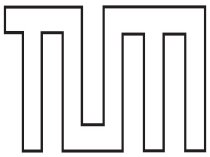
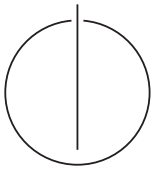


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



Intentional Meaning of Programs

Daniel Raţiu

9 July 2009

Institut für Informatik
der Technischen Universität München

Intentional Meaning of Programs

Daniel Raţiu

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Bernd Brügge, Ph.D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Univ.-Prof. Dr. Rainer Koschke, Universität Bremen

Die Dissertation wurde am 12.02.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.06.2009 angenommen.

To my family.

Acknowledgments

I would like to thank Prof. Manfred Broy for giving me the opportunity to work in a challenging and competitive research environment. I am grateful for his support and critical comments that made this work much better. I was in one of the few situations of PhD students doing a dissertation in the field of program understanding while being member in a group focused mostly on formal methods. This heterogeneity was very stimulating for me and gave me the opportunity to see beyond typical reverse engineering problems and the possibility to explore a broader research direction.

I would like to thank Prof. Rainer Koschke for being part of my thesis committee and for his support and friendliness during my visit in Bremen. His requirements and advices on performing good evaluations substantially improved the validation part of this dissertation.

During my research work, I had good collaborations with some of my colleagues at TUM with whom I published several research papers in the direction of this dissertation. Firstly, I would like to thank Florian Deißenböck for the work we did together in my early times at the chair. His target oriented, pragmatic and sharp approach to describe research problems and give feedback was very helpful and stimulated my research. I am also thankful to Jan Jürjens, who gave me useful insights about how to do research and how to write papers, and helped me with the formal parts of my dissertation. Special thanks go also to Martin Feilkas for the work we did together, for the hours of constructive discussions, for being a good friend, and last but not least for the excellent time we had at ICPC'08 in Amsterdam.

There are several persons that read (parts of) my dissertation and provided me very useful comments: Markus Herrmannsdörfer, Martin Feilkas, Jewgenij Botaschanjan, Florian Hölzl, Maria Spichkova, Christian Urban from TUM, Diana Raţiu and Stefan Schimanski from LMU, and Tudor Gîrba and Adrian Lienhard from SCG (Bern). Tudor also helped me during my research work at TUM, by acting several times as my postdoc. I am very grateful for your invested energy. Thank you!

I would like to thank Prof. Radu Marinescu and his group for making their reverse engineering infrastructures *iPlasma* and *inCode* available to me. While programming and performing different experiments, I interacted with several people that I would like to thank: Fei Xie, Yongming Li (TUM), Ioana Verebi, George Ganea, and Calin Drîmbau (Timisoara).

The experiments on extracting domain knowledge from domain-specific APIs were possible to perform due to the help of the following people that implemented exporters for different languages: Martin Feilkas (C#), Adrian Lienhard (Smalltalk), and Petru Mihancea (C++).

My work at TUM was made more enjoyable (and much easier) by: Silke Müller, Maria Spichkova, Mario Gleirscher, Martin Feilkas, Jan Romberg, Florian Hölzl, and Martin Fritzsche.

My student period in Romania was deeply marked by Prof. Radu Marinescu. Being my professor, diploma thesis supervisor, and a close friend in the same time, Radu was for me a model to follow, and I owe him my research background I had when I started my PhD. Tudor Gîrba helped me very much during my diploma thesis and I learned many things about doing research from him. I originally came to Munich due to the impulse Prof. Marius Minea gave me to attend the Marktoberdorf Summer School and to his support during my application.

Last but not least, I would like to thank my family, and especially my wife, for their continuous and unconditioned support, while I was struggling to do my research.

Abstract

Software engineering is a quest for appropriate modeling and abstraction. Writing programs that simulate parts of the real world requires programmers to fill the conceptual gap between the domain knowledge and computer languages. As a consequence of the conceptual distance between the business domain and the general purpose programming languages, clearly identifiable concepts at the domain level are implemented delocalized in the code and interleaved with each other and with a myriad of implementation details. This results in the loss of abstract information in programs. Once the code is written, it needs to be understood and this requires programmers to go the inverse way and to bridge the source code to the domain knowledge that it implements. Without doing this, the programs are meaningless for humans and are merely content interpretable only by machines. The recovery of domain specific information from programs is in the focus of the reverse engineering and program comprehension research efforts.

In this dissertation we develop a method for bridging the gap between domain knowledge and programs, by defining explicit mappings between program entities and domain concepts shared within domain ontologies. We call these mappings the *intentional interpretation* and they define the *intentional meaning* of a program. We use the intentional meaning to characterize the degree in which programs reflect the domain knowledge along four directions: the coverage of the domain in the code, the level of homomorphism between parts of the modeled domain and the code (distortion of knowledge), the distinguishability of domain concepts at the code level (diffusion of knowledge), and the logical redundancy in the implementation of domain concepts in the code. We investigate the measure in which the mismatches between the programs and the domain knowledge affect different quality attributes of programs such as: the extensibility of programs with new domain concepts, the conciseness and consistency in the implementation of domain concepts in the code and the protection against logical errors. By investigating different implementation strategies of concepts from the IEEE Suggested Upper Merged Ontology in Java programs, we show that many times mismatches originate in the limited conceptualization covered by the Java constructs and thereby they cannot be avoided at all.

In order to automate the conceptual analyses, we present a technique for automatic recovery of the intentional meaning based on the similarities between the names of concepts and the program identifiers. We discuss the limitations of the usage of identifiers for recovering the intentional meaning, with focus on their meaningfulness and ambiguity. Further, we investigate possible sources of domain ontologies that contain knowledge suitable for analyzing programs. We develop a method for extracting fragments of domain ontologies by analyzing similarities of domain specific APIs that implement the same domain. We present our experience with extracting fragments of domain ontologies from well-known APIs from Java, C++, .NET, and Smalltalk. Based on several case-studies, we show that a part of the intentional meaning can be automatically recovered, that it is feasible to automate the intentional analyses, and that they are useful for characterizing the conceptual coverage of APIs, the level of logical redundancy, and the level of diffusion. We show examples of mismatches between several domain ontologies fragments and parts of the Java standard API, and present our experience with performing intentional analyses of the Java systems JHotDraw and JEdit.

Kurzfassung

Im Zentrum des Software Engineerings steht die Suche nach geeigneter Modellierung und Abstraktion. Immer wenn Programme erstellt werden, die Teile der realen Welt simulieren, muss die konzeptuelle Diskrepanz zwischen der modellierten Domäne und den Programmiersprachen gefüllt werden. Programme müssen von Entwicklern verstanden werden. Dieses Verstehen erfordert, dass Entwickler die Lücke zwischen dem Programmtext und dem Domänenwissen, das dieser implementiert, schließen können. Ohne diesen Schritt sind Programme für Menschen sinnleer und sind nur Konstruktionen, die nur von Maschinen interpretiert werden können. Die Wiederherstellung von Domänenwissen aus Programmen ist im Zentrum des Forschungsbereichs des Programverstehens und des Reverse Engineerings.

In dieser Dissertation wird ein neuartiger Ansatz erarbeitet, um die Verknüpfung zwischen Programmen und Domänenwissen mit Hilfe von expliziten Abbildungen zwischen Programmelementen und Domänenkonzepten aus Domänenontologien zu schaffen. Wir nennen diese Abbildungen die "intentionale Interpretation" (intentional interpretation), welche die "intensionale Bedeutung" (intentional meaning) eines Programms definieren. Wir benutzen diese intensionale Bedeutung um die Art und Weise der Darstellung von Domänenwissen in Programmen zu charakterisieren, mit Fokus auf die folgenden Aspekte: Das Maß der Abdeckung einer Domäne in einem Programm, dem Grad in dem Teile der modellierten Domäne homomorph zum Quellcode sind (Verzerrung des Wissens), der Erkennbarkeit von Domänenkonzepten in Programmen (Diffusion des Wissens) und der logischen Redundanz in der Domänenkonzepte im Quelltext implementiert sind. Es wird diskutiert, in wie weit diese Divergenzen zwischen dem Domänenwissen und dem Code bestimmte Qualitätsattribute von Programmen beeinflussen, wie zum Beispiel deren Erweiterbarkeit, Konsistenz und Prägnanz, oder Absicherung vor logischen Fehlern. Zunächst wird eine Technik vorgestellt, um die intensionale Bedeutung basierend auf der Ähnlichkeiten zwischen Konzeptnamen und Programmidentifikatoren zu extrahieren. Wir diskutieren die Schwierigkeiten und Grenzen der Benutzung von Identifikatoren mit Fokus auf deren Ambiguität und Bedeutung. Um Wissen für die Analysen zu bekommen, wird eine Methode entwickelt, um Fragmente von Domänenontologien automatisch durch die Analyse von Ähnlichkeiten von mehreren domänenspezifischen APIs zu extrahieren.

Es wird die gewonnene Erfahrung mit der Extraktion von Ontologiefragmenten aus unterschiedliche APIs von Java, C#, C++ und Smalltalk dargestellt. Es werden Beispiele von Abweichungen der Java Standard APIs von deren modellierter Domäne gezeigt und logische Analysen von JHotDraw und JEdit vorgestellt. Der gewonnen Erfahrung nach enthalten Programme intensionale Bedeutung, von der ein großer Teil automatisch extrahiert werden kann und auf dieser Basis können konzeptionelle Analysen automatisiert durchgeführt werden.

Contents

Part I

Opening

1 Introduction

Software engineers use computer languages for describing parts of the real world. Ideally, in order to develop programs for a specific domain, engineers should use languages that directly support the description of that domain. Unfortunately, the currently widely used modeling, specification and programming languages are highly general, most of the times domain independent and ontologically neutral (van Lamsweerde, 2000). This leads to a big conceptual distance between the professional languages of domain experts and the languages used by software engineers. The same software engineering languages are used for describing a wide variety of situations. These languages make (almost) no difference whether they are used for building software for a vehicle, for a bank or for managing a hospital. One of the key challenges of modeling languages is the *abstraction challenge* namely, how can a language provide support for creating and manipulating problem-level abstractions as first-class entities (France and Rumpe, 2007). Whenever this challenge is not fulfilled, the generality of languages leads to *loss of conceptualization*: clearly defined and distinguishable concepts in the domain are not captured by language constructs and this subsequently leads to weakly defined models. The modeling and programming languages are "not aware" of the specifics of a domain and thereby the domain specific information is encoded informally as conventions or is simply lost.

languages

conceptualization loss

Intuitively, the situation in today's programming practice is similar to trying to explain to a four years old child (with a restricted vocabulary) advanced notions related to a particular software engineering project. Then give the same description, that is written in the vocabulary known by our child, to an experienced software engineer. None of them would understand what is the description about – the child due to the high amount of information and the engineer due to the loss of intentionality because of the different conceptual level at which the description is done.

Conceptual gap: *There is a big conceptual distance between the domain specific concepts and the today's widely used programming languages. As a result, many of the domain concepts are weakly defined and only partially (at best) captured in programs.*

Instead of programming from basic principles (e. g. implementing algorithms) the programmers are most of the times users of already existing code. The programming practice is biased from writing small and complex algorithms (programming in the small) towards mastering a large amount of knowledge about the existent libraries that many times have to be reused (programming in the large) (Meyer, 2000, p.80). Most system development involves extension of preexisting software systems and integration with legacy infrastructures (Finkelstein and Kramer, 2000). In the everyday programming activities, programmers are faced with the challenge of re-using large pieces of code, most of the times available in form of software libraries or collection of classes that model domain concepts. In Figure 1.1 we intuitively illustrate the

need for reuse

representation of domain knowledge in APIs (or general program interfaces) and how is this used by their clients to implement other programs.

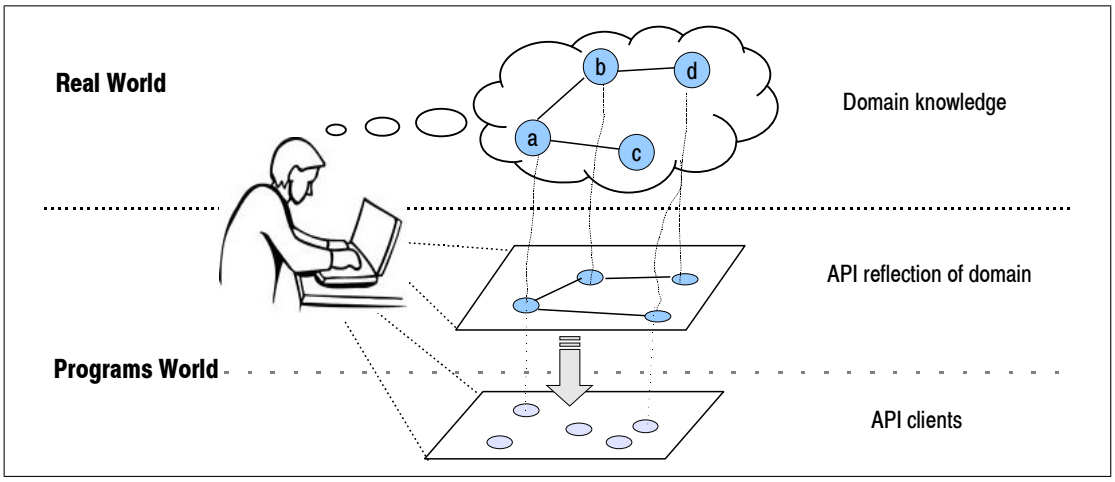


Figure 1.1: Programmers use the reflection of the domain knowledge in APIs

By having to implement business requirements in a pre-existing infrastructure, or to use APIs that do not fit perfectly the needs of the business domain, the programmers have to perform additional encoding steps. Whenever programmers implement software solutions by using abstractions that are inadequate to describe the problem, there occurs a high encoding bias – relations between concepts are not reflected anymore at the code level and the code structure is biased from the conceptual structure of the business domain.

Innapropriateness of implementation means: *Many times the programmers have to use innapropriate means (e. g. languages, libraries) for a specific programming task. In these cases the programmers need to perform an encoding of the desired domain concepts by using the languages or libraries at hand.*

dominant decomposition

The general purpose programming languages allow their users to decompose the domain along a main direction (problem also known as *tyranny of dominant decomposition* (Tarr et al., 1999)). Different views over the domain are today impossible to define and integrate in a single module and this leads to delocalization, a central problem in the program comprehension (Letovsky and Soloway, 1986; Rajlich and Wilde, 2002)

Delocalization: *The implementation of concepts from the business domain is spread across many software modules.*

Pieces of code that implement core domain functionality are scattered among pieces of code that refer to programming technologies. Many times pieces of code that are responsible for implementing more than one purpose are woven together in a single program part – phenomenon known as “interleaving” (Rugaber et al., 1995).

Interleaving: *The information belonging to the business domain is combined at the code level with information about technical domains.*

Each of the problems presented above is related to the *loss of intentionality* manifested in an ambiguous mapping between the concepts of the business domain and parts of programs. Due to this loss it is very difficult (or even impossible) to interpret programs from the point of view of the domain concepts that they implement and thereby to understand programs and to raise the abstraction level at which program analyses are performed.

intentionality loss

1.1 Thesis

In this dissertation we develop a method to express the meaning of a program in terms of the domain concepts that it implements. We call this the *intentional meaning of programs*.

intentional meaning

Thesis. To characterize the faithfulness of the implementation of domain knowledge in code, we need to capture explicitly the intentional meaning of programs.

We capture the intentional meaning with the help of mappings between concepts shared within domain ontologies and the program elements (e. g. classes, methods, attributes) that implement them. Our main *research hypotheses* are:

research hypotheses

1. Programs do exhibit intentional meaning and it is sensible to interpret program elements from the point of view of domain concepts that they implement.
2. Regarding programs from the point of view of the concepts that they implement increases the abstraction level at which software analyses are performed.
3. Light-weighted domain ontologies, made up of domain concepts and relations among them, can be used as semantic domain to perform conceptual analyses of programs.
4. It is feasible to automatically recover the intentional meaning based on the similarities between the program identifiers and the names of concepts.

1.2 Approach

Intentional meaning of programs. In Figure 1.2 we present an intuitive view over our method for defining the meaning of programs: we use ontologies to represent the semantic domain (the meaning) with respect to which a program is interpreted.

Use ontologies as semantic domains. Ontologies play a central role in different fields of artificial intelligence (e. g. natural language processing, semantic web) as means to model and share knowledge about the real world (McGuinness, 2003). At the core of representing (light-weighted) ontologies are concepts arranged in a taxonomy and a set of relations among them.

Intentional program abstraction. In order to map programs to ontologies, we need a representation of programs that is comparable with the chosen semantic domain. We abstract programs as graphs: the nodes are named program elements and the edges are program relations among these elements.

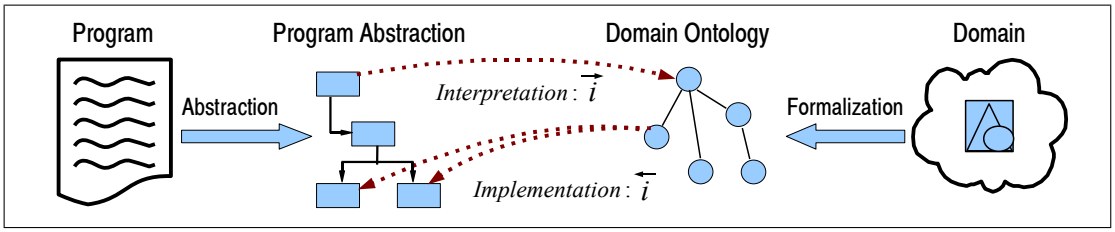


Figure 1.2: Intentional meaning intuition

Expressing intentional meaning. Through intentional meaning of programs we understand the links (specified with the *intentional interpretation* ($\overrightarrow{i}$) and *intentional implementation* ($\overleftarrow{i}$) functions) between its program elements and the domain concepts that they implement (Figure 1.2). With other words we make explicit the meaning of a particular program element in the program from the point of view of the domain knowledge that it implements. Starting from this, we can characterize programs from the point of view of the implemented concepts. We do this along three directions: the *reference of domain concepts* – how are the concepts be referenced at the code level; the *definition of domain concepts* – how are the concepts defined through program abstractions; and *representation of domain concepts* – how does the program encode the concepts internally.

The intentional implementation and interpretation are extensions of the *concept location* (Rajlich and Wilde, 2002) and *concept assignment* (Biggerstaff et al., 1994). The extension consists in the facts that the concepts are parts of domain ontologies and that they are assigned to program entities through well defined mappings. This has two implications: firstly, being part of a domain ontology, the concepts are part of a system of symbols (given by the ontology itself), are related to other concepts and thereby have *meaning*; secondly, *individual* program elements are mapped to concepts and thereby the mappings have a finer granularity.

Characterize the implementation of domain knowledge in programs. We use the intentional meaning in order to identify and characterize problems with the implementation of the domain knowledge in programs. We do this by measuring the level of isomorphism between a program and a domain ontology. We focus on the following issues:

1. *conceptual coverage*, meaning the measure in which programs implement domain concepts and the measure in which they can be extended to implement new domain concepts;

2. *distortion of domain knowledge in programs*, meaning the measure in which the program structure reflects the structure among domain concepts;

3. *diffusion of domain knowledge in programs*, meaning the measure in which different domain concepts are distinguishable at the program level; and

4. *logical redundancy*, meaning the conciseness in the implementation of domain concepts in programs.

Based on the explicit mappings between concepts, names and program elements, we characterize the *quality of identifiers* in terms of their meaningfulness (i. e. the measure in which they reflect the implemented concepts) and ambiguity (i. e. the measure in which the names are consistently used).

Automatic recovery of the intentional meaning. The structuring of programs and the information conveyed by the names of program elements play a central role in understanding programs. We use these observations to automate the recovery of the intentional meaning.

Programs are knowledge bases. Due to the program structure, the identifiers names appear in a program in relations (given by the structure) with other names. Thus, programs can be seen also as systems of names. We combine both the naming and structural information contained in programs and we *regard programs as knowledge bases*: the content of these knowledge bases is given by the set of identifiers and the knowledge representation language by a sub-set of the programming language. The program knowledge base contains a mix of knowledge varying from business domain to implementation details. In order to pull these dimensions apart and to be able to answer questions specific to a particular dimension (e. g. where is a domain concept implemented) we use the *intentional meaning*.

Automatic recovery of intentional meaning. We represent both programs and ontologies as graphs. The uniform representation enables us to (semi-)automatically recover the intentional meaning by mapping programs to ontologies using graph matching techniques. We define the similarity of nodes to be the similarity of names of program elements and respectively concepts; besides this we define a set of mapping strategies between the paths in the ontology and the program graphs.

Extracting domain knowledge from APIs. To make our approach usable in the practice we need a high amount of domain knowledge expressed as domain ontologies and that are at the level of abstraction appropriate for code analyses. In the practice, the ontologies available off-the-shelf are not suitable for analyzing programs due to their restricted conceptual coverage or to their different abstraction level. Furthermore, there are no ontologies that cover the domain of programming technologies (e. g. GUI, XML, data structures), that represent the most wide-spreaded form of knowledge contained in programs. In the practice there are many APIs that address the same domain. We take advantage of this fact and analyze the commonalities between different sets of APIs that address the same domain in order to extract domain knowledge in form of fragments of domain ontologies.

Experiments and relevance in the practice. In order to show the pervasiveness and (potential high) consequences of the mismatches between the domain knowledge and programs, we present examples of problems in the implementation of the domain knowledge in the Java standard APIs and two other Java systems (JHotDraw and JEdit). The case-studies that we perform have the following aims: to investigate the degree in which programs exhibit intentional meaning, the degree in which light-weighted ontologies can be used to represent the domain meaning, and the degree in which the intentional program analyses are automatable.

1.3 Contribution

This dissertation develops a program analysis technique, based on mapping program elements to concepts from a domain ontology, in order to define the intentional meaning of programs. Based on this, we define code analyses that capture the mismatches in the representation of domain knowledge in programs, with focus on public interfaces (e. g. APIs). The current work presents the following contributions to the current state of the art:

1. **Definition of intentional meaning:** We define the notion of *intentional meaning of programs*. Instead of describing what a program does in terms of the computation that it performs, the intentional meaning describes the program in terms of domain concepts that it implements. We show that there are many research works which, in a way or another, point out the need for "intentional meaning" of programs. We present a method for expressing the intentional meaning of a program by using ontologies as semantic domains. We describe the implementation of domain concepts along three directions: the manner in which programs reference domain concepts, the manner in which they define domain concepts at the code level, and the manner in which they represent internally domain concepts.

2. **Logical analyses of programs:** Based on the intentional meaning of programs, we identify, describe, and categorize four categories of mismatches in the implementation of the domain in programs: the conceptual coverage of the domain in the code, the level of homomorphism between parts of the modeled domain and the code (distortion of knowledge), the distinguishability of domain concepts at the code level (diffusion of knowledge), and the logical redundancy in the implementation of domain concepts in the code. We explain how these mismatches affect different quality attributes of programs such as their extensibility with new domain concepts, consistency, and conciseness. By investigating the implementation strategies of the concepts and relations from the Suggested Upper Merged Ontology (SUMO) in Java programs, we show that many mismatches cannot be avoided at all.

3. **Automatic recovery of intentions:** We define an algorithm and methodology for the automatic location of concepts in the code based on the similarity between the program identifiers and names of the concepts from the ontology. The automatic concepts location algorithm is a basis for automating intentional program analyses. We define a formal framework for describing the program identifiers, concept names and the relations between them. We use this framework to characterize the quality of names in terms of their ambiguity and meaningfulness.

4. **Extraction of domain knowledge from domain specific APIs:** By analyzing the commonalities between different domain-specific APIs that address the same domain, we extract domain knowledge in form of fragments of domain ontologies. Our focus is on technical domains typically covered by APIs such as XML, graphical user interfaces (GUI), or data structures. We show that (parts of) the standard APIs implement their domain in a similar manner and that by analyzing different APIs we can extract a large amount of knowledge about typical programming technologies.

1.4 Overview of the Dissertation

Figure 1.3 illustrates at a glance our contribution (left) and the structure of this dissertation (right).

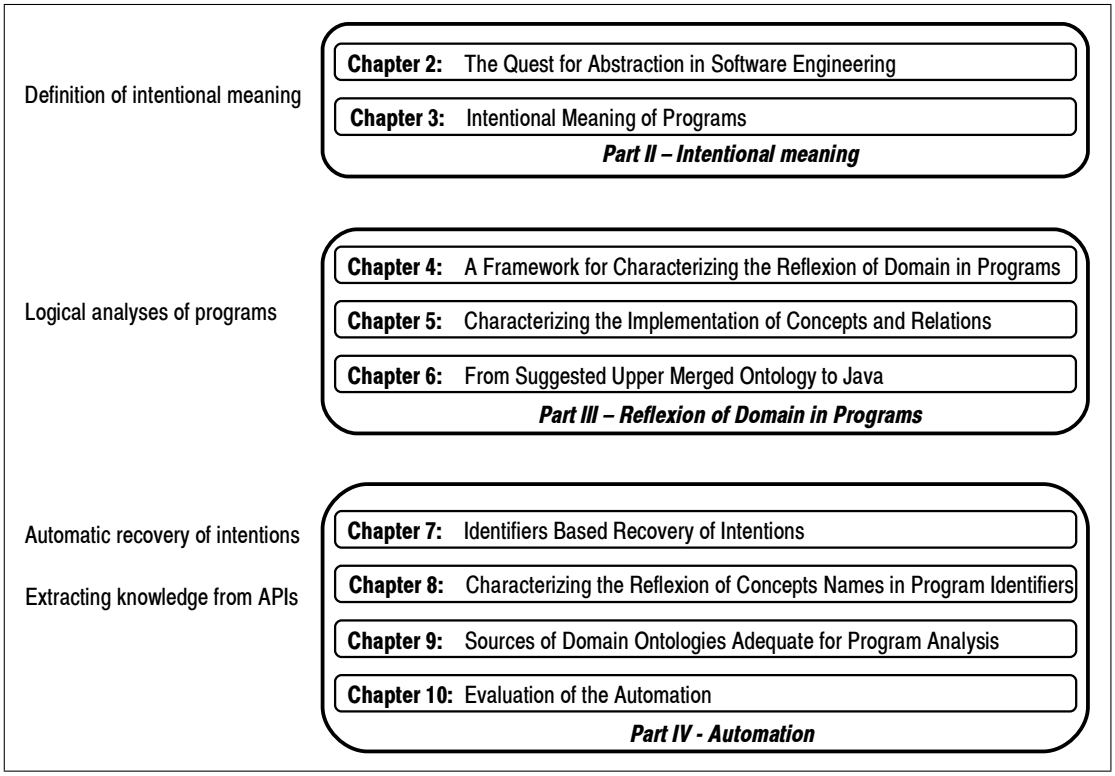


Figure 1.3: Overview of the main parts of this dissertation

Typographical conventions. Due to the high similarity between some of the domain concepts and Java programming elements, whenever there is a danger of confusion we use the following typographical conventions: through SMALL-CAPS we denote the domain concepts (e. g. CLASS, FILE) and through `type-writer` fonts we denote the entities from Java programs (e. g. `class`, `java.io.File`).

1.5 Origins of the Chapters

A significant part of this thesis was published in a series of scientific papers. These papers however use slightly different terminologies and formalizations. Our formal framework and our view over the relation between the domain concepts and programs, that represent the background material of our work, developed and matured in time. Even if the need for a coherent text demanded us to change the terminology and to broaden the scope, there is a clear correspondence

between parts of this dissertation and some (parts) of the published papers. Below we present this correspondence according to the chronology of the papers:

- (Ratiu and Deissenboeck, 2006b) – the intentional program abstraction and the general idea of giving meaning of programs by mapping them to ontologies.
- (Ratiu and Deissenboeck, 2006a) – formal definition of identifiers quality, general idea of logical duplications and their effects.
- (Deissenboeck and Ratiu, 2006) – the unified meta-model that contains the concepts, program elements and the names.
- (Ratiu and Juerjens, 2007) – the formalization of reflection of domain in the APIs and the presentation of a set of distortions.
- (Ratiu and Deissenboeck, 2007) – the formalization and description of diffusions of the domain knowledge in programs; the methodology for manually building domain ontologies that are fit for performing the intentional analyses.
- (Ratiu et al., 2008b) – the algorithm and methodology for extraction of domain ontologies from domain-specific APIs.
- (Ratiu and Juerjens, 2008) – the difference between the reference, definition and representation of domain concepts in the APIs; a formal definition and characterization of problems with the representation of domain concepts in code.
- (Ratiu et al., 2008a) – the presentation of the repository of technical knowledge, the issues in building it and the advantages of having it for enhancing program analyses.

Acknowledgments: The publications of these papers would have been impossible without the joint work and discussions with: Florian Deißenböck, Jan Jürjens, Martin Feilkas and Radu Marinescu. **Thank you!**

Part II

Intentional Meaning

2 The Quest for Abstraction in Software Engineering

X **lied** to Y =
X said something to Y
X knew that it was not true
X said it because X wanted Y to think that it was true
people think that it is bad if someone does something like this

The definition of "lie" in the "Natural Semantic Metalanguage"

Abstract: Programs simulate parts of the real-world: they act and respond to users actions as such they would "know" about a certain situation from the application domain. Writing programs that simulate real-world phenomena require programmers to fill the (huge) abstraction gap between the application domain and the machine. In this chapter we regard the software engineering efforts as a quest for abstraction. The artefacts produced during forward engineering are described in a few number of languages that are general enough to accommodate a wide variety of domains. These languages usually do not reflect in any way the semantic of the domain to which the applications are targeted. Furthermore, they introduce a considerable amount of implementation details that are interleaved with the information about the application domain. This leads to a loss of abstract information and intentionality even from the early phases of software engineering with a culmination on programming. A central theme of reverse engineering is the recovery of abstract information from programs. The current reverse engineering approaches are limited in terms of the abstraction level of the recovered information, in the precision with which the domain information is described, and the accuracy with which the recovery is performed. We advocate that in order to recover the lost abstraction and to characterize the faithfulness of the implementation of domain knowledge in programs, we need to explicitly take into consideration the relations between program elements and domain concepts that they implement.

Structure of this chapter. After the introduction (Section 2.1), in Section 2.2 we regard the software engineering process from the perspective of closing the gap between the application domain and the implementation technologies. In Section 2.3 we present examples of abstraction and intentionality loss in different artefacts produced in the software engineering process with emphasis on the source code. In Section 2.4 we present the reverse engineering approaches to recover the domain knowledge from programs. In Section 2.5 we advocate that we need a more rigorous manner to link the source code to the domain knowledge that it implements, in order to be able to characterize the effects of the loss of abstract information in programs. Section 2.6 ends this chapter with a summary and a road-map for the next parts of this dissertation.

2.1 Introduction

semantic primes

At the beginning of this chapter we presented an example from the Natural Semantic Metalanguage Project[1], that aims to identify the smallest number of words (called *semantic primes*) that can be used to describe all other words of the English language. We notice the intricacies that occur when we define a relatively simple word (i.e. "lie" in our case) in terms of these semantic primes. Let us make the following mental experiment: try to write a professional (e. g. software engineering, accounting) book using only the semantic primes and the words defined with these semantic primes. We envision here two strategies: the first is to define incrementally all the terms related to the professional domain (beginning with the most simple ones and continuing with the more complex ones) and only writing the book once the first phase is completed and all the needed words are defined; the second strategy is to define no professional terms but instead to describe them in-place by using the semantic primes whenever necessary.

The situation with the current programming practice is not much different: we have to use low-level programming constructs (similar to the semantic primes), in order to model a wide variety of domain situations, and to define a vocabulary consisting of program abstractions. Similarly with the two options from above, we can build incrementally the more complex vocabulary by using layers of definitions of increasingly complex concepts or we can directly use the already existent basic means (e. g. standard APIs) to directly define the complex concepts. These situations are exemplified below: before they are used, the concepts NAME, AGE, and HEIGHT are firstly defined explicitly (left); or the concepts are encoded using the primitive types (right). In the latter situation we lose conceptualization since clearly defined and distinguishable concepts from the application domain are now encoded in the program and (partially) not distinguishable anymore (i. e. the variables `height` and `age` are both integers).

```
class Name { ... }
class Age { ... }
class Height { ... }
class Person {
    Name name;
    Age age;
    ...
}
```

```
class Person {
    String name;
    int age;
    int height;
    ...
}
```

The *loss of conceptualization* brings the inability to directly address and manipulate the application domain concepts at the code level. In this chapter, we regard the software engineering process as a quest for abstraction and modeling: the challenge of transforming back and forth the domain knowledge to and from programs. Figure 2.1 presents the major phases of software engineering and the abstractions that correspond to these phases (Harandi and Ning, 1990):

software engineering phases and abstractions

- the *implementation view* abstracts away the concrete syntax of a program and the low level implementation details and represents programs as syntax graphs,
- the *structural view* reveals the program structure from different perspectives and abstracts away the program to a set of components (modules) and dependencies between them (e. g.

[1] `http://www.une.edu.au/bcss/linguistics/nsm/`

architecture, control flow, data flow),

- the *functional view* relates program modules to high-level functionality (e. g. sorting, performing persistence) and reveals the logical relations among them, and
- the *domain-level view* further abstracts the programs to a set of concepts specific to the application domain.

While the implementation view is close to programming and therefore can be automatically derived from the code, in the case of programs that implement non-technical domains (e. g. banking), the domain-level view is very different from programming and therefore hardly recoverable. The research in reverse engineering and program comprehension is focused on crossing these borders, namely on obtaining a higher-level description of programs. However, as we will show at the end of this chapter, *most of the current automatic analysis approaches are focused on the implementation, structural and functional views while a disciplined and systematic treatment of the domain-level view is currently missing.*

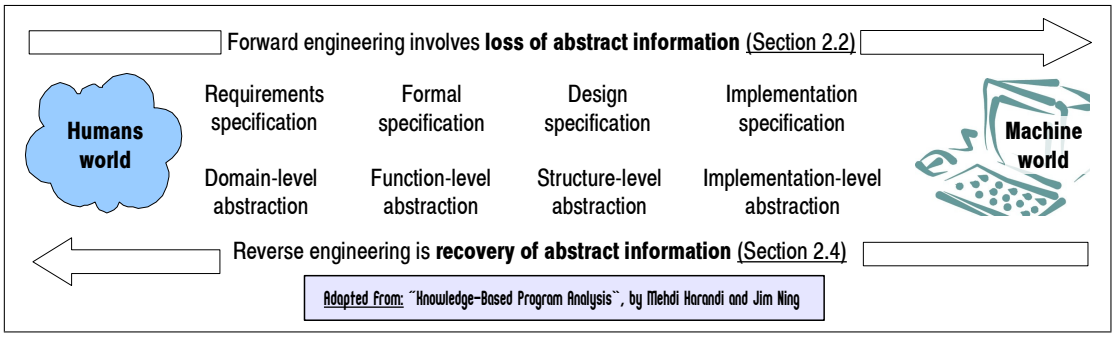


Figure 2.1: "Forward engineering" vs. "Reverse engineering" as the quest for implementing and recovering abstraction (adapted from (Harandi and Ning, 1990))

2.2 The Loss of Abstract Information during Forward Engineering

descriptions in software engineering

Almost everything that we do in software engineering is to describe various aspects of the application domain or of the computing domain (Jackson, 1995, p. 58). We use different descriptions in order to communicate with the stakeholders and with the computer itself. These descriptions are at different levels of abstraction and are meant to fill in the gap between the real-world phenomena (observations) and the computers world. Each of these descriptions is expressed in a language. Unfortunately, only a very small number of computer languages (and language constructs) that are currently employed in software engineering, must accommodate all real-world situations that need to be described. In contrast to these languages that make (almost) no difference between various problem domains, experts from various fields use professional languages to work in their domains – e. g. the set of concepts used by a biologist is very different in comparison to the concepts used by a mechanical engineer or an economist. Unfortunately, this difference is mostly ignored (not explicitly considered) in the process of producing software and

this inherently leads to loss of abstract information manifested as *conceptualization loss* – once implemented in a program, different domain concepts are not distinguishable anymore.

conceptualization loss

In the following parts of this section we describe the challenge of filling in the conceptual gap between the business and the computer domains through building cascades of models. These models have to be expressed in a small number of computer languages that are most of the times too general and inappropriate for capturing the domain information. This subsequently leads to difficulties (or inabilities) in referencing, distinguishing and manipulating the domain level concepts once they are modeled.

2.2.1 Programs Simulate the Real World

The meaning of programs, as illustrated in Figure 2.2, can be expressed in terms of their interactions with the world outside of computers (black-box view). For making our argument more clear, *we regard programs as communication partners* that interact with human users (e. g. the domain experts). This model of regarding programs as communication partners applies (at least partially) also in the case of embedded systems; in this case the communication partners are sensors, actuators, or other programs. We can imagine the interaction to take place through a set of questions and answers: the domain expert asks questions by providing all the necessary input and the program answers these questions. In the case of embedded systems, the questions are for example activated through interrupts and the information is transferred as parts of the memory (e. g. registers). The communication is mediated by an interface that the program provides for its clients. The questions and answers contain information about the application domain. The interface provides representations of domain concepts that domain experts can easily understand.

Ideally, the dialogue between a user and the program is done at the *knowledge level* (Newell, 1982) – the user thinks, acts and expects results from the program in terms of its domain of expertise, and according to the principle of rationality: the actions are performed in order to achieve well defined goals. In order to answer the questions, the program needs to 'know' about the concepts and the current state of affairs of the modeled domain.

knowledge level

Behind the user interface, programs use layers of representation of the domain knowledge: the texts and graphics of the user interface represent directly domain concepts that are understandable by the domain expert; behind the graphical user interface different domain abstractions are defined in a program. These domain abstractions are in turn represented in terms of a programming paradigm (e. g. the object-model in the case of object-oriented languages). In order to implement their persistency, the objects need to be saved in a data-base model (e. g. very often a relational model).

representation layers

Example 2.1: Examples of the interaction between domain-experts and programs

Let us consider that an economist works with a payroll system. The interaction between the program and the economist is done through a graphical user interface. For searching the information about an employee, the program asks the user for the name of the person (e. g. provides a dialog-box where the economist can enter the name of the employee). After entering the employee's name (and thereby answering the question of the program) the user presses the "start search" button (and thereby asks the program about the employee's information). The program

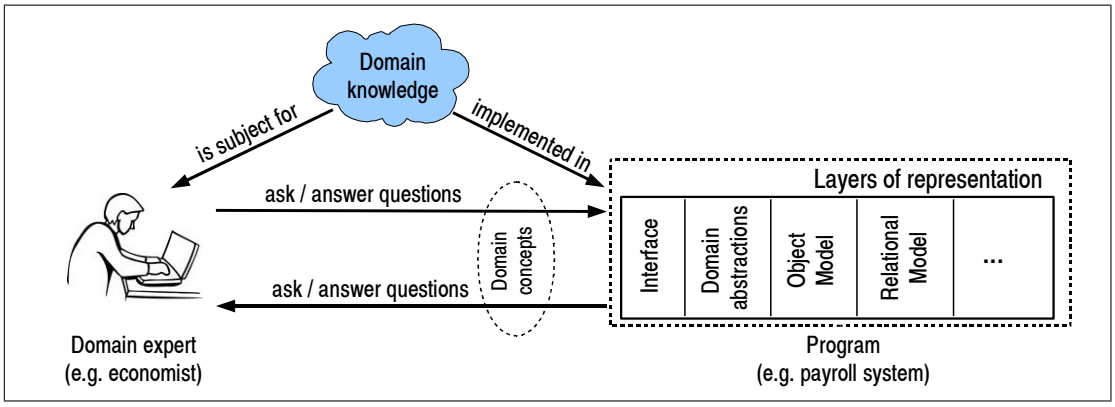


Figure 2.2: Programs interact with the real world

uses the name of the employee to perform the search and displays the information about the employee (answers the question). We can easily imagine that a person without knowledge about economy (e. g. an engineer) is not able to use the payroll system because she cannot provide the proper input or properly interpret the system's output.

□

Fact: Many programs simulate parts of the real world. Their inputs can be given and their results can be interpreted by domain experts solely in terms of their domain knowledge.

2.2.2 Languages Used in the Software Engineering

The artefacts that are produced during the development are described in different languages ranging from natural languages (e. g. requirements) to specification languages (e. g. Z (Spivey, 1992), Focus (Broy and Stolen, 2001)), modeling languages (e. g. UML) and programming languages (e. g. Java, C#). Bjørner characterizes the *professional languages* to be: *professional languages*

> "[...] the languages of domain-specific fields, such as the those used by those people for whom we make software." (Bjørner, 2006, p.218)

The professional languages are on the one hand *application-domain-specific language* (e. g. railways, banking, finance) and on the other hand *software engineering languages* (e. g. design, specification). The professional languages use precisely defined terminologies that denote specific phenomena from the domain:

> "The professional languages are characterized by a relatively precise use of terms. Certain verbs, nouns, adjectives and adverbs stand in relatively precise relations to the phenomena they designate they are, so to speak, part of the jargon of the professional trade." (Bjørner, 2006)

Each language provides its own set of constructs that can be used to describe a particular state of affairs in terms of a set of domain concepts – called *domain conceptualization*. The

link between the real-world things and the language can be visualized as in *Ullman's triangle* (Ullmann, 1972) (Figure 2.3): an entity from reality is abstracted to a concept and this concept is represented by a language construct. The language constructs can be used to (indirectly) refer to things from the reality. Furthermore, the interaction between entities is reflected in the language as relations and compositions between their corresponding constructs. These constructs and the compositions between them can be further used to describe situations from a domain. The constructs of modeling languages can then be interpreted according to the concepts that they represent (the "represented by" relation describes the *real-world semantics* of a language (Guizzardi, 2005, p.27)).

Ullman's triangle

real-world semantics

A language provides a set of concepts directly. These concepts can be used to represent more concepts indirectly.

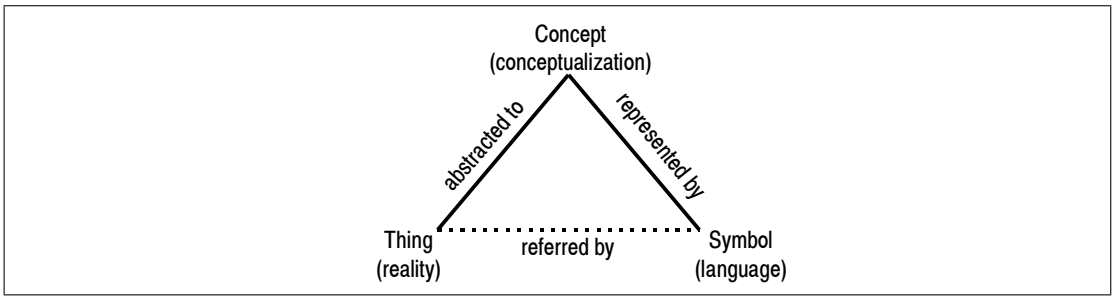


Figure 2.3: Ullman's triangle represents the relation between things in the reality and the constructs of a language (Ullmann, 1972)

Figure 2.4 illustrates the relation between abstractions of the real-world (model), a domain conceptualization, a modeling language and the specifications written with this language (Guizzardi, 2005, p.27). A conceptualization determines all possible valid state of affairs that are admissible in a domain and a language determines all possible specifications that can be constructed by using that language. When there is no clear relation between the domain conceptualization and the modeling language then the language is not appropriate to the domain. In these cases the relations between the real world situations and the specifications that describe them are also not clear: the same real world situation can be specified in completely different manners or vice-versa, the same specification can be interpreted to represent different situations.

abstraction challenge

One of the key challenges of the modeling languages is the *abstraction challenge*, namely how can one provide support for creating and manipulating problem-level abstractions as first-class modeling elements (France and Rumpe, 2007). The abstraction challenge is approached from two directions:

1. by using *extensible general-purpose modeling languages* that provide facilities for extension with domain-specific abstractions (e. g. the profiles mechanism make UML extensible);

2. by using *domain specific languages* that allow the direct representation of domain concepts (e. g. Bibtex allows the direct representation of bibliography information).

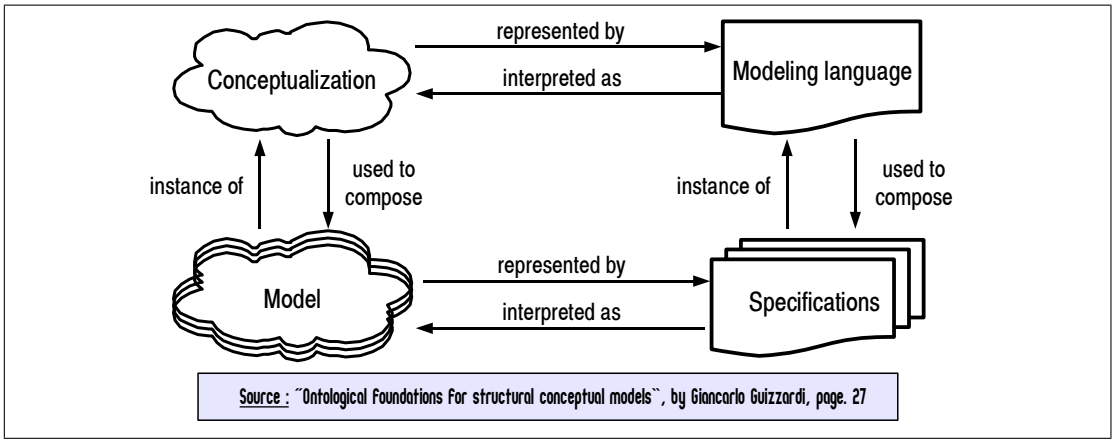


Figure 2.4: Relation between conceptualization, model, modeling language and specification (Guizzardi, 2005)

Language expressiveness. Besides the computational expressive power (e. g. Turing completeness) and abstraction power (Felleisen, 1991; Mitchell, 1993) of languages, the domain appropriateness represents another facet of their expressiveness[2]. (Krogstie, 2000) notes that: *language expressiveness*

> "[...] if one is looking for the suitability of a modeling language across a number of different, potentially unknown domains, one usually uses the term expressiveness."

The expressiveness of a language affects the understandability of the models built within this language (Selic, 2003). Lamsweerde notes in the context of the expressive power of specification languages two shortcomings (van Lamsweerde, 2000):

1. each specification paradigm "has some built-in semantic bias in order to be useful" – e. g. state-based specifications focus on sequential behavior and provide rich means to describe the structure of objects that are manipulated, and transition-based specifications focus on concurrent behavior while providing simple structures for defining the objects that are manipulated; *semantic bias*

2. beside the encoding bias, today's formal specifications suffer from the fact that they are based on low-level (programming-oriented) ontologies – namely that "the concepts in terms of which problems have to be structured and formalized are programming concepts" – and that for the future the formal specifications need to support richer, problem-oriented ontologies.

The different focuses of languages lead to *impedance mismatch*. The impedance mismatch denotes the ambiguities and the difficulties of translation between different languages that are used to represent the same domain phenomena (state of affairs) that are at a possibly different abstraction levels (Evermann and Wand, 2005). For example, there are difficulties in implementing data management tasks of a financial application with the help of relational database management systems (Rozen and Shasha, 1989): *impedance mismatch*

[2]Expressive means: " effectively conveying meaning" and meaning is defined as: " the thing, person, etc. for which a word or expression stands" (Merriam-Websters English Dictionary)

> "Developers of a Wall Street financial application were able to exploit a relational DBMS to advantage for some data management tasks (the good). For others, the relational system was not helpful (the bad), or *could be pressed into service only by means of major or minor contortions (the ugly).*" (Rozen and Shasha, 1989) (our emphasis)

Even if the above description of implementation difficulties is very expressive, it does not provide any information about the details, consequences or quantification of "contortions". In Chapters 4 and 5 we will formally describe different kinds of "contortions" and discuss their consequences on different programming and maintenance activities.

Example 2.2: Example of impedance mismatch when translating an E-R model into Java

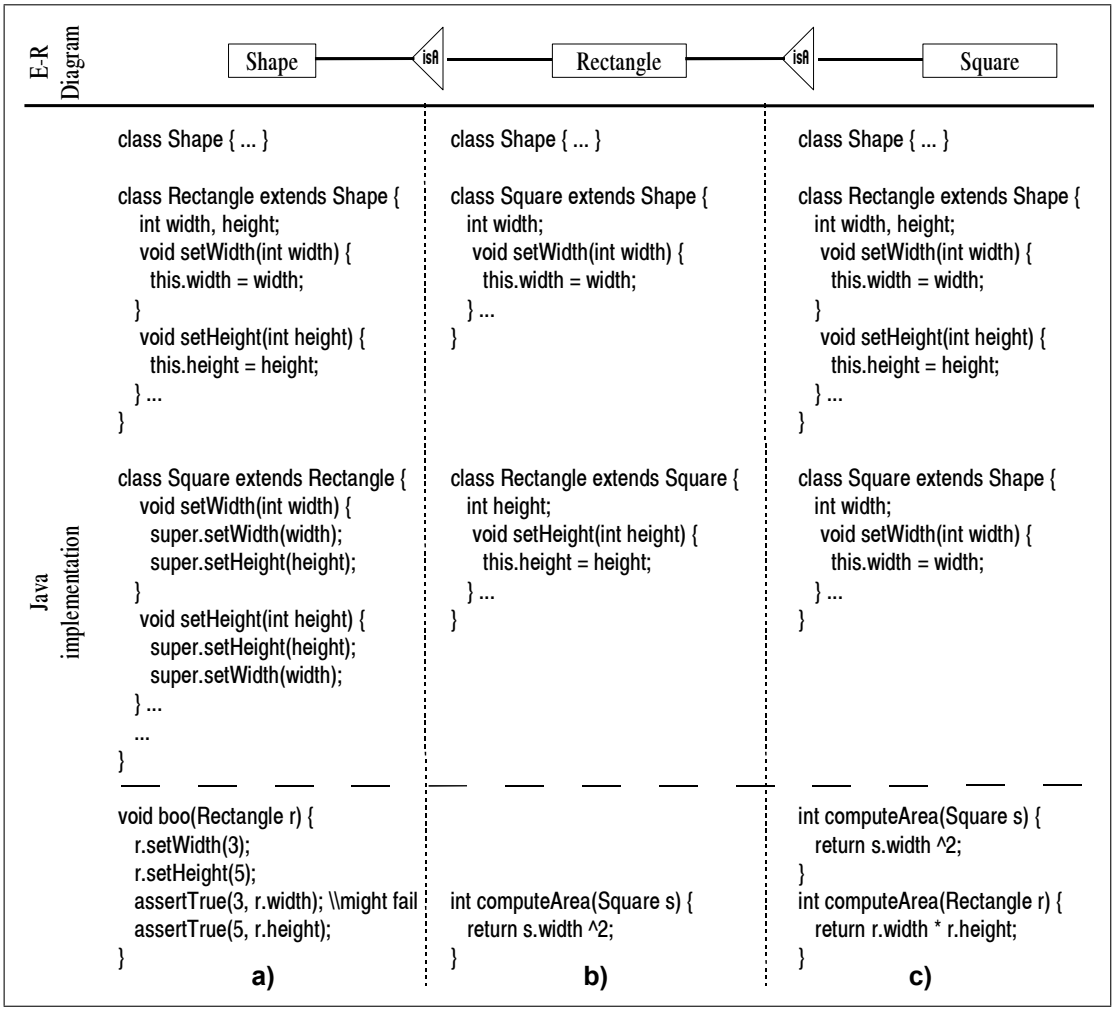


Figure 2.5: Impedance mismatch when translating portions of an E-R diagram into Java

Let's suppose that we have to implement a hierarchy of shapes that contains also 'rectangles' and 'squares'. In the upper part of the Figure 2.5 we present our hierarchy of shapes modeled

at a conceptual level with the help of Entity-Relationship diagrams. Even if this structure seems simple, to represent the same hierarchy in Java, as shown in the lower part of our figure, is rather tricky:

- In the case a) we have a violation of Liskov's substitution principle (Liskov, 1987) (also detailed in (Martin, 1996)). The assertion written in function `boo` will fail if the parameter is an object of type `Square`. The assertion illustrates the fact that a programmer that wrote the function `boo` assumed that by changing the width of rectangles their height will remain the same. The problem is that, from the point of view of their behavior, the objects of type `Square` do not behave like the objects of type Rectangle – the methods `setHeight` and `setWidth` from `Square` do not conform to the contracts of the corresponding methods from `Rectangle`.
- In the case b) we cannot substitute squares for rectangles (as we would expect) but vice versa. When substituting objects of type `Rectangle` where objects of type `Square` are expected, the result can be wrong as shown in the `computeArea` function that obviously returns a wrong value when called for a `Rectangle` object.
- The case c) is the best object-oriented design with respect to the implementation of the 'simple' conceptual hierarchy defined with the E-R diagram. This design however, does not reflect the relations between squares and rectangles.

This represents an example of 'contortion' of a model in another language – the relation between 'square' and 'rectangle' cannot be clearly translated between E-R diagrams and Java.

□

the quest for domain appropriate languages

Many of the advances in programming language design are concerned with raising the level of discourse at which software developers work. The ability of programming languages to "program the domain" was envisioned to be a "strategic direction" in programming languages and in software engineering (Gunter et al., 1996). Domain experts should be allowed to easily and effectively express programs in a language specific to their domain and thereby to eliminate the risks of inappropriate decisions made by non-experts in the domain. Furthermore, a language should enable its users to build only models that make sense from the point of view of the modeled domain. Ideally, the constructs of a language should be at the abstraction level of application domain and should correspond to the ontological categories (concepts) of the domain (Hruby, 2005). Depending on how abstract the language constructs are, the language can be used to express a wider or narrower domain as presented in Figure 2.6.

ontological analysis evaluates the domain appropriateness of languages

There are several works in the literature of information systems (e. g. (Krogstie, 2000; Wand and Weber, 1993; Guizzardi, 2005)) that analyze the domain appropriateness of languages used for conceptual modeling. In order to describe the (non-)correspondence between language constructs and domain concepts, Wand performs *ontological analysis* of the language by defining two mappings: one from ontological entities to language constructs (*representation mapping*), and the other from language constructs to ontological entities (*interpretation mapping*). The faithfulness of the language in reflecting domain is described through the following terms: *construct deficit* of a language, when the language is not powerful enough to express a domain concept, *construct excess*, when the language is able to express things that are not in the domain, *construct overload* when a language construct is used to express different things from

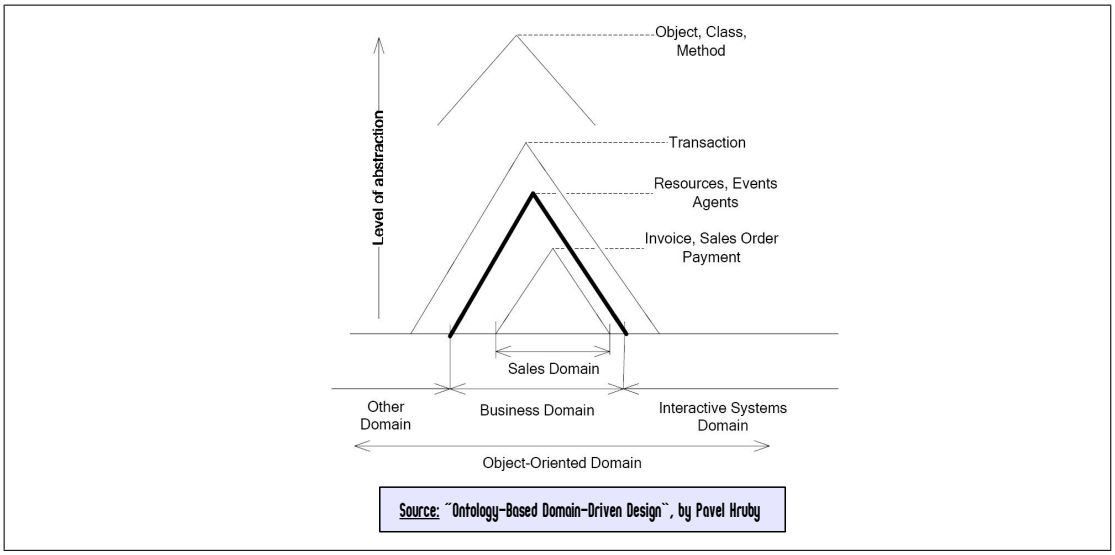


Figure 2.6: Levels of abstraction (here "abstraction" means "generality") and covered domain (Hruby, 2005)

the domain, and *construct redundancy* when two language constructs express the same domain concept (Figure 6.6).

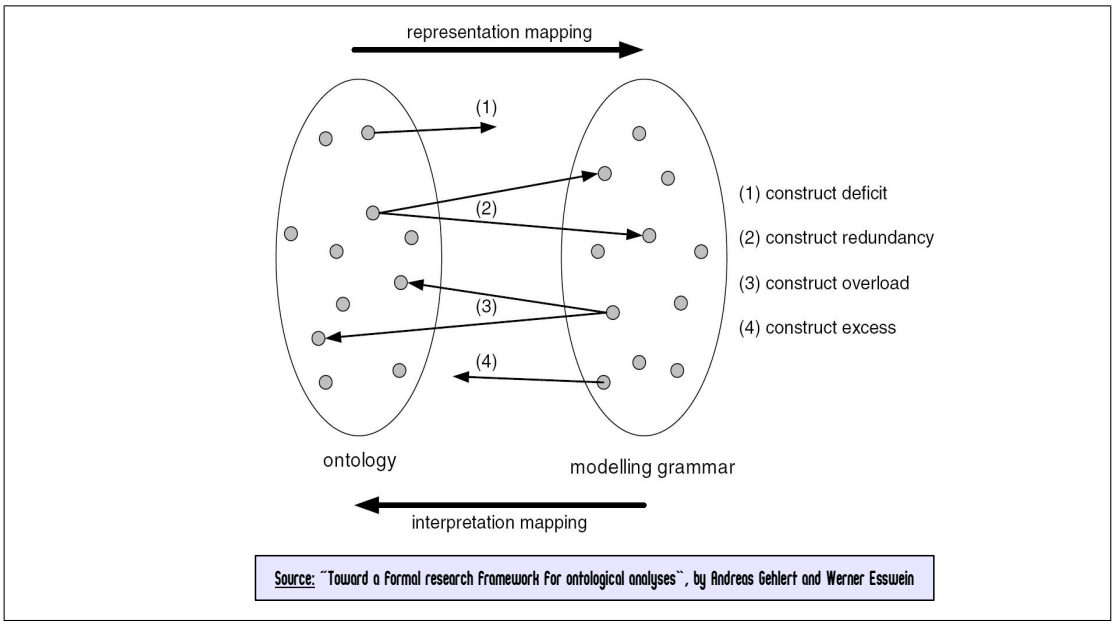


Figure 2.7: The (non-)correspondence between language constructs and domain concepts (Gehlert and Esswein, 2007)

The loss of intentionality resulting from encoding a domain concept in an inappropriate language is well known in the programming languages community:

effects of domain innapropriateness of languages

> "A language supports a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely enables the programmers to use the technique. For example, you can write structured programs in Fortran and type-secure programs in C, and you can use data abstraction in Modula-2, but it is unnecessarily hard to do so because those languages do not support those techniques." (Stroustrup, 1988)

Stroustrup does not describe exactly what "unnecessarily hard" or languages that "support techniques" mean. He uses in this case three 'domain – language' pairs: structured programming – Fortran, strong typing – C, and data abstraction – Modula-2. In this case the 'domains' are close to programming, but in the more general cases, when the domains resemble the real-world, the presented situation is essentially similar.

The same phenomenon, namely that programmers use a wide variety of ways for encoding abstractions in programs, is well known also in the context of reverse engineering:

> "It is a trivial realization that the programming language has to provide means to express what we want to detect. However, these means can be very rudimentary. Talented programmers are thoroughly able to simulate higher concepts with only little support by the language." (Koschke, 1999, p. 102)

Furthermore, the following situations can occur when implementing (domain) concepts:

> "[...] the concept is not properly defined because the programming language does not provide the means to model this concept or a programmer has ignored these means [...] if the language does not provide adequate means of expression, programmers find ways to simulate these means in part" (Koschke, 1999, p. 104)

Summary. The current state of the practice requires the usage of a particular (a priori given) language for modeling or implementing a certain aspect of a domain. Many times the given languages are not appropriate to the domain that they describe. Domain appropriateness of a language is affected by the *generality* and the *semantic bias*. On the one hand, a language that is too general can express different models in a manner that does not allow their distinction in the language – we call these situations *diffusions* (Section 5.4). However, this still allows the definition of a homomorphism between the domain models and their implementation in the language. On the other hand, a language that presents a semantic bias leads to different kinds of encodings of the situations that are non-conform to the bias – we call these situations *distortions* (Section 5.3). Both of these facts cause a big encoding step in which the domain concepts need to be twisted and compressed in order to be expressed in the language constructs.

Fact: The bigger the difference between the concepts directly supported by a language and the concepts that are implemented using that language, the more difficult it is to implement and recover the latter.

2.2.3 Interleaving and Delocalization

Beside the conceptual differences between different languages, the *loss of conceptualization* is also caused by *interleaving* of multiple dimensions of knowledge in the code and by the *delocalized* implementation of domain concepts in the code.

2.2.3.1 Interleaving

One way to investigate the knowledge contained in a program is to study the program identifiers. Empirical studies on the meaning of identifiers in large systems, presented in (Anquetil, 2001), classify the identifiers from Mosaic[3] system in three domains: general domain (e. g. mathematics, general activities such as read or write), computer science domain (e. g. data structures) and the application domain (e. g. web browsing). Circa 20% of the identifiers were classified to refer to concepts that belong to the application domain, 10% to the computer domain and 60% to the general domain. However, the identifiers belonging to the computer science and the application domains are used more frequently than the identifiers belonging to the general domain.

During an empirical analysis of the knowledge needed for understanding a program, Clayton (Clayton et al., 1998b) remarks that the various dimensions of knowledge are woven in programs: the same part of code can refer both to concepts from the modeled domain and to software engineering and programming concepts. A single fragment of the source code can contain knowledge about the modeled domain, design, programming technologies and the programming language. This situation is known in the programming comprehension literature as the *interleaving problem* (Rugaber et al., 1995) which occurs whenever a contiguous textual area of the code contains different fragments that accomplish unrelated purposes. One of the causes of interleaving is the inability to decompose programs in multiple directions – also known as the "the tyranny of the dominant decomposition" (Tarr et al., 1999). Other causes are the integration between the developed program and other libraries (e. g. in order to be integrable in a particular framework, classes that implement domain concepts need to implement specific interfaces), or the implementation details introduced by the basic programming language functionality (e. g. many classes implementing domain concepts need to implement methods such as `hashCode`).

interleaving problem

Example 2.3: The source code weaves information from the application domain and the programming (sub-)domains

Figure 2.8 illustrates the multitude of information that can be found in a fragment of the source code. In the same code fragment can occur both information about the application domain (e. g. Person) as well as information produced in the development steps such as design (e. g. Visitor), information related to programming technologies (e. g. XML) or information related to the programming language (e. g. `clone`). A class can implement in the same time a domain abstraction (e. g. class Person), a design pattern (e. g. play the role of a subject in the observer design pattern), can contain methods for conversion to and from XML (e. g. for serialization) and methods specific to the language infrastructure (e. g. inherit the `clone` method from `java.lang.Object`). In order to understand a piece of code (e. g. for implementing a

[3]Mosaic is the ancestor of the Netscape web browser

non-trivial change request) the programmers need to have knowledge about each of these domains and about how are their concepts implemented in the code.

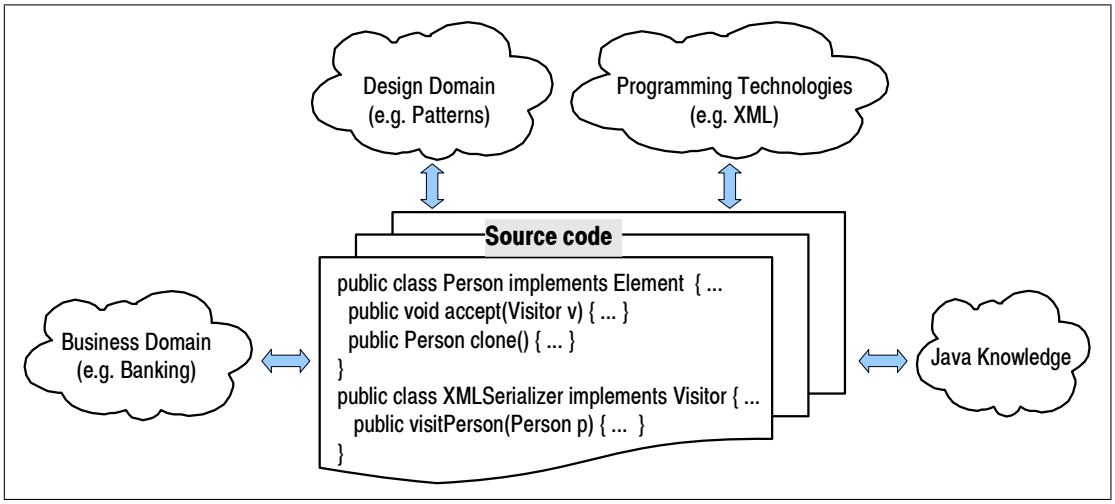


Figure 2.8: Multiple (sub-)domains are weaved (interleaved) in the source code

□

Fact: Knowledge about multiple domains is weaved in the same program fragment.

2.2.3.2 Delocalization

Brachman and his colleagues studied the typical activities done by maintainers during the maintenance of large systems at AT&T (Brachman et al., 1990). The study revealed the fact that up to 60% of the total effort spent by programmers is due to various searches in the code. The cause is the *delocalized information* in the system (firstly described in (Letovsky and Soloway, 1986)). *delocalization*
Pieces of information that logically belong together are implemented in different manners and spread. Consequently, the information for understanding a piece of code is spread across many files. One cause for the delocalization is the fact that the programs can be modularized in only one way at a time (aka. "tyranny of dominant decomposition" (Tarr et al., 1999)) and therefore clearly expressible concepts in the problem domain are spread across many modules or mixed together at the code level. Other causes of delocalization can be bad design, unexpected evolution or the need of integrating the system under development with existent technologies.

Example 2.4: Procedural and object-oriented decompositions

Below we present two program fragments written in Java that represent different decompositions in the implementation of figures and their functionality. The procedural decomposition (left) promotes the modularization along functionality and the object-oriented decomposition (right) promotes the modularization along data.

```
class Rectangle2D implements Shape { ... }
class Line2D implements Shape { ... }
class Graphics2D {
   void draw(Shape s) { ... }
   void fill(Shape s) { ... }
   void clip(Shape s) { ... }
}
```

```
interface Shape {
   void draw(Shape s);
   void fill(Shape s);
   void clip(Shape s);
}

class Rectangle2D implements Shape {
   void draw(Shape s) { ... }
   void fill(Shape s) { ... }
   void clip(Shape s) { ... }
}

class Line2D implements Shape {
   void draw(Shape s) { ... }
   void fill(Shape s) { ... }
   void clip(Shape s) { ... }
}
```

The different decomposition dimensions are illustrated below: the functionality dimension (on the left) and the data dimension (on the right).

```
draw
      Shape
      Line2D
      Rectangle2D
fill
      Shape
      Line2D
      Rectangle2D
clip
      Shape
      Line2D
      Rectangle2D
```

```
Shape
      draw
      clip
      fill
Line2D
      draw
      clip
      fill
Rectangle2D
      draw
      clip
      fill
```

In the object-oriented style the data is gathered together but the behavior is spread in more classes; in the procedural style the situation is vice-versa. These different modularization impact the maintenance and evolution. If during the evolution more figures are expected, then the object-oriented modularization is better since we can use the inheritance and polymorphism to add and use new classes for new shapes. In the case when during the program evolution more functionality will be added and the set of figures remains unchanged then the procedural organization is better since this will involve only localized changes.

□

Fact: The domain knowledge is dispersed in programs.

Conclusions. Due to todays technical constraints, clearly defined concepts at the domain level need to be expressed in pre-defined languages with a small set of constructs. The generality of implementation languages and their semantic bias involve both loss of abstract information about the modeled domain and addition of information in form of implementation details. The domain knowledge is interleaved in programs with knowledge about the implementation and is implemented delocalized in more modules.

2.3 Examples of Intentionality Loss

In the following, we present several examples from the literature that point out the problem of recovering the domain concepts (real-world meaning) from different artefacts produced in the software engineering process. Even if these problems have been acknowledged since a long time, currently there is no notion of (program) meaning that considers explicitly the domain concepts that these artefacts (intend to) implement.

Example 2.5: Comprehensibility of specifications

According to Liskov and Zilles (Liskov and Zilles, 1975), a specification is comprehensible when:

> "A person trained in the notation being used should be able to read a specification and then, with a minimum of difficulty, reconstruct the concept which the specification is **intended** to describe." (Liskov and Zilles, 1975) (our emphasis)

We remark in this definition that during the comprehension a person realizes a mapping between the specification and the concept that this describes. The concept that is only "intended to be described" means that the definition of the concept is not necessarily fully captured in the specification and a part of the meaning remains outside of the specification. The recovery of the intended concept is similar to a recognition process – based on the possibly incomplete information, we need to identify the concept.

□

Example 2.6: Formal specifications need to be interpreted in terms of their domain

van Lamsweerde (van Lamsweerde, 2000) gives a set of principles to be followed when writing formal specifications. Below we present an excerpt of these principles:

> "Formal specifications are **meaningless** without a precise, informal definition of how to interpret them in the domain considered. A formalization involves terms and predicates which may have many different meanings. The specification thus makes sense only if the meaning of each term/predicate is stated precisely, by mapping function/predicate names to functions/relations on domain objects. This mapping must be precise but necessarily informal (to avoid infinite regression)." (van Lamsweerde, 2000) (our emphasis)

We remark that the author considers that the meaning of formal specifications should be given by an informal, though precise, definition of how to interpret the terms and predicates in the domain. Here the term "meaning" is linked to a domain object (similar to a conceptual model) rather than a mathematical theory. Furthermore, "meaning" is treated distinctly from "semantics" – i. e. even if each specification has a well-defined semantics since it is written in a formally defined language, specifications can be "meaningless" with respect to the domain. The author refers to a meaning that is not covered by the mathematical objects. A possible approach for tackling the problem of having "meaningless specifications" is to rise the abstraction level at which the specifications are done towards specifications that are closer to the domain:

> "Specification techniques should move from functional design to requirements engineering; higher-level, **problem-oriented ontologies** must therefore be supported instead of program-oriented ones. [...] The terms in which problems have to be structured and formalized are programming concepts - most often, data and operations. It is time to raise the level of abstraction and conceptual richness found in informal requirements documents - such as, e. g. goals and their refinements, agents and their responsibilities alternatives, and so forth" (van Lamsweerde, 2000) (our emphasis)

□

Example 2.7: The "magic numbers" do not reveal their intentions

(Weissman, 1974) represents a pioneering work in empirical evaluation of the psychological complexity of programs. According to Weissman, the "magic numbers" are among the important factors that affect the understanding of programs:

> ""Magic numbers" are numbers whose **meaning** are not implied by their values. For example, DO I=1 TO 437 is perhaps less meaningful than DO I=1 TO TABLE_SIZE where TABLE_SIZE has previously equated to be 437." (Weissman, 1974) (our emphasis)

In this example, the word "meaning" denotes the intentions of the constants. In other words, the meaning in this context is given by the purpose of the constants, by the domain phenomena that they refer to. The mathematical meaning of 437 is clear but this number can be used to represent different real world concepts: the size of a table, the number of lines of a file, etc.

□

Example 2.8: The code structure does not entirely reflect the intent of the design

In the case of design patterns recovery, many of the state-of-the-art approaches are based exclusively on identifying the design patterns in terms of program structures. However, the design patterns have a design intent that is only incompletely (also weakly) captured by their structure. Well-known patterns with similar structure but different intent are: "Composite" and "Decorator" or "State" and "Strategy". Many of the relations between the components of patterns are determined mostly by the intent rather than by explicit language mechanisms:

> "There's no distinction in the programming language between aggregation and acquaintance. [...] Ultimately, acquaintance and aggregation are determined more by **intent** than by explicit language mechanisms. [...] Some patterns result in similar designs even though the patterns have different **intents**. For example, the structure diagrams of Composite and Decorator are similar." (Gamma et al., 1995) (our emphasis)

□

Example 2.9: Expressing the computational intent in "humans-oriented terms" and in "programming terms"

In their landmark work on concepts centered program comprehension, Biggerstaff and his colleagues (Biggerstaff et al., 1994) describe the information available in programs:

> "[...] it is qualitatively different for me to claim that the program "reserves an airline seat" than [...] to assert that 'if(seat = request(flight)) && available(seat) then reserve(seat, customer)'.
>
> Apart from the difference in the level of detail and formality, the first case expresses **computational intent in human-oriented terms**. [...] The first expression of computational intent is designed for succinct, intentionally ambiguous (i.e. informal) human-level communication, whereas the second is defined for automated treatment (e. g. program verification or compilation). Both forms of the information must be present for a human to manipulate programs [...] in any but most trivial way. Moreover, one must understand the association between the formal and informal expressions of computational intent.
>
> [...] human-oriented concepts such as [...] reserve airplane seat are decoupled from the formal patterns of their algorithms because they involve an arbitrary semantic mapping from operations expressed on numbers and data structures to computational intent expressed in terms of domain concepts (e. g. , [...] a seat). There is no algorithm [...] that allows us to recognize these concepts with complete confidence." (Biggerstaff et al., 1994) (our emphasis)

According to Biggerstaff, the computational intent can be expressed in human-oriented or programming oriented terms. The programming oriented concepts are signaled by the semantics of the language in which the program is written and can be inferred by analyzing the program. In contrast to this, the human oriented concepts appear in informal sources of information in a program (the most important one is the names of identifiers) and their identification requires a priori knowledge about the domain to which they belong. Furthermore, in order to perform any non-trivial operation on programs the programmers have to understand both these intents and the associations between them. The subject of this dissertation is to assign a humans-oriented meaning to programs.

Remark. Recovering the human-oriented intent is similar to the recognition rather than verification: the programmers suppose that the program part "reserves an airline seat" without any proof that this actually happens! If we want to be more precise we should re-write the above sentence to "tries to reserve an airline seat".

□

Example 2.10: Reflecting dependencies from the domain in the code

In an introductory book on programs design, Felleisen (Felleisen et al., 2001)[p. 36] presents two examples of code for calculating the profit obtained by selling tickets for an event. Regarding these fragments, Felleisen notes:

```
;; How to design a program
(define (profit ticket-price)
  (- (revenue ticket-price)
    (cost ticket-price)))

(define (revenue ticket-price)
  (* (attendees ticket-price) ticket-price))

(define (cost ticket-price)
  (+ 180
     (* .04 (attendees ticket-price))))

(define (attendees ticket-price)
  (+ 120
     (* (/ 15  .10) (- 5.00  ticket-price))))
```

```
;; How not to design a program
(define (profit price)
  (-  (* (+ 120
            (* (/ 15  .10)
               (-  5.00  price)))
        price)
     (+ 180
        (* .04
           (+ 120
              (* (/  15  .10)
                 (-  5.00  price)))))))
```

Source: "How to design programs", by Matthias Felleisen, page. 36

Figure 2.9: How (not) to design programs (Felleisen et al., 2001)[p. 36]

> "[...] it is easy to check that the two profit programs [...] produce the same profit when given the same ticket price. Still, it is also obvious that while **the arrangement on the left conveys the intention behind the program directly**, the program on the right is nearly impossible to understand. Worse, if we are asked to modify some aspect of the program, say, the relationship between the number of attendees and the price of the ticket, we can do this for the left column in a small amount of time, but we need to spend a much longer time for the right one." (Felleisen et al., 2001)[p. 36] (our emphasis)

The right program fragment from Figure 2.9 "conveys the intentions directly", since the structure of the program and the program abstractions mirror the application domain concepts. The measure in which the program mirrors the domain structure affects the extensibility of the program. In Section 5.2 we will discuss the conceptual coverage and extensibility of programs.

□

Intentionality is lost in all phases of the software engineering process and is manifested in the difficulty to discover the domain concepts once they are expressed in a computer language.

2.4 Reverse Engineering is in Search of the Lost Abstraction

In the previous section we pointed out several examples of intentionality loss when domain concepts are expressed in computer languages. In this chapter we present several approaches for recovering the lost (domain) abstractions. One of the most frequently cited definitions of *reverse engineering* is:

reverse engineering

> "Reverse engineering is the process of analyzing a subject system to

1. identify the system's components and their interrelationships and
2. create representations of the system in another form or at a higher level of abstraction." (Chikofsky and Cross, 1990)

From this definition, we remark that the recovery of abstract information from the code is one of the major goals of reverse engineering. What does "abstract information" mean and how "abstract" can it be?

2.4.1 Knowledge Needs During Maintenance

Antequil presents results of an empirical study about the knowledge needed by maintainers (Anquetil et al., 2003). The study was performed by observing programmers during the maintenance tasks and recording the atoms of knowledge they needed (think-aloud-protocol). The study identified 12 dimensions of knowledge divided in 3 categories:

knowledge dimensions

- **computer science knowledge:** programming (e. g. paradigms), programming language (e. g. syntax), development environment (e. g. tools used in development), application implementation (e. g. how is the application implemented), diagramming (e. g. knowledge on design techniques), organization's programming rules (e. g. naming conventions)
- **business knowledge:** application functionalities (e. g. what application does), problem analyzed (e. g. knowledge of the situation at hand), production environment (e. g. using some databases)
- **general knowledge:** additional tools (e. g. web browser), help (e. g. manuals), other knowledge

Maintainers made use of little business knowledge (12%) and general knowledge (13%) and used a lot the computer science knowledge (74%). Furthermore, maintainers use little new knowledge and they rather try to formulate hypothesis starting from what they already knew.

Rugaber and his colleagues investigates the roles played by the domain knowledge in program understanding (DeBaud et al., 1994; Clayton et al., 1998a; Rugaber, 2000). A domain model offers the reviewer a set of domain constructs (e. g. representations of real-world objects such as persons or accounts, of algorithms such as tax calculation) to look for in a program. A domain model can also act as a schema for controlling the program understanding and for organizing the analysis results – for example if the program reflects the modeled domain, the relations among domain concepts are hints for relations at the code level between their corresponding implementations.

role of domain knowledge

Ramal and his colleagues (Ramal et al., 2002) present their results after conducting an empirical study on the knowledge needed by developers for performing maintenance activities. The authors of this study came to the (disturbing) conclusion that the maintainers make little use of the knowledge about the business domain. An explanation for this fact is the extremely high level of details of a program and that most of the information needed for performing local maintenance tasks (that were observed in this study) is related to the programming and to the intricacies of implementation. The fact that the needs for business domain knowledge is rather restricted in comparison to the needs for programming and implementation knowledge when

performing maintenance tasks, is confirmed also by other studies. For example, Sillito (Sillito et al., 2006) presents the result of an empirical study as a set of questions that the programmers ask themselves when performing software evolution tasks. Relatively few of these questions are strongly related to the representation of domain concepts in the code; the other questions are related to understanding the intricacies of the implementation.

The multiple dimensions of knowledge weaved in a program and the intricacies in the implementation of domain knowledge in code are major burdens for maintenance.

2.4.1.1 Expressing the "meaning" in the program understanding community

In the literature of reverse engineering, the units of (domain) knowledge come along in different forms:

- **Clichés, plans:** represent knowledge about how typical programming algorithms (e. g. searching, iterating) are implemented in the code (Soloway and Ehrlich, 1984; Yu and Robertson, 1988). Many of the early program understanding approaches are focused on the recognition of plans in programs by using knowledge bases that contain programming plans. However, plans capture and express only functional abstractions and are not abstract enough to express domain information. For example, the approach of Harandi and Ning (Harandi and Ning, 1990) supports understanding by matching a program to a set of plans defined as patterns over the code and that are saved in a knowledge base (e. g. such a plan is the "forward-map-enumerator").

- **Concerns:** (Robillard, 2003, p. 1) defines concerns to "refer to any consideration a developer or team of developers might have about the implementation of a program". In (Robillard and Murphy, 2007) the concerns are defined to represent: "anything that stakeholders of a software project may want to consider as a conceptual unit. Typical concerns in a software project include features, nonfunctional requirements, design idioms and implementation mechanisms (e. g. caching)".

- **Business rules:** "A requirement on the condition or manipulation of data expressed in terms of the business enterprise or application domain" (Selfridge et al., 1993). Business rules are thus high-level knowledge about different conditions and manipulation of data in the application domain and that are desired in a particular project. They comprise a wide variety of conditions such as integrity constraints, computation rules or rules about the ordering of activities in a process.

- **Features:** are defined as requirements of a program that a user can exercise and which produce an observable behavior (Wilde and Scully, 1995; Antoniol and Gueheneuc, 2006). Domain concepts that belong to the functional requirements of a program are called "features" (e. g. add customer, create new account). Feature identification represents the identification of program parts that are activated when one of the features of the program are exercised. The typical manner of exercising the program features is through the user interface.

- **Concepts:** refer to general entities and can belong to the programming or business domain (Biggerstaff et al., 1993; Rajlich and Wilde, 2002). We will focus in the following on concepts-centered approaches for reverse engineering. In Section 3.2 we propose the usage of domain ontologies to specify the (domain) meaning of concepts. A concept belongs to a domain ontology and its meaning is given via its neighbours (i. e. concepts) from the domain ontology.

We remark that all these notions denote in a way or another different kinds of abstract knowledge: *clichés* represent knowledge very close to the implementation, *business rules* are knowledge about the dynamic of the system or integrity constraints, *features* are domain concepts that are accessible by the user through the program's interface.

2.4.2 Concepts Centered Program Understanding

The cognitive models of program understanding study the mental models that the programmers develop during comprehension. A *mental model* is a representation of the program under investigation and consists of different kinds of information at different abstraction levels (e. g. information about the program text, program structure, or about the application domain). The models of understanding study how do programmers increase their understanding of the program by exchanging information between different abstraction levels.

mental model

The concepts centered comprehension model (Rajlich and Wilde, 2002) is based on the assumption that in large software projects it is not feasible to assume that the programmers can know, read or understand the entire program or that they can have competence in all aspects of the application domain. Many maintenance tasks are goal driven and oriented on how specific concepts are reflected in the code. For example, for implementing a change request, programmers need to know how the concepts that are referred by the change request are implemented in the code. According to the concepts centered comprehension model, programmers do not comprehend the entire programs, but it is sufficient to locate concepts in the code. This operation is called *concept location*. The opposite operation namely of linking a piece of code with the concepts that it implements is called *concept assignment*:

concepts-centered understanding model

> "concept assignment is a process of recognizing concepts within a computer program [...] A degenerated case of this recognition process is the familiar process of parsing programming language for compilation." (Biggerstaff et al., 1994)

Rajlich emphasizes the problems of intricacies in the implementation and on linking concepts to code – "Frequently in program comprehension the programmer understands domain concepts, but not the code". This observation correlates well with the empirical observations made by Ramal and his colleagues that the maintainers are challenged mostly by the intricacies of the implementation of domain knowledge (e. g. interleaving, delocalization) and not about the domain knowledge per se (Ramal et al., 2002) – Section 2.4.1.

Remark. The programmers need to a priori know the concepts they are looking for (Biggerstaff et al., 1994; Rajlich and Wilde, 2002).

The concepts-centered program understanding shows that there exists a correspondence between program parts and the domain concepts.

2.4.3 Definitions and Descriptions of Concepts

In the literature of reverse engineering and program comprehension are several definitions of the term "concept" as shown in the following paragraphs.

Concepts as "units of human knowledge". Rajlich and Wilde (2002) give the following definition of concepts:

> "Concepts are units of human knowledge that can be processed by the human mind (short-term memory) in one instance" (Rajlich and Wilde, 2002)

Rajlich includes in his definition both domain concepts such as "credit card" and concepts related to implementation such as "I/O Error". This definition of concepts has the following limitations:

1. it does not allow to make the concepts explicit in order to be shared among the participants (different people can process different units of knowledge in one unit),
2. it does not allow to automate the concepts-based program analyses.

Concepts as words. There is a significant body of research in reverse engineering that considers that "concepts" are words contained in the names of identifiers, in the comments or in the names of files. For example Anquetil and Lethbridge (1998a) use the words contained in the names of source code files to obtain a conceptual decomposition of the software system: the files that contain the same word in their name belong to the same conceptual module. The definition of concepts simply based on words has the following limitations:

1. it does not consider the situations of naming ambiguities (synonymy and polysemy),
2. it does not consider the concepts that are described through multiple words (aka. compound words), and
3. it does not consider the structure of the conceptual space.

Concepts as clusters of words. The reverse engineering approaches based on Latent Semantic Indexing compute clusters of words of program identifiers based on some similarity metric (e. g. words that recurrently occur together belong to a cluster). According to these approaches, these words clusters represent concepts (Kuhn et al., 2005; Maletic and Marcus, 2001).

Example 2.11: Concepts described by clusters of words (LSI)

Below we present several examples of concepts described by identifiers, as are presented in (Kuhn et al., 2005). These clusters were identified by analyzing the words that form the identifiers of JEdit and that recurrently occur together.

1. cell, renderer, pane, scroller, frame
2. menu, VFSBROWSER, popup, show, adapter
3. key, stroke, traversal, bindings, event
4. directory, dir, file, interrupted, install
5. run, request, runnable, later, thread
6. plugin, unload, dependencies, deactivate, jar
7. area, display, manager, range, text
8. dirty, redo, position, undo, bh
9. font, hints, paint, opaque, metrics
10. mymatch, substitute, RE, sub, expr

□

The definition of concepts as clusters of words that recurrently occur together has the following drawbacks:

1. it offers a weak and ambiguous definition of concepts. The interpretation of concepts depends exclusively on the human who reads the cluster of words. Two different persons can understand different concepts – e. g. is the first cluster from above referring to TABLE or to more general graphical containers?

2. the concepts are isolated and there is no structure among them.

3. the relation between concepts and program parts is weakly defined since the program parts are regarded as flat text documents. By linking a cluster to a program part it is not clear how is the concept implemented in that part.

Concepts in formal concept analysis. The notion of "concept" also occurs in reverse engineering with the use of formal concepts analysis (FCA) (Tilley et al., 2003; Arévalo et al., 2005; Eisenbarth et al., 2003). FCA takes as input a set of objects, each object having its own attributes. These objects are clustered according to their common attributes. A concept is a maximal collection of objects that share common attributes and is described through a pair of sets – the set of objects (its extent) and the set of attributes (its intent). The concepts are arranged hierarchically in a concept lattice. Each node of the lattice represents a concept and the concepts are arranged in an is-a hierarchy. The concept that contains a set of attributes A has as subconcepts all those concepts that contain a set of attributes B that is included in A.

Example 2.12: Computing a concepts lattice using FCA

In Figure 2.10 we present an example of performing concepts analysis on a set of persons (acting as object) and a set of preferred sweets (acting as properties). The relation between the objects and their attributes is presented in the table on the left-hand side and the concepts lattice on the right-hand side. For example, we have the concept described by the pair $(\{Adrian,\ Stefan\}, \{ice\ cream,\ cake\})$ that contains the objects $Adrian$ and $Stefan$ that have the common attributes $ice\ cream$ and $cake$.

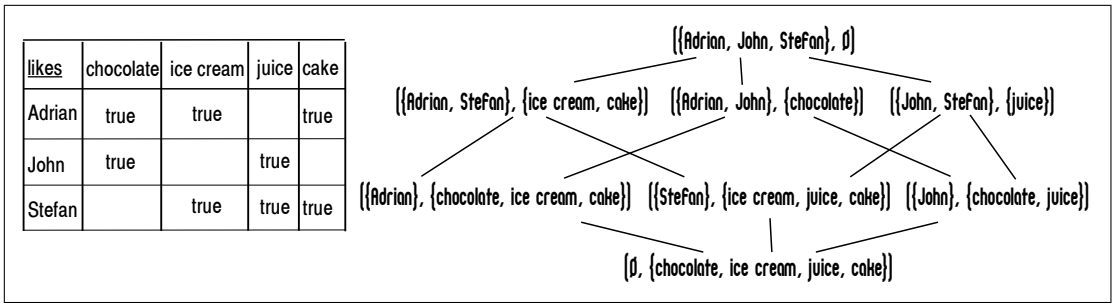


Figure 2.10: Example of a concepts lattice

□

The definition of concepts with FCA has the following limitations:

1. it does not assign the concepts a domain meaning. By simply considering that properties that occur together define a concept, we do not have any connection with the modeled domain.

2. it does not define a rich structure among the concepts (defines only the is-a hierarchy).

"Programming oriented" vs. "human oriented" concepts. In their landmark work on concept assignment, Biggerstaff et al. (1993) classify concepts into "programming oriented" (e. g. searching, sorting) or "human oriented" (e. g. customer, seat booking). The programming oriented concepts can be (mostly) recognized in the code in terms of sequences of steps that make up their algorithms and by using techniques similar to parsing. The programming oriented concepts are signalized by the language features or other information that can be algorithmically deduced from these features. In contrast to this, the human oriented concepts can be recognized by using the informal information contained in the program.

> "the recognition process depends heavily upon *a priori* knowledge about the application domain, the domain entities that are typically important and the typical relationships among them." (Biggerstaff et al., 1993)

Even if the work of Biggerstaff is the closest to our research, he does not define exactly what concepts are and does not detail on how could human concepts be made explicit. In our work we consider a framework for describing concepts as entities of domain ontologies. The meaning of concepts is made explicit by their place in the domain ontology (Section 3.2).

In all of the above approaches the meaning of concepts is (at most) weakly defined. This fact does not allow the unambiguous interpretation of concepts, their sharing between different programmers, or the automation of conceptual analyses.

2.4.4 Approaches for Making the Domain Concepts Explicit

In the reverse engineering literature are different approaches for making the domain knowledge explicit. These approaches vary in their formalization and in the typical knowledge that they can represent – e. g. on the one hand, by using computation patterns we can formally specify programming concepts; on the other hand, by using textual documentation we specify a much wider category of concepts but only informally and poorly structured.

Using computation patterns. The programming oriented concepts (e. g. sorting, accumulating, interchanging, lists) can be made explicit through patterns of computation, called *program plans*. The plans are saved into libraries (also called "plan bases") (Harandi and Ning, 1990).

Using graphs containing higher level information. The knowledge about the structure of a system can be captured by using different kinds of high-level diagrams (e. g. class diagrams) that describe different views of the architecture. The meaning of programs is then given by mapping the modules contained in these diagrams on the parts of the code (Murphy et al., 1995).

However, Murphy does not define the meaning of these modules – but only that they are used and interpreted by software engineers to communicate about the structure of a system. A concrete usage of these diagrams is to describe the design patterns in order to document the design rationale for a code fragment (Baniassad et al., 2003).

In their work on domain based reverse engineering (Clayton et al., 1997), Rugaber and his colleagues use the Object Modeling Technique (OMT) (Rumbaugh et al., 1991) for representing a conceptual model of the application domain.

Using ontologies. (Devanbu et al., 1990) is a pioneering work in reverse engineering that uses ontologies for representing the domain knowledge. Hsi et al. (2003) represent the knowledge that is implemented by an application with the help of an ontology that is built by manually analyzing the graphical interface (GUI) of the application.

Using textual documentation. The most common way to make the knowledge about a system explicit is to write textual documentation. Even if this is easily interpretable by humans, the natural language texts (due to their ambiguity) are difficult to access and manipulate by the machines. Many reverse engineering approaches define the meaning of programs by using traceability links between the source code and textual documentation (Antoniol et al., 2002; Witte et al., 2007).

2.4.5 Approaches for Assigning (Conceptual) Meaning to Programs

In the following we present several approaches to assign conceptual meaning to programs by mapping the code to different kinds of higher-level information.

2.4.5.1 Recovering programming plans

Programming plans (Soloway and Ehrlich, 1984) are stereotypic fragments of programs that experienced programmers use to implement typical algorithms (e. g. sorting, searching, accumulating a value). Programming plans can be automatically recognized by matching (patterns-based matching) their representation to parts of the analyzed programs. The recognition is similar to parsing. Harandi and Ning (1990) use a pattern-directed inference engine for performing and combining advanced mappings of plans to the code. Wills (1993) proposes a flexible approach for recovering programming plans by using graphs parsing. The flexibility resides in the possibility to vary the accuracy of the recognition (e. g. to allow near-miss recognition) by using different heuristics based on different analysis needs. There are two fundamental limitations of the recognition of plans: Firstly, all these approaches assume that all programmers implement plans in a similar manner. Secondly, the programming plans capture only knowledge very close to implementation (e. g. programming idioms) and ignore the other kinds of knowledge like about the business domain or knowledge about implementation technologies such as XML.

programming plans

2.4.5.2 Bridging domain models to architectural program abstractions

Clayton et al. (1997) propose a program understanding approach that takes as input the source code, a description of the application domain, and various sources of programming knowledge. The output contains three elements: a refined domain model, an abstract program description

(architectural abstraction of the program) and a mapping between the program description and the domain model that describes how the code implements the domain model. This approach has the following limitations: firstly, it defines the domain meaning of the program architecture and not of the source code itself; secondly, it is high granular since it maps domain knowledge on the architecture; thirdly, it does not specify clearly the semantics of mappings; and finally, it assumes that such a mapping is always possible (and thereby assumes a correspondence between the architecture and the domain model). The latter limitation is very pregnant in the presence of delocalization (DeBaud, 1996) and interleaving.

(Gall et al., 1996) proposes a method for transforming the procedural architecture of a system to an object oriented architecture by making use of domain knowledge. Without the domain knowledge, the transformation would be ambiguous – the procedural program structures could be interpreted in different ways in an object-oriented manner. During their work to transform the procedural into object-oriented architecture, Gall and his colleagues manually assign domain meaning to the program structures (semi-)automatically extracted from the code.

2.4.5.3 Reflexion models, intentional views and concern graphs

software reflexion models

In order to make the relation between the high-level models and the source code explicit, Murphy et al. (1995) propose the *software reflexion models*. Figure 2.11 presents the general approach for building the reflexion models: in the first step, a programmer defines high-level models of the structure of the software system; in the second step, the programmer chooses an abstraction of the source code (typically a kind of graph such as call graph or inheritance hierarchy) that is appropriate to be mapped to the high-level model and in the third step the programmer specifies how the models are to be mapped to the source code. All these three ingredients, namely the high level model, the mapping strategies and the code abstraction, are used as inputs of a tool that computes the reflexion of the high-level model in the code. Using the reflexion, the programmer can inspect where the code agrees or does not agree with the high-level model. The programmer interprets the reflexion and iteratively computes additional (more detailed) reflexion models. The approach described by Murphy is very general and can be used to map a wide variety of abstract models to programs. The high-level models used by Murphy and colleagues are mostly structural and they mostly target the architecture of the system and not domain knowledge.

Example 2.13: A simple reflexion model

In Figure 2.12 we present an example of a reflexion model. In our example, the high-level model represents the general 3-tier layered architecture: presentation layer, business logic and persistency. In this layered architecture there are links only between the presentation and the business logic and between the business logic and the persistency. In the lower part of the Figure 2.12 we define a set of mappings (based on the names of packages) for mapping the source code parts to the high-level model. Once the reflexion model is obtained, it can be compared to the original high-level model: we notice two kinds of differences in form of not wanted dependencies (between the presentation and persistency layers) and of missing dependencies in the code (between the business logic and the persistency layers).

□

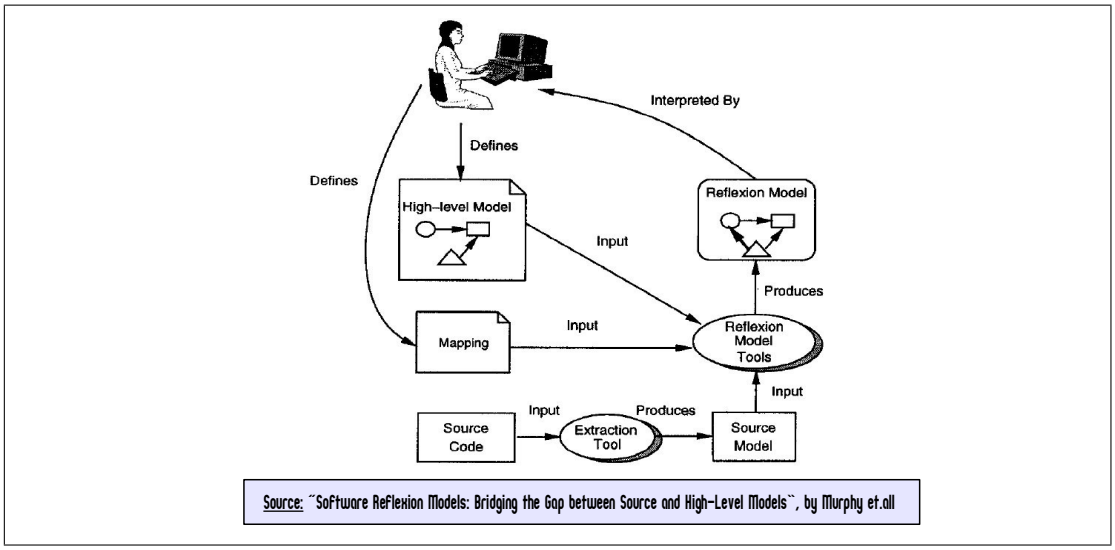


Figure 2.11: Software reflexion models overview (Murphy et al., 1995)

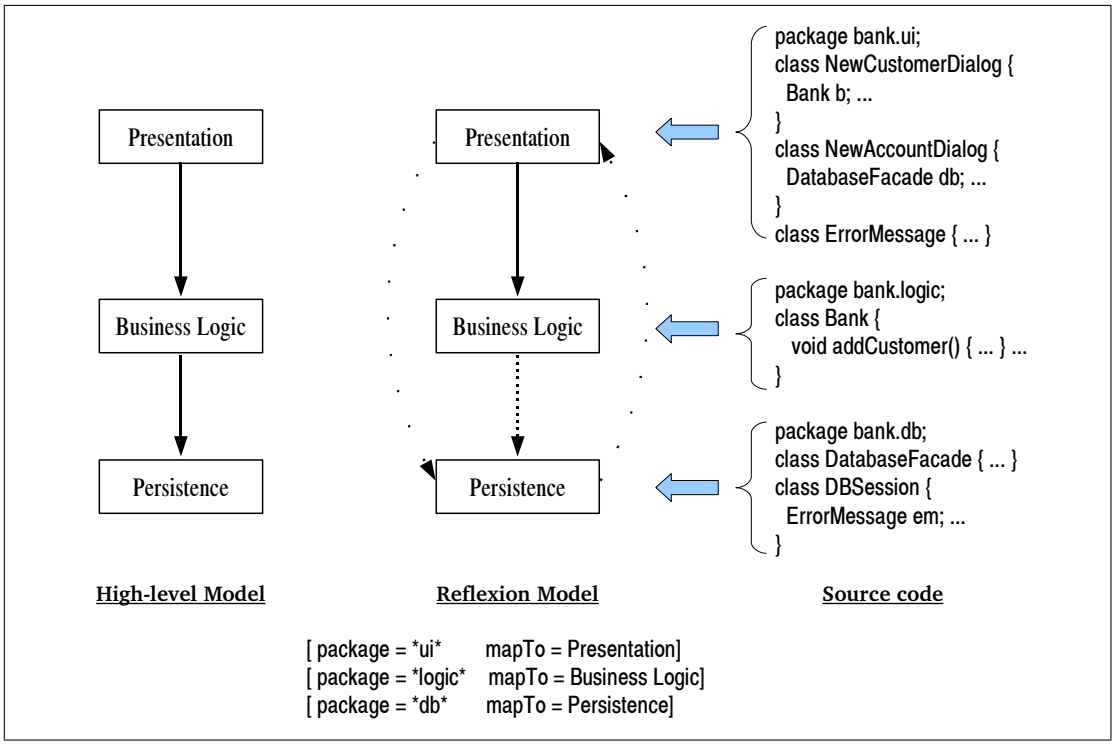


Figure 2.12: Example of defining, computing and using the "reflexion models"

Concern graphs. An extension of the reflexion models are the *concern graphs* (Robillard and Murphy, 2002; Robillard, 2003). A concern graph contains program elements and program relations among them that are relevant for implementing a concern. The goal of concern graphs

is to make the dependencies between contributing program elements explicit. A concern graph, mostly built manually, represents another decomposition of the program. As we showed in Section 2.4.1.1, a concern is "anything that can be considered as a conceptual unit". This definition is very general and therefore weakly connected to the real-world semantics of a domain.

Intentional views. Mens et al. (2003) develop intentional source code views to make explicit the conceptual structure of software systems and to provide means for reasoning about the abstract information contained in the code in an explicit manner. An intentional source code view is a set of related program entities (such as classes, instance variables, methods) that is specified by one or more alternative descriptions written in a declarative (meta-programming) language. These descriptions reflect the commonalities shared by the program entities and thereby they present the intention that is common to all of these elements. The descriptions of views are similar to the high-level models of Murphy et al. (1995). The intentional views can be used to define and compute a *conceptual structuring* of the code – the conceptual structure represents how we conceptually think of the code to be structured as opposed to how it is structured in the reality. Another example of use of intentional views is to check the architectural conformance of a software system (Mens, 2002) – i.e. whether the implementation corresponds to the abstract view of the architecture of the system. For example, one can discover and prevent the direct access of the data persistency layer of an application from the code that deals with the graphical user interface.

Example 2.14: Defining an intentional view

Below we present the definition of an intention through an example adapted from (Mens et al., 2003). In this example, the intention "uiClasses" is defined to contain all classes that belong to packages that contain in their names the substring "ui".

```
intention(uiClasses, byPackage, ?Class) if
    classInPackage(?Class,?Package),
    containsSubstring(?Package, "ui")
```

□

2.4.5.4 Programs as information systems

Brachman et al. (1990) studied the knowledge needed and the actions taken by maintainers during the maintenance of large systems at AT&T. These studies revealed the fact that up to 60% of the total effort spent by programmers is due to various searches in the code and in the domain model. The cause for this is the delocalized information in the system – namely pieces of information that logically belong together are spread across many files, many times slightly modified. The early code searches were exclusively based on "grep" and "find" and this leads to a high amount of noise and low efficiency.

The study of Brachman and colleagues lead to the idea of regarding the software itself (the code) as data that needs to be searched. A software information system (SIS) provides a domain model which contains an explicit representation of the business domain knowledge and a code model that contains information about the implementation.

software information system

LaSSIE. Devanbu et al. (1990) present an approach for creating a software information system that integrates architectural, conceptual, and code views of a software system into a knowledge base. This integration allows to overcome the invisibility by formulating queries that combine different views (a query can contain information about the domain, architecture and code). The most common usage scenario of LaSSIE was performing (semantic) queries over the software and the domain. Since the LaSSIE knowledge base is based on knowledge frames, queries are performed through classification. The knowledge base of LaSSIE was based on two ontologies:

ontologies of LaSSIE

- The application domain ontology contained around two hundred concepts of the telephony domain (the "Definity 75/85" telephony switch). In Figure 2.13 (left) we present the upper-level concepts of this ontology. This ontology was built after the program was constructed through a "reverse knowledge engineering" by analyzing different development artefacts (code, documentation, requirements) and reverse-engineering the domain knowledge. The upper concepts from the domain ontology are repeatedly referred in the specification and documentation. Based on these concepts are defined (taxonomically) concepts that are closer to the telephony domain: cable, microphones, cable-trunks. The domain ontology is based on DOERS (things that are capable of performing actions), OBJECTS (things on which actions are performed), ACTIONS (represent systems' functional components) and STATES (the state of the system after an action is performed).
- The code ontology Figure 2.13 (right) is rather high-granular and contains only around twenty concepts but many instances. The code ontology is populated automatically with instances obtained after a code analysis.

Using these ontologies the programmers could make queries about the code specific information (e. g. "Where is the variable with name "cable" used in the program?") and about the domain concepts (e. g. "What is a dial-tone generation?"). In order to allow combined queries about the code level and domain information (e. g. "Which files implement the dial-tone generation?") the users of LaSSIE manually made mappings between these ontologies.

The major advantage of LaSSIE was the integration of domain and code ontologies in a manner that facilitates querying, browsing and the change of perspectives. However, LaSSIE offered an integrated view of (only) the domain knowledge and the code while ignoring the other kinds of knowledge that are contained in the programs (e. g. knowledge about programming technologies used). The most important drawback of LaSSIE is that although the code ontology is populated automatically, the domain ontology and the connection between it and the code needed to be maintained by hand and this was very expensive. The relation between the models degraded during the evolution and after a while the programmers did not trust the LaSSIE system anymore to provide accurate information. Such a system with advanced searching proved to greatly support comprehension tasks but the overhead of manually synchronizing the models reduced the overall benefit (Welty, 2003).

Comprehensive software information system. Welty (1995) developed the notion of "comprehensive software information system" (CSIS). CSIS advances with respect to SIS along three directions:

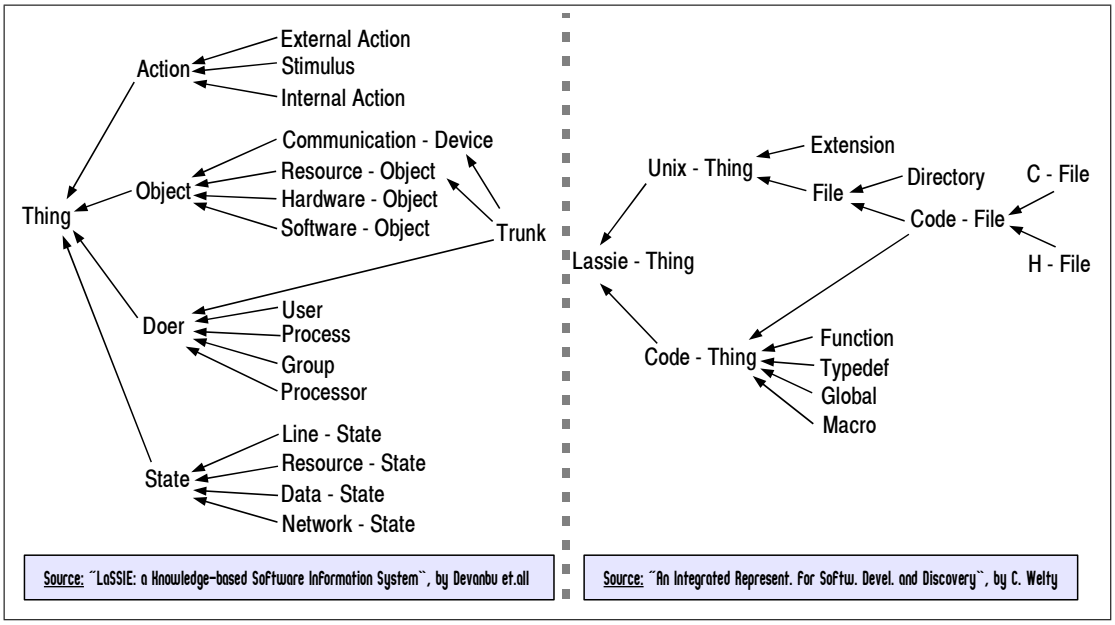


Figure 2.13: The upper parts of the code and domain ontologies used by LaSSIE (Devanbu et al., 1990; Welty, 2003)

1. it offers a richer ontology for the code model that is expressive enough in order to replace the need for programming languages (beside the basic structural information represented in SIS, CSIS represents also the information contained inside the methods),

2. it offers a deeper integration of the code and the domain models, and

3. it offers support for the development and modification of the code through the SIS.

A CSIS is a programming language that supports both the execution and the information retrieval. By developing programs using CSIS, programmers can take the advantages of having an information retrieval from up-front. However, from the point of view of analyzing already existent programs developed in conventional languages CSIS is basically a SIS and inherits all its disadvantages.

2.4.5.5 Recovering traceability between code and documentation

In the cases when the domain concepts are made explicit only as a textual documentation, we can assign meaning to program parts by linking them to their corresponding documentation fragments. There are several works in reverse engineering that use information retrieval techniques to map code parts to textual documentation (Antoniol et al., 2002; Marcus, 2003) and thereby to recover traceability links. This method to assign meaning to programs has two disadvantages: firstly, the traceability links are high granular (parts of programs are linked with parts (e. g. paragraphs) of the documentation); secondly, the semantic domain lacks structure (the documentation is a free text).

(Zhang et al., 2006; Witte et al., 2007) present an alternative approach for the recovery of traceability links between the source code and the documentation (Figure 2.14). In order to do this, the authors create an ontological representation of both source code and documentation and use semantic web technologies (ontology alignment) for mapping common concepts and instances of these two ontologies. The source code ontology is populated automatically by analyzing the sources and the documentation ontology is populated by analyzing the documentation with the help of natural language processing (NLP) tools. We need to remark that the ontologies are used here primary as a technology enabler: once the code and documentation are represented as ontologies (in a standard ontology language such as OWL[4]), semantic web specific techniques can be used to find the commonalities between them. The use of semantic web technologies is similar to the LaSSIE approach with the following differences:

1. instead of using a domain ontology (as in the case of LaSSIE), (Zhang et al., 2006) use the ontology of the documentation. This ontology contains a large number of concepts that are expected to be found in software documents and is automatically populated with instances extracted from the documentation by using NLP tools, and

2. the links between the code and the documentation ontology are established automatically through ontology alignment techniques.

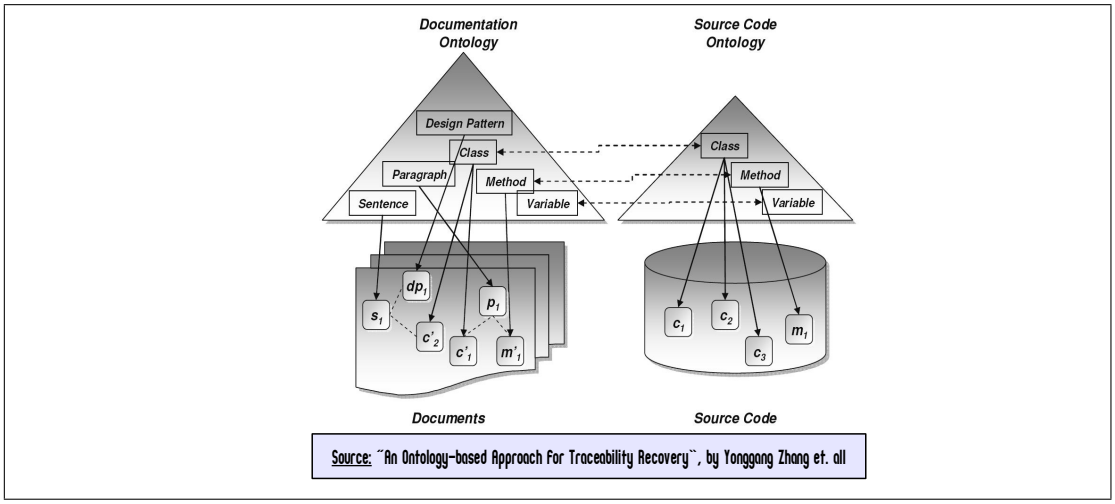


Figure 2.14: Overview over the ontology-based recovery of traceability between documentation and source code (Zhang et al., 2006)

[4] http://www.w3.org/TR/owl-features/

2.4.6 Summary

A central topic of reverse engineering is the recovery of abstract information contained in programs. The "abstract information" ranges from algorithmical knowledge (e. g. sorting), to architecture (e. g. components) and to knowledge about the business domain (e. g. account, customer). In Figure 2.15 we summarize the approaches employed in reverse engineering for giving meaning to programs by mapping the source code to higher-level knowledge. The approaches are classified along two directions:

- Firstly, according to the intentionality of information recovered, namely whether the abstract information is related to programming or business domains. For example, the intentional views or reflexion models are used mainly for recovering the structural information from the code, while LSI is used to recover the information that is closer to the domain.
- Secondly, according to the specification degree of the definition of abstract information and the granularity of the mappings, namely, the measure in which the domain meaning is captured and the precision of the mapping between the abstract information and the code entities. For example, in the case of reflexion models the high-level information is well defined and in the case of the LSI the knowledge is weakly defined (only as sets of words).

We remark that the current approaches either aim to recover information close to the programming domain (e. g. architecture), or lack structure in the level of specification of the knowledge in the case of more intentional domains that are farther from programming (e. g. business domain). A notable exception is LaSSIE (Devanbu et al., 1990) that used a rich representation of the domain (as an ontology) and precise mappings between the program elements and the domain concepts. However, the mappings between the LaSSIE ontology and the program are done manually, and the LaSSIE system did not contain information about the other dimensions of knowledge typically found in programs – e. g. programming technologies. Our work resembles mostly the LaSSIE approach and defines the domain meaning of programs by mapping them to domain ontologies.

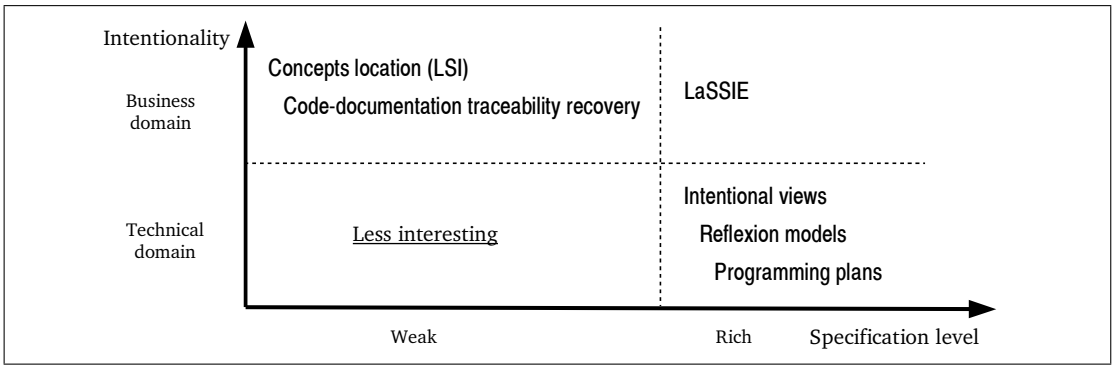


Figure 2.15: Spectrum of approaches for abstraction recovery in reverse-engineering

2.5 Towards an Intentional Meaning of Programs

Due to the big conceptual gap between the domain concepts and the programming languages constructs there is a strong and steep encoding of the domain knowledge in the code. As we presented in Section 2.2, during the forward engineering occurs a loss of abstract information manifested through the following sympthoms:

1. **Loss of conceptualization.** Concepts clearly distinguishable at the domain level need to be implemented through low-level programming constructs and this makes many times impossible their distinction at the code level.

2. **Encoding bias.** The use of inadequate (biased) languages or libraries, or the choice of unfortunate design decisions, leads to a high encoding of the representation of domain in the code and thereby to an ever increasing difference between the domain knowledge and how the program reflects it.

3. **Interleaving.** In single program parts is interleaved knowledge that belongs both to the business and to the programming domains. Distinguishing the knowledge atoms that belong to one dimension or to another is highly challenging.

4. **Delocalization.** Domain concepts are implemented in different parts of the code. Finding the (spread) code parts that implement a concept is highly desirable but very difficult.

These general sympthoms favour other mismatches between the domain knowledge and the code such as: logical redundancy, implementation details, or distortion of the implementation of groups of concepts in the code. Many programming and maintenance activities are strongly influenced by the mismatches between program parts and their business domains. In order to characterize the mismatches between the code and the domain, we need to analyze the code from the point of view of the domain knowledge that it implements. With other words, we need to systematically express the domain-meaning of programs 1) by using rich descriptions of the domain, 2) by mapping the domain description to the code in a fine-granular manner, and 3) by considering all dimensions of knowledge that are implemented in the code.

Our approach for assigning meaning to programs.

In Figure 2.16 we depict an intuitive view of our approach to define the meaning of programs in terms of their implemented domain concepts. There are two categories of actors that interact with programs: the program users and the program developers. The developers write programs that resolve problems from a particular domain and the program users use the programs according to their knowledge about that domain. The meaning of programs is described in the classical programming language literature with the help of programming language semantics – e. g. through a denotational theory. In order to analyze programs at a higher level of abstraction they need to be put in link with a humans-oriented description of the real-world. For doing this, we define the "intentional meaning" of programs as an explicit mapping between the program entities (e. g. classes, methods, variables) and the concepts of a domain ontology. The domain ontology is a product of knowledge engineering in the domain of interest and represents a body

of knowledge upon which both the program users and developers need to agree. The domain ontology is built by domain experts. By making the domain knowledge and the mapping of the program to the domain ontology explicit, we can characterize programs from the point of view of the concepts that they implement and thereby bridge the abstraction gap between the domain knowledge and the code. We use these mappings to characterize the faithfulness of the reflexion of domain knowledge in the code.

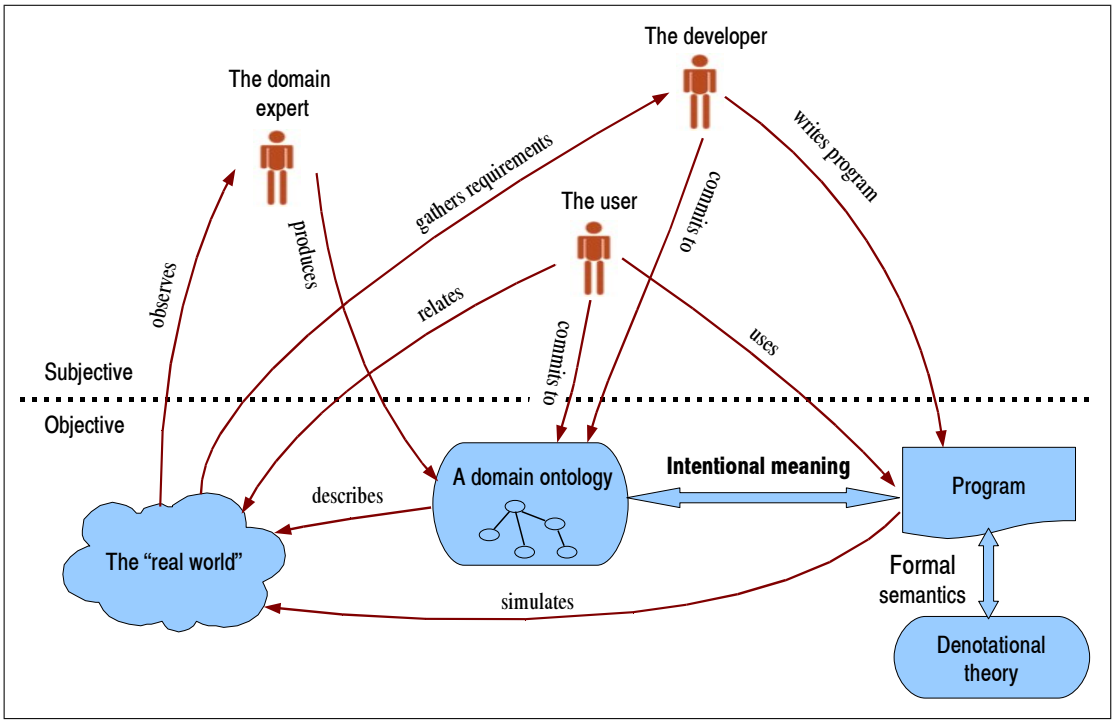


Figure 2.16: An overview of the intentional meaning

2.6 Summary and Roadmap

During the forward engineering process occurs a loss of abstract knowledge. Due to the big conceptual gap between the professional and the computer languages, domain concepts are encoded, commingled and interleaved in different degrees in all software artefacts with culmination in programs. Reverse engineering and program understanding approaches aim at recovering the abstract knowledge from programs. However, the current approaches either stop at the structural level or lack a clear definition of the domain information that is recovered.

In the next chapter we introduce the intentional meaning as an interpretation of programs with respect to a domain ontology. In Part III we use the intentional meaning to characterize the mismatches between domain knowledge and programs. In Part IV we present our approach to automate the recovery of intentional meaning.

3 Intentional Meaning of Programs

"In the works of man [...] it is the intention which is chiefly worth studying."

Goethe

Abstract: Due to the big conceptual gap between the application domain knowledge and programming languages, there is a myriad of possibilities to implement domain concepts in the code. Ideally, in order to be easy to understand, there should be a clear correspondence between program parts and the domain concepts that they implement. We call this correspondence the intent of a program part. Due to the intricacies of encoding of (abstract) domain concepts in (lower-level) programming constructs, the intent of a program part is not explicit and is many times lost and dissipated in a huge amount of implementation details. Instead of having a modularization that reflects the application domain, parts of programs are an amorphous and dense bulk of computation. In this chapter we define the meaning of programs with respect to the domain knowledge that they implement, and propose the use of ontologies as semantic domain for capturing the domain knowledge. We define the *intentional interpretation of programs*, by mapping program entities to concepts from a domain ontology that they implement. By doing this, we bridge the gap between the domain knowledge and programs, and we can interpret the source code from the perspective of the domain concepts that it implements. We show how does the intentional meaning relate to the programming language semantics and pragmatics. In order to enable a rigorous characterization of the correspondence between the domain concepts and the source code, we develop a formal framework that describes program elements, the domain concepts that they implement, and the relations between them. We further refine the intentional interpretation into more specific components: reference of concepts, definition of concepts, and representation of concepts in the code.

Structure of this chapter. After the introduction, in Section 3.2 we present domain ontologies as means to describe and share domain knowledge. Using the ontologies as semantic domains, we define the intentional meaning of programs as a function that maps program entities on concepts from domain ontologies (Section 3.3). Section 3.4 presents a formalization of programs and respectively domain ontologies as labeled graphs. The relation between program elements and the concepts that they implement is refined in Section 3.5 with the help of three functions: the reference of concepts, definition of concepts, and representation of concepts. Section 3.6 presents the mapping between the program and ontological relations. We end this chapter with a summary and an outlook on the next chapters.

3.1 Introduction

Once a piece of code is written, it is read many times, during a long period of time, by re-users or maintainers. Besides their role of interfaces between humans and computers, programs play an important role also in communicating among programmers. Thus programs can be regarded as works of humans targeted towards humans and therefore it is worth studying their intention.

Filling the conceptual gap between domain concepts and programming language constructs and the big amount of additional knowledge about programming technologies contained in programs, lead to approximations in reflecting the business domains in programs, to implementation details, and different distortions and diffusions of concepts. Due to the loss of conceptualization, clearly defined concepts in the problem domain can be hardly recognized in programs. Even the simple intent of a piece of code, in the sense of the domain concepts that it implements, is most of the times not explicit, but buried in a myriad of implementation details. In this chapter we propose a method to define the domain meaning of programs by explicit mappings between the program entities and the concepts from domain ontologies that they implement.

3.2 Specifying Meaning with Domain Ontologies

language semantics

The meaning of a language is given by a mapping between its syntax and a semantic domain. In a denotational style, the semantic mapping associates to each syntactic construct an object from the semantic domain. In the cases when the semantic mapping is compositional, the meaning of a program phrase is obtained by combining the atomic meanings of its atomic syntactic elements. Therefore, the semantics of programs written in a language is given in terms of the semantics of the language in which they are written and that is general enough to accommodate all programs that can be written in the language (Harel and Rumpe, 2004). The semantic domain specifies every concept that exists in the universe of discourse. By using a language, we cannot describe more than what is defined in its semantic domain. Conveying more information in a program, beside the one that is expressible in the semantic domain, can be done only by using an additional medium – e. g. domain information is communicated through the names of identifiers or documentation.

semantics of programs

> In this work we concentrate on the *domain-specific meaning of programs*. Therefore, our semantic domain should reflect the domain knowledge as close as possible to how humans understand it.

3.2.1 Real-World Meaning

Formal models are described by using logical frameworks such as First Order Logics (FOL). But FOL alone is not sufficient to describe a real world situation since FOL is ontologically free. Just like any general purpose programming language, FOL can be used to describe anything about anything. In order to describe a domain we need to fix a set of predicates, a set of functions and the universe of discourse.

Example 3.1: Simple formula in first-order logic

$$Woman(x) \equiv Person(x) \wedge Female(x)$$

In this sentence the symbols "(, $\equiv$, x" belong to the definition of FOL and the $Female$, $Person$, $Woman$ make the relation to a particular domain. The truth value of the formula depends on the universe of discourse, on the choice of predicates and on the choice of logical connectors (i. e. $Woman(x) \equiv Person(x) \wedge \neg Female(x)$ is obviously false).

□

A domain can be described in terms of a set of concepts. The important concepts that belong to the domain form the conceptualization of that domain. The same domain can be described from several points of view by using different conceptualizations (e. g. we can describe the information about the 'age' in two different ways: by using the concept 'age' or by using the 'birth date'). When FOL is used to describe a particular situation of the domain, one needs to decide on the sets of predicates – these predicates represent the *domain conceptualization* (Sowa, 2000). In the fields of semantic integration of information and semantic interoperability the word "semantics" means a mapping of objects in the model onto a domain conceptualization (Guizzardi, 2005, p.20) (Uschold, 2003). Domain conceptualizations are captured and shared in *domain ontologies*.

conceptualization

Terminological clarification. The word "ontology" has a multitude of meanings in the literature depending on the community that uses it (Guarino and Giaretta, 1995):

ontology

1) Ontology as a philosophical discipline. As a branch of philosophy, ontology is concerned with the nature and the organization of reality. It deals with entities that are general enough to be at the basis of more domains and with the organization of knowledge in its broadest sense. Typical philosophical questions with which the ontology deals are: What kinds of entities exist? What is the essence of things? What is being? What features are common to all beings? Is the world three or four dimensional?

2) Ontology as a conceptual system. According to this view, ontologies are conceptual systems which we may assume to underly a particular knowledge base. From this point of view, ontologies are similar to database schemas in the sense that they define what entities can be contained in the knowledge base.

3) Ontology as an "explicit, formal specification of a shared conceptualization". This definition is given in the seminal paper of (Gruber, 1995). The words "explicit" and "formal" mean that the ontology is expressed in a formal language whose semantic is explicitly defined. The fact that is "shared" means that the ontology reflects a common understanding of a certain domain that is assumed by a community of users. The users exchange information based on the vocabulary defined by the ontology; they commit to the ontology if they agree to use the vocabulary in a consistent manner. Sharing the same vocabulary does not mean that the ontology users share the knowledge since each user can know things that the others do not know.

4) Ontology as a representation of a conceptual system via a logical theory a) characterized by specific formal properties, or b) characterized only by its specific purpose. The first case means that an ontology is a logical theory of a certain kind (e. g. a terminological theory that describes concepts (called TBox) or an assertional one that describes individuals (called ABox)).

According to the sense b) an ontology is any logical theory and only its specific purpose (and not a particular form) makes it to be an ontology. For example, many researchers call "ontologies" – sense a) – any object that is represented in the OWL or RDFS languages.

5) Ontology as a vocabulary used by a logical theory. With other words, if a domain is described by a logical theory (e. g. a specification) then this logical theory will use predicates, relations and functions that are defined by the ontology. Guarino and Giaretta (1995) use the following definition of this sense of the word ontology: "A set of logical axioms designed to account for the intended meaning of a vocabulary."

We need to remark several differences between the interpretations of 'ontology' presented above. The first interpretation denotes a discipline of study while the following ones denote objects (artefacts). While in the second interpretation the object is a semantic description of a conceptual system, in the following ones the term ontology denotes a syntactic object. In the practice the term ontology is many times used in a polysemous manner. Below we give our definition of an ontology.

ontology

Definition 3.2.1 (Ontology)**:** *An* ontology *represents a conceptual model of a domain in form of named concepts arranged in a generalization / specialization hierarchy (taxonomy) and relations among them. The concepts and relations represent a consensual agreement on the domain by a category of domain experts.*

Discussion: 1) Our definition has two parts: the first part refers to the logical categories that can exist in a domain and the second part addresses a subjective agreement on these categories by the ontology users. We can regard an ontology as an instance of predicate logic with a choice of predicates whose names make the link with the natural language descriptions of a domain. Depending on the choice of predicates, we can express (or not) different parts of the knowledge about a domain.

2) Our notion of ontology is close to the definition given by Gruber (1995). We also regard ontologies to be specifications of a domain vocabulary (our concepts should have names) that is assumed by a community of users. The difference is that we do not require ontologies to be formally specified (but we do not restrict this). This difference is caused by the different use of ontologies: while Gruber uses them for knowledge sharing among agents (and therefore the semantics should be formally specified) we use them for program understanding (i. e. recognition of domain knowledge in programs) and based on this for evaluation of the clarity, adequacy and faithfulness of the implementation of domain concepts in programs. The intentional meaning of programs means the mapping of the program elements on the ontological entities that they (are supposed to) implement and not checking if the program elements implement the concepts.

3) The spectrum of details in the specification of ontologies is described in (McGuinness, 2003). From the point of view of the formalization degree, our understanding of ontologies is similar to conceptual schemas, or signatures of Σ-algebras. We call these ontologies to be "light-weighted". The interpretation of the entities from ontologies is given by the domain objects described through glossary entries.

light-weighted ontologies

4) Mylopoulos (1992) defines conceptual modeling to be "the activity of formally describing some aspects of the physical and social world around us for purpose of understanding and com-

munication". By the fact that our ontology is a "conceptual model" of the domain we require that each interesting domain concept to be explicitly reflected through an entity in the ontology. For example, if we are interested in the "family domain", the ontology needs to contain all the concepts FEMALE, PARENT and MOTHER explicitly, even if the concept MOTHER can be defined as a term composed from the intersection of concepts PARENT and FEMALE.

conceptual model

5) We require that ontologies represent an agreement among different domain experts (i. e. that they faithfully describe the knowledge from a domain). Prior to using an ontology, ontology users need to commit themselves to the ontology and take it as a faithful representation of the domain (Figure 3.1).

ontological commitment

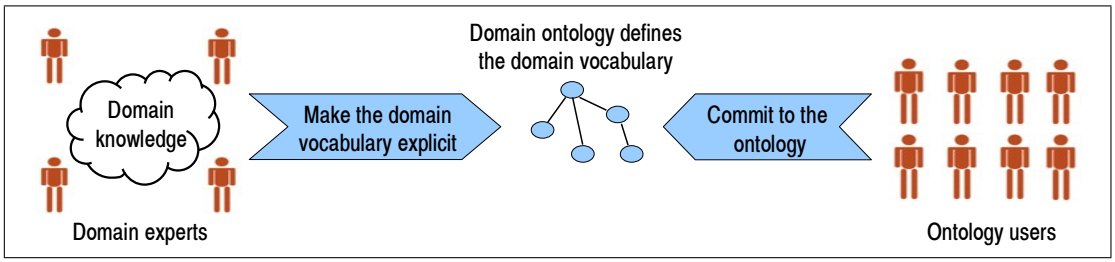


Figure 3.1: Ontologies share the domain knowledge by defining a vocabulary of the domain

Below are criteria for designing ontologies for knowledge sharing (Gruber, 1995):

criteria for good ontologies

- *Clarity:* The concepts defined in the ontology should clearly reflect the intended meaning in the domain (i. e. the ontology should represent a conceptual model of a domain).
- *Coherence:* The definitions in the ontology should be coherent (well-formed) – e. g. cycles of is-a relation are not allowed).
- *Minimal encoding bias:* The vocabulary should reflect the phenomena at the "knowledge level" and be independent of some specific encodings.
- *Minimal ontological commitment:* An ontology should have a clear focus and should define only the vocabulary that is necessary for a specific communication purpose. The users of an ontology that commit to it should agree only on the core information that is relevant for the actions they want to perform.

3.2.2 Examples of domain ontologies

Below we present several examples of fragments of domain ontologies. Each of these ontologies addresses a different kind of knowledge (e. g. business domain, design, Java), each of these kinds being representative for the knowledge contained in programs (Section 2.2.3.1) and needed during the maintenance (Section 2.4.1).

Example 3.2: An ontology fragment representing a family

In Figure 3.2 we present a domain ontology that contains concepts that describe the members of a family and their relations (the full arrows represent taxonomic is-a relations).

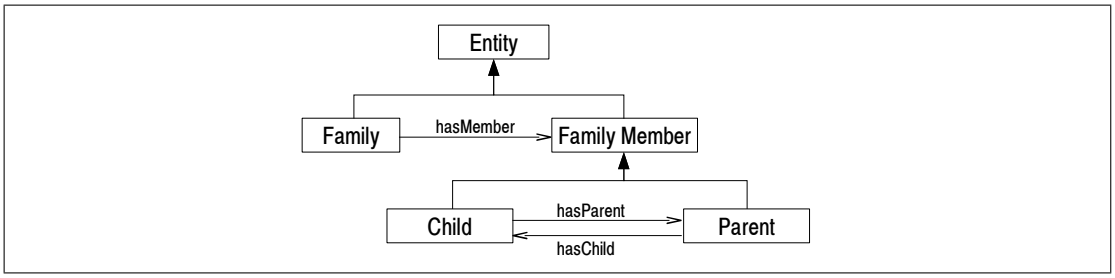


Figure 3.2: The family ontology

The meaning of the concepts is given as glossary entries in form of natural language descriptions as shown below:

Family	the basic unit in society consisting of two parents and their children
Family Member	a member of the family
Child	a son or daughter
Parent	a person who brings up and cares for another

□

Example 3.3: An ontology fragment representing a 'Pedestrian Traffic Lights Controller'

In Figure 3.3 we present a domain ontology that contains concepts that describe the behavior of a "pedestrian traffic lights" (PTL) system. This behavior can be modeled through a state automaton containing two states in which the traffic lights can be: RED and GREEN. These states are connected through two transitions: GREENRED and REDGREEN.

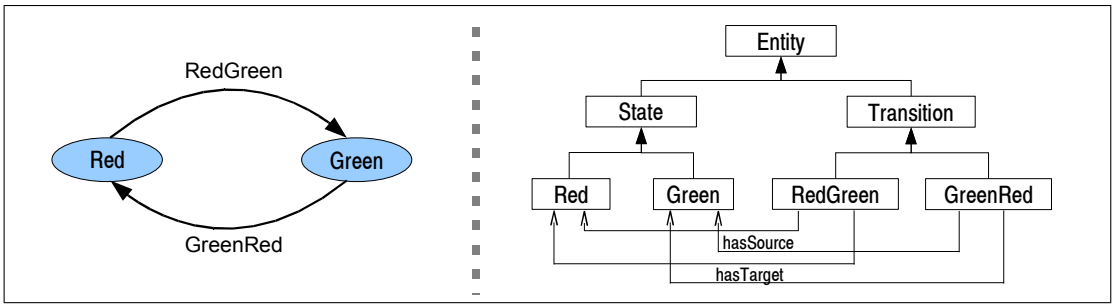


Figure 3.3: A state automaton that models a pedestrian traffic lights (left); a domain ontology fragment that represents the pedestrian traffic lights concepts (right)

The meaning of the concepts is given below:

State	the state of an automaton
Transition	the change of states of a state automaton
Red, Green	states of the pedestrian traffic lights system
RedGreen, GreenRed	transitions of a pedestrian traffic lights system

□

Example 3.4: A domain ontology representing the 'Observer' pattern

In Figure 3.4 we present a domain ontology that describes the "Observer" design pattern. There are two participants in this pattern: an OBSERVER and a SUBJECT. The observers register themselves to the subjects. Every time when the state of a subject changes, it notifies all its observers. The meaning of these concepts is described in prose text such as in the following (adapted from the description of Observer pattern in (Gamma et al., 1995)):

Subject	Provides an interface for attaching and detaching Observer objects.
Register	Adds a new observer to the list of observers observing the subject.
NotifyObservers	Notifies each observer by calling the notify method in the observer, when a change occurs.
Observer	Defines an updating interface for objects that should be notified of changes in a subject.
Notify	Updates the information the Observer has about the Subject's state.

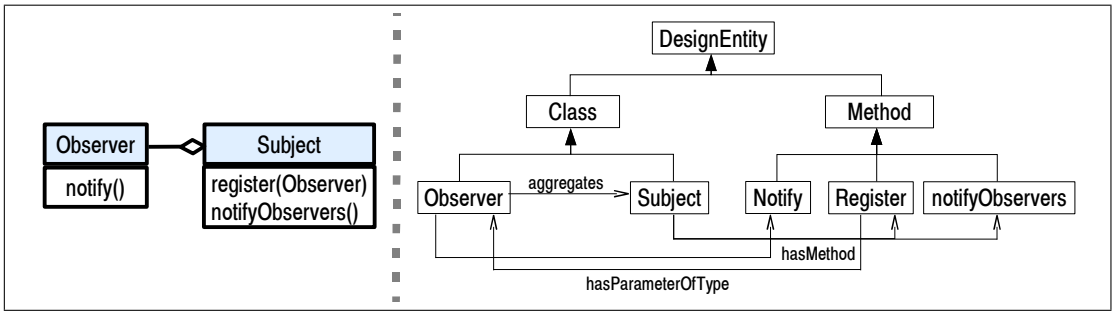


Figure 3.4: The "Observer" design pattern (left); a design patterns domain ontology that represents the "Observer" design pattern (right)

□

Example 3.5: A domain ontology representing the core constructs of Java

In Figure 3.5 we present a domain ontology fragment, in this case the "domain" is (a part of) the Java language. The concepts of this domain are the constructs of the Java programming language.

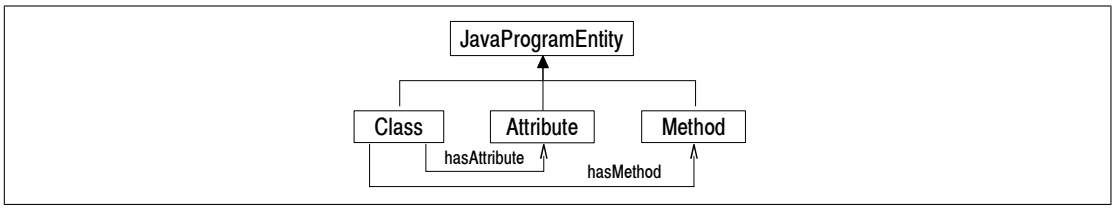


Figure 3.5: Fragment of the Java ontology

□

3.3 Intentional Meaning of Programs

Implementation versus interpretation. Figure 3.6 illustrates that for performing an implementation task, developers use a wide variety of knowledge belonging to several dimensions such as: knowledge about the business domain of the application, knowledge about the design, or knowledge about the programming technologies. Due to the fact that the general purpose programming languages are ontologically neutral, this knowledge is only partially reflected in programs. Figure 3.7 illustrates that for performing a maintenance task, maintainers need to interpret the program with respect to the knowledge that the code (intends to) implement(s) and thereby to understand (the meaning of) the code.

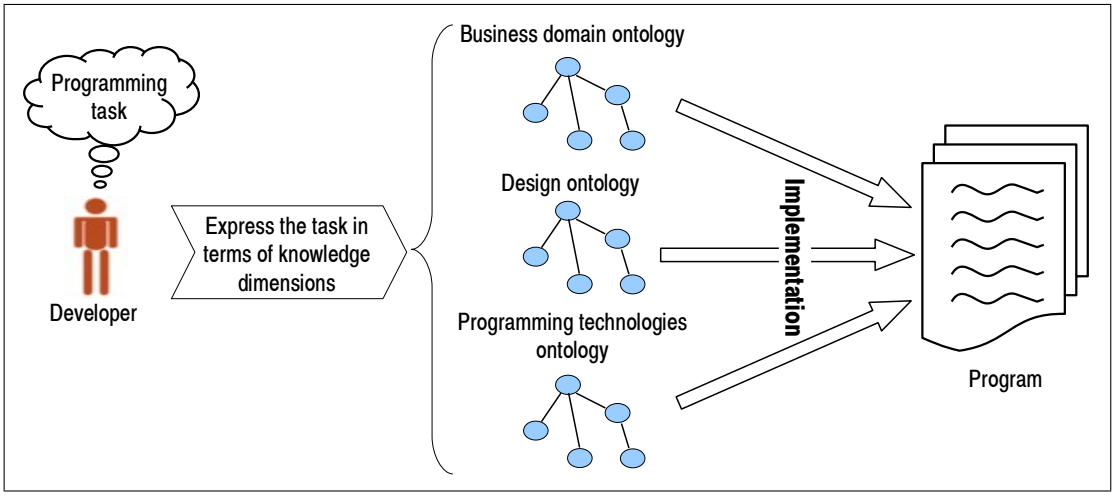


Figure 3.6: Developers implement a task by using different kinds (dimensions) of knowledge

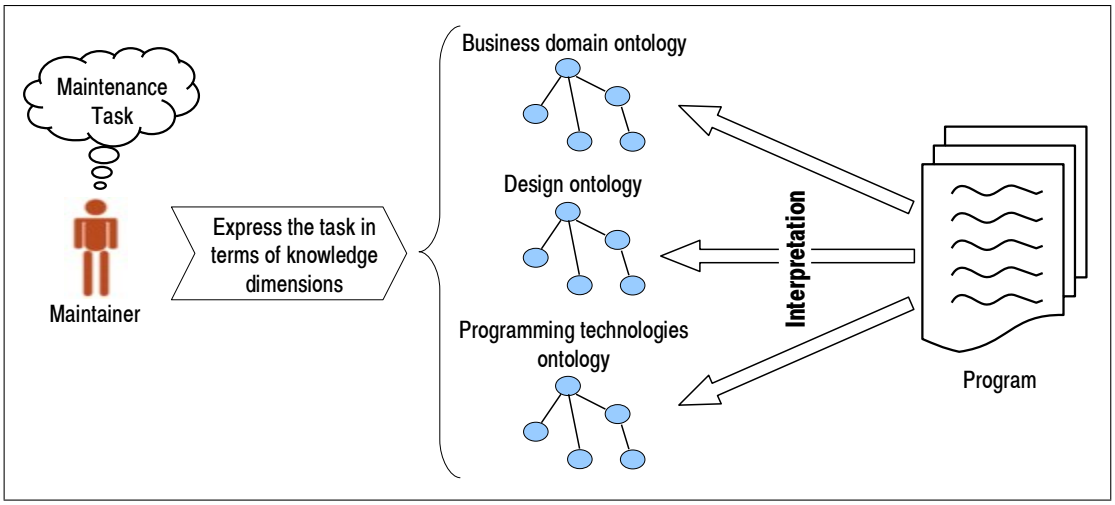


Figure 3.7: For performing a task, maintainers need to interpret the program with respect to the dimensions of knowledge that were implemented

Intuition. According to the *Oxford English Dictionary* [1] the adjective *intentional* means:

"Done on purpose, resulting from intention"

intention

the noun *intention* means:

"The way in which anything is to be understood"

We resume that "intentional" is how the purpose of something should be understood.

In the same dictionary the noun *meaning* is defined as:

meaning

"The sense or signification of a word, sentence, etc. [...] By extension: the thing, person, etc., for which a word or expression stands; the denotation or referent of a word or expression."

"Meaning" is a link between a word and entities from a domain for which it stands.

Combining the above definitions of "intentional" and of "meaning", the "intentional meaning (of something)" is the entity denoting how something should be understood. The current work has programs as subject of matter. As we described above, we define the domain entities through concepts in a domain ontology.

intentional meaning

The *intentional meaning of a program* is defined by linking the program entities (e. g. classes) and the concepts from domain ontologies which they intend to implement.

Example 3.6: Different interpretations of a program fragment

In Figure 3.8 we present fragments of three domain ontologies (right) that can possibly serve as interpretation for the same program fragment (left). Each domain ontology allows a different interpretation of program variables from a different perspective: In the first case (a) all variables are integers and thereby they refer to the same concept in the ontology (namely "integer variable"). In the second case (b), only three of the four variables are entities of an API (the first variable does not belong to the API since it is private). Java does not allow us to distinguish between APIs entities that are public and those that are published (Fowler, 2002). In the third case (c) all variables implement different concepts of the application domain.

Remark. In the cases a) and b) the meaning of a program element (the mapping between the element and its corresponding concept from the domain ontology) can be computed by using the information from the definition of the Java language (e. g. every variable with the type "int" is an "IntegerVar" and every program element with the keyword "public" in front of it is an "API program entity"). In case c), the interpretation requires information that is "outside" of the language (i.e. not captured in the language definition).

□

The same program fragment can have different intentional meanings depending on the domain ontology serving as semantic domain for the intentional interpretation.

[1] `http://www.oed.com`

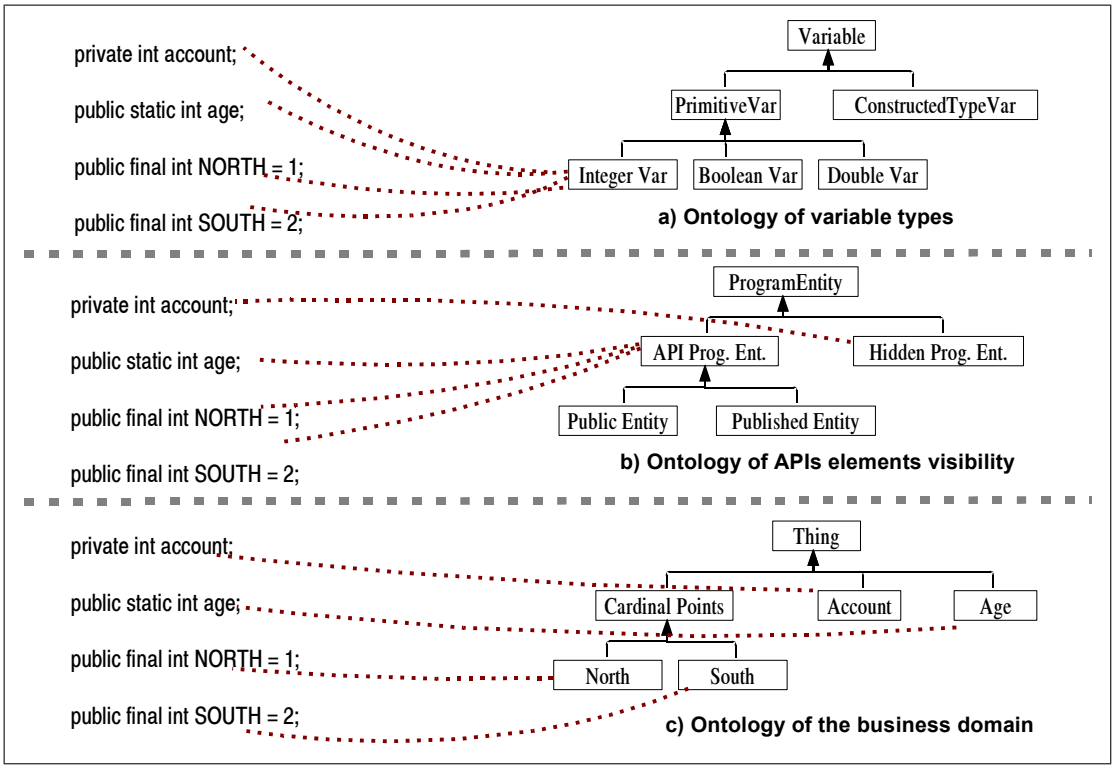


Figure 3.8: The same program fragment can have different intentional meanings

3.3.1 Definition

program

Definition 3.3.1 (Program)**:** *A program consists of a set of named program entities (e. g. classes, methods, and variables) and program relations among them. The program entities correspond to (part of) the nodes of the abstract syntax graph and the relations correspond to (part of) syntactic relations between these nodes.*

A formalization of programs with the help of labeled graphs is presented in Section 3.4.2. In in Chapter 6.3 we discuss in detail the kinds of program elements that we consider and their relations by presenting an ontology of Java programming knowledge. Similarly, a formalization of domain ontologies as labeled graphs is presented in Section 3.4.3. For the moment, in order to keep the presentation simple, it is enough to regard programs as sets of related program elements and ontologies as sets of related concepts, whereby concepts can be understood (simplified) as the content represented by words in the natural language.

intentional interpretation

intentional implementation

Definition 3.3.2 (Intentional interpretation and implementation)**:** *Let C be a set of concepts defined by a domain ontology and P be a set of program elements. The* intentional interpretation *is a function $\overrightarrow{i} : P \rightarrow \wp(C)$ that associates a program element to the set of concepts that it implements. The* intentional implementation *is a function $\overleftarrow{i} : C \rightarrow \wp(P)$ that associates a concept to the set of program elements that implement it.*

In Figure 3.9 we present an intuitive view over the intentional implementation and interpretation. For simplicity, we do not depict in the figure the types of relations between program elements or between concepts from the ontology.

Notation. The symbol $\overleftrightarrow{i}$ denotes both intentional implementation and interpretation.

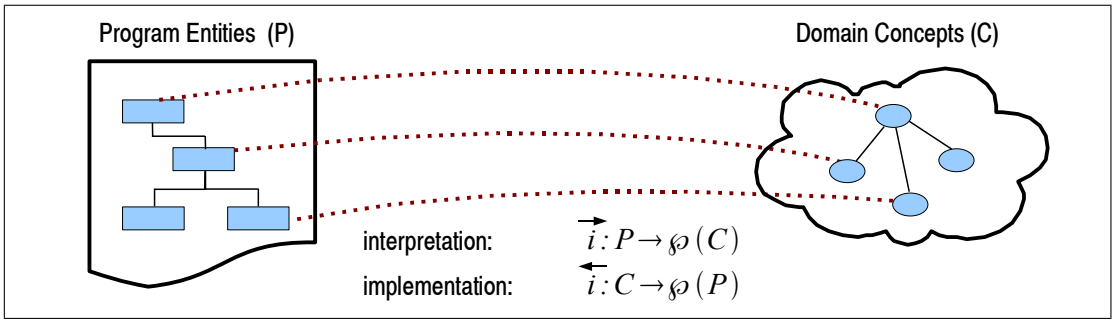


Figure 3.9: Intentional interpretation and implementation

Example 3.7: Interpreting a program

In Figure 3.10 we present a piece of code that implements a part of a traffic lights controller (left) and two domain ontologies (right): one contains concepts related to the modeled domain and the second contains concepts related to the design. The intentional interpretation, given by the relation between a program element to a set of concepts that it refers to, is presented with a dotted line. We can remark that the class `State` implements both the concept STATE (belonging to the business domain) and the concept SUBJECT (belonging to the design domain).

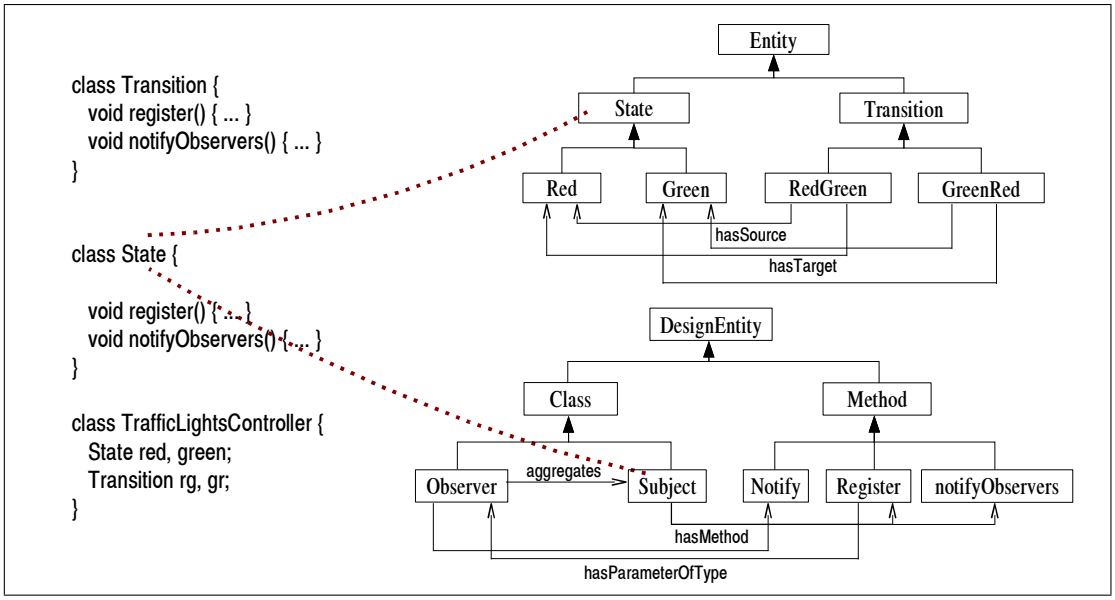


Figure 3.10: Program (left); domain ontologies (right); intentional interpretation (dotted line)

□

Discussion

1) What is the "domain" of a "domain ontology"? We have defined the intentional meaning as a mapping between the programs and a **domain ontology**. Depending on what the domain under consideration is, the intentional meaning can take many forms. Below we present several examples of domains:

kinds of domains

i) **Language syntax:** If the domain is the syntax of the Java language then the intentional interpretation is the classification of program elements according to their type in the syntax graph (as shown in Figure 3.8a). In this case the domain concepts are explicitly represented in programs and therefore there is an exact correspondence between a part of a given program and a concept from the ontology. In this case the intentional interpretation is recovered by parsing the program.

ii) **Design patterns:** If the domain is "design patterns" then the intentional interpretation represents the identification of design patterns (Florijn et al., 1997). In this case the meaning is represented by an ontology of design patterns (a fragment of such an ontology is presented in Figure 3.4).

iii) **Programming technologies:** The source code contains a great bunch of knowledge that is related to programming technologies such as XML, graphical user interface (GUI), databases or communication. Even if these domains exist mostly only in the "computers world" (i. e. they are unknown to non-programmers), they contain a big amount of knowledge that is used in every program. In Section 9.3 we will take a closer look at this domain and propose a method for automatically extracting the domain knowledge by analyzing the commonalities of several domain specific APIs.

iv) **Application (or business) domain:** The most interesting cases are represented by the domains that are completely independent from the "world of programming". Examples of such domains are banking or document processing, and are referred in this thesis as *application or business domains* . The characteristics of these domains is that most of the times there is no clear correspondence between their concepts and the language constructs.

business domain

Remark. While we do not exclude the domains closely related to programming languages – i. e. i) and ii) – *we focus in this work on highly intentional domains* (e. g. the business domain). The concepts of the highly intentional domains cannot be trivially (directly) expressed in programming language constructs.

2) Interleaving and delocalization. Many times a part of a program can implement different concepts. This phenomenon, known as *interleaving*, was presented in Section 2.2.3.1. We capture these situations by requiring the co-domain of $\overrightarrow{i}$ to be the power-set of concepts ($\wp(C)$). In many cases the interleaving involves concepts of different domains (described in different ontologies). In the case when more domains are interleaved in the code, then each domain ontology enables an interpretation of the program from the point of view of that domain. A program element can be related with more concepts from different ontologies (as we illustrated in Figure 2.8 and Figure 3.10).

Many times a concept (or aspects thereof) is implemented in different program parts – a phenomenon known as *delocalization* and presented in Section 2.2.3.2. We capture these situations by requiring the co-domain of $\overleftarrow{i}$ to be the power-set of program elements ($\wp(P)$).

3) What does "implement" mean? We defined the intentional interpretation to map a program element to "a set of concepts that it implements". In Section 3.5 we refine the notion of implementation of a concept along three directions: *reference of concepts*, *definition of concepts*, and *representation of concepts*.

4) How can the implementation and interpretation be recovered? Intuitively, the functions $\overleftrightarrow{i}$ are similar to the knowledge of a senior programmer ("project guru") that knows exactly where the domain concepts are implemented. If no guru is available, these functions can be defined by using other sources of information such as the documentation or the identifiers names. In Part IV we take the second approach and use the similarities between the names of identifiers and the names of concepts in order to recover $\overleftrightarrow{i}$. A fundamental restriction on the functions $\overleftrightarrow{i}$ is made by the available domain knowledge (ontology) – i. e. intuitively, if the "project guru" does not know a domain concept then he cannot find (or recognize) it in the code.

5) Limitations. With respect to the definition of $\overleftrightarrow{i}$ we notice the following limitations:

a) No program terms. We defined the intentional interpretation for individual program elements. Our definition does not take into consideration program phrases (program terms). With other words, our semantic is not compositional.

b) No conceptual terms. We assume that all the entities that can be recognized are explicitly defined as concepts in the ontology. We cannot use the predefined concepts to define new ones (i.e. we do not use any mechanism for defining terms at the conceptual level). In our view all concepts need to be lexicalized – this means that they should have a representation through a set of words (formally described by Definition 3.4.2).

3.3.2 Comparison with Other Notions of Program Meaning

The subject of this work is to define the meaning of a program in terms of the domain knowledge that it implements. In the following we compare the intentional meaning with program semantics and with pragmatics.

3.3.2.1 Program semantics

One of the most common ways to define the semantics of a programming language is in a denotational style. The semantics of a language is given in a denotational manner through a mapping of syntactic constructs ($Synt$) to a semantic domain (Sem):

denotational semantics of a language

$$M : Synt \rightarrow Sem$$

This mapping associates a mathematical object (e. g. number, tuple) to each phrase of the language. The semantic domain should be general enough in order to be able to express all pro-

grams that can be written in the language. In the case of general purpose languages, the semantics is defined thus in an ontologically neutral manner.

program semantics

The syntax of a language defines the language constructs and the basic rules to combine these constructs in order to form terms. What is not directly supported by the language constructs, is expressible (if possible) only as terms (e. g. programs). If the programming languages semantics is compositional, the semantics of a program is given through a composition of the semantics of its parts. Once the meaning of the language is defined then the meaning of its programs can be computed based on the recursive composition of the semantics of the program parts.

The generality of the semantic domain does not allow the explicit representation of domain concepts. As we present in the next example, an object can be interpreted as a state machine. Even if this view enables an accurate description of objects from the point of view of the computation that they perform, it is (most of the times) inappropriate for the manner in which people understand programs in analogy with the phenomena from reality.

Example 3.8: Objects can be regarded as state machines

Below we present an example of a simple class that models the concept FILE: it has two attributes of type boolean that are true if the file is OPEN and if it is READ ONLY. There are several methods that set the values of these attributes. In Figure 3.11 we present the state automaton that models the semantics of the objects of class `File`. The mathematical objects (e. g. state machines in our case) do not offer (almost) any information about the modeled domain.

```
class File {
    boolean isOpen = false, isReadOnly = false;
    void open() { isOpen = true; }
    void close() { isOpen = false; }
    void setReadOnly() { isReadOnly = true; }
    void setReadWrite() { isReadOnly = false; }
}
```

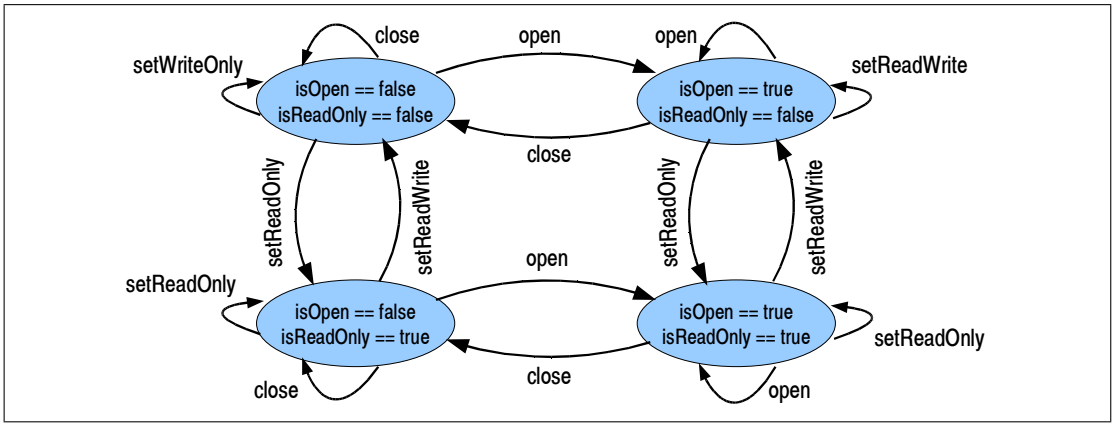


Figure 3.11: The state machine that describes the behavior of the objects of class `File`

In a similar manner, classes that implement other domain concepts such as PERSON, ACCOUNT (see below) can be interpreted as state machines. Describing objects of these classes as state

machines is inappropriate for communicating among programmers that take part in a software project, or with domain experts – e. g. a banking expert would not regard its clients and accounts as state machines.

```
class Account {
  String accountName;
  int sum;

  void setSum(int newSum) {
    sum = newSum;
  }
}
```

```
class Person {
  String personName;
  int age;

  void setAge(int newAge) {
    age = newAge;
  }
}
```

□

We define the intentional meaning of a program element $p \in P$ through the intentional interpretation function: $\overrightarrow{i} : P \rightarrow \wp(C)$. The meaning of a program element is given by a set of domain concepts that it implements. Each concept is part of a domain ontology and is defined by its relations with other concepts from the ontology. The concepts reflect the knowledge that humans have about the modeled domain.

As opposed to the *program semantics*, we focus on *domain meaning* of programs.

"semantic faces" of programs

The two *semantic faces of a program* are illustrated in Figure 3.12: On the one hand, in the lower part of the figure, we have a classical (mathematical) interpretation of a program based on the formal semantics of the language in which the program is written. On the other hand, in the upper part, we regard programs from the point of view of how they implement the domain concepts.

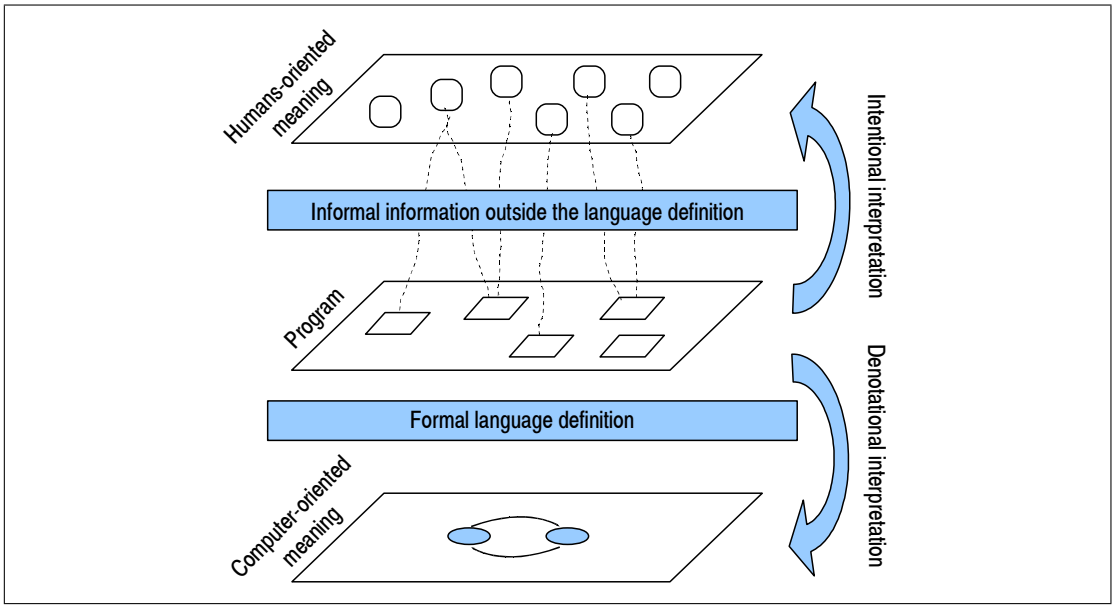


Figure 3.12: Human vs. computer - oriented interpretation of programs

In the following we present examples of differences between the semantics of programs and their intentional meaning.

Example 3.9: Programs with the same formal semantics but different intentional meanings

Below we give some code fragments that have the same semantics in the classical sense (programming language semantics) but have different meanings from the point of view of the application domain:

```
public static int main(String[] args) {
    int brutto = Integer.parseInt(args[0]);
    int taxRatio = Integer.parseInt(args[1]);
    int netto = brutto * (1 - taxRatio);
    System.out.println(netto);
}
```

```
public static int main(String[] args) {
    int pageWidth = Integer.parseInt(args[0]);
    int marginRatio = Integer.parseInt(args[1]);
    int textWidth = pageWidth * (1 - marginRatio);
    System.out.println(textWidth);
}
```

In the pieces of code from above we presented how two different real-world situations (i. e. computing the netto salary (left), and computing the text size (right)) can be encoded in a program in an identical manner. Even if these code fragments implement clearly different situations from the real world and thus have different intentions, the formal semantic of these fragments is identical. By ignoring the intent of these fragments (and thereby considering them to be the same), we can draw flawed conclusions (e. g. we might consider these code fragments to be code clones which is obviously not the case).

□

The difference between the intentional meaning and the semantics of the above code fragments can be remarked through different interpretations of the above fragments in plain English. If we take into account the intentional meaning we can explain the left code fragment with the following sentence: "This code fragment computes the netto salary given the brutto income and a tax ratio." If the intentional aspects would be left out, we can describe the same program fragment as: "Given x, y, this code fragment prints the result of computing $x * (1 - y)$". The latter explanation addresses the program fragment from the point of view of the language semantics and therefore, it is true for both code fragments. The former explanation cannot be used for the right code fragment (that computes the page size).

Two programs can have the same formal semantics but different intentional meanings.

Example 3.10: Programs with the same intentional meaning but different semantics

Below we present two different implementations of the concept PERSON. These code fragments have different formal semantics even though they obviously implement the same domain situation.

```
class Person {
  String name;
  Account myAccount;
}

class Account {
  String accountID;
  int accountStatus;
}
```

```
class Person {
  String name;
  String accountID;
  int accountStatus;
}
```

A more tricky example is given below and addresses two possible implementations of the searching functionality. Even if these two functions have different semantics (the function on the right is a flawed searching algorithm because it does not check the last position in the list) the methods have the same intentional meaning, namely to perform the action of searching.

```
Item search(ArrayList<Item> list, String name) {
  for (int i = 0; i < list.size(); i++)
      if (list.get(i).getName().equals(name))
        return list.get(i);
  return null;
}
```

```
Item search(ArrayList<Item> list, String name) {
  for (int i = 0; i < list.size() - 1; i++)
      if (list.get(i).getName().equals(name))
        return list.get(i);
  return null;
}
```

The above fragment suggests that we can define a bug to be a case where an implementation of an intention is not a model of the intention.

□

Two programs can have the same intentional meaning but different semantics.

Remark. An important distinction between the classical program semantics and intentional meaning of a program is that in the latter case the meaning usually stays outside of the language definition – i. e. there is no definable mapping between the language constructs and the application domain (e. g. we cannot define a mapping between the constructs of the Java language and the entities from the banking domain). From this point of view the intentional meaning is more related with the use of the programming language and thereby our work concerns the pragmatics.

3.3.2.2 Pragmatics

Pragmatics is the study of language use in contrast with the syntax and semantics, which are the study of language structure and meaning. In natural language processing, semantics versus pragmatics is the distinction between literal meaning and the speaker's meaning. The pragmatics considers the ability of the users of a language to communicate more than what is explicitly stated. The meaning of the speaker can be interpreted differently through different bindings of the indexicals (e. g. pronouns like "he", "she" can be bound differently in different contexts) or through different background knowledge.

Example 3.11: Interpreting the pragmatic meaning beside the literal meaning

For example, by saying that:

"John's wife is a mathematician."

one may understand (pragmatically) that John is married and that she knows calculus. This additional information is not conveyed directly by the sentence but it is due to the background knowledge (domain knowledge) of the reader.

□

In order to compare the pragmatics to intentional meaning, we need to decide whether the additional bits of information (e. g. identifiers is the most common source of information), that link the program to the modeled domain, belong to a program. For domains other than programming, the formal definition of programs (as terms over a language) does not tackle the additional information and thereby the intentional interpretation is outside the language definition.

> From the point of view of the programming languages, our work regards the pragmatics – we use background knowledge (ontologies) to interpret the programs.

3.4 A Rigorous Representation of Programs and Concepts

In this section we present our formalization of programs and domain ontologies as labeled graphs. This formalization will represent the basis for the further parts of this dissertation: the characterization of the reflexion of domain knowledge in programs (Part III), and the automation of performing intentional analyses (Part IV).

3.4.1 A Unified Meta-Model

In Figure 3.13 we present a meta-model that explicitly considers the concepts and conceptual relations (the domain ontology), the program elements and program relations, and the mappings between the program elements and concepts. Instances of this meta-model, or parts thereof, are (implicitly and mentally) built in many code comprehension activities. Having such a representation of programs explicit enables us to reason about programs in terms of the domain concepts that they implement.

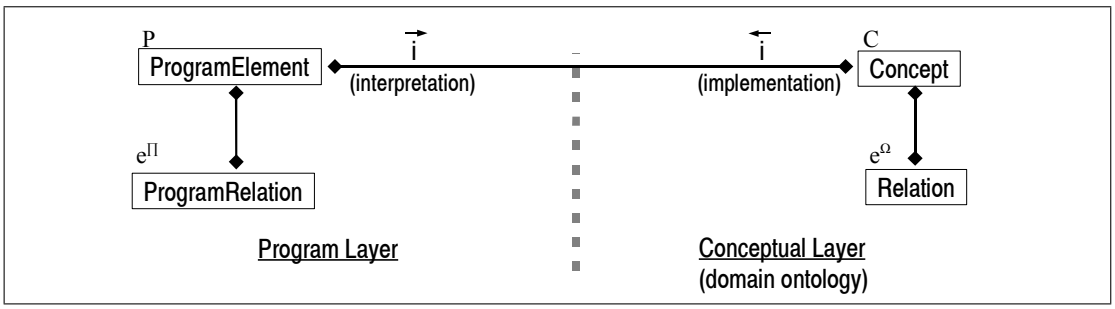


Figure 3.13: A unified meta-model for representing programs, their modeled domain and the relation between them

layered representation of programs

Our main aim is to use the intentional meaning to describe and quantify the reflexion of the domain knowledge in programs (Chapter 4). In order to do this we need a rigorous description of both programs and domain knowledge. Intuitively, we can regard this description as being layered according to the domain appropriateness of its entities (Figure 3.14): the programs are less appropriate to their domain in comparison with the domain concepts shared in the ontology. In Chapter 7 we will extend this layered representation with a new layer (placed between

conceptual and program layers) that contains the information about the names of program elements. This information is used in practice to link the program and the conceptual layers and will represent the basis for our automation.

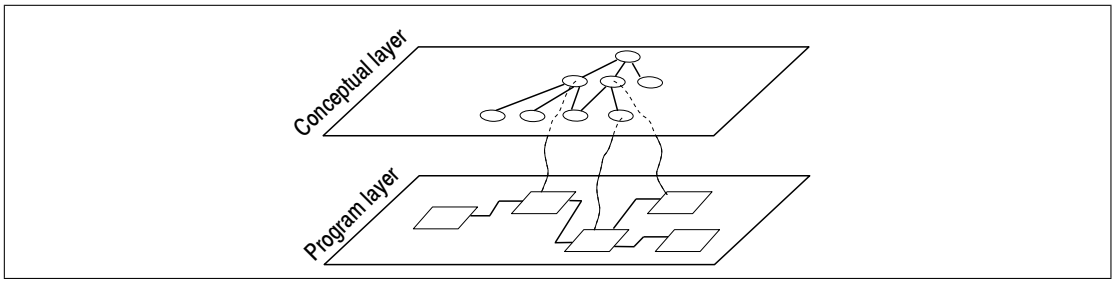


Figure 3.14: An intuitive view over the program and conceptual layers.

3.4.2 The Program Layer

We abstract programs as labeled directed graphs whose nodes are program elements and whose arcs are typed relations between these program elements.

Definition 3.4.1 (Program layer)**:** *We define the program layer* Π *to be the triple:* *program layer*

$$\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$$

where,

- P *is a set of program elements (e. g. classes, methods, variables),*
- Σ^{Π} *is the set of elements representing the types of program relations among the elements from the set* P*,*
- $e^{\Pi} : P \times P \rightarrow \Sigma^{\Pi} \cup \{\epsilon\}$ *is a function that returns the type of the edge between two program elements or* ϵ *if there is no relation between these program elements.*

Configurations of the program layer.

The set of program elements and of relations that are considered depends on the purpose of the analysis. In this thesis, we will use two configurations of the program layer:

1) The entire program (Π^{Prog}**).** Π^{Prog} represents the program layer obtained from the entire program. Depending on the programming language, the set of program elements types and of relations types can be different. In Section 6.3 we perform an in-depth analysis of the types of program elements in a Java program and on the relations among them. To discuss the implementation of domain concepts in programs, it is sufficient that the set P contains classes, attributes, methods, parameters and local variables and that the set $\Sigma^{\Pi_{Prog}}$ contains the following relations:

$\Sigma^{\Pi_{Prog}} = \{hasSupCls,\ hasType,\ hasMeth,\ hasAcc,\ hasAtt,\ hasRetType,\ hasParam\}$

The meaning of these relations is given in Table 3.1.

Relation	Source (S)	Target (T)	Description
hasSupCls	class or interface	class or interface	S extends (implements) T
hasAtt	class or interface	attribute	S has attribute T
hasMeth	class or interface	method	S has method T
hasAcc	class or interface	accessor method	S has accessor method T
hasType	variable	type	S has type T
hasParam	method	parameter	S has parameter T
hasRetType	method	type	S has return type T

Table 3.1: Relations types of the program layer

Remark. Beside the relations generated through the program syntax, in Table 3.1 is also a relation generated by Java programming idioms (hasAcc). In Section 6.3 we present a richer set of program relations by considering beside the syntactical relations also those generated by programming idioms and those between elements of the core library.

2) The API (Π^{API}). Π^{API} represents a projection of Π^{Prog} by considering only the program elements that belong to the public interface: public classes, public attributes, public methods and parameters of public methods. We use the same set of relations as in the case of Σ^{Π}_{Prog}.

Example 3.12: Example of a program layer

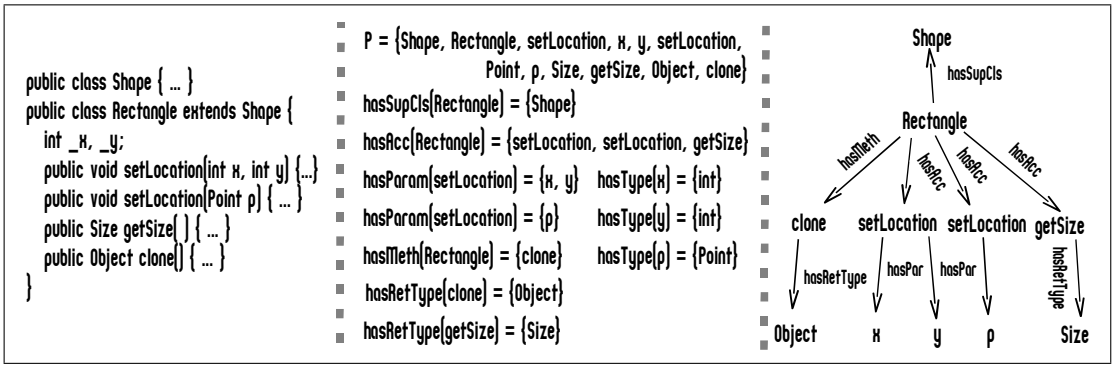


Figure 3.15: Program layer example (with the Π^{API} configuration)

In Figure 3.15 (left) we present an example of a program that contains two classes: `Shape` and `Rectangle`. The class `Rectangle` contains several public methods and two attributes with the "package" visibility. In the center of the same figure we present an instantiation of the program layer and illustrate the content of the sets that define Π^{API}. We need to remark that the program layer does not contain the attributes `_x` and `_y` because they are not declared public. On the right side of the Figure 3.15 we illustrate the graph-based representation of the program layer.

□

Remark. Given two nodes of the program graph, the type of the edge between them tells us whether the nodes represent classes, methods, attributes and so on. For example, hasParam(`setLocation`, `x`) tells us that `setLocation` is a method and `x` is a parameter.

3.4.3 The Conceptual Layer

As presented in the last sections, we use (light-weighted) domain ontologies to define the semantic domain. We describe domain ontologies through a set of triples of the form: "concept – relation – concept". These triples can be represented as labeled graphs whose nodes are the concepts and whose edges are the relations between these concepts.

concept

Definition 3.4.2 (Concept)**:** *We define a concept to be the tuple:*

$$(Names, Gloss)$$

where,

- $Names$ *is a set of names of the concept, and*
- $Gloss$ *is a gloss entry in natural language that uniquely describes the concept.*

Definition 3.4.3 (Conceptual layer)**:** *We define the conceptual layer* Ω *to be the triple:* *conceptual layer*

$$\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$$

where,

- C *is a set of concepts that are relevant for understanding the program; beside the concepts directly implemented in the program, this layer also contains concepts that are strongly related with them.*
- Σ^{Ω} *is the set of elements representing the types of conceptual relations,*
- $e^{\Omega} : C \times C \rightarrow \Sigma^{\Omega} \cup \{\epsilon\}$ *returns the type of the edge between two concepts or* ϵ *if there is no edge between these concepts.*

The conceptual layer has a generic form and, in order to be usable in practice, it needs to be instantiated with concrete relation types. Below we present a configuration of the conceptual layer. The types of the conceptual relations directly affect the expressivity of the conceptual layer – the more domain appropriate the relations are, the better (more explicit) can we describe that domain; but, the harder to map the conceptual layer on programs it is.

Configuration of the conceptual layer

We instantiate the set Σ^{Ω} with four relation types:

- *isA* defines the generalization / specialization relations between two concepts,

- *hasProp* between a concept and one of its properties,
- *actsOn* between a concept representing an action and the concept which stands for the entity on which it acts, and
- *isDoer* between a concept representing an agent and the concept that stands for the action that this agent performs.

$$\Sigma^{\Omega} = \{isA,\ hasProp,\ actsOn,\ isDoer\}$$

In Section 6.2 we present a richer set of conceptual relations based on the "Suggested Upper Merged Ontology" (Niles and Pease, 2001b). However, the above conceptual relation types suffice for describing a wide variety of domain conceptualizations and thereby for describing the important issues related to the way the domain knowledge is reflected in the code.

Example 3.13: Example of a conceptual layer

In Figure 3.16 (left) we present an example of a set of concepts described in fragments of (highly simplified) natural language sentences. Based on these fragments, we illustrate an instantiation of the conceptual layer (center). On the right-hand side of the figure we illustrate the conceptual graph that represents the ontology. In this example we have eight concepts (i. e. described in the middle of the figure by the set C) and two relations (i. e. *isA* and *hasProperty*).

As we notice, this conceptual layer is an ontology fragment of the geometry domain that is implemented by the program from Figure 3.15. In addition to the concepts directly implemented in the program, we have a set of other related concepts (i. e. WIDTH and HEIGHT).

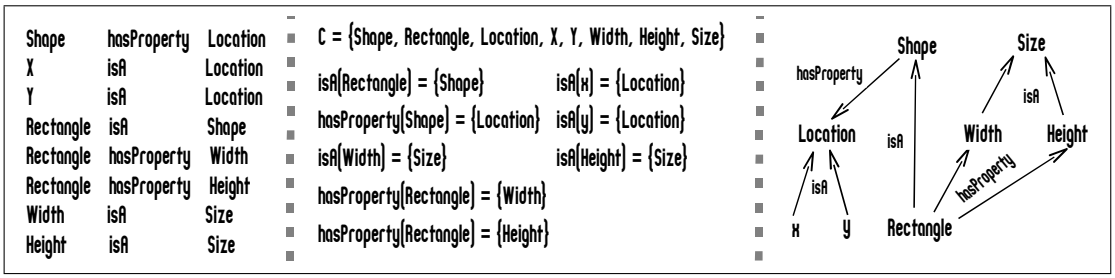


Figure 3.16: Conceptual layer example

□

Remark. In our definitions of the program and conceptual layers configurations, the set of relation types is fixed and the sets C and P are extensible with new concepts and respectively program elements. The sets Σ^{Π} and Σ^{Ω} define the "language" for expressing the program and the conceptual layer: Σ^{Π} defines our abstraction over programs and Σ^{Ω} defines what can we express in the conceptual layer.

Notations

1) Relations as predicates. Another manner to express the relation between two entities e_1, e_2 (program elements or concepts) is through a binary predicate – e. g. $hasType(a, int)$ means that the program element a has type int.
2) Relations as functions. Each relation type (program relation σ^{Π} or conceptual relation σ^{Ω}) can be written as a function that maps a program element (or a concept) to the set of related program elements (respectively of concepts) – e. g. $hasAtt : P \rightarrow \wp(P)$, $hasAtt(Rectangle) = \{\_x, \_y\}$.
3) Accessing the kind of a program element. Whenever we need to access directly the type of a program node we use a unary predicate that corresponds to the type of the node. For example Cls(`Person`) holds if the program element `Person` belongs to P and if it is a class.

3.5 Relating Program Elements and Domain Concepts

In Section 3.3 we defined the $\overleftrightarrow{i}$ functions that make the link between domain concepts and the program elements that "implement" them. We left the meaning of "implement" underspecified. In the next subsections we elaborate on this with the help of the following functions:

refinements of the implementation

- *Reference of concepts* in the code represents the weakest relation between the program and conceptual layers – namely the case when a program element refers to a concept.
- *Definition of concepts* in the code represents the classes that define the characteristics and functionality of a concept.
- *Representation of concepts* in the code represents the program elements that are used to internally encode the concepts.

We use these functions to describe different aspects of the implementation of domain concepts in programs.

Remark. Whenever we refer to one of these functions without caring about which one, we will use the notion of "implementation".

3.5.1 Reference of Concepts

There is a many-to-many relation between program elements and the domain concepts that they refer to. We capture this mapping through the functions $\overleftarrow{Ref}$ and $\overrightarrow{Ref}$.

Definition 3.5.1 (Reference of concepts)**:** *Let $\Omega = (C, \Sigma^{\Omega}, e^{\Omega})$ be a conceptual layer and $\Pi = (P, \Sigma^{\Pi}, e^{\Pi})$ be a program layer. We define the function*

reference of concepts

$$\overleftarrow{Ref} : C \rightarrow \wp(P)$$

that associates to a concept the set of program elements from the program that directly refer to it, and the function

$$\overrightarrow{Ref} : P \rightarrow \wp(C)$$

that associates to each program element a set of concepts that are referred by it. The correlation between these functions is expressed through the following equation:

$$\overrightarrow{Ref}(p) = \left\{ c \in C \mid p \in \overleftarrow{Ref}(c) \right\}$$

When we refer to both functions we use the following notation: $\overleftrightarrow{Ref}$.

Example 3.14: Examples of $\overleftrightarrow{Ref}$ functions

Based on the program layer from Figure 3.15 (extended with attributes `_x` and `_y`) and on the conceptual layer from Figure 3.16, we exemplify below several instances of $\overleftarrow{Ref}$ and $\overrightarrow{Ref}$:

$\overleftarrow{Ref}$(SIZE) = {`getSize`}	$\overleftarrow{Ref}$(SHAPE)={`Shape`}	$\overleftarrow{Ref}$(RECTANGLE)={`Rectangle`}
$\overleftarrow{Ref}$(X) = {`_x, x`}	$\overleftarrow{Ref}$(Y) = {`_y, y`}	$\overleftarrow{Ref}$(WIDTH) = $\emptyset$
$\overleftarrow{Ref}$(HEIGHT)=$\emptyset$	$\overleftarrow{Ref}$(LOCATION) = {`setLocation,setLocation`}	

$\overrightarrow{Ref}$(`getSize`) = {SIZE}	$\overrightarrow{Ref}$(`Shape`) = {SHAPE}	$\overrightarrow{Ref}$(`Rectangle`) = {RECTANGLE}
$\overrightarrow{Ref}$(`x`) = {X}	$\overrightarrow{Ref}$(`_x`) = {X}	$\overrightarrow{Ref}$(`setLocation`) = {LOCATION}
$\overrightarrow{Ref}$(`clone`) = $\emptyset$	$\overrightarrow{Ref}$(`p`) = $\emptyset$	$\overrightarrow{Ref}$(`setLocation`) = {LOCATION}

We notice that the concepts HEIGHT and WIDTH are not implemented, that the method `clone` does not refer any concept and that there are several concepts (e. g. LOCATION) referred by more program elements.

□

comparison between $\overleftrightarrow{Ref}$ *and concepts location and assignment*

Discussion. In the context of program comprehension, the function $\overleftarrow{Ref}$ is similar to *concept location* (Rajlich and Wilde, 2002) and the function $\overrightarrow{Ref}$ to *concept assignment* (Biggerstaff et al., 1993). However, there are several differences:

1. as *concept location* is defined by Rajlich and Wilde (2002), the concepts are mapped to fragments of code and not to individual program elements:

   > "All domain concepts should map onto one or more fragments of the code. The process of concepts location is the process that finds this code."

   Furthermore, when the location of concepts is based on names Rajlich and Wilde (2002) note:

   > "The concept is implemented not only in the place where the identifier was found, but also in previous and following statements, the variables that are used in these statements and so on."

This means that (arbitrary big) fragments of code implement a concept. According to our definition the concepts are assigned to individual program elements rather than program fragments.

2. the notion of 'implementation' defined by Rajlich and Wilde (2002) (see the above quote) is more ambiguous and under-specified. In contrast with this, we define $\overleftarrow{Ref}$ to map concepts only to program elements that **refer** to them (and not to arbitrary program fragments that **implement** them). A concept can be implemented by a fragment of program without any program element to directly refer to it; however, whenever a program element refers to a concept it also contributes to its implementation.

We assume a tighter relation between domain concepts and the code. Instead of linking the concepts to code fragments we link them to individual program elements and instead of using the "implementation" as meaning of the links we assume that program elements "refer" to the concepts.

In Chapter 5 we will use the $\overleftrightarrow{Ref}$ functions to characterize the reference of domain concepts in programs. The manner in which program elements refer to concepts influences the measure in which the implementation of domain concepts in programs can be explicitly accessed by programmers.

3.5.2 Definition of Concepts

Many concepts are not only referenced by some program elements but also defined as program abstractions. There is a many-to-many correspondence between the concepts and the program abstractions that define them: a concept can be defined in different program parts through different abstractions and an abstraction can define more concepts.

Definition 3.5.2 (Definition of concepts)**:** *Let* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *be a conceptual layer and* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *the program layer. We define the function* *definition of concepts*

$$\overleftarrow{Def} : C \to \wp(P),$$
$$\overleftarrow{Def}(c) = \{p \in P \mid Cls(p)\ \wedge\ c \in \overrightarrow{Ref}(p)\}$$

that associates to a concept the program elements that define it, and the partial function

$$\overrightarrow{Def} : P \hookrightarrow \wp(C)$$

that associates to a program element the set of concepts that it defines. The correlation between these functions is expressed through the following equation:

$$\overleftarrow{Def}(c) = \{p \in P \mid c \in \overrightarrow{Def}(p)\}$$

When we refer to both functions we use the following notation: $\overleftrightarrow{Def}$.

We consider that classes are the most important means in object-oriented languages for defining domain concepts. The function $\overrightarrow{Def}$ is defined only on classes and this is why it is partial (in the above definition we require that: $\overrightarrow{Def} : P \hookrightarrow \wp(C)$).

Example 3.15: Examples of definition of concepts

In Figure 3.15, the concept RECTANGLE is defined by the class `Rectangle`. Not all the concepts referred in the program are also defined. For example, in the program layer from Figure 3.15, the concept X is referred through the parameter `x` of the method `setLocation(int x, int y)`. However, the concept X is not defined in the program. In order to manipulate it at the program level, it is represented in the program through the type `int`.

□

Discussion. We make a distinction between the case when a program element defines a domain concept and the case when a program element only refers a domain concept. Once a domain concept is defined, it can be subsequently instantiated and used in a program. Since a definition of a concept can be used at the program level, domain concepts are defined only through classes: the interface of the class represents the way in which the implementation of the corresponding domain concept can be used in the program. The implementation of the class represents the semantic of the concept as defined in the program.

3.5.3 Representation of Concepts

Clearly, not every concept from the domain is defined in a program. However, in order to manipulate the concepts, a program provides representations for them. There is a many-to-many correspondence between the concepts and the data types that are used to represent them: a concept can be represented in different program parts through different data types and a data type can be used to represent more concepts.

concepts representation

Definition 3.5.3 (Representation of concepts)**:** *Let $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ be a conceptual layer and $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ be the program layer. We define the function*

$$\overleftarrow{Rep} : C \to \wp(P)$$

$$\overleftarrow{Rep}(c) = \{p \in P \mid \exists p' \in P.\ \overrightarrow{Ref}(p') = \{c\} \wedge\ hasType(p') = p\}$$

that associates to a concept the set of types that are used to represent it, and the function

$$\overrightarrow{Rep} : P \to \wp(C)$$

that associates to a program element the set of concepts that are represented by it. The correlation between these functions is expressed through the following equation:

$$\overrightarrow{Rep}(p) = \{c \in C \mid p \in \overleftarrow{Rep}(c)\}$$

When we refer to both functions we use the following notation: $\overleftrightarrow{Rep}$.

Intuitively, if a variable references a single concept then the concept is represented in the program through the type of this variable. For example, in Figure 3.15 the concepts X and Y are represented as integers.

Discussion. We consider that a concept is represented in the program through the type of the variable that refers to it. This is known in the literature of programming languages as "representation commitment" and is made many times during the development phase. From the programming language point of view choosing a representation is a "binding time commitment" (Gunter et al., 1996). The problem of changing this early commitments during the evolution of the software is challenging. Once the concept NAME is represented, for example, as a `String` and it is used in many places in a program, it is very difficult to change its representation (for example as a class `Name`).

binding time commitment

The importance of the correspondence between the concepts implemented in a program and the program abstractions used to represent them is widely acknowledged in the program comprehension literature. For example, Bastani and Iyengar (1987) empirically investigate the effects of using (or not) appropriate data structures for implementing abstract data types on errors identification. One of the experiments involved finding errors in the implementation of the 'push' operation of a stack. This operation was implemented in three procedures, each of them using a different data structure (e. g. array, linked list). The results of the experiments suggest that the difficulty in comprehending a program increases as the opaqueness of the mapping between the representation and abstract data types increases itself.

From the knowledge representation point of view, if a variable refers to a domain concept (e. g. the variable `name` refers to the domain concept NAME), then by choosing the type of this variable to be, for example, `String` we perform an *encoding bias*. One of the important criteria for building ontologies for knowledge sharing is the minimal encoding bias of concepts (Gruber, 1995). In the following example we present two representation anomalies.

encoding bias

Example 3.16: Examples of representation anomalies

```
int c = getColor();
int r = getRadius();
aCircle.setRadius(c);
aCircle.setColor(r);
```

```
Point p1 = aCircle._pos;
Point2D p2 = new Point2D(p.getX(), p.getY());
anotherCircle.move(p2);
```

Figure 3.17: Examples of anomalies introduced by representation

In Figure 3.17 we depicted examples of $\overleftarrow{Rep}$ and $\overrightarrow{Rep}$. The concept POSITION is represented in the program through the classes `Point` and `Point2D`. Therefore, in order to combine different instances of this concept at the program level we need to convert between representations (right). The primitive type `int` is used to represent the domain concepts COLOR and RADIUS and thus we can use different implementation of concepts at the program level in an inappropriate manner (left).

□

In Chapter 5 we present a set of defects generated by the inappropriate and non-homogeneous representation of concepts in programs.

Comparing the Reference, Definition and Representation of Concepts

Regarding the difference between the reference, definition and representation of concepts we make the following remarks:

1) While the reference of concepts affects their identification in the program, the representation of concepts affects the manners in which the clients can manipulate and compose the concepts at the program level. Figure 3.17 presents two such examples of anomalies which originate from using the same representation for two different concepts or using different representations for the same concept.

2) The definitions of concepts tell us about the explicitness of the program abstractions. Once a concept is defined at the code level, it can be subsequently used in other program parts. There is a duality between computation and structure in a program: on the one hand, once the concepts are explicitly defined, the computations needed to manipulate them are more simple; on the other hand, if concepts are not explicitly defined, the amount of computation increases.

3) All the functions defined above can be expressed in terms of $\overleftarrow{Ref}$. Therefore, the key for automatizing the computation of $\overrightarrow{Ref}$, $\overleftarrow{Rep}$, $\overrightarrow{Rep}$, $\overleftarrow{Def}$ and $\overrightarrow{Def}$ is the automation of $\overleftarrow{Ref}$. In Chapter 7 we present our approach for locating concepts in the code in an automatic manner, which is based on the similarities between the names of identifiers and the names of concepts.

Remark. These functions do not cover the whole spectrum of relations between domain concepts and the program elements. One could easily imagine other (more complex) functions that elaborate even more on the notion of "implementation". However, we claim that these functions are enough in order to characterize a wide variety of problems that typically occur in the programming practice and that are rooted in the mismatched reflexion of the domain in programs.

3.6 Relating Conceptual and Program Relations

There is a many-to-many mapping between the types of ontological relations (Σ^{Ω}) and the types of program relations (Σ^{Π}): a relation type between concepts can be reflected in the code through several relations types between program elements and vice-versa. We capture this correspondence through two functions: *relations implementation* $\overleftarrow{t}$ and *relations interpretation* $\overrightarrow{t}$.

Definition 3.6.1 (Implementation and interpretation of relations)**:** *Let* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *be a conceptual layer and* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be the program layer. We define the* implementation of relations *to be the function*

implementation of relations

$$\overleftarrow{t} : \Sigma^{\Omega} \rightarrow \wp(\Sigma^{\Pi})$$

that maps a relation type from the ontology to the corresponding relation types from the program graph. The interpretation of relations *is defined by the function*

interpretation of relations

$$\overrightarrow{t} : \Sigma^{\Pi} \rightarrow \wp(\Sigma^{\Omega})$$

that maps a relation type from the program graph to a corresponding relation type from the ontology. The duality between these functions is expressed through the following equation:

$$\overrightarrow{t}(\sigma^{\Pi}) = \{\sigma^{\Omega} \in \Sigma^{\Omega} \mid \sigma^{\Pi} \in \overleftarrow{t}(\sigma^{\Omega})\}$$

When we refer to both functions we use the following notation: $\overleftrightarrow{t}$.

Example 3.17: Examples of relations mapping

Below we present examples of mappings between relation types from our conceptual layer and relation types from the program layer. Intuitively, the *isA* relation is implemented through the *hasSupCls* or *hasType* relation, the *hasProp* relation is implemented through the *hasAtt* or *hasAcc* relation, the *actsOn* relation through *hasParam* and *isDoer* through *hasMeth*.

$$\overleftarrow{t}\,(isA) = \{hasSupCls, hasType\}$$
$$\overleftarrow{t}\,(hasProp) = \{hasAtt, hasAcc\}$$
$$\overleftarrow{t}\,(actsOn) = \{hasParam\}$$
$$\overleftarrow{t}\,(isDoer) = \{hasMeth\}$$

□

In Section 6.4.2 we present in more detail possible implementation strategies of different conceptual level relations. Also, due to different implementation decisions, many times the mapping between conceptual and program relations is more complex – e. g. a conceptual relation can be mapped to a sequence of program relations (Section 7.5.2). For example, as shown below, the property of a concept can be implemented either direct (b); as an attribute of the sub-class of the class implementing the concept (c); or as an attribute of a class that is aggregated by the class implementing the concept (d). The more degenerate the implementation is the harder it is to recognize the relations between concepts in programs.

Person hasProp Name	class Person {
	  String name;
	}
		class Person { ... }
		
		class Mother extends Person {
		  String name;
		}
			class Person {
			  Infos info;
			}
			
			class Infos {
			  String name;
			}

a)	**b)**	**c)**	**d)**

In Chapters 4 and 5 we discuss in detail the intricacies generated by different implementations of concepts and relations.

Limitation: We implicitly assume that the conceptual level relations and the program relations are at the same abstraction level – namely that different types of real-world relations among concepts are directly implemented through different types of program relations that connect the program elements implementing these concepts. *In our framework, we have no way to define new relation types in a program. The set of program relations types is fixed by the kind of program elements that form the program layer.* This is because in the current mainstream object-oriented languages there is no language construct that allows the definition of relations as first-class program entities. Abstract relations can be only defined through programming trickery such as data structures (e. g. maps) or attributes of classes. The advantages to explicitly define relations as first-class constructs in object oriented languages was recognized since the 80s (Rumbaugh, 1987).

3.7 Summary and Roadmap

In order to bridge the abstraction gap between (application) domain concepts and their implementation in the code, we proposed to explicitly map program entities to concepts from domain ontologies that they implement. We called this mapping the "intentional interpretation". We showed that this notion is complementary to the formal semantics (denotational theories) of programming languages and is more close to pragmatics. We presented a rigorous model of intentional meaning firstly by formalizing programs and domain ontologies as labeled graphs and secondly by defining a set of mappings between programs and domain ontologies: reference of concepts, definition of concepts, and representation of concepts in programs.

Roadmap. In the previous chapter we showed that abstract information is lost during the software engineering process. In this chapter we proposed the intentional meaning as a solution to recover this information by explicitly linking domain concepts with program entities. The core of this dissertation contains two more parts:

- in Part III we use the intentional meaning to characterize the reflexion of domain knowledge in the code. We do this by evaluating the level of isomorphism between the graph based representations of programs and domain ontologies.
- in Part IV we focus on practical means to automate the recovery of the intentional meaning. We concentrate on the use of similarities between the names of identifiers and of program elements in order to recover $\overleftrightarrow{Ref}$.

Part III

Reflexion of Domain in Programs

4 A Framework for Characterizing the Reflexion of Domain in Programs

A good FORTRAN programmer can write FORTRAN code in any language.

Fortran Folklore

Abstract: Due to the conceptual gap between the domain knowledge and programming constructs the choice of a specific implementation strategy of domain concepts in programs is most of the times ad-hoc. Many of the implementation decisions prevent a faithful correspondence between the business domain and programs and this leads in turn to a variety of intricacies in reflecting the domain knowledge in the code: domain concepts are left out from the implementation or program elements implement concepts that do not belong to the domain, distinct concepts are compacted in the same program elements, groups of concepts are implemented in a distorted manner and the same concept is implemented redundantly in more program parts. These issues lead in turn to code decay and to programs that are hard to understand and maintain. In this chapter we investigate these intricacies with the help of our formalization of programs and ontologies defined in the previous chapter. We develop a framework to characterize the reflexion of the domain in programs with respect to: the measure in which the program covers the domain concepts, the faithfulness of the implementation of groups of concepts, the ambiguity degree in the implementation of concepts, and the logical redundancy. In Chapter 5 we will instantiate this framework with concrete cases of mismatches and discuss their influence on typical programming and maintenance activities and on the usability of APIs.

Structure of this chapter. In the introduction (Section 4.1) we present the ideal implementation of concepts that maintains an isomorphism between the domain ontology and the program. After the introduction, we dedicate a section to each category of issues generated by the improper implementation of domain concepts in the code: Section 4.2 characterizes the coverage of the domain in programs, Section 4.3 characterizes the unfaithfulness in the implementation of groups of related concepts, Section 4.4 presents the diffusion of concepts in programs, and Section 4.5 presents the logical redundancy. The chapter ends with a summary (Section 4.6) and a short outlook on the next chapters.

4.1 Introduction

Most of the programming languages widely used today are Turing complete. This means that from a computational point of view, they are equally powerful: each program written in a language could be written in any other language as well. In the same time, there is a huge conceptual gap between the domain knowledge that is usually implemented in programs and the general purpose programming languages used for its implementation. With respect to this gap, the differences among the general purpose programming languages (most of the times syntactic) seem small for programmers. One can easily simulate the use of a language within another – e. g. one can write object-oriented or functional-style programs in the "ANSI C" language (Schreiner, 1994) or can use (almost) any programming language for writing Fortran-like code. However, when it comes to implementing a domain model in a program, the big conceptual gap can be filled in a multitude of ways by taking different implementation decisions.

In practice there is almost never a "best implementation decision" and choosing an implementation in favor for another is most of the times only a matter of technical trade-offs (and programming experience or taste) rather than an informed and systematically followed strategy. These decisions preserve (or not) the similarity between the program and the groups of concepts from the domain and this affects in turn different quality attributes of programs such as their extensibility, understandability or usability of APIs. In order to understand and characterize the relation between different implementation decisions and the quality attributes, we need to explicitly and rigorously characterize the reflexion of the domain in the code.

> We propose a framework for investigating the faithfulness of the implementation of domain concepts in programs that implement them by measuring the level of isomorphism between the ontology and program graphs.

In the following we present an introductory example in order to illustrate different implementation possibilities of the same domain situation.

Example 4.1: Different implementations of concepts about a family

In the upper-left part of Figure 4.1 we present a domain ontology that covers different members of a family, their relations and basic information such as their names and addresses. We present several code fragments that represent typical ways to implement these concepts. The ideal implementation, that mirrors the domain, is presented on the top-right side (Figure 4.1a). However, such implementations (similar to those found in object-oriented programming textbooks) can seldom be seen in the programming practice. Due to pragmatic constraints, programmers leave out some domain concepts while adding implementation details that fill the conceptual gap between the domain knowledge and the programming constructs (Figure 4.1b). Furthermore, due to different technical constraints (or lack of programming skills) the relations between related concepts (e. g. PARENT and HUMAN) can be distorted in the code (Figure 4.1c). In Figure 4.1d we present situations when the same program elements are used to implement different domain concepts. Thereby the differences between concepts disappear at the code level between their corresponding implementations (e. g. the concepts NAME and ADDRESS are both variables of type `String` and thereby they can be interchanged in a program). Finally, many times the same

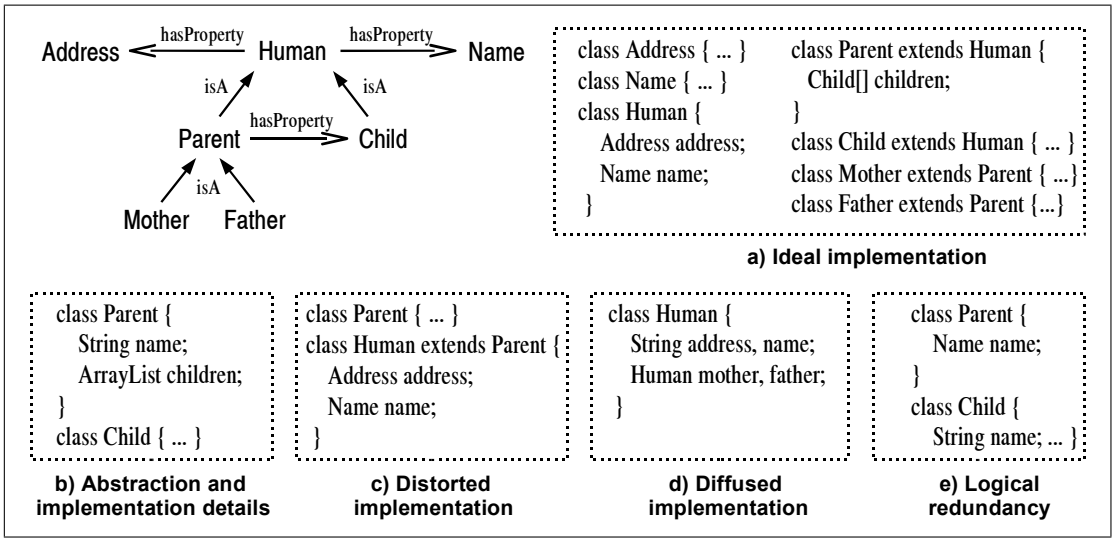


Figure 4.1: Introductory example: different implementations of family relations

concept is implemented redundantly in different ways in different parts of the program – in Figure 4.1e the concept NAME is defined as a class and represented as a `String`.

□

Notation (Function application). Let $f : A \to \wp(B)$ be a function that maps elements of a set A to subsets of a set B. Let $A' \subset A$ be a subset of A. We use the following notation:

$$f[A'] = \bigcup_{x \in A'} f(x)$$

Let $F \subset (A \to \wp(B))$ be a set of functions that map elements of a set A to subsets of a set B. We use the following notation:

$$[F](x) = \bigcup_{f \in F} f(x)$$

Notation (Navigating the graphs). Let $\Omega = (C,\ \Sigma^\Omega,\ e^\Omega)$ be a conceptual layer. To denote the neighbors of a concept $c \in C$ with respect to the relation type $\sigma^\Omega \in \Sigma^\Omega$ we use the following notation:

$$\sigma^\Omega(c) = \{c' \in C \mid e^\Omega(c, c') = \sigma^\Omega\}$$

Remark (Presentation structure). Each mismatch characterization has the same structure: we first define formally the (non-)mismatch situation, then we explain and give the intuition of the formula and we discuss the causes and consequences of the mismatch.

Note on reading the formulas. In this chapter we develop a formal framework to characterize the reflexion of domain models in programs. We do this by investigating the level of isomorphism between the conceptual and the program layers. Most of the formulas used in this chapter contain several basic building blocks as described below:

- $\overrightarrow{i}[\overleftarrow{i}(c)]$ – means the interpretation ($\overrightarrow{i}$) of the set of program elements that make up the implementation ($\overleftarrow{i}$) of the concept c.
  - $\overrightarrow{i}[\overleftarrow{i}(c)] = \{c\}$ means that each program element that implements the concept c does not implement any other concept (i. e. injectivity).
- $\overleftarrow{i}[\overrightarrow{i}(p)]$ – means the implementation ($\overleftarrow{i}$) of the set of concepts that make up the interpretation ($\overrightarrow{i}$) of the program element p.
  - $\overleftarrow{i}[\overrightarrow{i}(p)] = \{p\}$ means that each concept that is implemented by the program element p is not implemented by any other program element.
- $\overrightarrow{i}[\sigma^{\Pi}(p)]$ – means the set of concepts that are implemented by the program elements that are neighbors of the program element p with respect to the relation σ^{Π}.
- $\overleftarrow{i}[\sigma^{\Omega}(c)]$ – means the set of program elements that implement the concepts that are neighbors of the concept c with respect to the relation σ^{Ω}.
- $\overleftarrow{t}(\sigma^{\Omega})$ – denotes the set of program level relations that are used to implement the relation σ^{Ω}.
  - $[\overleftarrow{t}(\sigma^{\Omega})][\overleftarrow{i}(c)]$ – means the set of program elements that are neighbors with the one of the program elements that belong to the implementation of the concept c with respect to the implementation of the relation σ^{Ω}.

ideal implementation

Definition 4.1.1 (Ideal implementation)**:** *Let* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be a program abstraction and* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *a domain ontology. The implementation of* Ω *in* Π *is* ideal *iff:*

$$\forall c \in C.\ \overrightarrow{i}[\overleftarrow{i}(c)] = \{c\}\ \wedge\ \forall \sigma^{\Omega} \in \Sigma^{\Omega}.\ \overrightarrow{t}[\overleftarrow{t}(\sigma^{\Omega})] = \{\sigma^{\Omega}\}\ \wedge \tag{4.1}$$

$$\forall p \in P.\ \overleftarrow{i}[\overrightarrow{i}(p)] = \{p\}\ \wedge\ \forall \sigma^{\Pi} \in \Sigma^{\Pi}.\ \overleftarrow{t}[\overrightarrow{t}(\sigma^{\Pi})] = \{\sigma^{\Pi}\}\ \wedge \tag{4.2}$$

$$\forall \sigma^{\Omega} \in \Sigma^{\Omega}.\ \overleftarrow{i}[\sigma^{\Omega}(c)] = [\overleftarrow{t}(\sigma^{\Omega})][\overleftarrow{i}(c)]\ \wedge \tag{4.3}$$

$$\forall \sigma^{\Pi} \in \Sigma^{\Pi}.\ \overrightarrow{i}[\sigma^{\Pi}(p)] = [\overrightarrow{t}(\sigma^{\Pi})][\overrightarrow{i}(p)] \tag{4.4}$$

The *ideal* implementation represents the case in which there is a one-to-one correspondence between the entities and relations from the ontology and the entities and relations from the program. Figure 4.2 illustrates this situation intuitively. The above formula can be understood as a conjunction between the following affirmations:

- each concept and conceptual relation is implemented by a set of program elements and respectively program relation that do not implement any other concept respective conceptual relation (Formula 4.1) – i. e. there is a one-to-many relation between program elements and program relation types and concepts and conceptual relation types,
- each program element and program relation implements a set of concepts and respectively conceptual relations that are not implemented by any other program element respectively program relation (Formula 4.2) – i. e. there is a one-to-many relation between concepts and conceptual relation types and program elements and program relation types,
- every relation between two concepts is reflected in the program – in other words, the structure of the ontology is preserved in the program (Formula 4.3),
- every relation between two program elements reflects a relation between their corresponding concepts – in other words, the structure at the program level is not richer than that in the ontology (Formula 4.4)

Mathematically, the first two formulas taken together (4.1 and 4.2) define a bijection between the program elements and concepts on the one hand and the program relation types and conceptual relation types on the other; the next two formulas together (4.3 and 4.4) define the equality between structures. To sum up, *the conjunction of all these four formulas represent the case when there is an isomorphism between the program and the conceptual layer. In this case, the conceptual and program "worlds" are completely indistinguishable as far as our chosen representations (Ω and Π) are concerned.*

isomorphic implementation

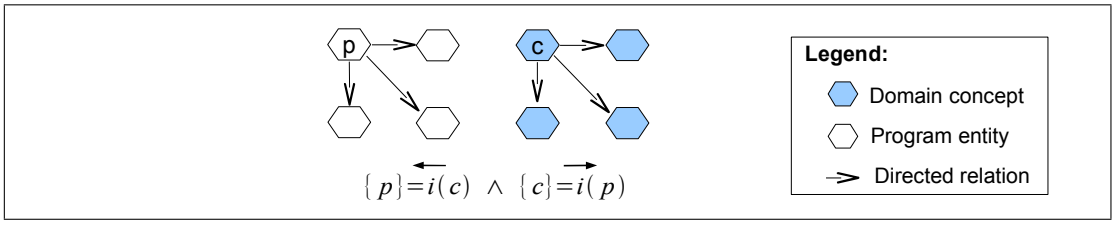


Figure 4.2: Ideal implementation intuition

Remark. Our abstractions of programs and domain knowledge (i. e. the program and the conceptual layers) are quite coarse grained and therefore not complete – i. e. we cannot describe all intricacies that occur in implementing domain concepts in programs. *However, our claim is that the framework presented in this chapter is powerful enough to capture a wide variety of mismatches that occur in practice and that have important consequences on different programming and maintenance activities.*

Discussion: The *ideal* implementation cannot be realized due to the conceptual gap between the high-level representation of the real world knowledge through domain ontologies and the low-level programming constructs. Below we enumerate the effects of this gap and how they influence the implementation of domain concepts.

1. Due to the high complexity of the modeled domain and the limited resources of programmers, an ideal implementation is not feasible in practice. This leads to the implementation of only an *abstraction* (Definition 4.2.1) of domain models in programs.

2. In order to bridge the conceptual gap, the programs contain many *implementation details* (Definition 4.2.2). The implementation details are program elements that are irrelevant from the point of view of the modeled domain.

3. Groups of concepts are implemented in the code in a *distorted* manner (Definition 4.3.1) and therefore the structure among them is changed in the code. The changed structure leads to (ever increasing) conceptual biases between the domain and the code.

4. Several concepts are implemented by single program elements or several conceptual relations by single relations at the program level. In these cases one cannot distinguish the borders between the implementations of two concepts or relations at the code level. We call these situations *diffusions* of the domain model in the code (Definition 4.4.2). The diffusion leads to (ever increasing) decay of the code due to the *conceptualization loss*.

5. A concept is implemented by more program elements and this leads to *logical redundancy* (Definition 4.5.2).

In the subsequent sections we present different relaxations of the constraints imposed by the ideal implementation. Each of the following section has a corresponding section in Chapter 5 where we study in detail each mismatch category.

4.2 Conceptual Coverage

Definition 4.2.1 (Abstraction)**:** *Let* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be a program abstraction and* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *a domain ontology. The implementation of a concept* $c \in C$ *and relation* $\sigma^{\Omega} \in \Sigma^{\Omega}$ *is an* abstraction *iff:*

abstraction

$$\sigma^{\Omega}(c) \supset \overrightarrow{i}[[\overleftarrow{t}(\sigma^{\Omega})][\overleftarrow{i}(c)]] \tag{4.5}$$

The *abstraction* implementation represents the case in which a part of the concepts related to c or the relations between them are not reflected at the code level (Figure 4.3). Step-by-step the formula should be read like this: the concepts related to c (denoted $\sigma^{\Omega}(c)$) form a superset of the interpretation (denoted $\overrightarrow{i}$) of the neighbors of the implementation of c (denoted $[\overleftarrow{t}(\sigma^{\Omega})][\overleftarrow{i}(c)]$). This can be noticed also by comparing Figure 4.3 with Figure 4.2: in the former figure we have less concepts and relations.

Discussion: The implementation exhibits *abstraction* due to the pragmatic decisions that programmers made when a system was planned and due to the way in which the boundaries of the system were chosen. Depending on what was left out from the modeled domain during the abstraction process, and on the relations between the left out and the already implemented entities, it can be more difficult or even impossible to extend or adapt a particular program. In the

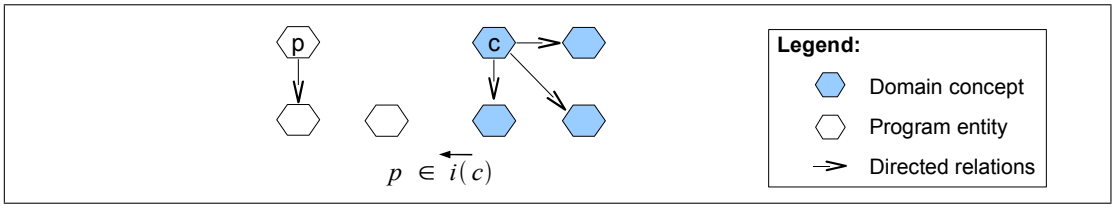


Figure 4.3: Abstraction intuition

case of APIs, the abstraction prevents their users to use (direct) implementations of domain concepts. In Section 5.2 we discuss several abstraction cases and how they influence the extension of programs.

Definition 4.2.2 (Implementation details)**:** *Let* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be a program abstraction and* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *a domain ontology. The implementation of a concept* $c \in C$ *and relation* $\sigma_i^{\Omega} \in \Sigma^{\Omega}$ *introduces* implementation details *iff:* *implementation details*

$$[\overleftarrow{t}(\sigma^{\Omega})][\overleftarrow{i}(c)] \supset \overleftarrow{i}[\sigma^{\Omega}(c)] \tag{4.6}$$

The *implementation details* represent the case in which additional to the program elements and relations corresponding to the implemented concepts, other program elements and relations appear on the program side (Figure 4.4). This can be noticed also by comparing Figure 4.4 with Figure 4.2: in the first figure we have more concepts and relations in the program graph.

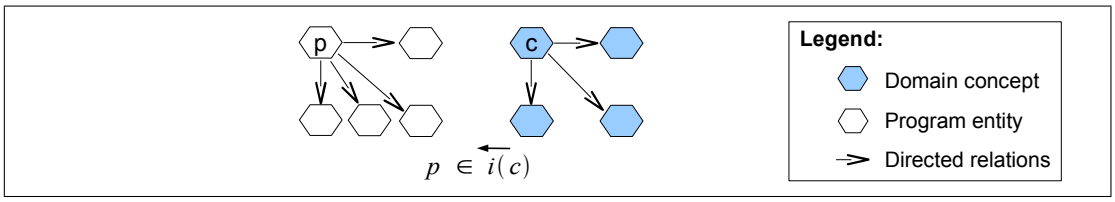


Figure 4.4: Implementation details intuition

Discussion: The *implementation details* are inherent due to the discrepancy between the declarative representations of the real-world knowledge and the operational implementation supported by the current languages. Many times the implementation details belong exclusively to the programming machinery. By introducing the implementation details that do not represent concepts from the modeled domain, programs lose their conciseness. Due to the implementation details, the vocabulary of APIs is cluttered with implementation related words and thus hard to learn, understand and use.

4.3 Distortion of Domain in Programs

Intuition. The "Oxford English Dictionary"[1] defines the word *distort* as in the following:

> "To put out of shape or position by twisting or drawing awry; to change to an unnatural shape; to render crooked, unshapely, or deformed"

In a similar way, groups of related concepts can be implemented in programs in a "deformed" manner: the structure of the concepts group is not preserved among the corresponding program elements in a program part; instead, they are reflected through non-equivalent relations. We call these "deformations" distortions.

Definition 4.3.1 (Distorted implementation)**:** *Let* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be a program abstraction and* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *a domain ontology. The implementation of related concepts* $c_1, c_2 \in C,\ c_2 \in \sigma^{\Omega}(c_1)$ *through the program elements* $p_1, p_2 \in P$ *such that* $p_1 \in \overleftarrow{i}(c_1)$ *and* $p_2 \in \overleftarrow{i}(c_2)$ *is* distorted *iff:*

distorted implementation

$$\exists \sigma^{\Pi} \in \Sigma^{\Pi}.\ p_2 \in \sigma^{\Pi}(p_1)\ \wedge\ \sigma^{\Pi} \notin \overleftarrow{t}(\sigma^{\Omega}) \tag{4.7}$$

Intuitively, the *distorted* implementation represents the situation when the program relation between the implementations of two related concepts does not reflect the conceptual relation. With other words, the program "lies" about the real-world (Figure 4.5).

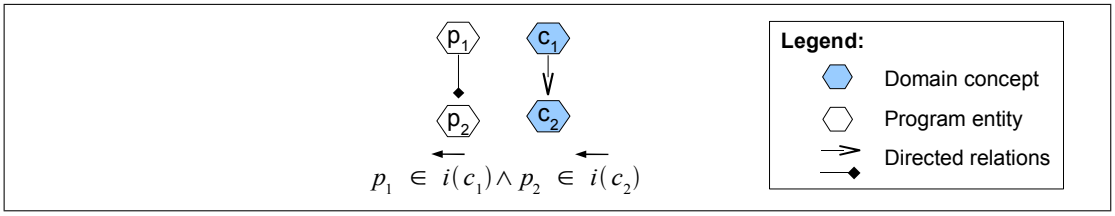


Figure 4.5: Distorted implementation intuition

Discussion: The *distorted implementation* deforms the reflexion of the ontology in the program. Many times the sources of distortion are the technical constraints imposed by the programming languages (see Section 2.2.2, *impedance mismatch* example), the need of integration with pre-existent software components, unfortunate design decisions, or not anticipated evolution. Distortion leads to a bias between the modeled domain and the code which subsequently influences the programming and maintenance activities. For example, in the case of an API, the API users cannot work with it in direct analogy to the modeled domain but have to take into consideration how is the domain implemented.

[1] `http://www.oed.com`

4.4 Diffusion of Domain in Programs

Intuition. The "Merriam Webster Dictionary"[2] defines the word *diffusion* as in the following:

> "[...] **3 a**: the process whereby particles of liquids, gases, or solids intermingle [...]
> **5**: the softening of sharp outlines in a photographic image"

In the same manner, the implementation of several concepts can be intermingled in the same program element, or the same relation type at the program level can be used to implement several conceptual relation types. In these cases the sharp distinction between different concepts (or relations) that can be made at the domain level is lost in the code – e. g. when inspecting a class that implements more concepts it is hard to find out which methods and attributes belong to the implementation of one concept or of another, or how do these concepts interact.

Definition 4.4.1 (Sub-ontology)**:** *Let* $\Omega = (C, \Sigma^{\Omega}, e^{\Omega})$ *be a domain ontology. We call* $\Omega' = (C', \Sigma^{\Omega'}, e^{\Omega'})$ *a sub-ontology of* Ω *(denoted* $\Omega' \subset \Omega$*) iff:* *sub-ontology*

$$C' \subset C \ \wedge \ \Sigma^{\Omega'} \subset \Sigma^{\Omega} \ \wedge \ \forall c_1, c_2 \in C'.\ e^{\Omega'}(c_1, c_2) = e^{\Omega}(c_1, c_2) \tag{4.8}$$

Intuitively, a sub-ontology of an ontology contains only a subset of the concepts and relation types of the initial ontology and preserves the remaining ontological relations among the concepts.

Definition 4.4.2 (Diffusion)**:** *Let* $\Omega = (C, \Sigma^{\Omega}, e^{\Omega})$ *and* $\Omega' = (C', \Sigma^{\Omega'}, e^{\Omega'})$ *be two domain ontologies and* $\Omega' \subset \Omega$*. Let* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be an intentional abstraction of a program. We call that between* Ω *and* Π *is a* diffusion *iff:* *diffusion*

$$\overrightarrow{i}[\overleftarrow{i}[C']] \supset C' \ \vee \ \overrightarrow{t}[\overleftarrow{t}[\Sigma^{\Omega'}]] \supset \Sigma^{\Omega'} \tag{4.9}$$

Intuitively, diffusion is the situation when interpreting the implementation of a set of concepts and conceptual relation types results in a bigger set of concepts or conceptual relation types. With other words, there are program elements that implement several concepts or program relation types that are used to implement several conceptual relation types (Figure 4.6).

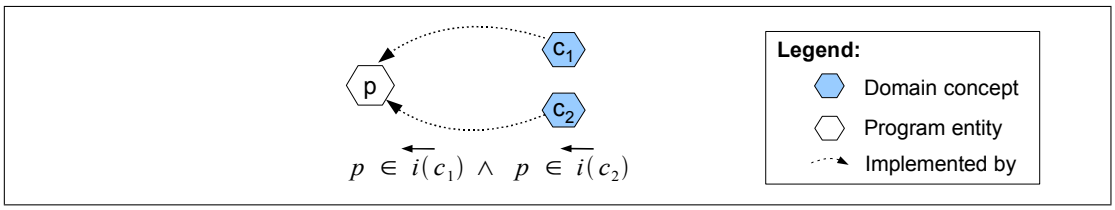


Figure 4.6: Diffusion intuition

Definition 4.4.3 (Directly implemented concept)**:** *A concept* $c \in C$ *is* directly implemented *iff* *directly implemented concept*

$$\overrightarrow{i}\left[\overleftarrow{i}(c)\right] = \{c\}$$

[2] http://www.merriam-webster.com

Intuitively, every time when a concept c is implemented in the code, the program elements that implement it do not implement any other concept. This represents the ideal case when individual concepts from the real world have individual counter-parts among the program elements within the program. In practice concepts are commingled with other concepts in the same program elements:

$$\left|\overrightarrow{i}\left[\overleftarrow{i}(c)\right]\right| > 1$$

directly implemented relation

Definition 4.4.4 (Directly implemented relation)**:** *A relation* $\sigma^{\Omega} \in \Sigma^{\Omega}$ *is* directly implemented *iff*

$$\overrightarrow{t}\left[\overleftarrow{t}(\sigma^{\Omega})\right] = \{\sigma^{\Omega}\}$$

Intuitively, every program relation that is used to implement a conceptual relation does not implement any other conceptual relation. In practice many program relations types can be interpreted as different conceptual relations types:

$$\left|\overrightarrow{t}\left[\overleftarrow{t}(\sigma^{\Omega})\right]\right| > 1$$

Discussion: As we explained in Section 2.2.2, a wide variety of situations from the application domain has to be encoded in programs by using only a small set of programming constructs. In Chapter 6 we show that the abstraction level of the current programming constructs is at the level of the most general concepts in the upper ontologies. In order to assure a lean transition to the domain knowledge, the programmers should incrementally implement more and more specific concepts until they reach the generality (specificness) level of their domain. In practice, this is not possible and most of the times there is a steep encoding step of the domain knowledge in programs. The consequence of this big encoding step is a diffusion of domain knowledge: once a domain fragment is implemented in a program, many concepts or relations clearly distinguishable at the domain level are not any more recognizable in programs. Instead of being explicit (reflected in the structure), the interaction between concepts is implicit and has an algorithmical nature. The diffusion of relation types is due to the limitations of the language to express different conceptual relations. This is similar to *construct overload*, as we presented in Section 2.2.2 (the same language construct is used to express different things from the domain).

4.5 Logical Redundancy

Intuition. The "Oxford English Dictionary"[3] defines the word *redundant* as in the following:

> "**1. b.** Characterized by superfluity or excess in some respect; having some additional or superfluous part, element, or feature."

Many times program parts are superfluous with respect to the domain knowledge that they implement: whenever a certain group of domain concepts is implemented several times, this

[3] http://www.oed.com

introduces redundancy from the point of view of domain knowledge. There are special cases when this redundancy is justified by technical constraints such as efficiency or different limitations of programming languages (Kapser and Godfrey, 2006), but as a general rule, redundancy in programs is regarded as a negative fact.

Definition 4.5.1 (Sub-program)**:** *Let* $\Pi = (P, \Sigma^{\Pi}, e^{\Pi})$ *be a program abstraction. We call* $\Pi' = (P', \Sigma^{\Pi'}, e^{\Pi'})$ *a sub-program of* Π *(denoted* $\Pi' \subset \Pi$*) iff:* *sub-program*

$$P' \subset P \ \wedge \ \Sigma^{\Pi'} \subset \Sigma^{\Pi} \ \wedge \ \forall p_1, p_2 \in P'. e^{\Pi'}(p_1, p_2) = e^{\Pi}(p_1, p_2) \tag{4.10}$$

Intuitively, a sub-program contains only a subset of the program elements of the program. Whenever these elements are related in the original program, the relations among them are preserved in the sub-program.

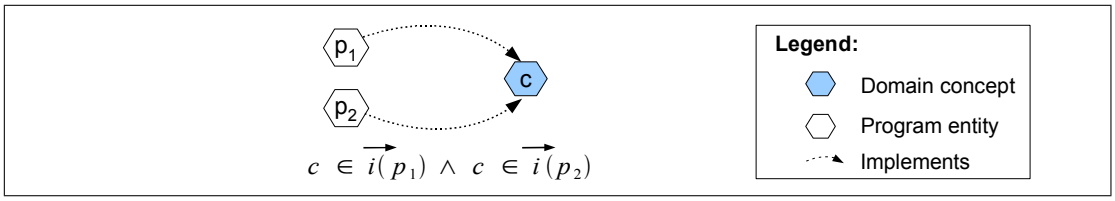


Figure 4.7: Redundancy intuition

Definition 4.5.2 (Logical redundancy)**:** *Let* $\Pi = (P, \Sigma^{\Pi}, e^{\Pi})$ *and* $\Pi' = (P', \Sigma^{\Pi'}, e^{\Pi'})$ *be two program abstractions and* $\Pi' \subset \Pi$*. Let* $\Omega = (C, \ \Sigma^{\Omega}, \ e^{\Omega})$ *be a domain ontology. We call that between* Π *and* Ω *is a* logical redundancy *iff:* *logical redundancy*

$$\overleftarrow{i}[\overrightarrow{i}[P']] \supset P' \ \vee \ \overleftarrow{t}[\overrightarrow{t}[\Sigma^{\Pi'}]] \supset \Sigma^{\Pi'} \tag{4.11}$$

Intuitively, redundancy is the situation when computing the implementation of the interpretation of a set of program elements and relations, results in a bigger set of program elements or relations. With other words, there are concepts implemented by several program elements and conceptual relation types implemented by several program relation types (Figure 4.7). This situation occurs in two cases: 1) when a concept or conceptual relation type is implemented redundantly in the code, or 2) when the domain ontology does not distinguish between two implementations of (potentially different) concepts.

Remark. The redundancy of relation types is due to the multiple implementation possibilities of the same conceptual relation. This is similar to the *construct excess* (Gehlert and Esswein, 2007) as we presented in Section 2.2.2 – the implementation language offers more possibilities for expressing the same conceptual-level situation.

Definition 4.5.3 (Concise implementation)**:** *Let* $\Pi = (P, \ \Sigma^{\Pi}, \ e^{\Pi})$ *be a program abstraction and* $\Omega = (C, \ \Sigma^{\Omega}, \ e^{\Omega})$ *a domain ontology. The program element* $p \in P$ *exhibits* concise implementation *iff:* *concise implementation*

$$\overleftarrow{i}[\overrightarrow{i}(p)] = \{p\}$$

Intuitively, the program element that is a concise implementation implements a concept that is not implemented anymore by other program elements. Many times the implementation of a concept is redundant:

$$|\overleftarrow{i}[\overrightarrow{i}(p)]| > 1$$

In comparison with the identification of *code* clones which is essentially a bottom-up approach, we address here the problem of *logical* redundancy in a top-down manner: we are looking for multiple implementations of concepts in the code.

Mind twister. The general aim of this work is to define the meaning of programs with respect to the domain that they implement. In order to do this, we use the functions $\overleftrightarrow{i}$.

- $\overleftarrow{i} : C \rightarrow \wp(P)$ – where $\overleftarrow{i}(c)$ means the *implementation* of a concept $c \in C$, and
- $\overrightarrow{i} : P \rightarrow \wp(C)$ – where $\overrightarrow{i}(p)$ means the *interpretation* of a program element $p \in P$.

Since we represent both programs and domain ontologies as labeled graphs, we can change the perspective and give the meaning of the domain ontology as it is defined by the program that implements it. To be more clear, we propose the following mental experiment: a programmer reads a program that implements a domain that is (partially) unknown to him. He could learn new things about that domain by reading the program. With other words, instead of interpreting the program with respect to the domain, he interprets the domain concepts with respect to the program. In this case the meaning is the program and the object to which meaning is assigned is an incomplete domain model. Due to the change of perspective (the semantic domain is the program and the syntactical object is the domain ontology), implementation and interpretation functions would exchange their meaning.

- $\overleftarrow{i} : C \rightarrow \wp(P)$ – where $\overleftarrow{i}(c)$ means the *"interpretation"* of a concept $c \in C$ with respect to the program, and
- $\overrightarrow{i} : P \rightarrow \wp(C)$ – where $\overrightarrow{i}(p)$ means the *"implementation"* of a program element $p \in P$ in the incomplete domain ontology.

This is why, maybe somehow unintuitive, diffusion is in our framework similar to redundancy (see Formulas 4.9 and 4.11). Redundancy means that our ontology cannot distinguish between two implementations of a concept and considers these two implementations to be the same (superfluous, redundant). Diffusion means that the program does not distinguish between two concepts and it assigns them a single program element.

4.6 Summary

In this chapter we presented a framework that allows a concise characterization of the mapping between domain models and the code along several directions:

1. the conceptual coverage of domain in the code,
2. the level of homomorphism between groups of concepts from the domain and the program parts that implement them (distortions),
3. the distinguishability of the implementation of different concepts in code (diffusions), and
4. the conciseness in the implementation of domain concepts (logical redundancy).

Remark. All these mismatches are defined with respect to a given domain ontology. As we presented in the previous chapter, a domain ontology always reflects the domain knowledge from a particular point of view. There can be more (equally good) domain ontologies that describe the same domain. Therefore, a program can exhibit mismatch with respect to a domain ontology and in the same time be faithful with respect to another ontology that describes the same domain.

Outlook. In the following chapter we take a closer look at a set of frequent types of mismatches between domain knowledge and programs and we discuss how these mismatches affect typical programming activities with focus on maintenance. In Chapter 6 we investigate how the most abstracts domain concepts (as defined by the Suggested Upper Merged Ontology (Niles and Pease, 2001b)) are implementable in Java programs. The results of this investigation suggest that the ideal implementation is in most of the situations impossible to realize.

5 Characterizing the Implementation of Concepts and Relations

You can do anything in this world if you are prepared to take the consequences.

Somerset Maugham[1]

Abstract: Due to the big conceptual gap between the real-world knowledge and programming languages and technologies, programmers have to take a multitude of design and implementation decisions. Many of these decisions (inevitably) lead to mismatches between the domain knowledge and the code that implements it. These mismatches have in turn consequences on the future programming and maintenance activities such as implementing new features or using the APIs. In the previous chapter we presented a framework that allows us to characterize the level of mismatch between the domain knowledge shared as domain ontologies and programs along the following directions: in what measure are the domain concepts implemented in programs (conceptual coverage), how faithful are they implemented (distortion and diffusion of concepts) and how concise are the domain concepts implemented (logical redundancy). In this chapter we instantiate our framework firstly by instantiating the general functions $\overleftrightarrow{i}$ into the more refined parts (i. e. $\overleftrightarrow{Ref}$, $\overleftrightarrow{Rep}$, and $\overleftrightarrow{Def}$) and discussing them in the context of the mismatches framework, and secondly by taking a closer look at particular cases of mismatches and discuss their influence on programming in general and on the use of APIs in particular. Our general aim is to identify, describe and investigate the consequences of concrete cases of mismatches that are often encountered in the practice. Furthermore, we aim to define these mismatch cases so that they can be (automatically) detectable in the practice.

Structure of this chapter. In Section 5.1 we give a brief overview of how the functions $\overleftrightarrow{Ref}$, $\overleftrightarrow{Rep}$ and $\overleftrightarrow{Def}$ can be combined with the coverage, diffusion, distortion, and logical redundancy. Each of the following sections correspond to a section from Chapter 4: in Section 5.2 we present in more detail the issues related to conceptual coverage of programs and define the notion of conceptual extensibility, in Section 5.3 we present a characterization of distortions, in Section 5.4 we look into more detail at diffusions, and in Section 5.5 we present a characterization of different cases of redundancy. In Section 5.6 we present related approaches that deal with different aspects of the conceptual mismatch between the code and higher-level artefacts. Section 5.7 ends this chapter with a summary and a set of conclusions.

[1] British novelist

5.1 Introduction

In the current chapter we take a more detailed look upon the categories of mismatches defined in Chapter 4 along two directions. Firstly, we instantiate the implementation and interpretation functions $\overleftrightarrow{i}$ into their more concrete correspondents: $\overleftrightarrow{Ref}$ (reference), $\overleftrightarrow{Rep}$ (representation), and $\overleftrightarrow{Def}$ (definition) (as defined in Section 3.5). In Table 5.1 we present how do these functions combine with different categories of mismatches. Secondly, we discuss concrete situations of mismatches and how they influence different programming and maintenance activities. Our general aim is to investigate how the implementation decisions leading to mismatches affect the programmers (API users or maintainers) in the future. With other words, we are aware that in many cases mismatches cannot be avoided and programmers have to use different (obscure) encodings of domain knowledge (e. g. "write Fortran code in another language"). However, we can alow these mismatches as long as we are aware of their future consequences.

	$\overleftrightarrow{Ref}$	$\overleftrightarrow{Rep}$	$\overleftrightarrow{Def}$
Conceptual coverage	x		x
Distortion of concepts			x
Diffusion of concepts	x	x	x
Logical redundancy		x	x

Table 5.1: How different kinds of implementation combine with the categories of mismatches

Remark. We describe each mismatch by following the same presentation structure: at first we define the mismatch rigorously by using our formal framework, we then present the intuition behind the formula, give examples of occurrences of the mismatch in the Java standard API and finally discuss the variation points and influences of the mismatch on the programming and maintenance activities in general and on the usage of APIs in particular. By presenting mismatch examples from the Java standard API (instead of toy-examples) we want to point out their pervasiveness in the practice. We present the mismatches from Java with respect to the WordNet off-the-shelf ontology that is described in more detail in Section 9.2. We detected these mismatches by performing (semi-)automatic analyses as we detail in Part IV.

All the mismatches formalized in this chapter are given by considering our intentional program abstraction $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ and a domain ontology $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$.

5.2 Characterizing Conceptual Coverage

As presented on the first line of Table 5.1 we consider the coverage from two perspectives: the reference of concepts and the definition of concepts.

pure implementation details

Definition 5.2.1 (Pure implementation details)**:** *A program element $p \in P$ exhibits* pure implementation details *iff:*

$$\overrightarrow{Ref}(p) = \emptyset$$

Intuitively, pure details means that, according to our conceptual level knowledge, the program element cannot be assigned to any concept (Figure 5.1a).

Example 5.1: Example of implementation details

An example of implementation detail is the method `hashCode` from the class `java.util.Calendar`. The method `hashCode` does not refer to any concept that is known by the WordNet ontology[2]. Even if they do not refer to any domain concept, the methods `hashCode` play an essential role in the Java programs. This is a typical case when the details are introduced by the underlying programming machinery.

□

Discussion. From the point of view of API clients the pure details represent accidental complexity in the library with which they have to deal. Even if the pure details do not have any meaning for the modeled domain, many times they have a central meaning for the programming language. From the point of view of working with the code at the abstraction level of the modeled domain, the pure details represent accidental complexity and are a burden for the programmers and maintainers. These details are introduced solely by the underlying programming (representation) infrastructure. During their daily work, the programmers need to be very much aware of these details since failing to do this leads to programs that do not work.

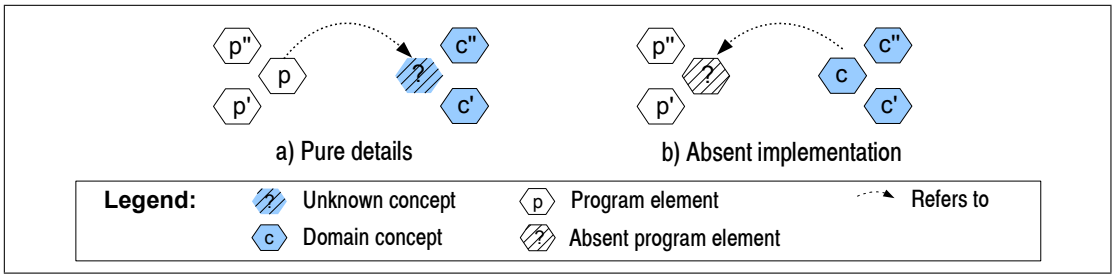


Figure 5.1: Conceptual coverage intuition: a) pure details; b) absent implementation

Definition 5.2.2 (Absent implementation)**:** *A concept $c \in C$ exhibits* absent implementation *iff:* *absent implementation*

$$\overleftarrow{Ref}(c) = \emptyset$$

Intuitively, absent implementation means that there are concepts from the modeled domain that are not referred by any program element (Figure 5.1b).

Example 5.2: Example of absent implementation

An example of absent implementation in the Java standard library (Figure 5.2) is the concept SOLAR CALENDAR that is defined in the WordNet ontology. Since it is not referenced by any program element, it cannot be used at the program level.

□

[2] `http://wordnet.princeton.edu`

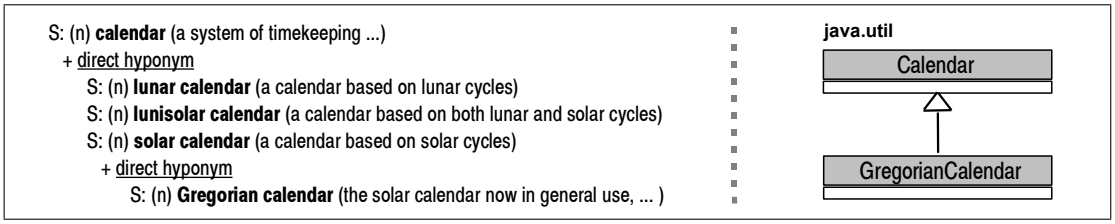


Figure 5.2: The concept SOLAR CALENDAR defined in the WordNet ontology (left) is not implemented in the Java API

Discussion. In the case of absent implementation, a concept is not referenceable in the program at all. So, even if it is implemented somehow in the code in an algorithmic way, it cannot be referenced explicitly, and consequently cannot be accessed and manipulated at the code level (e. g. through refactorings). In Section 10.5.1 we present our experience with assessing conceptual coverage of several well-known Java APIs by identifying absent reference.

Please note that the functions $\overleftrightarrow{Ref}$ represent the weakest form of mapping between programs and domain concepts. The concepts that are not referenced are not represented and not defined (since $\overleftrightarrow{Rep}$ and $\overleftrightarrow{Def}$ are defined based on $\overleftrightarrow{Ref}$). On the other hand, there are many situations when concepts are referenced in programs but are not defined.

absent definition

Definition 5.2.3 (Absent definition)**:** *A concept $c \in C$ exhibits* absent definition *iff:*

$$\overleftarrow{Def}(c) = \emptyset$$

Intuitively, absent definition of a concept means that it does not have any associated class that defines it at the program level.

Example 5.3: Example of absent definition

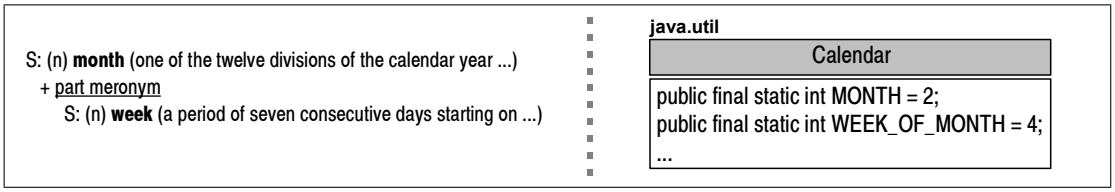


Figure 5.3: The concepts MONTH and WEEK are not defined in the Java API

An example of absent definition in the Java API is the concept MONTH (Figure 5.3). Even if this concept is referenced by several program elements, there is no class associated with it. Instead it is encoded as a constant and thereby the programmers can manipulate it only algorithmically. Furthermore, the concepts related to MONTH need also to be encoded. In the WordNet ontology, MONTHS have WEEKS as their parts (meronyms). In the Java implementation of CALENDAR this relation is only algorithmically implemented. From the point of view of the implementation, the concept MONTH is a second class citizen compared to the concept CALENDAR that is defined (as a class) in the Java library.

□

Discussion. The concepts that exhibit absent definition are not directly and explicitly defined. Instead, they are encoded in programs and their dependencies with related concepts are not visible in the program structure but instead are hidden behind algorithms. Even if this situation can be well motivated by performance concerns, it hampers the relation between the code and the domain that it implements. By making this relation implicit, it has a negative influence on the comprehensibility of the code, and this could negatively influence other programming and maintenance activities.

Definition 5.2.4 (Absent relation)**:** *Let $c_1, c_2 \in C$ be two related concepts with $c_2 \in \sigma^{\Omega}(c_1)$. The implementation of the concepts c_1 and c_2 through program elements $p_1, p_2 \in P$ with $p_1 \in \overleftarrow{Ref}(c_1),\ p_2 \in \overleftarrow{Ref}(c_2)$ exhibits* absent relation *iff:* *absent relation*

$$\nexists \sigma^{\Pi} \in \Sigma^{\Pi}.\ p_2 \in \sigma^{\Pi}(p_1)$$

In this case two related concepts are implemented in the code but the relation at the program level does not reflect the fact that the concepts are related. This situation can happen because:

1. The chosen program abstraction (i.e. the configuration of the program layer Π) is too weak to reflect this relation, even if it explicitly exists (somehow) in the code structure;
2. The relation is implemented only implicitly (algorithmically). In this case, the implementation is not explicit and is only hidden in the logic of the program. Generally, the less structure a program has, the more computation to implement different relations is needed.
3. The relation is not implemented in the code at all.

Example 5.4: Example of absent relation

As we presented in Figure 5.3, the WordNet meronymy relation between MONTH and WEEK is not (explicitly) implemented in the Java API. Instead, the relation between these concepts is only algorithmically implemented in the Java standard library.

□

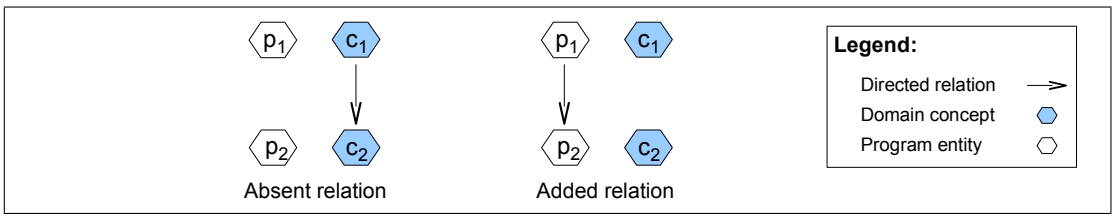


Figure 5.4: Relations coverage

Definition 5.2.5 (Added relation)**:** *Let $c_1, c_2 \in C$ be two unrelated concepts. The implementation of the concepts c_1 and c_2 through program elements $p_1, p_2 \in P$ with $p_1 \in \overleftarrow{i}(c_1),\ p_2 \in \overleftarrow{i}(c_2)$ exhibits* added relation *iff:* *added relation*

$$\exists \sigma^{\Pi} \in \Sigma^{\Pi}.\ p_2 \in \sigma^{\Pi}(p_1)$$

Intuitively, there are relations at the code level between program elements that implement unrelated concepts.

Example 5.5: Example of added relation

In the `java.util` package the `GregorianCalendar` class contains the attribute `second` even though these two concepts are independent (not directly related) in the WordNet ontology.

□

Discussion. This case represents a case of implementation details. Through the added relation we define unnecessary details at the level of the code between the implementation of concepts and this can make the program harder to understand and maintain. In the case of APIs, added relations represent the pollution of their interface.

Conceptual extensibility of programs

Definition 5.2.6 (Extension of a program)**:** *Let* $\Pi = (P, \Sigma^{\Pi}, e^{\Pi})$ *and* $\Pi' = (P', \Sigma^{\Pi'}, e^{\Pi'})$ *be intentional abstractions of two programs. We say that* Π' *is an extension of* Π *with* $\{p'_1, ..., p'_n\} \notin P$ *(denoted* $\Pi' = \Pi \oplus \{p'_1, ..., p'_n\}$*) iff:*

$$P' = P \cup \{p'_1, ..., p'_n\} \ \wedge \ \Sigma^{\Pi'} = \Sigma^{\Pi} \ \wedge \ \forall p_1, p_2 \in P. e^{\Pi}(p_1, p_2) = e^{\Pi'}(p_1, p_2)$$

Intuitively, a program Π' is an extension of the program Π when the extended program is obtained by adding a set of program elements to the original program. The program relations from the original program are preserved in the extended program.

Example 5.6: Examples of extension

Below we present two examples of program fragments, the one from right representing the extension of the program fragment from left. The right fragment (playing the role of Π') is obtained from the fragment from the left (playing the role of Π) by adding two new program elements – namely the attribute `age` and the type `int`.

```
class Person {
  String address;
}
```

$P = \{Person,\ String,\ address\}$
$e^{\Pi}(Person,\ address) = hasAtt$
$e^{\Pi}(address,\ String) = hasType$

```
class Person {
  String address;
  int age;
}
```

$P = \{Person,\ String,\ address,\ int,\ age\}$
$e^{\Pi}(Person,\ address) = hasAtt$
$e^{\Pi}(address,\ String) = hasType$
$e^{\Pi}(Person,\ age) = hasAtt$
$e^{\Pi}(age,\ int) = hasType$

□

Definition 5.2.7 (Completely implementable concept)**:** *A concept $c' \in C$ that exhibits absent implementation in the program Π is* completely implementable *if there is an extension $\Pi' = \Pi \oplus \{p', p_i, ..., p_j\}$ with $\overrightarrow{Ref}$(p') = $\{c'\}$ such that:*

completely implementable concept

$$\forall c \in C, \forall \sigma^{\Omega} \in \Sigma^{\Omega}.\ \sigma^{\Omega}(c') = c \ \wedge\ \overleftarrow{Ref}(c) \neq \emptyset \ \Rightarrow\ c \in \overrightarrow{Ref}[[\overleftarrow{t}\,(\sigma^{\Omega})](p')]$$

Intuitively, the concept c' is completely implementable if the program can be extended locally in order to implement the concept. A local extension is performed by simply adding a new program element (p') with the condition that the conceptual relations between the newly implemented concept c' and the concepts (c) that were already implemented ($\overleftarrow{Ref}(c) \neq \emptyset$) are reflected at the code level ($c \in \overrightarrow{Ref}[[\overleftarrow{t}\,(\sigma^{\Omega})](p')]$). We can also add other program elements ($\{p_i, ..., p_j\}$) that might be necessary for the implementation of p'.

Example 5.7: Examples of (non-)implementable concepts

Below we present an example of a domain ontology fragment that models persons and their addresses (left) and a program that implements a part of these concepts (right-top). On the one hand, the concept AGE is completely implementable (by simply adding a new attribute to the class `Person` – on the right-bottom of our example). On the other hand, the program is not conceptually extensible with respect to the concepts TOWN and POSTAL CODE since their relation with ADDRESS cannot be reflected in the program. In order to extend the program with these concepts we need to restructure the original code.

```
Person  hasProp Adress
Person  hasProp Age
Adress  hasProp Town
Adress  hasProp Postal Code
```

$C = \{Person,\ Address,\ Age,\ Town,\ Postal\,Code\}$
$e^{\Omega}(Person,\ Address) = hasProp$
$e^{\Omega}(Person,\ Age) = hasProp$
$e^{\Omega}(Address,\ Town) = hasProp$
$e^{\Omega}(Address,\ Postal\,Code) = hasProp$

```
class Person {
  String address;
}
```

$P = \{Person,\ String,\ address\}$
$e^{\Pi}(Person,\ address) = hasAtt$
$e^{\Pi}(address,\ String) = hasType$

```
class Person {
  String address;
  int age;
}
```

$P' = \{Person,\ String,\ address,\ age,\ int\}$
$e^{\Pi'}(Person,\ address) = hasAtt$
$e^{\Pi'}(address,\ String) = hasType$
$e^{\Pi'}(Person,\ age) = hasAtt$
$e^{\Pi'}(age,\ int) = hasType$

□

Discussion. In order to implement a new concept there can be many other constraints than the ones given by our definition of extension $\oplus$. A formal treatment of the semantic extensibility of programs can be found in (Krishnamurthi and Felleisen, 1998). In the work of Krishnamurthi

semantic versus conceptual extensibility

the extensibility is considered from a behavioral point of view. In comparison with Krishnamurthi's extensibility, our notion of extensibility ($\oplus$) represents the extensibility with respect to the domain model. Our notion of conceptual extensibility is clearly more conservative – i. e. the class of conceptually extensible programs is included in the class of behaviorally extensible programs defined in (Krishnamurthi and Felleisen, 1998). This happens because even if a program is not conceptually extensible, one can implement the extensions and integrate them with the existing implementation of the other domain concepts in an algorithmical manner. In the above example, the parts of the concept ADDRESS can be implemented as a convention on the format of the `string` object – e. g. at first comes the postal code and after this the name of the town such as "80336 München". However, in these cases we loose the explicitness in the correspondence between the program organization and the modeled domain. This correspondence would be realized only implicitly through computation.

5.3 Characterizing Distortion

Distortion occurs when the program relations between the implementation of related concepts correspond to not-similar relations at the conceptual level. In Figure 5.5 we present an intuitive overview over (non-)distorted situations. Each of these cases will be presented below in detail.

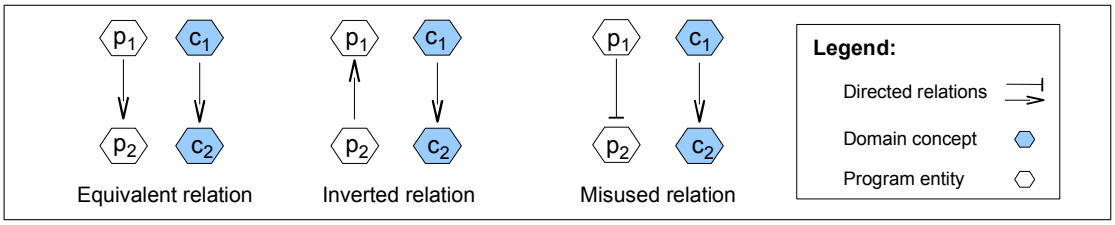


Figure 5.5: Equivalent, inverted or misused relations

equivalent relation

Definition 5.3.1 (Equivalent relation)**:** *Let $c_1, c_2 \in C$ be two related concepts with $c_2 \in \sigma^{\Omega}(c_1)$, and $p_1, p_2 \in P$ such that $p_1 \in \overleftarrow{Def}(c_1)$, $p_2 \in \overleftarrow{Def}(c_2)$. The implementation of the relation σ^{Ω} is* equivalent *iff:*

$$p_2 \in [\overleftarrow{t}(\sigma^{\Omega})](p_1)$$

Intuitively, if two related concepts are defined in the code, then the relation between their implementations reflects the relation from the domain ontology.

Example 5.8: Example of equivalent relation

As shown in Figure 5.6, in the Java API the WordNet hyponymy relation between LIST and STACK is implemented through a similar relation between the interfaces `java.util.List` and `java.util.Stack` (in this example we considered the hasSupCls relation to be transitive). □

Definition 5.3.2 (Inverted relation)**:** *Let $c_1, c_2 \in C$ be two related concepts with $c_2 \in \sigma^{\Omega}(c_1)$, and $p_1, p_2 \in P$ such that $p_1 \in \overleftarrow{Def}(c_1),\ p_2 \in \overleftarrow{Def}(c_2)$. The implementation of the relation σ^{Ω} is* inverted *iff:* *inverted relation*

$$p_1 \in [\overleftarrow{t}\,(\sigma^{\Omega})](p_2)$$

Intuitively, a relation between two concepts is implemented in the program through a similar relation but in the inverse sense. This case represents a form of distorted implementation which can lead to a non-intuitive (or false) usage of the implementations of concepts.

Example 5.9: Example of inverted relation

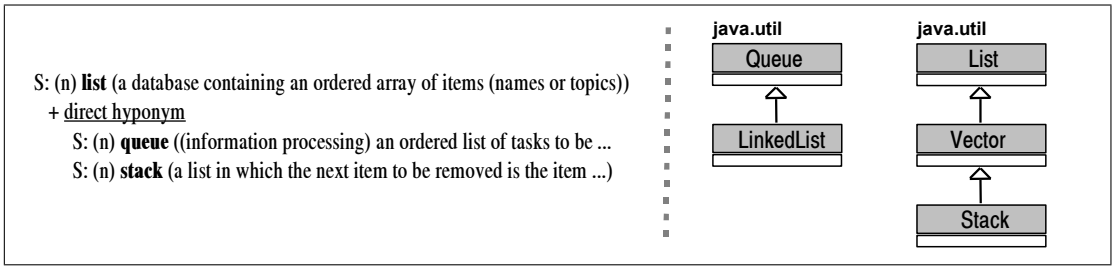


Figure 5.6: Example of equivalent and inverted implementation of relations

As shown in Figure 5.6, the WordNet hyponymy relation between LIST and QUEUE is implemented in the Java API through an inverted relation between the interface `java.util.Queue` and the class `java.util.LinkedList`. The consequence of the program relation is that whenever a `Queue` object is needed, we can use a `LinkedList` object. The latter can be easily used in a way that violates the `Queue` constraints. Below is an usage example of the `LinkedList` implementation for a `Queue` that breaks the ordering of elements within the queue: the `server_writeMessage` function writes always at the same end from which the client reads. The client will read the elements in an opposite order and thus our "queue" functions as a stack.

```
1.  Queue⟨Message⟩ aQueue = new LinkedList⟨Message⟩();
2.  void server_writeMessage() {
3.      ((LinkedList)aQueue).addFirst(new Message()); ...
4.  }
5.  ...
6.  Message client_readMessage(Queue aQueue) {
7.      return aQueue.element();
8.  }
```

□

Discussion. Many times the sources of distortion are the technical constraints imposed by the programming languages and the impedance mismatch between the conceptual level modeling and programming languages. For example, the well-known difference between sub-classing and sub-typing represents a source of distortion: when the inheritance is used primarily as a mechanism for reuse of implementation then it causes a distortion. In Figure 2.5 from Section 2.2.2

we presented an example of good and distorted object-oriented implementation of RECTANGLES and SQUARES. That example is similar with the example `Queue - LinkedList` from the Java API and that is presented above.

Definition 5.3.3 (Misused relation)**:** *Let $c_1, c_2 \in C$ be two concepts, related through the relation $\sigma^\Omega \in \Sigma^\Omega$ with $c_2 \in \sigma^\Omega(c_1)$, and let $p_1, p_2 \in P$ be two program elements such that $p_1 \in \overleftarrow{Def}(c_1)$ and $p_2 \in \overleftarrow{Def}(c_2)$. The implementation of the relation σ^Ω is* misused *iff:*

misused relation

$$\exists \sigma^\Pi \in \Sigma^\Pi.\ \sigma^\Pi \notin \overleftarrow{t}(\sigma^\Omega)\ \wedge\ p_2 \in \sigma^\Pi(p_1)$$

Intuitively, a relation between two concepts is implemented in the code through a relation with which it is not similar. This case represents a distortion since the model of the reality captured through the domain ontology is misleading at the code level.

Example 5.10: Example of misused relation

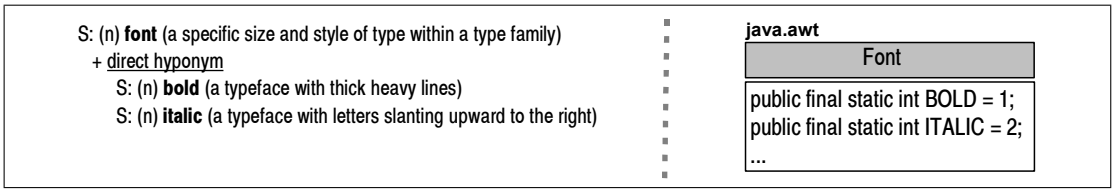


Figure 5.7: Example of misused relation

In Figure 5.7 we present an example of misused relation. In the `java.awt` package the BOLD FONT and ITALIC FONT concepts are implemented through the attributes `BOLD` and `ITALIC` of the class `Font`. This is contrary to the WordNet definition of BOLD FONT and ITALIC FONT according to which both of these concepts are hyponyms (sub-ordinates) of the concept FONT (and not parts as they are implemented). Adding more font types in this class will cause it to grow and will clutter its interface.

□

Remark. The usage of constants is a typical way to implement the hypernymy (is-a) relation in a light-weighted manner (not by creating subclasses). This is however an example of encoding of knowledge in programs since the relations between the program elements that implement the concepts are hidden.

Discussion. Many conceptual-level relations do not have a clear and unique interpretation in object-oriented languages. For example the triple: DRAW – actsOn – FIGURE can be implemented (in an object-oriented manner) as the method `draw` of the class `Figure` (`Figure` – hasMeth – `draw`), or can be implemented (in a procedural style) as parameter of the method draw (`draw` – hasParam – `Figure`). Whenever the relations are (slightly) misused, they increase the gap between the domain knowledge and the programs. The misused relations also hamper the extensibility of programs.

5.4 Characterizing Diffusion

The diffusions of concepts represent cases when pieces of the domain knowledge are commingled in the code. In the following subsections we discuss in detail the diffusions generated by the reference and representation of concepts.

5.4.1 Reference Diffusion

Definition 5.4.1 (Direct reference)**:** *Let $p \in P$ and $c \in C$. The program element p represents a* direct reference *iff:* *direct reference*

$$\overrightarrow{Ref}(p) = \{c\}$$

Intuitively, program elements that refer to only one concept are direct reference.

Definition 5.4.2 (Compacted reference)**:** *Let $p \in P$ and $c_1, ..., c_n \in C,\ n > 1$. The program element p represents a* compacted reference *iff:* *compacted reference*

$$\overrightarrow{Ref}(p) = \{c_1, ..., c_n\}$$

Intuitively, program elements that refer to several distinct concepts exhibit compacted reference (Figure 5.8). The concepts whose reference is compacted in the same program element are called *intimate program neighbours*.

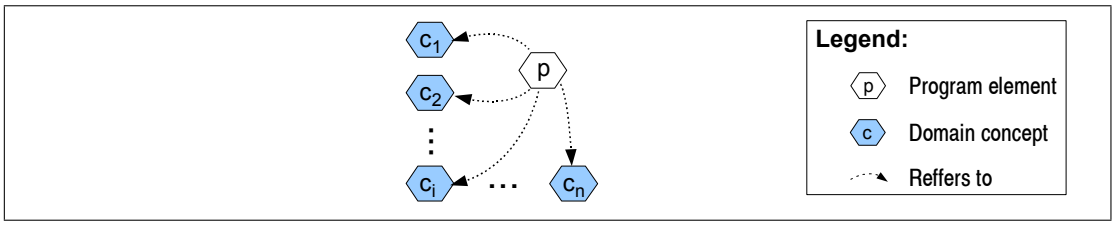


Figure 5.8: The program element p is a compacted reference; the concepts $c_1, ..., c_n$ are intimate program neighbours

Definition 5.4.3 (Intimate program neighbors)**:** *Let $C' = \{c_1, ..., c_n\}$ be a subset of the set of concepts C. C' represents* intimate program neighbors *of a concept $c \in C \setminus C'$ iff* *intimate program neighbors*

$$\forall c_i \in C'.c_i \in \overrightarrow{Ref}\left[\overleftarrow{Ref}(c)\right]$$

Intuitively, intimate neighbors of a concept c are all those concepts c_i that are referred by program elements that also refer to c. In Figure 5.8 we illustrate that the concepts c_i are intimate program neighbors of each other since the program element p refers to all of them.

Example 5.11: Examples of direct reference, compacted reference and intimate program neighbors

In the upper part of Figure 5.9 we present an ontology fragment that contains several concepts representing graphical figures, their properties and graphical operations. In Figure 5.9 (down-left) we present an example of the implementation of the drawing functionality in the Java AWT library (since Java 1.1). The references of concepts DRAW and FILL are compacted with the reference of concepts LINE, OVAL and RECTANGLE. Even if there is a clear relation between drawing actions and the geometrical figures (e. g. the draw action is performed over figures and thereby we have the relation "DRAW – actsOn– LINE"), these relations are encoded implicitly in the method bodies of `drawRect`, `drawLine`, etc. Furthermore, due to the compact reference, we notice also a pollution of the interface of `Graphics`: we have several program elements that implement combinations between these concepts (e. g. `drawOval()`, `fillOval`, `drawRect()`, `drawLine()`). Figure 5.9 (down-right) presents the direct reference of the same concepts by new program elements that were added in the Java API more recently (the version 1.2).

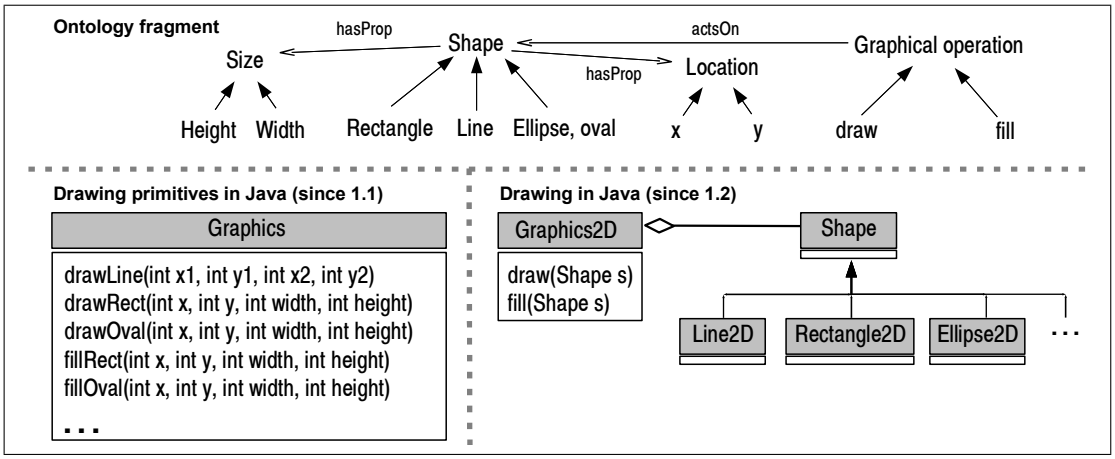


Figure 5.9: The concepts DRAW, FILL, LINE, OVAL and RECTANGLE are intimate program neighbors (down-left) or exhibit direct reference (down-right)

□

Discussion. Ideally, each program element should refer to only one concept and the program should combine the concepts exclusively by using the structuring and composition mechanisms provided by the modularization and composition features of the language (e. g. class membership, method call, parameter passing, overloading, overriding). In the case of compacted reference more concepts are weaved into a single program element. In many cases it is difficult to determine which parts of the program element implementation belong to a concept and which to the others. It is even more difficult to find where and how the composition between them is realized. Program elements that represent compacted implementation negatively affect the modularization of a program by intermingling more concepts and thereby by creating implicit and hidden coupling at the code level between the implementation of distinct concepts. In the

case of APIs the diffusion of reference prevents the API users to access individual concepts and allows the access of only pre-defined combination of concepts.

The compacted implementation is similar to the *interleaving* of programming plans (Rugaber, 2000). If we consider the implementation of each concept to be a plan, then compacted implementation represents two or more plans that are merged within a method, class, or variable.

5.4.2 Representation Diffusion

Definition 5.4.4 (Faithfully represented concept)**:** *A concept* $c \in C$ *is* faithfully represented *iff:* *faithful representation*

$$\overrightarrow{Rep}\,[\overleftarrow{Rep}(c)] = \{c\}$$

Intuitively, a type is used in the API to represent only one concept and each concept is represented through only one type. This is an ideal case when a program uses distinct representations for distinct concepts and the same representation whenever a concept is referenced several times in the program. Failing to do this makes the combination of the concepts unnatural and error-prone (as illustrated in Figure 3.17, p. 89).

Definition 5.4.5 (Representation overloading)**:** *The type* $p \in P$ *exhibits* representation overloading *iff:* *representation overloading*

$$\exists c_1, c_2 \in C.\ c_1 \neq c_2\ \wedge\ p \in \overleftarrow{Rep}(c_1)\ \wedge\ p \in \overleftarrow{Rep}(c_2)$$

Intuitively, a type exhibits an overloaded representation if it is used to represent more concepts (i.e. there are several variables with this type, each variable refers to a single concept (c_1 and c_2) and the concepts referred by these variables are distinct $c_1 \neq c_2$).

Example 5.12: Examples of representation overloading

In the fragment from the Java API presented in Figure 5.9, the type `int` exhibits representation overloading since it is used to represent the concepts X, Y, WIDTH and HEIGHT. This fact leads to unwanted combinations of concepts. A dangerous case of overloading is when two parameters of a method have the same type as in our example, since it allows clients to call the method with exchanged parameters and thereby to introduce bugs.

In Figure 5.10 we present another example of two parts of the Java library that implement functionality related to class loading. In this code fragment the type `String` represents an overloaded representation since it is used to represent both the CODEBASE address as well as the NAME of the class to be loaded. Furthermore, the order of parameters of these two methods is changed: the first parameter of the `RMIClassLoader.loadClass` has the meaning of the second parameter of `Util.loadClass` and vice-versa. Due to this fact, there are very high chances of misuse of `RMIClassLoader` by the programmers that already used the `Util` class.

□

```
package java.rmi.server;
public class RMIClassLoader { ...
    public static Class loadClass(
                String codebase,
                String name,
                ClassLoader defaultLoader) { ...
    }
}
```

```
package javax.rmi.CORBA;
public class Util { ...
    public static Class loadClass(
                String className,
                String remoteCodebase,
                ClassLoader loader) { ...
    }
}
```

Figure 5.10: Dangerous overloading example

Discussion. In practice, many times programmers do not use distinct representations for concepts. Instead, several concepts are represented through the same type and thereby are diffused – the language cannot recognize that these variables refer to different (and many times incompatible) concepts. The more concepts are represented through a single type, the more freedom in composing concepts we have and the easier it is for the clients to make uncaught logical mistakes. In the case of APIs, they are easy to misuse. Furthermore, the usage of representation overloading (different incompatible concepts are implemented as variables of the same type) nullifies the advantages of the type system.

The representation overloading seems to have been the root cause of the well-known Mars Climate Orbiter accident (NASA, 1999, p.16). The accident happened because the software used interchangeably two different concepts: the metric and the English based measurements of distances. The designers should have been used distinct representation for these distinct concepts and this should have warned them up-front and not allowed the interchanged usage of metric and English measurements.

Quantifying representation diffusion

overloading degree

Definition 5.4.6 (Overloading degree)**:** *Let $p \in P$ be a type. We define the* overloading degree (OD) *of p to be:*

$$OD(p) = \|\overrightarrow{Rep}(p)\|$$

Intuitively, the overloading degree quantifies the number of concepts represented through the same type. In the case of types that do not represent any concept ($\overrightarrow{Rep}(p) = \emptyset$) the value of OD is zero, otherwise the value of this metric is greater or equal than one; in the case when $OD(p) = 1$ then the type exhibits no overloading.

Example 5.13: Example of overloading degree

In case of the Java API fragment from our example from Figure 5.9 the overloading degree of `int` is 4 since `int` is used to represent four concepts – namely X, Y, WIDTH, and HEIGHT.

In our experiments performed on different Java APIs, and presented in Section 10.5.2, we found out that the primitive types and the type `java.lang.String` have a high overloading degree. This follows from the usage of numbers and strings as basic building blocks to represent real-world concepts whenever programmers do not have anything better at hand.

□

Discussion. Usually the primitive types exhibit a high overloading degree. This happens because some primitive types seem to be "natural" for representing domain concepts (e. g. `float width`, `String address`, `String name`). In our opinion this view is highly flawed since, for example, NAMES are not strings but rather more complex structures that have FIRST NAME, LAST NAME or even MIDDLE NAME; WIDTHS are not floating point numbers but also have a unit of measure, and so on. Furthermore, whenever two concepts are represented through the same type then the program cannot distinguish among them (in our example for the program NAMES can be interchangeably used with ADDRESSES. The higher the overloading degree, the lower the modeling level of the domain in the program and the higher decay the code exhibits. In Section 10.5.2 we present our experience with approximating the overloading degree for several Java systems.

5.4.3 Definition Diffusion

Definition 5.4.7 (Clear definition)**:** *A concept $c \in C$ is* clearly defined *iff:* *clear definition*

$$\overrightarrow{Def}\ [\overleftarrow{Def}(c)] = \{c\}$$

Intuitively, a concept is clearly defined when the classes that define it, do not define any other concept. This is an ideal case when a program uses distinct definitions for distinct concepts.

Definition 5.4.8 (Compacted definition)**:** *The class $p \in P$ exhibits* compacted definition *iff:* *compacted definition*

$$\exists c_1, c_2 \in C.\ c_1 \neq c_2\ \wedge\ p \in \overleftarrow{Def}(c_1)\ \wedge\ p \in \overleftarrow{Def}(c_2)$$

Intuitively, a class exhibits compacted definition if it is used to define more concepts. In this situation is not clear which parts of the class define which concepts and how are the definitions of different concepts combined in the class.

Discussion. Classes represent in object-oriented programs the basic modularization mechanisms. The compacted definitions represent cases when the modularization at the code level does not mirror the conceptual-level modularization. The classes that define distinct concepts have usually a small cohesion (the methods used to define different concepts do not interact with each other). Furthermore, in the case of APIs, classes that exhibit compacted definition are more difficult to use since their interface is big and refers to different sets of unrelated concepts. Many times compacted definition leads to the segregation of the interface of the class.

5.4.4 Diffusion of Relations

Definition 5.4.9 (Directly implementable relation)**:** *The relation $\sigma^{\Omega} \in \Sigma^{\Omega}$ is* directly implementable *iff:* *directly implementable relation*

$$\overrightarrow{t}[\overleftarrow{t}(\sigma^{\Omega})]\ =\ \{\sigma^{\Omega}\}$$

Intuitively, a relation type can be implemented in the code unambiguously.

ambiguous program relation

Definition 5.4.10 (Ambiguous relation)**:** *The relation* $\sigma^{\Pi} \in \Sigma^{\Pi}$ *is* ambiguous *iff:*

$$|\overrightarrow{t}(\sigma^{\Pi})| > 1$$

Intuitively, a relation type from the program can be interpreted in more ways at the conceptual level. This impacts negatively the reverse engineering of relations represented within programs.

Example 5.14: Ambiguous interpretation of relations

The *hasAtt* relation between a class and one of its attributes can be interpreted in two different ways: as the *hasProp* relation between a concept and one of its properties and as *hasPart* between a concept and one of its parts.

□

Discussion. Due to the big conceptual gap between the real-world and the general purpose programming languages, several (few) relation types at the code level have to be used to implement a big number of relation types from the modeled domain. This situation leads to diffusions of conceptual relations in programs. In Section 6.4.2 we study in detail the diffusions generated by the implementation of conceptual relations defined in the Suggested Upper Merged Ontology in Java programs. We show that due to the conceptual distance between SUMO and Java, diffusions are inherent. Diffusions of relations are similar to *construct deficit* (Gehlert and Esswein, 2007) presented in Section 2.2.2: namely, the implementation language offers a single program relation for expressing distinct relations at the conceptual-level.

5.5 Characterizing Logical Redundancy

As presented on the last line of Table 5.1, we consider the logical redundancy from two perspectives: redundancy in the representation and the definition of concepts.

5.5.1 Definition Redundancy

definition redundancy

Definition 5.5.1 (Definition redundancy)**:** *A concept* $c \in C$ *exhibits (logical)* definition redundancy *iff:*

$$\exists p_1, p_2 \in P.\ \overrightarrow{Def}(p_1) \ = \ \overrightarrow{Def}(p_2) \ = \ \{c\} \ \wedge \ p_1 \neq p_2$$

Intuitively, definition redundancy are different program elements that are used to define the same concept.

Example 5.15: The definition of the concept POINT in Java AWT exhibits redundancy

The POINT concept is defined in the AWT part of the Java standard library three times – namely, through classes `java.awt.Point`, `java.awt.geom.Point2D.Float` and `java.awt.geom.Point2D.Double`. The reasons for these definition redundancies are the different representations chosen for the coordinates – in the scopes of these classes we have $\overleftarrow{Rep}(x) = \{\texttt{int}, \texttt{float}, \texttt{double}\}$.

```
public abstract class Point2D ... {
    public static class Float extends Point2D {
        public float x, y; ...
    }

    public static class Double extends Point2D {
        public double x, y; ...
    }
    ...
}

public class Point extends Point2D ... {
    public int x, y; ...
}
```

□

Example 5.16: The definition of days of the week and of months in the Java library exhibit redundancy

Since the version 1.5 of Java we can define enumerations. Before the Java 1.5 version, programmers encoded enumerations through public constants. Below we present two examples of duplications in encoding of enumerations with constants. The concepts of the DAY OF THE WEEK: SUNDAY, MONDAY, TUESDAY, etc. are implemented identically as static constants in the classes `Calendar` and `BaseCalendar`.

```
package java.util;
public abstract class Calendar {
    public final static int SUNDAY = 1;
    public final static int MONDAY = 2;
    ...
    public final static int JANUARY = 0;
    public final static int FEBRUARY = 1;
    ...
}
```

```
package sun.util.calendar;
public abstract class BaseCalendar {
    public final static int SUNDAY = 1;
    public final static int MONDAY = 2;
    ...
    public final static int JANUARY = 1;
    public final static int FEBRUARY = 2;
    ...
}
```

Another example of logical redundancy is the implementation of months. As we can see in the above code, they are implemented as public static constants with different values (e. g. `Calendar.JANUARY = 0` and `BaseCalendar.JANUARY = 1`). We emphasize that in the Java library these two classes are used several times together (e. g. `BaseCalendar` is used in the implementation of `Calendar`). Below we give a possible (imaginary) example of a bug originating from the interchanged usage of the months.

```
aCalendar = new java.util.Calendar();
...
void trickySetDate(int year, int month, int day) {
    aCalendar.setMonth(year, month, day);
}
...
trickySetDate(2006, BaseCalendar.JANUARY, 30);
```

□

Discussion. Ideally, a domain concept should be defined only once in a program. When we use different definitions of a concept at the program level we introduce logical redundancy since we cannot use them in an uniform and seamless manner. We need to build adapters that transform between different definitions of the same concept. Many redundancies occur due to some inherent technical constraints such as efficiency: in the first example from above, the concept POINT is defined more times in order to accommodate both expressiveness (the coordinates are real numbers) and efficiency (integers are manipulable more efficiently); in the second example the redundancy was favored also by the lack of advanced constructs for defining enumeration in Java versions earlier than 1.5. In Section 10.5.3 we present our experience with the detection of definition redundancy in the Java AWT API.

5.5.2 Representation Redundancy

representation ambiguity

Definition 5.5.2 (Representation ambiguity)**:** *A concept* $c \in C$ *exhibits* representation ambiguity *iff:*

$$\exists p_1, p_2 \in P.\ p_1 \in \overleftarrow{Rep}(c)\ \wedge\ p_2 \in \overleftarrow{Rep}(c)\ \wedge\ p_1 \neq p_2$$

Intuitively, a concept is ambiguously represented in a program if it is represented through distinct types.

Example 5.17: Example of representation ambiguity

As explained in the previous section, the representation ambiguity leads to additional, not wanted complexity in combinations of concepts at the API level (Figure 3.17). Below we present cases of ambiguity in the Java SWING API: the concepts that refer to cardinal points are represented both as strings and as integers.

```
package javax.swing;
public class SpringLayout {
   public static final String NORTH = "North";
   public static final String SOUTH = "South";
}
```

```
package javax.swing;
public class SwingConstants {
   public static final int NORTH = 1;
   public static final int SOUTH = 5;
}
```

Figure 5.11: Example of representation ambiguity

□

Discussion. Whenever a concept is ambiguously represented, the programmers cannot uniformly use this concept at the program level and many times they should convert between its different representations. This leads to a significant redundancy that negatively affects the programmers in general. When the ambiguities occur in APIs their users cannot work with the API in direct analogy with the domain.

5.6 Related Work

In this chapter we investigated different conceptual problems generated by the implementation of domain knowledge in programs. In the reverse engineering and program analysis literature, concepts similar to ours (coverage, distortion, diffusion and redundancy) occur in different contexts. Below we present several examples of other reverse engineering works that deal (in a way or another) with the mismatches in the implementation of high-level concepts (e. g. domain concepts or design concepts) in the code.

Diffusion of relations. For example, the ambiguities in the implementation of composition, association, and aggregation relations from UML, require a considerable effort for their recovery from programs (Guéhéneuc and Albin-Amiot, 2004):

> "While we can identify easily the inheritance relationship in Java (extends keyword), how would the aggregation relationship between class B and class C be expressed? As a field? As a collection? How would the implementation reflect an aggregation or a composition relationship? More generally: How to define binary class relationships so we can detect them in implementation?" (Guéhéneuc and Albin-Amiot, 2004)

In this case, the "domain" with respect to which the code is interpreted are parts of UML, and therefore the domain ontology contains the UML concepts. Guéhéneuc also notes the difference between UML relations that are directly expressible in the code and those that are only ambiguously implementable. In the latter cases, the correct interpretation of implementation-level relations needs additional information or lacks accuracy.

Degeneration of design patterns. Much work in reverse engineering was carried to recover design patterns from existing programs. Beside the most simple implementation of patterns, many times the patterns are reflected in programs in a degenerated manner. These nonmodularities and degenerated situations are similar to our notions of diffusion and distortion.

> "Pattern occurrences were often "degenerated" in that many conceptual roles did not exist as distinct program elements, but were cluttered onto a few, more complex ones." (Florijn et al., 1997)

The "degeneration" of the design patterns implementation is similar to our notion of diffusion – i. e. more concepts (roles) are implemented by the same program element. However, our notion of diffusion is more general and is not defined only on patterns but for domain concepts (for a larger variety of domains) – if we consider a domain ontology about design patterns, then our definition of diffusion would match the Florijn's notion of degeneration.

Logical redundancy. The problem of redundant implementation is in-depth studied in the community of reverse engineering. However, with a few exceptions the redundancy is approached and evaluated at the syntactic level in form of the detection of code clones. We advocate that logical redundancy is even more pervasive and dangerous in the programs since it is most of the times not wanted and programmers (and most tools) are unaware of it.

Marcus and Maletic (2001) present a usage scenario for the Latent Semantic Indexing to identify conceptual clones in programs. LSI is used to determine semantic similarities between source documents (e.g. files, classes) and thereby to identify the documents that implement the same concepts – i. e. the documents with high semantic similarity are candidates to implement the same concepts. The definition of concepts in LSI is only vague (as a set of words) and the linking of concepts to the code is weakly defined. Our notions of representation ambiguity (redundancy) and definition redundancy allow us to define the logical redundancy more precisely.

Logical coverage. By examining the user interface of interactive programs (e. g. office applications) one can build an ontology of the domain. For example, Hsi et al. (2003) present an approach for building an ontology of the application domain of a program by manually analyzing its graphical interfaces (GUI). The concepts in the ontology are given by the labels of different graphical widgets (e. g. dialogs, menus) and the relations are chosen from is-a, has-a and different associations. This ontology is subsequently used to discover and investigate the central concepts of an application and that can have a significant impact on its evolution (logical extensibility). A major drawback of this work is that it does not take into consideration how are the domain concepts implemented in the program but only how are they reflected in the GUI. For example, due to different implementation decisions, a concept that is central in the GUI can be extended easier while a marginal concept from the GUI can be more difficult to extend.

5.7 Summary

Based on our mismatches classification framework presented in Chapter 4, in this chapter we characterize the logical mismatches that can occur between the domain knowledge and programs. For each mismatch category we discussed its influence on the typical maintenance activities and whenever we could, we gave examples from the Java standard API in order to point out the pervasiveness of these mismatches in practice.

Outlook. In the next parts of this dissertation, there are two other chapters that are related to the current one. Firstly, in the next chapter we investigate the measure in which domain knowledge can be faithfully implemented in the code. By studying the implementation strategies of the concepts and relations from the Suggested Upper Merged Ontology (SUMO) in Java programs, we will show that due to the conceptual gap between the code and the "real-world" knowledge (as described by SUMO), the implementation will be inherently mismatched. Secondly, in Chapter 10 we present our experience with detecting these mismatches in practice.

6 From Suggested Upper Merged Ontology to Java

> The limits of my language mean the limits of my world.
>
> Ludwig Wittgenstein[1]

Abstract: Programs implement parts of the real world. Programming language constructs are the means for reflecting the application domain knowledge in programs. There is a big conceptual gap between the domain knowledge and the object-oriented programming languages constructs. A small number of general language constructs need to accommodate a huge variety of domain-specific situations that should be implemented in programs. In the previous chapters, we showed that due to this gap, the implementation of domain concepts in programs leads to loss of conceptualization and this subsequently favors mismatches in the reflexion of domain models in the code. In this chapter we investigate possible implementations of real-world concepts that are at the generality level comparable with the constructs of object-oriented languages. We ground our study on the similarities between the concepts of the Suggested Upper Merged Ontology (SUMO) (Niles and Pease, 2001b) and the Java object-oriented programming constructs and its core library. We focus on a direct and accurate implementation of SUMO concepts and ignore other aspects such as efficiency, technical constraints, or interaction with other technologies. Our motivation is to analyze the ideal ways of how can the most general ontological concepts of SUMO be expressed by using the Java programming language. Thereby, we investigate the conceptual limitations of the mainstream object-oriented languages (with focus on Java) to accurately reflect parts of the real world in programs. These limitations are the root for many mismatches between programs and the modeled domain that can be hardly avoided (or, many times, cannot be avoided at all).

Structure of this chapter. After the introduction (Section 6.1), in Section 6.2 we present the Standard Upper Merged Ontology (SUMO) that comprises the most general concepts from the real-world. Section 6.3 presents an ontology of the core object-oriented programming knowledge that includes language constructs, core object oriented programming idioms and core standard libraries (we focus on the features commonly used in Java). In Section 6.4 we identify typical implementation possibilities of the most general real-world concepts by mapping SUMO on our ontology of object-oriented programming knowledge. In Section 6.5 we discuss the limitations of the Java language to explicitly reflect the SUMO concepts. In Section 6.6 we present the related work on ontological evaluation of modeling languages and in Section 6.7 we present a summary of this chapter.

[1] Austrian philosopher

6.1 Introduction

In the quotation at the beginning of this chapter the expression "my language" can be easily understood by computer scientists since everyone works daily with more (e. g. programming, modeling, specification) languages. The "limits" mean the measure in which the language supports expression of facts in direct analogy to the domain knowledge and the automation in analyzing and manipulating the content that is described with this language (e. g. programs, models, specifications). What does "my world" mean is however more problematic and the answer stays most of the times outside of computer science – most of the content we produce addresses other domains (e. g. banking, automotive). In this chapter we use a description of the world as it is given by the IEEE Suggested Upper Merged Ontology (SUMO) (Niles and Pease, 2001b) and compare it with an ontology of basic object-oriented programming knowledge (with focus on Java). In this manner we aim to investigate the limits of Java with respect to SUMO.

Terminological clarification. In the previous chapters we used the term ontology to denote an artefact – namely a set of concepts and relations that describe a domain. We needed these artefacts in order to analyze programs (other artefacts). In this chapter we focus on *ontological analysis* (Gehlert and Esswein, 2007), namely, we are interested in the kinds of entities that can exist in the real-world, the kinds of object-oriented features of Java and the mappings between them – i. e. how can the real-world concepts (given by SUMO) be implemented in Java.

ontological analysis

> The Suggested Upper Merged Ontology is an ontology (artefact) that was obtained through analysis of the categories of things that exist in the real-world.

Ontologies specificness spectrum. According to the domain specificity of the concepts that they represent, ontologies are classified into: domain specific, mid-level and upper-level – Figure 6.1(left). *Domain ontologies* contain concepts from a particular domain – e. g. an ontology about the SSL security protocol contains concepts used by this protocol such as X.509. *Mid-level ontologies* share concepts that cover more general domains that subsume families of related domains – e. g. an ontology of the data communication contains concepts about the data transfer. Mid-level ontologies act as bridges between domain ontologies and *upper ontologies* . The latter contain concepts that are abstract enough to subsume concepts from all domains. Upper ontologies originate from the studies of philosophers, linguists and knowledge representation experts.

domain ontology

mid-level ontology

upper ontology

> Upper-ontologies describe the organization of the real-world knowledge in terms of the most general concepts.

Remark. There is a strong similarity between the concepts from upper ontologies and core constructs of object-oriented programming languages (Figure 6.1 - top), between concepts from the mid-level ontologies and classes from standard software libraries (Figure 6.1 - middle) and between concepts of domain specific ontologies and entities of domain specific libraries (Figure 6.1 - bottom). In Section 6.4 we discuss the correspondence between the Suggested Upper Merged Ontology and the core object-oriented constructs from the Java programming language.

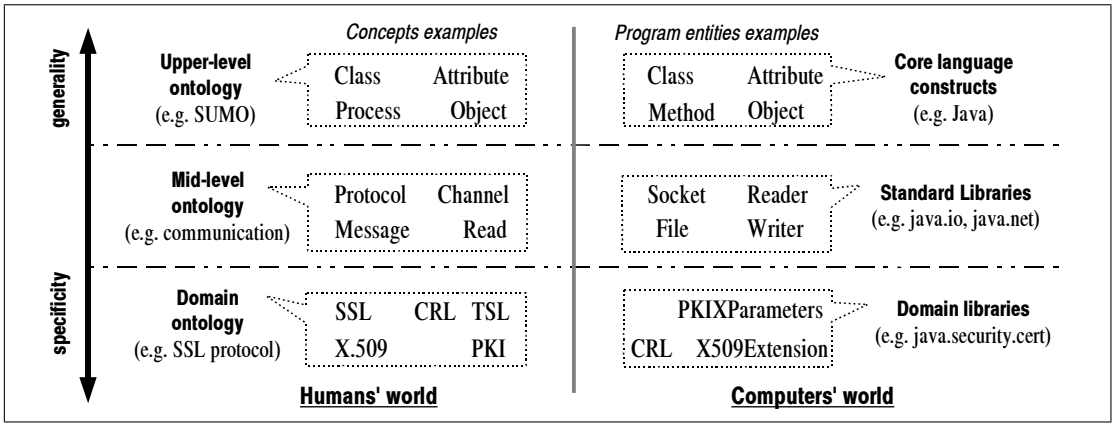


Figure 6.1: Ontologies specificness (left); Programming concepts specificness (right)

Based on this correspondence we investigate the possibilities to extend our set of conceptual level relations (Σ^{Ω}), our set of program relations (Σ^{Π}) and the mappings between them ($\overleftrightarrow{t}$).

Well-known upper ontologies. Describing the real-world as a whole is an extremely complex process due to the intricacies, the high number of details and the multitude of perspectives under which the same phenomena can be considered. There are currently several well-known upper ontologies that cover the abstract real-world knowledge from different perspectives: DOLCE (Masolo et al., 2003), Sowa's Upper Ontology (Sowa, 2000), or Suggested Upper Merged Ontology (SUMO) (Niles and Pease, 2001b). As we will see in the following, we have chosen SUMO because it describes the real-world in a similar manner to our conceptual layer – namely based on concepts and relations among them.

6.2 The Suggested Upper Merged Ontology (SUMO)

The IEEE Standard Upper Ontology Working Group[2] defines a general-purpose, upper-level, formal ontology that is extensible with domain specific ontologies. SUMO is designed to be useful for a large variety of purposes, such as integrating domain ontologies, design of new knowledge bases or enhancing the applications interoperability. The standardization effort is sustained by specialists from various domains like information science, philosophy and engineering, both from academia and from industry. SUMO did not start from scratch but by merging publicly available (fragments of) upper-level ontologies into a single comprehensive and cohesive structure (Niles and Pease, 2001a). In our work we will use the version 1.73 of SUMO published in September 2005[3].

> We analyze how can the most general real world concepts be implemented in programs. *We assume that SUMO is an accurate, technical independent and humans-oriented description of the most abstract concepts of the real world.*

[2] `http://suo.ieee.org/`
[3] `http://suo.ieee.org/SUO/SUMO/index.html`

6.2.1 SUMO Top-Level Concepts

In Figure 6.2 we present a subset of the most general concepts of SUMO[4]. All the other SUMO concepts (ca. 200) are derived through a (possible multiple) subordination relation from these concepts. As SUMO is intended to describe the real world (including feelings, natural sciences), and our interest is only restricted to the common knowledge captured in programs, we will present in the following a subset of SUMO concepts that satisfies our needs.

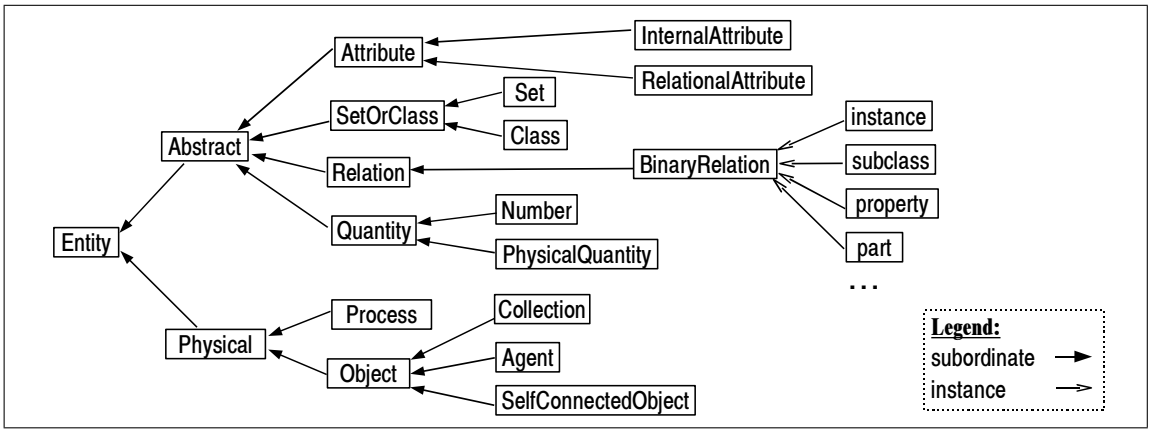


Figure 6.2: A part of the most abstract SUMO concepts

ENTITY is the most general concept of SUMO and represents the universal class of individuals. It is the root of the concepts hierarchy and thus, it subsumes the other concepts in the world. ENTITIES are exhaustively partitioned into: PHYSICALS, representing everything that has a well defined position in space and time; and ABSTRACTS, representing properties or qualities as distinguished from any particular embodiment in a physical medium.

OBJECTS are PHYSICALS that exist in full at any instant at which they exist at all. They correspond roughly to the class of ordinary objects in the real world. Among the kinds of OBJECTS distinguished by SUMO are:

1) AGENTS represent those OBJECTS that can act on their own and produce changes in the world (e. g. humans). AGENTS are the active determinants of PROCESSES.

2) COLLECTIONS represent those OBJECTS whose instances consist of disconnected parts that can be added or subtracted without thereby changing their identity (e. g. a "family" is a COLLECTION since the birth or the dead of one of its members do not affect its identity).

3) SELFCONNECTEDOBJECT represent those OBJECTS that do not consist of disconnected parts. COLLECTIONS are disjoint with SELFCONNECTEDOBJECT.

PROCESSES are the complement of OBJECTS in the PHYSICAL class and they subsume all entities that happen in time (e. g. reading). PROCESSES are only partially present at any time at which they are present. In the natural language, PROCESSES are denoted through verbs.

[4]For obtaining the descriptions of the concepts and relations from SUMO we used the following online ontology browser: `http://virtual.cvut.cz/ksmsaWeb/browser/title`

SETORCLASS is the common super concept of **SET** and **CLASS**. It subsumes any subordinate of ABSTRACT that has elements or instances. There are three differences between SETS and CLASSES: Firstly, CLASSES are not assumed to be extensional and thus distinct CLASSES might have exactly the same instances. SETS are extensional and two SETS with the same elements are always equal. Secondly, CLASSES typically have an associated 'condition' that determines their instances (e. g. common properties), while for SETS there is no associated condition that determines the membership. Thirdly, the instances of a CLASS may occur only once within the CLASS, while a SET can contain duplicated elements.

ATTRIBUTES are qualities which are not reified into subclasses. There are two kinds of ATTRIBUTES: INTERNAL ATTRIBUTES represent intrinsic properties of entities (e. g. shape, color); and RELATIONAL ATTRIBUTES are properties that entities have by the virtue of their relationships with other entities (e. g. social roles such as president).

RELATIONS are used to represent tuples of elements. Relations are classified according to their arity (e. g. BINARY RELATION, TERNARY RELATION). BINARY RELATIONS are relations defined between pairs of ENTITIES and are the most common relations encountered in the knowledge representation formalisms (e. g. binary relations are represented as slots in the frame-based knowledge representation paradigm). SUMO contains a large number of instances of BINARY RELATIONS defined between its concepts. A subset of these relations are presented in the following section.

6.2.2 SUMO Relations Between the Top-Level Concepts

Relation	Source (S)	Target (T)	Description
PROPERTY	ENTITY	ATTRIBUTE	S has an attribute T
INSTANCE	ENTITY	SETORCLASS	S is included in T
PART	OBJECT	OBJECT	S is part of T
EXPLOITS	OBJECT	AGENT	S is used by T for performing a PROCESS
MEASURE	OBJECT	PHYS.QUANT.	S is measured by the constant quantity T
AGENT	PROCESS	AGENT	T determines S
MANNER	PROCESS	ATTRIBUTE	S is qualified by T
INSTRUMENT	PROCESS	OBJECT	T is an instrument in performing S
RESOURCE	PROCESS	OBJECT	T is used and changed by S
CAUSES	PROCESS	PROCESS	S causes T
RESULT	PROCESS	ENTITY	T is the output of S
SUBPROCESS	PROCESS	PROCESS	S implies the execution of T
PATIENT	PROCESS	ENTITY	T is a patient of S
SUBCLASS	CLASS	CLASS	S is a subordinate of T
LESSTHAN	QUANTITY	QUANTITY	S is less than T
GREATERTHAN	QUANTITY	QUANTITY	S is greater than T

Table 6.1: A part of the SUMO relations between the most general concepts

In Table 6.1 we describe a part of the relations that are defined in SUMO between the concepts discussed in the previous section. These relations are good candidates for the set of relations at the conceptual layer (Σ^{Ω}).

6.3 An Ontology of Java Core Object-Oriented Programming Knowledge

As we presented in Section 3.4.2, we abstract programs as graphs whose nodes are named program entities and edges are relations among them. Up to this point, we considered the program nodes to be only the basic object-oriented program entities (classes, attributes and methods) and accessors. In this section we investigate in more detail the kinds of program entities that can make up the program layer and the types of relations among them. In Figure 6.3 we illustrate our Java programming knowledge ontology.

Remark. Some of the program entities and program relations presented below are defined explicitly by the syntax of the Java language and therefore they can be automatically extracted from the parse tree. Other program entities and relations are programming idioms and entities from the standard library and their identification in a program involves information that is not given by the Java language definition taken alone.

6.3.1 Named Program Entities

We consider only the program entities that have names associated with them (e. g. classes, methods, attributes, parameters). We categorize the program entities along the following directions:

1) Basic named entities. These entities represent all program entities that are explicitly defined by the language syntax and that have names.

- **Types** are either abstract datatypes (i. e. classes or interfaces), primitive types (e. g. int) and arrays (e. g. Object[]). **Classes** represent the main modularization mechanism. **Primitive types** are the most basic types of the Java language and are used as building blocks for defining abstract data types. **Arrays** are a convenience for representing fixed-size collections of objects.

- **Variables** are program entities used to denote references to run-time objects. According to where they are defined, variables are classified in: **attributes**, **parameters** and **local variables**.

- **Methods** are operations defined on classes. **Constructors** are special methods used for the basic initialization of objects at the creation time.

2) Idiomatic entities. Different languages provide different support for typical programming features. For example, in Java (as opposed to C# that has language constructs for properties)

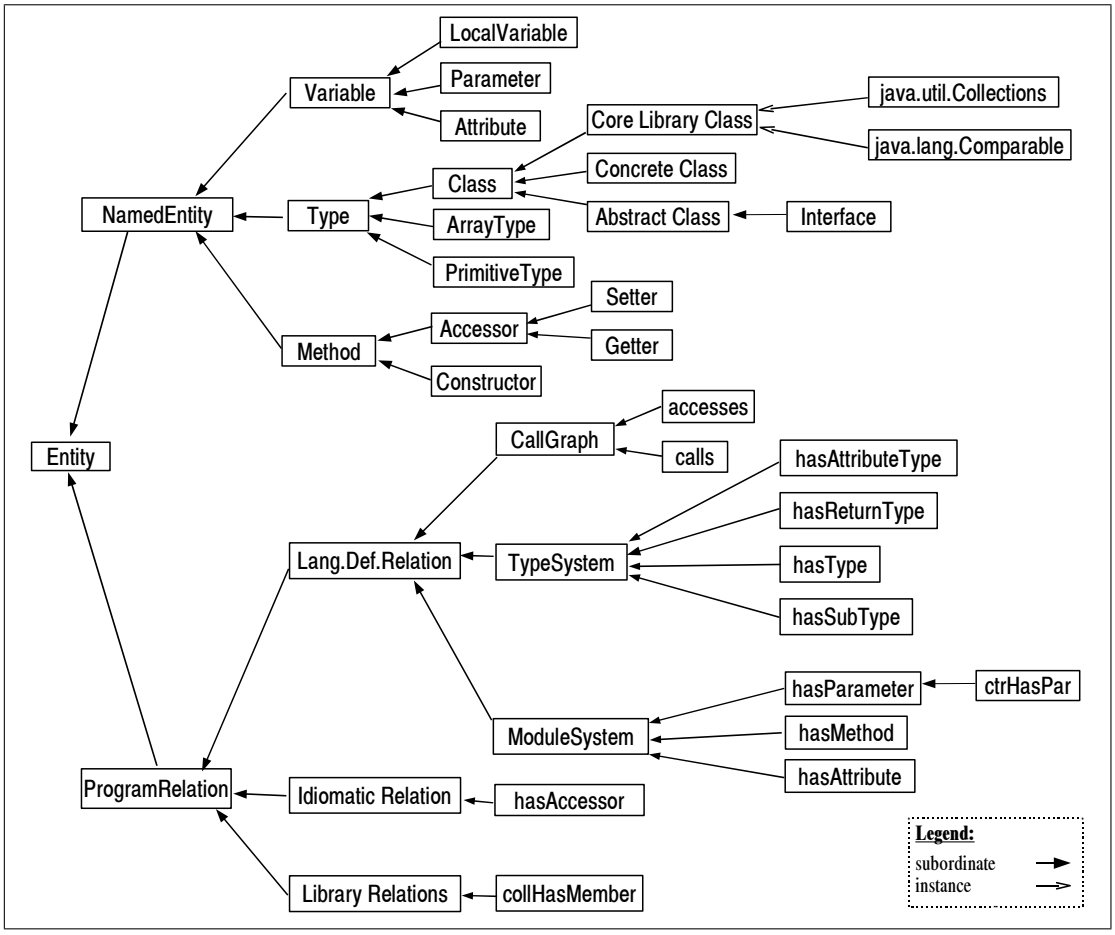


Figure 6.3: Java core programming knowledge

there is no built-in support for working with properties of objects. Such limitations are overcome in the practice through basic programming conventions (also known as idioms). **Accessor methods** (also known as setters and getters) are classical Java programming idioms. They are so important that they are at the core of wide-spreaded technologies such as JavaBeans.

> "If we discover a matching pair of "get<PropertyName>" and "set<PropertyName>" methods that take and return the same type, then we regard these methods as defining a read-write property whose name will be "<propertyName>". " (Sun, 1997, p. 55)

Programmers use idiomatic entities to overcome the lack of advanced features of programming languages.

3) Entities belonging to core libraries. Many general concepts are not reflected directly in the programming language syntax but are provided by their core libraries. Many times, there is no clear distinction between the programming language itself and its core libraries - i. e. the

semantics of the classes belonging to the Java core library goes beyond the "standard" semantics of user-defined classes. Among the Java standard library classes, we pay special attention to the following two:

- **Collections:** The objects of the interface `java.utils.Collection` are used as a standard Java mechanism for defining collections of arbitrary objects.
- **Comparable:** The objects of the interface `java.lang.Comparable` are used as a standard Java mechanism for defining ordering.

6.3.2 Program Relations

In addition to the named program entities, the relations represent an essential part of our program abstraction. In a similar manner with the classification of program entities, we distinguish among three categories of program relations (Table 6.2): 1) *language defined relations* are directly expressible within the language syntax, 2) *idiomatic relations* are relations between program entities representing programming idioms, and 3) *library-defined relations* are relations that involve entities from the core libraries.

Relation	Source (S)	Target (T)	Description
`hasSubType`	`Class`	`Class`	T is a subtype of S
`hasAtt`	`Class`	`Attribute`	S has attribute T
`hasMeth`	`Class`	`Method`	S has method T
`hasType`	`Variable`	`Type`	S has type T
`hasParam`	`Method`	`Parameter`	S has parameter T
`ctrHasPar`	`Constructor`	`Parameter`	S has parameter T
`hasRetType`	`Method`	`Type`	S has return type T
`calls`	`Method`	`Method`	S calls T
`accesses`	`Method`	`Variable`	S accesses T
`hasAcc`	`Class`	`Accessor`	S has method T
`collHasMember`	`Collection`	`Variable`	T is used in the `add` method call on S
`compareTo`	`Comparable` `Variable`	`Comparable` `Variable`	T is used as parameter of the `compareTo` method call on S

Table 6.2: Relation among core Java entities

6.4 From SUMO to Java

We presented in the last two sections the SUMO upper ontology and our ontology of the Java core programming knowledge. In the following we will give an answer to the question:

> *How can the SUMO concepts and relations be the most natural implemented using the programming concepts from our Java ontology?*

With other words, we aim to find the similarities and differences between the SUMO and Java ontologies. With respect to our question we emphasize two aspects: 1) we are interested in a faithful implementation, where the diffusions, distortions and implementation details are the smallest possible, and 2) we are interested in the Java core (object-oriented) features and we leave out other language details (e. g. primitive types).

Remark. Please note our typographical convention (presented also in Section 1.4): through SMALL-CAPS we denote the domain concepts (e. g. CLASS) and through `type-writer` fonts we denote the entities from Java programs (e. g. `class`).

6.4.1 Implementation of SUMO Concepts

Implementation of ENTITIES. ENTITIES are the most general concepts of SUMO and thus they subsume every concept that exists in the world. Therefore, they can be implemented in arbitrary program parts, as sequences of method-calls, any of their side effects, and so on. Our program abstraction is based on named program elements and relations among them, and this is why we are interested only in how the subconcepts of ENTITY are implementable by using the named program elements and relations.

Implementation of CLASSES. CLASSES are ABSTRACTS that have instances defined through a set of PROPERTIES. Correspondingly, Java `classes` are used to define instances, and each `instance` has a well defined set of properties (Java `attributes`). CLASSES are implemented through `classes` and `interfaces`.

Implementation of SETS. In SUMO, SETS are collections of arbitrary ENTITIES. The Java `arrays` and instances of the standard-library class `java.utils.Collection` provide support for dealing with arbitrary collections of `objects`. Therefore we consider that SETS are implemented by classes that implement the interface `Collection` or as variables with array type. The differences between SUMO CLASSES and SETS remain valid in Java: 1) different Java classes can have the same instances (classes that implement more interfaces) while collections with the same elements are equal; 2) every instance of a Java class has a set of common properties with the other instances (the Java attributes) while Java collections or arrays can contain arbitrary elements; 3) instances are singular within classes, while a collection can contain the same object several times (if the collection is not an instance of `java.util.Set`).

Implementation of PROCESSES. PROCESSES are ENTITIES that are unfolding in time. The execution of a program also unfolds in time and thus, PROCESSES are naturally implemented as

`methods`, parts of `methods`, or sequences of `methods`. With respect to our conceptualization of Java we notice two major limitations:

1. Since we only deal with the named program entities, and the parts of `methods` do not have a name, we consider that PROCESSES are implementable exclusively as `methods`.
2. In Java the methods are not first class entities (they cannot be sent as parameters nor can they be results of other methods). For this reason, many times the methods are encapsulated in objects (the well-known *Command* design pattern (Gamma et al., 1995, p. 233)). These classes whose sole purpose is to provide a wrapper for methods in order to allow them to be treated as first class entities is a typical example of implementation details.

Implementation of QUANTITIES. QUANTITIES can be explicitly implemented as instances of the class `java.lang.Comparable`. By doing so, the method `compareTo` is very similar to conceptual relations LESSTHAN and GREATERTHAN. However, these relations are implemented by an algorithm inside of the `compareTo` method.

Implementation of COLLECTIONS. Implementing COLLECTIONS as instances of `java.util.Collection` would be inadequate since in SUMO COLLECTIONS do not change their identity by adding or removing members. Thus, COLLECTIONS are implementable through classes which contain as one of their attributes an instance of `java.util.Collection`.

Implementation of other concepts. Beside the implementation of concepts described in the above paragraphs, there is no direct and explicit support in Java to implement other concepts from our SUMO fragment. For every other concept we can use a Java class to define it.

Implementation of RELATIONS. SUMO RELATIONS raise for us a special interest since they match very well the relations between concepts at the conceptual level. RELATIONS are defined in SUMO as tuples of ENTITIES. The implementation of arbitrary relations in object-oriented languages in general, and in Java in particular, is not directly supported through programming constructs. Even though, the importance of relations is widely acknowledged and led to different proposals for the extension of object-oriented languages (Rumbaugh, 1987; Bierman and Wren, 2005). In order to describe the fact that two program entities that correspond to two domain concepts are related we can:

1. use the small set of relations defined by the programming language (e. g. SUBCLASS is a binary relation directly expressible in Java), interpret the relations between program entities from the standard library (e. g. collHasMember is a relation between a collection and its members), or between different idioms (e. g. hasAcc is a relation between a class and one of its accessors),
2. use `attributes` to implement new relations between the class where they are defined and their types, or use `methods` to implement single-valued relations.

In the following we take the first approach and use the relations from our Java ontology to reflect the SUMO relations.

6.4.2 Implementation of SUMO Relations

For each SUMO relation presented in Section 6.2.2 we discuss its implementation in Java in one of the following paragraphs. Each paragraph has the following structure: at first, we briefly present the meaning of the relation in the SUMO ontology and then we discuss several strategies to directly reflect this relation in Java programs. To each paragraph corresponds a part of Figure 6.4 that illustrates with examples the implementation strategies.

Implementation of PROPERTY. PROPERTY(?Entity, ?Attribute) represents the relation between an ?Entity and one of its ?Attributes. As we illustrate in Figure 6.4a there are the following possibilities for implementing the PROPERTY relation:

1) `hasAtt(?Class, ?Attribute)` – between a `Class` and its `Attributes`,

2) `ctrHasPar(?Constructor, ?Parameter)` – between a constructor and one of its parameters.

3) `hasAcc(?Class, ?Accessor)` – between a class and one of its accessor methods.

Remark. The implementations presented above do not cover the whole spectrum of the PROPERTY relation – e. g. in SUMO one can define PROPERTIES of PROCESSES and they cannot be directly implemented in Java.

Implementation of INSTANCE. INSTANCE(?Entity, ?SetOrClass) represents the relation between ?Entity and the ?SetOrClass in which it is included. As we illustrate in Figure 6.4b there are two possibilities to implement the INSTANCE relation in Java:

1) `hasType(?Variables, ?Type)` – between a `variable` and its `type`,

2) `collHasMember(?CollectionVariables, ?Variable)` – between a `variable` whose type is `Collection` and another variable that is added to the collection (i. e. the actual parameter of a method `add`).

Implementation of MANNER. MANNER(?Process, ?Attribute) is a sub-relation of PROPERTY that means that ?Process is qualified by ?Attribute. In the natural language this relation is usually denoted through adverbs attached to verbs – e. g. MANNER(draw, quickly) means that quickly is a manner of drawing. The information about the MANNER in which a PROCESS is executed cannot be directly reflected in the source code since the methods cannot have attributes. However, an indirect possibility to implement PROCESSES that are executed in different MANNERS is via `parameters` that are used to switch between different execution "manners" of their methods. As we illustrate in Figure 6.4c, MANNERS can be as: `hasParam(?Method, ?Parameter)`.

Implementation of PART. PART(?Object1, ?Object2) is the basic mereological relation and expresses the fact that ?Object1 is part of ?Object2. As we illustrate in Figure 6.4d, the PART relation can be implemented through the `hasAtt(?Class, ?Attribute)` relation.

Implementation of RESULT. RESULT(?Process, ?Entity) means that ?Entity is the output of the ?Process – e. g. RESULT(translate, Point) means that Point is the result of move. As we illustrate in Figure 6.4e the RESULT relation between a PROCESS and its output ENTITY can be directly expressed in programs as the `hasRetType` relation between the `method` implementing

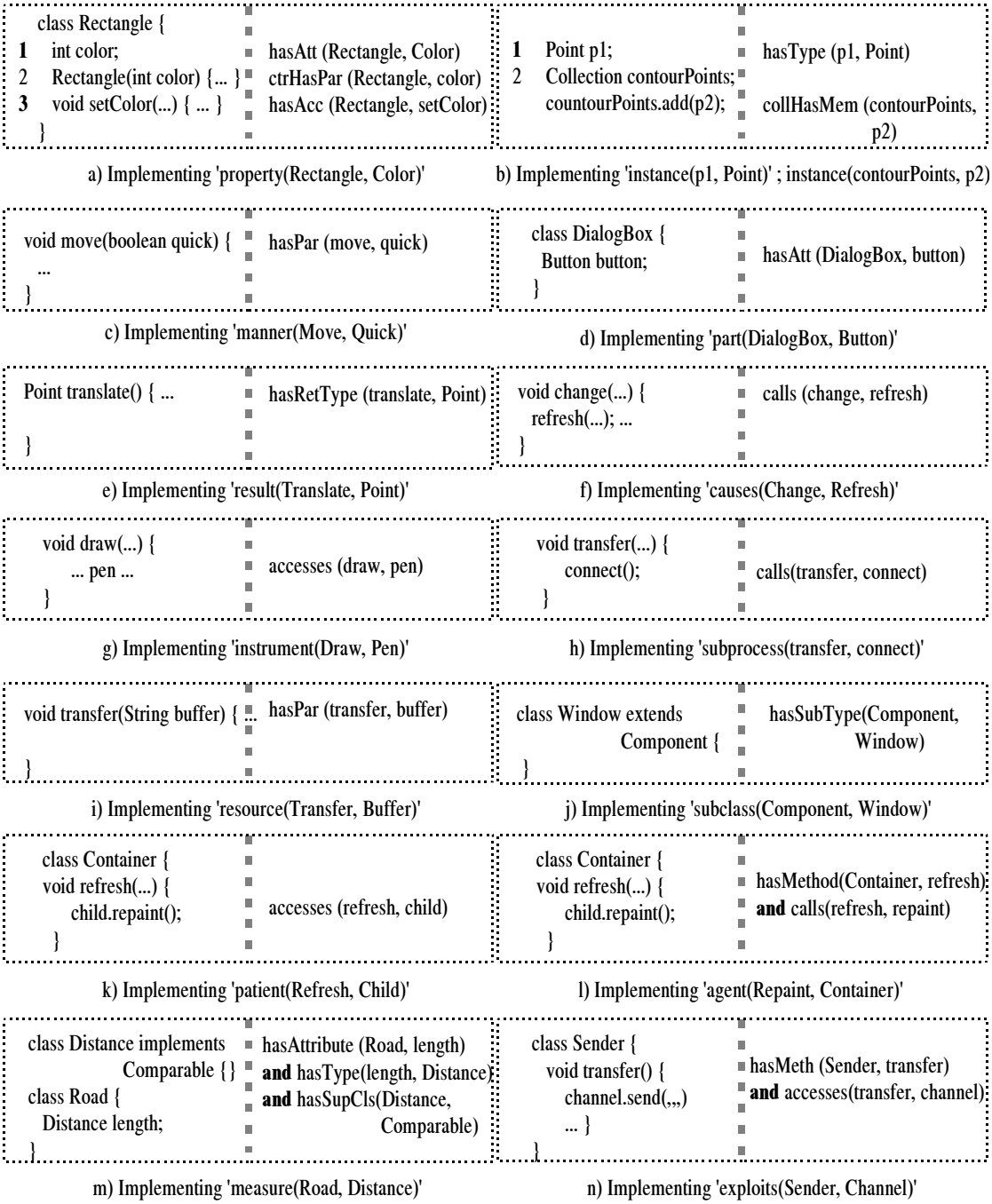


Figure 6.4: Possible implementations of SUMO relations in Java

that PROCESS and the return type that represents the class of the entity. However, it is often the case that the result of a method is a side-effect and these situations are not captured by our program abstraction.

Implementation of CAUSES. CAUSES(?Process1, Process2) means that ?Process2 is caused by ?Process1. As we illustrate in Figure 6.4f the CAUSES relation between two PROCESSES can be directly expressed in programs as the `calls` relation between the `method` that implements the source PROCESS to the `method` that implements the destination PROCESS. However, more sophisticated causality relations are not implementable by using our program ontology (we would need information about the control-flow). Furthermore, CAUSES can be also implemented by standard design patterns – e. g. in the Observer design pattern, the cause of an update is a change in the state of the subject.

Implementation of INSTRUMENT. INSTRUMENT(?Process, ?Object) denotes that ?Object is used by an agent to perform ?Process and ?Object is not changed. As we illustrate in Figure 6.4g, the INSTRUMENT relation can be represented in our program abstraction as the `accesses` relation between a `method` that implements the ?Process and a `variable` that refers the ?Object used as instrument.

Implementation of SUBPROCESS. SUBPROCESS(?Process1, ?Process2) denotes that ?Process1 implies the execution of ?Process2. As we illustrate in Figure 6.4h, the SUBPROCESS relation between two PROCESSES can be implemented through method calls.

Implementation of RESOURCE. RESOURCE(?Process, ?Object) relation between a PROCESSES and an OBJECT that it consumes can be directly represented as a relation between the `method` that implements the PROCESS and one of its `parameters` that refer to the resource (Figure 6.4i). However, Java methods can access other variables besides the parameters (e. g. `attributes`) and these variables can play the role of RESOURCES as well.

Implementation of SUBCLASS. The SUBCLASS relation is directly expressible in Java through the subtyping relations `hasSubType` between `classes` or `interfaces` (Figure 6.4j).

Implementation of PATIENT. PATIENT(?Process, ?Entity) is a relation between a PROCESS and an ENTITY that is a direct participant in the process. The PATIENT relation is used to express as broadly as possible the participants of processes – such a participant may or may not undergo changes as result of the process (e. g. INSTRUMENT, RESULT, RESOURCE are sub-relations of PATIENT). This relation can be directly represented as a relation between the `method` that implements the PROCESS and one of the variables accessed in the method representing the ENTITY (Figure 6.4k).

Implementation of AGENT. AGENT(?Process, ?Agent) is a relation between a PROCESS and one of its active determinants. The AGENT can be directly implemented as a relation between a `class` that implements the AGENT and one of the `methods`, that implement the PROCESS, and that are called from this `class` (Figure 6.4l).

Implementation of MEASURE. MEASURE(?Object, ?PhysicalQuantity) is a relation between an OBJECT and a quantity that measures it. This relation can be implemented between a `class` that implements the OBJECT and one of its attributes that is a sub-type of `Comparable` and that

implements the QUANTITY (Figure 6.4m).

Implementation of EXPLOITS. EXPLOITS(?Object, ?Agent) is a relation between an AGENT and an OBJECT that is used as a resource for performing a PROCESS that the AGENT determines. This relation can be implemented between a `class` that implements the AGENT and a `variable` (implementing the exploited OBJECT) that is accessed from a method that implements the PROCESS (Figure 6.4n).

Implementation of LESSTHAN and GREATERTHAN. LESSTHAN(?Quantity, ?Quantity) and GREATERTHAN(?Quantity, ?Quantity) are relations between two QUANTITIES. As we presented before, QUANTITIES are implemented as instances of the interface `java.lang.Comparable`. The LESSTHAN and GREATERTHAN relations can be implemented only algorithmically (implicit) in the body of the method `compareTo`.

6.5 Discussion on Conceptual Limitations of Java

The results from our analysis on typical mappings between the Java programming knowledge ontology and SUMO, show that Java is able to express explicitly only a few concepts from SUMO: CLASSES, PROCESSES, SETS, and QUANTITIES. Even these concepts can be only partially expressed in Java (not every relation between them can be reflected in the corresponding Java constructs). All the other SUMO concepts can be implemented (in the most explicit way) only as user defined classes. We remark that apart from a small set of RELATIONS instances that can be directly expressed through programming language defined relations (e. g. SUBCLASS), or relations among entities from standard libraries (e. g. `collHasMember`), general RELATIONS are not directly expressible in Java. The conceptual relations defined by SUMO are most of the times implementable through different kinds of program relations (redundancy) and program relations can be interpreted in different manners (diffusion). We summarize that:

- beside a few SUMO concepts that have correspondence in the Java ontology, all other SUMO concepts can only be encoded in Java;
- given a SUMO concept (or relation), there are more ways to map it on the Java ontology;
- the same (sequence of) program relation types can be used to implement distinct SUMO relations;
- in addition to the language constructs, the Java language provides "standard means" for implementing abstract concepts as part of its library; furthermore, basic programming idioms are many times used to implement more abstract concepts.

Regarding the implementation of individual concepts or relations, we identified the following issues:

- even if there is a good correspondence between SUMO PROCESSES and the Java `methods`, in order to manipulate the Java implementation of PROCESSES (e. g. to return a process as a result), we need to pack the `methods`, that implement the PROCESS, into `classes`;

- even if SUMO PROPERTIES can be implemented as Java `attributes`, there is no clear way to express the properties of PROCESSES;
- both PROPERTY and PART relations are implemented usually as attributes of classes and this leads to difficulties in their recovery;
- RESULTS of PROCESSES are implemented many times only as side-effects and this make their implementation difficult to recover;
- there is no clear manner to implement different kinds of participants in PROCESSES such as INSTRUMENT, PATIENT, or RESOURCE;
- complex relations between PROCESSES such as CAUSES or SUBPROCESS are not explicitly implementable (these relations are implemented only as method calls);
- the LESSTHAN and GREATERTHAN relations are encoded in the body of `compareTo` methods.

Remark. **1)** We performed the analysis of how can the real-world knowledge be implemented in programs only in a restricted manner. This restriction is because we base ourselves on two pre-defined conceptualizations: the SUMO ontology and our ontology of Java programming knowledge. Both of these ontologies represent only a description of the subject of matter (real-world and Java programs) from a particular point of view and thus are inherently incomplete. However, we claim that both of these ontologies are relevant since SUMO is an IEEE standard, and the Java ontology covers big parts of the knowledge about programs that is used in typical reverse engineering analyses.

2) We investigated the best possible implementations. We did not take into account different other constraints that can occur in the practical programs. For example, most of the times the implementation needs to be integrated in a framework, to interact with already existing programs, or to be performed by using existent means. These facts make the life of programmers much more complex than we described here.

6.6 Related work

Ontological evaluation of modeling languages. There are several works in the literature of information systems that concern the comparison of modeling grammars with respect to existent ontologies (Opdahl and Henderson-Sellers, 2002; Wand and Weber, 1993). By performing ontological analysis are detected different kinds of mismatches between the grammar and the ontology (Wand and Weber, 1993): *construct overload* – a modeling construct corresponds to several ontological concepts, *construct redundancy* – several modeling constructs represent the same ontological concept, *construct excess* – a modeling construct does not represent any ontological concept, *construct deficit* – an ontological concept is not represented by any modeling construct. As we presented in Section 2.2.2 (Figure), in order to identify these mismatches one needs to analyze two mappings (Wand and Weber, 1993):

1. *representation mapping:* from the ontology (semantic domain) to the modeling language. This mapping associates concepts from the ontology to the constructs of the modeling language. This mapping can be used to identify the construct redundancies or construct deficits.

2. *interpretation mapping:* from the modeling language to the reference ontology. This mapping associates constructs of the modeling language to the ontological concepts. By doing this, it can be used to identify modeling constructs that are non problem oriented (construct excess) or constructs that represent more concepts (construct overload).

The representation mapping is similar to our implementation strategies. However, we can notice the following differences: Firstly, the purpose of the above mentioned ontological analyses is to evaluate the constructs of conceptual modeling languages. Our purpose is to identify typical manners in which abstract real world concepts and relations (as defined by SUMO) are implemented in Java programs. Secondly, the conceptual gap between conceptual modeling languages (e. g. UML) and upper level ontologies is smaller than between Java and SUMO because the conceptual modeling languages provide many constructs (e. g. actors, events, states, transitions, relations or roles to name only a few) that are at a higher level of abstraction than the Java object-oriented constructs. Thirdly, we map our Java programming knowledge ontology to SUMO and the other works use Bunge-Wand-Weber ontology – e. g. (Opdahl and Henderson-Sellers, 2002).

6.7 Summary

In this section we investigated the similarities between the Standard Upper Merged Ontology and our ontology of basic Java programming knowledge. Based on these similarities we identified a set of mismatches that can occur when we try to implement SUMO concepts in Java programs: most SUMO concepts can only be encoded and are not directly implementable, a program relation can be interpreted as several SUMO relations and vice-versa. We considered SUMO to be an accurate description of the most general concepts of the real-world. Therefore the mismatches between SUMO and our Java ontology are likely to be perpetuated (and even accentuated) when implementing domain knowledge in programs.

Due to the big conceptual gap between the domain knowledge and general purpose programming languages constructs, many mismatches cannot be avoided.

Part IV

Automation

7 Identifiers Based Recovery of Intentions

Nomen est omen[1]

Latin proverb

Abstract: In this dissertation we define the meaning of programs through mappings between program elements and concepts from a domain ontology that they implement. When the domain is close to programming (e.g. design patterns, architecture) then domain concepts (e. g. the "Singleton" design pattern) can be identified in a program based only on the information captured in the language (e. g. the structure of the program). For example, numerous reverse engineering approaches deal with such problems: based exclusively on the structural information of a program they recover concepts from a (many times implicit) domain ontology about architectural knowledge. In our work we focus on the case when the domain is very different from programming and when the only hints about the mapping between a program and domain concepts are given by sources of information that are exterior to the language in which the program is written. In this chapter we focus our investigations on the use of informal knowledge contained in identifiers. We start by reviewing the roles of identifiers and program structure in program comprehension and the difficulties and pitfalls that can occur when using this information for understanding programs. We propose an approach to automatically detect concepts in the code by matching parts of the domain ontology with parts of the program based on the similarity between concepts and program elements names and between the conceptual and program relations. We develop a systematic approach for dealing with program elements names, extend our unified meta-model to take into consideration the names of identifiers and concepts, and present an algorithm for the location of concepts (and thereby recovering $\overleftrightarrow{Ref}$).

Structure of this chapter. After the introduction (Section 7.1), in Section 7.2 we review the role of identifiers and modularization in program understanding and the most important challenges in using them for concepts location. In Section 7.3 we present our view over programs as knowledge bases: the content of the knowledge bases is the identifiers and the knowledge representation language is the program structure. In Section 7.4 we present our unified meta-model that contains explicitly the concepts, program elements and the naming information. In Section 7.5 we present an algorithm for identification of concepts in programs using the similarities between the names of the concepts and the program entities. The chapter ends with an overview of the related work on concepts location (presented in Section 7.6) and a summary (presented in Section 7.7).

[1] Names contain a purpose

7.1 Introduction

Program identifiers are the most important source of information to link the code to its application domain. Even if they many times exhibit limitations (ambiguities and lack of meaningfulness), most of the times identifiers reflect the purpose of the program elements. In this chapter we use the identifiers in order to (semi-)automatically recover the intentional meaning by mapping programs to domain ontologies. In comparison to the existent automatic approaches that use identifiers in reverse engineering, we advance in two directions: 1) increase the accuracy of the mapping by detecting which program element refers to which concept, and 2) give semantic content to identifiers since they are part of ontologies.

7.2 Good Identifiers and Modularization Help Program Understanding

In this section we present a short literature survey on the benefits of good program identifiers and modularization for program understanding and on the issues that can occur due to the low quality of identifiers and of modularization.

7.2.1 The Role of Identifiers in Program Understanding

As we presented in the previous chapter, a large part of the domain concepts are not captured explicitly by the constructs of the languages used today in the development process. Most of the times, the domain concepts are only captured (hinted to) informally in the names of program entities. The central role of identifiers in program understanding was acknowledged by numerous works (Carter, 1982; Anquetil and Lethbridge, 1998b; Deissenboeck and Pizka, 2006). The names of identifiers bring an additional source of information to the programming language: the identifiers can be freely chosen and used to transmit knowledge that is not expressible in the language itself. Conventions on forming identifiers (for PL/I) were introduced almost 30 years ago:

> "First write down a list of all the English words that are likely to be used in the construction of identifiers. Even for large systems, this list numbers only in the hundreds, not thousands, of words [...] When making up an identifier, start with the English word or phrase that serves as a name. For each word in the name, pick the permitted form from the list, and connect the result with underscores." (Carter, 1982)

Not only good identifiers help program understanding, but scrambled identifiers (identifiers whose names are changed into meaningless strings of characters) are used as one of the basic methods of code obfuscation (Collberg et al., 1997).

Identifiers are the most important source of information that link programs to domains that are different from programming.

7.2.2 Issues with the Identifiers

Identifiers are only informal source of information, the names of program elements are chosen mostly in an ad-hoc manner or in bigger projects the naming conventions are only locally applied. Besides the typical naming conventions at the lexical level (e.g. Hungarian notation, Java naming conventions), there is remarkably few work that deals with the naming conventions at the conceptual level (a notable exception is (Deissenboeck and Pizka, 2006)). Identifiers exhibit a series of problems as presented below:

1. *Identifiers can be meaningless.* There are several studies in the literature of reverse engineering that report on the encountered identifiers which are meaningless with respect to the modeled domain. For example, Sneed (1996) reports about identifiers that represent names of football players or of programmers' girlfriends. Anquetil reports that even if the identifiers of Mosaic are meaningful in general, they encountered also strange situations like "mo_here_we_are_son" (Anquetil, 2001). Furthermore, many identifiers have only a single character and they do not reveal their intended meaning. However, even if the problems of meaningless names occurs very often, there is a general agreement that overall the identifiers are meaningful.

2. *Polysemy and synonymy.* Many times a name is used to refer to several concepts (synonymy) or a concept is referred in the code under different names (polysemy) (Deissenboeck and Pizka, 2006). The fact that people use a high variety of words to refer to the same things is recognized for a long time to be a central problem in human-computer communication and is known as the "vocabulary problem" (Furnas et al., 1987).

3. *Abbreviations and acronyms.* Many times the names of concepts are abbreviated in programs – instead of using the full name for referring to a concept (e. g. "length"), the programmers use short names (e. g. "len"). In the cases when both the abbreviated names and the full names are used, we have a special case of synonymy. Anquetil (2001) reports on the difficulties generated by dealing with acronyms – e. g. whether the acronym "IO" should be split or not.

4. *Compound identifiers.* At the code level, the identifiers that contain several words are named "compound identifiers" (e. g. "drawAndMoveColoredRectangle"). Splitting identifiers into basic words represents a problem in the analysis of identifiers (Anquetil and Lethbridge, 1998a; Feild et al., 2006). In order to deal with special cases, beside the general rules for splitting the compound identifiers, many times is used a list of "special strings" that do not comply to the general splitting rules. Furthermore, proper interpretation of the meaning of long compound identifiers might require the usage of natural language processing techniques (Deissenboeck and Pizka, 2006; Caprile and Tonella, 1999).

5. *Dealing with morphological variations of words.* Many times the same word occurs in a program in a variety of forms – e. g. infinitives, gerund, past tense or plurals. Recognizing that two strings represent the same word but in various inflexions is a challenging problem. Many times this is solved by approximating all identifiers to their basic form by using a words stemming algorithm (Anquetil, 2001) or by using a pre-defined dictionary.

6. *Compound words.* The natural language does not contain enough words for describing all the possible situations and only the most common concepts are lexicalized. The concepts that are not common enough to be lexicalized or variations of concepts are described through sequences of words, named "compound words" – e.g. "Gregorian Calendar" (Deissenboeck and Pizka, 2006). The identification and interpretation of compound words is another problem in dealing with identifiers.

7. *Special words.* Many times the words contained in identifiers do not represent common natural language words (that we might find for example in dictionaries). Many times identifiers refer to names of standards or other technical acronyms – e. g. "X059" is the name of a security standard. It is obvious that splitting the compound identifiers that contain these words require the usage of explicit knowledge about special names.

7.2.3 The Role of Modularization in Program Understanding

Beside the identifiers, other important factors that affect the understanding of programs are the modularization and structuring mechanisms. Empirical experiments made on the influence of modularization on program understanding show that modularization improves the comprehensibility. Pennington (1987) suggests the fact that programmers' mental representation of programs is procedural (structural) rather than functional and emphasize on the importance of the program structure for understanding.

Woodfield et al. (1981) represent an early work based on empirical studies of the influence of modularization and comments in comprehending programs. The authors conclude that the "abstract data type modularization" improved the comprehension even if the more modular program had a bigger size than the non-modular one.

Bastani and Iyengar (1987) draw the conclusion that the efforts for comprehending a program increase with the opaqueness between the abstract data structures (e. g. domain concepts) and the physical data structures used to implement them.

7.2.4 Issues with the Modularization

Even if the modularization mechanisms are defined in the programming language, the practice of software development shows the following problems:

1. *Dominant decomposition.* The programs are modularized along a single direction. This leads to the well-known problem of the "tyranny of dominant decomposition" (Tarr et al., 1999) that represents the inability of programmers to implement in a modular manner certain application concerns that do not fit in the current decomposition of the software in modules.

2. *Lack of modularization.* Many program parts lack an adequate modularization and big portions of the code belong to a single module. Furthermore, different programming languages provide different modularization capabilities: at one spectrum are the assembly languages that offer no mechanism for modularization; at the other spectrum are object-oriented languages that offer several modularization mechanisms.

Observation: From the point of view of the explicitness in the implementation of concepts in the code, "lack of modularization" is similar to "meaningless names". In an extreme case the program can be obfuscated by deleting all names and compressing the program structure (e.g. convert local variables into global ones, inline methods) (Collberg et al., 1997). In these cases our abstraction of programs would not function anymore.

> The automation in the recovery of intentions can be achieved if the program modularization follows (agrees to, is in concordance with) the modularization of its domain (as modeled by the ontology) and the names of program elements reflect the domain concepts that they refer to.

7.3 Programs are Knowledge Bases

In a broader sense, programming is a process of representing the domain knowledge in the world of computers. Within this perspective, programming languages can be viewed as knowledge representation languages. These observations bring us to regard *programs as knowledge bases*:

program knowledge base

- the *content of the knowledge base* is the names of program identifiers, and
- the *knowledge representation language* is a subset of the programming language.

Furthermore, by combining the program structure with the names of program elements, we regard *programs as semi-structured data*:

- the structure among identifiers is given by the relations between their corresponding program elements, but
- inside compound identifiers there is no structure since we consider that compound identifiers are flat lists of words.

Example 7.1: Example of programs seen as semi-structured data

In Figure 7.1 we present two examples of programs that implement the same domain concepts. In the first case, (Figure 7.1 top) there is obviously less structure in the implementations of concepts DRAW and OVAL. Since "drawOval" is a compound identifier there is no structuring between the words "draw" and "oval". The only structuring is between the program elements `Graphics` and `drawOval` (e.g. `Graphics` – hasMeth – `drawOval`) and `drawOval` and its parameters (e. g. `drawOval` – hasParam – `x`). In the second case, (Figure 7.1 down) we present a similar program, but better structured. In this case each program element refers to a lexicalized concept and the relations among concepts implementation are explicit: `Graphics` – hasMeth – `draw` – hasParam – `oval`).

□

> The modularization of a program induces structure in the corpus of identifiers.

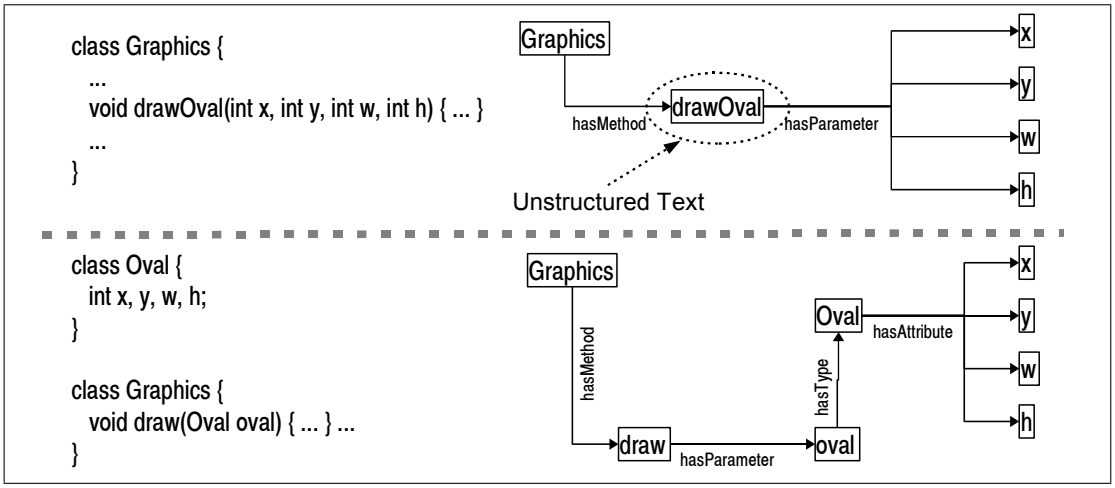


Figure 7.1: Programs as semi-structured data

7.4 A Unified Meta-model of Concepts, Names and Program Entities

In order to make the role of identifiers as links between the program elements and domain concepts explicit, we extend our unified metamodel (presented in Figure 3.13) to take into account the names of program entities, the names of concepts and the words they are composed of. In Figure 7.2 we present the extended meta-model that explicitly bridges the domain concepts to the code via identifiers (Deissenboeck and Ratiu, 2006). In order to describe the relations between the domain concepts, the names and the program elements we add to the meta-model from Figure 3.13 the *lexical layer*. The new layer explicitly describes the relation between program elements, their identifiers, the words that compose these identifiers on the one hand, and between the domain concepts, their names and the words that make up the concept names on the other hand. The words represent the glues that link the identifiers and the names of domain concepts. In Figure 7.2 we also show the functions that make the links between elements of these layers (e. g. P_2I links program elements to their corresponding identifiers).

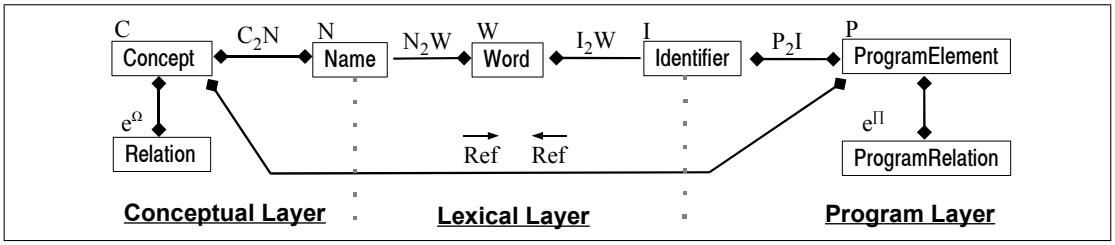


Figure 7.2: A unified meta-model for representing programs (right), domain ontologies (left) and naming information (center) (Deissenboeck and Ratiu, 2006)

The formalization of the program and conceptual layers was presented in Section 3.4.2 and Section 3.4.3. In the next section we present the formalization of the lexical layer.

7.4.1 The Lexical Layer

We describe the lexical layer as a tuple consisting of three sets: the set of program identifiers, the set of concept names and the set of words obtained by the reunion of both the words that make up identifiers and the ones that make up concept names.

Remark. In Definition 3.4.2 (on page 83) we formally defined a concept $c \in C$ to be a pair $(Names, Gloss)$, where $Names$ is the set of the concept names and $Gloss$ is a glossary entry that describes the concept.

Definition 7.4.1 (Lexical layer)**:** *Let* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be a program abstraction and* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *a domain ontology. The* lexical layer Λ *is:* *lexical layer*

$$\Lambda = (N, W, I) \tag{7.1}$$

where,

- N *is the set of elements representing the reunion of the names of concepts from* C *as defined below*

$$N = \bigcup_{c \in C} c.Names,$$

- I *is the set of elements representing the identifiers of program elements* P*,*
- W *is the set of morphologically normalized words obtained by splitting the elements of* N *and* I *into words.*

In Figure 7.3 (middle) we present an example of the lexical layer.

Definition 7.4.2 (Name-to-Words)**:** *Given the lexical layer* $\Lambda = (N, W, I)$*, we define the function* name-to-words *name-to-words*

$$N_2W : N \rightarrow W^*$$

that given a name returns the sequence of normalized words that compose it.

Definition 7.4.3 (Identifier-to-Words)**:** *Given the lexical layer* $\Lambda = (N, W, I)$*, we define the function* identifier-to-words *identifier-to-words*

$$I_2W : I \rightarrow W^*$$

that given an identifier computes the set of words that compose the identifier.

Computing the functions I_2W, N_2W

In order to compute N_2W or I_2W we need to perform two basic operations: 1) obtain the set of words from a name or an identifier, and 2) perform a morphological normalization of the words.

1) Splitting the identifiers. Identifiers that contain more words are named compound. They either denote stand-alone concepts that are formed of more words (e. g. Gregorian Calendar) or combinations of concepts (each concept being represented through a (compound-)word). The first step in assigning meaning to identifiers is to split them into words. The most common used techniques for splitting identifiers into words are based on CamelCase or the usage of special *words markers* such as underscores (see the example below). Various techniques for splitting identifiers into words are presented in the reverse engineering literature. Caprile and Tonella (1999) propose a technique for identifiers splitting that is based on a dictionary of already identified words. Anquetil and Lethbridge (1998a) propose a method for splitting the names of source code files that do not contain word markers based on two steps: 1) identify the abbreviations in the system, 2) split long names into words by making use of the abbreviations list. Feild et al. (2006) present a comparison of several techniques for splitting identifiers based on greedy algorithms for splitting and on neural networks.

word markers

Example 7.2: Example of identifiers splitting

'GregorianCalendar' $\mapsto$ {'Gregorian', 'Calendar'} 'gregorianCalendar' $\mapsto$ {'Gregorian', 'Calendar'}
'gregoriancalendar' $\mapsto$ {'gregoriancalendar'} 'gregorian_calendar' $\mapsto$ {'Gregorian', 'Calendar'}

□

We assume that the identifiers have word markers and that we can use these markers to split identifiers into words.

2) Morphological normalization of words. Many times the same word occurs in a program in various morphological forms – e. g. "house", "houses", "open", "opened", "opening". In these cases the string comparison simply is not sufficient for determining whether two strings refer the same word (possibly in two different forms). In order to enable the comparison of strings we need to morphologically normalize them by applying one of the following methods:

1. Use a stemming algorithm that given a word strips its suffix in order to obtain a normalized word (Porter, 1997); or
2. Use a dictionary that contains the word variations and their normalized form. The advantage of a dictionary, is that it can resolve more inflections of words than pure stemming – e. g. the WordNet normalized form of the words {"are", "was", "were"} is "be", and this cannot be obtained through stemming.

We use the WordNet dictionary in order to realize the normalization of words.

Example 7.3: Example of words normalization

'houses' $\mapsto$ 'house' 'moved' $\mapsto$ 'move' 'is' $\mapsto$ 'be'
'children' $\mapsto$ 'child' 'made' $\mapsto$ 'make' 'were' $\mapsto$ 'be'

□

Example 7.4: Example of the complete formalization of a program

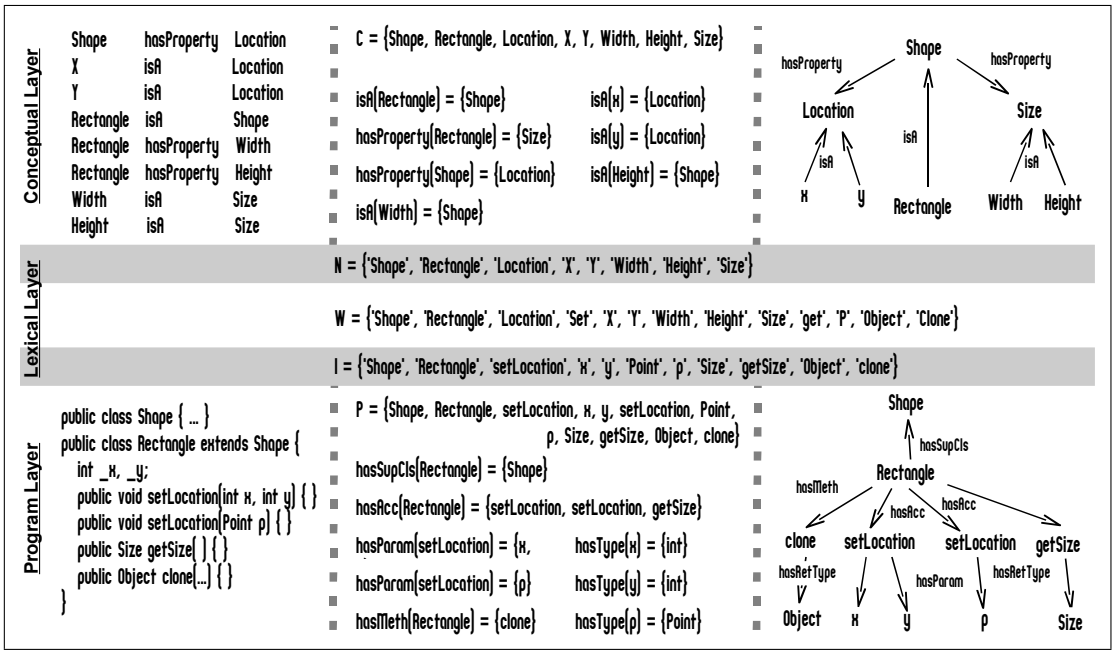


Figure 7.3: The three layers of a program

In Figure 7.3 we present an example of the program layers (the Program Layer containing only the API program elements). In the middle of this figure is presented the lexical layer: it contains the names of concepts (e. g. 'Shape', 'Location', 'Width'), the names of program elements (e. g. 'Shape', 'setLocation') and the set of normalized words obtained by the reunion of words contained in the name and in identifiers. The other layers were presented in Section 3.4.2 and Section 3.4.3.

□

7.4.2 Bridging the Layers

To obtain the names of concepts and program elements we define two functions: program elements-to-identifier (P_2I) gives the associated identifier to a program element and concept-to-name (C_2N) returns the set of names (synonyms) of a concept.

program element-to-identifier

Definition 7.4.4 (Program element-to-Identifier)**:** *Let P be a set of program elements and I a set of identifiers. We define the function P_2I (*program element-to-identifier*)*

$$P_2I : P \rightarrow I$$

to associate a program element to its corresponding identifier.

concept-to-names

Definition 7.4.5 (Concept-to-Names)**:** *Let C be a set of concepts and N their names. We define the function C_2N(*concept-to-names*)*

$$C_2N : C \rightarrow \wp(N)$$

to associate a concept to its corresponding set of names.

7.5 Automatic Identification of Concepts in the Code

In the following we present an algorithm for identification of concepts in programs based on the similarities between the concepts and program elements names on the one hand, and on the similarities between the structure of the ontology and of the program on the other hand.

7.5.1 Making Use of Naming Clues

In the following we define two functions that use the similarities between the identifiers and concept names as clues to bridge the program and conceptual layers.

candidate names

Definition 7.5.1 (Candidate names)**:** *Let I be a set of identifiers, $i \in I$ an identifier, and N a set of concept names. We define the function* candidate names *to be:*

$$CandNms : I \rightarrow \wp(N),$$
$$CandNms(i) = \{n \in N \,|\, N_2W(n) \sqsubseteq I_2W(i)\}$$

Intuitively, a concept name $n \in N$ is referred by an identifier $i \in I$ iff the sequence of words that make up the name of n is a subsequence of the sequence of words obtained from splitting the identifier i. By working with (sub)sequences and not with (sub)sets of words, we take into consideration the ordering among words. Based on the $CandNms$ we define below the candidate concepts function.

candidate concepts

Definition 7.5.2 (Candidate concepts)**:** *Let C be a set of concepts, P a set of program elements, and $p \in P$ a program element. We define the function* candidate concepts *to be:*

$$CandCts : P \rightarrow \wp(C),$$
$$CandCts(p) = \{c \,|\, c \in C \wedge i = P_2I(p) \wedge C_2N(c) \cap CandNms(i) \neq \emptyset\}$$

Intuitively, given a program element the function $CandCts$ computes, based on its name, the possible concepts that the program element refers to.

Example 7.5: Example of $CandNms$ and $CandCts$

In Figure 7.4 we present a configuration of the program, lexical and conceptual layers and below examples of CandNms and CandCts. We remark that according to our definitions of $CandCts$ and I_2W, the identifier 'getmonth' cannot be interpreted since it is not in CamelCase.

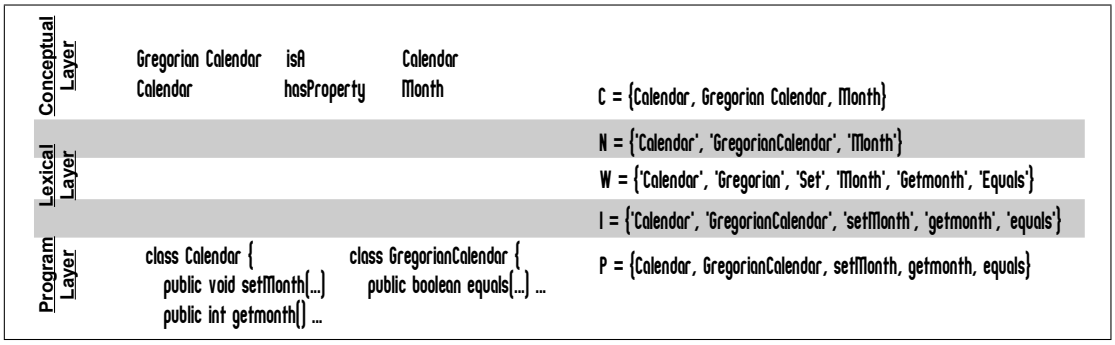


Figure 7.4: The names are used as clues for bridging the program and conceptual layers

CandNms(Calendar) = {'Calendar'}, CandNms(setMonth) = {'Month'},
CandNms(getmonth) = $\emptyset$, CandNms(equals) = $\emptyset$,
CandNms(GregorianCalendar) = {'Gregorian Calendar', 'Calendar'}

CandCts(Calendar) = {Calendar}, CandCts(setMonth) = {Month},
CandCts(getmonth) = $\emptyset$, CandCts(equals) = $\emptyset$,
CandCts(GregorianCalendar) = {Gregorian Calendar, Calendar}

□

Remark. While the functions $\overleftrightarrow{Ref}$ represent the real (exact) mapping between the concepts and the program elements that refer them, the function $CandCts$ represent an approximate mapping due to their similar names. The mapping provided by $CandCts$ is inexact because of the naming ambiguities (synonymy, polysemy), or is undefined (when identifiers are meaningless).

7.5.2 Making Use of Structural Similarities

In Section 3.6 we defined the functions $\overleftrightarrow{t}$ that map conceptual level relations on similar program relations. If the structure of the program is similar to the structure of the groups of concepts that it implements then the functions $\overleftrightarrow{t}$ would be enough. However, as we showed in Section 4.2 and 5.2, programs exhibit abstraction (i. e. leave out domain concepts) and implementation details (i. e. program elements that do not refer to any domain concept). In these cases, the function $\overleftrightarrow{t}$ needs to be extended as shown in the next example.

Example 7.6: Example of cases when simple relations mapping is insufficient

In Figure 7.5 we present examples of mappings between sequences of relations from our conceptual layer and sequences of relations from the program. In (a) the property of a concept is implemented as an attribute of the class that implements the concept; in (b) the property of

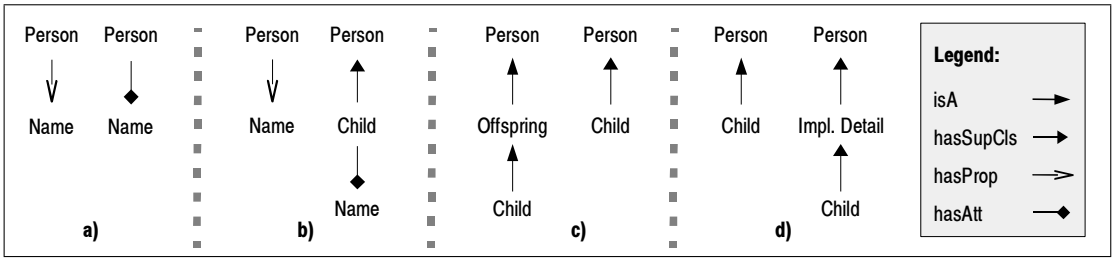


Figure 7.5: Implementation details and abstraction influence the mapping of relations

a concept is implemented as an attribute of the subclass that implements the concept; in (c) the concept in the middle of a isA relation sequence is left out; and in d) instead of implementing the isA relation directly we have an implementation detail.

□

In order to take into consideration these issues generated by abstraction and implementation details (i. e. *structural mismatches* between the program and the ontology), we define an extended version of the relations mapping function by defining mapping between sequences of conceptual and program relations.

Definition 7.5.3 (Extended implementation and interpretation of relations)**:** *Let* $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$ *be a conceptual layer and* $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$ *be the program layer. We define the* extended implementation of relations *to be the function*

extended implementation of relations

$$\overleftarrow{t_e} : \Sigma^{\Omega *} \rightarrow \wp(\Sigma^{\Pi *})$$

that maps a sequence of relation types from the ontology to the corresponding set of sequences of relation types from the program graph. The extended interpretation of relations *is a function*

extended interpretation of relations

$$\overrightarrow{t_e} : \Sigma^{\Pi *} \rightarrow \wp(\Sigma^{\Omega *})$$

that maps the sequence of relation types from the program graph to a corresponding set of sequences of relation types from the ontology. The duality between these functions is expressed through the following equation:

$$\overrightarrow{t_e}(\langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle) = \{\langle \sigma_k^{\Omega} ... \sigma_l^{\Omega} \rangle \in \Sigma^{\Omega *} \mid \langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle \in \overleftarrow{t_e}(\langle \sigma_k^{\Omega} ... \sigma_l^{\Omega} \rangle)\}$$

When we refer to both of these two functions we use the following notation: $\overleftrightarrow{t_e}$.

Example 7.7: Example of $\overleftarrow{t_e}$

In the following we present several instances of the $\overleftarrow{t_e}$ function for the example from Figure 7.5 ($-hasSupCls$ is the inverse of the $hasSupCls$ relation).

$$\begin{aligned}
\overleftarrow{t_e}(\langle hasProp \rangle) &= \{\langle hasAtt \rangle,\ \langle -hasSupCls, hasAtt \rangle\} \\
\overleftarrow{t_e}(\langle isA, isA \rangle) &= \{\langle hasSupCls \rangle\} \\
\overleftarrow{t_e}(\langle isA \rangle) &= \{\langle hasSupCls, hasSupCls \rangle\}
\end{aligned}$$

□

7.5.3 Concept Location Algorithm

In the following we present an algorithm that recovers the references of concepts in the code ($\overleftrightarrow{Ref}$). In order to overcome the naming ambiguities, our algorithm is based on matching the program and the ontology graphs. Below we present our algorithm in pseudo-code and in Figure 7.7 we present an example of how this algorithm works.

1. **for-each** $c \in C$ **do**
2. reflection$(c) = \{p \in P \mid c \in CandCts(p)\}$
3. **for-each** $\langle \sigma_i^\Omega ... \sigma_j^\Omega \rangle \in {\Sigma^{\Omega_e}}^*$ **with** $\overleftarrow{t_e}(\langle \sigma_i^\Omega ... \sigma_j^\Omega \rangle) \neq \emptyset$
4. neighbours$(c, \langle \sigma_i^\Omega ... \sigma_j^\Omega \rangle) = \{c' \in C \mid c' \in \langle \sigma_i^\Omega ... \sigma_j^\Omega \rangle(c)\}$
5. **for-each** $c' \in$ neighbours$(c, \langle \sigma_i^\Omega ... \sigma_j^\Omega \rangle)$
6. reflection$(c') = \{p' \in P \mid c' \in CandCts(p')\}$
7. **for-each** $p \in$ reflection(c), $p' \in$ reflection(c')
8. **check-if** $(\exists \langle \sigma_k^\Pi ... \sigma_l^\Pi \rangle \in \overleftarrow{t_e}(\langle \sigma_i^\Omega ... \sigma_j^\Omega \rangle)$ **with** $p' \in \langle \sigma_k^\Pi ... \sigma_l^\Pi \rangle(p))$
9. **if-yes** (found mapping $(c, p), (c', p')$)

Figure 7.6: Concepts location algorithm

Intuitively, the mapping between concepts and program elements is done by using a graphs matching algorithm that uses two operations:

1. maps program elements with their candidate concepts based on the names similarity given by $CandCts$ (lines 2, 6), and

2. maps sequences of edges from the program graph to equivalent sequences of edges from the concepts graph given by $\overleftarrow{t}$ (lines 4, 8).

Discussion: Below we make several remarks with respect to our algorithm.

1) Our algorithm for identification of concepts in programs is similar with algorithms for finding subgraph isomorphism. Even if the latter algorithms are known to be NP-hard, due to the fact that our mapping of paths is based on the functions CandCts and $\overleftarrow{t_e}$, that restrict the possible mapping space very much in practical situations, this algorithm does not pose any serious computational problems.

2) At each step the algorithm identifies a unit of knowledge (knowledge quark), that is expressed as a triple: concept – relation – concept, in the code. This allows us to avoid some of the mistaken identifications of concepts due to polysemy. However, as we present in Section 10.4, by mapping only a triple at a time we still have a significant amount of false positives (noise).

3) An important part of this algorithm is based on matching the identifiers names to the concepts names. Thus, our algorithm cannot map concepts to program elements if the names of these program elements do not offer strong enough clues that they implement the concepts. Moreover, we can recover only those implementations where the sequence of relations from the domain ontology is reflected in the code. Say it with other words, we recover the functions $\overleftrightarrow{Ref}$ only partially.

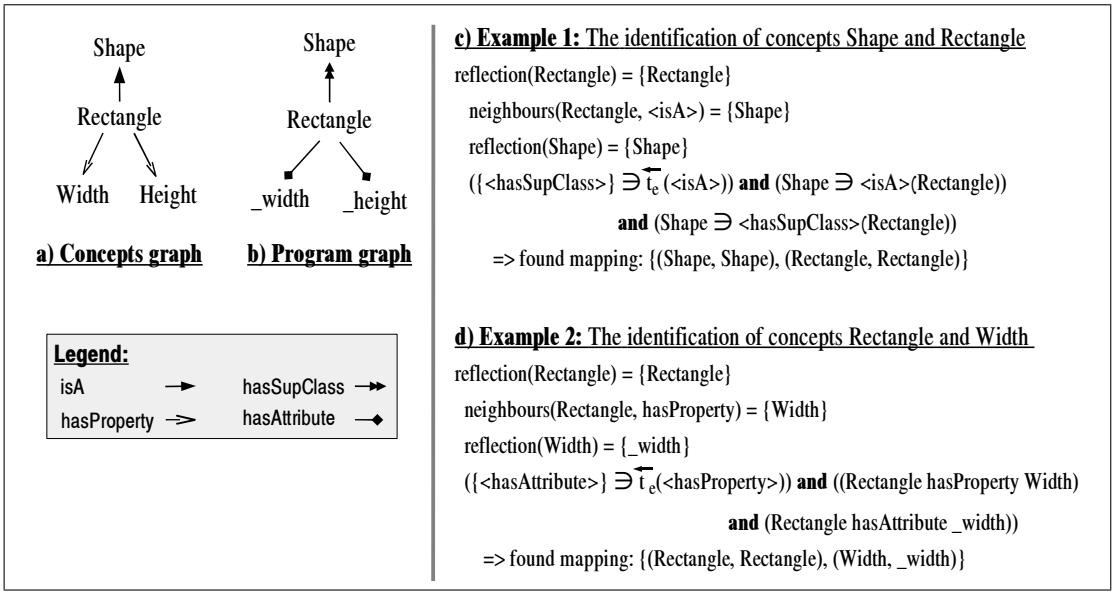


Figure 7.7: Concepts location algorithm examples

4) Once the mapping is established and concepts are identified in the code we can analyze the quality of identifiers and we identify naming problems such as polysemy and synonymy (Chapter 8). For the cases when the names have a good quality (and indeed reflect the link between the program entities and the domain concepts) the algorithm enables automatic investigation of how is the domain knowledge reflected in the code.

5) In Section 10.4 we present our experience with running the concepts location algorithm on several Java systems. The most important conclusions of our experiments are that the algorithm is powerful enough to recover $\overleftrightarrow{Ref}$ for a large number of program elements. However, due to the fact that in each step the algorithm matches only pairs of related concepts with pairs of related program elements, the algorithm can deliver a considerable number of false positives (i. e. program elements that were assigned to concepts by mistake).

7.5.4 Discussion on Automating the Concept Location Algorithm

In the following we examine our algorithm and discuss every line from Figure 7.6 with respect to the degree of automation:

Line 1: from the very first line we remark that in order to apply the algorithm we need to know the set C of concepts which are searched for in the code. Where we get these concepts from is the main subject of Chapter 9.

Lines 2, 6: as we already discussed, the $CandCts$ can be computed automatically based on the similarities between the names. The extraction of program layer and the slicing of identifiers for obtaining the CandNms, are topics already studied in the reverse engineering literature (Caprile and Tonella, 1999; Feild et al., 2006).

Lines 3, 4, 7: we need a concrete set of relations between concepts (Σ^{Ω}), a concrete set of program relations (Σ^{Π}) and an appropriate instance of the relations mapping function ($\overleftarrow{t_e}$). An approach for obtaining these ingredients (Σ^{Ω}, Σ^{Π}, $\overleftarrow{t_e}$) by performing ontological analysis was

presented in Chapter 6. In the experiments from Chapter 10 we will use only Σ^{Π} and Σ^{Ω} that we defined in Sections 3.4.2 and 3.4.3.

7.6 Related Work

Modeling identifiers and concept names. There are many papers in the reverse engineering literature that deal with the analysis of identifiers (Antoniol et al., 2007; Deissenboeck and Pizka, 2006; Lawrie et al., 2007). The modeling degree of identifiers varies between flat strings of characters and up to sequences of words obtained by different strategies for splitting compound identifiers into words and for morphologically normalizing the words. Most of the works (except (Deissenboeck and Pizka, 2006)) do not treat explicitly the differences between program identifiers and names of concepts.

We advance in this direction by defining and formalizing the lexical layer that makes a distinction between the names of concepts and program identifiers; compound words and compound identifiers. Furthermore, we define formally the relations between program identifiers and names (CandNms), the relation between names and the words they are composed of (N_2W), and between the identifiers and the words they are composed of (I_2W).

Concept location and assignment. According to Biggerstaff et al. (1994) two tasks are required to assign concepts to programs:

> "1. Identify which entities and relations out of the often overwhelming numbers in a large program are really important; 2. Assign them to known (or newly discovered) domain concepts and relations."

Biggerstaff describes a set of features that facilitate the assignment of human-oriented concepts to program parts:

> "1. natural language token meanings; 2. occurrences of closely related concepts"

The vision of Biggerstaff from early '90s on concepts assignment is very close to our concepts location approach described in this chapter. All major ingredients of our approach – 1) intentional program abstraction (i. e. program elements and relations), 2) the conceptual layer (i. e. concepts and relations) and 3) location of concepts based on the similarities between program elements and concepts names – represent practically only an operationalization of the vision of Biggerstaff.

Latent Semantic Indexing. One of the most widely-used approaches for using the information carried by identifiers is "Latent Semantic Indexing" (LSI) (Maletic and Marcus, 2001). LSI is an information retrieval technique based on computing clusters of words that recurrently occur together in different program parts. These clusters of co-occurring words form high-level concepts and thereby have a "semantic content". LSI was used by Marcus (2003) to link source code to documentation, to identify abstract data-types in the code or to identify high-level concept clones in the software. In most of the LSI-based concept location approaches, the clusters of

identifiers remain uninterpreted and thereby their meaning is not specified at all. There is however one exception: in the case when source code is mapped on documentation we can consider that the documentation represents an interpretation of the clusters – the meaning of the clusters is given by the natural language prose that is part of the documentation where the clusters were mapped to. However, this natural language interpretation is weak and lacks structure.

In our work, we use light-weighted domain ontologies to represent the meaning of domain the concepts. Furthermore, our graph-based algorithm aims to a higher precision in mapping the concepts to the code and thereby we can assign better defined concepts to individual program elements (and not to entire program parts).

Design patterns identification. In the case when the semantic domain describes design patterns, our approach for locating concepts is quite similar to the identification of design patterns. There is a large amount of work in the reverse engineering community that aimed at recovering design patterns from the code by matching program structure with a representation of design patterns collected in a knowledge base (and representing the semantic domain). At a certain abstraction level, all of these works are similar to our concepts location approach. Albin-Amiot and Guéhéneuc (2001) propose an approach to build a knowledge base of design patterns. The identification of design patterns in programs is done by mapping the code-model to the model defining design patterns. Tsantalis and Halkidis (2006) also use a graph matching algorithm for detecting design patterns. They formalize the design patterns as graphs, abstract programs as graphs and subsequently identify design patterns by matching the two graphs. The occurrences of design patterns are identified based on the similarities between the structures of the graphs. Compared to the works on design patterns detection our approach aims at locating (arbitrary) domain concepts.

7.7 Summary

Identifiers are the most important source of information for recovering the intentional meaning of programs. In this chapter we proposed an approach to automatic recovery of intentional meaning by using the similarities between the names and structure of concepts on the one hand and the program elements names and the program structure on the other hand. Our approach for locating concepts differ from the ones already existent in the literature along the following lines:

- *automation* – our approach is fully automatic once we have a domain ontology,
- *accuracy* – characterized along two directions: 1) the concepts are parts of domain ontologies and thereby have a well defined meaning, and 2) we map individual program elements to concepts,
- *domain of the applicability* – our approach can be used to locate concepts from a wide variety of domains (ranging from business domains up to domains closer to programming).

8 Characterizing the Reflexion of Concept Names in Program Identifiers

> [...] programs must be written for people to read, and only incidentally for machines to execute.
>
> Abelson and Sussman[1]

Abstract: The importance of good program elements names is widely acknowledged in program comprehension and reverse engineering communities. The similarities between the names of program entities and domain concepts are the most important hints that help in identification of non-technical concepts in the code. In the previous chapter we used the similarities between identifiers parts and domain concepts names to automate the location of concepts in the code. We assumed that the names of program elements are good enough in order to allow the location of concepts in the code. However, in practice the identifiers are far from being perfect and the quality of names influences the automatic location of concepts. In this chapter we use our framework to formally characterize the quality of names in terms of how good they reflect the domain concepts. By making the distinction between the functions $CandCts$ (apparent concepts reference based only on names similarity) and $\overleftrightarrow{Ref}$ (concepts actually referenced) we can formally describe naming anomalies along two directions:

1. *meaningfulness of identifiers* characterizes the measure in which identifiers reflect the concepts referred by their program element, and
2. *naming ambiguities* characterize the synonymy and polysemy caused by non-consistent naming.

Structure of this chapter. In the introduction (Section 8.1) we present a general discussion on the quality of names and what it means for a program element name to be ideal. In Section 8.2 we discuss and formalize the meaningfulness of identifiers. In Section 8.3 we present and formalize the naming ambiguities (i. e. synonymy and polysemy). We end this short chapter by presenting the related work in describing the quality of identifiers (Section 8.4) and with a summary (Section 8.5).

[1] Preface to the first edition of "Structure and Interpretation of Computer Programs" (Abelson and Sussman, 1996)

8.1 Introduction

As the quote at the beginning of this chapter says, programs must be written like prose in order to be easily readable by humans. One of the features of programs that make them readable is the quality of the identifiers. There are several works in the literature that characterize the quality of names in terms of informal rules (Carter, 1982; Anquetil and Lethbridge, 1998b) or by formally expressing conciseness and consistency rules (Deissenboeck and Pizka, 2006). Since the similarities of program elements and concepts names represent our most important source of information for automatically extraction of domain concepts in programs, we will characterize in the following the quality of names by using our framework.

Remark. All the mismatches formalized in this chapter are given by considering our intentional program abstraction $\Pi = (P,\ \Sigma^{\Pi},\ e^{\Pi})$, a domain ontology $\Omega = (C,\ \Sigma^{\Omega},\ e^{\Omega})$, and the lexical layer $\Lambda = (N, W, I)$.

ideal name

Definition 8.1.1: *A program element $p \in P$ has an* ideal name *iff:*

$$\overrightarrow{Ref}(p) = CandCts(p) \tag{8.1}$$

Intuitively, all the concepts that a program element refers to are reflected as parts of its name. Real-world programs contain many times identifiers whose names offer only partial clues about the implemented concepts or exhibit different inconsistencies. In the following two sections we characterize naming defects by describing the *meaningfulness* of names and *ambiguities* introduced by naming. We discuss the influence of different naming defects on program comprehension in general and on the usability of APIs in particular.

8.2 Meaningfulness of Identifiers

Many empirical studies on the nature of identifiers in industrial systems report the fact that some program identifiers are "meaningless" (Sneed, 1996; Anquetil, 2001). In the following we formally characterize the different cases of meaningfulness of identifiers.

non-suggestive name

Definition 8.2.1 (Non-suggestive name)**:** *A program element $p \in P$ has a* non-suggestive name *iff:*

$$\exists c \in C.\ c \in \overrightarrow{Ref}(p)\ \wedge\ c \notin CandCts(p)$$

Intuitively, at least one of the concepts that are referenced is not explicitly reflected in the program element name. In this case, the concepts that the program element refers to can be identified only by reading the code or the documentation. In the case of APIs non-suggestive names affect the usability of their interface negatively.

Example 8.1: Example of non-suggestive naming

The name of the method `getActualMaximum` of the class `java.util.Calendar` represents an example of non-suggestive naming: we can recognize the concept MAXIMUM but not the

other concepts it refers to. By investigating the Javadoc corresponding to this method we find out that[2]:

```
/**
  * Returns the maximum value that the specified calendar field could have,
  * given the time value of this Calendar. For example, the actual maximum
  * value of the MONTH field is 12 in some years, and 13 in other years in
  * the Hebrew calendar system. [...]
```

After reading the documentation we can propose a better name like for example "getMaximumFieldValue". In order to use this method, one needs to have a deep knowledge of the implementation details of the CALENDAR concept in the Java library.

□

Definition 8.2.2 (Clueless name)**:** *A program element $p \in P$ exhibits* clueless naming *iff:* *clueless naming*

$$\overrightarrow{Ref}(p) \neq \emptyset \ \wedge \ CandCts(p) = \emptyset$$

Intuitively, a program element refers to some domain concepts but its name does not give any clue about any of these concepts. When the names of all program elements are clueless, then we have a typical case of code obfuscation at the identifiers level. If the program element with clueless naming belongs to the public interface of the library, the only way to use this element is to inspect other sources of information beside its name - e. g. its documentation, its implementation, or usages of the program element in the code.

Example 8.2: Example of clueless naming

The method `java.awt.Rectangle.outcode(double x, double y)`, that for a given point determines its relative position with respect to the rectangle, has a clueless name.

□

Definition 8.2.3 (Misleading name)**:** *A program element $p \in P$ has* misleading name *iff:* *misleading name*

$$c \notin \overrightarrow{Ref}(p) \wedge c \in CandCts(p)$$

Intuitively, a program element has a misleading name if the name refers to concepts that are not referred by the program element. This represents a worse sub-case of code obfuscation since one can easily misuse this program element.

Example 8.3: Example of misleading name

An example of misleading name is the parameter `date` of the method `Calendar.set(int year, int month, int date)` from the Java API. Here, the name "date" is used to denote the concept DAY OF THE MONTH instead of a particular point in time as we would expect.

□

[2]Java API – `http://java.sun.com/javase/6/docs/api/java/util/Calendar.html`

8.3 Ambiguity of Identifiers

Another category of naming defects is represented by naming ambiguities. In these cases the concepts referenced by the program elements are reflected in their names but the usage of names for denoting concepts is inconsistent: one name is used for referring to more concepts or one concept is denoted by different names.

synonymy

Definition 8.3.1 (Synonymy)**:** *The names of the program elements p_1 and p_2 exhibit* synonymy *iff:*

$$\exists c \in C.\ \{p_1, p_2\} \subset \overleftarrow{Ref}(c)\ \wedge\ n_1 \in C_2N(c)\ \wedge\ n_2 \in C_2N(c)\ \wedge\ n_1 \neq n_2\ \wedge$$
$$n_1 \in CandNms(P_2I(p_1))\ \wedge\ n_2 \in CandNms(P_2I(p_2))$$

Intuitively, a concept is referred through two names within two program elements. When a concept from the real world is implemented in a program, the programmers should refer to that concept by using only one name. By avoiding synonymy, the homogeneity is increased and this facilitates the location of concepts.

Example 8.4: Example of synonymy

Below we present an example of synonymy from the Java AWT graphical library. The concept ELLIPSE is implemented in this library by using two names: 'ellipse' and 'oval'. As we can see in the code fragment below, this concept is implemented in a hierarchy of shapes through the class `Ellipse2D` and as drawing primitive through the method `drawOval()` of `Graphics`. When we look at the documentation of `drawOval()` we see that indeed this method refers to ELLIPSE. Furthermore, a closer inspection of these implementations reveals the fact that they are similar since they represent an ellipse by using the upper-left corner, the height and the width of the rectangle in which the ellipse is contained. This leads us to the conclusion that the ELLIPSE concept is defined using different names in two relatively close parts of the graphical library and at two levels of abstraction: firstly it is introduced as a "first class citizen" through a class, and secondly as a primitive method used in a drawing utility class[3].

```
1.  public abstract class Ellipse2D extends RectangularShape { ... }
2.
3.  public abstract class Graphics { ...
4.      public abstract void drawOval(int x, int y, int width, int height); ...
5.  }
```

□

polysemy

Definition 8.3.2 (Polysemy)**:** *The names of the program elements p_1 and p_2 exhibit* polysemy *iff:*

$$n \in C_2N(c_1) \wedge n \in C_2N(c_2) \wedge n \in CandNms(P_2I(p_1)) \wedge$$
$$n \in CandNms(P_2I(p_2)) \wedge p_1 \in \overleftarrow{Ref}(c_1) \wedge p_2 \in \overleftarrow{Ref}(c_2)$$

[3]More recent versions of the AWT library contain the class `Graphics2D` that solves this problem by offering the method `draw(Shape)`. However, as the original class was not removed the synonymy defect still exists.

Intuitively, two distinct concepts are referred through a single name by two program elements. When two distinct concepts are implemented in a program the names of their corresponding program entities should reflect the difference. The problems generated by polysemy at the level of program entity names negatively affect the location of concepts and increase the program ambiguity (Rajlich and Wilde, 2002).

Example 8.5: Example of polysemy

```
1.  class BorderLayout {
2.  ...
3.  /**
4.  * The north layout constraint (top of container).
5.  */
6.  public static final String NORTH = "North";
7.  ...
8.  /**
9.  * Constant to specify components location to
10. * be the north portion of the border layout.
11. */
12. Component north;
```

```
13. ...
14. public Object getConstraints(
15.                         Component comp) {
16.   ...
17.   if (comp == north) {
18.       return NORTH;
19.   } else if (comp == south) {
20.       return SOUTH;
21.   } else if (comp == west) {
22.       return WEST;
23.   }
24. ...
```

As we can see in the above code fragment, in the class `BorderLayout` we have two distinct concepts implemented as attributes and which are referred to through the same name: 'north'. The first concept refers to a position and is implemented through the attribute `NORTH` with type `int`. The other sense refers to a component which is placed at the top of the layout and is implemented through the attribute `north` with type `Component`. The only distinction between these names is their capitalization. On the right side is a code sequence taken from the library and which shows how these two concepts are defined. Interestingly, even the author of these lines seems to have been confused enough to use an obviously incorrect JavaDoc comment for attribute `north` with type `Component` (lines 9–10). There are further pieces of code that are affected by this naming ambiguity, e. g. in lines 17–23.

□

8.4 Related Work

Anquetil and Lethbridge (1998b) describe "reliable naming conventions" that are rules for characterizing identifiers at a semantic level:

> "Ideally, we would say that a naming convention is reliable if there is an equivalence between the names of the software artefacts and the concepts they implement."

This rule is expressed by our notion of *ideal name* presented in Definition 8.1.1. The authors present also a restricted version of the "reliable naming conventions" that refer only to naming (non-)ambiguities (synonymy and polysemy):

> "Two software artefacts with the same name should implement the same concept. [...] Two software artefacts with different names should implement different concepts."

In comparison with (Anquetil and Lethbridge, 1998b) our characterization is more detailed and formal.

The closest work to ours related to the formal characterization of names is (Deissenboeck and Pizka, 2006). The authors characterize the identifier names in terms of their consistency, conciseness and composition rules for compound words. In this paper the authors assume that a program element implements a single concept (that can be denoted through a compound word):

> "The concept identified by a compound identifier must be a specialization of the concept identified by the head of the compound identifier"

Our formal framework enriches the characterization of names along two directions:

- it allows us to characterize explicitly the difference between the apparent reference of concepts denoted through $CandCts$ and the real reference of concepts denoted through $\overleftarrow{Ref}$, and
- we make a distinction between a compound word (that refers to a single concept) and a compound identifier that can refer to more concepts.

Furthermore, once the intentional meaning is recovered, our framework allows the automatic identification of the naming defects.

8.5 Summary

Empirical studies in program understanding revealed that in practice the identifiers represent the most important source of information that link programs to domain concepts that they implement. However, even if the identifiers are generally useful, they exhibit many times a series of defects. In this chapter we use our framework for classifying the defects of identifiers along two directions: the meaningfulness of identifiers and the ambiguity of identifier names. We need to be aware of these defects since our algorithm for automatic identification of concepts in the code (presented in the previous chapter) uses the similarities between the concept and program element names as basis for the identification of concepts. Therefore our algorithm is highly restricted by the cases when the program element names are meaningless.

9 Sources of Domain Ontologies Adequate for Program Analysis

> The only source of knowledge is experience.
>
> Albert Einstein

Abstract: In this dissertation we present a new approach to define the meaning of programs by mapping their program entities to domain concepts shared within domain ontologies. A cornerstone of our approach is the fact that ontologies represent the semantic domain with respect to which the code is interpreted and characterized. Therefore, in order to apply our approach in practice, we need a considerable body of knowledge shared as (light-weighted) domain ontologies. In this chapter we present how we can obtain such knowledge from the following sources: 1) off-the-shelf ontologies, 2) ontology fragments built by analyzing the commonalities of domain specific APIs, and 3) ontologies built manually for performing particular analyses. While off-the-shelf ontologies are cheap to get, many times they have another focus and are not at the abstraction level of programs that need to be analyzed. Furthermore, currently there are no ontologies that cover the technical domains such as GUI or XML. To obtain fragments of ontologies that cover technical domains we analyze commonalities of domain specific APIs that implement the same domain. In the case of domains that are not covered by APIs and for which off-the-shelf ontologies are not available, we sketch a method to manually build ontologies based on the demands of a particular analysis. In this chapter we discuss in depth these three alternatives from the perspective of their support for automation and we focus on extracting ontologies fragments from domain specific APIs. We compare the advantages and drawbacks involved by using these sources of ontologies in order to enable the automatic conceptual analysis of software projects.

Structure of this chapter. After the introduction, presented in Section 9.1, we discuss in the following sections three sources of ontologies for analyzing programs. Section 9.2 presents the advantages and problems of using off-the-shelf ontologies for program analysis, with focus on WordNet. The core of this chapter is Section 9.3 that presents a method for extracting ontologies by analyzing the commonalities of more APIs that cover the same domain. A methodology for manually building ontologies is presented in Section 9.4. Section 9.5 presents a comparison of the three sources of ontologies with emphasis on their adequacy for program analysis. In Section 9.6 we present the related work, and Section 9.7 ends this chapter with a summary.

9.1 Introduction

A major landmark of our approach is the assumption that there exists a body of domain knowledge shared as domain ontologies. As consequence, a precondition for automation of our approach is to have (large enough) ontologies that represent adequate domain knowledge and at an appropriate level of abstraction that allows their mapping to programs.

A major distinction between our approach and the other reverse engineering approaches is the explicit assumption that this knowledge should be external to the program under analysis and that it should reflect the knowledge (and the experience) of domain experts.

> In this chapter through "ontology" we understand an artefact that represents a conceptual model of a domain. From the point of view of the specification level, we work with *light-weighted ontologies* that contain concepts and relations among them.

A domain ontology useful for analyzing a program should satisfy the following requirements:

1. **Modeling view:** The ontology should share domain knowledge from a point of view that is compatible with how is the domain implemented in the program. With other words, it should have "the same view over the world" as the program under analysis. For example, if the ontology uses other terminology for describing the domain then its concepts cannot be mapped (with our method) to programs.

2. **Completeness:** In order to analyze the domain coverage of a program the ontology must have a high completeness degree (i. e. all interesting domain concepts and relations need to be explicitly represented). If the ontology is not complete then the program elements that implement the missing concepts are seen as implementation details.

3. **Accuracy:** In order to identify diffusion and distortion of domain concepts in programs we need ontologies that reflect the domain knowledge as accurate as possible. If the ontology does not represent the domain knowledge in an accurate manner then the conclusions we draw (i. e. the mismatches we identify) are flawed.

4. **Format:** In order to enable automatic conceptual analyses based on our graph matching algorithm, the ontologies need to be represented in an adequate way (as graphs). Furthermore, in order to enable the graph matching, we need to define a priori the paths matching strategies ($\overleftrightarrow{t}$).

9.2 Off-the-Shelf Ontologies

Current off-the-self ontologies cover only restricted parts of some domains. Usually ontologies are built for a particular purpose and represent the domain concepts from a certain perspective that fits at best for achieving their purpose. Thus, even if the number of off-the-shelf ontologies is growing rapidly, most of the time there are no off-the-shelf ontologies that can be used to analyze a program.

The WordNet Ontology. WordNet[1] is an online dictionary of English inspired by psycholinguistic theories of human lexical memory. Instead of organizing the words according to their form, like the majority of other dictionaries do, WordNet organizes the words according to the meaning of the concepts they denote in sets of synonyms (synsets) (Miller et al., 1990). WordNet 2.0 contains over 150,000 words, of which more than 70% are nouns, grouped in more than 115,000 sets of synonyms. Due to polysemy, every word can express more lexicalized concepts and due to synonymy every lexicalized concept can be represented through more words. WordNet defines two different types of relations between the concepts denoted through nouns:

- *Hypernymy/Hyponymy (Generalization).* The synsets are organized hierarchically along the hypernymy (i. e. "is-a") relation. Every word definition consists of its immediate hypernym (superordinate) followed by distinguishing features. Hyponymy is the inverse relation of hypernymy. Both relations are transitive.
- *Holonymy/Meronymy (Aggregation).* In the case of nouns the distinguishing features that are explicitly encoded in WordNet are the meronyms (i. e. "part-of"). Meronyms, which represent parts of a whole, are features that can be inherited by hyponyms. Holonymy is the inverse relation of meronymy. Both relations are transitive.

S: (n) **calendar** (a system of timekeeping that defines the beginning and length and divisions of the year)
+ direct hyponym
S: (n) **lunar calendar** (a calendar based on lunar cycles)
S: (n) **lunisolar calendar** (a calendar based on both lunar and solar cycles)
S: (n) **solar calendar** (a calendar based on solar cycles)
+ direct hyponym
S: (n) **Gregorian calendar, New Style calendar** (the solar calendar now in general use, introduced by Gregory XIII ...)
+ part meronym
S: (n) **January, Jan** (the first month of the year; begins 10 days after the winter solstice)
S: (n) **February, Feb** (the month following January and preceding March)
S: (n) **March, Mar** (the month following February and preceding April)
S: (n) **April, Apr** (the month following March and preceding May)
S: (n) **May** (the month following April and preceding June)
S: (n) **June** (the month following May and preceding July)
S: (n) **July** (the month following June and preceding August)
S: (n) **August, Aug** (the month following July and preceding September)
S: (n) **September, Sep, Sept** (the month following August and preceding October)
S: (n) **October, Oct** (the month following September and preceding November)
S: (n) **November, Nov** (the month following October and preceding December)
S: (n) **December, Dec** (the last (12th) month of the year)
S: (n) **Julian calendar, Old Style calendar** (the solar calendar introduced in Rome in 46 b.c. by Julius Caesar ...)
S: (n) **Revolutionary calendar** (the calendar adopted by the first French Republic in 1793 and abandoned in 1805; ...)

Figure 9.1: Example of WordNet entries related to the concept CALENDAR

Figure 9.1 shows an example of how WordNet represents the calendar concept. We notice several hyponymy relations in the calendar hierarchy (e. g. SOLAR CALENDAR is a kind of CALENDAR) and twelve meronymy relations (e. g. JANUARY is a part of GREGORIAN CALENDAR). Figure 9.2 shows the conceptual graph containing these concepts.

[1] http://wordnet.princeton.edu

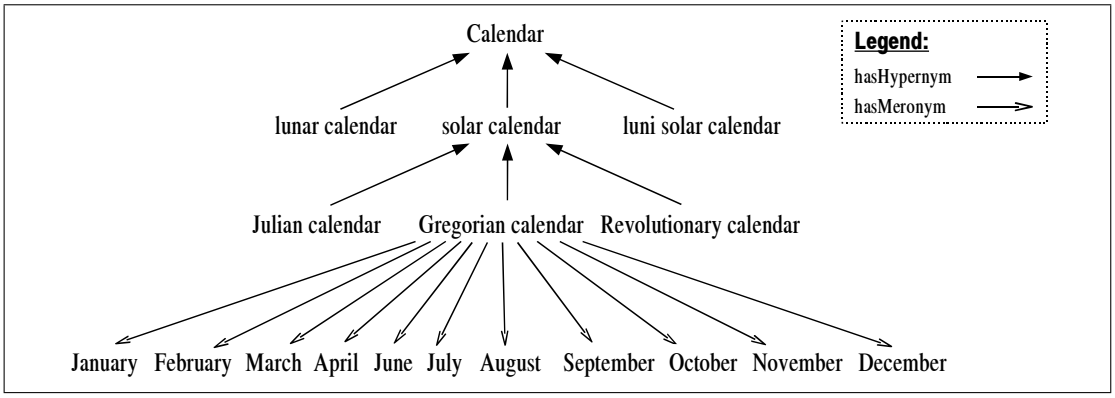


Figure 9.2: Example of a graph whose nodes are the concepts related to CALENDAR

Remark. Most of the relations in WordNet are hyponymy since all concepts are arranged in a taxonomy. The number of meronymy relations is much smaller. Due to the high number of words that it contains, there are many concepts shared in WordNet that are not relevant for the analysis of programs – e. g. in Figure 9.1 we notice the concept REVOLUTIONARY CALENDAR or LUNI SOLAR CALENDAR. In the same time, commonly used concepts that are specific to programming are not represented in WordNet – e. g. WordNet does not "know" technical concepts related to design patterns such as LISTENER, or related to graphical interfaces.

From these remarks, we can draw the conclusion that WordNet is too general to be systematically used as the main knowledge base in analyzing programs. However, due to the fact that it has a high quality (it has a large community of users and is validated in many applications) WordNet can be used for (incomplete) analyses of the mismatches in the reflexion of knowledge in programs. We present our experiences with using the WordNet for concepts location in Section 10.4.1.

> Mismatches examples from Chapters 5 and 8 are obtained by comparing parts of the Java API with the WordNet ontology.

9.3 Extracting Ontologies from Domain Specific APIs

Empirical studies on the knowledge needed by programmers during maintenance activities revealed the fact that most of the knowledge necessary for understanding programs is of technical nature – e. g. knowledge related to graphical user interfaces, networking, XML processing, communication (Anquetil et al., 2003, 2007). But technical knowledge in a machine processable format and that is at a proper abstraction level suitable for program analysis is not available off-the-shelf. One of the biggest sources of technical knowledge that is represented in a (semi-)structured form are the public interfaces of domain specific libraries. *Domain specific APIs* offer their clients ready-to-use implementations of domain concepts and we use this fact to extract ontology fragments. Extracting domain knowledge from APIs is difficult because:

domain-specific APIs

1. Due to the big abstraction gap between the domain knowledge and programs, in addition to the knowledge about their domain, the APIs are cluttered with a considerable amount of noise in form of implementation details.

2. An API offers a particular view on its domain and implements only a part of domain concepts.

In order to overcome these problems, we present a technique for extracting the domain knowledge based on the similarities of several APIs that cover the same domain. Different APIs that are developed by different programmers in different organizations and even in different programming languages, but that target the same domain, give us a much broader perspective of that domain and help us eliminate the noise. In Figure 9.3 we illustrate this situation intuitively: the upper part of this figure illustrates the forward engineering process of building APIs – starting from the same domain knowledge, different programmers provide different implementations of domain concepts. The lower part of Figure 9.3 illustrates our approach to extract the domain knowledge – starting from the commonalities of more APIs we extract fragments of a domain ontology.

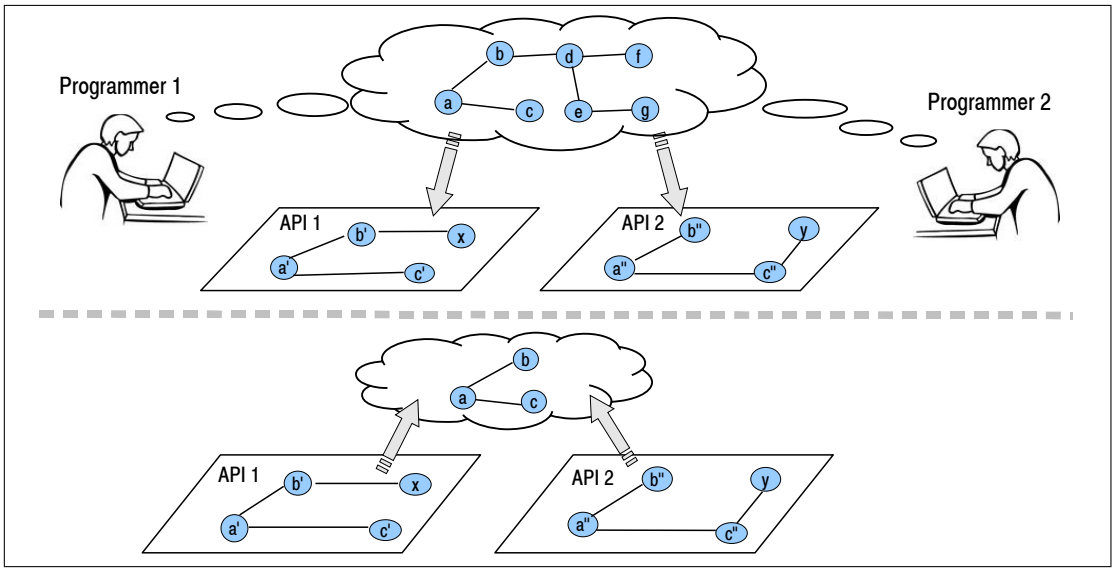


Figure 9.3: Reflecting domain knowledge in APIs (up); Extracting domain knowledge from APIs (down)

Example 9.1: Different APIs implement the same graphical concepts in similar manner

In the upper part of Figure 9.4 we present examples of basic concepts from the domain of graphical user interfaces and the relations among them (e. g. buttons are graphical components, graphical components have position and size). In the lower part we present how is this knowledge reflected in three of the most well-known APIs that implement graphical user interfaces (Java AWT, Eclipse SWT and .NET).

□

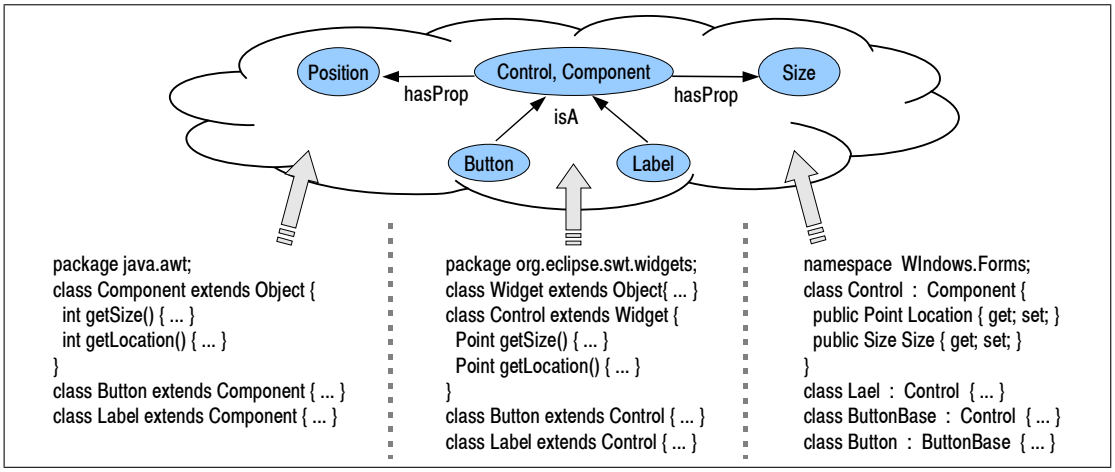


Figure 9.4: How are the graphical concepts implemented in Java AWT, Eclipse SWT and .NET

Since there is a big *conceptual gap* between the modeled domain and the general purpose programming languages, the domain concepts can be reflected in the code in a multitude of ways. To extract the domain knowledge automatically

- firstly, we need to identify a way to uniformize the possibly different implementations of the same real-world situation, and
- secondly, we need to filter out the noise introduced by particular implementation details.

9.3.1 Representing APIs

In order to identify domain concepts based on similarities of several APIs, we need to represent the APIs in a unified manner. We represent each API using the formalization of the program layer (Π^{API}) presented in Section 3.4.2. Intuitively, the program layer (Π^{API}) contains only program entities from the public interface. Based on the program layer we build the lexical layer (similar to the one presented in Section 7.4.1) that contains the reunion of words of the APIs under analysis.

APIs lexical layer

Definition 9.3.1 (APIs lexical layer)**:** *Let* $\Pi_1^{API} = (P_1,\ \Sigma^{\Pi},\ e_1^{\Pi})$ *and* $\Pi_2^{API} = (P_2,\ \Sigma^{\Pi},\ e_2^{\Pi})$ *be the program layers of two APIs. The* APIs lexical layer Λ^{API} *is:*

$$\Lambda^{API} = (I_1, W, I_2) \tag{9.1}$$

where,

- I_1, I_2 *are the set of identifiers of the program elements of* Π_1^{API} *and* Π_2^{API},
- W *is the set of morphologically normalized words obtained by the reunion of the sets of words after splitting the elements of* I_1 *and of* I_2,

Example 9.2: Examples of representing APIs

In the lower and upper parts of Figure 9.5 we present examples of two APIs: on the left side is the source code, in the middle is their instantiation according to our formal framework and on the right these APIs are represented as graphs. For example, the fact that the class `Widget` has attribute `size` is represented through the predicate: $hasAtt(Widget, size)$.

In API_2, between the nodes `Widget` and `Dialog` is the following path: $\langle hasSupCls^{-1}, hasSupCls^{-1}\rangle$. If we consider the inverse sense, namely between the node `Dialog` and `Widget` then the path is $\langle hasSupCls, hasSupCls\rangle$. Similarly, between the nodes `Widget` and `loc` the path is: $\langle hasCtr, hasParam\rangle$.

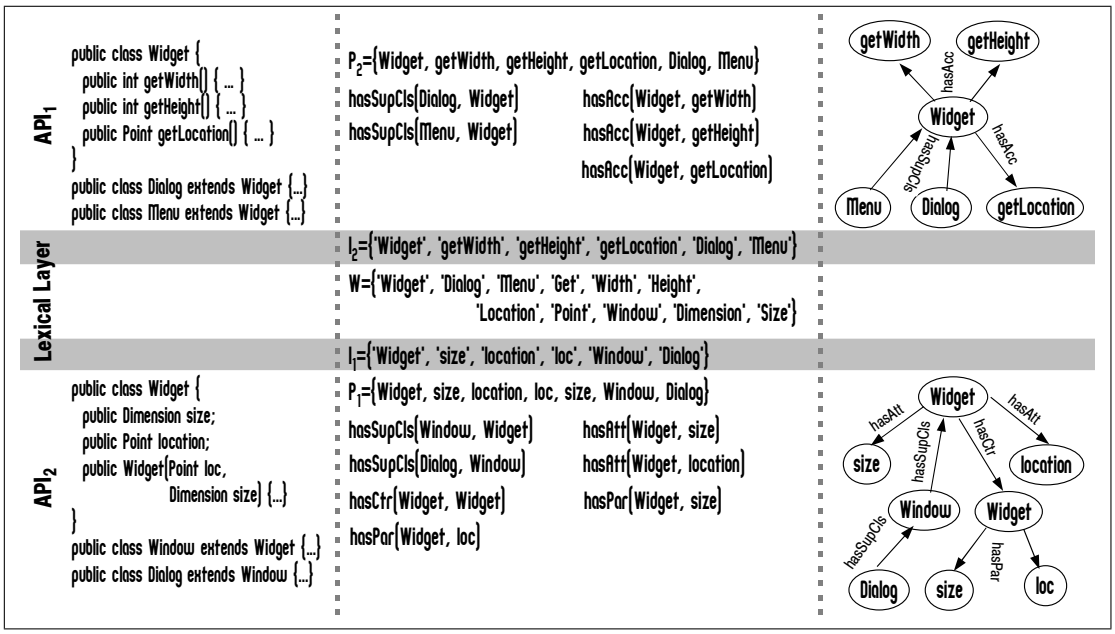


Figure 9.5: Uniformization of APIs in order to allow their comparison

□

9.3.2 Commonalities Between APIs: Names and Structure

As we presented in Section 7.2, the names of program elements and the structure of programs are the most important factors that aid the program comprehension. We make use of this information in order to extract domain knowledge from APIs.

9.3.2.1 Naming clues

We use the identifiers as hints for matching different program elements that refer to the same concept. In a similar manner with the natural language, we consider that the individual words represent the basic semantic carriers. Identifiers composed of more words can refer either to one complex concept or to several concepts (see also the discussion about compound identifiers in Section 7.2). Below we define a similarity between two program elements based on their name.

similar name

Definition 9.3.2 (Similar name)**:** *Given two program elements $p_1 \in P_1$ and $p_2 \in P_2$, the following holds:*

$$simNme(p_1, p_2) = true \Leftrightarrow \frac{\|W_1 \cap W_2\|}{\|W_1 \cup W_2\|} \geq 0.5,\ where$$
$$W_1 = I_2W(P_2I(p_1)),\ W_2 = I_2W(P_2I(p_2))$$

Intuitively, two program elements have similar names when at least half of their words are the same (the threshold 0.5 was choosen based on our experience).

Example 9.3: Example of $simNme$

$simNme(\texttt{BaseButton}, \texttt{Button}) = true$ since from the two words that these identifiers contain ('Base' and 'Button'), one word ('Button') appears in both identifiers.
$simNme(\texttt{ColoredButton}, \texttt{ColoredLabel}) = false$ since only one of the three words contained by these identifiers is shared between them.

□

9.3.2.2 Similar structure

The exclusive usage of names for the identification of commonalities between more APIs has two disadvantages:

1. based only on names we can identify the vocabulary of the domain but not the structure among the concepts, and
2. the names exhibit polysemy (i. e. a name refers to more concepts) and thereby matching two names does not imply the identification of a concept.

In order to overcome these problems we use a graph matching algorithm that extracts concepts based on the similarities of the API graphs. To apply our algorithm, we need to define similarity between paths in the program graphs and how to interpret them as conceptual level relations.

paths equivalence

Definition 9.3.3 (Paths equivalence)**:** *Let Σ^{Π} be a set of relation types among program elements and Σ^{Ω} a set of conceptual relation types. We define* paths equivalence *($\sim$)*

$$\sim \ \subset \ \Sigma^{\Pi *} \times \Sigma^{\Pi *}$$

such that,

$$\langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle \ \sim \ \langle \sigma_l^{\Pi} ... \sigma_k^{\Pi} \rangle \ \Leftrightarrow \ \exists \sigma^{\Omega} \in \Sigma^{\Omega}.\ \overleftarrow{t_e}(\langle \sigma^{\Omega} \rangle) \ \supset \ \{\langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle,\ \langle \sigma_l^{\Pi} ... \sigma_k^{\Pi} \rangle\}$$

Intuitively, $\sim$ is a relation between paths in the program graph that are used to implement the same conceptual relation. The relation $\sim$ is transitive and commutative. Below are some examples of $\sim$ between program paths (left) and the conceptual relation they correspond to (right) – in Table 10.1 we present the complete $\sim$:

T1. $\langle hasSupCls \rangle \sim \langle hasSupCls,\ hasSupCls \rangle \sim \langle attHasType \rangle \longrightarrow \mathbf{isA}$
T2. $\langle hasAtt \rangle \sim \langle hasCtr,\ hasParam \rangle \sim \langle hasAcc \rangle \longrightarrow \mathbf{hasProp}$
T3. $\langle hasMeth \rangle \longrightarrow \mathbf{isDoer}$
T4. $\langle hasParam \rangle \longrightarrow \mathbf{actsOn}$

Given two independent implementations of the same domain knowledge, there is a certain amount of variation represented by different implementation decisions. Basic variations were discussed in Chapter 6 (e. g. properties can be reflected as attributes, accessors or constructor parameters) and are captured in the equations T1 – T4. Below we discuss a case of heterogeneity in form of structural mismatches.

Structural mismatches. Above we presented the cases where the ontological relations are reflected directly in APIs. In practice, however, it is often the case that different APIs have slightly different views over the domain. For example, it is often the case that in one API the properties of a concept are implemented only in one of the sub-classes that refer to the concept. In Figure 9.6 we present two code fragments from two APIs where the relations "Widget – hasProp– Size" and "Widget – hasProp– Position" are implemented directly (left) and are implemented in a sub-class (right). From this observation we extend T2 with T5.

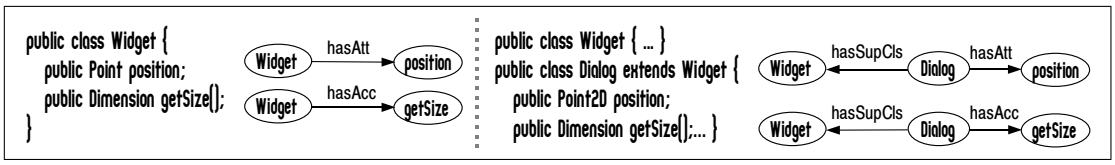


Figure 9.6: Structural mismatches in the implementation of relations: WIDGET – hasProp– SIZE and WIDGET – hasProp– POSITION

T5. $\langle hasSupCls^{-1}\ hasAtt \rangle \sim \langle hasSupCls^{-1}\ hasAcc \rangle \sim \langle hasAtt \rangle \sim$ ...

Remark. A thorough discussion about the heterogeneities that can be possibly produced in representing domain knowledge through ontologies is given in (Visser et al., 1998). Similar heterogeneities can occur in the implementation of domain knowledge in different APIs. We will consider only the heterogeneities due to the structural mismatches.

9.3.3 Ontology Extraction Algorithm

Having abstracted the APIs as graphs, the identification of similarities between several APIs is based on a graph matching algorithm (similar to our concept location algorithm presented in Section 7.5.3). Intuitively, the algorithm performs the following steps:

1. Match pairs of nodes of the two graphs based on the similarity of their names.
2. Search for compatible paths between every pair of nodes previously matched.
3. Whenever a match between nodes and paths is found, the algorithm extracts a pair of concepts and a conceptual relation between them.

Below we present our algorithm in pseudo-code and in Figure 9.7 we present an example of how this algorithm works. We consider that the APIs that are compared are $\Pi^{API} = (P, \Sigma^{\Pi}, e^{\Pi})$ and $\Pi^{API'} = (P', \Sigma^{\Pi}, e^{\Pi'})$.

1. **for-each** $p_1 \in P$ **do**
2. reflection$(p_1) = \{p'_1 \in P' \mid simNme(p'_1, p_1)\}$
3. **for-each** $\langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle \in \Sigma^{\Pi *}, \langle \sigma_k^{\Pi} ... \sigma_l^{\Pi} \rangle \in \Sigma^{\Pi *} . \langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle \sim \langle \sigma_k^{\Pi} ... \sigma_l^{\Pi} \rangle$
4. neighbours$(p_1, \langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle) = \{p_2 \in P \mid p_2 \in \langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle (p_1)\}$
5. **for-each** $p_2 \in$ neighbours$(p_1, \langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle)$
6. reflection$(p_2) = \{p'_2 \in P' \mid simNme(p'_2, p_2)\}$
7. **for-each** $p'_1 \in reflection(p_1), p'_2 \in reflection(p_2)$
8. **check-if** $p'_2 \in \langle \sigma_k^{\Pi} ... \sigma_l^{\Pi} \rangle (p'_1)$
8. **if-yes** saveRelation$(comNme(p_1, p'_1),\ comNme(p_2, p'_2),\ ontRel(\langle \sigma_i^{\Pi} ... \sigma_j^{\Pi} \rangle))$

where,

- $comNme$ takes two program elements as parameters and returns the intersection of the words of their identifiers – formally, $comNme(p, p') = I_2W(P_2I(p)) \cap I_2W(P_2I(p'))$. The words obtained through this intersection represent the name of the identified concept.
- $ontRel$ transforms a path in the program graph into its corresponding ontological relation:

$$ontRel : \Sigma^{\Pi *} \rightarrow \Sigma^{\Omega}$$

For example, $ontRel(\langle hasSupCls^{-1}\ hasAcc \rangle) = hasProperty$.

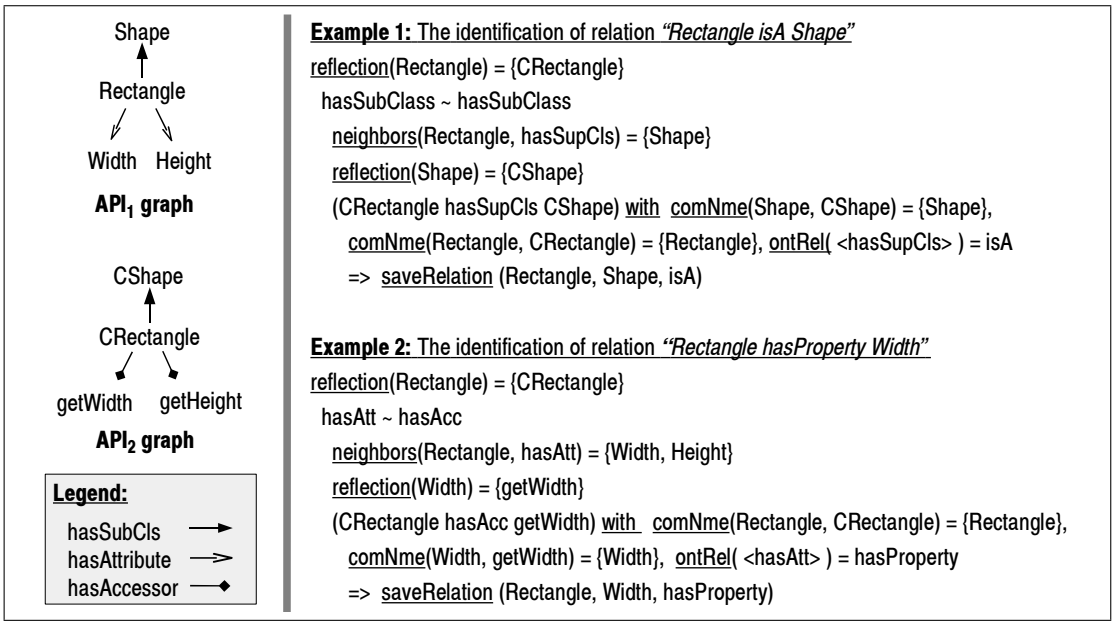


Figure 9.7: Examples of the identification of concepts and relations RECTANGLE – isA – SHAPE and RECTANGLE – hasProp – WIDTH

9.3.4 Knowledge Extraction Methodology

We developed an algorithm based on graph matching that is able to automatically find the similarities between different APIs as well as collect and interpret these similarities into fragments of a domain ontology. In order to apply this algorithm in practice we need to perform the following sequence of steps:

Step 1: Establish the scope of analysis. The very first step is to establish the scope of the analysis by answering the question: what is the domain we target?

Step 2: Select the set of APIs. In the second step we need to select the set of APIs that will be used for extracting the domain knowledge. Ideally, in order to avoid the implementation noise, these APIs should be implemented in different languages or should be provided by different organizations. Furthermore, in order to find the domain knowledge, these APIs should offer a compatible perspective over the modeled domain (e. g. should use the same terminology).

Step 3: Run the concepts identification algorithm. The running of the algorithm is fully automatic. The output of the algorithm are the ontological fragments in form of a set of concepts (candidates) and relations among them.

Step 4: Eliminate the noise. The algorithm can extract concepts and relations that are noise from the point of view of the domain knowledge. They are identified whenever more APIs exhibit the same implementation details. We eliminate the noise in two steps: Firstly, we manually inspect the list of concept candidates and eliminate the ones that do not make sense. Whenever a concept is eliminated, then all the triples that contain this concept are eliminated as well. Secondly, we manually inspect the remaining triples and remove the ones that do not make sense from the point of view of the domain.

In Section 10.2 we present our experience with extracting ontology fragments about data structures, XML and graphical user interfaces.

9.3.5 Towards a Repository of Programming Technologies Knowledge

Extracting fragments of an ontology that covers the knowledge from a domain that is modeled by several APIs is an important step to allow knowledge based reverse engineering. However, it is only a single step. Similar steps can be done by repeating the same knowledge extraction method and extracting knowledge from APIs that belong to other domains.

Empirical studies on the knowledge contained in programs (Anquetil et al., 2003) and on the knowledge used by programmers during the maintenance (Ramal et al., 2002), suggest that programmers make most frequently use of knowledge that relates to programming technologies. Programming technologies (e. g. GUIs, XML, databases, communication, data structures) represent domains that are well covered by APIs – almost all standard APIs that come with a programming language (e. g. Java, C#) implement a wide variety of concepts about programming technologies domains (e. g. DIALOG, SOCKET, FILE). The similarities between parts of the standard APIs that cover the same domain can be used as an entry point for building a knowledge repository (Ratiu et al., 2008a).

programming technologies domain

9.4 Manually Building Ontologies

Having an appropriate ontology is of capital importance for the practical applicability of our approach. Even if ontologies are envisioned to provide means for sharing large quantities of knowledge from different domains, current off-the-self ontologies cover only restricted parts of some domains. Usually the ontologies are built for a particular purpose and represent the domains from a certain perspective that fits at best for achieving their purpose. Similarly, APIs are also built for particular usage-scenarios and their interfaces offer better support for the implementation of central concepts. Thus, even if the number of the ontologies is growing rapidly, most of the times there are no available ontologies that can be used to evaluate a program. Furthermore, the automatic extraction of ontologies from domain specific APIs assumes the fact that there are already existent more APIs that cover the same domain and this is obviously not always the case.

In order to perform intentional analyses of arbitrary programs, many times we have no option but to manually build or enrich an already existing ontology. We propose a bottom-up approach for building fragments of ontologies by manually analyzing parts of the code. This approach is similar to the ontology building methodology presented in (Noy and Mcguinness, 2000).

Step 1: Establish the analysis scope. According to the particular analysis scope, we need to decide on the kinds of concepts that we want to represent through our ontology. The narrower the scope, the smaller the ontology is and therefore the easier to construct.

Step 2: Search for existing ontologies. If there exists an ontology that (partially) satisfies our analysis needs then we use it as a starting point in the manual approach.

Step 3: Gather concepts. From the list of identifiers of program elements, extract a list of names that are recognized as representing concepts relevant for the analysis; for each name add one or more concepts in the ontology. For each concept, give in addition to its name a description (glossary entry) in order to avoid confusion due to polysemy. Also record all the synonyms that refer to a concept.

Step 4: Build taxonomy. Arrange these concepts within a taxonomy based on the *isA* relation. Whenever there are sibling concepts and their parent name was not identified among the public identifiers add the parent of these concepts to the taxonomy.

Step 5: Build relations. Add additional relations between these concepts. The added relations need to be mappable to program relations (i. e. the function $\overleftrightarrow{t_e}$ need to be defined).

Step 6: Evaluation. Iteratively evaluate the consistency and accuracy of the ontology. This step is necessary in order to enable an objective evaluation of programs based on the ontology.

Step 7: Refinement. Iteratively refine the ontology by adding missing concepts and relations in order to achieve the needed completeness degree. The ontology should contain all the concepts and relations that are useful for the particular analysis. Due to the naming ambiguity such as polysemy, it happens that some concepts were not added in the ontology in Step 2. It is also possible that not all relevant relations are added in Step 4. In order to capture these situations, after mapping the ontology to programs, we should inspect the program elements that could not be mapped to any concept even if their names denote concepts from the chosen ontology scope.

Remark. Even if building the ontology manually is expensive, the knowledge contained in this ontology can be subsequently reused in the development process. Once an ontology (fragment)

is built it is a valuable artefact that reflects (a part of) the domain knowledge implemented in the program.

In Section 10.4.3 we present our experience with manually building ontology fragments.

9.5 Discussion on Ontology Sources

In the previous sections we presented different ways of obtaining domain ontologies in order to enable automatic analysis of programs. Each of these methods has its advantages and disadvantages (summarized in Table 9.5). Below we discuss the advantages (+), disadvantages (-) and neutral points (/) with respect to the cost, quality aspects (e. g. encoding bias, coverage), availability, and suitability of ontologies for analyzing programs.

Off-the-shelf ontologies. **+ Cost.** The publicly available off-the-shelf ontologies are easy to obtain. **+ Bias.** Since they are built by third parties, off-the-shelf ontologies are less biased and represent a source of knowledge external to our analysis. Furthermore, well-established ontologies (e. g. WordNet) were already used and validated by large communities and they represent an authoritative body of knowledge. **/ Scale.** The publicly available ontologies are very different in scale. Their size ranges between tens of concepts and relations up to hundreds of thousands of concepts and relations (e. g. WordNet). **- Availability.** Even if they are envisioned to share large quantities of knowledge, there are surprisingly few ontologies that are freely available and they cover a relative small quantity of knowledge. **- Focus.** The focus of publicly available ontologies can be different from what we need. **- Appropriateness for program analysis.** The off-the-shelf ontologies can contain complex relations among concepts that do not fit in our framework of automatically mapping ontologies on code.

Ontology fragments extracted from APIs. **+ Cost.** Such ontology fragments are relatively easy to obtain automatically. **+ Bias.** Since they are built by analyzing the commonalities between more APIs that are independently developed, these ontologies have a small bias.
+ Focus. The scope of these ontologies is mostly programming technologies since these domains are covered by more APIs. **+ Appropriateness for program analysis.** These ontologies are at the abstraction level close to programming and therefore are appropriate for program analysis.
/ Availability. These ontologies can be built only for domains that are covered by multiple APIs.
- Completeness. We extract only ontology fragments. The more APIs are available for analysis, the higher the completeness of the extracted ontology.

Manually built ontologies. **+ Focus.** These ontologies are very precisely targeted towards performing particular program analyses. **+ Appropriateness for program analysis.** Since they are built based on programs, these ontologies are at the abstraction level close to the analyzed program. **+ Availability.** Once the programmer possesses knowledge about the application domain, these ontologies can be built anytime. **+ Completeness.** The manually built ontologies have a high completeness with respect to the analysis to be performed. **/ Bias.** Since they are built by a person there is a danger that the domain knowledge is represented in a biased manner.

These ontologies need validation by third parties – e. g. review by a domain expert. **- Cost.** These ontologies are relatively expensive to build and the method does not scale up well.

	Off-the-Shelf	Extracted from APIs	Manually built
Cost	+	+	-
Availability	-	/	+
Bias	+	+	-
Focus	-	/	+
Coverage	-	+	+
Appropriateness	-	+	+

Table 9.1: Advantages (+), disadvantages (-), and neutral points (/) of ontologies sources

9.6 Related Work

Automatically extracting ontologies from software artefacts. Sabou (2004) presents a method for extracting an ontology that corresponds to an API by analyzing the javadoc documentation. The motivation for his work is to extract ontologies that provide semantical description for web-services for the domains where software APIs exist.

Yang et al. (1999) propose a method for representing programs as ontologies (e. g. transform records into classes of an ontology) and thereby eventually enhancing the level of abstraction at which different analyses can be performed. Yang proposes only a syntactic transformation of programs into a knowledge representation language.

In comparison with these approaches our work differs along two directions: firstly, we aim to capture the domain knowledge (as opposed to a syntactic conversion of pieces of programs into a language for representing ontologies), and secondly, we extract ontologies by analyzing the similarities between multiple APIs and not by performing natural language processing.

Building an application ontology by analyzing the application GUI. The domain concepts implemented in a program are most of the time accessible to the program users through the UI. By examining the user interface of interactive programs (e. g. office applications) one can build a domain ontology. For example, Hsi et al. (2003) present an approach for building an ontology of the application domain of a program by manually analyzing its graphical interfaces. The concepts in the ontology are given by the labels of different graphical widgets (e. g. dialogs, menus) and the relations are chosen from "is-a", "has-a" and different associations. This approach is completely manual and does not provide knowledge at the abstraction level of the code. However, it could be used as a starting point in building an ontology of the domain of a legacy application.

Comparing libraries. Michail and Notkin (1999) propose a method for comparing different libraries that address the same domain. In order to do this, they match similar components that

are provided by different libraries by using two methods: 1) names matching for identifying the components (e. g. classes, functions) that have the same normalized name in each library; 2) similarity matching that uses information retrieval techniques in order to assess the similarity of the classes and functions at the structural level. Even if it is similar in the techniques used (i. e. comparing the APIs based on program entities names), our work has another focus, namely, to extract domain knowledge from APIs. Furthermore, we use a graph based algorithm for identifying the similar structures and thereby to extract the domain concepts while Michail and colleagues compare the structural similarity by using information retrieval techniques.

Manually building domain ontologies useful for program analysis. Petrenko et al. (2008) present an approach to incrementally build fragments of a domain ontology in order to improve the code searching steps necessary for implementing a change request. The initial ontology fragment is built based on the domain knowledge contained in the change request and it is used for guiding the code searching process. Whenever necessary, the initial ontology is enlarged based on additional sources of knowledge. The ontology fragment that is built at a moment reflects the domain knowledge acquired by the programmers during the understanding process about the concepts implemented in the code. The approach of Petrenko and colleagues is similar to our manual method for building the ontology. However, their purpose for building the ontology is to mentally drive the comprehension process and not for automatically location of concepts in the code (as our aim is).

9.7 Summary

In this chapter we discussed three methods for obtaining domain ontologies adequate for analyzing programs. We focused especially on extracting domain knowledge by analyzing multiple APIs that address the same domain. In the next chapter we present our experience with extracting ontologies from APIs that cover several well-established technical domains. We also present our experience with using the WordNet ontology for locating concepts in the code, and with building manually fragments of an ontology about FIGURES for analyzing JHotDraw (Section 10.4.3).

10 Evaluation of the Automation

> In theory, there is no difference between theory and practice.
> But, in practice, there is.
>
> Chuck Reid

Abstract: We defined the intentional meaning of programs by linking program elements to concepts from domain ontologies. Even if the intentional meaning can be defined manually (by manually mapping programs to ontologies), due to the big size of the current programs, the applicability of the intentional analyses in the every-day reverse engineering depends on the measure in which they can be automated. In this chapter we take a closer look at the automation in the practice and present our experience with extracting knowledge about programming technologies from the standard APIs of Java, C#, C++ and Smalltalk, and with analyzing several well-known Java systems (e. g. parts of the Java standard API, JHotDraw and JEdit). We focus on the following practical aspects: 1) obtaining fragments of domain ontologies that make up the conceptual layer, 2) automatic recovery of the $\overrightarrow{Ref}$ function, 3) automatic identification of mismatches in reflecting the domain knowledge in programs with focus on domain coverage, quantifying diffusion, and identification of logical redundancy. We investigate the applicability of our approach, its automation degree, its precision and its costs. Our results show that programs (and especially public interfaces and APIs) exhibit high intentional meaning and its recovery is (partially) automatable with acceptable costs.

Structure of this chapter. After the introduction presented in Section 10.1, this chapter is logically structured in two parts. The first part is formed of Section 10.2 that presents three case studies about extraction of domain knowledge from domain specific APIs. In the second part we present our experience with recovering $\overleftrightarrow{Ref}$ and performing intentional analyses on Java systems. We start by presenting our tool support and the experimental setup in Section 10.3. In Section 10.4 we present our experience with automatically locating concepts by mapping ontologies to programs. Section 10.5 presents our experience with (semi-)automatic evaluation of the implementation of domain concepts in programs with respect to: conceptual coverage, diffusion, and logical redundancy. In Section 10.6 we present a detailed discussion about the threats to validity of our experimental results and in Section 10.7 we end this chapter with a summary.

10.1 Introduction

Our thesis is that the intentional meaning of programs is needed for evaluating the faithfulness of the reflexion of domain knowledge in programs. We advocated that the intentional meaning can (and should) be treated explicitly and systematically in order to increase the abstraction level of existent program analyses and define new program analyses. We started from the assumption that by matching program elements to entities from light-weighted domain ontologies we can recover the intentional meaning and thereby perform logical code analyses. In the previous chapters of this dissertation we presented many small examples of code pieces in order to support our theory. In this chapter we investigate the measure in which intentional meaning can be automatically recovered and the measure in which intentional analyses are automatically applicable in the practice. We perform our investigations with the help of several case studies that involve medium size programs. Even if these programs are not very big (i. e. they contain between one hundred and one thousand classes), they are similar to the programs currently written in the industry. By doing these case studies we investigate the differences between the theory and practice with respect to performing intentional analyses. More exactly we aim to answer a set of research questions (*RQ*) that can be subsumed to the categories presented below.

1. **Fundamental:** In what measure do real programs exhibit intentional meaning? In what measure is it expressible through light-weighted domain ontologies? (*RQ*1, *RQ*2, *RQ*9, *RQ*10, *RQ*11)

2. **Automation:** In what measure can we recover the intentional meaning of real programs? In what measure are the intentional analyses automatable? (*RQ*2, *RQ*6, *RQ*7, *RQ*9, *RQ*10, *RQ*11)

3. **Feasibility:** What are the precision and the recall of automatic intentional analyses? What are the costs of automatic intentional analyses? (*RQ*3, *RQ*4, *RQ*5, *RQ*6, *RQ*7, *RQ*8)

Remark. We are aware that these categories of questions are too ambitious, general and long-targeted to be answered in this chapter. However, we are convinced that the individual research questions (*RQ*) offer hints about the answers to these more generic categories of questions.

10.2 Extracting Ontologies from Domain Specific APIs

Research questions. In our experiments we aim to answer the following set of questions related to the extraction of domain ontology fragments from APIs:

RQ1) *Are the overlappings between different domain-specific APIs that address the same domain big enough for extracting domain ontologies?* This question addresses the most basic requirement of our ontology extraction approach – namely, that APIs (partly) exhibit the same intentional meaning (Section 10.2.1).

RQ2) *Can we identify the core concepts and relations from a domain?* This question addresses the relevance of the extracted concepts for the modeled domain (Section 10.2.2).

RQ3) *What is the amount of noise in the extracted ontology?* This question addresses the

precision of the extraction algorithm and the feasibility of eliminating the noise (Section 10.2.3).

RQ4) *What is the coverage of the extracted ontology?* This question addresses the recall of the extraction algorithm (Section 10.2.4).

RQ5) *What is the effort for extracting ontologies and eliminating the noise?* This question addresses the feasibility of the approach for extracting fragments of domain ontologies from APIs (Section 10.2.5).

Experimental setup. We present our experience with extracting domain knowledge from APIs that cover the following domains:

1. **Data structures.** We used the standard libraries from Java (the collections framework of Java 1.5), .Net (the `System.Collections` namespace), the implementation of collections provided by Squeak[1] (an implementation of the Smalltalk language), and the implementation of the C++ Standard Template Library provided by Silicon Graphics[2].

2. **XML.** In this case we chose the following APIs: the package `org.w3c.dom` is the implementation of the W3C DOM (Document Object Model) available in the Java standard library; dom4j[3] open source library for working with XML; jdom[4] library for accessing, manipulating, and outputting XML data; the XML processing API from the .NET platform (namespace `System.Xml`) and the Yaxo 2.1 API[5] from Squeak (Smalltalk dialect).

3. **Graphical user interfaces.** We chose the following APIs: the AWT and SWING APIs from the Java standard library, the Eclipse Standard Widget Toolkit (SWT), the .NET API from the namespace `Windows.Forms` and the Morphic API[6] from Squeak (Smalltalk dialect).

The reason for choosing the data structures domain is the fact that it is well defined, established, very wide-spreaded and relatively small. We have chosen the domain of graphical interfaces because it is big and relatively non-standardized (there are many variants of GUI libraries). The XML is in-between: it is bigger than data structures but smaller than GUIs; less standardized than data structures but better standardized and more focused than GUIs.

In Figure 10.1 we present an overview of the tools used for the extraction of knowledge from APIs. We remark that different APIs are exported in a common format (i. e. program graphs formally described through Π^{API}) which is fed subsequently into the knowledge extractor module (i. e. an implementation of the graph matching algorithm from Section 9.3.3). The output of the knowledge extractor are concept-relation-concept triples that are candidate fragments of a domain ontology. These triples need to be further manually reviewed in order to eliminate the noise. Besides the program graphs the knowledge extraction module is parameterizable with the set of equivalent program paths and their resulting conceptual relation (i. e. the functions $\overleftrightarrow{t_e}$ and $ontRel$).

[1] `http://www.squeak.org/`
[2] `http://www.sgi.com/tech/stl/`
[3] `www.dom4j.org`
[4] `www.jdom.org`
[5] `http://www.squeaksource.com/XMLSupport.html`
[6] `http://wiki.squeak.org/squeak/30`

Acknowledgments: The analysis of APIs from languages other than Java was made possible due to the exporters written by the following people: Martin Feilkas[7] (.NET), Adrian Lienhard[8] (Smalltalk) and Petru Mihancea[9] (C++). The knowledge extraction module, initially built by me, was improved by Yongming Li. Thank you!

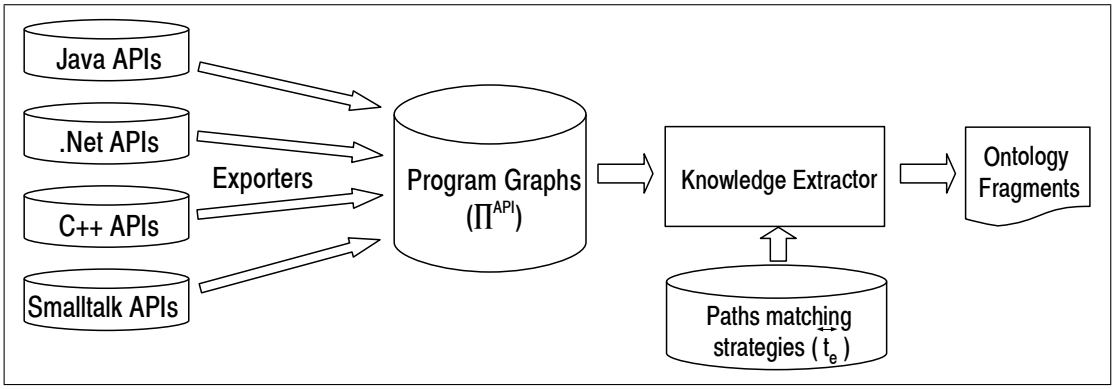


Figure 10.1: Knowledge extraction overview

Paths mapping. In Table 10.1 we present the concrete instance of the equivalence of program paths (right) and their corresponding ontological relation (i. e. the result of applying the $ontRel$ function of a program path) (left).

isA	$\langle hasSupCls \rangle \sim \langle hasSupCls,\ hasSupCls \rangle \sim \langle hasType \rangle \sim$ $\langle hasType,\ hasSupCls \rangle$
hasProp	$\langle hasAtt \rangle \sim \langle hasSupCls,\ hasAtt \rangle \sim \langle hasAcc \rangle \sim$ $\langle hasSupCls,\ hasAcc \rangle \sim \langle hasCtrPar \rangle \sim \langle hasCtrPar,\ hasType \rangle$
actsOn	$\langle hasParam \rangle \sim \langle hasParam,\ hasType \rangle \sim \langle hasParam,\ hasType,\ hasSupCls \rangle$
isDoer	$\langle hasMeth \rangle \sim \langle hasSupCls,\ hasMeth \rangle$

Table 10.1: Paths equivalence: the ontological relation (left) and the paths equivalence (right)

10.2.1 Assessing APIs Overlappings

Vocabulary Overlapping. The most naïve measurement of overlapping is at the vocabulary level. The overlapping of vocabularies shows the agreement at the terminological level between different APIs, namely the measure in which the API builders use the same vocabulary for denoting (presumably the same) concepts that are implemented in the APIs. In Tables 10.2, 10.3,

[7] http://www4.informatik.tu-muenchen.de/~feilkas/
[8] http://www.adrian-lienhard.ch/
[9] http://www.cs.utt.ro/~petrum/

10.4 we present the number of words belonging to the intersection of vocabularies of APIs from several domains. Due to the relative big differences among sizes of APIs vocabularies (even for APIs that belong to the same domain), we choose to present the overlapping in absolute numbers and not as percent. We can see that the overlappings range between under 100 words in the case of the data structures APIs and up to 800 words in the case of the graphical widgets APIs.

	Java	**.Net**	**C++**	**Smalltalk**
Java Collec. Framew.	-	64	56	90
System.Collections (.Net)	-	-	64	99
STL (C++)	-	-	-	111
Squeak Collec. (Smalltalk)	-	-	-	-

Table 10.2: Vocabulary overlapping for data structures APIs

	DOM4J	**JDOM**	**W3C.DOM**	**Yaxo**	**.Net**
DOM4J (Java)	-	163	180	103	209
JDOM (Java)	-	-	123	88	151
ORG.W3C.DOM (Java)	-	-	-	85	205
Yaxo (Smalltalk)	-	-	-	-	98
System.Xml (.Net)	-	-	-	-	-

Table 10.3: Vocabulary overlapping for XML APIs

	AWT	**SWING**	**SWT**	**QT**	**Morphic**	**Wnd.Frm**
AWT (Java)	-	583	471	821	402	499
SWING (Java)			483	856	375	553
SWT (Java)	-		-	644	322	471
QT (C++)	-		-	-	589	804
Morphic (Smalltalk)	-		-	-	-	346
Windows.Forms (.Net)	-		-	-	-	-

Table 10.4: Vocabulary overlapping for graphical widgets APIs

There is a high number of words that belong to the intersection of the APIs from the same domain.

The relative big number of common words is promising for extracting the knowledge but raises the following question: Are the common words relevant for the domain of these APIs? In Figure 10.2 we present the intersection of vocabularies of Java and .Net standard APIs for data structures (containing 64 words). This set of words is split in two parts: on the left hand side of Figure 10.2 we present the words that after the manual review we found that they refer to concepts related to data structures (44 words – ca. 66% from the number of common words) and on the right hand side of Figure 10.2 the words (22 words – ca. 33% from the number of common words) that are false positives (i. e. do not refer to concepts related to data structures). When we inspect closer the false positives we notice that most of them are either meaningless

words (e. g. 'a', 'o'), prepositions (e. g. 'at', 'as') or denote concepts related to programming languages (e. g. 'clone', 'new', 'object'). Similar results were obtained after the inspection of the common vocabularies of other API pairs.

Hashtable, Queue, Push, List, Capacity, First, Hash, Next, Peek, Vector, Trim, Add, Clear, Comparator, Entry, Search, Size, Insert, Equals, Remove, Range, Reverse, Item, Max, Contain, Binary, Previous, Factor, Last, Dictionary, Type, Stack, Collection, Array, Copy, Keyed, Linked, Pop, Value, Index, Enumerable, Load, Dest, Sort,	Code, Is, Initial, Get, All, O, Synchronized, Of, At, New, Clone, C, A, Has, To, I, Set, String, Object, As, Obj, T

Figure 10.2: Intersection of the Java and .Net vocabularies of data structures – words that belong to the data structures domain (left); words that do not belong to the data structures domain (right)

APIs that address the same domain contain a large number of common words that belong to the domain.

What about the words not belonging to the intersection of the vocabularies? Are they relevant for the domain? To answer these question we manually inspected the words that do not belong to the intersection of the vocabularies. In Figure 10.3 we present as example the words belonging either to Java or to .Net data structures vocabularies but not to both. By manually inspecting these words we remark that 85 words (ca. 60% from the different words) represent concepts not related to data structures (Figure 10.3-right) and 64 words (ca. 40% from the different words) represent words that belong to the vocabulary of data structures (Figure 10.3-left). By manually inspecting the words that belong to the vocabulary of data structures but are not found in the intersection of Java and .Net collection APIs we identified the following cases:

- usage of different abbreviations to denote the same concepts – e. g. Comparator – Comp – Cmp or Collection – Coll – Col. In Figure 10.2(left) we remark the occurrence of the words 'Collection' and 'Comparator'. From this we conclude that when the full name was used to denote a concept it was identified to belong to the common vocabulary.

- concepts missing from one API – e. g. the part of the .Net collections API that we analyzed does not contain the concept TREE.

- naming ambiguities – e. g. the .Net uses (synonymously) both words 'size' and 'length' while Java uses only 'size'. The common name was identified to belong to the vocabularies intersection.

Due to differences in terminology and different focuses, many words denoting domain concepts belong only to an API. Therefore, in order to get a good coverage of the domain, we need to analyze more APIs.

Head, Put, Tail, From, Coll, Val, Iterable, Order, Rotate, Cmp, Identity, Enum, Elem, Swap, Shuffle, Comp, Disjoint, Priority, Empty, Increment, Elements, Map, Random, Frequency, Iterator, Rest, Rnd, Sequential, Unmodifiable, Tree, Min, Access, Ensure, Fill, Complement, Replace, Col, Root, Reset, Enqueue, Count, Mask, Convert, Start, Pair, After, Table, Match, Create, Ordered, Equality, True, Comparison, Each, Exists, Dequeue, Found, Length, Move, Grow, Node, Find, Data, Before	Abstract, Expected, Target, Singleton, S, Deep, Source, Into, Offer, Weak, K, Retain, Old, More, N, An, Distance, Src, V, Poll, Checked, None, Sub, M, E, Class, J, Try, Hcp, Sender, Disposable, Callback, Y, B, Read, Case, Name, Info, Default, Deserialization, Int, Action, Not, Fixed, Ctor, X, Xor, Base, D, Util, Culture, Inner, Single, Cloneable, Context, Or, Byte, Bits, And, Output, Insensitive, Provider, Interface, Boolean, Message, Sync, Parameter, Repeat, System, Serializable, By, Section, Generic, Exception, On, Hybrid, Streaming, Current, Adapter, For, Only, Excess, Serialization,

Figure 10.3: Words not belonging to the intersection of the Java and .Net vocabularies of data structures. On the left-hand side are words that belong to the data structures domain and on the right-hand side are words that do not belong to the data structures domain

	#Concepts	#Relations	#isA	#hasProp	#isDoer	#actsOn
Data Structures	115	281	52	12	166	51
XML	314	1056	126	379	253	298
GUI	1134	3772	504	1345	1132	791

Table 10.5: Automatically obtained ontologies fragments (a quantitative overview)

Structural overlapping. In Table 10.5 we present the number of extracted concepts and relations after running our algorithm on all of the domain APIs and mapping all APIs pairs. We can notice the big difference in size between these domains – the GUI domain is ca. 10 times bigger than the data structures domain.

The conceptual overlapping between different APIs that belong to the same domain is big enough for extracting fragments of domain ontologies.

10.2.2 Identifying the Core Concepts and Relations

In order to identify the core concepts we rank the importance of the automatically extracted concepts by counting how many times they participated in a match. Analogously we did for the relations between these concepts: we counted how many times a relation was identified. Below are examples of the concepts and relations with high frequency from data structures (left), XML (center) and GUI (right) domains. In Figure 10.4 we present the most frequent 20 concepts and some frequencies (e. g. the concept LIST was identified in 746 matches in the data structures APIs).

In Figure 10.5 we present the most frequent 10 "concept – relation – concept" triples and their frequencies. We remark that in the case of data structures the most frequent relations involve the operations "remove" and "add" that represent the basic terms builders in the algebra of data structures – e. g. the triple "list – isDoer – remove" was identified 182 times. The centrality of frequent concepts shows the agreement among APIs on the central concepts and relations. In the case of GUIs we remark that many triples refer to events manipulation and the Observer pattern.

list (746), collection (737), add, remove, set, dictionary (181), contain, size, index of (152), link list, object (128), array, clear, map (85), array list (78), key (74), remove all (70), value (66), copy, hash map (56)	element (1350), attribute (828), name (775), document, string (498), node, processing instruction, value, namespace (286), text, entity, document type, node type, xml writer (170), parent, remove, add, clone, content, character datum (101), object (100)	add listener (1630), listener (1614), button, scroll bar, list, string (979), label, size, remove listener, event, x, y (636), name, color, add, font, object, remove, text, html element, e, menu, component, control (433), progress bar, value, draw, paint (387)

Figure 10.4: Concepts with the highest matching frequency (data structures (left), XML (center), GUI (right))

List–isDoer(182)–Remove	Element–hasProp(179)–Attribute	Add Listener–actsOn(1117)–Listener
List–isDoer(137)–Add	Element–hasProp(151)–Name	Name–isA(298)–String
Collection–isDoer(106)–Add	Attribute–hasProp(147)–Value	Remove Listener–actsOn(293)–Listener
List–isA(99)–Collection	Element–hasProp(110)–Namespace	E–isA(264)–Event
Collection–isDoer(88)–Remove	Name–isA(96)–String	Button–isDoer(182)–Add Listener
Set–isDoer(74)–Add	Attribute–hasProp(86)–Name	Button–isDoer(182)–Remove Listener
Set–isDoer(71)–Size	Element–isDoer(84)–Remove	Draw–actsOn(129)–X
List–isDoer(70)–Contain	Element–hasProp(79)–Text	Draw–actsOn(129)–Y
List–isDoer(67)–Index Of	Attribute–hasProp(68)–Namespace	Scroll bar–hasProp(109)–Size
Array–isA(57)–Collection	Attribute–actsOn(60)–Name	Insert–actsOn(102)–Index

Figure 10.5: Relations with the highest matching frequency (data structures (left), XML (center), GUI (right))

The frequency with which domain concepts and relations are identified is a good indicator of the centrality of these concepts in the domain.

10.2.3 Eliminating the Noise

Starting from the automatically extracted ontology, we manually inspect the concepts and the relations in two steps:

1. Firstly, we inspected the list of concepts candidates and eliminated the concepts with meaningless names and those that do not belong to the domain. For each concept that we eliminate, we remove also all the triples in which it takes part.

2. Secondly, we inspected the remaining triples (that contain now only concepts that we recognized to belong to the domain) and eliminated the triples that do not make sense from the point of view of the domain.

In Table 10.6 we present the results of the manual inspection. We remark that ca. 45% of the concepts and relations automatically extracted were classified to belong to the modeled domain of the API.

	#Concepts	#Relations	#isA	#hasProp	#isDoer	#actsOn
Data Structures	70	143	10	9	98	26
XML	138	403	36	284	48	35
GUI	526	1747	105	1106	362	174

Table 10.6: Ontologies fragments after noise elimination (a quantitative overview)

10.2.4 Coverage Estimation

We estimate the coverage along two dimensions: 1) the degree in which the concepts from our extracted ontology cover the domain, and 2) the degree in which the relations between them (i. e. from the ontology) cover the domain relations.

The coverage of the ontology can be assessed from two perspectives: the "absolute" coverage representing the percentage of the absolute number of domain concepts and those that are contained in the recovered ontology, and the "relative" coverage representing the percentage of the number of domain concepts that are implemented by at least of one of the analyzed API and that are contained in the recovered ontology. The "relative coverage" represents the recall of the algorithm.

Coverage of concepts in the case of data structures. To assess the relative conceptual coverage of the data structures ontology, we determined the difference between the set of all words belonging to the analyzed data structures APIs and those that belong to the extracted ontology. We obtained the words that denote concepts related to data structures and that are implemented in at least one API. We manually inspected this difference and selected 70 words that refer to concepts that belong to the data structures domain. From the point of view of the extracted concepts we obtained a recall of ca. 50%.

> The data structures ontology contains ca. 50% of the domain concepts that are implemented (and provided) by the analyzed data structures APIs.

To assess the "absolute" conceptual coverage we used as reference the "Dictionary of Algorithms and Data Structures" [10]. We consider that this dictionary (containing ca. 1500 entries) contains all lexicalized concepts from the domain of data structures and algorithms. From the point of view of the absolute conceptual coverage (under 5%), the extracted ontology seems to be dissapointing. However, the concepts contained in our ontology fragments are central to programming technologies; in the same time, many of the concepts from the above-mentioned dictionary (e. g. PLANAR STRAIGHT-LINE GRAPH) are very seldom used in the programming practice.

> The automatically extracted data structures ontology contains only a small fraction of the absolute number of concepts that belong to the data structures domain. However, the concepts extracted are commonly used concepts about data structures.

[10] `http://www.nist.gov/dads/`

Coverage of relations in the case of data structures. The more relations a concept takes part into, the better is the concept defined by the ontology. By inspecting the automatically extracted ontologies, we noticed big differences in the number of relations that were extracted for different concepts. For example, the concept LIST takes part in 15 relations in our ontology while the concept VECTOR only in two relations: VECTOR – isdoer – EQUAL, and VECTOR – isdoer – HASHCODE.

There is a big difference in the specification of concepts in our automatically extracted ontologies. However, the more central a concept is the more relations with its neighbours it has.

Remark. In order to assure a good coverage of the domain, the automatically obtained domain ontologies need to be manually extended. In Sections 10.4.2 and 10.5.1 we show that the coverage of extracted ontologies (fragments) is big enough for concepts location and for evaluating the conceptual coverage of APIs that belong to the same domain.

Coverage in the case of XML and GUI. In the case of the other ontologies (XML and GUI) estimating the conceptual coverage is more difficult. The cause for this is that the XML and GUI domains are less clearly defined and thereby is difficult to state wether a concept belongs to these ontologies or not. Furthermore, they contain much more concepts described through composed words (as opposed to the data structures domain that is relative simple and most concepts are denoted through single words). However, if we consider the ratio of words contained in the ontology fragments on those that refer to valid concepts, we obtain a recall around 40%. Due to the complexity of concepts (described through compound words) and the less clear boundaries of these domains, this approximation is very rough.

10.2.5 Effort Estimation

In Table 10.7 we present the duration measured in hours of the ontology extraction steps. These results represent only the experiments and do not take into account the programming efforts. We spent most of the time in selecting the set of APIs and in preparing them for analysis (e. g. removing the tests). The numbers presented are approximations and are based on our experience with Java APIs (for the APIs belonging to other languages than Java we made approximations for estimating the selection of APIs and preparation of APIs). We can notice that once we have the APIs ready for analysis, the extraction of ontological fragments and the elimination of noise are relatively quick. From these four steps only one (i. e. running the algorithm) is automatic.

API / Operation	Data Struc.	XML	GUI
Selection of APIs (manual)	2	4	7
Preparation of APIs (manual)	2	3	4
Algorithm running (automatic)	0.05	0.1	0.9
Manual noise elimination (manual)	0.5	1.0	3.0

Table 10.7: Estimation of the effort (in hours)

Once a set of APIs that cover the same domain is available, the extraction of domain ontologies fragments is (relatively) inexpensive.

10.2.6 Programming Technologies Knowledge Repository

In this section we presented our experience with extraction of knowledge about GUIs, XML and data structures. As we advocated in Section 9.3.5, in a similar manner one can extract knowledge about other programming technologies by systematically comparing different (parts of) standard libraries. We started to build such a knowledge repository that contains ontology fragments that cover technical domains and that were automatically extracted from APIs. Our repository covers currently the following domains: graphical user interfaces, XML, common data structures, databases, communication, calendar and networking.

By using APIs that cover the same domain we can inexpensively extract domain knowledge especially about common programming technologies. Our knowledge repository can be found on the web at the following address: `www4.in.tum.de/~ratiu/knowledge_repository.html`.

10.3 Tool Support and Experimental Setup for Intentional Analyses

In the following sections of this chapter we present our experience with performing intentional program analyses. In this section we present our knowledge-based reverse engineering framework that we use to perform our analyses and the systems we studied. Once the intentional meaning is recovered (the functions $\overleftrightarrow{Ref}$) then the other analyses are immediate. In order to automate the recovery of $\overleftrightarrow{Ref}$ we need the following ingredients:

1. **Program layer.** We need tools to build the program layer by extracting the interesting facts from the programs under analysis. In order to do this we use the "inCode" reverse engineering platform (Section 10.3).

2. **Lexical layer.** The similarities between the names of program elements and of concepts represent the most important source of information for automatic recovery of the intentional meaning. We use the CamelCase convention for splitting the program identifiers into words and the WordNet dictionary for performing the morphological normalization of words.

3. **Conceptual layer.** A fundamental characteristic of our approach is the assumption that domain ontologies are available beforehand. We evaluate three sources of ontologies: the use of the WordNet off-the-shelf ontology, extracting domain ontologies from domain specific APIs, or manually building ontology fragments that target specific analysis needs.

We also discuss several methodological aspects that should be followed in order to perform intentional analyses in practice.

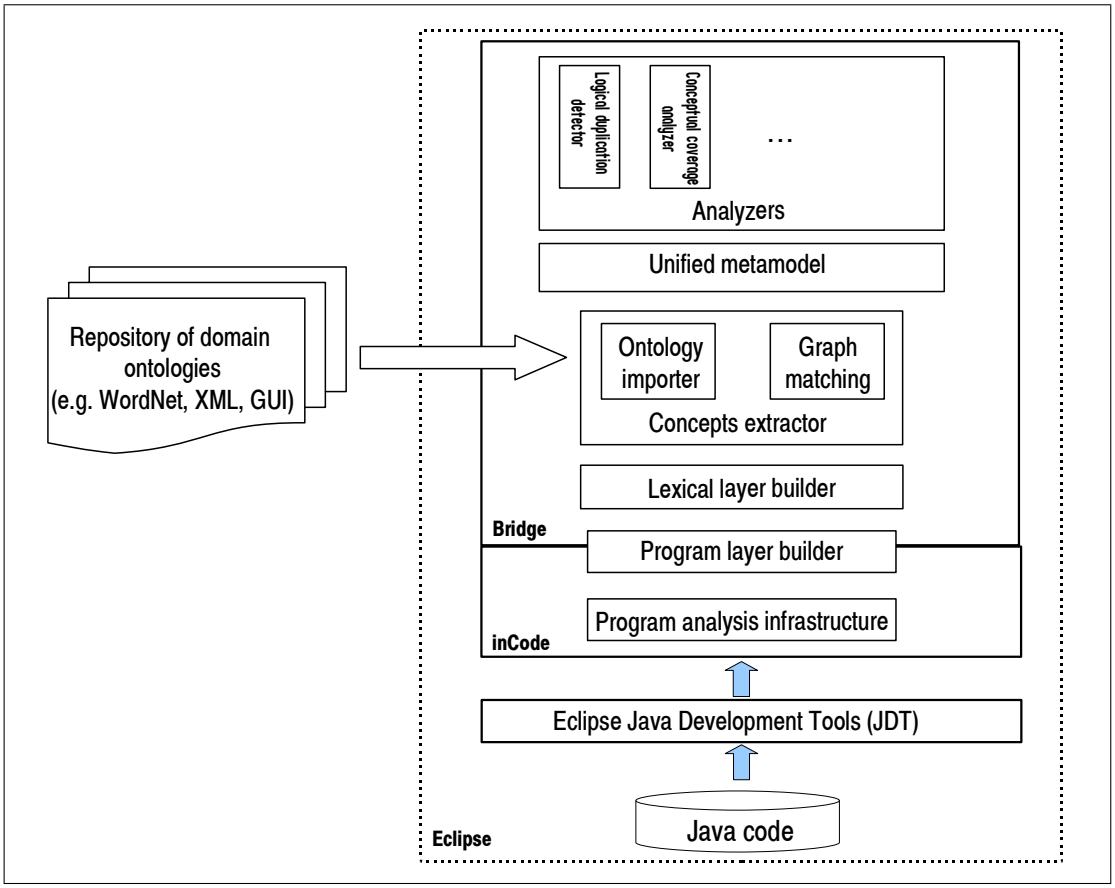


Figure 10.6: Tools architecture overview

Tool support. For performing our experiments we used our tool named **Bridge**. Bridge is based on the reverse engineering and program analysis framework "inCode"[11] developed at the "LOOSE Research Group"[12] from "Politehnica University of Timisoara". Both Bridge and inCode are Eclipse plugins. In Figure 10.6 we present an overview over our tools infrastructure. Firstly, we need to remark that the domain knowledge (fragments of domain ontologies) is outside our analysis framework. The Java code is read and parsed by the Eclipse Java Development Tools infrastructure (JDT)[13]. The inCode framework provides support for advanced program analyses and for the building of the program layer. The Bridge tool is based on inCode and consists of three modules: a module for building the program layer (makes the link with inCode), a module for building the lexical layer, and a module for extracting the concepts. Based on these builders we obtain an instance of the unified meta-model presented in Figure 7.2 (p. 152). Based on this meta-model we can define a wide variety of analyses.

[11] `http://loose.upt.ro/incode/`
[12] `http://loose.upt.ro/`
[13] `http://www.eclipse.org/jdt/`

The instantiation of our unified meta-model represents the key point in the automatic analyses. Once (a part of) this meta-model is built, many conceptual analyses are immediate. A key step in instantiating the meta-model is the recovery of $\overleftrightarrow{Ref}$. This is why we dedicate Section 10.4 only to investigate the issues with the automatic recovery of $\overleftrightarrow{Ref}$.

Systems studied. In Table 10.8 we present a short overview of the software systems on which we performed our experiments: the number of classes of analyzed systems (this number does not include the anonymous classes), the number of program entities that these systems contain (classes, attributes, methods, parameters and local variables defined in the program), the number of identifiers and the number of words.

We have chosen these case studies for the following reasons: the parts from the Java standard library because they are wide-spreaded APIs that model established domains in daily programming practice; JHotDraw[14] (version 7.0.9) because this framework (used for technical and structured graphics) is used as a benchmark in the reverse engineering community, JEdit[15] (version 4.3) is the biggest system that we analyzed, and SWT (version 3.2.2)[16] because it is one of the latest developed (and widely used) GUI APIs and therefore we assume that it has a very good quality.

Name	**#Classes**	$\|P\|$	$\|W\|$	$\|I\|$
Java 1.4.2 Collec.(Π^{API})	30	979	106	176
Java 1.5.0 Collec.(Π^{API})	37	1175	123	211
Eclipse SWT (Π^{API})	245	5854	852	2001
Java AWT (Π^{API})	354	10881	1299	3193
Java SWING (Π^{API})	719	15961	1087	4337
JHotDraw (Π^{API})	371	7358	715	1606
JHotDraw (Π^{Prog})	418	14449	1151	2916
JEdit (Π^{Prog})	1022	28555	2211	7980

Table 10.8: Overview over case studies

10.4 Automatic Location of Concepts

Research Questions. The recovery of $\overleftrightarrow{Ref}$ functions is the basic (and most important) step towards automation of our analyses. We evaluate the possibility to automate the recovery of $\overleftrightarrow{Ref}$ by mapping different ontologies on several Java systems. More concrete we will answer the following questions:

RQ6) *How appropriate is WordNet for locating concepts in code?* This question addresses the relevance of the WordNet ontology for concept location (Section 10.4.1).

RQ7) *How appropriate are the ontology fragments extracted from APIs for locating concepts*

[14] `http://sourceforge.net/projects/jhotdraw/`
[15] `http://www.jedit.org/`
[16] Without the "internal" packages.

in code? What are the precision and recall of the concepts location algorithm? This question addresses the relevance of the extracted ontologies for concept location (Section 10.4.2).

RQ8) *What is the effort necessary to build fragments of domain ontologies that are suited for conceptual analysis?* This question addresses the feasibility of integrating ontology building during reverse engineering (Section 10.4.3).

10.4.1 Using the WordNet Ontology for Concept Location

We mapped the WordNet ontology on several programs by using the paths mapping strategies ($\overleftrightarrow{t_e}$) presented in Table 10.9. The first line of this table represent the paths corresponding to the isA relation and the second line to those corresponding to the $hasPart$ relation.

Output	WordNet path	Program path
isA	$\langle hasHypern \rangle \sim$ $\langle hasHypern,\ hasHypern \rangle \sim$ $\langle hasHypern,\ hasHypern,\ hasHypern \rangle$	$\langle hasSupCls \rangle \sim \langle hasType \rangle \sim$ $\langle hasSupCls,\ hasSupCls \rangle$
hasPart	$\langle hasHolon \rangle \sim$ $\langle hasHypern,\ hasHolon \rangle$	$\langle hasAtt \rangle \sim$ $\langle hasAtt,\ hasSupCls \rangle \sim$ $\langle hasAcc \rangle \sim$ $\langle hasAcc,\ hasSupCls \rangle$

Table 10.9: Paths equivalence between WordNet (center) and the paths in the program (right)

In Table 10.10 we present the results of the automatic mapping WordNet to several Java systems. We present the number of identified concepts, identified relations and of program elements that could be mapped on WordNet concepts ('known program elements'). We remark that the number of isA relations is much bigger than $hasPart$. This can be explained by the fact that WordNet contains a much higher number of $hypernym$ relations than the number of $holonym$ relations.

Name	#Concepts	#isA	#hasPart	#Known Program Elem.
Java AWT API	46	19	4	42
Java 1.5 Collec. API	4	2	0	12
JHotDraw	79	54	9	179

Table 10.10: Identified concepts with WordNet

By manually analyzing the results obtained through the automatic mapping of WordNet on these Java programs we found out that many of the concepts that were identified represent noise – i. e. noise are those concepts that were erroneously identified to occur in programs. For example, in Figure 10.7 we present the erroneous identification of the concept BAR (with WordNet gloss entry "a heating element in an electric fire; 'an electric fire with three bars' ") and COMPONENT (with WordNet gloss entry "an artifact that is one of the individual parts of which a composite

entity is made up; especially a part that can be separated from or attached to a system 'spare components for cars' ").

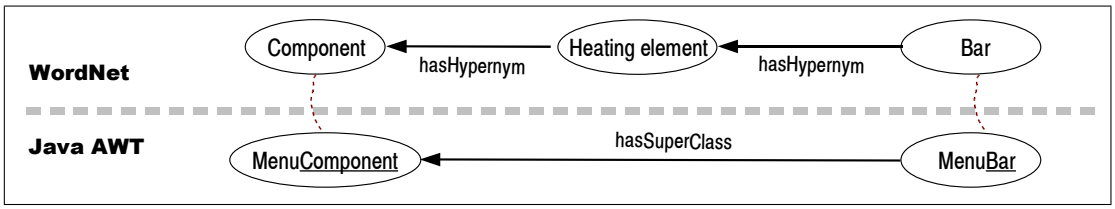


Figure 10.7: Example of erroneously identified concepts by mapping AWT to WordNet

In Table 10.11 we present the results of mapping WordNet to different programs after we eliminated the noise (false positives). The explanation for the big number of false positives is the big size of WordNet (over 100.000 concepts), and of the fact that it is a general purpose ontology (and thereby lacks focus). These facts lead in turn to mistaken identification of concepts.

Name	#Concepts	#isA	#hasPart	#Known Program Elem.
Java AWT API	15	8	1	22
Java 1.5 Collec. API	2	1	0	3
JHotDraw	27	16	1	41

Table 10.11: Concepts identified with WordNet (after manual review)

By mapping WordNet to programs we can generally identify a small number of concepts. Our results suggest that WordNet is too general to obtain a good coverage in locating concepts in the code. The identified concepts were mapped to a small fraction of the program elements (under 1%).

10.4.2 Using the Automatically Extracted Ontologies for Concept Location

In this section we answer to RQ7, namely how appropriate the ontology fragments extracted from APIs are for locating concepts in programs. We present our experience with recovering $\overrightarrow{Ref}$ for JHotDraw and JEdit. Please note that (a part of) RQ7 is answered also in Section 10.5.1 where we use the ontology fragments to evaluate the conceptual coverage of APIs. In order to evaluate our algorithm we compute the precision and recall as described below.

Computing the "precision": After running the concepts location algorithm we obtain a set of candidate mappings between concepts and program elements. Some of these mappings are false positives – i. e. the program elements do not actually refer to that concepts. We compute the precision of our algorithm by dividing the number of correct mappings (obtained after manual inspection) to the number of mappings identified automatically by our algorithm.

Computing the "recall": In order to (roughly) estimate the recall of our algorithm, we computed the number of program elements p that were correctly mapped to a concept by our algorithm ($\overrightarrow{Ref}(p) \neq \emptyset$) divided by the maximum number of program elements p' that could refer

to a known concept ($CandCts(p') \neq \emptyset$). The formula below represents a (very) conservative approximation of the recall since it ignores the polysemy cases. In the case of polysemy, even if a program element has a known name ($CandCts$ is not empty), it does not refer to any concept (the $\overrightarrow{Ref}$ is empty). Therefore, the recall of our algorithm is bigger than $Recall^{API}$.

$$Recall^{API} = \frac{|\{p \in P \mid \overrightarrow{Ref}(p) \neq \emptyset\}|}{|\{p \in P \mid CandCts(p) \neq \emptyset\}|}$$

Besides the $Recall^{API}$, that is a global measurement of the recall, we evaluate the recall of individual concepts. Our aim is to investigate whether there are differences in the recall between different concepts. Given a concept c, we conservatively approximate its recall by using the following formula:

$$Recall^{Conc}(c) = \frac{|\{p \in P \mid c \in \overrightarrow{Ref}(p)\}|}{|\{p \in P \mid c \in CandCts(p)\}|}$$

Remark. Our estimation of the recall assumes that the names of the concepts referred by a program element are contained in its identifier. In the case of many meaningless identifiers of program elements that refer to a concept c, we would erroneously obtain a high $Recall^{Conc}$ for that concept.

In Table 10.12 we present the mapping strategies between paths in the program and paths in the ontology (i. e. the functions $\overleftrightarrow{t_e}$).

Data structure concepts. In JHotDraw our concepts location algorithm automatically identified 15 concepts that belong to the data structures ontology. The manual inspection of the program elements automatically assigned to these concepts revealed the fact that ca. 20% of the identified program elements were false positives (they do not refer to data structure concepts). By computing the $Recall^{API}$ we obtained an extremely small value (ca. 6 - 7%). The manual inspection of the program elements with known names (ca. 2170) revealed that many program elements contain in their identifiers names of data structure concepts even if that program elements do not refer to any concept contained in the data structures ontology. Prominent examples are the name "set", that is used in many accessor methods in JHotDraw (ca. 270 setters) without referring to the data structure concept SET; or the name "value" that is used in ca. 400 program elements that do not actually refer to the concept VALUE from the data structures ontology (in our ontology of data structures many operations act on VALUES – i. e. we have the triple ADD - actsOn - VALUE). After the manual inspection, that removed the false mappings, we can estimate the recall to be around 25 - 30%. By computing the $Recall^{Conc}$ (i. e. approximating the recall for individual concepts) we found out that the central concepts of the data structures domain (e. g. LIST, MAP, STACK) have very high values of $Recall^{Conc}$ (over 50%). In the same time, the concepts with small recall are those that are either not central to the data structures domain (e. g. KEY, VALUE) or have names commonly used as names of program elements (e. g. ADD, REMOVE). Many of the program elements in the latter cases do not refer the concepts from the data structures domain. By mapping triples the precision of the mappings is over 90%.

In Figure 10.8 we present two examples of the mapping of LIST concept to JHotDraw code. On the left side we have a good mapping: we identified that JHotDraw implements a LIST in

Output	Ontology path	Program path
isA	$\langle isA \rangle \sim$ $\langle isA,\ isA \rangle$	$\langle hasSupCls \rangle \sim \langle hasType \rangle \sim \langle hasSupCls,\ hasSupCls \rangle$ $\sim \langle hasType,\ hasSupCls \rangle \sim \langle ctrHasClass,\ hasSupCls \rangle$
hasProp	$\langle hasProp \rangle \sim$ $\langle isA,\ hasProp \rangle \sim$ $\langle isA^{-1},\ hasProp \rangle$	$\langle hasAtt \rangle \sim \langle hasSupCls,\ hasAtt \rangle \sim$ $\langle hasType,\ hasAtt \rangle \sim \langle hasType,\ hasSupCls,\ hasAtt \rangle \sim$ $\langle ctrHasClass,\ hasAtt \rangle \sim$ $\langle ctrHasClass,\ hasSupCls,\ hasAtt \rangle \sim$ $\langle hasAcc \rangle \sim \langle hasAcc,\ hasParam \rangle \sim$ $\langle hasAcc,\ hasParam,\ hasType \rangle \sim \langle hasSupCls,\ hasAcc \rangle$ $\sim$ $\langle hasType,\ hasAcc \rangle \sim \langle hasType,\ hasSupCls,\ hasAcc \rangle \sim$ $\langle ctrHasClass,\ hasAcc \rangle \sim$ $\langle ctrHasClass,\ hasSupCls,\ hasAcc \rangle \sim \langle ctrHasPar \rangle$
isDoer	$\langle isDoer \rangle \sim$ $\langle isA,\ isDoer \rangle$	$\langle hasMeth \rangle \sim \langle hasSupCls,\ hasMeth \rangle \sim$ $\langle hasType,\ hasMeth \rangle \sim$ $\langle hasType,\ hasSupCls,\ hasMeth \rangle \sim$ $\langle ctrHasClass,\ hasMeth \rangle \sim$ $\langle ctrHasClass,\ hasSupCls,\ hasMeth \rangle$
actsOn	$\langle actsOn \rangle \sim$ $\langle actsOn,\ isA^{-1} \rangle$	$\langle hasParam \rangle \sim \langle hasParam,\ hasType \rangle \sim$ $\langle hasParam,\ hasType,\ hasSupCls \rangle$

Table 10.12: Paths equivalence between the automatically extracted ontologies (left) and the paths in the program (right)

the class `ReversedList`. On the right-hand side we have a false positive since the word 'list' of the class `ListFigure` was erroneously mapped to the concept LIST from our data structures ontology. The reason for the false mapping is that the super-class of `ListFigure`, namely `GraphicalCompositeFigure`, has methods that were confused with typical operations on lists (e. g. ADD, REMOVE, CONTAIN).

In Figure 10.9 we present a program fragment example of how is the knowledge about data structures interleaved with knowledge about design and about graphical widgets. In order to understand this code fragment programmers need to have knowledge about all domains that are weaved in the code – data structures (e. g. concepts like TABLE, KEY, PUT), design patterns (e. g. concepts like SUBJECT, OBSERVER), or user interface (e. g. concepts like TOOL BAR).

In JEdit the concepts location algorithm identified 45 concepts that belong to the data structures ontology. These concepts were mapped to 556 program elements. The manual inspection of these program elements revealed that the false positives rate is ca. 10%. By computing the $Recall^{API}$ we obtained again a small value (ca. 10 - 12%). By inspecting the program elements with known names (i. e. whose identifiers contain concepts names), we found out again that many of these program elements do not refer to data structures concepts (e. g. similar with the case of JHotDraw, we have many setters that contain the word 'set' in their names, the word

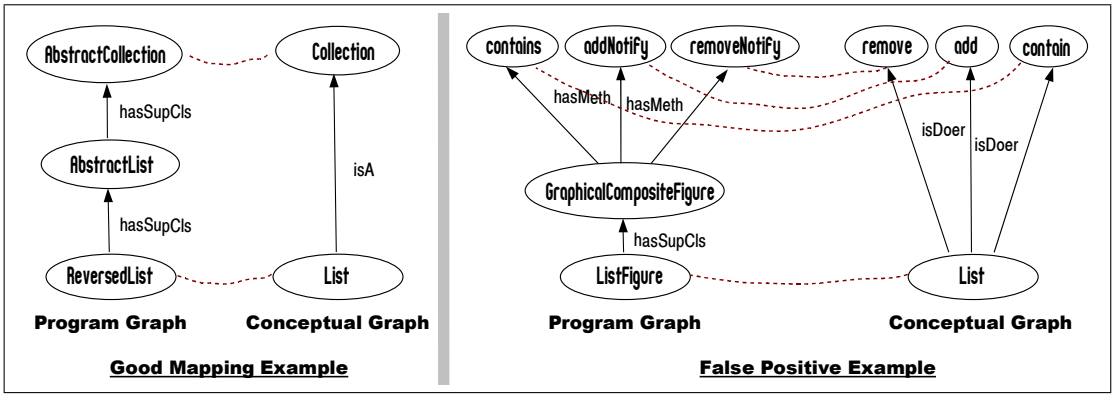


Figure 10.8: Example of identification of data structures concepts in JHotDraw

'value' occurs very often, etc). Thereby, we estimate that the recall is ca. 30-40%. Similarly with JHotDraw, concepts central to the data structures domain have a much higher recall – e. g. the recall for the concepts STACK, LIST, MAP, VECTOR is much higher (over 60%).

Remark. Many concepts from the data structures ontology are highly general and therefore many times it is difficult to decide (even during manual inspection) whether a program element refers to such a concept or not. For example, in Figure 10.8-right we illustrate a false mapping of the triple LIST - isDoer - ADD to the code. However, it is difficult to decide whether the method `addNotify` refers to the concept ADD that adds elements in a COLLECTION or the method refers actually to the concept ADD NOTIFY related to the design pattern OBSERVER.

XML concepts. In the case of JHotDraw, the concepts location algorithm automatically identified 34 concepts that refer to XML. These concepts were assigned to 758 program elements. By inspecting the program elements assigned to XML concepts, we discovered the fact that JHotDraw contains classes that use the `nanoxml`[17] API (we did not use this API for extracting our XML ontology fragments). This represents a sanity check for our approach as we validate that the XML concepts contained in our ontology are general enough and do not depend on a particular XML API. The $Recall^{API}$ was computed to be ca. 20%. After manually investigating the program elements with known names (and eliminating those that do not refer to XML concepts – e. g. the name "value" occurs in ca. 460 program elements from JHotDraw and only a small fractions of these elements refer to the "value" concept defined in the XML ontology; the word "name" is very often and occurs in ca. 537 identifiers of JHotDraw, but only a small fraction of these identifiers refer to the concept NAME that is related to XML), we estimate the recall to be ca. 40%.

In the case of JEdit, our algorithm recovered 37 concepts and these concepts were mapped to 764 program elements. We discovered however that JEdit contains yet another implementation of XML parsers given in the package `com.microstar.xml`. Beside this old implementation, JEdit uses also `org.xml.sax`. In this case our manual inspection of the mapped program elements revealed a higher amount of noise – over 50% of the program elements were mapped to

[17] `http://nanoxml.cyberelf.be/`

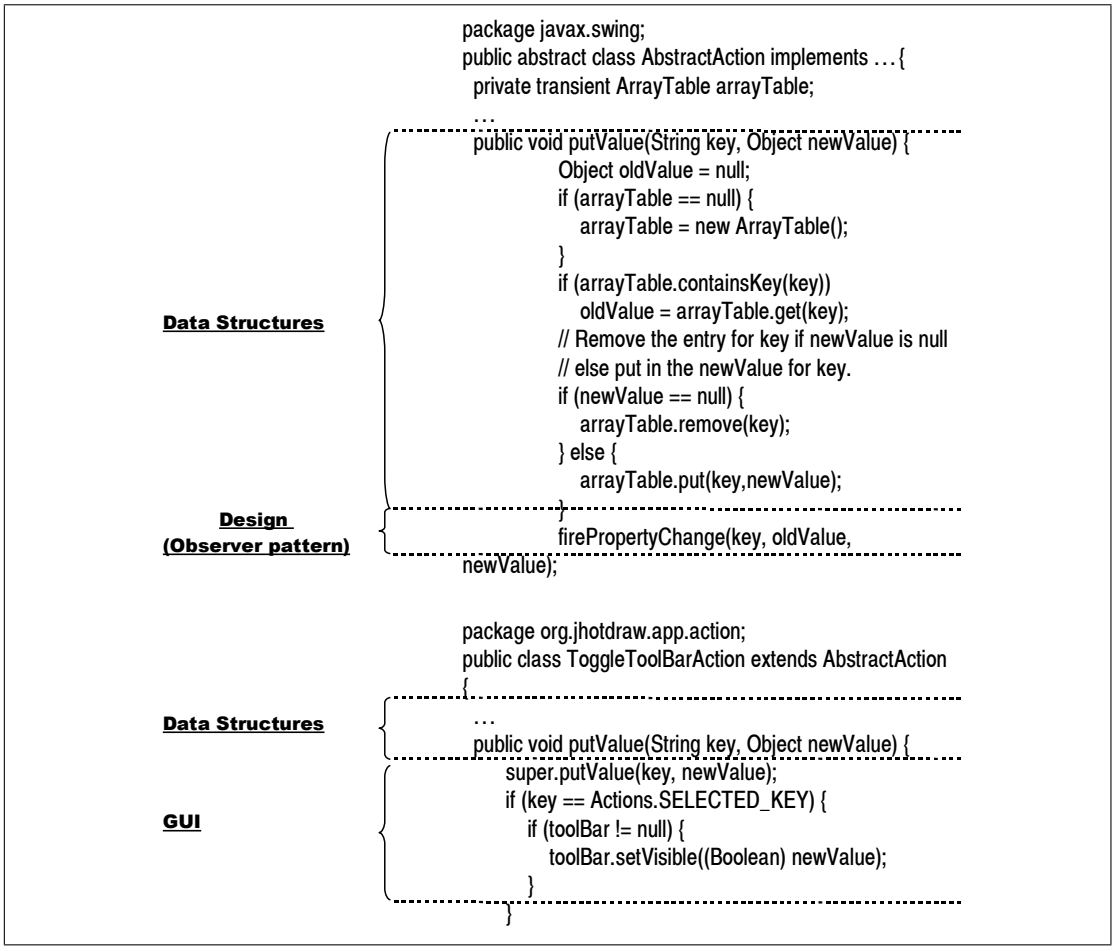


Figure 10.9: Example of interleaving of knowledge dimensions in JHotDraw

XML concepts by mistake. The reason for this is that our XML ontology contains many triples that are more general purpose (i. e. do not belong only to the XML domain). For example, the triple NODE – hasProp – PARENT belongs both to our XML and to our GUI ontologies. Furthermore, our algorithm maps only a "concept – relation – concept" triple to a pair of program elements at a time and this generates confusions in the identification of concepts. This shows a fundamental limit of our approach that generates many *false positives*: we identify only a triple (knowledge quark) at a time. The $Recall^{API}$ was computed to be ca. 10%. The reason for this relatively small value of the $Recall^{API}$ is that among the program elements that have a known name (program elements that contain in their identifiers names of concepts from our XML ontology), most of them do not refer to XML concepts. After the manual inspections, we estimate the recall to be ca. 25%.

GUI concepts. Not surprisingly, the most concepts identified in JHotDraw belong to our GUI ontology (138 concepts). These concepts were mapped to 2437 program elements. In this case the number of false positives is under 20%. By computing the $Recall^{API}$ we obtained a value

of 31%. After we manually inspected the results, we estimate the recall to be around 40%.

The manual inspection of JHotDraw revealed that FIGURE, one of the central concepts of JHotDraw, could not be located. This is due to the fact that none of our ontologies contain this concept. In Section 10.4.3 we present how we manually constructed fragments of an ontology that describes FIGURES.

In the case of JEdit, our algorithm identified 192 concepts and they were mapped to 4279 program elements. The manual inspection revealed a relatively high number of false positives (ca. 30%) due to the confusions that occurred when matching the triples. By computing the $Recall^{API}$ we obtained a value of 29%. By computing the recall values for individual concepts, we found out that $Recall^{Conc}$ for the central graphical concepts is much higher (for one fifth of the most central GUI concepts $Recall^{Conc}$ was computed to be over 70%).

In Table 10.13 we summarize our results on mapping the data structures, XML and GUI ontology fragments on our case-studies. This table has the following structure: the number of concepts that were identified, the number of isA, hasProp, isDoer, and actsOn relations, the number of program elements (#Prog. El.) that could be mapped to at least one concept, namely $\overrightarrow{Ref}(p) \neq \emptyset$, and the conceptual coverage representing the ratio of the program elements that could be mapped to at least one concept (Cov.). The conceptual coverage is obtained by dividing the #Prog. Elem. by the total number of program elements presented in Table 10.8. We remark that in all of these systems, ca. 20% of the number of program elements were identified to refer to concepts from one of the programming technologies domains covered by the data structures, XML or GUI ontology fragments.

Name	#Concepts	#isA	#hasProp	#actsOn	#isDoer	#Prog. El.	#Cov.
$JHotDraw^{entire}$	182	45	970	63	334	3091	22%
$JEdit^{entire}$	262	65	1317	93	486	5117	18%

Table 10.13: Locating GUI, XML, and data structures concepts

Summary about using the technology ontologies for concept location

False positives represent the program elements that were erroneously mapped to concepts that they do not actually reference. The false positives occur due to polysemy and one of the following causes: our algorithm matches only pairs of concepts with pairs of program elements, the relaxation of the paths mapping ($\overleftrightarrow{t_e}$), and the fact that our ontologies contain (also) information that belong to more domains. The manual inspection revealed a relatively good precision of our concepts location algorithm, the number of false positives being under 30%.

False negatives represent the concepts that were not mapped to program elements even if the program elements reference these concepts. The more false negatives, the lower the recall of our algorithm. The reason for false negatives represent the fact that no similarity was identified between the structure of the program and the structure of the ontology. We approximated the recall of our algorithm by computing the $Recall^{API}$. For both JEdit and JHotDraw we found that $Recall^{API}$ is between 10-30%. As we explained above, $Recall^{API}$ is a conservative approximation, after the manual inspection we estimate the recall to be 40-50%. However, the more

central a concept in the ontology is, the higher its recall is ($Recall^{Conc}$) – for central concepts the recall is around 70%.

Unknown concepts represent the concepts that are implemented in programs but are not known to our ontologies. The reason for this relatively low coverage is that our knowledge bases do not contain any information from the core of the application domain of our case studies. For example, our used ontologies (i. e. GUI, data structures, XML) do not know at all about FIGURES that are central concepts in JHotDraw. In the next section we present our approach for manually extending the knowledge base with the concept FIGURE and its neighbours.

> The ontology fragments related to GUI, XML and data structures that were automatically extracted from APIs support the recovery of $\overrightarrow{Ref}$ for 15 - 20% of program elements.

10.4.3 Manually Building Ontology Fragments for Concept Location

In the following we present our experience with manually building ontology fragments and thereby we answer RQ8. The semantic domain is manually built by analyzing the identifiers of the program layer and recognizing domain concepts and relations between these concepts. We use the steps from our ontology building methodology presented in Section 9.4.

Identifying the concept FIGURE in JHotDraw. As we presented in the previous section, one of the central concepts that are implemented in JHotDraw is FIGURE. Since this concept is not contained (not surprisingly) by any of our technologies ontologies, it was not located in the code. In order to study how is this concept implemented in the code, we have no choice but to manually build an ontology (fragment) that contains the FIGURE concept. In the public interface of JHotDraw ($JHotDraw^{API}$) the word 'figure' is used to name 302 program elements (ca 4% of the total number of program elements contained in the public interface of JHotDraw). Below we present the manual steps we performed in order to build the ontology triples that describe the concept FIGURE:

- **Step 1: Build the initial ontology fragment from the interface `Figure` (20 min).** The interface `org.jhotdraw.Figure` represents the core of the implementation of the concept FIGURE in JHotDraw (the concept FIGURE is referenced in many other program parts that do not reference this interface). We manually reviewed the vocabulary of this interface and built ontology fragments that contain the relation between figure and other concepts identified in the interface `Figure`.
- **Step 2: Run the concepts location algorithm and investigate the references to Figure that were not identified (10 min).** The concepts location algorithm (ran on $JHotDraw^{API}$) identified that 175 program elements refer to FIGURE. 127 program elements that refer to FIGURE could not be identified.
- **Step 3: Add new concepts to the ontology (30 min).** We inspected the remaining program elements referring to FIGURE (that have the word 'figure' in their names) and incrementally added new concepts that are neighbors at the program level with FIGURE.

Constantly we ran the concepts location algorithm and inspected only the unknown program elements that contain the word 'figure'.

After one hour of building the ontology and running the algorithm we could identify 241 program elements (recall is ca. 80%) that contain in their name the word 'Figure'. The remaining program elements could not be identified by our concepts location algorithm. We classified the remaining program elements in the following categories:

- **Diffusions.** Many of these program elements represent methods whose names contain both the action to be performed and the object on which the action will be performed (e. g. `findFigure`). They represent cases of reference diffusion and since the relation between the action and the object is not explicit they cannot be mapped to the ontology. In order to identify these references we need to interpret the compound identifiers (Section 10.5.2).

- **Events.** Many graphical events are triggered with the help of methods and many of these methods contain the word 'figure' in their name – e. g. `figureRemoved`, `figureAttributeChanged`. Our concepts location algorithm cannot identify the occurrences of the concept FIGURE in these methods since our conceptual layer relations cannot capture events (i. e. we cannot say for example TRIANGLE –isDoer – FIGURE REMOVED).

- **States.** There are several methods that query the state of a class – e. g. `isFigureSelected`. The reference to the concept FIGURE could not be identified in these methods.

For the concepts central to an application it is feasible to manually build ontology fragments that can lead to a good coverage (ca. 80%) in the recovery of $\overleftrightarrow{Ref}$ for that concepts.

10.5 Evaluating the Reflexion of Domain in Programs

Research questions. The following questions deal with several aspects of the reflexion of domain in programs presented in Chapter 4: conceptual coverage (*RQ*9), diffusion of domain in the code (*RQ*10), and logical redundancy (*RQ*11).

RQ9) *Can the conceptual coverage of APIs be automatically assessed? Can the automatically extracted ontologies be used for assessing the coverage of APIs from the same domain?* This question addresses the issue of automation in evaluating the conceptual coverage and conceptual extensibility of APIs. Furthermore, it addresses the relevance of the automatically extracted ontologies for assessing the quality of other APIs (Section 10.5.1).

RQ10) *Can we identify diffusion of concepts?* This question addresses the measure in which we can detect the diffusion (Section 10.5.2).

RQ11) *Can we identify redundant definitions in APIs? Can we identify redundant representations of domain concepts in APIs?* This question addresses the identification of logical redundancy in APIs (Section 10.5.3).

10.5.1 Assessing Conceptual Coverage of APIs

We assess the conceptual coverage of an API by analyzing $\overleftrightarrow{Ref}$. The domain concepts whose reference cannot be identified in the API represent the concepts not implemented in the API. In our experiments we use the ontologies that we automatically extracted from APIs in order to assess the conceptual coverage of individual APIs from the same domain. Below we present our experience with evaluation of the coverage of three systems: the Java collections framework, Java AWT, and Eclipse SWT.

Java collections. We use the data structures ontology to identify the domain concepts that are (or are not) implemented in two versions of the Java collections framework that correspond to the JDK versions 1.4.2 and 1.5. In Table 10.14 we present the number of concepts and conceptual relations from the data structures ontology that our automatic concepts location algorithm could (or could not) map to the program entities and program relations of the Java collections framework. We remark that the Java collections framework has a relatively good conceptual coverage (ca. 80%) of the data structure ontology. We can also observe that in the version 1.5 we have more domain concepts and relations implemented.

	#Found conc.	**#Not found conc.**	**#Found rel.**	**#Not found rel.**
Java collec. 1.4.2	58	13	121	38
Java collec. 1.5.0	59	12	128	31

Table 10.14: Results of mapping the data structures ontology on Java collections framework

On the left side of Figure 10.10 we present the concepts that were not identified in the version 1.4.2 of the Java collections framework; on the right-hand side of the same figure we present the concepts that were not identified in the version 1.5.0. We notice that the concept QUEUE was not identified in version 1.4.2 but was identified in version 1.5.0. The manual inspection revealed that QUEUE is implemented in the Java collections framework only since version 1.5.0. We notice that apart from QUEUE, there are other concepts that were not identified in both versions of Java collections. The meaning of some of the concepts that were not found in the Java collections API is presented in Figure 10.11 – this figure presents triples of the data structures ontology that contain these concepts. In this figure we see for example that ADD BEFORE and ADD AFTER are operations on LINKED LISTS that are not provided by the Java collection framework. By manually inspecting the class `java.util.LinkedList` we found out that it contains a method `addBefore` but this method is private. The functionality of ADD BEFORE is implemented with the method `add(int index, E element)`. We also note that among concepts that were not identified are variations of FIND. These concepts were not identified due to a terminological mismatch – the Java library uses the method `indexOf` for searching. We can also notice that the LIST NODES are not explicitly referenceable in the Java collections API.

Java AWT. In order to assess the conceptual coverage of the Java AWT API we use the GUI ontology. After running the algorithm for concepts location (i. e. recovering $\overleftrightarrow{Ref}$) we identified

add before, add after, capacity, count, entry, find, find index, find last, list node, next, queue, search, top	add after, add before, capacity, count, entry, find, find index, find last, list node, next, search, top

Figure 10.10: Concepts from the data structures ontology and that are not referenced by the Java collections framework (version 1.4.2 (left) and 1.5.0 (right))

Queue – isA – Collection	Linked List – isDoer – Add After	Map – isDoer – Find
Queue – isDoer – Peek	Linked List – isDoer – Add Before	List – isDoer – Find
Queue – isDoer – Remove	Stack – isDoer – Top	Link List – isDoer – Find
Queue – isDoer – Remove All	Stack – isDoer – Be Empty	List – isDoer – Find Last
Queue – isDoer – Be Empty	List – hasProp – Capacity	Link List – isDoer – Find Last
Queue – isDoer – Empty	Array List – hasProp – Capacity	List Node – hasProp – Value
Queue – isDoer – Clear	List – isDoer – Search	List Node – hasProp – Next

Figure 10.11: Concepts not identified in version 1.4.2 of the Java collections framework and some of their neighbours in the data structures ontology

that 266 concepts are not referenced by the program elements from the Java AWT API. In Figure 10.12 we present a part of the concepts that were not identified to be referenced in the AWT.

box, browser, check box, check box menu item, combo box, document, find, font dialog, form, group box, html document, list box, print dialog, progress bar, radio button, scroll bar, slider, spinner, status bar, table, text box, text style, tool bar, tool tip, tree

Figure 10.12: Concepts not referenced in the Java AWT API

After manual inspection, we notice that in most of the concepts from Figure 10.12 are indeed not implemented in AWT. In order to find out the meaning of these concepts, we have to look at their neighbours in the GUI ontology. In Figure 10.13 we present the most important triples from our ontology in which some of the missing concepts from Figure 10.12 occur. We remark that many of the missing concepts represent advanced graphical features (e. g. tooltips, special dialogs, html browsing support). Indeed, these concepts are not implemented by AWT – the explanation for the lack of advanced components is that AWT is the lowest-common denominator for GUI components defined for all Java host environments.

By consulting a comparison[18] of AWT, SWING and SWT (Feigenbaum, 2008) we found out that we accurately identified most of the missing concepts from Java AWT (presented in Table 10.15). However, by manually investigating the documentation of AWT we found out that some of the concepts that we identified to be missing (e. g. ones presented in the lower part of Figure 10.13) were in fact (indirectly) supported by AWT. For example, even if AWT does not provide direct support for radio buttons, they can be simulated through check-boxes. The class `java.awt.Checkbox` implements also radio buttons; to create a radio button one needs to create a checkbox and add it to a group. Another example, are the concepts CUT, COPY and

[18] `http://www.ibm.com/developerworks/grid/library/os-swingswt/`

Browser – hasProp – Url	Html Document – isA – Document
Browser – isDoer – Layout	Html Document – hasProp – Link
Browser – isDoer – Back	Html Document – hasProp – Link Color
Browser – isDoer – Forward	Html Document – hasProp – Active Link Color
Browser – isDoer – Refresh	Html Document – hasProp – Visit Link Color
Browser – isDoer – Stop	Html Document – isDoer – Parse
Combo Box – hasProp – Foreground	Html Document – isDoer – Open
Combo Box – hasProp – Item	Html Document – isDoer – Focus
Combo Box – hasProp – Select Index	Print Dialog – isA – Dialog
Combo Box – hasProp – Select Item	Print Dialog – hasProp – Printer
Combo Box – isDoer – Insert Item	Print Dialog – hasProp – Print To File
Combo Box – isDoer – Find	Print Dialog – hasProp – Text
Combo Box – isDoer – Select	List View – hasProp – Item
Combo Box – isDoer – Paint	List View – hasProp – Alignment
Combo Box – isDoer – Update	List View – hasProp – Select Item
Text – isDoer – Copy	Radio Button – isA – Button
Text – isDoer – Cut	Radio Button – isA – Toggle Button
Text – isDoer – Paste	

Figure 10.13: Examples of concepts not identified in AWT and some of their neighbours in the GUI ontology

PASTE. Figure 10.13 shows that these concepts belong to the set of actions that are done by/on texts. From here, we deduced that the AWT does not provide the functionality for copying text inside any of its component. The manual inspection of the AWT documentation revealed that the `java.awt.TextArea` provides this kind of functionality. However, it is implemented in another manner and because of this we could not find it automatically. The AWT implementation of these text operations (e. g. in the classes `TextComponent` and `TextArea`) is of algorithmic nature – the CUT, COPY and PASTE concepts are implementable through a combination of methods `selection`, `insert` and `replace`.

These false negatives show one of the fundamental limits of our approach, namely, by describing the domain as light-weighted ontologies we cannot capture algorithmic combinations of concepts such as the simulation of advanced operations on texts with primitive operations.

Beside the graphical concepts missing from AWT and presented in Table 10.15, we identified additional concepts that are not implemented in AWT – e. g. SPINNER, STATUS BAR, SLIDER, SPLITTER, GROUP BOXES.

Eclipse SWT. We performed the same experiment by analyzing the conceptual coverage of Eclipse SWT. We identified several concepts that after the manual inspection proved not to be implemented in SWT – e. g. AFFINE TRANSFORM and FOCUS TRAVERSAL POLICIES. However, our experiments returned many false positives (i. e. concepts that we could not find even if supported by SWT). The cause for this is the fact that many graphical components are implemented only in an algorithmic manner and this is not reflected in the structure of the program. Many special widgets are implemented as more general components that can be parameterized

Feature not available in Java AWT[19]	Missing identified
Display an image	yes
Display text and image	no
ToolTip pop-up help	yes
Styled text entry	yes
Simple push button with text and/or image	yes
Enter text or select from a drop-down list	yes
Display an insertion caret	yes
Web browser	yes
Generic container of other controls with a border and title	no
Arrow buttons	no
Display simple message dialog	yes
Display simple prompting dialog	no
Display a tool bar	yes
Display a progress bar	yes
Divide space between areas	yes
Display tabbed areas	no
Display tabular info	yes
Format table columns	yes
Display hierarchical info	yes
Select from range of values	yes
Select from discrete range of values	yes
Add items to the system tray	no

Table 10.15: Features not available in Java AWT (after (Feigenbaum, 2008))

with certain constants. For example, we erroneously identified that the following concepts are missing from SWT (even if this is not the case): POPUP MENU (this is implemented in SWT as the constant `SWT.POP_UP`), CHECK BOX BUTTON and CHECK BOX MENU ITEM (are implemented in SWT as buttons and menu items that are initialized with the constant `SWT.CHECK`), RADIO BUTTONS (are implemented as buttons initialized with the constant `SWT.RADIO`).

Automatically extracted ontologies from domain specific APIs can be used to assess the conceptual coverage of APIs that address the same domain. Whenever a domain concept is identified to be implemented in an API it proved to be really implemented. However, due to implementation intricacies, there are many concepts that are implemented (encoded) in APIs but are not identifiable through our method.

10.5.2 Assessing Diffusion

Reference diffusion. Reference diffusion occurs when a program element refers to more concepts (Definition 5.4.2). In order to detect the diffusion automatically, we need to be able to interpret the compound identifiers (identifiers that contain more words) and identify the concepts to which they refer. Most of the compound identifiers are the names of classes and of methods.

1. Reference diffusion in classes. When more concepts are referenced by the name of a single class, then the interface of the class is segregated between these concepts – e. g. a subset of the methods, accessors or attributes of that class are related with a concept and another subset with another concept. In Figure 10.14 we present an example of a class (`TextAreaFigure`) from JHotDraw that was identified to exhibit reference diffusion. On the left-hand side we present a class diagram and on the right-hand side we illustrate how the concepts TEXT and FIGURE were identified.

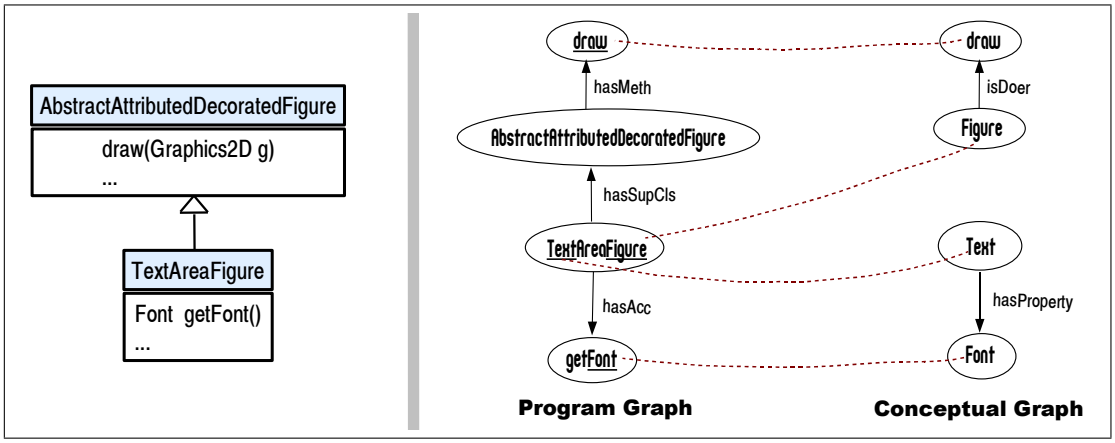


Figure 10.14: Example of identification of diffusion in classes

In the case of JHotDraw we identified 25 classes that exhibit reference diffusion. The completeness and accuracy in the identification of diffusion is limited by the accuracy and coverage of our concepts location algorithm.

2. Reference diffusion in methods. In Section 10.4.3 we showed that many references of the concept FIGURE could not be identified in JHotDraw because they are inside methods names. These methods refer to both the action that is performed and to the object on which the action is performed – e. g. the identifier 'findFigure' refers both to the action FIND and to the object FIGURE on which the action is performed. Based on our experience, we propose the following heuristic to interpret the compound identifiers – for each compound identifier i of a method such that $i = \langle w_1, ..., w_n \rangle$, we considered that the action is represented by the first word w_1 and that one of the next words $(w_2, ..., w_n)$ refers to the object on which the action is executed. In Figure 10.15 we illustrate how we interpret the compound identifier 'findFigureBehind' and how we identify the concepts FIND and FIGURE.

Using this heuristic, we identified $\overrightarrow{Ref}$ for 199 methods from $JHotDraw^{API}$. All these methods exhibit reference diffusion.

By interpreting the compound identifiers of methods we can identify diffusions in the concepts referenced by methods.

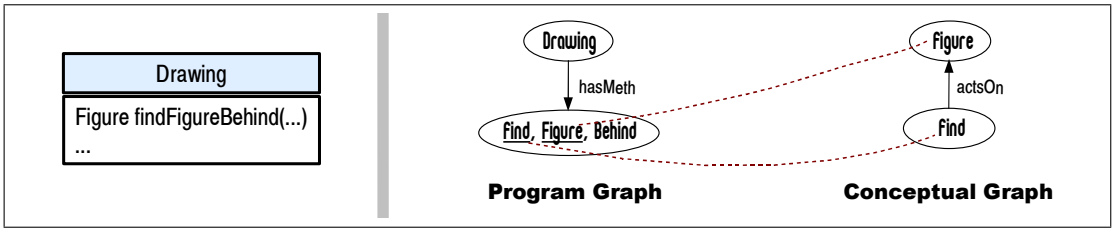


Figure 10.15: Example of identification of diffusion in methods

Representation diffusion. The diffusion of the representation occurs when the same type is used to represent distinct concepts (Definition 5.4.5) and is quantified through the overloading degree (Definition 5.4.6). We approximate the representation overloading degree by computing the number of variables with the same type but with different names. In order to do this approximation we used the "reliable naming conventions" assumption (Anquetil and Lethbridge, 1998b) (also described in Section 8.4) and approximate that two variables with different names refer to different concepts and two variables with the same name refer to the same concept.

Type	OD
int	1065
MediaType	148
float	82
double	75
boolean	65
String	52
Object	50
Color	33
long	29
SystemColor	26

$JavaAWT^{API}$

Type	OD
int	451
String	288
boolean	109
Object	100
Tag	82
Region	59
Attribute	54
Color	28
float	24
Component	24

$JavaSWING^{API}$

Type	OD
int	821
boolean	75
short	69
String	41
float	31
int[]	14
Control	11
char	9
Color	9
Object	8

SWT^{API}

Type	OD
int	89
String	83
double	58
AttributeKey	41
boolean	33
Object	28
float	25
Double	19
Figure	17
Map	11

$JHotDraw^{API}$

Figure 10.16: The Top 10 overloaded types from AWT, SWING, SWT and JHotDraw

In Figure 10.16 we present examples of types with the highest overloading degree from the APIs of AWT, SWING, SWT and JHotdraw. We remark that the primitive types, especially `int` and `String` are highly overloaded. The explanation for this is that every time when the API providers do not have a better solution, they tend to encode the concepts as numbers or strings of characters. High overloading of the primitive types represent cases of underspecification at the program level. In case a non-primitive type T exhibits a high overloading, the concepts referred by the variables of this type are many times sub-concepts of the concept referred by the type T – e. g. the type `Html.Attribute` from SWING is used as a type-safe enumeration of the Html attributes (e. g. COLOR, SIZE); the type `MediaType` from AWT is a type-safe enumeration of possible paper sizes (e. g. A0, A4, ISO_C7_ENVELOPE). We also remark that apart from enumerations, the user-defined types have a much smaller overloading degree.

The overloading degree can be approximated by using the "reliable naming conventions" assumption. The highest overloading is exhibited by the basic types `int`, `String`, `double`, `boolean`, `Object`, and `float`.

10.5.3 Assessing Logical Redundancy

Once we recover $\overleftrightarrow{Ref}$ (and thereby we instantiate the unified meta-model), the identification of definition redundancy is straight-forward. In order to identify the definition redundancy, we have to identify the concepts that are defined by more classes (Definition 5.5.1); in order to identify representation redundancy we have to find the concepts represented through more types (Definition 5.5.2).

Our automatically obtained results showed that 43 GUI concepts were redundantly defined in the AWT API. The redundant definitions affect 114 classes (this represents ca. 30% of the total number of public classes of AWT). The extremely high number of classes that represent redundant definitions of concepts was a surprise for us. After we manually inspected these classes, we found out the following categories of redundancies (the manual investigation revealed that the other classes, beside the ones presented below, were false positives):

- *Parallel inheritance hierarchies:* Twenty classes from the graphical components hierarchy (from package `java.awt`) have a corresponding interface in the package `java.awt.peer` (Figure 10.17). By consulting the documentation corresponding to the package `java.awt.peer` we found out that indeed these classes interface the functionality of the graphical components and are useful for porting AWT to other platforms. The "peers" are not intended to be used or to be extended by the end-users of AWT, only by its developers. Since our domain ontology, (the automatically extracted GUI ontology) does not "have any knowledge" about porting graphical libraries, we identified the pairs of components and their peers (e. g. `Component` − `ComponentPeer`, `Window` − `WindowPeer`) to be logical definition redundancy of GUI concepts (e. g. COMPONENT, WINDOW).
- *Optimizations of the speed:* The manual investigation revealed several situations where due to the optimizations of the used memory, the same shapes were implemented as having integers and floats as coordinates (as illustrated in Figure 10.18).

In Figure 10.18 we also notice that the concepts referring to coordinates are redundantly represented (e. g. the concept X is represented as `int`, `float`, and `double`).

By automatically recovering the $\overrightarrow{Ref}$ we can identify logical redundancy. A significant part of the redundancy cannot be however avoided due to the constraints of the current programming languages.

10.6 Threats to Validity

In this chapter we investigated several research questions that refer to the manner in which programs exhibit intentional meaning and to the possibility of automating intentional analyses.

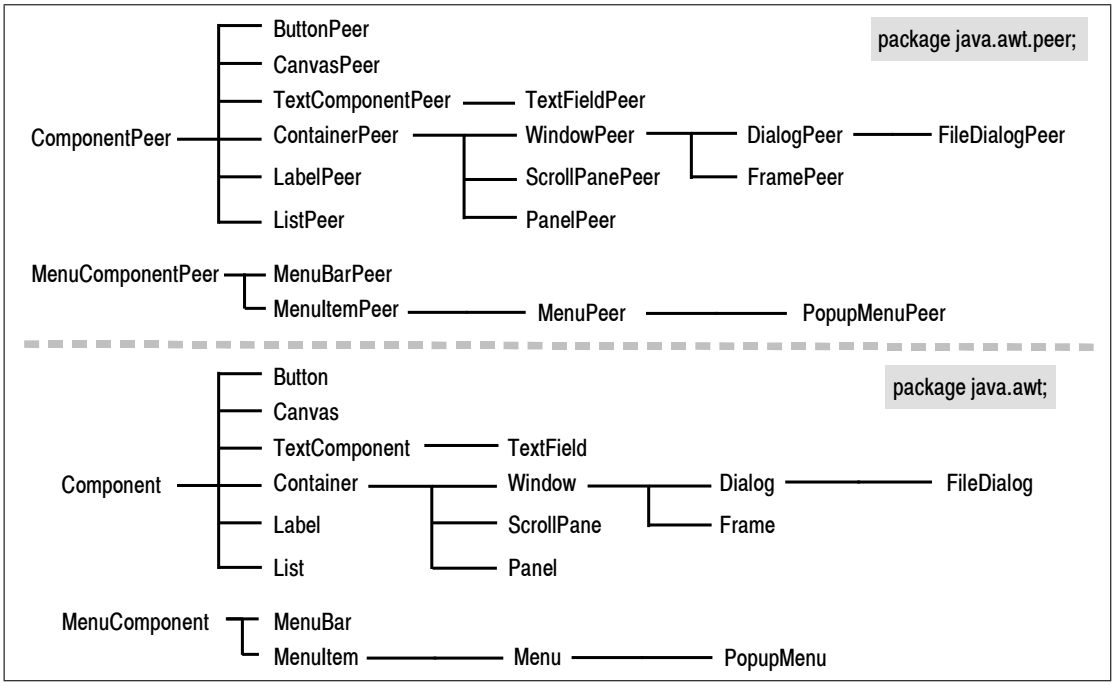


Figure 10.17: Many graphical components from AWT have a corresponding "peer"

Even if the results of our experiments seem promising to us, we are aware that due to the limited case studies investigated and the many variability points in our approach, the exact interpretation and generalization of our results is not trivial. Below we explicitly illustrate some of the key issues that could influence the answers to our research questions (internal validity) and that could prevent the generalization of our conclusions (external validity).

Internal Validity

The "internal validity" represents the validity of causal inferences (relations) that we made (Trochim, 2006). Due to the nature of the case studies (e. g. the results are influenced by our interpretation of domain concepts, by the informal knowledge contained in the programs) many of the answers to our research questions can be biased. Below we discuss the most important sources of noise that could affect our results.

Quality of identifiers. Our automatic analyses are strongly based on the similarity between the names of concepts and the program identifiers. What happens when the identifiers are meaningless, when they are ambiguous, or when they are misleadingly chosen?

- When the identifiers are meaningless we cannot identify concepts. The meaningless identifiers directly affect the analyses referring to conceptual coverage. Our analyses about coverage used parts of the Java standard APIs as case studies and whose identifiers are mostly well chosen. Furthermore, we validated the coverage analysis by using an external

```
public class Rectangle extends Rectangle2D {
    public int x, y;
    public int width, height;
}

public class Rectangle2D {
    public static class Double extends Rectangle2D {
        public double x, y;
        public double width, height;
    }

    public static class Float extends Rectangle2D {
        public float x, y;
        public float width, height;
    }
}
```

```
public class Point extends Point2D {
    public int x, y;
}

public class Point2D {
    public static class Double extends Point2D {
        public double x, y;
    }

    public static class Float extends Point2D {
        public float x, y;
    }
}
```

Figure 10.18: Definition and representation redundancy caused by optimizations

source of information (Feigenbaum, 2008) and this makes us more confident about our results.

Our estimation of $Recall^{API}$ assumes that the identifiers are meaningful. If this was not the case we would not be able to estimate the number of occurences of a concept in the code ($CandCts$ would deliver completely unreliable results) and thereby we would obtain a very high (but very erroneous) value for $Recall^{API}$.

- When the identifiers are ambiguous (i. e. exhibit synonymy or polysemy) then the "reliable naming conventions" do not work well. The overloading degree (OD) metric is strongly affected. However, we interpret the results of these two metrics only qualitatively – i. e. primitive types have a high overloading degree – and do not emphasize on the quantitative aspects of OD.

- When the identifiers are meaningful (i. e. $CandNms(i) \neq \emptyset$) but misleading (Definition 8.2.3) we would obtain completely unreliable results. All our analyses are based on the fact that the identifiers are 'fairly' chosen. More simply said, the programmers do not systematically choose misleading names – e. g. it never happens to have a class named "window", one of its subclasses named "dialog" and these classes to refer to geometrical figures (e. g. "shape" and "square").

Quality of the used ontologies. Most of the results of our analyses depend strongly on the used ontologies. Since we consider that ontologies represent the semantic domain with respect to which the programs are interpreted, every analysis that uses an ontology is (at most) as reliable as the ontology itself. While WordNet is a widely used ontology and it was validated in different applications by large communities of users in more than fifteen years, the other ontologies (fragments) that we extracted from APIs or we manually built are far less validated and therefore open to debate. However, whenever possible, we validated the results that we

presented in these case studies by using external sources of information. This gives us confidence that the conceptual biases in the ontologies that we used in our analyses did not significantly distort our results.

External Validity

The "external validity" represents the possibility to generalize the causal inferences (relations) identified in a certain study to other studies (Trochim, 2006). Our case studies deal with well-known (pieces of) APIs, a small number of systems and with a small number of ontologies. Therefore, can the results that we presented be generalized for arbitrary programs?

Using adequate ontologies. The measure in which we can recover $\overleftrightarrow{Ref}$ for a certain program depends strongly on the used ontology. As we presented in Section 10.4.1 (mapping WordNet on programs), by using an inadequate ontology we can find only a few concepts and give the intentional meaning of few program elements (ca. 1%). In a similar manner, trying to analyze let's say an operating system kernel by mapping it to our GUI ontology would be highly inappropriate and the concepts location algorithm would identify only noise (if anything).

Type of the program. We are aware that the intentional analyses depend strongly on the type of the code that is analyzed. Our basic research hypothesis is that beside a purely mathematical view in which programs are interpreted as mathematical terms, programs can be interpreted from the perspective of their modeled domain. In many situations this assumption is too strong – e. g. in the case of generated code, highly optimized programs, (part of) embedded software, this hypothesis does not hold. The latter kinds of programs might not be (easily) interpretable from the point of view of the domain, namely, we cannot find a clear correspondence between the program elements and the domain concepts that they implement since the domain concepts are highly diffused in the code. In these cases program structure does not mirror the domain but the program is rather a "computational soup" out of which the domain behavior (somehow) emerges.

Programming style and programming language. All our case studies about intentional analysis were performed on Java programs. While we are confident that the results can be generalized to programs written in other object-oriented languages, making any claims about programs written in other paradigms (e. g. functional, logic, data-flow languages) or in scripting languages (e. g. 4GL) is impossible. Furthermore, the same limitation is applied to Java programs that are written using a non object-oriented programming style (e. g. a functional style).

10.7 Summary

In this chapter we presented our experience with applying intentional analyses to several Java systems. Our investigations can be classified in three main categories: the existence of intentional meaning, the level of automation that can be achieved by the intentional analyses, and the

feasibility and precision of the intentional analyses. Below we summarize the most important results of our experiments:

- The analyzed programs do exhibit intentional meaning. Therefore, by considering them from the perspective of the domain knowledge they implement, we could apply intentional program analyses.
- Different domain specific APIs that address the same domain contain enough common knowledge to enable the extraction of ontology fragments. Furthermore, it is feasible to extract domain knowledge from such APIs by investing a relative small manual effort.
- Ontology fragments automatically extracted from domain specific APIs can be used for performing analyses of the conceptual coverage and logical redundancy. When mapped on programs, they can be used to recover the references of domain concepts in the code.
- The WordNet ontology contains too general knowledge to be useful for coverage analyses.

Limitations of the concepts location algorithm. Even if the concepts location algorithm could identify the function $\overrightarrow{Ref}$ for a considerable number of program elements and was helpful in evaluating the domain coverage of APIs, there are many program elements that could not be mapped to any concept and the algorithm produces (quite) many false positives. Below we present the most important reasons for these facts:

1. **Diffusion.** Our algorithm is based on the similarities between names of concepts and identifiers and on the similarities between the structure of the ontology and of the program. A major limitation of our algorithm is that we currently do not interpret the composition of concepts at the level of identifiers (e. g. the name of the method `findFigure` could be interpreted as the conceptual triple FIND – actsOn– FIGURE).
2. **Knowledge base limitations.** Even if the knowledge bases used contain a big number of concepts, there are many more domain concepts not contained in our knowledge bases and therefore they are not recoverable. Furthermore, between our concepts from the used ontologies are a relatively small number of relations. Therefore, the structures of the ontology and that of the code are not similar and our algorithm does not recover the concepts.
3. **Representation of the knowledge as triples.** Since a "concept – relation – concept" triple represents the basic unit of knowledge, and only these triples are used for the identification of concepts, we obtain a (relatively) high number of false positives.

Part V

Closing

11 Summary and Future Work

In Section 11.1 we present a summary of our work and our most important contributions in context. In Section 11.2 we suggest several directions for extending this research.

11.1 Summary

Intentionality in software engineering. The software engineering process can be seen as a quest for closing the conceptual gap between the knowledge of the domain to which the software is addressed and the implementation languages. Due to the differences between the conceptualizations of the application domain and of the implementation languages, during forward engineering occurs a loss of abstract information about the (intended) relation between software artefacts and domain knowledge (Section 2.2). Researchers from different software engineering sub-branches (from specification to programming) fight with the loss of intentionality that occurs in the forward engineering (Section 2.3). At the level of programs, the intentionality loss is amplified by the interleaving of multiple dimensions of knowledge and delocalization of the implementation of concepts. Reverse engineering and program understanding have in center the recovery of abstract information from the code. Many approaches are focused on the recovery of the relations between programs and their business domain (Section 2.4). We showed that the many approaches for program understanding aim at recovering the lost intentionality by identifying domain concepts in the code. However, when they tackle the concepts related to the business domain (as opposed to for example, design), they suffer from the lack of precision and structuring in the definition of concepts and of their mappings to programs.

Intentional meaning of programs. In this dissertation, we worked out a new kind of interpretation of programs that is defined in terms of the domain concepts that programs implement. The meaning of these concepts is given by domain ontologies they are part of (Section 3.2). With other words, domain ontologies represent the semantic domain with respect to which programs (or parts thereof) are interpreted. We defined the intentional implementation ($\overleftarrow{i}$) and intentional interpretation ($\overrightarrow{i}$) functions (Section 3.3) that link concepts from domain ontologies to named program entities (i. e. classes, methods, variables). Using these explicit links we bridge the gap between the domain knowledge and the code, and thereby we can characterize faithfulness in the implementation of domain knowledge in the code.

Operationalization. In order to operationalize our approach we defined abstractions for both programs and domain ontologies as labeled graphs (Section 3.4). In the case of programs, the nodes represent the named program entities and the edges are program relations between them. In the case of ontologies, the nodes represent concepts and the edges represent conceptual relations. We defined a unified meta-model that contains program entities and their corresponding concepts that they implement. Instantiating this meta-model represents the starting point for our

automatic analyses. We further refined the implementation and interpretation functions ($\overleftrightarrow{i}$) into more specific components: reference of concepts ($\overleftrightarrow{Ref}$), definition of concepts ($\overleftrightarrow{Def}$), and representation of concepts ($\overleftrightarrow{Rep}$) (Section 3.5).

Intentional analyses of programs. We use the intentional meaning to define new program analyses about the coverage, consistency, explicitness, and conciseness (Sections 4.2 – 4.5) of implementation of domain knowledge in the code. All these analyses are expressed by measuring the level of isomorphism between the programs and the domain ontology (both represented as graphs). We investigate the problems caused by typical implementation mismatches (Sections 5.2 – 5.5) and we present code examples from the Java standard API where such problems occur. We use our formal framework also for characterizing the quality of identifiers from the point of view of their meaningfulness and ambiguity (Section 8.2 and 8.3).

Conceptual adequacy of Java. By analyzing the implementation strategies of the concepts from the Standard Upper Merged Ontology (SUMO) in Java, we identified typical cases that inherently lead to mismatches in reflection of domain knowledge in Java programs (Section 6.4). We showed that these mismatches are due to the conceptual differences between the Java and SUMO, and are (mostly) not avoidable in the practice. We draw the conclusion that whenever we want to implement certain phenomena from the business domain in Java, we introduce a conceptual bias in the implementation with respect to the business domain (no matter for example how good the design is).

Automatic recovery of intentional meaning. In order to automate the intentional analyses, we develop a method to recover the intentional meaning by using the similarities between the identifier names and the names of the concepts that they implement, and a pre-defined set of implementation strategies for typical categories of concepts and conceptual relations. We represent both programs and ontologies as graphs and this enables us to (semi-)automatically recover the intentional meaning by mapping programs to ontologies using graph matching (Section 7.5).

Obtaining ontologies fit for intentional analyses. We discuss different sources of domain ontologies that are fit for performing intentional analyses (Sections 9.2 – 9.4) and we focus on automatic extraction of fragments of domain ontologies from domain specific APIs (Section 9.3). The ontology fragments obtained from the analysis of APIs offer a high conceptual coverage of the technical domains that are commonly used in programs (e. g. GUI, XML).

Experiments and relevance in the practice. Throughout this dissertation we presented examples of code fragments and discuss their deficiencies from the point of view of the implementation of domain concepts. Whenever possible, we presented examples of problems in the implementation of the domain knowledge in the Java standard APIs. In Chapter 10 we presented our experience with performing intentional analyses on several Java systems. The case studies that we performed had the following aims: to investigate the degree in which programs exhibit intentional meaning, the degree in which light-weighted ontologies can be used to represent the domain meaning, and the degree in which the intentional program analyses are automatable. The defects that we identified give us insides in the nature of programming today, in the manner in which domain concepts are reflected, represented and composed in the programs. With the help of our case studies, we showed that it is feasible to perform intentional analyses in the practice and they can be automated.

11.2 Future Work

This dissertation represents only a new step in the direction of using domain knowledge for analyzing programs. We are aware that there are many improvements to be done and the use of domain ontologies for performing intentional analyses of programs or different kinds of models (Section 11.2.1). Furthermore, our results suggest that the intentionality loss can be avoided only by raising the abstraction level at which the current development is performed (Section 11.2.2).

11.2.1 Intentional Analyses

Below we enumerate several directions of future work in the direction of recovering the intentional meaning and using it for enriching other program analyses with domain information.

Intended dynamics of programs. In this dissertation we did not consider the dynamic of programs. Our notion of "intentional meaning" is (mostly) structural. A next logical step is to extend the intentional meaning by taking into consideration the intended dynamic of the business domain. For example, in the banking domain "calculating the interest for some lend money" is also a domain concept, that in comparison with the "account" concept, has a dynamic nature. How can the intentional meaning be extended with intended dynamics is still an open question.

Instantiating the layers. While the identifier slicing and obtaining the words do not raise (almost) any problems, the concrete abstraction of programs (i. e. a concrete set of program relation types – Σ^{Π}), the abstraction of the domain (i. e. a concrete set of conceptual relation types – Σ^{Ω}) and the mapping strategies between them are far less clear and have more variability points. Depending on the domain under consideration and on the concrete language that is used to produce the content under analysis (e. g. programs, models), we can instantiate the programs layer, the conceptual layers, and the mapping functions $\overleftrightarrow{t_e}$ differently.

Intentional analysis of models described in other languages. The focus of this work was the analysis of Java programs. We can extend our framework towards other languages by choosing a different Σ^{Π} – e. g. we can use the same approach to analyze the intentionality of UML models and thereby Σ^{Π} will contain relation types like "aggregation", "composition" from the class diagrams part of UML, and "next activity" for the activity diagrams part of UML. By instantiating the Σ^{Ω} with different relation types we can describe other (more domain specific) conceptualizations like for example modeling security concerns.

Extending the set of intentional functions and describe other mismatches. We refined the intentional interpretation $\overleftrightarrow{i}$ into three mappings: the reference ($\overleftrightarrow{Ref}$), representation ($\overleftrightarrow{Rep}$), and definition ($\overleftrightarrow{Def}$) of concepts. We used these functions to describe concrete cases of mismatches between a domain ontology and the code and the consequences of these mismatches on different (maintenance) activities. The $\overleftrightarrow{i}$ can be further refined with other mappings between concepts and programs, that capture other aspects of the "implementation of domain concepts". Based on the new mappings, new mismatches between the domain and programs could be characterized.

Enriching program analysis with intentional meaning. Many of the currently wide-spreaded static analyses in reverse engineering (e.g. design quality assessment) are at a syntactical level. However, the proper interpretation of their results requires semantical information about the relation of program parts to the business domain. For example, if a class is reported as affected by a design flaw, the criticality of the flaw could be weighted by the *conceptual centrality* of that

class. Similarly, domain knowledge might lead to a better detection of architectural violations (e. g. GUI concepts that occur in the persistency layer). Furthermore, we can use the intentional meaning to compare the structural decomposition of a system with a conceptual decomposition of the business domain and thereby to better assess the modularity of the system.

Recovering the intentional interpretation. We presented a method to recover the $\overleftrightarrow{Ref}$ functions by using a graph matching algorithm. This algorithm can be improved in order to increase its precision and recall. Furthermore, we envision also other methods (manual or automatic) for recovering the mappings between the code and domain knowledge.

Building a large knowledge repository. The existence of domain knowledge in form of domain ontologies is a central prerequisite for the applicability of our approach on a larger scale. Having more knowledge, our algorithm could achieve a higher recall and a bigger coverage of the code. The work that we already started by building the programming technologies knowledge repository needs to be extended (ideally) by a community of users. More APIs that cover different domains can be analyzed and used as a source of domain knowledge. Furthermore, methods for ontology mining from text could be used to enrich the knowledge repository.

Integrating intentional analyses in the development process. In this thesis we presented a method to assign meaning to programs, and showed our experience with performing (reverse engineering like) program analyses. These analyses can be integrated in the development process and extended to other artefacts besides programs (e. g. design). Building fragments of domain ontologies while developing the code and linking them to the program could be an effective manner of documentation, or could guide the design of the application.

Intentional "Integrated Development Environments" (IDEs). A semantically enhanced IDE can "look over the shoulder" of programmers and can help in performing different programming or maintenance tasks. For example, it could guide developers to browse the development artefacts based on the concepts they refer to, or it could serve as (active) documentation for APIs. A more visionary enhancement is to enable a kind of conceptual type checking – by "knowing" the intentional meaning of two variables the IDE might give warnings when one is assigned to the other in the case when the concepts that they represent are not compatible.

11.2.2 Domain Specific Modeling and Domain Specific Languages

The focus of this dissertation is programs analysis with emphasis on reverse engineering. However, *many parts of our work can be seen and understood as a case for domain specific languages and domain specific modeling.* On the one hand, we showed that the loss of intentionality cannot be avoided when there is a big conceptual gap between the languages which express the problem and those in which we express the solution. On the other hand, if programs were written in a domain specific language that is appropriate enough to their domain, the intentional meaning of those programs would be exactly the meaning that is defined based on the underlying (domain specific) language. In these cases the mismatches between the domain knowledge and programs would be minimized.

We strongly advocate that domain specific modeling and domain specific languages represent essential approaches to fight against the intentionality loss.

Bibliography

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA. 163

Albin-Amiot, H. and Guéhéneuc, Y.-G. (2001). Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of the 1st ECOOP Workshop on Automating Object-Oriented Software Development Methods*. 162

Anquetil, N. (2001). Characterizing the informal knowledge contained in systems. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, page 166, Washington, DC, USA. IEEE Computer Society. 40, 149, 164

Anquetil, N., de Oliveira, K. M., de Sousa, K. D., and Dias, M. G. B. (2007). Software maintenance seen as a knowledge management issue. *Information and Software Technology*, 49(5):515–529. 172

Anquetil, N., de Oliveira, K. M., Dias, M. G. B., Ramal, M., and de Moura Meneses, R. (2003). Knowledge for software maintenance. In *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*, pages 61–68. 47, 172, 179

Anquetil, N. and Lethbridge, T. (1998a). Extracting concepts from file names: a new file clustering criterion. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 84–93, Washington, DC, USA. IEEE Computer Society. 50, 149, 154

Anquetil, N. and Lethbridge, T. C. (1998b). Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '98)*, page 4. IBM Press. 148, 164, 167, 168, 212

Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., and Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983. 53, 58

Antoniol, G., Gael, Y., Merlo, E., and Tonella, P. (2007). Mining the lexicon used by programmers during sofware evolution. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07)*. 161

Antoniol, G. and Gueheneuc, Y.-G. (2006). Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641. 48

Arévalo, G., Ducasse, S., and Nierstrasz, O. (2005). Lessons learned in applying formal concept analysis to reverse engineering. In *Proceedings of 3rd International Conference on Formal Concept Analysis (ICFCA'05)*, pages 95–112. 51

Baniassad, E. L. A., Murphy, G. C., and Schwanninger, C. (2003). Design pattern rationale graphs: linking design to source. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 352–362, Washington, DC, USA. IEEE Computer Society. 53

Bastani, F. B. and Iyengar, S. S. (1987). The effect of data structures on the logical complexity of programs. *Communications of the ACM*, 30(3):250–259. 89, 150

Bierman, G. M. and Wren, A. (2005). First-class relationships in an object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag. 138

Biggerstaff, T. J., Mitbander, B. G., and Webster, D. (1993). The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering (ICSE '93)*, pages 482–498. IEEE CS Press. 49, 52, 86

Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E. (1994). Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82. 22, 45, 49, 161

Bjørner, D. (2006). *Software Engineering, vol 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science. Springer. 33

Brachman, R. J., Devanbu, P., Selfridge, P. G., Belanger, D., and Chen, Y. (1990). Toward a software information system. *AT & T Technical Journal*, 69(2):22–41. 41, 56

Broy, M. and Stolen, K. (2001). *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 33

Caprile, B. and Tonella, P. (1999). Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pages 112–122. 149, 154, 160

Carter, B. (1982). On choosing identifiers. *ACM SIGPLAN Notices*, 17(5):54–59. 148, 164

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17. 47

Clayton, R., Rugaber, S., Taylor, L., and Wills, L. (1997). A case study of domain-based program understanding. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, page 102, Washington, DC, USA. IEEE Computer Society. 53

Clayton, R., Rugaber, S., and Wills, L. (1998a). Dowsing: A tool framework for domain-oriented browsing of software artifacts. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE '98)*, page 204, Washington, DC, USA. IEEE Computer Society. 47

Clayton, R., Rugaber, S., and Wills, L. (1998b). On the knowledge required to understand a program. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, page 69, Washington, DC, USA. IEEE Computer Society. 40

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical Report 148, The University of Auckland. 148, 151

DeBaud, J.-M. (1996). Lessons from a domain-based reengineering effort. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 217, Washington, DC, USA. IEEE Computer Society. 54

DeBaud, J.-M., Moopen, B., and Rugaber, S. (1994). Domain analysis and reverse engineering. In *Proceedings of the International Conference on Software Maintenance (ICSM '94)*, pages 326–335. 47

Deissenboeck, F. and Pizka, M. (2006). Concise and consistent naming. *Software Quality Journal*, 14(3):261–282. 148, 149, 150, 161, 164, 168

Deissenboeck, F. and Ratiu, D. (2006). A unified meta-model for concept-based reverse engineering. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies (ATEM'06)*. Johannes Gutenberg-Univ. Mainz. 26, 152

Devanbu, P. T., Brachman, R. J., Selfridge, P. G., and Ballard, B. W. (1990). LaSSIE: a knowledge-based software information system. In *Proceedings of the 12th International Conference on Software Engineering (ICSE '90)*, pages 249–261, Los Alamitos, CA, USA. IEEE Computer Society Press. 53, 57, 58, 60

Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224. 51

Evermann, J. and Wand, Y. (2005). Toward formalizing domain modeling semantics in language syntax. *IEEE Transactions on Software Engineering*, 31(1):21–37. 35

Feigenbaum, B. (2008). SWT, Swing or AWT: Which is right for you. 208, 210, 215

Feild, H., Binkley, D., and Lawrie, D. (2006). Identifier splitting: A study of two techniques. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*. 149, 154, 160

Felleisen, M. (1991). On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75. 35

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2001). *How to design programs: an introduction to programming and computing*. MIT Press. 45, 46

Finkelstein, A. and Kramer, J. (2000). Software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 3–22, New York, NY, USA. ACM Press. 19

Florijn, G., Meijers, M., and van Winsen, P. (1997). Tool support for object-oriented patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, pages 472–495. 74, 127

Fowler, M. (2002). Public versus published interfaces. *IEEE Software*, 19(2):18–19. 71

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE '07)*, pages 37–54, Washington, DC, USA. IEEE Computer Society. 19, 34

Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. (1987). The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971. 149

Gall, H., Klösch, R., and Mittermeir, R. T. (1996). Using domain knowledge to improve reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 6(3):477–505. 54

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 44, 69, 138

Gehlert, A. and Esswein, W. (2007). Toward a formal research framework for ontological analyses. *Advanced Engineering Informatics*, 21(2):119–131. 38, 105, 124, 130

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal Human-Computer Studies*, 43(5-6):907–928. 65, 66, 67, 89

Guarino, N. and Giaretta, P. (1995). Ontologies and knowledge bases: Towards a terminological clarification. *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing*, pages 25 – 32. 65, 66

Guéhéneuc, Y.-G. and Albin-Amiot, H. (2004). Recovering binary class relationships: putting icing on the UML cake. In *Proceedings of the 19th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 301–314, New York, NY, USA. ACM. 127

Guizzardi, G. (2005). *Ontological Foundations for Structural Conceptual Models*. PhD thesis, University of Twente, The Netherlands. 34, 35, 37, 65

Gunter, C., Mitchell, J., and Notkin, D. (1996). Strategic directions in software engineering and programming languages. *ACM Computing Surveys*, 28(4):727–737. 37, 89

Harandi, M. T. and Ning, J. Q. (1990). Knowledge-based program analysis. *IEEE Software*, 7(1):74–81. 30, 31, 48, 52, 53

Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72. 64

Hruby, P. (2005). Ontology-based domain-driven design. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, San Diego, CA, USA. 37, 38

Hsi, I., Potts, C., and Moore, M. (2003). Ontological excavation: Unearthing the core concepts of the application. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'03)*, volume 0, page 345, Los Alamitos, CA, USA. IEEE Computer Society. 53, 128, 182

Jackson, M. (1995). *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. 31

Kapser, C. and Godfrey, M. W. (2006). "Cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 19–28, Washington, DC, USA. IEEE Computer Society. 105

Koschke, R. (1999). *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institut für Informatik, Universität Stuttgart. 39

Krishnamurthi, S. and Felleisen, M. (1998). Toward a formal theory of extensible software. *SIGSOFT Software Engineering Notes*, 23(6):88–98. 115, 116

Krogstie, J. (2000). Evaluating UML: a practical application of a framework for the understanding of quality in requirements specifications and conceptual modelling. In *Norwegian Informatics Conference (NIK)*. 35, 37

Kuhn, A., Ducasse, S., and Girba, T. (2005). Enriching reverse engineering with semantic clustering. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE '05)*, pages 133–142, Washington, DC, USA. IEEE Computer Society. 50

Lawrie, D., Feild, H., and Binkley, D. (2007). An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution*, 19(4):205–229. 161

Letovsky, S. and Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49. 20, 41

Liskov, B. (1987). Keynote address - data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 17–34, New York, NY, USA. ACM. 37

Liskov, B. and Zilles, S. (1975). Specification techniques for data abstractions. In *Proceedings of the International Conference on Reliable Software*, pages 72–87, New York, NY, USA. ACM Press. 43

Maletic, J. I. and Marcus, A. (2001). Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 103–112, Washington, DC, USA. IEEE Computer Society. 50, 161

Marcus, A. (2003). *Semantic-driven program analysis*. PhD thesis, Kent State University, Kent, OH, USA. Director-Jonathan I. Maletic. 58, 161

Marcus, A. and Maletic, J. I. (2001). Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE '01)*, page 107, Washington, DC, USA. IEEE Computer Society. 127

Martin, R. C. (1996). The Liskov Substitution Principle. *C++ Report*, 8. 37

Masolo, C., Borgo, S., Gangemi, A., Guarino, N., and Oltramari, A. (2003). WonderWeb Deliverable D18 - Ontology Library. Technical report, University of Trento. 131

McGuinness, D. L. (2003). Ontologies come of age. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential, D. Fensel et al., eds.*, pages 171–194. MIT Press. 21, 66

Mens, K. (2002). *Automating architectural conformance checking by means of logic meta programming*. PhD thesis, Universiteit Brussel. 56

Mens, K., Poll, B., and González, S. (2003). Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM '03)*, page 169, Washington, DC, USA. IEEE Computer Society. 56

Meyer, B. (2000). *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition. 19

Michail, A. and Notkin, D. (1999). Assessing software libraries by browsing similar classes, functions and relationships. In *Proceedings of the 21st International Conference on Software engineering (ICSE '99)*, pages 463–472, Los Alamitos, CA, USA. IEEE Computer Society Press. 182

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. J. (1990). Introduction to WordNet: An on-line lexical database. *Intternational Journal of Lexicography*, 3(4):235–244. 171

Mitchell, J. C. (1993). On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141–163. 35

Murphy, G. C., Notkin, D., and Sullivan, K. (1995). Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd Symposium on Foundations of Software Engineering (FSE'95)*, pages 18–28, New York, NY, USA. ACM. 52, 54, 55, 56

Mylopoulos, J. (1992). Conceptual modeling and telos. In P. Loucopoulos, R. Z., editor, *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*. John Wiley and Sons Inc. 66

NASA (1999). Mars climate orbiter - mishap investigation board phase I report. Technical report, NASA. 122

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, 18:357–377. 32

Niles, I. and Pease, A. (2001a). Origins of the IEEE Standard Upper Ontology. In *Working Notes of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology*, pages 37–42, Seattle. 131

Niles, I. and Pease, A. (2001b). Towards a standard upper ontology. In *Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS '01)*, pages 2–9, New York, NY, USA. ACM. 84, 107, 129, 130, 131

Noy, N. F. and Mcguinness, D. (2000). Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford University. 180

Opdahl, A. and Henderson-Sellers, B. (2002). Ontological evaluation of the uml using the Bunge-Wand-Weber model. *Software and Systems Modeling Journal*, 1:43 – 67. 143, 144

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341. 150

Petrenko, M., Rajlich, V., and Vanciu, R. (2008). Partial domain comprehension in software evolution and maintenance. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*, volume 0, pages 13–22, Los Alamitos, CA, USA. IEEE Computer Society. 183

Porter, M. F. (1997). *An algorithm for suffix stripping*, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 154

Rajlich, V. and Wilde, N. (2002). The role of concepts in program comprehension. In *Proceedings of the International Workshop on Program Comprehension (IWPC'02)*. IEEE CS Press. 20, 22, 49, 50, 86, 87, 167

Ramal, M. F. N., de Moura Meneses, R., and Anquetil, N. (2002). A disturbing result on the knowledge used during software maintenance. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, pages 277–. 47, 49, 179

Ratiu, D. and Deissenboeck, F. (2006a). How programs represent reality (and how they don't). In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 83–92. IEEE Computer Society Press. 26

Ratiu, D. and Deissenboeck, F. (2006b). Programs are knowledge bases. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC '06)*, pages 79–83. IEEE CS Press. 26

Ratiu, D. and Deissenboeck, F. (2007). From reality to programs and (not quite) back again. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC '07)*. 26

Ratiu, D., Feilkas, M., Deissenboeck, F., Juerjens, J., and Marinescu, R. (2008a). Towards a repository of common programming technologies knowledge. In *Proceedings of the International Workshop on Semantic Technologies in System Maintenance (STSM'08)*. 26, 179

Ratiu, D. and Juerjens, J. (2007). The reality of libraries. In *Proceesings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE CS Press. 26

Ratiu, D. and Juerjens, J. (2008). Evaluating the reference and representation of domain concepts in APIs. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*. IEEE CS. 26

Ratiu, D., Juerjens, J., and Feilkas, M. (2008b). Extracting domain ontologies from domain specific APIs. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*. IEEE CS. 26

Robillard, M. P. (2003). *Representing Concerns in Source Code*. PhD thesis, University of British Columbia. 48, 55

Robillard, M. P. and Murphy, G. C. (2002). Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 406–416, New York, NY, USA. ACM. 55

Robillard, M. P. and Murphy, G. C. (2007). Representing concerns in source code. *ACM Transactions on Software Engineering Methodology*, 16(1):3. 48

Rozen, S. and Shasha, D. (1989). Using a relational system on Wall Street: the good, the bad, the ugly, and the ideal. *Communications of the ACM*, 32(8):988–994. 35, 36

Rugaber, S. (2000). The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-4):143–192. 47, 121

Rugaber, S., Stirewalt, K., and Wills, L. M. (1995). The interleaving problem in program understanding. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'95)*. 20, 40

Rumbaugh, J. (1987). Relations as semantic constructs in an object-oriented language. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'87)*. ACM Press. 91, 138

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 53

Sabou, M. (2004). From software APIs to web service ontologies: A semi-automatic extraction method. In *International Semantic Web Conference (ISWC'04)*, pages 410–424. 182

Schreiner, A.-T. (1994). *Objekt-orientierte Programmierung mit ANSI-C*. Hanser, Munich. 96

Selfridge, P., Waters, R., and Chikofsky, E. (1993). Challenges to the field of reverse engineering. In *Proceedings of Working Conference on Reverse Engineering (WCRE'93)*, pages 144–150. 48

Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25. 35

Sillito, J., Murphy, G. C., and Volder, K. D. (2006). Questions programmers ask during software evolution tasks. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering (FSE'06)*, pages 23–34, New York, NY, USA. ACM Press. 48

Sneed, H. M. (1996). Object-oriented cobol recycling. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 169, Washington, DC, USA. IEEE Computer Society. 149, 164

Soloway, E. and Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609. 48, 53

Sowa, J. F. (2000). *Knowledge representation: logical, philosophical and computational foundations*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA. 65, 131

Spivey, J. M. (1992). *The Z Notation: A reference manual*. Prentice Hall International Series in Computer Science, 2nd edition edition. 33

Stroustrup, B. (1988). What is object-oriented programming? *IEEE Software*, 05(3):10–20. 39

Sun (1997). JavaBeans API specification. Technical report, Sun Microsystems. 135

Tarr, P., Ossher, H., Harrison, W., and Stanley M. Sutton, J. (1999). N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107–119, Los Alamitos, CA, USA. IEEE Computer Society Press. 20, 40, 41, 150

Tilley, T., Cole, R., Becker, P., and Eklund, P. (2003). A survey of formal concept analysis support for software engineering activities. In *Proceedings of the First International Conference on Formal Concept Analysis (ICFCA'03)*. Springer-Verlag. 51

Trochim, W. M. (2006). *The Research Methods Knowledge Base*. Atomic Dog Publishing. `http://www.socialresearchmethods.net/kb`. 214, 216

Tsantalis, N. and Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909. 162

Ullmann, S. (1972). *Semantics : an introduction to the science of meaning*. Blackwell, Oxford. 34

Uschold, M. (2003). Where are the semantics in the semantic web? *AI Magazine*, 24(3):25–36. 65

van Lamsweerde, A. (2000). Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering (ICSE '00)*, pages 147–159, New York, NY, USA. ACM Press. 19, 35, 43, 44

Visser, P. R. S., Jones, D. M., Bench-Capon, T. J. M., and Shave, M. J. R. (1998). Assessing heterogeneity by classifying ontology mismatches. In *Proceedings of Formal Ontology in Information Systems (FOIS'98)*, pages 148–162. IOS Press. 177

Wand, Y. and Weber, R. (1993). On the ontological expressiveness of information systems analysis and design grammars. *Journal of Information Systems*, 3:217–237. 37, 143

Weissman, L. (1974). Psychological complexity of computer programs: an experimental methodology. *SIGPLAN Notes*, 9(6):25–36. 44

Welty, C. A. (1995). *An Integrated Representation for Software Development and Discovery*. PhD thesis, Rensselaer Polytechnic Institute. 57

Welty, C. A. (2003). Software engineering. In Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors, *Description Logic Handbook*, pages 373–387, New York, NY, USA. Cambridge University Press. 57, 58

Wilde, N. and Scully, M. C. (1995). Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62. 48

Wills, L. M. (1993). Flexible control for program recognition. In *Proceedings of the First Working Conference on Reverse Engineering (WCRE'93)*, pages 134–143. 53

Witte, R., Zhang, Y., and Rilling, J. (2007). Empowering software maintainers with semantic web technologies. In *Proceedings of the European Semantic Web Conference (ESWC'07)*, pages 37–52. 53, 59

Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. (1981). The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*, pages 215–223, Piscataway, NJ, USA. IEEE Press. 150

Yang, H., Cui, Z., and O'Brien, P. (1999). Extracting ontologies from legacy systems for understanding and re-engineering. In *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC '99)*, page 21, Washington, DC, USA. IEEE Computer Society. 182

Yu, C.-C. and Robertson, S. P. (1988). Plan-based representations of Pascal and Fortran code. In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 251–256, New York, NY, USA. ACM. 48

Zhang, Y., Witte, R., Rilling, J., and Haarslev, V. (2006). An ontology-based approach for traceability recovery. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM '06)*. 59

A Notations

Π	program layer (p. 81)
Π^{Prog}	program layer corresponding to the entire program (p. 81)
Π^{API}	program layer corresponding to an API (p. 82)
P	the set of program elements (p. 81)
p	a particular program element (an element of P)
Σ^{Π}	the set of relation types from the program layer (p. 81)
σ^{Π}	a relation type from the program layer (an element of Σ^{Π})
$e^{\Pi}(p_1, p_2)$	the type of the relation between program elements p_1 and p_2 (p. 81)
$\sigma^{\Pi}(p_1, p_2)$	between program elements p_1 and p_2 is a relation whose type is σ^{Π} ($e^{\Pi}(p_1, p_2) = \sigma^{\Pi}$)
Ω	conceptual layer (p. 83)
C	the set of concepts (p. 83)
c	a particular concept (an element of C)
Σ^{Ω}	the set of relation types from the conceptual layer (p. 83)
σ^{Ω}	a relation type from the conceptual layer (an element of Σ^{Ω})
$e^{\Omega}(c_1, c_2)$	the type of the relation between concepts c_1 and c_2 (p. 83)
$\sigma^{\Omega}(c_1, c_2)$	between concepts c_1 and c_2 is an edge whose type is σ^{Ω} ($e^{\Omega}(c_1, c_2) = \sigma^{\Omega}$)
$\overrightarrow{i}$	intentional interpretation (p. 72)
$\overrightarrow{i}(p)$	the set of concepts representing the interpretation of the program element p
$\overleftarrow{i}$	intentional implementation (p. 72)
$\overleftarrow{i}(c)$	the set of program elements representing the implementation of the concept c
$\overleftrightarrow{i}$	implementation and interpretation functions
$\overrightarrow{t}$	interpretation of relations (p. 90)
$\overrightarrow{t}(\sigma^{\Pi})$	the set of conceptual relations representing the interpretation of the program relation σ^{Π}
$\overleftarrow{t}$	implementation of relations (p. 90)
$\overleftarrow{t}(\sigma^{\Omega})$	the set of program relations representing the implementation of the conceptual relation σ^{Ω}
$\overleftrightarrow{t}$	implementation and interpretation of relations functions
$\overrightarrow{t_e}$	extended interpretation of relations (p. 158)
$\overrightarrow{t_e}(\langle\sigma_i^{\Pi}, ..., \sigma_j^{\Pi}\rangle)$	the set of sequences of conceptual relations representing the interpretation of the sequence of program relations $\langle\sigma_i^{\Pi}, ..., \sigma_j^{\Pi}\rangle$
$\overleftarrow{t_e}$	extended implementation of relations (p. 158)
$\overleftarrow{t_e}(\langle\sigma_i^{\Omega}, ..., \sigma_j^{\Omega}\rangle)$	the set of sequences of program relations representing the implementation of the sequence

	of conceptual relations $\langle \sigma_i^{\Omega}, ..., \sigma_j^{\Omega} \rangle$
$\overleftrightarrow{t_e}$	extended implementation and interpretation of relations functions
$\overrightarrow{Ref}(p)$	the set of concepts referred by the program element p (p. 85)
$\overleftarrow{Ref}(c)$	the set of program elements that reffer the concept c (p. 85)
$\overleftrightarrow{Ref}$	reference functions
$\overrightarrow{Def}(p)$	the set of concepts defined by the program element p (p. 87)
$\overleftarrow{Def}(c)$	the set of program elements defined by the concept c (p. 87)
$\overleftrightarrow{Def}$	definition functions
$\overrightarrow{Rep}(p)$	the set of concepts represented by the program element p (p. 88)
$\overleftarrow{Rep}(c)$	the set of program elements that represent the concept c (p. 88)
$\overleftrightarrow{Rep}$	representation functions
Λ	lexical layer (p. 153)
N	the set of names of concepts
I	the set of names of program identifiers
W	the set of lexically normalized words
$N_2W(n)$	the set of lexically normalized words corresponding to name n
$I_2W(i)$	the set of lexically normalized words corresponding to identifier i
$CandNms$	candidate names (p. 156)
$CandCts$	candidate concepts (p. 156)
$\sim$	relation of equivalence between two paths from the program graph (p. 176)

Index