In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

Chapter 12
BIM programming

Julian Amann, Cornelius Preidel, Eike Tauscher, André Borrmann

Abstract This chapter describes different possibilities for programming BIM applications with particular emphasis on processing data in the vendor-neutral Industry Foundation Classes (IFC) exchange format. It describes how to access data in STEP clear text encoding and discusses the differences between early and late binding. Given the increasingly important role of ifcXML in the exchange of IFC data, the chapter also examines different access variants such as SAX (Simple API for XML) and DOM (Document Object Model), and discusses the different geometry representations of IFC and their interpretation. Furthermore, the chapter gives a brief overview of the development of add-ins as a means of allowing existing software to be adapted to user-specific needs. The chapter ends with a brief overview of cloud-based platforms and a short introduction to visual programming.

12.1 Introduction

As described in earlier chapters, a wide range of different software products have been developed to serve specific tasks in the construction industry, with new software tools emerging all the time. To make efficient use of these tools in the value-added chain, data exchange at a high semantic level is paramount. Today, this is increasingly achieved using open data formats such as the Industry Foundation Classes (IFC) (see Chap. 6). To use information contained in an IFC instance file,

Julian Amann · Cornelius Preidel · André Borrmann
Technical University of Munich, Chair of Computational Modeling and Simulation, Arcisstraße 21, 80333 Munich, Germany
e-mail: julian.amann@tum.de, cornelius.preidel@tum.de, andre.borrmann@tum.de

Eike Tauscher
Bauhaus-Universität Weimar, Chair of Computing in Civil Engineering, Coudraystraße 7, 99421 Weimar, Germany
e-mail: eike.tauscher@uni-weimar.de

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

it needs to be accessed using the respective programming language. This chapter outlines the different methods and practices.

12.2 Procedures for accessing data in the STEP format

The most commonly used format for the storage of IFC instance file is STEP Clear Text Encoding (ISO 10303-21, 2016), also known as STEP P21. Techniques for reading and writing STEP files can be categorized into two key approaches: early binding and late binding.

12.2.1 Early binding

With the early binding approach, the entities of the EXPRESS schema in the STEP P21 file are mapped to the target programming language (host language) using a suitable mapping method. Early binding make it possible to map the STEP file to entities of the host language, i.e. to read a STEP file, and subsequently to convert the host entities back into a STEP file, i.e. to write a STEP file. While it is theoretically possible to implement early binding manually, it is not recommended for the IFC data model due to the large number of entities (several hundred) and accompanying risk of introducing programming errors through manual implementation.

As a rule, a code generator is used that takes an EXPRESS schema as input and produces entities (e.g. classes) of the host language as output. This mapping and the associated code generation need only be performed once for a given EXPRESS schema, and need only be repeated if the underlying EXPRESS schema changes. In the case of the IFC, this is comparatively rare and when changes are made, a new version number for the corresponding EXPRESS schema is issued (e.g. IFC4, IFC2x3 TC1, IFC2x3, IFC2x2, etc.).

From a technical viewpoint, if several different IFC schema versions need to be supported in parallel, each version requires its own separate early binding. Figure 12.1 shows an overview of the early binding process. The code generator generates a corresponding mapping for each entity in the target programming language, e.g. for the C++ programming language, a C++ class called `IfcAddress` is generated for the EXPRESS entity `IfcAddress`. There are no standardized rules for mapping EXPRESS entities to a programming language, and the developer of the code generator therefore has a free hand. In object-oriented programming languages, EXPRESS entities are typically mapped to classes, inheritance is implemented with the inheritance syntax, and references are realized with pointers, smart pointers (pointers that deal with memory management) or references in the target programming language.

A code generator needs to be able to parse the EXPRESS grammar by means of a lexer that generates tokens from the input symbols of the STEP file, processes

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

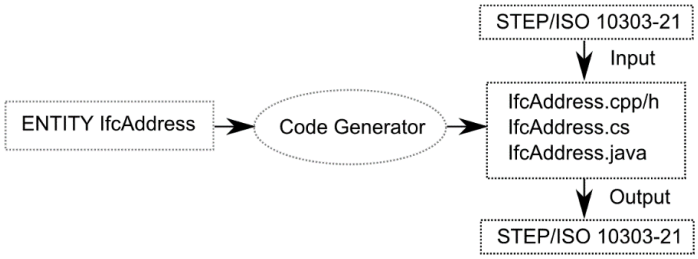

Fig. 12.1 Scheme of an early binding. For each entity, a corresponding class is created for the target programming language.

them using a parser to create a syntax tree and then validates the syntax of the respective EXPRESS schema for correctness. The code generator should ideally be able to produce a valid mapping for the target language in one step from the EXPRESS schema without any manual intervention. In practice, however, not all code generators are able to convert any valid EXPRESS schema and may need a pre-processing step or additional manual effort up front. IfcOpenShell[1] is an example of a code generator for the target programming language C++, while the JSDAI library[2] can be used for the programming language Java.

The following listing shows the use of the TUM Open Infra Platform Early Binding EXPRESS generator[3] with the IFC4 schema:

```
// create a model
ifc_model = shared_ptr<Ifc4Model>(new Ifc4Model());
// ...

// create a point with the coordinates (9,10)
shared_ptr<IfcCartesianPoint> pnt =
    std::make_shared<IfcCartesianPoint>(id++);
ifc_model->insertEntity(pnt); // add point to model
// set coordinates of point
pnt->m_Coordinates.push_back(
    std::make_shared<IfcLengthMeasure>(9.0)
);
pnt->m_Coordinates.push_back(
    std::make_shared<IfcLengthMeasure>(10.0)
);
// ...

// write a STEP P21 file
shared_ptr<IfcStepWriter> step_writer =
    std::make_shared<IfcStepWriter>()
std::stringstream stream;
stream.precision(20);
```

[1] https://github.com/IfcOpenShell/IfcOpenShell/tree/master/src/ifcexpressparser

[2] http://www.jsdai.net

[3] https://bitbucket.org/tumcms/oipexpress

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3


```
step_writer->writeStream(stream, ifc_model);
std::ofstream myFile("MyFile.ifc");
myFile<<stream.str().c_str();
```

The program creates an instance of the entity `IfcCartesianPoint` with the coordinates (9.0, 10.0). Subsequently, an `IfcStepWriter` object is created, which is used to convert the generated model (`ifc_model`) to a STEP P21 file.

12.2.2 Late Binding

In contrast to early binding, late binding uses a fixed interface called the standard data access interface (SDAI). The SDAI is an application programming interface (API) that provides a defined set of functions and methods to read and write STEP files, and is standardized in abstract form in the ISO 10303-22 standard (ISO 10303-22, 1998) . In addition, the STEP standard defines three different bindings for three different programming languages: Part 23 (ISO 10303-23, 2000) defines a C++ binding, part 24 a C binding, and part 27 a Java binding. Bindings for other software languages, such as C#, have been implemented by other software vendors which, while not standardized, are based on the standardized bindings.

The following examples use the C binding of the SDAI, but the principles are transferable to other bindings. In the C binding, all variable names, constants, alias types (`typedef`) and function names start with the prefix `sdai`. SDAI operations are always executed in the scope of an SDAI session. The data of a STEP file is stored in a SDAI model, which is also part of a SDAI repository. The following listing illustrates the use of the C-SDAI API:

```
SdaiSession session = sdaiOpenSession(); // open a new session

// open a new repository
SdaiRepo repository = sdaiOpenRepositoryBN(session,
                                           "MyFile.ifc");

// create a model
int ifcModelId = sdaiCreateModelBN(0, "MyModelName", "IFC4.exp");

// create a point with the coordinates (9,10)
int ifcCartesianPointId =
    sdaiCreateInstanceBN(ifcModelId, "IfcCartesianPoint");
int ifcCoordinatesId =
    sdaiCreateAggrBN(ifcCartesianPointId, "Coordinates");
sdaiAdd(ifcCoordinatesId, sdaiREAL, 9.0);
sdaiAdd(ifcCoordinatesId, sdaiREAL, 10.0);
sdaiSaveChanges(ifcModelId);
sdaiCloseRepository(repository);
sdaiCloseSession(session);
```

The above program, as before, produces an `IfcCartesianPoint` entity, which is stored in a STEP file.

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

Table 12.1 Overview of common STEP/IFC libraries.

Name	Language	License	STEP	IFC	Visualization	URL of website
IfcPlusPlus	C++	MIT	No	Yes	Yes	https://github.com/ifcquery/ifcplusplus
IfcOpenShell	C++/Python	OSGPL	No	Yes	Yes	http://ifcopenshell.org/
JSDAI	Java	AGPL v3	Yes	No	No	http://www.jsdai.net/
xBIM Toolkit	C#	CDDL	No	Yes	Yes	https://github.com/xBimTeam
IFC Tools Project	Java/C#	CC BY-NC 4.0 DE	Yes	Yes	Yes	http://www.ifctoolsproject.com
IFC Engine	C++/C#	proprietary	Yes	Yes	Yes	http://rdf.bg/ifc-engine-dll.php
STEPcode	C++/Python	BSD	Yes	Yes	No	http://stepcode.org/mw/index.php/STEPcode
ifc-dotnet	C#	BSD	No	Yes	No	https://code.google.com/p/ifc-dotnet/

While with early binding an equivalent for each entity of the EXPRESS schema is created in the host programming language, this intermediate code generation step is not necessary when using late binding. The late binding approach can therefore respond flexibly to changes in the EXPRESS schema. This is achieved using a generic approach that allows both the instantiation and access to entities based on the underlying EXPRESS schema during program runtime.

To accomplish this, however, the interface must frequently be called with strings used as parameters, which denote, for example, which entity should be created, which attribute should be read or which function should be executed. This requires in-depth knowledge of the underlying EXPRESS schema, not least because the automatic code completion functionality of modern development environments cannot help here. In addition, this is problematic from a programming perspective because, for example, syntax errors within such a string are not recognized by the compiler and thus only come to light when the program is run.

The handling of IFC files using the SDAI is much more difficult since the same entities with the same attributes, inheritance hierarchies and relations in the host language are not available as they are with early binding, and therefore cannot be checked during compiling to reliably exclude such errors. In theory, a key advantage of the SDAI is that an SDAI implementation from one vendor can be swapped with another, as they are standardized. In practice, however, this is not always as straightforward because some vendors integrate advanced SDAI functions into their APIs that are not part of the standard.

Table 12.1 shows a (non-exhaustive) overview of different libraries that can be used to read or write STEP and IFC files.

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

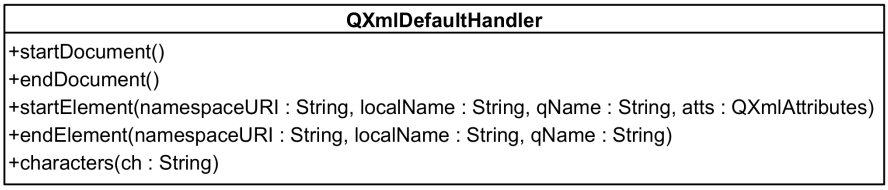

Fig. 12.2 Class diagram that shows a small part of the Qt SAX framework. The member methods of the class `QtXmlDefaultHandler` are incomplete.

12.3 Accessing XML encoded IFC data

In recent years, Extensible Markup Language (XML) has established itself as a standard and cross-industry approach for describing schemas and instance data. Both Microsoft's .NET framework and Java Standard Edition include an XML parser for handling XML files. There are numerous libraries for C++ for reading and writing XML files, for example, the Qt-libraries or the Xerces C++ XML parser, which is particularly suitable for very large XML files. In short, support for reading and creating XML documents or the availability of middleware for this task is much better for XML than for STEP.

Starting with version 4 of the IFC standard, an XML schema is also available as an equivalent to the EXPRESS schema. XML Schema Definition (XSD) is used as the description language. This defines the structure of XML instance files and allows them to be validated against the corresponding schema. Most major frameworks include XSD validators for this purpose.

Although from a programming standpoint, data access via XML is easier to implement using XML for the reasons mentioned above, it is currently far less widespread than its STEP counterpart. This can be attributed in part to the historical development of IFC, which was based on STEP and the data modeling language EXPRESS. A further reason is the size of ifcXML files, which are often multiple times larger than their STEP counterparts due to the XML tag syntax (see Chap. 6). ifcXML4 has improved on this through the definition of a more compact representation which may help the XML mapping of IFC data gain popularity in future. However, it must be noted that the XML schema of the IFC contains none of the inverse attributes, rules or functions included in the original IFC EXPRESS schema.

There are three commonly used approaches to reading and writing XML files: SAX, DOM and class generators.

SAX (Simple API for XML) was initially a Java library for sequentially reading XML documents. The software architecture of the original SAX implementation has become the de-facto standard and found its way into numerous other frameworks. Figure 12.2 shows a small part of the class `QtXmlDefaultHandler` of the Qt-SAX framework. Object-oriented programming languages usually offer a base classes or interface (cf. `QtXmlDefaultHandler`) which can be tailored by the developer via inheritance to serve a custom purpose. The SAX parser reads the XML document and invokes the appropriate method upon finding a specific XML element. For example,

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

parsing the root tag invokes the `startDocument` method, while the `endDocument` method is called at the end. XML elements are treated in a similar fashion, calling `startElement` or `endElement` at the start or end of the element respectively. The SAX parser is, however, only capable of verifying whether an XML file is valid or not while reading it.

Like SAX, the DOM (Document Object Model) is a common method for accessing XML and is likewise supported in multiple frameworks. The following listing shows the use of DOM in Qt:

```
QDomDocument doc; // create a DOM document
QDomProcessingInstruction header = // create XML header
    doc.createProcessingInstruction("xml",
                                    "version=\"1.0\"");
doc.appendChild(header); // Add XML header to DOM document
QDomElement root = doc.createElement("root");
root.setAttribute("version", getApplicationVersionString());
doc.appendChild(root);

// save entity
QDomElement xmlAlignments = doc.createElement("Alignments");
root.appendChild(xmlAlignments);

QFile file(filename.c_str()); // save XML file
file.open(QIODevice::WriteOnly))
QTextStream ts(&file);
ts << doc.toString();
```

The last approach is the equivalent of an early binding method for XML. Here too, multiple tools are available for generating a class hierarchy from an XML schema and providing read/write methods for the respective XML file. This approach has the same advantages and disadvantages as STEP-based early binding.

12.4 Interpretation of IFC geometry information

Alongside semantic information, geometric information plays an important role in IFC-based data exchange, due to the relevance of geometry in the design, construction and operation of buildings. It is essential that all software tools correctly interpret this data when visualizing or processing the geometric information contained in an IFC file. While most available geometry models support the export of geometric information into the IFC format (see Chap. 6), the provision of import functionalities is more complex because software systems need to support all geometric representation methods defined by the exchange requirements (see Chap. 7).

A large part of the IFC geometry descriptions is based on definitions in the ISO standard 10303-42 (ISO 10303-42, 2014). IFC version 4 supports the following approaches of geometry descriptions (see Chap. 2 for more details):

- *Constructive Solid Geometry (CSG)*: Solids formed by the result of Boolean operators – union, difference or intersection – on two or more solids.

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

- *Half-space solids*: Solids which are bounded on one side by a surface.
- *Extrusion bodies*: Solids produced by extrusion of a surface along a vector, one or more polylines, curves, splines or other mathematical functions.
- *Boundary Representation*: Solid bodies described by means of the surfaces delimiting them.
- *Tessellated objects*: Sets of triangulated surfaces.
- *Geometric groups*: Groups of geometric elements that do not have a topological structure, such as 2D or 3D points, lines, curves, surfaces.
- *Non-Uniform Rational B-Splines (NURBS)*: Representation of surfaces based on B-splines.

In the following, two of the above models are presented in more detail to illustrate the complexity of interpreting geometry.

Geometric models are divided into two primary categories: evaluated (or explicit) and non-evaluated (or implicit) models. Evaluated models are relatively easy to interpret since all geometric information relevant to the representation or further processing is explicitly available within the IFC data model. The Brep representation used in IFC can be taken as an example for such a model. All vertices of a body are already present with the correct coordinates and need no further calculation. The limiting surfaces (faces) result from the topological relationship between the vertices and edges.

Non-evaluated geometry models require the execution of sometimes complex geometric operations, because the implicit geometric information for representing an object must first be processed. An example of this is CSG modeling.

The basis of a CSG model, as described in detail in Chap. 2, is its so-called construction tree. It describes the construction history of the arising object, with the result of all Boolean operations at the root of the tree. The primitive bodies are located at the leaves of the tree, and are combined by the inner nodes of the tree using regularized Boolean set operations.

The calculations required to produce the final geometric bodies can be very complex. Different calculation models exist for different methods: for example, if the operands are provided as triangulated surface bodies, the calculation can be performed according to (Laidlaw et al., 1986) and (Hubbard, 1990). This method effectively checks all triangles of each operand against each other for intersection. If two triangles intersect, new triangles are formed by the cutting edge. The triangles of both operands are then classified depending on whether they are within the other body, outside the other body, on the surface of the other body with the same surface normal, or on the surface of the other body with opposing surface normal. After classification, the triangles are merged using the respective Boolean operator as shown in Table 12.2.

In this case, not only geometric primitives (spheres, cones, cylinders, etc.) can be found on the leaves, but also arbitrary complex solids. The only prerequisite is that it be a valid body, i.e. that two surfaces adjoin on each body edge. This procedure is often used in IFC models, for example to “cut” openings in walls or ceilings.

Table 12.3 shows a brief selection of different libraries that can convert different geometric representations into a triangle representation. In many cases, these

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

Table 12.2 Classification of triangles for area selection for the resulting body as a function of the Boolean operation (* = triangles with inverted orientation)

Operation	Triangles from A				Triangles from B			
	within B	outside of B	same normal	reverse normal	within A	outside of B	same normal	reverse normal
$A \cup B$	No	Yes	Yes	No	No	Yes	No	No
$A \cap B$	Yes	No	Yes	No	Yes	No	No	No
$A \setminus B$	No	Yes	No	Yes	Yes*	No	No	No

Table 12.3 A selection of libraries which can convert different geometrical representations into a triangle representation.

Name	Language	License	URL of website
OpenCASCADE	C++	LGPL	http://opencascade.org
Carve	C++	MIT	https://github.com/Vertexwahn/carve
csg.js	JavaScript	BSD	http://evanw.github.io/csg.js/
GTS	C++/Python	LGPL	http://gts.sourceforge.net

libraries are not limited to a certain programming language, because bindings are available that permit the use of other programming languages, such as Java or C#.

12.5 Add-in development for commercial BIM applications

Numerous software applications used for or in conjunction with Building Information Modeling provide a means of implementing add-ins and plugins to extend their functionality. Add-ins can typically be written in various programming languages such as C++ or languages within Microsoft's .Net framework (C#, VisualBasic.Net, J# etc.).

This section briefly outlines the development of a simple C# add-in for Autodesk Revit. Full documentation and details for programmers on developing a Revit add-in is available on the Autodesk website [4].

Microsoft Visual Studio can be used to program a C#-based Revit extension. Usually, the first step is to create a class library as a new project. If access to the Revit API is required, the class has to reference the corresponding Revit libraries (RevitAPI.dll and RevitAPIUI.dll). It is also important to use the correct target framework: the .Net version used in Visual Studio – e.g. 2.0, 3.0, 3.5, 3.5 Client Profile or 4 – must match that used by Revit (a common pitfall). The following is a minimal add-in for Revit:

[4] https://www.autodesk.com/

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3


```
using System;
using System.Collections.Generic;
using System.Linq;

using Autodesk.Revit.DB;
using Autodesk.Revit.DB.Architecture;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;

[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IexternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {
    UIApplication uiApp = commandData.Application;
    Document doc = uiApp.ActiveUIDocument.Document;

         // here you can use the revit API

    return Result.Succeeded;
  }
}
```

In addition to this library, an add-in XML manifest file must be created to define the global settings of the add-in and saved under `Autodesk\Revit\Addins\2014\`. Revit will then automatically load the add-in on start-up and it can be used via the toolbar. Further information can be found in the documentation.

12.6 Cloud-based platforms

Several companies offer so-called cloud-based platforms that enable developers to develop cloud-based applications based on these platforms. Table 12.4 gives some examples of such cloud-based platforms.

Table 12.4 Some examples of cloud-based platforms.

Provider	Name	URL of website
Autodesk	Autodesk Forge	https://forge.autodesk.com/
Nemetschek	BIM+	https://bimplus.net/
Trimple	Tekla BIMsight	http://www.teklabimsight.com/

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

These cloud-based platforms build on RESTful web services. REST has the advantage that it can be used from almost every programming language since it needs only to send HTTP requests encoded with JSON or XML messages. This means it is straightforward to use these services within an application that is programmed in C++, C#, Java, Ruby, Java Script or any other similar powerful programming language. Besides this, sometimes high-level APIs are provided for different programming languages to make programming more comfortable for software developers.

The previously mentioned platforms allow to store data like IFC files, photographs, any files such as for example Word Documents, Excel Sheets or scanned PDFs. Besides data management, also view capabilities are provided. There is a 3D viewer available on all the above-named platform that is also running directly in a web browser that allows inspecting your data in real time.

Besides this, each platform offers different APIs. For instance, Autodesk Forge offers a Reality Capture API that makes it possible to convert photographs to a textured 3D model. For more details consider the corresponding documentation of the platforms.

12.7 Visual programming

In recent years, Visual Programming Languages (VPL) have made inroads into the field of digital construction. Users can employ these languages as a tool to make repetitive work or the creation of variants and their evaluation a lot easier without the need for detailed programming knowledge (Chao, 2016; Cooper et al., 2000).

A visual language is defined as a formal language with visual syntax and semantics. It describes a system of signs and rules on the syntactic and semantic level with the help of visual elements, which are more readily understandable for non-professional programmers. Visual programming languages are often referred to as flow-based, since they represent complex structures as a flow of information (Hils, 1993).

Typically, the user interface of visual language applications comprises a canvas that serves as a basic workspace, and a library of individual components (nodes). Nodes are placed on the campus and arranged and linked to one another by so-called edges or wires (see Fig. 12.3). The resulting system can be stored as a graph system and passed on to other project participants or documented accordingly.

An essential distinguishing characteristic between different visual programming languages is the level of granularity. This granularity describes how finely the individual functionalities are resolved, i.e. whether functions within a node are entirely encapsulated or whether each sub-step is available and visible as a separate node. Encapsulation (low granularity) reduces the number of elements present on the workspace and contributes to the clarity, handling and comprehensibility of the overall system. At the other end of the scale (fine granularity), the resulting canvas is a more detailed representation of the information process in which the user can access and adapt each individual step as required.

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

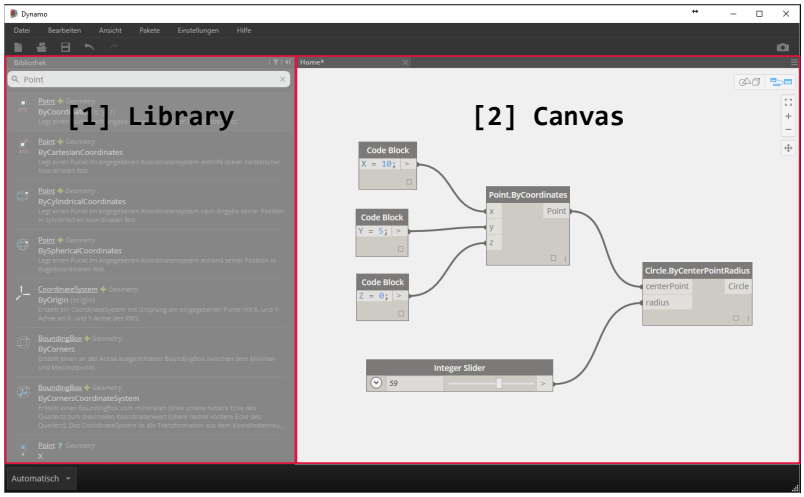

Fig. 12.3 Typical environment of a visual programming language: [1] Library containing the usable node elements, and [2] the workspace canvas. As an example, the interface of Autodesk Dynamo is shown.

Visual programming languages are controversial, particularly among programmers. The most commonly stated disadvantage is that programs created with a VPL rarely meet the high requirements of a professional programming environment. Furthermore, more complex situations, such as recursion, can often not be implemented or are hard to understand. A common argument is that users who design processes using a visual language should also be able to describe the information process with a conventional textual programming language (Chao, 2016).

On the other hand, VPLs are more user-friendly and make it easier for inexperienced users to get started with programming. Due to its abstract representation, it is easier for people without programming knowledge to understand and therefore to achieve results more quickly. Images can communicate ideas more simply and more clearly, and aid visual comprehension and remembering, not least also because there are no language barriers (Shu, 1988). A study by Cataraci and Santucci (1995) attests to the user-friendliness of visual languages using an example based on the common query language SQL.

In digital construction, VPLs are mainly used in two application areas: for generative purposes to generate geometric as well as semantic information, or for checking or querying information on existing models. In some VPL-based applications and environments the boundaries between the two are fluid and clear classification is not always possible. Most of the visual programming environments provide the ability for developers to extend libraries with their own functions to extend a program's functionality or field of application (Kurihara et al., 2015). Table 12.5 shows an overview of common visual programming languages.

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

Table 12.5 Selected VPL environments and libraries

Name	Application	Manufacturer	Programming Interface	URL
Dynamo	Standalone; Autodesk Revit add-in	Autodesk	C#, Iron Python	http://dynamobim.org/
Google Blocky	Web-Based	Google	JavaScript	https://developers.google.com/blockly/
Grasshopper3D	Rhinoceros3D plug-in	Open-Source	C++, C#, Python	http://www.grasshopper3d.com/
Grasshopper3D ArchiCAD	ArchiCAD plug-in	Graphisoft	C++, C#	https://www.graphisoft.com/archicad/rhino-grasshopper/
Marionette	Vectorworks plug-in	Vectorworks	Python	http://www.vectorworks.net/training/marionette
Scratch	Web-based	MIT	JavaScript	https://scratch.mit.edu/
TUM.CMS. VplControl	Standalone	CMS Chair	C#	https://github.com/tumcms/TUM.CMS.VPLControl

12.8 Summary

This chapter provided a brief overview of ways to read and write IFC files. Presently, the most common format for exchanging IFC data uses STEP clear text encoding, which is standardized in Part 21 (ISO 10303-21, 2016) of the STEP standard.

The difference between the early binding approach, in which the entities of the EXPRESS schema are mapped to entities of a high-level language, and the late binding approach, in which a generic, data-model-independent interface is used to access instance data, is explained and their respective advantages and disadvantages discussed. A key advantage of the early binding approach is that the majority of programming errors can be detected at compile time. A key disadvantage is that the class structure has to be generated in the host language using a code generator, and that this process must be repeated every time a schema changes. An alternative to using STEP-Part 21 to communicate IFC data is to use the XML-based format ifcXML. The SAX and DOM methods provide a means for the programmatic implementation of accessing IFC data as XML. The popularity and widespread support of the XML format means that it is likely that ifcXML will become an increasingly important means of communicating IFC data in future.

An important aspect of processing IFC data is the interpretation of geometric information. IFC data models present a challenge because they support very different means of geometric representations, including implicit representations. These must normally be processed and converted into a triangle network to render the geometry and make it available for further processing.

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

Where the existing possibilities offered by the vendor-neutral IFC and corresponding BIM tools are not sufficient, it is possible to extend commercially available software applications by developing add-ins or use cloud-based platforms providing additional functionality.

Finally, Visual Programming Languages (VPL) represent a new genre of programming tools that make it possible for AEC professionals to develop customized BIM solutions, without in-depth programming skills being required.

References

Cataraci, T., & Santucci, G. (1995). Are Visual Query Languages Easier to Use than traditional Ones? An Experimental Proof. In: *People and computers X: Proceedings of HCI '95, Huddersfield*, edited by Kirby, M.A.R, Dix, A., Finlay, J.E., Cambridge programme on human-computer interaction. Cambridge University Press.

Chao, P.-Y. (2016). "Exploring students' computational practice, design and performance of problem-solving through a visual programming environment", *Computers & Education*, Vol. 95, pp. 202–215.

Cooper, S., Dann, W., & Pausch, R. (2000). "Alice. A 3-D tool for introductory programming concepts", *Journal of Computing Sciences in Colleges, 15*(5), pp. 107–116.

Hils, D.D. (1993). *A visual programming language for visualization of scientific data*. University of Illinois at Urbana-Champaign.

Hubbard, M. (1990). *Constructive Solid Geometry for Triangulated Polyhedra*. Department of Computer Science, Brown University, Providence, Rhode Island 02912, CS-90-07.

ISO 10303-21:2016-03 (2016). *Industrial automation systems and integration - Product data representation and exchange - Part 21: Implementation methods: Clear text encoding of the exchange structure*. Standard, International Organization for Standardization, Geneva, CH.

ISO 10303-22:1998 (1998). *Industrial automation systems and integration - Product data representation and exchange - Part 22: Implementation methods: Standard data access interface*. Standard, International Organization for Standardization, Geneva, CH.

ISO 10303-23:2000 (2000). *Industrial automation systems and integration - Product data representation and exchange - Part 23: Implementation methods: C++ language binding to the standard data access interface*. Standard, International Organization for Standardization, Geneva, CH.

ISO 10303-42:2014 (2014). *Industrial automation systems and integration - Product data representation and exchange - Part 42: Integrated generic resource: Geometric and topological representation*. Standard, International Organization for Standardization, Geneva, CH.

Kurihara, A., Sasaki, A., Wakita, K., & Hosobe, H. (2015). "A Programming Environment for Visual Block-Based Domain-Specific Languages", *Procedia Computer Science, 62*, pp. 287–296.

Laidlaw, D., Trumbore, B., & Hughes, J. (1986). Constructive Solid Geometry for Polyhedral Objects. *Proceedings of SIGGRAPH '86, Computer Graphics, 2*, ACM, New York, USA.

Shu, N. C. (1988). *Visual programming*. New York: Van Nostrand Reinhold.

In: Borrmann, A.; König, M.; Koch, C.; Beetz, J. (Eds): Building Information Modeling - Technology Foundations and Industry Practice, Springer, 2018, DOI: 10.1007/978-3-319-92862-3

Index