Informatik
Computer Science

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Design and implementation of *Envision* - a visual programming system

Dimitar Asenov

Master Thesis

Supervisor: Prof. Dr. Peter Müller

Chair of Programming Methodology
ETH Zürich

7th April, 2011

Summary

In the following work we will present the concept and initial implementation of *Envision*. *Envision* combines the features of an integrated development environment with the functionality of an information system to offer a unified tool for software engineers. The system is independent of a specific programming language and offers a visual programming approach for developing applications. *Envision* is designed for general purpose development and targets professional programmers that work on software projects of all scales.

We first describe the basic requirements for the *Envision* platform. They define a flexible framework which can be extended and customized to provide a wide variety of functionality used in everyday software engineering.

The core of this work consists of designing a modular architecture for *Envision* and implementing a first version of the system. The focus of the current implementation is on mechanisms for visual object oriented programming. We describe the modules that constitute *Envision* and outline how they achieve the requirements and goals set earlier. The current state of the platform and the progress regarding visual programming is illustrated using examples from the running tool.

Finally we outline our plans for further development of *Envision* in the areas of visual interaction, semantic version control, statistics, software verification and others.

Contents

List of Figures

Acknowledgements

I am very grateful to my supervisor Prof. Dr. Peter Müller for helping me pursue my interest in the field of visual programming and software development tools. His continued support was vital for the development of the concept of *Envision* and the realization of the first version of the system, which is the result of many fruitful and inspiring discussions.

Additionally I would like to thank everybody at the Chair of Programming Methodology at ETH for their feedback regarding my work. In particular I want to thank Joseph Ruskiewicz, Valentin Wüstholz and Cédric Favre for their ideas and suggestions, and Malte Schwerhoff for our many discussions on programming, software visualization and development techniques.

Finally I would like to thank my girlfriend Lucia for her suggestions and valuable feedback regarding the concept of *Envision*.

1 Introduction

In this chapter we outline our motivation for building a new integrated development environment based on a visual programming approach. We then introduce the concept of *Envision* and the core features of the platform. At the end we briefly explore related work in the area of software visualization and IDEs.

1.1 Motivation

As computers become an ever larger part of our everyday life the requirements for software applications are getting more and more demanding. Computer programs today are among the most complex systems man has ever built and many different factors contribute to this. Nowadays applications run on a wide variety of hardware, ranging from high-performance clusters, to world-wide distributed systems, to personal computers and embedded devices. Software is designed for many different fields such as scientific simulations, transportation, health care, entertainment, finance, etc. Its cost can also vary significantly as proprietary systems compete with free open-source solutions. Increasing scalability, improving reliability and decreasing cost are only a few examples of the requirements facing institutions and developers today.

In order to cope with increasing complexity many aspects of software development have also evolved. Organizations optimize their operation by following established best practices such as the CMMI model for development [1]. The significant growth [2] and popularization of open-source software has introduced new distributed development practices fueled by community effort which can deliver high-quality alternatives to proprietary solutions at a fraction of the cost. On a more technical level: programming language paradigms have shifted from unstructured, to structured code and have continued to offer new higher abstraction concepts such as objects and aspects. Further improving the way we build software are advances in the area of distributed and parallel computing, formal methods and verification, testing and numerous other areas.

To take advantage of new ideas and techniques, it is essential to use appropriate tools. Tools for collaboration, project management, verification, debugging, version control, etc. play an important role in a developer's every day job. A need arises to integrate such applications into a single environment [3]. This allows developers to control the design and implementation processes from a unified platform. Modern integrated development environments (IDEs) are an indispensable tool for larger projects. Applications such as Eclipse[1] or Microsoft Visual Studio[2] not only offer many features in a unified interface but are also extensible by the user.

Despite the many advances of development tools there is one fundamental area where progress has been slow - the way we write, read and store programs. With rare exceptions, a program today is typically just a collection of text files in a particular programming language. New languages are continuously created and old ones improved in order to support emerging concepts in application programming. These concepts are what we as developers are interested in. The syntax of a specific language however is just a tool that helps us express these ideas. It is precisely this tool that has been neglected and causes a number of issues:

- In addition to understanding general programming paradigms, developers have to learn a strict syntax for each programming language they use. The syntax for a single concept usually differs between languages and may even change in newer versions of a language.
- The more expressive a language is the more complicated its syntax becomes thus making it harder to understand or write programs.
- The current program representation is limited to text. This exercises mostly symbolical cognitive processes and leaves out other tasks that humans are naturally skilled at such as processing visual and spatial information. Furthermore other useful artifacts of software engineering such as graphs, tables or images cannot be part of the source code of an application.
- The organization of a project into directories and files does not accurately reflect the internal organization of programs and requires time to maintain. For example in C or C++ the

[1]`http://www.eclipse.org/`
[2]`http://www.microsoft.com/visualstudio/en-us/`

separation between header files and implementation files can create non-trivial dependencies between parts of the code and needs to be carefully considered. Additionally when using templates, one is forced to put all such code in header files further complicating the maintenance of these classes, especially in cases where this functionality needs to be exported from a shared library. Another example where it is important to carefully structure the different files of the application is when automatically generated code is mixed with user defined routines in GUI programming. Languages such as Java deal with this problem by defining two separate classes (one deriving from the other) in two different files that contain the different parts of the implementation. In C# a different solution is possible - using partial classes[3]. This allows the developer to split the definition of classes, interfaces, structures and methods across multiple files. In each of these cases additional effort is required when building software, simply because it is stored as files.

- Because of the wide variety of available languages, current IDEs provide only a basic support on the generic language level. This includes features such as keyword coloring and text templates. More advanced features must be reimplemented separately for each language even if they concern identical concepts. Such a feature for example is recognizing different semantic elements of a program like method calls, literals or symbol definitions in order to enable better syntax coloring or refactoring.

To resolve these problems we have designed *Envision*. It is a visual programming system that concentrates on concepts rather than languages. It provides a unique mix of features that try to improve the way we develop software by separating the concerns of reading, writing and storing programs. Such a tool can offer many advantages to developers.

First, presenting a program in a two dimensional plane using a combination of text and visual elements such as shapes, colors and icons has the potential to make certain types of interactions more natural and speed up comprehension. During a debugging session, for example when examining a call stack trace, instead of switching between different files, we could simply arrange the execution objects we are interested in, in a linear sequence on the screen. The sequence could indicate the order of the calls and can consists of methods as well as other program fragments like classes or individual statements. Additional information such as variable values, or associated bug reports could also be presented directly in the environment to aid the developer.

Additionally, a visual organization of a software project can also be very useful for programmers joining the development team who are still unfamiliar with the application's design. At a more abstract zoom level one can get a good overview of the structure of the project, while finer zoom levels reveal details about the implementation. It is easy to navigate such a hierarchical design in a visual environment. It is also possible to embed software artifacts such as diagrams, tables or documentation that will further facilitate the understanding of the program.

Furthermore, modifying a program that is presented visually is possible in a number of ways. Novice programmers can use visual tools such as an object palette to explore the capabilities of the system and design applications in a more interactive manner using drag and drop techniques. Professional developers can use powerful command-like textual tools that quickly allow them to manipulate every aspect of a project. In both cases programmers directly manipulate visible objects and work with concepts, as opposed to a specific programming language. Thus there is no complicated and strict syntax to learn and understand. Even textual commands can feature a simple invocation mechanism that allows for mistyped or imprecise input.

Finally, supporting all this functionality is a flexible model for applications. This model defines entities based on concepts such as classes, methods, visibility scope etc. While these entities can be mapped to constructs in a specific programming language, they are not inherently associated with any language. Thus many tools for refactoring, analysis, verification and other useful tasks can be created once and applied to the generic model therefore covering a wide range of programming

[3]http://msdn.microsoft.com/en-us/library/wa80x488.aspx

languages. This model also frees the programmer from the duty of maintaining a development directory for the project. Persistence is managed by the system and is performed at the level of logical objects such as classes and methods. This eliminates the burden of properly structuring program fragments into files, which could often be quite a hassle as we have shown above. Moreover, such an approach allows semantic version control of fine-grained elements, such as methods or even individual statements. This enables a more accurate trace of the evolution of a project, collection of statistics, etc, which could facilitate collaboration or decision making.

These are just some of the features of *Envision* that make such a system desirable. The remainder of this chapter introduces the concepts behind the platform in more detail and then discusses related work. In chapter 2 we outline the main requirements for the system. Next, in chapter 3 the architecture of *Envision* is presented and its design and implementation are described in more detail. In chapter 4 we guide the user around the currently implemented interface and demonstrate its functionality. Finally we summarize our contributions and outline future work in chapter 5.

1.2 The concept of *Envision*

Here we introduce our vision about the *Envision* system as a whole. The current implementation developed as part of this work covers only a fraction of the concepts discussed below. The implementation and its limits are described in more detail in chapter 3.

Envision is an IDE and an information system. A core principle of the platform is the separation of concerns of reading, writing and storing programs.

Traditionally developers create programs by writing text. The same text is also used when someone is trying to understand what a program does to modify it. The same text is also used to store the program on disk. The reasons for this are mostly historical, with this being the only possibility in the early days of computing. Nowadays however the absolute coupling of these three activities is no longer necessary. *Envision* implements a Model-View-Controller approach to software development that clearly separates program storage, comprehension and modification and allows for their optimization.

The ***Model*** is the data structure that defines an application. This is similar to the abstract syntax tree of a programming language. A model can contain executable fragments, comments, images, tables, documentation, graphs, plans, schedules or any other software engineering artifact. It has no inherent representation, neither textual nor visual. Here are the main features that *Envision*'s model will offer:

- Rich content - the model can contain many different software artifacts along with the program structure. For example graphs, images or tables embedded in the model, can be presented right next to the classes and methods they refer to. This is particularly useful for developers which have just began working on an existing project as it gives them immediate access to supporting documentation. Ultimately this will make it easier to explore and understand how an existing system works.
- Content linking - information from different parts of the system can be linked to ease navigation and maintain consistency. For example, values of program constants can be derived from values which appear in the documentation of the project. This will help eliminate the typical inconsistency of only modifying the source code and postponing the update of the corresponding documentation. In addition it will be possible to follow hyper-links that interconnect all software artifacts.
- Statistics collection - a better integration of software artifacts enables a broad range of statistical data to be easily collected and analyzed. The statistics themselves can become

part of the application model. For example one could query the ratio between bug reports and test case coverage over time, which may be used to help with decision making.

- Semantic version control - version control can be performed for logical objects such as classes or methods as opposed to files. This will make it easier to track the evolution of the program and even find version related bugs. For example, the fragile base class problem can be more easily diagnosed by looking directly at the differences between two versions of the same class.
- Flexible and implicit persistence - projects can be automatically persisted on disk, in a database or in a different way defined by the programmer. It is no longer necessary to manually structure files on disk and programmers can focus on the logical structure of the application they are developing.
- Language independent model - *Envision*'s model is based on programming concepts and not on the features of a particular programming language. Programmers use *Envision* to create a model of an application using these concepts. Concepts for example are classes, inheritance, dynamic dispatch, static methods, constants, operators, generics and many other commonly found features in modern programming languages. Such a model can then be translated to any programming language as long as that language has support for all used concepts. This has the advantage that many tools can be created to operate on the generic model and their results will then automatically apply to a wider range of programming languages. This includes tools for refactoring, analysis, visualization and others. Naturally it will still be possible to parametrize such a tool with a specific target programming language when this is essential for the tool's operation (for example profiling and optimization tools).

The ***View*** is a particular way to visualize the model to the user. A view may display the entire model or part of it. It can display a segment of the program in complete detail or it can show a summary of the more important features of that part. The view can consist of text, visual elements and any combination of the two. Some of the key features of this approach are:

- Rich and flexible visuals - programs are represented as a mixture of textual and visual elements and are not limited to text files. It is possible to display images, tables, animations or other artifacts in the same view as the program fragments that they refer to. As discussed in 'rich content' above this helps developers to explore and learn the program faster. Furthermore working in a more visual landscape will help developers stay oriented when performing complicated tasks.
- Analysis overlays - it is possible to present the results of various types of analyses as an overlay directly on top of the program segment they refer to. This could be used for example to highlight errors found by a type system or a verifier, to indicate which branches of a method have not been covered by test cases and others.
- Multiple views - a model may have many associated views which visualize it in a number of ways suitable for different tasks. For example one view may show the contents of a class while another one shows the class within an inheritance hierarchy together with other classes used by it.
- Visual cognitive processing - rendering programming constructs using visual elements such as lines, colors and positions will enable their processing to happen mostly on the level of the visual system in the brain. This yields better retention rates as compared to symbolic cognition [4, 5]. Using a visual programming system could thus improve the memory of programmers with respect to parts of an application that they have worked on in the past.
- Spatial navigation - a program can be navigated in a two dimensional canvas to improve developer orientation and focus. For example, instead of switching between multiple tabs, a programmer can simply put all relevant pieces of the program next to each other for a better overview.

- Semantic zoom - the developer can zoom in and out to observe the application at different levels of detail. This is especially helpful when one is exploring an unfamiliar program as it makes it possible to quickly gain an overview of the entire project and selectively see more detail about a particular part by zooming in on it. Once a programmer is familiar with the application different zoom levels can still be useful by displaying different information about the system. The finest level will include the expressions, statements and similar constructs, while coarser levels could provide statistical information, architectural overviews, etc.

The ***Controller*** is the mechanism by which the *Model* can be modified. Unlike conventional IDEs where programmers directly modify the structure of the program by writing text, in *Envision* they rely on the aid of the *controller* to do this more efficiently. Some key ideas of our approach are:

- Direct manipulation - programmers can interact in a natural way directly with the visual objects that represent a segment of the model. This helps preserve focus, which could otherwise be lost, for example if the developer needs to go through a series of menus to perform a task.
- Keyboard centric interaction - while keyboard input is standard for traditional text-based programming, many visual programming systems are centered on mouse interactions. Unlike these systems, *Envision* focuses heavily on keyboard input and uses the mouse to complement it. We believe that in the hands of an experienced developer, the keyboard provides a very effective input mechanism and provides for better productivity compared to mouse based interactions.
- Command based interaction - one way to modify the model is via textual commands issued by the user. Such commands are context sensitive and have a lightweight syntax in order to simplify program manipulation. They can quickly change the model in arbitrary and complex ways. For example in a method that has an integer argument `count` issuing the command 'for count' could automatically examine accessible symbols and create a `for` loop that iterates from 0 to `count` since count is defined as a numerical type. A command like 'add pattern Visitor' issued within a class, could automatically add methods that implement support for the Visitor pattern to the class and all of its derived classes. Such a task might require a significant amount of time in a traditional text-based environment.

A pervasive feature throughout *Envision*'s design is extensibility. The system supports extensions that can add new functionality to all of its core elements. This enables us to experiment with different settings and features so that we can fine-tune the platform and maximize productivity.

1.3 Related Work

In [6] we report on a preliminary study on the feasibility of the *Envision* concept. Previous work in the field of visual programming languages (VPLs) and software visualization is discussed in more detail there. Here only a brief summary of related work is presented.

The idea of visual programming is not new. Initial research was focused on creating new VPLs and introducing new programming paradigms better suited for visual programming. This includes languages such as Vista [7] by Schiffer and Fröhlich, Seity [8] by Chang, Ungar and Smith and VIPR [9] from Citrin et al.

Efforts at creating a general-purpose visual programming language have not succeeded however. This is evident in the fact that today the most widely used languages in industry are textual. Initially the VPL community had high expectations and disappointed by the lack of results many researchers gradually shifted their focus to software visualization and the broader field of human-machine interaction.

Nevertheless, visual programming has been very successful in domain specific areas. A possible explanation for this is that domain specific languages offer components and levels of abstraction specific to the task at hand which greatly simplifies the programming process. National Instrument's LabView[4] software is one of the most successful commercial visual programming systems. It is designed for advanced data acquisition, analysis and visualization purposes for electronics and electrical engineers. It supports a wide variety of concepts such as data flow programing, object orientation, parallel multi-core programming and others. Plant Simulation[5] from Siemens is another visual data-flow programing environment. It is designed for simulations in production and transport logistics and has a number of interesting features such as a discrete simulation engine, object oriented design and 3D visualization of execution.

In Green and Petre's classical work on Cognitive Dimensions [10] they put traditional textual programming languages represented by FORTRAN against VPLs such as LabView and Prograph. They found that visual languages make certain tasks easier, due to the reduced need to learn specific syntax. Visual languages were found to have their own issues such as difficult management of screen estate and high viscosity (difficulty of making local changes to an existing program). The authors developed the framework of Cognitive Dimensions that has been used since by many researchers to evaluate new VPL designs. Learning from their results, *Envision* tries to address the issue of managing screen estate by using semantic zoom and flexible visual representations. We address the issue of high viscosity, which is crucial in VPLs, by focusing on keyboard and command driven input. It is important to note that this does not void the benefits of visual programming that come with the reduced need to learn a specific language syntax. The command input of *Envision* uses a simple syntax with minimal amount of punctuation and special characters and provides a built-in auto-complete feature. It is even possible to automatically correct for some typos and ambiguous input. This approach is similar to "Sloppy programming" as defined by Little et al. [11, 12, 13].

More recently there has been some effort to enhance existing IDEs with features typical for VPLs. CodeBubbles by Bragdon et al. [14, 15] is a plug-in for Eclipse that provides programmers with two dimensional surface on which they can work with the program source code. This surface is populated with bubbles - a light-weight window-like widgets that can contain the source code of a single class or method. DeLine and Rowan have created Code Canvas [16] - a development prototype interface for Microsoft Visual Studio. It presents the user with a "canvas" where fragments of source code appear structured according to the class and directory they belong to. Developers can navigate a project in two dimensional space, turn on or off different layers which overlay information on the canvas and use semantic zoom features. It is also possible to filter out certain entities based on user defined criteria, such as search results. Both of these tools feature modern techniques in software visualization which are also part of *Envision*'s design. Nevertheless, our platform differs in one major aspect. *Envision* has at its core a generic program model, while CodeBubbles and Code Canvas still operate on top of text files. This allows us to not only surround classes or other program fragments by a visual box and group them together, but also to visualize any segment of the program in an arbitrary way. For example mathematical functions such as summation or integration, or objects such as sets and matrices can be visualized using their standard look and feel as opposed to just a sequence of symbols. The concept of *Envision* also extends beyond visual programming which is the focus of CodeBubbles and Code Canvas. An advantage of these tools is that they can be directly used with existing source code, whereas *Envision* will require existing sources to be imported into the generic model provided by the system.

[4] http://www.ni.com/labview/

[5] http://www.plm.automation.siemens.com/en_us/products/tecnomatix/plant_design/plant_simulation.shtml

2

Requirements

This chapter specifies the requirements for the *Envision* platform. Each of them is assigned a unique id in order to allow referencing. The requirements are grouped according to the system aspect they describe.

The requirements we set here define an extensible system and focus on the core architecture and design issues. Thus only high-level requirements are provided. Specific details are not given for functionality which is customizable or extensible.

The requirements specified in this chapter are of two types:

1. those which influence the architecture of *Envision* by defining how different modules interact with each other.

2. those which influence the design of a particular module by defining what minimal set of features and extensibility mechanisms should be supported.

In either case only requirements for the first implementation of *Envision* are discussed. Further developments are outlined in section 5.1.

2.1 General

These requirements pertain to the *Envision* platform as a whole. They enable the achievement of core goals of the project.

GEN-1: General-purpose
The *Envision* system should be designed for general-purpose software development in the same sense in which Eclipse and Microsoft Visual Studio using Java, C++ and C# are also general-purpose. It should be suitable for creating applications in a wide variety of domains. Visual programming environments are already successful in domain-specific areas and our goal is to try and bring this success to the general case.

GEN-2: Support for experiments
Envision should be suitable for conducting experiments. Many of features of the platform are not well understood and quantitative results in the field of VPLs are scarce [17]. Different implementations of these features will need to be tested in order to find a better approach to visual programming. Conducting experiments with programmers will be a frequent activity in the development cycle of *Envision* and our goal is to have appropriate support for it.

GEN-3: Different scale support
The platform should provide appropriate support for software projects of different scales. The target size of applications developed with *Envision* ranges from small class room exercises to large projects such as operating systems. This will enable the evaluation of the system by both students and professional programmers which will allow us to tweak our design and make it suitable for a wide range of tasks.

GEN-4: Extensibility
The functionality of the programming system should be extensible by third party programmers. They should be able to introduce entirely new features to *Envision* or improve existing functionality. This requirement is a crucial step towards GEN-2.

GEN-5: Customization
Each component of the *Envision* platform should be customizable. This means that the user can alter the component's behavior without the need to recompile *Envision* or understand details of its design. The platform should provide a framework that facilitates this. Components should be free to choose in what way their behavior can be customized. This may include operation modes, keyboard shortcuts, look and feel and other settings. This requirement is essential for achieving GEN-2.

2.2 Technical

The requirements describe here pertain to major technical aspects of the implementation of *Envision*. They assure that the developed solution will be of high quality. Since *Envision* is a long-term project it is essential to have a well built platform in order to facilitate future development.

TECH-1: Cross-platform
Envision should be a cross-platform IDE, capable of running on major operating systems used for software development. At least Linux, Microsoft Windows and Apple MacOS should be supported. Apart from reaching a wider audience of developers this will also improve the quality of our solution

as cross-platform development requires discipline and forces a better structuring of the program.

TECH-2: Plug-in based
Different components in *Envision* should be implemented as plug-ins. Plug-ins are shared libraries that can be added to the system at run-time. They are also the main mechanism by which the functionality of the platform can be extended. A plug-in can provide arbitrary new functionality and is allowed to use the services of any other plug-in, which is explicitly defined as a dependency. The system should be able to mange different versions of a plug-in. A plug-in based design is essential for extensibility (GEN-4) and also improves the software quality as it requires clear separation of concerns.

TECH-3: Tested
The different components of *Envision* should be tested to assure a high-quality implementation. This could be done as a variety of unit or integration tests. Tests, and especially regression tests will greatly facilitate the future development of the platform.

2.3 Application model

As discussed in the concept of *Envision* the (application) model is the central data structure that contains all programming structures and software engineering artifacts associated with a software project. Here we will describe the most essential requirements that define that model.

MODEL-1: Different model types
Envision should support different types of models. A model type is a model data structure that supports a specific set of features and programming concepts. For example a model for object oriented programming will support concepts such as classes, inheritance, statements, loops and so on. Ideally there will be only one model that allows the user to create an arbitrary program. This is desirable as supporting tools like refactoring and verification would only need to be created once. However this scenario is in most cases not possible. A need for having different model types arises when some features are incompatible with each other and thus cannot be easily integrated into a single data structure. For example shell programming (e.g Bash) has a very different structure compared to OOP and will therefore require a different model type in order to be supported in *Envision*. While a model type such as OOP is meant to be generic and provide a wide support for constructs typical for the programming languages it represents, sometimes this might not be feasible. For example there might not be a suitable way to provide a common model for Java and Scala that supports all of the features of both languages. Although both support OOP, Scala additionally has many features typical for functional programming. In such cases a derived model can be created that further specializes the features of the generic model in order to provide functionality specific to a language. Nevertheless common functionality should be supported in a unified model wherever possible in order to facilitate tool building. The user should be able to choose what model to use for the current application under development.

The rest of this section describes the characteristics that each model should have, regardless of what type it is and what programming concepts and languages it supports.

MODEL-2: Rich application model
An application model should allow for a wide variety of content to be part of it. The model should be able to contain program source code, documentation, graphs, figures, tables, requirements, animations and so on. This follows the core concepts for *Envision*'s role as an information system that integrates all artifacts of software engineering.

MODEL-3: Modular model
A model should provide convenient means to write modular applications. It should be possible to clearly separate the different parts of an application and define hierarchies. This could include concepts such as projects, groups, modules, libraries, packages or classes and so on. As modern application grow in size it is important to manage this by structuring them properly.

MODEL-4: Tree shape
Each application model should be implemented as a tree data structure. Nodes in the tree should represent programming constructs and other software engineering artifacts. Edges should represent containment. Therefore each programming construct or artifact is made of other constructs and artifacts. This establishes a hierarchical model that matches the structure of programs developed with today's modern programming languages. A tree structure is also desirable since there exists a large body of algorithms that operate efficiently on such structures.

MODEL-5: Tree access
Flexible means to access the elements of the model should be provided. This includes mechanisms to access individual tree nodes, traverse the entire structure, search for nodes and query a node for information regarding the application. This will facilitate the development of components for *Envision* which work with the model to perform tasks such as compilation, refactoring, verification, statistics and others.

MODEL-6: Simple tree manipulation
It should be possible to manipulate the application model at the tree level without having specific knowledge of the model type. This entails that generic operations such as removing, adding or replacing sub-trees should be supported. This allows low-level generic components to perform actions on all available models. For example a simple copy and paste operation that duplicates a sub-tree of a model, should always be possible regardless of the specifics of the model.

MODEL-7: Advanced model manipulation
Building on top of the generic manipulation operations, more advanced modification mechanisms should be available. These can depend on the particular type of a model. Using these operations it should be possible to perform actions commonly associated with developing applications for that model. For example in a model targeting object oriented programming, there should be an easy way to create classes, methods, fields, expressions or statements. The programmer should not need to construct such objects manually, node by node. This will enable the faster building of more specialized components such as a verification or a refactoring plug-in.

MODEL-8: Extensible model
The application model should be extensible. This means that plug-ins developed by third party developers should be able to add support for new concepts to an existing model, or to make new derived models. This reflects the extensibility goal GEN-4 for *Envision*.

MODEL-9: Undo
Undo and redo commands commonly found in today's development tools should be provided by *Envision*. This functionality should be available at the level of the application model. A limited history of the most recent modifications to the model tree should be kept. It should be possible to undo and redo actions such as the insertion of new nodes, the modification of existing nodes or node deletion. This is an essential usability feature for any modern IDE.

MODEL-10: Concurrency
The model should provide support for concurrent access to the tree structure. Users should have

fine grained control with respect to how different sub-trees are accessed. Software engineering nowadays is not just about writing code. Programmers need to perform many additional tasks such as run test cases, compile programs, verify and analyze the source code, profile and optimize applications and others. Many of these tasks can be performed in parallel in order to reduce the overall development time. Because improving programmer productivity is the main goal of *Envision* adequate support for concurrency is vital.

2.4 Persistence

PERS-1: Automatic persistence
The user should not need to explicitly specify how a model is persisted, the structure of a persisted model should be based on the logical hierarchy of the nodes it contains and should be automatically established by the system. This means for example that if a model is persisted as files, *Envision* automatically creates an appropriate directory and file structure without the involvement of the developer. As we have discussed in section 1.1 this frees the developer of the burden to manually maintain an appropriate structure and lets them focus on the logic of the program. Ultimately this should lead to an increase in efficiency.

PERS-2: Persistence of generic models
It should be possible to store any model type to a non-volatile storage using a generic mechanism. Since new models can be created by deriving existing ones and existing models can be extended with new concepts, it is important that plug-in developers can do this quickly without the need to implement new persistence mechanisms. This will encourage the development of extensions for *Envision*.

PERS-3: Storage media
The persistence mechanism should allow the saving of a model to an arbitrary medium, such a file system or a data base. Plug-in developers should be able to add new extensions to *Envision* that provide additional storage media. This will improve the flexibility of the system and will enable the interoperability with a wider range of tools.

PERS-4: Model retrieval
Once a model is stored to a persistent store it should be possible to load it from that store. A loaded model should be identical to the one that was stored. This is just to ensure that a persistent store maintains the consistence of models.

PERS-5: Partial loading
A mechanism should be implemented that allows the partial loading of a model. This means that some nodes from the model may be partially loaded. A partially loaded node is one that does not load its entire subtree, but rather only essential parts of it. It should be possible to fully load a node that is partially loaded. This is useful in cases where the developed application is large and loading it entirely in memory is not an option. For example, all methods in a big project may be partially loaded. This will load the method name, arguments and return type but will omit the method body. When a programmer needs to work with a particular method its body can be fully loaded. This will enable *Envision* to be used on large scale projects.

2.5 Visualization

VIS-1: Model and node visualization
It should be possible to visualize an entire model or parts of it. An appropriate visualization should be provided for each individual node, so that nodes can also be displayed by themselves outside of their original context. This adds a lot of flexibility when constructing visual representations of the model. Different parts of it can be shown next to each other in varying detail to allow for a better overview and task oriented arrangement. For instance, the programmer might need to debug a particular method of a class, and visualize only this method without the rest of the class. As the debug session progresses other components of the model might be needed and the developer should be able to visualize exactly what is needed. This will improve the concentration of the programmer by eliminating the need to browse through a large code base before reaching a point of interest.

VIS-2: Views
Envision should support the concept of views. A view is a particular representation of the model that is shown on screen. This view can represent a segment of the model or the entire model and the used representation can include arbitrary visual and textual elements. Specialized views such as an object explorer, a project navigator and others are a common part of modern IDEs and help programmers perform a specific set of tasks faster.

VIS-3: Simultaneous views
The system should allow multiple views to be concurrently visible. They can show different parts of the same model or even different models. This is very useful for a programmer that needs to inspect the application under development from different perspectives to gain a better understanding of it. For example one view can show the inheritance hierarchy for a class while another one shows which other classes are used by it so that the developer sees all relationships in which the class participates.

VIS-4: Multiple representations of the same data
It should be possible to simultaneously visualize a node in different ways. This is needed to support different views of the same data VIS-3.

VIS-5: Default visualization
A default visualization should be provided for each node of a model that does not have a customized representation. This visualization should allow the user to explore the tree structure of the node and make basic changes. This functionality should be independent of the model type used. This will greatly aid third party plug-in developers who want to introduce a new concept to envision but do not have the time to implement a visualization for it. In this case, the new concept can still be added and used with the default visualization, until a better one is created.

VIS-6: Customization
Each visualization should be customizable. This should include as many properties as possible such as colors, outlines, shadows, positions, layouts, etc. *Envision* should provide a framework that simplifies this for new plug-in developers. This requirement facilitates the goal to conduct experiments with *Envision* with different parameters of its features as outlined in GEN-2 and GEN-5.

VIS-7: Rich content
It should be possible to visualize different types of content such as text, graphs, images, animations, etc. This is necessary to appropriately represent nodes from the model which are inherently

expressed in such a way. Having a wide variety of visualization primitives will also facilitate future experiments with the system.

VIS-8: Two dimensional canvas
Visual objects should be positioned on a two dimensional canvas. Navigating to different program elements should be possible by looking at different locations on the canvas. The idea is to allow the programmer to navigate a consistent surface of objects as opposed to switching between different tabs. This is also a natural concept in visually oriented systems. Similar functionality already exists in tools such as CodeBubbles and Code Canvas and shows a lot of potential for improving the interaction with the IDE.

VIS-9: Layers
Visualizations should support layers. Different pieces of information can be put on those layers such as class diagrams, project health status, search results, etc. The user should be able to decided which layers are visible. The concept of layers has long been available in CAD systems, image manipulation programs and other specialized software where it is a highly useful tool. The ability to present an overlay of useful information on top of a visual representation of a software segment has a lot of potential to improve the programmer experience. This needs to be better explored in *Envision*.

VIS-10: Semantic Zoom
The user workspace should allow zooming in and out of the current application. Different zoom levels should present the user with different information appropriate for the current level. Such functionality can facilitate the exploration of the developed application. The programmer can zoom to a level that provides just the right type of detail without including too much information, thereby improving focus.

VIS-11: Navigation map
It should be possible to navigate the visualization of a model using a map. The map should always show the entire project and allow the quick repositioning of the main view to a location of interest. Such an interaction is natural for any two dimensional space and will improve navigation.

VIS-12: Legend
There should be an option to present a legend on screen that will make it easier for unexperienced users to recognize objects on the display. This will be especially useful for users new to *Envision* or programming to get to know the system faster. Unlike reading a system manual, a legend on screen shows relevant information exactly when and where it is needed.

2.6 Interaction

INTER-1: Direct manipulation
The user should be able to select and directly manipulate any object that is visible on screen. Actions possible through direct manipulation are determined by the selected visual item. Interaction should be possible using the mouse or the keyboard. Direct manipulation techniques have long been present in visual programming and allow programmers to perform actions directly with the objects they see, as opposed to opening new windows, changing views or invoking menu commands. This improves focus as the programmers work flow is not interrupted. Further benefits of this approach is the immediate feedback that users receive from the system and that it naturally fits a visual representation.

INTER-2: Full model control
The interaction mechanisms should give the user full control over the model tree. It should be possible to change the structure of the tree and therefore the application under development in an arbitrary way. Just like modern IDEs, *Envision* should be a self-sufficient tool for creating software. This includes not just visualizing a model but also creating and modifying it in any way.

INTER-3: Common manipulation operations
A set of common manipulation operations should be available for each visual item, regardless of its type

- selecting objects
- copying objects to the clipboard
- pasting objects from the clipboard
- deleting objects

This requirement is a consequence of MODEL-6 and will enable the basic manipulation of tree nodes even when no specific interaction mechanisms have been developed yet. This also simplifies the development of new model extensions.

INTER-4: Keyboard navigation
Envision should make it possible to navigate to any part of the program using just the keyboard. The user should be able to select any individual visual item from the project without the help of the mouse. This will improve productivity for users acquainted with the environment and is in accordance with *Envision*'s goal to provide a keyboard-centric IDE.

INTER-5: Configurable interaction
The interaction for each visual item type should be configurable. This can include possible actions, shortcut definitions, etc. This enables customization and promotes the conducting of experiments as discussed in GEN-5 and GEN-2.

INTER-6: Default object properties control
There should be a default and uniform way to set properties of objects. E.g. member visibility, read/write permissions, etc. This feature is useful for newly introduced extensions to a model that do not yet have a custom implementation of an interaction mechanism. This allows for the quick development of model extensions in order to test features without the need to create specialized interaction functionality.

INTER-7: Command prompt
The user should be able to interact with *Envision* by typing textual commands in a prompt widget. These commands could alter the visualization, invoke system functions or alter the program model. This is an essential feature in *Envision* as introduced in section 1.2 and is part of the concept of a keyboard-centric interaction. The main goal is to allow direct manipulation by textual commands in order to improve the productivity of developers.

INTER-8: Context-sensitive commands
The input commands should be context sensitive. The context is determined by the currently selected visual object and its underlying model. Plug-ins should be able to define which commands are available for a particular context. This allows for a great flexibility within the range of commands available in a model. It is also intuitive that different objects on screen support different sets of manipulation commands. By offering a specialized set of commands, the interaction with a visual object better supports the direct manipulation principle.

INTER-9: Flexible command syntax
The command text that a user needs to write to achieve a particular result should be minimal and flexible. Depending on the context, textual shortcuts in the form of abbreviations, shortened command names or even mistyped commands could be accepted as valid input, as long as the input component can unambiguously determine what operation to invoke. A simple syntax that supports error recovery will help developers get familiar with the command input mechanism faster and will improve their productivity.

3

System design

In order to allow for new extensions to the basic core of *Envision* a modular architecture was designed. This design was inspired by the one of the Eclipse IDE, which has proven its solid foundation and suitability for large scale general-purpose software development.

In this chapter we will describe the architecture of *Envision* and give more details about the design of its components. Technical details about various aspects of the system are presented in the Appendices at the end.

3.1 Architecture overview

The plug-in based architecture of *Envision* is depicted on figure 3.1. The design consists of three layers on the top level - executable, generic and model specific. The last two consist of plug-ins that implement all functionality. Within a layer plug-ins can have arbitrary dependencies. Plug-ins can also depend on functionality provided by lower layers. The design fulfills the requirements for extensibility GEN-4 and plug-ins TECH-2.

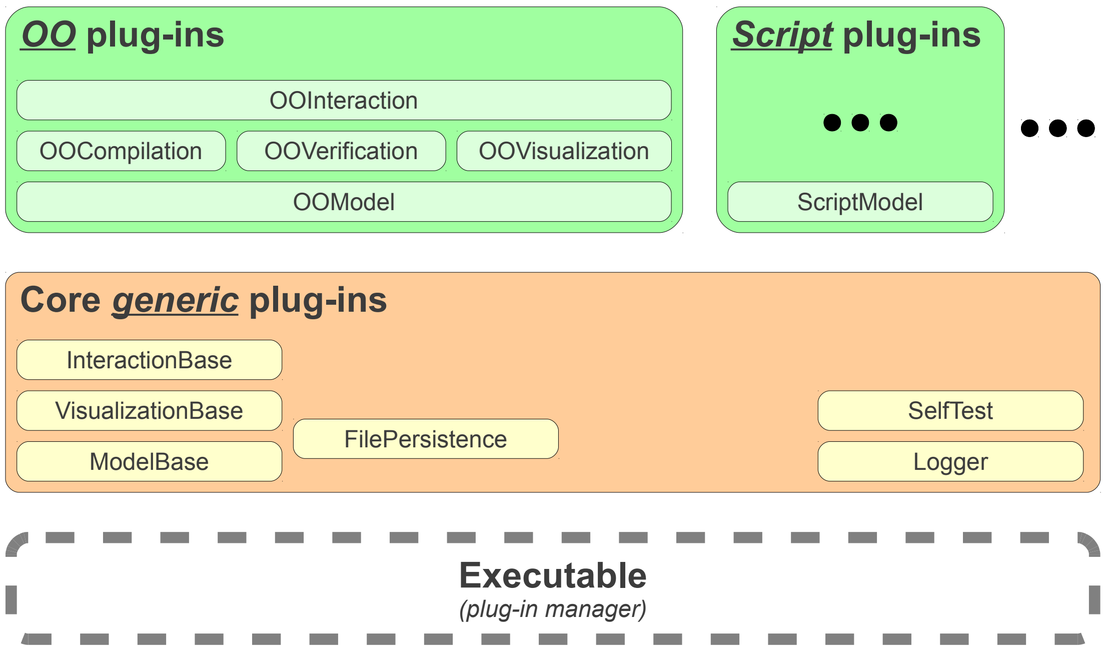


Figure 3.1: The high-level architecture of *Envision*.

The ***executable*** layer, marked with gray, consists only of the application executable. It is solely responsible for loading and managing plug-ins.

The ***generic*** layer, marked with orange, consists of all plug-ins which form the core functionality of *Envision*. This is where the foundation of the Model-View-Controller approach to programming is established. All other plug-ins rely on and extend the functionality implemented here. This is where the interaction between the different sub-systems is defined and where the interfaces of the generic components are specified. Here is a brief description of each plug-in from this layer:

- *Logger* provides a simple means to keep a log of events that have occurred in the system.
- *SelfTest* implements a small test framework for *Envision* plug-ins.
- *ModelBase* defines the foundation for application models. This includes some basic nodes, concurrent access mechanisms, persistence interfaces and others.
- *VisualizationBase* defines the basic visualization behavior of *Envision*. It provides graphical representations for the generic nodes defined in *ModelBase*, a number of layouts, shapes and other useful visual items.
- *InteractionBase* defines the interaction framework that processes user input and modifies the state of *Envision*. Custom behavior for some of the visualizations in *VisualizationBase* is implemented here.

- *FilePersistence* implements a persistent store that can save any model to an XML file. Additionally, it provides access to the system clipboard.

The ***model specific*** layer, marked with green, consists of plug-ins that are typically centered around a particular programming model. As defined in the model requirements (section 2.3) different model types can be supported. In the graph above there is a cluster of plug-ins related to object oriented programming and a second one for script programming. Any other model type can also be supported in this layer such as functional or logic programming or programming targeting a specific language like Scala for example. New models can be created by deriving from the generic functionality provided in the previous layer or by building on top of existing models. The current implementation of *Envision* is targeting the object oriented programming paradigm for which the following plug-ins are planned (only the first two are implemented):

- *OOModel* introduces many new model tree nodes specific for OOP, such as classes, methods, fields etc.
- *OOVisualization* implements visualizations for all the constructs in OOModel.
- *OOInteraction* defines mechanisms to create and work with an OO model and its visual representation.
- *JavaCompilation* provides a simple converter that can produce Java source code from an OO *Envision* model.
- *OOVerification* defines rules for checking the correctness and consistency of an OO model. This includes type checking, formal verification, etc.

Note that the last layer is focused on paradigms and techniques rather than a specific programming language. Thus it is possible to compile an *Envision* application designed in the OO Model to any programming language (Java, C++, C#) as long as there is an appropriate plug-in that provides this conversion service.

3.2 Using the Qt framework

Envision uses the Qt[1] framework at the core of its design. My experience with this framework together with the numerous advantages it offers makes it an excellent choice for the development of *Envision*.

Qt is a mature application and UI framework originally designed by Trolltech and currently developed and supported by Nokia. Its first version was released in 1992. Currently the framework is open-source software under the LGPL[2] license and is available on many different platforms including Linux, Microsoft Windows, Mac OS X and others. It is natively written in C++ but bindings to many other languages are also available. Thanks to this and to the exceptionally good quality of its documentation, Qt enjoys a vast community of developers who target the platform and offer support to others.

Here are some of the more important features of Qt that *Envision* uses:

- Cross-platform development - Qt greatly simplifies the process of writing cross-platform code by providing a single interface to many features commonly present in different operating systems, such as shared libraries, input devices and synchronization mechanisms for multi-threaded programming.

[1] `http://qt.nokia.com`
[2] `http://www.gnu.org/licenses/lgpl.html`

- Better collections - the collection classes offered by Qt have a richer interface and are generally more convenient compared to their alternatives from the C++ standard library.
- Unicode strings - Qt provides true Unicode character and string classes.
- XML processing - classes for working with XML files and data structures are provided and different access methods are supported (DOM and SAX).
- Excellent graphics framework - Qt's GraphicsView framework[3] is a perfect match for *Envision*'s visualization requirements. It offers a powerful toolkit for designing custom visualizations and managing huge hierarchies of graphical objects. It also supports rendering through OpenGL for optimized user experience.

3.3 Plug-in extension model

Except for a few service tasks all of the functionality of *Envision* is provided by plug-ins. A plug-in may use the services of other plug-ins and provide services that others can use. The architecture does not enforce any particular mechanism to achieve this and is very flexible in this respect.

The plug-in developer can chose how to offer services to others. Here are two suggestions which are currently used in *Envision*:

- **Using interfaces -** a plug-in might offer interface classes (abstract classes with pure virtual methods) to others which request its services. The actual implementation of the functionality is done in derived classes which are private to the plug-in. This approach is flexible: as long as the interface is fixed the implementation can change in any way and client code will not need to be recompiled or adjusted. The drawback is that code cannot be reused, since clients are only aware of the interfaces.
- **Using classes** - as an alternative a plug-in can instead export a complete class together with all other classes from its inheritance hierarchy. Clients can then further extend this hierarchy and reuse functionality from base classes. The drawback of this approach is that changes to base classes might more easily invalidate client code.

Envision uses primarily the second approach, but also employs interfaces. This is typical for C++ systems: Qt for instance, also exports complete classes which clients can inherit from and extend. This provides a more natural way to use the library. Special techniques such as using a ”d-pointer” developed by Trolltech's Arnt Gulbrandsen can mitigate the problems of this approach. This technique together with some other considerations are discussed in [18].

Regardless of what way plug-ins offer services to others. This is always done via exporting symbols. These could be classes, functions or global variables. Although there is no enforced restriction the exporting of functions or variables is strongly discouraged and not used within *Envision*. Exporting of classes (including static methods and static fields) is supported by *Envision* by using built-in functionality provided by the Qt framework. Details about exporting functionality are discussed in appendix C.

3.4 Executable

The application executable is responsible for initializing the application and starting all plug-ins. Once the application is loaded this module is mostly passive unless a plug-in requests its services.

Here are all the tasks performed on startup in order of execution:

[3] http://doc.qt.nokia.com/stable/graphicsview.html

1. The `QApplication` object of Qt is initialized.
2. All plug-ins are loaded. Plug-ins which have dependencies are only loaded after the plug-ins they depend on have been loaded. If a plug-in cannot be started due to dependency problems, *Envision* is halted with an exception.
3. After all plug-ins are loaded, any tests which were specified on the command line are enqueued.
4. The Qt event loop is started. At this point plug-ins will receive a request for self-testing if the user specified this.

As soon as a plug-in is loaded, its initialization routine is called. If any events are enqueued by the plug-ins during their initialization, these will precede the requests for self-testing. Running tests after all other plug-ins have been loaded and initialization is finished can be helpful in discovering integration problems.

As Qt requires, the main application window and the corresponding Qt event queue are part of the executable. This module itself however does not use the window in any way. Plug-ins are responsible for defining the interface of *Envision*.

3.4.1 Plug-in interface

Each *Envision* plug-in consists of at least two files:

- *Shared library* - a binary file in the format of the operating system (e.g. 'so' on Linux or 'dll' on Windows) that implements the functionality of the plug-in.
- *Meta information* - a text file in XML format that contains the plug-in's identifier, description, version number and dependency information. This format is specified in appendix A.1.

Plug-ins are discovered by searching for meta information files. The name of this file should match the name of the shared library.

The shared library interface that each plug-in must implement is specified by the executable module. This interface is defined according to Qt's `QPlugin` design principles. It includes methods for initialization and testing. When a plug-in is initialized it is given a reference to an `EnvisionManager` interface. Using this interface a plug-in can draw on the main window or query the system about the state of *Envision*. In particular it is possible to get information about what plug-ins are installed and running.

3.5 Logger

The *Logger* plug-in provides facilities to keep a system log. It exports a `Log` class that can store messages of different error levels. A client plug-in can use the log to record an event or to browse existing log events. Plug-ins (such as a visualization) can subscribe to the *Log* so that they get notified whenever a new event appears. In this way users can be notified as as soon as a new entry occurs.

3.6 SelfTest

This plug-in provides a basic unit test framework that can be used by other plug-ins to implement self-testing. Plug-in developers are encouraged to write code that tests their implementations using the services of *SelfTest*, but it is also possible to use other testing approaches.

Testing the functionality of plug-ins is a good way to assure a high quality implementation. As described in requirement TECH-3 this also makes the *Envision* system suitable for future extensions.

Clients who want to implement tests can do so with minimal efforts by using convenience macros defined within *SelfTest*. All tests for a plug-in are automatically registered with a `TestManager`. It can be used to run all tests or a specified test and report statistics. *SelfTest* offers a collection of commonly used assertions and checks and provides exception handling support.

Classes for test cases are only allocated on request and do not occupy memory if tests are not running.

3.7 ModelBase

This plug-in builds the foundation of the application model in *Envision* by implementing the corresponding requirements specified in section 2.3. It defines the structure of the model and supporting functionality. Other plug-ins can extend the classes defined here in order to provide different types of models.

The application model is the primary source that defines an application. As per requirement MODEL-2 it also contains artifacts supporting the software engineering process. The model is a tree structure as described in requirement MODEL-4 which is roughly similar to the abstract syntax tree (AST) associated with a textual program. Two major differences are that the model is not generated based on a different representation but is the original source and that it can contain arbitrary data. Similarly to an AST the different nodes in the model represent programming constructs and the edges of the tree represent containment. The root of the tree is an entity which represents the entire application that is being developed.

3.7.1 Structure of the application model

The application model in *Envision* is a tree structure where each node is derived from the base class `Node`. Each such tree structure has an associated `Model` object that manages various aspects of the model. Together these two classes form the core of the model in *Envision*.

A node in the tree corresponds to a logical element in the application under development. This could be a for instance a class, a method, a statement, a numerical value, an image, a comment, etc. A node's children are elements that define it. For example a class is defined by its name, methods, fields and others, all of which appear in the node's sub-tree. `Node` serves as a base class for all nodes and defines the minimum interface they should provide which contains the following functionality:

- Node registration - each node type (each class inheriting from `Node`) has to be registered in the system before it can be used. This is typically done during the initialization of the plug-in where this new node type is declared.
- Tree navigation - nodes can navigate to their parents, children and to other nodes. A number of convenience methods provide different ways to do this.

- Persistence - a node should be able to store or load its contents. Such operations involve an object which implements the `PersistentStore` interface. An implementation can store the node to disk, data base or any other medium.
- Modification - to enable undoing and redoing of node operations, modification happens in a controlled manner based on the command pattern.
- Meta information - nodes have various properties such as class name, class id and revision which are required for the correct operation of the system.

Whenever possible default behavior is already implemented for the functions of `Node` to simplify development of new node types.

The `Model` class implements functionality for managing a single model tree. A `Model` object might not be associated with a tree, but a tree always corresponds to exactly one such object. Here are the management functions that it provides:

- Tree creation - A newly constructed model does not yet have an associated tree. The `Model` object can be used to construct the root node of the model.
- Saving and loading - the `Model` coordinates the saving and loading of applications using a `PersistentStore`. This is described in more detail in section 3.7.5.
- Access control - In order to assure correct behavior of the system when there is concurrent access to the model tree, the access is controlled in the `Model` object. The exact mechanism is discussed in section 3.7.4.
- Undo support - this object keeps a history of previous commands and provides methods that restore the tree structure to a known previous state. See section 3.7.6 for more details.

3.7.2 Predefined node types

Basic node types which are commonly used in programming are defined in *ModelBase* for convenience. Plug-in developers are encouraged to use these types, but this is not an absolute requirement.

The predefined types include:

- `Integer` is a 32 bit signed integer.
- `Boolean` is a boolean value.
- `Float` is an 80 bit double precision floating point value.
- `Character` is a Unicode character.
- `Text` is a Unicode string.
- `Reference` is a node that contains a string reference to another node. The meaning of the string and how to resolve references are defined individually by each node type.
- `List` is a node that contains an ordered list of nodes. This list is untyped.
- `TypedList` is a thin wrapper around `List` that provides a statically typed equivalent to it.
- `ExtendableNode` is a node that can be extended to contain children of arbitrary type. Plug-in developers can use the functionality of this node to quickly design new node types. This is discussed in more detail in section 3.7.3.

3.7.3 Node extensibility

In order to create new types of nodes, a plug-in developer can inherit an existing predefined node or `Node` directly. The `ExtendableNode` class is specifically designed for implementing new custom nodes. It provides two main features to support extensibility:

A class inheriting from `ExtendableNode` can specify what attributes (child nodes) it will have. The number and type of attributes can be arbitrary. A longer inheritance chain is also possible, where each derived class specifies additional child nodes. Attributes defined in base classes are also available in derived classes.

Nodes deriving from `ExtendableNode` also support an extension mechanism. An extension can be dynamically registered for a node type at run-time as long as no instances of this class have been created yet. The attributes specified in the extension will be added to the attributes that are already defined in the node type being extended. This change will apply to all nodes that derive from the extended class. This feature can be used by plug-ins to attach new information to a model tree.

Details about the implementation of `ExtendbaleNode` can be found in appendix F.

Figure 3.2 illustrates these mechanisms with an example. A node type `BinaryNode` is defined that inherits from `ExtendableNode` and specifies two attributes: `left` and `right`. The `IntegerBinaryNode` derives from `BinaryNode` and introduces an additional attribute - `value`. In addition, the `Position` extension has been registered for `BinaryNode`. This inserts the attributes `x` and `y` into `BinaryNode` and all its subclasses - in this case only `IntegerBinaryNode`. Finally one more extension has been registered for `IntegerBinaryNode` which provides that node type with the new attribute `editable`. On the right we see a summary of what attributes are available for the two different classes.

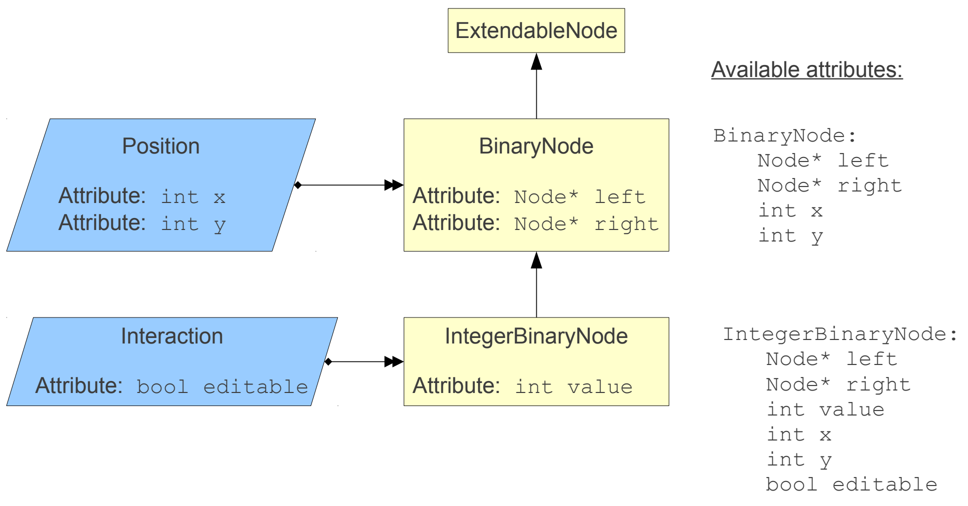


Figure 3.2: A binary node hierarchy with extensions.

There is no limit to how many extensions a node can have. Extensions are implemented as single classes and can be applied to any node that inherits directly or indirectly from `ExtendableNode`. A simple method is provided for asking if an extension has been registered for a node type and accessing its corresponding attributes.

Persistence operations are automatically handled for any class deriving from `ExtendableNode` for all attributes and attributes of extensions.

This generic node support greatly simplifies the task of extending a model by adding new types of nodes. The developer can focus on important aspects, namely what attributes and what operations are available when defining new nodes. Trivial functionality is handled automatically.

3.7.4 Concurrent access

In today's software engineering practices developers use many different tools and techniques to be more productive and to create high quality applications. Many of these tools work with the source code or the application binary. This includes compilation, testing, verification, profiling, and others. Modern IDEs are capable of performing some of these tasks in parallel in order to speed up development. For example Eclipse can automatically compile a Java source file while the user is typing. This allows the environment to show compilation mistakes or warnings immediately after the programmer writes the lines that generate them. This is an excellent feature that helps quickly eliminate trivial problems in the source code, without explicitly compiling the program first. Another feature popular among programmers is auto-complete - the ability of an IDE to suggest a complete symbol name based on just a few characters typed by the developer. Enabling this is a background process that keeps an up-to-date table of defined symbols by constantly parsing the source code and reading in new definitions. In some cases, such as the Spec# language in Visual Studio, the IDE even performs background verification of contracts while the user is typing.

Receiving information about the program such as compilation and verification errors, testing statistics, profiling reports, etc. in real time can be a huge boost to productivity and software quality. Therefore having a mechanism that enables concurrent access to the source code or the application model in the case of *Envision* is crucial. The *ModelBase* plug-in implements a such a mechanism that controls the access to the tree structure of a model. For maximum flexibility a number of different ways to access the tree exist. This allows tools with different needs to use an appropriate access mode and optimally utilize resources.

The different access modes to an application model are presented below:

Write - only one thread at a time is allowed to have write access to the tree structure. This assures that modifying threads will operate in a model that will stay in a consistent state. This is typically the thread which processes user input and acts on the tree structure accordingly.

Global read - Any number of threads can simultaneously have global read access to the tree. The system guarantees that the tree will not be modified while there are active global readers. Any thread that needs to modify the tree will have to wait until all global readers have finished their operation. This mode can be used for threads which must operate on a consistent data structure throughout the entire model. Such tasks include persistence, compilation and others.

Local read - Local readers can run in parallel with threads that modify the tree state. To better manage this process there are two types of local read access:

- **Blocking** - these readers lock a part of the tree called an access unit. Any number of readers can read the same access unit, but a writer has to wait for all blocking readers to finish their execution before it is granted permission to modify that access unit. A blocking local reader can simultaneously operate on multiple access units. This access mode is designed for background tasks which work with the model but should not hinder the programmer from manipulating it. Especially when coupled with the `InterruptibleThread` thread class described below, the blocking access mode is a good choice for tasks such as symbol indexing, type-checking, verification or collection of statistics.
- **Non-synchronized** - such a reading thread does not use the built-in synchronization mechanism and may operate in parallel with any other thread on the same access unit. The system provides no guarantees about the consistency of nodes read by such a reader. This access mode should be used with caution and is designed to be used only in cases where other synchronization means are employed.

The concept of access units and the implementation of synchronization mechanisms in the tree model in *Envision* are illustrated in figure 3.3. All synchronization is achieved by means of Reader-Writer locks provided by the Qt framework. Two such locks exist in the `Model` object corresponding to the tree. Optionally a node can declare that it is a new access unit and thus contains a lock of its own. When a node contains a lock the corresponding access unit spans that node and all its children reachable without passing through another node which contains a lock. In figure 3.3, the nodes A, B, C, D, E and F contain a lock. The corresponding access units are colored in orange. If the root node does not contain a lock then the root access unit lock inside the `Model` is used to define the root access unit as shown. Otherwise that lock is not used at all.

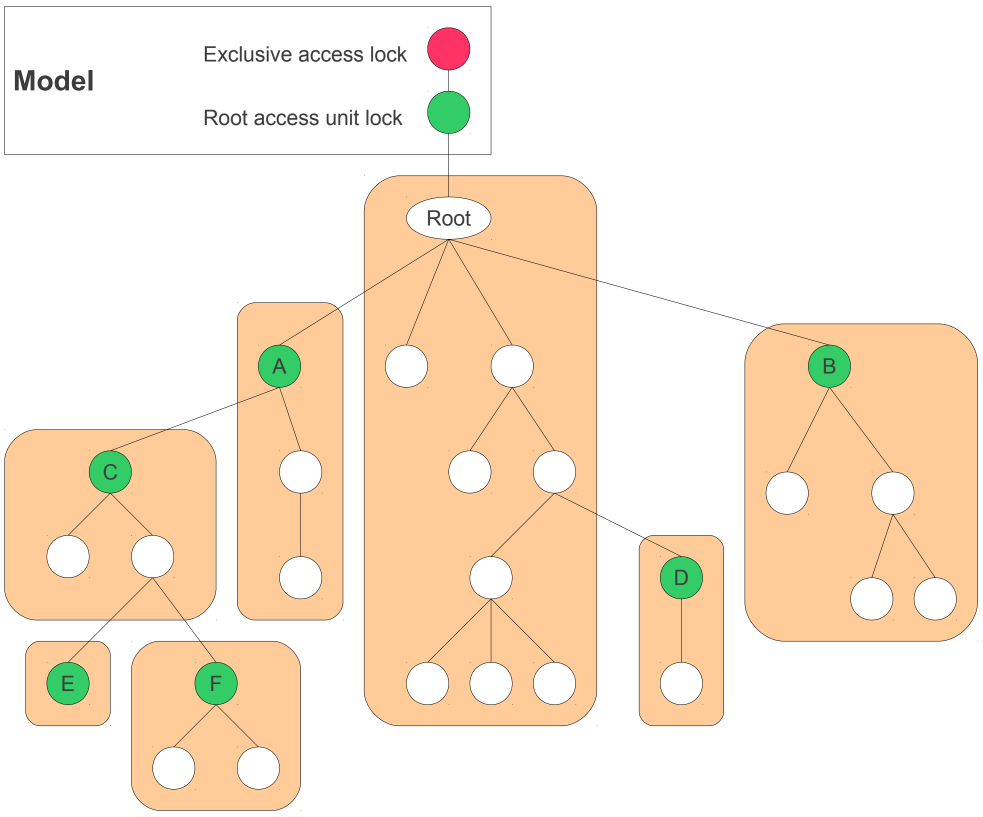


Figure 3.3: An example of an abstract model with access locks.

A thread that wants to write must obtain the exclusive access lock inside the `Model`. To do this a plug-in needs to begin a modification session by calling the `beginModification` method on a model object. This grants the plug-in write access. Afterwards it can modify any node belonging to a single access unit. During a modification session the plug-in can request access to a different access unit. A writing thread must therefore also obtain the lock for that unit. To avoid deadlocks in this case it must first release the access lock it currently holds. This behavior is enforced by the system. A writing thread which tries to acquire the locks of two different access units at the same time will be terminated with an exception. The exclusive access lock can be released once the thread is finished modifying the application tree by calling `endModification`. A deadlock situation can only occur if a thread tries to acquire both read-only and write access permissions. Plug-ins which need to modify the model should directly initiate a write session which also allows the reading of tree nodes.

To receive global read access a thread acquires the exclusive access lock in the model thereby preventing any writers from accessing the tree. Such a thread does not acquire any other locks. Global access is granted in the form of an access session that can be started by calling `beginExclusiveRead` on a model object and can be terminated with `endExclusiveRead`. There is no possibility of a deadlock since all readers can run in parallel, and writers do not hold any locks while global readers are executing.

Blocking local readers should only acquire a lock corresponding to the access unit containing the nodes they wish to read. The lock that defines the access unit of a node can be obtained by calling the node's `accessLock` method. It is possible to acquire the locks of multiple access units. There is no danger of a deadlock since multiple readers can hold the same lock at the same time, and a writer can only hold one such lock at any time.

A blocking reader can optionally implement the `InterruptibleThread` interface. When a writer thread tries to acquire a lock which currently belongs to an `InterruptibleThread` this thread will be signaled that there is a writer waiting. The reader may then decide to yield the lock and let the writer modify the tree. This supports the running of long operations such as verification, without blocking the responsiveness of the environment to normal user interaction.

Non-synchronized readers do not acquire any locks. Developers should be careful not to read tree nodes in an inconsistent state. This access mode is provided for use in cases where other synchronization means are used.

3.7.5 Persistence interface

Each node of the model supports an interface for saving and loading itself to and from a persistent store. This is a minimal interface that can work with integers, strings, floating point numbers and composite nodes. It is easy for plug-ins to provide implementations of this interface that work with specific storage media: files, databases, version control systems, etc.

To reduce run-time memory load a node can support partial loading as defined in requirement PERS-5. A partially loaded node may load only a subset of its children from the persistent store. The node can be fully loaded at any time later when all of its subtree is needed. This is useful for example to skip the loading of method bodies, unless the user specifically wants to look at the implementation of a method.

In order to enable granular management of the persistent store a node can declare that it is a persistence unit. For example a node representing classes might be declared to be a persistence unit which could signal to a file persistence store that each class should be saved in a separate file. In general being a persistence unit is only a hint for the underlying store. It can also be ignored which is the case, for instance, when copying nodes to clipboard.

3.7.6 Undo

The *ModelBase* plug-in provides undo functionality for tree operations. Following the command pattern each operation on the tree is executed as a command that can undo its effect. Of course once a command is undone it can be redone again. The command stack is part of the `Model` object managing each tree.

Only writer threads are allowed to execute commands which modify the tree and only writer threads can request and undo or a redo operation. Each modification command is associated with a specific node. A writer thread must have acquired the lock corresponding to the access unit of that node. The interface of `Model` provides functionality for initiating and ending a write session and for changing the current write lock.

3.7.7 Signals

Client plug-ins can subscribe to a number of signals that are emitted by a *Model* object in order to receive notifications about model events. This is done using Qt's signals and slots mechanism. Signals emitted include:

- `rootCreated` - the specified node was just created as a root node for this model. A visualization plug-in can use this signal in order to start visualizing the newly created model.
- `nodesModified` - the specified nodes were modified. This signal is emitted after a part of the model tree has been modified. Many different plug-ins have interest in this event. For example a visualization plug-in should examine the list of nodes that have been modified and update their representation. A verification plug-in can examine the modifications and look for semantic errors. An incremental compiler might compile the new version of the model in the background.
- `nameModified` - the name of a symbol definition node was modified. This can be used for refactoring and consistency purposes. For example, a plug-in that keeps the tree in a consistent state can find all references to this node and update the symbol they use to the new name.
- `nodeFullyLoaded/nodePartiallyLoaded` - the specified node was just fully/partially loaded. This can be used by a plug-in to update the node's visualization or to persist its state.

3.8 FilePersistence

This plug-in implements two XML based persistent stores for application models.

With the help of the `FileStore` class a model can be stored as files on disk, where each persistence unit is saved to a different file. The precise data format is described in appendix A.2.

The `SystemClipboard` class provides access to the system clipboard. It offers functionality to copy a node or a list of nodes to the clipboard. Nodes are copied as an XML text which is described in appendix A.3. Nodes from the clipboard can be pasted into a node of a list type or can be used to replace other nodes in the model.

3.9 VisualizationBase

This plug-in sets the foundations for model visualization in *Envision*. It is based on Qt's Graphics View[4] framework. In that framework a `QGraphicsScene` is a virtual canvas where `QGraphicsItem` objects are drawn. A `QGraphicsView` is used to visualize the scene or parts of it.

VisualizationBase defines classes built upon the functionality of the Graphics View framework to provide essential features for visualizing and navigating application models. A number of extensibility concepts are also implemented which make it easy to add custom functionality on top of it.

3.9.1 Scenes

A `Scene` is a data structure that contains the visual representation of one or more models and possibly other graphical entities. It is a virtual canvas that is not visible itself but serves as the

[4]http://doc.qt.nokia.com/stable/graphicsview.html

source for one or more views. Objects on this canvas are instances of `Item` or derived classes. Items which are part of a model visualization are structured in a hierarchy similar to the one of the model tree. Other graphical objects can have an arbitrary structure.

A `Scene` has an associated `ModelRenderer` which is used to construct a visualization(instance of `Item`) from an arbitrary node in a model. Classes deriving from `Item` which represent a node must be registered with a `ModelRenderer` indicating what node type they can depict. The renderer is then used to draw visualizations of a model or of individual nodes. It is possible to have multiple renderers and switch between them in order to use different sets of visualizations.

The scene accepts user input events from associated views and propagates them to the appropriate item (e.g. the item under the cursor or the focused item). Each scene also has a default `SceneHandlerItem`. User actions which are not processed by any other item on the scene will be forwarded to this object. Default interaction mechanisms can be implemented there. The scene is automatically updated after the user interacts with any of its associated views.

Items on the scene need to be updated whenever a change in the model or in a visual representation occurs. In order to avoid unnecessary updates which are costly this process has been optimized to only adjust affected visual elements. The mechanism relies on the hierarchy of items on the scene. Whenever the visualization of an item becomes invalid, only it and all of its ancestors will be updated. Siblings which explicitly indicate that their visualization depends on the one of their parent, may also be updated.

3.9.2 Views

To display the contents of a scene a `View` widget must be placed on the main screen of *Envision*. Multiple views can be used to show the same scene from different viewpoints simultaneously. Views can visualize the scene using a variety of transformations such as scaling, rotation and translation. Views support zooming in and out, moving to different parts of the scene and also provide scene interaction.

A view can be either a master view (i.e. has no parent) or a child view which must specify a parent view. The position and size on screen of child views are controlled by their parent. Master views are managed by the `VisualizationManager` singleton. It is responsible for positioning and resizing the views within *Envision*'s main window.

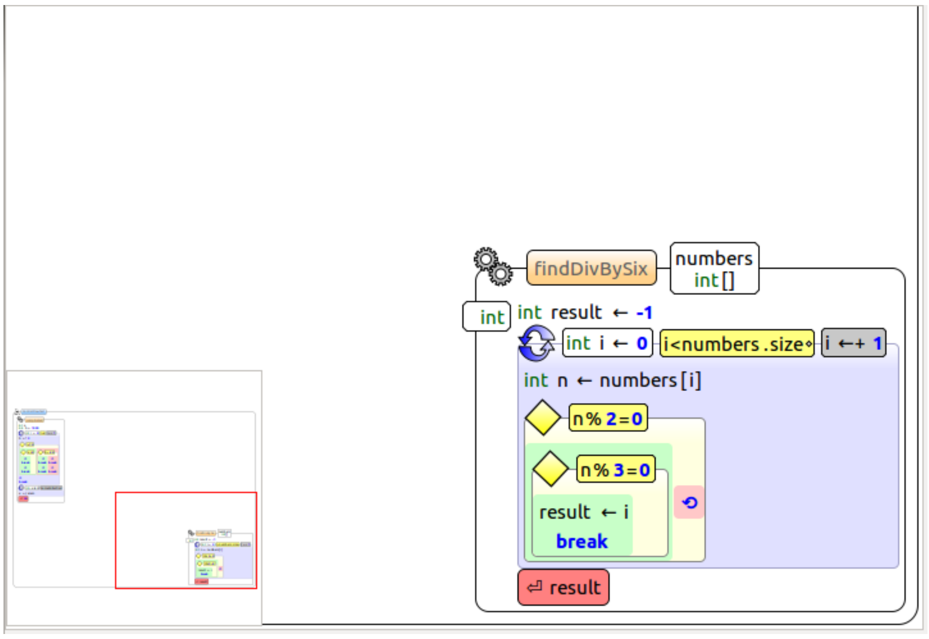
VisualizationBase provides a default view which is an instance of the scheme above. The `MainView` master view provides two distinct features: a fixed step zoom behavior controlled by the mouse wheel and a mini map navigation. The mini map navigation is implemented in the child view `MiniMap` which at all times shows the entire scene and outlines in red the region currently displayed by its parent. The user can navigate the main view by clicking or dragging within the mini map located in the lower left corner.

Figure 3.4 shows a screen shot of the two views. Here only part of the canvas is visible and the mini map helps the user to stay oriented.

3.9.3 Items

A central concept in *VisualizationBase* is the item. Items are the building blocks of the visualizations in *Envision*. An item belongs to a scene and can draw anything on it: text, shapes, images, etc. Items can also have child items thus allowing a natural hierarchical structure.

Items in *Envision* inherit from the `Item` class. It implements facilities to configure the geometry of an item, to perform updates, to communicate with the scene, to process input events and to handle the item's style and shape. Subclasses are responsible for specifying what should be drawn and what children, if any, an item has.



```
findDivBySix  numbers
              int[]
int  int result ← -1
     int i ← 0  i<numbers.size  i ←+ 1
     int n ← numbers[i]
     n%2=0
       n%3=0
         result ← i
         break
     ⏎ result
```

Figure 3.4: Mini-map example.

Generally items are used to visualize nodes from the application model. Some items directly correspond to a node (inherit from `ItemWithNode`), while others perform service functions in the visualization. For example a method item corresponds to a construct from the application tree. Other items, such as icons, layouts, lines, connectors, etc. might be part of a method's visualization but are only used to achieve a certain look are not directly associated with a node. Typically one of the ancestors of these items is an `ItemWithNode`.

Each item has a style. A style is a collection of properties which define how an item should be drawn. This could include for example, a color scheme for the item, positions for children items, a style for children items and others. Styles allow for the customization of most aspects of the visualization without the need to understand or modify program source code, therefore simplifying the task of experimenting with different appearances.

The developer of a subclass of `Item` decides what visualization parameters the new class should contain. During the construction of an `Item` or at a later time a particular style can be set that defines the values of all required properties. *VisualizationBase* provides a convenient way of instantiating styles based on XML files. Defining a specific look and feel in such a way is very flexible: the files provide reuse mechanisms which make it easy to create a family of styles that have a common base. This flexibility is granular and can be applied to an entire set of properties or singe parameters. Using this built-in mechanism provided by the singleton `Styles` a single style object will be shared among all items which share the same properties. The format of style files and more details on how to use them are discussed in appendix A.4.

Specified in each item's style is an optional shape. A `Shape` is a static image drawn behind the item, such as for example a box. By default the item's content is fully enclosed by the shape and appears on top of it, but this behavior can be overridden if desired. Changing the appearance of the shape or even substituting a shape for another one can be achieved easily by modifying the item's style file. A shape itself has properties which are also defined there. Shapes already defined in *VisualizationBase* include a `Box` and a `Diamond`.

Interaction

Interaction with the scene is performed at the item level. When the user presses a key on the keyboard, moves the mouse or clicks a mouse button, a corresponding event is sent to the scene which then dispatches it to the appropriate `Item` object. This is for example the top-most visible object under the cursor or the focused item for keyboard events.

An item can either handle or ignore an event. By default all keyboard and most mouse events are ignored. An ignored mouse event is propagated to the the next visible item under the current one. Some basic selection behavior is enabled for all items. All default mechanisms are provided by Qt.

VisualizationBase also introduces a 'command' handling mechanism for all items. Each item can receive a string command and execute it. The exact mechanism is not specified in *Visualization-Base*, but rather only the necessary infrastructure is created.

To enable more advanced item interaction, event and command processing for each item is performed by an `InteractionHandler` object. Such an object is associated with each item class. The default implementation simply ignores commands and forwards events to the underlying Qt event handling methods. Plug-in developers can create new handlers and replace the default one in order to allow custom interaction. A handler can override the behavior for mouse, keyboard and command events.

Predefined items

A number of items which are frequently used in visualizations are provided for convenience by *VisualizationBase*:

`TextRenderer` - This is an abstract item that can be used to render any Unicode string on screen. Any other item which has an inherent textual representation can inherit from `TextRenderer` and use the facilities it provides which include text selection and modification (in conjunction with an appropriate interaction handler). This is suitable for literals, identifiers, keywords, comments, special symbols, etc. `TextRenderer` supports a rich style definition, that allows the configuration of the font, size, mode, color and background in both the normal and the selected state.

`Text` - A thin wrapper around `TextRenderer` that just renders some text on screen. The text to display can be set at run-time and may optionally be editable.

`Symbol` - An item that renders static text on screen. The text to render is determined by the style file of the item. This is useful for defining the visualization of 'fixed' items, such as unary and binary operator symbols, keywords, etc.

`LayoutProvider` - This item makes it easy to design a new visualization based on a layout. Typically a client will inherit from `LayoutProvider` specifying a particular layout to use. `LayoutProvider` will automatically take care of all drawing and service functions and the derived class must only populate the layout with children elements.

`SVGIcon` - This item displays the content of an SVG file as an icon. The file to use and the icon size are specified in the item's style.

`VBoolean`, `VCharacter`, `VInteger`, `VFloat`, `VText`, `VReference`, `VList`, `VExtendbale` - As a convention item names of the form ***V****node* are used when they visualize the correspondingly named node. Thus these items are used to show the similarly named nodes defined in *ModelBase*. Most of these items are thin wrappers around `TextRenderer`. `VList` uses a `SequentialLayout` and shows the contents of a `List` object. `VExtendable` is based on a `PanelBorderLayout` layout and serves as a generic visualization for any class deriving from `ExtendableNode`. It lists all the attributes of such an item as a pair of attribute name and attribute visualization per row.

Figure 3.5 is a screen shot from *Envision* showing how some of the predefined items are visualized. It displays a list with a gray background which consists of three items. The first two items are `VExtendable` objects representing a simple `BinaryNode` node. The last item is a `VText` object rendering of a `Text` node. A binary node has a name, and optionally a left and right child nodes. `VExtendable` shows all attributes of the node it represents as a vertical list of pairs. In gray we see on the left the name of the attribute. On the right we see the visualization of the attribute value. Since `VExtendable` can be used to visualize any node that inherits from `ExtendableNode` there needs to be away to know exactly what node type is being visualized. The name of the node type appears in gray in the top bar of the visualization. In case there is an attribute called 'name' `VExtendable` puts this in the title before the type name, instead of the normal list of attributes. Thus in figure 3.5 the first two elements of the list are called 'First node' and 'Empty node'. Furthermore we see that both children elements of 'First node' are present and are called 'left node' and 'right node'. All instances of `VExtendable` represent `BinaryNode` objects as indicated by their type in gray text.

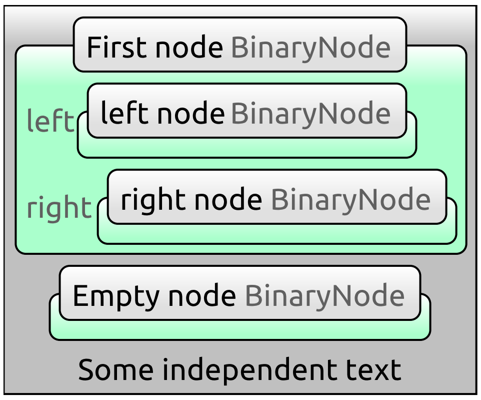


Figure 3.5: The default visualization of generic items.

Layouts

Layouts are items whose only purpose is to contain other items and manage their position on screen. A composite item can use one or several layouts to simplify the management of children. For example the visualization of a node representing a class can contain a layout that displays all of the class' methods in a linear sequence, or in a grid.

Several common layouts are provided by `VisualizationBase`. Most of the options that control aspects of their visualizations are configurable in styles. Only more interesting options will be mentioned in the discussion below.

`SequentialLayout` - This layout arranges its children items in a linear sequence. Its interface is similar to a collection, items can be inserted and removed anywhere in the layout. The way the items are visualized depends on the layout's style and includes options for ordering, orientation, alignment and spacing. Figure 3.6 shows four instances of a sequential layout visualizing the same items (**A**,**B** and **C**) with different style settings.

This layout is useful for arranging objects which logically belong to an ordered sequence, such as the statements of a method.

`PanelLayout` - This layout visualizes a linear sequence of up to three items called 'first', 'middle' and 'last'. The layout will stretch along the sequence direction in order to fill any space that the parent reports as available. The 'first' item will be placed in the beginning of the space available to the layout, the 'middle' one at the center and the 'last' item will be located at the end. Figure

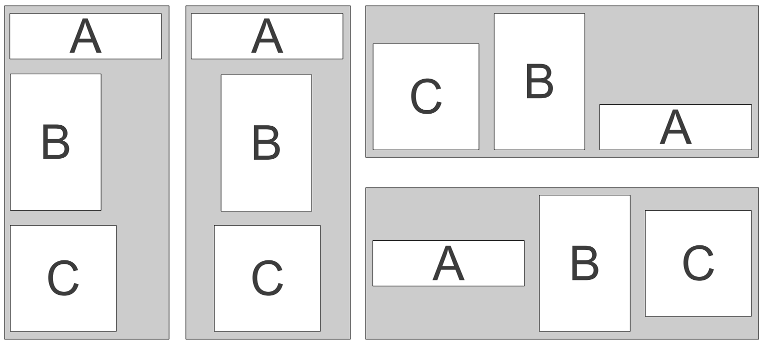


Figure 3.6: A sequential layout with different orientations, item orderings and alignments.

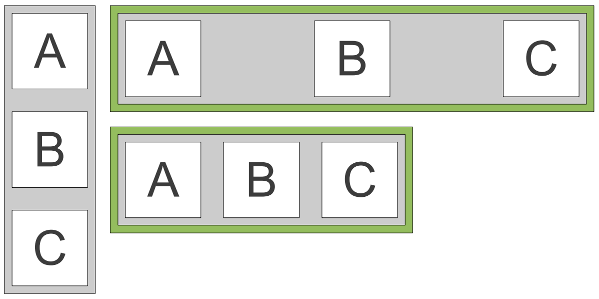


Figure 3.7: A panel layout. Available space reported by the parent is indicated in green. If the parent reports more available space than required, the layout will stretch to fill this space.

3.7 illustrates this concept with a few examples.

This layout is useful for headers of objects. For example an icon can be placed on the left, while the object title appears in the middle.

`PanelBorderLayout` - This is a complex layout that consists of one item in the middle, surrounded by four instances of `PanelLayout` that serve as a border. The objects on the border are optional. Figure 3.8 depicts the look of this item.

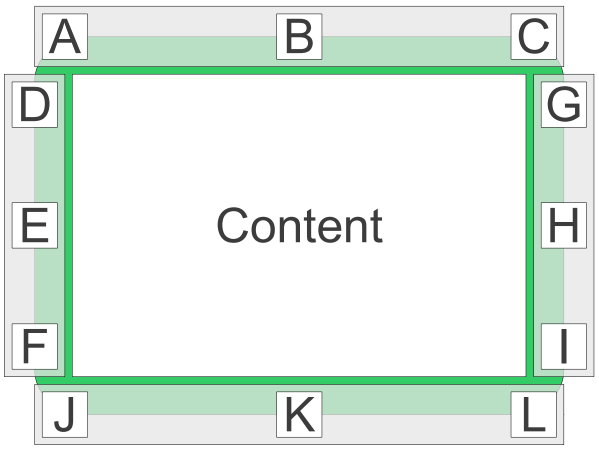


Figure 3.8: A panel border layout. The background shape is shown in green.

`PanelBorderLayout` can also override the default drawing of its background shape, if any. In this case the shape is not drawn to encompass all objects in the layout, but rather goes through the middle of the panel layouts on the border.

This layout is useful for displaying objects which represent higher level composite entities. For example classes or methods visualized in such a way can include an icon and a name in the top bar while having their content displayed in the middle. Additional indicators can be placed on the side bars.

`GridLayout` - Items in this layout are arranged in a grid as demonstrated on figure 3.9. It is possible to adjust the horizontal and vertical alignment of items as well as the space between them. For maximum convenience it is possible to have empty grid spaces.

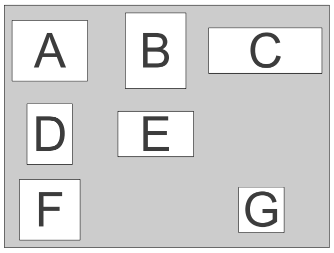


Figure 3.9: A grid layout where the contained items are horizontally and vertically centered. Not all grid positions are filled.

This layout can be used to show a group of items more concisely compared to a `SequentialLayout`. Some constructs such as matrices are also naturally represented by a grid arrangement.

`PositionLayout` - This layout arranges items which contain a position. Visualizations which can be used in this layout must represents nodes that have the `Position` extension. This extension class is defined in *VisualizationBase* and can be registered for any node type that inherits from `ExtendableNode`. It provides x and y coordinates that define the position of the node. Figure 3.10 shows an example visualization of three items.

This layout is suitable for the rendering of unordered sets or lists, such as a collection of classes or methods.

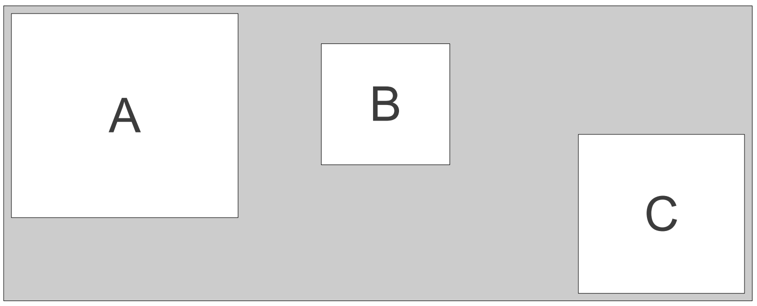


Figure 3.10: A position layout. The location of each item is determined from its `x` and `y` coordinates stored in the application model.

3.10 InteractionBase

This plug-in defines the basis of the interaction mechanisms available in *Envision*. It implements the corresponding requirements described in section 2.6. Interaction refers to the activity where the user manipulates the visualizations on the canvas and the underlying application model. So far two ways of achieving this have been conceptualized and a preliminary implementation is provided by *InteractionBase*.

The first way is by direct manipulation of the visualizations - an approach common to many visual programming environments. As defined in requirements INTER-1 and INTER-3, direct manipulation includes actions such as selecting and moving items, copying and pasting objects, editing text fields or invoking context menu functions. The merits of such an interaction are that users stay focused on a single item and view, that they get immediate feedback of their actions and that it feels natural to manipulate visual representations in a direct way therefore simplifying many tasks. This technique is widely used in other visual professional systems such as image and diagram editing tools, CAD programs and modeling applications.

On a technical level, what direct manipulation actions have in common is that they are directly associated with user input from the mouse or keyboard. Each item class has an associated *handler* that controls how user actions should be interpreted. By creating new handlers or extending existing ones the interaction behavior of items can be changed. Handlers can be assigned to item classes at run time.

The second way of manipulating the application model is by issuing textual commands to items. This is an implementation of the requirements for keyboard-centric interaction (INTER-7, INTER-8 and INTER-9) in *Envision*. The motivation is that experienced developers will be more productive when provided with text based input mechanisms. This is further discussed in section 3.10.2.

The support for command based input in *InteractionBase* is provided by a visual prompt object which allows the user to issue a command to any item on the scene. Textual commands however need not be issued by the user and can be also be executed programmatically.

3.10.1 Interaction handlers for items

Each class inheriting from `Item` provides an interface to set the class' interaction handler. An implementation of a handler is simply a class that inherits from `InteractionHandler` and overrides one or more of the virtual methods which process specific events such as, for instance `keyPressEvent` or `mouseReleaseEvent`. `InteractionHandler`'s default behavior is to execute the standard Qt action for any event.

InteractionBase provides custom handlers for all of the item types defined in *VisualizationBase*. Most functionality is implemented in `GenericHandler` and all other handlers inherit from it. It provides the following features:

- Command list - this handler contains a list of commands which are available for any item that uses it. New commands can be added to the list at any time. See section 3.10.2 for more details on commands.
- Command prompt invocation - any item which uses this handler can display a command prompt on the screen. This is discussed in more detail in section 3.10.2.
- Copy and Paste - this handler allows the copying of any number of nodes into clipboard. The generic paste functionality is limited to replacing a selected node with one from the clipboard. More sophisticated paste behavior must be implemented in handlers designed for specific item types, such as a list.

- Undo and redo - it is possible to rewind and replay changes to the application model.
- Spatial navigation - this handler provides a framework for navigation between items in a two dimensional space. Using the arrow keys of the keyboard a user can change the current focus to an adjacent item. Items which contain other items must implement a mechanism to allow the correct child to be selected when navigating in this way.
- Item selection - the generic handler allows items to be selected when the user drags the mouse around them. Which items are selected depends on the structure of the visualization and the hierarchy of the underlying application model.

Here is a list of the other handlers provided by *InteractionBase*

`HText` - handles interaction with items based on `TextRenderer`. It allows the text to be edited in a natural way and change the underlying node if any. The changes to the text are filtered through the item's `setText` method. For example typing a character in a text field which represents an integer will have no effect.

`HList` - this handler can process events for `List` items and derivatives. In particular it provides copy and paste functionality so that a collection of items can be copied into the list.

`HExtendable` - this handler can be used with a `VExtendable` visualization and allows the collapsing of the visualization to a one line header, containing only the name and type of the underlying node.

`HPositionLayout` - allows items in a `PositionLayout` to be moved.

`HSceneHandleritem` - implements a handler for the default scene item. It will receive events from the scene when they are ignored by all other items. This handler contains the top-level commands available in *Envision* such as 'exit' for closing the application.

`HCommandPrompt` - handles interaction in a `CommandPrompt` object. It processes key presses in order to update the suggested commands list (auto-complete) or to execute a command.

3.10.2 Interaction through textual commands

It is typical for visual programming environments to primarily use interaction methods focused on the mouse and direct object manipulation. In doing so, often there is no means of textual or keyboard input. An alternative that is sometimes present in such environments is a separate view with purely textual representation of the program that allows modification which updates the main graphical view. This however voids the direct manipulation advantage of graphical environments and requires the user to work with two different notations at the same time.

We believe that being able to create programs using mostly the keyboard is a must for professional programmers that want high productivity. Thus we provide an alternative approach to manipulating visual objects - using textual commands. The idea is that the user can work with a visual canvas and request a small command prompt to appear that is associated with the currently selected object. This prompt appears close to that object so that the user's attention is not distracted. In this prompt it will be possible to write commands which manipulate the selected object or the model in general. Therefore the sense of direct manipulation is still preserved while the user can quickly accomplish his or her task.

Here is an example that illustrates the power of this approach. Imagine a canvas with some geometrical shapes. One of the shapes is a circle which the user needs to resize to a specific new width and height. Here are possible methods to achieve this:

- Use the mouse and drag one of the corners of the circle's bounding rectangle to reach the desired size.

  - **Advantages:** Single view, direct manipulation of the object. It does not matter how many other properties an object may have.
  - **Drawbacks:** Using the mouse might not be possible in situations where the new size cannot be matched to pixels on screen. Additionally dragging the mouse to a specific position may be a slow action and decrease productivity if done often.

- Open a dialog that contains all the properties of the circle and change the value in the size field.
  - **Advantages:** No precise mouse movement necessary. The size of the circle can easily be set to the exact value desired by the user.
  - **Drawbacks:** The dialog is similar to a second view of the object and could break the user's focus. Dialogs which include a lot of options and tabs might require significant time to find the desired setting.
- Open a purely textual representation of the circle and modify its size by changing the text.
  - **Advantages:** Like above, no precise mouse movements and the exact size can be set.
  - **Drawbacks:** The user needs to understand the format of the textual representation of the circle. Working with two alternative representations will decrease the user's focus. If the circle has many properties its textual representation will be more complex and harder to work with.
- Use textual commands: select the circle and press a key to open a command prompt for it. Then just type a command to change it's size. e.g: "resize 42 10.25".
  - **Advantages:** No precise mouse movements required. The exact size can be set quickly without having to open a dialog or change to a different view and the number of properties an object may have plays no role. The object is manipulated directly and quickly, thus the user stays focused on the current task.
  - **Drawbacks:** The user must know the command that is needed to resize a circle.

This is just a simple situation that illustrates the potential of this manipulation technique. As we can see the more properties an object has and the more precise its manipulation needs to be, the more we can benefit from a command based approach. When we are dealing with the structure of a program, elements are complex and require precise modification. Our approach is an excellent match for this case.

When using a more sophisticated model it is possible to have more expressive commands (akin to shell commands in operating systems) that perform complex modifications in a single step. Coupled with spatial navigation using the keyboard arrow keys it is possible to avoid the requirement of using the mouse to select an object for manipulation, thus further increasing productivity.

The drawback of a command based approach is that users need to know what commands are available and how to use them. Existing techniques such as auto-completion or self-documenting commands can help reduce this problem. Ultimately a programmer experienced in using *Envision* will learn how to use the available commands in a fast and efficient way and will not require additional help.

Processing of commands is built-in in *Envision*. *VisualizationBase* requires that each item's interface provides a method for processing string commands. A framework for doing this is implemented in *InteractionBase*. All handlers inheriting from `GenericHandler` contain a list of commands that are available for any item type associated with such a handler. In addition `GenericHandler` can display a command prompt for the currently focused item. The prompt and the commands themselves are discussed in more detail next.

Command execution

A command is a derived implementation of the `Command` abstract class. The main facility it provides is to perform an action based on a list of text tokens. A plug-in programmer can also ask a command if it can process a specific list of tokens before attempting this. Additionally commands can provide help to the user in three ways: by reporting different forms of the command, by auto-completing command text and by providing a detailed description and usage instructions.

It is worth noting that suggestions for auto-completion and command help can be arbitrary visual objects that take full advantage of facilities provided by *Envision*. Of course visual auto-complete suggestions must be accompanied by a corresponding command text.

Command execution is controlled by a `CommandExecutionEngine`. When a command is sent to an item, it invokes the execution method of its associated handler. `GenericHandler` further delegates this to an execution engine, which could be customized. The default engine executes the command in the following way:

1. Verifies that any quote symbols are properly matched. If not returns an error.
2. Uses the navigation prefix of a command (if any) to navigate to the indicated visual item, starting from the currently selected item. This new item becomes the target of the command. This is similar to using "." and ".." when working with file systems.
3. Extracts a list of tokens from the command.
4. Tries to find a command in the command list of the target that can process the specified tokens. This process is repeated for all ancestors of the target item until the command is processed or until there are no more ancestors.
5. If the command was not processed by any item then it will be passed to the default scene item.
6. At the end if the command was not processed an error will be returned, otherwise the result of the command will be returned.

This behavior can be altered by sub-classing `CommandExecutionEngine` and setting the new engine in `GenericHandler`.

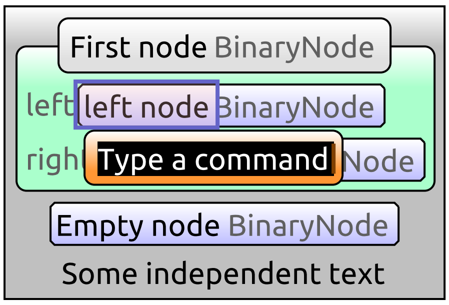
The result of executing a command (also in case of an error) is a `CommandResult` object. This object contains a visual representation of what went wrong and can be of arbitrary content. It can include explanations, suggestions, links, etc.

Command prompt

The command prompt is a visual item that can be used to enter textual commands for any other item on the scene. It features a text field where the user can enter commands to be executed. The prompt will automatically suggest suitable ways to complete the current command on every keystroke. The suggestions are based on what commands the current item can process and are obtained with the help of the execution engine.

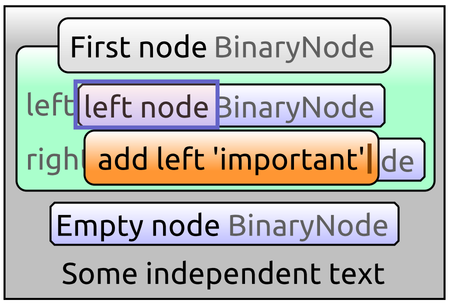
When the user executes a command, the prompt will disappear or, in case the command failed, will display the error messages of the command. A command error message might be accompanied with suggestions on how to change the command in order for it to execute properly. The user can easily pick a suggestion to execute using the keyboard.

Figures 3.11 through 3.14 show the command prompt in various situations.



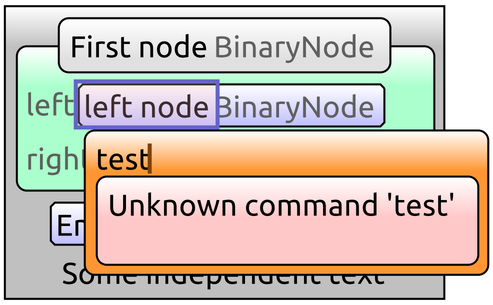
Type a command

Figure 3.11: A command prompt has just been shown. It invites the user to start typing a command. The command will be executed for the selected item, shown with blue outline.



add left 'important'

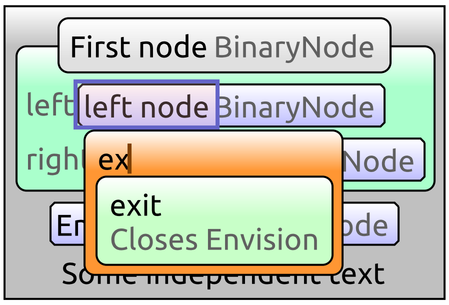
Figure 3.12: A command example. The given command could for example create a new child node as the left branch of the selected item with the name "important".



test

Unknown command 'test'

Figure 3.13: A command could not be understood and a corresponding error message is shown.



ex

exit
Closes Envision

Figure 3.14: The user has started typing 'ex' and the system automatically suggest 'exit'. This command will close *Envision* and is provided by the default scene handler item.

3.11 OOModel

This plug-in extends *ModelBase* by defining many new node classes representing constructs typical in object oriented programming. The current implementation is preliminary and simply defines what properties(sub-nodes) each node contains. Additional node types and functionality are planned for future versions of *Envision*. Further developments are discussed in section 5.1.1.

Figure 3.15 shows the top-level entities that define the structure of an OO application model in *Envision*. Each application is represented by a top-level project. A project possibly includes libraries, other sub-projects, modules and classes. Libraries are projects which provide functionality to the application. Modules are similar to packages in Java and help with better separation of functionality when developing an application. They may consist of other modules or classes.

A class has a visibility specifier and may have base classes. It consists of methods and fields which also have visibility specifiers and can be static or not. A method has formal arguments and results and has a body which is a list of statement items.

A statement item is most commonly an executable statement, but could also be a comment, an image, or any other node which can appear in the body of a method or composite statements. Figure 3.16 shows the different statements that are defined in *OOModel*.

Figure 3.17 shows all expressions currently supported by *OOModel* and finally figure 3.18 shows the different available types and other miscellaneous nodes.

Almost all of the node classes in *OOModel* are based on `ExtendableNode`. This makes it possible for plug-in that use *OOModel* to register extensions for these nodes classes thereby adding additional attributes. This functionality is used for example by *OOVisualization* to add a position to some nodes.

The current set of nodes implemented in the *OOModel* is a good starting point that allows us to test and tweak many of the features of *Envision*. A more feature full implementation that supports, generics, exceptions and others is outlined in section 5.1.1.

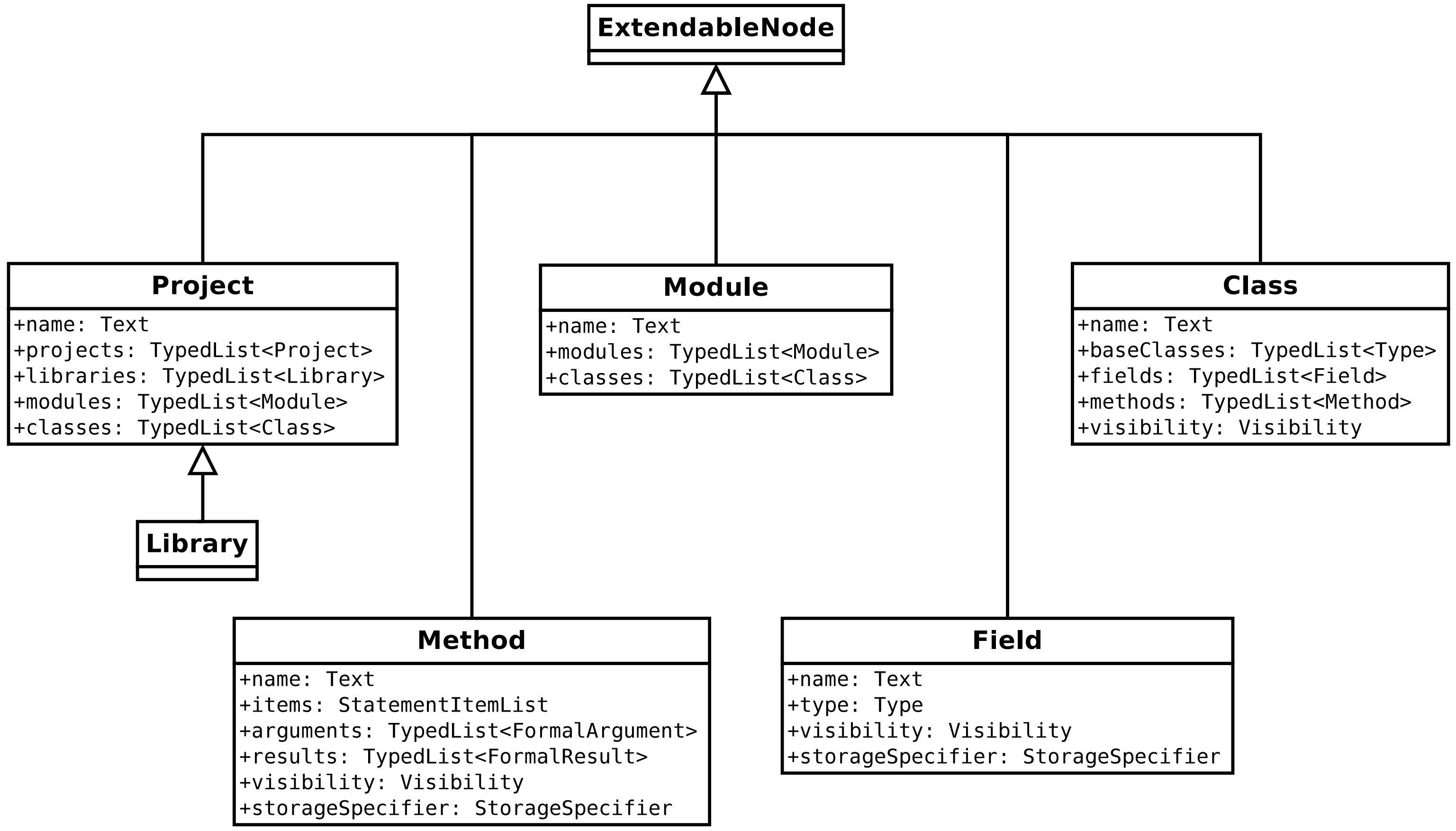

ExtendableNode
Project
+name: Text
+projects: TypedList<Project>
+libraries: TypedList<Library>
+modules: TypedList<Module>
+classes: TypedList<Class>
Module
+name: Text
+modules: TypedList<Module>
+classes: TypedList<Class>
Class
+name: Text
+baseClasses: TypedList<Type>
+fields: TypedList<Field>
+methods: TypedList<Method>
+visibility: Visibility
Library
Method
+name: Text
+items: StatementItemList
+arguments: TypedList<FormalArgument>
+results: TypedList<FormalResult>
+visibility: Visibility
+storageSpecifier: StorageSpecifier
Field
+name: Text
+type: Type
+visibility: Visibility
+storageSpecifier: StorageSpecifier

Figure 3.15: Class diagram of the top-level OO constructs provided by *OOModel*.

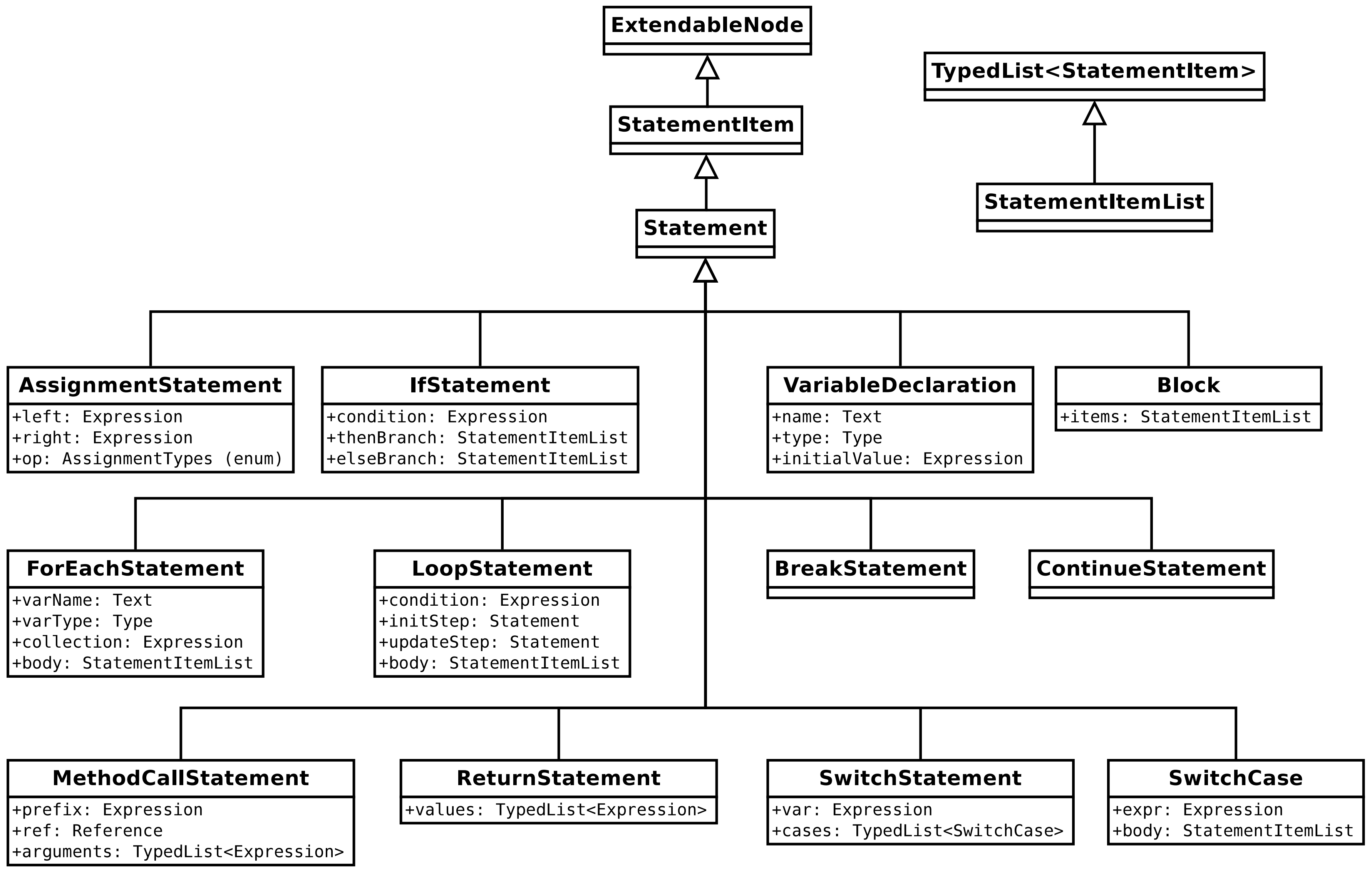


Figure 3.16: Class diagram of the statements provided by *OOModel*.

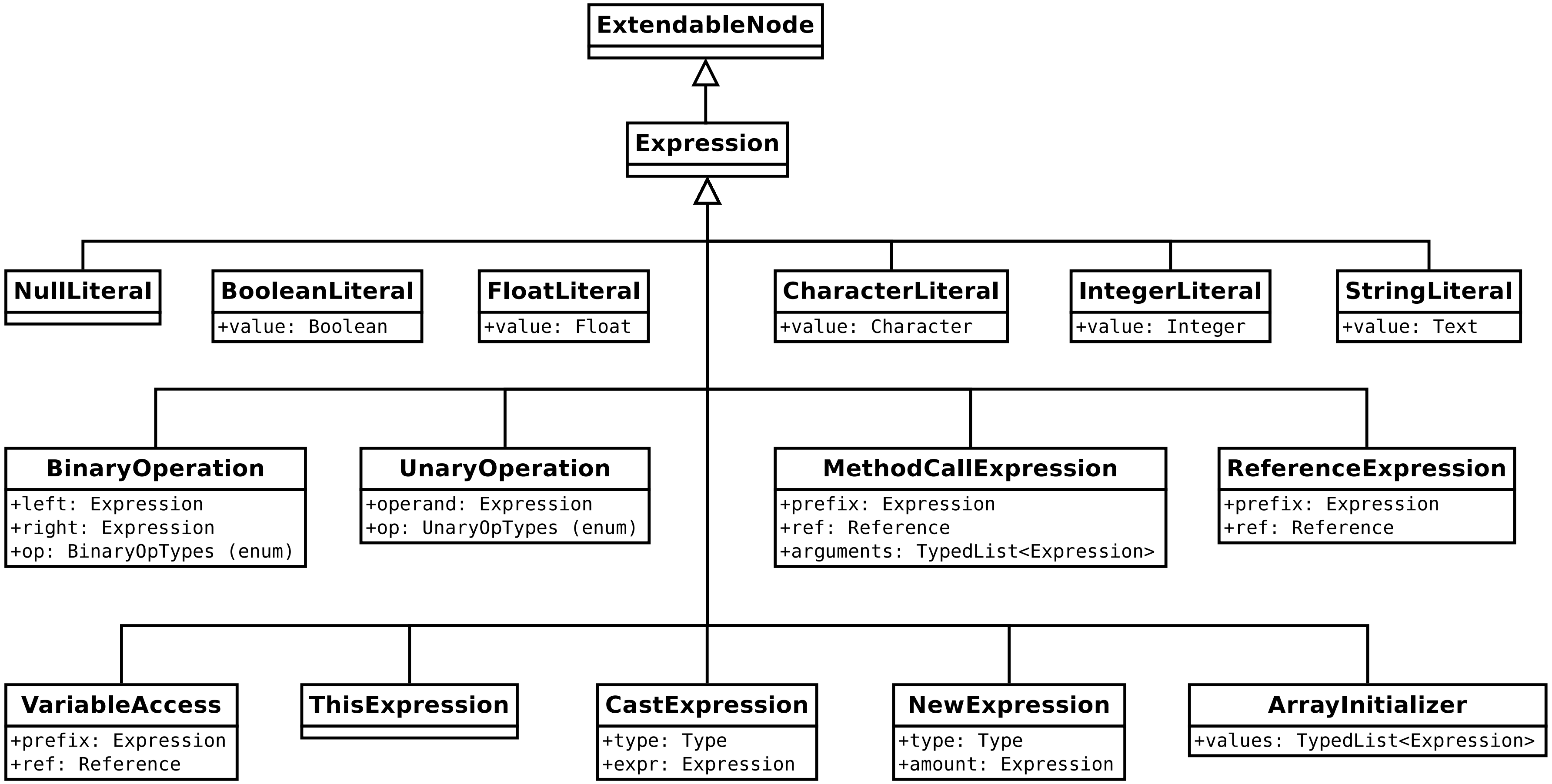


Figure 3.17: Class diagram of the expressions provided by *OOModel*.

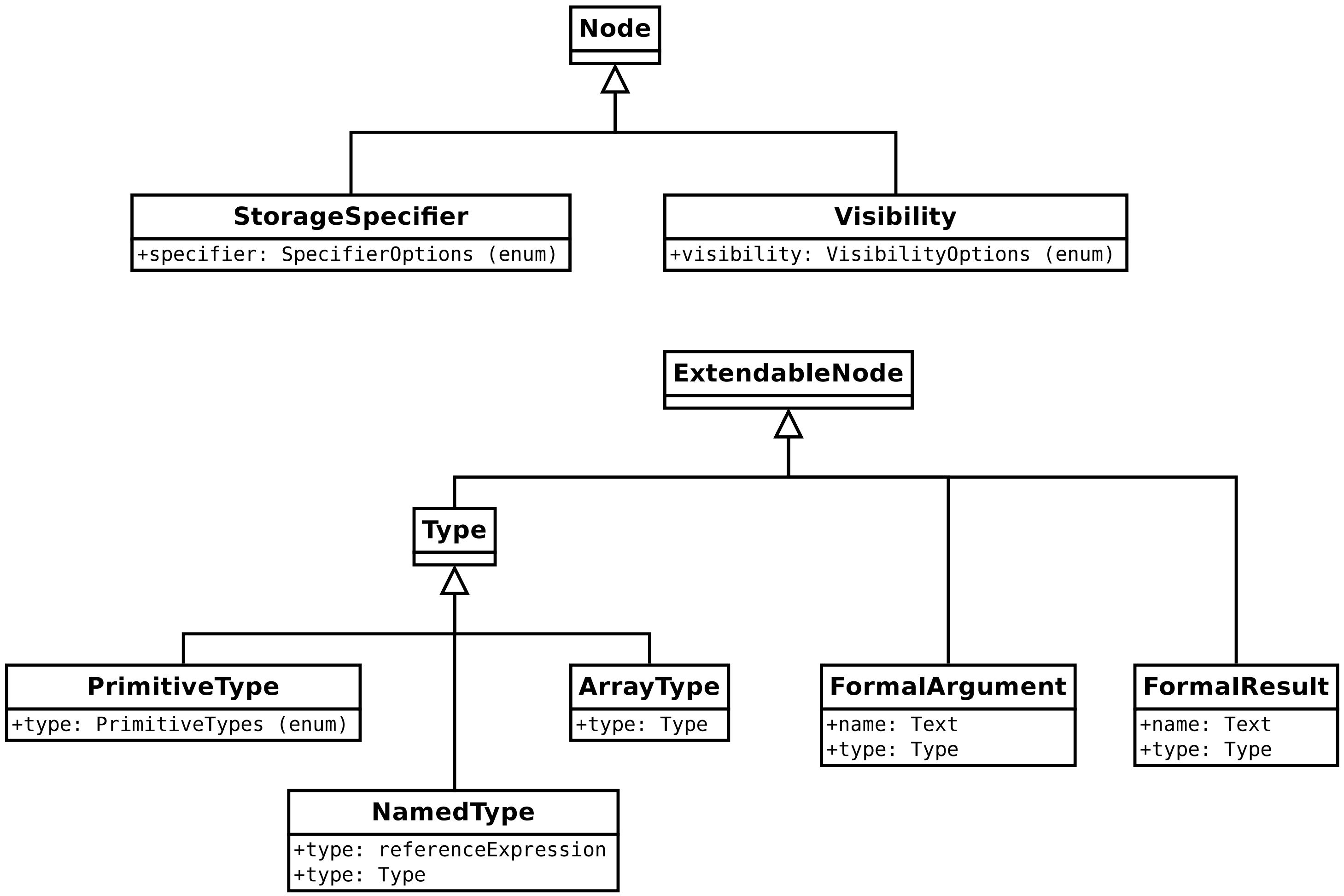


Figure 3.18: Class diagram of miscellaneous constructs provided by *OOModel*.

3.12 OOVisualization

This plug-in provides custom visualizations for most constructs introduced in *OOModel*. These graphical objects use layouts, icons, shapes and styles defined in the `VisualizationBase` plug-in.

In this section we will show some of the more interesting visualization items and will comment on their features and how they make use of the functionality developed in `VisualizatoinBase`. It is important to note that the images shown here serve to illustrate the flexibility of our framework and their ability to increase the performance of programmers, if any, has not been tested yet. We discuss good visualizations in section 3.15.

A ”Hello World” program is shown in figure 3.19. The outermost box is a class as indicated by its icon. The name of the class is `HelloWorld` and is displayed in green font which means that the class is public. The class contains the `main` method - methods have two gears as an icon. This method is public and static, as indicated by the color and underline of the name. The method has a single argument - `args`. The type of the argument - a string array - appears under its name. The method consists of a single statement that calls the `println` method of `Java.System.out` with the string argument ”Hello World”.

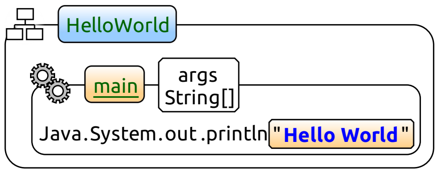


Figure 3.19: A HelloWorld class.

Let us examine how our framework helps us build these visualizations. We will focus on the item representing the `main` method - `VMethod`. It is based on `LayoutProvider` and uses a `PanelBorderLayout`. The top bar of the layout is only used for a header (`SequentialLayout`) on the left side. The header consists of three consecutive elements: an icon (`SVGIcon`) a text field (`VText`) and an arguments list (`VList`). The argument list contains visualizations of type `VFormalArgument`. Such an item simply displays the name of the argument above its type, by using a vertically oriented `SequentialLayout`. The body of the method is visualized as the content in the middle of the layout. This is again a `VList` item, that contains the visualizations of all statement objects. The layout of the method has a shape of type `Box` with white background and rounded corners. Since this is a `PanelBorderLayout` it does not simply draw the shape around it, but rather draws it around the content, thus allowing the borders (in this case only the top bar) to intersect with the shape.

As we can see the method visualization relies heavily on features provided by *VisualizationBase*. This allows us to spend minimal effort designing such a visualization: the source code for `VMethod` contains only declarative statements. It is only needed to specify what items and layouts are part of the method and where they are located. The rest is handled automatically by the framework including updates, rendering, etc.

For additional flexibility all the visualizations are configurable through styles. Changing the style does not require recompilation of the source code and can be used to quickly experiment with different options. For example some of the properties of the `HelloWorld` application which can be configured using styles include:

- **Boxes** -
  - the line color and width of all outlines
  - the background color, or gradient

  - the corner type and radius

- **Text -**
  - font family, style, size and color
  - font family, style, size and color for selected text
  - background shape
- **Sequential layouts**
  - orientation and alignment
  - space between elements
  - background shape
- **Icons**
  - filename that defines the icon image
  - size
  - background shape

While more complicated changes to the look of an item require changes to the source code or running additional plug-ins, the flexibility of the visualization framework makes it is easy to modify the current appearance. For example instead of indicating the scope of an object (public, private, etc.) with the color of its name, it is possible to add a second icon that indicates this. We could also change the item's existing icon based on the scope. These adjustments will require changes in only a few lines of code.

Figure 3.20 shows the `HelloWorld` class in the context of a project[5]. This project uses the `Java` library. The figure shows only the elements from the library which are necessary for the `HelloWorld` class (NOTE: The visualization of the library used here closely resembles the real structure of the standard Java classes, but is not meant to be an exact match). The library contains the `System` and the `String` classes and the `io` module. The system class has one field `- out -` of type `io.PrintStream`. Fields in a class are displayed in the upper left corner, while methods can be placed anywhere inside the class. The gray box next to the name of the `String` class indicates its explicitly specified base classes. Finally we see the `io` module contains one class (`PrintStream`) with a single method (`println`).

Projects, Libraries, Classes and Modules all use the `PanelBorderLayout` layout as basis for their visualization. The content placed in the middle is an instance of `PositionLayout` allowing the user to place items anywhere inside these visualizations. For example on figure 3.20 we see the `Java` library on the right side of the `HelloWorld` class. Both are part of the `HelloWorld` project. The `Java` library itself contains three entities `System`, `String` and `io` positioned such that they fit nicely together. The position of elements is part of the model and will be persisted across sessions. To achieve this *OOVisualization* registers the `Position` extension for nodes of type `Project`, `Module`, `Class` and `Method` which were originally defined by `OOModel`.

Figure 3.21 shows a method that computes the factorial of a number. The method's result type is indicated on the left under its icon. The body consists of three statements: a variable declaration, a conditional and a return statement. The condition of the 'if' statement is displayed in the header next to the icon. The 'then' and 'else' branches are characterized by light green and red backgrounds respectively. If the condition is true the loop on the left is executed otherwise the assignment statement on the right is in effect.

[5]The tree and the book icons in figure 3.20 are taken from the Open Clip Art Library and are licensed under a creative common license. They can be found here `http://www.openclipart.org/people/gurica/gurica_tree.svg`, `http://www.openclipart.org/people/Anonymous/books-aj.svg_aj_ashton_01.svg`

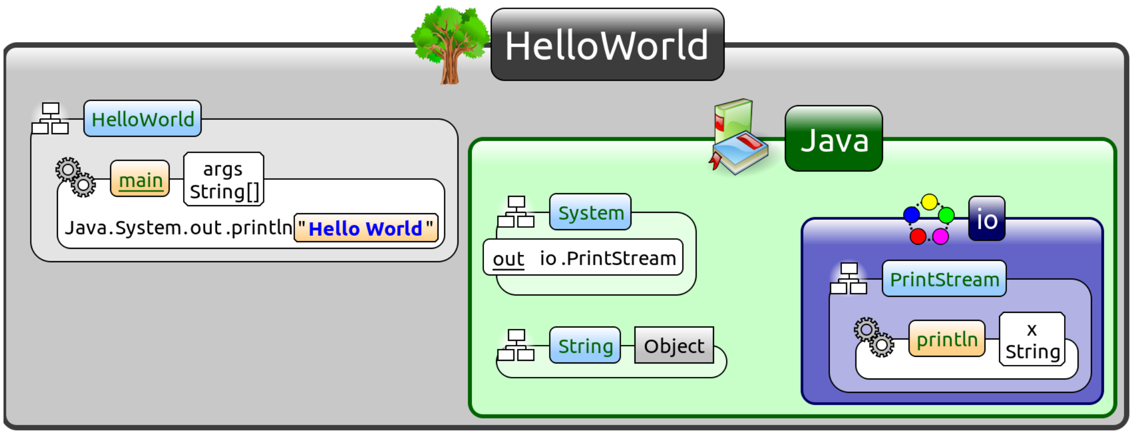


Figure 3.20: A HelloWorld project with the interface of a Java library.

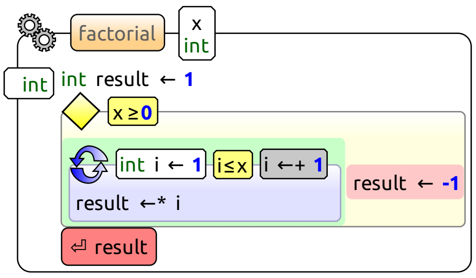


Figure 3.21: A method which returns the factorial of a number.

Since we are no longer constrained by a textual representation we can indicate various operations or concepts in ways which are more intuitive and natural. For example operations such as comparisons or assignments can be indicated by the standard symbols used in mathematics and computer science namely $\leq, \neq, \geq, \leftarrow$. Going beyond symbols, concepts such as conditionals, loops or others can be indicated by a representative icon. This may look like an insignificant change, but such icons are interpreted by visual cognitive processes as opposed to symbolical ones. An advantage of images over text is that they can be also perceived with one's peripheral vision and do not necessarily require the user's focused attention. This makes it easier to stay oriented inside the program model regardless of what the programmer is focusing on right now, since he or she is also aware of the surrounding context.

A further example of how a visual environment can be of benefit is shown in figure 3.22. Two variables of array types are defined and initialized. While a one-dimensional array initializer is simply shown as a list, a two-dimensional initializer is shown as a matrix, which more naturally represents the underlying concept. The first index of `matrix` represents rows and the second - columns.

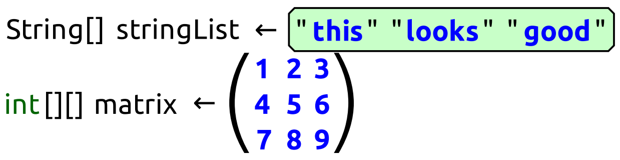


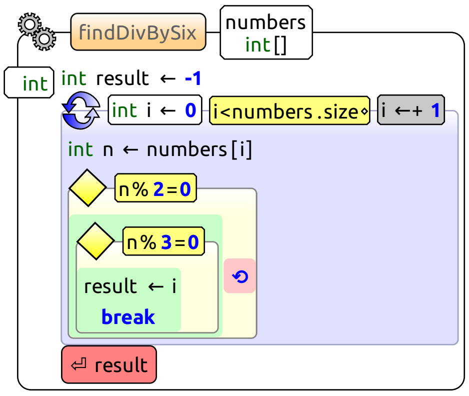
Figure 3.22: An array initializer shown as a matrix.

3.13 ControlFlowVisualization

To demonstrate the flexibility of our visualization framework, we developed a second type of representation for some OO nodes. The *ControlFlowVisualization* plug-in provides an alternative way to display methods, loops, conditionals, statement lists, break, continue and return statements. The idea is that the new rendering will explicitly indicate the control flow in a method by showing its content as a control flow diagram.

Figure 3.23 illustrates this with an example. We see the standard and the control flow representation of a method. This is a fully automatic rendering of the control flow constructed without user involvement. The user can switch between display modes by double clicking on the header of the method. This new visualization is not static and is a fully editable version of the application model: it supports interaction, including command based input just like any other visualization in *Envision*.

The control flow visualization is still preliminary and could be optimized to make it more compact. Even in its current state however it demonstrates the power of *Envision*'s architecture. We are able to extend the standard project visualizations by adding an additional plug-in. This is important since we need to be able to support different model representations in different situations such as editing, debugging, error reporting, refactoring, architecture overview, etc. For larger software projects, it is feasible for a developer to implement a custom plug-in to visualize aspects of the model in a way specific to the project domain.



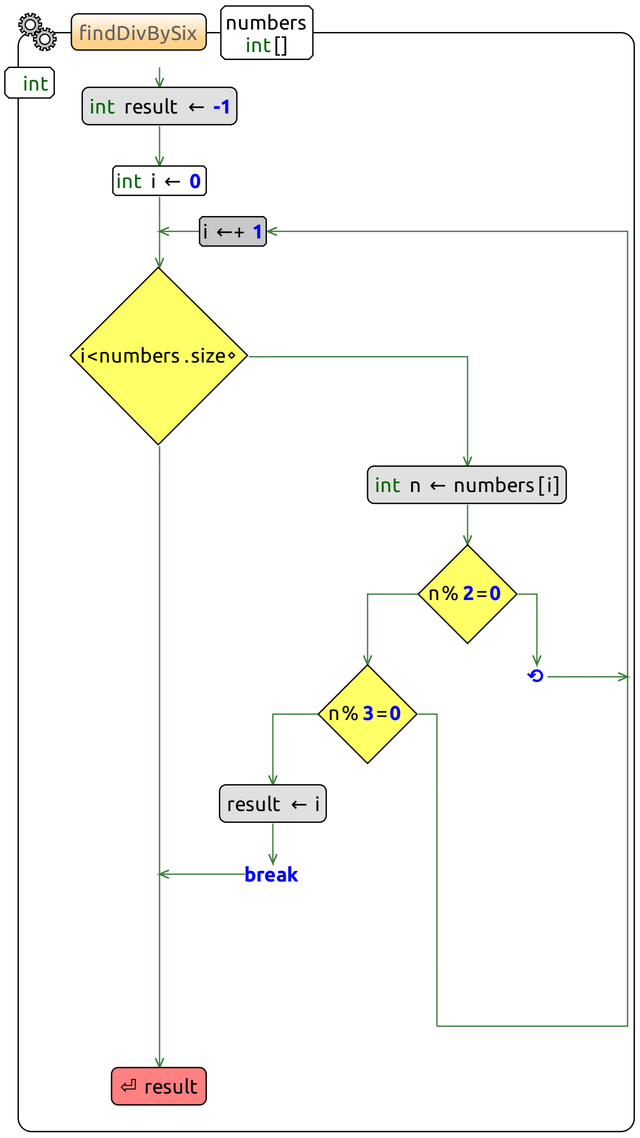


Figure 3.23: A method that returns the index of the first item divisible by 6. The left image is the normal representation as defined by *OOVisualization*. The image on the right is an alternative visualization provided by *ControlFlowVisualization*.

3.14 CustomMethodCall

In order to further demonstrate what can be achieved within our visual programming environment we developed the *CustomMethodCall* plug-in. Once it is deployed it allows the programmer to choose how calls to a method are visualized.

We illustrate this with an example. Figure 3.24 shows an excerpt from the class `Collection`. It defines an ordered list of integer values. A few operations are available to work with the list:

- `find` - returns the index of an element with the value `x`.
- `insert` - inserts the value `x` at the end of the list.
- `empty` - returns `true` if the list is empty and `false` if it contains some elements.
- ∃ - returns `true` if an element with the value `x` exists in the list, otherwise returns false. Note that we use the standard symbol for 'exists' from mathematics as the name of this method. This is possible since all strings in *Envision* use Unicode characters.
- `sum` - returns the sum of all items in the list between the `from` and `to` positions.

The `test` method is an example routine that makes a call to all of these methods. It makes sure that the number 42 is in the list and then simply returns the sum of all items from the beginning of the list until the first instance of 42.

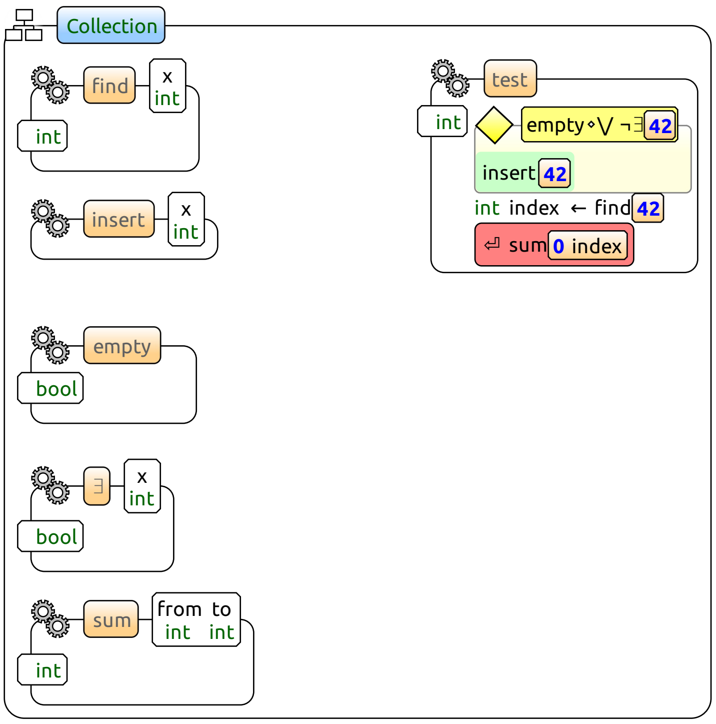


Figure 3.24: A standard visualization for method calls.

Figure 3.25 shows the same class with the addition of altered method call visualizations[6]. The visualization of method calls to `find`, `empty`, `insert` and `sum` have been customized. This is

[6]The box and the plus sign icons in figure 3.25 are taken from the Open Clip Art Library and are licensed under a creative common license. They can be found here http://www.openclipart.org/people/Anonymous/Anonymous_Package.svg, http://www.openclipart.org/people/Anonymous/tasto_2_architetto_franc_01.svg

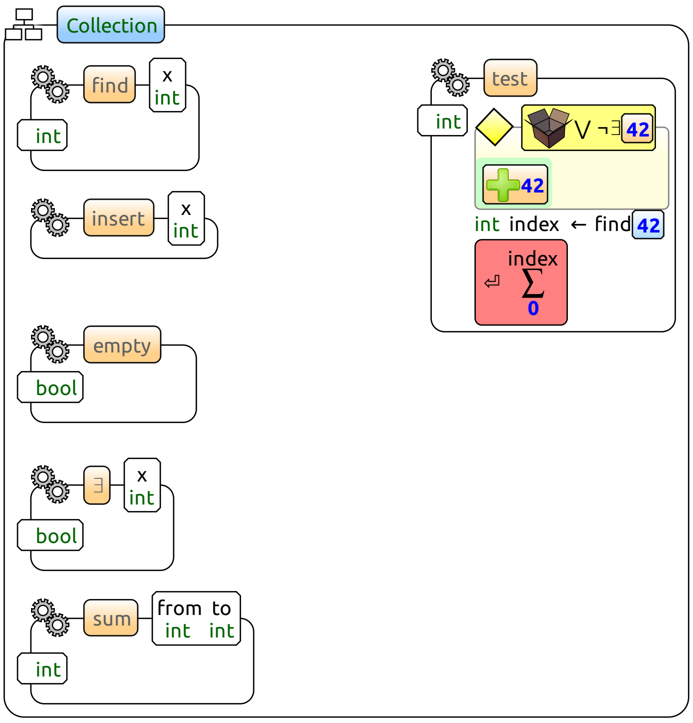


Figure 3.25: A custom visualization for method calls.

achieved by attaching a new attribute to the `Method` node. This attribute is a string name of a visualization that should be used when rendering a call to this method. Additionally the default visualization of method calls has been overriden by the plug-in so that it first tries to find a customized setting.

It is possible to completely alter the way a method call is shown to the user. In case of the `find` method we have just altered the background of the argument list. `empty` and `insert` are now represented by icons of an empty box and a plus sign respectively. The appearance of `sum` has also changed so that it more closely resembles the corresponding mathematical operation.

3.15 A discussion about good visualizations

Throughout this work we show different visual representations of programming constructs. This includes simple textual elements, different colors, icons, nested boxes and different layouts. Are these visualizations good? What is a good visualization?

When judging how good a visualization is there are many criteria to consider. On a more basic level a visualization's size, color scheme and shape should be easy to understand and work with. For example high contrast features are easy to distinguish and therefore desirable, whereas objects of similar colors might blend and be hard to identify. Taking multiple visualizations together it is important that the overall look and feel is consistent. Inconsistent interfaces will confuse the user and will result in less productivity. On a higher level a visualization's suitability is determined by the semantics of the object it represents and even by the context of the task in which this object is currently used. A class diagram visualization is very useful to see the relationships of a class, but is a poor tool when we want to inspect details of its implementation. We can combine all this information and present it at once but then the result might be too cluttered to be useful. Thus a suitable visualization is one that shows to the programmer exactly what is needed, when it is needed and in a form that is easy to understand and manipulate. Ultimately a good visualization is one that makes the programmer more productive.

So how to judge a visual representation by these criteria? Many of the characteristics explained above are subjective and a single visualization will be perceived differently by different developers. Thus there is no formula that can help us design a good visual programming environment. Nevertheless it is possible to empirically determine the usefulness of visualizations. Studies in visual programming often include experiments where programmers have to perform a typical development task such as creating a class, finding and removing a bug, refactoring an existing design, etc. It is possible to measure for example the time it takes to complete such tasks with a visual and a textual environment and compare the two. This gives an assessment of how well a particular visual approach performs against traditional text-based programming. Whitley provides a summary of such experiments concerning visual programming languages in [17]. One of the main messages of the author is that more empirical results are needed in the field.

Therefore, one of the main goals of our work is to create a platform that will support empirical studies with a wide range of visualizations. The visualizations presented so far have not been part of a user study and therefore we have no claims about their suitability. They only serve to demonstrate the customization and extensibility features of *Envision* which will allow us to quickly create different representations for programming models. Once different visualizations are developed we can test and compare them with the help of empirical studies in order to find out which ones can increase productivity.

4

Guided tour

This chapter will guide the user through the features currently implemented in *Envision* and will give suggestions on how to explore the system.

4.1 Accessible features

Envision is currently in an early stage of development. While a lot of features are already implemented, many are still missing. Most notably the interaction mechanisms are not fully developed yet, but the basic framework is in place and we plan to improve the interaction experience soon.

While these features are already implemented, currently there is no way to directly experience model persistence and concurrent node access. Several self-tests however exercise this functionality and the user can run these tests.

Other important functionality which is missing is the ability to create new or delete old programming constructs from an application model. Modifying existing constructs is also limited to basic text modification.

4.2 Starting *Envision*

The current version of *Envision* needs to be started by indicating a plug-in to test at the command line. Otherwise an empty application screen will be displayed as until now there is no plug-in which works with the main window outside of specific test cases.

The user has a choice of running tests for the non-graphical components such as *Logger*, *ModelBase* or *FilePersistence* and checking the test statistics which are printed to the standard output or running a test for a plug-in that involves visualization. Plug-ins which display content on screen include *VisualizationBase*, *InteractionBase*, *OOVisualization* and *ControlFlowVisualization*.

NOTE: Any tests involving visualizations are designed to run on their own. Specifying more than one such test on the command line will result in a damaged image being displayed.

Appendix B.1 explains the general syntax for running plug-in tests on the command line. Here we will only give interesting cases to try out:

command	**effect**
`envision --test logger`	Runs self-tests for the logger. A few log messages will be displayed on the standard output and the test statistics will be shown.
`envision --test modelbase`	Runs all self-tests for the *ModelBase* plug-in. This is a comprehensive test suite that covers most of the functionality of the plug-in. Test statistics are shown at the end.
`envision --test filepersistence`	Runs all tests for saving and loading models to and from disk. Displays statistics at the end.
`envision --test visualizationbase`	Starts *Envision* and displays the visualization of a simple generic model tree.
`envision --test interactionbase`	Behaves identically to the previous command.
`envision --test oomodel`	Runs all tests for *OOModel* and displays statistics at the end.
`envision --test oovisualization`	Starts *Envision* and displays a sample OO project that contains libraries, classes, methods, statements and others.
`envision --test controlflowvisualization`	Starts *Envision* and displays a class with two methods. They include non linear control flows which are interesting to examine in the control flow view. See section 3.13.
`envision --text custommethodcall`	Starts *Envision* and displays a class with a few methods that illustrate custom method call visualizations. See section 3.14.

The next section will show the various types of interaction currently available to the user when a program model has been visualized.

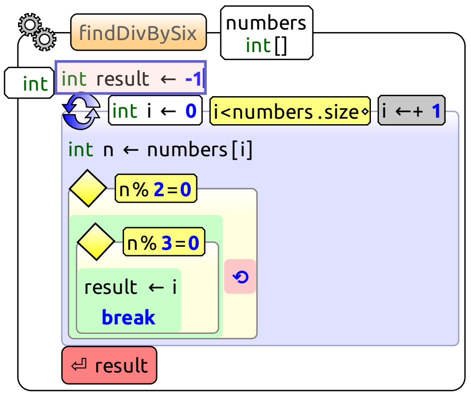
4.3 Interaction

4.3.1 Selection

To test this run: `envision --test controlflowvisualization`.

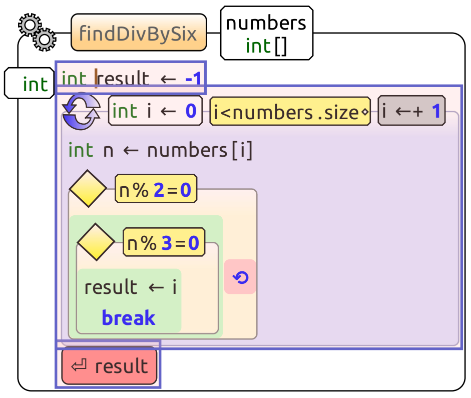
Using the mouse it is possible to select a single item by clicking on it. Selected items are highlighted with a blue outline as shown in figure 4.1 where the first statement of the method is highlighted. Once an item is selected press the different arrow keys to navigate between items without the mouse. The order in which items are selected depends on their layout. Notice that it is possible to select any item using the keyboard. This easy keyboard interaction is essential for a programming environment striving to improve developer productivity.

To select multiple items, click on an item and drag the mouse while holding down the left button. This actions filters out items so that the resulting selection is logically consistent. For example, clicking and dragging inside a single statement will select different parts of the statement. If the



```
findDivBySix  numbers int[]
int  int result ← -1
     int i ← 0  i<numbers.size  i ←+ 1
       int n ← numbers[i]
       n%2=0
         n%3=0
           result ← i
           break
     ⏎ result
```

Figure 4.1: A single selected item: an assignment statement.



```
findDivBySix  numbers int[]
int  int result ← -1
     int i ← 0  i<numbers.size  i ←+ 1
       int n ← numbers[i]
       n%2=0
         n%3=0
           result ← i
           break
     ⏎ result
```

Figure 4.2: Multiple selected items: all method statements.

mouse cursor leaves the statement borders, the entire statement will be selected instead of its parts. Furthermore if the cursor now moves over a different statement, that second statement will also be selected as a whole. Finally if the cursor leaves the current statement list, the entire list will be selected. Multiple selection is shown on figure 4.2.

4.3.2 Modification

To test this run: `envision --test controlflowvisualization`.

A few modification mechanisms are currently implemented in *Envision*. It is possible to click on a text item and directly edit it. Note however that many text items are marked as not-editable. Items which can be modified directly are names of classes and methods, names of variables in variable declarations and literals. Some textual items, such as integer literals, impose restrictions on the possible modifications. Integers for example can only be modified in such a way that the resulting string can be converted to an integer.

It is also possible to copy and paste items. Try selecting the outer 'if' construct by clicking just above its condition. Press CTRL+C to copy the item to clipboard. If you wish you can open a text editor and paste the clipboard contents to examine the structure of the nodes. It is even possible to modify this text and copy the new version to clipboard. After one or more items have been copied it is possible to insert them into any list visualization. As the 'if' construct is still selected, simply press CTRL+SHIFT+V. This pastes the contents of the clipboard after the currently selected item.

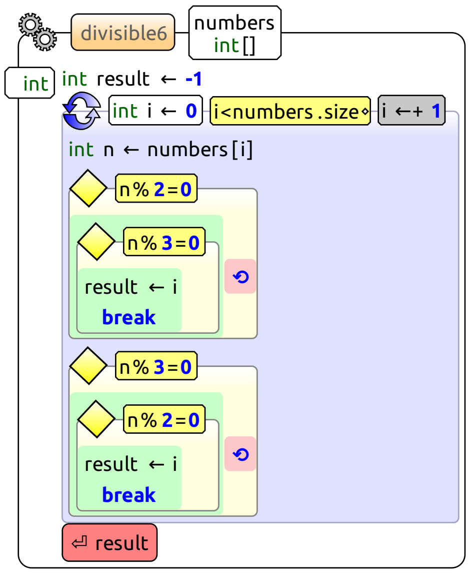

```
divisible6  numbers int[]
int
int result ← -1
int i ← 0   i<numbers.size   i ←+ 1
int n ← numbers[i]
n%2=0
  n%3=0
    result ← i
    break
n%3=0
  n%2=0
    result ← i
    break
⏎ result
```

Figure 4.3: The modified `findDivBySix` method.

Figure 4.3 shows the result of renaming the `findDivBySix` method, copying the outer ’if’ statement and changing the order of the conditions.

To undo and redo changes made to the model press CTRL+Z and CTRL+Y respectively. Note that there is an issue currently and undo and redo only work if there is an item selected.

4.3.3 Command prompt

To test this run: `envision --test controlflowvisualization`.

When an item is selected the user can show a command prompt and give this item commands as discussed in section 3.10.2. Select any visual item and press ESCAPE. An orange command prompt will appear. The only command currently implemented is ’exit’. Type ’exit’ and press Enter to close *Envision*. As soon as you start typing the system will suggest commands that match the current input string. Type ’e’ in an empty command prompt and the suggestion for ’exit’ will appear underneath. Press TAB to auto-complete the current command.

If you type a command that is not understood by the system an error message will be shown under the command prompt. This message will remain visible until you press Enter again. Auto-complete suggestions will appear under the error message.

While typing a command, the prompt appears on top of the application model visualization. If you need to see something under the prompt, simply press ESCAPE again. The prompt will disappear. Pressing escape a third time will show it again with all the content it had when it was hidden. This is very convenient to quickly look something up that was obscured by the prompt. Note that if you change the selected object when the prompt is hidden, the currently entered command string will be cleared.

4.3.4 Taking screen shots

In order to allow visualizations to be used in documents or other media, it is possible to take screen shots. There are two different ways to capture the image of the currently displayed model: the entire scene or only the part visualized by the current view. To take a screen shot of the entire scene press CTRL+SHIFT+PRINTSCREEN. To take a screen shot of the current view only press CTRL+PRINTSCREEN.

In both cases, three files will be created that contain the screen shot in different formats: PNG, SVG and PDF. The PNG file contains a raster image and is guaranteed to look identically to the screen. It is rendered at twice the resolution of the screen to improve the quality of the graphic. The SVG and the PDF store the image in a vector format but may contain artifacts as not all screen content is perfectly representable in this way.

Qt libraries are used to create all these images. The files are placed in the same folder as *Envision*'s executable.

4.4 Control flow visualization

To test this run: `envision --test controlflowvisualization`.

Envision can visualize a model in an arbitrary way. Different visualizations are used to facilitate different tasks. We developed a control flow visualization for method bodies to illustrate the flexibility of our framework. This was further discussed in section 3.13.

To switch between the different representations double click the left mouse button just above the method body outline.

4.5 Custom method call visualization

To test this run: `envision --test custommethodcall`.

To further demonstrate the capabilities of our framework we developed an extension which allows the programmer to specify a custom visualization for all calls to a particular method. Simply run the command above to see the `Collection` class that we discussed in section 3.14.

5

Conclusion

We have presented the concept and initial implementation of *Envision* - a visual programming system.

We discussed our motivation for building a new general purpose IDE independent of a specific programming language and defined requirements for *Envision*'s design that will allow it to achieve new levels of integration between all artifacts involved in software engineering by combining a visual approach for application development with novel interaction mechanisms. The resulting system's goal is to improve the productivity of professional programmers.

In our description of *Envision*'s functionality we outlined the components that form the model-view-controller pattern at its core. The current implementation was shown to feature a modular design where functionality is entirely provided by plug-ins. The resulting system could be easily extended to support new visualization features. In addition we demonstrated that many aspects of the current framework are customizable.

The layered architecture of *Envision* highlights the responsibilities of each module and provides a clear path for further extensions. With the current work we have established the fundamental of the system and we plan to expand the capabilities of the IDE as discussed next.

5.1 Future Work

Envision is a work in progress in the early stage of its development. There are many aspects of the system that need to be further defined and implemented. Here we will briefly outline some of the features and experiments we are planning to do.

5.1.1 Additional functionality

Before *Envision* can be used for developing software a number of crucial modules need to be improved or developed.

Interaction

The current interaction mechanism needs to be extended to provide specific behavior for all available constructs of the object oriented model. Of particular importance is to design an effective way to create and modify method statements. For these items a mostly textual representation is desirable as it is makes it easy to convey details in a compact manner. Suitable ways need to be found to manipulate these representations and the underlying model. The challenge is to do this in such a way, that programmer productivity is similar or even better than what can be achieved with standard text-based IDEs.

Object oriented model

Additional constructs need to be added to the object oriented model of *Envision*. Most prominently this includes generics and exceptions, but could also extend to templates and others. Visualizations appropriate for these constructs should also be designed.

Compilation

As part of *Envision*'s design the programmer creates a new application in a language independent way. At some point however the application will have to be compiled for a specific platform. Functionality should therefore be implemented that allows the user to compile an application model to a series of source files in a target programming language such as C++ or Java. The challenge here is to implement this functionality in a flexible way so that the target programming language and platform for compilation should be independent of the model and can be chosen by the user.

Some auxiliary functionality that will help with the development process includes:

Alternative visualizations

As demonstrated in this work *Envision* is capable of representing the same application model using different visualizations. So far we have just developed a default visualization for object oriented programs and two extensions for it: control flow and custom method call. However to speed up the development process it is desirable to create a wider variety of visualizations optimized for specific tasks such as:

- debugging
- error reporting
- project overview
- refactoring
- documenting

Consistency and Verification

It would also be interesting to integrate software verification methods inside *Envision*. We want to explore how such formal techniques can benefit from a visual programming environment. The rich application model of *Envision* also needs a consistency framework to be designed, that can keep different parts of larger applications in a consistent state (for example synchronizing comments and source code).

Statistics

Because of the higher level of integration in *Envision* it is possible to build various statistics directly into the system and use this as another tool for project management and overview. Possible developments with regards to statistics include:

- keeping track of which users created/modified part of the program tree and when
- automatically summarizing and visualizing test results
- visualizing code metrics
- monitoring bug reports

Implementing a suitably designed database storage for *Envision* might greatly improve the collection and analysis of data for statistical purposes.

Semantic version control

Version control is essential for any professional project. Our plan is to introduce a semantic version control module in *Envision* that will operate at the level of a model node. This means that version control will not be performed with files as it is today, but rather at a logical level that follows the structure of a program. Different versions of classes, methods and even statements can be preserved for maximum flexibility. This is a challenging task that involves defining semantic version control in general and integrating it with *Envision*'s existing application model.

5.1.2 Experimental evaluation

Envision makes a unique mix of features available to developers. In order to test the performance of our approach and improve it, it is essential to conduct regular experiments with the system. Many different empirical results are of interest when it comes to visual programming and interaction, such as how much time does a developer need for a given task, how many navigation steps are needed to find a specific part of the application, etc. Running such experiments in parallel with *Envision* and other modern IDEs such as Eclipse can provide valuable feedback about the effectiveness of our approach.

Appendices

Appendix A

File formats

This section describes the format of all files currently used or created by *Envision*. Every file described here is in an XML format.

A.1 Plug-in meta data (.plugin)

The file with the meta information about a plug-in is primarily used to define and resolve dependencies. Here is an example file:

```
<!DOCTYPE EnvisionPlugin>
<plugin id="modelbase" name="Model base" version="0.1">
  <dependencies>
    <dependency pluginid="logger" version="0" />
    <dependency pluginid="selftest" version="0" />
  </dependencies>
</plugin>
```

Listing A.1: The plug-in meta data file of *ModelBase*.

The top-level element is always called `plugin`. Its attributes define the plug-in id, short name(description) and full version string.

The `dependencies` element specifies all other plug-ins that are needed in order to run the current one. A dependency is comprised of a plug-in name and a specific major version of that plug-in. The major version is everything before the first '.' in the version string of a plug-in. A dependency will be loaded before the current plug-in.

A.2 Persisted model

This file type is used by *FilePersistence* to store application models on disk. To describe it we will use the simple model shown in figure A.1. It shows a model called 'units' that contains 8 nodes. The node ids are indicated in the corresponding circle. Nodes 0,2,4 and 6 are of type `BinaryNode`. A binary node always has a name an optionally has a left and right nodes. The name is just a string. Node 2 in the figure is a persistence unit. Persistence units are saved in separate files by the `FileStore` class implemented in *FilePersistence*.

There is a master file which belongs to the main persistence unit (the model) and a file for each other persistence unit if any. Here is an example of a master file:

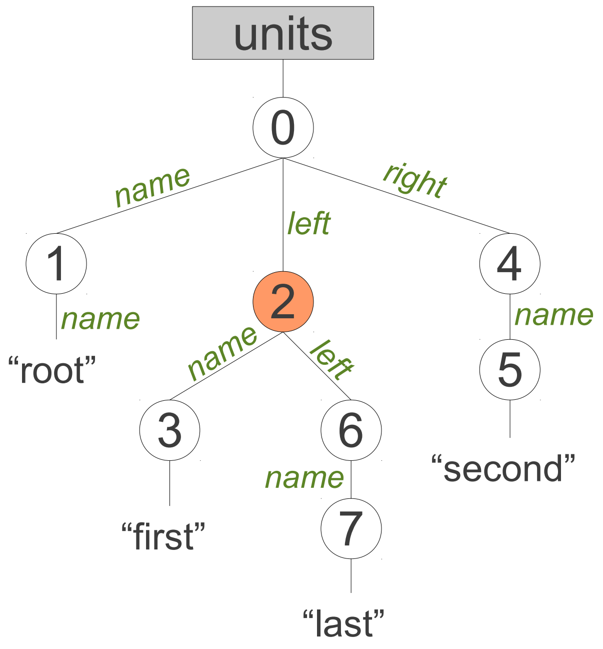


Figure A.1: A binary tree model with a persistent unit.

```
<!DOCTYPE EnvisionFilePersistence>
<model nextid="8">
  <BinaryNode id="0" name="units" partial="0">
    <Text id="1" name="name" partial="0">S_root</Text>
    <persistencenewunit name="left">S_2</persistencenewunit>
    <BinaryNode id="4" name="right" partial="0">
      <Text id="5" name="name" partial="0">S_second</Text>
    </BinaryNode>
  </BinaryNode>
</model>
```

Listing A.2: The master file of a simple application model consisting of binary nodes.

The root element is `model` and its `nextid` attribute indicates the id that should be given to the next node that is created. This is necessary in order to give each node a unique id.

The tag name of all elements other than the model defines the type of the node they represent. The model always contains only one node - the root node. In this case a `BinaryNode`. Each node has the following attributes:

1. **id** - the integer id of the node.
2. **name** - the name of the node. This is always set by the parent of the node. The name of the root node is the model (application) name. This is the name by which the parent identifies the node. If the node has a name of its own (such as the name of a method node), this will appear as a sub-element of the node, rather than an attribute.
3. **partial** - whether the node should be partially loaded if possible. If this is 1, nodes which support partial loading should only load their content partially to avoid too much memory overhead. This property is controlled by the parent.

The sub-elements of a node are defined by the `store` method of the node. They represent the content of the node. There are only three exceptions: String, Integer and Double value nodes. The value of such nodes is saved directly as text into the XML file. Depending on the node type the value is prefixed with 'S_', 'I_' or 'D_' for String, Integer and Double respectively. The root node above has sub-nodes 'name', 'left' and 'right'.

`persistencenewunit` is a special node which indicates that its content is defined in a different file. The type of this node is also defined in that file. The file name is represented as a string and is therefore preceded by 'S_'. In the example above this filename is simply '2'. Here is the content of that file:

```
<!DOCTYPE EnvisionFilePersistence>
<BinaryNodePersistenceUnit id="2" name="left" partial="0">
  <Text id="3" name="name" partial="0">S_first</Text>
  <BinaryNode id="6" name="left" partial="0">
    <Text id="7" name="name" partial="0">S_last</Text>
  </BinaryNode>
</BinaryNodePersistenceUnit>
```

Listing A.3: A file representing a persistent unit.

Its structure is identical to the one of the master file, with the exception of the `model` top-level element which is absent. It is possible to have nested persistence unit nodes, in which case each new unit will be saved in its own file.

A.3 Clipboard copy of nodes

The format of nodes in clipboard is similar to the ones persisted on disk. Here is an example of two nodes being copied to clipboard:

```
<!DOCTYPE EnvisionFilePersistence>
<clipboard>
 <BinaryNode name="0" partial="0">
  <Text name="name" partial="0">S_first</Text>
 </BinaryNode>
 <BinaryNode name="1" partial="0">
  <Text name="name" partial="0">S_second</Text>
 </BinaryNode>
</clipboard>
```

Listing A.4: Representation of nodes copied to clipboard.

The main element is `clipboard` and it contains one sub-element for each node that was copied. The structure of these sub-node elements is identical to the one discussed in appendix A.2, with the following differences:

- There is no node id attribute. This attribute is meaningless across projects as it is specific for a model.
- The partial hint is always set to 0. Nodes loaded from the clipboard cannot be partially loaded.
- For direct children of `clipboard` the 'name' attribute is simply an index that ranges from 0 to one less than the number of nodes copied to the clipboard.

A.4 Visualization style

Each visual item in *Envision* has a style, which is a set of properties that define how the item should be drawn. The values of these properties are defined by a style file. A sample file is shown below:

```
<!DOCTYPE EnvisionVisualizationStyle>
<style prototypes="item/Item/default">
  <shape prototypes="shape/Box/default">
    Box
    <backgroundBrush>
      <gradient>
        <stopPoints>
          <e1>
            <first>1.0</first>
            <second>
              <alpha>255</alpha>
              <red>192</red>
              <green>192</green>
              <blue>192</blue>
            </second>
          </e1>
        </stopPoints>
      </gradient>
    </backgroundBrush>
    <cornerType>0</cornerType>
    <cornerRadius>5</cornerRadius>
  </shape>
  <itemsStyle prototypes="layout/SequentialLayout/default">
    <direction>2</direction>
    <spaceBetweenElements>5</spaceBetweenElements>
  </itemsStyle>
</style>
```

Listing A.5: The 'default' style for `VList`.

The root element is always called `style`. The tags of its child elements are the names of the properties whose value needs to be defined. A value can be a simple text, boolean, number or a composite style that itself consists of other values. The meaning of these properties is defined by the style class.

Each property may have one or more prototypes defined using the `prototypes` attribute. A prototype is a style file. This enables a specialization scheme for composite properties. When such a property has at least one prototype, it is not necessary to explicitly specify all of its sub-elements. The value of any elements that are not specified will be taken from the first prototype in the prototype list that has a definition for this value.

In the example above the entire style uses a prototype - the default Item style. This is a common practice as the default item style specifies properties which each item must define such as a shape. Afterwards two properties are defined - `shape` and `itemsStyle`. The `shape` property overrides the one from the default item style and specifies that a box shape will be used. This shape has many properties which are not explicitly specified here, therefore the shape's default style is used as a prototype. Only the sub-properties of the shape that differ from the prototype are defined in the current style file. This is essentially the second color of the background gradient, the corner type and the corner radius. The other property - `itemsStyle` - defines the direction and the space between elements of the underlying sequential layout that is used within `VList` to display its content. Any other properties that the sequential layout style requires are taken from the default prototype.

Multiple prototype entries in the `prototypes` attribute can be specified by using a comma. The paths specified there can be either relative to the current file, or can begin in the 'styles' folder of a deployed *Envision* installation.

Appendix B

Plug-in interface

Each plug-in in *Envision* is implemented as a platform specific shared library. The library uses the standard Qt plug-in framework that facilitates cross-platform development. In this framework each plug-in must expose a specific interface that can be used by the host application to communicate with it. In *Envision* this interface is defined in the *Executable* module and is called `EnvisionPlugin`. It consists of three methods:

`bool initialize(EnvisionManager& manager)`
This method is called immediately after the plug-in shared library is loaded. Plug-ins can execute custom initialization code here and must report if their initialization was successful or not. The `manager` parameter is a reference to an object which can be used to get information about the system. It can be used to obtain the main window for drawing, or to ask about other plug-ins that are loaded. Plug-ins do not need to explicitly check for their dependencies, this is performed by the system prior to loading the plug-in.

`void selfTest(QString testid)`
After all plug-ins are loaded, this method will be called for any plug-in for which the user specified that a test case should be run. The *testid* parameter indicates a particular test that should be executed. If this argument is an empty or a null string, all tests for the plug-in should be executed.

`~EnvisionPlugin()`
The destructor is called when the application is closing. Plug-ins can use this to perform clean up operations.

B.1 Running tests

Indicating that a specific plug-in should be tested after *Envision* is initialized can be done using the `--test` switch on the command line. Here is the syntax for this command:

```
envision --test pluginid[:testid] [pluginid:[testid]] ...
```

`pluginid` is a mandatory argument that specifies which plug-in needs to be tested. If `testid` is specified only the corresponding test will be run, otherwise all tests for the indicated plug-in will be executed. To test multiple plug-ins or to run multiple tests for a plug-in, specify additional `pluginid:testid` pairs. Here are some examples:

```
envision --test modelbase
```
Runs all tests for the *modelbase* plug-in.

```
envision --test modelbase:RemoveOptional
```

Runs only the `RemoveOptional` test for the *modelbase* plug-in.

```
envision --test modelbase:RemoveOptional modelbase:ListCreation oomodel
```

Runs the `RemoveOptional` and `ListCreation` tests for the *modelbase* plug-in and all tests for the *oomodel* plug-in. The tests will be run in the order indicated on the command line.

Appendix C

Exporting classes

Envision is built by plug-ins which rely on other plug-ins and export functionality to clients. Since plug-ins are compiled to platform native shared libraries it is important to properly export requested functionality in the source code. The way symbols (classes, global variables and functions, name spaces, etc.) are exported differs from operating system to operating system.

Fortunately Qt provides facilities to make this as easy as possible. Combined with the *Plugin-Generator* developed for *Envision*, exporting functionality is even easier. In the following we will demonstrate the exporting of classes so that they may be used by client plug-ins. Exporting of other symbols is identical.

There are three ingredients needed for exporting a class, which are discussed next.

C.1 API definition header file

Each project must have a file that defines an export and import macros. This file is typically named '*pluginname*_api.h', where *pluginname* is the lowercase plug-in id. Here is the content of this file for the *InteractionBase* plug-in:

```
#ifndef INTERACTIONBASE_API_H_
#define INTERACTIONBASE_API_H_

#include <QtCore/QtGlobal>

#if defined(INTERACTIONBASE_LIBRARY)
#  define INTERACTIONBASE_API Q_DECL_EXPORT
#else
#  define INTERACTIONBASE_API Q_DECL_IMPORT
#endif

#endif /* INTERACTIONBASE_API_H_ */
```

Listing C.1: The contents of `interactionbase_api.h`.

This file defines the `INTERACTIONBASE_API` macro. This macro is equivalent to `Q_DECL_EXPORT` when compiling the *InteractionBase* plug-in, and is therefore used for exporting classes. It is equivalent to `Q_DECL_IMPORT` in any other plug-in and is thus used for importing when compiling other plug-ins that include this header file.

The `Q_DECL_EXPORT` and `Q_DECL_IMPORT` macros are defined by Qt and expand to a platform specific attribute modifier for symbols.

If a plug-in is initially generated with *Envision*'s plug-in generator this file will be created automatically.

C.2 The library compilation flag

In order for the import/export macro above to be defined with the correct value it is important that its control symbol is defined only when compiling the target library. In the example above this means that only when compiling the *InteractionBase* plug-in should `INTERACTIONBASE_LIBRARY` be defined. The easiest way to achieve this is to include the
`-DINTERACTIONBASE_LIBRARY`
flag on the command line of each compilation operation. When using Qt's QMake project file, this can done by including the following line anywhere in that file:

```
DEFINES += INTERACTIONBASE_LIBRARY
```

Once again, if a plug-in was created by the plug-in generator, its project file will already contain this setting.

C.3 Macro modifier for class definitions

After the infrastructure for exporting is in place it is necessary to specify what will be exported on a per symbol basis. This is done simply by including the export macro in front of any newly defined symbol.

Here is the relevant part of the `GenericHandler.h` file which is part of *InteractionBase*:

```
#ifndef GENERICHANDLER_H_
#define GENERICHANDLER_H_

#include "../interactionbase_api.h"

...

namespace Interaction {
  class INTERACTIONBASE_API GenericHandler ...
  {
    ...
  };
}

#endif /* GENERICHANDLER_H_ */
```

Listing C.2: A segment of GenericHandler.h. Missing parts are indicated with '...'.

When a class needs to be exported, its header file must first include the plug-in's API header. Then just after the `class` keyword and before the name of the class insert the export macro as in the example above.

This has to be done separately for each class that is exported. Moreover, all symbols which the class uses in its public interface should also be exported. Most notably this includes all other classes in its inheritance hierarchy.

NOTE: Template classes should not be exported. Since a template is instantiated at the time of use and each client will create its own copy of it. The only exception is explicitly instantiated templates. In that case clients should see an import definition for each explicit instantiation and the source library should explicitly export the class definition.

C.4 Importing classes

One last consideration is important for clients that wish to use services provided by other plug-ins. In this case it it important that the compilation command includes a reference to the shared libraries that include the imported functionality.

This is most easily achieved by including the following line in the QMake's project file:

```
LIBS += -lPLUGINNAME
```

where `PLUGINNAME` is the name of the plug-in in lowercase.

Appendix D

Project Organization

Here we will briefly present the organization of the *Envision* project within the Eclipse development environment and on disk.

D.1 Eclipse projects

The development process of *Envision* is optimized for the Eclipse IDE. Each plug-in in *Envision* is a separate Eclipse C++ project in the main workspace. There is also a C++ project for the main executable called *Core*. Finally there are three more generic Eclipse projects:

- *DebugBuild* - this is a container for all compiled shared libraries produced by a debug build. This is used to run the application during the development phase.
- *ReleaseBuild* - this is a container for all compiled shared libraries produced by a release build. These binaries can be used for distributing end-user versions of *Envision*.
- *PluginGenerator* - this project contains a Bash script and a plug-in template. It is used to quickly create the skeleton of a new plug-in for *Envision*.

The C++ Eclipse projects are partially managed by the Qt integration plug-in. Each project is configured to depend on all projects from which it imports classes. A project also includes a launcher that automatically starts *Envision* and runs all tests for the project.

The most important part of a project is Qt's QMake project file (with a .pro extension). These files define shared library dependencies, needed Qt modules and various other build parameters.

D.2 Folder structure

D.2.1 Deployed installation

A deployed version of *Envision* has the following structure:

path	**description**
`/`	The root folder of the application. It contains the envision executable. Any screen shots produced by the application will also be placed here.
`/plugins`	This folder contains all shared libraries and all plug-in meta information files. New plug-ins can be added by simply copying them here.
`/styles`	Contains the styles for all classes used in *Envision*.
`/styles/item`	Contains the styles for visual items. Each item is associated with a folder that matches the name of the item's class. Inside this folder there can be an arbitrary number of files that contain different styles for this item.
`/styles/shape`	Contains the styles for background shapes of items. Similarly to above, each shape has a folder that matches its class name, and may contain multiple style files.
`/styles/layout`	Contains the styles for layout items. The structure of this directory is identical to that of the `item` folder.

D.2.2 Plug-in project

Individual plug-in projects have the following general structure:

path	**description**
`/`	The root folder of the plug-in contains the Eclipse project files, the plug-in meta information file and the QMake project file.
`/debug`	This is used internally by Qt to create temporary files during a debug build compilation.
`/headers`	This folder contains the headers of all entities that contain exported symbols. The internal structure of this folder is defined by the programmer and should mimic the one of `src`.
`/release`	This is used internally by Qt to create temporary files during a release build compilation.
`/src`	This folder contains all the source code files for the project except for headers placed in `/headers`.
`/styles`	If this plug-in uses styles, the corresponding style files should be placed in this folder.
`/test`	Any code that relates to unit or integration testing of the plug-in should be placed in this folder.

Appendix E

PluginGenerator

The plug-in generator is one of the projects in the *Envision* workspace. It consists of a Bash shell script and a plug-in template. The script can be invoked to create the skeleton of a new plug-in based on the template. The programmer will need to provide a few details, such as plug-in name and description. The newly generated Eclipse project is a complete *Envision* plug-in that can be compiled and used immediately.

The generated plug-in project has support for self-testing and logging based on the *SelfTest* and *Logger* plug-ins. An API header file is created automatically as discussed in appendix C.1. Additionally a plug-in exception class is defined. The generated plug-in includes a single example test case that shows how the self-testing framework can be used. A custom Eclipse launcher is also generated, that will automatically start *Envision* with an option to execute all test cases for this plug-in.

The automatically generated plug-in is a convenient starting point from which a developer can add arbitrary custom functionality this new module.

Appendix F

Node extension mechanism of `ExtendableNode`

The `ExtendableNode` class provides convenient functionality that simplifies the creation of new nodes and allows existing node types to be extended by plug-ins thereby adding new attributes to them. Here we will explain with an example the implementation details of this mechanism. The implementation is a variant of the `Properties pattern` which is described here: `http://steve-yegge.blogspot.com/2008/10/universal-design-pattern.html`.

`ExtendableNode` and each class deriving from it contains a static member structure that contains meta data. This meta data describes exactly what attributes a class has. Attributes are child nodes and are specified by a name, a node type and three flags - optional, persisted and partially loaded.

When a class `Foo` derives from `ExtendableNode` it should specify what attributes it has by registering them with its associated meta data object. When a particular instance of `Foo` is created, the new instance's constructor does not do anything. Instead the base constructor of `ExtendableNode` initializes the object. The object contains a table (vector of vectors) of child nodes which is initialized according to the meta information of all derived classes. Once the object is initialized, it can be used just like any other object by calling methods directly on the instance.

This scheme also allows for arbitrary extensions to be registered for a node type. For example it is possible to register the extension `Bar` for all nodes of type `Foo`. This will simply insert additional attributes in the meta information object for class `Foo`. When new instances of that class are created they will automatically have these new attributes. Thus an extension should only be registered before any instances of an object exist. To use an extension the programmer can simply call the `extension` method on any object deriving from `ExtendableNode`. This will return a pointer to an extension instance object which can be used to access the attributes defined by the extension. If the requested extension was not registered with the target node type, a NULL pointer will be returned.

The `nodeMacros.h` file contains many convenience macros which simplify the use of this extension mechanism.

Bibliography

[1] C.P. Team. CMMI® for Development, Version 1.3. 2010.

[2] A. Hars and S. Ou. Working for free? Motivations of participating in open source projects. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, page 9. IEEE, 2002.

[3] W. Harrison, H. Ossher, and P. Tarr. Software engineering tools and environments: A roadmap. *The Future of Software Engineering*, 2000.

[4] R.N. Shepard. Recognition memory for words, sentences, and pictures1. *Journal of Verbal Learning and Verbal Behavior*, 6(1):156–163, 1967.

[5] L. Standing. Learning 10000 pictures. *The Quarterly Journal of Experimental Psychology*, 25(2):207–222, 1973.

[6] D. Asenov. A feasibility study for a general-purpose visual programming system. `http://www.pm.inf.ethz.ch/education/theses/student_docs/Asenov_Dimitar/Report`, 2010.

[7] S. Schiffer and J.H. Fröhlich. Visual programming and software engineering with Vista. *Visual object-oriented programming: concepts and environments*, pages 199–227, 1995.

[8] B.W. Chang, D. Ungar, and R.B. Smith. Getting close to objects: Object-focused programming environments. *Visual object-oriented programming: concepts and environments*, pages 185–198, 1995.

[9] W. Citrin, M. Doherty, and B. Zorn. The design of a completely visual object-oriented programming language. *Visual Object-Oriented Programming: Concepts and Environments. Prentice-Hall, New York*, 1995.

[10] T.R.G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[11] G. Little and R.C. Miller. Translating keyword commands into executable code. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, page 144. ACM, 2006.

[12] G. Little, T.A. Lau, A. Cypher, J. Lin, E.M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 946. ACM, 2007.

[13] G. Little and R.C. Miller. Keyword programming in java. *Automated Software Engineering*, 16(1):37–71, 2009.

[14] A. Bragdon, R. Zeleznik, S.P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J.J. LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512. ACM, 2010.

[15] A. Bragdon, S.P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J.J. LaViola Jr. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 455–464. ACM, 2010.

[16] R. DeLine and K. Rowan. Code Canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 207–210. ACM, 2010.

[17] K.N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109–142, 1997.

[18] KDE e.V. KDE TechBase. Policies/binary compatibility issues with c++. http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++, accessed March 2011.