Bachelor Thesis

January 20, 2013

# Interactive Exploration

## A Touch-Based Visual Studio Extension for Software Exploration

**Christian Lüthold**

of Frauenfeld, Switzerland (09-714-981)

**supervised by**

Prof. Dr. Thomas Fritz

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

# Interactive Exploration

## A Touch-Based Visual Studio Extension for Software Exploration

**Christian Lüthold**

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

**Bachelor Thesis**

**Author:** Christian Lüthold, christian.luethold@uzh.ch

**Project period:** 23.07.2012 - 23.01.2013

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

# Acknowledgements

I would like to thank several people for supporting me during the time I wrote this thesis. First and foremost, I thank Prof. Dr. Thomas Fritz for giving me the opportunity to realize this project at the *Software Evolution and Architecture Lab* and for his support in every phase of my work. As there exists a very spare amount of literature or documentation concerning the development of a Visual Studio plug-in, I appreciate the nice touch of István Novák, who offered me the material of his unpublished book about the fundamentals of Visual Studio Package development for free. Furthermore, I would like to thank Štěpán Šindelář for his preliminary efforts in order to make the *Graphviz*[1] base functionalities available for WPF developers. In addition to that, I could benefit from his fast and viable replies. Another acknowledgement goes out to David Shepherd, whose professional opinion was of valuable use when implementation decisions were concerned. Last but not least, I thank all those people over the world, who tried to help me out or giving me hints in public developer communities and forums whenever possible.

[1] http://www.graphviz.org/

# Abstract

With the paradigm shift from simple mouse and keyboard interactions to more intuitive and natural input mechanisms, one software takes advantage of such novel *Natural User Interfaces* after another as they allow new ways of how users can interact with digital content. Instead of simple clicks and keystrokes, more advanced movements can be captured and interpreted. In case multi-touch enabled screens are employed, a wide range of natural finger or hand gestures could act as commands. Even though such touch screens are omnipresent in our daily environment, yet little effort has been put in the research of their application in the area of software exploration.

We tackle this lack of corresponding exploration tools by the development of a *Microsoft Visual Studio* extension, which will suit the purpose of a touch driven source code investigation. This plug-in, named *Interactive Exploration*, provides the opportunity to use simple touch gestures in order to analyse code elements and their dependencies. A well-structured graph serves as mental model, allows the navigation along entity relations and supports the annotation of particular elements.

The conducted evaluation revealed that our approach fulfils the necessary requirements in order to be intuitive. The design of the user interface, as well as the supported gestures and the layout of the so-called *Exploration Graph*, were highly appreciated by the evaluators.

# Zusammenfassung

Durch den Paradigmenwechsel von simplen Maus- und Tastaturbefehlen zu intuitiveren und natürlicheren Eingabemechanismen wechseln mehr und mehr Programme auf die Verwendung solcher *Natural User Interfaces*, da diese den Benutzern neue Wege der Interaktion mit digitalem Inhalt erlauben. Anstelle von simplen Klicks und Tastaturanschlägen treten fortgeschrittenere Bewegungen, welche mittels Sensoren aufgezeichnet und interpretiert werden. Bei der Verwendung eines Multi-touch basierten Bildschirms können verschiedene natürliche Finger- oder Handbewegungen als Eingabebefehle dienen. Obwohl solche Touchscreens in unserem Alltag praktisch omnipräsent sind, wurden bisher wenige Nachforschungen angestellt, die den Einsatz solcher Technologien im Bereich der Untersuchung von Software erforscht hätten.

Mit unserer Arbeit nehmen wir diesen Mangel an entsprechenden Werkzeugen in Angriff. Durch die Entwicklung einer *Microsoft Visual Studio* Erweiterung stellen wir ein solches bewegungsgesteuertes Analyseprogramm zur Verfügung. Dieses Plug-in mit dem Namen *Interactive Exploration* bietet die Gelegenheit bestimmte Elemente des Quellcodes, sowie auch deren Abhängigkeiten, mittels einfachen Gesten zu untersuchen. Ein sauber strukturierter Graph dient dabei als mentales Modell und erlaubt die Navigation entlang solcher Entitätsbeziehungen. Ausserdem unterstützt er die Annotation bestimmter Elemente.

Die nachfolgend durchgeführte Evaluation legte offen, dass unser Ansatz die nötigen Anforderungen einer intuitiven Verwendung erfüllt. Sowohl das Design der Benutzeroberfläche, als auch die unterstützten Gesten und das Layout des sogenannten *Exploration Graph*, wurde von den Bewertern wertgeschätzt.

# Contents

# List of Figures



# List of Listings

# Chapter 1

# Introduction

## 1.1 Motivation

With this thesis we tackle the implementation of *Interactive Exploration*, a *Microsoft Visual Studio 2010* extension, whose major task is to support developers in the process of source code exploration. In order to assist, we focus on a tidy visual representation of the current workspace as well as modern interaction mechanisms. Our essential motivations are forthwith presented.

### 1.1.1 Code Elements and Relations

The architecture of a software system is the fundamental organisation of code components and their relationships. During the development phase, the code base of a software project changes and expands in its quantity. This evolutionary process leads to a decay of the initial design, such that the finished software might differ in its content. The recovery of architecture therefore is an important reverse engineering activity in order to understand what design decisions were made during the development phase. Many approaches and tools to leverage this recovery processes exist already. Most of them are directly integrated into the respective development environment. Although those numerous aids and assistants have been introduced and allow the users to drill down and refine views from a high-level perspective of code elements, little emphasis is put on the importance of the dependencies between them [LL07].

The enrichment of software engineering tools with the ability to show code component relations, like call graphs, inheritance diagrams or the visualization of code-flow, might be of importance and bear great potential for supporting the understanding of large evolving software systems.

### 1.1.2 Natural User Interfaces

It is a very well known fact that the creation of software is a complex undertaking. The processes of designing, developing and maintaining software systems become more difficult and require suitable and supportive tools. Today, the majority of such tools forces their users to interact with them by using mouse and keyboard input. Interestingly, the emerging field of so-called *Natural User Interfaces* provides new ways of interactions with digital content. These new technologies avail themselves of sophisticated sensors to acquire input data of any form and transform them into commands. One of the most famous approaches is the multi-touch technology that nowadays is a common way to operate with many mobile devices by means of habitual finger and hand

movements. Such *Natural User Interfaces* allow their users to interact in a more intuitive and — as the name suggests — natural manner.

An obvious conclusion is to take advantage of such input mechanisms and try to improve existing software engineering tools or to come up with totally new approaches. In combination with adequate abstraction and meaningful representation of information, these new ways could be particularly helpful.

## 1.2 Outline

Having presented our motivation for a novel implementation of a software exploration tool, we continue in the following chapter with the listing and explanation of several approaches, whose principles or features are related to our work. Chapter 3 then analyses weak points and compromises of some of those existing tools and proposes appendages of amelioration. Furthermore, it contains a concrete definition of our main goals. The successive Chapter 4 is dedicated to encapsulate concrete details about the realization of our approach, namely the *Interactive Exploration* extension. Among the features and limitations of our software, we also provide information about the most difficult challenges and third-party tools that we employed in order to achieve our goals. Chapter 5 reports our methodologies of evaluation. It contains a usability study, that was applied to a prototype of the extension, and presents the corresponding results. Additionally, the final version of the plug-in is checked for the compliance with our goals. Finally, we conclude our thesis with Chapter 6, where we list our contributions and suggestions for possible future work.

# Chapter 2

# Related Work

This chapter provides an overview about several state-of-the-art approaches concerning information visualization techniques, source code exploration and navigation in the context of software engineering activities. Furthermore, some findings about the usage of *Natural User Interfaces* and their benefits for collaborative work are presented. Sure, most of the approaches to be mentioned deliver assistance in multiple of these categories, but we extract their core ideas and list them accordingly. After the presentation of the related works, we finally conclude the most interesting features and requirements.

## 2.1 Information Visualization Techniques

### 2.1.1 Diagrams

Quite some research has been accomplished concerning how and why developers use diagrams in their daily work. Cherubini *et al.* [CVDK07] found that such informal notations are often used to support face-to-face communication and they claim that current software engineering tools are not capable of supporting this need, because they do not help developers to externalize their mental models of the code. On account of their findings they demand spatial features from future tools. They also state that conventional levels of abstraction should be introduced to show microscopic details (*i.e.*, mechanics of classes and methods) of the code on the one hand, but also the macroscopic high-level structure (*i.e.*, concepts such as modules and systems) on the other hand.

Lee *et al.* [LMFA08] have also been aware of the fact that current tools could be greatly improved with a better support for diagrams. They not just investigated what kind of diagrammatic tool support is desired, but also when certain content is useful. Furthermore they descried what kind of information such diagrams should contain in any particular context. Developers seemed to generally have many kinds of information they want to see in a diagramming tool. The researches therefore categorized those diagram content requirements among their levels of abstraction (*i.e.*, package, class, method, etc.) and conclude that future tools either must be flexible and adaptive enough or have to understand precisely when and where certain diagrams are utile.

Diagrams are widely used by software developers in order to deal with the demanding level of complexity. Such diagrams, which reach from primitive sketches to high-quality posters, are mostly used to understand an execution behaviour, to refactor code, to explain code to co-workers or to design a user interface. DeLine *et al.* [DVR] at Microsoft Research remarked as well that IDEs capable of tying source code and such diagrams, which serve as mental models, together, do not yet exist. Furthermore they censure the outcomes of this lack of coupling, like the necessity of

switching media or the disorientation that origins from this tool-switching as well as the inability to share diagrams and models with co-workers. In order to improve the support of visual code diagrams in modern development environments, they launched a project to engineer and design a Microsoft Visual Studio plug-in prototype. This plug-in, finally called Code Canvas[1], should carry out developers' tasks from the tabs into an interactive map, which is coupled and synchronised with its underlying source code, contains all of the required information needs and can even be shared among team mates. The goal of Code Canvas is to replace the usage of multiple tabs, which are acquired for each task, with one single map per task. Those maps can be managed and filtered independently, such that instead of having a confusing amount of open tabs, only one tab per task is used and all of a project's documents (code files, icons, user interface designs) are placed onto a code map. Another work by the same research team thus describes the analysis of the diagramming behaviour of several developers to define the design of this map [CVD07]. The final design was basically an architectural layer diagram sprinkled with types containing method signatures. It additionally included features like representations of planned but not yet existing code or colourized identifier fragments to aid visual searching. Because the content of many tasks is mentionable large, monitor space could be overwhelmed by this amount of visualized information. Code Canvas therefore uses a technique called *Semantic Zoom* to show different levels of detail with distinct levels of zoom. An important lesson from this project was that developers assign meaning to the spatial layout of the code and that they become familiar with the layout because they often are working with it for a notable amount of time. This approach therefore allows programmers to be better grounded in the code.

With *Interactive Exploration* we present another Visual Studio plug-in, whose main mission is to deal with the visualization of the explored code. A well-structured single view of boxes and arrows, each containing just a decent amount of relevant information, allows the externalization of the code under discussion. Similar principles as with Code Canvas for navigation and zooming are used in order to help the developers to stay oriented.

### 2.1.2 Dependency Graphs

The object-oriented programming paradigm introduces an implicit arrangement of source code elements into various modules. At the same time, dependencies between them arise and frequently gain much less attention than the modules which are known to represent the complex functionality. Hence Bohnet and Döllner [BD06] combine static and dynamic analysis techniques to extract the function call graphs of certain features. With the aid of this call graph, users are able to explore and interpret their C/C++ source code in a different way than just by endless document scrolling. The fundamental idea behind their work is to provide developers with a visual call graph representation and to assist them in localizing and understanding certain feature implementations. This is an advantageous approach as it reduces the amount of code which has to be inspected. A so-called *Graph Exploration View* provides various kinds of information on the function currently under inspection. The neighbourhood of the method (*i.e.*, callers and callees) as well as the architectural context are made available.

The GEVOL system presented by Collberg *et al.* [CKN+03] also takes advantage of a graph drawing technique for the visualization of large graphs, but additionally considers the evolutionary aspects of software. It aids in the discovery of the structure of legacy systems by visualizing the changes the system has gone through. Precisely speaking, GEVOL extracts information about some Java source code from a version control system and not only constructs call graphs, but also inheritance and control-flow graphs. Furthermore, it displays the changes those graphs have

[1] http://research.microsoft.com/en-us/projects/codecanvas/

gone through since the initial commit. By interpreting those colour-coded graphs, developers are able to answer questions about why a program was structured the way it is.

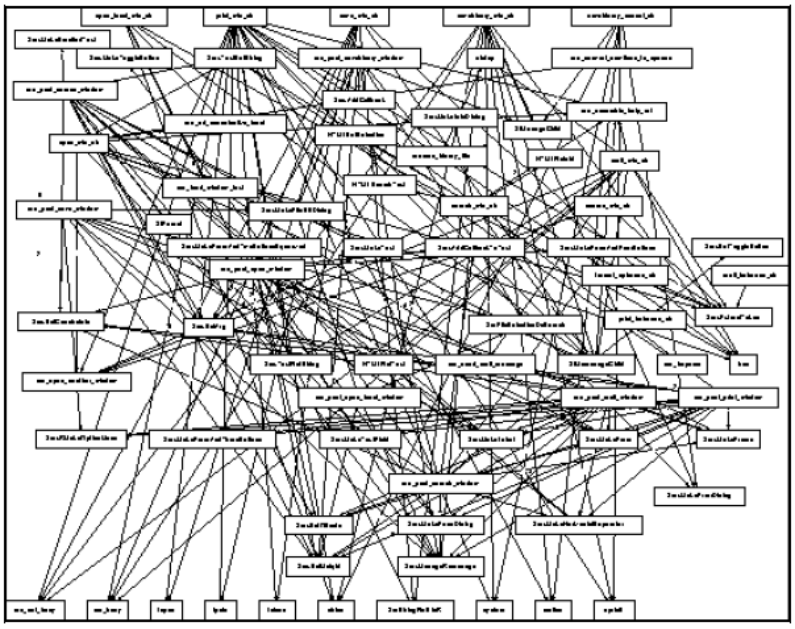

**Figure 2.1**: A complex and unstructured call graph visualization. [YM97]

With large and complex systems come large and confusing graphs. Representing various code elements and their relations on different abstraction layers in a depicted manner requires clever graph layout algorithms or fade-out of information. Otherwise the interpretation of such unstructured graphs, like the one in Figure 2.1, requires soon as much of a comprehension task as it would without any graph at all. This is because those graphs differ by far from the structures of the cognitive model created by the developers. Young and Munro [YM97] claim that a large number of tools appear to have put little effort into layout and presentation and thus address those challenges. Under their statement that layout matters, they demonstrate a new view of call graphs that represent the necessary information in a readily understandable and intuitive manner. Strictly speaking, they show the CallStax visualization technique which takes advantage of a 3D representation and shows function calls as stacks in a virtual environment. CallStax benefits from it's flexibility as the edges (*i.e.*, the function calls) are perceived implicitly. Further space is gained by dislocation of code information to other views or by representation of details, such as metrics, by visual conventions and metaphors.

Hassan and Holt [HH03] complain about other shortcomings of dependency graphs, namely the lack of annotation of dependencies. As they don't want to miss the rationale (*i.e.*, the reason behind why a dependency was introduced or removed), the time of existence, the inter-dependency-patterns nor who created a specific dependency, they propose the usage of so-called *Annotated Dependency Graphs* (ADG). Edges show not only which entities are related but also contain the listed information.

The *Exploration Graph* of our extension is a dependency graph as well, although it does not

represent control flows. Instead, statically read out call-relations and inheritance hierarchies are visualized in the best possible tidy manner. Furthermore, annotation of graph nodes is supported, however, we do not yet provide the possibility to assign meaning to the explicitly displayed edges. Also, our plug-in is not able to deliver information about the evolutionary aspects of the analysed software.

## 2.2 Source Code Exploration

Often the makers of software exploration tools structure the code of the analysed project into modules which represent software element entities like packages, classes or methods. Softwarenaut, a prototype for the exploration of large software systems and introduced by Lungu and Lanza [LL06], follows this idea and allows the hierarchical top-down decomposition of such modules. By starting with a high-level view and continuously refining it with operations and filters, relevant views of the architecture can be found. Additionally to an exploration perspective, which shows the modules and their relationships in a graph-like representation, a detail perspective, offering details about the selected entity, and a map perspective to preserve orientation within the hierarchy are provided. The exploration view makes use of various visual metaphors in order to encode information about the code. Edges are coloured among their relation types (*i.e.*, invocations and inheritances) and the entity-representing nodes are squares. The size of the area of each square is proportional to the size of the incorporated module. But also the edges may have different stroke broadnesses to indicate distinct cardinalities.

Many more software exploration tools use diverse views that are of valuable assistance. However, Favre [Fav01] describes that the generation of all those different kinds of software views is not cost effective. This is due to an impressive amount of diverse types of entities, relationships, software models and their representations. On top of that, multiple different perspectives (*e.g.*, abstraction layers) could be of interest. Therefore, Favre introduces an object-oriented, dynamic and easily adoptable framework to overcome this problem. $G^{SEE}$ (Generic Software Exploration Environment) provides functionalities to specify and create new exploration tools very easily. It does not depend on a particular kind of data or visualization and is hence easily customizable such that views and graphical representations for any data sources can be configured.

Another interesting approach of how large and complex software systems can be explored arose from the research of Bragdon *et al.* [BRZ+10]. They built a prototype IDE user interface, called Code Bubbles, for the exploration of Java code. The strategy of this novel user interface is to start with an empty canvas and add items as the user searches and browses the project. The idea behind is to show multiple editable fragments simultaneously and to support high-level interactions between working sets. This should reduce navigations and support developers to perform complex code understanding tasks. Bubbles serve as metaphors for investigated code fragments since they - in contrast to windows - have a minimal border decoration (*i.e.*, scroll- and tilebars) and because bubbles are light, they push each other away in case overlapping occurs. Beside this repositioning of bubbles, support for automatic syntax-aware reflow and elision of code is granted. This functionality serves as mechanism to maintain a justifiable level of abstraction. Bubbles can for example be created to enable the search for certain code snippets within the project. Classes and methods that contain matching results will be presented in further bubbles which then can be filtered to refine the results. As one navigates through the found relevant code, new bubbles for each method that is inspected will be created. All those editable bubbles are linked together to indicate the hierarchy of the project architecture.

*Interactive Exploration* provides a hierarchical overview representing the different code levels. Projects, types and methods are neatly nested in order to offer an overview of the context under exploration. As with Code Bubbles, the initial workspace is empty and elements are continuously added. It is not mandatory to start with a single entry, but rather a whole subset of entities can serve as the origin of interaction. The whole plug-in takes advantage of colours, conventions and metaphors as well since they significantly support the cognitive factor.

## 2.3 Source Code Navigation

Ko *et al.* [KMCA06] found that developers spend on average 35 percent of their time performing navigation when they try to understand an unknown code base. The researchers observed that developers typically searched first manually or with the aid of search tools for a relevant entry point. Afterwards, they usually follow in- and outgoing dependencies of the located code.

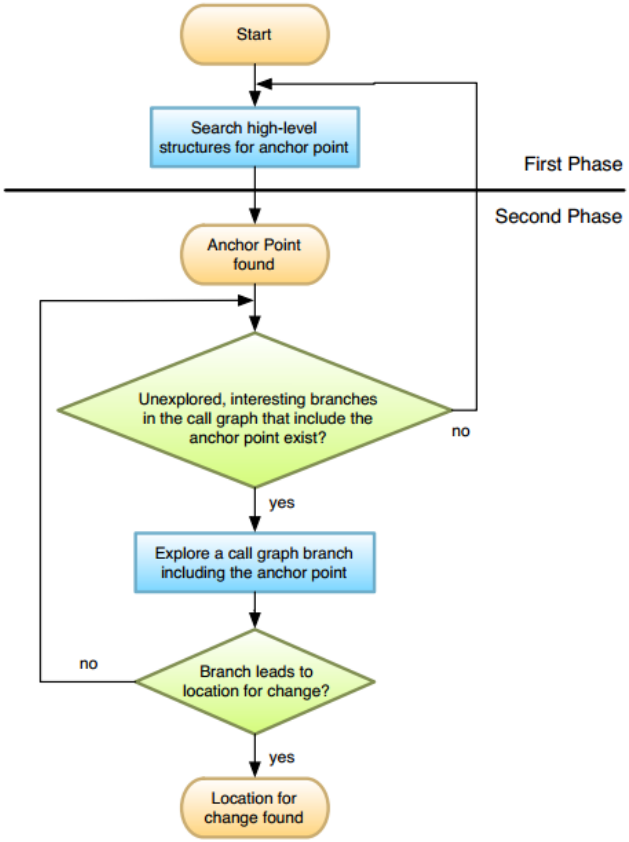


**Figure 2.2**: The two-phased navigation behaviour model. Phase 1: searching an entry point. Phase 2: navigation through the call graph. [KKD+11]

A similar two-phase model regarding the method call graph has been addressed by Krämer *et al.* [KKKB12]. Concerning this model, shown in Figure 2.2, developers search for an anchor

point in phase one and then proceed with following different paths until they find the code fragment they want to work with. To simplify the process of back-and-forth navigation (*i.e.*, the second phase), Karrer *et al.* [KKD+11] focused in an initial step on the development of Stacksplorer. Stacksplorer is a plug-in for the *XCode* programming environment and visualizes the call graph neighbourhood of a certain method and supports the navigation through it. At the same time, it provides additional information scent, like names and classes, about those calling and called methods. Furthermore, paths that have been traversed can be annotated and bookmarked for later use. In order to support the first phase (*i.e.*, the search for an entry point) as well, Krämer *et al.* [KKKB12] introduced Blaze, yet another XCode plug-in, which works pretty much like Stacksplorer. In contrast, the paths of the method, which the developer is focussing, are automatically updated and important additional information is provided. When the entry method is found, it can be locked and thus the anchor point is made explicit. With both tools the intention of the researchers was to reduce the risk of introducing side-effects while maintaining software by making available an easy to use navigation through callers. On the other side, stepping through called methods should help developers to better find and understand the implementation details.

In order to improve how complex source code and its information spaces can be explored, Storey *et al.* [SBM+02] developed SHriMP (Simple Hierarchical Multi-Perspective) which allows the investigation of several different perspectives and abstraction levels. Although much effort has been attached to how the informative data should be visualized, SHriMP merges different sources of information to enhance the way how developers browse and explore. That is why it makes use of animated panning and zooming motions and provides continuous orientation and contextual cues to developers while they navigate through the different perspectives.

By means of a hierarchical overview and a method filter, a suitable or specific entry point can be found with our Visual Studio extension too. The navigation along incoming or outgoing dependencies is possible as well, however, the neighbourhood is not shown all the time and visible only on command. Among these contextual cues, we also enable animated panning and zooming in order to keep the user's orientation. On the opposite, we did not realize different abstraction levels.

## 2.4 Multi-touch Collaboration

Jetter *et al.* [JGR+12] demonstrate the high relevance of modern user interface evolution. Various approaches, such as touch or motion tracking systems, do their contributions to more natural user interfaces. They describe the movement from the traditional *WIMP* (Windows, Icons, Menus, Pointers) paradigm, where a single user works with mouse and keyboard, to a new generation of interactive spaces with *Natural User Interfaces*. For this reason, Jetter and his colleagues discuss haptical and gestural interfaces like Microsoft's *Kinect*[2] gaming controller - which has found to be not only of interest for gaming due to his sensory capabilities - that captures body movement, gestures and voice as input or tangible user interfaces like the Microsoft *PixelSense*[3] tabletop. They state that such interface types can help not only to exploit our natural motor skills, but also could support face-to-face collaboration and increased group awareness.

Similar statements have been made by Hilliges *et al.* [HTB+07], but they concentrate more on the users and how they behave while working together with such novel technologies. In their study they present a number of design goals for electronic systems to support so-called *Collab-*

[2] http://www.microsoft.com/en-us/kinectforwindows/
[3] http://www.microsoft.com/en-us/pixelsense/default.aspx

*orative Creative Problem Solving*, that is the exchange of knowledge, the coordination of different skills, the interpretation of information and the creation of new ideas. Based on those guidelines, they designed and implemented an interactive environment which runs on a large wall display. In addition, Hilliges and colleagues argue that such collaboratively tangible surfaces not only offer new possibilities for the design of systems that can exploit physical and social requirements of traditional face-to-face meetings, but also benefit from the digital technology. Precisely, they name the storage and reloading of data, the fast access of information and the ability to easily edit and traverse data.

Another concrete approach, where a tangible user interface is used for a collaborative user interaction, has been developed by Müller *et al.* [MWFG12]. In order to make code reviews more desirable, they moved the process of reviewing from a lonesome workspace to a collaborative environment, namely a multi-touch tabletop. Their tool, called *SmellTagger*, provides a graphical guide that leads through the detection and investigation of so-called *Code Smells* which often indicate architectural inconsistencies. Furthermore, problem-relevant code can be annotated (*e.g.*, for later refactoring) by means of audio notes. This approach also comes with various visualizations, like graphs, diagrams and metaphors, and allows the definition of own gestures to interact with the virtual environment. In addition to that, much effort has been spent on making the interaction collaborative for all users around the touch interface. On these grounds, every single view can be duplicated, rotated, resized and removed in order to guarantee the concurrent interaction.

*Interactive Exploration* is another multi-touch driven software exploration tool where the interaction is intended to take place on a large wall screen. Of course, mouse and keyboard can be used as well, however, taking advantage of a touch enabled interface allows multiple developers to simultaneously interact with the content. Rather than visualizing different kinds of diagrams or code metrics, a single dependency graph is the matter of examination.

## 2.5 Conclusion

The analysis of developers' diagramming behaviours has shown that visualization techniques should not only contain information about code elements and support the building of mental models, but also allow programmers to reflect about code and to discuss it with co-workers. They should be helpful in processes like understanding, refactoring, designing or explaining source code. Additionally, such tools must provide mechanisms to support developers in finding those information fragments they need. Metaphors or colour-coded informations could help visual searching, whereas animations, such as panning and zooming, could help users to stay oriented. Information scents, like the neighbourhood of a certain data type, are often used to assist the developer while navigating through the code base. In order to allow fast access to the code for accomplishing comparative tasks, the graphical representation and the underlying code are mostly tightly coupled. As developers seem to have many information needs, exploration tools, which provide different kinds of context-based information and even on various levels of abstraction, would be of great use. Graph-like representations are frequently considered to show the interactions of diverse information or code element types and their dependencies. Some approaches even take evolutionary aspects into account and hook them into the graphs as well. However, great attention has to be paid when visualizing large graphs as the layout might quickly become confusing. The possibility of interacting with tools by means of tangible *Natural User Interfaces* does not only allow developers to use habitual and common gestures, but also aids in collocated team work. Since multiple user inputs can be handled at the same time, team collaboration is boosted and group awareness is increased.

# Chapter 3

# Problem Definition

In this chapter we present the targets tackled by our thesis. First, we show where we found appendages of improvement and why it is relevant to come up with yet another software exploration approach. Hence, we list some requirements our tool has to fulfil and also highlight and discuss the differences to other currently existing tools. Secondly, we concretely formulate the goal of this thesis.

## 3.1 Requirements

We discuss some aspects of widely spread techniques of existing software exploration tools and argue that some parts require further attempts to be observed. We also note some requirements we want to lay the focus on in our work, even though they have been examined in detail.

### 3.1.1 Visualization Space

Most work is still done using source code editors despite the fact that these textual representations cannot easily convey many dependencies and relations of code elements. Many tools accommodate these shortcomings with visual graphs which show all necessary interactions. However, a common issue that quickly arises, is the lack of space for such huge constructs. Even modules that are assumed to be little often contain an immense number of hidden relations. The problem becomes obvious if one catches a glimpse at graphs created by tools like GEVOL [CKN+03] or Softwarenaut [LL07].

Thus, our approach is intended to be of most valuable use when large screens are employed. We will design for such dimensions right from the start like Hilliges *et al.* [HTB+07] did with their interactive brainstorming system. A further advantage that arises from focussing to bigger visualization spaces is the additional room which possibly will allow more users to gain information from the same screen or to simultaneously interact with it.

### 3.1.2 Facility of Inspection

On the one hand, current tools, such as CallStax [YM97], try to help developers staying oriented by providing facile interpretable graphical representations or metaphors. On the other hand, they make available many informations that have been acquired by smart algorithms (*e.g.*, the *Graph Exploration View* presented in [BD06]) while totally plastering the user interface at the same time.

The difficulty is to find the right middle course in order to help the developers and not to confuse them even more. The findings of Lee *et al.* [LMFA08] could thus be of interest as they

already found which information is relevant on which level of abstraction and also when it is important to show it based on the context.

### 3.1.3 Meaningful Graph-Layout

Apart from having enough space to visually present huge graphs and filtering the information overflow to relevant parts, the layout process of that graph should contribute to make the representation as meaningful as possible. Young and Munro [YM97] offered a solution with CallStax to overcome crossing edges by totally hiding them and making the perception of dependencies implicit. The investigation of diagramming conventions conducted by Cherubini *et al.* [CVDK07] have shown that developers often use boxes and arrows in an informal manner to represent entities and relationships. Furthermore, they often observed visual arrangements where the relationships had a dominant flow, such as left-to-right or top-down.

Our mission is to look for an advantageous layout of a 2D graph rather than making use of a novel 3D approach. We further want to provide possibilities to automatically layout the graph in several different manners in order to support the spatial orientation along dependencies (*e.g.*, left-to-right navigation from calling to called methods). On top of that, entities should be grouped into some higher-order structures as this helps to understand the architectural context.

### 3.1.4 Incremental Exploration

The search of relevant code inspection entry points is equally important as the subsequent navigation along incoming and outgoing dependencies. Stacksplorer [KKD+11] and Blaze [KKKB12] both provide mechanisms to support developers in applying these phases where methods are concerned. However, they do not allow users to navigate among multiple paths concurrently. The multi-view visualization proposed by Bohnet and Döllner [BD06] also provides techniques to search and navigate the code base, but contrarily shows all existing paths of a module and thus does not provide ways to incrementally explore and build up only specific dependency paths.

A combination of both approaches might be worth a test, where the two-phased exploration model as well as the simultaneous constitution of particular dependency graphs are supported. Furthermore, giving early notice of possible outgoing travel directions of an entity would serve as information scent and therefore assist the routing towards parts of interest.

### 3.1.5 Visual Orientation Cues

The minds behind SHriMP [SBM+02] showed that the animation of visualizations can aid the developers to stay continuously oriented within the graphical concept of the underlying code. The same principle is used by tools like Code Canvas [DR10] or Code Bubbles [BRZ+10] where the visual content can easily be investigated by zooming in and out and by panning the view such that content of interest is centred. Additionally, a view presenting the affiliation in the explored hierarchy of the visible modules should offer a sense of context and orientation. The *Map Perspective* of Softwarenaut [LL06] demonstrates such an approach.

We do not only want to provide visual information about where a module resides within the hierarchy tree, but also which modules are currently under exploration. On account of this, developers will be able to keep track of the architectural nesting and the selected subsets simultaneously. The tool should also provide capabilities to zoom out to yield a better insight about the overall structure.

### 3.1.6 Touch-based User Interaction

Several efforts have already been made in order to move the user interaction paradigm from only keyboard and mouse input to more natural input methods. Those modern input mechanisms promise to be more intuitive as habitual hand and finger movements can be used to interact with the tools. *SmellTagger*, introduced by Müller *et al.* [MWFG12], demonstrates one approach where a multi-touch display serves as input interface. Their research results have shown that the exploration of source code can be executed using touch input only.

With our tool, we will also support a touch-based interaction in addition to the traditional input methods. However, action triggering gestures have to be simple in order to facilitate the paradigm shift to more sophisticated user interfaces. Catching a glimpse at some of the latest mobile devices, such as smartphones or portable tablet computers, bares that many manufacturers use the same facile touch gestures to interact with the displayed content. Because of this observation, we want to take advantage of those well-known gestures for the interaction and navigation of our tool too. Apart from switching the kind of interaction, multi-touch tables or wall displays also open new ways for developers to operate collaboratively. This is not only due to the bigger screen size, but also to the fact that multiple users are able to draw their gestures on the surface in order to interact with the tool concurrently.

## 3.2 Goal

The goal of this thesis is to develop a new software exploration tool, which deals with the visualization of code elements and the relations between them, while all aforementioned requirements should be respected. Concretely, the two-phased exploration model (see Figure 2.2), namely the search and concrete selection of one or multiple specific code elements, as well as the subsequent navigation along dependencies, has to be supported. The representation of those code elements and their dependencies should be visualized by means of a dynamic graph to allow the consecutive attachment of more and more nodes and edges. Furthermore, the finished tool has to be able to cope with the emergence of increasing constructs. This visual code navigation approach should assist developers in reviewing, analysing and annotating existing code by making available a well structured visualization to support their individual mental models of the selected code fragments. A moderate employment of the available space used for the presentation of information is prescribed. In addition, this software should enable developers to use multi-touch interactions in order to operate with it. Simple, intuitive and natural gestures to manipulate and navigate through the visualized content should be provided. In particular, program elements and their relationships should become incrementally explorable by means of simple touch gestures.

The user interface has to be structured for the primarily use of a huge wall display, namely the *PQLabs G3 Monitor*, which serves as touch interface. More details about this screen can be gleaned in Appendix A.

The finished version of our approach should finally run as a *Microsoft Visual Studio 2010* extension and make available its features to .NET C# developers. The plug-in should come with its own views that directly integrate with the rest of the programming environment.

# Chapter 4

# Approach

In this chapter we provide information about the concrete realisation of the Visual Studio extension called *Interactive Exploration*. Among features and their implementation details we present also the major challenges we had to address. We thus cover some workarounds that were introduced to lessen their impact. In addition to that, we further demonstrate and discuss the underlying graph calculation engine.

## 4.1 The Visual Studio Extension

This section is intended to deliver an overview about the extension we developed. We hence list the implemented features and show how they are intended to be applied. Also part of this section are limitations that were made to our software, as well as statements to reason their absence. Further topics then discuss more concrete aspects of our work, such as the basic architecture of the plug-in and some implementation details.

### 4.1.1 Features

The *Interactive Exploration* tool comes with a bunch of features which will be introduced one after another in the following part. The listing is done in such a manner that the features are discussed in a logical order which would make sense to a new user, since it demonstrates a possible interaction workflow.

#### Extension Tool Window

The *Interactive Exploration* plug-in basically consists of a single view which can easily be embedded between other arbitrary tool windows of Visual Studio. The user is allowed to dock the extension's frame wherever he wants since the graphical user interface dynamically adapts its size and allows for a user-specific layout of the personal workspace. However, in case one likes to take advantage of a huge display, the recommended way is to represent the extension's tool window in full screen mode. The composition of the whole user interface is pictured in Figure 4.1. Like any other integrated development tool, our extension received an entry in Visual Studio's tool menu. The interactive plug-in content can be opened as soon as a solution is loaded within the IDE. The compendium of *Interactive Exploration*'s single graphical representation consists of three parts, whose meanings will all be discussed in detail later on. The intention here is to provide one main part, that contains the explorable content through which can be navigated. The other two parts provide additional views, that encapsulate information about the analysed context, but are

not meant to be of importance all the time. For this reason, those two panels can easily be hidden whenever they are not necessary for the current task at hand. This allows the main exploration view to consume more space and to layout its focused elements in a broader range.

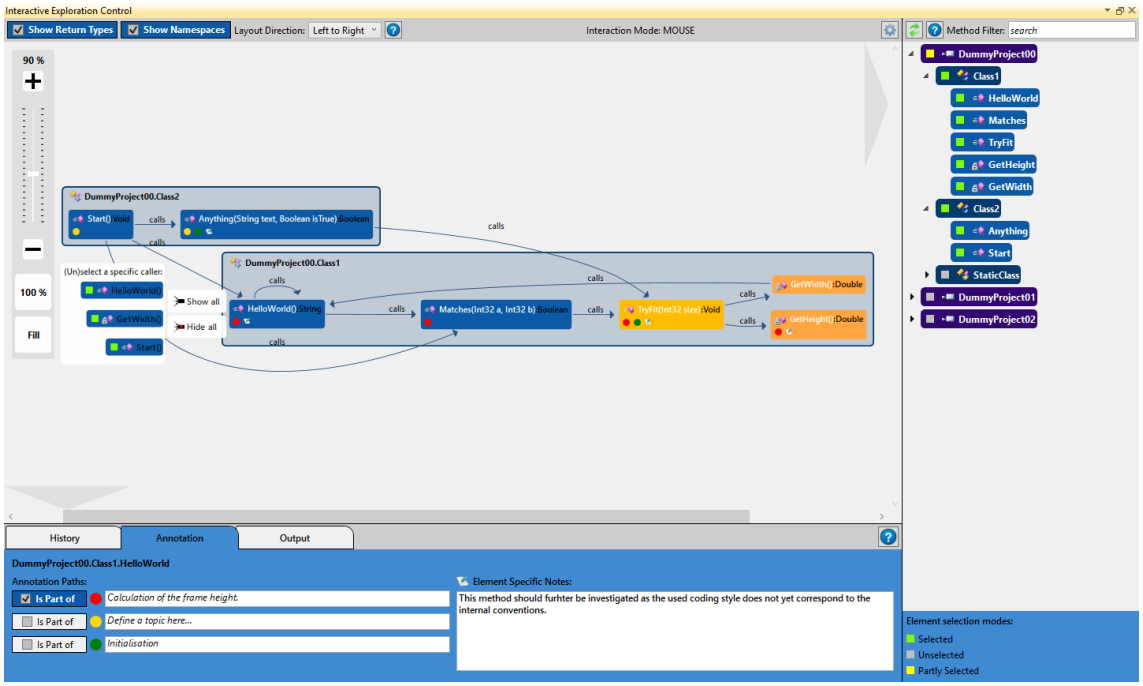


**Figure 4.1**: The whole user interface of the *Interactive Exploration* extension. All parts are discussed in this chapter.

## Hierarchical Overview

The first of the two aforementioned additional parts is reserved for an overall hierarchical overview about the underlying and analysed solution. All elements of the current workspace are aggregated and displayed in a meaningful manner as can be seen in Figure 4.2. This means that all projects the solution consists of are alphabetically listed, as well as every type which is defined within one of them. If a type element represents a class, all its specified methods are itemized in the hierarchy as well. Having an optionally visible view, dedicated to the entire code element hierarchy, makes sense to the user as he or she most likely wants to gain an overview of the context. This is also based on the fact that Visual Studio 2010 does not itself provide an inbuilt representation of the workspace in such a manner. The very well-known *Solution Explorer* indeed makes available a similar hierarchy, but it only lists the solution's content in a file based manner, that also shows the projects and the code files associated with them. Such an approach does not guarantee that every defined type is represented as most programming languages allow the definition of multiple types in a single file. Furthermore, the Solution Explorer does not contain any information about methods at all. In order to explore the solution's code elements, the user either may have a specific entry point or want to search for an interesting one. The hierarchical overview in addition is intended to serve this reason. A developer interested in a specific method can make use of the provided search box which is able to filter the hierarchy among a key word. This al-

lows for a more precise and quicker discovery of a starting point. Any element can be selected or unselected, no matter whether a whole project wants to be inspected or only a specific method of a certain class type. A little coloured square beneath every selectable item indicates information about the current selection status of the element such that the chosen parts of the solution can easily be distinguished from the rest. Once a code element is selected, it gets inserted into the *Exploration Graph*, which brings us to the next feature.

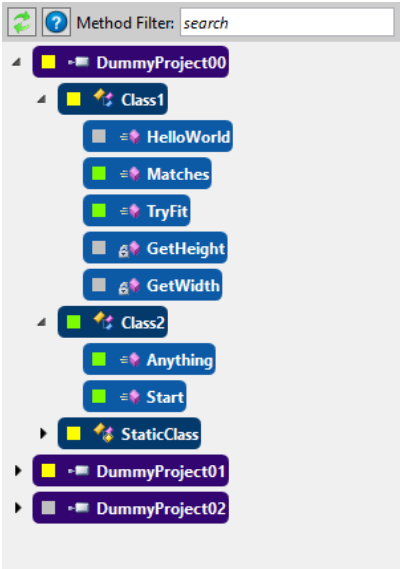


**Figure 4.2**: A hierarchical overview of the solution's content.

## Exploration Graph

As we have already mentioned, the main part of the extension's view consists of the navigable part, whose duty is to take care of the so-called *Exploration Graph*. This graph, pictured in Figure 4.3, contains all selected code elements and shows their relations. Concretely speaking, every selected method is added as a node. Those nodes consist of the method's signature (*i.e.*, name and parameter list) and optionally its return type. Besides, a little icon is displayed, which provides compressed information about the method, such as its level of accessibility or whether it is static or virtual. In the event of adding multiple methods that are all defined by the same type, the respective nodes are aggregated and drawn nearby each other in a common sub-graph. Sub-graphs thus represent types and encapsulate all type-related information that arise during the exploratory process. It contains the type's name, which optionally can consist of the full .NET namespace description, and again, an icon takes over the compression of further information. Among methods and types, the graph includes two species of code element relations which are added as different kinds of edges. A directed edge between two nodes represents a call-relation between the two connected methods. If, on the other hand, two sub-graphs are linked, a type inheritance relation is represented.

The default layout direction of the graph is left-to-right in order to simplify the interpretation of the visualized call-relations. However, this layout can easily be changed to top-to-bottom for example, such that type inheritances are more intuitively arranged. Of course it is possible to

remove any code element from within the graph in order not to be dependent of the probably hidden hierarchy view every time. This functionality is accessible through a simple little context menu which is available for every method or type in the graph. The successive feature explains how the *Exploration Graph* can be navigated.

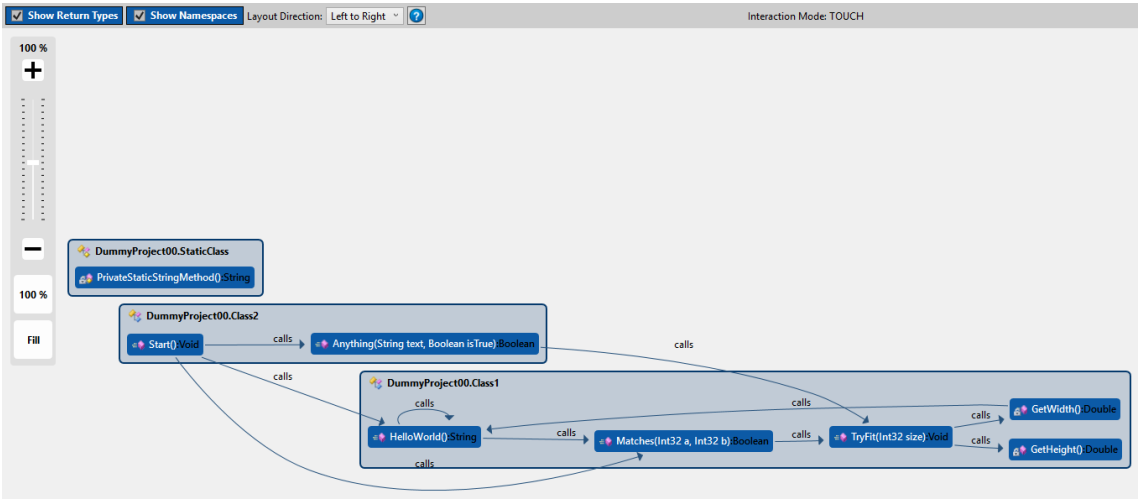


**Figure 4.3**: The *Exploration Graph* contains all selected code elements and shows the dependencies between them.

## Multi-touch Navigation

We announced to support not only the first step of the two-phased navigation behaviour model (see Figure 2.2), but also the subsequent walk through the call relations. For this reason, several navigation options have been added to the graph's nodes. Every displayed method element is able to open a menu, containing a list of all called methods, to it's right side of the node (see Figure 4.4), or another menu to the left, providing information about every calling method respectively. Those listings should help the user to analyse the direct neighbourhood of a method. Among the enlisted names of calling or called methods, also their particular selection states are indicated, which allows to gain a quick survey about whether they are already represented by the graph or not. By means of selection or unselection of such an enlisted item, the relative method element is added or removed from the graph. In addition, two buttons per navigation menu accompany the list. The first one allows all itemized methods to be added at once, the second button removes them respectively. The graphical user interface of the *Exploration Graph* is touch-enabled, meaning that a user can interact with it by using simple touch gestures. To open the navigation menu to head for a method's callees, a simple swipe gesture to the right on top of the method node is sufficient. The same applies for a left swipe, where a navigation menu with callers pops up. A single tap on a node in the graph either closes open navigation menus or toggles the visibility of the already mentioned context menu.

Depending on the user's needs, quite a bunch of nodes and edges may arise while increasing the size of the graph. Assuming a scenario, where the graph is large enough to not fit the boundaries of the view any more, the exploration view provides a zoom control (depicted in Figure **??**), which allows the graph to be zoomed out or in, such that its whole extent is visible without any border overflows. This functionality is controllable using touch gestures as well. In order to move

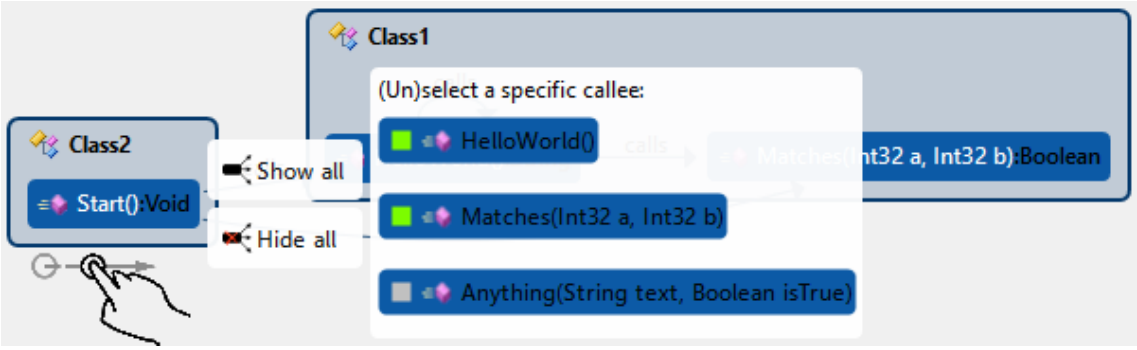


**Figure 4.4**: By means of a swipe gesture to the right, the navigation menu for called methods can be opened.

the current viewport around, one finger can be used to pan the graph. The zoom factor can easily be changed by means of two fingers. Pinching the distance of the two fingers initiates the view to zoom out, whereas spreading the distance zooms in. This interaction approach is very well known as *Pinch-to-Zoom* and is almost always used when content displayed by a touch screen has to be resized.

## Code Element Annotation

The second part of the extension's user interface that optionally can be retracted consists of three tabs. One tab is reserved for a control area which deals only with the annotation of the various code elements (see Figure 4.5). Any type or method can be assigned with different kinds of annotation. In order to focus any sub-graph or node, it is enough to touch it and make it automatically matter to the annotation control. The simplest way of annotation is to write an element-specific note and attach it to the particular element. The control provides a text box where comments, critics or questions can be written, edited and read. Once such a memo has been composed, an icon is added to the representative in the graph so as to give hints where comments are deposited (see Figure 4.1). Another possible way of annotation is to first define a topic due to which the source code is explored. The annotation tab allows to specify three such topics at the same time and adds an identifiable colour to it. As the *Exploration Graph* is investigated and code elements are found to be relevant to the topic, they can be made members of it by simply adding a tag to the elements. This tagging just adds a little circle of the topic's colour to the sub-graphs or nodes. This procedure allows for example to create and be aware of travelled paths or to just make groups of elements which are relevant to certain inquisitions.

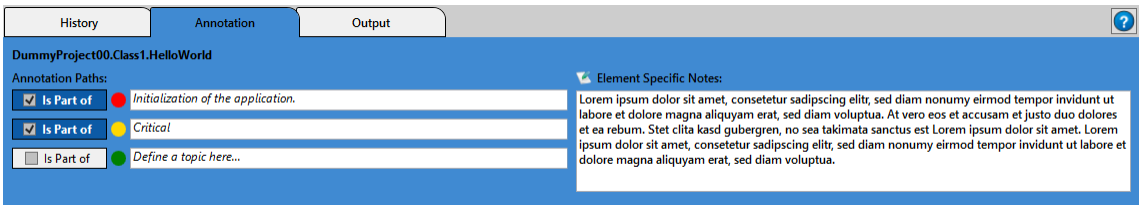


**Figure 4.5**: The annotation control allows to add elements to a certain annotation path or to compose a specific note.

### Exploration History

As a user interacts with the *Exploration Graph*, selects and removes methods or types and follows call-relations, he probably wants to browse his navigation history at some point. To support this, a further feature has been added to the extension, namely a history which contains all those steps. Hence, another tab, shown in Figure 4.6, is provided by the extension's interface, containing this information and listing an item for every interaction step. All entries possess a detailed description of what exactly happened to the graph as well as a separate button. This specific button is dedicated to permit the user to reverse any interaction in the history. In case a method has been unintentionally selected, the button can be considered to undo this step. This not only allows to correct any possible missteps, but rather gives the opportunity to turn back the clock at a later moment of the exploratory process. Once a change in the graph has been made undone, the discussed button alters its functionality. Concretely speaking, clicking this button again would result in the representative change to be redone. The whole behaviour is similar to the well-known undo- and redo-buttons that are provided by a variety of editor tools or web page browsers. However, the history control of *Interactive Exploration* admits the user to intervene in any point of time and not only in a stacked manner.

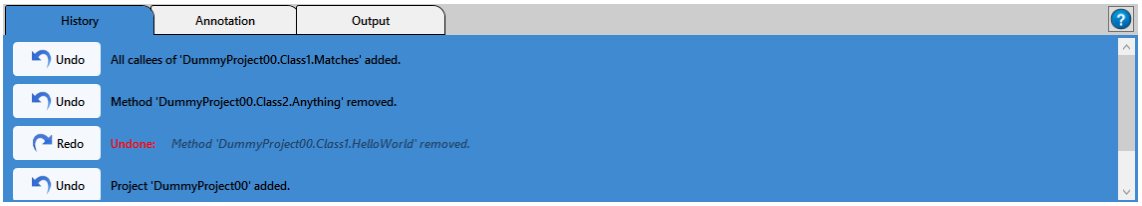


**Figure 4.6**: The history control allows to undo and redo certain modifications that were made to the *Exploration Graph*.

## 4.1.2 Limitations

Although our Visual Studio plug-in has some useful abilities, it does not yet provide all of the most important features for software exploration. We will briefly discuss some lacking functionalities which are exposed to future work (see Section 6.3).

### Dynamic Runtime Behaviour

*Interactive Exploration* is only intended to support software investigation of Visual Studio projects that are developed with the C# programming language. Since C# operates under the rules of the object-oriented programming paradigm, the real runtime behaviour cannot be precisely predicted by just statically analysing the source code. This issue can especially be proven when call-relations are considered. The object-oriented approach allows function calls to be dynamically bound at runtime. It is not possible to deal with this polymorphic behaviour in advance as the *Common Language Runtime* decides, based on the running code's context, which inheritance layer should receive the call. Another but similar problem is that of dynamic callback mechanisms and delegates, where again the runtime correctly assigns the responsibilities.

The current implementation of our extension's code element and relation assembling algorithm does its work just in a static manner. In case of polymorphism or callbacks, we are not able to explore those possible dependencies using the *Exploration Graph*.

### Direct Source Code Access

It is beyond dispute that a good software exploration tool should provide access to the source code of the currently inspected code elements. Catching a quick glimpse of the implementation details is essential for the developer in order to decide, whether a certain type or method is relevant for the exploration or not. The code contains valuable information about the algorithms and architecture. During the navigation process, it is important to have the possibility to compare the bodies of two methods, especially if they are call-related, such that the software's logic can be interpreted and analysed. The user should be given a chance to rapidly decide, whether he understands and accepts the code, or thinks that rather a notice or comment in form of an annotation is appropriate.

Unfortunately, *Interactive Exploration* does not yet support this very inspection method. This is due to the fact that the *Visual Studio 2010 SDK* did not provide an apparent or documented way to access any code element of the solution and therefore we used a workaround to get the solution's content at all. This issue will be explained in more detail in Section 4.2.

### Levels of Abstraction

Right from the start, we wanted to give the user the ability to have a look at the *Exploration Graph* from different perspectives. With the introduction of abstraction layers, more or less information in the graph should be shown in order to deal with the visual complexity of a possible huge construct. We therefore added a further graph to our extension, which represented just types and their inheritance relations, while methods and call-relations were totally hidden. Unfortunately, the graph layout calculation unity was unable to cope with the simultaneous computation of multiple graphs. This resulted in incomplete graphs with broken edges and empty sub-graphs in every second case. We will present the underlying mechanisms and their limitations in more depth in Section 4.3. Due to this fact, we were forced to removed this problematic type graph and leave the implementation of well-oiled levels of abstraction to future investigations.

### Save, Export and Share

Although the plug-in works correctly and encapsulates some neat features to build up an extensive graph, there is no saving mechanism that would allow the storage of the assembled workspace yet. Such a functionality would be of good use to the user as the current state of work could be interrupted and readopted at a later date. However, at the moment, our extension does only allow to explore a Visual Studio solution for a single session.

A further feature we discussed during the requirements analysis, was to integrate an option to export the current workspace. As soon as such a feature would be available, graphs could easily be shared among co-workers. There indeed exist other software exploration tools that enable their users to share content in order to support better collaboration and team work. However, this feature got a very low priority and was, like the ability to save everything, not part of the final release due to given limitations of such a thesis.

## 4.1.3 Architecture

This chapter is dedicated to give a concrete insight into the architecture of the *Interactive Exploration* plug-in. First, we explain how Visual Studio can be extended and what mechanisms were used in order to host our software. We then talk about its architectural pattern and cursorily dive into the data model that forms the basis of the whole implementation.

## Extension Point

The architects of the *Microsoft Visual Studio* programming environment made their software extendible in many forms. Anyhow, the right choice between those different and unequal powerful extension methods has to be made in order not to lose access to certain functionalities. Among macros, add-ins, text-editor specific plug-ins (*i.e.*, using the *Managed Extensibility Framework*[1] (MEF)) and packages, we decided to go for the most powerful of all extension mechanisms, namely the *Visual Studio Packages*. The majority of all functionalities of the IDE, including programming languages, editors, the debugger and the project system, are implemented by means of such packages. From a developer point of view, adding a new package to Visual Studio is just like appending core functionality to the environment as if it were developed by Microsoft. Consequently, Visual Studio is not a monolith application, but rather a set of so-called *Dynamically Linked Libraries* (DLL). The core functionality, called the *Shell*, hosts independent packages as soon as they are needed. This composition is depicted in Figure 4.7. Furthermore, they are able to communicate and co-operate with each other through services and automation objects. The power of any package comes from its ability to use the core services of the IDE. Furthermore, they can provide own windows and views which can be directly integrated and placed between other tool windows of the environment.

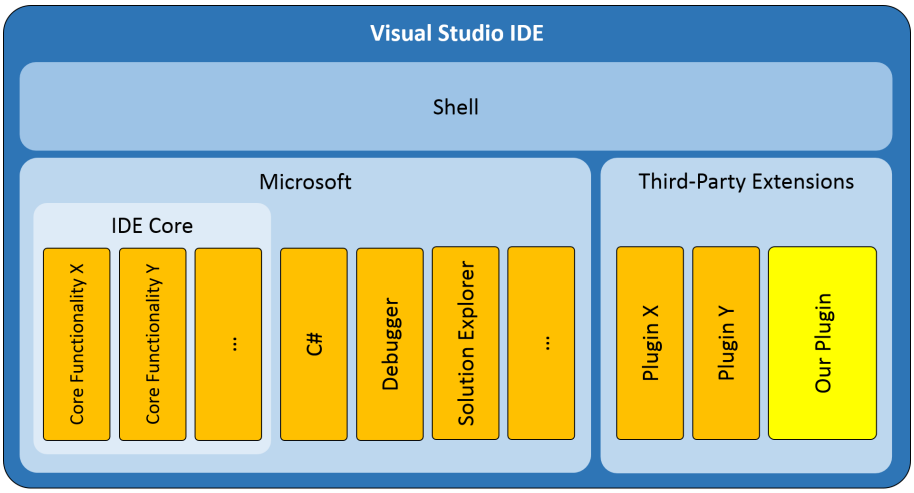


**Figure 4.7**: The package architecture of the *Visual Studio* IDE. A plug-in is, like all other core functionalities, a *Visual Studio Package* and hosted by the *Shell*.

## MVVM Pattern

As the main design concept of our software we applied the *Model View ViewModel* (MVVM) architectural pattern which is largely based on the well-known *Model View Controller* (MVC) pattern. The principles of MVVM originate from Microsoft and are often considered by developers when *Windows Presentation Foundation* (WPF) applications are built. This pattern allows to decouple the implementation of the logic and the user interface. While we described the layout of the extension's graphical user interface with the *Extensible Application Markup Language* (XAML), we implemented the underlying data model with C#. The connection between the view and the

[1] http://msdn.microsoft.com/en-us/library/dd460648.aspx

model is done by means of the *View Model* which serves as a controller. This means it is partly responsible for the view's logic and therefore manages and exposes the model data objects. The view communicates with its view model through so-called *Data Bindings* that allow a powerful and stable synchronisation between the two.

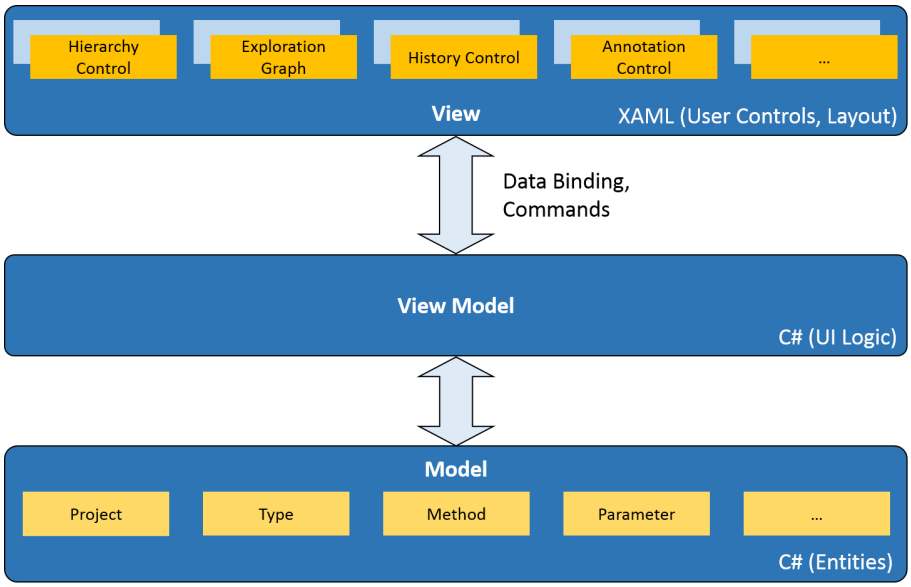


**Figure 4.8**: The *MVVM* architectural pattern. View and view model are synchronized by means of data binding.

Every piece of *Interactive Exploration's* user interface is separated and specified in a proper user control which consists of a layout defining mark-up file and a tightly coupled code file, as it is depicted in Figure 4.8. This so-called *Code-Behind File* includes some model unrelated view logic, like animations or other calculations. Furthermore, a single view model was defined in order to synchronize all those user controls that define the different aspects of the plug-in. This means, it makes the data objects of the underlying model available and ensures a concurrent update to all views via data binding. In the event of user interaction, this middle layer is again informed and executes the corresponding logic which may also include the modification of particular model data.

## Data Model

Since the fundamental idea of the extension is to explore code elements of the underlying development project as well as their relations, we decided to constitute our own model which should accurately represent the solution under inspection. This model should allow us to manipulate and modify its content among our needs. Going for such a flexible model has proven to be essential for our research, as it allowed an easy adoption in case changes had to be made or new requirements came up.

According to Visual Studio's nested entity pattern, we defined own model types and interfaces as depicted in Figure 4.9. We defined a *SolutionModel.cs* class which serves as the main model type and takes care of all the solution's projects. Furthermore, an overall abstract base class, called *HierarchyMember.cs*, has been defined which contains basic information about an element such as its

name or selection status. In addition, we equip every element with a unique *GUID* in order to be able to distinguish them. As a representative for any assembly of the solution, such as class libraries or executables, we introduced an inheriting *Project.cs* class. It is obvious that the mission of this entity is to encapsulate all types that are declared by the underlying solution's project. Of course, also the types have been given their respective counterpart in the data model. Among several informations about type-specific attributes, the most important job of this *Type.cs* class is to take care of any declared method within the type's body. We originally added equivalent classes in order to represent fields and properties, however, they are not used yet for the plug-ins functionality, but rather serve the completion of the derived model. On the opposite, a very central class was defined in order to incorporate methods. The *Method.cs* class takes care of its calling and called neighbours, as well as of its parameters and return type. Method input parameters obtained an own class called *Parameter.cs*. The characteristics of types, methods, fields and properties have in common that they are part of an assembly and each have got a parent (*e.g.*, a method's parent is a type). We therefore declared another intermediary abstract *ProjectMember.cs* class which inherits from *HierarchyMember.cs*, but additionally manages the parent object. The focused representatives of the solution's code elements, namely projects, types and methods, each contain an essential property called *ReflectionInfo*. This object contains a bunch of information about the particular entity and is used as source of extraction. We will discuss this topic later on and in more detail (see Section 4.2.1).

We modularized the whole implementation of our data model, which consists of the mentioned data types and other less important model items, and declared an accordant *InteractiveExploration.Model.dll* class library. This approach of an independent namespace allowed us in a later point in time not to make use of this model types only in our own plug-in project, but also in further important third-party projects, such as the *Graphviz4Net* application which will be discussed in section 4.3.2. The definition of an appropriate model namespace therefore avoids forbidden cross-references as soon as those data types are used in plug-in external content.

## 4.2 Challenges

During the development phase of the *Interactive Exploration* extension we encountered several obstacles that prevented us from meeting all our initial demands. In the following, we discuss our major drawbacks and how we dealt with them.

### 4.2.1 Prohibited Abstract Syntax Tree Access

In order to empower the *Interactive Exploration* extension to search for entry points and navigate through element dependencies, the underlying model has to be created first. We initially tried to make use of a possible official *API* provided by Microsoft, however, we encountered a rough disappointment as there seems to be no such comfortable interface to access the source code's syntax tree. Instead, we had to go for a more roundabout way. It's an established approach among Visual Studio *Add-In* and *Macro* developers to take advantage of the so-called *DTE Object* which represents the Visual Studio .NET IDE and is the top-most object in the automation model hierarchy. The package class creates a single instance of this service as can be seen in Listing 4.1. This application object encapsulates information about the currently loaded solution and its projects.

```
public static DTE2 AutomationObject
{
   get
   {
```

```
        return _automationObject ?? (_automationObject = GetGlobalService(typeof(
            DTE)) as DTE2);
    }
}
```

**Listing 4.1**: *Interactive Exploration* makes use of the Visual Studio automation service called *DTE Object*.

Our workaround makes use of this services as it first figures out the assembly paths of every open project on the computer's hard disk. Of course this procedure has a weak point since those files actually have to exist, what is not the case before the projects are compiled. This means that a project cannot be explored in case it contains a compiler error and therefore this particular assembly is not able to be built. Furthermore, the loader always expects the assemblies not to be signed and to reside in the default debug folder. As soon as the file paths are valid, the corresponding assemblies are loaded by means of reflection and the *SolutionModel* type (see Section 4.1.3) can be created by passing a list of the projects' reflection information to its constructor. The static *ModelBuilder.cs* class takes care of this setup process.

```
private static SolutionModel BuildModelReflectionBased(DTE2 applicationObject)
{
    if (applicationObject == null)
        return null;

    // load the assembly for each project
    var projectAssemblies = new List<Assembly>();
    foreach (EnvDTE.Project project in applicationObject.Solution.Projects)
    {
        if (project.FullName.Equals(String.Empty))
            continue;

        // create file path of the project
        var path = project.FullName.Substring(0, project.FullName.LastIndexOf(@"
            \\", StringComparison.Ordinal))
            + "\\bin\\Debug" + project.FullName.Substring(project.FullName.
                LastIndexOf(@"\\", StringComparison.Ordinal));

        // check all assembly types (i.e., .dll and .exe)
        foreach (var assemblyType in AssemblyTypes)
        {
            var assemblyPath = path;
            assemblyPath = assemblyPath.Replace("csproj", assemblyType);

            // get the assembly, if file exists
            var fileInfo = new FileInfo(assemblyPath);
            if (fileInfo.Exists)
            {
                var assembly = Assembly.LoadFile(assemblyPath);
                projectAssemblies.Add(assembly);
                Outputs.Add(String.Format("'{0}' assembly loaded.", assemblyPath));
                break;
            }
```

```
            Outputs.Add(String.Format("'{0}' assembly not exists.\nMaybe it has to
                be built first.", assemblyPath));
        }
    }
    return new SolutionModel(projectAssemblies);
}
```

**Listing 4.2**: The *ModelBuilder.cs* class is responsible for the reflection-based creation of the model by means of the *DTE* automation object.

The *SolutionModel*'s constructor reads out all the project's members in a gradual manner while constantly considering the services of the respective reflection objects. For each project, its declared types are readout and immediately injected in our hierarchy model. It is a simple task to finally extract every method from all type-representing reflection objects.

### 4.2.2 Call-Relation Readout

The aforementioned absence of an official *API* for the access of source code elements prevented us also from gaining information about existing method call-relations in an easy way. An intensive enquiry laid open that most Visual Studio package developers started to implement their own C# parsers in order to figure out the existences of those calls. A prominent example of this is *JetBrains' ReSharper*[2] plug-in which supports .NET programmers with a bunch of very useful developer tools. They started an own implementation of the codes abstract syntax tree, called *Program Structure Interface*[3] (PSI), whose main responsibility is the lexing and parsing of the code. Another package is the *Sando*[4] code search engine developed by a voluntary community around David Shepherd. The team was not able to make use of an official API too and hence the underlying services of this search dedicated plug-in are responsible for parsing, splitting and indexing code fragments.

As the implementation of an own parser would have blasted our time frame, we decided to implement a simple service which is able to fake method calls, such that we at least could test our plug-in. Unfortunately, this means that *Interactive Exploration* is not able to read out real call-relations of the solution's context at the moment. However, after all this limitation affects only call-relations. When type inheritance relations are concerned, we can derive those by means of the information provided by the type-representing reflection objects and a simple algorithm. The static *RelationBuilder.cs* class, whose main task is to establish relations of any kind, was extended by a functionality that randomly generates call-relations between methods of a project. This does not include assembly comprehensive calls, anyhow, it is enough to test and warrant the functionalities of our extension. The corresponding implementation details can be seen in Listing 4.3.

```
public static void GenerateFakesForProject(Project project, int
    percentOfRelationChance)
{
    if (project == null)
        return;

    // aggregate all methods of a project
    var projectMethods = new List<Method>();
```

[2] http://www.jetbrains.com/resharper/
[3] http://confluence.jetbrains.net/display/ReSharper/2.1+Architectural+Overview
[4] http://sando.codeplex.com/

```
    foreach (var typeContainer in project.Types)
    {
        projectMethods.AddRange(typeContainer.Methods);
    }
    // foreach method create a relation with a certain chance
    foreach (var possibleCaller in projectMethods)
    {
        foreach (var possibleCallee in projectMethods.Where(possibleCallee =>
            Random.Next(101) % (100/percentOfRelationChance) == 0))
        {
            Method.CreateCallRelation(possibleCaller, possibleCallee);
        }
    }
}
```

**Listing 4.3**: The *RelationFaker.cs* is able to fake call-relations such that *Interactive Exploration*'s features can be run and tested.

### 4.2.3 Meaningful Graph Layout Algorithm

Another major challenge was to find a suitable way to guarantee a well-structured layout of the *Exploration Graph* (see Section 4.1.1). Therefore, we looked for a possible existing framework able to deal with the layout calculation. Such a tool should be capable of drawing given nodes and edges in a handsome manner and furthermore should be totally free of charge. Concretely speaking, crossing edges should be avoided as far as possible and nodes should be distributed wisely and without overlaps. Unfortunately, the *Windows Presentation Foundation* (WPF) does not innately offer such a functionality and we figured out that the .NET developer community is indeed in desperate need of such a framework. Of course, several third-party libraries are available, but always come with expensive license costs. Nevertheless, we found a promising open source project called *Graph#*[5] which provides some graph layout algorithms and controls for WPF. It furthermore allows visualized vertices to be dragged by means of user input. Regrettably, this framework was not initially intended to run in the context of a Visual Studio extension and all efforts to port this framework's code failed, although we received support of Graph#'s author. Anyhow, this turned out to be a minor backlash since we also wanted our layout algorithm to be able to manage sub-graphs and edges that have been established between any graph elements. We were aware of the open source graph visualization tool called *Graphviz* and it's abilities to do exactly what we wanted. But again, no corresponding support for WPF was given. Further cumbersome searches luckily lead us to the *Graphviz4Net*[6] project which allowed us to build a graph according to our imagination. We will introduce this framework and how it lays out a graph's elements in more detail in Section 4.3.2.

In order to build up the *Exploration Graph*, we introduced an algorithm to take care about the management of its elements. The update of any change in the graph should take part in the lowest adaptation layer, namely the level where method representing vertices are managed. Thus, we implemented a function for appending a method to the graph (see Listing 4.4) and another for the deletion respectively. Those implementations are responsible for the correct actualization of method element vertices, their type sub-graph assignments and the production of any relation edges. Those two procedures are used in all possible scenarios. In case the user wants to add

[5] http://graphsharp.codeplex.com/
[6] http://graphviz4net.codeplex.com/

a whole project or type at once, all contained methods are consumed by those functions in a sequential manner. The view model (see Section 4.1.3) holds a central dictionary to ensure a quick and safe management of the visualized type sub-graphs.

```
private void AddMethodToGraph(Method method)
{
   if (InteractiveExplorationPackage.SettingsViewModel.IsDeclaredOnly && !
       method.IsDeclared)
      return;

   // check if method is already in graph
   if (GraphContainsMember(method))
      return;

   method.SelectionMode = HierarchyDefinitions.SelectionMode.Selected;

   // check whether subgraph already exists
   var type = method.Parent;
   if (_typeSubDic.ContainsKey(type.Id))
      _typeSubDic[type.Id].AddVertex(method);

   // else create new type subgraph
   else
   {
      var typeSubgraph = new SubGraph<Method> { Label = type.Name };
      _relationGraph.AddSubGraph(typeSubgraph);
      _typeSubDic.Add(type.Id, typeSubgraph);

      // add new selected method
      typeSubgraph.AddVertex(method);

      // add existing siblings as well (move from graph to subgraph)
      foreach (var sibling in TypeTopLevelMethods(type.Id).ToList())
      {
         _relationGraph.RemoveVertex(sibling);
         typeSubgraph.AddVertex(sibling);
      }

      // update the inheritance relations between types
      UpdateInheritanceRelations(type as Type, typeSubgraph);
   }

   // update the call relations between drawn methods
   UpdateCallRelations(method);
}
```

**Listing 4.4**: The *ExplorationToolViewModel.cs* manages the addition and deletion of method elements. Sub-graphs are managed by means of a dictionary.

From Listing 4.4 can be derived that the updated graph is investigated for any possibly new

introduced call-relations that should be visualized. An equivalent method takes care of inheritance relations in case a new sub-graph was added to the graph.

### 4.2.4 Aid to Orientation

Since the structuring of the graph is evaluated in an external tool, whose functionalities we will reveal in Section 4.3.1, we don't have any influence of how this very algorithm will lay out the elements. The disadvantage here is that, in case nodes or sub-graphs are added, we cannot guarantee for no element that it will be redrawn at the same position. Unfortunately, this queer behaviour does not help the user to stay oriented in the graph.

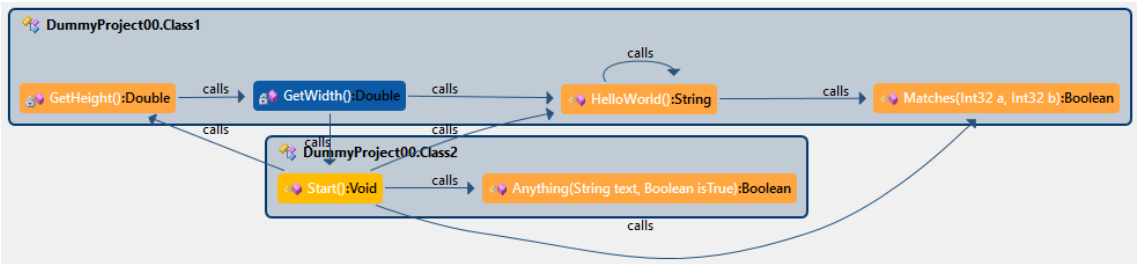


**Figure 4.10**: The highlighting mechanism of the *Exploration Graph*. Depicted are newly added outgoing calls of the *Start()* method.

Instead of leaving the user to his own resources, we defined a couple of highlight mechanisms which should serve a faster localization of the elements under focus. After a modification was made to the graph, it is thus not necessary to laboriously search for the vertices of interest, but rather they are dyed with a flashy colour. As pictured in Figure 4.10, newly added code hierarchy members obtain a orange background, whereas the source of interaction (*e.g.*, the method that ordered all callees to be shown) receives a dark yellow.

### 4.2.5 Gesture Recognition

We wanted to support both mouse and touch interaction on the *Exploration Graph* at the same time. However, this turned out to be more complicated than originally assumed. The main reason for this is the stacking of independent user interface elements that should not interfere with each other as input is received and interpreted. This concerns the graph control, its underlying zoom panel as well as navigation and context menus of the vertices. Since WPF interprets untreated touch input automatically as mouse events, we were not able to support both interaction mechanisms simultaneously on every single control. Fortunately, we could partly put some things right by making use of the *Surface 2.0 SDK*[7], which provides a couple of default controls that deal with this tricky input event handling. Anyway, we were forced to leave the decision to the user, whether he wants to make use of mouse or touch interactions and provided a corresponding settings option.

Since WPF already provides a bunch of handy information about any touch interactions on the screen, it was no big deal to interpret gestures, such as panning or pinching, and couple them with the respective commands. On the other side, this does not hold for the capture of mouse

[7] http://www.microsoft.com/en-us/download/details.aspx?id=26716

movements. Here, we developed an own gesture tracker which that attention to the cursor and interprets gestures where needed.

## 4.3 External Technologies and Frameworks

The *Interactive Exploration* plug-in makes use of some third-party open source libraries and frameworks which will be presented in the following. In order to reconcile the external functionality with our proper code, several likewise illustrated issues had to be dealt with.

### 4.3.1 Graphviz and Dot

As we mentioned before, the layout of our well-structured *Exploration Graph* is calculated by an external tool. We assigned this task to *Graphviz*[8], an open source and platform independent software package originally developed by *AT&T* and the *Bell-Labs*. Its core functionalities are the computation of node positions, such that the overall graph is laid out in a clear manner. This includes the bending of edges, the minimization of connection lengths and the avoidance of crossing lines. The required information about the graph elements' dimensions are stored in a text file which is then consumed by Graphviz in order to optimize the layout.

Graphviz supports some different input text formats to support various kinds of layout algorithms. Anyhow, we made use of the so-called *DOT* format whose main capability is to offer support for hierarchical graph structures. It optionally lays out all edges in either a horizontal or vertical manner. The mentioned input text file is composed with the *DOT Language* which is basically a powerful syntax to describe graph requirements. It provides plenty of keywords and attributes in order to describe the graph as precise as possible.

The *Interactive Exploration* extension makes use of the *dot.exe* command line tool which is able to generate and process this format. Anyhow, it is inevitable to have an entity between the Graphviz command line tool and WPF, which finally creates and renders the plug-ins user interface. That is why we took advantage of the forthwith discussed Graphviz4Net approach.

### 4.3.2 Graphviz4Net

The *Graphviz4Net* project is an open source solution, originally developed by Štěpán Šindelář and further adopted by *CodePlex* community members. It provides a couple of .NET APIs for the generation of *dot* inputs and is also able to consume Graphviz' output. Furthermore, it brings a graph layout user control that can easily be integrated by WPF applications. By contrast with other tools, it allows also the definition of sub-graphs and diverse kinds of links. In addition, it permits to style every single part of the graph. We hence could design the appearance of nodes, edges and sub-graphs according to our imagination.

In order to have textual input for the external command line tool, the whole graph has to be defined by means of C# data structures. Once this is done, Graphviz4Net generates the corresponding DOT syntax. The encoder asks WPF to measure the dimensions of the styled elements' visual representatives, such that a precise arrangement can be calculated. The output text file, returned by *dot.exe*, contains the computed positions of nodes and sub-graphs. Besides, it contains coordinates for the edges such that WPF can render the curved lines through those points. A concrete example of how the syntax looks like is demonstrated in Appendix B.

[8] http://www.graphviz.org/

### 4.3.3 Problems & Modifications

Although we were glad to make use of the Graphviz4Net project, it did not serve all our needs right from the start. Several modifications were simple, such as the adaptation and customization to our model or the definition of own view models. On the other side, we encountered other difficulties that were not as easy to deal with.

#### Execution Locking Mechanism

The interface between our graph modification algorithms and the coupled events of Graphviz4Net did not play well in the beginning. The nature of the *Exploration Graph* allows the user to insert or delete multiple elements at the same time. Stupidly, for every single change that was made to the graph by our management algorithms, an immediate execution of the externalized layout calculation followed. That is why the *dot.exe* tool was not able to perform all its obligations in reasonable time at all and led the whole framework to drift in an unstable state. Instead of a well-organized graph description, an error was returned.

For this reason, we established a locking mechanism which prevents the Graphviz4Net framework from handling every single update. Our graph management algorithms enable this lock as soon as modifications are announced. Once, the graph's data structure is successfully adjusted, the lock is released and the layout computation is finally forced to execute all changes at the same time.

#### Scalability

Although we could find a workaround for the aforementioned issue, this did not prevent the Graphviz4Net project to throw further random exceptions, complaining about invalid DOT output or missing key values, from time to time. Even though we tried to figure out what is wrong with its code, we did not yet discover the source of the problem. At least, such exceptions are rare and we therefore assume that it highly depends on the amount of modifications made to the graph in a single step. Anyhow, Graphviz4Net turned out to not be well scalable and thus suffers from internal errors, whose occurrences are due to a bad support for larger graphs.

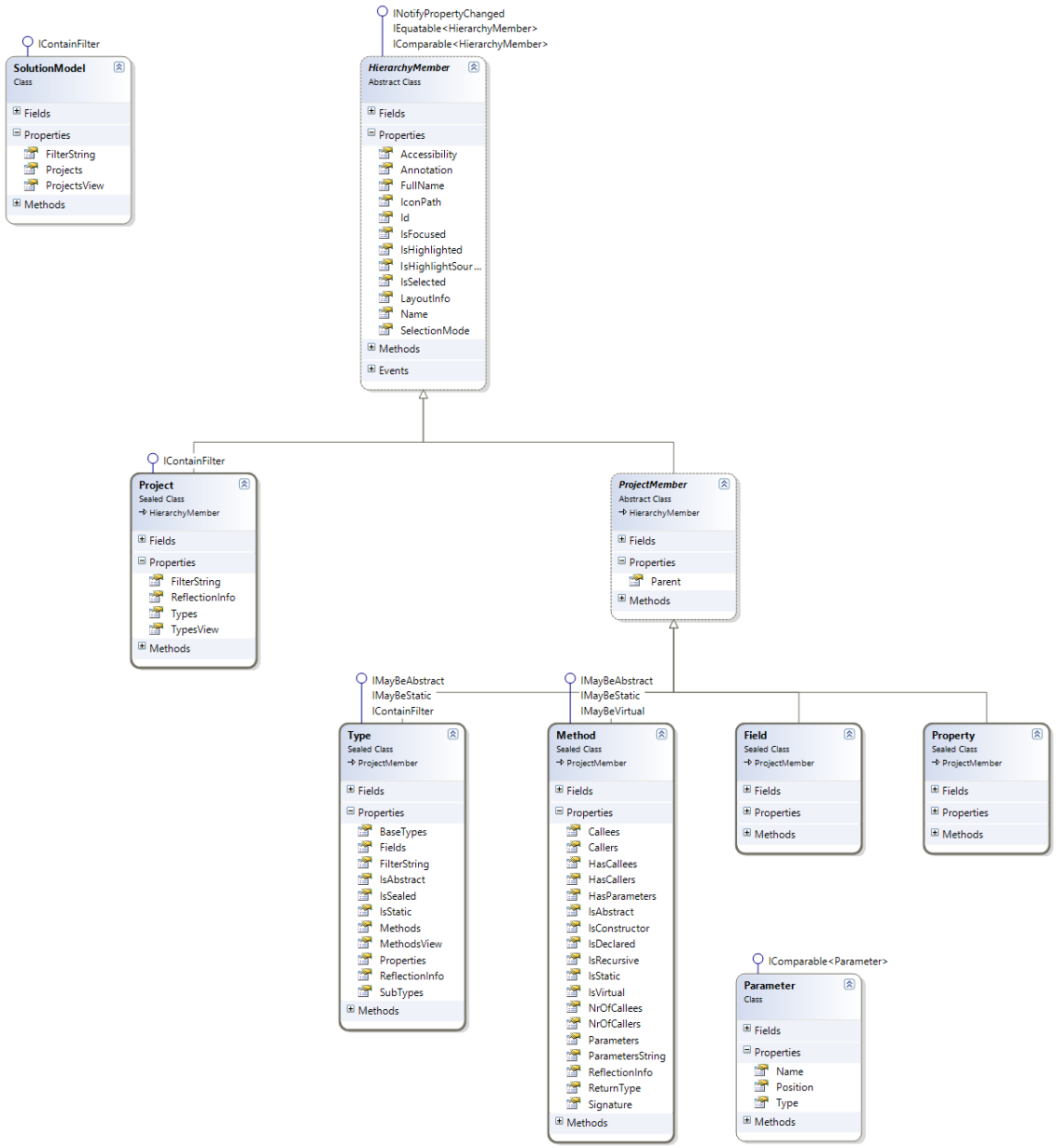


**Figure 4.9**: The data model of the *Interactive Exploration* extension. The abstract class *HierarchyMember.cs* serves as overall base class. *SolutionModel.cs* represents the analysed solution and encapsulates its projects.

# Chapter 5

# Evaluation

The first phase of our implementation process included the development of the basic functionalities of the extension. The result was an almost fully featured prototype (see Appendix C). However, we did not spend much effort with the visual representation. Since our *Interactive Exploration* plug-in should meet the preliminary stated promises and design decisions, we conducted a usability study in order to gain advice from several Visual Studio developers. According to their feedback, we focused specifically on the criticized points which requested further improvement of several parts of the user interface. Based on this, we finally defined a whole new look. Hence, the objective of our evaluation is twofold. Contiguous to the illustration of the mentioned study, we check whether we achieved our own goals.

## 5.1 Usability Study

In order to evaluate the usability of our prototype plug-in, we checked several aspects of the user interface. For this purpose, we analysed all of the ten *Usability Heuristics* of Jakob Nielsen [NM90] that can be used to explain a very large proportion of the problems one observes in user interface designs. They are not specific guidelines but rather rough rules of thumb and therefore allow to have only a small set of evaluators to examine and judge the interface. Generally, every examiner finds a different set of usability problems and more experienced designers tend to find even more issues. Therefore, it is possible to improve the effectiveness of this method significantly by involving multiple evaluators.

This section shows how we conducted the evaluation and what we found. Besides explaining briefly what the investigated heuristics' principles are targeted on, we illustrate how good our prototype performed when they were applied. We finally conclude our findings and show which parts of the user interface were accordingly adapted shortly after.

### 5.1.1 Methodology

In order to study our tool, we employed a 65 inch wall screen (see Appendix A) as touch interface and run the application on it. We considered the opinion of four developers as Nielsen suggests three to five evaluators. One usually does not find much additional issues when asking more participants [NM90]. The evaluators went through the interface two times. The first time was meant to gain a basic idea of what views exist and how they interact with each other. A general understanding and an insight how everything fits together was obtained by this step. In the second pass, they had to focus on specific interface elements in order to compare them to the ten heuristics. An evaluation form, containing the principles, was used to keep track of the findings.

The evaluators were not allowed to just say that they did not like a certain feature or design. A concrete critique with reference to one of the heuristics had to be stated in order to be valid. Additionally, we allowed the testers to bring up further principles that came to mind, even if they found no representative heuristics. We also declared that positive feedback would be appreciated and could of course be mentioned. We provided no special assistance or introduction about how the tool should be used since the evaluators were all domain experts, meaning that they had fundamental practical experience with Visual Studio. We assured that in case the evaluators had questions, we would assist with accurate explanations. However, we did not foreclose questions such that we could first observe how they interacted with the content.

### 5.1.2 Heuristics and Findings

This section presents the ten heuristics and provides the according results from our user study. Also mentioned are concrete modifications we made due to specific criticism or lack of intuitive design.

#### Visibility of System Status

This principle suggests that the system should provide timely and accurate feedback about it's status such that users are always kept informed about what is going on.

The evaluators agreed about the needlessness of a particular progress bar for the hierarchy overview since the underlying model was always built up very quickly such that the solution's hierarchy tree could be immediately shown. In dependence of how many of the entities were selected to be shown in the graph and the graph's current dimension, the algorithm calculating the new layout not always performed equally fast. Therefore a text replaces the graph in calculation and indicates the running computations. This feature was appreciated by the users. However, the resulting layout of nodes and edges often introduced a new placement for this entities and lead to confusion as they suddenly vanished from the evaluators focuses. Consequently, those repositioned nodes had to be tediously located before the exploration process could be continued. We therefore introduced the in Section 4.2.4 demonstrated concept that highlights the most recently added elements in order to better distinguish them from the rest and to give the user a prompt feedback. The toggle buttons residing in the tool bars, which provide different options of what informations should be visible, were denounced to not clearly represent whether they are checked or not. The testers always first clicked those buttons in order to investigate their current status. We solved this issue by means of a more intuitive control.

#### Match Between System and the Real World

Another proposition for better comprehensibility of the interface design is to use terms and concepts that are familiar to the user. The system should, rather than speaking with system-oriented terms, follow real-world conventions and metaphors in order to display information in a natural and logical manner.

Representing the *Exploration Graph* in a left-to-right manner has proved to be a good way to emphasize the flow of method calls. This mainly left-right navigation was said to be supportive in the assembly of a personal mental model. Furthermore, one evaluator complimented the option to change this main flow direction, although he saw no need for the bottom-up and right-left options. Almost all evaluators had difficulty to interpret the little coloured square to the left of each hierarchy tree element and it required some time for them to notice that they actually

represent the respective current selection states. An issue associated to this was the fact, that the navigation menus of methods within the graph made use of different symbols in order to denote the corresponding entities' current selection states which indicated whether they were shown by the graph or not. In those menus, a cross icon was shown to notify the user that an interaction would result in the removal of that method, whereas the absence of that cross would have caused the opposite. The solution of this was to use the squares in all kinds of menus and to provide a legend for the meanings of each colour.

### User Control and Freedom

While working with tools, users often unintentionally navigate to a wrong state although they actually wanted to do something else. Therefore, it is a good idea to provide functionality to return to previous states. In general, users should always perceive control about what happens as they interact with the tool.

Once the principles of how entities can be selected and shown in the graph were understood by the testers, they found it easy to insert and remove items. Also the functionality, to do so directly from within the graph by means of the provided navigation menus, was highly appreciated. However, they consistently missed an overall history of actions they have performed so far and an associated functionality that would allow to undo certain actions (*e.g.*, hiding unintentionally added callees of a method). For this reason, we implemented the in Section 4.1.1 presented history.

### Consistency and Standards

Wherever possible, all user interfaces should be consistent such that users do not have to deal with multiple graphical representations or actions that actually mean the same thing. This concerns icons, user controls and terminologies. Sometimes it is appropriate to use a certain industry or platform convention to which the audience is used to.

The assignment of an element-specific colour to both the hierarchy tree items and the graph nodes convinced the evaluators. The usage of popular icons, which are used by Visual Studio's *IntelliSense* (*i.e.*, Microsoft's auto-completion implementation) and several third-party extensions, that are consistently used throughout all views and menus, lead to the same result. Furthermore, the styling of toggle and refresh buttons had always the same appearance. As the users were acquainted with the *UML* modelling standard, they instantly recognized the accordingly styled edges and directly assigned the correct meaning to them. On the opposite, they again criticized the inconsistency of the representation of an elements selection state that differed in the hierarchy and the navigation menus of the graph. A major critic was that the zoom functionality did not work well with the pinch gesture, although it was intended to work. An evaluator advised that nowadays pretty much every multi-touch based user interface allows their users to zoom with this very well-known pinch gesture. On account of this we replaced the prototype's zoom control with a totally new one. The reimplementation of this functionality finally led to the zoom based features stated in Section 4.1.1.

### Error Prevention

The best method to prevent users from committing errors is to carefully design the user interface in such a manner that it clearly communicates the consequences of a possible interaction. This

stops the occurrence of problems to the greatest possible extend.

A hint, which was worth to be respected, showed that the ability to open the hierarchy and the exploration control independently from each other is nonsense since they are tightly coupled and pretty much useless without each other. Based on this feedback, we joint and nested the different views of our plug-in in such a way, that some views can easily be hidden when not needed any more (see Section 4.1.1). Concerning the tool bars, we received the suggestion to ask the developers, in case they clicked any refresh buttons, whether they really want to proceed rather than directly initializing all computations. One evaluator recommended not to give the ability to show and search just for methods that are defined within their embracing types, but rather to completely remove this feature as it only led to confusion. We followed his suggestion. Again, the lack of an overall history was mentioned. The testers all missed a button with which they could go back and undo the least recent action in order they encountered an error.

### Recognition Rather Than Recall

Users should be provided with familiar icons, actions and options in order not to force them to recall interaction possibilities from another view and to reduce their memory load. Instructions for functionalities should be easily and early retrievable (*e.g.*, with the aid of tool tips).

With respect to this principle, we received good feedback for the general layout of the interface elements since we arranged them in the same manner as original Visual Studio tool windows are laid out. Once more the consequent usage of same colours, styles and icons - except those in the graph node's navigation menus - for the same types of information was praised as it helped the evaluators to interpret the content. However, assuming the user has absolutely no idea what the extension is good for, it seemed he would have forced to try just randomly any buttons or gestures in order to discover functionality and features. This was due to the fact that absolutely no guideline has been given. We therefore introduced a tooltip for every single view of the extension such that the most important guidelines can easily be accessed. Another statement indicated that a possible user might quickly loose the orientation within the graph's constructs right after a rather big modification was applied. The consequent recommendation was to provide visual information about where modifications were made to the graph. We implemented this very idea which resulted the highlighting mechanisms.

### Flexibility and Efficiency of Use

This principle recommends to give expert users so-called *accelerators* in order to speed up the interaction. With this, at least the most frequent tasks should become easy and efficient. Less experienced users however should not have to care about those fast access options and should be able to use the tool the normal way without any difficulty.

Having the ability to directly resize the graph according to the actual dimensions of the view or to zoom to its original size was highly appreciated and said to be very useful. On the opposite, the employment of the slider control causing the view to scale accordingly was cumbersome. Based on this, we replaced the slider control with a more precise and touch friendly one. Furthermore, more sorting options for all kinds of lists throughout the interface were wished. The navigation menus which allow a preview of calling or called methods should for example be sortable alphabetically, among their access level or position of declaration. Unfortunately, we were not able to come up with such filters by the time. Anyhow, the most missed feature was the absence of an option to directly access the underlying source code as this would help the general understanding

and importance of a certain element. Concerning this lack of functionality, we already made an accordant statement in Section 4.1.2.

### Aesthetic and Minimalist Design

This heuristics argues for a minimalistic design of the user interface since irrelevant or uncommonly needed information just wastes space and competes with the relevant or interesting part of the visualization. Concerning the aesthetic aspect of the visual layout, contrast, repetition as well as the alignment and proximity of graphical elements should be chosen with caution.

The toggle buttons in the tool bars showed to come up to their intended functionality, namely leaving the decision, whether namespaces of classes or return types of methods should be visible, to the user. The intention to compress information about the code elements' attributes into a single icon promised to be successive as well. It has been stated that the growing dimensions of the graph could become problematic when orientation aspects are concerned. The majority of the testers therefore suggested to introduce some few levels of abstraction where information could be further packed on higher layers. This should allow to regain clarity, however we were not able to come up with an acceptable solution so far (see Section 4.1.2).

### Help Users Recognize, Diagnose and Recover from Errors

In case errors are occurring, those should not be displayed as cryptic system codes, but should rather indicate the underlying problem in plain language. Furthermore, the user experience can be greatly improved with the delivery of constructive suggestions about how the encountered error could be recovered.

Our plug-in was not supportive at all with respect to error recovery. Although information about the occurrence of a possible error has always been given in a separate channel of Visual Studio's output window, hints about how to recover from that error were rarely provided. However, we moved this channel from the official output to a corresponding plug-in intern view and added more accurate messages. In addition, the existence of an overall interaction history now at least allows the user to reconstruct the path of failure.

### Help and Documentation

Even though the best way would be to create such an intuitive user interface that no help is required at all, there always might be cases where it is essential to access further information about the usability. For this reason, a comprehensible documentation and user support should be easy to find.

A lively discussion, about whether a help should be integrated or not, took part between the evaluators and resulted in the general accommodation that our plug-in does not necessarily need a separate help documentation. In fact, the usage of the aforementioned tooltips which are directly integrated in the corresponding view serve as local guideline right away.

## 5.2 Realisation

By dint of this section, we dissect our plug-in in order to clarify whether we met our own expectations, requirements and goals that were declared in Section 3.2.

The main downsides of our finalized extension are the inability to catch the underlying code elements by means of an official API. Instead, a cumbersome workaround (*i.e.*, reflection) is used to get the workspace context, as explained in Section 4.2.1. It is also a pity that call-relations cannot be identified at all, as reasoned in Section 4.2.2, although the plug-in should serve as supportive dependency navigator. However, in order to test the requirements of our software, we took advantage of a mockup project and faked some method calls, such that the extension's functionality could be proven.

Even though we had to deal with those hindering issues, we successfully respected and implemented our requirements. Our exploration tool runs as a single-windowed Visual Studio plug-in and promises to make use of all available screen space. Temporarily dispensable parts of the user interface can easily be hidden and recovered as soon as they are needed. In the event of using a rather large touch screen, like the *PQLabs G3 Monitor* discussed in Appendix A, even multiple developers are able to review and examine code elements and dependencies concurrently. While the display is multi-touch enabled and the tracking of diverse exploration paths is supported, we do not yet recommend a simultaneous user interaction, since the graph's limitations would hamper this experience, and instead suggest to have a single operator taking care of the navigation.

Furthermore, no complicated interaction chains have to be followed in order to get information about a specific code fragment. All worth knowing facts about an element are directly visible. Besides information, like protection level or other element-specific attributes, that is summarized by a little icon, each vertex contains further icons in case it is annotated. The tool is able to answer questions about call or type hierarchies, references or the place of declaration.

Due to the layout calculation algorithms provided by Graphviz, the *Exploration Graph* always profits from the best possible structure. Even sub-graphing and clustering of method nodes are supported, such that the requirement of a higher-order structure is maintained. A user is able to switch the current layout direction in order to modify the graph in such a manner that it allows a better understanding of the represented connections.

*Interactive Exploration* allows a developer to search for a suitable starting point by providing a hierarchical overview of the workspace at hand. In case a specific anchor point is already known, it can easily be accessed by means of the provided search box. The graph supports all basic navigation operations as far as methods are concerned. The nodes' navigation menus provide early information about existing paths and allow the concurrent composition of multiple exploration paths. However, travelling along type-representing sub-graphs is not yet supported and at the mercy of future work.

We provide several different ways to change the zoom level in order to gain a better overview or to focus on just specific parts of the interconnected elements. Any modification to the zoom factor is fairly animated, such that the chance to get disoriented is reduced to a minimum. By means of panning, the current viewport can easily be repositioned in order to follow dependencies or to relocate the focus. A good understanding of the overall solution's structure is supported by the hierarchy view and the respective graph representation which groups methods of the same type. The colours of the selection states allow code fragments under inspection to be clearly distinguishable from unselected ones.

In order to interact with the plug-in, both mouse and touch input is, even though not at the same time, supported. Although primarily designed for touch-based commands, the user interface optionally understands congruent mouse movements. As with established touch interfaces, *Interactive Exploration* is aware of single or double tapping, swiping or pinching and spreading gestures. A disadvantage of gesture driven interaction is the tedious entering of text in the method filter or the annotation tab. After all, the Windows integrated screen keyboard could be considered.

# Chapter 6

# Conclusions

## 6.1 Conclusions

With this thesis we successfully implemented a *Microsoft Visual Studio 2010* extension which encapsulates a bunch of features that allow C# developers to explore the current solution's code base. In the centre of attention is the visualization of code elements and their relations, whose existences can only be perceived by an implied manner in case prevalent editors are concerned. Both, the search or definition of an appropriate element set, that should serve as the exploration's entry point, as well as the subsequent navigation along dependencies, are supported by *Interactive Exploration*. Another benefit is the employment of a touch enabled *Natural User Interface* that allows users to command the plug-in by application of habitual, well-known finger and hand movements.

The accomplishment of our evaluation revealed that the requirements and goals we had regarding to the development of a novel software exploration tool could largely be satisfied. The user interface, the interaction mechanisms and the navigation cycle have shown to be facile and intuitive. However, our software lacks in the configuration of call-relations that are contained in the analysed context. This, as well as the unsupported fast source code access, is due to the unfortunate absence of suitable APIs, whose provision we assumed to be guaranteed by an official authority.

## 6.2 Summary of Contributions

The sheer fact that virtually no extension for Visual Studio 2010 assisting the visualization of code fragments or dependencies exists, makes our software inimitable. Although the IDE has some integrated diagramming capabilities, they do not provide the navigation along relations or the ability to leave an annotation, but rather serve the purpose of documentation. Apart from the mentioned multi-touch supported interaction, we mainly separated our tool from similar implementations on other platforms by allowance of frequently occurrent weak points or handicaps. Our approach is able to deal with an ever-growing graph by means of optimal assignment of screen space and suitable options to relocate the viewport. This is intended to assist in collaborative exploration scenarios, where multiple developers discuss, review or annotate the analysed source code. We adhered to display just a decent number of the most necessary information in order to keep the view clear. A major difference of our approach is the well-structured and meaningful arrangement of graph elements. We ensure the layout to generate additional value since always the best possible structure is calculated in order to facilitate interpretations.

## 6.3 Future Research

Although we afford a versatile extension, our work opens the door for several further possible researches and implementations. This section is dedicated to a listing of our personal suggestions, where future effort could be applied.

The most important thing to realize would be an authentic setup of the model. A possible approach is to implement both a source code parser and an indexer in order to keep track of a program's entities. Possibly, this could already be achieved by making advantage of the open source code of the *Sando*[1] search engine extension. This would, in addition, allow to provide access to the entity-related code statements.

In order to deal with the lack of call-relations, yet another third-party tool could be employed. We are aware of *SrcML.NET*[2], which is a C# framework for the analysis of source code. It is the .NET counterpart of *srcML*[3], whose main functionality is to combine code and abstract syntax tree information by means of an XML document providing, among other things, full access at the structural and syntactical levels.

Providing various levels of abstraction is essential for the exploration of large and complex software projects. This is another issue that could not be dealt with so far because of the problematic calculation of multiple graph layouts (see Section 4.1.2 for more information). Either the computation can be parallelised or a new approach should be found.

Making the *Exporation Graph*'s visual representation more dynamic and reactive to interaction input would further increase the user experience. However, this is of less importance. We already identified and marked possible entry points in the Graphviz4Net source code in order to add the ability to drag or resize vertices.

Future additions of new types of entities or relations with only a few source code changes is possible as well, since Graphviz4Net allows programmers to define new kinds of vertices or edges in a relatively homely way. Therewith, further dependencies could be introduced, once the underlying model is given.

Another major upgrade to our plug-in would be the addition of evolutional information about the code under inspection. A possible way to achieve such a target, is the tapping of an attached version controlling system, such as the *Team Foundation Server*[4].

[1] http://sando.codeplex.com/
[2] https://github.com/vinayaugustine/SrcML.NET
[3] http://www.sdml.info/projects/srcml/
[4] http://www.microsoft.com/visualstudio/eng/products/visual-studio-team-foundation-server-2012

# Appendix A

# PQLabs G3 Monitor

The *Interactive Exploration* Visual Studio extension is intended to be used together with large wall displays, such that advantage of the additional space can be taken. For this reason, we employed the *PQLabs G3 Monitor*, whose multi-touch overlay is a plug- and play solution for both programmers and consumers. This interactive wall can be brought to service by simply connecting the reactive overlay's USB 2.0 cable with a computer of choice. Touch gestures are recognized by the screen and forwarded to the *PQLabs MultiTouch Platform and Screen Driver*, which has to be installed on the running machine. A VGA interface serves the transmission of the image. Some further specifications[1] are hereby listed:

- **Size:** 65"
- **Resolution:** Full HD (1920 x 1080)
- **Touch Points:** Up to 32 points simultaneously
- **Touch Technology:** PQLabs LED Cell Imaging
- **Touch Method:** Finger, stylus or gloved hand
- **Accuracy:** $\pm$ 1.5mm
- **Response Time:** 7 - 12ms
- **Interface:** USB 2.0 (full speed), HID compliant, plug-and-play compatible
- **OS:** Windows, Mac OS X

[1] http://mymultitouch.de/

# Appendix B

# The DOT Language

We want to briefly demonstrate, how the DOT language works exactly. For this reason, we show input and output files of Graphviz' *dot.exe* command line tool. The following shows how the structure of the graph depicted in Figure B.1 is processed. To calculate the positions of those subgraphs, nodes and their directed edges, the input shown in Listing B.1 has to be passed to the command line tool. The corresponding output is presented in Listing B.2. This information is subsequently passed to the WPF rendering engine, which is now able to lay out the graph in a meaningful manner.

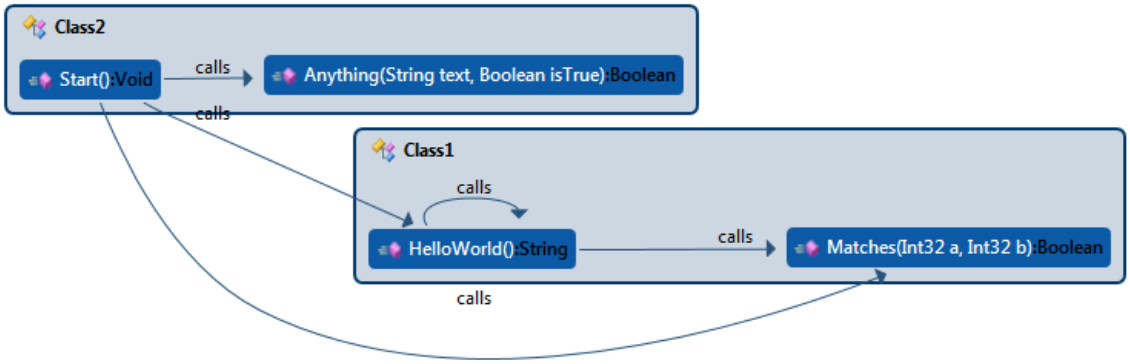


**Figure B.1**: A simple example graph, whose structure was computed by *dot.exe*.

```
digraph g {
graph [rankdir="LR" ,compound="true" ];
   subgraph cluster0 {
      graph [label="Class1" ];
      1 [ width="2.9861111111111" ,height="0.416666666666667" ,shape="rect" ,
          fixedsize="true" ];
      2 [ width="1.93055555555556" ,height="0.416666666666667" ,shape="rect" ,
          fixedsize="true" ];
   };
   subgraph cluster3 {
      graph [label="Class2" ];
      4 [ width="3.84722222222222" ,height="0.416666666666667" ,shape="rect" ,
```

```
          fixedsize="true" ];
      5 [ width="1.33333333333333" ,height="0.416666666666667" ,shape="rect" ,
          fixedsize="true" ];
   };
   5 -> 4 [ label="calls" ,comment="6" ];
   5 -> 1 [ label="calls" ,comment="7" ];
   2 -> 1 [ label="calls" ,comment="8" ];
   5 -> 2 [ label="calls" ,comment="9" ];
   2 -> 2 [ label="calls" ,comment="10" ];
}
```

**Listing B.1**: The DOT language content describing the graph in Figure B.1

```
digraph g {
  graph [rankdir=LR, compound=true];
  node [label="\N"];
  graph [bb="0,0,746,238.2"];
  subgraph cluster0 {
    graph [label=Class1,
      bb="235,48.201,738,150.2"];
    1 [width="2.9861", height="0.41667", shape=rect, fixedsize=true, pos="
        622,72.201"];
    2 [width="1.9306", height="0.41667", shape=rect, fixedsize=true, pos="
        313,71.201"];
    2 -> 1 [label=calls, comment=8, pos="e,514.41,71.854 382.51,71.424
        418.35,71.541 463.21,71.687 504.32,71.821", lp="483,79.701"];
    2 -> 2 [label=calls, comment=10, pos="e,342.88,86.476 283.12,86.476
        276.48,95.607 286.44,104.2 313,104.2 328.77,104.2 338.69,101.17
        342.76,96.799", lp="313,111.7"];
  }
  subgraph cluster3 {
    graph [label=Class2,
      bb="8,158.2,460,230.2"];
    4 [width="3.8472", height="0.41667", shape=rect, fixedsize=true, pos="
        313,184.2"];
    5 [width="1.3333", height="0.41667", shape=rect, fixedsize=true, pos="
        64,181.2"];
    5 -> 4 [label=calls, comment=6, pos="e,174.29,182.53 112.04,181.77
        127.4,181.96 145.37,182.18 164.12,182.4", lp="143,189.7"];
  }
  5 -> 1 [label=calls, comment=7, pos="e,586.14,57.173 70.535,166.14
      83.153,134.35 117.41,60.919 174,32.201 307.67,-35.631 492.87,21.078
      576.67,53.451", lp="313,39.701"];
  5 -> 2 [label=calls, comment=9, pos="e,278.19,86.26 98.826,166.14
      143.05,146.44 220.54,111.93 268.92,90.387", lp="143,159.7"];
}
```

**Listing B.2**: The DOT language content describing the corresponding output of *dot.exe*.

# Appendix C

# Visual Appearance of the Prototype

This appendix provides an insight to the main visual differences of the prototype version of *Interactive Exploration* (Figure C.1) and the final release, as presented in Chapter 4 (see Figure 4.1).

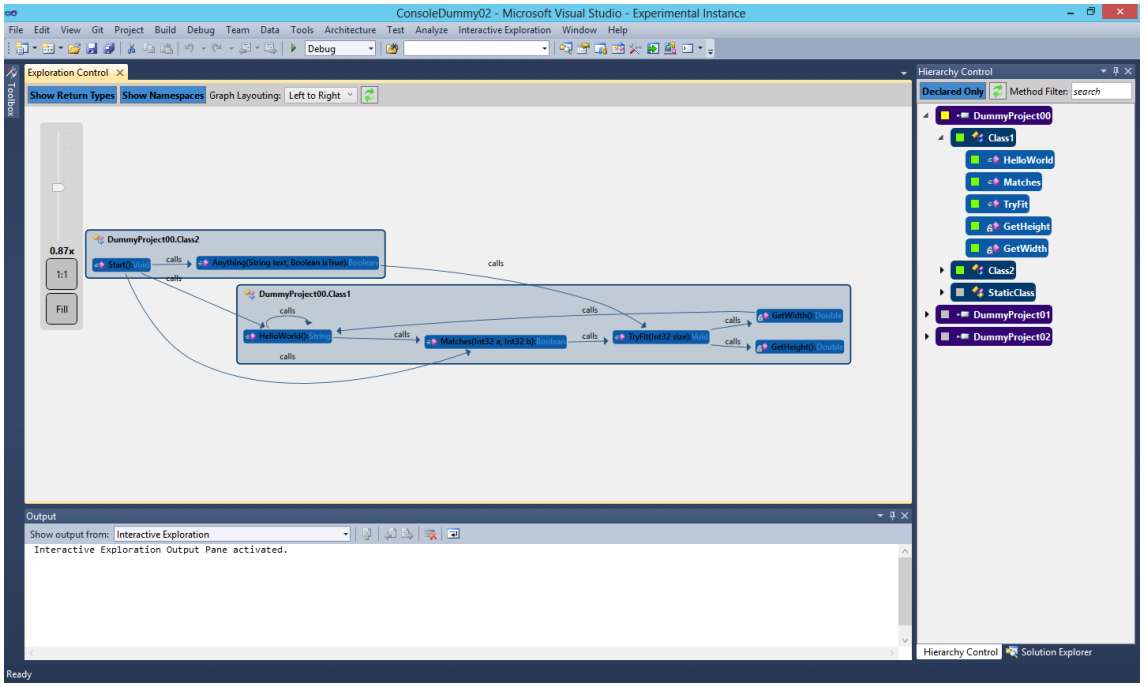


**Figure C.1**: The user interface of the plug-in's prototype.

The hierarchical overview and the exploration area each made use of a particular tool window. Besides, the extension initially received a separate channel in the IDE's output view. However, the different windows were joined in the final version and a history control, as well as an annotation panel, were added. Furthermore, the whole zoom control and several interaction options were changed in order to provide a better user experience. More detailed modifications are listed in Section 5.1.2.

# Appendix D

# Used Tools and Frameworks

- **Microsoft Visual Studio 2010 Ultimate - http://www.microsoft.com/visualstudio/**
  Microsoft's multi-language development environment. Used in this thesis for developing, debugging and testing the software. In addition, the *Experimental Hive* instance of this environment was used to run and test the *Interactive Exploration* plug-in.

- **Microsoft Expression Blend 4 - http://www.microsoft.com/expression**
  Microsoft's user interface tool for creating graphical interfaces for web and desktop applications. Used for the front-end design of the *XAML*-based *Windows Presentation Foundation* plug-in interface.

- **Visual Studio 2010 SDK**
  This SDK provides tools and templates for building Visual Studio extensions. Used in order to build tool windows and to create menu commands.

- **Microsoft Surface 2.0 SDK**
  Microsoft's Surface development kit for improved support for Windows touch-enabled devices. The developed extension makes use of several provided controls in order to make them reactive to both mouse and touch input.

- **Graphviz 2.28.0 - http://www.graphviz.org/**
  An open source graph visualization software. Provides several main graph layout algorithms. Used in this thesis to calculate the structure and layout of graphs and sub-graphs.

- **PQLabs MultiTouch Platform and Screen Driver - http://www.pqlabs.com**
  PQLab's driver for Windows to handle multi-touch input. Used in this thesis for the capturing of multi-touch gestures on the *PQ Labs G3 Monitor*.

- **C# 4.0**
  Modern, multi-paradigm, general-purpose and object-oriented programming language. Used for the implementation of the *Visual Studio* plug-in.

- **Windows Presentation Foundation (WPF)**
  Graphical subsystem for rendering Windows-based user interfaces. Uses the *XAML* language to define and link various user interface components. Used in this thesis to describe the graphical representations of the extension's views.

- **TeXstudio 2.5.1 - http://texstudio.sourceforge.net/**
  An open source integrated environment for Windows for the creation of *LaTeX* documents. Used for the composition of this thesis document.

- **Paint.NET - http://www.getpaint.net/**
  An open source image and photo editing software for Windows. Used for the generation and manipulation of icons and images used by the plug-in and the thesis.

# Appendix E

# Installation Manual

**Installation by means of the *Microsoft Visual Studio Extension Installer*:**

- In order to launch the setup, the *InteractiveExploration.msi* file has to be launched. This file resides in the source code folder that is provided with the CD-ROM under the following path: *Source Code/InteractiveExploration/InteractiveExploration/bin/Debug*.
- Install *Graphviz 2.28.0* by means of the provided executable (*graphviz-2.28.0.msi*) in the installer folder. This installs the *dot.exe* command line tool required by the extension. In case Graphviz cannot be installed to the default destination, make sure to modify the path to *dot.exe* in the plug-in settings.
- Install the *PQLabs MultiTouch Platform and Screen Driver*, which is also available from the installer folder (*PQLabsMultiTouchWinDriver.exe*). This driver is required in order to recognize multi-touch gestures on the *PQLabs G3 Monitor*.

**Installation from source code:**

- Open the Visual Studio 2010 solution residing in the provided source code folder of the CD-ROM (*Source Code/InteractiveExploration/InteractiveExploration.sln*).
- Build the solution. This creates the corresponding *DLL* files in the debug folder of the project (*Source Code/InteractiveExploration/InteractiveExploration/bin/Debug*).
- Copy the following files from the debug folder to Visual Studio's *Private Assembly Folder* (*C:/Program Files (x86)/Microsoft Visual Studio 10.0/Common7/IDE/PrivateAssemblies*) such that they are available at runtime:

  *Antlr3.Runtime.dll*

  *Graphviz4Net.dll*

  *Graphviz4Net.WPF.dll*

  *InteractiveExploration.dll*

  *InteractiveExploration.Helper.dll*

  *InteractiveExploration.Model.dll*

  *ZoomAndPan.dll*

- In case those *DLLs* should be debugged, copy the corresponding *PDB* files from the debug folder to the private assembly folder as well.
- Install *Graphviz 2.28.0* and the *PQLabs MultiTouch Platform and Screen Driver* as instructed in the installation guideline above.

# Appendix F

# Contents of the CD-ROM

The following files are stored on the enclosed CD-ROM:

- **Zusfsg.txt**
  The German version of this thesis' abstract.
- **Abstract.txt**
  The English version of this thesis' abstract.
- **Bachelorarbeit.pdf**
  A copy of this thesis.
- **SourceCode.rar**
  The source code of the software described in this thesis.
- **Installer.rar**
  The executables needed in order to install the software described in this thesis.

# Bibliography


[BD06] Johannes Bohnet and Jürgen Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, pages 95–104, New York, NY, USA, 2006. ACM.

[BRZ+10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph 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*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[CKN+03] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03, pages 77–ff, New York, NY, USA, 2003. ACM.

[CVD07] Mauro Cherubini, Gina Venolia, and Rob DeLine. Building an ecologically valid, large-scale diagram to help developers stay oriented in their code. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '07, pages 157–162, Washington, DC, USA, 2007. IEEE Computer Society.

[CVDK07] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 557–566, New York, NY, USA, 2007. ACM.

[DR10] Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 207–210, New York, NY, USA, 2010. ACM.

[DVR] Robert DeLine, Gina Venolia, and Kael Rowan. Software development with code maps. *Queue*, 8:10:10–10:18.

[Fav01] J.-M. Favre. Gsee: a generic software exploration environment. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 233 –244, 2001.

[HH03] Ahmed E. Hassan and Richard C. Holt. Adg: Annotated dependency graphs for software understanding, 2003.

[HTB+07] Otmar Hilliges, Lucia Terrenghi, Sebastian Boring, David Kim, Hendrik Richter, and Andreas Butz. Designing for collaborative creative problem solving. In *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition*, C&C '07, pages 137–146, New York, NY, USA, 2007. ACM.

[JGR+12] Hans-Christian Jetter, Florian Geyer, Harald Reiterer, Raimund Dachselt, Gerhard Fischer, Rainer Groh, Michael Haller, and Thomas Herrmann. Designing collaborative interactive spaces. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, AVI '12, pages 818–820, New York, NY, USA, 2012. ACM.

[KKD+11] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 217–224, New York, NY, USA, 2011. ACM.

[KKKB12] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. Blaze: supporting two-phased call graph navigation in source code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 2195–2200, New York, NY, USA, 2012. ACM.

[KMCA06] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971 –987, dec. 2006.

[LL06] M. Lungu and M. Lanza. Softwarenaut: exploring hierarchical system decompositions. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 2 pp. –354, march 2006.

[LL07] Mircea Lungu and Michele Lanza. Exploring inter-module relationships in evolving software systems. In *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, pages 91 –102, march 2007.

[LMFA08] Seonah Lee, G.C. Murphy, T. Fritz, and M. Allen. How can diagramming tools help support programming activities? In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, pages 246 –249, sept. 2008.

[MWFG12] S. Muller, M. Wursch, T. Fritz, and H.C. Gall. An approach for collaborative code reviews using multi-touch technology. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2012 5th International Workshop on*, pages 93 –99, june 2012.

[NM90] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, NY, USA, 1990. ACM.

[SBM+02] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. Shrimp views: an interactive environment for information visualization and navigation. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '02, pages 520–521, New York, NY, USA, 2002. ACM.

[YM97] Peter Young and Malcolm Munro. A new view of call graphs for visualising code structures, 1997.