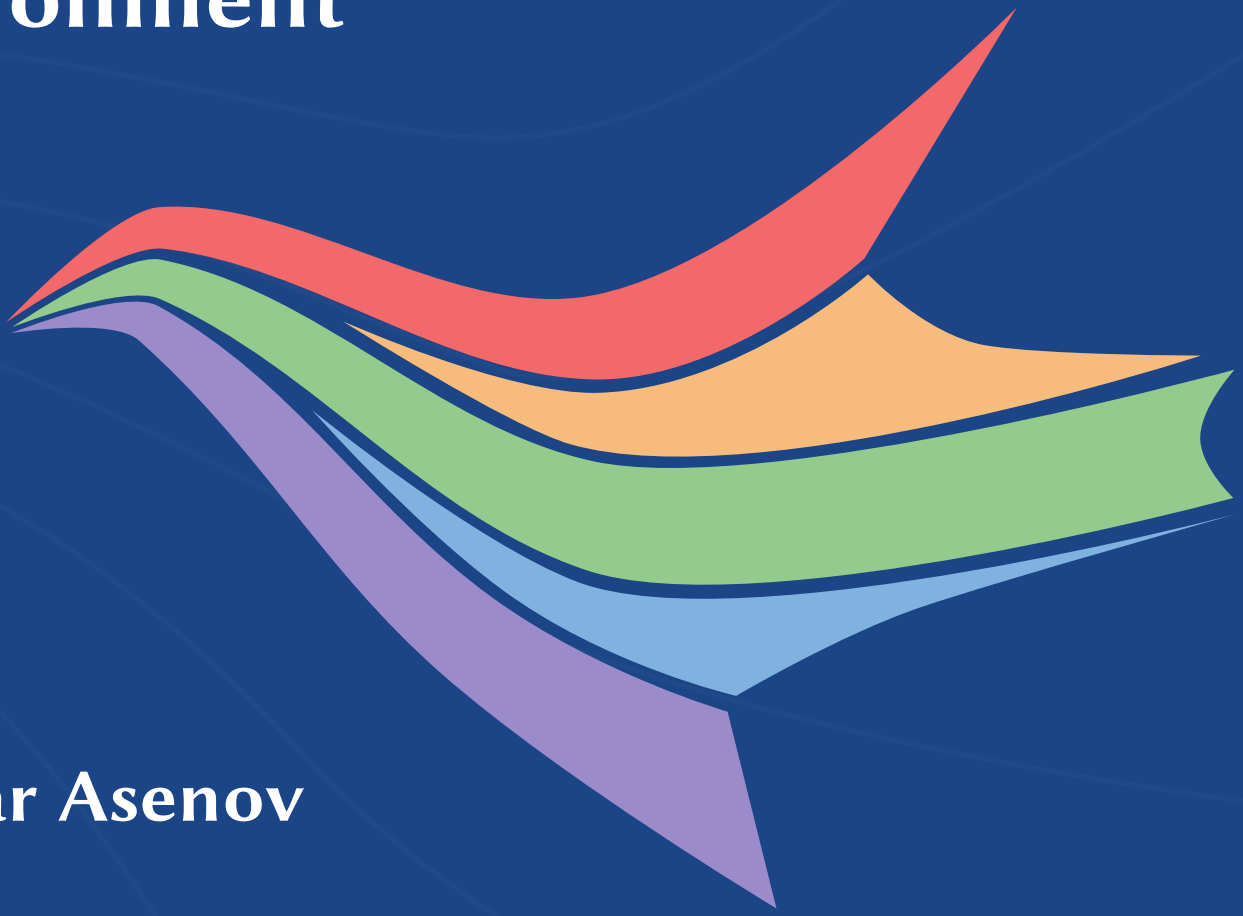
Envision: Reinventing the Integrated Development Environment



Dimitar Asenov

Diss. ETH No. 24074

DISS. ETH NO. 24074

Envision: Reinventing the Integrated Development Environment

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by

Dimitar Dimov Asenov
MSc ETH CS, ETH Zurich

born on 24.12.1986
citizen of Bulgaria

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Otmar Hilliges, co-examiner
Prof. Dr. Brad Myers, co-examiner
Dr. Robert DeLine, co-examiner

2017

Abstract

For decades, professional programmers have been reading and writing programs using a text editor to directly view and modify source files. Despite the ubiquity of this approach, it suffers from three inherent limitations caused by the tight coupling of the editing interface and the information stored on disk. First, programming notations are limited to text, even though people have additional capacities to process information such as visual and spatial perception. Text editors also preclude the use of established graphical notations from specific domains such as math or electrical engineering. Second, the content of source files is limited to what is comfortably readable by developers, even if additional information can help program comprehension or improve tools. For example, storing unique identifiers for each syntax token can help improve tracking code in version control systems or identifying code fragments between tools, but will result in a complete loss of readability. Third, many development tools such as version control systems process data on the textual level of source files, disregarding the structure of programs. This not only misses opportunities to improve the results of such tools, but also makes it harder to share data on a logical level between tools. These three limitations prevent a wide range of improvements in the efficiency of software construction. In this dissertation, we challenge the established practice of reading and writing programs as text by building a development environment in which programming interfaces are not tightly bound to the content of source files. Our approach removes the inherent limitations discussed above and enables us to explore new and more efficient ways of creating programs:

To help programmers more quickly understand programs, we explore visually rich presentations of code as an alternative to syntax highlighted text. Even though syntax highlighting is the main presentation used for improving code readability today, its effects and the effects of richer presentations of code have not been systematically evaluated. We provide additional evidence for the effects of rich code presentations by conducting a user study with developers, comparing syntax high-lighted text to two richer presentations of code. Our results show that richer visualizations help developers to more quickly understand code structure without causing visual overload, contrary to the subjective opinion of most participants in our study. We also explore how code presentation can be customized according to different types of context and domains, such as the nature of a developer's current task or what APIs are being used. We demonstrate customizations allowing domain-specific APIs from a library to use non-standard notations, thereby making domain code easier to read.

To enable programmers to efficiently and directly edit rich visual notations of code, we introduce novel interaction techniques for structured and visual code

editors. Such editors have traditionally suffered from poor usability due to disallowing syntactically or semantically invalid program states or requiring many extraneous actions when making local changes to code, a problem known as high viscosity. We design and evaluate a set of interaction techniques specifically created to tackle these problems. Results of CogTool simulations suggest that our interaction techniques permit programming with visually rich notations in a structured editor to be as efficient as editing text in a modern text editor.

To provide programmers with more precise information and additional automation during version control operations, we designed a version control system that uses additional meta-data saved directly in program sources. Mainstream version control systems work directly with the text of a program, disregarding the rich structure of code, resulting in inaccurate and confusing diffs, unnecessary conflicts, and incorrect merges, that waste developers' time. In order to eliminate these issues, we have designed version control algorithms that work with the tree structure and semantics of programs supported by additional automatically-generated meta-data stored within the program's source code. Unlike other research prototypes of advanced version control systems, our algorithms are designed to interoperate with existing version control tools and infrastructure such as GitHub, making them more practical. An evaluation of our approach on popular Java projects shows substantially improved merge results compared to Git.

To enable developers to effectively combine information from different sources and tools, we embed a scriptable information system directly in the development environment. Despite the fact that developers regularly work with many different sources of information such as the program code, version repositories, issue trackers, or profilers, tool support for combining, visualizing, and acting on such information is lacking. Our approach overcomes these deficiencies by enabling developers to write information queries in order to process and combine information from arbitrary sources, visualize the results in diverse forms (e.g., using highlights, diagrams, tables, or custom visualizations), and automatically perform actions (e.g., changing the code or filing a bug report). We demonstrate the versatility of our information system by applying it to a number of practical scenarios encountered by developers.

To explore how the techniques above can complement each other, we have built Envision – an open-source IDE that we used as a vehicle for our research and that integrates all of our work. Envision is highly extensible and customizable in order to suit the needs of both professional programmers who need to fine-tune their environment and researchers who need a framework for rapid experimentation.

Zusammenfassung

Seit Jahrzehnten lesen und schreiben professionelle Softwareentwickler Programme mit einem Texteditor der es ermöglicht Quelldateien direkt anzusehen und zu ändern. Trotz der Allgegenwart dieses Ansatzes leidet er unter drei inhärenten Einschränkungen, die durch die enge Kopplung der Editierschnittstelle und die auf der Festplatte gespeicherten Informationen verursacht werden. Erstens, Programmnotationen sind auf Text beschränkt, obwohl Menschen zusätzliche Kapazitäten zur Verarbeitung von Informationen haben, wie, z. B. visuelle und räumliche Wahrnehmung. Texteditoren schliessen auch die Verwendung von etablierten grafischen Notationen aus bestimmten Bereichen wie Mathematik oder Elektrotechnik aus. Zweitens, der Inhalt der Quelldateien ist beschränkt auf das, was von Entwicklern bequem lesbar ist, auch wenn zusätzliche Informationen das Verständnis des Programms oder die Verbesserung der Werkzeuge unterstützen könnten. Beispielsweise kann das Speichern von eindeutigen Identifikatoren für jedes Syntax-Token dazu beitragen, das Verwalten des Codes in Versionskontrollsystemen zu verbessern oder Codefragmente zwischen Werkzeugen zu identifizieren. Allerdings führt die Einführung dieser Metadaten zu einem vollständigen Verlust der Lesbarkeit. Drittens, viele Entwicklungswerkzeuge wie Versionskontrollsysteme verarbeiten Daten auf der textlichen Ebene von Quelldateien und ignorieren die Struktur von Programmen. Dies versäumt nicht nur Möglichkeiten, die Ergebnisse solcher Entwicklungswerkzeuge zu verbessern, sondern macht es auch schwieriger, Daten auf einer logischen Ebene zwischen den Werkzeuge zu teilen. Diese drei Einschränkungen verhindern eine breite Palette an Verbesserungen bei der Effizienz der Softwareentwicklung. In dieser Dissertation fordern wir die etablierte Praxis des Lesens und Schreibens von Programmen als Text heraus, indem wir eine Entwicklungsumgebung entwickeln, in der Programmnotationen nicht eng an den Inhalt der Quelldateien gebunden sind. Unser Ansatz entfernt die inhärenten, oben diskutiert Einschränkungen und ermöglicht es uns, neue und effizientere Wege der Entwicklung von Programmen zu erkunden:

Um Programmierern zu helfen, Programme schneller zu verstehen, erforschen wir visuell vielfältige Präsentationen von Code als eine Alternative zu syntax-hervorgehobenem Text. Obwohl die Syntax-Hervorhebung die Hauptpräsentation ist, die heute zur Verbesserung der Codelesbarkeit verwendet wird, wurden ihre Effekte und die Effekte vielfältigerer Darstellungen von Code nicht systematisch ausgewertet. Wir bieten zusätzliche Beweise für die Auswirkungen von vielfältigen Codedarstellungen, durch eine Anwenderstudie mit Entwicklern, wobei wir syntax-hervorgehobenen Text mit zwei vielfältigen Codedarstellungen vergleichen. Unsere Ergebnisse zeigen, dass vielfältige Visualisierungen den Entwicklern helfen, Codestrukturen schneller zu verstehen, ohne visuelle Überlastung zu verursachen, im Gegensatz zu der sub-

jektiven Meinung der meisten Teilnehmer in unserer Studie. Wir erkunden auch, wie die Codedarstellungen an verschiedene Kontexte und Anwendungsdomänen angepasst werden können, z. B. die Art der aktuellen Aufgabe eines Entwicklers oder welche APIs verwendet werden. Wir zeigen Anpassungen, die es ermöglichen, dass domänenspezifische APIs aus einer Bibliothek nicht standardmässige Notationen verwenden, wodurch der Domänencode einfacher zu lesen ist.

Um es Programmierern zu ermöglichen, vielfältige Codedarstellungen effizient und direkt zu bearbeiten, führen wir neuartige Interaktionstechniken für strukturierte und visuelle Code-Editoren ein. Solche Editoren haben traditionell eine schlechte Nutzbarkeit, weil sie syntaktisch oder semantisch ungültige Programmzustände nicht zulassen oder bei lokalen Änderungen am Code viele eigentlich unnötige Aktionen erfordern; ein Problem, das als „hohe Viskosität“ bekannt ist. Wir entwerfen und bewerten eine Reihe von Interaktionstechniken, die speziell entwickelt wurden, um diese Probleme anzugehen. Von uns durchgeführte CogTool-Simulationen lassen vermuten, dass unsere Interaktionstechniken die Programmierung mit visuell vielfältigen Notationen in einem strukturierten Editor so effizient wie die Bearbeitung von Text in einem modernen Texteditor sein lassen.

Um Programmierern präzisere Informationen und zusätzliche Automatisierung während der Versionskontrolle zu bieten, haben wir ein Versionskontrollsystem entwickelt, das zusätzliche Metadaten verwendet, die direkt in den Programmquellen gespeichert werden. Standard-Versionskontrollsysteme arbeiten direkt mit dem Text eines Programms, ohne Rücksicht auf die vielfältige Struktur des Codes, was zu ungenauen und verwirrenden Diffs, unnötigen Konflikten und fehlerhaften Merges führt, die die Zeit der Entwickler vergeuden. Um diese Probleme zu beseitigen, haben wir Versionskontrollalgorithmen entwickelt, die mit der Baumstruktur und der Semantik von Programmen arbeiten und die durch zusätzliche automatisch generierte Metadaten unterstützt werden, die im Code des Programms gespeichert sind. Anders als andere Forschungsprototypen von fortschrittlichen Versionskontrollsystemen sind unsere Algorithmen darauf ausgelegt, mit bestehenden Versionskontrollwerkzeugen und Infrastrukturen wie GitHub zusammenzuarbeiten, was sie praxistauglicher macht. Eine Auswertung unseres Ansatzes bei gängigen Java-Projekten zeigt deutlich bessere Ergebnisse bei Merges im Vergleich zu Git.

Um Entwicklern die effiziente Kombination von Informationen aus verschiedenen Quellen und Werkzeugen zu ermöglichen, haben wir ein skriptfähiges Informationssystem direkt in die Entwicklungsumgebung eingebettet. Trotz der Tatsache, dass Entwickler regelmässig mit vielen unterschiedlichen Informationsquellen wie Programmcode, Versionsrepositories, Fehlerverfolgungssystemen oder Profilern arbeiten, fehlt es an Werkzeugunterstützung für die Kombination, Visualisierung und Handhabung solcher Informationen. Unser Ansatz überwindet diese Defizite, indem er es Entwicklern ermöglicht, Informationsabfragen zu schreiben, um Informationen aus beliebigen Quellen zu verarbeiten und zu kombinieren, die Ergebnisse in unterschiedlichen Formen zu visualisieren (z. B. mit Hilfe von Highlights, Diagrammen, Tabellen oder benutzerdefinierten Visualisierungen) und automatisch Aktionen durchzuführen (z. B. Änderung des Codes oder Einreichung eines Fehlerberichts). Wir demonstrieren die Vielseitigkeit unseres Informationssystems, indem wir es auf eine Reihe von Szenarien anwenden auf die Entwickler in der Praxis regelmässig stossen.

Um zu erforschen, wie die oben genannten Techniken einander ergänzen können, haben wir Envision entwickelt — eine Open-Source-Entwicklungsumgebung, die wir

als Vehikel für unsere Forschung verwendet haben und die alle unsere Arbeit integriert. Envision ist flexibel erweiterbar und anpassbar, um sowohl auf die Bedürfnisse von professionellen Programmierern einzugehen die ihre Entwicklungsumgebung frei konfigurieren möchten, als auch auf die von Forschern, die ein Framework für schnelle Experimente benötigen.

Acknowledgements

Many people have influenced my work and myself, showing me new horizons and making me a better researcher. I am grateful for their support, dedication, wisdom, and inspiration. I would like to acknowledge:

My advisor, Peter Müller, who not only took a leap of faith by allowing me to dive into an area away from his core expertise, but also stood by me throughout this endeavor. He was always a solid support and a source of encouragement, leaving me energized to press onwards after each of our meetings. Through his wisdom and experience, I learned much from him: how to conduct proper research, to communicate clearly, and to inspire. Thank you, Peter, for believing in me, for our motivating discussions, for your advice and alternative perspectives, and for teaching the most engaging and thought-provoking course I have ever attended, which brought us together in the first place.

My co-advisor, Otmar Hilliges, for his invaluable insights on statistics and experiment design and for finding the time to comment on my dissertation.

The co-examineers of my dissertation, Brad Myers and Rob DeLine, whose research and the tools they have built have inspired me greatly. Thank you for taking the time out of your busy schedules to attend my defense and provide valuable insight and comments on my work.

My colleagues Hermann Lehner and Marco Eilers, for finding the time to read early drafts of this dissertation and providing me with critical and very helpful feedback.

My wife, Lucia, for reading and commenting on early drafts of this dissertation and for supporting me throughout my studies. Thank you for your interest in this project, your insights of an experienced software engineer, and your continued encouragement.

My colleague Malte Schwerhoff, for giving me valuable feedback on the introduction of this dissertation and for proofreading the German version of the abstract.

My students who helped make Envision what it is today: Lukas Vogel, for our collaboration on Envision's information system, for his work on debugging support and C++ importing functionality, and for being so dedicated, systematic, and supportive; Patrick Lüthi, for his work on semantic zoom and self-hosting Envision and for his dedication and attention to detail; Balz Guenat, Martin Otth, and Vaishal Shah, for our collaboration on Envision's version control system and their dedication to the project; Manuel Galbier, for his hard work on Envision's code review functionality and for being persistent, systematic, and putting in the extra effort to polish Envision's version-control visualizations; Andrea Helfenstein, for designing the declarative framework for Envision's visualizations; Cyril Steimer, for improving

Envision's support for tasks, views and exploration; and Jonas Trappenberg and Sascha Dinkel, for implementing Envision's rich comment support.

John Boyland, for our fruitful discussions on version control and automatic proof systems.

All members of the Chair of Programming Methodology at ETH for providing a stimulating and friendly work environment, which made me happy to go to work every morning. I would like to especially thank Ioannis Kassios, Alex Summers, Uri Juhasz, Malte Schwerhoff, Lucas Brutschy, Valentin Wüstholz, Maria Christakis, Milos Novacek, Caterina Urban, Marco Eilers, Arshavir Ter-Gabrielyan, and Jérôme Dohrau for our many stimulating discussions on the design of programming languages and tools; and Marlies Weissert, for making sure that everything in the office runs smoothly and for providing invaluable insights about Switzerland.

My parents, Ginka and Dimo, for their open-mindedness, unconditional support, and infinite patience. I am forever grateful to have grown up with your sage advice in the peaceful and empowering environment that you created.

Thank you.

Contents

Introduction 1

What do architects, music producers, animators, and software engineers have in common? They are people who use tools to produce complex digital artifacts by creating and manipulating information structures of their respective domain. *Information structures* are objects and relations between objects, which are used in the process of creating a digital artifact and for expressing the final artifact itself. For example, animators may use a 3D-modeling tool such as Autodesk 3ds Max [3ds] to make a 3D animated movie, using information structures like a script, key frames, object deformations, and surface shaders. Analogously, software engineers use integrated development environments (IDEs) such as Eclipse [Ecl] or code editors such as Vim [Vim] to make an application by manipulating structures like specifications, documentation, classes, tests, and build rules. The ever-increasing complexity of some types of digital artifacts, such as software, necessitates an improvement in the efficiency with which the artifacts can be constructed. For experts in any given domain, significant improvements in efficiency do not come from further education or experience, but only from improved information structures and better tools.

Improving information structures means to introduce new types of structures or tweak existing types so that it becomes easier or possible to create digital artifacts with desired properties. Such improvements typically happen when our understanding about a domain improves or when the domain evolves. For example, in architecture, the increasing development of telecommunications led to the introduction of structured cabling, which standardizes complex network infrastructure [LL96], making it easier to create blueprints for buildings with advanced networking needs. An example from software engineering is the transition from `goto` statements to structured programming [Dij68], which made it easier to write more understandable and correct code.

Improving the tools for creating digital artifacts means to make it easier to create digital artifacts using existing types of information structures and to facilitate the understanding of these structures: their properties, their relationships, and their behaviors. For example, the development of powerful graphics rendering hardware enabled animators to more quickly experiment with different materials [Kil00], making it easier to achieve the look they want. In software engineering, the introduction of interactive debuggers [ED66] made it easier to understand and fix the behavior of programs in certain situations.

Improving information structures and creating better tools are interrelated. On the one hand, when information structures evolve, new tools are often needed to handle the new structures, for example, introducing lambda expressions to a programming language requires an update to the compiler. On the other hand, some

tools require additional information structures in order to function. For example, a program verifier may require code to be annotated with specifications, which introduce new information related to the code. As a consequence, if either the information structures or the tools in a given domain are inflexible or slow to evolve, this can hinder improvement in both and prevent more efficient ways of creating digital artifacts in that domain. We argue that this is the case in software engineering. Next, we explain how the continued use of a text-editor to directly read and write textual source files impedes improvements in the efficiency of creating software.

1.1 Issues of programming as creating text

> Why, in the early 21st century, do programmers still insist that their tools have to draw exactly one glyph on the screen for each byte in their source files? No one expects AutoCAD or Microsoft Word to do this; even grizzled old Unix fanatics don't expect to be able to open a relational database with Vi or Emacs. One of the great ironies of the early 21st century is that secretaries can easily put organizational charts or cubicle floor plans in e-mail messages, but the programmers who made that possible can't put class diagrams in their code.
>
> *Gregory V. Wilson* [Wil04]

Today, like in the past several decades, developers still use a text editor to read and edit source files directly. While programming languages have evolved and text editors have received many improvements such as syntax highlighting, text decorations, code folding, or integration with other development tools, the fundamental experience of programming as writing and reading text has not changed. Source files are the central container for information structures in software engineering, and the fact that their contents is coupled to a specific type of tool creates three fundamental issues: limited information, rigid notation, and basic tools.

Limited information structures in source files

One limitation of source files is that they contain, almost exclusively information that is conveniently expressed as text. Even though diagrams, tables, and other non-textual artifacts are very useful in software engineering, such artifacts are rarely included in source files. Even when they are included, non-textual artifacts are limited to what is possible to express as static ASCII characters, e.g., the table in Figure 1.1. Non-textual and other artifacts typical for software engineering projects such as requirements, design descriptions, diagrams, models, and user documentation are, thus, created and maintained separately from source code. This separation often prevents effective links between code and other artifacts and creates inconsistencies. For example, a rename refactoring may not be propagated to a table like the one from Figure 1.1 and is almost certainly not reflected in any files outside of source code.

Another limitation of source files is that they are unable to contain dense annotations and meta-data without a catastrophic reduction in readability. *Dense* annotations and meta-data are pieces of information that are associated with arbitrary

```
/* list of SSA commands
 *
 * I use three address code (TAC) as low-level IR.
 * It will be implemented with indirect triples.
 *
 * +------------------+------+------+---------+
 * |      command     | arg1 | arg2 |   op    |
 * +------------------+------+------+---------+
 * |    t = a + b     |  a   |  b   |    +    |
 * +------------------+------+------+---------+
 * |    t = a - b     |  a   |  b   |    -    |
 * +------------------+------+------+---------+
 ...
```

Figure 1.1: A diagram in source code using ASCII symbols [Jus].

nodes of the AST or text tokens and are orthogonal to the syntax of the used programming language. An example of dense meta-data are unique IDs for each node of a program's AST (see Figure 1.2). Such dense data is desirable, for example, to improve the operation of tools (e.g., version control as we show in Chapter 7). Dense annotations and meta-data require specialized tool support, in order to remain hidden from the user and be properly maintained when the program is edited. Using a text editor that directly shows the underlying sources is infeasible – imagine trying to read a Java file shown as in Figure 1.2.

```
a /* db7cd599-596e-4caf-b1e9-ebb165ccc2e4 */
> /* c787ca8e-29f8-466c-93c3-aec52139788c */
b /* 8bb2d61e-31d2-4d5b-9410-dea50da30b02 */
? /* 6807c9a8-732a-4b2d-bf6b-0f79f151837b */
a /* 8413cdc6-4b05-4991-a94e-ebd9c5cea56f */
: b /* 3fa8c837-37ee-486c-94cf-605edee106a5 */
```

Figure 1.2: The expression `a>b?a:b` with a 128-bit unique ID for each AST node.

Rigid notation

The users of a text editor directly observe and manipulate the underlying files, which forces programmers to mostly work with a textual notation. However, depending on the current task of the developer a visual notation might be more effective [WNF06]. Kosar et al. [KMC12] show that for specific tasks programmers are more productive using domain-specific languages compared to general-purpose ones. While their study looked at textual languages, some domains have a natural and established visual notation (e.g., music, electrical circuits, state machines, etc.), which is, however, incompatible with text-based editors. Acknowledging the usefulness of visual notations, modern IDEs provide tools to visualize class hierarchies and other parts of the code, and research prototypes such as Code Bubbles [BRZ+10] and Code Canvas [DR10] even offer ways to visually structure code fragments on a two-dimensional canvas. While these are important advances, all of these approaches

still feature a standard text-editor and a fixed textual notation for the majority of programming tasks. As we will show in Chapter 5, even if only the basic structures of code are considered, such as expressions, statements, and methods, a text-only presentation is inferior to richer presentations that mix text and graphics based on the program's structure. The rigid notation imposed by text-editors precludes more efficient notations.

Another issue with textual source files is that developers need to deal with formatting and to precisely spell out code according to a language syntax, as if semicolons and curly braces were an inherent part of describing computation. While novices are more significantly affected by these issues, experts also waste time on them – a recent study at Microsoft [ABBS14] finds that 9% of code reviews contain comments related to formatting. Encoding ASTs in a compact textual syntax may also lead to cumbersome notations when programming languages evolve. For example, recent extensions of the C++ language add the new keywords `nullptr`, `co_await`, and `co_yield`. These keywords are suboptimal and not the ones language designers would have picked for a new language, but the preferred alternatives `null`, `await`, and `yield` could not be used, because they would clash with a large number of existing identifier names. Such a compromise in notation would not have been necessary if token types were explicitly stored in the source file, as opposed to deduced from the syntax. However, token types are dense meta-data, which is infeasible when using a text-editor, as we have seen above.

Basic and isolated tools

Textual source files are the lowest common denominator across all development tools. While there are tools which work with the structure and semantics of a program (e.g., compilers or static analyzers), many frequently used tools in software engineering remain on the textual level. A prominent example is the version control system, which completely disregards the code's real structure. For example, Figure 1.3 shows a merge conflict reported by Git. Because Git sees code as lines of text, inserting different lines at the same location is a conflict, even though in this case the order is irrelevant and the two changes can be automatically merged. A developer would have to manually resolve such conflicts, wasting time.

Using a lowest common data format as basic as text makes it harder to create

```
import rx.subscriptions.BooleanSubscription;
import rx.subscriptions.Subscriptions;
<<<<<<< master
import rx.util.BufferClosing;
import rx.util.BufferOpening;
=======
import rx.util.OnErrorNotImplementedException;
>>>>>>> dev
import rx.util.Range;
import rx.util.Timestamped;
```

Figure 1.3: A merge conflict where two branches insert import declarations at the same location in the file. The developer resolved the conflict by using all three imports [Rea].

smarter development tools because it is more difficult for tools to share data on a high-level. Thus, even when smart tools are built, they often exist in isolated silos and the developer has to manually combine information from such tools. For example, Kuhn [Kuh12] reports on his experience developing an Eclipse plug-in that shows a code map. Even though the information needed to construct the map (call hierarchies) was already computed by another standard Eclipse plug-in, there was no convenient way to access this information, and it was ultimately retrieved by scraping the user interface components of the standard plug-in. This is an unnecessarily complex and error-prone solution, though it is still preferable to computing the information from scratch. Kuhn's further analysis reveals that almost none of the default Eclipse plug-ins have mechanisms for sharing data.

Resolving the issues above can result in substantial productivity gains for developers. Next, we outline our approach and the open challenges for achieving these gains.

1.2 Improving the programming experience

1.2.1 Approach

We can observe that virtually all domains where digital artifacts are produced, except programming, share a common property: information structures and their containers (files) are independent of tool interfaces. For example, even though SVG images are stored on disk as XML files, graphics designers do not use a text-editor to create SVG images, but rather a dedicated graphical interface. The decoupling of information structures and editing interfaces achieves two goals. First, there are no longer any restrictions on what type of data the information structures contain. For example, SVG files can contain dense meta-data, because there is no need for a human to read the raw XML code directly. Second, interfaces are not bound to a particular notation and can selectively show information, offering a variety of user experiences to match the current task. For example, when aligning objects, SVG editors may show additional guides directly on the canvas to enable a designer to visually achieve the look they want by dragging; with complex images, designers can hide certain layers, which lets them focus on particular visual elements.

We argue that using rich information structures and interfaces that offer dedicated support for different tasks, unlocks many possibilities for making software construction more efficient, such as making more tools aware of code structure and enabling new ways to understand programs. For example, meta-data enables grouping scattered pieces of code together to form working sets for particular kinds of tasks, as suggested by Storey et al. [SCBR06] and dense meta-data, such as node IDs, allows fine-grained and accurate version control, which, in the absence of such meta-data, can only be approximated with time-consuming tree-matching algorithms [Bil05, FWPG07, Rei08, FMB+14]. To guide the design of rich information structures and dedicated interfaces, we ask the following questions:

- *What information structures are needed to describe programs and the software engineering process?* Developers work daily with many types of information structures. Source code holds, perhaps, the most important ones such as expressions, statements, methods, classes, packages, tests, build rules, etc.

Others are found in external sources such as bug reports, software versions, profile and test results, software architecture and design documents, etc.

- *How are these structures connected and used, in what context, and for what purpose?* There are many different dimensions along which this question can be answered. One dimension is the context of a developer's current activity: fixing a bug, creating a new feature, profiling, exploring unfamiliar code, writing documentation, creating a high-level design, etc. Another dimension is the target domain of a piece of code: e.g., physics simulations, network communications, or accounting. Yet another dimension is the familiarity of the developer with a particular code base, implementation language, or organization. An important aspect of how developers use information is that different information types are interlinked and developers often need to combine information from various sources [KDV07, SMDV08, FM10].

- *What presentations of the information structures enable quick comprehension?* Different notations for the same concepts can be more or less effective depending on the task of a user [GP92, WNF06]. Thus a good tool should be able to provide appropriate, and possibly rather different, notations depending on the context. The context might even include the developer, so that the presentation of code may be automatically derived from code structure and customized to each developer's individual preferences.

- *What interactions and interfaces enable efficient creation and manipulation of each information structure?* As we have seen, a dedicated interface is preferred over a generic one. A dedicated interface is one that fits a particular information structure well and provides intuitive and direct interactions for manipulating it on a logical level.

These questions show that the space for designing rich information structures and interfaces for software development is vast, which poses many challenges.

1.2.2 Challenges

It is our goal to support the majority of professional programmers today. This means designing general-purpose techniques and tools that can be used efficiently, scale well to large projects, can be customized, and are realistically adoptable. Achieving all of these qualities is non-trivial even with traditional programming approaches, and poses a number of additional challenges in the context of novel programming interfaces and rich information structures.

Code presentation

Programming interfaces that are not bound to a textual presentation of code offer a huge space for exploring programming notations. Despite decades of research on visual programming, there is little evidence to guide the design of visually-enhanced notations for wide-spread programming languages and paradigms such as object-oriented programming. Recent tools such as JASPER [CKM06], Code Bubbles [BRZ+10, BZR+10], Code Canvas [DR10], and Debugger Canvas [DBR+12] only wrap a standard code notation (syntax-highlighted text) in a visual shell. Other, more visual tools are either designed for specific domains (e.g., LabView [Lab]),

target novice programmers (e.g., Scratch [MRR+10], MIT App Inventor [Wol11], or Alice [Coo10]), or are primarily used by end-users (e.g., spreadsheets). Barista by Ko and Myers [KM06] introduces richer notations for programming, but to our knowledge these notations and the usability of the system have not been evaluated. In general, there is only scarce empirical research about the notation of wide-spread programming languages and paradigms [MSH+16] and most of it focuses on aspects of textual syntax (e.g., [McI01, SS13]) or performance of novice programmers (e.g., [AB15]). In their seminal work on the Cognitive Dimensions framework [GP96], Green and Petre discuss many different aspects of notations, but only compare a traditional programming language (Basic) to visual languages using a different programming paradigm (data-flow programming), and do not investigate different notations for the same paradigm. Generally, the Cognitive Dimensions framework cannot be used to directly guide us in creating visually-enhanced notations for code.

Efficient editor interactions

In order to prevent information overload, rich information structures require code editors that can work with selected logical fragments of the program. Thus, such editors must operate on code structure – so called structured editors. There is considerable amounts of research of structured and visual program editors as an alternative to text [TR81, HN86, Cow87, GKM90, RW91, MPMV94, SH06, KM06, VSBK14]. Unlike text editors, structured editors do not see code as a sequence of characters, but rather as a structure and provide an interface that lets developers edit the units of this structure in order to modify the program at a logical level.

While many of the useful features of modern text editors can be easily transferred to structured editors (e.g., syntax highlighting, text decorations, auto-completion), structured editors typically suffer from significantly lower usability compared to text-editors, due to the severe restrictions structured editing imposes on the way programs are edited, resulting in cumbersome and slow interactions [MPMV94]. Visual programming environments, which are a particular kind of structured editors, have been shown to have a much higher viscosity compared to textual languages [Gre90, GP96]. A high viscosity means that making local changes to the program requires a lot of operations, thereby reducing programmer efficiency. The high viscosity of structured and visual programming environments presents a major usability barrier for professional developers who prefer the unconstrained manipulation of textual code.

Successful blocks programming environments [MRR+10, Coo10, Wol11] sidestep the issue of high viscosity, since their target audience is novice programmers who have different interaction patterns, which are much less affected by viscosity compared to professionals. Professional programmers are especially deterred by the strong focus on mouse-based interactions typical of visual and blocks environments. Stride [McK12, PBL+16] is a structured editor for novices that achieves good usability with keyboard-based interactions, but it is limited to a fixed set of code presentations and does not allow customization. Barista [KM06] also enables editing of graphical code presentations with the keyboard by converting them to text, which prevents interactive graphical interfaces. Overcoming high viscosity for expert developers is the central challenge in designing practical structured editors. Providing good usability becomes even more challenging if an editor should support a diverse set of customizable and directly editable notations.

Working with diverse information

Despite the fact that developers use many different types of information daily [FM10, KDV07, SMDV08], modern IDEs are not designed to function as information systems and lack good support for storing, querying, combining, and visualizing arbitrary information. Since IDEs do not offer sufficient channels for sharing information between tools, IDE plug-ins are typically isolated [Kuh12], preventing effective collaboration between tools. Research tools such as Ferret [dAM08] or the information fragments model [FM10] provide the ability to pose questions directly within an IDE, but are often limited to only a few types of information, a fixed set of output visualizations, or read-only queries. Other tools such as JunGL [VEdM06] and Rascal [HKV12] offer the ability to automate actions, but are limited to working with source code and do not integrate other information sources. It is a challenge to create extensible data formats and tools that can store and process arbitrary information, while still allowing flexible visualizations, data exchange between tools, and automated actions.

Interoperability with the textual ecosystem

Apart from requiring new program editors, enhancing the information structures for programs introduces another challenge — how to interoperate with the rich ecosystem of existing text-based tools that developers rely on. Converting rich information structures to text could work for legacy tools that only read the text such as compilers and debuggers: the output messages of such tools could be mapped back to objects in the richer structures. However, tools that perform changes to the code (e.g., search and replace via regular expressions, or merging files in the version control system) need to be adapted or redesigned. Systems that work with structured data often use a dedicated set of tools and backends (e.g., EMFStore [KH10], Odyssey [MCPW08, OMW05], TouchDevelop [PBMM15] or MolhadoRef [DMJN07]), which prevents developers from using widely-available and cost-effective infrastructure such as GitHub. The challenge here is to benefit from rich information structures while utilizing existing tools and infrastructure whenever possible.

Integrating a new generation of tools

In addition to solving the individual problems discussed previously, a major challenge is to provide a platform that integrates all of these solutions and enables them to interoperate effectively, forming a powerful toolbox — an IDE. Such an IDE could be used by a broad group of researchers for rapidly prototyping and testing new ideas and by tool designers as a model for a next-generation programming system. We are not aware of any ongoing research efforts in this direction.

In our work we tackle these challenges in order to create a next-generation IDE that works with rich information structures and features a visual structured code editor. Next, we list the contributions that we make in this dissertation.

1.3 Contributions and outline

In our work, we have designed and evaluated novel techniques for building programming environments. Inspired by modern tools in other domains, we offer an advanced interface that works with structures richer than traditional source files and supports diverse domains and context-sensitive notations. We provide keyboard-based interactions throughout the program editor, enabling expert developers to write code as efficiently as they can in traditional systems. Our approach enables new visualization and information capabilities for IDEs, turning them into information systems for software development.

The product of our ideas is Envision – a prototype IDE which we designed to match the usability of text editors, to overcome their deficiencies, and to offer advanced features for constructing, understanding, and maintaining software in a professional setting. We designed Envision from the ground up using a holistic approach that considers both the human experience and the technical requirements of scalability, extensibility, and versatility in professional settings. Our design allows us to combine a significant number of visualization, interaction, and information processing techniques, achieving a level of integration not often seen in research tools. We have evaluated our work on Envision using various methods such as case studies, simulations, and a user study.

We make the following five key contributions.

1.3.1 Envision as a platform for experimentation

To enable comprehensive research of rich information structures and of dedicated interfaces for software development, we designed and created the Envision open-source IDE. Envision is a stand-alone tool, not based on any existing IDE. Unlike existing IDEs, Envision is built from the ground up to support non-textual and dedicated interfaces for programming and make use of a rich and flexible source code model that may hold any types of data (in particular, dense meta-data and annotations discussed in Section 1.1). This design has enabled us to explore programming interfaces more diverse than those of mainstream IDEs and to enhance source-code with data that helps to improve typical programming activities such as version control. Unlike other approaches that use structured and visual editors such as Intentional Software [SCC06] or MPS [VSBK14], Envision is not a language workbench and supports enhanced interfaces for general-purpose languages. We see Envision as an essential research platform that has enabled our research and may serve as a foundation for additional explorations of programming interfaces and tools, both our own and the explorations of other researchers.

In Chapter 3 we list the fundamental principles that guided the design of Envision to make it an effective platform for our research. These principles were derived from our research goals: to enrich the software structures contained in source files; to improve the programming interface; and to design a scalable and flexible tool, suitable for professional developers. The principles can guide the design of any system with a purpose similar to the goals of our research.

In Chapter 4, we describe the architecture of Envision [AM14a] and explain how it reflects the fundamental principles. Two highlights of our approach are its modularity and extensibility, which have enabled us to explore a diverse set of

research topics and integrate various tools into a single system, for example, domain-specific visual notations for software, scalable code maps, fine-grained version control, and information integration systems. The tool and its developer documentation are available at github.com/dimitar-asenov/Envision.

1.3.2 Flexible visual notations for programming

In Chapter 5 we address a key limitation of text-based editors: their limited notation. We describe an alternative editor approach that allows textual and graphical notations to be freely mixed [AM14b]. This provides a number of benefits to programmers. First, it enables the presentation and integration of non-textual artifacts of software engineering within the program code. For example, code comments in our system can use rich formatting and support elements such as embedded diagrams (editable directly in the IDE), tables, and interactive HTML/Javascript fragments. Second, the flexibility of notations makes it possible to better support embedded domain-specific languages [AM13]. For example, libraries can customize the notations and interfaces of APIs the libraries provide. Third, even for standard code structures such as expressions or statements, our approach enables notational improvements by using richer visualizations that enable developers to more quickly understand the structure of code. In a user study [AHM16], developers were able to answer questions about the structure of Java methods up to four times faster when using rich visualization compared to using standard syntax-highlighted text. We achieve these benefits without sacrificing the scalability of traditional editors and show how our approach is applicable to large code bases.

1.3.3 Efficient interactions in structured editors

In Chapter 6 we introduce a set of efficient interaction techniques for structured editors. As we described in Section 1.2, usability is a central issue with structured editors. To improve usability for expert programmers, we introduce a standard two-part interaction mechanism that uses the keyboard for fast input and applies across all notations. On the one hand, our system enables command-prompt-like interactions for controlling the IDE and for creating top-level code structures such as classes or methods. On the other hand, developers can type statements and expressions freely in a way that mimics text-based entry, even though all fragments of the program are maintained in a fully-structured form. In addition to these standard interactions, our system can be enhanced with custom interactions for each programming construct or domain-specific notation, which we also demonstrate. Using CogTool [JPSK04, BJRT10] we perform simulations of expert users and show that our approach allows code manipulation in structured editors as fast as in a text editor [AM14b].

1.3.4 Fine-grained and precise version control of trees

In Chapter 7 we describe version control algorithms that we designed specifically to support rich information structures represented as trees [AGMO17]. Version control of tree structures, ubiquitous in software engineering, is typically performed on a textual encoding of the trees (e.g., source code), rather than the trees directly. However, applying standard line-based diff and merge algorithms to such encodings

leads to inaccurate diffs, unnecessary conflicts, and incorrect merges. Traditional encodings are also unsuitable for rich information structures and dense data. To address these problems, we propose a novel format for storing trees on disk and novel algorithms for computing precise diffs between two versions of a tree, and for three-way merging of trees. Unlike most other approaches for version control of structured data, our approach integrates with mainstream version control systems such as Git. Our merge algorithm can be customized for specific application domains to further improve the quality of merge results. An evaluation of our approach on abstract syntax trees from popular Java projects shows substantially improved merge results compared to Git.

1.3.5 Scriptable information system within the IDE

In Chapter 8 we describe an approach for searching, integrating, and visualizing information from diverse sources directly within the IDE. Developers work with information from numerous sources such as the source code itself, compiler output, debugging and program analysis tools, version control information, issue trackers, project and API documentation, project wiki pages, and community resources like wikipedia.org and stackoverflow.com. In trying to meet their information needs, developers are faced with three issues. First, they often need to combine information from more than one source, but tool support for piecing information together is lacking [SMDV08] and developers are forced to manually connect the different pieces of information, which is an error-prone and time-consuming process. Second, most tools present information in a fixed form, which is not always a good match for a developer's specific information need (e.g., a visual call graph is better suited for detecting recursion than a long stack trace). Third, even after a developer finds the information they need, they often have to take action manually, for example, to make a change to the code or create a new bug report. Repeatedly performing an action manually is time-consuming, error-prone, and frustrating. Our approach [AMV16] solves these three problems by allowing developers to write queries that combine information from source code with external sources such as the file system, version control system, web services, etc. Combining different types of information is possible thanks to a unified data exchange format, which allows queries to be piped together. To improve the comprehension of query results, the results can be displayed in diverse ways, including entirely custom visualizations created by the developer using HTML and Javascript. The system can be extended with arbitrary data sources or computations thanks to an integrated support for Python scripts. Our approach provides a significantly more powerful alternative to standard text-based tools like searching with regular expressions, and it enables smart tools to more easily share high-level data with each other.

In the next chapter, we familiarize the reader with Envision's interface by means of a brief guided tour showcasing the tool's notation and interactions.

A quick tour of Envision 2

In this chapter we introduce the user interface of Envision and showcase key design features of our approach. Getting familiar with Envision's visualizations and interaction flow will facilitate the understanding of the core contributions of our work. For a video introduction to Envision, please see youtu.be/5YMaCQEoPe0. We will explore Envision in two different scenarios – creating a simple Java program from scratch and exploring an existing larger Java project. While Java is currently the best-supported language in Envision, the tool also allows support for other languages. For more details regarding Envision's architecture and implementation we refer the reader to Chapter 4.

2.1 Writing code from scratch

In this section we will explore Envision's code editing features by creating a simple application, step by step.

When Envision is started, the user is presented with the minimal interface shown in Figure 2.1. It consists of a white canvas where the program's code will later appear, and a command prompt, which is a central input mechanism in Envision. The interface is purposefully minimal, avoiding components such as menus, toolbars,

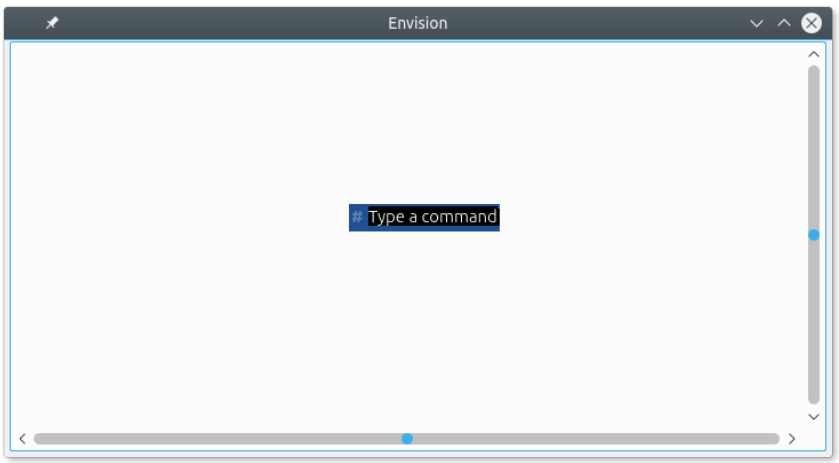


Figure 2.1: The initial screen a developer sees after starting Envision.

Figure 2.2: The `class` command creates a class called `Hello`.

Figure 2.3: A newly created class with the name `Hello`. The font size is deliberately large, so that it contrasts well with the font size of method names (medium) and the font size of statements and expressions (small).

and other views. This is reminiscent of terminal text editors like vim or graphical editors with "zen" or minimal distractions mode such as Atom [Ato] or Visual Studio Code [Visb]. As Envision was designed primarily for expert programmers, we enable them to focus on the code and offer access to all IDE features via keyboard shortcuts or the command prompt. This focus on keyboard interactions enables developers to quickly access any IDE function, without having to navigate hierarchical menus or lift their hands from the keyboard. For a discussion regarding the learnability of this interface we refer the reader to Section 2.3.

The command prompt not only provides access to IDE functions (e.g., loading a project or searching for code), but also offers commands for creating code. We will start by creating the `Hello` class, by invoking the `class` command, as illustrated in Figure 2.2. A command may have optional arguments, like `public` and `Hello` in this case, and its name does not need to appear in the first position in the prompt. This enables flexible input text, for example, to mimic familiar declarations in programming languages. The prompt also provides an auto-completion menu for commands, making it easier to remember available commands and quicker to type them. Pressing Enter executes the command and creates the corresponding class, which will then be displayed on the canvas, as shown in Figure 2.3. The prompt can be shown or hidden at any time by pressing ESC. The prompt is context-sensitive, and the available commands depend on the currently selected object, which, right now, is the freshly created class. When a class is selected, it is possible to create a method using the `method` command as shown in Figure 2.4.

In that figure, we show another feature of the prompt – its support for abbreviated commands. As long as the input string is unambiguous, command and argument names do not have to be spelled in full, and with ambiguous input the auto-completion menu can be used to select the desired command. Support for abbreviated commands makes it even quicker to execute IDE functions and create code.

Executing the `method` command creates the corresponding method shown in Figure 2.5. After the method is created, we can use the keyboard to select its various

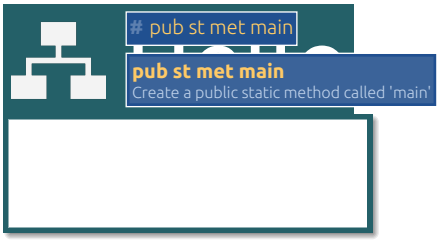


Figure 2.4: A usage of the `method` command. The command is abbreviated to `met` along with its arguments `public` and `static`.

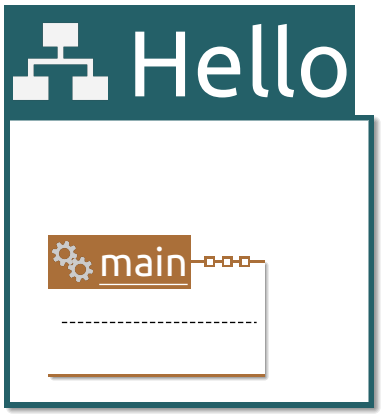


Figure 2.5: The newly created method `main`.

elements and add additional code. To do this we use a fundamental component of Envision's keyboard-focused interactions – a flexible cursor for making selections. Using the arrow keys (or, alternatively, the mouse), this cursor can be moved anywhere within the code in order to select existing code entities, placeholders for optional code elements, or the space between elements in vertical or horizontal sequences. These various selections are illustrated in Figure 2.6. The selection

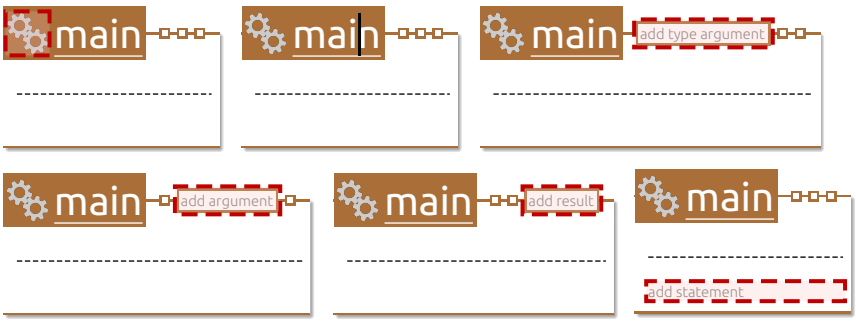


Figure 2.6: The cursor selecting various parts of the `main` method: icon, name, generic type arguments, arguments, result type, or body.

cursor's ability to take different forms and move to any position enables developers to reach and modify any code location using only the keyboard, regardless of how textual and graphical objects are composed on the canvas.

The command prompt is convenient for creating top-level code structures like classes or methods, which are mostly independent of peer constructs and are created infrequently. The prompt would offer poor usability for creating low-level code structures, like statements or expressions, which are more frequently manipulated and have a specific location with respect to their peers and context. Envision features another input mechanism for easily creating low-level code structures directly within the canvas. Expressions can simply be typed, left-to-right, as in a traditional text editor. Figure 2.7 illustrates expression entry. The input is parsed on each keystroke and converted to the corresponding AST nodes. All input sequences, including invalid ones, are accepted and stored as AST nodes – special error nodes are used to encode invalid fragments of the input. This allows for a flexible manipulation of expressions, where intermediate states are not restricted to error-free ASTs.

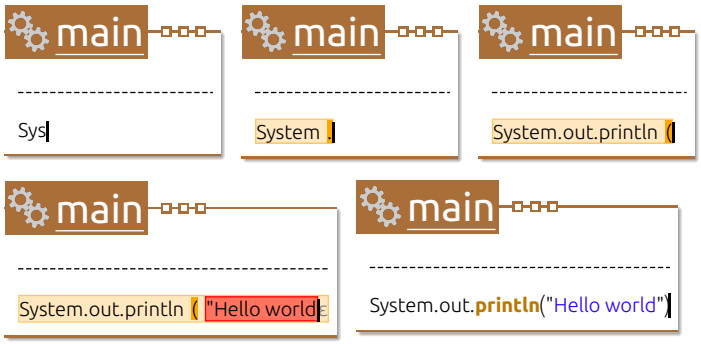


Figure 2.7: Typing an expression from left to right. During this process, syntactically incomplete operators are shown with a yellow background and text which is not understood, such as unfinished literals or unknown operators, is shown with a red background.

Most expressions in Envision are arranged horizontally as in the example above, but the visualizations are not limited to a horizontal arrangement. For example, names of method parameters appear above the corresponding parameter type as illustrated in Figure 2.8 and nested array initializers can be shown using a matrix notation as shown in Figure 2.9.

Many code components in Envision can be arranged in a two-dimensional layout,

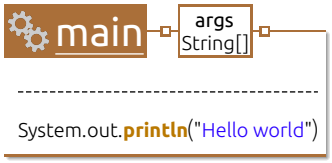


Figure 2.8: A `main` method with the usual `args` parameter. The parameter name appears above the parameter type.

Figure 2.9: A two-dimensional array rendered as a matrix. The matrix is editable. The textual equivalent is `int[][] identity = {{1,0},{0,1}}`.

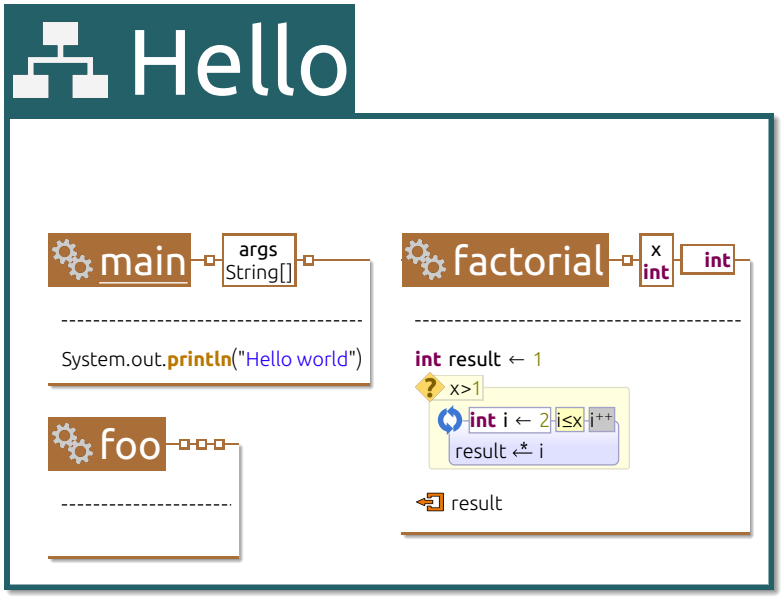


Figure 2.10: The `Hello` class holding three methods in a semi-grid: to reduce wasted space, by default all objects from the same column are vertically stacked independently of other columns.

similar to a grid. Work by Henley and Fleming [HF14] suggests that arranging code in a grid allows faster navigation compared to the traditional sequential arrangement in files, or to freely positioning code elements anywhere on the screen, e.g., as in Code Bubbles [BRZ+10]. To demonstrate the usage of grid arrangement in Envision, we add two additional methods to the `Hello` class, as shown in Figure 2.10. The cursor can be moved between the elements in a column and in between columns in order to create new elements or columns, respectively.

As can be seen on the different figures so far, and especially in Figure 2.10, Envision uses a visually richer presentation of code that is a mixture of text and graphics, unlike the purely textual presentations of traditional development environments. In Section 5.3 we discuss a user-study, which shows that these visualizations make it easier for developers to understand code structure. To explain Envision's code presentation, we contrast it to traditional syntax-highlighted text in Figure 2.11, which shows a factorial method as rendered by Eclipse and by Envision's default visualizations. To reduce syntactic noise, Envision eliminates semicolons at the end of statements, uses outlines instead of curly braces for showing scope, and shows alternating background colors instead of commas or semicolons for separating list elements. To improve structure detection, Envision uses icons instead of some keywords, and colored backgrounds for some statements and expressions. For a

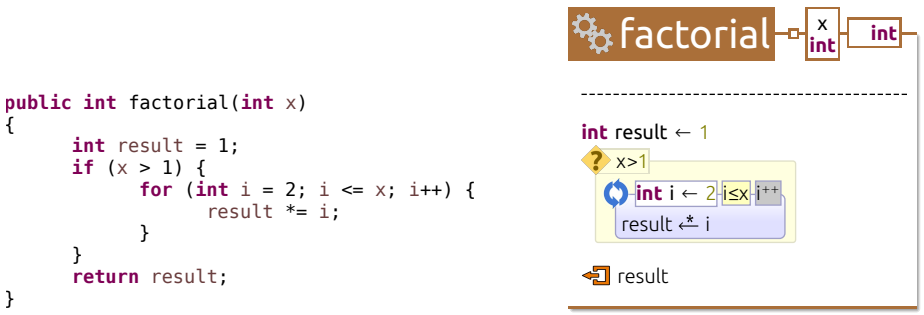


Figure 2.11: An identical method that computes the factorial of `x`, rendered by Eclipse and Envision, using default settings in both cases.

detailed discussion of the visual design of Envision we refer the reader to Section 5.2.

Despite their visual nature, the presentations can be created entirely using the keyboard. Creating the `factorial` method from Figure 2.11 in Envision is done with keystrokes that are almost identical to the ones used for creating the textual version. For example, to create a "for" loop we can type `for` and press Space – the original `for` text is automatically converted in a visual loop structure and the cursor is placed in the loop's initialization step element. We can then type an expression that initializes the loop variable and press ; , which will automatically place the cursor in the loop condition. After we finish writing the loop condition and step expressions, pressing Enter will position the cursor at the beginning of the loop's body, ready to create additional statements or expressions. Such a keyboard-based code entry is familiar to developers and helps to more quickly create code compared to, for example, dragging and dropping code constructs from a palette.

In addition to the default visualizations, any code fragment in Envision can also be displayed using custom visualizations. To demonstrate this we developed a control-flow visualization for method bodies, shown in Figure 2.12. Custom visualizations may be explicitly selected by the user, or automatically chosen, based on context. This enables support for customizing the presentation of code for specific tasks or domains as we show in Section 5.4.

Envision's flexible visualizations enable non-textual artifacts to be embedded directly in the source of a program. Thus, the program's source is not restricted to executable code and can be enhanced with rich information such as documentation, diagrams, and others. For example, we can enhance the `factorial` method with a rich comment like the one shown in Figure 2.13. This opens the possibilities for improved documentation in code and allows the integration of different artifacts from software engineering such as requirements, diagrams, design, documentation, and code, which could be processed together by tools, e.g., when performing a refactoring.

We have seen how Envision's interface enables working with a small code fragment. In the next section, we will explore a larger code-base.

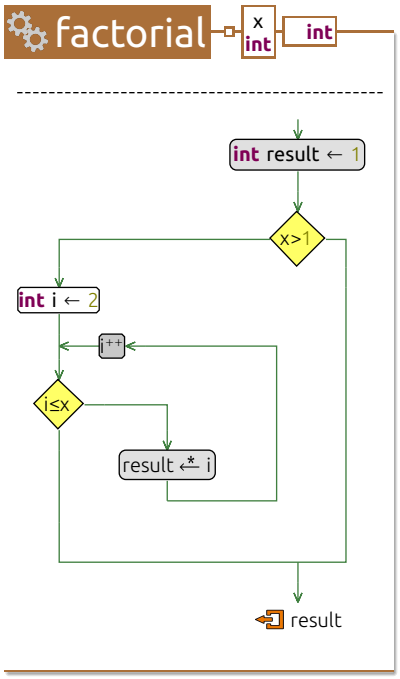


Figure 2.12: The `factorial` method, rendered using an alternative visualization, which highlights the control flow.

2.2 Working with a big project

We have designed Envision's visualization framework so that it can simultaneously display millions of visual objects (representing information from more than 100 000 lines of code) while still remaining responsive and permitting interactions. This is useful, for example, to view a map of an entire project, as illustrated in Figure 2.14. The figure shows the entire source code of the jEdit text editor – about 170 000 lines of code, all rendered at once. The map contains package, class, and method labels to make it easier to spot the different components. We can zoom in and pan the map to focus on a particular part of the code as shown in Figure 2.15. The map uses the standard presentation of the program and also allows the usual interactions for editing code when the view is sufficiently zoomed in.

The user can enable a mini-map in the lower left-corner. The mini-map shows the entire project; a red rectangle highlights the currently displayed part of the project, helping the developer to stay oriented. The mini-map can also be used for navigation – clicking on it will pan the screen to the corresponding location.

Envision offers the developer the possibility to use different views of the same code base simultaneously. From the code-map, shown in one view, a developer can find code fragments of interest and open them in another uncluttered view for inspection or editing.

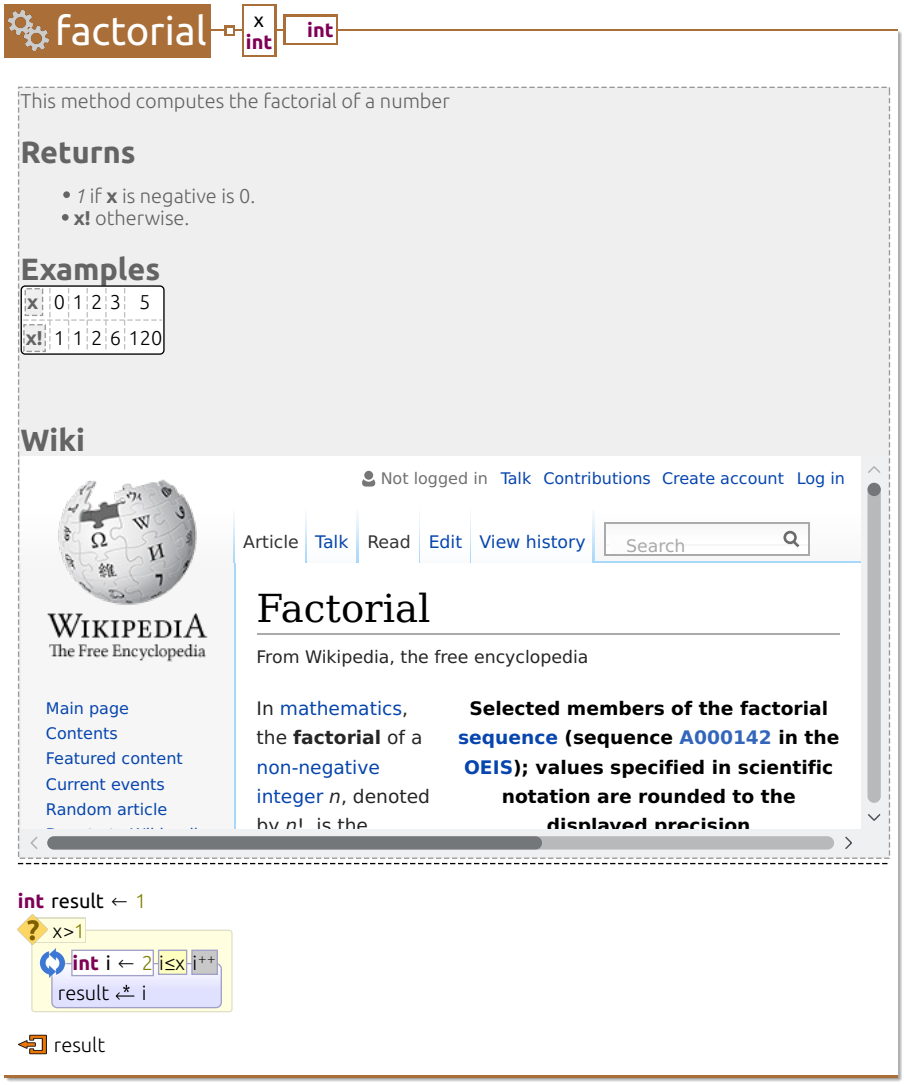


Figure 2.13: The `factorial` method with a rich comment header that includes rich text-formatting, a table, and an embedded browser.

2.3 A note on learnability

In our research, we have deprioritized learnability so that we can focus our efforts on exploring programming interfaces for power users that we assume are familiar with Envision's interface. Thus, the current version of Envision has a steep learning curve and its minimal user interface does not provide many aids to help its users discover various features. This can be compared to power tools such as the vim editor and the Unix command line. Implementing features to improve learnability and feature discovery is possible, but would require significant effort. For example, the system could offer users who are new to its interface an initial tutorial explaining how to

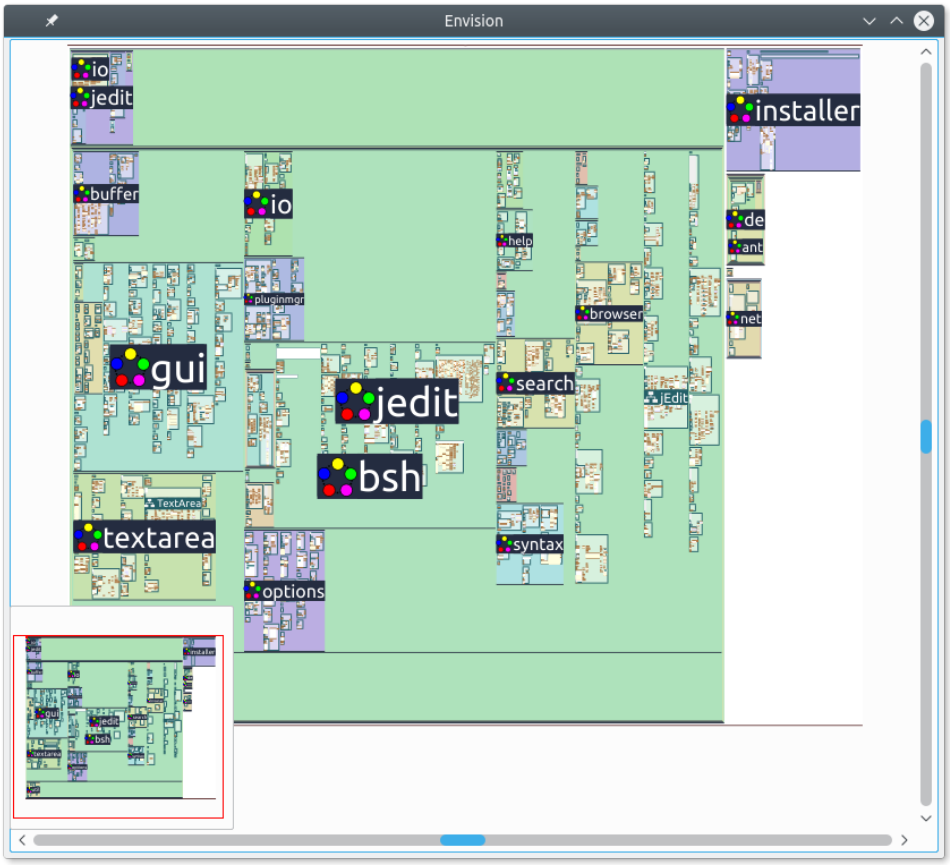


Figure 2.14: A map view of the entire source of the jEdit open-source text editor.

write programs and listing common shortcuts. More common functionality could be included in a traditional menu or toolbar.

Similarly, we assume that users of Envision would already know how to program in a supported language (currently, only Java). It would be interesting to explore how Envision's visual interface might be used to help teach programming to novices. For example, the tool could provide drag-and-drop interactions to construct programs from visual templates and it could help prevent mistakes by disallowing structurally incorrect code. Once again, we defer such explorations to future work.

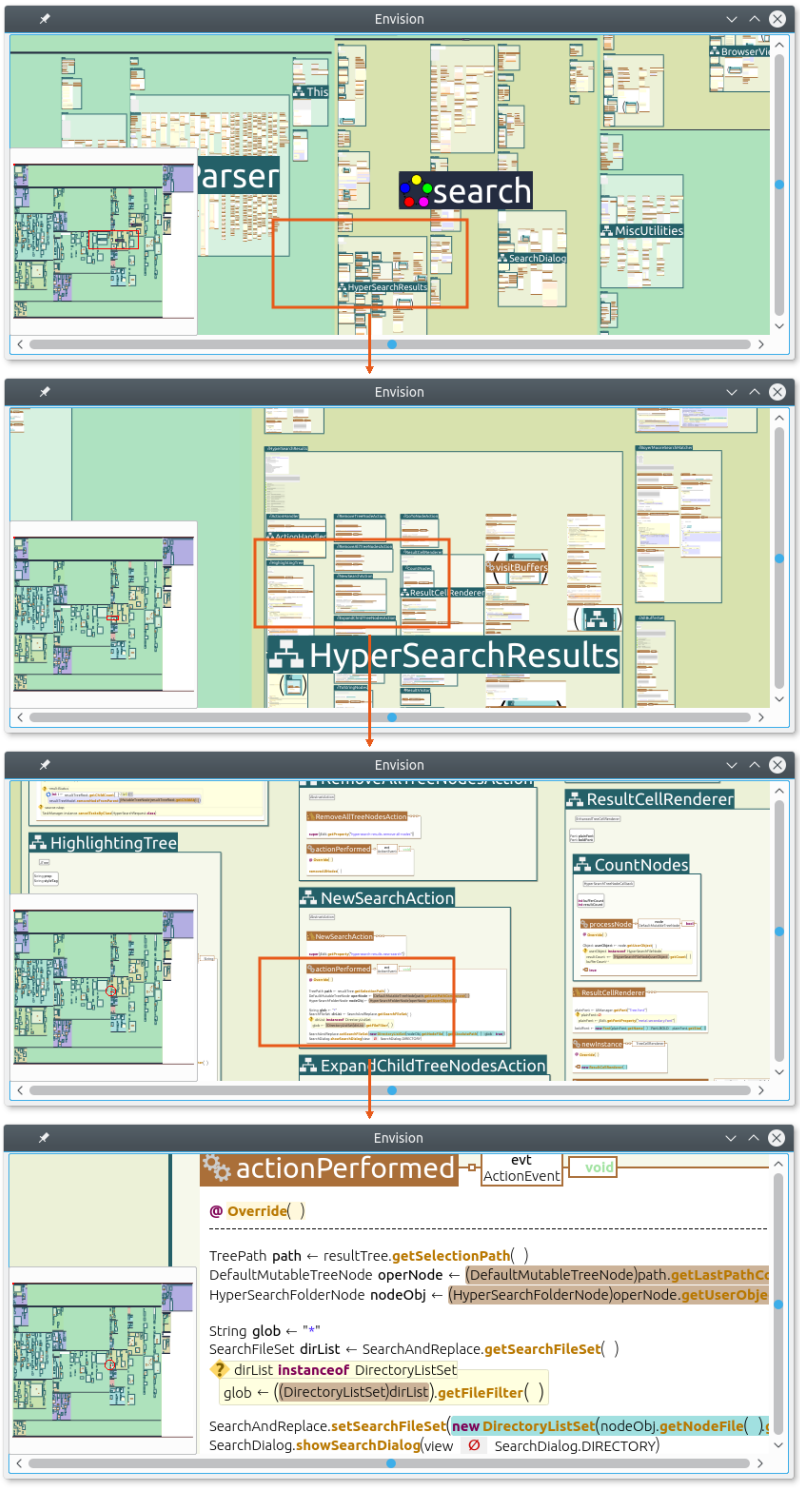


Figure 2.15: Different stages of zooming in on the `actionPerformed` method of the `NewSearchAction` inner class, part of the `HyperSearchResults` class, which belongs to the `search` package. The mini-map in the lower-left corner shows which portion of the project is currently displayed.

Design principles of Envision 3

In this chapter we outline the principles that have guided the development of Envision. They are organized in three sections corresponding to the high-level goals that motivate them. We define each principle, describe the motivation behind it, and point to its consequences.

3.1 Rich information and smarter tools

In order to improve the efficiency of software development, our approach advocates the use of enhanced information structures throughout all types of development tools. The following three principles promote a design facilitating this approach.

3.1.1 Decouple information structures from interfaces

Definition and examples

The information structures that define a program should be independent from the interfaces that people use to observe, modify, and create the structures. In particular, the amount and kind of information stored on disk or exchanged between tools should not be limited in any way by the operation of an interface.

For example, the amount of meta-data should not be restricted due to the inability of an interface to hide such information when it is not needed.

Motivation and consequences

The decoupling of information structures and interfaces is at the core of our approach. It is contrary to the established practice of writing, reading, and storing programs as text files. The direct benefit of this decoupling is that information structures can be expanded to include additional concepts and meta-data that facilitate the development process. For example, as we show in Chapter 7, adding unique identifiers to all AST nodes enables improved version control by allowing perfect tracking of code locations as opposed to approaches based on heuristics and approximations [Flu08, Rei08, FMB+14]. Others [KM06] have also recognized the utility of associating meta-data with code elements.

As a consequence of richer information structures, interfaces have to be adapted to enable developers to selectively see the information they need. This opens an exciting opportunity for the redesign of programming interfaces, since they are no longer bound to showing a particular textual encoding of the underlying structures. Thus,

interfaces may be designed using a wide variety of options to present information, for example, using graphics, text, multiple-dimensions, animations, and other, non-visual means. This variety makes it challenging to explore the design space for interfaces, which is needed in order to maximize their efficiency.

3.1.2 Support processing and integration of diverse types of information

Definition and examples

Software engineers work with many different types of information and tools should support developers by acting as information systems. This means enabling the storage, integration, querying, and manipulation of all types of information that are part of the software engineering process, not just executable program code.

For example, developers should be able to store design decisions and diagrams with a program or write a single information query that combines data from a dynamic analysis with data from the version control system.

Motivation and consequences

This principle is about taking an information perspective on software engineering and seeing the development process as a way to communicate and refine ideas, not simply as a means to make a computer perform a specific computation. When performing their daily tasks, developers ask a variety of questions that require diverse information [KDV07, SMDV08, LM10, FM10]. Characterizing the information needs of developers poses challenging questions, e.g.: what information is needed; how should information be presented; how can information be combined? Since there are no fixed constraints on developers' information needs, IDEs should be designed with general and extensible support for information processing. For example, information work can be facilitated by enabling a variety of visualizations or by providing a data-model that can easily be extended when the need arises. The development environment should not impose barriers to information by limiting its scope or preventing its collection.

3.1.3 Use and share rich information to make tools smarter

Definition and examples

An IDE should provide tools with easy access to rich information and enable tools that produce useful information to share it with other tools, to make it easier to build on top of existing functionality.

For example, the AST and type information should be easily accessible to a tool that performs type-checking; a tool that computes a call graph should make it possible for other tools to use the final call-graph information, say, to visualize it on the screen.

Motivation and consequences

The key goal of this principle is to make it easier to create smart development tools. *Smart tools* are those that make extensive use of a program's structure and semantics,

and of additional information, such as meta-data.

As a foundation for creating smart tools, basic access to the information they require needs to be provided. However, to make it easier to create such tools, they should have access to appropriate data models. For example, if a tool needs to perform semantic analysis of a code fragment, providing the tool with a raw textual encoding of the code is inefficient. Not only will the tool author need to develop a preprocessing step to parse and understand the encoding, but this computation will also happen independently of other similar efforts, wasting time. In this example, it would be much more convenient to create the analysis starting from an AST and type information provided directly by the IDE or another tool. Once the analysis is complete, it should be sharable with other tools and with the user who might want to also annotate the analysis. Such user annotations are an example of meta-data which should be preserved and processed by tools in subsequent tool runs, essentially sharing meta-data across different runs of the same tool. Sharing high-level information between tools in an IDE is also considered essential by Kuhn [Kuh12] who was forced to scrape the visual interface of the Call Hierarchies plug-in in Eclipse, because there was no other way to extract the information from the plug-in. We share Kuhn's view that IDEs should act as open data platforms in order to facilitate information exchange and make it easier to create tools like his Codemap. This could be achieved, for example, by defining a unified format for exchanging data between different tools and plug-ins.

3.2 Better programming interfaces for people

People are the second big focus point of our approach to building programming environments. Building tools for people means to carefully evaluate and craft interfaces that facilitate understanding and agency. The next three principles reflect this goal, and provide concrete guidelines for achieving it.

3.2.1 Design flexible interfaces that adapt to context

Definition and examples

For optimal efficiency, a developer should observe and manipulate information at a level of abstraction that matches their current needs. An interface should adapt to the current context, which includes what the current task is, who is working on the task, what the domain is, what APIs are used, etc.

For example, when debugging a state machine, it might be helpful to see a visual depiction of the transitions that took place, but when creating a new state machine the developer might prefer a tabular presentation.

Motivation and consequences

The context of a developer's task dictates the information and the amount of detail they need. The task and the programming language are two types of context, that are traditionally supported by IDEs, e.g., by the various perspectives in Eclipse such as Java or Debug. However, context has many facets that could be used to improve the efficiency of an interface, such what else is currently shown on the screen, or

what part of the code a fragment is located in. The same piece of information, say, a code fragment, may need to be presented differently in different contexts. For example, when exploring the public interface of a class, it is enough to see the signature of its methods, but when implementing the class, the developer needs to work with the full code. To support this variability, an IDE needs two features: a finer notion of context that enables the IDE to select an appropriate interface; and support for a diverse set of visualizations and interactions in order to enable context-specific interfaces. Supporting only textual notations, for example, severely limits the possibilities for visualizing information.

3.2.2 Leverage expert developers' existing skills with languages and using the keyboard

Definition and examples

In order to be practical for a wide range of professional developers an IDE should improve the programming experience for common programming paradigms, instead of imposing new ones, and should focus on keyboard-based input.

For example, the IDE should support popular object-oriented languages and enable developers to continue to type to create and edit programs.

Motivation and consequences

This principle is about meeting developers where they are and working with their existing skills, instead of designing a programming environment that imposes completely new workflows or habits.

First, our goal is to enhance the programming experience of a wide range of professional developers using richer information and better interfaces, not to make programmers express computation in a different way. This is in contrast to other programming tools with non-traditional interfaces such as LabView [Lab], which is built around a data-flow programming paradigm, or MPS [MPS], which is a language workbench, in which developers define and use their own domain-specific languages.

Second, the majority of expert developers type their programs and use command-line tools. It is not surprising that the keyboard is the central input device for experts – it offers much quicker input compared to other common input devices such as mice and enables developers to quickly translate their intentions to actions or precise code changes. This is why keyboard-based interactions are crucial for any interface that needs to be efficient and appeal to experts. This is in contrast to interfaces for beginners, such as the block environments offered by Scratch or Mit AppInventor, which offer palettes and drag-and-drop interactions to improve the discoverability of language or tool features at the expense of slower program construction.

3.2.3 Make better use of people's perceptual and cognitive abilities

Definition and examples

Programming interfaces should make use of well developed human abilities, for example, visual perception or spatial cognition.

Motivation and consequences

To communicate information more efficiently, interfaces should make use of the full range of human abilities. Unlike mainstream programming tools, which focus heavily on the textual presentation, programming tools could use richer presentations that offer a mix of text and graphics in order to make more thorough use of people's visual perception capabilities. Such presentations could also include code and information maps that utilize people's spatial cognition.

A consequence of this principle is that a development environment should provide a framework that enables diverse ways to interact with people. Thus, interfaces should not be rooted in a particular presentation (e.g., text), but rather be based on the structure of the underlying data and offer flexibility of form. In our work we explore only rich visual interfaces, but other modalities are also possible, for example, tactile or aural for visually impaired developers.

3.3 Support for general-purpose languages and large-scale projects

The last group of principles stems directly from the target audience of our research and their needs. First, we are targeting professional developers who spend most of their time using an IDE. They require a flexible tool that can be customized to suit their particular needs. Second, we are not targeting professionals from any particular domain, but are interested in designing tools for general-purpose programming. Third, our tool should support large-scale projects, and not only toy examples, which means that the interfaces and performance should scale well.

3.3.1 Design a tool that can be used conveniently for extended periods of time

Definition and examples

Programming interfaces should allow developers to work in a state of flow and be productive. Cumbersome or unintuitive interfaces and unnecessary distractions should be avoided.

For example, rarely used interface elements should not be shown by default.

Motivation and consequences

This principle is about being mindful of the usability of a tool and how the usability is affected by the design of a particular interface or interaction. Poor usability is bad for any tool, but professionals can be especially demanding when it comes to their code editor. The lack of a particular feature, cumbersome interactions, or too much bloat can lead to the failure of an otherwise great tool.

Professionals use a development environment during most of their day, and they need to be comfortable with it. As a crude check for usability we have found it helpful to continuously ask the following question:

> *Do we imagine that professional developers or even we can create software using such an environment every day, all the time, and be comfortable and efficient?*

In a number of cases, we have rejected a particular interface because it failed to pass even this basic test. For example, an initial prototype of Envision's expression editor, while technically advanced and elegant, did not allow text-like input and was scrapped because we were not comfortable with its usability.

3.3.2 Enable a high degree of customization and flexibility

Definition and examples

A tool should support customization and extensibility at every level of its design, in order to satisfy the needs of professionals and to facilitate research explorations. Particular features and design choices that might be affected by strong personal preferences, or which need scientific exploration, should be easily configurable.

For example, color schemes or icons should not be hard-coded, but it should be possible for users to pick their own, and for researchers to experiment with different schemes.

Motivation and consequences

Professional developers require tools that can be molded to fit their own preferences and work environment. Unlike most professional development tools, research prototypes are rarely customizable, skipping over the challenges of providing customizability, but also losing the benefits it provides. Researchers can also benefit from highly customizable tools, which help to more easily implement and evaluate new ideas. This principle is about supporting both professionals and researchers with flexible tools. Making tools flexible means making existing features customizable and adding support for making new extensions.

To facilitate extensibility, there should be clear boundaries between the different components of the environment and its design should be modular. A key boundary in Envision is between the model of a program and its presentation, which helps us extend each of the two independently of the other. Existing components should be configurable without recompiling the tool, wherever possible. To facilitate this, the environment can provide a standard framework that handles the processing of configuration files and user preferences.

3.3.3 Support a wide range of project domains and sizes

Definition and examples

In order to be applicable in a wide range of contexts, a programming environment should support programming in general-purpose languages and allow both small and large software projects.

For example, the IDE design should not be limited to a particular programming language or use case.

Motivation and consequences

Our aim is to improve the tools of mainstream developers, regardless of the particularities of their projects. The development environment can permit extensions that provide specializations for a particular domain, programming language, or

company-specific processes, but should not be limited to them or to specific kinds of programming tasks.

In particular, one important aspect of a software project is its size. Our aim is to support projects of millions of lines of code and associated information. This poses two challenges: On the one hand, interfaces need to provide meaningful information and interactions at large scale. For example, simply writing a million lines of code with a tiny font on the screen is unhelpful, but presenting a visual summary of the major components of the entire project can be useful. On the other hand, there is the technical challenge of making interfaces perform well with huge code-bases. This requires regular performance evaluations, identifying bottlenecks, and developing scalability solutions, such as partial and lazy loading of program models. Performance issues could also be caused by third-party libraries and be challenging to work around.

The architecture of Envision 4

In this chapter, we provide more details about Envision's architecture and important system design aspects. We show what mechanisms form the foundation of the tool's flexibility, which makes it suitable for exploring novel programming interfaces and rich information integration. We also discuss performance-related details and provide guidelines for achieving good performance in complex visualizations.

4.1 Overview

Envision is implemented in C++ and is a stand-alone IDE, rather than an extension for another IDE or code editor. Because existing IDEs and code editors for general-purpose languages utilize text-based editors, Envision does not reuse any of their interface components and instead has a visual editing interface implemented from scratch using low-level drawing routines provided by the Qt Graphics View framework [QGV]. All text-like editing interactions in Envision's interface, including the cursor's behavior, are also implemented from scratch in order to better integrate with Envision's visually-rich presentations of code. At present, Envision's implementation spans more than 160 000 lines of C++ code in over 950 classes.

Envision is built using a modular plug-in architecture — virtually all of Envision's functionality is implemented via plug-ins, each of which is a dynamically loadable library that is automatically discovered and loaded at run-time. We logically separate the different plug-ins into layers: plug-ins may only use services from their own layer and layers underneath. The layers of Envision's architecture and the currently implemented plug-ins are shown in Figure 4.1. Next, we provide a brief description of each layer and some of the plug-ins contained within.

At the bottom of all the layers is the **Initialization** layer. It contains the *Launcher*, which is the executable that starts Envision and the *Core* library, which provides a few basic utility functions and manages plug-in loading and unloading. The main task of this layer is to scan for available plug-ins and load them dynamically. Practically all of the functionality of Envision is implemented by plug-ins from higher layers.

The next layer is **Development Support**. It includes two plug-ins that provide services for logging warnings and errors and for performing unit-tests. These are primarily useful during the development of Envision itself.

The functionality in the first two layers is completely generic and does not yet have features specific to a tool for software development. The **IDE Base** layer establishes the foundations of Envision's development features and interface. It is the first layer with IDE-specific functionality. Plug-ins from this layer are completely

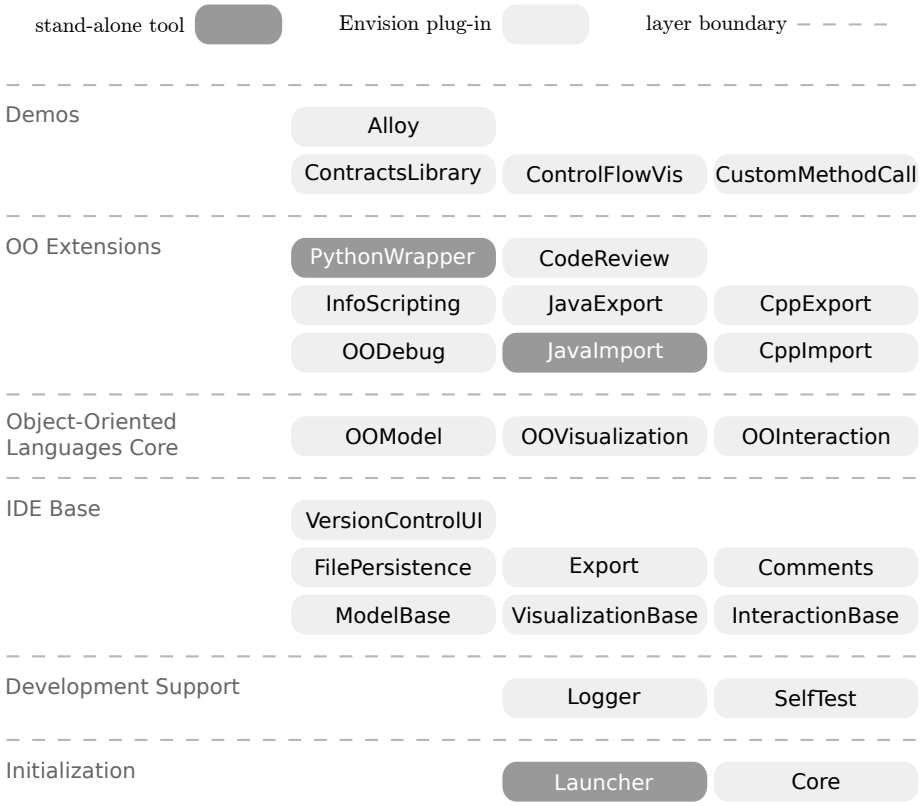


Figure 4.1: The layers of Envision's architecture including all current plug-ins and stand-alone tools.

independent of any programming language or programming paradigm and define generic versions of basic programming concepts such as literals or code versions. The three most important plug-ins in this layer implement the different parts of a Model-View-Controller (MVC) framework that is the foundation of Envision. The *ModelBase* plug-in implements generic functionality for specifying a program tree. A program tree is similar to an AST, but supports rich information and non-executable tree nodes. The *VisualizationBase* plug-in implements Envision's visualization engine using Qt's Graphics View framework [QGV]. The *InteractionBase* plug-in implements the generic interactions available in the system, such as the selection cursor and the command prompt. The remaining plug-ins from this layer provide support for: rich comments, exporting a program tree as traditional programming language text, saving and loading program trees directly in Envision's custom file format, and version control for program trees (Chapter 7). This layer provides services that higher layers can use for supporting any programming language or paradigm. Our work has focused exclusively on object-oriented (OO) languages, because they are prevalent in the mainstream. However, it is possible to support other types of programming such as scripts, data-flow programming, or logic programming.

Next comes the **Object-Oriented Languages Core** layer. It features three plug-ins which are the OO extensions of the generic MVC implementations: *OOModel*,

OOVisualization, and *OOInteraction*. These plug-ins define standard AST nodes for OO programs (e.g., classes, methods, and expressions), their default visualizations, and interactions. It is worth pointing out that each of these plug-ins provides a dedicated set of services and the plug-ins do not have to be used in an all-or-nothing fashion. For example, it is possible to load only the OOModel plug-in, which will enable the loading and semantic analysis of programs, but not their OO-specific visualizations or convenient manipulation by the user. If also the OOVisualization plug-in is loaded, but not OOInteraction, then a program can be visualized using dedicated OO presentations, but there will be no OO-specific interactions. Such a configuration may be used for exploring code, but will not permit efficient code manipulation. Of course, the full functionality of the OO-specific interfaces will be available if all three plug-ins are loaded. This clear separation of concerns gives us additional flexibility for defining the code model independently of the way it is visualized and the way it is manipulated.

The **OO Extensions** layer builds on top of the core OO functionality and provides a wide variety of features. There are plug-ins for debugging, code reviews, accessing and combining information from diverse sources (Chapter 8), and for importing and exporting Java and C++ code to and from Envision's program tree. To import textual source code into Envision, we use the parser of Eclipse's Java compiler [JDT] for Java sources, and the clang parser [Cla] for C++ sources.

Finally, there is the **Demos** layer, which features four projects that illustrate the flexibility of Envision's interfaces. *ControlFlowVis* is a plug-in that implements the alternative control-flow visualization of method bodies that we showed in Section 2.1. The control-flow visualization works for any method and can be manually selected by the user. *CustomMethodCall* is a plug-in that implements custom visualizations for method calls to four specific methods, illustrating how code-presentation can be customized based on the context of the code – in this case the target of a method call. The customized visualizations are configured to be used automatically by the IDE whenever they are applicable. The *ContractsLibrary* provides interface customizations for efficiently working with the APIs of Microsoft's Code Contracts. We provide more detail on this use case in Section 5.4. The *Alloy* plug-in builds on top of the ContractsLibrary plug-in. It extracts code contracts written using the Code Contracts API and uses MIT's Alloy tool to visualize data structure models that satisfy the contracts.

Envision's architecture promotes a clear separation of concerns and facilitates extensibility. The separation into layers and plug-ins makes it easy to identify where a new feature should be implemented. The overhead of implementing new features is reduced by the reuse of existing building blocks, plug-in services, and customization concepts. For example, throughout Envision, we use a declarative API to create new visualizations by combining existing ones.

Envision's design has also been beneficial for us as researchers. It has allowed us to quickly experiment with a variety of interfaces and information processing techniques. For example, when creating a new visualization, we can define many of its properties such as colors, shapes, distances, font properties, icons, etc. using the built-in styles API, defined in VisualizationBase. Using the API, property values are automatically read from an XML file on disk and can be changed without recompiling the program. Next, we highlight more of Envision's customization and extensibility features that make it a good research vehicle and enable personalized customization by users.

4.2 Extensibility and customization

Support for extensibility and customization permeates the entire design of Envision and is evident on several levels.

At the architecture level, the fundamental extension mechanism is the modular plug-in system, which allows for unlimited functionality to be added to Envision. Users of Envision can choose what set of plug-ins should be loaded in order to customize what features are available.

At the plug-in level, extensibility and customization are facilitated in two primary ways. First, plug-ins provide a large amount of functionality that can be readily reused by others. This often includes generic and parametric components that are specifically designed to be used in extensions. Second, many plug-ins' operations may be extended to new contexts, using dedicated customization APIs and extensible class hierarchies. For example, the generic reference-to-declaration binding mechanism can be configured to work with different programming languages and type systems.

At the configuration level, many plug-ins rely on external files that can be easily modified or provided by the user to customize Envision without compiling code. This ranges from XML files that define visual styles, to the configuration of keyboard shortcuts, to Python scripts, e.g., as we show in detail in Chapter 8. For example, by modifying visual style files, it is possible to achieve significant variation in Envision's interfaces, changing colors, icons, visual arrangement, and even behavior. This allowed us, for example, to easily create the two alternative visual styles of Envision shown in Figure 4.2, which we used for the user study described in Section 5.3.

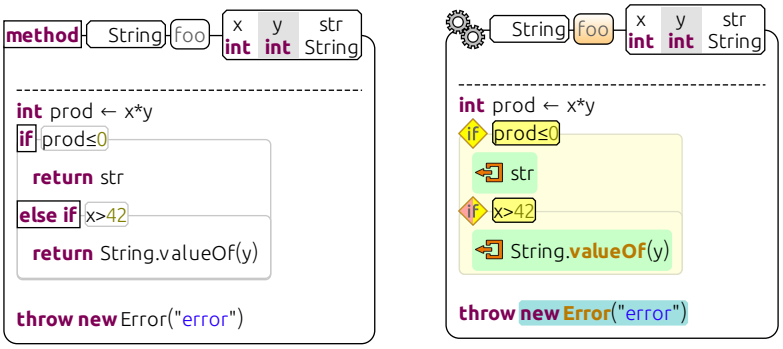


Figure 4.2: Two alternative visual styles for displaying methods in Envision.

Finally, some plug-ins also allow customization by including annotations in code in projects that are developed in Envision. This enables library designers to tweak specific IDE aspects for APIs of their libraries. For example, in Section 5.4 we use annotations in a library to make the APIs it provides look like native language features.

The customization and extension mechanisms in Envision cross-cut all aspects of the MVC framework, in order to enable extension to new types of information and support for dedicated interfaces. For example:

- The ModelBase plug-in implements a framework for defining types of nodes for a program's tree. It uses the framework to define some basic AST node

types such as text literals or lists, but other plug-ins can contribute additional types, e.g., `Class`, contributed by OOModel.

- The VisualizationBase plug-in enables existing visualizations to be extended with new snippets of information. This can be done either inline using the concept of visualization slots (a pre-determined visual space associated with any visual item, where other plug-ins can inject additional visual objects) or using visual overlays, which are independent of the underlying visualization. We use the former in Section 5.4 to inject a visualization of a method's contracts in its signature, and the latter in Section 8.4.3 to display information associated with a statement.
- The InteractionBase plug-in offers a general hook for processing keyboard and mouse events from the user, which enables other plug-ins to register custom handlers for these events. InteractionBase also enables others to contribute new commands for the command-prompt, or fine-tune the behavior of the cursor in specific interfaces.

Throughout Chapters 5, 6, 7, and 8 we will demonstrate many specific use cases of customizations in Envision. The effort to create such customizations and the intended customization designer depend on the complexity and type of a customization:

- **Modifying a style file** can be done by anyone, including ordinary users of Envision, but is limited to adjusting predefined parameters of visualizations and interactions. Such customizations are very simple as style files are mostly self-explanatory and typically do not require any knowledge about the inner workings of Envision.
- **Adding annotations to code** is intended for library designers who want to change certain predefined aspects of the look and feel of library APIs. Library designers need to know what customization annotations are available, but using them is as simple as annotating the library code.
- **Writing an external script** is intended for power users of Envision. Scripting offers a lot of flexibility, but script designers would need to be familiar with Envision's scripting APIs and would have to invest time creating and testing the scripts.
- **Writing a plug-in** for Envision is intended for power users and plug-in designers. This method offers unlimited extension capabilities, but requires the most effort and knowledge. Plug-in designers need to be familiar with Envision's C++ API and the APIs of other plug-ins they want to use and would need to use an environment configured for compiling the Envision codebase. To make creating new plug-ins easier, we support plug-in writers in two ways: First, we offer a plug-in generator that, in a few seconds, creates a functional generic plug-in ready to be specialized. Second, we provide many simple APIs designed specifically to make customization easier. To illustrate this point, next, we present two customization examples in more detail.

Adding attributes to program nodes

The program model defined by the ModelBase plug-in allows other plug-ins to not only contribute new node types, but also enhance existing types with new attributes

on the fly. For example, the OOVisualization plug-in adds two optional `x`- and `y`-position attributes to the `Class` and `Method` types, allowing classes and methods to be visually positioned on a two-dimensional grid, which itself is implemented by the VisualizationBase plug-in. We make this process as straight-forward as possible:

First one must define the `Position` extension as shown in Figure 4.3. This

```
// In Position.h
class Position
{
  DECLARE_EXTENSION(Position)

  EXTENSION_ATTRIBUTE_VALUE(Model::Integer, x, setX, int)
  EXTENSION_ATTRIBUTE_VALUE(Model::Integer, y, setY, int)
};

// In Position.cpp
DEFINE_EXTENSION(Position)
DEFINE_EXTENSION_ATTRIBUTE(Position,x,Integer,false,false,true)
DEFINE_EXTENSION_ATTRIBUTE(Position,y,Integer,false,false,true)
```

Figure 4.3: The definition of a `Position` class that lists two new integer attributes, which can be attached to arbitrary node types from the program model. The different macros are used to declare the name and type of the new attributes and specify whether they are partially loadable, optional, and persisted to disk.

definition consists of the usual split between header and source files in C++, but is otherwise minimal thanks to the use of custom macros that help to quickly declare the desired attributes.

After the `Position` class is declared, any plug-in can register this extension to any existing node type, as shown in Figure 4.4.

```
Class::registerNewExtension<Position>();
Method::registerNewExtension<Position>();
```

Figure 4.4: The registration of the `Position` class as an extension to the existing `Class` and `Method` nodes. Registration code can appear in any plug-in.

Finally, any code that is aware of the extension can use it as shown in Figure 4.5.

```
auto pos = someClass->extension<Position>();
pos->setY(42);
assert( pos->y() == 42 );
```

Figure 4.5: An example usage of the newly registered extension.

Declaring new visualizations

The VisualizationBase plug-in defines a framework for creating visualizations by using a declarative C++ API. Plug-ins are free to create visualizations with or without using this framework, but its use simplifies development significantly. Thus, this framework is used by many other plug-ins, e.g., by OOVisualizations, in order to create new interfaces. An example of declaring the visual layout for the components of a for-each loop is shown in Figure 4.6.

4.3 Performance and scalability

In order to support work on large software projects, we have designed dedicated performance features such as a dead-lock free concurrent access scheme for the program model or a variety of rendering optimizations. As many techniques for scaling to large projects are known from existing IDEs (e.g., incremental compilation), in the discussion below we focus our attention on Envision's most performance critical novel aspect – the rich visual presentation of programs.

Envision's code visualizations scale well to large programs with millions of visual objects on the screen. For example, the user could load a code-map like the one from Section 2.2, and smoothly zoom-in and out and navigate around the entire code base – a visual rendering of 170 000 lines of Java code in that example. We have also experimented with bigger code bases, and the interactions remain fluid, at the expense of increased RAM usage. We achieve scalability thanks to Envision's performance-sensitive design and optimizations. Some of these optimizations are inspired by video game rendering techniques; others, such as reducing the number of memory allocations, are made possible by the technology we use to implement Envision. Below we formulate the most important performance aspects as guidelines for tool designers.

Performance guidelines

Structure visualizations in a visual tree. Envision renders a large amount of two-dimensional visual objects using drawing functionality from the Qt library. Qt itself uses collision detection algorithms for a number of tasks related to rendering and input handling, for example, to determine which objects are under the mouse cursor or which objects are currently not visible on the screen and thus, do not need to be repainted. Although visual objects in Qt are logically organized as a tree, by default the bounding box of each object depends only on what the object draws itself and disregards children. Thus, to calculate collisions, Qt iterates over all objects, which is slow. Collision detection can be much faster if the bounding box of each item completely encompasses all of its children, because this enables efficient spatial partitioning of the visual tree. Qt has support for restricting the visual extent of child items to their parents, but enforcing this restriction incurs a significant overhead due the use of clipping operations. We contributed a new feature to Qt to skip this enforcement, while still assuming that children are fully contained in their parents. The enforcement is not necessary in Envision, because all visualizations are within their parent's bound by construction. This resulted in significant performance benefits when drawing a large number of objects at once. Note that this optimization

```
void VForEachStatement::initializeForms()
{
  // The loops's header is a linear sequence of elements:
  // An icon, the loop variable's type and name, and the
  // collection to iterate over.
  auto header = (new GridLayoutFormElement{})
    ->setHorizontalSpacing(3)->setColumnStretchFactor(3, 1)
    ->setVerticalAlignment(LayoutStyle::Alignment::Center)
    ->put(0, 0, item<Static>(&I::icon_,
      [](I* v) {return &v->style()->icon();}))
    ->put(1, 0, item<NodeWrapper>(&I::varType_,
      [](I* v){return v->node()->varType();},
      [](I* v){return &v->style()->varType();}))
    ->put(2, 0, item<VText>(&I::varName_,
      [](I* v){return v->node()->varNameNode();},
      [](I* v){return &v->style()->varName();}))
    ->put(3, 0, item<NodeWrapper>(&I::collection_,
      [](I* v){return v->node()->collection();},
      [](I* v){return &v->style()->collection();}));

  auto body = (new GridLayoutFormElement{})
    ->setNoBoundaryCursors([](Item*){return true;})
    ->setNoInnerCursors([](Item*){return true;})
    ->setColumnStretchFactor(0, 1)
    ->put(0, 0, item(&I::body_,
      [](I* v){return v->node()->body();}));

  // The shape is the visual background of the loop.
  auto shapeElement = new ShapeFormElement{};

  // The header, body and shape are arranged using a
  // a layout based on position and size constraints.
  addForm((new AnchorLayoutFormElement{})
    ->put(TheTopOf, body, 3, FromBottomOf, header)
    ->put(TheTopOf, shapeElement, AtCenterOf, header)
    ->put(TheLeftOf, shapeElement, -10, FromLeftOf, header)
    ->put(TheLeftOf, shapeElement, 5, FromLeftOf, body)
    ->put(TheRightOf, header, AtRightOf, body)
    ->put(TheRightOf, shapeElement, 3, FromRightOf, header)
    ->put(TheBottomOf, shapeElement, 3, FromBottomOf, body)
    ->put(TheRightOf, shapeElement, 3, FromRightOf, body));
}
```

Figure 4.6: The complete definition of the visual layout of the components that form the presentation of a for-each loop. This code defines how elements are arranged, but their look is loaded from styles, which are either explicitly selected or automatically determined by the system.

does not restrict the flexibility of Envision's visualizations, since peer objects are allowed to overlap.

Draw only what is necessary. Qt already provides support for culling off-screen objects by using collision detection to decide what is not visible. However, when observing a canvas with many visual objects at a high zoom level, most of the objects are within the visible region and therefore drawn. This is especially of concern to Envision, since the mini-map that we provide in the tool (see Section 2.2) is actually just a second rendering of the same canvas, where all objects are always visible. With millions of objects, the standard Qt approach was too slow and unusable. At a very high zoom level, many visual objects are extremely small and practically invisible, but were nevertheless drawn. We made a further contribution to Qt's drawing routines – an option to skip the drawing of any visual item whose size in any dimension is less than a configurable value, for example, half a pixel. Note that this also automatically skips the drawing of all child items. This results in a significant speedup when drawing a large number of objects, like in the mini-map.

Together the first two optimizations effectively make the rendering performance independent of the size of the visual canvas and the number of objects. Only objects that are truly visible on screen need to be rendered and the amount of such objects is influenced by other factors such as screen resolution.

Cache text renderings. At medium zoom levels, thousands of objects could be within the visible region of the screen and not small enough to be discarded by the optimizations discussed so far. This is especially a problem for textual elements (labels, expressions, etc.) as drawing so much anti-aliased text at once can be slow. We used the functionality of Qt's `QStaticText` class to speed up text drawing by caching results of drawing operations. In this way, text has to be redrawn only when it is updated or when the zoom level changes. This resulted in a noticeable speed-up, without a significant memory cost.

Decouple updates from rendering. Modern graphics frameworks minimize the number of painting operations, but painting still occurs quite often, for example, when zooming or panning. In many of these cases, all or almost all objects on the visual canvas remain unchanged and do not need to be updated. For achieving good performance, it is essential that updates are decoupled from the regular painting operations, and are only performed when necessary. When the user modifies a program, only visualizations that are affected should be updated. When using a tree organization for visualizations, an edit typically affects only a particular visualization and all of its ancestors in the tree.

Avoid recreating objects. Constant allocation and deallocation of memory can significantly slow down a program. We experienced this a number of times in Envision and had to implement optimizations that keep objects around and reuse them if possible. This was especially useful for loading large programs from disk — using manual memory management enabled by C++, we were able to reduce loading times by half. We also preserve visual objects with most updates.

Use mature, open-source graphics frameworks designed for performance. Envision is based on the Qt framework, which features a mature rendering engine. It has a large community of users and is updated frequently with new features and optimizations. This choice has had a big impact on Envision's performance. The fact that Qt is open-source was also crucial for developing Envision – before our custom patches, which implemented the first two optimizations mentioned above, rendering a mini-map or huge code maps was not possible with bigger code bases.

Rich and customizable zprogram presentations

5

In this chapter we present and evaluate Envision's visually rich program interfaces that can be customized for different contexts.

First, we outline key flexibility concepts of our approach and show how they shape the design of Envision's visualization engine. In contrast to many existing approaches to code visualizations or visually enhanced IDEs [BZR+10, DR10], Envision does not simply use rich visualizations to complement or decorate textually rendered code. Instead, rich visualizations are at the core of our code editor, allowing arbitrary compositions of textual and graphical elements to be used at all levels of code presentation: from literals and expressions, to methods and classes, to entire software projects. Our approach also supports customizing visual notations and interactions based on different types of context such as what is currently shown on the screen or what domain a particular part of the code pertains to. Interface customizations can be helpful, especially in large software projects, which use a variety of libraries that: provide APIs for particular domains such as writing database queries, staging, constraint solving or define an embedded domain-specific language (eDSL). Our approach enables presentations of APIs and eDSLs to be customized in order to improve comprehension.

Second, we discuss the current design of Envision's presentations of programing constructs. We ground the design in findings from cognitive psychology and visual perception theories and show how the design has evolved over time.

Third, we show two different alternatives for visually rich code presentations enabled by Envision's design and compare them to mainstream syntax highlighting in a user study [AHM16]. Syntax highlighting is the main visual lens through which developers perceive their code today, and yet its effects and the effects of richer code presentations on code comprehension have not been evaluated systematically. Our study helps to fill this knowledge gap; the results of the study show that Envision's richer code visualizations reduce the time necessary to answer questions about code structure, and that contrary to the subjective perception of developers, richer code visualizations do not lead to visual overload. Based on our results we outline practical recommendations for tool designers.

Finally, we explore how Envision's support for customizations can be applied to improve comprehension of code in a specific domain. We demonstrate the benefits of our approach by customizing Envision for the .NET Code Contracts API [AM13]. Our case study shows in particular that we can customize many aspects of visualization and interaction with little effort.

5.1 The visualization framework of Envision

Guided by the principles we outlined in Chapter 3, in this section, we define key concepts for enabling flexible visualizations in a code editor and use them to motivate and describe Envision's visualization engine.

5.1.1 Key concepts of flexible visualizations

The following concepts enable visually rich code presentations and the customization of the look and feel of interfaces in a programming environment:

1. Decouple the storage format from visualization and interaction. As we described in Chapter 1 the strong coupling of source files to text editors greatly restricts flexibility. A prerequisite for flexible visualizations is, thus, to decouple the storage format of a program from the representation that is used for rendering and editing. Editors should operate on a program model or an AST, even when the visualization and interactions are textual.

2. Provide basic building blocks for visualizations and use them to create reasonable default presentations that others can build on. Once a program is not viewed just as text, there are many options for visualizing it. Editors must provide a toolbox for easily creating visualizations from basic elements such as text, shapes, outlines, images, icons, layouts, etc. These elements should be used to create suitable default visualizations that serve as a stepping stone and facilitate further customizations. A first choice might be to just represent program elements with the text they are usually associated with. Graphical and hybrid notations can also be used.

3. Show only information necessary for the current task. Programmers have to deal with a lot of information in terms of source code, documentation, tool and program output, etc. To avoid information overload programming tools should display only what is needed for the current task and remove distractions. For example, a developer that is debugging a method might want to see a slice that contains the relevant statements. Everything which is not part of the slice could be completely hidden. The visualization mechanism needs to be flexible and allow only partial visualization of program fragments or constructs.

4. Choose visualizations based on context. In a customizable editor, program fragments and language constructs can be visualized in different ways. The kind of construct being rendered should not be the sole determinant of the visualization to use. Customization designers should have the freedom to define suitable visualizations based on the following additional factors:

- **Construct instance:** What is the specific instance or value of the programming construct? For example, a method call could be rendered differently based on its target method; a string constant that represents a URL might be rendered as a hyperlink, permitting additional interactions.
- **AST context:** Where in the structure of the program is this construct? For example, a stand-alone list can be rendered as a sequence, whereas if the list is nested in another list, both can be rendered as a matrix.
- **Visualization context:** What visualizations will appear alongside the construct that is being rendered? For example, when rendering a sequence of variable

assignments, their corresponding parts could be visually aligned.

- **Visualization purpose:** Why is this construct being rendered, what is the purpose of the current task? For example, when debugging, variable references could show the current value of the variable in addition to the variable's name.

- **Personal preferences:** Has the developer requested a particular visualization? For example, they might prefer to represent regular expressions as automata instead of text.

5. Allow customization of interactions and make each visualization interactive. It is essential that customization designers are able to define interactions for the visualizations they create such as new shortcuts, options, context-menus, commands, etc. Providing merely "read-only" visualizations might help with program comprehension, but not with manipulation; programmers would need to switch to the default representation of the host language for editing, which is cumbersome and reduces the benefits of customizations. Existing visualizations may also benefit from new, customized interactions. For example, one could create a new interaction for string literals that simplifies the input of file paths by overriding the TAB key to perform a name match, like a command terminal.

6. Make creating simple customizations easy, facilitate composition, and enable advanced customizations. The editor should permit customization designers to develop visualizations and interactions, and ship them together with their libraries. These customizations should go beyond simple style files for syntax highlighting and not require detailed knowledge of the editor implementation. For instance, designers should be able to quickly implement common visualizations such as text, boxes, icons, and lists, and create new visualizations by composing existing ones. The interactions of new compositions should automatically emerge as the aggregation of the interactions of their parts. This will greatly reduce the efforts required to implement customizations. Moreover, developers who are well familiar with the APIs provided by the editor should be empowered to create advanced customizations.

We designed Envision's visualization system by adhering to these concepts. Next, we provide an overview of how single visualizations can be added to Envision and how the system uses them to show code.

5.1.2 Creating and customizing visualizations in Envision

Program fragments (AST nodes) in Envision can be visualized in arbitrary ways. The default is to use text for low-level constructs such as expressions, and graphical notations for top-level constructs such as classes and methods. A visualization is a C++ class and new visualizations can be added to Envision by plug-ins, which we discussed in Section 4.2. Registering a new visualization in the system includes specifying the context in which it is applicable. All factors specified in concept 4 from Section 5.1.1 can be used to determine the context. When rendering an AST node, Envision chooses a visualization by scoring all visualizations applicable in the current context and picking the best one.

Figure 5.1 shows all factors that determine what visualization is used to render an AST node. Visualizations are associated with a specific node type and are chosen from the **visualization pool**. One node type may have multiple alternative visualizations. In addition to the node type, a visualization might also depend on

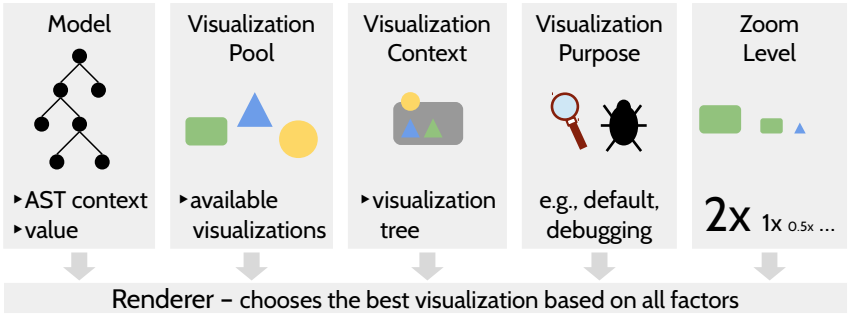


Figure 5.1: Factors that determine what visualization is used for an AST node.

the current context, which has many facets. First, the entire program model can be used to determine the **AST context**. For example, inner classes may be rendered differently from top-level ones, and calls to special methods can use a dedicated style that differentiates them from standard method calls. Second, the **visualization context** is determined by all other visualizations that are currently on the visual canvas (e.g., parent visualization). For example, if a method is rendered within its enclosing class, only the method's simple name could shown, but if the method is rendered on its own, a fully qualified name might be rendered. Third, the context of a visualization includes the **visualization purpose**. A visualization purpose is simply a string that describes what the developer is currently doing or what they want to see, e.g., "default", "debug", or "control-flow". The purpose can be set globally for all visualizations, or locally for individual code fragments. The user can manually choose a purpose, or it can be set automatically when the user performs certain actions (e.g., start the debugger). Plug-ins can register new types of purpose and provide corresponding visualizations. Fourth, a visualization's context also includes the current **zoom level**, which enables visualizations to be dynamically adjusted when the user zooms in and out. All of this information is fed into the **renderer**, which decides what visualizations should be used by scoring all visualizations applicable in the current context and picking the best one. For a given programming construct and context, only a single visualization will be used and there is no support for composing multiple available visualizations.

Each visualization has a handler that defines its interactions. Like visualizations, handlers are also C++ classes and can be reused and extended, making it easier to create new visualizations. For example, in Chapter 6 we describe Envision's selection cursor and expression parsing functionality, which can be used by any visualization. We use them to enable the manipulation of expression AST nodes using the keyboard, like a standard text editor, and customization designers can use the same mechanism to enable text-like editing for their visualizations. We describe handlers and Envision's interaction components in detail in Chapter 6.

To take advantage of Envision's customization mechanisms for visualizations, designers have two options. First, predefined properties of visualizations are easy to tweak by any user by modifying style files or inserting annotations in code as we show in Section 5.4. However, this approach is limited. Second, power users of Envision and language and library designers may define advanced customizations in a separate plug-in. This approach is very flexible, but requires familiarity with

Envision's visualization framework and APIs. However, based on our own experience and that of students using and improving Envision's implementation, we find that once a programmer is familiar with Envision's visualization framework, they require little effort to implement new visualizations and interactions. For example, when composing visualizations, the system automatically provides a cursor that enables keyboard-based navigation and selection. Typically this desired behavior requires no extra code in new visualizations, therefore reducing their implementation effort, in line with concept 2.

Next, we show how visualizations in Envision can be composed to create a rich code presentation on screen.

5.1.3 Composing and rendering visualizations

Envision's visualization engine builds on three key components of Qt's graphics view framework [QGV] for composing and rendering visualizations: Items, Scenes, and Views. The relation between the three components is illustrated in Figure 5.2 and we explain them in more detail below.

Scenes and items

A *scene* is an off-screen staging area that consists of arbitrary items (visualizations), which are composed in a tree hierarchy. A scene corresponds to a large two-

[1]Icons made by Freepik from www.flaticon.com are licensed by CC BY 3.0.

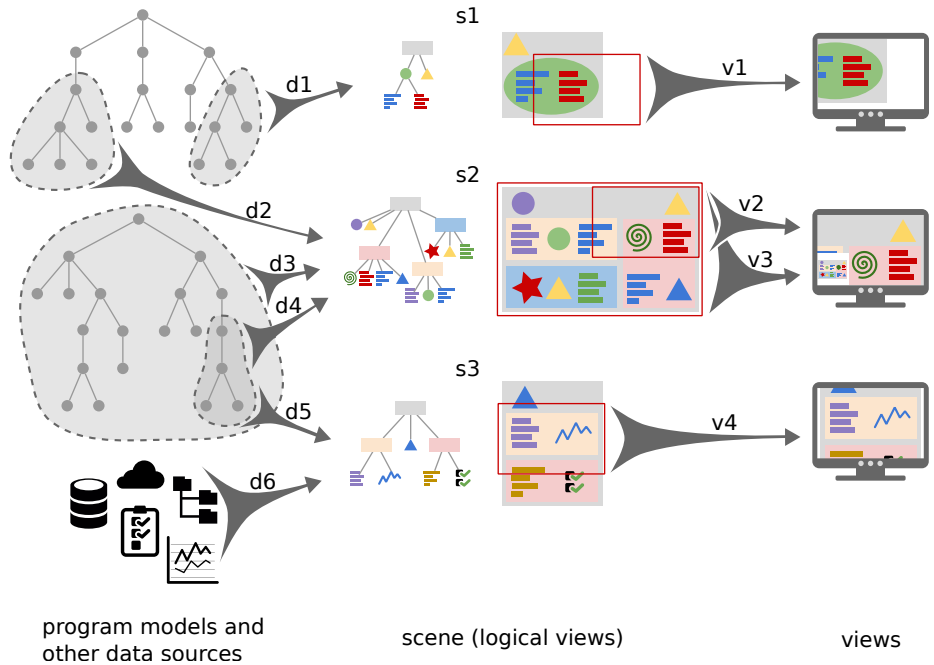


Figure 5.2: An illustration of composing and rendering visualizations in Envision. There are six (d)ata sources; three (s)cenes; and four (v)iews.[1]

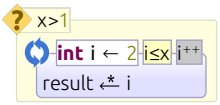


Figure 5.3: An item rendering an if-statement. The item draws the yellow background shape and is comprised of child items: the icon, the condition and the body of the then-branch. The then-branch itself contains an item representing a for-loop.

dimensional canvas which contains many different "drawings", its items. Scenes in Envision correspond to different logical views of collections of code fragments. For example, one scene might contain a map of the code, laid out according to the containment hierarchy of the various code fragments, while another scene might contain just a few different methods that the developer put together to form a working set for a particular task.

An *item* is a single visual unit, which may draw content on the scene and may have child items. The visualization framework is completely agnostic to how complex an individual item is and what it draws on the scene. For example, an item could draw a piece of text, a shape, an image, or even a dynamic HTML/Javascript page. Items that have children, can compose their children in a specific layout (e.g., horizontal, grid, constraint-based) in order to achieve a desired look on the scene. This is illustrated in Figure 5.3, where the item representing an if-statement both draws its background shape directly on the canvas and contains child items representing the elements of the if-statement.

Envision's visualization framework is very flexible with respect to how items are created and combined into a tree structure in order to form a scene. Figure 5.2 illustrates this flexibility with four use cases.

First, a common case is when the structure of the item tree closely matches the one of a corresponding program model fragment. This is illustrated by s1 and d1 and could result in a rendering like the one from Figure 5.3.

Second, a single tree of visual items might be constructed from multiple program models, e.g., s2 is constructed from d2, d3, and d4. This enables combining information from different parts of a program or even from different programs and is useful, for example, when a developer is working on a task that spans several projects.

Third, a single model fragment might be visualized using different types of items or it might be visualized in multiple scenes (e.g., d4 and d5 in s2 and s3, respectively) or both. This enables the same code to be rendered differently depending on context.

Fourth, item trees are not limited to visualizing only programming models. An item can also represent data from other information source such as the file system, databases, analysis results, etc. Such items can be freely mixed with items representing code as illustrated by d5 and d6. This is useful to integrate information and code, for example, to help developers answer questions as we show in Chapter 8.

Views

Views are the physical projections of a scene on the screen. A view can render an entire scene (v3 in Figure 5.2) or just a part of a scene (v1, v2, and v4). A view can also be scaled (v3), which enables zooming in and out. Multiple views can be

displayed at once (v2 and v3). We use this feature in Envision to provide a mini-map – a scaled-down rendering of the entire scene which helps the developer stay oriented when panning and zooming the main view.

Combining different types of items in a scene rendered by one or more views is the uniform way to display all information and interfaces in Envision, including the program code, messages and visualizations from other tools (e.g., compilers or profilers), and even UI elements such as the command prompt. This uniform visualization space combined with the high flexibility of items enable the creation of rich and context-specific interfaces. Next, we discuss the evolution and current design of the standard presentations of code in Envision.

5.2 The design and evolution of Envision's program visualizations

In this section, we show and motivate our designs for visualizing the structure of object-oriented programs in Envision. We discuss the evolution of the visualizations and outline possible avenues for future improvement. The presentations shown here are the ones that we created for our research and are just one particular design enabled by Envision's visualization framework. In this section, we discuss only Envision's standard presentations of code applicable to all contexts. For an example of context specific presentations, see Section 5.4.

5.2.1 Basics of rendering code structure

Our primary goal when designing the standard program presentation of Envision is to make the program structure more apparent and easy for developers to understand at all levels. Our aim is twofold:

- to further improve on the usefulness of traditional syntax highlighting for understanding the structure of methods.
- to provide visualizations that facilitate the understanding of global aspects of a project, e.g., the classes and packages that comprise the project.

To achieve these two goals we utilize a mixture of text and graphics for presenting code elements.

We purposefully include a wide range of graphical elements, because findings from cognitive psychology [WKL+06] and visual perception theories such as the Semiology of Graphics [Ber83] suggest that increasing the variety of visual elements used to represent a composition of objects can enable the different objects to be more quickly identified. The Semiology of Graphics theory defines key visual variables (e.g., shape, luminosity, color, size, and containment) and explores which of these are selective. A *selective* variable is one that enables differences in its values to be perceived instantly by a human. Finding a particular object in a visual composition requires a linear scan of all objects, if the target object has no distinguishing visual variables that are selective. If, however, the object has distinguishing selective variables, it can be found much more quickly, without performing a scan of all other objects. This

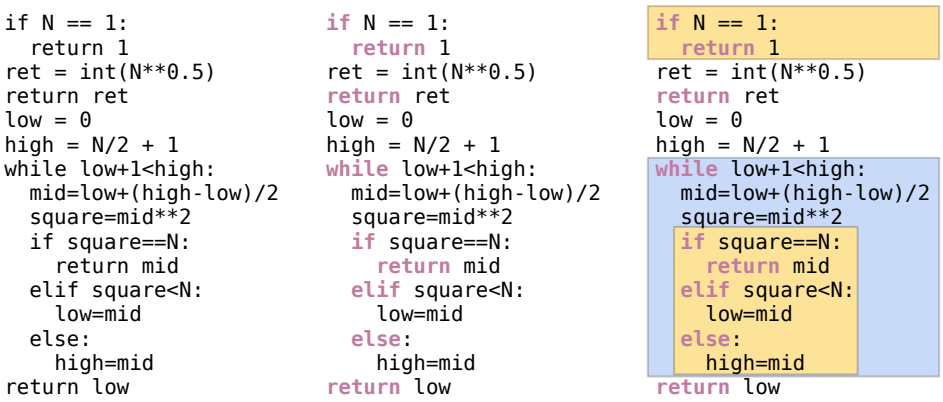


Figure 5.4: Three different presentations of the same code. *Left:* plain text. *Middle:* text with highlighted keywords. *Right:* text with highlighted keywords and colored containment boxes.

is best illustrated with the example in Figure 5.4. If a developer is interested in whether there are nested loops in a piece of code, using the presentation on the left in Figure 5.4, the developer would have to scan the entire text to find where there are loop statements and if they are nested. Traditional syntax highlighting (middle of Figure 5.4) makes this task easier, because now the developer needs to look only at the keywords. Since color is a selective visual variable, humans can instantly tell apart objects that have sufficiently different colors. However, with syntax highlighting the developer would still need to do a linear scan over all keywords. In the right of Figure 5.4, the syntax-highlighted code is enriched with additional color boxes for compound statements, indicating containment – another selective visual variable. Using this representation it is almost instant to see that there are no two nested loops, because there are no two nested blue boxes. Clearly showing the containment hierarchy obviates the need to scan over all keywords when searching for compound statements.

Guided by these findings, we attempted to maximize the visual variety of Envision's visualizations by using the following graphical elements:

- **Color:** We use a wide range of colors to highlight different parts of code structure. Light colors are used as backgrounds for compound statements, and some expressions. Dark colors are used for text or as the background for the names of top-level elements such as methods, classes, or packages. Icons used throughout the system also come in different colors.

- **Outlines and containment:** We use outlines to indicate scope and containment, which directly reflect the structure of object-oriented programs. Packages, classes, methods, compound statements, and some types of lists are all clearly delineated using an outline. These outlines feature different properties: width, color, padding to the content, corner radius, and the presence of a shadow.

- **Icons and decorations:** We use colored icons to indicate the type of various objects on screen. This includes all top-level constructs such as projects,

packages, classes and methods, as well as many low-level constructs such as statements and some expressions. We occasionally use other graphical decorations to make the structure of a construct clearer. For example, a dashed line separates a method's body from its signature and methods have small squares to the right of their name that serve as placeholders for adding arguments and results, if these have not been added to the method already.

- **Texture:** Texture represents the "roughness" of an object's appearance. Different icons often have different texture. A solid background color has a different texture than a striped background. Another possibility for texture is a smooth gradient. Envision primarily uses texture as part of icons and as gradients for the backgrounds of some compound statements.
- **Grouping and layout:** We use grid-based layouts to position top-level objects within their parent, e.g., classes within packages or methods within classes. We use sequential layouts to represent lists, e.g., statements or method arguments. In all cases, we group objects by proximity – objects within a layout are separated by a visible gap and the gap is larger for more top-level objects.
- **Text style:** We use text as a component of many visualizations, e.g., the names of types and variables and for most expressions. In addition to text properties that are often customized by traditional syntax highlighting systems such as color, weight, or slant, we also use variations of font and size.

By combining different graphical elements in a single programing construct we strive to make the construct quick to identify in different contexts. For example, an if-statement has a distinctive icon, a background color shared among conditional branching structures (if- and switch-statements), and an outline typical for compound statements. This enables if-statements to be selectively recognized based on different information needs, e.g., containment of statements or conditionality of execution.

Next, we discuss specific examples of Envision's visualizations and how they have evolved over the course of our work.

5.2.2 Design and evolution

Envision's code visualizations are entirely designed by the author of this dissertation, including all color, shape, and layout choices. In earlier versions of Envision, most icons were sourced from [Ope], whereas the current design includes many custom-drawn icons. The initial design goal was to use the full capabilities of Envision's visualization framework to maximize visual variety. Designing a large amount of diverse visualizations from scratch is challenging and doing so without the help of a trained designer or the capacity to test each individual choice inevitably led to some poor design choices and mistakes. After we conducted the user study discussed in Section 5.3, some of these design flaws became apparent to us. Using feedback from the study and new design insight, we have improved some aspects of Envision's code visualizations, though there is certainly potential for additional improvements.

Below, we will explore noteworthy elements of Envision's code presentations and discuss their evolution. Figure 5.5 shows a toy method exemplifying a diverse set of programming constructs as they were rendered when we carried out the user study from Section 5.3. Figure 5.6 shows the same method with Envision's updated design. We will look at each individual part of this method in turn.

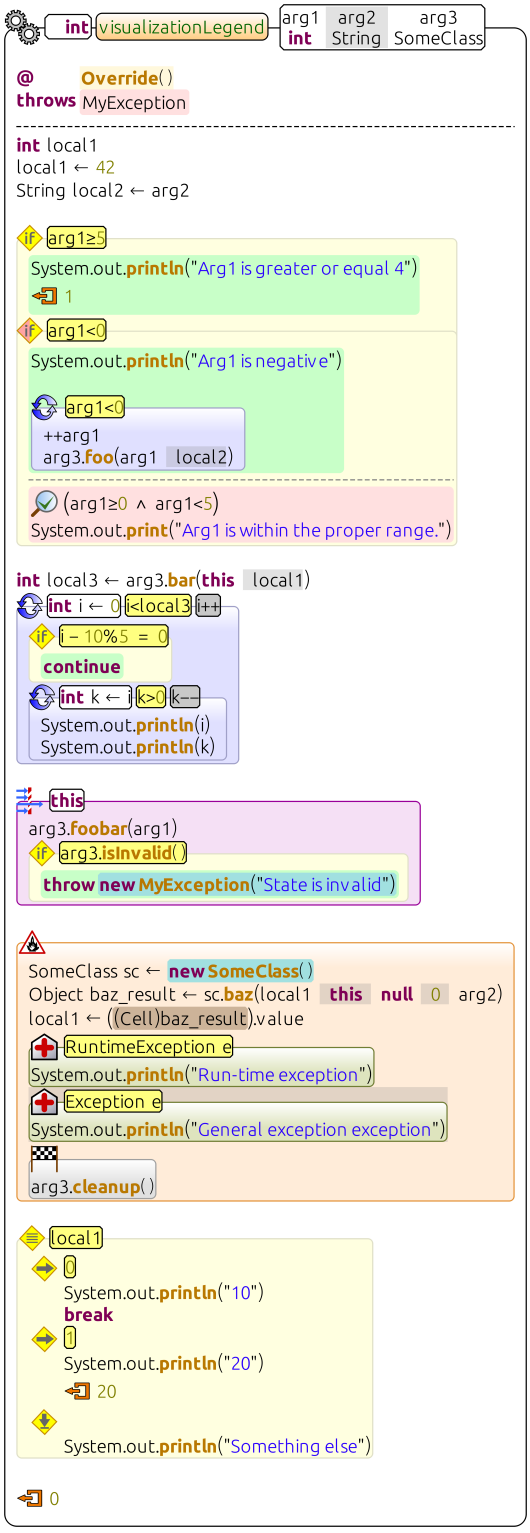


Figure 5.5: A method consisting of diverse programming constructs rendered using an early Envision visual design.

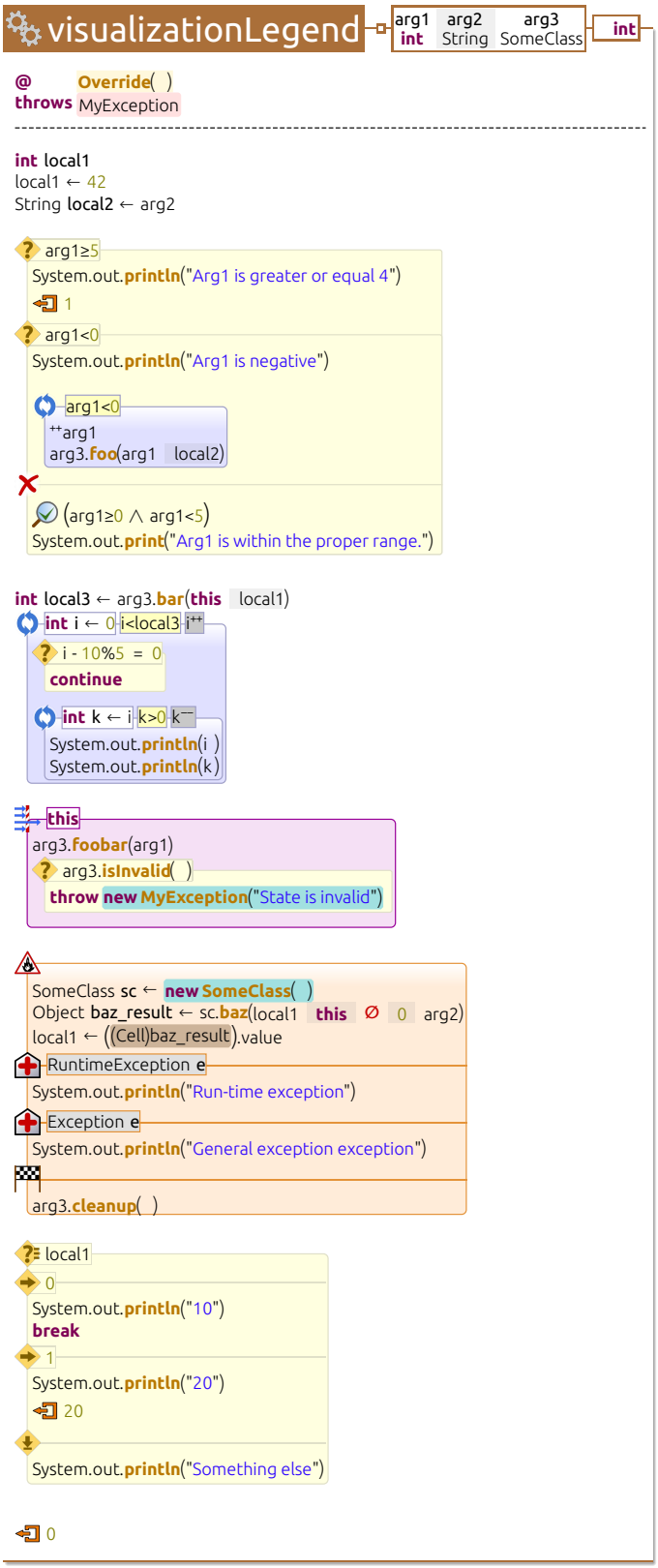


Figure 5.6: The method from Figure 5.5 rendered using Envision's current visual design.

Method outline and signature

Figure 5.7 shows the rendering of method signatures in Envision. In order to make methods stand out and, thus, the code structure more apparent, we use outlines to surround the method body. Initially we used a thin rectangle with rounded corners surrounding the method. To reduce clutter in the current version of Envision, we remove the left border of the outline, which was superfluous due to two factors. First, the method's contents already creates a strong visual boundary on the left of the method since all the content is left-aligned. Second, all methods have a white background, whereas the background of classes and other top-level constructs is typically non-white. This contrast produces a natural border at the boundary of methods within classes or other constructs. In the current design, we have also simplified the method's outline by using straight instead of rounded corners. Additionally, to make methods stand out, they cast a slight shadow on the underlying object.

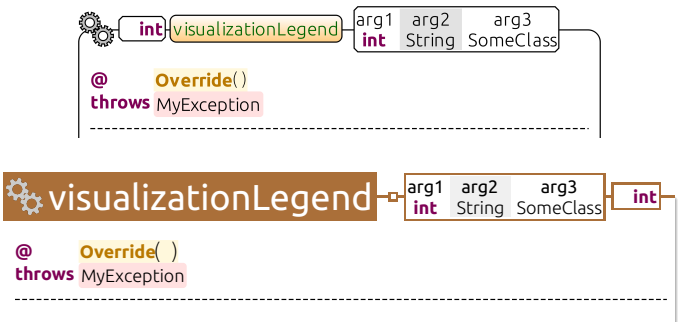


Figure 5.7: A method ( ) signature using an early (top) and the current (bottom) Envision visual design.

We use a non-textual method header to enable programmers to more quickly scan for methods. A typical feature for methods, as well as many other objects in Envision, is the icon in the top-left corner, which is unique for a given programming construct. Designing an icon for an abstract object like a method or a class is challenging, because such an object typically has no inherent or commonly-accepted representation that is easily recognized by people. For methods, we chose an icon representing two interlocked gears, symbolizing the inner workings of a machine. To the right of the icon come the method's name, arguments, and return type. The visual design of these objects has also evolved to reflect the design principle of contrast. Initially the name used the same font size as the rest of the objects in the method. The current version uses a much bigger font size, white text, and a dark background, that make the name easy to read even when the view is zoomed out and the body of the method is no longer readable. The list of arguments of the method uses an alternating white-gray background, instead of commas, in order to visually delineate the different list elements. By generally employing visual elements such as outlines and colors, instead of textual symbols such as braces and commas, we expect to enable parts of the code to be perceived more quickly thanks to the well-developed human visual system. To further increase visual variety, we display an argument's name above the argument's type, which also makes the list of arguments

more compact.

Under the method header we place the rest of the components that comprise the method's logical signature. This is the list of annotations on the method and the list of exceptions the method may throw. We separate the method's signature from its body with a dashed line.

Simple statements and expressions

Figure 5.8 shows the rendering of a few simple (non-compound) statements. They resemble very closely how expressions are rendered in standard syntax-highlighted text.

Figure 5.8: Simple expressions in an early (left) and the current (right) Envision visual design.

Envision allows improved visualizations of some symbols, independently of how an expression is typed. For example, an assignment is shown using a left arrow (←), an intuitive symbol that indicates the direction of data transfer and also used in many textbooks for showing the pseudo code of algorithms. Users can continue to type [=] to create assignments – the keystroke is independent of the visualization. If this is confusing for a user, the presentation of assignments or the way to type them can be customized on an individual basis.

In addition to using specialized symbols, we also eliminate semicolons as they are unnecessary in a structured programming environment. To further increase visual variety in the current design of Envision, we highlight variable declarations by using a bold font.

Compound statements

All compound statements (e.g., if-statements, loops, or try-catch-blocks) in Envision feature an outline, header, and a body. The outline and distinct background color serve to indicate a new scope and visually separate the compound statement from its surrounding statements. The header consists of a distinct icon in the top-left corner and other expressions that are not part of the body, for example the condition of an if-statement, or the initialization of a for-loop. While this basic design has persisted throughout the different versions of Envision, we have iterated on particular details of the design.

Figure 5.9 shows an if-statement with an else-if part and a final else branch. From the user study we discuss in Section 5.3, we learned that the early design of the if-statement's rendering was confusing to users. In that design, then branches and else-if branches had a green background color, while an else branch had a red background. More problematic was the fact that an else branch had no icon. Instead, else branches were separated with a dashed line from the then branch. While this design has good visual variety, it confused programmers because it lacked consistency: then and else-if branches could be identified by a distinct icon, whereas there was

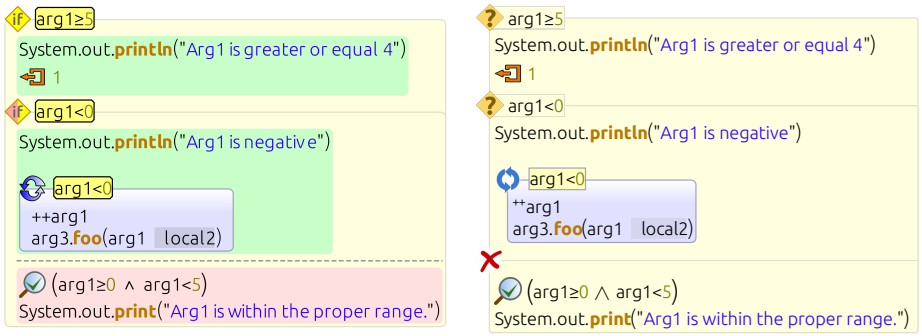


Figure 5.9: An if-statement (?) with an else-if (?) and else branches (✕) in an early (left) and the current (right) Envision visual design.

no icon for signaling else branches. The current design of Envision corrects this inconsistency by using an icon for else branches.

To improve the aesthetics of Envision, which participants in our study found unappealing, we familiarized ourselves with basic graphic design principles [Wil14] and applied them to the renderings of various programming constructs, including compound statements. To improve consistency, we redesigned the icons of if-statements and loops (Figure 5.10), removing text from the icons and giving them a flatter appearance. We also adopted a universal line and icon separator design for compound statements with multiple branches or bodies, such as if-statements, switch-statements (Figure 5.11), or try-catch-finally blocks (Figure 5.12). Each branch or body now appears under the previous one, separated by an uninterrupted line, and marked with an icon to the left of the line. We deliberately made constructs that serve similar semantic purpose look similar, while still giving them distinct icons, for example, an if-statement with several else-if branches and switch-statements. To improve the appeal of the design, we adjusted background and outline colors to a more consistent palette, removing neon yellow colors, and the red/green backgrounds of if-statements.

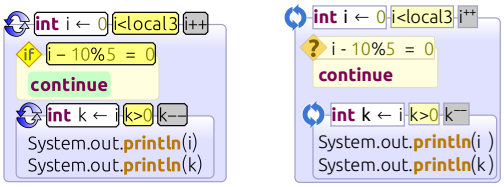


Figure 5.10: Nested for-loops (↻) in an early (left) and the current (right) Envision visual design.

Miscellaneous

To further increase the visual variety of Envision's visualizations we enhanced some expressions with two additional visual components. First, we added background

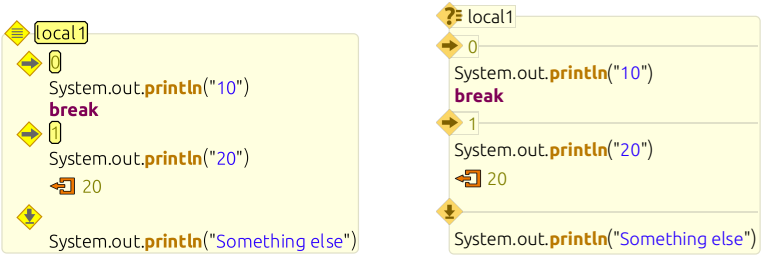


Figure 5.11: A switch-statement ( ) with two specified cases ( ) and a default case ( ) in an early (left) and the current (right) Envision visual design.

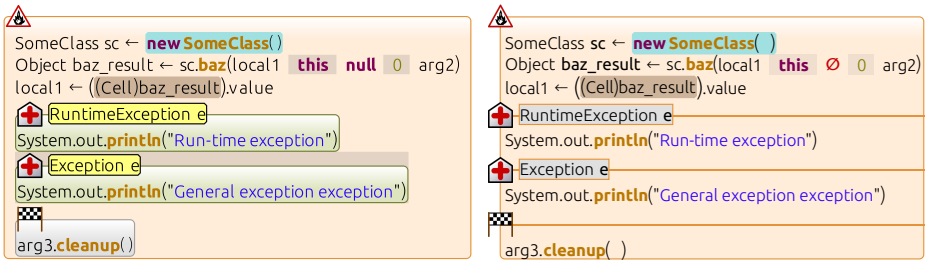


Figure 5.12: A try-catch-finally statement ( ) with two catch blocks ( ) and a finally block ( ) in an early (left) and the current (right) Envision visual design.

colors for expressions that manipulate the heap, such as new-expressions, and potentially dangerous expressions, such as casts. The former has a blue background that can be seen in the first line of Figure 5.12 and the last line of Figure 5.13, whereas the latter has a brown background that can be seen in the third line of Figure 5.12. Second, we automatically increase blank spaces in more complex expressions to indicate precedence. For example, the assert statement ( ) near the end of Figure 5.9 has a complex condition rendered with varying spaces. Since the and-expression (shown as $\wedge$) binds less strongly than $\geqslant$ and $<$, it is separated by additional white space from its operands compared to the $\geqslant$ and $<$ operators.

Another noteworthy point regarding the evolution of Envision's design is the arrangement of top-level constructs such as methods, classes, or packages, within their parent. Originally, Envision allowed such constructs to be freely positioned at any point within the body of their parent. The parent and its shape would automatically resize to fit all its children. While this design enabled higher visual variety, it suffered from a flaw: children could overlap, which frequently occurred inadvertently when adding more code to a child that had been placed near another child. To avoid overlap, we could either automatically push children or use a more restrictive, grid-based layout to dictate the position of children. Based on findings by Henley and Fleming [HF14], we chose the latter design. They compared their Patchworks code editor that has a 2×3 scrollable grid for showing code fragments to the Code Bubbles [BZR+10] editor that allows free positioning of code fragments

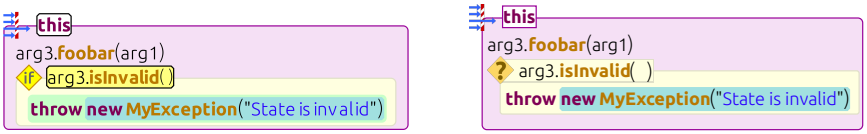


Figure 5.13: A synchronized-statement ( ) in an early (left) and the current (right) Envision visual design.

on the screen and automatically pushes code aside to avoid overlaps. In their experiments, Henley and Fleming found that developers could navigate more quickly and precisely using Patchworks and that having a restricted set of positions (i.e., the grid) where code could be placed actually reduced the burden on developers to arrange code on the screen.

Envision's grid-based layout does not implement a traditional grid, but has a major axis (usually the horizontal axis) and a minor axis (usually the vertical one). The major axis determines in which list (column) a child element should be, while the minor axis determines the position of the child within its list (column). An example is illustrated in Figure 5.14. Unlike a traditional grid, the height of a row in a column is independent from the heights of rows in other columns. Such a design still enables developers to easily choose a slot for child elements like in Patchworks, while producing a more compact overall appearance.

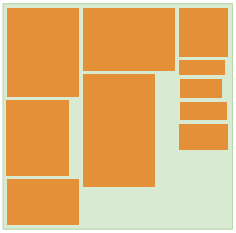


Figure 5.14: An example of Envision's grid-based layout for arranging the children of top-level programming constructs such as classes and packages.

Next, we present a user study that evaluated the early version of Envision's rich visualizations by comparing them to traditional syntax-highlighted text.

5.3 The effects of rich code presentations on code comprehension

To evaluate whether the kinds or rich code presentations enabled by Envision can be helpful to developers, in this section, we show two alternative presentations for Java code in Envision and compare them to the default syntax highlighted text of Eclipse. Our evaluation provides important evidence in an area that has scarcely been empirically investigated. While recent research in programming environments

[BZR+10, DBR+12, HF14, LBM14, OLDR11, OVH15] has explored a variety of novel techniques for visually navigating, augmenting, decorating, or abstracting code fragments, all of these enhancements pertain either to visualizations external to the code or to code fragment decorations, leaving the presentation of the actual code unchanged, unlike Envision's rich code presentations. To our knowledge no study compared different visual code presentations of the same source code, while keeping the programming paradigm and language constant across conditions. To investigate improvements to code presentation as a way to complement other research on development environments we pose the following research questions:

- **RQ1:** Do richer code visualizations affect the speed with which code features can be detected?
- **RQ2:** Do richer code visualizations affect the ability to correctly answer questions about code structure?
- **RQ3:** Do visually enhanced code constructs impair the readability of unenhanced ones (e.g., due to visual overload)?

Understanding code structure is part of more complex programming activities that developers perform regularly. Therefore, we asked 15 questions about code structure in a controlled study with 33 developers and compared their performance when using traditional syntax highlighting to two richer code visualizations from Envision. The results strongly indicate that Envision's richer visualizations reduce response times on a wide range of questions about code structure and do not lead to visual overload, contrary to developers' feedback.

5.3.1 Evaluation method

To determine if enhanced code presentations can help developers to more quickly comprehend code, we conducted a controlled within-subjects experiment comparing three different levels of visual variety. Below we provide details about the participants and the various components of the experiment.

Experimental conditions

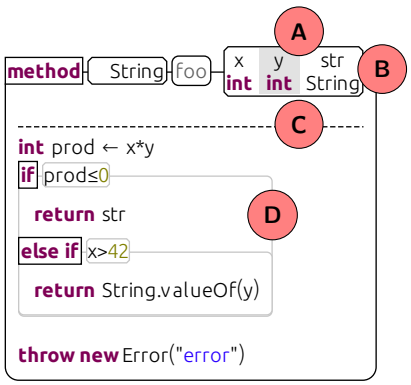
We showed participants screenshots of methods rendered with three different levels of visual variety:

- **v-low** corresponds to the default Java syntax highlighting in Eclipse.
- **v-med** is a presentation from Envision, which adds additional visual enhancements over **v-low**.
- **v-high** is also a presentation from Envision, which further increases visual variety over **v-mid**. This presentation is the early design discussed in Section 5.2.2.

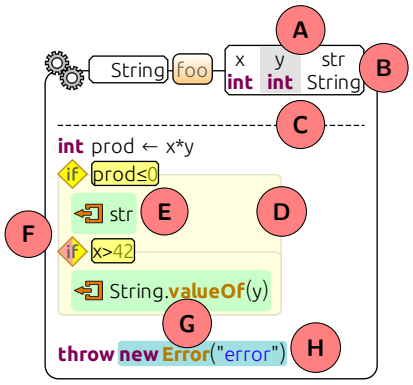
Figure 5.15 illustrates the most important differences between the three levels. At the time we conducted the experiment, **v-high** was Envision's default code presentation, and **v-med** was created by modifying only XML style files, without writing C++ customizations.

```
String foo(int x, int y, String str)
{
   int prod = x*y;
   if (prod <= 0)
      return str;
   else if (x>42) {
      return String.valueOf(y);
   }

   throw new Error("error");
}
```

(a) **v-low** (Eclipse - default settings)



(b) **v-med** (Envision - alternate settings)



(c) **v-high** (Envision - default settings)

Figure 5.15: A Java method rendered using three different levels of visual enhancements. *v-low* shows Eclipse using default settings. *v-med* shows a variant of Envision with the following enhancements over *v-low*: (A) all lists use alternating white and gray background instead of commas; (B) the names of formal parameters appear above the types; (C) a dashed line separates the method body from its signature; (D) blocks are visually outlined instead of showing curly braces. *v-high* also shows a variant of Envision, with the following additional enhancements over *v-med*: (E) some constructs, like compound statements have a background color; (F) many (but not all) keywords are replaced with icons; (G) method and constructor calls have orange text; (H) some expressions have a specific background color. The visualizations shown here are the ones from our evaluation, but they use an outdated visual style; see Section 5.2 for more details on Envision's visual design.

Method screenshots

For each level, we took screenshots of 298 methods from the open-source Java text editor jEdit. We chose the jEdit project in order to increase the ecological validity of the experiment. We used all methods from the jEdit codebase that are complex, but still fit on one screen, by selecting the ones that matched the following criteria:

- belong to the core jEdit packages: `org.jedit` or `org.gjt.sp.jedit`.
- have at least 2 parameters.
- have at least 3 block statements.
- screenshots in all three conditions have a size of at most 1920x1080 pixels so that they fit fully on a standard monitor.
- do not have any syntactical features that are not yet supported by Envision (anonymous classes, labels, final variables, long hex literals).

Some of the 298 methods that we selected required minor adjustments:

- All comments including Javadoc were removed, since Envision did not yet support comments at arbitrary code locations (e.g., in the middle of expressions). We have subsequently implemented support for Javadoc comments of methods and comments of constructs within method bodies.
- Annotations that suppress warnings were removed.
- Rarely, a few lines inside of a method were manually rearranged in order to make the method fit on the screen.

Participants

For our within-subjects study, we recruited 33 non-color blind participants with at least 1 year of Java experience. Demographics and experience data about the participants are shown in Table 5.1 and Figure 5.16.

	min	average (SD)	max
Age (32 responses)	21	28 (5.1)	42
Experience with Java	1	5.5 (4.2)	20
Experience with any language	3	11 (6.2)	28
Professional experience	0	2 (2.7)	12
Experience with Eclipse	0	3.5 (2.2)	8

Table 5.1: The age and programming experience of participants in years. Except for the first row, $n = 33$.

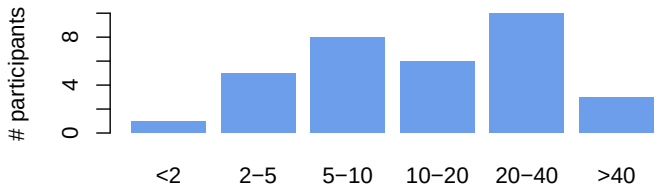


Figure 5.16: Weekly average number of hours that participants spent programming in the last 6 months.

Questions and experimental procedure

Each participant was presented with the 15 yes/no questions shown in Table 5.2 in random order. Within each question, we showed participants 15 method screenshots for each of the three visual variety levels. The order of the levels was randomized, but all screenshots from one level were shown in succession. Thus participants saw 45 screenshots per question, all of which were of randomly chosen methods from our pool of 298. In total we recorded $15 \times 3 \times 15 = 675$ answers per participant. For each combination of question and visual variety level, we drop the first three answers and average the remaining 12 for each participant in order to account for the learning curve. The choice to remove 3 samples was made before the experiment was conducted. We guarantee that the 45 methods that participants see for a particular question are all different, but participants may see screenshots of the same method in different questions. However, we believe this repetition has no effect, because participants see each method only for a few seconds.

We iteratively designed the 15 questions so that they match the following criteria:

- **simple**: we ask only questions about method structure that can be answered within several seconds without complex reasoning. This enables us to attribute any differences to the speed of comprehension even when developers have very different experience;
- **matching visual enhancements**: to test our hypotheses we need questions pertaining to constructs that are either enhanced differently or unenhanced by **v-mid** and **v-high**;
- **practical** and **representative**: Each question is a component of a practical programming activity and similar low-level questions are typical when reading code.

Such questions appear, for example, as parts of more complex questions that developers frequently ask during maintenance tasks as observed by Sillito et al. [SMDV06]. For instance, imagine a developer who is inspecting a method in order to improve its performance. The developer is looking for time-consuming operations, such as loops, especially nested loops (**Q8**), and calls to other methods, especially within loops (**Q13**). Generally, questions about code structure occur frequently as sub-tasks of many programming activities, such as looking up APIs (**Q7**, **Q9**), searching for errors or exceptions (**Q1**, **Q2**, **Q3**, **Q15**), tracing local definitions (**Q2**, **Q5**), understanding method structure and control flow (**Q6**, **Q8**, **Q10**, **Q12**, **Q13**, **Q14**), optimizing code (**Q8**, **Q13**), and tracking object life-times and state (**Q4**, **Q11**).

Id	Question
Q1	Does the method throw exceptions directly in its body using a throw statement?
Q2	Are all local variables immediately initialized (assigned) as part of their declaration?
Q3	Is subtraction (-) used in an expression? Any use counts, for example: a-b, -1, –i, x -= 3.
Q4	Is the 'this' identifier used in the method?
Q5	Is there a local variable (not a method parameter) of type String?
Q6	Is there an if statement with an else branch? Both else, and if else count.
Q7	Is the type of the second method parameter 'int'?
Q8	Is there a loop nested inside another loop?
Q9	Does the method have exactly 3 parameters?
Q10	Is there a top-level loop (not nested inside any other statement) that appears after some code containing an if-statement? The if-statement may be nested.
Q11	Does the method explicitly create new objects of any type, including arrays or exceptions.
Q12	Does the method catch any exceptions?
Q13	Is there a loop that contains two or more method/constructor calls? The calls can be anywhere inside the loop, including in nested statements, or arguments.
Q14	Are there multiple points from which the method can return (the end of the body is usually one such point)? Throwing exceptions does not count.
Q15	Does the method use an explicit type cast?

Table 5.2: The questions about code structure that we asked in our study. These are divided in three categories based on which visual variety levels enhance the fragments of code relevant for answering: Q1-Q5: all visual variety levels use the same textual (unenhanced) presentation; Q6-Q10: both v-med and v-high provide substantial enhancements; Q11-Q15: only v-high provides a substantial enhancement.

Before the study, participants received a brief introduction to the three code presentations and were given four sheets (see Appendix A) that serve as a visual legend that participants could use during the entire study. The study itself is implemented as an OpenSesame [MST12] script, which also includes a brief tutorial explaining the experiment and the interface of the experimental software. After the main part of the study, participants were asked to rank the three different code presentations, answer several Likert scale questions, provide demographics data, and optionally provide free-form feedback. At the end we gave each participant a chocolate bar as a gratitude for their efforts. To enable replication or additional analysis, all data and scripts are available for download from [Env].

We make the following hypotheses:

- **H1:** Response times are faster in questions that pertain to visually enhanced constructs (Q6-Q15).
- **H2:** Response times in questions pertaining to unenhanced constructs (Q1-Q5) are unaffected by richer visualizations of other constructs.
- **H3:** Correctness is unaffected by richer visualizations.

5.3.2 Results

Figure 5.17 shows a plot of the raw data we collected for both time and correctness, whereas Figure 5.18 presents an estimation analysis of the response time. This analysis avoids null-hypothesis significance testing, following Cumming [Cum14] and Dragicevic [Dra16]. Null-hypothesis testing has been shown to yield imprecise results in certain circumstances, where the resulting p-values could incorrectly suggest significance or insignificance. This may happen when an experiment's sample data distribution does not precisely match the distribution of the real population. In such cases, it is preferable to report confidence intervals, since they provide more information about the experimental sample and a more informed basis for comparison between samples. As the data of our experiment is not normally distributed we use Wilcox' robust bootstrapped estimation with trimmed means (B=2000, γ=.2) [Wil12]. Response times across all visualizations are similar for questions **Q1**-**Q5**, as expected, but also for **Q6**, even though it pertains to visually enhanced code. This supports **H2**. For **Q7**-**Q10**, we observe that both **v-med** and **v-high** outperform **v-low**. The reduction in mean response time is substantial and varies between 21% and 63%. For **Q11**-**Q15**, we observe that **v-high** outperforms **v-low** and, except for **Q14**, also **v-med**. Again, the reduction in mean response time is substantial: between 29% and 75%. Except for **Q6** the data supports **H1**.

Due to a clear ceiling effect, the correctness data is inconclusive. **H3** seems to hold for the simple questions that we asked, but this cannot be generalized for more complex ones.

Participant Feedback

Overall, participants preferred **v-med**, which received the best average rank (2.4) on a 1 to 3 scale, followed by **v-high** (2.0) and **v-low** (1.7). The participants commented that **v-med** was helpful while still feeling more familiar than **v-high**.

Figure 5.19 shows responses to three Likert scale questions regarding the default visualizations of Envision. Participants found Envision's icons and symbols easy to remember, but gave a mixed assessment for how easy it is to remember Envision's colors. Overall, participants reported that Envision's visualizations make method feature understanding easier compared to Eclipse.

30 out of the 33 participants provided textual feedback. 20 users found some aspects of **v-med** and **v-high** helpful. A clear theme from 23 responses is that **v-high** can sometimes feel overwhelming. 9 participants indicated that they would prefer a version that is a mix between **v-med** and **v-high**, primarily toning down (but not completely removing) colors and icons. 3 participants said they imagine using enhanced visualization levels, but only for viewing code (e.g., code review) and they would rather write code using a traditional notation. 5 participants stated that the rendering of `else` branches, using a dashed lined instead of an icon, is confusing, which might explain the insignificant differences in **Q6**. We have since improved the visualization of if-statements in Envision as discussed in Section 5.2.2.

5.3.3 Discussion

RQ1: In every question we investigated, increasing visual variety over **v-low** either had no effect, or substantially reduced the time to detect structural features of methods. The same effect was also observed when switching from **v-med** to **v-high**.

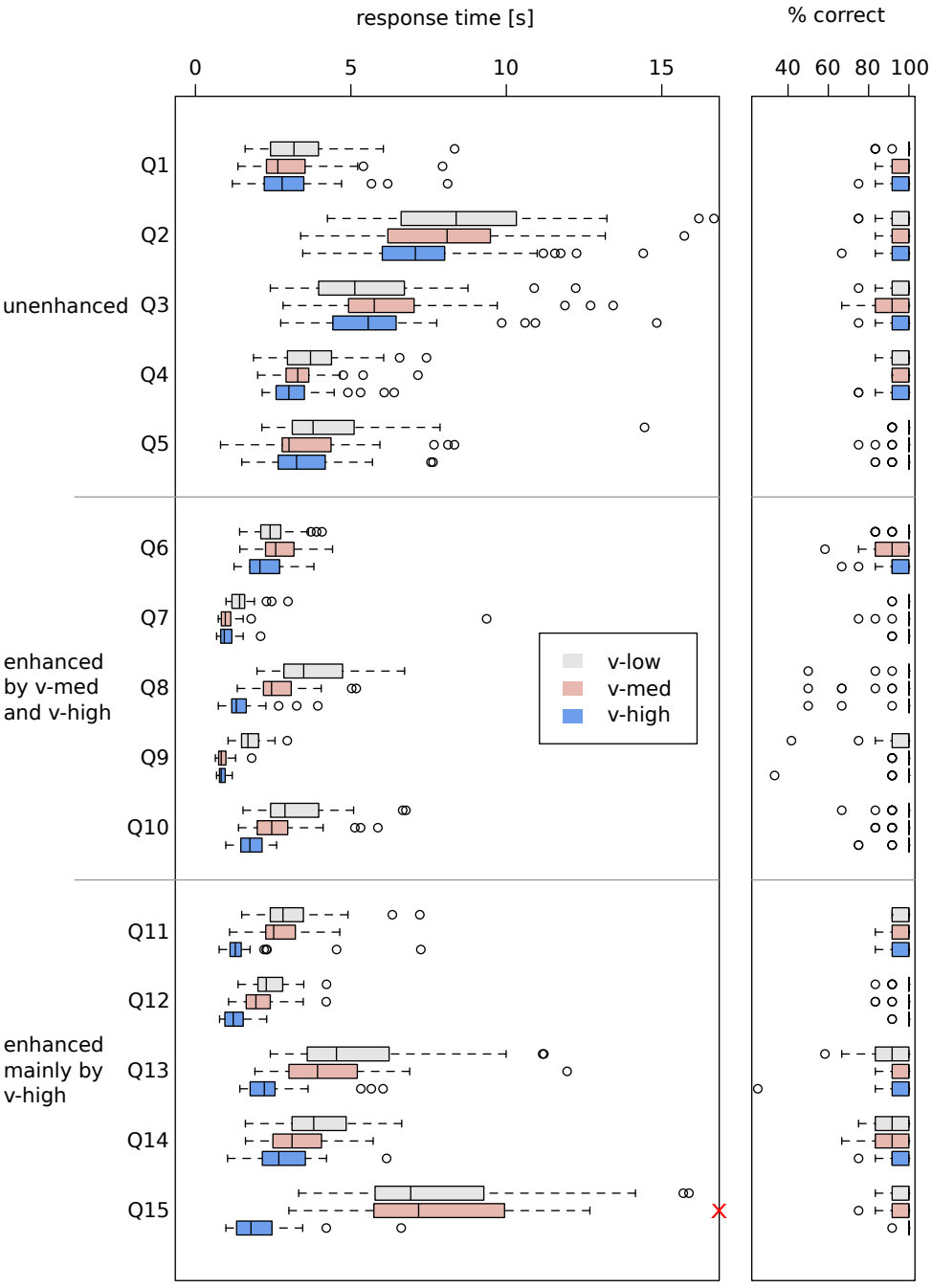


Figure 5.17: Box plots ($n = 33$) of response time and correctness per question and tool. The $\times$ indicates two out-of-graph outliers at 19.4s and 17.3s. We observe that increasing the level of visual variety lowers response times and does not affect correctness due to a clear ceiling effect. The whiskers represent the lowest/highest data points still within 1.5 $\times$ interquartile range (IQR) of the lower/upper quartile.

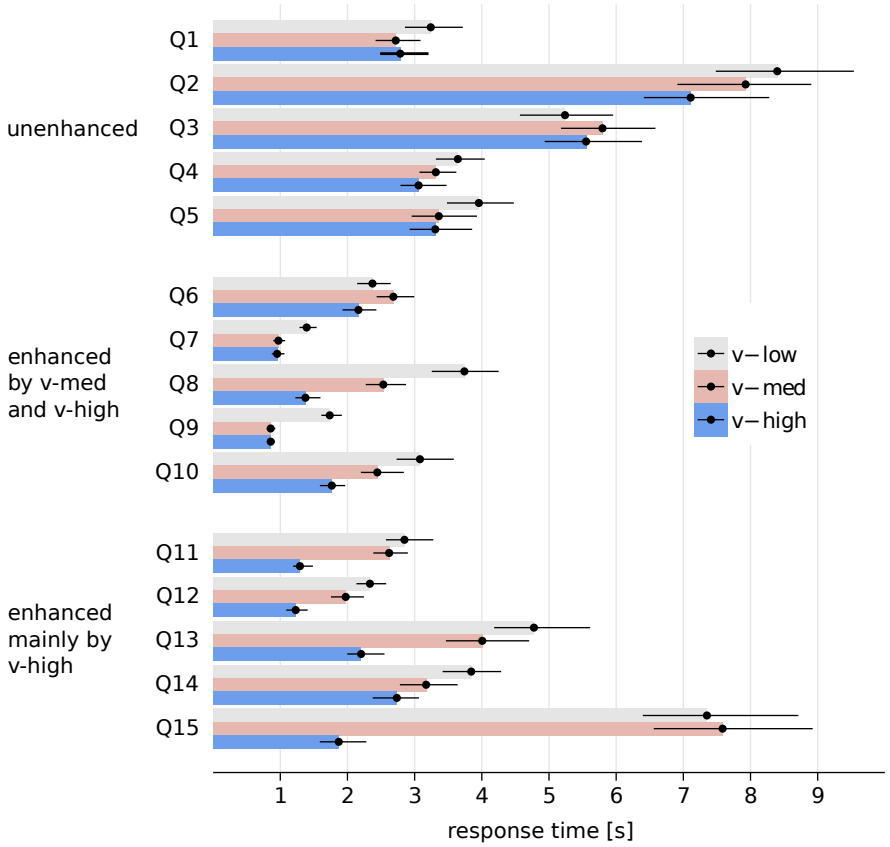


Figure 5.18: Estimated mean response times and 95% confidence intervals. A bootstrapped, trimmed means approach was used (B=2000, γ=.2).

The results show a medium to large reduction in response time (21%-75% or 0.5 - 5 seconds) in **Q7**-**Q15**. We believe that developers will ask similar low-level questions often, resulting in a tangible benefit. Richer visualizations can help to more quickly detect method features and we speculate that this might help developers maintain a state of flow and improve productivity.

RQ2: For the simple questions that we asked, participants almost always provided a correct answer, which resulted in a strong ceiling effect in the correctness data. This suggests that for such simple questions, traditional and richer visualizations are equally able to guide developers to the correct answer. It remains to be further investigated, what effect richer visualizations might have for more complex questions.

RQ3: The most interesting finding is that richer visualizations did not cause any measurable visual overload, even in **Q1**-**Q5**, where answers pertain to unenhanced constructs. This finding represents a mismatch between the performance of participants, which was best with **v-high**, and their overall preference for **v-med** as well as the feedback of 23 participants, who reported some sort of subjective visual overload

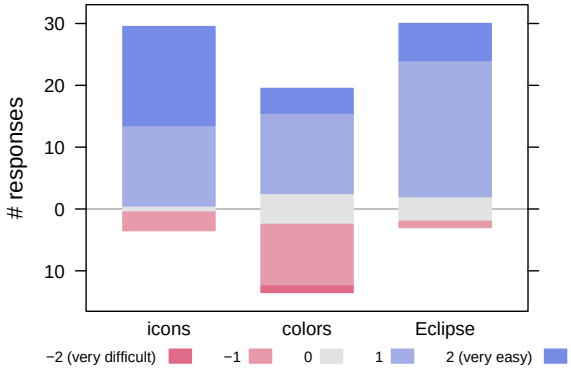


Figure 5.19: Responses for the three likert scale questions about v-high (Envision's default visualizations): "How easy was it to remember Envision's *icons* and symbols?"; "How easy was it to remember Envision's *colors*?"; "Overall, how easy was it to understand Envision's visualizations compared to *Eclipse*?"

or confusion. The open-ended feedback reveals three main factors that contribute to the participants' preference of **v-med** over **v-high**:

- Aesthetics: icons and colors in **v-high** were often criticized for being unappealing and messy.
- Else visualization: some participants specifically pointed out the inconsistent visualization of if-statements with else branches.
- Familiarity: **v-med** is closer to standard syntax highlighting and participants found **v-med** more intuitive.

This suggests that aesthetics and the users' feeling about a design are important for adoption: an unappealing design will likely not be used even if it can be helpful. It is worth investigating whether a visualization that is more aesthetically appealing than **v-high**, but with a similar visual variety, will be more popular. Since we conducted the study, we have updated Envision's design to make it more appealing and consistent, but we have not evaluated this new version.

Limitations

Our study has several limitations.

First, we tested the participants' responses on a limited number of questions in a controlled setting. We observed a ceiling effect in that developers almost always answered our simple questions about code structure correctly. It is possible that richer visualizations may make it harder to answer other types of questions or high-level questions about code. To increase the applicability of our findings we picked questions that occur as components of more high-level regular programming tasks. Thus, we believe that it is unlikely that other types of questions will be negatively affected by our richer visualizations of code.

Second, we draw all our sample methods from a particular Java code base, and results might not generalize to other code or other languages. To increase ecological validity we used an established, large, and actively maintained open-source project. Participants used **v-low** screenshots of the code as it was formatted by the developers of jEdit. Formatting is important for comprehension and it is possible that a different formatting style will significantly affect the responses. The original formatting of the code only slightly affects **v-med** and **v-high**, as only empty lines from the original sources are preserved and everything else is automatically laid out by Envision's visualization framework.

Third, as Envision does not fully support Java yet and does not allow comments in arbitrary places in the code, we had to filter and slightly alter the source code as described earlier. To mitigate any consequences of this limitation, we performed all code manipulations for all three visual variety levels.

Fourth, we measure only code comprehension. Nevertheless, reading code is an inherent part in most programming activities including writing, debugging, and testing, which suggests that improved visualizations could have an overall productivity benefit.

Fifth, more generally, rich presentations of code need to be studied further. The two such presentations that we evaluated performed well overall, but participants identified issues with the readability of specific programming constructs (e.g., else blocks) and there was a general consensus that the visualizations were not aesthetically appealing. While our code presentations included a variety of graphical elements such as outlines, colors, icons, and non-linear layouts, we only evaluated the presentations as a whole and cannot directly determine how individual graphical elements affect performance. Our findings should guide further exploration of rich code presentations. First, it is worth carrying more fine-grained experiments investigating the effect of individual graphical improvements, for example, outlines. Second, it is worth experimenting with more appealing visualizations. We speculate that as the visual appeal and coherence of code presentations increases, the visualizations will be better accepted by programmers and would yield even higher efficiency gains. Since the study, we have updated the visual design of Envision to make it more appealing and consistent, but we have not yet evaluated the improved visual design.

Recommendations for tool designers

Based on our results we make two recommendations for tool designers.

Our results show that **v-high** outperformed **v-med**. This is in line with Bertin's SoG theory [Ber83] which suggests that increased color variety can improve perception, and extensive use of color is the major difference between **v-med** and **v-high**. Based on these findings we recommend that designers of text editors boost the syntax highlighting capabilities of their tools by using a wider variety of colors by default and enabling the highlighting of more constructs. For example, designers could enable different highlighting for different keywords, identifiers, and code blocks. Our recommendation is practical since syntax highlighting is universal and improving it requires only marginal effort while being risk-free: one could simply revert to a classical coloring theme.

Generally, tool designers are encouraged to experiment with non-textual visualizations or visually-rich editors that allow flexible program rendering and, thus, increased visual variety. Our results show that using more visual elements such

as colors, icons, outlines, and non-linear arrangement of constructs can speed up detection of code features. Visually-enhanced programming environments can also provide advantages beyond the single methods which we evaluated in our study, for example, by improving navigation [BZR+10, HF14] or by enabling context-specific visualizations for domain-specific APIs and languages as we show next.

5.4 Using context-sensitive customizations

In this section, we demonstrate one of the more advanced aspects of Envision's context-sensitive visualizations: customizing the presentation of code that uses a specific API or DSL. Large software systems frequently make use of specialized APIs or DSLs to describe specific aspects of the system in a concise way. Examples include DSLs for specifying database queries, business processes, or security policies. Generally, there are two approaches to using domain-specific functionality:

- Use an external DSL (e.g., SQL in the domain of database queries), which allows programmers to concisely express high-level concepts from a particular domain in a (textual or graphical) notation that is suitable for that domain. Code written in such DSLs has been shown to be easier to comprehend [KMC12] than equivalent code in a general-purpose language. However, external DSLs require their own tool support such as parsers, code generators, etc., which makes their development costly. Additionally, external DSLs cannot be easily intermixed with code from other languages and domains, which makes interoperability more difficult.

- Use domain-specific abstractions, APIs, and embedded DSLs implemented as libraries in a general-purpose language, such as Querydsl [Que] for database queries in Java. This approach leverages the tool infrastructure of the host language, but also restricts the syntax and IDE support to that of the host language. Even though some host languages such as Python and Scala provide ways to customize syntax for embedded DSLs, for instance, through operator overloading, they do not fully support specialized notations. For instance, SQL's `WHERE` clause is implemented as a method call in Querydsl and treated as such by the IDE. Therefore, the IDE offers neither specific visualizations nor interactions for domain-specific code, missing potential opportunities to improve productivity.

Envision's support for customizations enables developers to combine both approaches: to implement domain-specific functionality as a library, but still design customized interfaces for accessing this functionality. For example, it is possible to display and edit a call to Querydsl's `where` method in the familiar SQL syntax. Domain-specific interfaces give library APIs and embedded DSLs much of the flexibility of external DSLs, while retaining the benefits of operating inside a host language supported by an IDE.

In the remainder of this section, we demonstrate the usefulness of Envision's context-sensitive interfaces by customizing the tool's visualizations and interactions for Microsoft's Code Contracts library for .NET. A video demonstrating our customizations can be seen at youtu.be/GD0W5HEteu8.

5.4.1 Code Contracts for .NET

The .NET Code Contracts library [FBL10, Codb] allows programmers to annotate code with assertions such as method pre and postconditions. Most assertions are expressed via calls to static methods of a class `Contract` as illustrated by Figure 5.20. Due to the library approach, the code annotated with Code Contracts remains standard C# code and can be handled by the standard compiler. Additional tools support documentation generation, run-time checking, static analysis, and automatic test case generation.

```
public int factorial(int x) {
  Contract.Requires(x >= 0);
  Contract.Ensures(Contract.Result<int>()>0);
  return x <= 1 : 1 ? x * factorial(x - 1);
}
```

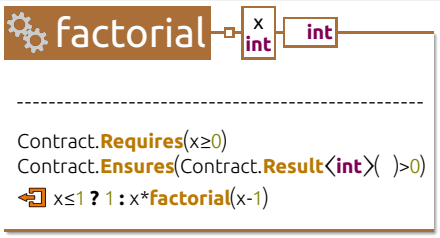


Figure 5.20: A C# factorial method using a standard textual notation and the equivalent (non-customized) presentation in Envision. The first two statements are calls to Code Contract methods to express the pre and postcondition of the method. The call to `Result` in the postcondition refers to the return value of the method; its type argument is needed to satisfy the type system of the host language.

The small example in Figure 5.20 already demonstrates three issues of Code Contracts' APIs:

- (A) Code Contracts are quite verbose compared to contract languages with designated syntax such as Eiffel.
- (B) The notation is sometimes inconvenient since it needs to satisfy the rules of the host language; for instance, the call to `Result` requires a type argument even though it is always the result type of the enclosing method. Even bigger inconvenience occurs for interfaces and out-parameters, as we illustrate later.
- (C) Calls encoding pre and postconditions occur within the method body, although conceptually they belong to the client-visible method signature.

In the rest of this section, we will show how to customize Envision in three steps to address these issues and to give Code Contracts the convenience of native language support despite being just a library.

5.4.2 Custom visualizations for contract methods

Following concept 1 from Section 5.1.1, we do not simply visualize calls to contract methods in C# syntax, but apply custom visualizations (here, based on the target method of the call). To address issue (C) above, we visualize these calls as part of the method signature, separated from the body by a dashed line. Associating contracts with the method signature is not only conceptually sound, but may also have other positive effects. For example, a special view with an overview purpose that shows only method signatures would now also include the contracts. Moreover, to address issue (A) above, we visualize calls to contract methods using a keyword style. The effect of these customizations is shown in Figure 5.21.

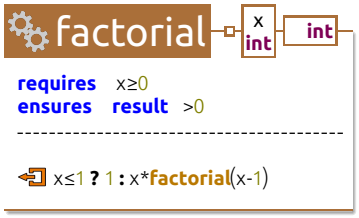


Figure 5.21: The factorial method from Figure 5.20 with custom visualizations. Contracts are visualized using keywords and visually separated from the method body. Contracts remain editable directly in this new form and new contracts can be created using shortcut keywords as described in Section 5.4.4.

Displaying contracts as part of the method signature is achieved by composing customizations according to concept 6. All visualizations in Envision have slots for optional *add-ons*, which are simply other visual items. Add-ons allow one to display additional information within a visualization. For contracts, we created an add-on for methods that displays additional items in the signature. This add-on reuses the existing visualization of the (entire) method body. To avoid that the method body is visualized twice, we apply *list filtering* as a second customization. Envision allows visualizations of list nodes to filter which elements get displayed. For method contracts, we installed a filter for statement lists that is sensitive to the visualization context (see concept 4 from Section 5.1.1): If the visualization appears in a method signature, only contract calls are displayed; otherwise, everything except contract calls is shown. Together, these two customizations render contracts as part of the method signature. The customizations are implemented in the *ContractsLibrary* plug-in, which needs to be distributed together with the library.

To apply the keyword style for contract methods, we apply a built-in visualization that shows method calls as keywords instead of using the normal method name. To change the visualization of all calls to a method, the method's definition simply needs to be annotated with the attribute `EnvisionKeywordVisualization( style )`. Here, *`style`* identifies a style in an XML file that allows visualizations to be easily configured without recompilation. It is possible to change the text, font, color, background, placement, and other parameters. It is also possible to specify an icon instead of text. The complete style for preconditions is shown in Figure 5.22; it specifies the keyword `requires` and uses defaults for all other parameters of the style.

```
<style prototypes="default"><keyword>
    <symbol>requires</symbol>
</keyword></style>
```

Figure 5.22: The style for precondition visualizations.

To illustrate the flexibility of visualization styles, consider the example in Figure 5.23. The second postcondition refers to the value of the `size` field, at the time of the method call, the so-called *old* value of `size`. In C# syntax, the old value is denoted by `Contract.OldValue(size)`. Again, we apply a context-dependent visualization, which renders calls to `OldValue` using the superscript keyword `OLD`.

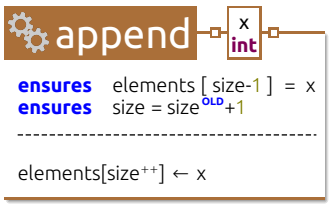


Figure 5.23: A keyword visualization with a different style – `old` is rendered as a superscript.

Note that all of these visualizations are fully interactive following concept 5: contracts can be added, edited, and removed directly in the signature. To add a contract, the user can either type the method call as usual or use the shortcut shown in Section 5.4.4, after which the call will be automatically visualized using the keyword presentation. Once this presentation is used by the system, the pre and postcondition expressions can be edited normally using the keyboard.

5.4.3 Custom visualizations for interfaces

Since interface methods do not have bodies, there is no place in the interface definition where one could write calls to contract methods. To work around this limitation, Code Contracts force developers to create a dummy contract class that implements the interface and whose sole purpose is to contain the contracts. This special contract class and the interface are linked by attributes as shown in Figure 5.24. This solution requires a lot of boilerplate code and makes reading the contracts of an interface difficult.

To address issue (B) above and shield the programmer from this inconvenient notation, we perform two customizations. First, we create another add-on for methods. It is active within the context of interface method declarations and displays the contracts from the associated contract class, as shown in Figure 5.25. This example illustrates that visualizations may depend on the entire AST (see concept 4 from Section 5.1.1), in this case, code contained in a different class. Again, this visualization is fully interactive according to concept 3: programmers may edit the contracts using the add-on, as if the contracts were specified in the interface itself. Second, in line with concept 3, we hide the contract class entirely as it is no

```
[ContractClass(typeof(ICalcContract))]
interface ICalc {
  int op(int x, int y);
}

[ContractClassFor(typeof(ICalc))]
abstract class ICalcContract : ICalc
{
  int ICalc.op(int x, int y)
  {
    Contract.Requires( x != y );
    return 0;
  }
}
```

Figure 5.24: An interface with an associated contract class. The attributes in square brackets link the interface and the class. The `return` statement is necessary to satisfy the C# compiler.

longer necessary and also omit the attribute in the interface referring to that class. Both of these customizations are implemented in the *ContractsLibrary* plug-in.

5.4.4 Custom interactions for contract methods

In order to help with writing contracts, we have also introduced new interactions.

First, contract method calls can be created by simply typing a keyword preceded by a backslash, e.g., `\requires` for preconditions. This feature resembles code snippets in modern IDEs, but registering such shortcuts in Envision can be done by library developers by simply annotating the method for which they want a shortcut.

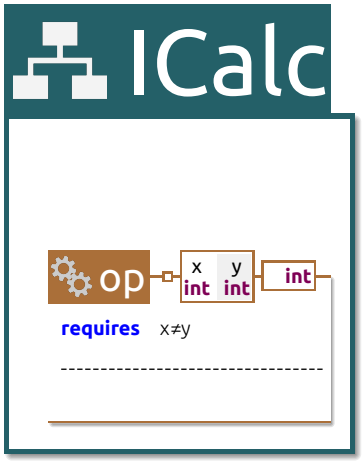


Figure 5.25: The `ICalc` interface from Figure 5.24 with a contract visualization add-on. The corresponding contract class is hidden as it is no longer needed.

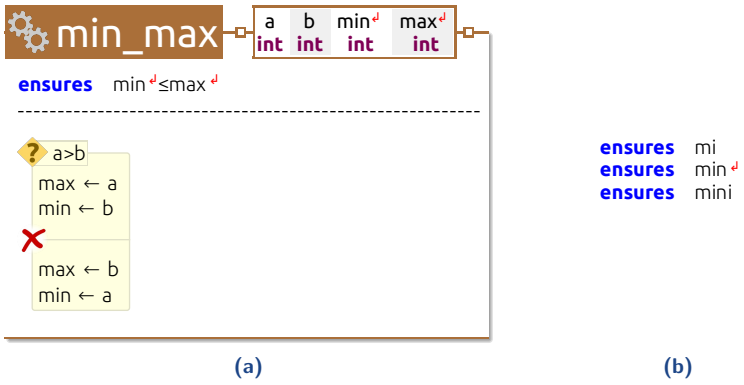


Figure 5.26: Wrapped references to the `min` and `max` out-parameters (a) and automatic wrapping/unwrapping during typing (b).

In the case of preconditions the annotation is `EnvisionShortcut("requires")`. If multiple libraries register the same shortcut the current implementation simply uses the last registered one, though it is possible to implement better conflict management.

Second, we customize expression editing to reduce typing in certain cases. Postconditions in Code Contracts may refer to out-parameters. Since postconditions appear at the beginning of the method bodies, the C# compiler complains that the value of an out-parameter appears to be read before it is initialized. The Code Contracts solution is to wrap all such accesses to out-parameters in a call to the `Contract.ValueAtReturn` method. In our custom visualization, we omit these calls, but we make them explicit in Figure 5.26a using a '↵' icon for better illustration. The same icon is used to indicate which parameters are out-parameters. To address issue (B) above and avoid the inconvenience of calls to the `ValueAtReturn` method, Envision does not only omit them in visualizations, but also inserts them automatically when the programmer types an out-parameter within a postcondition. Following concept 5 from Section 5.1.1, we achieve this by adding a listener for expression modifications implemented in the *ContractsLibrary* plug-in. If a method call to the `Ensures` method is edited, its arguments are "sanitized": all accesses to output arguments are wrapped, all other accesses are unwrapped. For instance, assume a programmer types `mini` within a postcondition of the `min_max` method as shown in Figure 5.26b. When the programmer types the 'n' key, the symbol is resolved to an out-parameter and automatically wrapped. As soon as the second 'i' is typed, the symbol will be unwrapped as it no longer refers to an out-parameter.

5.4.5 Discussion and limitations

One of the major goals for Envision is to make the environment very customizable. We believe our design largely achieves this goal. Even though we demonstrated only a single use case for customization, we used a variety of techniques and were able to customize visualizations as we wanted, without modifying the underlying program model in any way.

A second design goal for Envision is making customization easy. Here, we achieve mixed results. While Envision offers many abstractions that simplify customizations,

most customizations need to be implemented via plug-ins, which requires customization designers to be familiar with Envision's C++ implementation and APIs. This makes especially personal customzations less practical. Further effort is needed to support additional customization methods that enable ordinary users to personalize their environment without the need to write plug-ins.

Finally, the context-sensitive nature of visualizations may also cause ambiguities at times when there is no most specific context. For example, if a programming construct has a specialized visualization for zoomed-out views and also has a specialized visualization for the debug purpose, it is not clear which of these two visualizations should be used in a zoomed-out debug view. We currently resolve this in Envision by prioritizing some context types over others in the visualization scoring strategy and by enabling users to manually pick a visualization. The scoring strategy can be configured, but there is no clear priority order between the different contexts and perhaps there are even more suitable ways to disambiguate between available visualizations.

5.5 Related work

5.5.1 Evaluating code visualizations

Green and Petre [GP92, GP96] and Whitley et al. [WNF06] show that the usability of notations varies with the programmer's task, and neither textual nor visual notations are generally superior. However, they analyze notations for distinct programming models, and do not compare different notations for a single programming language. Hendrix et al. [HCM02] show that a control structure diagram of the code, embedded in the indentation area to the left of the text, can improve comprehension. However, the code itself is presented as plain text without even syntax highlighting. Feigenspan et al. [FKA+13] investigate a specific use of color showing that different backgrounds for the components of a software product line help to identify which component some code belongs to.

In the first study of syntax highlighting that we are aware of, Hakala et al. [HNS06] investigate users' performance using the default coloring scheme of the Vim code editor, code without highlighting, and one other coloring scheme. Surprisingly, they found no overall difference between the three schemes. The only other two studies of syntax highlighting that we know of, focus on eye tracking [Sar15] and a specific programming domain [Dim15]. Both suggest benefits of syntax highlighting, but they only compare plain text to syntax highlighting, unlike our study which compares syntax highlighting to richer visualizations.

Conversy [Con14] proposed a framework based on the Semiology of Graphics (SoG) [Ber83] to model the visual perception of code. SoG recognizes seven visual variables: shape, luminosity, color, position, size, orientation, and linking marks. In our work, increasing levels of visual variety correspond to more extensive use of these visual variables, which Conversy suggests can improve the performance of readers.

Nuñez and Kiczales [NK12] conducted a study on registration based abstractions (RBAs). Using an RBAs enabled code-editor, certain patterns in the source code are shown with an alternative visualization. These alternative visualizations are typically more concise and express the pattern more directly. RBAs can be used to emulate additional syntactical features, not present in the underlying language. While

Envision does support a similar mechanism, it can also enhance code presentation without mimicking new language features.

Petre [Pet95] discusses the importance of secondary notation in electrical engineering drawings and source code. In both domains, good use of secondary notation is important in facilitating comprehension, and misuse of secondary notation can even lead to confusion and misunderstanding. An important characteristic of Envision compared to traditional text-editors is the automatic handling of almost all forms of secondary notation. Users can only insert new lines in method bodies and write comments. Other forms of secondary notation, such as indentation and alignment, are part of the built-in, automatic visualizations. This simultaneously assures that secondary notation is always available and that it is never misleading.

Wiedenbeck [Wie91] shows that code beacons, which are "key features that typically indicate the presence of a particular structure or operation" - facilitate the initial stages of high-level understanding of code. Similarly, visually enhancing code can serve as a "structural beacon", that facilitates the early understanding of the local structure of code on a low-level. This might even improve the speed at which code beacons are detected.

5.5.2 Tools with rich code presentations

Researchers have investigated many tools for making code more readable and easy to understand. Early efforts [BM86] focused on typographic improvements and pretty printing. Recent work on programming environments [BZR+10, DBR+12, DR10, HF14, OVH15] shows that visual enhancements in IDEs can also be beneficial. However, all of these recent tools still use standard syntax highlighting for presenting code. Barista [KM06] is a research prototype of structured code editors that allows flexible visualization of code fragments. The initial prototype of the system shares many ideological similarities with Envision, but that prototype was not developed any further and has not been empirically evaluated. Unlike Envision, Barista does not explicitly focus on supporting customizations.

Lieber et al. [LBM14] developed Theseus, an IDE extension which augments code with additional coloring and statistics based on run-time information, such as how many times a method is called. A quantitative study shows no effect when using Theseus, but qualitative studies are favorable. Such an approach to enhancing code visualizations using run-time data is complementary to our work, which focuses on making static structures more explicit.

Caserta and Zendra [CZ11] compiled a survey of tools for visualizing software. These and similar tools present an abstracted view of the code and fulfill a different role than syntax highlighting and rich code presentations.

5.5.3 Visualization customizations

Modern development environments have advanced customizations that go beyond syntax highlighting using external plug-ins. For instance, the Code Contracts Editor Extensions [Coda] for Visual Studio enhance a method declaration by showing inherited contracts and, for interfaces, contracts from contract classes using keywords. However, unlike our work, these visualizations are not editable—changes must still be made at the source location where contracts appear as normal method calls. The

Editor Extensions also do not support the other customizations presented in this paper.

Eisenberg and Kiczales [EK07] describe an approach for presentation extensions using an Eclipse plug-in as a way to achieve language extensions. Their tool allows one to embed graphical notations inside an Eclipse editor, using annotations to mark code fragments that are extended. These annotations are used by their editor for custom presentations, and by a preprocessor to generate standard Java code corresponding to the extensions. In contrast, our work requires no preprocessing, and no changes to the code using an embedded DSL to achieve custom presentation.

Davis and Kiczales [DK10] describe an approach to recognize syntactical patterns in code and visualize them with a more convenient notation. Our approach similarly allows the editor to choose an appropriate visualization for some code, but the decision is not just based on syntax and name bindings—the entire AST can be used together with the visual context and purpose. We also support customization of interactions.

Intentional software [SCC06] and MPS [Fow05] are language workbenches that allow developers to create new textual and visual DSLs, integrated with a specially designed host language. To our knowledge neither tool supports the automatic context-sensitive visualization of program components based on the wide range of contexts supported by Envision, and unlike these tools, our approach works without introducing new languages or changing the host language.

Erwig and Meyer propose a framework for mixing textual and visual languages [EM95]. They use a text editor for the host language that allows language constructs to be created using graphical notations, particularly for specialized domains. Unlike our work, their approach does not allow the user to simply customize the appearance of arbitrary language constructs. Moreover, parsing a program with mixed notations requires user-defined visual grammars.

Efficient interactions in a structured editor

6

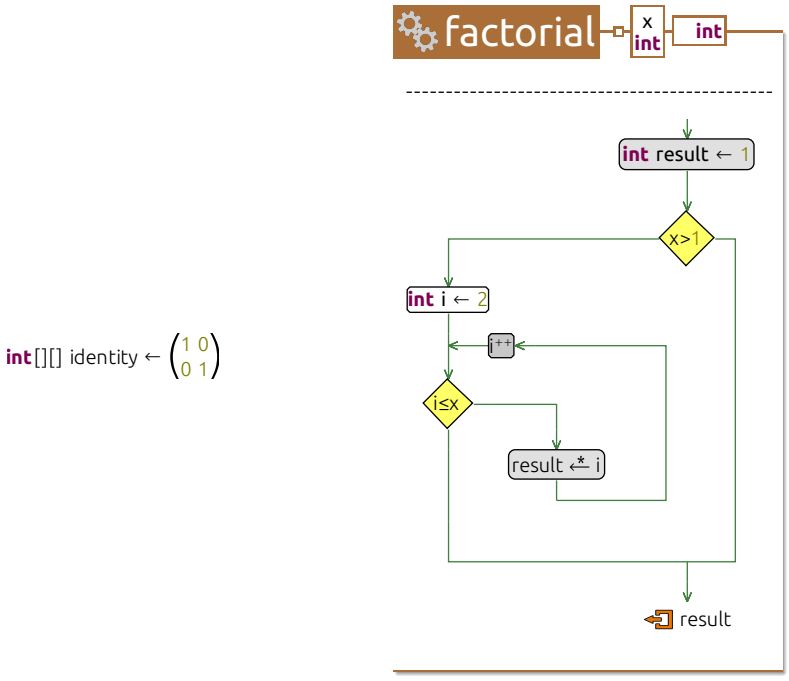
In the previous chapter we explored Envision's visualization framework and showed how it enables rich and customizable visualizations. In this chapter we look at the second major component of Envision's interface, the interactions that enable developers to efficiently create and edit programs. First, we describe the main challenges in designing efficient interactions for structured and visual code editors. Next, we provide detailed descriptions of the key interaction components that enable efficient manipulation of programs in Envision. Our approach is sufficiently general to address interaction issues also in other structured editors or visual programming tools. Lastly, we use CogTool [JPSK04, BJRT10] to compare Envision to Eclipse on three typical code editing tasks. The results of the CogTool simulations show that our code manipulation techniques enable program edits as efficient as edits in Eclipse [AM14b], thus overcoming a major usability barrier for structured editors for expert developers.

We remind the readers of Envision's introductory video (youtu.be/5YMaCQEoPe0), which shows the interactions we discuss in this chapter.

6.1 Challenges and requirements of interactions in structured editors

A key strength of text editors is that they provide a universal set of interactions that developers are well familiar with and can use efficiently. Thanks to these efficient interactions, editing code is fluid, takes only a small fraction of the overall development time, and requires only little cognitive overhead to think about the mechanics of editing. In contrast, structured and visual code editors are typically characterized by cumbersome interactions that impede efficient program manipulation and are only rarely used by professional developers. There are three main usability issues with such editors:

- **AST-based editing and navigation are slow:** Structured editors typically restrict developers to make edits that always leave the AST in a valid state. This limits how programs can be constructed, for example, by forcing program elements to be created in a top-down order according to the AST [MPMV94, VSBK14], which can be cumbersome, especially for constructs like expressions. Navigation might also be limited to follow the AST structure [MPMV94], which causes unintuitive cursor movement that does not allow users to directly navigate between objects as they see them on the screen.



(a) (b)

Figure 6.1: Different visual notations in Envision: (a) textual (declaration of `identity`), mathematical (matrix initialization); and (b) graphical.

- **Visual program editors have high viscosity:** Visual editors such as LabView [Lab] suffer from *high viscosity* [Gre89, Gre90, GP96], which means that local modifications of code are difficult and require many more operations than creating new code. For example, in LabView, inserting a new block on the visual canvas requires that the developer first clear space for the new block by rearranging existing blocks or wires.
- **Complex visualizations are often not edited directly:** Some tools such as Barista [KM06] offer flexible read-only visualizations of code and revert to a textual representation to allow edits. This approach works around the usability issues above, but prevents developers from always working with optimal notations.

To be successful with a broad range of professional programmers, a visual structured code editor must allow users to edit code structures at least as fast as the users can do this with a text editor. It should allow developers to edit the code quickly and directly. As we have seen, designing such efficient interactions is challenging in itself, and it becomes even more difficult to achieve in a highly customizable environment such as Envision. For example, supporting a mixture of different types of notations (e.g., the ones from Figure 6.1) necessitates some notation-independent interaction mechanisms. However, to allow the intuitive manipulation of each notation, it should be possible to also provide specialized interactions where

needed, for example, text-like notations should be editable as in a text editor for optimal efficiency.

To support different notations for programming interfaces and interface customization like we have seen in Section 5.4, and to enable efficient program edits, we define four key interaction requirements for visual structured editors:

- **Requirement of keyboard-based interactions:** All edits should be achievable by using just the keyboard, like in a standard text editor. This includes creating new structures such as methods, editing expressions, and navigating between different parts of the visualization. Developers can be very efficient using the keyboard and it should be the primary input device in programming tools for experts.
- **Requirement of direct interactions:** All visualizations should be editable directly, without the need to switch between simpler, but editable, visualizations and more elaborate ones that are read-only. In particular, it should be possible to directly edit non-linear and graphical visualizations. Using a consistent set of visualizations reduces the cognitive burden on users.
- **Requirement of genericity:** Built-in generic interactions, such as copy, paste, undo, and redo should work across all programming constructs regardless of how a construct is visualized. Having a set of core interactions makes the system more intuitive and transparent for the user. Furthermore, generic interactions reduce the implementation overhead for new visualizations, thereby promoting alternatives.
- **Requirement of customizability:** In addition to the functionality provided by generic interactions, it should be possible to create custom interactions. In this way, existing visualizations could be adapted to user preferences or new domains, and new visualizations with complex behavior could be created. Customizability also facilitates direct manipulation.

Next, we describe the key interaction components of Envision, which satisfy the requirements above.

6.2 Interaction components

In this section, we provide an overview of the basic interaction framework of Envision and explain its key components.

6.2.1 Interaction framework basics

In Chapter 5 we showed how Envision's program model is decoupled from its visualizations and explained that visualizations are created from elementary visual units called items that are connected in a tree structure to form a scene. Items are also the basic interaction unit in Envision. Whenever the user presses a key on the keyboard or clicks a mouse button, the currently focused item receives a notification of this low-level event. Events are not directly processed by the item, but rather by an event *handler* associated with the item. Handlers are C++ classes that can be associated with items in three ways of increasing priority:

1. **not specified:** If no handler is explicitly registered for an item type, the item will use the handler associated with the item's supertype. The supertype of all items uses a default handler.

2. **per-type:** If a particular handler is registered for an item's type, this handler will be used.

3. **per-instance:** It is also possible to register a dedicated handler for a particular item instance. In this case, this most-specific handler is used.

A single handler may be associated with multiple items or item types. Using different handlers is the first of two means of supporting context-sensitive event processing in Envision. The ability to assign handlers to items in three different ways makes it easier to customize input processing to specific domains or interfaces.

The **default handler** implements many basic features such as copy and paste support, undoing and redoing actions, arrow key navigation (see Section 6.2.2), standard command handling over the command prompt (see Section 6.2.3), and others. The current implementation maintains only a single undo stack for all code. To better support developers' fine-grained needs for undo [YM15] the implementation would need to maintain separate undo stacks for more fine-grained code fragments. Other handlers may reuse the functionality of the default handler, which facilitates the creation of interaction extensions.

If an item's associated handler is unable to process an event, this event is forwarded to another handler. This forwarding is the second means of context-sensitive event processing in Envision. First, the event is sequentially forwarded to the handlers of the item's ancestors in the visual tree of the scene, until the event is processed. For example, if a user issues a command while the icon of an if-statement is selected, this command event will be propagated first to the if-statement handler, then to the handler of the body that contains the if-statement (e.g., a method or another statement), then to the encompassing method, then to the encompassing class, etc. If none of these handlers process the event, it is forwarded to the handler of the current view, which enables view-specific functions to be invoked, e.g., zooming or scrolling. Finally, if the event remains unhandled, it is forwarded to the global event handler, which provides access to basic IDE functions such as loading and saving projects or switching views.

A handler has full access to the items from the scene, the program model, and generally all data from the IDE. Based on this information a handler may, for example, ignore the event, perform an action that changes the model (e.g., create a new statement), alter the visualizations (e.g., change the focus to another item, which will receive the next input), or invoke other functionality provided by plug-ins. This flexibility allows completely dedicated interfaces to be implemented for each different context. Building on top of the handler infrastructure, we created the three key higher-level interaction components that we describe next.

6.2.2 Universal visual cursor

A prerequisite for any application that uses keyboard input is the cursor. In textual environments, it indicates where the text that the user types will appear and developers can directly position the cursor anywhere on screen, providing two essential freedoms of interaction:

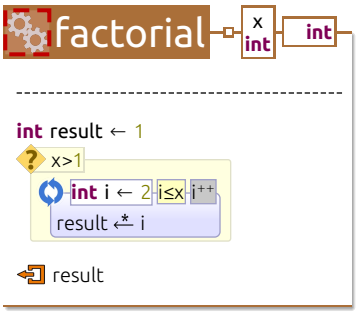
- **Freedom of placement:** Where the cursor can be placed is independent of the syntax or semantics of the underlying text.

- **Freedom of movement:** Navigating from one cursor position to another can be done directly on the presentation without having to understand the structure of the underlying program.

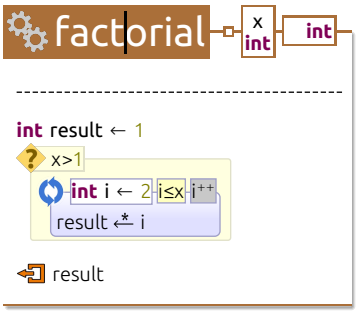
Visual and structured editors typically do not offer these freedoms: the cursor is often limited to selecting whole visual objects or manipulating text fields and navigation might be tightly coupled to the structure of the program. In contrast, we designed a cursor that provides the freedoms of a text editor in a visual setting.

Using the keyboard arrows or mouse clicks, it is possible to position the cursor virtually anywhere on the canvas:

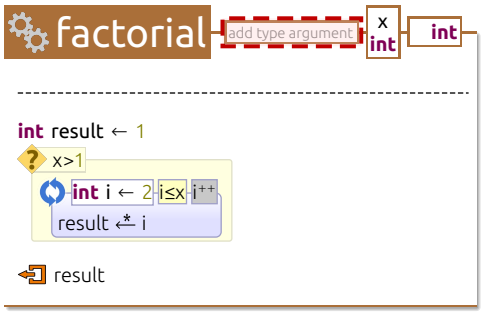
- On top of an item, thus selecting the item, which is a typical interaction in visual tools. For example, we can select a method's icon (to the left of the method name) to copy or delete the method:



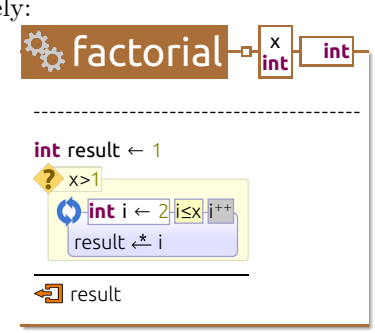
- Inside text. This is typical for text editors and text box widgets. For example, we can place the cursor in the middle of a method's name:



- Selecting placeholders. Using placeholders is useful to remind the users of available options for creating additional code fragments. For example, we can place the cursor after a method's name, selecting a placehoder that allows us to enter the method's generic type arguments:



- Between items, in empty spaces. This is an unusual cursor location for visual editors, even though it provides a lot of possibilities for interactions. Being able to mark empty space is especially useful for creating new structures. For example, we can place the cursor between statements in a method, or arguments in a list, allowing us, with a single key press, to create new statements or arguments, respectively:



Almost all of the cursor's functionality is provided by the default handler in conjunction with the visualization framework. No code is required for the cursor to work in a standard way with new types of visualizations. Next, we describe how the cursor works and how it may be customized for specialized interfaces.

Each visual item in Envision has a bounding rectangle and can declare regions within this rectangle that can be occupied by the cursor. For example, an item showing a piece of text will have cursor regions between characters as well as before and after the text. Typically, regions occupied by child items (e.g., the icon in the visualization of a method) are automatically declared so that children can be selected. Regions, such as between children or at the corners or edges, can also be declared to enable additional interactions, such as removing a child by pressing Del. Pressing an arrow key moves the cursor in the desired direction to the closest cursor region within the same item. If the cursor is already at the edge, it is moved to the visually closest region of the closest item in the desired direction. For example, in Figure 6.2, if the cursor is at the end of the `println` expression of the `main` method, pressing → will move the cursor to the right. Since there are no other characters to the right of the cursor in the current method, the cursor will be moved to the visually closest position rightwards that declares a cursor region – the beginning of `int result ← 1` in the `factorial` method. This mechanism also works for more complex visualizations like the control-flow from Figure 6.1b. This behavior allows users to navigate based on what they see on the screen, regardless of the underlying code structure.

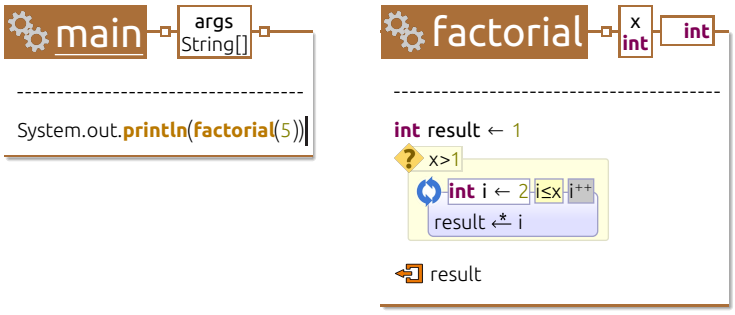


Figure 6.2: Two methods in Envision with the cursor at the end of the statement in the left method. Pressing → will move the cursor to the closest cursor region on the right: the beginning of `int result ← 1` in the `factorial` method.

In many common cases, appropriate regions can be automatically declared even for new visualizations. For example, when creating a visualization that uses the standard list layout provided by Envision, the layout will automatically declare cursor regions for list items and for empty spaces in the list. This automatic behavior supports generic interactions that work across visualizations. It is also possible to manually define cursor regions in a visualization to achieve a custom interaction. This is done within the definition of a visualization within a C++ plug-in.

When using the arrow keys to move the cursor, there are two interesting situations that need special treatment. The first one arises when objects are placed far apart and the next cursor position could be quite far away on the screen. Large jumps in the location of the cursor can confuse the user, so we limit the distance the cursor is allowed to move when a key is pressed, that is, the key press is ignored if the cursor would jump too far away from its current position. Originally, Envision never ignored navigation cursor presses and would instead scroll the view to show the cursor at its new location. However, we found this behavior undesirable when the cursor moved large distances, because scrolling the view to a completely new code location would change all the code on the screen, which was highly disorienting for us.

The second situation arises with adjacent cursor regions. A common case is illustrated in Figure 6.3. It is an expression consisting of four items: two identifiers (`base` and `offset`); the `+` operator; and the parent container. Each item has a number of cursor regions. Imagine, for example, that the cursor is positioned after the letter "a" and a user wants to use the keyboard to position the cursor after the "o". Going through all the cursor regions will take 8 key presses instead of the desirable 4. To eliminate this issue, Envision provides a way to mark adjacent cursor regions as equivalent and treat them as one region during navigation. Using this feature, moving the cursor in the case from Figure 6.3 behaves as desired. Many of Envision's default visualizations and layouts automatically mark edge cursors as equivalent to any adjacent cursors. With some existing visualizations, equivalent cursors regions can be manually tweaked using XML style files, but generally, non-standard cursor region equivalence is specified in the definition of visualizations, in plug-ins.

The cursor is solely based on the structure of the visual items in a scene and is independent of the program model. Thus, the cursor's behavior and movement are predictable and intuitive, and the user needs no knowledge of how a program's

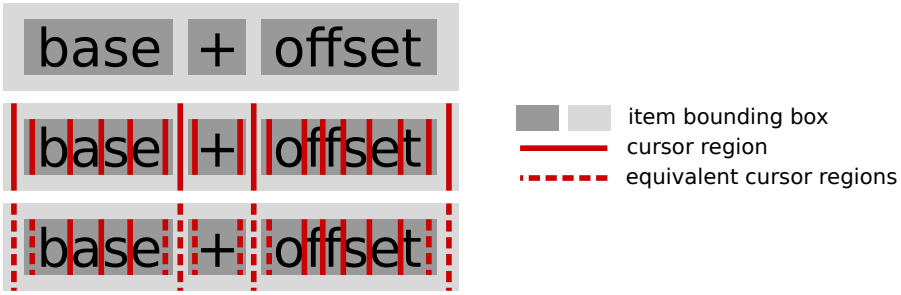


Figure 6.3: Four visual items representing the expression `base+offset` wrapped in a container. Top: the items' bounding boxes; middle: cursor regions without equivalent regions; bottom: cursor regions with equivalent regions.

internal model is structured, unlike structured editors such as Gnome [GM84]. Because it only relates to visualizations, the cursor can also be used in dedicated interfaces for rich data or interfaces that work with information other than the program model. This makes the cursor a universal tool for editing in Envision. The cursor enables three fundamental interactions in Envision:

- Marking positions for the insertion or deletion of objects via a single key press such as Enter or Back←.
- Selecting a context for invoking a context-sensitive command prompt.
- Providing a familiar text-like edit functionality for objects that resemble text, such as expressions.

We present the latter two next.

6.2.3 Context-sensitive command prompt

Envision offers a context-sensitive command prompt which is a universal mechanism for controlling the environment. The prompt provides access to actions specific to the currently selected item as well as all of the IDE functionality available in the current context. Unlike context-sensitive menus that are typical in visual tools, the prompt can be invoked and used entirely using the keyboard and does not restrict the actions available to the user to what can comfortably fit in a menu. The prompt functions similarly to a shell terminal or the command mechanisms in text editors like Vim or Emacs, all of which are tools that enable experts to be very efficient in controlling their environment.

We also use prompt commands to create high-level AST structures. Compared to expressions, high-level AST nodes such as classes and methods are created less frequently and usually in a top-down fashion, for example, developers first create a class and only then its fields, methods, and method bodies. These interaction patterns can be performed efficiently on a structural level, which makes it possible to keep high-level AST nodes always in a consistent state, and eliminates the need for error nodes, unlike when editing expressions, which need to support transient invalid states as we discuss in Section 6.2.4.

Figure 6.4: A command for creating a class.

Figure 6.5: An error message in the prompt.

Prompt commands are registered with one or more event handlers. Each handler supports an extensible set of commands. What commands are available in a particular prompt instance is determined by the handler of the currently focused item (the item that has the cursor). Pressing Esc or clicking the right mouse button shows the command prompt right below the cursor and allows the execution of:

- commands associated with the handler of the focused item. For example, the handler of a class has commands for creating methods, fields, and inner classes.
- commands associated with the handlers of the item's ancestors in the containment hierarchy. For example, if a class is selected, the available commands will also include a command for creating a package within the parent package of the class. This command comes from the handler of the class' parent package.
- IDE commands such as *find*, *save project*, or *switch view*.

Like any event handler, a command has access to the entire program model, all items from the scene, and all other IDE data. Therefore, there are no restrictions on what a command does: it can manipulate the program, change properties of visualizations, or even execute a function of the operating system.

When typing commands, Envision provides a list of auto-completion suggestions and a brief description of each command as can be seen in Figure 6.4. This helps explore the available commands and reduces the need to remember the list of arguments. If the user invokes an incorrect command, an error message will be displayed directly in the prompt as shown in Figure 6.5.

A further feature simplifying command entry and reducing the need to remember the precise syntax of commands is the support for abbreviations as we demonstrated in Chapter 2. For example, instead of typing the complete form of the `method` command to create a method (e.g., `public static method main`), it is sufficient to abbreviate the command name and arguments as long as the abbreviations are not ambiguous (see Figure 6.6). In case an abbreviation is ambiguous, the auto-completion menu can be used to select the desired command.

Figure 6.6: An abbreviated command for creating a method.

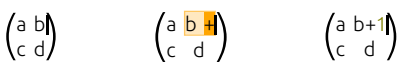
The prompt works across all visualizations and fulfills the requirement for generic interactions. Envision provides a number of built-in commands, but it is also possible to register new ones, according to the customizability requirement. New commands can be added to existing and new handlers by implementing the commands in plug-ins. A command is a C++ class that defines execution actions and also how the command can be auto-completed in the prompt.

6.2.4 Text-like expression editing

In a text editor, edits are unrestricted and allow the developer to temporarily break the code structure to quickly achieve the desired results with only a few keystrokes. For example, typing `{{a,b},{c,d}}` from left to right goes through many invalid states such as `{{a,b` and `{{a,b},`. Providing such a quick interaction has traditionally been a problem for visual editors, especially if non-linear visualizations like the one from Figure 6.7 must be editable. Quick editing is crucial for expressions since they are the leaves of ASTs, the larger and frequently edited part of a program. Unlike top-level AST nodes such as classes or methods, expressions are not usually created in a top-down fashion, for example, typing `a++`, first creates the child node (the reference to `a`) and then its parent node (the `++` operator). Enabling convenient editing of expressions in Envision requires an interaction mechanism with the following properties:

- Expressions can be typed with the keyboard, from left to right, like in a text editor.
- Expressions that are currently focused (contain the cursor) can be modified directly for any position of the cursor.
- No user input is ignored, even if that input results in invalid AST fragments.
- The number and size of invalid AST fragments is minimized, because such fragments can use only suboptimal generic visualizations.
- Free expression editing works even for complex, non-textual and non-linear visualizations (e.g., the matrix from Figure 6.7).

To achieve these properties, we use two ingredients. First we use special error nodes to represent incorrect and partial tokens resulting from user input, similarly to how JPie [BG05] and Barista [KM06] relax the edit-time grammar. This enables users to create expressions quickly by temporarily going through invalid program states. Second, we use a dedicated event handler for expression items. This handler processes user input to change Envision's program model, creating error nodes if necessary. Because the handler needs to work with arbitrary code visualizations and also allow linear left-to-right input, it works by linearizing visualizations to text,



(a) (b) (c)

Figure 6.7: Editing an expression visualized in a matrix form.

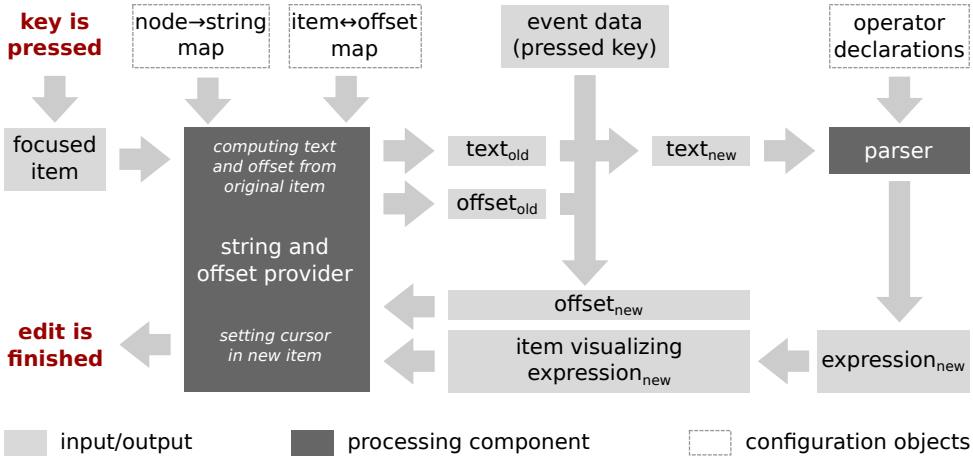


Figure 6.8: The process of editing an expression after a key has been pressed.

performing linear edits on this text, and reparsing the text to create new expressions. Next, we explain in more detail the operation of the expression handler, which is illustrated in Figure 6.8.

On every keystroke, the visualization of the edited expression is first mapped to a corresponding text string and an integer offset that represents the current position of the cursor in the string. For example, in Figure 6.7a, the matrix visualization is mapped to the string `{{a,b},{c,d}}` and the offset of the cursor is 5 (just after the `b`). This mapping is done by the string and offset provider component which we describe in more detail later. Next, a new string and offset are computed based on what key is pressed. For example, pressing a letter inserts it into the string and increments the offset, whereas pressing Back← removes the symbol before the offset and decrements the offset. In Figure 6.7a, pressing + results in the new text `{{a,b+},{c,d}}` and offset 6. The new text is then parsed to produce a new expression and corresponding visualization items. We provide more details about the parser later. The new expression visualizations and the new offset are fed into the string and offset provider, which moves the visual cursor inside the new visualization to a location that matches the specified textual offset. For example, in Figure 6.7b the visual cursor appears right after the `+`. Moving the cursor in this way allows uninterrupted left-to-right typing even in complex visualizations. Despite having multiple processing steps, the entire editing process lasts under a few milliseconds. The slowest component is the parser, which we have specifically optimized to efficiently process large expressions.

Next, we describe the two processing components involved in expression edits.

String and offset provider (SOP)

The SOP has two functions.

First, it can map a visual item to a string (but not the other way around). In Figure 6.8, this first function of the SOP is used to compute text_{old}. Any item, no matter how complex, can be mapped to a string, provided that an appropriate mapping is registered by a plug-in. To make providing such a mapping easier, we

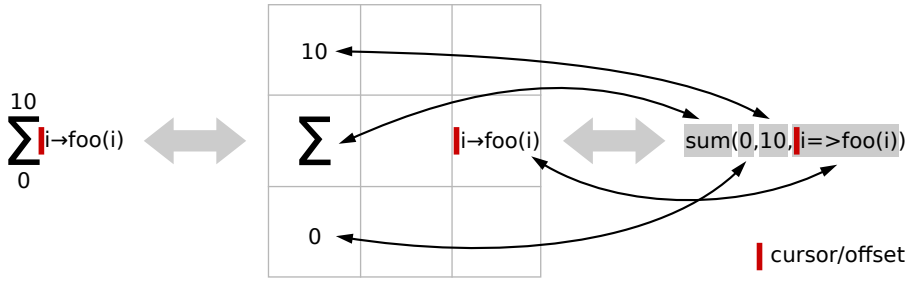


Figure 6.9: An example of the intermediate mapping to a virtual grid that facilitates the bi-directional mapping between a visual cursor's position and a textual offset.

include defaults for many of Envision's existing visual items. For example, visual items that represent AST nodes use the mapping of the corresponding node and visual items that represent lists are mapped to a concatenation of the mappings of their children. Thus, for editing expressions, it is sufficient to provide a mapping from each type of expression AST node to a string. For example, a throw expression E is mapped to `"throw $E->expr$"`, where `"$E->expr$"` is recursively substituted for the mapping of that expression.

Second, the SOP can map the current visual cursor to an offset into the textual representation of the current item and vice-versa. In Figure 6.8, this second function of the SOP is used to compute $\text{offset}_{\text{old}}$ and to reposition the visual cursor after the new expression is visualized. A mapping between the visual cursor and a textual offset can be computed regardless of the complexity of a visualization, if an appropriate mapper procedure is registered with the SOP. To facilitate creating these bi-directional mappings, we introduce an intermediate mapping to a virtual grid as illustrated in Figure 6.9. An item's child elements are mapped to separate cells in a grid. Each such cell is in turn mapped to a substring from the item's textual representation computed by the SOP. To determine the textual offset that corresponds to a visual cursor, the SOP uses the cell that corresponds to the child item that has the cursor. If the cursor is not on an edge of the cell, the SOP recursively explores the child element to determine the correct offset. If the cursor is on an edge, like in Figure 6.9, then it is mapped differently depending on what key was pressed. If the user has pressed a key that inserts a symbol (e.g., Q or *), the cell to the right of the cursor is used. This cell can be either the current cell, if the cursor is on that cell's left edge as in Figure 6.9, or the adjacent cell, if the cursor is at the right edge of the current cell. Analogously, if the user has pressed a key that removes a symbol (e.g., Back←), the cell to the left of the cursor is used. The final offset is either the start or the end of the substring that corresponds to the chosen cell, depending on whether the cursor is on the left or right edge of the cell, respectively.

Note that not all cells have adjacent cells. For example, in Figure 6.9, there is no cell to the left of the cursor. In such cases, the current cell is used regardless of the key that was pressed. This enables fine-grained control over the cursor↔offset mapping. For example, pressing Back← in the situation from Figure 6.9 would remove the comma before `i=>foo(i)`, bringing this expression to the top of the sum,

right after the `10`. While this may seem like a strange outcome, it is a mechanism that allows developers to temporarily break the expression structure when making edits and more directly write the expression they want. For example, in Figure 6.9, a developer typing quickly might write "`19, i=> foo`" before realizing that they meant to type `10` on top. The grid setup from Figure 6.9 would allow this developer to simply delete the last few characters using Back← until they delete the `9` and retype the rest of the expression correctly, just like they might do in a text editor. Alternative grid configurations are also possible for the Σ visualization, for example, if we removed the empty column in the middle of the virtual grid, pressing Back← would remove the `m` of `sum`, because the `sum` cell would now be adjacent to the left of the cursor. Another alternative is to have two cursor regions: one at the left of the `i=>foo(i)` cell as shown in Figure 6.9, and one on the right of the Σ cell. This would enable finer-grained control of the edit behavior.

The reverse mapping from offset to visual cursor position is performed in an analogous fashion, also taking the type of the pressed key into account. Visualization designers are free to provide alternative mapping mechanisms for their custom items. The recursive nature of the SOPs operation enables different mapping strategies to be transparently composed. For example, we provide a specialized mapping for purely textual items, which directly return an offset based on the currently selected cursor region. Visualization designers can customize the SOP and associated mappings for new visualizations by writing an Envision plug-in using two SOP-related APIs: simpler mapping declarations for common cases and full SOP specialization for maximum flexibility.

The bi-directional mapping performed by the SOP is essential for providing flexibility of notations in Envision and enabling all notations to be directly editable using the keyboard, without the need for separate textual interfaces for performing edits. This is in contrast to other tools like Gnome or Barista, which have a tighter coupling between syntax and editable visualizations.

Parser

The parser component needs to accept any input and produce error nodes for input fragments that are not understood. A key requirement towards the parser is that when the input string is not syntactically correct or complete, only a minimal number of error nodes should be produced, and each node should span as small a substring of the input as possible. This assures that as a user is typing an expression, well-formed parts of it remain well-formed and are visualized according to their meaning. More precisely, we distinguish two types of AST nodes for recording malformed input: *error nodes*, for parts of the input that are not understood (e.g., the symbol `#` where an identifier is expected), and *unfinished operator nodes*, for parts of the input that include fragments of an operator, but are not yet fully complete (e.g., the partly finished plus operator `b+` from Figure 6.7b). Our parser algorithm prefers unfinished operators to errors.

When parsing malformed input strings, to find a parsing with minimal error and unfinished operator nodes, the parser essentially processes the input string in all possible ways by inserting error and unfinished operator nodes in different positions. As exploring all options for longer inputs can be very expensive, we use a number of heuristics and rules to speed up parsing. For example, we score each parsing based on the number and size of errors and unfinished expressions it contains and we use the

best discovered parsing to prune the search of worse parsings. Additionally, we treat expressions wrapped in matching parenthesis independently of other expressions. For example, the input

```
foo(a+b) + objects[23] + bar(c
```

will trigger three independent parser runs:

- sub_1 = `a+b`
- sub_2 = `23`
- and `foo(`sub_1`) + objects[`sub_2`] + bar(c`, where sub_1 and sub_2 are considered atomic entities and are not recursively processed.

These optimizations make the parser sufficiently fast so that it can be invoked on every key press, even for large and deep expressions. In our experience parsing usually takes only a few milliseconds. In exceptional cases, for example, an expression representing a list of millions of bytes, parsing could take significant time. However, all such expressions that we have encountered have always been automatically generated and not edited by the user.

Our parser employs a hybrid of regional and global recovery schemes as described by Hammond and Rayvvard-Smith [HRS84]: parsing errors are detected locally and are "corrected" by backtracking and exploring multiple parallel forward parses similar to Lévy's approach [Lév75]. While parallel parsing approaches like Lévy's are normally considered impractical, our specific optimizations and the fact that we parse only expressions (as opposed to entire programs), make such an approach feasible.

This parser design makes it possible to provide accurate and context-sensitive visualizations immediately after a key press, even for incomplete or erroneous expressions. Additionally, syntactic issues with an expression are focused on small fragments of it and are clearly visualized, guiding the developer in finishing the edit.

The string offset provider and the parser are not limited to operating together or to editing only expressions. Any other visualization that requires text-like interactions can use these components. For example, we use the string offset provider with a simple regular-expression-based parser to provide convenient typing of the information queries discussed in Chapter 8.

The presented techniques for fluid expression editing are our second attempt at achieving efficient interactions in a structured editor. Initially we designed an editor without parsing or mapping to text, guided by the findings of Ko et al. [KAM05b] about the editing behavior of developers. In this first prototype, we implemented a "pure" structured editor where syntactic consistency would always be assured and quick editing was achieved through specialized support for frequent kinds of edit interactions, e.g., automatically creating a complete list by typing only its left delimiter. This is a similar approach to the built-in editor in MPS [VSBK14]. This first design suffered from two major issues, which ultimately caused us to abandon it. First, developers would still have to learn to use a somewhat unusual editor with unintuitive cursor behavior, even within textual notations. Second, providing appropriate interactions required many specialized rules for each different kind

of expression visualization, an approach that would be difficult to extend to new visualizations.

Editing complex visualizations

While complex visualizations do use the expression handler that we described above, they may provide additional interactions to make edits easier. We will illustrate this by creating and editing the specialized matrix notation for two-dimensional arrays that we showed in Figure 6.7a.

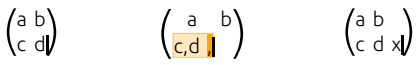
To create the initial matrix, the user can directly type `{{a,b},{c,d}}`. Note that before the user types the last `}`, the matrix visualization is not yet used. The visualization is automatically chosen by Envision (instead of the more generic array initializer visualization) only once two nested array initializers are detected, which happens when the user completes the outer initializer. This is shown in Figure 6.10.

(a) (b)

Figure 6.10: Typing a matrix (two-dimensional array) from left to right: before (a) and after (b) the final `}` is typed.

If the user prefers to use the specialized notation earlier, they may choose to initially type only `{{}}` thereby making Envision detect and show an empty matrix:

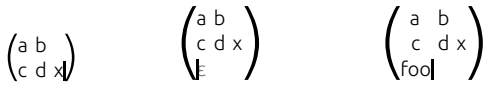
Once nested array initializers (a two-dimensional matrix) are detected and the specialized matrix visualization is shown, the user may continue editing using this specialized notation. Edits to individual elements of this notation work via linearization as described earlier in this section. For example, to add a new element `x`, the user needs to type "`,x`" after an existing element, or "`x,`" before an existing element, just like they would do in a text editor. This is illustrated in Figure 6.11.



(a) (b) (c)

Figure 6.11: Adding an element to a matrix row: before (a) and after (b) typing ",", and after typing "x" (c).

The handler of a complex (or indeed any) visualization may also provide additional interactions that bypass linearization and directly change the underlying AST in order to enable quick edits specific to the visualization. For example, if a user presses Enter in the matrix visualization, a new row is created under the currently selected one, as shown in Figure 6.12. Indeed, there is no other way to create new rows in the matrix visualization, since there is no cursor position that is equivalent to the space between the inner arrays of the nested array initializer underlying the matrix. A matrix visualization with such a cursor region could be designed and edits would



(a) (b) (c)

Figure 6.12: Adding a new row to a matrix: before (a) and after (b) pressing Enter, and after adding a new element `foo` to the row (c).

work using linearization as usual, but we believe the current behavior is preferable and provides a cleaner look.

As described above, custom visualizations for specific code constructs are only used when the corresponding construct is detected by Envision. Custom visualizations will be used for as long as their required code pattern (or other contextual requirements) are met. Some edits may eliminate required contextual conditions and cause Envision to use default visualizations once again. For example, if a cursor is placed just after a matrix visualization and the user presses Back←, this will delete the closing brace of the outer array initializer and there will no longer be two nested array initializers. As a consequence Envision will switch to using default visualizations. This is illustrated in Figure 6.13.

(a) (b)

Figure 6.13: Deleting the last brace of a two-dimensional array initializer shown in matrix form: before (a) and after (b) pressing Back←.

6.3 Evaluation

To evaluate the efficiency of edits in Envision, we used CogTool [JPSK04, BJRT10]. Our goal is to show that the speed of edits in Envision is comparable to text-based tools and, therefore, editing in Envision, remains an insignificant part of software development. Our evaluation resembles what Green and Petre did for their Cognitive Dimensions framework [GP96] when they measured the high viscosity of visual programming tools. CogTool is well suited for such an evaluation. In CogTool, one creates a model of several user interfaces and specifies how a task can be achieved using each one. The tool then uses a cognitive model to simulate the performance of expert users, who know how to achieve the given task without hesitation. The CogTool models used for this evaluation can be downloaded from [Env].

We modeled how three typical code editing tasks are accomplished in Envision and Eclipse, similarly to what other researchers have done in the past [BJRT10]. The steps to complete each task and the final results are assumed to be known to the developer — the programs just need to be edited without the need to deliberate.

The models assume correct text entry and bug-free coding. Next, we briefly describe the three tasks.

HelloWorld — This task involves the creation of a new project, a class within the project, and a simple `main` method that prints the string "Hello world". In Envision, we create the project, class, and the method using the command prompt and abbreviated commands. The call to the `println` method is created without using auto-completion. In Eclipse, we use the main menu to create the new project and a context menu to create the new class. We did not use the option to automatically create the `main` method in the class wizard dialog, as this is not a part of the usual Eclipse interactions and will skew the results. Envision does not have such a shortcut, but one could be easily added. When editing the `main` method, we used Eclipse's built-in code snippet facility to speed up writing. As in the Envision case, we did not use auto-completion.

Rocket — This task is an adaptation of a task from the work on Cognitive Dimensions of Green and Petre [GP96] and is originally based on work by Curtis et al. [CSKB+89]. The task is to modify a method that computes the trajectory of a rocket to account for air resistance. It requires the addition of five new statements to a method. Using this task, Green and Petre showed that visual programming environments can suffer from high viscosity: users of visual environments were up to 8 slower compared to users of a textual environment. We adapted the task to the Java programming language and modeled it in Eclipse and Envision. The actions required to complete the task in both tools are almost identical.

Tetris — In this task, a new shape is added to a game of Tetris. This consists of adding a new element to an existing enumeration and editing a three-dimensional array, a loop condition, and another array. In Envision, the three-dimensional array is edited as a matrix of arrays, and there is no need to manually insert space breaks to align array elements. The mini-map is used to navigate from one edit location to another. In Eclipse, space breaks are inserted to nicely align the elements in the three-dimensional array. Navigation between different edit locations is done via the file tabs and the scroll bar.

Results

The simulation results in Figure 6.14 show that for all tasks the differences in completion times between Eclipse and Envision are within the empirically observed error margin of ±10% in CogTool's underlying KLM cognitive model. The results indicate that editing code in Envision is as fast as in a text editor, and therefore remains an insignificant part of software development. We are not aware of another visual code editor that provides such efficient interactions and offers visualizations as flexible as the ones in Envision.

Our own experience with editing code in Envision

We have some limited experience with editing code using Envision, mostly from testing the implementation and developing toy examples. Generally, writing code feels natural and quick. The automatic formatting and code layout is a welcome change compared to text, as it allows us to directly type without worrying about the whitespace within expressions. Creating visual structures (e.g., for loops) by just typing their keyword is easy to get accustomed to and is also convenient because it

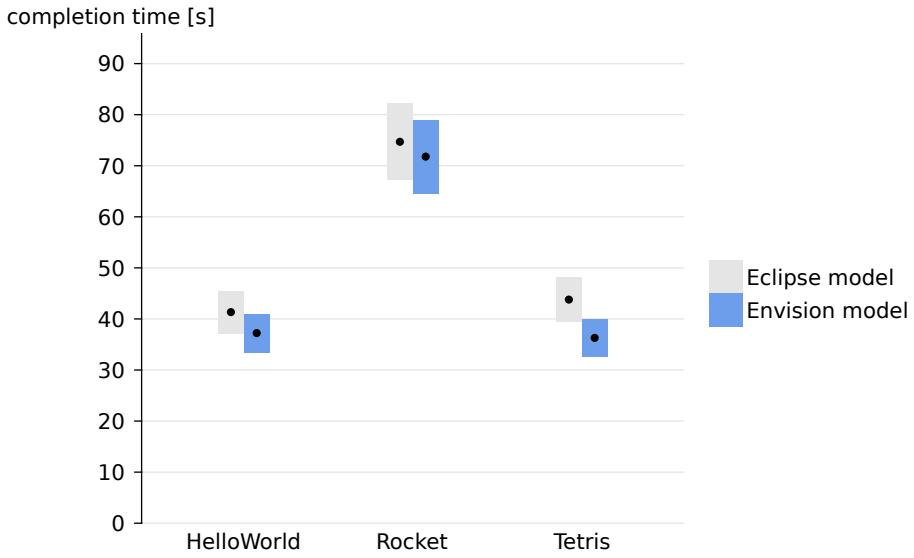


Figure 6.14: CogTool estimations of task completion times. The boxes indicate a ± 10% range of each estimated time.

automatically establishes a visual scope.

The implementation suffers from two main problems that prevent long sessions of editing. First, Envision can occasionally crash due to incorrectly handled input. This happens primarily with infrequently used visualizations whose interaction handlers are not well tested and contain bugs. Second, custom visualizations are sometimes lacking specific interactions that would be useful for a specific case, making it hard to perform that edit. Both of these issues can be resolved if more development time is invested in polishing the implementation of interactions in Envision.

Limitations

Below we discuss four limitations of our evaluation and our interaction approach.

We have evaluated Envision's interaction mechanisms on only three CogTool models. The performance might differ on models of other programming tasks or in a user study. To improve the relevance of the simulations we modeled long tasks that test a combination of all edit interactions. The empirical results also match our own experience of using Envision, so we speculate that a user study with experienced Envision users would yield similar results.

Envision's interaction design enables a diverse set of programming notations and interfaces. However, in some cases there is no single best way to define more complex interaction flows. For example, when editing non-linear expression visualizations, such as the one from Figure 6.9, it is not always clear how the graphical visualizations should be linearized. In the discussion of Figure 6.9, we mentioned three possibilities for configuring the virtual grid and cursor regions and different users might have different preferences. While Envision supports all these configurations, each must be separately implemented in plug-ins, which is beyond what we expect of ordinary users of the system. Thus, designers of complex visualizations need to provide reasonable

default interactions and perhaps anticipate different interaction preferences in order to provide easily configurable alternatives through styles.

The support for complex and editable visualizations may also hinder the performance of inexperienced users. Unlike plain text, where it is clear how to achieve any edit, users of complex visualizations might be unaware of what interactions are available or how to achieve an edit efficiently. While a complex visualization may enable left-to-right typing and edits similar to text, oftentimes it can provide additional interactions specific to its domain to make certain tasks simpler. Users unaware of these interactions might struggle. Further research is needed to make interactions with complex visualizations more transparent.

Generally, most interaction customizations can only be performed via Envision plug-ins. Such customizations need to be implemented by power users or Envision developers with knowledge of Envision's interaction APIs. A further inconvenience with customizations is that, unlike standard text-editing, Envision's interactions mechanisms are more complex, which makes interaction customizations harder to debug and prone to bugs. In order to allow wider and more robust personalization of the programming environment, it is necessary to develop additional mechanisms that enable a larger set of customizations to be performed by ordinary users in a more controlled manner, without the need to write Envision plug-ins.

6.4 Related work

Miller et al. [MPMV94] describe early structured editors used in education. Early systems such as the Cornell Program Synthesizer [TR81], Gandalf [HN86], or Gnome [GM84] rendered programs in a textual form that looked like a standard programming language syntax, but these editors eliminated syntax errors by imposing constraints on how the program is manipulated: edits are always made in a top-down fashion according to the program's AST and invalid AST states are impossible. Navigation was also performed along AST nodes as opposed to the text structure on screen. This navigation mode meant that students need to map from movements on the text to movements in terms of AST nodes, which further reduced usability. Such interactions may be admissible when educating novice programmers, but they are prohibitive for advanced developers. In particular, the constraint of always syntactically-correct programs is severe, because it prohibits quick edits via invalid states, an edit pattern developers frequently use: Omar et al. [OVH+16] report that in data of character-level edits collected over 1460 hours by Yoon and Myers [YM14], 44.2% of the states of source files were not syntactically valid. Structured editors, such as MacGnome [CGG+85] or ACSE [PM93], addressed some of these usability issues by focusing more on mouse-based interaction (e.g., for navigation) and by allowing the program to be edited also using a standard text-editor, thus also negating some of the benefits of structured editing. Unlike Envision, none of these editors were aimed at expert programmers.

Ko et al. [KAM05b] analyzed the code edit patterns of Java programmers and found that the full flexibility of text editors is not used. They proposed a set of common code edit patterns that, if supported by structured editors, might provide good usability. The MPS [Fow05, VSBK14] commercial programming system features a modern structured editor, which can be considered to support specific interactions as suggested by Ko et al. MPS offers many features that improve usability, but

editing code in MPS is constrained to valid ASTs. Users of MPS report that editing feels unfamiliar at first and takes significant time to learn. More proficient MPS users are generally satisfied with the usability of the tool, but still experience situations where the constrained editing imposes specific ways to write code, for example, classes or interfaces need to be defined before they can be used. Intentional software [SCC06] is another commercial programming system with a structured editor, which, to the best of our knowledge, seems to work similarly to the one in MPS. In contrast to Envision, which targets mainstream programming languages, both MPS and Intentional software are language workbenches, which take a non-standard approach to software engineering in which programmers define and compose their own domain-specific languages.

Barista [KM06] is an implementation framework for creating visual code editors. It has features similar to Envision, such as augmenting code with HTML documentation, the ability to present source code in different ways, keyboard navigation, and text-like editing. Our work goes beyond Barista in a number of important aspects. Firstly, not all visualizations in Barista are editable. The tool's authors demonstrate pretty-printed views of mathematical operators, but for editing revert to visualizations that closely match the tokens of the concrete language. This is not necessary in Envision, since its interaction framework enables any visualization to be directly editable, provided the visualization's designer implements the appropriate handlers. Secondly, we focus strongly on customizability, through explicit mechanisms in the existing implementation and through our plug-in based architecture. Thirdly, to our knowledge Barista has not been evaluated.

LabView is perhaps the most successful visual programming tool used by professionals. Unlike Envision, LabView targets the domain of measurement automation and control systems, and is based on a non-mainstream, data-flow driven programming model. LabView and other visual programming environments have been studied extensively by Green and Petre [Gre89, Gre90, GP96], who identify the major usability issue of high viscosity. For example, due to high viscosity in LabView it might be difficult to modify an existing expression, because the developer has to rearrange existing subexpressions on the visual canvas. Visual programming environments typically have a strong focus on mouse-based interactions, in contrast to Envision.

Many visual programming tools such as Alice [Coo10], Scratch [MRR$^+$10], MIT App Inventor [Wol11], and JPie [BG05] are designed for students learning to program. Such tools focus on easily constructing executable programs via visual means and displaying a live execution while programming. Users can use drag and drop gestures to put different program fragments together, which eliminates most if not all syntactical errors. Unlike tools designed for experts such as Envision, visual tools for novices typically have no strong focus on keyboard interactions, freedom of editing the program via invalid states, or customizability.

The Stride structured editor [McK12, PBL$^+$16], which is part of the Greenfoot tool, has been designed to help novices program in a language that is semantically identical to Java. Evaluations show that editing code in Stride is at least as fast as editing in a standard text editor. Unlike Envision, Stride uses a specific visual structure to present code and does not support arbitrary editable visualizations.

Eco [DT14] is a recent tool that allows language composition, which is achieved by using an editor based on language boxes. Within a language box developers can freely type text corresponding to one language and also insert language boxes

of other languages. Each box is edited independently of other boxes and is not restricted to a text editor. Unlike Eco, Envision makes it possible to customize the appearance and edit interactions of individual language constructs, not just of entire languages.

Code Bubbles [BZR+10, BRZ+10], Code Canvas [DR10, DVR10], and Debugger Canvas [DBR+12] are recent programming tools that visually enhance professional IDEs. Building on the well-established Eclipse and Visual Studio platforms, these tools provide alternative ways to navigate code in a two-dimensional canvas, but neither provides a structured editor or rich and flexible visualizations of arbitrary code fragments.

Another big group of visual programming tools are applications designed for end-users, for example, spreadsheets. While the interfaces of such systems can be very efficient, these environments mostly use their own programming model and workflows and lack support for general software engineering.

Precise version control of tree structures

7

In the previous two chapters, we addressed the issue of limited notations of mainstream programming tools. We presented flexible visual interfaces that are decoupled from the underlying program model. This decoupling allows us to enrich the information structures of source files, and in particular to include dense data. In this chapter we investigate a specific application of dense data – improving version control algorithms for trees.

Version control of tree structures, ubiquitous in software engineering, is typically performed on a textual encoding of the trees, disregarding the tree structure. Applying standard line-based diff and merge algorithms to such encodings leads to inaccurate diffs, unnecessary conflicts, and incorrect merges. To address these problems, we have designed novel algorithms for computing precise diffs between two versions of a tree and for three-way merging of trees [AGMO17]. Unlike most other approaches for version control of structured data, our approach integrates with mainstream version control systems. Our merge algorithm can be customized for specific application domains to further improve merge results. An evaluation of our approach on abstract syntax trees from popular Java projects shows substantially improved merge results compared to Git. While the algorithms we present can be used with traditional source files, the algorithms are much more efficient and produce better diffs and merges with source files that include dense data.

7.1 Challenges of versioning trees

Tree structures such as XML, JSON, and source code are ubiquitous in software engineering, but support for precise version control of trees is lacking. Mainstream version control systems (VCSs) such as Git, Mercurial, and SVN treat all data as sequences of lines of text. Standard diff and merge algorithms disregard the structure of the data they manipulate, which has three major drawbacks for versioning trees. First, standard line-based diff algorithms may lead to *inaccurate and confusing diffs*, for instance when differences in formatting (e.g., added indentation) blend with real changes or when lines are incorrectly matched across different sub-trees (e.g., across method boundaries in a program, as illustrated in Figure 7.1). Inaccurate diffs do not only waste developers' time, but may also corrupt the result of subsequent merge operations. Second, standard merge algorithms may lead to *unnecessary conflicts*, which occur for incompatible changes to the formatting (e.g., breaking a line at different places), but also for more substantial changes such as merging two revisions that each add an element to an un-ordered list (for instance, a method at the end of

```
-InformationEdge* Graph::addDirectedEdge(InformationNode* from, InformationNode* to, const QString& name)
+InformationEdge* Graph::addEdge(InformationNode* from, InformationNode* to, const QString& name,
+                                InformationEdge::Orientation orientation)
 {
    // We only allow existing nodes:
    Q_ASSERT(from == findNode(from));
    Q_ASSERT(to == findNode(to));
+   // TODO this might be a possible performance hit if we have many edges.
    for (auto edge : edges_)
    {
-      if (edge->from() == from && edge->to() == to && edge->name() == name)
+      // TODO if directed only this condition
+      bool directedMatch = edge->from() == from && edge->to() == to;
+      bool undirectedMatch = edge->from() == to && edge->to() == from;
+      if (edge->name() == name && ((orientation == InformationEdge::Orientation::Directed && directedMatch)
+                      || (orientation == InformationEdge::Orientation::Directed && (directedMatch || undirectedMatch))))
       {
          auto existingEdge = edge;
-         Q_ASSERT(existingEdge->isDirected());
+         Q_ASSERT(existingEdge->orientation() == orientation);
          existingEdge->incrementCount();
          return existingEdge;
       }
    }
-   auto edge = new InformationEdge(from, to, name);
-   edges_.push_back(edge);
-   return edge;
-}
-
-InformationEdge* Graph::addEdge(InformationNode* a, InformationNode* b, const QString& name)
-{
-   // We only allow existing nodes:
-   Q_ASSERT(a == findNode(a));
-   Q_ASSERT(b == findNode(b));
-   for (auto edge : edges_)
-   {
-      if ((edge->from() == a || edge->from() == b) && (edge->to() == b || edge->to() == a) && edge->name() == name)
-      {
-         auto existingEdge = edge;
-         Q_ASSERT(!existingEdge->isDirected());
-         existingEdge->incrementCount();
-         return existingEdge;
-      }
-   }
-   auto edge = new InformationEdge(a, b, name, InformationEdge::Orientation::Undirected);
+   auto edge = new InformationEdge(from, to, name, orientation);
    edges_.push_back(edge);
    return edge;
 }
```

Figure 7.1: A confusing diff shown by GitHub. On first sight, the large red section in middle gives the impression that the developer removed this code, including the closing brace, to merge the two `addEdge` methods into one. However, upon closer inspection one notices that the last two statements of both methods are identical (shown in blue boxes). A more intuitive diff would show the bottom method as fully deleted, the second to last statement of the top method as modified, and the last two statements of the top method, including the closing brace, as unchanged.

the same class). Unnecessary conflicts could be merged automatically, but instead require manual intervention from the developer. Third, standard merge algorithms may lead to *incorrect merges*; for instance, if two developers move the same tree node to two different places, a line-based merge might incorrectly duplicate the node. Incorrect merges lead to errors that developers need to detect and fix manually.

To solve these problems, we have developed a novel approach to versioning trees. Our approach builds on a standard line-based VCS (in our case, Git), but provides diff and merge algorithms that utilize the tree structure to provide accurate diffs, conflict detection, and merging. In contrast to VCSs that require a dedicated backend for trees [KH10, KHvWH10, MCPW08, NMB05, NNPN10], employing a standard VCS allows developers to use established infrastructures and workflows (such as GitHub or BitBucket) and to version trees and text files such as documentation in the same VCS. Our diff algorithm relies on the optimized line-based diff of the underlying VCS, but utilizes the tree structure to accurately report changes, including moved sub-trees, which line-based algorithms do not track as illustrated in Figure 7.2. Building on the diff algorithm, we designed a three-way merge algorithm that reduces unnecessary conflicts and incorrect merges by using the tree structure and, optionally, domain knowledge such as whether the order of elements in a list is relevant.

Diff and merge algorithms rely on matching different revisions of a tree to recognize commonalities and changes. The optimal way to obtain such a matching is to associate each tree node with a unique ID, which is dense data that remains unchanged across revisions. This approach yields precise matchings and makes it very efficient to recognize changed and moved nodes, but requires a custom storage format that supports dense data and a corresponding editor such as Envision. Alternatively, one can use traditional textual encodings of trees without IDs (e.g., source code to represent an AST) and compute matchings using an algorithm such as ChangeDistiller [FWPG07] or GumTree [FMB+14]. However, such algorithms require significant time and produce results that are approximate and, thus, lead to less precise diffs and merges. Our approach supports both options; it benefits from the precise matchings provided by node IDs when available, but can also utilize the results of matching algorithms and, thus, be used with standard editors. We will present the approach for a storage format that includes node IDs, but our evaluation shows that even with approximate matchings computed on standard Java code, our approach achieves substantially better results than a standard line-based merge.

In the remainder of this chapter, we will first present a textual encoding of generic trees that includes node IDs and that enables precise version control of trees within standard line-based VCSs such as Git. Afterwards, we will describe an algorithm for computing the difference between two versions of a tree based on the diff reported by a line-based VCS. Following the diff algorithm, we will describe an algorithm for a three-way merge of trees, which allows the customization of conflict detection and resolution. Finally, we will discuss an evaluation of the algorithms on several popular open-source Java code bases.

7.2 Tree versioning with a line-based VCS

The algorithms we designed work on a general tree structure. In order to enable precise version control of trees, we assume, without loss of generality, that each tree

```
-bool Commit::getFileContent(QString fileName, const char*& content, int& contentSize) const
+bool Commit::getFileContent(QString fileName, const char*& content, int& contentSize, bool exactFileNameMatching) const
 {
-   QHash<QString, CommitFile*>::const_iterator iter = files_.find(fileName);
-   if (iter != files_.constEnd())
+   // name of file must match fileName exactly
+   if (exactFileNameMatching)
    {
-      contentSize = iter.value()->size_;
-      content = iter.value()->content();
+      QHash<QString, CommitFile*>::const_iterator iter = files_.find(fileName);
+      if (iter != files_.constEnd())
+      {
+         contentSize = iter.value()->size_;
+         content = iter.value()->content();

-      return true;
+         return true;
+      }
    }
+   // name of file contains fileName
    else
-      return false;
+   {
+      QHash<QString, CommitFile*>::const_iterator iter = files_.constBegin();
+      while (iter != files_.constEnd())
+      {
+         if (iter.key().contains(fileName))
+         {
+            contentSize = iter.value()->size_;
+            content = iter.value()->content();
+
+            return true;
+         }
+         iter++;
+      }
+   }
+   return false;
 }
```

Figure 7.2: An undetected move making a diff harder to understand. At first sight, it seems that the entire method body was replaced with new code. In fact, most of the old body was simply moved into a newly inserted if-statement.

node is a tuple with the following elements:

- *id*: a globally unique ID. This ID is used to match and compare nodes from different versions and track node movement. IDs can be stored on disk with source files that support dense data, or generated on demand for traditional source files, as long as matching nodes from different versions have the same ID, which can be achieved by using a tree-matching algorithm such as GumTree [FMB+14]. We use a standard 128-bit universally unique identifier (UUID).

- *parentId*: the ID of the parent node. The parent ID of the root node is a null UUID. All other nodes must have a parentId that matches the ID of an existing node.

- *label*: a name that is unique among sibling nodes. The label is essentially the name of the edge from parent to child node. This could be any string or number, e.g., $1, 2, 3, \ldots$ for the children of nodes representing lists.
- *type*: an arbitrary type name from the target domain. For example, types of AST nodes could be `Method` or `IntegerLiteral`. Types enable additional customization of the version control algorithms, used to improve conflict detection and resolution. In domains without different node types, one type can be used for all nodes.
- *value*: an optional value, e.g., the text of string literals.

A *valid tree* is a set of nodes which form a tree and meet the requirements above.

7.2.1 Textual encoding of valid trees

In order to efficiently perform version control of trees within a line-based VCS, we encode trees in a specific text format, which enables using the existing line-based diff in the first of two stages for computing the changes between two tree versions. A valid tree is encoded into text files as illustrated in Figure 7.3. The key property of the encoding is that a single line contains the encoding of exactly one tree node with all its elements. In Figure 7.3, each line encodes a node's label, type, UUID, the UUID of the parent node, and the optional value in that order. A reserved node type *External* indicates that a subtree is stored in a different file (Figure 7.3b).

This encoding allows two versions of a set of files to be efficiently compared using a standard line-based diff. The different lines reported by such a diff correspond directly to a set of nodes that is guaranteed to be an overapproximation of the nodes that have changed between the two versions of the encoded tree.

The order of lines within a file and their indentation has no meaning, but for efficient parsing, we indent each child node and insert children after their parents (Figure 7.3), enabling simple stack-based parsing. The names of the files that comprise a single tree is irrelevant, but for quickly finding subtrees, it is advisable to include the UUID of the root of each file's subtree in the file name.

```
2 Method {9c2c..} {e0b6..}
  modifiers Modifier {8842..} {9c2c..} 1
  name Name {3269..} {9c2c..} foo
  body StatementList {1023..} {9c2c..}
    0 If {f3c2..} {1023..}
      condition BinOp {b0a0..} {f3c2..}
        left Text {f7c3..} {b0a0..} two\nlines
```

(a)

```
12 Class {5414..} {425d..}
  methods List {e0b6..} {5414..}
    0 External {e239..} {e0b6..}
    1 External {5db1..} {e0b6..}
    2 External {9c2c..} {e0b6..}
```

(b)

Figure 7.3: (a) An example encoding of an AST fragment in Envision's file format. For brevity, only the first 2 bytes of UUIDs are shown here. (b) A file that references external files, which contain the subtrees of a class's methods. The last line refers to the file from (a). At most two lines in different files may have the same ID, and one of them must be of type *External*.

7.2.2 Diff algorithm

The diff algorithm computes the delta between two versions of a tree (T_{old} and T_{new}). The delta is a set of changes, where each change represents the evolution of one node and is a tuple consisting of:

- *oldNode*: the node tuple from T_{old}, if it exists (node was not inserted).
- *newNode*: the node tuple from T_{new}, if it exists (node was not deleted).
- *kind*: the kind of the change – one of *Insertion*, *Deletion*, *Move* (change of parent and possibly label, type, or value), *Stationary* (no change of parent, but change in at least one of label, type, or value).

These elements provide the full information necessary to report precisely how a node has changed. The encoding from Section 7.2.1 enables an efficient two-stage algorithm for computing the delta between two versions of a tree. The operation of the algorithm is illustrated in Figure 7.4.

The **first stage** computes two sets of nodes $oldNodes \subseteq T_{old}$ and $newNodes \subseteq T_{new}$, which overapproximate the nodes that have changed between T_{old} and T_{new}. The sets are computed by comparing the encodings of T_{old} and T_{new} using a standard line-based diff [MM85, Mye86, Ukk85]. Given two text files, a line-based diff computes a longest common subsequence (LCS), where each line is treated as an atomic element. The LCS is a subset of all identical lines between T_{old} and T_{new}. The diff outputs the lines that are not in the LCS, thus overapproximating changes: lines from the "old" file are marked as deleted and lines from the "new" file are marked as inserted. In the middle of Figure 7.4, lines B, E, G, H, and D on the left are marked as removed and lines B', E', G, H, and X on the right are marked as inserted. The combined diff output for all files is two sets of removed and inserted lines. The nodes corresponding to these two sets, ignoring nodes of type External, are the inputs to the second stage of the diff algorithm.

The **second stage** (Algorithm 7.2.1) filters the overapproximated nodes and computes the final, precise delta between T_{old} and T_{new}. The algorithm essentially compares nodes with the same id from *oldNodes* and *newNodes* and if they are different, adds a corresponding change to the delta. A node from oldNodes might be identical to a node from newNodes, for example, if its corresponding line has moved,

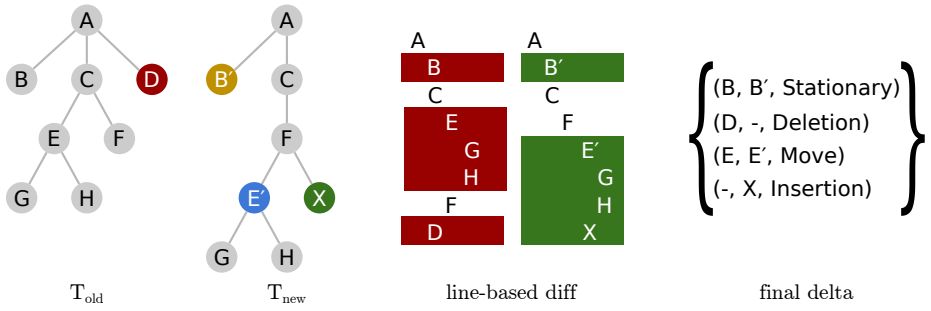


Figure 7.4: A tree modification and the outputs of the two stages of the diff algorithm.

```
1: function TreeDiffStageTwo(oldNodes, newNodes)
2:     changes ← ∅
3:     for all {(old, new) ∈ (oldNodes × newNodes) | old.id=new.id ∧ old≠new}
   do
4:         if old.parentId = new.parentId then
5:             changes ← changes ∪ {(old, new, Stationary)}
6:         else
7:             changes ← changes ∪ {( old, new, Move)}
8:         end
9:     end
10:    for all {old ∈ oldNodes | old.id ∉ IDs(newNodes)} do
11:        changes ← changes ∪ {(old, NIL, Deletion)}
12:    end
13:    for all {new ∈ newNodes | new.id ∉ IDs(oldNodes)} do
14:        changes ← changes ∪ {(NIL, new, Insertion)}
15:    end
16:    return changes
17: end
```

Algorithm 7.2.1: The second stage of the *TreeDiff* algorithm. IDs is the set of all identifiers of nodes from the input set. A more detailed version of this algorithm and a proof of correctness can be found in [Gue15].

but is otherwise unchanged. This is the case for nodes G and H in Figure 7.4, where the final delta consist only of real changes to the nodes B, D, E, an X. This is in contrast to a line-based diff, which will also report G and H as changed, even though they have not.

In the absence of unique IDs stored with the tree, it is possible to compute matching nodes using a tree match algorithm, enabling our diff and merge algorithms to be used for traditional encodings of trees, such as Java files. To achieve this, the first stage needs to be replaced so that it parses the input files, computes a tree matching, and assigns new IDs according to the matching. However, this will greatly reduce the efficiency of the diff, and will reduce the precision of the final delta, since the node matchings are heuristic.

The described diff algorithm eliminates (with unique IDs), or greatly reduces (using a tree matching algorithm) inaccurate diffs. This is because the formatting of the encoding is irrelevant, changes are expressed in term of tree nodes, and moved nodes are tracked, even across files. The diff provides a basis for improved merges, discussed next.

7.3 Merging trees

Building on the diff algorithm from Section 7.2.2, we designed an algorithm for merging two tree revisions T_A and T_B given their common ancestor T_{base}. At the core of the merge is the change graph – a graph of changes performed by the two revisions, which includes conflicts and dependencies. In this section, we describe the change graph and how it is used to merge generic trees; in Section 7.4, we will outline

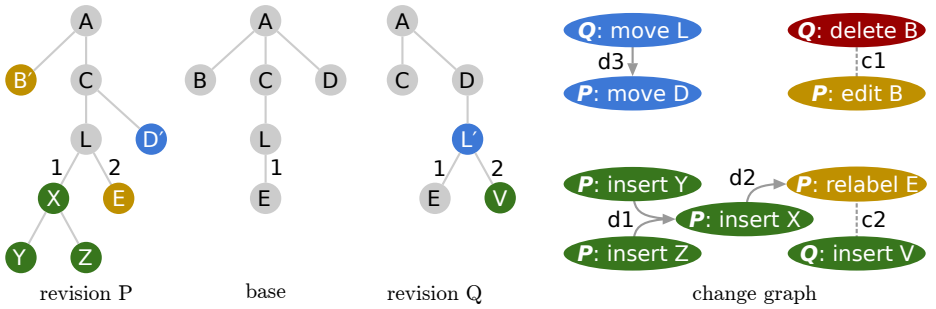


Figure 7.5: A base tree with two modifying revisions and the corresponding CG. Each edge in the CG is labeled with the dependency or conflict type that the edge represents.

additional merge customizations, which use knowledge about the domain of a tree to improve conflict detection and resolution. Unlike the diff algorithm, the merge does not build on its line-based analog, which is unaware of the tree structure and may produce invalid results. For example, if two revisions move the same node (line) to two different parents, which are located in different parts of a file or in different files, a line-based algorithm would simply keep both lines, incorrectly duplicating the subtree, whereas our algorithm will report a conflict.

7.3.1 Change graph and merge algorithm

The purpose of the *change graph* (CG) is to bring together changes from two diverging revisions and facilitate the creation of a merged tree. The nodes of the CG are changes, similar to the ones reported by the diff. The changes are connected with two types of edges, which constrain when changes may be applied. A change may require another change to be applied first, expressed as a directed *dependency edge*. For example, a change inserting a node might depend on the change inserting the parent node. Two changes may be in conflict with each other, expressed as an undirected *conflict edge*. For example, if both revisions change the same node differently, these changes will be in conflict. An example change graph is illustrated in Figure 7.5.

To merge T_A and T_B into a tree T_{merged}, first an inverse topological ordering of the CG is computed using the dependency edges. Changes are applied according to this ordering, if possible. A change is applicable if it does not depend on any other change and has no conflict edges. Changes that form cycles in the CG may be applied together, in one atomic step, provided that all changes (i) have no conflict edges; (ii) are made by the same revision or by both revisions simultaneously; and (iii) do not depend on any change outside the cycle. Essentially, changes in such cycles are independent of other changes and are compatible with both revisions, making them safe to apply. These restrictions ensure that each application of a change preserves the validity of the tree. Applied changes are removed from the CG along with any incoming dependency edges. Once all applicable changes have been applied, any remaining changes represent conflicts and will be reported to the user. Next, we explain how the CG is constructed.

Merge changes

A *merge change* is a tuple that extends the change tuple from the diff algorithm with one new element, *revisions*, which indicates which revisions make this change: *RevA*, *RevB*, or *Both*. The nodes of the CG are the merge changes obtained by running the diff algorithm twice to compute the delta between T_{base} and T_A and between T_{base} and T_B, respectively. First, each change from the two deltas is associated with either *RevA* or *RevB* to create a corresponding merge change. Then, we organize the elements of a tree node into two *element groups*: (i) parent and label; and (ii) type and value. Each group contains tuple elements whose modification by different revisions is a conflict. Any merge changes that modify both element groups are split into two merge changes: one for each element group. For example, if a node is moved to a new parent and its value is modified, this will appear as two separate and independent merge changes within the CG. This separation reduces conflicts and dependencies in the CG, since the two groups are independent. Finally, any identical changes made by different revisions are combined into a single merge change with *revisions=Both*, which ensures that identical changes are applied only once.

Dependencies between merge changes

A dependency $X \rightarrow Y$ means that change X cannot be applied before Y, and is the first of two means that restrict applicable changes. Dependencies prevent three cases of tree structure violations.

(d1) orphan nodes: (a) Before a change IM inserts or moves a node N, N's parent destination node P must exist. If P does not already exist in T_{base}, then there must be an insertion change I, which inserts it. An edge $IM \rightarrow I$ is added to indicate that I must be applied before IM can be applied. In Figure 7.5, nodes Y and Z depend on the insertion of X. (b) Before a change D deletes a node N, all of N's children must be deleted or moved. An edge $D \rightarrow DM$ is added between D and each change DM that moves or deletes a child of N. In both (a) and (b), the changes I and DM are guaranteed to exist if they are necessary, because the merge changes were computed from the deltas of valid trees. Note that dependencies that prevent orphan nodes cannot form cycles on their own.

(d2) clashing labels: Before a change IMR inserts, moves, or relabels (modifies the label of) a node N, there must be no sibling at the destination of N with the same label. If a node with the same label as N's final label exists in T_{base} at the destination parent of N then there must be a change DMR that deletes, moves, or relabels that sibling. An edge $IMR \rightarrow DMR$ is added to the CG. In Figure 7.5, node X depends on the relabeling of E. Such dependencies may form cycles. For example, swapping two elements in a list yields two relabel changes, where each change depends on the other.

(d3) cycles: If a change M_N moves a node N, N must not become its own ancestor. Such a situation occurs, for example, if a revision A moves an if-statement IF_1 into an if-statement IF_2, and revision B moves IF_2 into IF_1. To prevent such issues, move changes are applied only if the destination subtree does not need to be moved. This is enforced using dependencies. If M_N moves N to a subtree that needs to be moved, let M_P be the change that moves the subtree. An edge $M_N \rightarrow M_P$ is added to the CG. Move changes from different revisions may create dependency chains that form a cycle in the CG. For example, the move of IF_1 will depend on the move of IF_2, which will itself depend on the move of IF_1. Such a cycle means

that the two revisions perform incompatible moves and the changes from the cycle cannot be applied. Move changes from different revisions do not always result in a cycle. For example, in Figure 7.5, the move of L depends on the move of D, which is independent.

Conflicting merge changes

Conflicts that would result in a node becoming its own ancestor are indirectly represented in the CG in the form of dependency cycles described above. Other conflicts cannot be expressed with dependencies and appear directly as conflict edges, which are the second means for restricting change application. There are three cases of direct conflicts.

(c1) same node: If two revisions make non-identical changes X and Y to the same node, these changes may be conflicting. Deletions conflict with all other changes. Other changes conflict only with changes of the same element group. Conflicting changes are connected with an undirected edge $X \sim Y$ in the CG. An example of such a conflict is the modification and deletion of B in Figure 7.5.

(c2) label clash: If a change IMR_N inserts, moves, or relabels a node N, and another change IMR_Q inserts, moves, or relabels a node Q such that N and Q have identical final labels and parent nodes, the two changes are in conflict. An edge $IMR_N \sim IMR_Q$ is added to the CG. In Figure 7.5, such a conflict is the relabeling of E and the insertion of V.

(c3) deletion clash: If a change D_N deletes a node N, and another change IM_Q inserts or moves a node Q as a child of N, the two are in conflict. An edge $D_N \sim IM_Q$ is added to the CG.

Validity of merge changes application

Below we argue why applying a change from the CG always results in a valid tree. We justify each of the properties of a valid tree separately:

- **One root, no other nodes without a parent:** Without loss of generality, we assume that the root node of a tree is never moved and thus, remains the root node in all revisions. Move and Insert changes require a non-null ID as the target parent node, and thus cannot introduce an additional root. Edges of types **d1** and **c3** ensure that no nodes are left without a parent when inserting or deleting nodes.

- **No cycles:** For a cycle to occur in a tree, a change would have to move a node into its own subtree. Dependencies of type **c3** effectively prohibit this. Within a single revision, these dependencies dictate a specific change application order, which guarantees that at no step there are cycles in the tree. With two tree revisions, conflicting changes might exist such that there is no application order that prevents a cycle. In such cases, would-be tree cycles correspond to cycles in the CG that prevent the corresponding changes from being applied.

- **One parent per node, except for the root:** Each tree revision includes at most one change that may affect the parent of a node. If both revisions have changes affecting the parent of a node, these changes will either be merged, if they are identical, or form a conflict of type **c1** and neither will be applied. Thus a node may not accidentally get more than one parent.

- **Unique IDs:** Only insertions introduce new node IDs. Two insertion changes for the same ID must come from different revisions, and are either merged into a single merge change, if they are identical, or are marked with a conflict of type **c1**. Thus, there cannot be two different nodes with the same ID.
- **Unique labels for siblings:** Within a single revision, dependencies of type **d2** prevent changes from introducing a duplicate label. With two revisions, at most two changes introduce the same label within a parent node. If these changes pertain to the same child node, they are either identical and therefore merged, or they are conflicting according to **c1**. If these changes pertain to different child nodes, they are conflicting according to **c2**.

In contrast to line-based merges, applying changes using the CG prevents incorrect merges by taking the tree structure into account. The algorithm, as described so far, does not have any knowledge about the domain of the tree, and misses opportunities for improved merges and better error reporting. Next, we explain customizations, that improve the merge results and inform the developer of potential semantic issues.

7.4 Domain-specific merge customizations

Merging two tree revisions without any domain knowledge, as described so far, can lead to suboptimal merges. Figures 7.5 and 7.6 illustrate one such example, where revision P inserts a node X in the beginning of list L and revision Q inserts a node V at the end. These two changes conflict, because the label of E in P is identical to the label of V in Q. Despite this conflict, intuitively these changes can be merged by relabeling V. To achieve better merge results, we allow the merge process to be customized by taking domain knowledge into account. Customizations use domain knowledge, such as the semantics of specific node types or values, to tweak the CG, eliminating conflicts and dependencies, and thus, enabling additional changes to be applied. Customizations may also produce *review items*, which are messages that inform the user of a potential semantic issue with the final merge. Review items have two advantages over conflicts. First, unlike changes in a conflict or their depending changes (even if not in a conflict), which are not applicable, review items are not part of the CG and do not prevent the application of changes. Applying more

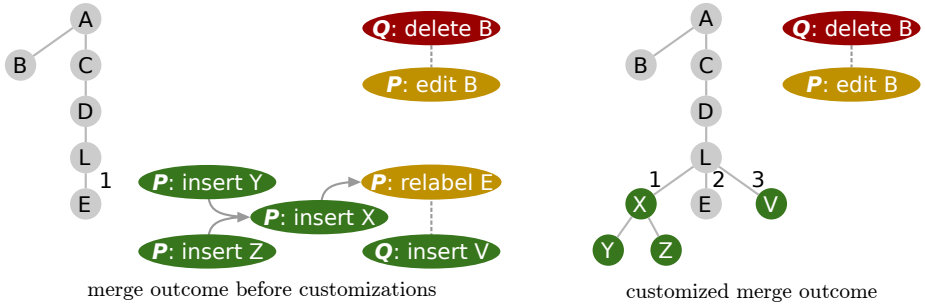


Figure 7.6: The resulting T_{merged} and CG after applying all possible changes from Figure 7.5 (left) and after additional customizations (right).

changes is desirable because the final merge more closely represents both revisions and the user has to review issues with only a selected group of nodes, instead of manually exploring many unapplied changes. Second, review items provide semantic information to the user, making it easier to take corrective action, unlike conflicts, which represent generic constraints on the tree structure. Similarly, review items are preferable to conflicts in line-based merges, because review items are more focused and provide semantic information. Next, we present two examples of customizations, which we have found useful for achieving high-quality merges.

7.4.1 List-merge customization

Data from many domains (e.g., ASTs, UML models) has list entities. Merging lists is challenging [KK14, SUW15], as it is not trivial to determine the order of the merged elements and to detect and resolve conflicts. In addition, the CG often contains label clash conflicts in lists (e.g., for nodes E and V in Figure 7.6), which are usually easy to resolve automatically. We developed the List-merge customization, which is crucial for merging list nodes well. Essentially, the customization computes a total order of all list elements from both revisions. This total order is used to relabel all elements, giving each element a unique label. Thus, all conflicts or dependencies due to previously clashing labels are removed from the CG, allowing many more changes to be merged. Next, we describe at a high-level the computation of the total order and how ambiguities are handled.

The total order is computed in three steps as illustrated in Figure 7.7. In the **first step**, a three-way longest common subsequence (LCS) between L_{Base}, L_A, and L_B is computed and used to create an alternating sequence of stable and unstable chunks. The *stable chunks* are a partition of the LCS – elements in a single chunk are adjacent in all lists. There are two stable chunks in Figure 7.7: $[A]$ and $[D]$. An *unstable chunk* consists of one element span per list, each span containing elements that are not in the LCS. There are two unstable chunks in Figure 7.7: $[XB, BC, CY]$ and $[V, \epsilon, W]$. Elements from different chunks are totally ordered using the order of the chunks, e.g., A before X and D. Elements from the same stable chunk are totally ordered using their order within the chunk. In the **second step**, for each unstable chunk C, two two-way LCSs $lcs_a = LCS(C_{base}, C_a)$ and $lcs_b = LCS(C_{base}, C_b)$ are computed. Elements from lcs_a are totally ordered with respect to elements from lcs_b using the order in C_{base}. In Figure 7.7 these are B and C. The remaining elements from C_a and C_b are ordered with respect to elements from lcs_a and lcs_b,

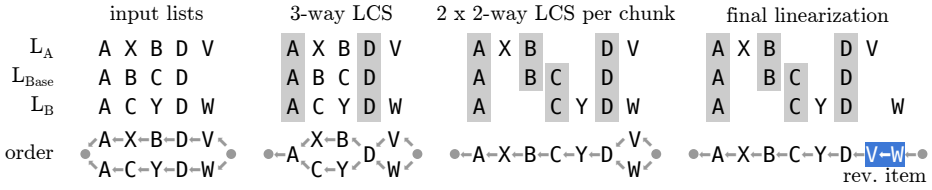


Figure 7.7: Computing a total order for the elements of a merged list. Stable chunks have a light-gray background. At the end, V and W are linearized and added to a review item. In the merged list, B and C will be removed to reflect changes from revisions.

respectively. Such elements are totally ordered using the order from one revision, if there are no elements from the other revision in the corresponding chunk (X and Y in Figure 7.7). Otherwise, the elements are not totally ordered (V and W in Figure 7.7). In the **third step**, unordered elements are linearized in an arbitrary order. If the list represents an ordered collection within the domain, a review item is created to inform the user of the ambiguity. The detailed pseudocode of this algorithm is shown in Algorithm 7.4.1, which we explain next.

The final total order of the elements of a list L is represented by a mapping from IDs of children of the list, to fractional labels in the CG. The idea is to use a unique label for all current (in T_{base}) and potential future list elements (from changes in the CG). Each list element mapping is from a node ID to a node label, and also includes information from which tree version the mapping was inferred. All elements from the list L_{base} (L as it appears in T_{base}) are mapped once to their corresponding labels from L_{base}. Then, independently for each revision, each element E that is newly inserted, relabeled, or moved into the list, is mapped to a fractional label consisting of a base index and an offset (e.g., 8.3). The base index indicates after which element from L_{base} an element E should be located, and the offset determines the order of elements with the same base index. Intuitively, this makes the labels of new elements relative to labels of elements in L_{base}. Considering only a single revision, it is guaranteed that no labels will clash. Considering both revision, some fractional labels might clash (e.g., if both revision insert a new element at the same position). The IDs of such elements are included in a domain conflict, and are remapped to new labels by adding ".a" and ".b" suffixes corresponding to each element's revision, in order to allow them to be merged into the list. Note that an element may be associated with up to three different positions (e.g., an element from L_{base}, that both revisions relabel to different locations). In such cases all of the possible labels are "reserved" for that element and subsequent applications of changes from the CG will determine the final label of the element. If both revisions map the same element to the same or adjacent locations, the mappings are merged.

The List-merge customization brings essential domain knowledge about lists to the merge algorithm. The customization not only resolves many conflicts automatically, but also reports merge ambiguities on a semantic level. Thus, it lets developers deal with less conflicts and do so more easily, saving time.

7.4.2 Conflict unit customization

Our merge algorithm is able to merge changes at the very fine-grained level of tree nodes, which is not desirable in some domains. For example, if $x < y$ in an AST is changed to $x \leqslant y$ in one revision, and to $x < y + 1$ in another, these two changes can be merged as $x \leqslant y + 1$, which is not intended. A common case where fine-grained merges might result in semantic issues is when changes affect nodes that are "very close" according to the semantics of the tree domain. We designed the Conflict unit (CU) customization to detect such situations. In essence, the customization partitions the tree into small regions called CUs and creates review items for each CU that is changed by both revisions. The customization does not alter the change graph.

The CU customization is parametrized by a set of node types – the *CU types*. The *conflict root* of a node N is its closest reflexive ancestor that is of a CU type. The tree root is always a conflict root. The set of nodes that have the same conflict

```
1: function UniqueListLabelsMap( L_base, L_A, L_B )
2:     labelMap ← IdsToLabelsFrom( L_base, Base )
3:     baseIndex ← 0
4:     topLevelChunks ← LCS( L_base, L_A, L_B )
5:     for all top ∈ topLevelChunks do
6:         if top is stable then
7:             baseIndex ← top.Elements(Base).last.label
8:         else
9:             offset = 1
10:            for all rev ∈ {RevA, RevB} do
11:                subChunks ← LCS(top.Elements(Base), top.Elements(rev))
12:                for all sub ∈ subChunks do
13:                    if sub is stable then
14:                        baseIndex ← sub.Elements(Base).last.label
15:                        offset ← 1
16:                    else
17:                        for all  e ∈ sub.Elements(rev) do
18:                            labelMap.add( e.id → baseIndex + "." + offset, rev )
19:                            offset ← offset + 1
20:                        end
21:                    end
22:                end
23:            end
24:        end
25:    end
26:    for all m_A,m_B ∈ labelMap | m_A.id=m_B.id ∧ Adjacent(m_A.label,m_B.label)
   do
27:        labelMap.Replace(m_A, m_A.id → m_A.label, RevA + RevB)
28:        labelMap.Remove(m_B)
29:    end
30:    for all m_A, m_B ∈ labelMap | m_A.label = m_B.label do
31:        AddDomainConflict("Clashing list elements", { m_A.id, m_B.id } )
32:        labelMap.Replace(m_A, m_A.id → m_A.label + ".a", m_A.rev)
33:        labelMap.Replace(m_B, m_B.id → m_B.label + ".b", m_B.rev)
34:    end
35:    return labelMap
36: end
```

Algorithm 7.4.1: The *UniqueListLabelsMap* function that returns a unique mapping between list element IDs and labels. LCS computes the longest common subsequence (*lcs*) of two or three lists, and returns a series of alternating stable and unstable chunks. Stable chunks contain the elements from the *lcs*, whereas unstable chunks contain the uncommon elements in-between.

root constitute a CU (see Figure 7.8). If two revisions make changes to nodes from the same CU, there is a potential for a semantic issue and this is reported with a review item.

With an appropriate choice of CU types, the CU customization can be useful

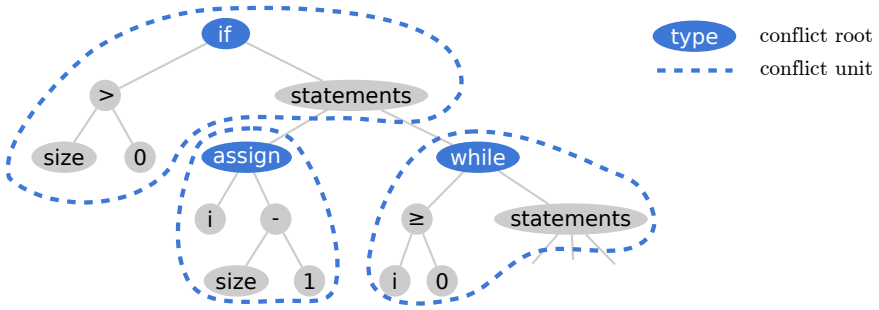


Figure 7.8: A tree with three conflict units.

in identifying potential semantic issues. For example, in ASTs, if statements are CU types, like in Figure 7.8, a change in one statement is semantically independent from a change in another statement, but two changes in the subtree of the same statement will result in a review item. In this setting, if a developer changes one part of the $i \geqslant 0$ expression, while another developer changes another part, these changes will no longer be silently merged, but a semantic issue will be reported.

Structure- and semantics-based review items are more precise and meaningful than the line-based conflicts produced by standard algorithms. A line-based conflict might incorrectly arise due to compatible changes (e.g., moving a declaration in one revision and adding a comment in another revision) or it might be due to formatting (e.g., changing a method name in one revision and moving the opening brace to a new line in another revision). In contrast, our CU approach is precise, predictable, and uses domain knowledge to report issues on a semantic level.

7.5 Evaluation and discussion

Even though Envision supports unique node IDs, we use Gumtree [FMB+14] to evaluate our approach on large existing Java projects to show its applicability on large trees with a long history. We inspected the default branch of the most popular (having more than 10000 stars) Java projects on GitHub, 19 in total. Six of the projects did not contain any merges of Java files. In the remaining 13 projects, we evaluate each merge of a Java file by comparing the merge results of Git and our implementation. We focus on the merge here since the merge operation depends on the diff and thus, reflects its quality. The results are presented in Table 7.1. All tests were run on an Intel i7-2600K CPU running at 3.4 GHz, 32GB RAM, and an SSD.

A *divergent merge* (DM) is one that results in conflicts (C) or one where the automatically merged file is different from the file committed by the developer. One exception are successful automatic merges in Envision that only differ from the developer committed version by the order of methods or import declarations. Since this order is semantically irrelevant, we do not consider such merges divergent – they are counted as *order difference* (OD). For Envision, we also list the number of files whose merge produced review items due to linearized list elements (RI_l) or changes to the same conflict unit by two revisions (RI_{cu}). For conflict unit types we use all statement and declaration node types. The total and average merge times are

Table 7.1: Comparison between merges by Git (G) and Envision (E). DM – divergent merge; C – merge with conflicts; OD – merge where only order differs; RI – review item (I – due to linearized list elements, cu – due to multiple changes in a CU).

project	number of files									merge	ovrhd.	avg.merge	
	all	DM		C		OD	RI_l	RI_{cu}		[s]	[min]	[ms]	
	merges	G	E	G	E	E	E	E		G	E	E	G	E
ReactiveX/RxJava	354	82	46	74	12	19	22	38		3	125	122	9	353
elastic/ elasticsearch	2677	863	547	821	281	29	157	525		10	276	266	4	103
square/retrofit	49	14	13	12	10	0	3	12		0	3	4	3	53
square/okhttp	163	4	4	4	1	0	1	11		1	23	22	4	138
nostra13/Android-Universal-Image-Loader	56	15	10	14	6	0	4	8		0	2	4	5	36
iluwatar/java-design-patterns	59	2	1	2	0	0	1	1		0	0	3	5	3
JakeWharton/ butterknife	18	0	0	0	0	0	0	0		0	1	2	6	73
greenrobot/ EventBus	10	4	3	4	2	0	0	4		0	1	1	7	54
square/picasso	40	0	0	0	0	0	0	3		0	3	4	5	75
PhilJay/ MPAndroidChart	169	32	35	24	20	0	9	21		1	13	15	4	76
square/leakcanary	13	1	0	1	0	0	0	2		0	0	1	7	31
bumptech/glide	76	69	54	69	29	2	22	32		0	3	6	3	39
spring-projects/spring-framework	339	14	4	14	1	0	3	5		1	2	19	3	5
total	4023	1100	717	1039	362	50	222	662		16	452	469	5	80

reported for both tools (merge and avg. merge). Merging Java sources with Envision incurs a significant overhead (ovrhd.) in addition to the merge time, because (i) the sources have to be parsed, (ii) the different revisions have to be matched to the base using Gumtree, and (iii) these two-way matchings are tweaked to enable a three-way merge. Almost all of this overhead can be avoided by using IDs directly stored on disk.

Tree-based merging results in significantly fewer divergent merges, 717, compared to the standard line-based approach, 1100. The difference in conflicts is even more substantial, with 362 for the tree-based approach, and 1039 for Git. Our approach also reports a significant number of files with review items for lists, 222, and conflict units, 662. Unlike textual conflicts, review items describe the semantic issue they reflect and report the minimal set of tree nodes that are affected, which makes it easier for developers to understand review items, and act accordingly.

To get more insight, we manually investigated all 46 cases of Envision's diverging merges in the RxJava project. All of these merges also diverge when using Git. There are 34 merges with real conflicts or merges where the developer made a semantic change, neither of which can be automatically handled. In the remaining 12 cases, we observed two reasons for divergence in Envision.

First, in six cases the result of a tree merge was, in fact, correct, but the version committed by the developer was incorrect. This occurred when a manual merge via Git results in a conflict which the developer resolves incorrectly, even though the resulting code compiles. For example, a conflict marker (`<<<<<<< HEAD`) inserted

by Git was forgotten in the middle of a block comment. Another example is the accidental omission of an `@Test` annotation which appeared just before a conflict marker. This omission potentially disabled one of the test cases in the code and went unnoticed for nearly three years until the RxJava developers accepted our patch for fixing it. Using our approach, all of these cases are automatically merged correctly.

Second, six merges diverge due to the suboptimal matchings produced by GumTree. For example, if a particular Java `import` declaration is present in the base version, but is deleted in both revisions, GumTree may match the deleted `import` to two different newly inserted `import`s from the different revisions. Our merge algorithm detects this as a conflict and fails to merge the file.

In terms of run-time, merging files with Envision is, on average, 16 times slower compared to Git. Nevertheless, Envision still allows merging at a rate of 12.5 files a second, which is significantly faster than manually resolving conflicts.

However, if the files are not stored using the format we described in Section 7.2.1, and require parsing and tree-matching, there is significant overhead, which further slows down Envision by a factor of 60. In this case merging a single file could take about one minute. To further investigate the effect of the matching on the merge result, we implemented a simple tree-matching algorithm and used it instead of Gumtree on the RxJava project. Our tree-matching produces worse matchings compared to GumTree, but incurs less overhead (81 minutes instead of 122). The simpler matcher resulted in more divergent merges (70) and more conflicts (40), compared to using GumTree, but the results are still better than using Git. These results suggest that our approach is most useful for storage formats that support dense data and include unique node IDs such that matching algorithms are avoided altogether.

Threats to validity

We evaluated our implementation on 13 Java repositories. Our results might not apply to other projects, other languages, or trees that are not ASTs. Nevertheless, the code bases we used provide a wide variety of tree-merge situations, and we used popular projects in order to increase the ecological validity of the results.

The tool we used to convert Java files into files encoded as we described in Section 7.2.1 omits some rarely-used Java constructs such as multiple type bounds for generic types. It is possible that a conflict in Git is due to a part of the code, which is missing in the new encoding. We are not aware of such cases.

We discard the text formatting and some comments. To handle such unstructured data with our approach the data would have to be encoded as part of the AST, e.g., by attaching a textual prefix node to each AST node.

7.6 Related work

Researchers have proposed a number of systems for version control of structured data. Molhado [NMB05] is a powerful stand-alone framework for versioning object-oriented data. It is based on an extensible model that could be used to version arbitrary types of objects. Molhado requires deep integration with the development environment, making Molhado the “heart of the environment”, in contrast to our more lightweight approach. OperV [NNPN10] is another approach for versioning of structured tree

data with fine granularity, which, unlike our system, is operation-based, thereby requiring additional data and more complex tool support. Unlike our approach, both Molhado and OperV introduce a custom storage backend and do not integrate with an existing VCS.

Altmanninger has surveyed various systems for versioning models [ASW09]. One of the most popular model repositories is EMFStore [KH10], part of the Eclipse Modeling Framework. There is continued interest in the research community in improving EMFStore, e.g., by formalizing merging for models [Wes10] or performing semantics-based mering [ASK10]. Odyssey [MCPW08, OMW05] is another model VCS, which targets UML models and features advanced merge capabilities. EMFStore, Odyssey, and most systems for versioning models are not often used to version trees, and unlike our approach, they use a custom backend and do not integrate with standard line-based VCSs. Our approach may be applied to graph models, e.g., by expressing them as containment trees, similar to Mikhaiel et al. [MTN+13].

Mens [Men02] provides an overview of different approaches for merging program sources. Newer approaches based on the full [ALL12] or partial structure [ALB+11] of source files have been proposed by Apel et al. These approaches improve on the merge results of Git, and can be fast and practical, but unlike our approach they do not work with unique IDs stored as part of the files, and thus may be inaccurate. Other approaches, rely on storing unique IDs, for example, the version control system of TouchDevelop [PBMM15] or MolhadoRef [DMJN07]. However, TouchDevelop is designed for a specific language and automatically resolves conflicts by ignoring one of the revisions, and MolhadoRef is an operation-based system, in contrast to our approach. Neither of the two integrate with a standard VCS like our approach.

There are also approaches to enhance VCSs for software with additional knowledge about the semantics of code and refactoring in order to improve merging [DMJN07, EA04, NND+15]. Our customization mechanism can also be used to provide similar semantics-based improvements to the merge.

Ghezzi et al. [GWGG12] propose that a pluggable framework be built on top of traditional VCSs in order to provide additional services and analysis capabilities. Our algorithms can be seen as an instance of their suggestion.

Lorenz and Rosenan [LR13] propose a JSON format for storing structured data and integrating it with a traditional VCS. Their proposal however uses the VCS only for storage and performs versioning on its own – one version of the JSON file in the VCS stores itself all previous versions of the objects that comprise it. In contrast, our approach uses the underlying VCS for both storage and versioning.

Lindholm [Lin04] proposes a way to merge XML documents using the XML tree structure. Their approach focuses on the particular class of document-oriented XML files, whereas our approach is designed for arbitrary trees.

MPS [VSBK14] is a commercial system which stores programs as XML files and implements custom merge hooks to integrate with traditional VCSs. It relies on IDs for precise merging, but the system does not seem to be customizable or easily usable for other data.

Schwägerl et al. have designed a graph-based algorithm [SUW15] for merging ordered collections. Unlike our List-merge customization, their algorithm only works with inserted, deleted, and relabeled elements, and there is no treatment for elements which are moved in or out of the list to another subtree and possible conflicts with these operations.

The IDE as a scriptable information system

8

In this chapter we bring together aspects of the flexible interfaces which we explored in Chapters 5 and 6 and smarter tools working on rich data as we have seen in Chapter 7, in order to help developers work with diverse information such as the source code, version repositories, issue trackers, or web resources. Existing tools for working with these different kinds of information lack good support for three key activities:

- combining information from different sources
- flexibly presenting collected information to enable easier comprehension
- automatically acting on collected information, for example, to perform a refactoring

Poor support for these activities makes many common development tasks time-consuming and error-prone. We propose an approach that directly addresses these three issues by integrating a flexible query mechanism into the development environment, turning the IDE into an information system, in which tools share data to help the user. Our approach enables diverse ways to process and visualize information and can be extended via scripts. We demonstrate how an implementation of the approach can be used to rapidly write queries that meet a wide range of information needs.

8.1 Problems developers encounter when working with information

Software development is an information-intense activity. While programming and designing software, developers ask a wide variety of questions [FM10, KDV07, LM10, SMDV08] and seek information from numerous sources such as the source code itself, compiler output, debug and program analysis tools, version control information, issue tracker, project and API documentation, colleagues, project wiki pages, and community resources like wikipedia.org and stackoverflow.com. In trying to meet their information needs, developers are faced with three issues.

First, developers often need to combine information from more than one source, but tool support for piecing information together is lacking [SMDV08]. For example, in order to understand a performance regression, it is useful to combine information from the source code (code structure and control flow), the version control system

(recent commits and changes to affected code), performance analysis tools (run-time measurements), and an issue tracker (bugs associated with relevant commits). In situations that require diverse information, developers are forced to manually connect the different pieces of information, which is an error-prone and time-consuming process. Such an information search is also tedious to refine as this usually requires the developer to manually repeat a part of the process.

Second, tools most often present information in a fixed form. Typical presentations include a list of items, a tree-view, or a visual graph. For example, searching with regular expressions results in a list of matches; querying a program with Ferret [dAM08] results in a hierarchical tree-view. These one-size-fits-all presentations are not always a good match for a developer's specific information need (e.g., a visual call graph is better suited for detecting recursion than a hierarchical list), but there is very little or no flexibility for customizing the presentation in existing tools. This could hinder the comprehension of the results and makes domain- and project-specific visualizations impossible.

Third, even after a developer finds the information they need, they often have to take action manually. For example, to understand how a set of methods are called, one has to manually set breakpoints or insert print statements in the code. Another example is a developer manually creating bug reports as a result of an analysis that detects certain code patterns. Repeatedly performing an action manually is time-consuming, error-prone, and frustrating. While some tools support automation (e.g., JunGL [VEdM06] and Rascal [HKV12] for refactoring), they cannot integrate arbitrary information sources and are usually limited to certain modifications of source code.

To address these three issues, we designed a query system that integrates directly with the IDE. We integrated the system in Envision, but the approach is also applicable to other IDEs. The system supports the integration of diverse information resources, flexible result presentations, extensibility via scripts, and automated execution of actions. Our evaluation shows that the system is applicable in a wide range of use cases with diverse information needs. A video demonstrating the system can be seen at youtu.be/kYaRKuUy9rA.

Next, we motivate our approach with two practical examples.

8.2 Motivating examples

8.2.1 Investigating a regression

Suppose that a developer is investigating a recently reported regression, where the incorrect behavior occurs after a specific button is pressed. To investigate this problem the developer will need two main pieces of information:

- The source code, more specifically, the code that is executed after the handler of the button click.
- The version repository, to see what recent changes could cause this issue.

With current tools, the developer will likely first explore what code is being called from the button handler and manually correlate that to recent changes. This could be a rather time-consuming task if a lot of code is potentially reachable from

the handler or if there are many changes that have happened in the mean time. In particularly hard cases, it might pay off to design a specific test case for this regression and run a binary search on the version repository in order to find the offending commit (e.g., using `git bisect`). Both of these approaches are rather time-consuming due to ineffective ways of combining source code information (the call graph) with version information (what changed recently).

Our approach offers an alternative solution. The developer can select the handler of the button in the source code, bring up a query prompt and type:

```
callgraph -nodes | changes -c 5 -nodes
```

The `callgraph` query returns the nodes (methods) in the callee graph of the currently selected method. The bug is likely among these methods, but there may be many of them. To narrow down the search, the methods from the callee graph are piped into the `changes` query, which returns only those methods from the graph that have changed in the last five commits. After the query is executed, the relevant source code fragments will be highlighted and help the developer to more quickly find the issue.

Enabling this workflow are three key components of our approach:

- A **context-sensitive query prompt** that enables developers to quickly type and combine queries.
- Diverse **queries** that can access arbitrary data resources such as the program's source code or version repository.
- A **unified data format** that enables queries to be combined in order to refine searches.

8.2.2 Heatmap of code execution

Imagine that a developer wants to get a visual overview of the often-executed parts of the code. The goal is not to optimize specific code, but rather gain a general understanding of which classes are relevant for performance and what is their general function. Thus, it is preferable to see frequently executed methods in a broader context.

Profiling tools typically provide timing information in the form of a chart, graph, or a list. However none of these presentations fits the developer's need in this example, as the code around performance critical methods is also important. The developer will have to manually switch between the profiler and the code they want to explore.

This is one example where the presentation of information is critical for understanding and where our approach's support for flexible visualizations can help. The developer could, for example, export the timings to a CSV file and use a query to import it into the IDE and visualize the results:

```
importProfileCSV profile.csv | heatmap
```

The first time they do this, they will have to write the `importProfileCSV` Python script shown in Figure 8.1. The script reads the CSV file into the data format understood by our system and is straight-forward to write, thanks to the freely available `csv` Python library. The read data is piped into the `heatmap` query, which highlights

```
import csv

# Returns the AST node corresponding to the provided method name
def findMethod(methodName):
  name = methodName.split('(')[0]
  # Execute another Envision query that returns matching methods
  astTuples = Query.methods(['-g', '-n={}'.format(name)],[])[0]
  for astTuple in astTuples.tuples('ast'):
    return astTuple.ast

# Read the CSV file and associate methods with a count
with open(Query.args[0]) as csvfile:
  CSVReader = csv.reader(csvfile, delimiter=';', quotechar='"')
  next(CSVReader, None)
  for row in CSVReader:
    m = findMethod(row[0])
    count = int(float(row[1]))
    values = [('count', count), ('ast', m)]
    Query.result.add(Tuple(values))
```

Figure 8.1: A Python script that imports profile files generated by Visual VM.

different parts of the code with a color in the red-green spectrum based on the value of a number. An example heatmap is shown in Figure 8.2.

This example illustrates two more essential components of our approach:

- Tight integration with a mainstream **scripting language**, allowing easy exten-

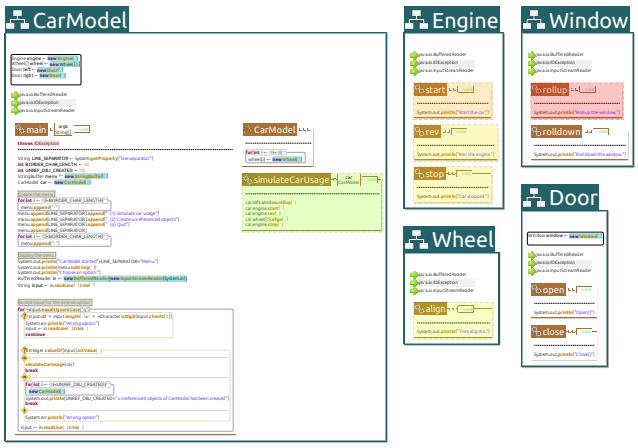


Figure 8.2: A heatmap overlayed on top of Envision's usual code presentation. The heatmap is visualized as a set of translucent overlays on top of methods; each overlay has a color in the spectrum between red and green, indicating how often a method was executed.

sion to new information resources and highly-customized queries.

- **Flexible visualizations**, which enable task-specific rendering of information to facilitate comprehension.

Next, we describe our approach and its components in more detail.

8.3 Approach

Our goal was to design a system that is highly expressive and extensible by the user in order to satisfy a wide range of information needs. The architecture of our system is shown in Figure 8.3. Using a query prompt, a developer can invoke and combine queries. Queries can access diverse information resources, make computations, and produce visualizations, and can be either native or implemented via scripts. A unified data exchange format facilitates the cooperation between multiple queries. An execution engine orchestrates query execution and the information flow between queries. Below, we discuss each component in detail.

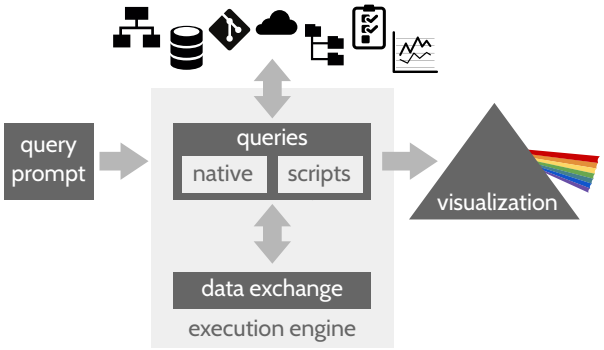


Figure 8.3: The architecture of our information system.[1]

8.3.1 Query execution model

The core of our approach is the ability to compose and execute queries. This functionality is provided by the execution engine, which implements a simple computation model. It enables queries to be connected in a directed acyclic graph, where the edges between queries represent data flow. Such a network of queries is illustrated in Figure 8.4.

The execution engine provides each query with a context, which is the AST node (e.g., a method) on which the command prompt was invoked. A query has two additional ways to access information.

First, a query may be connected to the output of other queries via any number of required or optional inputs, which comprise the inter-query data exchange. For example, a query might receive on its input a set of method AST nodes, and output

[1] The git logo by Jason Long and icons made by Freepik from www.flaticon.com are licensed by CC BY 3.0.

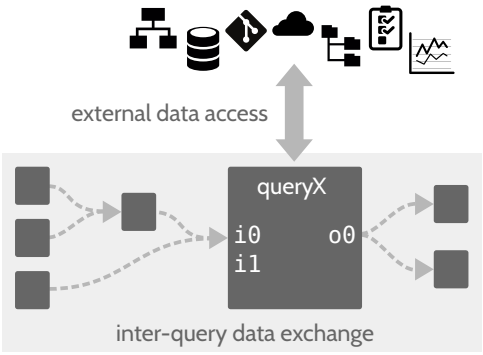


Figure 8.4: A query network with seven interconnected queries. *queryX* is shown in detail. It has two inputs (`i0`, which is the union of two outputs from other queries and `i1`, which is unused here), and one output (`o0`, which is duplicated).

their names as a set of strings. To enable communication between queries, it is key that the data format flowing between queries is unified (see Section 8.3.3).

Second, a query may access (that is, read and modify) external resources (see Section 8.3.2), such as the program's source code (e.g., to perform a refactoring), call a method of the IDE (e.g., to access the AST or show a message), read from a file (e.g., to import external data), or use a REST service (e.g., to create a bug report).

A query is executed only after all of its inputs are read. When a query is run, it can perform arbitrary computation, which typically includes accessing external resources and computing outputs. Once a query has finished executing, its outputs are forwarded to any downstream queries.

8.3.2 Query types

In principle a query can perform arbitrary computation and use many information resources. However, to facilitate composition, we divide queries into three types (Figure 8.5): **resource-access**, **visualization**, and **operator**. Below, we define each type and explain how we designed corresponding queries in order to improve usability.

Resource access

Resource-access queries are used to read and modify resources external to the execution engine. They connect a network of queries to the AST and external tools and data. A resource-access query can be a source of data for other queries. For example, a query may read the contents of a file or it may extract version information from the project's repository and provide them as inputs to other queries for processing. A resource-access query may also modify external data. For example, it could create a new record in a database, or modify the program. There is no limit on the type of external resources a query may access; some common ones are the program code or AST, the version repository, the issue tracker, the file system, and on-line services.

To facilitate composability, we designed resource-access queries according to the following guidelines. First, a resource-access query provides access to only

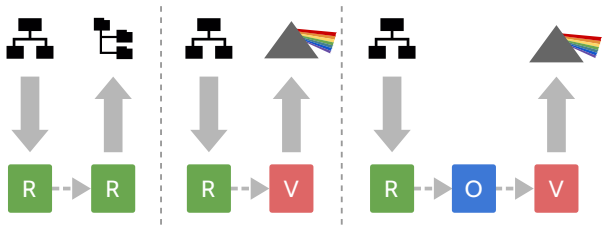


Figure 8.5: The three common types of queries: (R) resource-access, (V) visualization, and (O) operator. On the left a resource-access query extracts information from an AST and forwards it to another query, which writes it to a file. In the middle, this information is being visualized instead of written to file. On the right, an additional filter is inserted to refine what information is displayed.

one resource. This restriction allows accessing one resource without imposing requirements on another one. Second, complex resources are accessed by multiple queries, which enables each query to focus on a particular aspect of the resource. For example, when reading the program's AST, one query is used to select nodes while another provides control-flow information. Third, when integrating tools that have a command line interface, we created queries with a similar interface. This enables developers to transfer some of their existing knowledge from the terminal command to the query. For example, in a query that accesses a Git repository, commits can be specified by commit id, branch name, or reference like the `git` command allows.

A noteworthy class of resources available through resource-access queries is external programming tools and the IDE itself. A query may call available IDE APIs to get information or to perform IDE functions. For example, most IDEs maintain a code model that provides easy access to the program's AST. Such queries are not limited to extracting information. Depending on available IDE APIs, queries might be used for navigation between code fragments, setting breakpoints, running tests, displaying warnings or errors, and refactoring code, all of which and more are supported by Envision. Combining queries effectively provides a user-controlled way for sharing data between different development tools.

Visualization

Visualization queries are used to render information on the screen. Different visualization queries can be used to render the same piece of information in different ways in order to better match the specific information needs of the developer. For example, Envision supports three ways to visualize relations between code elements:

- Show the relations using a textual notation – useful for a dense summary.
- Highlight on screen all code elements that appear in the relations – useful when searching for particular patterns.
- Show the relations using arrows between code elements – useful when exploring call graphs or data flow.

Visualizations can be provided by the underlying IDE or via external tools. Common visualizations in mainstream IDEs are highlighting a program fragment, showing a list or a tree of result entries, and displaying error messages and warnings. More visual IDEs, such as Envision, could provide a map of the code that enables intuitive arrow overlays to explore connections between elements or even a heatmap visualization similar to Figure 8.2.

Our approach imposes no limit on how information can be visualized. If an IDE exposes general drawing routines, a query could use those to implement an entirely custom visualization. Some IDEs, such as Envision or Eclipse, provide a built-in HTML rendering engine, which could be used to easily and quickly implement new ways for visualizing information. Using a combination of HTML5 and Javascript, it is even possible to create interactive visualizations as we demonstrate in Section 8.4.1. This high degree of flexibility is indispensable for domain- and project-specific visualizations.

We designed visualization queries according to the following guidelines in order to make them easier to use. First, each available visualization mechanism has its own query, which makes it clear what will appear on the screen when it is invoked. Second, visualizations impose as few requirements on the input data as possible, so that one visualization can be easily swapped for another. Third, if a visualization query is not explicitly provided by the user, but there is unconsumed data at the end of a query-network execution, a visualization is automatically chosen based on the structure of the result. This frees developers from the need to always explicitly specify a visualization that could be automatically inferred.

Operator

Operator queries (operators) are used to perform internal computation, for example, to refine results. Operators do not access any external resources, but rather help filter and combine data in complex query networks. They work solely with the unified data format, which we discuss in Section 8.3.3. As operators work with sets and relations, they naturally map to operations from set and relational algebra such as union, intersect, select, and join. Building on these primitives, we have also pre-defined more elaborate operators in order to simplify common cases. For example, in Section 8.4.2, we demonstrate the `reachable` operator, which filters out elements unreachable from a starting point via a relation – in essence a combination of transitive closure and selection. Such a convenience operator is very useful in answering reachability questions, which are very common [LM10].

8.3.3 Inter-query data exchange

To enable query composition, all queries communicate using a simple unified structure for exchanging data. This unified exchange structure is a **set of tuples**, because it is sufficiently expressive and provides a simple mental model for developers to work with.

Each input and output of a query is a set of tuples, and each tuple consists of an arbitrary number of named elements. The names of elements within a single tuple have to be unique. The elements of a tuple may be strings, integers, and references to AST nodes. Each tuple has a tag, which is an identifier that is either explicitly provided or is identical to the name of the tuple's first element. This

minimal structure allows us to conveniently encode and access typical structures such as sets, lists, and graphs. For example, the set of all methods whose name starts with `get` could be:

```
{  (node: getAge), (node: getAddress)  }
```

The tuple tags could also be provided explicitly:

```
{  node: (node: getAge), node: (node: getAddress) }
```

It is also easy to express relations. For example, all methods transitively called from `rest` could be expressed using the `calls` relation:

```
{  calls: (caller: rest,  callee: watchTV),
   calls: (caller: rest,  callee: sleep),
   calls: (caller: sleep, callee: dream)    }
```

Since there is no restriction on what the set contains, we could also merge the two sets above:

```
{  (node: getAge), (node: getAddress),
   calls: (caller: rest,  callee: watchTV),
   calls: (caller: rest,  callee: sleep),
   calls: (caller: sleep, callee: dream)    }
```

It is also easy to relate information from different data sources. For example, in:

```
{  (commit: "bcdef01", author: "John" ),
   changes: (commit: "bcdef01", node: sleep),
   calls: (caller: rest,  callee: watchTV),
   calls: (caller: rest,  callee: sleep),
   calls: (caller: sleep, callee: dream)    }
```

We can see that John made a change to `sleep`, which is called by `rest`.

Filtering and combining can be easily done based on the name or value of elements, or the tags of tuples. Result visualizations can be automatically selected based on the tags and tuples present in the final output. For example, a tuple with the tag `message` could be shown as a message associated with a code location.

8.3.4 Query prompt

In order to make information access quick and convenient, we designed a specialized input mechanism for invoking queries – the query prompt. The query-prompt resembles the standard command prompt of Envision, but provides functionality specific for queries. Below, we list the key features of this interface.

The prompt is normally hidden and does not take space on screen. Using a keyboard shortcut, the developer can show the prompt on top of an arbitrary code fragment. Like the command prompt, the query prompt is context-sensitive — it records the location of the cursor inside the source code at the time the prompt was shown and forwards it to queries. The queries can then use this context (e.g., a class or a method) to implement their behavior.

To use the prompt, developers simply type the queries they want to execute. This keyboard-based input allows experienced developers to efficiently invoke queries.

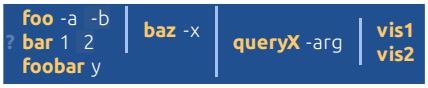


Figure 8.6: A complex composition of queries in the query prompt. The outputs of `foo` and `bar` are merged and forwarded to `baz`, whose output is merged with the output of `foobar` and forwarded to `queryX`. The output of `queryX` is duplicated and forwarded to both `vis1` and `vis2`. The resulting network is the one from Figure 8.4.

Multiple queries can be composed by typing the pipe character '|'. The similarity of this interface to a typical Unix command prompt ensures that developers are already familiar with the interaction flow and composition using pipes.

The query prompt is also used for displaying errors. First, on a more basic level, if the user tries to execute a nonexistent query or a query with incorrect arguments, the prompt will display an error message shown in red. Second, on a more logical level, each query from a network of queries can halt execution and return an error message if it encounters a run-time error or an unexpected condition. If no errors are encountered all queries are executed silently, producing only the side effects that the user requested. If the user incorrectly composed queries, which nevertheless resulted in a full execution, the users will need to manually detect and handle the logical error in their query composition.

The input field of the prompt is not a standard text box, but a custom widget that allows non-linear queries. Figure 8.6 shows an example of non-linear input. The prompt does not allow the creation of an arbitrary query network, and uses at most one input and one output per query. Developers can create parallel paths by pressing a keyboard shortcut and can direct data flow using multi-line pipes. There are currently two multi-line pipes: joining and subtracting. Both types have a single output that is duplicated among all downstream queries. A joining multi-line pipe outputs a union of all of its inputs. A subtracting multi-line pipe subtracts from its first input all remaining inputs and outputs the result. Even though this mechanism cannot construct an arbitrary query network, we find that it is sufficiently expressive for many practical tasks.

While we find the query prompt a convenient mechanism, the rest of our system is independent of it and there are other ways for invoking queries. For example, a lighter approach could be to show a text-input on right click and parse linear commands. This could also be extended with support for named pipes. On the other hand, for creating truly custom queries, a graphical editor could be created, which allows manually wiring queries in a complex network.

8.3.5 Extensibility via scripts and native queries

Using our system, the simplest way to access and manipulate information is to compose existing queries directly on the query prompt and get results immediately. Query composition covers a broad range of common needs with minimal effort, but inevitably, developers will need specialized behavior where composition will not be enough. To support custom information processing we provide two extensibility mechanisms.

```
for node in Query.input.tuples('ast'):
  if isinstance(node.ast, IfStatement):
    if node.ast.elseBranch.size() > 0:
      Query.result.add( node )
```

Figure 8.7: A Python script that filters input nodes.

The first one is to implement new queries as light-weight scripts, in our case using Python. Scripts allow orchestrating complex query flows and provide access to specialized resources, which is made easier by existing libraries for the scripting language. Scripts have access to a limited API: the context, inputs, and outputs of the query they represent, and the program's AST. This API is enough to make scripts versatile while keeping them extremely simple. For example, Figure 8.7 shows the complete script for selecting only if statements that have else branches from the input. Implementing a query via a script is as easy as writing the script file. As Envision does not support Python yet, scripts have to be implemented in another editor. Envision has no dedicated support for debugging Python scripts, but script writers can use standard visualizations to display intermediate results directly in Envision during a script's execution. Any available scripts are directly invokable on the query prompt and scripts are seamlessly composable with other queries. The execution engine translates the inter-query data format to and from the native environment and the scripting language's virtual environment.

The second extension mechanism is to create new native queries within the host IDE of our information system, in our case within Envision. Native queries give the developer unlimited power to perform specialized computation and allow deep IDE integration — unlike scripts, native queries have access to all IDE APIs. The drawback of this approach is that it is more demanding and time-consuming than writing a script, since developers will have to, essentially, extend the host IDE (e.g., by writing a plug-in).

In practice, native queries, which provide deep IDE integration, and scripts, which access a custom resource, complement each other well, as we show next.

8.4 Evaluation and case studies

Vogel [Vog15] evaluated an implementation of our query approach in Envision by using it to answer questions that developers frequently ask. He investigated 89 questions compiled from the works of Fritz and Murphy [FM10], Hajiyev et al. [HVM06], and Urma and Mycroft [UM15]:

- 64 questions pertained to information sources for which support is not yet implemented in Envision, and thus, could not be answered. All remaining questions could be answered.
- 20 questions could be answered using only built-in queries.
- 5 questions could be answered by writing additional scripts.

All queries and scripts can be found in [Vog15]. These results suggest that our approach is effective at covering the information needs of developers for implemented information sources.

Below, we show the applicability of our approach by demonstrating its use in a variety of practical programming scenarios. For each case, we motivate its practical relevance, show a possible solution, and discuss how our approach addresses the needs of programmers. All examples can be expressed in Envision.

8.4.1 Callgraph of selected method

Programmers frequently need to understand control flow and, in particular, follow paths through several method calls [LM10]. IDEs often have built-in call graph views, and there is a number of research tools that further facilitate the exploration of call graphs [KKD+11, KKKB12, LM11]. Let's examine a simple scenario:

What is the callgraph of this method?

As this is a commonly needed piece of information, we provide a native query:

```
callgraph
```

Because it is context-sensitive, this query will automatically return the callee graph of the method that contains the cursor. The query itself just returns a set of tuples and does not produce any visualization. As no visualization is explicitly provided, the execution engine automatically chooses one based on the structure of the result. By default, results that represent relations between AST nodes are visualized as arrows connecting the nodes' representations on screen as shown on Figure 8.8(a). Alternatively, the developer could show all result tuples in a table (Figure 8.8(b)):

```
callgraph | table
```

or highlight visually the methods that are part of the call graph without showing arrows (Figure 8.8(c)):

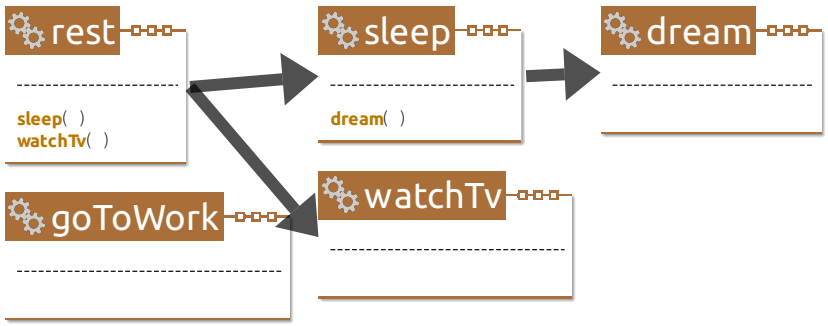
```
callgraph -nodes
```

Here, the execution engine will detect that the output is a set of AST nodes and will highlight them on the screen. These three visualization options illustrate an important aspect of our approach: the results of queries are decoupled from their visualizations. This decoupling enables flexible visualizations that allow the presentation of the results to more accurately match the information need of the developer. Common presentations such as highlights, graphs, arrows, and tables can be readily provided by the IDE. As we show next, developers can also add their own visualizations.

In some projects or domains, specialized visualizations enable better information comprehension. A very convenient way of achieving custom visualizations is to quickly create them by using HTML and existing developer skills. Most full-fledged IDEs and also Envision come with a built-in HTML rendering engine, and our approach allows visualization queries to utilize such capabilities. For example, we used the freely available open-source *vis.js* Javascript library and a custom Python script to implement the visualization from Figure 8.8(d). With a total of 80 lines of Javascript and Python code we can run the query:

```
callgraph | toHtmlGraph
```

(a)

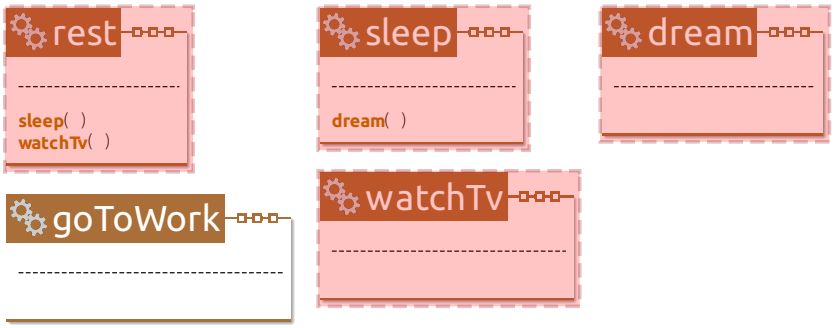


(b)

calls

callee	caller
Person.watchTv	Person.rest
Person.sleep	Person.rest
Person.dream	Person.sleep

(c)



(d)

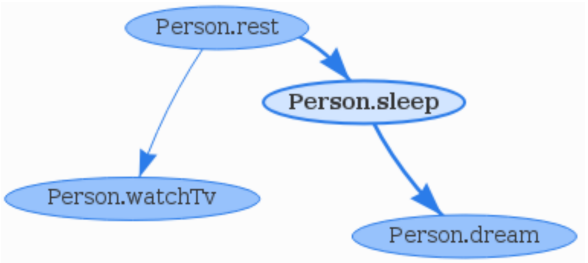


Figure 8.8: Different ways of visualizing results that describe a relation: (a) arrows between related elements; (b) a table; (c) highlighting relation elements; (d) a custom HTML visualization: in this case, an interactive graph rendered using the `vis.js` Javascript library.

and get an interactive HTML view of a graph that enables users to select and rearrange nodes. The `toHtmlGraph` script converts the tuple set from our system into an HTML page that uses *vis.js*, which in turn provides the rendering and interactions. See Appendix B for the complete script and the wrapper HTML page.

8.4.2 Recently changed recursive methods

Sillito et al. [SMDV08] identify the lack of support for writing refined queries and combining information as two of the three major gaps in tool support for answering developers' questions. One such question is:

Which recursive methods have changed recently?

This question is a refinement of the questions we showed in both Section 8.2.1 and Section 8.4.1. Compared to the former, it adds the requirement of *recursive* methods. Compared to the latter, it adds the need for another information source, the version repository. Our approach enables answering this question directly:

```
callgraph -global | reachable -self | changes -c 5 -nodes
```

The `-global` argument makes `callgraph` return the call graph of the entire program, ignoring context. The result is then piped into the `reachable` query, which filters AST nodes that cannot reach themselves following the relations from the input. The output contains only recursive methods. Finally, we use the `changes` query to select only those methods from the input that have changed in the last 5 commits.

Two things are noteworthy. First, it was possible to easily refine the call graph query by inserting the `reachable` filter in order to get a set of recursive methods. And second, it was also easy to combine the result with another information source. Other research tools rarely provide both of these features, and mainstream tools such as regular expression search are thoroughly inadequate for such tasks.

8.4.3 Why is this code the way it is?

Programmers often need to know the reason some code exists or looks the way it does (e.g., questions 8-10 from Fritz and Murphy [FM10]). There are multiple ways to interpret and answer this question, but a common approach is to connect a piece of code with version control and bug database information. Here is how a query answering this question could look:

```
ast -type Statement -topLevel
| changes -intermediate
| join change.id,commit.message,ast
  -as data
| associatedBugs
```

This is a more complex query, but building it piece by piece is rather straightforward. We want information about each statement in a method, so we first get all top-level (non-nested) statements using `ast -type Statement -topLevel`. The result is piped into the `changes` query, which yields change information about all commits that modify any of the input statements. The result consists of two different kinds of tuples: the first one associating each node with the id of the commit where it changed, the second associating a commit id with the commit's meta data (e.g., the commit message). Using the `join` query, we merge those two different kinds of tuples

```
import re
from github3 import GitHub

gh = GitHub()
repo = gh.repository('username', 'repository')

def referencedIssues(commit):
  for issueId in re.findall('#(\d+)', commit):
    yield repo.issue(issueId)

# Build an HTML message from commit and issue data
for data in Query.input.tuples('data'):
  text = '<b>Commit</b><br/>{}<br/>'.format(
        data.message.replace('\n', '<br/>'))
  for issue in referencedIssues(data.message):
    text += '<b>Issue #{}</b><br/>{}<br/>'.format(
            issue.number, issue.title)

  t = Tuple([ ('message', text),
      ('ast', data.ast), ('type', 'info') ])
  Query.result.add(t)
```

Figure 8.9: A Python script that fetches issue information from a GitHub repository.

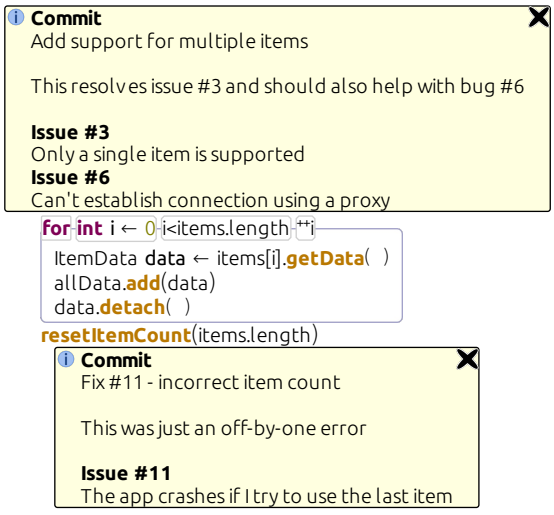


Figure 8.10: A for-loop and a method call with corresponding explanations for why they were last changed. The information bubbles are standard visualizations in Envision.

into a single kind that relates nodes and commit messages and that we call `data`. We use this name in the `associatedBugs` script (Figure 8.9), which is invoked next. This script scans the text of each commit message, looking for references to issue numbers and fetches the corresponding issues' descriptions from the GitHub issue tracker using a REST request. It uses the freely available `GitHub3` Python library to communicate with GitHub. The output of the script is a set of tuples representing `info` messages, which are automatically shown using standard Envision information bubbles next to the corresponding code as shown in Figure 8.10. Developers might want to execute this query often. They can conveniently create an alias to it and use this name to call it in the future. The alias may also appear as a subquery in other even more complex queries.

Functionality for explaining the reason for a piece of code is also available in other tools like the `blame` command for version control systems, or the Eclipse annotate feature. However, our approach allows building this information from elementary blocks and precise controlling of what is included. Developers can also link to arbitrary additional sources to fit the answer of this question to their needs.

8.4.4 Which upstream changes possibly conflict with mine?

This question might arise in fast moving projects where developers' local branches may quickly diverge from the development branch. Answering this question precisely is impossible in general, but one way to get an approximate answer is to compare local changes to changes from the remote branch. One possible query to perform this comparison and the corresponding result are illustrated in Figure 8.11.

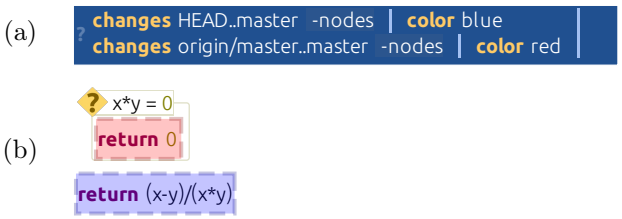


Figure 8.11: (a) Two parallel queries that will highlight local changes and changes between the `master` branch and the `origin` repository in different colors. (b) The resulting highlights on part of the code. The current implementation ignores changes for AST nodes that are not currently displayed, but this behavior can be adjusted in the future, e.g., to show such nodes in an additional view.

We run two queries in parallel that fetch the latest changes and highlight the changed nodes in different colors. The query prompt's support for non-linear queries enables the flow of query results to be split and joined to form a more complex query graph, and can often yield a simpler and more intuitive solution compared to a linear approach.

8.4.5 Instability metric

A common application of program query tools is to compute software metrics [AHR11, MSV+08]. To illustrate the computation of metrics using our approach,

```
# Returns the fully qualified package of a node
def packageOf(node):
  package = ''
  node = node.parent
  while node:
    if type(node) is Module:
      package = node.symbolName() + '.' + package
    node = node.parent
  return package

# Returns a list of all packages a class imports
def dependsOnPackages(aClass):
  result = []
  for decl in aClass.subDeclarations:
    if type(decl) is NameImport:
      package = ''
      name = decl.importedName
      while type(name) is ReferenceExpression:
        package = name.name + '.' + package
        name = name.prefix
      result.append(package)
  return result

allPackages = set()
eff = {}
aff = {}

# Loop over input classes to collect
# package dependencies
for tuple in Query.input.take('ast'):
  p = packageOf(tuple.ast)
  allPackages.add(p)
  deps = dependsOnPackages(tuple.ast)
  if deps:
    eff[p] = 1 + (eff[p] if p in eff else 0)
    for dep in deps:
      aff[dep] = 1 + (aff[p] if p in aff else 0)

# Compute the instability of each package
for p in allPackages:
  e = eff[p] if p in eff else 0
  a = aff[p] if p in aff else 0
  i = str( e/(e+a) ) if e+a > 0 else 1
  t = Tuple([('package', p), ('instability', i)])
  Query.result.add(t)
```

Figure 8.12: A Python script that computes instability of all packages.

we will compute an instability metric [Mar94]. The instability I of a Java package could be defined as:

$$I = \frac{\textit{Efferent Couplings}}{\textit{Efferent Couplings} + \textit{Afferent Couplings}}$$

Where *Efferent Couplings* is the number of classes inside the package that depend on (import) classes outside the package, and *Afferent Couplings* is the number of classes outside of the package that depend on classes within the package. The higher the instability, the easier it is to change a package without affecting other packages. To compute this metric we could use the following query:

```
ast -type Class -global | instability | table
```

This query fetches all `Class` AST nodes, forwards them to the `instability` script, and displays the results in a table. The `instability` script (Figure 8.12) iterates over all classes, collecting dependency information in order to compute the metric. Our approach's support for scripts enables the easy computation of metrics and more generally of code analyses.

8.4.6 Modifying recursive methods

Programmers often have to change existing code at scale. Development tools typically provide a limited set of refactoring options, and one is lucky if one's use case is covered by these. The fallback solution is most often textual search and replace using regular expressions. However, many situations are non-standard and are simply not expressible using regular expressions. Let's consider one such example – a variation of Section 8.4.2:

Modify the program so that each recently changed recursive method prints the values of its arguments when called.

This modification is for instance useful if a program crashes due to a stack overflow after recent changes and one wants to better understand what code is being executed. We could add the necessary print statements using the query:

```
callgraph -global
| reachable -self
| changes -c 5 -nodes
| insertArgPrinting
```

Like we showed in Section 8.4.2, the first three queries get all recursive methods that have changed recently. Then we pass these methods to the `insertArgPrinting` script (Figure 8.13), which inserts code that prints the name and lists all arguments of each method.

To help deal with accidental changes to the code during the development of refactoring scripts, code changes performed via scripts can be undone just like manual changes by pressing Ctrl + Z. A further help with developing refactoring scripts would be to also display a preview of the potential code changes, but Envision's current design does not support this.

Three things are worth noting here. First, we executed a query which modified a resource, in this case the source code. Second, this modification uses both non-trivial program properties (recursion) and information other than the source code itself (version information). Thus, it is outside the reach of regular expressions and also

```
for t in Query.input.tuples('ast'):
  m = t.ast
  if type(m) is Method:
    call = 'System.out.println("calling ' +
           m.name + ': "'
    for a in m.arguments:
      call += ' + "' + a.name + '=" + ' + a.name
    call += ')'

    m.beginModification('add print statement')
    nodeExpr=AstModification.buildExpression(call)
    printStmt = Node.createNewNode(
      'ExpressionStatement', None)
    printStmt.expression = nodeExpr
    m.items.prepend(printStmt)
    m.endModification()
```

Figure 8.13: A Python script that inserts code to print all arguments at the beginning of a method.

impossible in typical refactoring languages such as JunGL [VEdM06] or Rascal [HKV12], which cannot integrate additional information sources into refactoring decisions. Third, the support for scripts makes it easy to perform program edits within the overall query mechanism. Just like any other query, edits can also depend on input from other queries and external data resources, which enables data-driven changes to the code.

8.5 Implementation

We implemented our information system approach as a plug-in for Envision. Each native query is implemented as a single C++ class, which has access to our framework's infrastructure as well as all IDE APIs. Additional native queries can be added by creating new query classes in new plug-ins. We use existing Envision features to implement queries that show highlights, arrows, and information messages, and to render HTML. The advanced features of the `changes` query (e.g., version information on a per-AST node basis), are enabled by Envision's own fine-grained version control system, which we discussed in Chapter 7.

To enable advanced interactions and non-linear queries in the query prompt, we parse the user's input on every keystroke and create a parse tree of the current input. This interface uses the string offset provider, which we discussed in Section 6.2.4, and a simple regular-expression-based parser. The parse tree is used to create and render parallel queries, and it is mapped to a string representation that enables copying and pasting. This string representation allows users to share queries as text and is also used when creating aliases to queries.

Queries are executed sequentially by a simple data-flow programming engine, which can be easily extended to parallel execution and streaming of data.

We use *boost.python* to integrate Python scripting. Python scripts have access to the inputs, outputs, and context of a query, as well as Envision's AST model. A few helper features are also available, such as the ability to invoke other queries and to modify the program. Access to all these features is provided via the `Query` namespace which is automatically made available in each script.

8.6 Related work

8.6.1 Questions developers ask

A number of recent studies investigate what questions developers ask and how they seek answers to these questions.

Ko et al. [KDV07] observed 17 professional developers in 90-minute sessions and recorded the information they needed and how they acquired it. They present a list of 21 information types and associated questions that developers often asked. The authors suggest that tools should be able to share information and let users transform it as needed, both of which our system directly supports.

Sillito et al. [SMDV08] observed 27 developers during two studies and compiled a list of 44 questions that were frequently asked during program evolution. They list three areas for improvement of information seeking tools:

- support for asking more precise and refined questions
- using context when searching for information and displaying results
- support for combining information

Our approach directly addresses these three gaps with the ability to pipe queries, to use context for queries and show results in-line with the program, and to combine information from different sources.

In three separate studies LaToza and Myers [LM10] observed that developers often asked reachability questions and suggest that answering such questions is difficult and time-consuming in large code bases. Reachability questions are about a feasible path through a program, for example, in the program control flow or data flow graphs. Our approach directly supports reachability questions.

Fritz and Murphy [FM10] interviewed 11 professional developers about questions they face frequently and learned 78 questions that span different domains such as the source code, bug database, version history, test cases, etc. Most of these questions require linking different information together, which is supported by our approach.

Sadowski et al. [SSE15] collected data from in-browser code search queries of 27 developers at Google and characterized their search behavior. One of their observations is that developers frequently perform quick searches to navigate code and the authors suggest the integration of search tools directly within the IDE to facilitate quick searches without context switching. Our system enables this workflow.

8.6.2 Tools for seeking information

Researchers have developed a variety of languages and techniques for querying source code [DRNKJ11, JDV03, MSV+08, SEHM06], which have also been analyzed in

comparative studies [AHR11, dAMR07]. Some tools also offer a natural language interface [KMM11, WGRG10]. Such query tools offer powerful and efficient ways to query program properties, but many are restricted to the program source and do not integrate additional information resources. Such query engines could be integrated as a single information resource in our platform, which will enable the combination of their output with additional information. For such an integration, it is important that query results are reified with other entities (e.g., AST nodes) so that they are usable in the rest of the system [DRNKJ11].

A number of tools do allow the combination of different data sources. ABSINTHE [KDRN+11] was designed specifically to enable queries over different versions of software. More generally, as the basis of the Ferret tool, Alwis and Murphy [dAM08] present a model for integrating information from different sources, which they call Spheres. For example, one sphere could represent source code, while another could capture run-time information such as a call stack. Two spheres can be linked if they contain matching elements, and these matchings have to be predefined by the tool designers. Later, Fritz and Murphy [FM10] proposed another approach for integrating data from different sources – the *information fragment model*. Unlike Ferret, the information fragment model allows the automatic inference of links between different kinds of information so that it can be easily composed. Our approach is different from these approaches in several ways. First, the link between different sources does not need to be predefined, but the developer has full control over what information is linked. Second, the simple tuple set interface of queries allows for the easy addition of a wide range of information resources. Third, result visualizations are flexible to better match specific information needs.

Recognizing the importance of accessing and combining information from different sources, Myers [Mye98] suggests using open data models for IDEs. Similarly, Schiller and Lucia [SL12] formalize an open model for inter-plug-in communication and cooperation within an IDE. They suggest that plug-ins should share data and should allow users to put information from plug-ins together via pipes and filters similarly to the Unix Shell and the Windows Powershell. Our system is also inspired by the Unix shell, but additionally is concerned with visualizations and allows more flexible plug-in and script mixtures. In a similar spirit, Kuhn [Kuh12] suggests that IDEs should become open platforms that facilitate the data exchange between plug-ins. In his vision, plug-ins should make all data they compute public and available to other plug-ins for consumption. To share data, he suggests that plug-ins use meta-models to describe the data they produce in a unified system. Our approach also features a unifying component – the tuple set exchange format – which we believe is easier for developers to understand and use to compose complex queries.

8.6.3 Visualization of information

There is a wealth of tools for visualizing information related to software. However these are typically coupled to a specific kind of inquiry. Most tools that provide general query capabilities provide only a single way to view results or only basic visualization flexibility. Common result presentations are list or tree views [dAM08, DRNKJ11, FM10, JDV03, KMM11, WGRG10] and graphs [LM11, SEHM06]. Some tools allow configurable views or advanced interfaces. The prototype implementation for Fritz and Murphy's information fragment model [FM10] presents the results in a tree view, which allows different projections of the data affecting the hierarchy

of objects in the tree. SemmleCode [MSV+08] can present results as a list or as a number of predefined chart types. Reacher [LM11], Stacksplorer [KKD+11], and Blaze [KKKB12] enable the interactive exploration of call graphs. Our approach is more general and, unlike these other tools, offers flexible and easily extensible visualizations. We believe it is possible to implement such specialized visualizations and interfaces within Envision.

8.6.4 Scripting actions and refactoring

Existing program querying tools that go beyond displaying information are typically limited to refactoring code. To support the implementation of complex or project-specific refactorings, researchers have designed scripting languages for refactoring such as JunGL [VEdM06] or Rascal [HKV12], which offer powerful capabilities to analyze and transform the source code. Our system also enables complex refactorings via Python scripts and it offers two major advantages. First, scripts can use external resources in addition to the source code, enabling data-driven refactorings. Second, modifying code is just a special-case for our system's general support for automating arbitrary actions.

Conclusion and future work 9

In this dissertation, we have presented techniques for overcoming notational, information, and tooling deficiencies of state-of-the-art IDEs. We have challenged the well-established practice of developing software by directly editing text files and demonstrated an alternative approach that enables expert developers to understand code more easily and to work more flexibly with information, thanks to smart and collaborating tools. To validate our approach, we have developed the Envision IDE, which integrates all proposed techniques, demonstrating their usability and synergies between them.

Summary

Decoupling the programming interface from the way programs are stored on disk unlocks a number of opportunities for transforming programming environments and potentially improving the efficiency of software development. In a lab study, we observed that a visually rich code notation that mixes textual and graphical elements can reduce the time that programmers need to answer questions about method structure by up to 75%. Contrary to the subjective opinion of most participants from our evaluation, we did not measure any visual overload with richer code notations. These results provide an important and counterintuitive insight about code notations, and offer a new direction for tool designers and researchers to explore, beyond the ubiquitous and decades-old syntax highlighting. We also demonstrated an approach for customizing code visualizations according to context, e.g., the domain of an API. To make these highly customizable and visual program notations directly editable, we introduced a number of keyboard-based interaction techniques and evaluated them using CogTool simulations. The simulations show that expert users can be as efficient when typing in a structured editor, as they can be in a text editor. Thus, our proposed programming interface combines high flexibility and usability, which makes it a promising step towards developing practical visual interfaces for professional developers.

Taking advantage of unrestricted information structures, we designed accurate version control algorithms that can use dense data (AST node IDs) stored within source files. Our diff algorithm tracks moved pieces of code and efficiently computes precise and fine-grained deltas between code versions, preventing inaccurate or confusing diffs. Our merge algorithm and domain-specific customizations eliminate merges leading to incorrect program structures, reduce unnecessary conflicts, and report semantic issues, improving the merge result. Our evaluation shows that our approach achieves a substantial improvement in merge quality, automation, and error reporting compared to a standard line-based version control tool. These results

show not only that essential programming practices can be improved by enriched information structures, but also that such improvements can be achieved while continuing to use existing infrastructure, such as GitHub, making our approach more practical than alternatives requiring a dedicated storage backend.

Finally, we showed an approach for turning the IDE into a powerful and customizable information system that integrates diverse information sources and tools. Due to its familiar command interface, flexible visualizations, and scripting support, our system allows composing built-in queries and custom extensions with minimal effort, in order to answer their questions and automate actions. To achieve this result we had to integrate many aspects of our system, demonstrating that the various techniques we designed fit together well.

In addition to providing insight and novel techniques to tool designers and researchers, our work has also focused on developing the Envision IDE as a platform for experimentation. We have purposefully developed an open-source tool with a coherent vision for extensibility and modularity, so that we and others can easily use it to prototype and evaluate novel ideas. Envision has served us extremely well, allowing us to effectively integrate all of our work in a single system and to get a more complete understanding of the advantages and open challenges of such a next-generation integrated development environment.

Future work

Our work provides a basis for the future exploration of several directions.

In order to further validate our approach, we would like to achieve self-hosting for Envision, allowing us to develop the system using Envision itself and obviating the need for a separate IDE. Self-hosting is an important milestone for any development tool, as it establishes the tool's practicality and applicability to real-world problems. We have already invested significant effort in the biggest conceptual challenge for self-hosting: enabling Envision to import its own code-base of over 150 000 lines of code written in C++. Unlike languages such as Java, C++ relies on a textual preprocessor for achieving modularity (via `#include` directives) and for achieving certain kinds of genericity (e.g., macros and conditional compilation). Because Envision's tree-based program model does not support such purely textual features, we implemented a high-level meta-programming facility [Lüt15] modeled after the structured macro systems of languages such as Scala and also developed a C++ import plug-in for Envision, which automatically translates C++ preprocessor features to meta-programming constructs on a case-by-case basis. Many additional engineering challenges still remain until Envision can be self-hosting, e.g., supporting the full expressiveness of C++ within Envision's program model, and improving support for smart services such as code completion for C++.

A promising area for future research is further improving programming interfaces. To maximize improvements in efficiency, it is worth exploring additional context-specific interfaces and also evaluating the effects of richer visualizations on programming activities other than comprehension, such as writing and debugging code, code review, and live coding. Classifying the information needs of developers in different contexts could be used to guide the design of appropriate programming visualizations and interfaces. Not only the information available in interfaces, but also their visual appeal needs to be investigated. In our user study, many participants expressed concerns with the aesthetics of Envision's visualizations. To gain a better

understanding of the role of aesthetics, it is worth further improving Envision's visualizations with the help of experienced graphical designers and experimenting with these updated code presentations.

Decoupling programming interfaces from text also enables the exploration of novel non-visual interfaces, for example, for visually impaired developers. Such developers currently rely mostly on screen readers that work directly with the program's text, but an approach that works with the logical structure of the program, while still focusing on keyboard-based input, might yield better interfaces. Such interfaces would work with the additional information available from the semantics and structure of a program, which enables further interface customization to suit the needs of visually impaired developers. Envision is well suited for exploring interfaces for such developers, since the visual abilities of a user can be considered another type of context to which interfaces could be adapted.

Finally, another avenue for future research is additional enhancement of information structures within programs, allowing additional information integration and more semantic tools. For example, in order to reduce duplication and inconsistencies, program fragments could be automatically generated from integrated documentation such as tables or diagrams. To help understanding and maintenance, it is worth exploring information structures for recording system design decisions and their rationale directly within the code. Working directly with a rich program structure opens many opportunities for better semantic tools. For example, we have started exploring a merge customization that can detect renamings of declarations in an AST in one revision and apply them automatically to another on merge. Such high-level merge customizations might help to further reduce conflicts and detect additional semantic incompatibilities between revisions. An alternative might even be to record meta-operations such as renamings directly in Envision's rich program structure and make these accessible to version control tools. Such meta-data would help with selective undo and provenance, needed by tools for advanced navigation over program versions, e.g., [YM15]. Richer information structures could also be used to improve navigation within the IDE, which takes a significant time for developers [KAM05a]. Navigation information could be directly stored as part of the program and may include, for example, waypoints similar to tagSEA [SCBR06], but also groups of code locations that pertain to some feature, a table of contents for programs, or even guided tutorials that introduce new programmers to a particular part of a program or show the steps for a routine task. Smart tools are not limited to working with only static data, and static and run-time information can be mixed. For example, we have done preliminary experiments with setting breakpoints from within interactive queries, pausing the queries mid-way in order to allow the program to run, and collecting data during its execution when a breakpoint is hit. This ability to collected run-time data effectively makes this data available as another information source for queries and paves the way for execution-based refactorings or algorithm visualizations that enhance classical approaches such as Incense [Mye83] and BALSA [BS84] with support for customization and query refinement. We see the continuing transformation of IDEs into full-fledged information systems as promising and inevitable given the high complexity of today's software. As approaches for integrating data resources like Envision's information system evolve, we speculate that standards will emerge. These standards should make it easier to integrate information, breaking down barriers to cooperation between tools and making developers more productive.

Bibliography

[3ds] *Autodesk 3ds Max – official website*. http://www.autodesk.com/products/3ds-max/overview. Accessed: 2017-02-02.

[AB15] A. Altadmri and N. C. Brown. *37 million compilations: Investigating novice programming mistakes in large-scale student data*. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, 522–527 (ACM, New York, NY, USA, 2015). ISBN 978-1-4503-2966-8. doi:10.1145/2676723.2677258. URL http://doi.acm.org/10.1145/2676723.2677258.

[ABBS14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. *Learning natural coding conventions*. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 281–293 (ACM, New York, NY, USA, 2014). ISBN 978-1-4503-3056-5. doi:10.1145/2635868.2635883. URL http://doi.acm.org/10.1145/2635868.2635883.

[AGMO17] D. Asenov, B. Guenat, P. Müller, and M. Otth. *Precise version control of trees with line-based version control systems*. In *Fundamental Approaches to Software Engineering (FASE)* (2017). To appear.

[AHM16] D. Asenov, O. Hilliges, and P. Müller. *The effect of richer visualizations on code comprehension*. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, 5040–5045 (ACM, New York, NY, USA, 2016). ISBN 978-1-4503-3362-7. doi:10.1145/2858036.2858372. URL http://doi.acm.org/10.1145/2858036.2858372.

[AHR11] T. Alves, J. Hage, and P. Rademaker. *A comparative study of code query technologies*. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, 145–154 (2011). doi:10.1109/SCAM.2011.14.

[ALB+11] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. *Semistructured merge: Rethinking merge in revision control systems*. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, 190–200 (ACM, New York, NY, USA, 2011). ISBN 978-1-4503-0443-6. doi:10.1145/2025113.2025141. URL http://doi.acm.org/10.1145/2025113.2025141.

[ALL12] S. Apel, O. Leßenich, and C. Lengauer. *Structured merge with auto-tuning: Balancing precision and performance.* In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, 120–129 (ACM, New York, NY, USA, 2012). ISBN 978-1-4503-1204-2. doi:10.1145/2351676.2351694. URL http://doi.acm.org/10.1145/2351676.2351694.

[AM13] D. Asenov and P. Müller. *Customizing the visualization and interaction for embedded domain-specific languages in a structured editor.* In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, 127–130 (2013). ISSN 1943-6092. doi:10.1109/VLHCC.2013.6645255.

[AM14a] D. Asenov and P. Müller. *Envision: A fast and flexible visual code editor with fluid interactions.* Technical report, ETH-Zürich, 2014. doi:10.3929/ethz-a-010140807. Available at www.pm.inf.ethz.ch/publications.

[AM14b] D. Asenov and P. Müller. *Envision: A fast and flexible visual code editor with fluid interactions (overview).* In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 9–12 (2014). doi:10.1109/VLHCC.2014.6883014.

[AMV16] D. Asenov, P. Müller, and L. Vogel. *The ide as a scriptable information system.* In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 444–449 (ACM, New York, NY, USA, 2016). ISBN 978-1-4503-3845-5. doi:10.1145/2970276.2970329. URL http://doi.acm.org/10.1145/2970276.2970329.

[ASK10] K. Altmanninger, W. Schwinger, and G. Kotsis. *Semantics for accurate conflict detection in smover: Specification, detection and presentation by example.* IJEIS, 6(1):68–84, 2010. doi:10.4018/jeis.2010120206. URL http://dx.doi.org/10.4018/jeis.2010120206.

[ASW09] K. Altmanninger, M. Seidl, and M. Wimmer. *A survey on model versioning approaches.* International Journal of Web Information Systems, 5(3):271–304, 2009. doi:10.1108/17440080910983556. http://dx.doi.org/10.1108/17440080910983556, URL http://dx.doi.org/10.1108/17440080910983556.

[Ato] *Atom Zen mode.* https://atom.io/packages/zen. Accessed: 2017-02-06.

[Ber83] J. Bertin. *Semiology of graphics: diagrams, networks, maps* (University of Wisconsin press, 1983).

[BG05] B. E. Birnbaum and K. J. Goldman. *Achieving flexibility in direct-manipulation programming environments by relaxing the edit-time grammar.* In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 259–266 (IEEE Computer Society, Washington, DC, USA, 2005). ISBN 0-7695-2443-5. doi:10.1109/VLHCC.2005.15. URL http://dl.acm.org/citation.cfm?id=1092357.1092400.

[Bil05] P. Bille. *A survey on tree edit distance and related problems.* Theor. Comput. Sci., 337(1-3):217–239, 2005. ISSN 0304-3975. doi:10.1016/j.tcs.2004.12.030. URL http://dx.doi.org/10.1016/j.tcs.2004.12.030.

[BJRT10] R. Bellamy, B. John, J. Richards, and J. Thomas. *Using CogTool to model programming tasks.* In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, 1:1–1:6 (ACM, New York, NY, USA, 2010). ISBN 978-1-4503-0547-1. doi:10.1145/1937117.1937118. URL http://doi.acm.org/10.1145/1937117.1937118.

[BM86] R. Baecker and A. Marcus. *Design principles for the enhanced presentation of computer program source text.* In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, 51–58 (ACM, New York, NY, USA, 1986). ISBN 0-89791-180-6. doi:10.1145/22627.22348. URL http://doi.acm.org/10.1145/22627.22348.

[BRZ+10] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. *Code Bubbles: rethinking the user interface paradigm of integrated development environments.* In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, 455–464 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-719-6. doi:http://doi.acm.org/10.1145/1806799.1806866. URL http://doi.acm.org/10.1145/1806799.1806866.

[BS84] M. H. Brown and R. Sedgewick. *A system for algorithm animation.* In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, 177–186 (ACM, New York, NY, USA, 1984). ISBN 0-89791-138-5. doi:10.1145/800031.808596. URL http://doi.acm.org/10.1145/800031.808596.

[BZR+10] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. *Code Bubbles: a working set-based interface for code understanding and maintenance.* In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, 2503–2512 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-929-9. doi:http://doi.acm.org/10.1145/1753326.1753706. URL http://doi.acm.org/10.1145/1753326.1753706.

[CGG+85] R. Chandhok, D. Garlan, D. Goldenson, P. Miller, and M. Tucker. *Programming environments based on structure editing: The gnome approach.* Managing Requirements Knowledge, International Workshop on, 00(undefined):359, 1985. doi:doi.ieeecomputersociety.org/10.1109/AFIPS.1985.47.

[CKM06] M. J. Coblenz, A. J. Ko, and B. A. Myers. *Jasper: an eclipse plug-in to facilitate software maintenance tasks.* In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, eclipse '06,

65–69 (ACM, New York, NY, USA, 2006). ISBN 1-59593-621-1. doi: http://doi.acm.org/10.1145/1188835.1188849. URL http://doi.acm.org/10.1145/1188835.1188849.

[Cla] *clang: a C language family frontend for LLVM*. https://clang.llvm.org/. Accessed: 2017-02-02.

[Coda] *Code Contracts editor extensions for Microsoft Visual Studio*. https://visualstudiogallery.msdn.microsoft.com/02de7066-b6ca-42b3-8b3c-2562c7fa024f. Accessed: 2017-02-02.

[Codb] *Microsoft Code Contracts – official website*. https://www.microsoft.com/en-us/research/project/code-contracts/. Accessed: 2017-02-02.

[Con14] S. Conversy. *Unifying textual and visual: A theoretical account of the visual perception of programming languages*. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, 201–212 (ACM, New York, NY, USA, 2014). ISBN 978-1-4503-3210-1. doi:10.1145/2661136.2661138. URL http://doi.acm.org/10.1145/2661136.2661138.

[Coo10] S. Cooper. *The design of Alice*. Trans. Comput. Educ., 10:15:1–15:16, 2010. ISSN 1946-6226. doi:http://doi.acm.org/10.1145/1868358.1868362. URL http://doi.acm.org/10.1145/1868358.1868362.

[Cow87] M. Cowlishaw. *Lexx - a programmable structured editor*. IBM Journal of Research and Development, 31(1):73–80, 1987. ISSN 0018-8646. doi:10.1147/rd.311.0073.

[CSKB+89] B. Curtis, S. B. Sheppard, E. Kruesi-Bailey, J. Bailey, and D. A. Boehm-Davis. *Experimental evaluation of software documentation formats*. J. of Systems and Software, 9(2):167 – 207, 1989. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/0164-1212(89)90019-8. URL http://www.sciencedirect.com/science/article/pii/0164121289900198.

[Cum14] G. Cumming. *The new statistics: Why and how*. Psychological Science, 25(1):7–29, 2014. doi:10.1177/0956797613504966. http://pss.sagepub.com/content/25/1/7.full.pdf+html, URL http://pss.sagepub.com/content/25/1/7.abstract.

[CZ11] P. Caserta and O. Zendra. *Visualization of the static aspects of software: A survey*. Visualization and Computer Graphics, IEEE Transactions on, 17(7):913–933, 2011. ISSN 1077-2626. doi:10.1109/TVCG.2010.110.

[dAM08] B. de Alwis and G. Murphy. *Answering conceptual queries with ferret*. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, 21–30 (2008). ISSN 0270-5257. doi: 10.1145/1368088.1368092.

[dAMR07] B. de Alwis, G. Murphy, and M. Robillard. *A comparative study of three program exploration tools.* In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, 103–112 (2007). ISSN 1092-8138. doi:10.1109/ICPC.2007.6.

[DBR+12] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. *Debugger Canvas: industrial experience with the Code Bubbles paradigm.* In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, 1064–1073 (IEEE Press, Piscataway, NJ, USA, 2012). ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337362.

[Dij68] E. W. Dijkstra. *Letters to the editor: Go to statement considered harmful.* Commun. ACM, 11(3):147–148, 1968. ISSN 0001-0782. doi:10.1145/362929.362947. URL http://doi.acm.org/10.1145/362929.362947.

[Dim15] G. M. Dimitri. *The impact of syntax highlighting in sonic pi.* In *Proceedings of the 26th Annual Conference of the Psychology of Programming Interest Group (PPIG 2015)* (2015).

[DK10] S. Davis and G. Kiczales. *Registration-based language abstractions.* In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, 754–773 (ACM, New York, NY, USA, 2010). ISBN 978-1-4503-0203-6. doi:10.1145/1869459.1869521. URL http://doi.acm.org/10.1145/1869459.1869521.

[DMJN07] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. *Refactoring-aware configuration management for object-oriented programs.* In *29th International Conference on Software Engineering (ICSE'07)*, 427–436 (2007). ISSN 0270-5257. doi:10.1109/ICSE.2007.71.

[DR10] R. DeLine and K. Rowan. *Code canvas: zooming towards better development environments.* In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, 207–210 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-719-6. doi:http://doi.acm.org/10.1145/1810295.1810331. URL http://doi.acm.org/10.1145/1810295.1810331.

[Dra16] P. Dragicevic. *Fair statistical communication in HCI.* In J. Robertson and M. Kaptein (editors), *Modern Statistical Methods for HCI* (Springer, 2016). In press.

[DRNKJ11] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. *The soul tool suite for querying programs in symbiosis with eclipse.* In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, 71–80 (ACM, New York, NY, USA, 2011). ISBN 978-1-4503-0935-6. doi:10.1145/2093157.2093168. URL http://doi.acm.org/10.1145/2093157.2093168.

[DT14] L. Diekmann and L. Tratt. *Eco: A language composition editor*. In B. Combemale, D. Pearce, O. Barais, and J. Vinju (editors), *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, 82–101 (Springer International Publishing, 2014). ISBN 978-3-319-11244-2. doi:10.1007/978-3-319-11245-9_5. URL http://dx.doi.org/10.1007/978-3-319-11245-9_5.

[DVR10] R. DeLine, G. Venolia, and K. Rowan. *Software development with code maps*. Commun. ACM, 53:48–54, 2010. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/1787234.1787250. URL http://doi.acm.org/10.1145/1787234.1787250.

[EA04] T. Ekman and U. Asklund. *Refactoring-aware versioning in eclipse*. Electron. Notes Theor. Comput. Sci., 107:57–69, 2004. ISSN 1571-0661. doi:10.1016/j.entcs.2004.02.048. URL http://dx.doi.org/10.1016/j.entcs.2004.02.048.

[Ecl] *Eclipse – official website*. https://eclipse.org. Accessed: 2017-02-02.

[ED66] T. G. Evans and D. L. Darley. *On-line debugging techniques: A survey*. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), 37–50 (ACM, New York, NY, USA, 1966). doi:10.1145/1464291.1464295. URL http://doi.acm.org/10.1145/1464291.1464295.

[EK07] A. D. Eisenberg and G. Kiczales. *Expressive programs through presentation extension*. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, 73–84 (ACM, New York, NY, USA, 2007). ISBN 1-59593-615-7. doi:http://doi.acm.org/10.1145/1218563.1218573. URL http://doi.acm.org/10.1145/1218563.1218573.

[EM95] M. Erwig and B. Meyer. *Heterogeneous visual languages-integrating visual and textual programming*. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, VL '95, 318–325 (IEEE Computer Society, Washington, DC, USA, 1995). ISBN 0-8186-7045-2. URL http://dl.acm.org/citation.cfm?id=832276.834317.

[Env] *Envision webpage at ETH Zurich*. http://www.pm.inf.ethz.ch/research/envision.html. Accessed: 2017-02-02.

[FBL10] M. Fähndrich, M. Barnett, and F. Logozzo. *Embedded contract languages*. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 2103–2110 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-639-7. doi:10.1145/1774088.1774531. URL http://doi.acm.org/10.1145/1774088.1774531.

[FKA+13] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. *Do background colors improve program comprehension in the #ifdef hell?* Empirical Software Engineering, 18(4):699–745, 2013. ISSN 1382-3256.

doi:10.1007/s10664-012-9208-x. URL http://dx.doi.org/10.1007/s10664-012-9208-x.

[Flu08] B. Fluri. *Change distilling. Enriching software evolution analysis with fine-grained source code change histories.* Ph.D. thesis, 2008. URL http://www.zora.uzh.ch/16421/.

[FM10] T. Fritz and G. C. Murphy. *Using information fragments to answer the questions developers ask.* In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, 175–184 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-719-6. doi:10.1145/1806799.1806828. URL http://doi.acm.org/10.1145/1806799.1806828.

[FMB+14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus. *Fine-grained and accurate source code differencing.* In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, 313–324 (ACM, New York, NY, USA, 2014). ISBN 978-1-4503-3013-8. doi:10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982.

[Fow05] M. Fowler. *A language workbench in action - mps.* [Online] Available: http://martinfowler.com/articles/mpsAgree.html, 2005.

[FWPG07] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. *Change distilling: Tree differencing for fine-grained source code change extraction.* IEEE Trans. Softw. Eng., 33(11):725–743, 2007. ISSN 0098-5589. doi:10.1109/TSE.2007.70731. URL http://dx.doi.org/10.1109/TSE.2007.70731.

[GKM90] E. P. Glinert, M. E. Kopache, and D. W. McIntyre. *Exploring the general-purpose visual alternative.* JVLC, 1(1):3 – 39, 1990. ISSN 1045-926X. doi:10.1016/S1045-926X(05)80032-1. URL http://www.sciencedirect.com/science/article/pii/S1045926X05800321.

[GM84] D. B. Garlan and P. L. Miller. *Gnome: An introductory programming environment based on a family of structure editors.* In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, 65–72 (ACM, New York, NY, USA, 1984). ISBN 0-89791-131-8. doi:10.1145/800020.808250. URL http://doi.acm.org/10.1145/800020.808250.

[GP92] T. Green and M. Petre. *When visual programs are harder to read than textual programs.* In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 57 (Citeseer, 1992).

[GP96] T. Green and M. Petre. *Usability analysis of visual programming environments: A "Cognitive Dimensions" framework.* JVLC, 7(2):131 – 174, 1996. ISSN 1045-926X. doi:10.1006/jvlc.1996.0009. URL http://www.sciencedirect.com/science/article/pii/S1045926X96900099.

[Gre89] T. R. G. Green. *Cognitive dimensions of notations.* 443–460, 1989. URL http://dl.acm.org/citation.cfm?id=92968.93015.

[Gre90] T. R. G. Green. *The cognitive dimension of viscosity: A sticky problem for hci.* In *Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*, INTERACT '90, 79–86 (North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1990). ISBN 0-444-88817-9. URL http://dl.acm.org/citation.cfm?id=647402.725762.

[Gue15] B. Guenat. *Tree-based Version Control in Envision.* BSc. Thesis, ETH Zurich, 2015.

[GWGG12] G. Ghezzi, M. Würsch, E. Giger, and H. C. Gall. *An architectural blueprint for a pluggable version control system for software (evolution) analysis.* In *Proceedings of the Second International Workshop on Developing Tools As Plug-Ins*, TOPI '12, 13–18 (IEEE Press, Piscataway, NJ, USA, 2012). ISBN 978-1-4673-1820-4. URL http://dl.acm.org/citation.cfm?id=2667062.2667065.

[HCM02] D. Hendrix, I. Cross, J.H., and S. Maghsoodloo. *The effectiveness of control structure diagrams in source code comprehension activities.* Software Engineering, IEEE Transactions on, 28(5):463–477, 2002. ISSN 0098-5589. doi:10.1109/TSE.2002.1000450.

[HF14] A. Z. Henley and S. D. Fleming. *The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes.* In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, 2511–2520 (ACM, New York, NY, USA, 2014). ISBN 978-1-4503-2473-1. doi:10.1145/2556288.2557073. URL http://doi.acm.org/10.1145/2556288.2557073.

[HKV12] M. Hills, P. Klint, and J. J. Vinju. *Scripting a refactoring with rascal and eclipse.* In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, 40–49 (ACM, New York, NY, USA, 2012). ISBN 978-1-4503-1500-5. doi:10.1145/2328876.2328882. URL http://doi.acm.org/10.1145/2328876.2328882.

[HN86] A. N. Habermann and D. Notkin. *Gandalf: Software development environments.* IEEE Transactions on Software Engineering, SE-12(12):1117–1127, 1986. ISSN 0098-5589. doi:10.1109/TSE.1986.6313007.

[HNS06] T. Hakala, P. Nykyri, and J. Sajaniemi. *An experiment on the effects of program code highlighting on visual search for local patterns.* Psychology of Programming Interest Group, 38–52, 2006.

[HRS84] K. Hammond and V. Rayward-Smith. *A survey on syntactic error recovery and repair.* Computer Languages, 9(1):51 – 67, 1984. ISSN 0096-0551. doi:http://dx.doi.org/10.1016/0096-0551(84)90012-2. URL http://www.sciencedirect.com/science/article/pii/0096055184900122.

[HVM06] E. Hajiyev, M. Verbaere, and O. Moor. *ECOOP 2006 – Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings*, chapter codeQuest: Scalable Source Code Queries

with Datalog, 2–27 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006). ISBN 978-3-540-35727-8. doi:10.1007/11785477_2. URL http://dx.doi.org/10.1007/11785477_2.

[JDT] *Eclipse JDT Core Component*. https://eclipse.org/jdt/core/. Accessed: 2017-02-02.

[JDV03] D. Janzen and K. De Volder. *Navigating and querying code without getting lost*. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, 178–187 (ACM, New York, NY, USA, 2003). ISBN 1-58113-660-9. doi:10.1145/643603.643622. URL http://doi.acm.org/10.1145/643603.643622.

[JPSK04] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger. *Predictive human performance modeling made easy*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, 455–462 (ACM, New York, NY, USA, 2004). ISBN 1-58113-702-8. doi:10.1145/985692.985750. URL http://doi.acm.org/10.1145/985692.985750.

[Jus] *GitHub project: justanothercoder/Compiler*. https://github.com/justanothercoder/Compiler/blob/79a44744be646cf8edfeefff44c28f776889ed6e/threeaddresscode.hpp. Accessed: 2017-02-02.

[KAM05a] A. J. Ko, H. Aung, and B. A. Myers. *Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks*. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, 126–135 (ACM, New York, NY, USA, 2005). ISBN 1-58113-963-2. doi:http://doi.acm.org/10.1145/1062455.1062492. URL http://doi.acm.org/10.1145/1062455.1062492.

[KAM05b] A. J. Ko, H. H. Aung, and B. A. Myers. *Design requirements for more flexible structured editors from a study of programmers' text editing*. In *CHI '05 extended abstracts on Human factors in computing systems*, CHI EA '05, 1557–1560 (ACM, New York, NY, USA, 2005). ISBN 1-59593-002-7. doi:http://doi.acm.org/10.1145/1056808.1056965. URL http://doi.acm.org/10.1145/1056808.1056965.

[KDRN+11] A. Kellens, C. De Roover, C. Noguera, R. Stevens, and V. Jonckers. *Reasoning over the evolution of source code using quantified regular path expressions*. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 389–393 (2011). ISSN 1095-1350. doi:10.1109/WCRE.2011.54.

[KDV07] A. J. Ko, R. DeLine, and G. Venolia. *Information needs in collocated software development teams*. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, 344–353 (IEEE Computer Society, Washington, DC, USA, 2007). ISBN 0-7695-2828-7. doi:http://dx.doi.org/10.1109/ICSE.2007.45. URL http://dx.doi.org/10.1109/ICSE.2007.45.

[KH10] M. Koegel and J. Helming. *Emfstore: A model repository for emf models.* In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, 307–308 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-719-6. doi:10.1145/1810295.1810364. URL http://doi.acm.org/10.1145/1810295.1810364.

[KHvWH10] M. Koegel, M. Herrmannsdoerfer, O. von Wesendonk, and J. Helming. *Operation-based conflict detection.* In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, 21–30 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-960-2. doi:10.1145/1826147.1826154. URL http://doi.acm.org/10.1145/1826147.1826154.

[Kil00] M. J. Kilgard. *A practical and robust bump-mapping technique for today's gpus.* In *Game Developers Conference 2000* (2000).

[KK14] T. Kehrer and U. Kelter. *Versioning of ordered model element sets.* Technical Report 2, University of Siegen, 2014.

[KKD+11] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. 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, 217–224 (ACM, New York, NY, USA, 2011). ISBN 978-1-4503-0716-1. doi:10.1145/2047196.2047225. URL http://doi.acm.org/10.1145/2047196.2047225.

[KKKB12] J.-P. Krämer, J. Kurz, T. Karrer, and J. 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, 2195–2200 (ACM, New York, NY, USA, 2012). ISBN 978-1-4503-1016-1. doi:10.1145/2212776.2223775. URL http://doi.acm.org/10.1145/2212776.2223775.

[KM06] A. J. Ko and B. A. Myers. *Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors.* In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, 387–396 (ACM, New York, NY, USA, 2006). ISBN 1-59593-372-7. doi:http://doi.acm.org/10.1145/1124772.1124831. URL http://doi.acm.org/10.1145/1124772.1124831.

[KMC12] T. Kosar, M. Mernik, and J. Carver. *Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments.* Empirical Software Engineering, 17:276–304, 2012. ISSN 1382-3256. 10.1007/s10664-011-9172-x, URL http://dx.doi.org/10.1007/s10664-011-9172-x.

[KMM11] M. Kimmig, M. Monperrus, and M. Mezini. *Querying source code with natural language.* In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, 376–379 (IEEE Computer Society, Washington, DC, USA, 2011). ISBN 978-1-4577-1638-6. doi:10.1109/ASE.2011.6100076. URL http://dx.doi.org/10.1109/ASE.2011.6100076.

[Kuh12] A. Kuhn. *Ides need become open data platforms (as need languages and vms)*. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, 31–36 (2012). doi:10.1109/TOPI.2012.6229807.

[Lab] *LabView – official website*. https://www.ni.com/labview/. Accessed: 2017-02-02.

[LBM14] T. Lieber, J. R. Brandt, and R. C. Miller. *Addressing misconceptions about code with always-on programming visualizations*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, 2481–2490 (ACM, New York, NY, USA, 2014). ISBN 978-1-4503-2473-1. doi:10.1145/2556288.2557409. URL http://doi.acm.org/10.1145/2556288.2557409.

[Lév75] J. P. Lévy. *Automatic correction of syntax-errors in programming languages*. Acta Informatica, 4(3):271–292, 1975. ISSN 1432-0525. doi:10.1007/BF00288730. URL http://dx.doi.org/10.1007/BF00288730.

[Lin04] T. Lindholm. *A three-way merge for xml documents*. In *Proceedings of the 2004 ACM Symposium on Document Engineering*, DocEng '04, 1–10 (ACM, New York, NY, USA, 2004). ISBN 1-58113-938-1. doi:10.1145/1030397.1030399. URL http://doi.acm.org/10.1145/1030397.1030399.

[LL96] M. C. Longo and P. Lockhart. *Structured cabling: foundations for the future*. Healthcare information management: journal of the Healthcare Information and Management Systems Society of the American Hospital Association, 10(4):59, 1996.

[LM10] T. D. LaToza and B. A. Myers. *Developers ask reachability questions*. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, 185–194 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-719-6. doi:10.1145/1806799.1806829. URL http://doi.acm.org/10.1145/1806799.1806829.

[LM11] T. LaToza and B. Myers. *Visualizing call graphs*. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, 117–124 (2011). ISSN 1943-6092. doi:10.1109/VLHCC.2011.6070388.

[LR13] D. H. Lorenz and B. Rosenan. *Source code management for projectional editing*. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, 83–84 (ACM, New York, NY, USA, 2013). ISBN 978-1-4503-1995-9. doi:10.1145/2508075.2508092. URL http://doi.acm.org/10.1145/2508075.2508092.

[Lüt15] P. Lüthi. *Self-hosting the Envision Visual Programming Environment*. MSc. Thesis, ETH Zurich, 2015.

[Mar94] R. Martin. *Oo design quality metrics*. An analysis of dependencies, 12:151–170, 1994.

[McI01] L. K. McIver. *Syntactic and semantic issues in introductory programming education*. Ph.D. thesis, Monash University, School of Computer Science and Software Engineering, 2001.

[McK12] F. McKay. *A prototype structured but low-viscosity editor for novice programmers*. In *Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers*, BCS-HCI '12, 363–368 (British Computer Society, Swinton, UK, UK, 2012). URL http://dl.acm.org/citation.cfm?id=2377916.2377967.

[MCPW08] L. Murta, C. Corrêa, J. a. G. Prudêncio, and C. Werner. *Towards odyssey-vcs 2: Improvements over a uml-based version control system*. In *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*, CVSM '08, 25–30 (ACM, New York, NY, USA, 2008). ISBN 978-1-60558-045-6. doi:10.1145/1370152.1370159. URL http://doi.acm.org/10.1145/1370152.1370159.

[Men02] T. Mens. *A state-of-the-art survey on software merging*. IEEE Transactions on Software Engineering, 28(5):449–462, 2002. ISSN 0098-5589. doi:10.1109/TSE.2002.1000449.

[MM85] W. Miller and E. W. Myers. *A file comparison program*. Software: Practice and Experience, 15(11):1025–1040, 1985. ISSN 1097-024X. doi:10.1002/spe.4380151102. URL http://dx.doi.org/10.1002/spe.4380151102.

[MPMV94] P. Miller, J. Pane, G. Meter, and S. Vorthmann. *Evolution of novice programming environments: The structure editors of carnegie mellon university*. Interactive Learning Environments, 4(2):140–158, 1994.

[MPS] *MPS – official website*. https://www.jetbrains.com/mps/. Accessed: 2017-02-02.

[MRR+10] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. *The Scratch programming language and environment*. Trans. Comput. Educ., 10(4):16:1–16:15, 2010. ISSN 1946-6226. doi:10.1145/1868358.1868363. URL http://doi.acm.org/10.1145/1868358.1868363.

[MSH+16] B. A. Myers, A. Stefik, S. Hanenberg, A.-J. Kaijanaho, M. Burnett, F. Turbak, and P. Wadler. *Usability of programming languages: Special interest group (sig) meeting at chi 2016*. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '16, 1104–1107 (ACM, New York, NY, USA, 2016). ISBN 978-1-4503-4082-3. doi:10.1145/2851581.2886434. URL http://doi.acm.org/10.1145/2851581.2886434.

[MST12] S. Mathôt, D. Schreij, and J. Theeuwes. *Opensesame: An open-source, graphical experiment builder for the social sciences*. Behavior Research Methods, 44(2):314–324, 2012. doi:10.3758/s13428-011-0168-7. URL http://dx.doi.org/10.3758/s13428-011-0168-7.

[MSV+08] O. Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, chapter .QL: Object-Oriented Queries Made Easy, 78–133 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008). ISBN 978-3-540-88643-3. doi:10.1007/978-3-540-88643-3_3. URL http://dx.doi.org/10.1007/978-3-540-88643-3_3.

[MTN+13] R. Mikhaiel, N. Tsantalis, N. Negara, E. Stroulia, and Z. Xing. *Differencing UML Models: A Domain-Specific vs. a Domain-Agnostic Method*, 159–196 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013). ISBN 978-3-642-35992-7. doi:10.1007/978-3-642-35992-7_4. URL http://dx.doi.org/10.1007/978-3-642-35992-7_4.

[Mye83] B. A. Myers. *Incense: A system for displaying data structures*. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '83, 115–125 (ACM, New York, NY, USA, 1983). ISBN 0-89791-109-1. doi:10.1145/800059.801140. URL http://doi.acm.org/10.1145/800059.801140.

[Mye86] E. W. Myers. *An O(ND) difference algorithm and its variations*. Algorithmica, 1(1):251–266, 1986. ISSN 1432-0541. doi:10.1007/BF01840446. URL http://dx.doi.org/10.1007/BF01840446.

[Mye98] B. A. Myers. *The case for an open data model*. Technical report, Carnegie Mellon University, 1998.

[NK12] J.-J. Nuñez and G. Kiczales. *Understanding registration-based abstractions: A quantitative user study*. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, 93–102 (2012). ISSN 1092-8138. doi:10.1109/ICPC.2012.6240513.

[NMB05] T. Nguyen, E. Munson, and J. Boyland. *An infrastructure for development of object-oriented, multi-level configuration management services*. In *Proceedings of the 27th International Conference on Software Engineering, (ICSE 2005)*, 215–224 (2005). doi:10.1109/ICSE.2005.1553564.

[NND+15] H. V. Nguyen, M. H. Nguyen, S. C. Dang, C. Kästner, and T. N. Nguyen. *Detecting semantic merge conflicts with variability-aware execution*. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 926–929 (ACM, New York, NY, USA, 2015). ISBN 978-1-4503-3675-8. doi:10.1145/2786805.2803208. URL http://doi.acm.org/10.1145/2786805.2803208.

[NNPN10] T. Nguyen, H. Nguyen, N. Pham, and T. Nguyen. *Operation-based, fine-grained version control model for tree-based representation*. In D. Rosenblum and G. Taentzer (editors), *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, 74–90 (Springer Berlin Heidelberg, 2010). ISBN 978-3-642-12028-2. doi:10.1007/978-3-642-12029-9_6. URL http://dx.doi.org/10.1007/978-3-642-12029-9_6.

[OLDR11] F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes. *Enabling program comprehension through a visual object-focused development environment.* In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, 127 –134 (2011). ISSN 1943-6092. doi:10.1109/VLHCC.2011.6070389.

[OMW05] H. Oliveira, L. Murta, and C. Werner. *Odyssey-vcs: A flexible version control system for uml model elements.* In *Proceedings of the 12th International Workshop on Software Configuration Management*, SCM '05, 1–16 (ACM, New York, NY, USA, 2005). ISBN 1-59593-310-7. doi:10.1145/1109128.1109129. URL http://doi.acm.org/10.1145/1109128.1109129.

[Ope] *Open clipart.* https://openclipart.org/. Accessed: 2017-02-03.

[OVH15] J. Ou, M. Vechev, and O. Hilliges. *An interactive system for data structure development.* In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, 3053–3062 (ACM, New York, NY, USA, 2015). ISBN 978-1-4503-3145-6. doi:10.1145/2702123.2702319. URL http://doi.acm.org/10.1145/2702123.2702319.

[OVH+16] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer. *Hazelnut: A bidirectionally typed structure editor calculus.* CoRR, abs/1607.04180, 2016. URL http://arxiv.org/abs/1607.04180.

[PBL+16] T. W. Price, N. C. Brown, D. Lipovac, T. Barnes, and M. Kölling. *Evaluation of a frame-based programming editor.* In *ICER '16* (ACM, 2016).

[PBMM15] J. Protzenko, S. Burckhardt, M. Moskal, and J. McClurg. *Implementing real-time collaboration in touchdevelop using ast merges.* In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*, MobileDeLi 2015, 25–27 (ACM, New York, NY, USA, 2015). ISBN 978-1-4503-3906-3. doi:10.1145/2846661.2846672. URL http://doi.acm.org/10.1145/2846661.2846672.

[Pet95] M. Petre. *Why looking isn't always seeing: readership skills and graphical programming.* Commun. ACM, 38:33–44, 1995. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/203241.203251. URL http://doi.acm.org/10.1145/203241.203251.

[PM93] J. F. Pane and P. L. Miller. *The acse multimedia science learning environment.* In *Proceedings of the 1993 International Conference on Computers in Education*, 168–173 (1993).

[QGV] *The Qt Graphics View Framework – official website.* https://doc.qt.io/qt-5/graphicsview.html. Accessed: 2017-02-02.

[Que] *QueryDSL – official website.* http://www.querydsl.com. Accessed: 2017-02-02.

[Rea] *GitHub project: ReactiveX/RxJava (manual merge).* https://github.com/ReactiveX/RxJava/pull/281/commits/1667386b61de0c9f1157dce157bf2529e3715cb6. Accessed: 2017-02-02.

[Rei08] S. P. Reiss. *Tracking source locations.* In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, 11–20 (ACM, New York, NY, USA, 2008). ISBN 978-1-60558-079-1. doi:10.1145/1368088.1368091. URL http://doi.acm.org/10.1145/1368088.1368091.

[RW91] J. R. Rasure and C. S. Williams. *An integrated data flow visual language and software development environment.* JVLC, 2(3):217 – 246, 1991. ISSN 1045-926X. doi:10.1016/S1045-926X(06)80007-8. URL http://www.sciencedirect.com/science/article/pii/S1045926X06800078.

[Sar15] A. Sarkar. *The impact of syntax colouring on program comprehension.* In *Proceedings of the 26th Annual Conference of the Psychology of Programming Interest Group (PPIG 2015)*, 49–58 (2015).

[SCBR06] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. *Waypointing and social tagging to support program navigation.* In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, 1367–1372 (ACM, New York, NY, USA, 2006). ISBN 1-59593-298-4. doi:10.1145/1125451.1125704. URL http://doi.acm.org/10.1145/1125451.1125704.

[SCC06] C. Simonyi, M. Christerson, and S. Clifford. *Intentional software.* In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, 451–464 (ACM, New York, NY, USA, 2006). ISBN 1-59593-348-4. doi:http://doi.acm.org/10.1145/1167473.1167511. URL http://doi.acm.org/10.1145/1167473.1167511.

[SEHM06] T. Schafer, M. Eichberg, M. Haupt, and M. Mezini. *The sextant software exploration tool.* IEEE Transactions on Software Engineering, 32(9):753–768, 2006. ISSN 0098-5589. doi:10.1109/TSE.2006.94.

[SH06] Z. Suvajdžin and M. Hajduković. *A structure editor for the program composing assistant.* volume 3, 65–76 (2006).

[SL12] T. Schiller and B. Lucia. *Playing cupid: The ide as a matchmaker for plug-ins.* In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, 1–6 (2012). doi:10.1109/TOPI.2012.6229805.

[SMDV06] J. Sillito, G. C. Murphy, and K. De Volder. *Questions programmers ask during software evolution tasks.* In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, 23–34 (ACM, New York, NY, USA, 2006). ISBN 1-59593-468-5. doi:http://doi.acm.org/10.1145/1181775.1181779. URL http://doi.acm.org/10.1145/1181775.1181779.

[SMDV08] J. Sillito, G. Murphy, and K. De Volder. *Asking and answering questions during a programming change task.* Software Engineering, IEEE Transactions on, 34(4):434–451, 2008. ISSN 0098-5589. doi: 10.1109/TSE.2008.26.

[SS13] A. Stefik and S. Siebert. *An empirical investigation into programming language syntax.* Trans. Comput. Educ., 13(4):19:1–19:40, 2013. ISSN 1946-6226. doi:10.1145/2534973. URL http://doi.acm.org/10.1145/2534973.

[SSE15] C. Sadowski, K. T. Stolee, and S. Elbaum. *How developers search for code: A case study.* In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 191–201 (ACM, New York, NY, USA, 2015). ISBN 978-1-4503-3675-8. doi:10.1145/2786805.2786855. URL http://doi.acm.org/10.1145/2786805.2786855.

[SUW15] F. Schwägerl, S. Uhrig, and B. Westfechtel. *A graph-based algorithm for three-way merging of ordered collections in {EMF} models.* Science of Computer Programming, 113, Part 1:51 – 81, 2015. ISSN 0167-6423. doi:http://dx.doi.org/10.1016/j.scico.2015.02.008. Model Driven Development (Selected & extended papers from {MODELSWARD} 2014), URL http://www.sciencedirect.com/science/article/pii/S0167642315000532.

[TR81] T. Teitelbaum and T. Reps. *The cornell program synthesizer: A syntax-directed programming environment.* Commun. ACM, 24(9):563–573, 1981. ISSN 0001-0782. doi:10.1145/358746.358755. URL http://doi.acm.org/10.1145/358746.358755.

[Ukk85] E. Ukkonen. *International conference on foundations of computation theory algorithms for approximate string matching.* Information and Control, 64(1):100 – 118, 1985. ISSN 0019-9958. doi:http://dx.doi.org/10.1016/S0019-9958(85)80046-2. URL http://www.sciencedirect.com/science/article/pii/S0019995885800462.

[UM15] R.-G. Urma and A. Mycroft. *Source-code queries with graph databases—with application to programming language usage and evolution.* Science of Computer Programming, 97, Part 1:127 – 134, 2015. ISSN 0167-6423. doi:http://dx.doi.org/10.1016/j.scico.2013.11.010. Special Issue on New Ideas and Emerging Results in Understanding Software, URL http://www.sciencedirect.com/science/article/pii/S0167642313002943.

[VEdM06] M. Verbaere, R. Ettinger, and O. de Moor. *Jungl: A scripting language for refactoring.* In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, 172–181 (ACM, New York, NY, USA, 2006). ISBN 1-59593-375-1. doi:10.1145/1134285.1134311. URL http://doi.acm.org/10.1145/1134285.1134311.

[Vim] *Vim – official website.* http://www.vim.org. Accessed: 2017-02-02.

[Visa] *The vis.js Javascript library for rendering interactive data – official website.* http://visjs.org. Accessed: 2017-02-02.

[Visb] *Visual Studio Code.* https://code.visualstudio.com/. Accessed: 2017-02-06.

[Vog15] L. Vogel. *Augmenting software development with information scripting.* MSc. Thesis, ETH Zurich, 2015.

[VSBK14] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. *Towards user-friendly projectional editors.* In B. Combemale, D. Pearce, O. Barais, and J. Vinju (editors), *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, 41–61 (Springer International Publishing, 2014). ISBN 978-3-319-11244-2. doi:10.1007/978-3-319-11245-9_3. URL http://dx.doi.org/10.1007/978-3-319-11245-9_3.

[Wes10] B. Westfechtel. *A formal approach to three-way merging of emf models.* In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, 31–41 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-960-2. doi:10.1145/1826147.1826155. URL http://doi.acm.org/10.1145/1826147.1826155.

[WGRG10] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. *Supporting developers with natural language queries.* In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, 165–174 (ACM, New York, NY, USA, 2010). ISBN 978-1-60558-719-6. doi:10.1145/1806799.1806827. URL http://doi.acm.org/10.1145/1806799.1806827.

[Wie91] S. Wiedenbeck. *The initial stage of program comprehension.* International Journal of Man-Machine Studies, 35(4):517 – 540, 1991. ISSN 0020-7373. doi:http://dx.doi.org/10.1016/S0020-7373(05)80090-2. URL http://www.sciencedirect.com/science/article/pii/S0020737305800902.

[Wil04] G. V. Wilson. *Extensible programming for the 21st century.* Queue, 2(9):48–57, 2004. ISSN 1542-7730. doi:10.1145/1039511.1039534. URL http://doi.acm.org/10.1145/1039511.1039534.

[Wil12] R. Wilcox. *Chapter 4 - confidence intervals in the one-sample case.* In R. Wilcox (editor), *Introduction to Robust Estimation and Hypothesis Testing (Third Edition)*, Statistical Modeling and Decision Science, 103 – 136 (Academic Press, Boston, 2012), third edition edition. ISBN 978-0-12-386983-8. doi:http://dx.doi.org/10.1016/B978-0-12-386983-8.00004-4. URL http://www.sciencedirect.com/science/article/pii/B9780123869838000044.

[Wil14] R. Williams. *The non-designer's design book: design and typographic principles for the visual novice* (Pearson Education, 2014).

[WKL+06] J. M. Wolfe, K. R. Kluender, D. M. Levi, L. M. Bartoshuk, R. S. Herz, R. L. Klatzky, S. J. Lederman, and D. M. Merfeld. *Sensation & perception* (Sinauer Sunderland, MA, 2006).

[WNF06] K. N. Whitley, L. R. Novick, and D. Fisher. *Evidence in favor of visual representation for the dataflow paradigm: An experiment testing labview's comprehensibility.* International Journal of Human-Computer Studies, 64(4):281 – 303, 2006. ISSN 1071-5819. doi: 10.1016/j.ijhcs.2005.06.005. URL http://www.sciencedirect.com/science/article/pii/S1071581905001163.

[Wol11] D. Wolber. *App inventor and real-world motivation.* In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, 601–606 (ACM, New York, NY, USA, 2011). ISBN 978-1-4503-0500-6. doi:10.1145/1953163.1953329. URL http://doi.acm.org/10.1145/1953163.1953329.

[YM14] Y. S. Yoon and B. A. Myers. *A longitudinal study of programmers' backtracking.* In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 101–108 (2014). ISSN 1943-6092. doi:10.1109/VLHCC.2014.6883030.

[YM15] Y. S. Yoon and B. A. Myers. *Supporting selective undo in a code editor.* In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, 223–233 (IEEE Press, Piscataway, NJ, USA, 2015). ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818784.

Appendices

Supplementary material for user-study participants

A

Comparison between *Eclipse*, *Envision reduced*, and *Envision default*

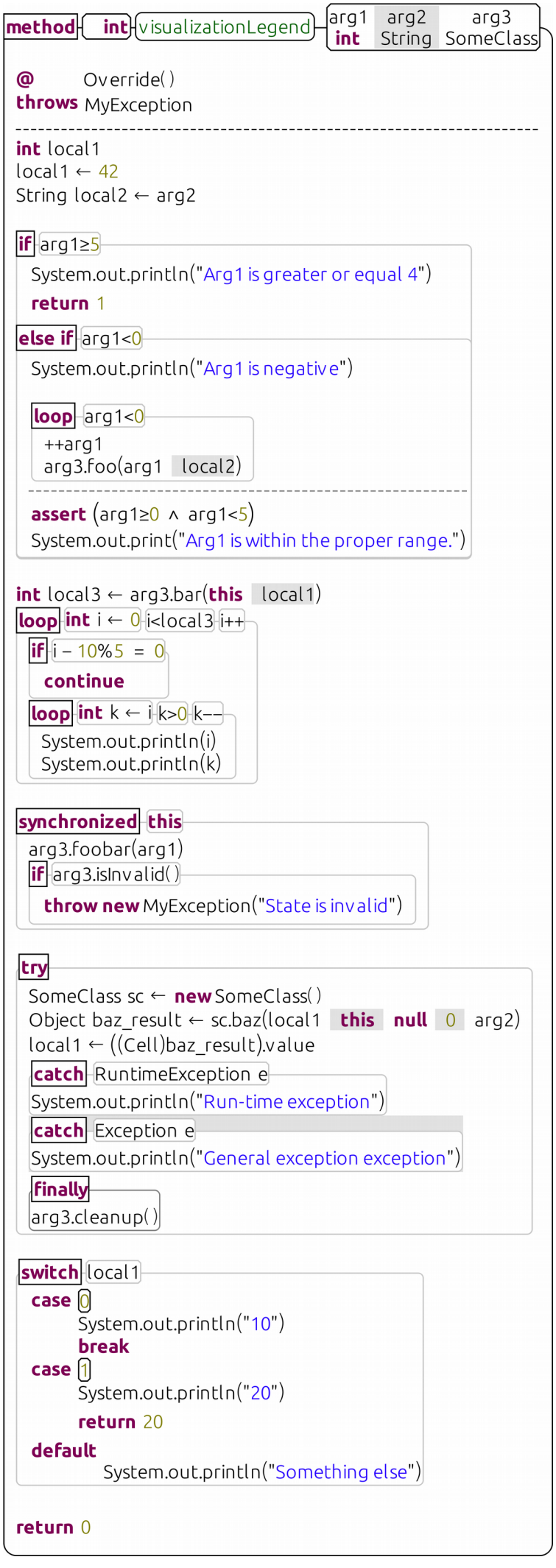
Differences:

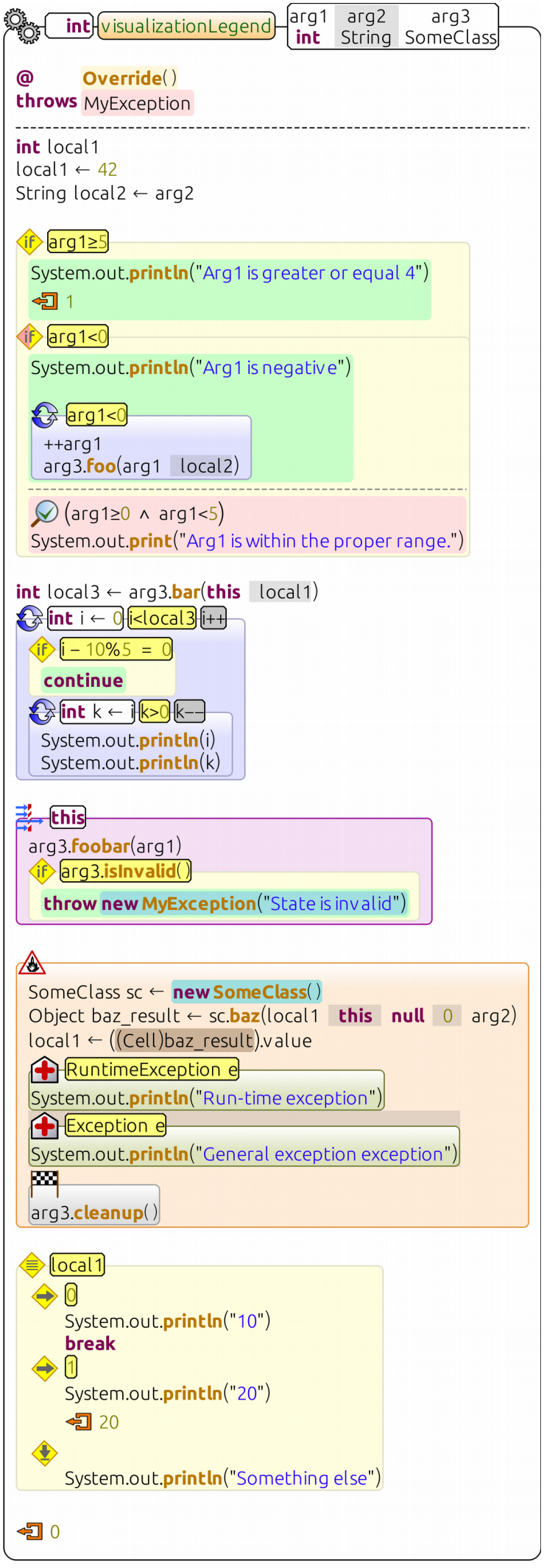
Feature	Eclipse	Envision reduced	Envision default
block/body	`{ }`	*outlines*	
method icon	*none*	**method**	
method parameters	`(String x, int y)`	x y String **int**	
signature end/ body start	`{ }`	`------`	
local variable	`SomeType foo;`	SomeType foo	
assignment	`=`	←	
comparison	`==`	=	
operators	`<= >= != && || !`	≤ ≥ ≠ ∧ ∨ ¬	
semicolon	`;`	*no semicolon*	
lists (e.g. method args)	`a, b, c, d`	a b c d	
method/constructor call	`foo(a, b);`	foo(a b)	**foo**(a b)
object creation	**new** `Foo(x);`	**new** Foo(x)	**new** Foo(x)
cast	`(String) foo`	(String) foo	(String) foo
return	**return** `x;`	**return** x	 x
assert	**assert** `x;`	**assert** x	 x
if	**if**	**if**	 *background*
else if	**else if**	**else if**	 *background*
then branch	`{ }`	*white background*	*background*
else branch	**else** `{ }`	`------`	`------` *background*
loops	**for** **while** **do**	**loop**	 *background*
try	**try**	**try**	 *background*
catch	**catch**	**catch**	 *background*
finally	**finally**	**finally**	 *background*
synchronized	**synchronized**	**synchronized**	 *background*
switch	**switch**	**switch**	 *background*
case	**case**	**case**	 *background*
default case	**default**	**default**	 *background*

No differences:

Feature	Eclipse	Envision reduced	Envision default
null	**null**	**null**	
this	**this**	**this**	
throw	**throw**	**throw**	
break, continue	**break** **continue**	**break** **continue**	
operators	`+ - * / ++ --`	+ − * / ++ −−	

```
@Override
public int visualizationLegend(int arg1, String arg2, SomeClass arg3)
throws MyException
{
	int local1;
	local1 = 42;
	String local2 = arg2;

	if (arg1 >= 5) {
		System.out.println("Arg1 is greater or equal 4");
		return 1;
	} else if (arg1 <0) {
		System.out.println("Arg1 is negative");

		while (arg1<0) {
			++arg1;
			arg3.foo(arg1, local2);
		}
	}
	else {
		assert(arg1 >=0 && arg1 <5);
		System.out.print("Arg1 is within the proper range.");
	}

	int local3 = arg3.bar(this, local1);
	for(int i = 0; i<local3; i++) {
		if (i - 10 % 5 == 0) continue;
		for (int k = i; k>0; k--) {
			System.out.println(i);
			System.out.println(k);
		}
	}

	synchronized(this) {
		arg3.foobar(arg1);
		if (arg3.isInvalid())
			throw new MyException("State is invalid");
	}

	try {
		SomeClass sc = new SomeClass();
		Object baz_result = sc.baz(local1, this, null, 0, arg2);
		local1 = ((Cell)baz_result).value;
	}
	catch(RuntimeException e) {
		System.out.println("Run-time exception");
	}
	catch(Exception e) {
		System.out.println("General exception exception");
	}
	finally {
		arg3.cleanup();
	}

	switch(local1) {
		case 0:
			System.out.println("10");
			break;
		case 1:
			System.out.println("20");
			return 20;
		default:
			System.out.println("Something else");
	}

	return 0;
}
```


method int visualizationLegend arg1 int arg2 String arg3 SomeClass
@ Override()
throws MyException
int local1
local1 ← 42
String local2 ← arg2
if arg1≥5
System.out.println("Arg1 is greater or equal 4")
return 1
else if arg1<0
System.out.println("Arg1 is negative")
loop arg1<0
++arg1
arg3.foo(arg1 local2)
assert (arg1≥0 ∧ arg1<5)
System.out.print("Arg1 is within the proper range.")
int local3 ← arg3.bar(this local1)
loop int i ← 0 i<local3 i++
if i − 10%5 = 0
continue
loop int k ← i k>0 k−−
System.out.println(i)
System.out.println(k)
synchronized this
arg3.foobar(arg1)
if arg3.isInvalid()
throw new MyException("State is invalid")
try
SomeClass sc ← new SomeClass()
Object baz_result ← sc.baz(local1 this null 0 arg2)
local1 ← ((Cell)baz_result).value
catch RuntimeException e
System.out.println("Run-time exception")
catch Exception e
System.out.println("General exception exception")
finally
arg3.cleanup()
switch local1
case 0
System.out.println("10")
break
case 1
System.out.println("20")
return 20
default
System.out.println("Something else")
return 0


int visualizationLegend
arg1 arg2 arg3
int String SomeClass
@ Override()
throws MyException
int local1
local1 ← 42
String local2 ← arg2
if arg1≥5
System.out.println("Arg1 is greater or equal 4")
1
if arg1<0
System.out.println("Arg1 is negative")
arg1<0
++arg1
arg3.foo(arg1 local2)
(arg1≥0 ∧ arg1<5)
System.out.print("Arg1 is within the proper range.")
int local3 ← arg3.bar(this local1)
int i ← 0 i<local3 i++
if i – 10%5 = 0
continue
int k ← i k>0 k--
System.out.println(i)
System.out.println(k)
this
arg3.foobar(arg1)
if arg3.isInvalid()
throw new MyException("State is invalid")
SomeClass sc ← new SomeClass()
Object baz_result ← sc.baz(local1 this null 0 arg2)
local1 ← ((Cell)baz_result).value
RuntimeException e
System.out.println("Run-time exception")
Exception e
System.out.println("General exception exception")
arg3.cleanup()
local1
0
System.out.println("10")
break
1
System.out.println("20")
20
System.out.println("Something else")
0

Python and Javascript files for interactive rendering of graphs

B

The `vis.js` Javascript library can be used to render an interactive graph inside a web browser. We also use the library to show interactive graphs directly within Envision. To achieve this, it is necessary to write a wrapper HTML file that uses the library, similar to the one shown here:

```
<html>
<head>
  <script type="text/javascript"
    src="file://@path/scripts/vis-graph/vis.js"></script>
  <link href="file://@path/scripts/vis-graph/vis.css"
    rel="stylesheet" type="text/css" />
</head>
<body>
<div id="graph"></div>
<script type="text/javascript">
  var nodes = new vis.DataSet([ @nodes ]);
  var edges = new vis.DataSet([ @edges ]);

  var container = document.getElementById('graph');
  var data = { nodes: nodes, edges: edges };
  var options = {};
  var network = new vis.Network(container, data, options);
</script>
</body>
</html>
```

The `vis.js` and `vis.css` files used above can be downloaded from [Visa]. `@nodes` and `@edges` are placeholders for the elements of the graph. They are substituted with strings that correspond to the structure of a graph within Envision. The strings are computed by the Python script on the next page.

```
import os
import random
import json

def getNames(node):
  if node:
    names = getNames(node.parent)
    if node.symbolName():
      names.append(node.symbolName())
    return names
  else:
    return []

def getLabel(tuple):
  if isinstance(tuple.value, Node):
    return '.'.join(getNames(tuple.value)[-2:])
  return str(tuple.value)

def getId(tuple):
  if isinstance(tuple.value, Node):
    return '.'.join(getNames(tuple.value))
  return str(tuple.value)

nodeLabels = {}
nodes = []
edges = []

#Handle color tuples specially:
if len(Query.args) > 0 and Query.args[0] == 'useColor':
  for colorTuple in Query.input.take('color'):
    nodeId = getId(colorTuple[1])
    nodeLabels[nodeId] = nodeId
    nodes.append(json.dumps({'id': nodeId, 'label': nodeId,
                             'color': colorTuple.color}))

# Record all nodes and their labels
for tuple in Query.input.tuples():
  for i in range(0, tuple.size()):
    id = getId(tuple[i])
    if not id in nodeLabels:
      valueLabel = getLabel(tuple[i])
      nodeLabels[id] = valueLabel
      nodes.append(json.dumps({'id': id, 'label': valueLabel}))

# Record edges
for tuple in Query.input.tuples():
  if tuple.size() >= 2:
    edges.append(json.dumps({'from': getId(tuple[0]),
                             'to': getId(tuple[1]), 'arrows':'to'}))
nodesText = ','.join(nodes)
edgesText = ','.join(edges)

with open('scripts/vis-graph/vis-graph.html') as htmlFile:
  htmlText = htmlFile.read().replace('@path', os.path.abspath("."))
         .replace('@nodes', nodesText).replace('@edges', edgesText)
  t = Tuple([('html', htmlText)])
  Query.result.add(t)
```