Augmenting software development with information scripting

Master Thesis

Lukas Vogel

Supervisors: Dimitar Asenov, Prof. Dr. Peter Müller

Chair of Programming Methodology

ETH Zürich

December 1, 2015

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Software projects are associated with an overwhelming amount of information in form of the source code and in additional information sources such as version control systems, debuggers, bug trackers, task management systems, and more. Yet current integrated development environments lack the possibility to effortlessly combine information from different information sources.

We introduce a query framework that lets developers access, combine, and modify information from different sources through short textual queries. Our system also supports Python scripting which enables the user to extend our system. We integrate the query framework in Envision, a visual programming environment, which enables us to visualize the results of queries. We show the usefulness of our tool with an example set of 25 questions that developers find difficult to answer.

Contents

List of Figures

Listings

Acknowledgements

I would like to express my gratitude to my supervisor Dimitar Asenov for all the support throughout the whole thesis. His advises on both technical and writing related issues were extremely helpful.

Furthermore I want to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this project. My thank also goes to everyone who participated in the insightful discussions after the intermediate presentations.

I would like to thank my parents for supporting through the course of the studies. Last but not least I would like to thank my loved ones who supported me enormously.

1 Introduction

To manage increasingly large software projects various tools, such as version control systems, bug trackers, task management systems, debuggers, analysis tools, and others are used. The source code in combination with all the various tools delivers essential information to developers involved with the software project. Research has shown that engineers spend a significant amount of time searching and combining information [4, 17, 18, 15, 13]. Yet existing tools often lack the ability to combine information.

Consider an imaginary developer Bob, who is debugging a new regression. Bob updated his code version with the version control system. He finds the updated code crashes somewhere after the current breakpoint in the method `addTracks()`. To find the cause Bob would like to know what changes occurred in the callgraph of the method `addTracks()` during the last week. Bob thus has to answer the more abstract question: "Find recent changes in the callgraph of function `x`". To answer this, Bob needs to combine two pieces of information: 1. recent changes, 2. the callgraph of method `x`. Finding those two pieces of information independently is well supported by IDEs through plug-ins. To combine the information gathered from different information sources, is however a manual and often tedious task. Bob would have to look through all the changes and see which of the changes affect methods in the callgraph of the method `addTracks()`.

In this work we aim to design and build an IDE framework that enables developers to query and combine different sources of information. Through a query prompt the user of our system should be able to specify what information he requests and how it should be combined through short textual queries. Answering Bob's question in our system should be as simple as opening a query prompt on the `addTracks()` method and typing the command: `callgraph|changes -c 5`, where the argument `-c 5` indicates that last 5 commits should be considered.

The framework should be integrated in Envision. Envision is a prototype IDE [1], which uses visual and textual elements to represent source code. To present the results of queries we would like to make use of the visual representation of source code artifacts in Envision.

The problem of combining software related information has been approached before [3, 4, 16]. We present contributions in the following areas:

1. Extensibility by the user: Through Python scripts users can extend our system easily.

2. Presentation of the results: By using the visual nature of Envision we can present answers to questions in a visual way.

3. Modification of source data: A user of our system can not only search for information but he can also use the results to modify the underlying data.

This work presents many challenges. First of all we have to design a data format which is suitable to contain information from various sources without knowing all of them in advance. This format should simplify the composition of different queries. A second challenge is the integration of Python scripts in the system. Since the IDE and the rest of the framework is built in C++ we have to enable the interaction between the Python scripts and the C++ framework. A third challenge we see in finding suitable visualizations for the results of queries. Depending on the question a developer asks, different visualizations make sense.

The rest of this report is structured as follows: We first present a vision of the final system and from this deduce the goals on the essential characteristics of our system in chapter 2. We then present the individual components of our system: The data format we use to pass data between queries in chapter 3, queries which extract information from existing information sources in chapter 4, and the presentation of the results from queries in chapter 5. In chapter 6 we present details about the implementation of our system. We discuss related work in chapter 7. We evaluate and compare our system to related work in chapter 8. We end this report with an outlook on future work in chapter 9 and a conclusion in chapter 10. In appendix A we present all available queries in the current system.

2 Vision and overview of the system

In this chapter we present our vision of the final system. Section 2.1 presents usage scenarios we aim to support. From the usage scenarios we derive the goals on the essential characteristics of the system in section 2.2. Section 2.3 describes the concept of an information source. In section 2.4 we briefly introduce the components of our system. Our system is heavily inspired by the Unix shell and we explain the similarities to it in section 2.5.

2.1 Usage scenarios

The final system should be able to help a developer in various activities. This ranges from simple searches, to complex refactorings implemented with the help of a script. To motivate such a system we present the following scenarios:

2.1.1 Find all recently introduced recursive methods

Alice dislikes recursive methods because they can often be replaced by faster iterative methods. As a project leader she wants from time to time get an overview of new recursive methods in the code base.

Alice works with Eclipse and uses the version control plug-in to find recent changes. To find new recursive methods she has to manually go through all changes. Whenever she finds a new method she notes the name of the method. Later she manually checks which of the noted methods is recursive.

We believe that all those steps can be automated. With applying semantics to the change set a tool should be able to extract all methods from it. Another tool could work on the AST and check if a method is recursive by analyzing its call graph. That would leave us with two *modules* that can answer the two individual parts of this problem.

We want our system to be able to *compose* the two *modules* in a *simple* way. Alice should not have more overhead than typing a few queries and specifying how they are connected:

```
callgraph -global | canReach -self -r calls | changes -nodes
```

The `callgraph` query gets the call relations of all methods, then the `canReach` query extracts all methods that can call themselves. The `changes` query filters the methods in the input and returns only the recently changed methods. The system should automatically highlight all the methods that are returned by the last query.

2.1.2 Show the coverage of all recently changed statements

Code coverage analysis is well suited to report what code is tested by a certain test case. However to get code coverage data specific coverage tools are required.

Carol had the idea of building a coverage analysis using only a debugger and a few helper programs. The idea is simple: With a small program she extracts all statements in a project and then sets a breakpoint on each of the statements. By recording which breakpoints were hit during the execution of the program and automatically resuming them, Carol can get an overview of which statements were executed.

While existing tools are very good in analyzing and showing coverage, it requires a lot of manual work to extract which of the statements have recently changed.

We would like to be able to support Carol's idea in our query system so that we can *compose* it with version control information. A query to do this could look like this:

```
changes -t Statement -nodes | addBreakpoints | traceExecution | yield | heatmap
```

The `changes` query gets all recently changed statements, to which breakpoints are added with the `addBreakpoints` query. The `traceExecution` query executes the program and makes sure that all hit breakpoints will be recorded. The `yield` command specifies that the `heatmap` query should only run once the program started by `tracExecution` has finished.

The result is presented as a heatmap, that means frequently tested statements appear in red and less frequently tested statements in green. Not tested statements are not highlighted at all.

2.1.3 Put a tag in all methods that take more than 10ms to execute

Bob works on a performance critical application. He needs to measure the performance and check which code parts are bottlenecks. To do this Bob uses the VisualVM profiling tool to create a CSV file which contains timing information for all methods that were executed. He usually goes through the profiling data an marks all methods that run longer than 10 ms with a tag `SLOW` in the source code. Whenever he has time Bob searches for the `SLOW` tag and checks if he can optimize something in the slow methods.

Bob finds this approach quite cumbersome for two reasons: First associating the profile data with the source code is quite time consuming because he has to search for all the methods in the source code. Second, the tags he adds to the source code as comments clutter the source code and make it less clear.

We want our system to be *extensible* with scripts, so that a script can find the association between the profile data and the source code. With this association it should be possible to automatically tag slow methods. Since the tags are only relevant for this task they should per default be hidden and only be shown when Bob requests them. The hiding of tags is enabled by Envision's capability to flexibly show and hide parts of the source code model.

The script which maps the methods from the profile data to the actual methods could look like the script in listing 2.1.

```python
import csv

def findMethod(profileName):
    # This assumes no nested classes/namespaces, i.e. the format should be:
    # ClassName.methodName(...)
    nameWithoutBracket = profileName.split('(')[0]
    astTuples = Query.methods(['-global', '-name', nameWithoutBracket], [])[0]
    # Assume we have exactly one matching:
    return astTuples.tuples('ast').pop().ast

with open(Query.args[0]) as csvfile:
    profileCSVReader = csv.reader(csvfile, delimiter=';', quotechar='"')
    next(profileCSVReader, None) # Ignore the header
    for row in profileCSVReader:
        if float(row[2]) > 10.0:
            m = findMethod(row[0])
            Query.result.add(Tuple([('ast', m)]))
```

Listing 2.1: Example of a script that extracts information from a file. It identifies all methods that take longer than 10ms to execute.

Adding tags to the extracted methods should be a matter of *composing* the script with the tags query:

```
extractSlowMethods profileData.csv | addTags -n SLOW
```

With such a system the whole task of Bob is automated. To find slow methods at a later time Bob can query for all `SLOW` tags and extract the AST nodes those tags are attached to:

```
tags -n SLOW | filter -extract tag.ast
```

The tag query returns all tags with the name `SLOW` and indicates to which AST nodes those tags are associated. The `filter` query then extracts the AST nodes so that they can be highlighted by the system.

2.1.4 Generating a visitor stub with a script

Dan wants to implement the visitor pattern. He wants a visitor to visit all classes that are subclasses of the class `Node`. To implement a proper visitor pattern *Dan* thus has to add `accept` methods in all subclasses of `Node` and a corresponding `visit` method in the visitor class.

With this task Dan faces two time consuming tasks: First he has to find all classes that are subclasses of Node. In a second step he then has to add the accept method in each class that he found and a corresponding visit method in the visitor.

In our system it should be possible to find all classes that are subclasses of Node through a query. With a script it should be possible to add all the required method stubs automatically. The query to do this would then look like this:

```
bases -global | canReach -r baseclasses -n Node | addToVisitor
```

It first gets the baseclass relation of all classes and then canReach only keeps classes which are subclasses of Node, finally the query calls the script to add the visitor stub. Since the script assumes that this query is executed on the the visitor class, it is important that the query prompt is opened at the correct location. The script which creates the method stubs is shown in listing 2.2.

```python
if not type(Query.target) is Class:
    raise Exception('Needs to be executed on a class')

visitorName = Query.target.name

def addArgument(argType, argName, toMethod):
    argument = Node.createNewNode('FormalArgument', None)
    argument.typeExpression = AstModification.buildExpression(argType)
    argument.name = argName
    toMethod.arguments.append(argument)

def createVoidMethod(name):
    m = Node.createNewNode('Method', None)
    m.name = name
    returnValue = Node.createNewNode('FormalResult', None)
    returnValue.typeExpression = AstModification.buildExpression('void')
    m.results.append(returnValue)
    return m

def addStatementFromString(expression, toMethod):
    nodeExpr = AstModification.buildExpression(expression)
    printStmt = Node.createNewNode('ExpressionStatement', None)
    printStmt.expression = nodeExpr
    toMethod.items.append(printStmt)

for astTuple in Query.input.tuples('ast'):
    if astTuple.ast.name == visitorName:
        continue

    # First create the accept method in the current class
    acceptMethod = createVoidMethod('accept')
    addArgument(visitorName, 'visitor', acceptMethod)
    addStatementFromString('visitor.visit(this)', acceptMethod)

    astTuple.ast.beginModification('adding method')
    astTuple.ast.methods.append(acceptMethod)
    astTuple.ast.endModification()

    # Now create the visit stub in the visitor class
    visitMethod = createVoidMethod('visit')
    addArgument(astTuple.ast.name, 'node', visitMethod)
    addStatementFromString('System.out.println("Visiting
        {}")'.format(astTuple.ast.name), visitMethod)

    Query.target.beginModification('adding method')
    Query.target.methods.append(visitMethod)
    Query.target.endModification()
```

Listing 2.2: Example of a script that creates a visitor stub. It adds a `accept` methods to all classes in the input and a `visit` method in the target class.

With this script *Dan* has the complete time consuming task automated. More details on the script API can be found in section 4.2.

2.2 Goals

From the use cases we can deduce four core characteristics which we aim to achieve.

Modularity Queries in our system should be able to work on their own, independently from other queries. For example a query to list methods with a certain name can be a standalone query and produce meaningful results.

Composability and interoperability It should be possible to combine different queries to gain connected information. That means queries need to be able to exchange information in a common format. For example the output of an AST query should be understood by a script or another query.

Extensibility The system should make it easy to add new types of queries. For simple cases it should be possible to create a Python script which can interact with other queries. For more advanced queries the framework should make it simple to contribute a query as a C++ plug-in for Envision.

Ease of use Our system should be easy to use. That means queries should have descriptive names and arguments. The combining of queries should be as simple as writing a pipe operator `|` to indicate data flow. Python scripts should have a well defined interface and should be able to access existing queries.

2.3 Information sources

A large software project has many information sources. An information source provides access to existing information in the software ecosystem. Information sources are used by the query system to gather and manipulate data. Examples of information sources are:

AST The AST information source contains all information that is related to the source code. In our case this information is stored in Envision's model.

Tags A tag in software is a bit of information attached to a certain code location. Tags can be useful to provide navigation in software[20, 19]. For example to add a new plug-in to Envision there are several steps involved, with the helps of tags a developer could be guided through those steps. As part of this project we added tags as an extension to Envision's model.

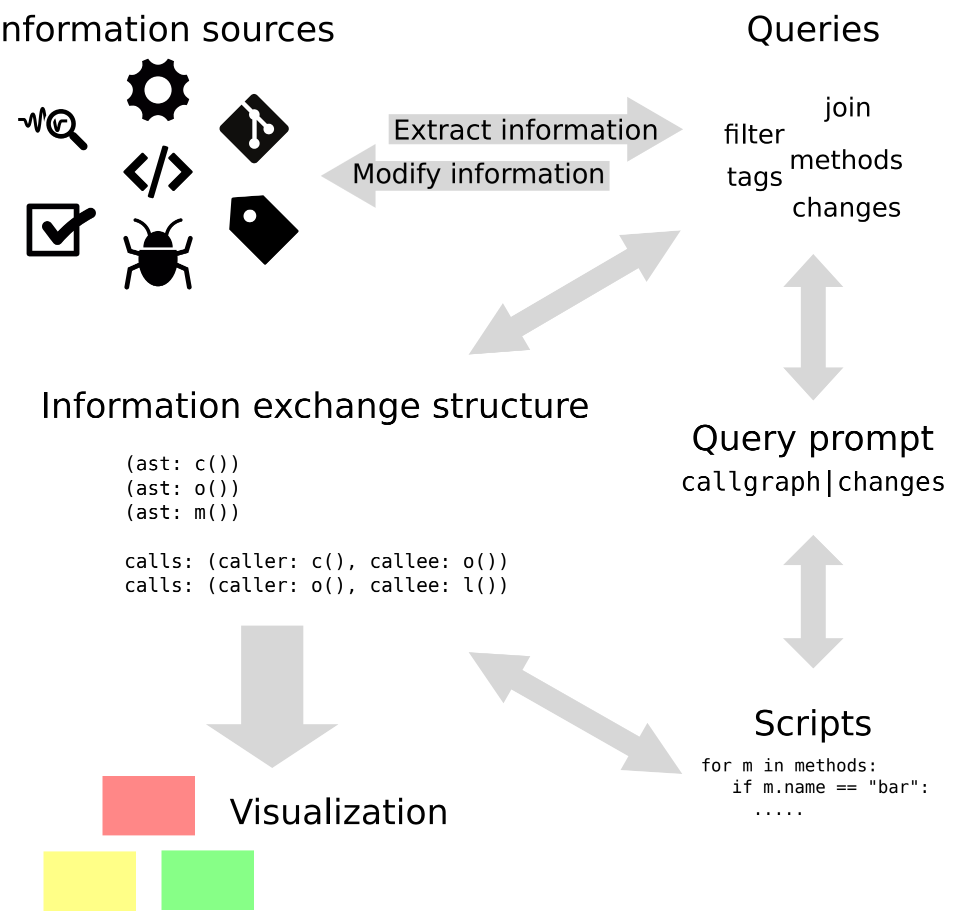
Version control The version control information source provides information about the change history, commit ids, authors of code, and other data. We use Envision's version control system built on top of Git [14, 5].

Debugger The debugger information source contains information about a program's execution state. It can also be used to modify and check breakpoints and the programs execution state. We use Envision's debugger.

Issue tracker A issue tracker provides information about open and closed tasks and bugs in a project. The current system does not support any issue tracker, but we believe that support could be added through a script.

2.4 System components

This section provides a brief overview on the components in the system. Each of the components is described in full detail in separate chapters.



```
(ast: c())
(ast: o())
(ast: m())

calls: (caller: c(), callee: o())
calls: (caller: o(), callee: l())
```

```
for m in methods:
   if m.name == "bar":
    .....
```

Figure 2.1: Overview over the components in the system[1].

Information exchange structure This is the data structure that all other components use to communicate with each other.

Queries Queries can both create data in the information exchange structure from any of the information sources, and manipulate data they receive as input.

Query prompt The prompt is used by the user to interact with our system. It enables the composition of queries in a flexible way.

Scripts Scripts provide the possibility for a user to create new queries in Python.

Visualization The visualization component is used to visualize the results of queries with visualization primitives like arrows and highlights.

[1]Icons made by Freepik, Linh Pham, Catalin Fertu from flaticon.com

2.5 Inspiration from Unix shell

Our system is heavily inspired by the Unix shell. We were especially motivated by the simplicity of combining commands with a single pipe character. This lets users flexibly choose what programs or queries they compose and in which way. All components of the system have an analog in the Unix shell:

System component	**Unix shell analog**
Information exchange structure	Lines of text
Queries	Binary programs (e.g. `ls`, `grep`)
Query prompt	Command line
Scripts	User scripts (e.g. `Bash`, `Python`, `Perl`, etc. scripts)
Visualization	Output on command line or to a file

Unix programs and queries likewise can both extract information existing in the system and work on the input. The user can define connections between programs through the input prompt and the data is passed in specific format between different programs. Through scripts both systems are user extensible with low effort.

3 Information exchange structure

This chapter discusses the unified information exchange structure. We analyze the core requirements for such a structure in section 3.1. We present set of tuples, the structure which fulfills all those requirements in section 3.2. In the early stages of this project we also explored an alternative graph structure which we discuss in section 3.3. In section 3.4 we evaluate the set of tuples.

3.1 Requirements

In chapter 2 we argue that we want a single information structure which can be understood by all queries of the final system. Our structure needs to be able to incorporate various bits of information from the whole software environment. Since we strive for extensibility we cannot know in advance what data the structure will contain. Nonetheless we know that what we store is always either an information bit or a relation between information bits.

3.1.1 Atoms

We call a single existing information bit *atom*, we have the following requirements for atoms:

Identifyability An atom of a certain information source has to be uniquely identifiable. If this would not be the case, the information source could not decide if a certain information piece is already contained in the structure.

Accessibility To make an atom accessible we require a tag. The tag defines how to interpret an information atom. The tag conveys both the information about the information source and the type of the atom. A tag should therefore not be reused for different types of information.

A good example for an information atom, is an AST node. The AST node is uniquely identifiable since each node is only once in the AST. With a tag “ast” this node gets accessible in the structure since the tag specifies that we talk about an AST node. Note that the exact tag name does not matter as long as we know what the underlying information presents.

3.1.2 Relations

We need a way to model relations between any information atoms. Relations require additional properties, examples include:

- Tags: To identify what this relation describes. For example between AST nodes we can have call relations as well as dataflow relations.
- Count: To describe multiple relations with the same tag between two atoms. For example in the callgraph from an execution trace a call to a method can be executed multiple times.

3.2 Set of tuples

Our proposed information exchange structure is a set of n-tuples with record type. The n-tuples have the following properties:

1. Elements are ordered, i.e. $(a, b, c) \neq (b, c, a)$
2. All elements are named with a tag, as described in section 3.1.1. We denote a tagged tuple as follows: `(tag: value)`
3. A tag can only appear once in a tuple, i.e. `(ast: x, ast: y)` is not allowed.
4. The number of elements is finite.
5. The n-tuple is identified with a tag, this is either an explicit tag or the tag of the first value. We denote a tuple with an explicit tag as follows: `explicit: (tag: value)`

With the help of the example code in listing 3.1 we explain how this format is used to store information.

```
// TODO
void c() {
    o();
}

// FIXME
void o() {
    l();
}

void l() {
    c();
}
```

Listing 3.1: Code example to illustrate how extracted information can be presented in a set of tuples.

To store information atoms, we can use a single valued tuple: For the methods extracted from the code in listing 3.1 the set of tuples is shown in listing 3.2.

```
{(ast: c()), (ast: o()), (ast: l())}
```

Listing 3.2: Set of tuples with the method nodes extracted from listing 3.1.

To represent a relation we can use a multi valued tuple: For the call relations in listing 3.1 the set of tuples is shown in listing 3.3.

```
{
 calls: (caller: c(), callee: o()),
 calls: (caller: o(), callee: l()),
 calls: (caller: l(), callee: c())
}
```

Listing 3.3: Set of tuples with call relations extracted form the code in listing 3.1.

To relate the `TODO` and `FIXME` tags in listing 3.1 with the corresponding code parts we can use the multi valued tuples shown in figure 3.4.

```
{(tag: TODO, ast: c()), (tag: FIXME, ast: o())}
```

Listing 3.4: Tuples that associate `TODO` and `FIXME` tags with the code shown in listing 3.1.

Since the format is a set of tuples all the shown information can be combined in a single set of tuples, as shown in listing 3.5.

```
{
 (ast: c()),
 (ast: o()),
 (ast: l()),
 calls: (caller: c(), callee: o()),
 calls: (caller: o(), callee: l()),
 calls: (caller: l(), callee: c()),
 (tag: TODO, ast: c()),
 (tag: FIXME, ast: o())
}
```

Listing 3.5: Set of tuple which contains various information about the code in listing 3.1.

3.3 Alternative structure – graph

Before selecting the set of tuples as our information exchange structure we evaluated a graph representation. In this graph both the nodes and the edges can hold any amount of information with the help of key value pairs. Such a structure fulfills all requirements stated in section 3.1.

Yet there are two main issues with this approach. Both of them stem from the observation that nodes in the graph have to be unique:

Consider the question: "What has changed in the call graph of `x()`". To answer this question we need to combine version control information of the method nodes, with the call graph of `x()`. We will only get more insight if the nodes representing the AST nodes from the version control information and the call graph information are the same if the AST node is the same.

3.3.1 Problem 1: node equality

If we want to combine information about the call graph and version control information in a graph, we would need a node equality test. However such a test is not easy to define since the nodes can have any amount of properties stored. In some cases it could be that two nodes should be treated as equal even though they have some amount of differing properties, for example because they refer to the same AST node. In other cases we want to treat nodes with differing properties differently, for example if we want to count the changes on an AST node per author, nodes with the same AST node value should not in every case be treated as equal.

3.3.2 Problem 2: node merging

Assuming a solution for problem 1 exists, we still have the problem of merging 2 equal nodes in a single node. Consider the union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. There might be a property conflict: $v_1 = v_2 \wedge v_1.x \neq v_2.x; v_1 \in V_1, v_2 \in V_2$. For such a conflict there are multiple resolution strategies, but the correct one can only be defined with more knowledge about the data.

3.3.3 Comparison with set of tuples

With a set of tuples the problem of equality is well defined and solved: Two tuples are equal if all their values are equal. We do not need to merge anything. We can just add a tuple which contains additional information. Potential conflicts can then be solved at a higher abstraction level.

3.4 Assessment of set of tuples

3.4.1 Advantages

Extensibility A set of tuples can contain various sorts of data in a standard encoding. Adding information from an information source is thus just a matter of wrapping the data in this encoding.

Composability The standard structure that all components of the system understand enables composition of the different components.

Simplicity A set of tuples is a relatively simple structure, and it is therefore easy to think and reason about.

Well founded Tuples are defined in mathematics, and operations on them are therefore well founded.

3.4.2 Disadvantages

Detached relations Relations can be between information atoms which are also stored as separate tuples. If we remove the tuples which participate in relations but not the relations itself, it could be that the format is in an inconsistent state. In a graph structure such a problem would not occur since by removing a node one would also remove all connected edges. Such a situation can be resolved on a higher abstraction level by removing all relations which contain an atom that is to be removed from the set.

Path overhead Path queries are expensive on our structure. Consider a call graph stored in tuples where each tuple contains a single call relation. For one call relation all other call relations have to be checked if they are a continuation of the path. This searching would be cheaper in a graph structure since for each node only the incident edges have to be considered.

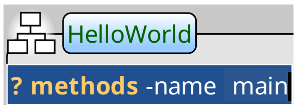
4 Queries

Queries are used to create and manipulate data in the information exchange format that we described in section 3. A query can produce data in the information exchange format: For example the query `methods` returns the methods in the current scope. A query can also be just an operation applied on the input: For example a filter operation filters out method nodes that do not have a certain name.

The user inputs a query in a special prompt in Envision which we explain in section 4.1. The extensibility of our system is discussed in section 4.2.

4.1 Query prompt

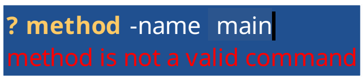
The query input enables users to execute queries in Envision. The prompt provides a visual interface that clearly distinguishes commands and their arguments, as shown in figure 4.1.



```
HelloWorld
? methods -name main
```

Figure 4.1: Query prompt with syntax highlighting on a Class element.

In case of a misspelled command the query prompt reports the problem to the user, as shown in figure 4.2.



```
? method -name main
method is not a valid command
```

Figure 4.2: Query prompt reporting a misspelled command.

The query prompt is quickly accessed through the keyboard shortcut Ctrl + Q. The prompt supports copy and paste actions. The ↵ Enter key triggers the execution of the query on the prompt and visualizes the results on the Envision canvas.

4.1.1 Context

The query prompt is always opened on a node in Envision's canvas. This node provides context to the query. The context can be used by the query to produce local results. Below we list some scenarios where context is useful:

AST node contents In the query prompt shown in figure 4.1, the context is the class `HelloWorld`. The `methods` query uses the context to search for methods in this class.

AST node relations Similar to querying content on AST nodes we can also query for relations on the context node. For example `bases` returns the base class relations of the context class and `callgraph` returns the call relations of the context method.

Tags The `tags` query can take advantage of locality: If a task is bound to a specific module or even class, it makes more sense to only search for the corresponding tags in the local scope, rather than searching globally.

Version history If a developer spots an odd pattern in a method, the history of the method can probably explain why this pattern is used. Thus the developer just wants the history of the method and no other irrelevant information.

Naturally a query is not required to make use of the context and may operate globally or on the input.

4.1.2 Composability

A single query is rarely sufficient to answer more complex questions. A combination of different queries can help in such cases. With the help of the query prompt queries can easily be composed. The user can simply type a pipe (`|`) character to connect two queries. The pipe character is visualized by the query prompt with an arrow (➡) to indicate the data flow.

Assume we want all methods which contain "visit" in the name and are in a class which contains "Node" in the name: We would first get all classes that contain "Node" in the name and from those classes extract the methods. Thus we pass the result from the classes query on to the method query as is indicated in figure 4.3.

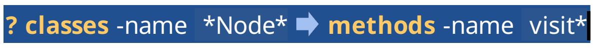
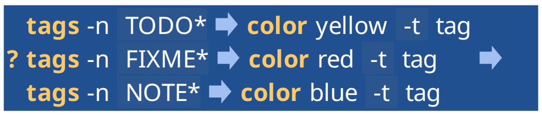

? **classes** -name *Node* ➡ **methods** -name visit*

Figure 4.3: Query prompt showing the data flow between two composed queries.

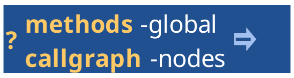
Unlike a command line, the query prompt can also pass data to multiple queries or combine data from different queries. This can be especially useful for visualizing results in different ways. For example coloring task tags, such as `TODO`, `FIXME`, and `NOTE`, can give a visual overview of a project's task state. To achieve this we first have to query for an individual tag and then color it in the color we want. If we do this for each tag we have the colors set but we still have to combine the result to get a single visualization, since the system per default only visualizes the results of a single output. In figure 4.4 we show how this can be done with the query prompt: at first we have individual query execution streams and in the end we combine the results. Similar to combining results from multiple execution streams we can also split the result of a single execution stream to multiple streams.



```
  tags -n TODO* ➡ color yellow -t tag
? tags -n FIXME* ➡ color red -t tag      ➡
  tags -n NOTE* ➡ color blue -t tag
```

Figure 4.4: Query prompt with individual execution streams which are combined in the end.

In addition to the combine operation the prompt also supports subtraction. The user can type `|-` to request a subtractive pipe. The prompt indicates the `|-` with an arrow with a minus (⇨). This is for example useful if we want to check which methods are not reachable from a certain method `x`. We list all methods and subtract the methods in the call graph of `x` as shown in figure 4.5.



```
  methods -global
?                  ⇨
  callgraph -nodes
```

Figure 4.5: Query prompt with individual execution streams which are combined with a subtractive behavior.

We offer a set of queries which aid to process the data from query composition:

`filter` Filters the input in different ways. It can keep only tuples with a specific tag, or tuples which contain a specific value. For example `filter ast` keeps all tuples with the tag `ast`. `filter color.color blue` keeps all tuples with the tag `color` and in which the value of `color` is blue.

`join` Joins two tuples with different tags based on some value which occurs in both tuples. For example `join -v change.id,commit.author,ast -as astAuthor` joins the tuples with tag `commit` and the tag `change`. The resulting tuple will have the name `astAuthor` and contains the values `id, author, ast`. The `join` query per default tries to match the `id` values of both tuples. A user can also specify a different value to join on with the `-on` argument.

`count` Counts how many times a combination of values occurs in the tuples. For example `count color.color` counts how many times each value of `color.color` appears in the input. Note that count only makes sense on multi valued tuples, since single value tuples can only appear once as they are stored in a set.

`toParent` Transforms AST values in the input to parent nodes of a certain type. For example if we have methods in the input we can get their parent classes as follows: `toParent -t Class`.

`canReach` Checks which AST relations in the input can reach a node with a certain name, and outputs all AST nodes in all paths that can reach the value. For example if we have all call relations in the input of the statement `canReach -r calls -n someFunc` returns all methods that call `someFunc` via some call path.

A more detailed explanation of all the queries in the current system can be found in appendix A.

4.2 Extensibility

While the query prompt alone is already quite powerful, there are often project specific queries that require some domain specific knowledge that cannot easily be expressed in the query prompt. To support these cases, our system can be extended easily by the user. For this we offer two mechanisms, either a developer can create a new query class in C++, or write a script in Python. In this section we will solely focus on the Python script approach. We explain the C++ API for queries in section 6.1. The scripting approach offers a lighter alternative, which is powerful enough for many use cases.

Scripts can leverage the full power of Python. With the help of an API, scripts can integrate with the query system. The `Query` module provides access to all query related information:

`Query.args` Are the textual input arguments as written in the query prompt.

`Query.input` Is the input in the information exchange format from a potential previous query.

`Query.target` Is the target node of the query, which provides the context as explained in section 4.1.1.

`Query.result` Is the variable to store the results of the script in the information exchange format.

`Query.executeQuery()` Is used to execute a query pipeline from a string. The first argument is the query string and the second argument is the data passed to the first query. Note that with this function a user can easily define an alias to a query pipeline that he uses often in form of a script.

`Query.QueryName()` Executes the query with name `QueryName` individually. The first argument is a list of textual argument and the second argument is the data passed to the query.

Additionally scripts have access to the public API of the Envision model, and can thus query and modify the data in it. In case of errors Python scripts can raise exceptions. The exception message is displayed to the user similar to the error message shown in figure 4.2. By means of two examples we present how such scripts can look like in practice.

In the first example we describe a script that can find code by using domain specific knowledge: Many C++ projects, such as Envision, use project specific versions of dynamic casts, which have less computational overhead. For such a project it makes sense to have a query in form of a script which can detect dynamic casts which can be replaced. In listing 4.1 we show a script which finds dynamic casts that can be replaced in Envision's code. As a first step the scripts runs a query pipeline to find all the classes referenced in cast expressions. This is done by first finding all the cast expressions, and from them extracting the cast type attribute. The cast type attribute is then passed to the `definitions` query to find the classes this type refers to. For each of the referenced classes we then check if they contain a `typeIdStatic` method, which is the method required in the project specific version of the dynamic cast. If it exists we can thus add the cast type attribute of the cast expression to the result. To show the actual cast we have to convert the result in the end to the first cast expression parent.

```
classUses = Query.executeQuery('ast -t=CastExpression | attribute -at=castType |
    definitions -t=Class', [])

def hasTypeIdMethod( cl ):
    for method in cl.methods:
        if method.name == 'typeIdStatic':
            return True
    return False

for definitionTuple in classUses[0].tuples('definition'):
    if hasTypeIdMethod(definitionTuple.definition):
        values = [('ast', definitionTuple.reference)]
        Query.result.add(Tuple(values))

Query.result = Query.toParent(['-t=CastExpression'], [Query.result])[0]
```

Listing 4.1: Example of a project specific script. The script is used to find dynamic casts which can be replaced with project specific casts.

The script can be saved and can then directly be called from the query prompt. For example this script is saved as `filterCasts.py` and can be called from the query prompt as `filterCasts`.

In the second example we show how a script can modify the source code: The script in listing 4.2 transforms an if statement to a while statement. We first check if the target node is really an if statement, if not we report to the problem to the user by raising an exception. If we indeed have an if statement we first have to extract the `condition` and the `then` branch from the if statement. Then we replace the `condition` and the `then` branch with empty nodes. Empty nodes can be created by calling `Node.createNode()`. The replacement is needed so that the two nodes are detached from the if statement, so that they can be added to another node. We then create the while node and attach the `condition` as `condition` and the `then` branch as loop `body`. We then have to replace the if statement with the while node.

```
if type(Query.target) is IfStatement:
    parent = Query.target.parent
    parent.beginModification('If to while')

    condition = Query.target.condition
    thenBranch = Query.target.thenBranch

    Query.target.replaceChild(condition, Node.createNewNode('Expression', None))
    Query.target.replaceChild(thenBranch, Node.createNewNode('StatementItemList',
        None))

    whileNode = Node.createNewNode('LoopStatement', None)
    whileNode.condition = condition
    whileNode.body = thenBranch

    parent.replaceChild(Query.target, whileNode)
    parent.endModification()

else:
    raise Exception('ifToWhile command only works on if statements')
```

Listing 4.2: Example of a script that transforms the target node from an if statement to a while statement.

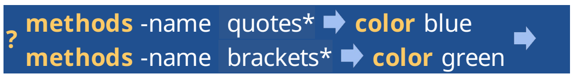
While this example is very simple, scripts could also be used to implement more complex refactoring actions.

5 Visualization

Since Envision lays code out on a 2D canvas, it offers interesting possibilities to visualize results of queries. We present three different visualization primitives we use to visualize results.

5.1 Highlights with text

Highlights are very helpful to show all the AST nodes in the results of queries. For example the simple `methods` query would highlight all methods of the target class. Since highlights can have different colors we can also use them to give an overview of different concerns in methods. For example if we look for methods that either contain `quotes` or `brackets` in the name we can use the query shown in figure 5.1 to produce the visualization that is shown in figure 5.2.



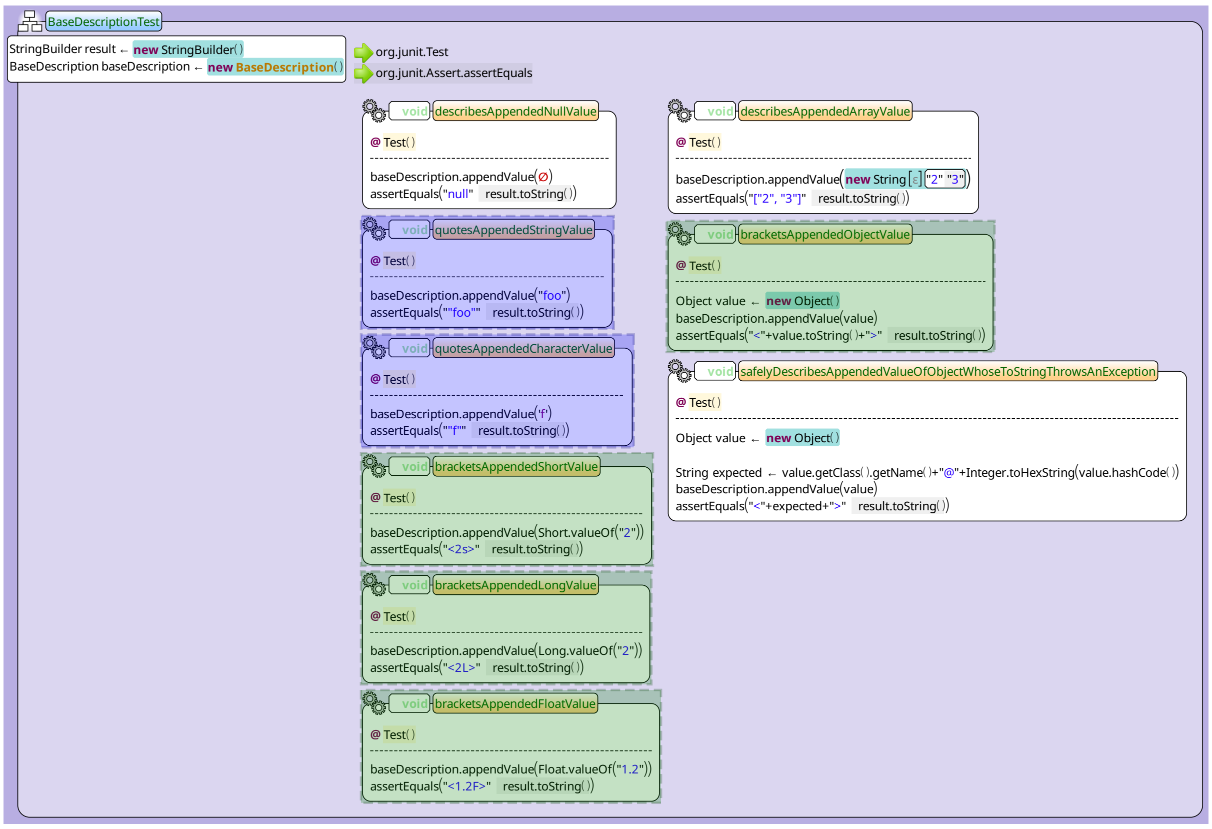
```
? methods -name quotes* ➡ color blue
  methods -name brackets* ➡ color green  ➡
```

Figure 5.1: Query to highlight methods differently depending on the name.

Often the user would like to see additional information on top of the highlighted nodes. For example the question "Who did change methods in the last commit?" requests for an author name. We can display this information directly on top of the relevant highlighted node, as shown in figure 5.3.

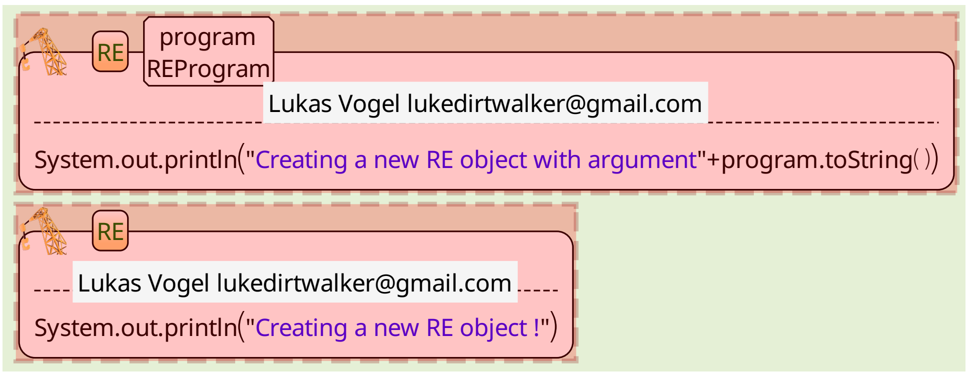
5.2 Heatmaps

A heatmap can visualize a range of numbers by a range of colors. A light green color means a low value and a red color means a high value. A heatmap can for example be used to visualize profiling data or the frequency of commits. An example of how a heatmap looks in Envision is shown in figure 5.4.



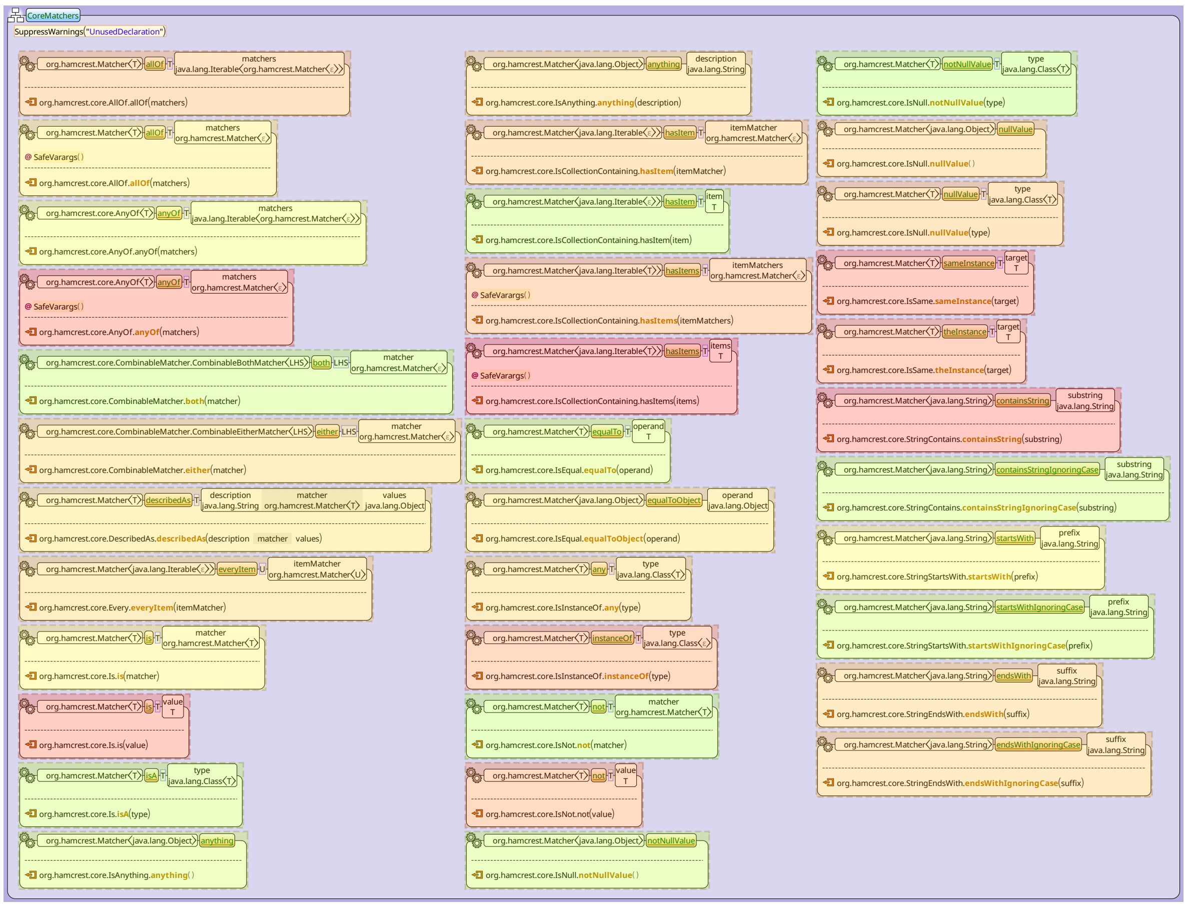
```
BaseDescriptionTest
StringBuilder result ← new StringBuilder()
BaseDescription baseDescription ← new BaseDescription()

org.junit.Test
org.junit.Assert.assertEquals

void describesAppendedNullValue
@ Test()
--------------------------------
baseDescription.appendValue(∅)
assertEquals("null"  result.toString())

void quotesAppendedStringValue
@ Test()
--------------------------------
baseDescription.appendValue("foo")
assertEquals(""foo""  result.toString())

void quotesAppendedCharacterValue
@ Test()
--------------------------------
baseDescription.appendValue('f')
assertEquals(""f""  result.toString())

void bracketsAppendedShortValue
@ Test()
--------------------------------
baseDescription.appendValue(Short.valueOf("2"))
assertEquals("<2s>"  result.toString())

void bracketsAppendedLongValue
@ Test()
--------------------------------
baseDescription.appendValue(Long.valueOf("2"))
assertEquals("<2L>"  result.toString())

void bracketsAppendedFloatValue
@ Test()
--------------------------------
baseDescription.appendValue(Float.valueOf("1.2"))
assertEquals("<1.2F>"  result.toString())

void describesAppendedArrayValue
@ Test()
--------------------------------
baseDescription.appendValue(new String[ε]"2" "3")
assertEquals("["2", "3"]"  result.toString())

void bracketsAppendedObjectValue
@ Test()
--------------------------------
Object value ← new Object()
baseDescription.appendValue(value)
assertEquals("<"+value.toString()+">"  result.toString())

void safelyDescribesAppendedValueOfObjectWhoseToStringThrowsAnException
@ Test()
--------------------------------
Object value ← new Object()

String expected ← value.getClass().getName()+"@"+Integer.toHexString(value.hashCode())
baseDescription.appendValue(value)
assertEquals("<"+expected+">"  result.toString())
```

Figure 5.2: Example of methods colored depending on the name.



```
RE program REProgram
Lukas Vogel lukedirtwalker@gmail.com
--------------------------------
System.out.println("Creating a new RE object with argument"+program.toString())

RE
---- Lukas Vogel lukedirtwalker@gmail.com ----
System.out.println("Creating a new RE object !")
```

Figure 5.3: Example of author information displayed on top of code elements.

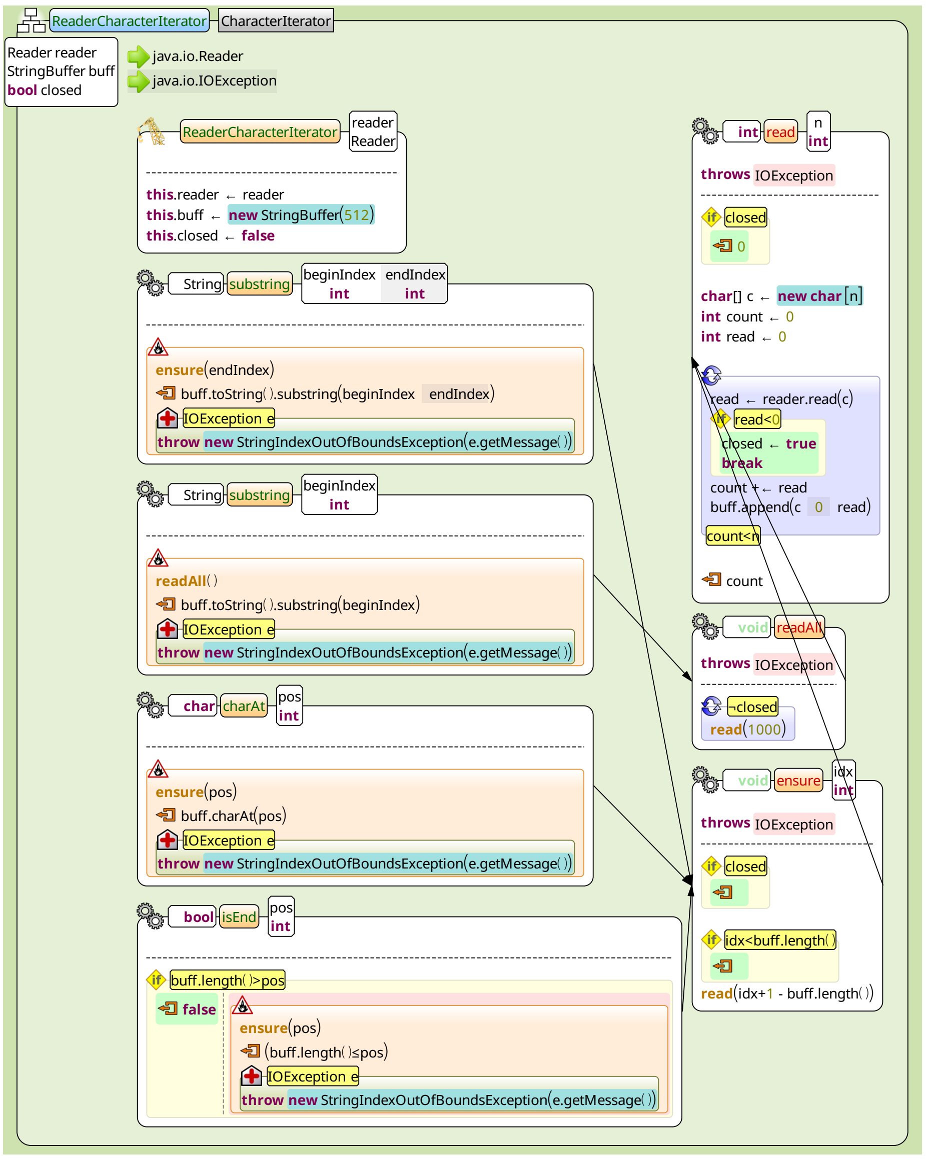


```
CoreMatchers
SuppressWarnings("UnusedDeclaration")

org.hamcrest.Matcher⟨T⟩ allOf T  matchers java.lang.Iterable⟨org.hamcrest.Matcher⟨ε⟩⟩
org.hamcrest.core.AllOf.allOf(matchers)

org.hamcrest.Matcher⟨T⟩ allOf T  matchers org.hamcrest.Matcher⟨ε⟩
@ SafeVarargs()
org.hamcrest.core.AllOf.allOf(matchers)

org.hamcrest.core.AnyOf⟨T⟩ anyOf T  matchers java.lang.Iterable⟨org.hamcrest.Matcher⟨ε⟩⟩
org.hamcrest.core.AnyOf.anyOf(matchers)

org.hamcrest.core.AnyOf⟨T⟩ anyOf T  matchers org.hamcrest.Matcher⟨ε⟩
@ SafeVarargs()
org.hamcrest.core.AnyOf.anyOf(matchers)

org.hamcrest.core.CombinableMatcher.CombinableBothMatcher⟨LHS⟩ both LHS  matcher org.hamcrest.Matcher⟨ε⟩
org.hamcrest.core.CombinableMatcher.both(matcher)

org.hamcrest.core.CombinableMatcher.CombinableEitherMatcher⟨LHS⟩ either LHS  matcher org.hamcrest.Matcher⟨ε⟩
org.hamcrest.core.CombinableMatcher.either(matcher)

org.hamcrest.Matcher⟨T⟩ describedAs T  description java.lang.String  matcher org.hamcrest.Matcher⟨T⟩  values java.lang.Object
org.hamcrest.core.DescribedAs.describedAs(description matcher values)

org.hamcrest.Matcher⟨java.lang.Iterable⟨ε⟩⟩ everyItem U  itemMatcher org.hamcrest.Matcher⟨U⟩
org.hamcrest.core.Every.everyItem(itemMatcher)

org.hamcrest.Matcher⟨T⟩ is T  matcher org.hamcrest.Matcher⟨T⟩
org.hamcrest.core.Is.is(matcher)

org.hamcrest.Matcher⟨T⟩ is T  value T
org.hamcrest.core.Is.is(value)

org.hamcrest.Matcher⟨T⟩ isA T  type java.lang.Class⟨T⟩
org.hamcrest.core.Is.isA(type)

org.hamcrest.Matcher⟨java.lang.Object⟩ anything
org.hamcrest.core.IsAnything.anything()

org.hamcrest.Matcher⟨java.lang.Object⟩ anything  description java.lang.String
org.hamcrest.core.IsAnything.anything(description)

org.hamcrest.Matcher⟨java.lang.Iterable⟨ε⟩⟩ hasItem T  itemMatcher org.hamcrest.Matcher⟨ε⟩
org.hamcrest.core.IsCollectionContaining.hasItem(itemMatcher)

org.hamcrest.Matcher⟨java.lang.Iterable⟨ε⟩⟩ hasItem T  item T
org.hamcrest.core.IsCollectionContaining.hasItem(item)

org.hamcrest.Matcher⟨java.lang.Iterable⟨T⟩⟩ hasItems T  itemMatchers org.hamcrest.Matcher⟨ε⟩
@ SafeVarargs()
org.hamcrest.core.IsCollectionContaining.hasItems(itemMatchers)

org.hamcrest.Matcher⟨java.lang.Iterable⟨T⟩⟩ hasItems T  items T
@ SafeVarargs()
org.hamcrest.core.IsCollectionContaining.hasItems(items)

org.hamcrest.Matcher⟨T⟩ equalTo T  operand T
org.hamcrest.core.IsEqual.equalTo(operand)

org.hamcrest.Matcher⟨java.lang.Object⟩ equalToObject  operand java.lang.Object
org.hamcrest.core.IsEqual.equalToObject(operand)

org.hamcrest.Matcher⟨T⟩ any T  type java.lang.Class⟨T⟩
org.hamcrest.core.IsInstanceOf.any(type)

org.hamcrest.Matcher⟨T⟩ instanceOf T  type java.lang.Class⟨ε⟩
org.hamcrest.core.IsInstanceOf.instanceOf(type)

org.hamcrest.Matcher⟨T⟩ not T  matcher org.hamcrest.Matcher⟨T⟩
org.hamcrest.core.IsNot.not(matcher)

org.hamcrest.Matcher⟨T⟩ not T  value T
org.hamcrest.core.IsNot.not(value)

org.hamcrest.Matcher⟨java.lang.Object⟩ notNullValue
org.hamcrest.core.IsNull.notNullValue()

org.hamcrest.Matcher⟨T⟩ notNullValue T  type java.lang.Class⟨T⟩
org.hamcrest.core.IsNull.notNullValue(type)

org.hamcrest.Matcher⟨java.lang.Object⟩ nullValue
org.hamcrest.core.IsNull.nullValue()

org.hamcrest.Matcher⟨T⟩ nullValue T  type java.lang.Class⟨T⟩
org.hamcrest.core.IsNull.nullValue(type)

org.hamcrest.Matcher⟨T⟩ sameInstance T  target T
org.hamcrest.core.IsSame.sameInstance(target)

org.hamcrest.Matcher⟨T⟩ theInstance T  target T
org.hamcrest.core.IsSame.theInstance(target)

org.hamcrest.Matcher⟨java.lang.String⟩ containsString  substring java.lang.String
org.hamcrest.core.StringContains.containsString(substring)

org.hamcrest.Matcher⟨java.lang.String⟩ containsStringIgnoringCase  substring java.lang.String
org.hamcrest.core.StringContains.containsStringIgnoringCase(substring)

org.hamcrest.Matcher⟨java.lang.String⟩ startsWith  prefix java.lang.String
org.hamcrest.core.StringStartsWith.startsWith(prefix)

org.hamcrest.Matcher⟨java.lang.String⟩ startsWithIgnoringCase  prefix java.lang.String
org.hamcrest.core.StringStartsWith.startsWithIgnoringCase(prefix)

org.hamcrest.Matcher⟨java.lang.String⟩ endsWith  suffix java.lang.String
org.hamcrest.core.StringEndsWith.endsWith(suffix)

org.hamcrest.Matcher⟨java.lang.String⟩ endsWithIgnoringCase  suffix java.lang.String
org.hamcrest.core.StringEndsWith.endsWithIgnoringCase(suffix)
```

Figure 5.4: Example of a heatmap visualization.

5.3 Arrows

To present relations, such as call relations we use an arrow overlay. While the arrows can show relations, one can quickly loose the overview when many arrows are shown at once. We leave the improvement of this visualization for future work. An example of the arrows are shown in figure 5.5.



```
ReaderCharacterIterator  CharacterIterator
Reader reader
StringBuffer buff
bool closed
java.io.Reader
java.io.IOException

ReaderCharacterIterator  reader Reader
----------------------------------------
this.reader ← reader
this.buff ← new StringBuffer(512)
this.closed ← false

String substring  beginIndex int  endIndex int
----------------------------------------
ensure(endIndex)
buff.toString().substring(beginIndex  endIndex)
IOException e
throw new StringIndexOutOfBoundsException(e.getMessage())

String substring  beginIndex int
----------------------------------------
readAll()
buff.toString().substring(beginIndex)
IOException e
throw new StringIndexOutOfBoundsException(e.getMessage())

char charAt  pos int
----------------------------------------
ensure(pos)
buff.charAt(pos)
IOException e
throw new StringIndexOutOfBoundsException(e.getMessage())

bool isEnd  pos int
----------------------------------------
if buff.length()>pos
    false
    ensure(pos)
    (buff.length()≤pos)
    IOException e
    throw new StringIndexOutOfBoundsException(e.getMessage())

int read  n int
throws IOException
----------------------------------------
if closed
    0
char[] c ← new char[n]
int count ← 0
int read ← 0
    read ← reader.read(c)
    if read<0
        closed ← true
        break
    count +← read
    buff.append(c  0  read)
count<n
count

void readAll
throws IOException
----------------------------------------
¬closed
    read(1000)

void ensure  idx int
throws IOException
----------------------------------------
if closed
    ←
if idx<buff.length()
    ←
read(idx+1 - buff.length())
```

Figure 5.5: Example of the callgraph visualization.

6 System design

In this chapter we present interesting details about the implementation of our framework. We outline the query framework in section 6.1. In section 6.2 we discuss our data flow query engine. To expose C++ code to the scripting environment we wrote a binding generator, it is presented together with the script bindings in section 6.3. Section 6.4 explains how we extend Envision's model to support tags. The implementation of the result visualizations is explained in section 6.5. In section 6.6 we present smaller issues of the current implementation.

6.1 Query framework

This section discusses the building blocks for the query support.

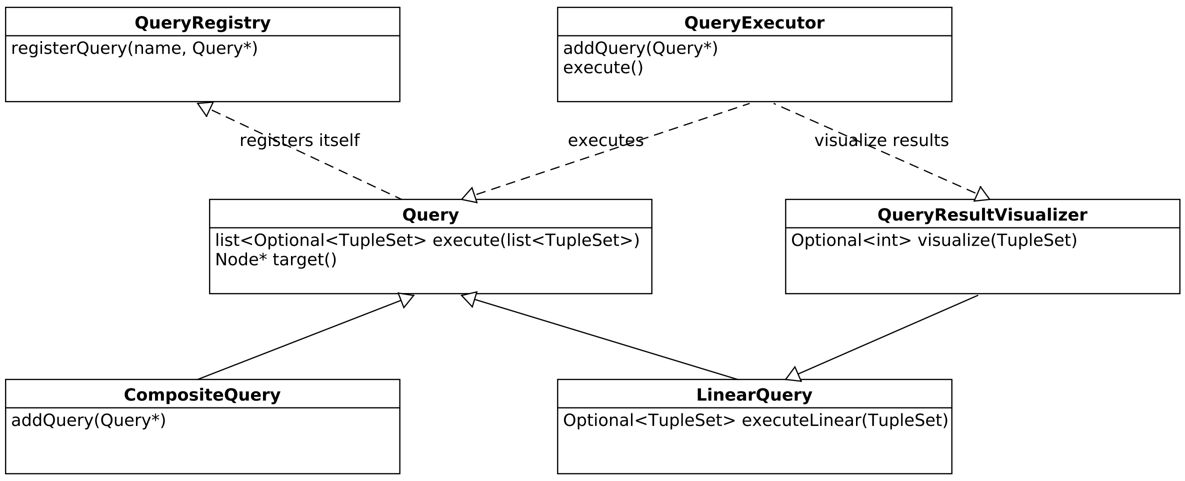


Figure 6.1: Overview of the most relevant classes and functionality in the query framework.

6.1.1 Query and LinearQuery classes

The `Query` class is the base class of all queries. It specifies that each query has to have the `execute` method. This method takes a list of `TupleSet` as input and returns a list of optional `TupleSets`. The `Optional` return value is to indicate any errors that might have happened during execution. Since most queries, apart from operators, work on a single input and produce a single output we offer a common abstraction for those queries in form of the abstract `LinearQuery` class. The `LinearQuery` class requires implementors to implement the `executeLinear` method. The default implementation of the `execute` method in the `LinearQuery` class just applies the `executeLinear` function for each input.

To ease the implementation of queries we offer the `ArgumentParser` and `ArgumentRule` classes. They offer functionality to parse arguments of a query and to create rules that specify which arguments have to be passed. We explain how they can be used in section 6.1.4.

6.1.2 QueryRegistry

To make the query functionality available through the query prompt we need a registry of all the available queries in the system. A query can register all queries it provides in the `QueryRegistry` class with the `registerQuery` function. When parsing the user input we use this registry to build the appropriate queries. If the user typed a query name which is not registered we check if there is a corresponding script with this name and if so we execute this as part of a `ScriptQuery`. This makes it possible to add scripts with minimal overhead, adding a file in the `scripts` directory is enough.

6.1.3 QueryExecutor

The `QueryExecutor` manages the execution of queries. In most cases it will execute a single Query, which might contain multiple subqueries in case of a `CompositeQuery` (c.f. section 6.2). However in some cases the user might have to wait on a result before it makes sense to continue the execution of the query pipeline. For example if one query starts a program and the second query requires some parts of the result for the execution. To make this use case possible we offer the `yield` keyword which splits the query pipeline into two parts. We store the parts of the query pipeline in the `QueryExecutor` as a queue. A query before the `yield` command has to make sure that the `execute` method of the `QueryExecutor` is called whenever the long running action is finished to continue the execution of the query pipeline. In the case of running a program a callback that is executed on program termination, can do this call.

6.1.4 Adding a new C++ Query

Adding a new C++ query is quite simple in our system. Adding a single class and registering it is enough. We explain the most important points to consider with the help of the example query in listing 6.1.

Our example uses the `LinearQuery` as a base class. The public interface of a query is often only the `execute` or in the linear case the `executeLinear` method together with the `registerDefaultQueries` function. The `executeLinear` function is called when the query should be executed. The `registerDefaultQueries` function should register the query to the `QueryRegistry`. The developer has to make sure that this function is called at the initialization of the plug-in in which the query resides. The `registerQuery` function takes the name of the query as a first argument and all other argument are passed to the constructor of the query class.

The default signature of the constructor should always have a `Node` pointer as first argument and a `QStringList` for the arguments as second argument. The pointer is the target of the query an can just be passed on to the parent constructor. The `QueryRegistry` will always prepend the query name to the argument list. Because of this and the forwarded arguments, queries should always be constructed through the `QueryRegistry`. Thus a query constructor is usually private and the `QueryRegistry` is marked as a friend class.

If the query wants to add command line arguments it can specify them at initialization of the `arguments_` field. The names of the arguments are stored in string lists as shown in lines 23 and 24. The `ArgumentParser` class takes a list of `QCommandLineOption` and the arguments list itself. The options can be declared to expect a value like in line 30, or as a flag without value like in line 31. To check at construction time that arguments which are required are present, we can create a list of `ArgumentRules`. This list can be declared at the registry of the query (line 17) and is then forwarded to the constructor where the rules can be checked since the arguments are parsed. In our example we want that either a color argument is given of the blue flag is specified. Thus we describe the rule as shown on line 17 and 18.

```cpp
class Color : public LinearQuery
{
    public:
        virtual Optional<TupleSet> executeLinear(TupleSet input) override
        {
            // check if an argument is set
            if (arguments_.isArgumentSet(BLUE_ARGUMENT[1]))
                ...
            // get an argument
            QString colorValue = arguments_.argument(COLOR_ARGUMENT[1]);
            // color the tuples
            ...
        }

        static void registerDefaultQueries() {
            QueryRegistry::registerQuery<MyQuery>("queryName",
                std::vector<ArgumentRule>{{ArgumentRule::RequireOneOf,
                    {{COLOR_ARGUMENT[1]}, {BLUE_ARGUMENT[1], ArgumentValue::IsSet}}}});
        }

    private:
        friend class QueryRegistry;

        static const QStringList COLOR_ARGUMENT{"c", "color"};
        static const QStringList BLUE_ARGUMENT{"b", "blue"};

        ArgumentParser arguments_;

        MyQuery(Model::Node* target, QStringList args, std::vector<ArgumentRule> rules)
        : LinearQuery{target}, arguments_{{
            {COLOR_ARGUMENT, "Color argument description", COLOR_ARGUMENT[1], "red"},
            QCommandLineOption{BLUE_ARGUMENT}
            }, args}
        {
            for (const auto& rule : argumentRules)
                rule.check(arguments_);
        }
};
```

Listing 6.1: Example of a simple Query class in C++

6.2 Query dataflow engine

To be able to manage the dataflow between different queries we introduced a `CompositeQuery` class. This class allows us to flexibly define how queries are interconnected. It provides functionality to map output i of a query to input j of another query. The `CompositeQuery` then assures that a query only executes when all its inputs have been calculated. This is achieved by applying topological sort on all the queries a `CompositeQuery` contains. In essence the `CompositeQuery` implements a standard dataflow execution engine.

6.3 Script bindings

To make the C++ parts of the Envision model accessible to the python scripting environment we decided to use the boost python library[1]. Several properties lead us to this decision:

Concise: To expose C++ symbols to the python environment one can write relatively short wrapping code with the help of boost python. For example to expose a class symbol `Node` the following code would be sufficient: `class_<Node>("Node");`.

Non intrusive: The example shows that this library is a layer on top of an existing system, which does not require any changes to the existing C++ code.

Future proof: Since boost is the the most widely used library collection for C++, apart from the standard library, we can safely assume that this library will stay well maintained in the future.

In section 6.3.1 we outline some aspects of the boost python library. This basic knowledge is required to discuss the code generation facility for the wrapping code of the Envision model in section 6.3.2. In section 6.3.3 we explain how python scripts are embedded into the main IDE process during runtime.

6.3.1 Boost Python concepts

In this section we present a short overview of concepts of Boost Python which are relevant for our implementation of the Python bindings. The reader should have basic knowledge about the library, good introductions[2] and examples[3] can be found online.

Value and pointer types

Boost Python makes it possible to have a minimal overhead when passing data structures to the Python environment. This is achieved by thin wrappers around pointers of existing objects. The wrapping code for the exposed types enables a Python script to access C++ objects just like they were Python objects. The most common use case for the scripts in our use case, will be to access the Envision model, thus this possibility is very helpful for us. Furthermore this enables us to deal with memory management only on the C++ side, and avoids many potential problems related to it. This is since we know that the object we are accessing, i.e. parts of the Envision model, always lives longer than the Python environment. To expose an object via a pointer to Python the `ptr` function is used. When writing wrapping code for methods which

[1]http://www.boost.org/doc/libs/1_59_0/libs/python/doc
[2]https://www.youtube.com/watch?v=GE8EsGUsC2w
[3]https://github.com/TNG/boost-python-examples

return a value or a pointer the correct return value policy has to be selected. In our case the correct policy for pointers to `Nodes` is `return_internal_reference`[4] since we want to prevent Python from managing the memory of the Envision model.

To deal with value objects like strings, lists, and sets object converters have to be written.

Python properties

Envision's model is based heavily on attributes, special fields which always have setter and getter methods. Fortunately the Python language can model this concept in a nice way with properties[5]. To wrap attributes of Envision's model we can thus use the `add_property` method of the Boost Python library.

6.3.2 Wrapping code generation

Envision's model consists of hundreds of classes, which might change over time. To make sure that we can keep the python binding code up to date without much effort we wrote a tool that generates the wrapping code of those classes. The implementation is based on the Clang tooling infrastructure[6]. The choice of the Clang interface was motivated by the fact that we already use it for importing C++ code into Envision, so we could reuse the knowledge gained in this domain. A custom tailored tool was suitable for this use case, since we can use domain specific knowledge about the Envision model. Figure 6.2 outlines the interactions between the most important classes of this tool. Each class is individually discussed in respective subsections.

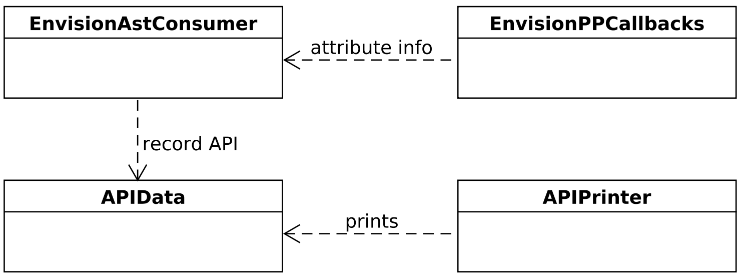


Figure 6.2: Overview of the interactions in the python wrapper generator tool

The tool can be configured through the `config.json` file in the tool's directory. The Envision path should point to a directory that contains the source code of Envision. In that directory there should also be a `compile_commands.json` file. This file can be generated by building Envision with CMake.

[4] http://www.boost.org/doc/libs/1_59_0/libs/python/doc/v2/return_internal_reference.html
[5] https://docs.python.org/3.5/library/functions.html#property
[6] http://clang.llvm.org/docs/LibTooling.html

APIData

The `APIData` class is a container class, which stores all the relevant data for the wrapping code. It contains information about classes and their methods and attributes, enums, etc. To generate correct wrapping code, the class hierarchy has to be respected, i.e. base classes have to appear before their child classes. To respect this ordering we create a tree to store the hierarchy graph. A breadth first traversal of this tree guarantees a correct ordering of the produced wrapping code.

EnvisionPPCallbacks

To export the attributes of Envision's model as Python properties we register usages of `ATTRIBUTE` macros with the help of the `EnvisionPPCallbacks` class. This data is then used by the `EnvisionAstConsumer` to detect which methods should be wrapped as part of a python property and which methods should be standalone.

EnvisionAstConsumer

The `EnvisionAstConsumer` class records all the class and method information. It uses the information from the Clang AST to generate the correct wrapping code.

During a class traversal, all methods are traversed twice. In the first run we identify all methods which belong to attributes as found by the `EnvisionPPCallbacks` class. For methods which do not belong to an `ATTRIBUTE` macro it could still be that they model an attribute. For example the class `Class` in Envision's model has the methods `constructKind()` and `setConstructKind()`, which model the construct kind attribute. We identify all such methods in the first run in a possible attribute set. Using this information in the second run we can decide if a method should be wrapped as a standalone method or as an attribute.

Since all of the Model should be owned by Envision and never by the Python environment we can select the return value policy of methods based on whether a method returns a value type or not. In case of a pointer or reference type return value, the policy should always be `return_internal_reference`.

6.3.3 Embedding the Python interpreter

To run Python scripts as part of a query execution, we implemented a special `ScriptQuery` class. This class adheres to the `Query` interface. Once an object of this class is executed, the respective python script is loaded and executed. To reduce the typing overhead of a user we inject symbols of the Envision model into the Python name space before the execution. Additionally we expose the query input as `Query.input`. At the end we collect the results from the `Query.result` field and return them. Note that the Python interpreter is only started once, at the initialization of the information scripting plug-in, this is mostly to protect from possible issues we could have when running it multiple times[7].

[7] https://docs.python.org/3.5/c-api/init.html

6.4 Tags

Since Envision's model previously had no notion of tags we had to add them. Since tags should persist and be shared between team members we decided to add them to the Envision model. With the extension mechanism of Envision's model we can make use both of the persistence and version control capabilities. We use a simple string as the format of tags. With this format users can have their own preferred structure.

6.5 Visualizations

The visualization of the results is handled with the `QueryResultVisualizer` class. This class adheres to the query interface and provides the queries `show` and `info`. The `show` query shows all tuples with `ast` tag in red or in the color specified by `color` tuples. Per default the `QueryExecutor` executes the `show` query at the end of the execution. Since each execution of the `QueryResultVisualizer` removes previous visualizations we have to make sure that it is run only once per query pipeline. Thus if the user explicitly runs `show` or `info` as part of the query pipeline, the `QueryResultVisualizer` will notify the `QueryExecutor` about this.

To visualize the data we use the existing overlay framework of Envision. We introduced the `HighlightOverlay` class, which is an overlay that can both highlight a region of code in any color, and print some text on top of the highlight. To visualize relations we use the existing `ArrowOverlay`.

The creation of heatmaps is done in a separate `Heatmap` class. The class is just an independent query that translates `count` tuples to `color` tuples. This is done by mapping the range of different count input values to an rgb range. The color tuples are then used by the `QueryResultVisualizer` to highlight sections of code in the given color.

6.6 Known issues

This section presents various smaller implementation issues of the current system:

- We currently lack a query which can consider modifiers of declarations. Such a query could be useful to answer many questions.
- In the current implementation of the query prompt it is not possible to escape quotes. That means quoted arguments that contain both spaces and quotes are not correctly parsed at the moment.
- Python scripts can only create tuples that contain strings, numbers or AST nodes. To add support for more data types the function `DataApi::convertTuple` would have to be adapted.
- The arrow visualization is very crude. Support for bent arrows to make it possible to not let arrows overlap important information.

- The visualization cannot handle multiple colors for a single node. If two different color tuples are specified for an AST node only the first color that is found will be shown.

7 Related work

7.1 Questions developers ask and tools to answer them

Many researchers conducted user studies to identify questions which are difficult to answer in a software ecosystem. Work in this area should be used to guide the design of systems like ours.

Latoza and Myers [12] surveyed 179 developers about questions which they recently had to answer. From the 371 reported questions they built a valuable questions catalog with 21 categories across three topics: changes, properties of elements, and relations between elements. While some of those questions depend on personal or team guidelines, e.g. “Should I branch or code against the main branch?”, others question like “By whom was this code changed?” we can answer with our tool (see section 8).

Fritz and Murphy [4] surveyed a group of 11 developers about questions, that require a combination of information from different tools to answer. They present a list of 78 questions and state over which information domains each question ranges. A large amount of the questions the authors identified, they can answer with the help of a tool they developed as part of their work. To compose information from different domains they present an information fragment model, which is a graph based structure. They focus on automatically matching data from different domains with the help of identifiers. Through projections they can vary the presentation of a result. Unlike our approach, a user of their tool has to rely on the given fragments and cannot manually refine results with the help of simple scripts. Furthermore we have the possibility to work on finer grained source code elements since we work directly with an AST.

Sillito et al. [17] analyzed questions developers have during change tasks. With two user studies the authors identify a set of 44 questions. While the first study focused more on *what* developers need to know during change task, the second study also focused on *how* professionals discover the required information. The authors observed that existing tools do not provide sufficient means to answer the questions that were raised.

Alwis and Murphy [3] present a model to compose similar information from different information spheres. A sphere can be thought of as a perspective on a program. For example static information, such as call relations, form the static sphere. To find out which calls in a call graph are executed a developer can thus combine information from the static and dynamic sphere. In their information exchange format they explicitly distinguish between information elements and relations between those elements. This helps them to apply different composing strategies for relations. With our approach we can define composing strategies per relation or even elements with the help of scripts. In terms of visualization we could make use of the visual elements of Envision, while Alwis et al. used simple tree views.

Schiller and Lucia [16] present Cupid, a plug-in ecosystem which enables users to combine different data driven plug-ins with each other. They formalize the underlying architecture using a polymorphic lambda calculus. With the help of this formalism and typing information they prevent nonsensical plug-in combinations. While this might be helpful to guide the user in some cases, we believe that it is in general too restrictive. In our approach it could be that two independent queries are connected, even though their type would not match. The rationale behind this is that both queries can independently add data, which could be used in a third query in a useful way.

Würsch et al. [23] designed a pyramid of ontologies called Seon. Seon is designed with the goal to study software evolution and thus combines information from version control systems and the source code. Iqbal et al. [8] also use an ontology model to combine information from various domains. The problem with ontology approaches is that the model is often precomputed and thus offers little flexibility in what it contains. To update the precomputed model once the underlying information sources change is also not easy to do efficiently. Another open problem is how to change the underlying information through this structure. With our approach we compute the data lazily, only when it is requested. While that in theory could have a negative impact on performance it removes the problem of outdated information and comes at a lower memory overhead. Additionally we do not have to impose any fixed structure on the data.

To write queries targeting an ontology based system is often cumbersome for an end user. Both Würsch et al. [24] and Jiang et al. [9] present interfaces which translate natural language queries into the underlying query language. While those approaches are very user friendly, the underlying system still has the ontology related problems we describe above. We believe that a natural language interface could be added to our system as well.

7.2 Tags

Storey et al. [20] observe that navigation is a key activity in software development. To ease navigation they propose the concept of tags. A tag is attached to a source code location and stores additional relevant information for this source location. To simplify a recurring software task a team can tag all the involved source code locations. Once a user wants to do such a task, he can simply navigate through the tagged locations. In a user study [19] of their prototype users in general found tags useful, but most of them kept tags short because they clutter the source code. Our approach to tags does not suffer from this limitation because per default tags are not shown. In this way the user can focus on the relevant information and show and hide tags depending on the task.

7.3 Visualization of information

Latoza and Myers [11] found that reachability plays a role in more than 9 questions developers ask during a day. A large part of those questions were reported to be difficult to answer. The authors thus built the tool Reacher [10] that supports reachability questions. Reacher can visualize call graphs and aids developers to search along callgraphs. In a user study they found that their tool is very helpful to answer control flow questions. Our tool provides a primitive visualization of call graphs. Since we focused more on the combination of information, than on this specific use case our visualization should still be improved in future work. We believe that by rearranging the method nodes in our approach the output could be vastly improved.

Team and version control history data integration has been investigated by various researchers. Biehl et al. [2] designed FASTDash a dashboard visualization for team artifacts. Guzzi et al. [6] studied how historical information can be presented at code locations. Both projects show that there is a need for team and version control coordination in IDEs. While we support the extraction of author information we currently do not offer any team information. Furthermore we can still improve on how the interaction with author information works. For example it could be useful to be able to directly contact an author of a code region of interest, like Guzzi et al. propose.

7.4 Refactoring with domain specific languages

Verbaere et al. [22] found that many refactoring actions in IDEs create wrong results. They designed a custom scripting language, called JunGL, which helps to build refactoring actions. JunGL is a hybrid of a functional and a logic query language. They argue that such a language can optimally support dataflow properties. The authors show that they can solve various refactoring actions correctly with their proposed solution. Our approach also offers scripting capabilities for changing the source code. However, how to optimally support complex refactorings in our system remains to be investigated in the future.

8 Evaluation

8.1 Comparison with related work

In this section we validate our prototype by answering questions other researchers investigated and tried to answer with similar tools. We show that we can answer all questions, that pertain to information sources we currently support, and with the help of scripts we provide flexibility in how exactly a question is interpreted.

8.1.1 Information Fragments

Fritz and Murphy [4] interviewed 11 professional developers and identified 78 questions, which require information from various sources to answer: source code, change sets, teams, work items, comments, web, stack traces, and test cases. Since we only support version control and source code information sources we only look at the questions 14 to 27 as listed in their work. Some of the questions also involve team information but they can also be interpreted as addressed to a specific person. The text in square brackets (`[]`) in a question denotes a clarification by the authors to a question stated explicitly by a study participant.

Question 14: **What has changed between two builds [and] who has changed it?**

Usually builds of a software are indicated with a tag in the version control system. Thus we can extract the changes between the two build tags. Since the changes query returns commit meta information and the changed nodes in separate tuples we have to join them to get author information on the AST nodes. We can then show the authors on top of all the changed AST nodes. The query is shown is listing 8.1.

```
changes -in build1..build2 | join -v change.id,commit.author,ast -as=astAuthor | info
    astAuthor.author
```

Listing 8.1: Query to find changes and authors thereof between two builds.

If a user often has to associate author information with AST nodes it could make sense to define an alias to the join query in form of a script as we describe in section 4.2. The alias script is shown in listing 8.2.

```
Query.result = Query.executeQuery('join -v change.id,commit.author,ast
    -as=astAuthor', Query.input)[0]
```

Listing 8.2: Script which aliases the join query on author information.

If we save this script as `developer.py` we can simplify the query in listing 8.1 to the one shown in listing 8.3.

```
    changes -in build1..build2 | developer | info astAuthor.author
```

Listing 8.3: Simplified query from listing 8.1 by using the alias script from listing 8.2

Question 15: **Who has made changes to my classes?**

First we have to define what ownership of a class means. We could define it as the class that a person has changed most. We could also define it just as a list of classes the user defines. Once we have the list of classes we can pass it to the changes query and get the list of authors, as shown in listing 8.4.

```
changes -c all | developer | count astAuthor.ast,astAuthor.author | info
    count.author,count.count -sort count
```

Listing 8.4: Query to find the most frequent authors of classes provided by the user.

The `changes` query extracts the change information for all the classes in the input. We then get the author information with the `developer` query. To provide the amount of changes each person did we use the `count` query to count the changes per person. We then show this data sorted on top of the AST nodes with the `info` query.

Question 16: **Who is using that API [that I am about to change]?**

For this question we first have to extract the API from a certain code location. This can be done with a simple script as shown in listing 8.5. Since we define what belongs to the API in a script we can flexibly adapt this definition according to user needs.

```
def addToResultIfPublic(ast):
    if ast.modifiers.isSet(Modifier.ModifierFlag.Public):
        Query.result.add(Tuple([('ast', ast)]))

if type(Query.target) is Class:
    tClass = Query.target
    for method in tClass.methods:
        addToResultIfPublic(method)
    for field in tClass.fields:
        addToResultIfPublic(field)

else:
    raise Exception('extractAPI only works on classes')
```

Listing 8.5: Example script to extract the API of a class.

Once we have all the definitions from the API we associate the definitions with their usages through the `usages` query. We then extract all usages into `ast` tuples and get version control information for those nodes. From the version control information we get recent authors of the usages. The `info` query will highlight all usages of the API and list developers that recently modified the usages, who are likely the users of the API. The complete query is shown in listing 8.6.

```
extractAPI | usages | filter -extract usage.use -as ast | changes | developer | info
    astAuthor.author
```

Listing 8.6: Query to find developers who use an API.

Question 17: **Who created the API [that I am about to change]?**

Similar to question 16 we first have to define what the API is, here we just reuse the same definition given by the script in listing 8.5. We then only have to extract author information of those AST nodes. The query to do this is shown in listing 8.7.

```
extractAPI | changes -c all | developer | info astAuthor.author
```

Listing 8.7: Query to find all authors of an API.

Question 18: **Who owns this piece of code? / Who modified it the latest?**

To answer this question we can use a local query and directly open the query prompt on the code region of interest. We first get the latest commit for this code region and then extract the author information and display it. The query is shown in listing 8.8.

```
changes -local -c 1 | developer | info astAuthor.author
```

Listing 8.8: Query to find the latest author of a piece of code.

Question 19: **Who owns this piece of code? / Who modified it most?**

Similar to question 18, we can use a local query, but instead of looking at a single commit, all commits have to be considered. Since we want to know who modified the code most, we also have to count how many time a developer changed the code and then print a sorted list of all developers. The complete query is shown in listing 8.9.

```
changes -local -c all | developer | count astAuthor.ast,astAuthor.author | info
    count.author,count.count -sort count
```

Listing 8.9: Query to list the most frequent authors of the target code.

Question 20: **Who to talk to if you have to work with packages you haven't worked with?**

If a developer has to work on code, which he has no knowledge about, he would likely want to contact a person who has a lot of experience in this code. So a good way to answer this question is just to list the most frequent authors of the code in a sorted manner. Thus this question reduces to question 19 and can be answered with the query in listing 8.9.

Question 21: **What classes have been changed?**

This question does not specify the timespan the developer is interested in. But since that is an argument to our `changes` query the user is quite flexible with this. For the timespan of 5 commits the query is shown in listing 8.10.

```
changes -c 5 -t Class -nodes
```

Listing 8.10: Query to find the classes which changed in the last 5 commits.

Question 22: **[Which] API has changed (to see which methods are not supported any more)?**

For this question we first have to define the API, similar to how we define the API in question 16 with the script shown in listing 8.5. The AST nodes of the API we can then pass as input to the `changes` query. This returns all nodes from the input that were modified in the last 5 commits. The complete query is shown in listing 8.11.

```
extractAPI | changes -c 5 -nodes
```

Listing 8.11: Query to find how an API changed in the last 5 commits.

***Question 23:* What's the most popular class? [Which class has been changed most?]**

To answer this question we first get all change informations at class level. In a second step we count how many commits change each class. Then we present the results as a heatmap. This presentation will not only answer the initial query, but it also gives an overview on how frequent other classes were changed. The query is shown in listing 8.12.

```
changes -c all -t Class | count change.ast | heatmap
```

Listing 8.12: Query to produce a heatmap from the frequency of changes in classes.

***Question 24:* Which other code that I worked on uses this code pattern / utility function?**

This question consists of two parts, first we have to find other code that the developer has worked on, and then we have to filter out the code which uses a certain pattern or utility function. For the first part we get the changes at method resolution, then extract author information and filter this by author name and then we extract all AST nodes from the result.

```
changes -c all -t Method | developer | filter astAuthor.author "Bob Developer*" |
    filter -extract astAuthor.ast
```

Listing 8.13: Query to find all code locations that were changed by "Bob Developer".

The result of the query in listing 8.13 we can then pass to a script which filters out methods which do not use the requested pattern or utility function. The complexity of such a script depends on what exactly the developer wants to check for. For example checking if it uses a certain method can be done relatively simply but looking for a specific pattern could be more involved.

***Question 25:* Which code has recently changed that is related to me?**

First we have to define what "related to me" means. It could mean code that the developer has ever changed but it could also have a different meaning. We focus on the first interpretation, other interpretations can be implemented with the help of scripts. The query to answer this question is shown in listing 8.14.

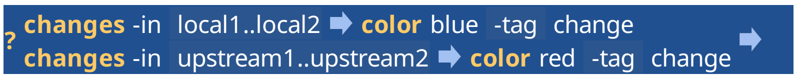
```
changes -c all | developer | filter astAuthor.author "Lukas Vogel*" | filter -extract
    astAuthor.ast | changes -nodes
```

Listing 8.14: Query to find recent changes in all code that I ever changed.

To find code that the developer has ever changed we can first query all change information and then attach the author information to the code. We can then filter by the author to only get changes that the developer did. From the changes that the developer authored we then extract the AST nodes and pass them to the `changes` query to find recent changes.

Question 26: **How do recently delivered changes affect changes that I am working on?**

To help answering this question we suggest to highlight the changes that were applied upstream in a different color to changes that were applied on the local branch. The query to do this is shown in figure 8.1.



```
? changes -in local1..local2 ➡ color blue -tag change
  changes -in upstream1..upstream2 ➡ color red -tag change   ➡
```

Figure 8.1: Query to highlight changes differently.

Question 27: **What code is related to a change?**

This question asks, what code was changed in a single commit. Thus we specify that we want all nodes that changed in this one commit. Listing 8.15 shows the query for this question.

```
changes -in a12dbf -nodes
```

Listing 8.15: Query to find all affected nodes in a commit.

8.1.2 CodeQuest

Hajiyev et al. [7] developed a source code querying tool called CodeQuest. The queries in CodeQuest are written in Datalog, a logic language. The authors present queries their tool can answer from various domains:

General queries, which are of general interest in software.

Project specific queries, which check for properties that are specific to a project.

Program understanding queries, which mimic the use case of a developer getting to know a project's source code.

Refactoring queries, which should help to find candidates for refactoring.

In the list below we use the domain as prefix to the queries. We did not include the program understanding queries, since they are all relatively simple and do not add any additional value to our evaluation.

General 1: Find all non final public fields

Similar to the datalog query we first get all fields and then filter the output. However we cannot directly filter for the public and final modifiers from the query prompt but we have to write a small script for this purpose. The query is shown in listing 8.16 and the `nonFinalPublicFilter` script is shown in listing 8.17. We think that here our system shows room for improvement in the future: There should be a specific query that can check for modifiers.

```
ast -global -t Field | nonFinalPublicFilter
```

Listing 8.16: Query to find all non final public fields

```
for astTuple in Query.input.tuples('ast'):
   if type(astTuple.ast) is Field:
      modifiers = astTuple.ast.modifiers
      nonFinalPublic = modifiers.isSet(Modifier.ModifierFlag.Public) and not
         modifiers.isSet(Modifier.ModifierFlag.Final)
      if not nonFinalPublic:
         Query.input.remove(astTuple)

Query.result = Query.input
```

Listing 8.17: Script to filter out all fields which are public and non final.

General 2: Find all methods M that write a field whose type is a subtype of T

In our system we approach this question bottom up. That means we first get all assignment expressions. Then we filter those with a script, which is shown in listing 8.19. The script discards all assignments which are not to a field of subtype `T`. Then we get the method parent of those assignments, which is the answer of the question. The query is shown in listing 8.18.

```
ast -t AssignmentExpression -global | writesFieldOfSubtype T | toParent -t Method
```

Listing 8.18: Query to find all methods that write a field of subtype of `T`.

```python
for astTuple in Query.input.tuples('ast'):
    leftExpr = astTuple.ast.left
    if type(leftExpr) is ReferenceExpression and type(leftExpr.target()) is Field:
        field = leftExpr.target()
        if type(field.typeExpression) is ReferenceExpression:
            typeTarget = field.typeExpression.target()
            if type(typeTarget) is Class:
                if typeTarget.name == Query.args[0]:
                    Query.result.add(astTuple)
                else:
                    for aClass in typeTarget.allBaseClasses():
                        if aClass.name == Query.args[0]:
                            Query.result.add(astTuple)
```

Listing 8.19: Script to filter out all assignment expressions that write a field of a subtype of the first argument.

General 3: Find all implementations M2 of an abstract method M1

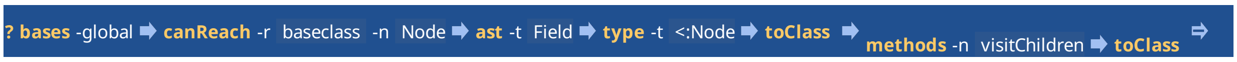
If we want to know this for a specific method, we open the query prompt on this method and run a script. The script can then run a query to get all methods in the system and for each method check if it overrides the target method. The script is shown in listing 8.20.

```python
if type(Query.target) is Method:
    allMethods = Query.methods(['-global'], [])[0]
    for astTuple in allMethods.tuples('ast'):
        if astTuple.ast.overrides(Query.target):
            Query.result.add(astTuple)
else:
    raise Exception('Finding overrides of a methods only works on a method.')
```

Listing 8.20: Script to find all methods that override the target method.

Project specific 1: Find every class that implements the `Node` interface that has a field which is of subtype `Node` and does not implement a `visitChildren` method

To answer this question we first have to find all classes that implement the `Node` interface, for this we check if the base class relation can reach the node interface. We then extract all fields of those classes and filter out fields which have a subtype of `Node`. Since we want to check the methods of those classes, we first have to get the containing classes again. We then subtract all classes which have a `visitChildren` method from the result, which gives us the final result.

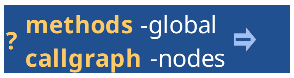


```
? bases -global ➡ canReach -r baseclass -n Node ➡ ast -t Field ➡ type -t <:Node ➡ toClass ➡ methods -n visitChildren ➡ toClass ⇨
```

Figure 8.2: Query to find all classes that implement to `Node` interface and have a field that is of subtype `Node` and does not implement a `visitChildren` method.

***Project specific 2:* Find all methods that are not called from main**

To answer this question we first get all methods of the project. Then we subtract the methods which are in the callgraph of the `main` method. Note that here it is important that the user executes the query shown in figure 8.3 on the main method. The same query could be applied to any other method, by simply changing the location where the query prompt is opened.



```
? methods -global
  callgraph -nodes ⇨
```

Figure 8.3: Query to find all methods that are not called from the target method.

***Refactoring 1:* Find all classes that contain `Combined` in the name and have a method name `getSubplots`**

This query can be stated relatively simply: We first get all the classes which contain `Combined` in the name. From those classes we extract the methods with name `getSubplots` and then get the parent classes of those methods again.

```
classes -global -name *Combined* | methods -n getSubplots | toClass
```

Listing 8.21: Query to find all classes that contain `Combined` in the name and have a method called `getSubplots`

***Refactoring 2:* Find method signatures that are common between a set of classes**

Since we only have a limited set of classes to check we can use a simple approach: For each method that exists in one class we check if all other classes have this method as well. We can do this by passing the classes to check to the script in listing 8.22.

```
astTuples = Query.input.tuples('ast')
firstClass = astTuples.pop().ast

for m in firstClass.methods:
    for otherClass in astTuples:
        for otherM in otherClass.ast.methods:
            if m.name == otherM.name:
                Query.result.add(Tuple([('ast', m)]))
                Query.result.add(Tuple([('ast', otherM)]))
```

Listing 8.22: Script to find common methods among the classes in the input.

8.1.3 Wiggle queries

Urma and Mycroft [21] designed Wiggle, a tool which supports source code queries with the help of a graph database. The authors list 4 source code queries which their tool is able to answer.

All information required for the queries they list, so called overlays, had to be precomputed in Wiggle. For example the type hierarchy is a precomputed overlay which helps to answer the first query. Since Wiggle only works on source code the queries only work on AST information.

Find all classes that are directly or indirectly subtypes of `java.lang.Exception`

To answer this query we first have to query for the base class relation of all classes. We can do this by using the `bases` query globally. This returns a set of relations, which form base class paths. If we now check all base class paths if they contain `java.lang.Exception` we know which classes are sub types of `java.lang.Exception`. The full query is shown in listing 8.23.

```
bases -global | canReach -r baseclass -n java.lang.Exception
```

Listing 8.23: Query to find all classes which are a subtype of `java.lang.Exception`

Find classes containing recursive methods

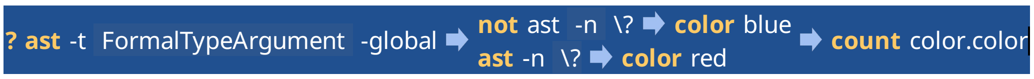
This query is very similar to the query to find base classes above. We first want to get the call relations of all methods and then find all paths which can reach themselves. This will return all methods that call themselves, which means they are recursive. By getting the respective parent classes we have all classes that contain recursive methods.

```
callgraph -global | canReach -self -r calls | toClass
```

Listing 8.24: Query to find all classes which contain recursive methods

Report the ratio of the usage of generic wildcards

To get the ratio of generic wildcards we first have to query for all type arguments. We then split the output. In one branch we check if the type argument is generic, i.e. the name is `?`. In the other branch we check the opposite, i.e. we apply the `not` operator on the same query. By coloring them differently we not only get the ratio but also get an overview of where the generic wildcards are. The whole query is shown in figure 8.4.



```
? ast -t FormalTypeArgument -global ➡ not ast -n \? ➡ color blue
                                      ast -n \? ➡ color red        ➡ count color.color
```

Figure 8.4: Query to highlight generic wildcard in contrast to normal generic arguments.

Find all uses of covariant array assignments

To find all covariant array assignments we first have to find all assignment nodes. We can then apply a script to filter out non covariant assignments. The query itself is shown in listing 8.25 and the `isCovariantArrayAssignment` script is shown in listing 8.26. Note that the script is only correct if the source code we are working in is Java or a similar language with covariant arrays, otherwise the check on line 6 would not be correct.

```
ast -global -t AssignmentExpression | isCovariantArrayAssignment
```

Listing 8.25: Query to find all covariant array assignments

```
for astTuple in Query.input.tuples('ast'):
   assignment = astTuple.ast
   leftType = assignment.left.type()
   rightType = assignment.right.type()
   if type(leftType) is ArrayType and type(rightType) is ArrayType:
      if not leftType.elementType().equals(rightType.elementType()):
         Query.result.add(astTuple)
```

Listing 8.26: Scripts that filters covariant array assignments if the source code is Java.

8.2 Modifying queries

While there are many systems that offer the querying of data, most of them do not allow to modify the underlying data. In our system modification of the underlying data is possible. A simple example is the modification of tags through tag queries, like `addTags` and `removeTags`. The modification of source code can be achieved with scripts as an extension to the query system. While we currently do not offer any building blocks for complex refactoring simple but interesting use cases can be implemented with relatively small effort as we show in the following example:

8.2.1 Code instrumentation: Put a print statement in all recently modified methods

During a debug session it could be helpful to have a print statement on all recently modified methods. With this in place one can quickly rule out methods which are not executed. We can do this with our system by first getting all the changes at method resolution and then modify all the methods with a script, as shown in listing 8.27. The script, shown in listing 8.28 can leverage the parser of Envision to build expression nodes which the script can then attach to the methods.

```
changes -t Method | addPrintMethodName
```

Listing 8.27: Query to add a print statement to all recently modified methods

```
for changeTuple in Query.input.tuples('change'):
    method = changeTuple.ast
    printExpr = AstModification.buildExpression('System.out.println("Executing method
        {}")'.format(method.name))
    printStmt = Node.createNewNode('ExpressionStatement', None)
    printStmt.expression = printExpr
    method.beginModification('Add print statement')
    method.items.prepend(printStmt)
    method.endModification()
```

Listing 8.28: Script to modify the source code, by adding a print statement at each method in the change tuples.

9 Future work

9.1 Path queries and visualizations

Path queries are very helpful to answer questions that consider relations in programs. LaToza and Myers [11] have shown that data flow and reachability queries are relevant for many problems developers face. Especially correct refactorings require the ability to answer path queries as Verbaere et al. [22] have shown. Our system currently has no support for such queries. It could be interesting to investigate how such queries can be used in combination with our existing framework. We think the visualization of paths and especially data flow information integrated in our visual system can provide helpful insights to users.

9.2 Refactoring support library

With our prototype we support the manipulation of source code through scripts. Yet non trivial refactoring actions will have a high implementation cost. It could be interesting to investigate various refactoring use cases and abstract common operations in a library. Such a library could then provide building blocks for complex refactoring actions by end users.

9.3 User evaluation

We have shown that our prototype is capable of answering various queries. To validate how well users can make use of this system we will need to evaluate our tool in a user study with professional developers. We think the following questions have to be investigated:

Simplicity of queries: How easy is it to understand the individual queries and their arguments? How could the syntax be improved in case it is unclear?

Simplicity of combination of queries: In case a combination of queries is needed how easy can participants combine the correct queries? Is the selected data format suitable to think about?

Presentation of results: Are the results of queries presented in a way that helps to answer the initial question? What other presentations would be helpful?

9.4 More information sources

We currently only access a few information sources. Adding support for querying more information sources will make it possible to answer a wider range of questions. Some examples of possible additional information sources to integrate in the system are:

Compiler Compiler information would be useful for refactoring actions. For example when renaming a field the compiler could be used to find all references which have to be updated.

Source host With the integration of code hosting solutions, such as Github, we can access discussions about code from reviews. Additionally the issue tracker could be used as described below.

Issue tracker and team The integration of a issue tracker will provide access to open and closed issues in the software. Together with team information, a developer of a team could easily find issues to work on. Through the query system open issues could be associated source code that has to be investigated. Fixed issues could provide information about which commits did fix the issue. This information could then be used to find out why the source code is the way it is.

10 Conclusion

In this thesis we extended Envision, a research IDE that features a visual structured editor with information scripting capabilities. We argued that combining information from different domains in a large software project is often a tedious and time consuming task. With the framework we introduced information from different information sources can be combined with ease. We have evaluated our system by showing how it can be used to answer various questions developers have stated.

We presented the information exchange format as foundation of this framework, the queries which create this format from information sources, and how the user can combine the queries in the query prompt and extend the system with scripts. We have shown how the results of queries are visualized in our system.

Since our framework is designed to be extensible we think that users and developers likewise will be able to use our system in unforeseen ways. With the addition of outlined future ideas such as path queries and refactoring support we think our system can be even more helpful. While the visual presentation of results is already useful we believe that enhanced visualizations with graphs etc. can make the system even more advantageous. Through user studies we would like to find further directions for improvements of the current system.

Appendix A

List of implemented queries

In this appendix we present all queries that our system currently has. Note that to make the tuples in the examples readable we use textual code whenever we refer to AST nodes, in the actual implementation we use node pointers to refer to AST nodes.

Aggregating queries

Aggregating queries only work on the input tuples. They provide implicit information in the input in an explicit tuple in the output.

`count`

Description Counts tuples in the input that match on the specified values and differ on others.

Arguments:

`0 (positional)` specifies by which tags to count.

Aliases: None.

Example: To count how many times a color appears in the input, the following query can be used: `count color.color`. For the following input:

`{(color: blue, ast: x), (color: blue, ast: y), (color: red, ast: z)}`

the output would be:

`{(count: 2, color: blue), (count: 1, color: red)}`

An application of this case can be found in the query in figure 8.4.

`join`

Description Joins tuples with different tags in the input and outputs the joined tuples together with the input.

Arguments:

`-value, -v` specifies the values that should be contained in the joined tuple.

`-on, -o` the tagged values that have to match in the tuples. Per default the id value of both tuples is used.

`-as, -a` specifies the tag of the joined tuple.

Aliases: None.

Example: In listing 8.1 we first execute the `changes` query, which produces tuples with `commit` and `change` tags. We then use the `join` query to get author information associated with the AST node. An example input could look like this:

```
{
 commit: (id: 3b975d,..., author: Bob),
 change: (id: 3b975d, ast: x),
 change: (id: 3b975d, ast: y)
}
```

If we now execute `join -v change.id,commit.author,ast -as astAuthor` on this input, `join` automatically detects that it should join on the `id` values. The output would then be:

```
{
 commit: (id: 3b975d,..., author: Bob),
 change: (id: 3b975d, ast: x),
 change: (id: 3b975d, ast: y),
 astAuthor: (id: 3b975d, author: Bob, ast: x),
 astAuthor: (id: 3b975d, author: Bob, ast: y)
}
```

AST queries

AST queries extract information from the Envision model.

`ast`

Description Returns AST nodes that are in the current scope, globally, or nodes in the scope of the input. The user has to either specify the name or the type argument.

Arguments:

`-type, -t` specifies the requested type of the node.

`-name, -n` specifies the requested name of the node.

`-global` applies the query globally.

Aliases:

`methods = ast -t Method`

`classes = ast -t Class`

Examples: To get the methods in a class, a user can open the query prompt on this class and run the `methods` query, this will return all methods in the current class. Similarly if we want all statements of the methods from the query above we can pass it on as follows: `methods | ast -t Statement`. The second query extracts all statement nodes in the methods that it gets as input and only returns those statements in `ast` tuples.

attribute

Description Extracts the specified attribute from the input nodes or the target node and outputs all found attribute nodes in ast tuples.

Arguments:

`-attribute, -at` specifies the name of the attribute to extract.

`-global` applies the query globally to all nodes which have an attribute of the specified name.

Aliases: None.

Example: We use this query in the `filterCasts` script in listing 4.1. A possible input could look like this: `{(ast: dynamic_cast<SomeClass>(...))}` and the output of `attribute -at castType` applied on this input looks like this: `{(ast: SomeClass)}`.

bases

Description Provides the base class relation on classes in the input or on the target class.

Arguments:

`-global` applies the query globally.

`-nodes` If set the query will return all base classes as `ast` tuples instead of relations.

Aliases: None.

Example: Assume we have three classes `C<:B<:A` if query we execute `bases -global` we get the following output:

```
{
 baseclass: (childClass: C, baseClass: B),
 baseclass: (childClass: B, baseClass: A)
}
```

If we would execute `bases -nodes` on the `C` class we would get the following tuples: `{(ast: B), (ast: A)}` since `C` is not a base class of itself.

`callgraph`

Description Similar to `bases`, this query provides the calls relation on methods in the input or on the target method.

Arguments:

`-global` applies the query globally.

`-nodes` returns the nodes in the callgraph as `ast` tuples instead of relations.

Aliases: None.

Examples: Consider the following callgraph: `x()` → `y()` → `z()`.

If we execute `callgraph -global` we get the following tuples:

```
{
 calls: (caller: x(), callee: y()),
 calls: (caller: y(), callee: z())
}
```

If we get the method `y()` as input and execute `callgraph -nodes` we get the following tuples: `{(ast: y()), (ast: z())}`.

`toParent`

Description Takes input tuples with tag `ast` and transforms them to the parent node of the specified type.

Arguments:

`-type, -t` the parent type to convert to.

Aliases:

`toClass = toParent -t Class`

Example If we have methods `x()` and `y()` residing in class `Foo`, inputting `{(ast: x()), (ast: y())}` to the query `toClass` returns the following tuple `{(ast: Foo)}`.

`type`

Description Finds all AST nodes that have the signature specified by the type argument, either locally, in all input nodes, or globally.

Arguments:

`-type, -t` the type signature the returned nodes should have.

`-global` applies the query globally.

Aliases: None.

Examples: If we execute the query `type -t int()` on a class the query returns `ast` tuples containing all methods in this class that return an `int` and take no arguments. If the brackets are omitted the query searches for fields and variable declarations. To leave a type unspecified an underline character can be used, e.g. `type -t int(_)` returns all methods that return an `int` and have one argument of any type.

`definitions`

Description For all references in the input, this query returns tuples which relate the references to the definitions.

Arguments:

`-type, -t` specifies the requested type of the definition node.

`-name, -n` specifies the requested name of the definition node.

`-global` applies the query globally.

Aliases: None.

Examples: We use this query in the `filterCasts` script in listing 4.1. A possible input could look like this: `{(ast: refTo SomeClass)}`, where `refTo` indicates a reference node, the output of `definitions -t Class` applied on this input looks like this: `{definition: (reference: refTo SomeClass, definition: SomeClass)}`.

Debugger queries

`addBreakpoints`

Description Sets breakpoints on the input nodes. The breakpoints are set on all nodes of type `ExpressionStatement`. If the input contains nodes of other types, all children of type `ExpressionStatement` are used to put breakpoints on. For each breakpoint that this query sets it puts a tuple in the following form: `(breakpoint: statement, visible: false)` to the output, where the statement refers to the statement on which a breakpoint was set, and visible specifies if the breakpoint is visible by the user.

Arguments:

`-visible, -v` specifies if the breakpoints should be visible to the user. From invisible breakpoints the debugger will automatically resume once it called all registered callbacks.

Aliases: None.

`traceExecution`

Description For all breakpoint tuples in the input, registers callbacks that counts how many times each breakpoint was hit. The query also sets a callback that restarts the `QueryExecutor` after the program terminated. It then executes the current project. Note after this query a `yield` should be called.

Arguments: None.

Aliases: None.

Filtering queries

`filter`

Description Filters the input depending on the specified conditions.

Arguments:

`0 (positional)` if a single tuple tag is specified only keeps tuples with this tag. If a tagged value, e.g. `tag.value`, is specified either one of the other arguments has to be set.

`1 (positional)` specifies the value that the tagged value specified in the first argument should have. Filter will only keep tuples where the value is the same.

`-extract` extracts the value in `tag.value` in a separate tuples with tag `value` only containing the value.

Aliases: None.

Examples: If the input is:

```
{
 calls: (caller: x(), callee: y()),
 (ast: z())
}
```

the output of `filter ast` is `{(ast: z())}`

the output of `filter -extract calls.caller` is `{(caller: x())}`

the output of `filter calls.callee y()` is `{calls: (caller: x(), callee: y())}`

`canReach`

Description Checks if a relation between AST nodes can reach a certain node and outputs all the nodes that can and nothing else. The specified tuples with the `relation` argument should contain exactly two values which are AST nodes and any other values.

Arguments:

`-name, -n` specifies the name of the node to reach.

`-relation, -r` specifies the name of the relation to work on.

`-self` if self is set only nodes that can reach themselves through some path are returned.

Aliases: None.

Examples: Consider the following callgraph: `x()` → `y()` and `z()` → `z()`.

The query: `callgraph -global | canReach -r calls -n y`

returns `{(ast: x()), (ast: y())}`.

The query `callgraph -global | canReach -r calls -self`

returns `{(ast: z())}`.

Tag queries

`tags`

Description Lets the user query and modify tags in the Envision model. It either works on the target node or on the input nodes.

Arguments:

`-name, -n` specifies the requested name of the tag.

`-add, -a` specifies that the tag should be added.

`-remove, -r` specifies that the tag should be removed.

`-pesistent, -p` specifies that the change should be added to the nodes. This is set per default. If it is not persistent the changes are only applied to the data format.

Aliases:

`addTags = tags -a`

`removeTags = tags -r`

Examples: If we have the methods `x(), y(), z()` in a project the following query: `methods -global | addTags -n TODO` adds a `TODO` tag to the three methods and returns the following tuple `{(tag: TODO, ast: x()), (tag: TODO, ast: y()), (tag: TODO, ast: z())}`.

Version control queries

`changes`

Description Provides access to the version control information source. If AST nodes are in the input the query attaches commit information to those nodes, if they were modified in the specified commits. If no input is provided the query returns commit information for all changed nodes in the specified commits.

Arguments:

`-count, -c` specifies the amount of commits to consider.

`-type, -t` specifies the minimal type of the nodes returned.

`-in` specifies a specific commit or a range of commits to look at.

`-local` specifies that only changes in the target node should be reported.

Aliases: None.

Visualizing queries

`heatmap`

Description Generates color tuples from a range of numbers attached to AST properties in the input tuples. The query does not visualize something on its own, but the default visualizer recognizes color tuples.

Arguments:

`-name, -n` specifies the tag of the tuples that contain the value to generate a color from, per default this is count.

Aliases: None.

Example: For the input `{(count: 1, ast: m()), (count: 10,ast: x())}` the query returns `{(color: green, ast: m()), (color: red, ast: x())}`. Note that the actual colors are specified in hexadecimal format, but for the sake of readability we used names in this example.

`show`

Description Shows the results of queries. With no arguments this query can be omitted since it is executed per default at the end of a query pipeline.

Arguments:

`-info, -i` specifies a list of tagged values that should be printed on the attached ast node.

`-sort, -s` if a info argument is specified, sort the info values by this value.

Aliases:

```
info = show -info
```

References

[1] 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)*, pages 9–12, 2014.

[2] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. Fastdash: A visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 1313–1322, New York, NY, USA, 2007. ACM.

[3] Brian de Alwis and Gail C. Murphy. Answering conceptual queries with ferret. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 21–30, New York, NY, USA, 2008. ACM.

[4] Thomas Fritz and Gail 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, pages 175–184, New York, NY, USA, 2010. ACM.

[5] Balz Guenat. Tree-based version control in envision. Bachelor's Thesis, 2015.

[6] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. Supporting developers' coordination in the ide. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, pages 518–532, New York, NY, USA, 2015. ACM.

[7] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 2–27, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] Aftab Iqbal, Oana Ureche, Michael Hausenblas, and Giovanni Tummarello. Ld2sd: Linked data driven software development. In *In 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 09*, 2009.

[9] Shihai Jiang, Liwei Shen, Xin Peng, L.V. Zhaojin, and Wenyun Zhao. Understanding developers' natural language queries with interactive clarification. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 13–22, March 2015.

[10] T.D. LaToza and B.A. Myers. Visualizing call graphs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 117–124, Sept 2011.

[11] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM.

[12] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[13] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[14] Martin Otth. Fine-grained software version control based on a program's abstract syntax tree. Master's thesis, 2014.

[15] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[16] Todd W. Schiller and Brandon Lucia. Playing cupid: The ide as a matchmaker for plug-ins. In *Proceedings of the Second International Workshop on Developing Tools As Plug-Ins*, TOPI '12, pages 1–6, Piscataway, NJ, USA, 2012. IEEE Press.

[17] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, July 2008.

[18] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 21–. IBM Press, 1997.

[19] M. Storey, J. Ryall, J. Singer, D. Myers, Li-Te Cheng, and M. Muller. How software developers use tagging to support reminding and refinding. *Software Engineering, IEEE Transactions on*, 35(4):470–483, July 2009.

[20] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Waypointing and social tagging to support program navigation. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, pages 1367–1372, New York, NY, USA, 2006. ACM.

[21] Raoul-Gabriel Urma and Alan 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. Special Issue on New Ideas and Emerging Results in Understanding Software.

[22] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: A scripting language for refactoring. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 172–181, New York, NY, USA, 2006. ACM.

[23] Michael Würsch, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald Gall. Seon: A pyramid of ontologies for software evolution and its applications. *Computing*, 94(11):857–885, 2012.

[24] Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 165–174, New York, NY, USA, 2010. ACM.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Augmenting software development with information scripting

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Vogel	Lukas

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 1.12.2015	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.