EVOLVE: AN EXTENSIBLE SOFTWARE VISUALIZATION FRAMEWORK

by
Qin Wang

School of Computer Science
McGill University, Montreal

June 2002

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Existing visualization tools typically do not provide a simple mechanism for adding new visualization techniques, and are often coupled with inflexible data input mechanisms. This thesis presents EVolve, a flexible and extensible framework for visualizing program characteristics and behaviour. The framework is flexible in the sense that it can visualize many kinds of data, and it is extensible in the sense that it is quite straightforward to add new kinds of visualizations.

The overall architecture of the framework consists of the core EVolve platform that communicates with data sources via a well defined data protocol and which communicates with visualization methods via a visualization protocol.

Given a data source, an end user can use EVolve as a stand-alone tool by interactively creating, configuring and modifying one or more visualizations. A variety of visualizations are provided in the standard EVolve visualization library. EVolve can also be used to build custom visualizers by implementing new data sources and/or new kinds of visualizations.

The thesis presents an overview of the framework, its data protocol and visualization protocol, and a detailed case study showing how to extend EVolve.

Résumé

Les outils de visualisation existants ne fournissent habituellement pas un mécanisme simple pour l'ajout de nouvelles techniques de visualisation, et de plus utilisent souvent un mécanisme d'entrée de données inflexible. Cette thèse présente EVolve, un cadre d'applications flexible et extensible permettant la visualisation des charactéristiques et du comportement d'une application. Le cadre tire sa flexibilité du fait qu'il peut visualiser plusieurs types de données et son extensibilité du fait qu'il est simple d'ajouter de nouveaux types de visualisations.

L'ensemble de l'architecture du cadre est constitué du noyau de la plateforme EVolve, qui communique avec les sources de données via un protocole de données bien défini et qui communique avec les méthodes de visualisation via un protocole de visualisation.

Partant d'une source de données, un utilisateur final peut utiliser EVolve comme un outil autonome en créant, configurant et modifiant interactivement une ou plusieurs visualisations. Une variété de visualisations est incluse dans la librairie de visualisation standard d'EVolve. EVolve peut aussi être utilisé dans le but de construire des visualisateurs sur mesure en implémentant de nouvelles sources de données et/ou de nouveaux types de visualisation.

La thèse présente un survol du cadre d'applications, de son protocole de données et de son protocole de visualisation, ainsi qu'une étude de cas détaillée démontrant comment étendre EVolve.

Acknowledgments

This thesis would not be written without the help from others. First of all, I'd like to give special thanks to my supervisors, Laurie Hendren and Karel Driesen, for giving me great help in academic research, and providing all kinds of support and consistent encouragement throughout the course of my study at McGill University.

I would also like to thank all those people who contributed to EVolve and helped to make it a reality. In particular, I would like to thank Rhodes Brown for providing valuable suggestions and great ideas on the design of EVolve, and for being the first user; John Jorgensen for giving all kinds of technical support; Bruno Dufour for testing EVolve and translating the abstract of this thesis to French; and all the students of the CS308-764 course (winter 2002) for using EVolve and giving me essential feedback to improve it.

Special thanks go to my friends in Montreal, who helped me in various aspects, and made my life here interesting and colorful. In particular, Huaizhi, Yichi, Michelle, Vincent, Xiaolong, Hai, Xiangrong, and Sanna.

I'm always thankful to my wonderful family. They are always there when I'm in trouble, and without their full support, I wouldn't be able to study at McGill. Thanks Dad, Mom, Feng, Li, Bing, and Peng.

Finally, I would like to thank my wife, Rui, for all she has done for me and my family in the last two years. Without her, this thesis would not be possible.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Overview

Object-oriented programming has the advantages of flexibility and reusability, which facilitate developing large and complicated software systems. However, the growth of complexity may make it more difficult to understand the system. This difficulty, in turn, causes many problems in performance analysis, optimization, and maintenance of the system. Different kinds of techniques have been used to solve these problems, and software visualization tools provide one way for programmers to understand the structure and performance of large software systems.

Figure 1.1 shows one way of representing a software visualization process. First, the data provider profiles the program and generates trace data that contains information about some characteristics of the program. Then, the visualization provider analyzes the trace data and produces the corresponding visual representation of these characteristics. Finally, the end user uses the visual representation to facilitate understanding of the program.

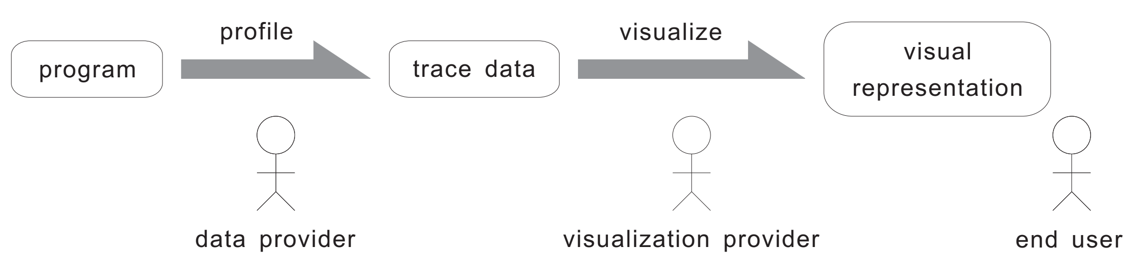

Figure 1.1: Software visualization process.

For example, to help understanding the run-time memory usage of a program, the data provider may instrument the source code of the program, so that each time when an object is created, the type of the object, the size of memory allocated for the object, and the method which creates the object is recorded in a trace data file. The trace data is then used by the visualization provider to generate graphs and charts which can facilitate end users finding the answers to questions such as: "Which types of objects are allocated most frequently?", "Which methods have allocated most of the memory?", and so on.

Because of the usefulness and importance of software visualization, many software visualization systems have been developed in the last decade[1, 3, 17]. These systems often focus on visualizing a fixed set of program characteristics, and contain several built-in visualizations that can provide useful information of these characteristics. However, in some cases, users may still want to have additional visualizations so that they can get a better understanding of their program. Unfortunately, most existing software visualization systems are designed without flexibility and extensibility in mind, so adding new visualizations to these systems is difficult, if not impossible, for the following reasons.

First, in order to transmit the trace data, the data provider and visualization provider must agree on a certain data protocol. If the system is not extensible, the data protocol usually has to be modified to support new visualizations. Consequently, the data provider must change the format of the trace data to fit the new data protocol, and the old traces have to be updated correspondingly. What's more, the visualization provider also has to update the old visualizations according to the new data protocol.

Second, trace data usually contains large amounts of information, so a software visualization system must provide data manipulating functionalities, such as finding, partitioning, and filtering subsets of the data, to facilitate end users finding the information they need. These functionalities demand that all the visualizations in the system should be able to communicate with each other. Therefore, if the system is not extensible, adding new visualizations to it usually requires updating all the old visualizations so that they can cooperate with the new ones.

The goal of this work is to provide a software visualization framework that is extensible and flexible. The EVolve framework provides the ability to visualize different types of trace data, and simplifies the task of implementing new visualizations and incorporating them into the framework.

The structure of EVolve is shown in Figure 1.2. The essential parts of the framework are the EVolve platform, the data protocol, and the visualization protocol. The

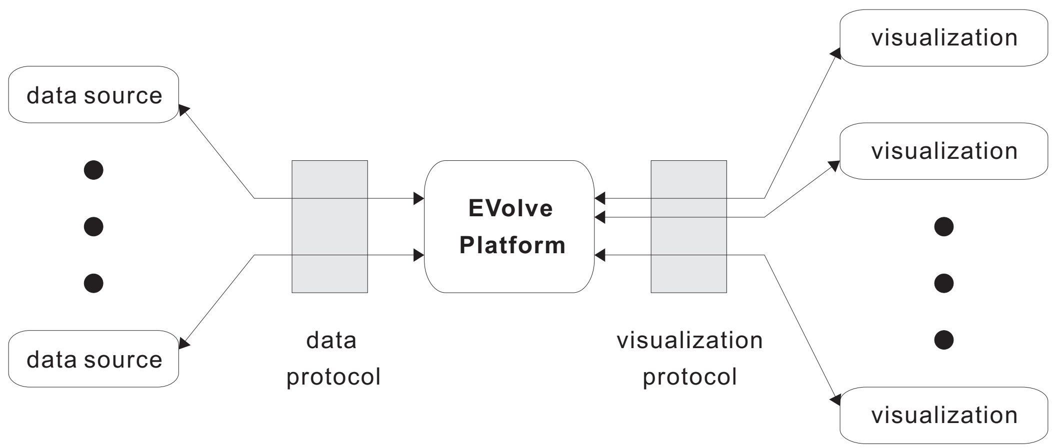

Figure 1.2: Structure of EVolve.

EVolve platform is the core of the framework. It's the connection between the data source and the visualizations, and it supports the communication among the visualizations. What's more, the platform provides a set of data manipulating functionalities for end users.

The extensibility and flexibility of EVolve mainly comes from the two protocols. The EVolve platform receives trace data through the data protocol, which uses a flexible method to represent the data. The data representation technique used in EVolve is general enough to represent various kinds of information, and can provide additional information for the visualizations to interpret the data. Furthermore, the data protocol doesn't specify the source of the data. Therefore, EVolve can receive trace data from different kinds of data source, such as a file, a data stream, or a database.

The visualization protocol handles the communication between the platform and the visualizations. In EVolve, there's no direct interaction among the visualizations, so adding new visualizations to the framework won't require updating the existing ones. With the visualization protocol, EVolve can provide data manipulating functionalities while still keeping the visualizations independent.

Based on the EVolve framework, we have implemented various visualizations and data sources, and tested the framework on several different tasks for large Java programs. The result of these experiments has verified the extensibility and flexibility of EVolve.

1.2 Background

Roughly speaking, related work of EVolve falls into the following categories[1]:

Performance tuning: there are many profiling tools for program performance tuning and optimizing (e.g. Jinsight[1], JProbe[2], and Optimizeit[3]), and these tools often focus on visualizing the usage of system resource, such as CPU time and memory, to help programmers find the bottleneck of their programs.

Software structure visualizing: many software visualization systems are developed to facilitate reverse engineering, such as Rigi[24] (using SHriMP views[23]) and Moose[11]. These systems help developers understanding the hierarchy and structure of their systems by visualizing static information (classes, methods, fields, and etc.) that is usually obtained from parsing the source code.

Run-time behavior visualizing: some software visualization systems, for example, Bloom[17], focus on visualizing various sorts of dynamic information (such as object interaction, memory allocation, and field access) that is collected during the execution of programs. These systems can provide detailed information about the run-time behaviour of software, which most performance tuning tools cannot, and they are mainly used in research fields.

These systems are usually developed to solve some specific problems. Few of them allow users using their own data source, and even fewer support adding new visualizations.

As depicted in Figure 1.1, users of a software visualization system can take three kinds of different roles: end user, data provider, and visualization provider. Unlike most of the above systems, which are designed only for the end users, EVolve can facilitate all these three:

1. **End users:** EVolve provides a set of built-in visualizations to visualize various types of program characteristics, including both run-time behaviour and static information. For end users, EVolve helps them visualize their trace data and find the most important information from it, and it is similar to the other visualization systems.

2. **Data providers:** Trace data can be sent to EVolve in two ways. One is using the built-in data source of EVolve directly, which requires the trace data to be

[1]Chapter 6 discusses related work in detail.

stored in files according to a certain format. Another way is that users can implement their own data source, and send trace data to EVolve through the data protocol. This is convenient for those users who have special requirements. For data providers, EVolve is a flexible visualization toolkit.

3. **Visualization providers:** In case the built-in visualizations are not suitable to visualize some specific trace data, or cannot provide enough information, users can create new visualizations that fit their need, and incorporate them into EVolve easily. For visualization providers, EVolve is a framework that helps them implementing various visualization techniques.

EVolve is part of the Sable Toolkit for Extensible Profiling (STEP), but it can work as an independent framework as well. Like the other parts of STEP, EVolve is also implemented in Java.

1.3 Thesis Contributions

The contributions of this thesis are the design, implementation, and experimental validation of the EVolve framework.

1.3.1 Design

The design of EVolve mainly includes the following:

- A flexible data representation technique that is capable of representing various types of program characteristics, and can provide additional information for interpreting the data.
- A data protocol that specifies the communication between the framework and the data source, allowing the framework to use different kinds of data source.
- A visualization protocol that specifies the communication between the framework and the visualizations, making it easy for users to add new visualizations to the framework.

1.3.2 Implementation

The implementation of EVolve consists of:

- The EVolve platform.
- A data interface and a visualization interface that are implemented according to the corresponding protocols.
- A set of data manipulating functionalities, such as partitioning and filtering subsets of the data.
- A group of built-in visualizations.

The source code and documentation of EVolve is available at:
`http://www.sable.mcgill.ca/evolve/`

1.3.3 Experimentation

Great effort has been placed in testing the flexibility and extensibility of EVolve. In particular, 20 graduate students in an upper level graduate course (CS308-764: Run-time Support for Object-Oriented Programming Languages) at McGill used EVolve to characterize the run-time behaviour of Java programs. This experiment demonstrated the ease of use and clarity of EVolve for the students claimed that EVolve facilitated and enhanced program understanding.

Furthermore, various visualization techniques that are used in other software visualization systems can be implemented and incorporated into EVolve without changing any part of the EVolve platform and/or the interfaces.

1.4 Thesis Organization

The rest of the thesis is organized as follows. First, Chapter 2 introduces the architecture of EVolve and some basic concepts used in the framework. Then, Chapter 3 describes the data protocol of EVolve, and Chapter 4 describes the visualization protocol, followed by a detailed case study in Chapter 5. Finally, Chapter 6 discusses some related work, and Chapter 7 gives our conclusions and the future work.

Chapter 2

Overview and Basic Concepts

The extensibility of a software visualization system mainly depends on its architecture, and in an extensible framework like EVolve, every visualization should be able to work independently (not only independent from other visualizations, but also independent from the trace data).

On the other hand, the flexibility of a software visualization system largely depends on how the trace data is represented in the system. Therefore, EVolve needs a data representation technique that is capable of representing various kinds of program characteristics.

This chapter describes the most fundamental parts of EVolve, the architectural design and the data representation technique, which make EVolve extensible and flexible. To make the discussion more concrete, an illustrative example is used in this chapter. Assume that a user wants to visualize method invocation and object allocation information that is collected during the execution of a program. For every method invocation, the following information is included in the trace data: the method executed, the defining class of this method, and the thread in which the method is invoked.

For every object allocation, the trace data includes: the type of the object, the size of memory allocated, the method which creates the object, the defining class of this method, and the thread in which the object is created.

2.1 Architectural Design

As mentioned before, the essential parts of EVolve are the platform, the data protocol, and the visualization protocol. The relationship among these three parts is shown in Figure 2.1.

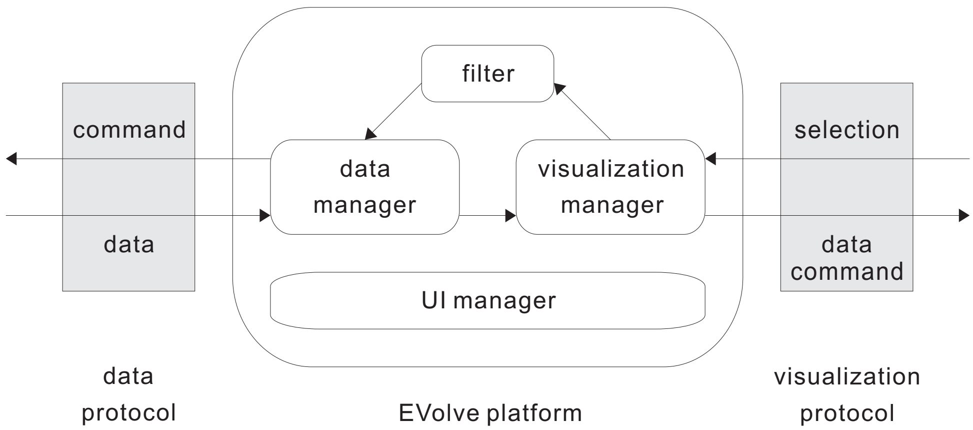

Figure 2.1: Architecture of EVolve.

The EVolve platform mainly consists of four components: a user-interface manager that interacts with the end user, a data manager that communicates with the data source, a visualization manager that controls the visualizations, and a filter that provides data manipulating functionalities.

The major responsibility of the user-interface manager is sending commands from the end user to the other components of the platform and controlling the layout of the visualizations. But it also provides other functions such as exporting visualizations to images and monitoring data processing.

Externally, the data manager communicates with the data source through the data protocol, and the visualization manager controls the visualizations through the visualization protocol. The difference is that data manager sends commands to the data source and receives data from it, while the visualization manager sends both commands and data to the visualizations. What the visualization manager receives from the visualizations are subsets of the data that the end user selected in the visualizations, and these selections are used by the filter to perform data manipulating operations.

Internally, within each visualization cycle, the data manager first sends data to the visualization manager, and after the data is visualized in the visualizations, the visualization manager sends the selections that the end user made to the filter. Then, the filter uses these selections to determine which part of the data should be visualized in the next visualization process, and this information is sent pack to the data manager.

Because of this architecture, in EVolve, a visualization can only communicate with the data source and the other visualizations through the EVolve platform. This means that there's no direct interaction between the data source and the visualizations, and there's also no direct interaction among the visualizations themselves. Therefore, the architecture of EVolve keeps the data source and the visualizations independent, and this allows EVolve to be extensible.

2.2 Data Representation

2.2.1 Element, Event, and Entity

In EVolve, the basic information unit is called an *element*, and the trace data is represented as a sequence of different types of elements. In the example mentioned above, there are five types of elements: method invocation, object allocation, method, class, and thread.

Elements are linked together by *references*, and an element can have several references to another type of elements. Figure 2.2 shows the references among the elements that are related to object allocation. Note that object allocations have two references to classes: one indicating the type of the new object, and the other showing the allocating class.

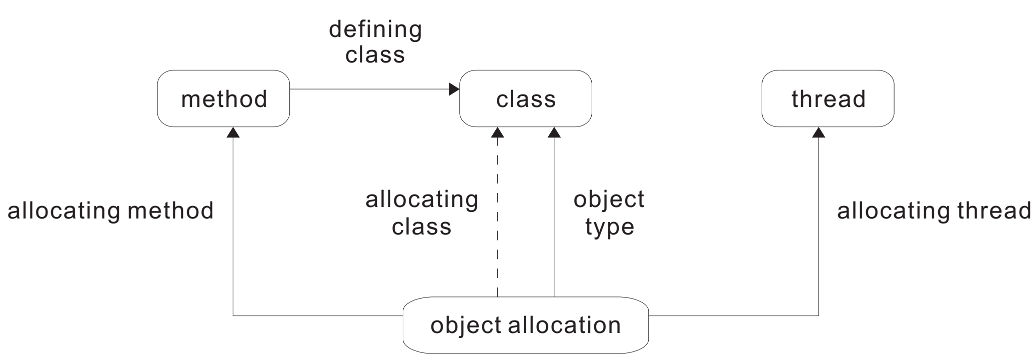

Figure 2.2: References among elements.

A reference can be *direct* or *indirect*. In Figure 2.2, the allocating class is an indirect reference because it is the combination of other references (in this case, it's the combination of allocating method and defining class, i.e. the allocating class of an object allocation is the defining class of the allocating method of that object allocation), and all the other references are direct.

In EVolve, an element can be either an *event* or an *entity*. Events represent all kinds of things that occurred during the execution of programs, such as method invocation and object allocation. The sequence of events in the trace data indicates the order of their occurrence, and changing the sequence will lead to misinterpreting the data in visualizations.

On the contrary, entities represent elements that don't "occur", such as methods, classes, and threads. Unlike events, the sequence of entities in the trace data doesn't have any specific meaning, and changing it won't affect the visualizations.

Furthermore, events and entities are different in the following aspects:

1. Elements can only have references to entities, events cannot be referred to.
2. Entities usually have names, but events don't.
3. Trace data doesn't contain copies of the same entity, but the same events can appear many times.
4. Normally, the number of events in a trace is much larger than the number of entities. (The following sub-section describes this in detail.)

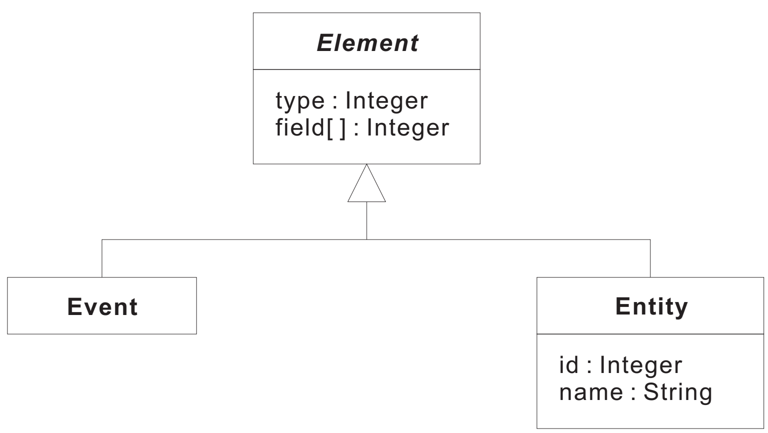

Figure 2.3: Class diagram of element, event, and entity.

Figure 2.3 shows the class diagram of element, event, and entity. Every element has a group of *fields*, which can be either values related to the element, or references to entities (represented as entity identifier). For example, an object allocation has a value indicating the size of memory allocated, and references to method, class, and thread.

An event doesn't contain additional information, but an entity has a name and an identifier.

Entity vs. Event

As mentioned above, in trace data, the number of events is much larger than the number of entities. To demonstrate this, we counted the number of four different types of elements created during the execution of six SPEC[4] JVM98 benchmarks. Among these elements, classes and methods are entities, while method invocation and object allocation are events.

Table 2.1 shows the number of each type of elements and Table 2.2 shows the total number of entities and elements, as well as the percentage of entities among all the elements.

benchmark	class	method	method invocation	object allocation
db	155	998	160137	16171
jess	352	1399	667999	66345
mpegaudio	208	1161	1225781	19494
jack	274	1332	3318260	371115
mtrt	207	1129	5572704	311660
compress	179	986	17416524	13913

Table 2.1: Number of different types of elements in six SPEC JVM98 benchmarks.

benchmark	all entities	all elements	percentage
db	1153	177461	0.65
jess	1751	736095	0.24
mpegaudio	1369	1246644	0.11
jack	1606	3690981	0.044
mtrt	1336	5885700	0.023
compress	1165	17431602	0.0067

Table 2.2: Percentage of entities among elements.

These two tables also show that:

- although there's a big difference among the number of events occurred during the execution of these benchmarks, the difference among the number of entities involved is relatively small.

- for each of the six benchmarks, only less than one percent of the elements are entities.

- as the number of elements increases, the percentage of entities among elements decreases.

2.2.2 Element Definition

Events and entities can represent various kinds of program characteristics. However, to visualize the elements, visualizations still need additional information. For example, visualizations must know that among the items of an element, which are values and which are references. Also, if an item is a reference, which type of entity it refers to. What's more, some visualizations are designed to visualize elements that have a certain property (for example, a tree-like visualization can only visualize entities that belong to a certain hierarchy), so visualizations also have to know the properties of the elements.

In EVolve, the additional information is represented by *element definitions*, and an element definition consists of the definitions of the fields that belong to the corresponding elements.

A *field definition* indicates whether the field is a reference or not, and if it is, which type of entity it refers to. What's more, the definition also includes the properties that the field have, and these properties are checked by visualizations when choosing the appropriate fields and elements to visualize. The next section discusses this in detail.

2.3 Visualization Characterization

Corresponding to the data representation technique described above, EVolve also uses a visualization characterization technique that forms a connection between trace data and visualizations.

2.3.1 Subject and Dimension

In order to visualize the trace data, a visualization has to know which elements should be visualized, and how to map the fields of these elements to their corresponding visual representation. In EVolve, the *subject* and *dimensions* of a visualization are used to associate elements and fields with the visual representation.

The subject of a visualization indicates which type of elements it visualizes, and the visualization manager sends elements to the visualizations according to their subjects. So in EVolve, every visualization only receives elements that belong to its subject (i.e. method invocation events won't be sent to a visualization that visualizes object allocations).

The dimensions of a visualization determine how it interprets the elements. To explain this, assume that a bar chart (Figure 2.4) is used to visualize object allocations. The bar chart has two dimensions, one is its X-axis, the other is the height of the bars (the Y-axis). For object allocation events, if the type of the object created is mapped to the X-axis and the size of memory allocated is mapped to the height of the bars, then the bar chart shows how much memory is allocated for each type of objects. And if the allocating method is mapped to the X-axis while the size of memory allocated is still mapped to the height of the bars, then the bar chart shows how much memory is allocated by each method.

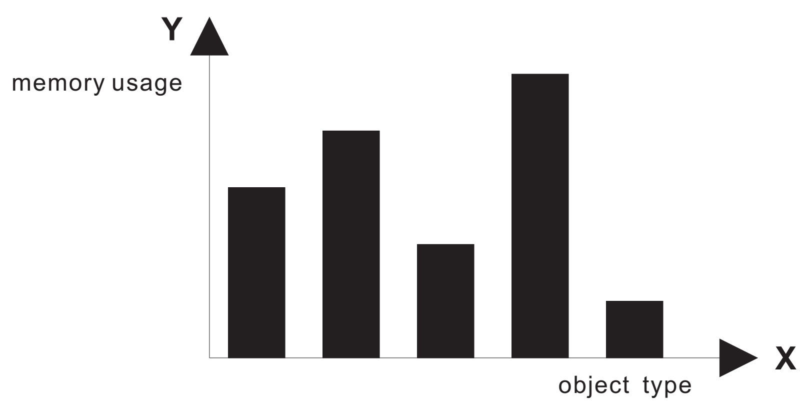

Figure 2.4: Bar chart example.

The bar chart is usually depicted as a two-dimensional graph, and the two dimensions of the graph are equivalent to the dimensions of the visualization. However, this is not always the case. For example, the color and texture of the bars can be two additional dimensions of the bar chart.

2.3.2 Visualization Definition

Before visualizing the trace data, the subject and dimensions of a visualization must be determined according to its *definition*. The definition of a visualization describes what kinds of fields can be mapped to its dimensions.

For each dimension of the visualization, only fields that have a certain property can be mapped to it. For example, in a bar chart, the X-axis is usually used to represent entities and the height of the bar normally represents additive values. So to visualize object allocations with a bar chart, object type and allocating method can be mapped to the X-axis, and the size of memory allocated can be mapped to the height of the bar. If items are mapped in the opposite way, the bar chart cannot work properly. So the visualization definition has to tell what property is required by each dimension.

The properties used in visualization definition are same to those that are used in element definition, so data provider and visualization provider must use the same set of properties. EVolve defines some general-purpose properties, and users also can define new properties that are specific to their requirements.

Amount vs. Coordinate

Two of the predefined properties need to be mentioned here, *amount* and *coordinate*[20], which represent different types of values. Coordinates are values that represent temporal or spatial points in a particular frame of reference, such as the address where an object is allocated in memory. On the other hand, amounts are values that represent certain kinds of quantities, such as the size of memory allocated for an object. Naturally, coordinates are non-additive and amounts are additive.

It is important to differentiate amounts and coordinates because normally they should be visualized using different types of visualizations. For example, in the bar chart, the size (height) of the bars is a quantitative measurement so it should be used to visualize amounts, and visualizing coordinates with a bar chart is inappropriate. Similarly, plot charts are designed to visualize coordinates, and they shouldn't be used to visualize amounts.

Figure 2.5 shows the use of bar chart and plot chart for visualizing where five objects are allocated in the memory. The addresses of memory are coordinates, and it's obvious that using bar chart for them is inappropriate, because that suggests certain amounts of memory are allocated. On the contrary, using plot chart to visualize the addresses is effective.

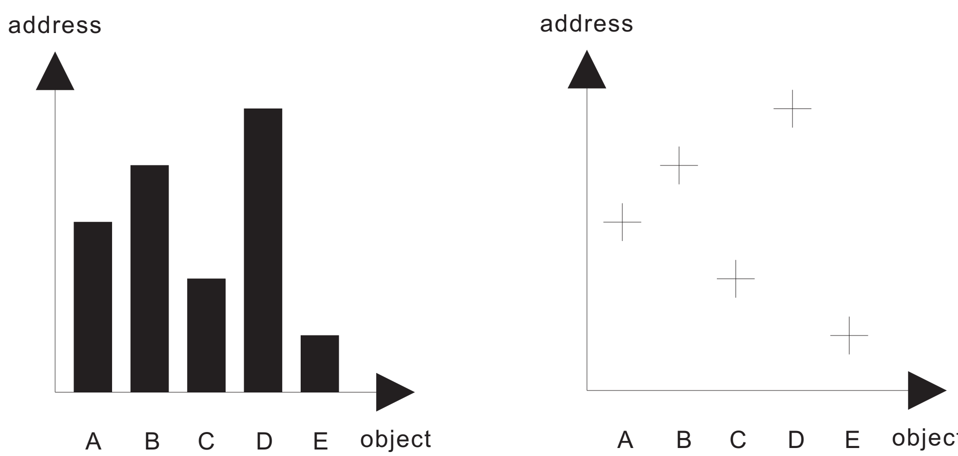

Figure 2.5: Using bar chart and plot chart for coordinates.

Chapter 3

Data Protocol

In order to use EVolve to visualize trace data, a data source has to create elements (events and entities) that represent the trace data and send these elements as well as their definitions to the EVolve platform.

EVolve defines a `DataSource` interface that specifies the interaction between the EVolve platform and the data source. By using the `DataSource` interface, the EVolve platform can read various types of data traces, and this guarantees the flexibility of EVolve.

Furthermore, EVolve provides `ElementBuilder` to simplify the process of defining and creating elements. Element builders also verify the elements and guarantee that the EVolve platform will be able to use these elements.

Figure 3.1 depicts the relationship between the user-defined data source (in this case, the MyDataSource) and EVolve. The user-defined data source needs to implement the `DataSource` interface and use element builders to create elements and their definitions. These elements and element definitions are then sent to the EVolve platform through the `DataSource` interface.

The following two sections describe the `DataSource` interface and how to use the `ElementBuilder`, and the last section of this chapter discusses the data storage technique used in EVolve.

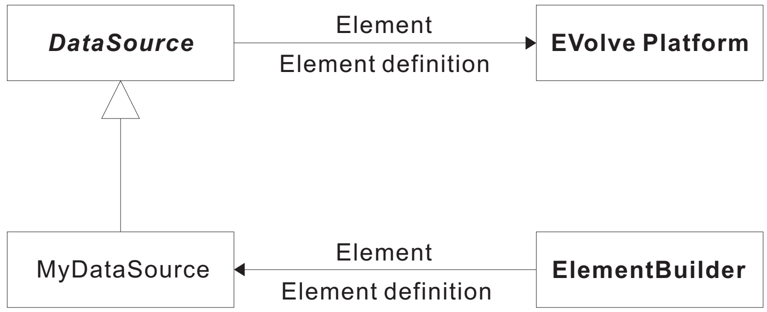

Figure 3.1: Relationship between user-defined data source and EVolve.

3.1 Data Source Interface

The `DataSource` interface specifies the interaction between the EVolve platform and the data source. It consists of seven methods that are called by the EVolve platform to read trace data from the data source, and data providers must implement these methods according to the specific format of their data traces.

The first method called by the EVolve platform makes a connection with the data source:

`public void init()`
: Initializes the data source, such as opening a file or connecting to a database.

The rest of the `DataSource` interface are actually three pairs of methods that allow the EVolve platform to read element definitions, entities, and events from the data source.

The first pair of methods ask the data source to build the element definitions and then read all the definitions from it. Because the element definitions define the format and properties of the elements, the EVolve platform reads all these definitions before starting to read the elements.

`public void startBuildDefinition()`
: Asks the data source to start building the element definitions (both entity definitions and event definitions). This method is called only once by the EVolve platform.

`public ElementDefinition getNextDefinition()`
: Gets the next element definition (returns `null` if all the definitions are sent).

The EVolve platform keeps on calling this method until all the definitions are read.

The other two pairs of methods are similar. After all the element definitions are received, the EVolve platform first reads all the entities, then starts to read all the events from the data source.

`public void startBuildEntity()`
Asks the data source to start building the entities.

`public Entity getNextEntity()`
Gets the next entity (returns `null` if all the entities are sent).

`public void startBuildEvent()`
Asks the data source to start building the events.

`public Event getNextEvent()`
Gets the next event (returns `null` if all the events are sent).

The `DataSource` interface of EVolve is simple and general enough so that data providers can send various sorts of data traces to the EVolve platform easily.

3.2 Element Builder

As shown in Figure 2.3, internally, the fields of an element are represented as an array of integers instead of objects. The advantage of this is that it allows the visualizations to process the elements more efficiently. However, this also makes it more difficult for data providers to create the elements directly, and makes debugging the data sources more complicated.

Therefore, EVolve provides `ElementBuilder` to facilitate the process of defining and creating elements. What's more, the element builder is also a verifying mechanism that helps data providers to debug their data sources.

The element builder of EVolve is based on the well-known builder pattern[8]. As shown in Figure 3.2, interaction between a data source and an element builder mainly consists of two phases, builder initialization, which specifies the format of the elements that will be created by the builder, and element creation, which creates the elements according to the format. Each of these two phases has three steps.

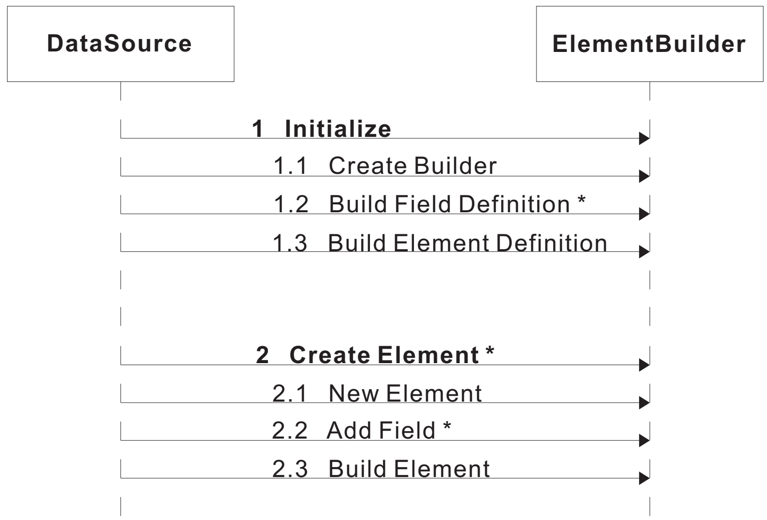

Figure 3.2: Interaction between a data source and an element builder.

To help explaining these two phases and demonstrating how to use the element builders, assume that the data source needs to send elements that represent the following piece of information to the EVolve platform:

```
ClassA.MethodA() {
    ClassB b = new ClassB();
}
```

To do so, the data source must create the following elements: two class entities that represent ClassA and ClassB, respectively, a method entity that represents MethodA, and an object allocation event that represents the creation of object b. Figure 3.3

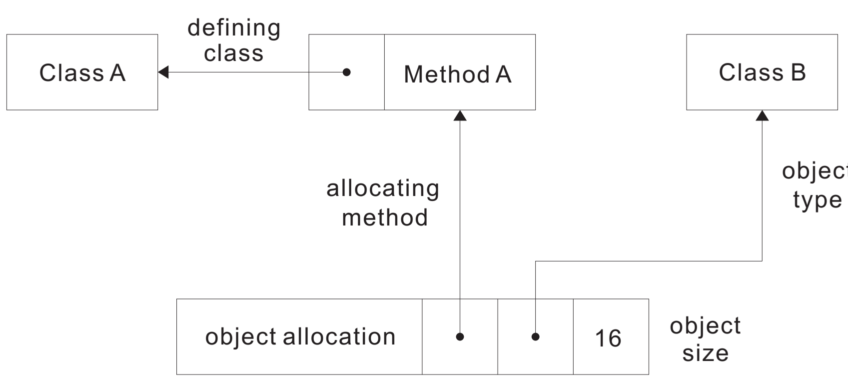

Figure 3.3: Relationship among the elements of the example.

depicts the relationship among these elements.

3.2.1 Builder Initialization

During the builder initialization phase, the data source creates element builders, specifies the format of the elements that will be created by the builders, and uses the builders to create element definitions.

Create Builder

In order to create elements and element definitions, the data source must first create an element builder (either an entity builder or an event builder, as shown in Figure 3.4[1]) for every type of element:

`public EntityBuilder(String entityName, String entityDescription)`
    Creates an entity builder.

`public EventBuilder(String eventName, String eventDescription)`
    Creates an event builder.

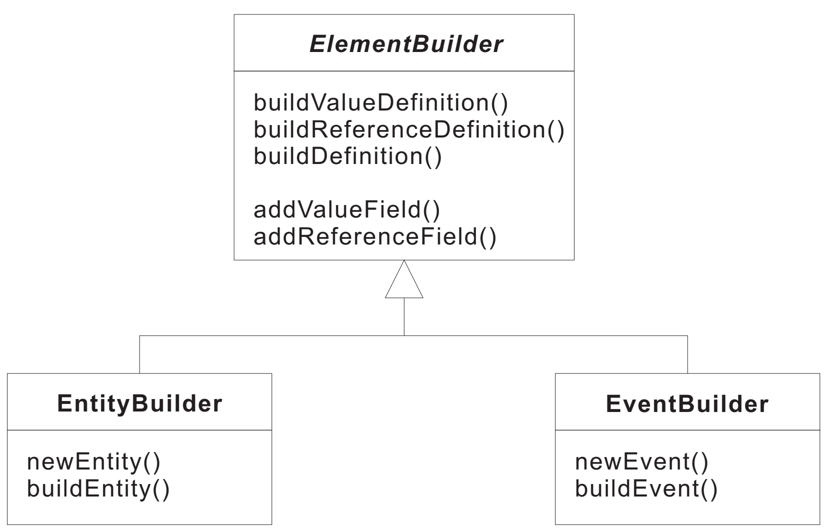

Figure 3.4: Class diagram of element builder.

[1]Methods included in Figure 3.4 are described in the following sections.

There are three types of element in the example above: class, method, and object allocation. Correspondingly, the data source has to create two entity builders and an event builder:

```
EntityBuilder classBuilder = new EntityBuilder("Class", null);
EntityBuilder methodBuilder = new EntityBuilder("Method", null);
EventBuilder allocationBuilder = new EventBuilder("Allocation", "Object allocation");
```

Note that the allocation builder adds a description[2] for allocation events.

Build Definition

After the element builders are created, the data source needs to use the builders to build field definitions and element definitions. Building the definitions serves two purposes. First, the data source has to send the element definitions to the EVolve platform, as required by the `DataSource` interface. Second, these definitions specify the format of the elements to be built by the element builders, and this information allows the builders to validate the trace data when creating elements.

As shown in Figure 3.2, in order to build the element definition, the data source first has to build all the field definitions. Following are the related methods:

```
public FieldDefinition buildValueDefinition(String fieldName,
    String[] fieldProperty, String fieldDescription)
```

Builds the definition of a value field.

```
public FieldDefinition buildReferenceDefinition(String fieldName,
    EntityBuilder referenceBuilder, String[] fieldProperty,
    String fieldDescription)
```

Builds the definition of a reference field.

```
public ElementDefinition buildDefinition()
```

Builds the element definition.

In the above example, class entities don't have any fields (see Figure 3.3), so there's no field definitions to be built and the element definition of class entities can

[2]Descriptions of elements and fields are used to help end-users configuring the visualizations, i.e. selecting the subjects and dimensions of the visualizations. See Chapter 4 for more information on visualization configuration.

be built directly:

```
ElementDefinition classDefinition = classBuilder.buildDefinition();
```

Method entities have references to their defining classes. This means that each method entity has a reference field indicating its defining class. Therefore, the element definition of method entities contains a corresponding field definition of a reference field, and the field definition has to be built before building the element definition of method entities:

```
FieldDefinition definingClass = methodBuilder.buildReferenceDefinition("Defining Class",
    classBuilder, null, "Defining class of the method");

ElementDefinition methodDefinition = methodBuilder.buildDefinition();
```

The above code specifies that the name of the reference field is “Defining Class”, and the description is “Defining class of the method”. It also specifies that the field refers to class entities (by using the builder of class entities as a parameter), and the field has no specific properties.

As shown in Figure 3.3, an object allocation event has three fields: a value field representing the size of memory allocated for the object, a reference field indicating the type of the object, and another reference field indicating the method where the object is allocated.

Because the allocating method of an object allocation event also has a reference to its own defining class, EVolve will automatically generate an indirect reference from the object allocation event to the class entity. This indirect reference represents the allocating class of the object allocation event, and the data source doesn't need to specify it.

Building the element definition of object allocation events is similar to building the definition of method entities, except that the object size field has a specific property (amount), as shown below:

```
String[] propertyAmount = {"amount"};

FieldDefinition objectSize = allocationBuilder.buildValueDefinition("Object Size",
    propertyAmount, "Size of memory allocated");
```

FieldDefinition objectType = allocationBuilder.buildReferenceDefinition("Object Type", classBuilder, null, "Type of the object");

FieldDefinition allocatingMethod = allocationBuilder.buildReferenceDefinition("Allocating Method", methodBuilder, null, "Method that creates the object");

ElementDefinition allocationDefinition = allocationBuilder.buildDefinition();

In EVolve, value fields don't have default properties, so the data source must specify the properties when building the field definition of value fields. On the other hand, reference fields have a default property which indicates that the fields are references to regular entities.

After the element definitions are built, element builders are ready to create elements.

3.2.2 Element Creation

As shown in Figure 3.2, the element creation phase also consists of three steps. To create an element, the data source first needs to inform the element builder to start building a new element. This also causes the element builder to initialize its data verifier.

`public void newEntity(String entityName)`
Starts building new entity.

`public void newEvent()`
Starts building new event.

Second, the data source can start to add the fields of the element to the element builder. To add a field, the data source must provide the corresponding field definition as the key.

`public void addValueField(FieldDefinition fieldKey, int value)`
Adds a value field.

`public void addReferenceField(FieldDefinition fieldKey, Entity reference)`
Adds a reference field.

After all the fields are added correctly (the element builder validates the data when the fields are added), the data source can get the element from the element builder.

`public Entity buildEntity()`
Builds the entity.

`public Event buildEvent()`
Builds the event.

In the above example, the data source first needs to create two class entities. Because they don't have any fields, the class entities can be created directly:

```
classBuilder.newEntity("ClassA");
Entity classA = classBuilder.buildEntity();

classBuilder.newEntity("ClassB");
Entity classB = classBuilder.buildEntity();
```

To create a method entity, the data source must first add a reference field by using the corresponding field definition as the key and specifying which entity the field refers to:

```
methodBuilder.newEntity("MethodA");
methodBuilder.addReferenceField(definingClass, classA);
Entity methodA = methodBuilder.buildEntity();
```

The data source can create an allocation event in a similar way (assume that the size of the object created is 16 bytes):

```
allocationBuilder.newEvent();
allocationBuilder.addValueField(objectSize, 16);
allocationBuilder.addReferenceField(objectType, classB);
allocationBuilder.addReferenceField(allocatingMethod, methodA);
Event allocation = allocationBuilder.buildEvent();
```

Through the `DataSource` interface, the data source can send these elements to the EVolve platform after they are built. Chapter 5 presents a detailed example showing how to use the element builders to implement the `DataSource` interface.

3.3 Data Storage

In general, a data source can send trace data to the software visualization system in two different ways. The first is that the trace data is sent only once and the visualization system stores all the data in memory. This is the most efficient way when visualizing a small amount of data. However, in some cases, especially when visualizing the run-time behaviour of programs, the trace data generated is often quite large, thus storing all the data in memory is infeasible.

The second way is just the opposite, and none of the data is stored in memory. Therefore, whenever the visualization system needs any information, it has to read it from the data source. Although this can solve the problem of visualizing large amount of data, it's much less efficient when users manipulate the data, because every operation (such as selecting a subset of the data) requires reading data from the data source.

In EVolve, these problems are solved by finding a balance point between these two extremes, and the strategy used in EVolve is based on the following:

First, when selecting subsets of the data, users are usually interested in selecting a group of entities. For example, users may want to visualize the invocation of a set of methods, or the allocation of certain types of objects. Therefore, most data manipulation functionalities only involve entities.

Second, as shown in Table 2.1 and Table 2.2, in most traces, the number of events is much larger than the number of entities. The difference between the two is usually over a factor of 100, and it's much higher in large traces.

Due to these factors, in EVolve, element definitions and entities are stored in memory and only need to be read from the data source once, while events are read from the data source when necessary (this means that the first five methods of the `DataSource` interface are called only once by the EVolve platform and the last two methods are called when necessary).

This strategy allows end users to manipulate large amount of data efficiently because EVolve only needs to read the trace data when generating the visual representation.

Chapter 4

Visualization Protocol

As shown in Figure 1.2, the visualization protocol of EVolve specifies the interaction between the visualizations and the EVolve platform, and it can be viewed as a cycle (Figure 4.1) consisting of four phases: creation, configuration, visualizing, and data manipulation.

Creation: during the creation phase, a visualization's definition is sent to the EVolve platform so that it can be configured in the next phase. The visualization also needs to be initialized, and one major step of the initialization is creating a canvas on which the data will be visualized.

Configuration: in the configuration phase, an end user selects the subject and dimensions of a visualization (i.e. mapping data fields to the dimensions), as well as specifies other parameters (such as sample size), causing the visualization to update itself.

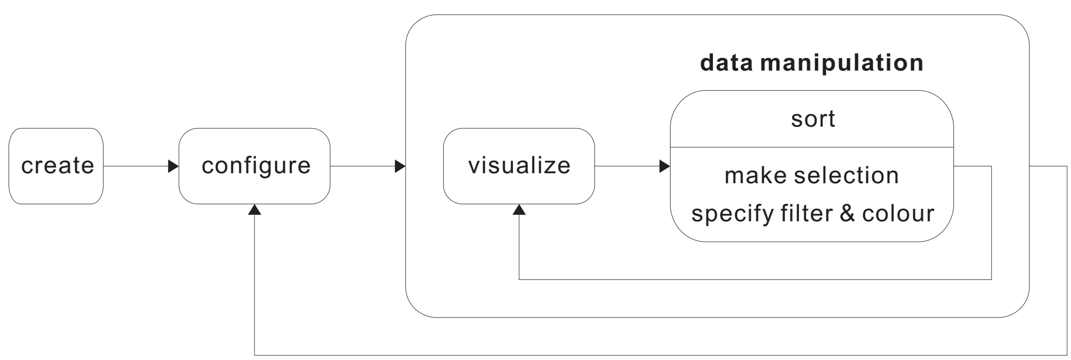

Figure 4.1: Visualization cycle of EVolve.

Visualizing: during the data processing phase, the visualization receives elements that belong to its subject from the visualization manager, and generates the visual representation of these elements. This time-consuming phase can be shared by different visualizations (i.e. the trace data is read only once and sent to various visualization).

Data manipulation: in EVolve, an end user can sort the content of a visualization according to a certain order, make selections in the visualization, and use these selections for filtering or coloring subsets of the data.

EVolve has a number of built-in visualizations, such as bar charts, hot spot graphs[7], prediction graphs, etc. End users can use EVolve as a stand-alone tool and create one or more of these built-in visualizations. However, EVolve is also designed to be extensible, so the visualization protocol makes it simple to add new kinds of visualizations.

In this chapter, Section 4.1 gives an overview of a typical end user interaction with EVolve, and the other sections describe how a visualization provider can build a new visualization by implementing the four phases of the visualization cycle.

4.1 End User Visualization

Given a data source, an end user can run EVolve as a stand-alone program and interactively create and modify one or more visualizations of the data. The end user can select from any data provided by the data source and can visualize using any visualization currently available.

To create and modify a visualization, the end user goes through the process shown in Figure 4.1. The following sections illustrate this process using a sequence of screen shots of EVolve (Figure 4.2 to Figure 4.9). This example shows four different visualizations of method invocations from the Volano benchmark[5].

4.1.1 Bar Chart Visualization

Assume that the end user first wants to use a bar chart to show which invoking locations are most active. To do so, the end user needs to indicate that he/she wants to create a bar chart by choosing the corresponding menu item (Figure 4.2) from the available visualizations; this corresponds to the box labeled `create` in Figure 4.1.

Second, the end user configures the bar chart by specifying which data should be visualized and which fields should be mapped to the axes (Figure 4.3), i.e. selecting the subject and dimensions of the bar chart (the configure box in Figure 4.1). The configuration dialog is generated according to the visualization definition and the element definitions, so that only valid choices are available in the combo-boxes.

In this case, the end user selects method invocations as the subject of the bar chart, and maps the location of the invocations to the Y-axis and the number of invocations to the X-axis. Note that the number of invocations on the X-axis is an amount, i.e. a value than can be summed to produce the total number of invocations. In EVolve, configuration is optional, and if no configuration is given by the end user then default choices are made by EVolve.

Until this point no data has actually been displayed. After completing the configuration, the end user indicates that the data should be visualized and the appropriate visualization is computed and displayed (the visualize box in Figure 4.1). The result is shown in Figure 4.4.

Note that 88 method invocation locations were each executed up to 479 times.[1] The Y-axis of the bar chart shows invocation locations sorted in lexical order, and the length of the bars on the X-axis indicates the number of times each location was active in the visualized program run. The bar chart therefore stresses the importance of an invocation location over the whole program run. Users can find the name of the method being invoked by placing their mouse over the bars in the chart.

On the first visualization, all of the data is displayed in default color (black). However, the end user may want to perform various data manipulations on the visualization (see the data manipulation box in Figure 4.1). He/she may select only parts of the data (filter), assign colors to different parts of the data, or change the sorting order on some axis.

For the bar chart example, the end user has selected the `com.volano` locations using the mouse (Figure 4.5), and then assigned the color of this part blue/black[2]. Similarly, he/she has also assigned the color of the `java` locations green/grey. After specifying the modifications the end user then indicates that the data should be re-visualized (i.e. going from the data manipulation box back to the visualize box in Figure 4.1). Now the bar chart is computed and displayed again, and the new color scheme is applied (Figure 4.6).

[1]To make this example a reasonable size, only 88 method invocation locations are shown here, the whole benchmark has significantly more invocation locations.

[2]The electronic version of the thesis includes color images.

This process of specifying data manipulations and re-visualizing may iterate until the end user is satisfied with the result. At any point the end user may decide to completely reconfigure the visualization, and return to the configure step where different subject or dimensions are assigned to the visualization (i.e. going back to the visualize box).

4.1.2 Hot Spot Visualization

From the bar chart, the end user can easily find the most active invoking locations. However, a bar chart cannot provide enough information about the run-time behaviour of the program, so the end user may also want to use a hot spot visualization.

Figure 4.7 shows a lexical hot spot graph of the same 88 method invocation locations. It is produced in a similar manner to the bar chart, so only the differences from the bar chart are mentioned here. The Y-axis still represents the invocation locations, and the X-axis indicates time as number of bytecodes executed since start of program (a coordinate) for a total of 1,322,582 executed bytecodes.

A hot spot graph shows when a particular invocation location is active. For example, from the visualization one can see that the `com.volano` method invocations (blue/black) do not start until about half way into the execution, whereas the first half of the code has only `java` method invocations (green/grey). Presumably this shows that the first phase of the program execution consists of program/JVM initialization and class loading, whereas the second half executes mostly `com.volano` code, with some calls to `java` libraries.

The end user can create another hot spot visualization, a temporal hot spot, as shown in Figure 4.8. The X-axis of this visualization remains the same as for the lexical hot spot, encoding time as number of bytecodes executed. The Y-axis shows the same 88 method invocation locations, but the locations are now sorted by the time they are first activated (temporal order). A temporal hot spot groups together invocations that execute together, emphasizing the phases of program execution. The blue/black[3] `com.volano` invocations therefore appear clustered together high on the Y-axis.

Actually, the visualizations used to generate these two hot spot graphs are identical, only the sorting schemes used on their Y-axes are different. EVolve provides two default sorting schemes (lexical and temporal), but is designed to facilitate painless integration of other sorting schemes.

[3]The coloring scheme is kept from the previous step.

4.1.3 Prediction Visualization

In order to facilitate understanding polymorphism, the end user can use a prediction visualization that displays the predictability of events (Figure 4.9). In this case, the prediction visualization shows predictor misses of virtual method invocations. The X-axis and Y-axis are identical to the temporal hot spot graph, but the visualization provides its own coloring scheme.

Method invocations appear in light blue/light grey when the invoked target method does not change within the time period visualized (i.e., the invocation is not polymorphic, in these cases, some sort of inline cache for virtual method calls would be expected to work well). On the contrary, method invocations appear in red/black when the invoked target method does change, representing a polymorphic invocation.

The name "prediction" for this visualization stems from the technique used to generate the colors: a simple last-value predictor guesses that an invocation location will invoke the exact same method as the last time it was executed. The blue/grey areas therefore indicate perfect prediction accuracy, the red/black areas show when the predictor guesses the wrong target method. More sophisticated and accurate predictors can be visualized by plugging them into the framework, and more accurate predictors should reduce the amount of red/black points in the graph.

The visualization in the example shows that the `com.volano` invocation locations exhibit a higher degree of polymorphism than the `java` methods. For example, in the startup phase, most invocation locations never change method targets.

The following sections use the hot spot visualization as an example to describe the implementation of the visualization cycle, and the prediction visualization is used as a case study in Chapter 5.

4.2 Creation

Compared to the data protocol, the visualization protocol of EVolve is much more complicated, because it not only involves data transmission and processing, but also includes end user interaction and the user-interface. Therefore, if the visualization protocol is simply specified by an interface, visualization providers will have a lot of work to do to implement a new visualization in EVolve.

Fortunately, most of the work involved in implementing a new visualization is general to any kind of visualization. Therefore, EVolve provides a `Visualization` class which implements all the functions that are general. By extending the `Visualization` class, visualization providers only need to implement functions that are specific to their new visualizations.

This section and the following sections describe how to build a new visualizations in EVolve using the `Visualization` class. The hot spot visualization described in the previous section is used as an example.

As shown in Figure 4.1, the first phase in the visualization cycle is the creation phase. During this phase, the visualization is created and initialized, its definition is sent to the EVolve platform and the canvas where the data will be visualized is created.

4.2.1 Visualization Factory

End users often need to use several visualizations of the same type at the same time (in Figure 4.9, two hot spot visualizations are used). Because of this, EVolve uses the factory pattern[8] to create visualizations.

Figure 4.10 shows the relationship between visualization and visualization factory. According to this relationship, in order to build a new type of visualization in EVolve, a visualization provider not only needs to build a visualization by extending the `Visualization` class, but also has to build a visualization factory by extending the `VisualizationFactory` class.

The `VisualizationFactory` class has three abstract methods that must be implemented:

`public String getName()`
: Gets the name of the visualization.

`protected VisualizationDefinition createDefinition()`
: Creates visualization definition.

`public Visualization createVisualization()`
: Creates a visualization.

For example, here's the visualization factory for hot spot visualization:

```
public String getName() {
   return "Hot Spot Visualization";
}
```

```
protected VisualizationDefinition createDefinition() {
    DimensionDefinition[] dimensionDefinition = new DimensionDefinition[2];

    /* only fields that are coordinates can be mapped to the X-axis */
    dimensionDefinition[0] = new DimensionDefinition("X-axis", "coordinate");

    /* only fields that are references can be mapped to the Y-axis */
    dimensionDefinition[1] = new DimensionDefinition("Y-axis", "reference");

    return new VisualizationDefinition(dimensionDefinition);
}

public Visualization createVisualization() {
    return new HotSpotVisualization();
}
```

The visualization definition defines the dimensions of a visualization and the property of these dimensions. A hot spot visualization has two dimensions, its X-axis and Y-axis. The property of the dimensions determines that only value fields that are coordinates can by mapped to the X-axis, and any kinds of reference field can be mapped to the Y-axis.

4.2.2 Creating Visualization

Two major fields of the `Visualization` class are the panel (the canvas where the data elements are visualized) and the dimensions, as shown in Figure 4.11. During the creation phase, these two fields must be initialized.

In order to extend the `Visualization` class, a new visualization must implement nine abstract methods that are specified by the `Visualization` class. These methods belong to the difference phases of the visualization cycle. The following two methods belong to the creation phase and create the panel and dimensions of the visualization, respectively:

`protected JPanel createPanel()`
: Creates the panel.

`protected Dimension[] createDimension()`

Creates the dimensions.

The following piece of code creates the panel and the dimensions for hot spot visualization:

```
private ValueDimension xAxis;
private ReferenceDimension yAxis;

protected JPanel createPanel() {
    AxesPanel returnVal = new AxesPanel();
    return returnVal;
}

protected Dimension[] createDimension() {
    xAxis = new ValueDimension();
    yAxis = new ReferenceDimension();

    Dimension[] returnVal = new Dimension[2];
    returnVal[0] = xAxis;
    returnVal[1] = yAxis;
    return returnVal;
}
```

Note that the hot spot visualization creates an `AxesPanel` instead of a `JPanel`. Because many visualizations use two axes (such as bar chart, plot chart, hot spot visualization and etc.), EVolve provides `AxesPanel` to simplify the implementation of these visualizations. The `AxesPanel` class extends the `JPanel` and the following sections describe how to use it.

The hot spot visualization creates a value dimension (X-axis) and a reference dimension (Y-axis). In EVolve, the dimensions created by the `createDimension` method must conform to the visualization definition created by the `createDefinition` method of the visualization factory.

4.3 Configuration

In EVolve, when the end user needs to configure a visualization, a configuration dialog (Figure 4.12) is popped up, where the end user can change the title of the visualization, select the subject and dimensions of the visualization, and choose other parameters.

The `Visualization` class generates the configuration dialog according to the visualization definition, so that only valid choices are available in the combo-boxes. For example, because only coordinates can be mapped to the X-axis of the hot spot visualization, the combo-box that represents the X-axis only provides choices that are coordinates.

Because the configuration of the title, subject and dimensions is general to all kinds of visualizations, it is implemented by the `Visualization` class. Therefore, visualization providers only need to implement the following two methods to support the configuration phase:

`protected JPanel createConfigurationPanel()`
: Creates a panel for parameters input (returns `null` if the visualization doesn't need other parameters, such as bar chart).

`protected void updateConfiguration()`
: Updates the visualization according to the configuration.

For hot spot visualization, an end user can specify the sample size (interval) in the configuration phase, and the code below shows how to add items to the configuration dialog:

```
private int interval;
private JTextField textInterval

protected JPanel createConfigurationPanel() {
    interval = 1000;
    textInterval = new JTextField("1000");

    JPanel returnVal = new JPanel();
    returnVal.add(new JLabel("Interval:"));
    returnVal.add(textInterval);
    return returnVal;
}
```

When an end user applies the configuration, the `updateConfiguration()` method is called, and the hot spot visualization needs to get the interval value from the configuration dialog and update the axes panel:

```
protected void updateConfiguration() {
    /* gets the interval from the dialog */
    interval = Integer.parseInt(textInterval.getText());

    /* gets the name of the axes from the dimensions */
    ((AxesPanel)panel).setName(xAxis.getName(), yAxis.getName());

    /* draws an empty graph */
    ((AxesPanel)panel).setImage(null);
    panel.repaint();
}
```

After the configuration phase, an empty hot spot visualization is ready for visualizing elements in the next phase (Figure 4.13).

4.4 Visualizing

The visualizing phase of the visualization protocol consists of three steps. First, before starting to process the data trace, most visualizations need to do some preparation (such as initializing some variables). After the preparation, the visualization starts to receive and process the elements, and when all the elements are received, the visualization can generate the visual representation of these elements. Figure 4.14 shows these three steps.

Correspondingly, the following methods of the `Visualization` class belong to the visualizing phase:

`public void preVisualize()`
: Prepares for receiving elements.

`protected void receiveElement(Element element)`
: Receives an element.

`public void visualize()`
: Generates the visual representation of the elements.

For a hot spot visualization, each time when an element is received, it needs to draw a point on the axes panel. An AxesPanel uses a BufferedImage[4] as the content of the panel and EVolve provides an AutoImage to facilitate generating the BufferedImage. So in the first step, a hot spot visualization has to create an AutoImage:

```
private AutoImage image;

public void preVisualize() {
    image = new AutoImage();
}
```

When an element is received, a visualization can use its dimensions to get the corresponding fields from the element. If the field is a value, the dimension gets the value from the element; if the field is an entity, the dimension returns the index of the entity. The following code shows how a hot spot visualization draws a corresponding point of an element (the X-position is determined by the value on the X-axis and the interval, the Y-position is the index of the entity on the Y-axis:

```
public void receiveElement(Element element) {
    image.setColor(xAxis.getField(element) / interval, yAxis.getField(element),
        EVolve.getColor());
}
```

Note that the EVolve.getColor() method returns the color of the current element determined by the filter of EVolve.

After all the elements are received, a hot spot visualization only needs to generate the BufferedImage from the AutoImage and update the content of the AxesPanel:

```
public void visualize() {
    ((AxesPanel)panel).setImage(image.getImage());
    panel.repaint();
}
```

Figure 4.15 shows the hot spot visualization after the visualizing phase. From the graph, an end user can know when a particular invocation location is active. For example, it is obvious that the execution of the program consists of two major phases,

[4]BufferedImage is a class included in the java.awt.image package.

and most invoking locations that are active during the first phase (the left half) are no longer active in the second phase (the right half).

4.5 Data Manipulation

EVolve provides three types of data manipulation: filtering, coloring, and sorting. Filtering and coloring are mainly accomplished by the filter of the EVolve platform, and visualizations only need to allow end users to make selections among the data. EVolve also simplifies the process of sorting and provides two default sorting schemes: lexical and temporal.

In EVolve, sorting and making selections only involve entities, so in order to implement the data manipulation phase, a visualization provider has to understand the `ReferenceDimension` class, because reference dimensions represent entities.

4.5.1 Reference Dimension

When an entity is created by an element builder, its id is assigned automatically. However, when implementing a visualization (especially the data manipulation phase), it is complicated to use the entity id directly. Therefore, for every entity, a reference dimension generates a corresponding entity index and sorted index to simplify the implementation of the data manipulation phase, as shown in Figure 4.16.

To generate the entity indices, a reference dimension filters out all the entities that are unnecessary for a visualization. For example, a data trace may contain 100 methods, but only 10 of them create objects, so for a visualization that shows objects created by methods, only these 10 method entities are useful while the other 90 are unnecessary.

Sorted indices indicate the sorted order of the entities under a certain sorting scheme. Unlike entity indices, which are generated during the visualizing phase, sorted indices are generated during the data manipulation phase when the end user applies sorting schemes to the reference dimension.

The following methods of the `ReferenceDimension` class are related to the conversion of entity id:

```
public int getMaxEntityNumber()
```

Gets the maximum number of entities that belong to this dimension (the number of entity ids). For the previous example, this will return 100. Note that this method can be called in the visualizing and data manipulation phase.

`public int getEntityNumber()`
: Gets the number of entities that are useful (the number of entity indices). For the previous example, this will return 10. Note that this method should be called after all the elements are received, i.e. in the data manipulation phase and the last step of the visualizing phase.

`public Entity getEntity(int entityIndex)`
: Gets the corresponding entity.

`public int getSortedIndex(int entityIndex)`
: Gets the sorted index.

4.5.2 Making Selections

Different types of visualization have different ways for making selections. For example, an end user can select a group of rows in a table, or select several bars in a bar chart, or select an area in a hot spot visualization (Figure 4.17).

In EVolve, the following method is called when an end user needs to make a selection:

`public void makeSelection()`
: Makes a selection of entities.

To implement this method, a hot spot visualization first needs to know which entities are selected (i.e. which entities are enclosed in the selecting box of Figure 4.17), and then sends the *sorted index* of these entities to the reference dimension to make the selection. For example, a hot spot visualization can get the range of the selection box from the `AxesPanel` and use its Y-axis to make the selection[5]:

```
public void makeSelection() {
    /* gets the sorted index of the entity that corresponds to the bottom edge */
    int y1 = ((AxesPanel)panel).getEndY();
```

[5]Note that this example only uses the top and bottom boundaries of the selecting box, and ignores the left and right boundaries, Chapter 5 gives an example that also uses the left and right boundaries.

```
    /* gets the sorted index of the entity that corresponds to the top edge */
    int y2 = ((AxesPanel)panel).getStartY();

    /* adds all the entities between the two edges to the selection */
    int[] selection = new int[y2 - y1 + 1];
    for (int i = y1; i <= y2; i++) {
        selection[i - y1] = i;
    }

    yAxis.makeSelection(selection);
}
```

4.5.3 Sorting

When the end user applies a certain sorting scheme to a reference dimension, the dimension generates new sorted index of the entities and the following method is called:

`public void sort()`
  Sorts the content of the visualization.

Normally, a visualization needs to sort the content of the visualization according to the sorted index of the entities (such as rearranging the rows in a table). However, the `AutoImage` used by the hot spot visualization can sort the content of the image automatically:

```
public void sort() {
    /* sorts the Y-axis of the image */
    ((AxesPanel)panel).setImage(image.getSortedImage(null, yAxis).getImage());
    panel.repaint();
}
```

The `AutoImage` can get the sorted indices from reference dimensions and generate sorted image accordingly.

4.6 Summary

This chapter describes the visualization protocol of EVolve and how to build a visualization by extending the `Visualization` class. To summarize, here are the nine methods that need to be implemented by the visualization provider:

```
protected JPanel createPanel()

protected Dimension[] createDimension()

protected JPanel createConfigurationPanel()

protected void updateConfiguration()

public void preVisualize()

protected void receiveElement(Element element)

public void visualize()

public void makeSelection()

public void sort()
```

The next chapter gives a complete example showing how to implement a more complicated visualization in EVolve.

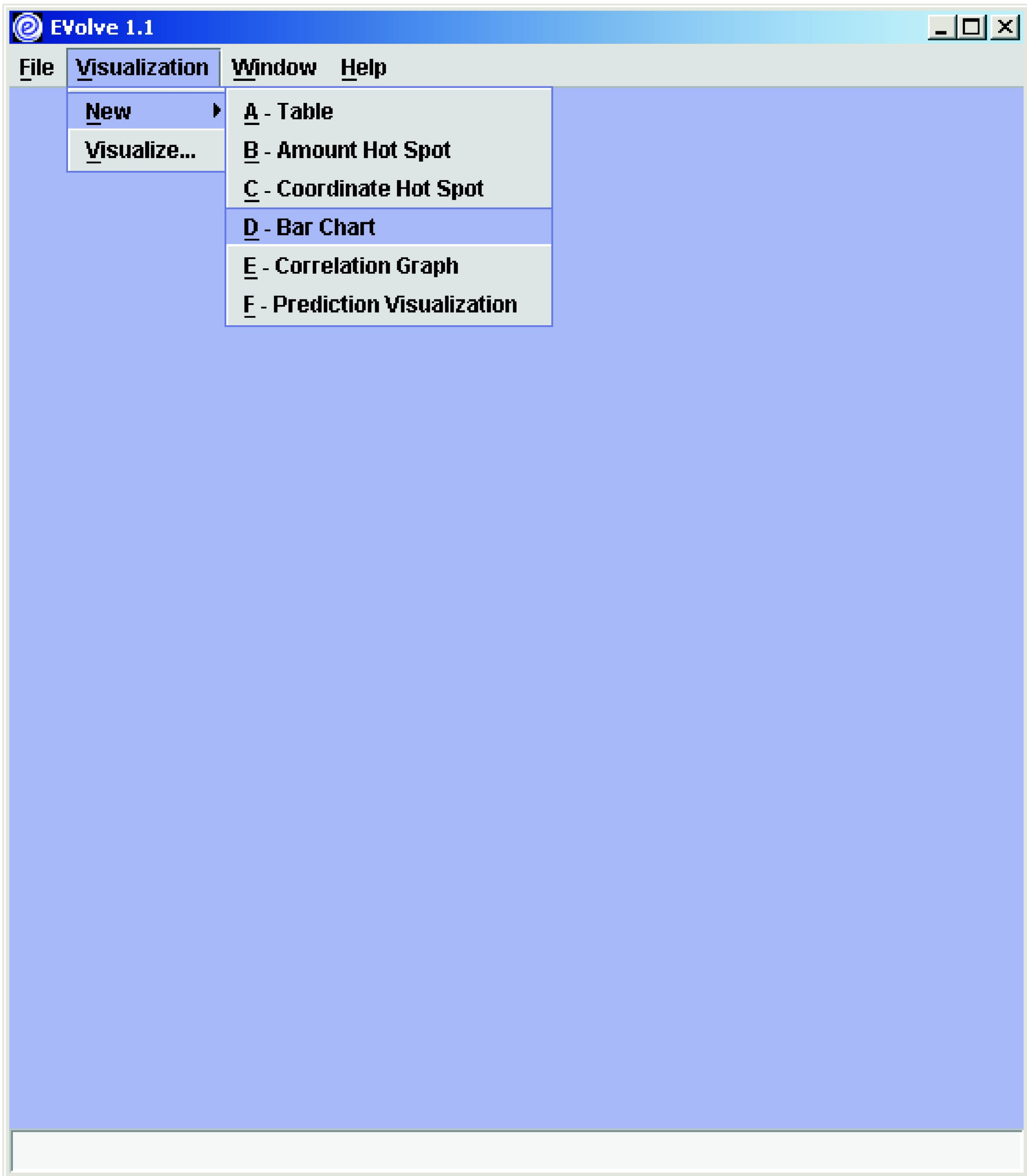

Figure 4.2: Creating a bar chart.

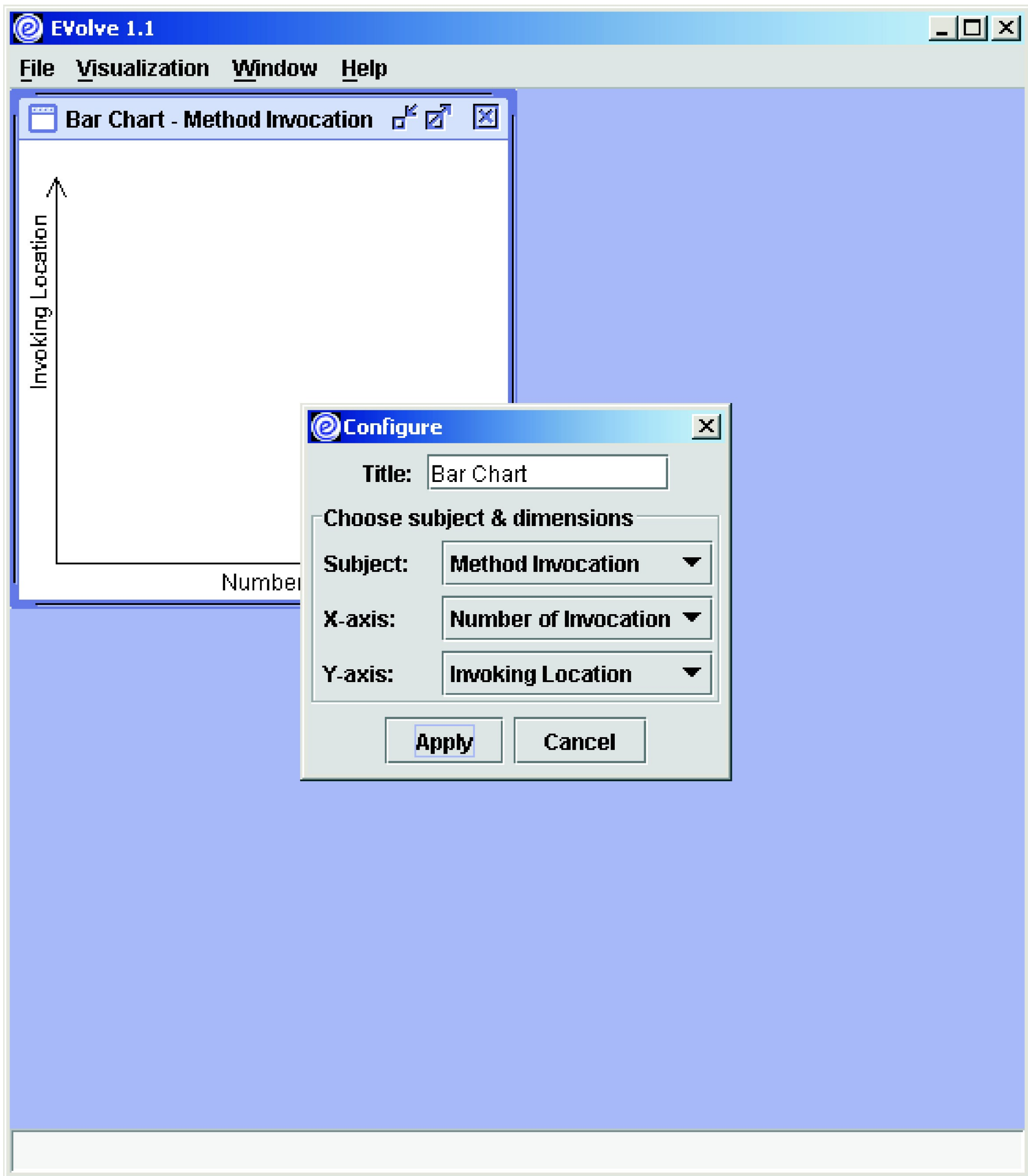

Figure 4.3: Configuring the bar chart.

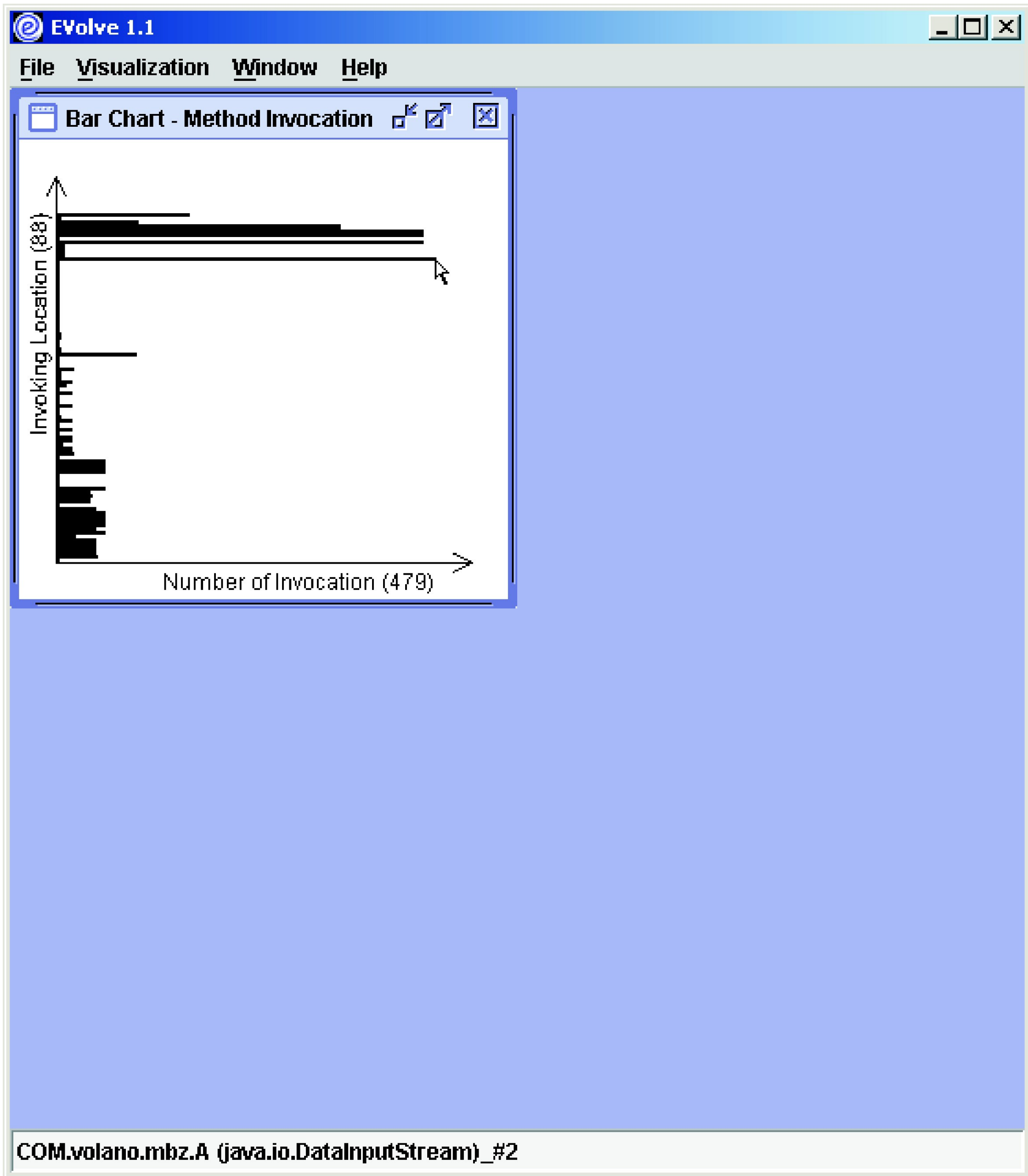

Figure 4.4: Bar chart without color.

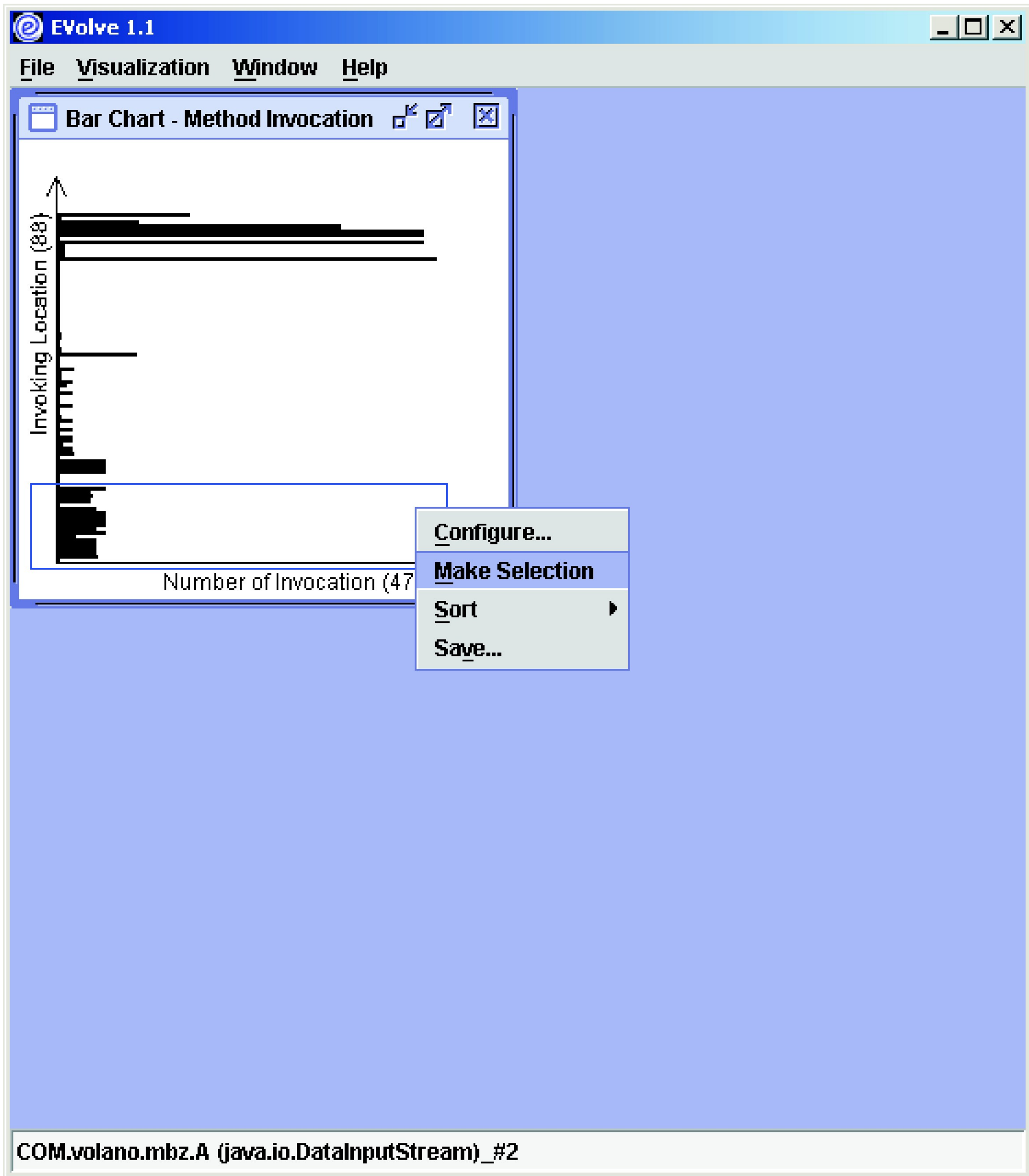

Figure 4.5: Making selections on the bar chart.

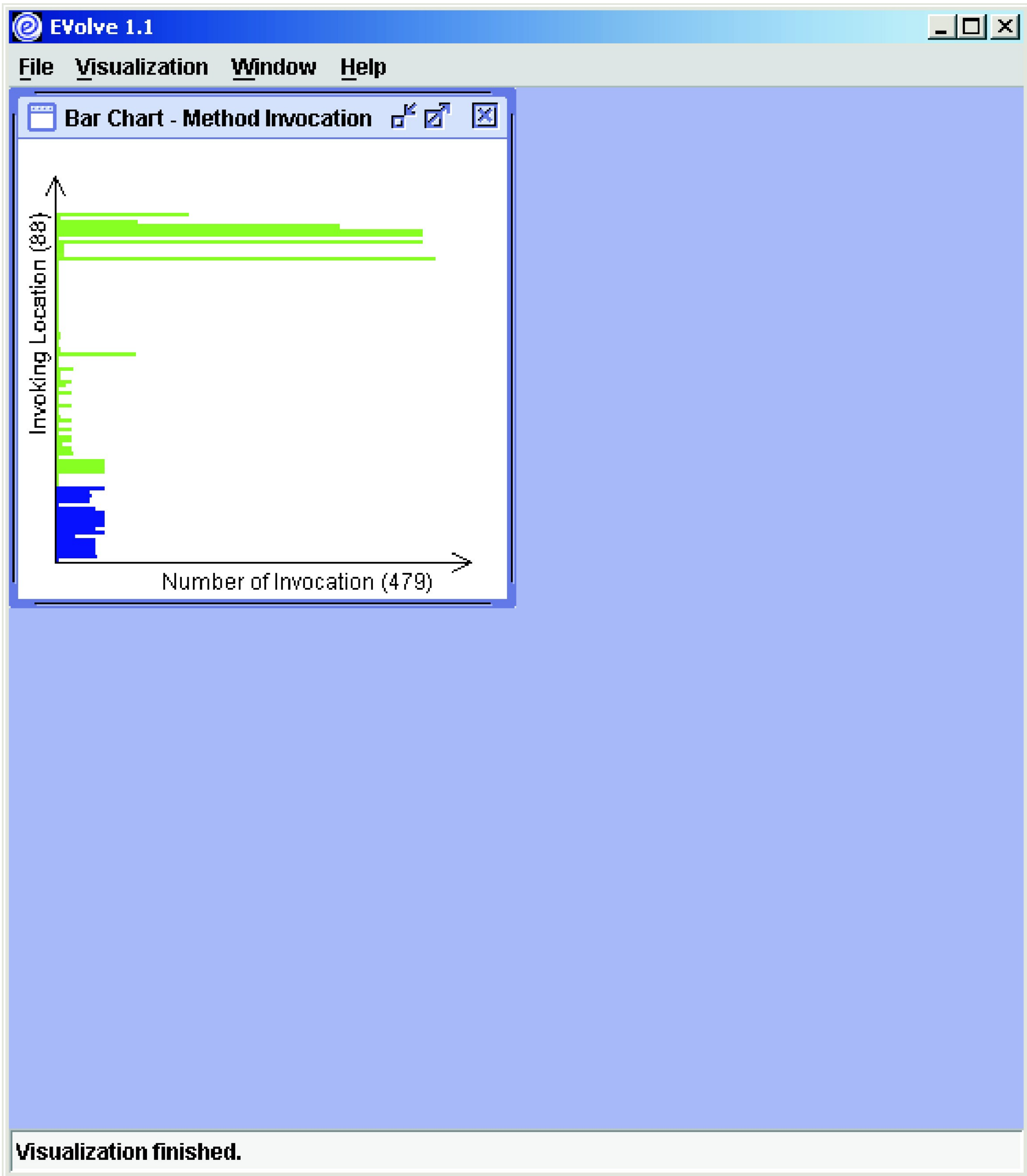

Figure 4.6: Bar chart with color.

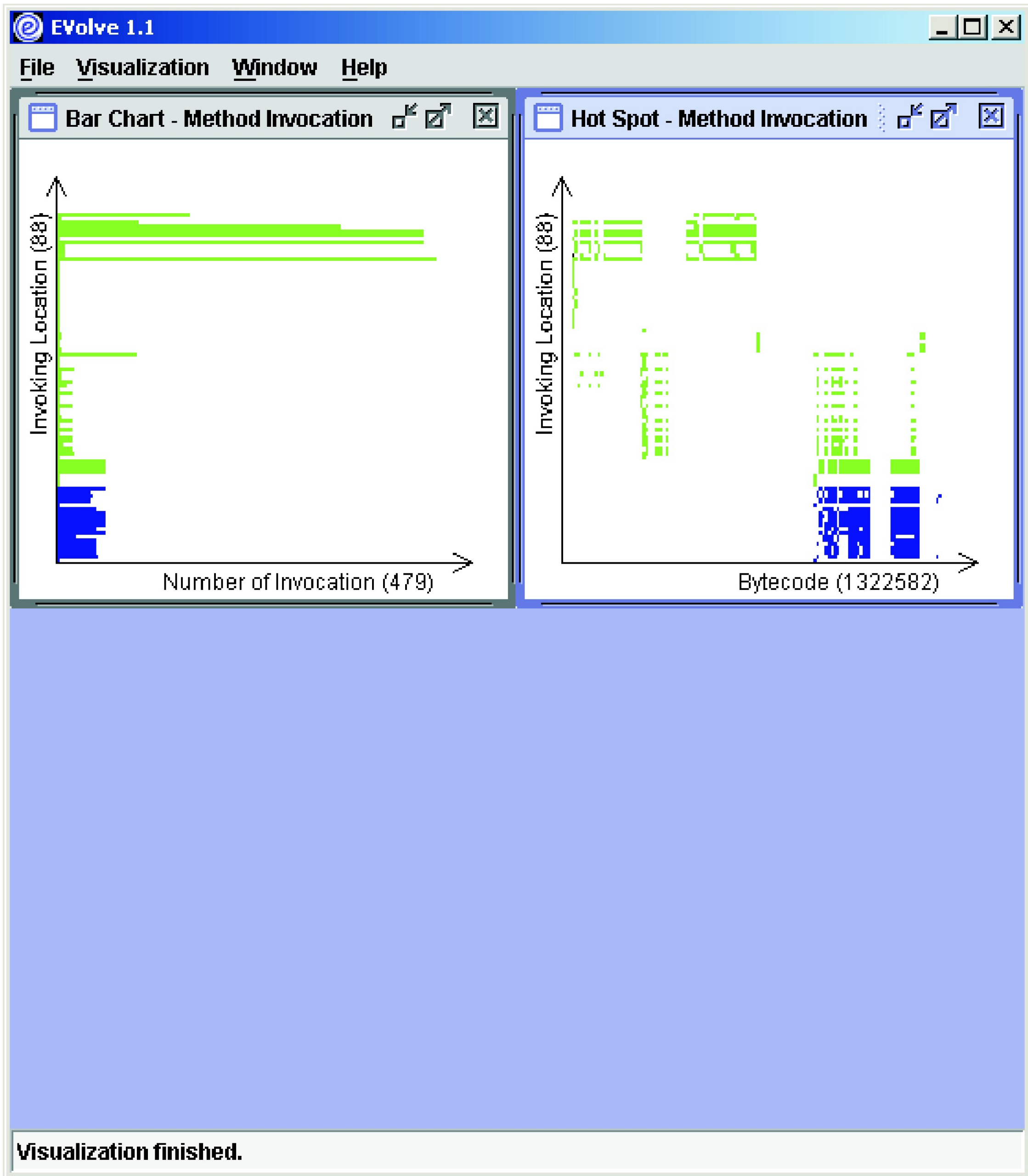

Figure 4.7: Lexical hot spot.

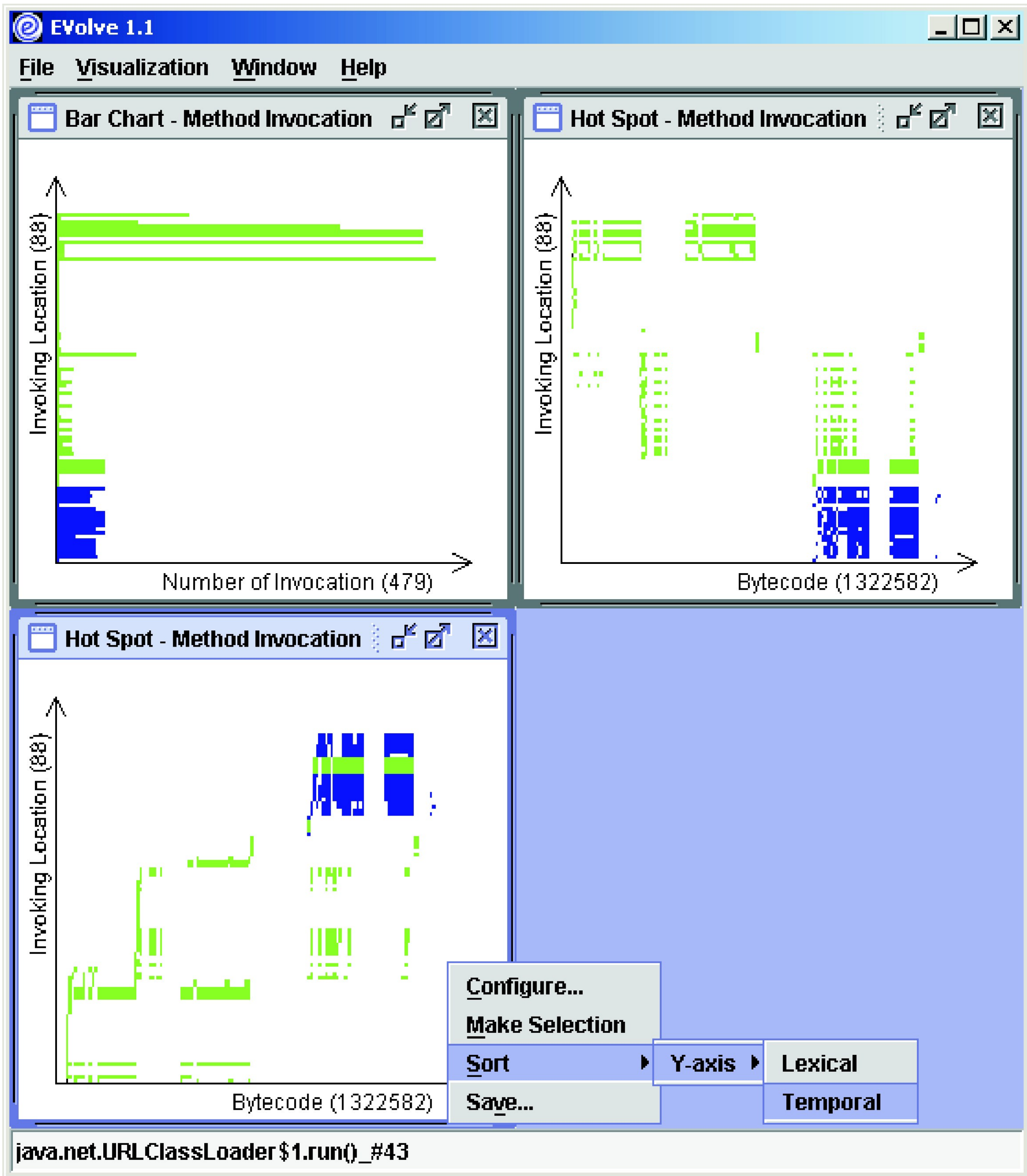

Figure 4.8: Temporal hot spot.

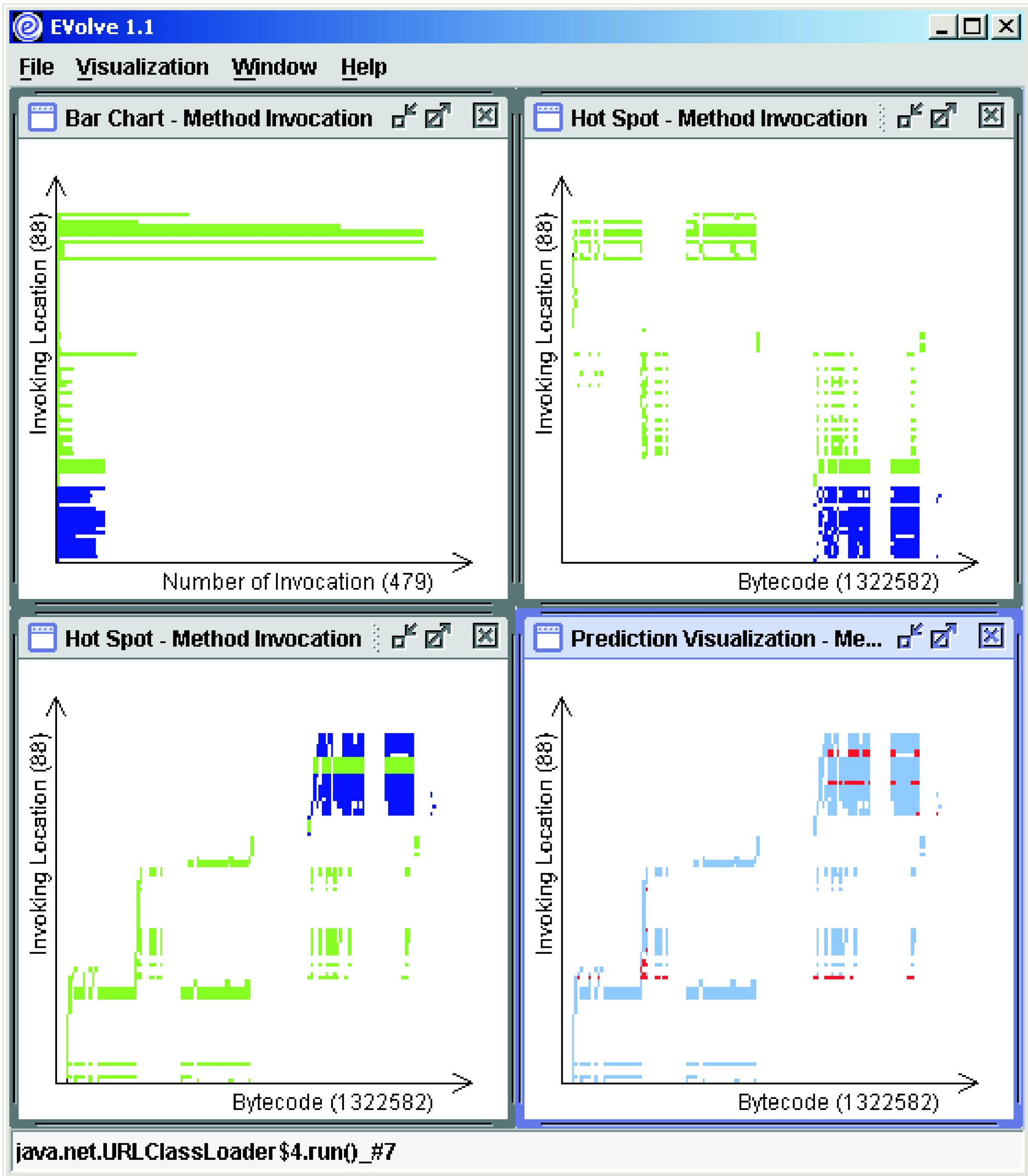

Figure 4.9: Screen shot of EVolve.

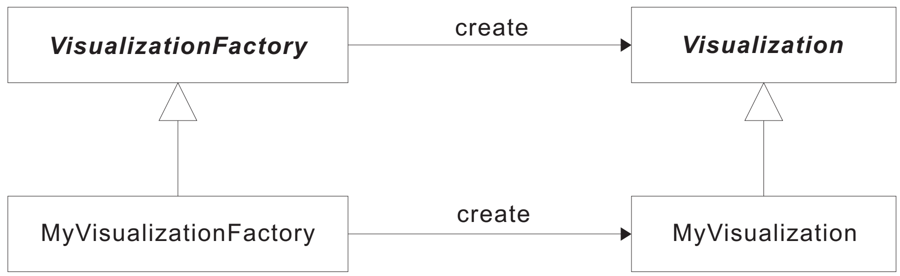

Figure 4.10: Relationship between visualization and visualization factory.

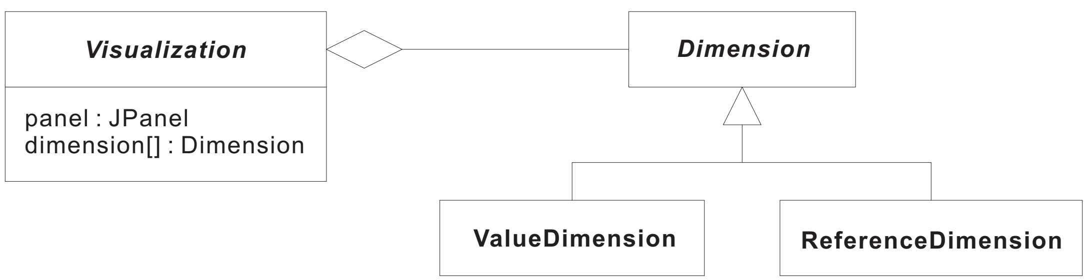

Figure 4.11: Class diagram of visualization.

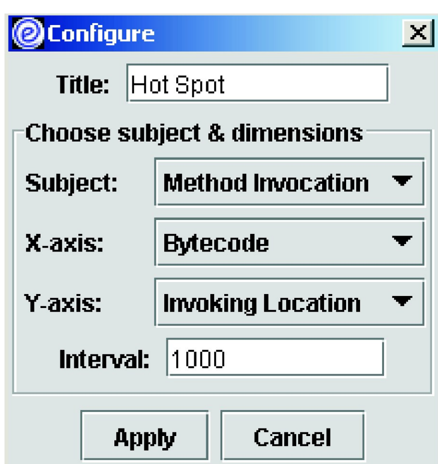

Figure 4.12: Configuration dialog.

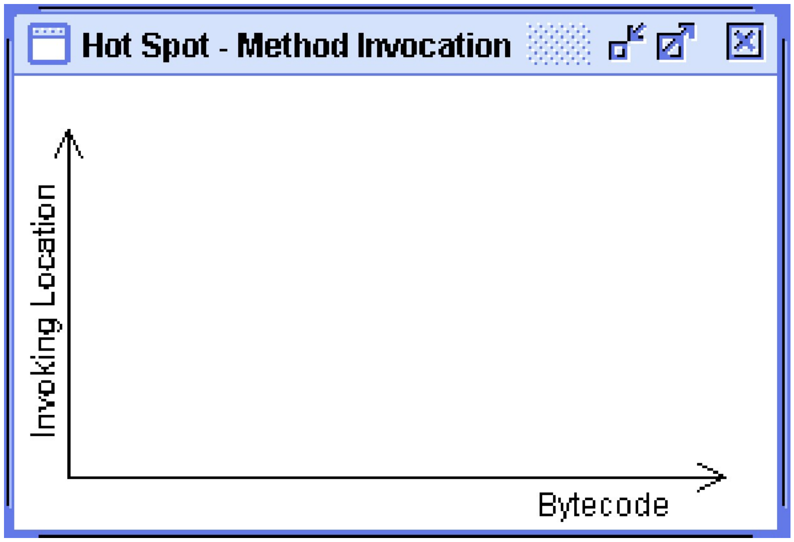

Figure 4.13: An empty hot spot visualization.

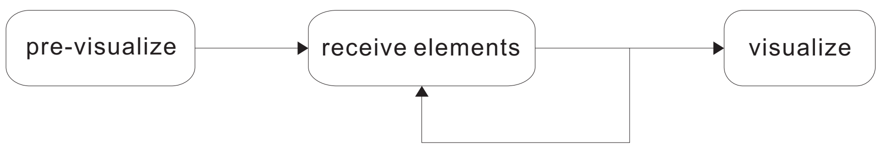

Figure 4.14: Visualizing phase of the visualization protocol.

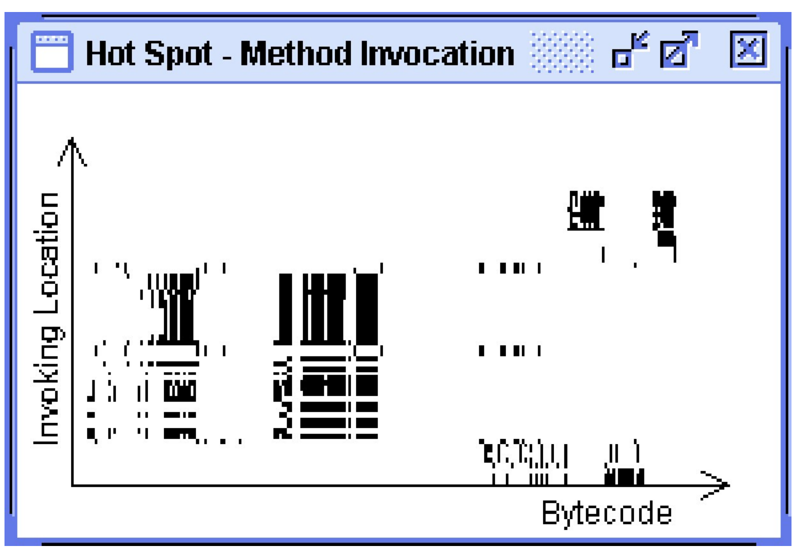

Figure 4.15: A hot spot visualization.

Figure 4.16: Conversion of entity id.

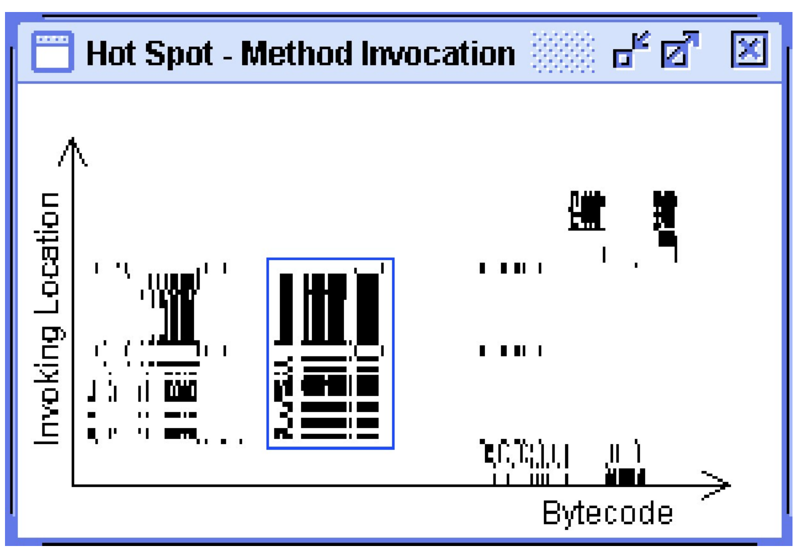

Figure 4.17: Making selections on hot spot visualization.

Chapter 5

Case Study: Visualizing Polymorphism

Chapter 3 and Chapter 4 described the data protocol and visualization protocol of EVolve, and how to implement data sources and visualizations. To demonstrate how to use EVolve solving real problems, this chapter presents a step-by-step example to serve as a case study.

The goal of this case study is to visualize polymorphism, and it serves the following purpose:

- The visualization used in this case study was implemented after the whole framework was built. However, adding the new visualization to the framework didn't require any modification of the framework, and this validates the extensibility of EVolve.
- It only took about two hours to implement the new visualization, and this demonstrates that extending EVolve is easy.
- Polymorphism is an important characteristic of object-oriented programs, and the visualization provides a straightforward way to understand it.

Section 5.1 first discusses the problem needs to be solved in the case study, and the requirement of the data source and the visualization. Then Section 5.2 and 5.3, respectively, describe the implementation of the data source and the visualization. Finally, Section 5.4 shows how to integrate the data source and the visualization into EVolve.

5.1 Problem Description

An interesting and important aspect of the run-time behavior of object-oriented programs is polymorphism, especially for system optimizations (such as in-lining). So the goal of this case study is to provide a visualization technique to facilitate understanding and comparing different prediction strategies.

To do so, the first step is to generate a data trace that contains information about method invocations. For each method invocation event, the trace data has to provide the invoking location and the actual method invoked. There are various ways to obtain the trace data, such as instrumenting the source code, using an instrumented JVM (Java Virtual Machine), or using the JVMPI (Java Virtual Machine Profiler Interface)[25]. The discussion of how to create the trace data is beyond the scope of this thesis and in this chapter, it is assumed that the trace data is generated and stored in a file using a certain format.

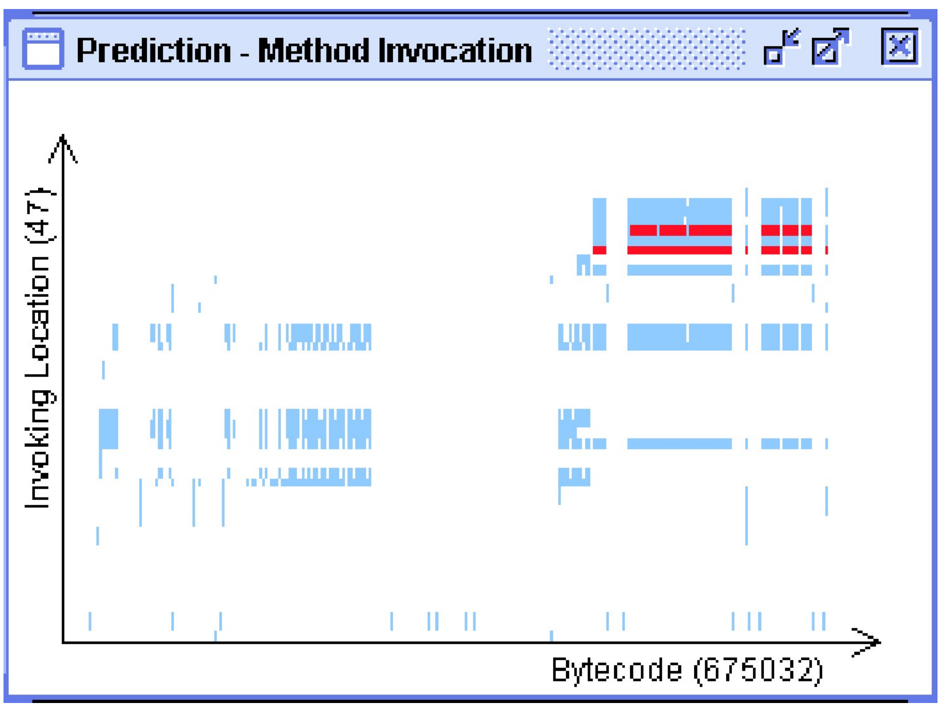

Figure 5.1: Prediction visualization.

After the trace data is generated, the data source must be able to read the trace data and create the corresponding elements: methods, invoking locations, and method invocations. Obviously, methods and invoking locations are entities, while method invocations are events.

The visualization used in this example is similar to the hot spot visualization described in Chapter 4, except that it uses a different coloring scheme so that the color now indicates whether the method invocations are polymorphic or not. As shown in Figure 5.1, red/black[1] indicates invocations that are polymorphic, while light blue/light grey indicates invocations that are monomorphic.

5.2 Implementing the Data Source

5.2.1 Format of the Data Trace

For the case study, assume that the data trace is stored as a pure text file using the format shown in Table 5.1.

Method Invoked	`M: methodName methodId`
Invoking Location	`L: locationName locationId`
Invocation	`I: bytecode locationId methodId`

Table 5.1: Format of the data trace.

For example, the following piece of information shows that when executing the 151347th bytecode, method `java.lang.Math.max(int,int)` is invoked by method `java.util.jar.Manifest.read()`, and the offset of the invoking location within the method is 18.

```
M: java.lang.Math.max(int,int) 115
L: java.util.jar.Manifest.read()#18 238
I: 151347 238 115
```

5.2.2 Data Source

The data source of the case study needs to parse the data trace according to the format described in Table 5.1 and generate the corresponding elements. Because there are three types of element: method, invocation location, and invocation, the data source requires three element builders, as shown in Figure 5.2. Furthermore, in

[1]The electronic version of this thesis includes color images.

order to create the invocation events, the data source also needs to keep track of the entities created by the entity builders. Therefore, it uses the `TreeMap` provided in the `java.util` package to store the entities and their ids (the ids used in the data trace, not the internal EVolve entity ids).

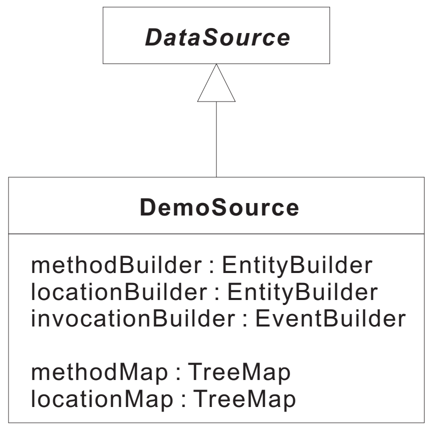

Figure 5.2: Class diagram of the data source of the case study.

5.2.3 Initialization

During the initialization, the data source needs to open a trace data file that is chosen by the end user using the `JFileChooser` dialog provided in the `javax.swing` package. If the file cannot be opened, or the end user canceled the action, the data source throws an `DataProcessingException` defined by EVolve to indicate that there's an exception occurred when processing the data.

```
import java.io.*;
import java.util.*;
import javax.swing.*;

public class DemoSource implements DataSource {
    private RandomAccessFile file;

    private EntityBuilder methodBuilder;
    private EntityBuilder locationBuilder;

    private EventBuilder invocationBuilder;
    private FieldDefinition invocationBytecode;
```

```
    private FieldDefinition invocationLocation;
    private FieldDefinition invocationMethod;

    private TreeMap methodMap;
    private TreeMap locationMap;

    public void init() throws DataProcessingException {
        JFileChooser fc = new JFileChooser();

        /* opens the file chooser */
        if (fc.showOpenDialog(EVolve.getFrame()) ==
            JFileChooser.APPROVE_OPTION) {
            try {
                /* opens the file that the end user chose */
                file = new RandomAccessFile(fc.getSelectedFile(), "r");
            } catch (IOException e) {
                throw new DataProcessingException("File loading failed.");
            }
        } else {
            /* the end user cancels the initialization */
            throw new DataProcessingException("File loading canceled.");
        }
    }
}
```

5.2.4 Definition Building

To send the element definitions to the EVolve platform, the data source first needs to create the element builders and build the element definitions. The element definitions are stored in an array and are then sent to the EVolve platform one by one.

```
private ElementDefinition[] definition;
private int definitionCounter;

public void startBuildDefinition() {
    definition = new ElementDefinition[3];

    methodBuilder = new EntityBuilder("Method", "Method");
```

```
    definition[0] = methodBuilder.buildDefinition();

    locationBuilder = new EntityBuilder("Location", "Location in method");
    definition[1] = locationBuilder.buildDefinition();

    invocationBuilder = new EventBuilder("Invocation", "Method invocation");

    /* a value field that represents the bytecode sequence */
    String[] propertyCoordinate = {"coordinate"};
    invocationBytecode = invocationBuilder.buildValueDefinition("Bytecode",
        propertyCoordinate, "Bytecode number");

    /* a reference field that represents the method invoked */
    invocationMethod = invocationBuilder.buildReferenceDefinition("Method",
        methodBuilder, null, "Method that is invoked");

    /* a reference field that represents the invoking location */
    invocationLocation = invocationBuilder.buildReferenceDefinition("Location",
        locationBuilder, null, "Invoking location");

    definition[2] = invocationBuilder.buildDefinition();

    /* initializes the counter of element definitions */
    definitionCounter = -1;
}

public ElementDefinition getNextDefinition() {
    /* sends the element definitions to the EVolve platform */
    definitionCounter++;

    if (definitionCounter < definition.length) {
        /* sends an element definition */
        return definition[definitionCounter];
    } else {
        /* all the element definitions have been sent */
        return null;
    }
}
```

5.2.5 Entity Building

To facilitate parsing the data trace, the data source needs a `getSub(String line, int part)` method[2] that returns a certain part of the line. For example, if the current line is `"M: java.lang.Math.max(int,int) 115"`, the first part of the line is the name of the method, and the second is the id of the method.

When an entity is built, it also needs to be stored in a `TreeMap` so that it can be used later on when building the events.

```
public void startBuildEntity() throws DataProcessingException {
    try {
        /* starts reading the data trace file from the beginning */
        file.seek(0);
        methodMap = new TreeMap();
        locationMap = new TreeMap();
    } catch (IOException e) {
        throw new DataProcessingException("File processing failed.");
    }
}

public Entity getNextEntity() throws DataProcessingException {
    try {
        /* reads a line from the data trace file */
        String line = file.readLine();
        Entity returnVal = null;

        while ((returnVal == null) && (line != null)) {
            char ch = line.charAt(0);
            if (ch == 'M') {
                /* uses the method name to create a method entity */
                methodBuilder.newEntity(getSub(line, 1));
                returnVal = methodBuilder.buildEntity();

                /* stores the method entity in the tree map using its id as key */
                methodMap.put(getSub(line, 2), returnVal);
            } else if (ch == 'L') {
                /* builds a location entity, same as building a method entity */
                locationBuilder.newEntity(getSub(line, 1));
```

[2] Implementation of this method is ignored here.

```
            returnVal = locationBuilder.buildEntity();
            locationMap.put(getSub(line, 2), returnVal);
        }

        if (returnVal == null) {
            /* reads another line if the current line represents an invocation event */
            line = file.readLine();
        }
    }

    return returnVal;
} catch (IOException e) {
    throw new DataProcessingException("File processing failed.");
}
}
```

5.2.6 Event Building

Building the invocation events is similar to building the entities, except that every invocation event has a value field indicating the bytecode number, and two reference fields, one indicating the method invoked, the other indicating the invoking location.

```
public void startBuildEvent() throws DataProcessingException {
    try {
        /* starts reading the data trace file from the beginning */
        file.seek(0);
    } catch (IOException e) {
        throw new DataProcessingException("File processing failed.");
    }
}

public Entity getNextEvent() throws DataProcessingException {
    try {
        /* reads a line from the data trace file */
        String line = file.readLine();
        Event returnVal = null;

        while ((returnVal == null) && (line != null)) {
```

```
            char ch = line.charAt(0);
            if (ch == 'I') {
                /* starts the create a new invocation event */
                invocationBuilder.newEvent();

                /* gets the bytecode number from the line and adds a value field */
                invocationBuilder.addValueField(invocationBytecode,
                    Integer.parseInt(getSub(line, 1)));

                /* used the location id to get location entity
                    and adds a reference field */
                invocationBuilder.addReferenceField(invocationLocation,
                    (Entity)(locationMap.get(getSub(line, 2))));

                /* used the method id to get method entity
                    and adds a reference field */
                invocationBuilder.addReferenceField(invocationMethod,
                    (Entity)(methodMap.get(getSub(line, 3))));

                /* builds the invocation event */
                returnVal = invocationBuilder.buildEvent();
            }

            if (returnVal == null) {
                /* reads another line if the current line doesn't represents
                    an invocation event */
                line = file.readLine();
            }
        }

        return returnVal;
    } catch (IOException e) {
        throw new DataProcessingException("File processing failed.");
    }
}
```

5.3 Implementing the Prediction Visualization

5.3.1 Predictor

One of the design goals of the prediction visualization is to allow users compare different kinds of prediction strategies. Therefore, the prediction visualization should be able to use various "plug-in" predictors. This is similar to the way that various visualizations are "plugged-in" to EVolve, so the predictors are also created using the factory pattern as shown in Figure 5.3.

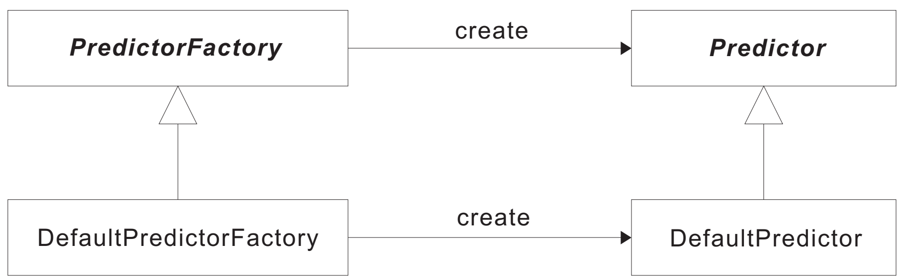

Figure 5.3: Predictor factory.

The `PredictorFactory` class is simple and it only consists of the following two abstract methods:

`public String getName()`
Gets the name of the predicting strategy.

`public Visualization createPredictor()`
Creates a predictor.

For the predictors, no matter what the predicting strategy is, whenever a target is touched (i.e. a method is invoked), it makes a prediction according to the previous targets and tells whether the prediction is correct or not. Therefore, the `Predictor` class has these two abstract methods:

`public void newTarget(int target)`
Receives a new target.

`public boolean isCorrect()`
Tells if the prediction is correct.

For example, here's the factory that creates the default (one-bit) predictors:

```
public class DefaultPredictorFactory extends PredictorFactory {
    public String getName() {
        return "One-Bit Predictor";
    }

    public Predictor createPredictor() {
        return new DefaultPredictor();
    }
}
```

The implementation of the one-bit predictor is simple because it only compares the new target with the previous one, and if there are same, the prediction is correct. Note that the first time when a predictor is used, it always tells that the prediction is correct because there's no previous target.

```
public class DefaultPredictor extends Predictor {
    private int lastTarget;
    private boolean correct;

    public DefaultPredictor() {
        lastTarget = Integer.MIN_VALUE;
        correct = true;
    }

    public String getName() {
        if (target == lastTarget) {
            correct = true;
        } else {
            if (lastTarget != Integer.MIN_VALUE) {
                /* miss-prediction occurs */
                correct = false;
            }
            lastTarget = target;
        }
    }

    public boolean isCorrect() {
```

```
        return correct;
    }
}
```

5.3.2 Dimension and Factory

The first step for implementing a visualization is determining the dimensions of the visualization and the property of these dimensions. Like the hot spot visualization, a prediction visualization also has an X-axis (coordinate) and an Y-axis (entity). In addition, a prediction visualization needs an additional dimension to represent what to predict, and only entities can be mapped to this dimension. For method invocation, the bytecode number should be mapped to the X-axis, the invoking location should be mapped to the Y-axis, and the method invoked should be mapped to the third dimension.

A major difference between the factory of prediction visualization and other visualization factories is that it needs to allow users adding various predicting strategies to it (i.e. adding various predictor factories), and these predictor factories are used when creating prediction visualizations, as shown below:

```
public class PredictionVizFactory extends VisualizationFactory {
    private ArrayList factoryList;

    public PredictionVizFactory() {
        factoryList = new ArrayList();
    }

    public void addPredictorFactory(PredictorFactory factory) {
        /* adds a predicting strategy */
        factoryList.add(factory);
    }

    public public String getName() {
        return "Prediction Visualization";
    }

    protected VisualizationDefinition createDefinition() {
        DimensionDefinition[] dimensionDefinition = new DimensionDefinition[3];
```

```
        dimensionDefinition[0] = new DimensionDefinition("X-axis", "coordinate");
        dimensionDefinition[1] = new DimensionDefinition("Y-axis", "reference");
        dimensionDefinition[2] = new DimensionDefinition("Prediction", "reference");
        return new VisualizationDefinition(dimensionDefinition);
    }

    public Visualization createVisualization() {
        /* generates an array of predictor factories */
        PredictorFactory[] factory = new PredictorFactory[factoryList.size()];
        for (int i = 0; i < factory.length; i++) {
            factory[i] = (PredictorFactory)(factoryList.get(i));
        }

        /* uses the predictor factories to create a prediction visualization */
        return new PredictionViz(factory);
    }
}
```

5.3.3 Visualization Creation

The creation of a prediction visualization is similar to that of a hot spot visualization, except that a prediction visualization needs to know which predictor factories are available.

```
public class PredictionViz extends Visualization {
    private ValueDimension xAxis;
    private ReferenceDimension yAxis;
    private ReferenceDimension prediction;
    private PredictorFactory[] factory;

    public PredictionViz(PredictorFactory[] factory) {
        this.factory = factory;
    }

    protected Dimension[] createDimension() {
        xAxis = new ValueDimension();
        yAxis = new ReferenceDimension();
        prediction = new ReferenceDimension();
```

```
        Dimension[] returnVal = new Dimension[3];
        returnVal[0] = xAxis;
        returnVal[1] = yAxis;
        returnVal[2] = prediction;
        return returnVal;
    }

    protected JPanel createPanel() {
        AxesPanel returnVal = new AxesPanel();
        return returnVal;
    }
}
```

5.3.4 Configuration

Compared to the hot spot visualization, during the configuration phase, a prediction visualization needs to allow end users to choose a predicting strategy from available ones. Figure 5.4 shows the configuration dialog of prediction visualization, which uses a combo-box for predictor choosing.

The following code creates the additional configuration panel and receives all the parameters from the dialog:

```
private PredictorFactory selectedFactory;
private JComboBox comboPredictor;
private int interval;
private JTextField textInterval;

protected JPanel createConfigurationPanel() {
    /* creates the additional configuration panel */
    JPanel returnVal = new JPanel();

    returnVal.add(new JLabel("Predictor:"));

    /* selects the default predictor factory */
    selectedFactory = factory[0];
```

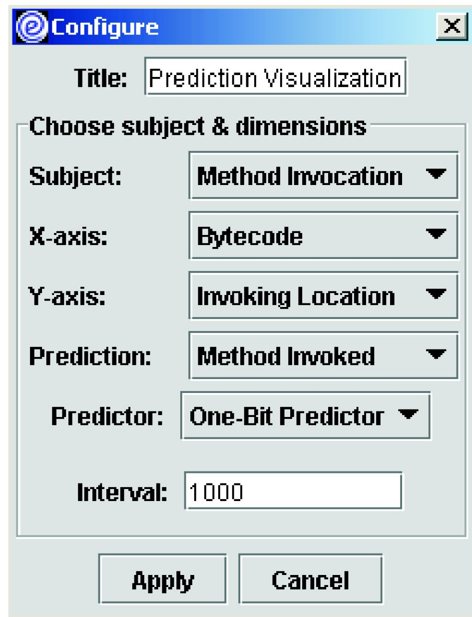

Figure 5.4: Configuration dialog of the prediction visualization.

```
    /* creates a combo-box and and adds all the available predictor factories to it */
    comboPredictor = new JComboBox();
    for (int i = 0; i < factory.length; i++) {
        comboPredictor.addItem(factory[i].getName());
    }
    returnVal.add(comboPredictor);

    returnVal.add(new JLabel("Interval:"));

    /* the default interval is 1000 */
    interval = 1000;

    /* creates the text field for input */
    textInterval = new JTextField("1000");
    returnVal.add(textInterval);

    return returnVal;
}

protected void updateConfiguration() {
```

```
    /* selects the predictor factory */
    selectedFactory = factory[comboPredictor.getSelectedIndex()];

    /* gets the interval */
    interval = Integer.parseInt(textInterval.getText());

    /* sets the name of the axes */
    ((AxesPanel)panel).setName(xAxis.getName(), yAxis.getName());

    /* draws an empty graph */
    ((AxesPanel)panel).setImage(null);

    panel.repaint();
}
```

5.3.5 Visualizing

Generating a prediction graph is similar to generating a hot spot graph except that the coloring scheme is different. In a prediction visualization, the color is determined by the predictors and every entity on the Y-axis (in this case, the invoking locations) needs a predictor. Each time when a method is invoked at a location, the index of the method is sent to the corresponding predictor and the predictor returns the appropriate color.

For end users, it's also useful to know the range of the graph, i.e. how many invoking locations have been touched (range of the Y-axis), and totally how many bytecodes have been executed (range of the X-axis). As shown in Figure 5.1, the prediction visualization provides this information on its two axes. The range of the Y-axis can be obtained directly from the reference dimension, but the prediction visualization has to keep track of the maximum value on the X-axis to know the range of the X-axis.

```
private Color colorRed = new Color(255, 0, 0);
private Color colorBlue = new Color(120, 160, 255);
private Predictor[] predictor;

private AutoImage image;
private int xMax;
```

```
public void preVisualize() {
    /* initializes the predictors */
    predictor = new Predictor[yAxis.getMaxEntityNumber()];
    for (int i = 0; i < predictor.length; i++) {
        predictor[i] = selectedFactory.createPredictor();
    }

    image = new AutoImage();
    xMax = 0;
}

public void receiveElement(Element element) {
    /* sends the target to the corresponding predictor */
    predictor[yAxis.getField(element)].newTarget(prediction.getField(element));

    /* calculates the X-position */
    int x = xAxis.getField(element) / interval;

    if (predictor[yAxis.getField(element)].isCorrect()) {
        if (image.getColor(x, yAxis.getField(element)) == null) {
            /* if the prediction is correct and the corresponding point
                hasn't been drawn yet, draws a blue point */
            image.setColor(x, yAxis.getField(element), colorBlue);
        }
    } else {
        /* miss-prediction occurs, draws a red point */
        image.setColor(x, yAxis.getField(element), colorRed);
    }

    /* keeps track of the maximum value on the X-axis */
    if (xAxis.getField(element) > xMax) {
        xMax = xAxis.getField(element);
    }
}

public void visualize() {
    /* sets the name and range of the axes */
    ((AxesPanel)panel).setName(xAxis.getName() + " (" + xMax + ")",
        yAxis.getName() + " (" + yAxis.getEntityNumber() + ")");
```

```
    /* draws the graph using the selected sorting scheme */
    ((AxesPanel)panel).setImage(image.getSortedImage(null, yAxis).getImage());
}
```

5.3.6 Data Manipulation: Making Selections

Making selections on a prediction visualization is identical to that on a hot spot visualization. However, Chapter 4 only introduced how to make a selection of entities (i.e. only considering the top and bottom edges of the selection box in Figure 5.5), but sometimes end users are also interested in selecting events that occurred during a certain period of time. For example, in a prediction visualization, an end user may only want to visualize the events that occurred between the left and right edges of the selection box.

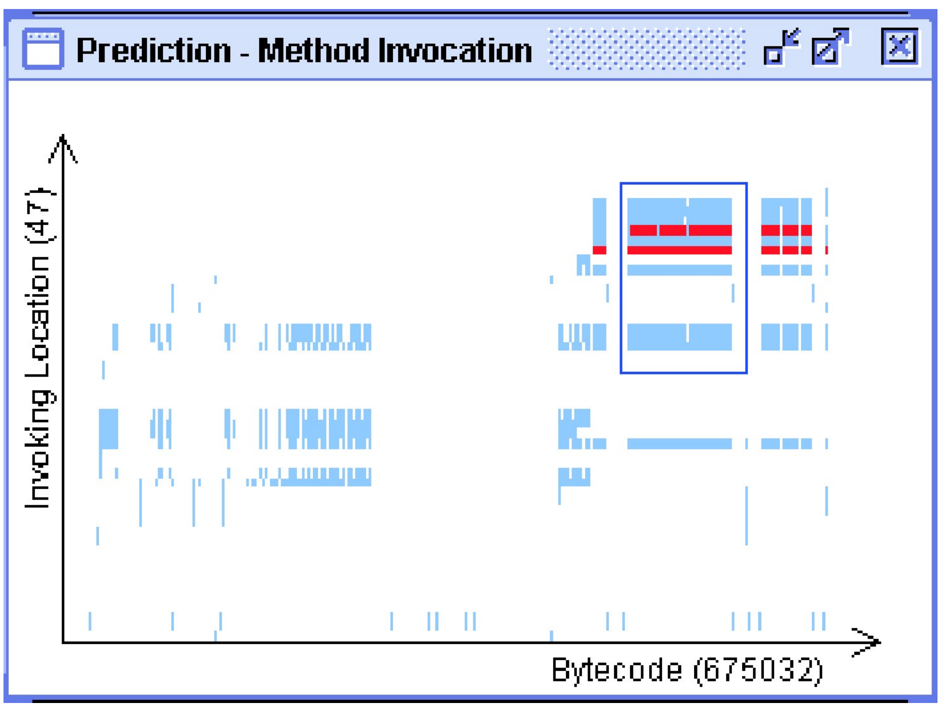

Figure 5.5: Making selections on prediction visualization.

Because events don't have ids, visualization providers cannot specify a certain event directly. However, because events occurred in a certain sequence, the EVolve platform automatically assigns a sequence number to every event when it reads the events from the data source, and visualization providers can use the sequence number to specify a period of time when making selections.

To do so, a visualization needs to keep track of the sequence numbers. For a prediction visualization, it has to remember the corresponding sequence number of each point on the X-axis, so that when end users make selections, it knows which sequence numbers the left and right edges of the selection box represent.

Therefore, the first time when an X-position is touched, the prediction visualization has to store the corresponding sequence number in a list, and this requires the following modifications to the visualizing phase:

```
private ArrayList eventCounter;

public void preVisualize() {
    /* uses and array list to keep track of the sequence numbers */
    eventCounter = new ArrayList();
}

public void receiveElement(Element element) {
    int x = xAxis.getField(element) / interval;
    /* keeps track of the sequence numbers */
    while (x >= eventCounter.size()) {
        eventCounter.add(new Integer(EVolve.getEventCounter()));
    }
}
```

After that, a prediction visualization can use the sequence numbers stored in the list to specify the time period enclosed in the selection box:

```
public void makeSelection() {
    /* gets the X-position of the left edge */
    int x1 = ((AxesPanel)panel).getStartX();

    /* gets the X-position of the right edge */
    int x2 = ((AxesPanel)panel).getEndX();

    /* gets the sorted index of the entity that corresponds to the bottom edge */
    int y1 = ((AxesPanel)panel).getEndY();
```

```
    /* gets the sorted index of the entity that corresponds to the top edge */
    int y2 = ((AxesPanel)panel).getStartY();

    /* adds all the entities between the two edges to the selection */
    int[] selection = new int[y2 - y1 + 1];
    for (int i = y1; i <= y2; i++) {
        selection[i - y1] = i;
    }

    /* makes the selection and specifies the corresponding time period */
    yAxis.makeSelection(selection, ((Integer)eventCounter.get(x1)).intValue(),
        ((Integer)eventCounter.get(x2)).intValue());
}
```

5.3.7 Data Manipulation: Sorting

Sorting a prediction visualization is exactly same to sorting a hot spot visualization. However, for prediction visualizations, end users may want to know which locations cause the most miss-predictions. Therefore, prediction visualization needs an additional sorting scheme on its Y-axis.

In EVolve, adding a sorting scheme is done by adding an `EntityComparator` to a reference dimension. An entity comparator compares entities according to a certain sorting scheme and EVolve provides a `ValueComparator` which uses an array of integers to determine the order of the entities. To use the value comparator, a prediction visualization needs to calculate the number of miss-predictions during the visualizing phase:

```
private int[] miss;

public void preVisualize() {
    /* initializes the miss-prediction counter */
    miss = new int[yAxis.getMaxEntityNumber()];
    for (int i = 0; i < miss.length; i++) {
        miss[i] = 0;
    }
}
```

```
public void receiveElement(Element element) {
    /* calculates the miss-prediction counter */
    if (!predictor[yAxis.getField(element)].isCorrect()) {
        miss[yAxis.getField(element)]++;
    }
}

public void visualize() {
    /* adds a sorting scheme to the Y-axis */
    yAxis.addComparator(new ValueComparator("Miss Prediction", false, miss));
}
```

The comparator is created in the last step of the visualizing phase by giving the name of the sorting scheme ("Miss Prediction") and specifying the order of sorting (descending, because the second parameter is false).

5.4 Integration

To use the data source and the prediction visualization, EVolve also needs a wrapping class that integrates them together:

```
public class Main {
    /* the entry point */
    public static void main(String[] args) {

        /* creates the data source */
        DemoSource dataSource = new DemoSource();

        /* creates the factory of prediction visualization */
        predictionFactory = new PredictionVizFactory();

        /* adds the default predicting scheme to the factory */
        predictionFactory.addPredictorFactory(new DefaultPredictorFactory());

        VisualizationFactory[] factory = new VisualizationFactory[1];
        factory[0] = predictionFactory;
```

```
        /* starts EVolve using the data source and the factory */
        EVolve.start(dataSource, factory);
    }
}
```

The wrapping class provides an entry point which first creates the data source and the available visualization factories (the example above only creates the factory of prediction visualization), and then uses them to start EVolve.

Chapter 6

Related Work

Software visualization has a long history[15] and many software visualization systems have been developed including some commercial tools (such as JProbe[2] and Optimizeit[3]) which mainly focus on facilitate performance tuning by providing various views to visualize the run-time usage of system resources, such as CPU time and memory.

Other general purpose software visualization systems include Jinsight[1, 13, 14] and BLOOM[16, 17]. Jinsight focuses on applying visualization, pattern extraction, database query, and other techniques to solve problems of performance analysis, memory leak diagnosis, debugging, and general program understanding. Although Jinsight is not an extensible tool, it provides many useful views to visualize different aspects of the run-time behaviour of programs.

BLOOM is a system that provides various 2D and 3D visualization strategies and data analyzing techniques (including a visual query language). The architecture of BLOOM is extensible and visualizations are independent from data sources. Furthermore, BLOOM provides a group of control panels for end users to choose parameters and specify how the trace data should be visualized in the visualizations.

In the domain for object-oriented program visualization, Jinsight and BLOOM are closest to EVolve. However, BLOOM emphasizes the richness of visualization techniques at the cost of user-interface facility, without providing guidance to the users as to which visualizations are especially revealing for characterizing object-oriented program behaviour. Jinsight, on the contrary, strives to provide a set of visualizations which are useful for object-oriented program profiling and debugging (e.g. memory leaks), and emphasizes performance, while not allowing extensions. EVolve lies somewhat between Jinsight and BLOOM. It facilitates extensions, and provides a given set of visualization techniques for object-oriented languages, some of which are not available in either of the two other tools.

There are also many software visualization tools that are designed for more specific purpose. Shaham[21] and Röjemo[18] provide a technique to help finding memory leak by visualizing the lifetime of objects. Jerding[9] presents an efficient way to visualize the object and class interactions occurred during the execution of programs. Lange[10] facilitates framework understanding by visualizing design patterns.

Software visualization techniques are also widely used to facilitate reverse engineering. Rigi[24] is an interactive tool designed to simplify understanding and re-document software systems. Moose[11] is a re-engineering environment that uses *class blueprints* to help understanding the purpose of a class and its inner structure.

Algorithm animation[6, 12, 22] is another major part of software visualization, and these tools normally generate detailed visualizations that describe the behaviour of a specific program and its data, and this allows the users to understand various algorithms easily.

Unlike EVolve, most of the above tools tend not to be extensible—they use built-in or specific profiling front-ends to generate trace data and use a fixed set of visualizations to interpret the data.

Extensibility is more often seen in information visualization (vs software visualization) systems. These systems tend to concentrate on extensibility, because they are designed to solve general purpose problems. Visage[19], for example, is an information visualization environment for data-intensive domains that supports and coordinates multiple visualizations and analysis tools. Furthermore, Visage provides an interactive tool to facilitate creating new visualizations. EVolve of course is designed for a much more specific domain.

Chapter 7

Conclusions and Future Work

This thesis presented EVolve, an extensible framework for visualizing the characteristics and behaviour of object-oriented programs. The architecture of EVolve is designed to facilitate the addition of new data sources as well as new visualization techniques. EVolve allows data providers to examine a new data source immediately using a wide range of visualizations, and allows visualization providers to test a new visualization technique on a variety of existing sources.

The EVolve framework consists of three major parts:

- the core **EVolve platform** connects the data source and the visualizations, and supports the communication among the visualizations. Furthermore, the EVolve platform also provides a set of data manipulating functionalities to help end users navigating the data.
- the **data protocol** uses a flexible method to represent different kinds of trace data. Through the data protocol, the EVolve platform can read trace data from different types of data sources. The data protocol of EVolve is not only flexible but also efficient so that end users can manipulate large amount of data.
- the **visualization protocol** handles the communication between the platform and the visualizations. Because there is no direct interaction among the visualizations, adding new visualizations to EVolve won't require updating the existing ones. With the visualization protocol, EVolve can provide data manipulating functionalities while still keeping the visualizations independent.

The thesis described the use of EVolve, both as a stand-alone software visualization tool, as well as a framework that can be extended easily. For end users, EVolve provides a set of built-in visualizations to visualize various types of program characteristics, and helps them find the most important information from their trace data. For data providers and visualization providers, EVolve is a framework that helps them visualizing different kinds of trace data and/or implementing various new visualization techniques easily.

The thesis also presented a case study showing how to use EVolve solving real problems. The case study not only served as a tutorial for data providers and/or visualization providers, but also demonstrated the extensibility and flexibility of EVolve.

Great effort has been placed in testing EVolve for different types of users. In particular:

- **End users:** twenty graduate students at McGill used EVolve to characterize the run-time behaviour of Java programs. This experiment demonstrated the ease of use and clarity of EVolve for the students claimed that EVolve facilitated and enhanced program understanding.
- **Data providers:** EVolve has been used with traces generated from JVMPI, a customized Java Virtual Machine, and several internal formats, and these experiments demonstrated the flexibility of EVolve.
- **Visualization providers:** the extensibility of EVolve was shown by adding new, custom visualizations without any modifications of the framework. Implementing these new visualizations was a straightforward process, requiring relatively little coding and minimal time (a few hours).

Future work of EVolve mainly includes the following:

- extending EVolve's repertoire of visualization techniques, and testing these visualizations on more data sources. Since extensibility is built-in, the core of the EVolve platform does not need to change.
- investigating other user-interface and comparison techniques that may improve comprehension of the resulting visualizations.
- adding more functionalities to facilitate end users using EVolve, such as saving the configurations and selections so that they can be shared by different data traces.

Appendix A

Getting Started

The source code and class files of EVolve are available at (a small trace data is also included): http://www.sable.mcgill.ca/evolve/EVolve.jar

Here's a short description about how to get started:

- Use the following command to install EVolve:
  `jar xvf EVolve.jar`

- EVolve requires at least JDK1.4 (available at http://java.sun.com/), because it uses several new features. Start EVolve under the `classes` directory where it's installed:
  `java Main`

- EVolve uses assertions to validate the trace data, so to compile the source code of EVolve, use the `-source 1.4` parameter:
  `javac -source 1.4 Main.java`

- By default, assertions are disabled to avoid overhead. To enable assertions when debugging, use:
  `java -ea Main`

Appendix B

Built-in Visualizations of EVolve

EVolve now has the following seven built-in visualizations:

- Table
- Hot Spot Visualization (Coordinate)
- Hot Spot Visualization (Amount)
- Vertical Bar Chart
- Horizontal Bar Chart
- Correlation Graph
- Prediction Visualization

B.1 Table

Method Invoked	Number of Invocation
java.util.jar.Attributes$Name.isAlpha(char)	5859
java.util.jar.Attributes$Name.isValid(char)	5859
sun.reflect.ByteVectorImpl.add(byte)	2702
sun.reflect.ClassFileAssembler.emitByte(...	2702
java.util.jar.Manifest.toLower(int)	996
java.util.jar.Manifest$FastInputStream.rea...	978
java.util.jar.Manifest$FastInputStream.rea...	978
java.util.jar.Attributes$Name.isDigit(char)	693
java.util.jar.Attributes$Name.[init](java.lan...	479
java.util.jar.Attributes$Name.isValid(java.l...	479
java.util.jar.Attributes$Name.hashCode()	471
java.util.jar.Attributes.putValue(java.lang.S...	462
java.util.jar.Attributes.put(java.lang.Object,...	462
sun.reflect.ClassFileAssembler.emitShor...	428
java.io.DataOutputStream.incCount(int)	340
sun.reflect.ClassFileAssembler.cpi()	264
java.util.jar.Attributes.[init](int)	258
java.util.jar.Attributes.read(java.util.jar.Man...	258

Figure B.1: A table.

Figure B.1 shows a table. A table has two dimensions, the left column and the right column. The left column contains entities, and the right column shows the summary of certain amount values related to the entities. For example, Figure B.1 provides information about how many times each method is invoked during the execution of a program.

(Source code: `/evolve/visualization/TableViz.java`)

B.2 Hot Spot Visualization (Coordinate)

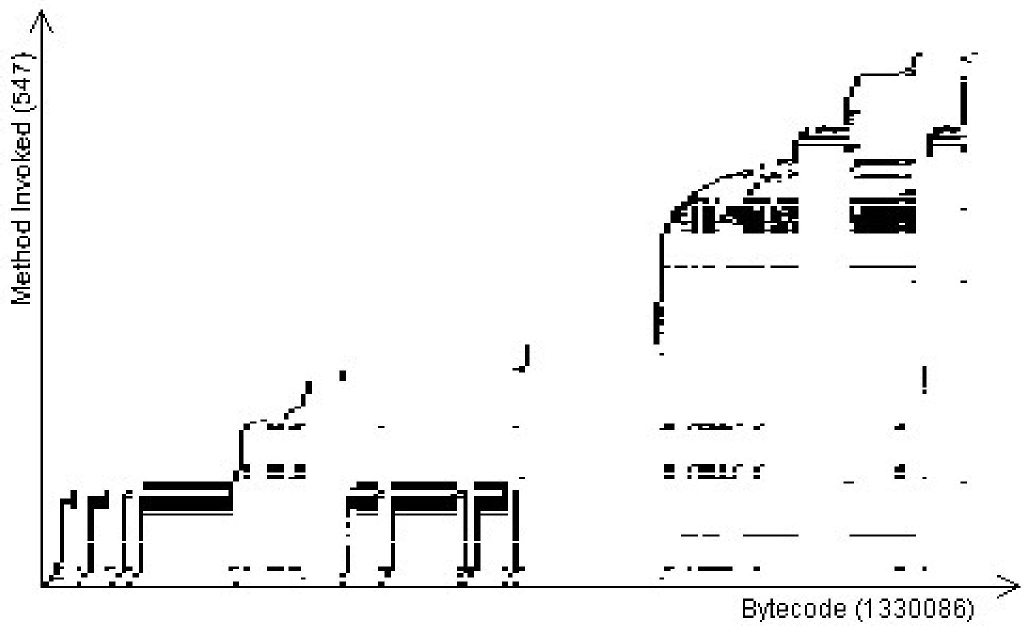

Figure B.2: A hot spot visualization (coordinate).

A hot spot visualization shows when different events occurred during the execution of a program. A hot spot visualization has two dimensions, the X-axis and the Y-axis. The Y-axis represents entities and the X-axis represents time in a certain kind of measurement.

Figure B.2 shows a hot spot visualization and the property of the X-axis is coordinate. The visualization shows when each method is invoked and time is measured as the number of bytecodes executed since the start of program.

(Source code: `/evolve/visualization/HotSpotCoordinate.java`)

B.3 Hot Spot Visualization (Amount)

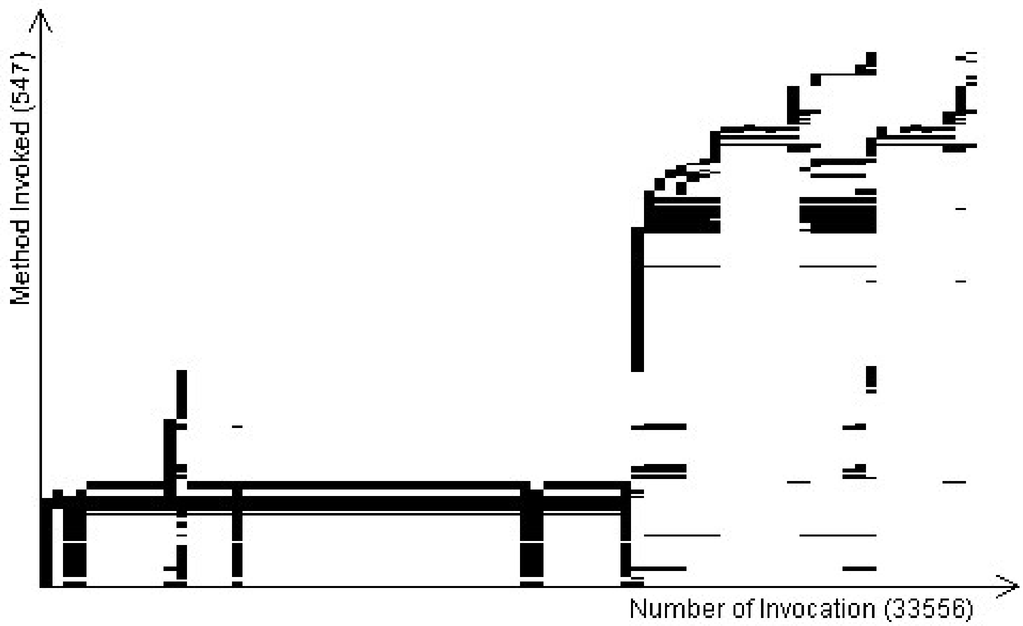

Figure B.3: A hot spot visualization (amount).

Figure B.3 shows another hot spot visualization and it's similar to the one shown in Figure B.2 except that the property of the X-axis now is amount. So in Figure B.3, time is measured as the number of invocations occurred since the start of program.

(Source code: `/evolve/visualization/HotSpotAmount.java`)

B.4 Vertical Bar Chart

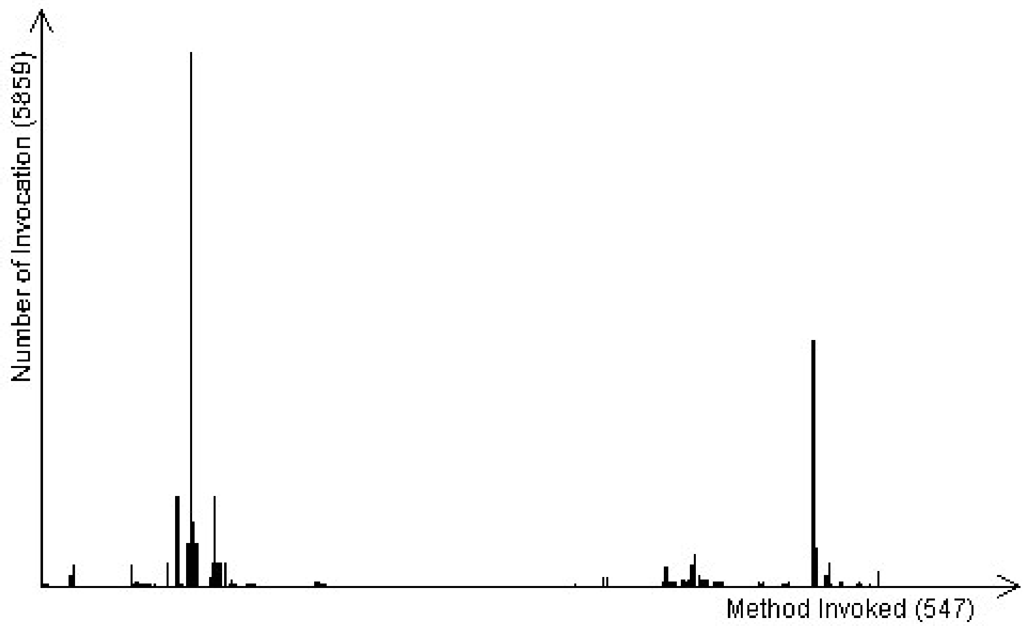

Figure B.4: A vertical bar chart.

Like tables, bar charts provide summary of amount values. Figure B.4 shows a vertical bar chart. Its X-axis represents entities (method invoked) and Y-axis represents amount values (number of invocations). From the graph, it is easy to find which methods are invoked the most.

(Source code: `/evolve/visualization/BarChartVertical.java`)

B.5 Horizontal Bar Chart

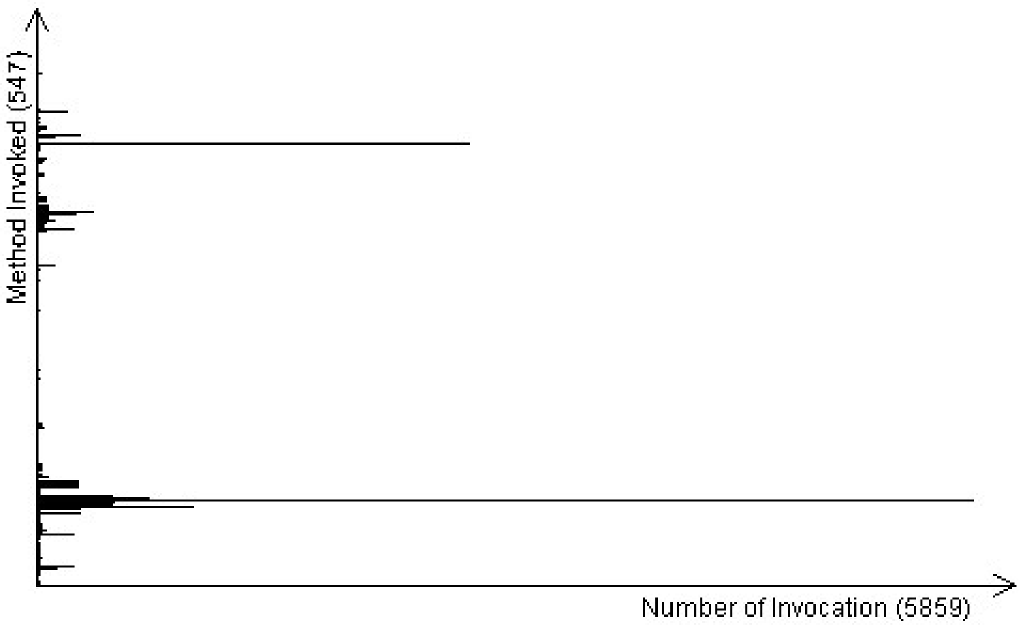

Figure B.5: A horizontal bar chart.

Figure B.5 shows a bar chart similar to the one shown in Figure B.4 except that the bars are in horizontal direction. So now the Y-axis represents entities and the X-axis represents amount values. This visualization is useful especially when working together with other visualizations whose Y-axis also represents entities (such as the hot spot visualization), because it's easy to compare the result.

(Source code: `/evolve/visualization/BarChartHorizontal.java`)

B.6 Correlation Graph

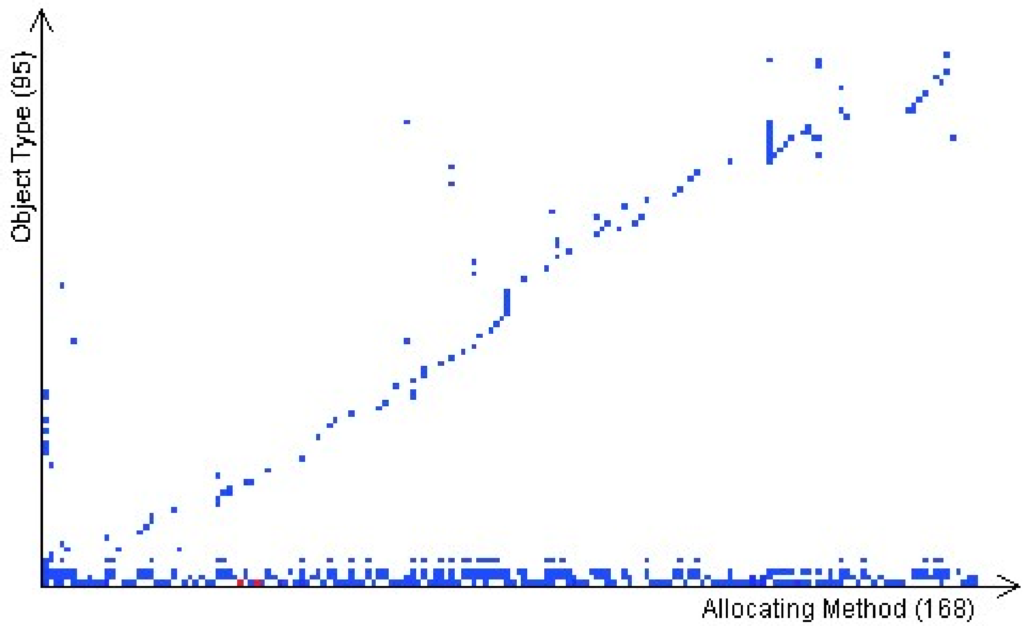

Figure B.6: A correlation graph.

A correlation graph provides information about the correlation between entities. For example, Figure B.6 depicts the correlation between the allocating methods and the type of objects allocated. The correlation visualization uses color to indicate how often the events occurred (events that are red occurred more frequently than those that are blue).

(Source code: `/evolve/visualization/CorrelationViz.java`)

B.7 Prediction Visualization

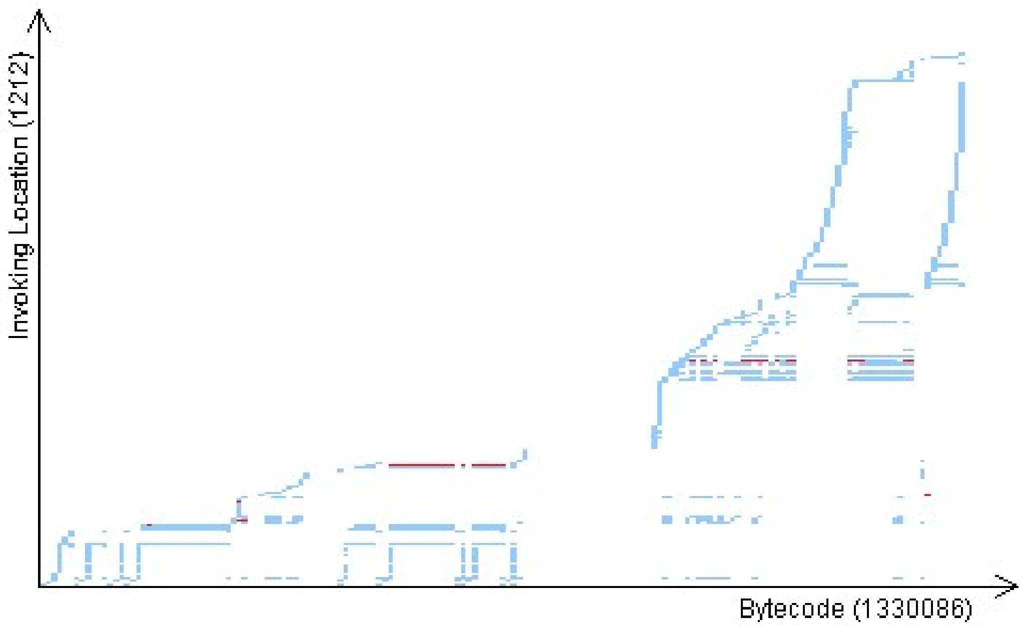

Figure B.7: A prediction visualization.

A prediction visualization is similar to a hot spot visualization, except that it uses color to indicate the predictability of events (red indicates miss-predictions). Figure B.7 shows the predictability of method invocations, so the red points correspond to method calls that are polymorphic and the blue points correspond to those that are monomorphic.

(Source code: `/evolve/visualization/PredictionViz.java`)

Bibliography

[1] *Jinsight*. http://www.research.ibm.com/jinsight/.

[2] *JProbe*. http://www.sitraka.com/software/jprobe/.

[3] *Optimizeit*. http://www.optimizeit.com/.

[4] *Standard Performance Evaluation Corporation*. http://www.spec.org/.

[5] *Volano Benchmark*. http://www.volano.com/benchmarks.html.

[6] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, (1):28–39, 1985.

[7] Karel Driesen, Nagi Basha, David Eng, Matt Holly, John Jorgensen, Georges Kanaan, Babak Mahdavi, and Qin Wang. Visualizing hot spots in various domains. In *Software Visualization Workshop in conjunction with the International Conference on Software Engineering (ICSE 2001),*, pages 65–70, 2001. Available at http://www.cs.mcgill.ca/resrchpages/tech2001.html.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[9] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of the International Conference on Software Engineering (ICSE'97)*, pages 360–371, 1997.

[10] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 342–357, 1995.

[11] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*, pages 300–311, 2001.

[12] Ralph L. London and Robert. A. Duisberg. Animating programs using Smalltalk. *Computer*, (8):61–71, 1985.

[13] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, pages 326–337, 1993.

[14] Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan. Drive-by analysis of running programs. In *Software Visualization Workshop in conjunction with the International Conference on Software Engineering (ICSE 2001)*, pages 17–22, 2001. Available at http://www.research.ibm.com/jinsight/.

[15] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, (3):211–266, 1993.

[16] Steven P. Reiss. Software visualization in the Desert environment. In *Proceedings of the 1998 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engeneering (PASTE'98)*, pages 59–66, 1998.

[17] Steven P. Reiss. An overview of BLOOM. In *Proceedings of the 2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engeneering (PASTE'01)*, pages 2–5, 2001.

[18] Niklas Röjemo and Colin Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 34–41, 1996.

[19] Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina C. Gomberg, Michael B. Burks, Philip J. Stroffolino, John A. Kolojejchick, and Carolyn Dunmire. Visage: A user interface environment for exploring information. In *Proceedings of Information Visualization, IEEE*, pages 3–12, 1996.

[20] Steven F. Roth and Joe Mattis. Data characterization for intelligent graphics presentation. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI'90)*, pages 193–200, 1990.

[21] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 104–113, 2001.

[22] John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, (9):27–39, 1990.

[23] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of International Conference on Software Maintenance*, pages 275–285, 1995.

[24] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'97)*, pages 606–607, 1997.

[25] Deepa Viswanathan and Sheng Liang. Java virtual machine profiler interface. *IBM Systems Journal*, (1):82–95, 2000.