McGill University
School of Computer Science
Sable Research Group

# EVolve: An Extensible Software Visualization Framework

Sable Technical Report No. 2002-6

Qin Wang, Rhodes Brown, Karel Driesen, Laurie Hendren and Clark Verbrugge
[qwang21,rhodesb,karel,hendren,clump]@cs.mcgill.ca
Sable Research Group, School of Computer Science, McGill University

**Abstract**

Existing visualization tools typically do not allow easy extension by new visualization techniques, and are often coupled with inflexible data input mechanisms. This paper 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 be used to build custom visualizers by implementing new data sources and/or new kinds of visualizations.

The paper presents an overview of the system, examples of its use, and discusses our experiences using the tool both as an end-user and as a developer building custom visualizers.

# 1 Introduction

Visualizing software is very useful for both program understanding and performance tuning. Visualizing software is also helpful in the design of compiler optimizations and runtime systems. This is particularly true when dealing with complex programs written in relatively high-level languages like Java, where the behaviour and optimization opportunities are not obvious.

Whereas existing special purpose visualization systems may be useful for typical visualizations such as detecting performance hotspots (profiling tools) or visualizing connections between classes (program understanding tools), we found that existing systems did not always provide enough flexibility to visualize the kinds of dynamic information needed for understanding the complex or specialized program behaviours that are of interest to compiler and runtime system developers. Thus, we have designed and implemented EVolve, a flexible and extensible visualization framework.

The EVolve 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. This makes it applicable to a wide variety of visualization problems, and in particular for situations where stock visualization techniques are inadequate, or do not exist.

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 was designed to build custom visualizers by implementing new data sources and/or new kinds of visualizations.

The paper is organized as follows. In Section 2 we discuss the overall architecture of the system. In Section 3 we explore the design of data sources in more detail and in Section 4 we provide an overview of the visualizations. We have had quite positive experiences using the system with both end-users and in extending the system by providing new data sources and new visualizations. These experiences are summarized in Section 5. Finally, we give related work in Section 6 and conclusions in Section 7.

# 2 Architecture

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 data source).

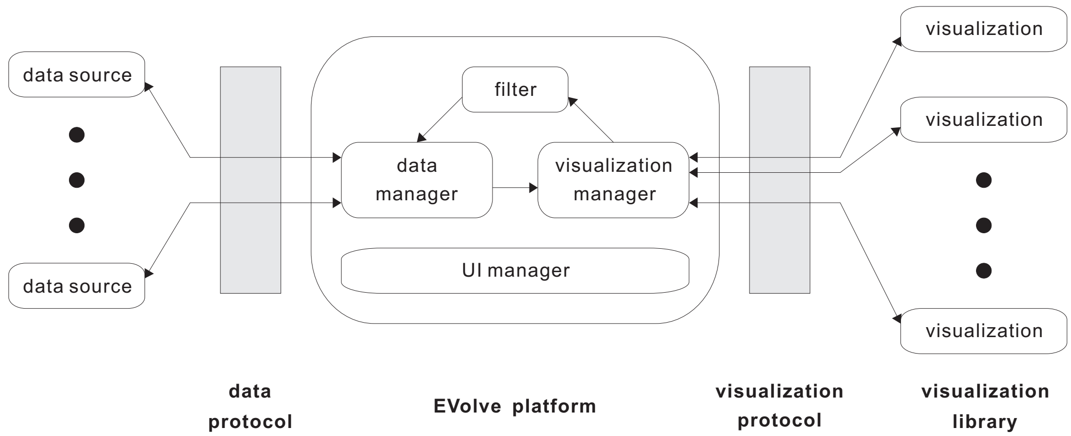


Figure 1: Architecture of EVolve.

Figure 1 shows the architecture of EVolve. The core of the whole framework is the EVolve platform. Externally, the EVolve platform communicates with the data source through the data protocol, and controls the visualizations through the visualization protocol. In EVolve, a visualization can only communicate with the data source and the other visualizations by going through the EVolve platform—there is no direct interaction between the data source and the visualizations, and there is also no direct interaction among the visualizations. Data sources and the visualizations are independent, and using different data source or adding new visualizations does not require changing unrelated parts of the EVolve platform.

Internally, the EVolve platform consists of four parts: a data manager, a visualization manager, a user-interface manager, and a filter. Within each visualization process, the data manager first reads data from the data source and sends the data to the visualization manager. After the visualizations read the data from the visualization manager and generate the corresponding visual representation, an end-user can perform data manipulations on the visualizations, such as selecting subsets of the data. The data manipulation information is fed back to the visualization manager and is used by the filter to determine what should be visualized in the next visualization process.

# 3 Data Representation

EVolve is designed to accept input data from a variety of sources (see left side of Figure 1). These sources are expected to be modular, in the sense that the same visualization should be able to function with different sources (and vice versa). An additional requirement is imposed by the size of the input data—large trace files can be gigabytes in size. The source format should therefore avoid redundancy, and be amenable to non-core storage during visualization.

To satisfy these constraints, EVolve requires input data to be formatted in a specific, though flexible manner. Input data used by EVolve is composed of *elements*, aggregated data akin to a simple C structure. Each element is either an *entity* or an *event*. Entities describe the static, unordered data that does not change during the course of visualization construction; e.g., a data type. Events are used to describe the bulk of visualization data: events are ordered and duplicate events are sensible (e.g., the same event happens twice).

Once the various entities and elements are defined, actual creation of a data source is straightforward. To assist in the construction of elements, a variant of the well-known builder pattern[5] is provided. Malformed data and other minor problems are thus prevented.

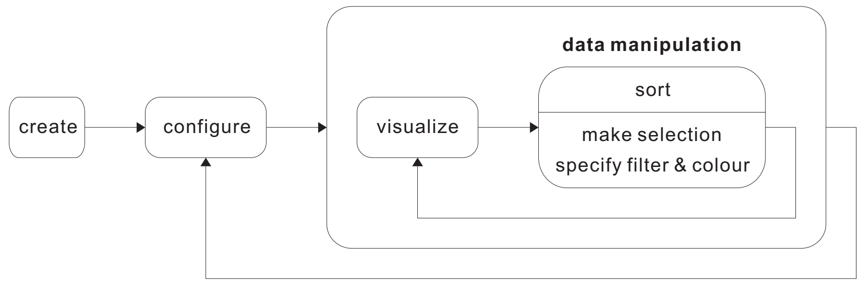


Figure 2: Visualization Cycle

## 3.1 Reducing Storage Requirements

Similar or duplicate trace events often include repetitive information. EVolve's input format reduces this by allowing entities to include an identifier or reference handle; this allows particular entities to be referenced by other elements. Complex, arbitrarily-partitioned entities can thus be formed by references between entities. More importantly, events can use entity references to refer to specific, unchanging data without unnecessary repetition; e.g., data representing runtime method invocations might include many invocation events, each describing the invocation target by a reference to the same method description entity.

Entities are used to represent often-referenced but rarely-changing information, and so it is appropriate to keep entities in main memory during runtime. Events, however, are processed sequentially and need only be examined once in order to generate a visualization, and so secondary storage (disk) is sufficient. In our typical runs over 99% of the input data is comprised of events; by keeping only entities and not events in memory storage cost is manageable even for large input data.

## 3.2 Data Characterization

Visualizations typically have a restricted domain of interest: it is not necessary for a visualization of dynamic memory allocation to receive events describing method invocations. Efficient filtering of input data is enabled by associating each visualization with a specific *subject*, and ensuring only elements appropriate to the visualization subject are delivered.

Distinct visualizations also demand certain properties of the actual data being visualized. A bar graph, for instance, requires ordered data on at least one axis. In general, given a visualization with various degrees of freedom it is important to know which element fields can be mapped onto which "dimensions" of the visualization.

Data source definitions therefore include *properties* along with element fields. Basic properties include whether the datum represents an *amount* or a *coordinate*. The former describes numeric, summable data (e.g., memory allocation size, duration of a function call, etc), whereas the latter is used for non-numeric or non-additive data (e.g., type name, memory address of a variable, etc). Basic properties such as these are common to many visualizations[10] and so are "built-in." Data characterization requirements of new or future visualizations are supported by the ability to specify and query user-defined properties.

# 4 Visualization

The actual visualizations provided by EVolve are produced by the EVolve platform (central part of Figure 1) and a library of visualizations (right part of Figure 1). The visualization protocol specifies the interface between the platform and the visualizations.

EVolve has a number of standard visualizations, including support for bar charts, hot spot graphs, correlation graphs and prediction graphs, which are provided in EVolve's standard visualization library. End-users can use EVolve as a stand-alone tool and create one or more of these standard visualizations.

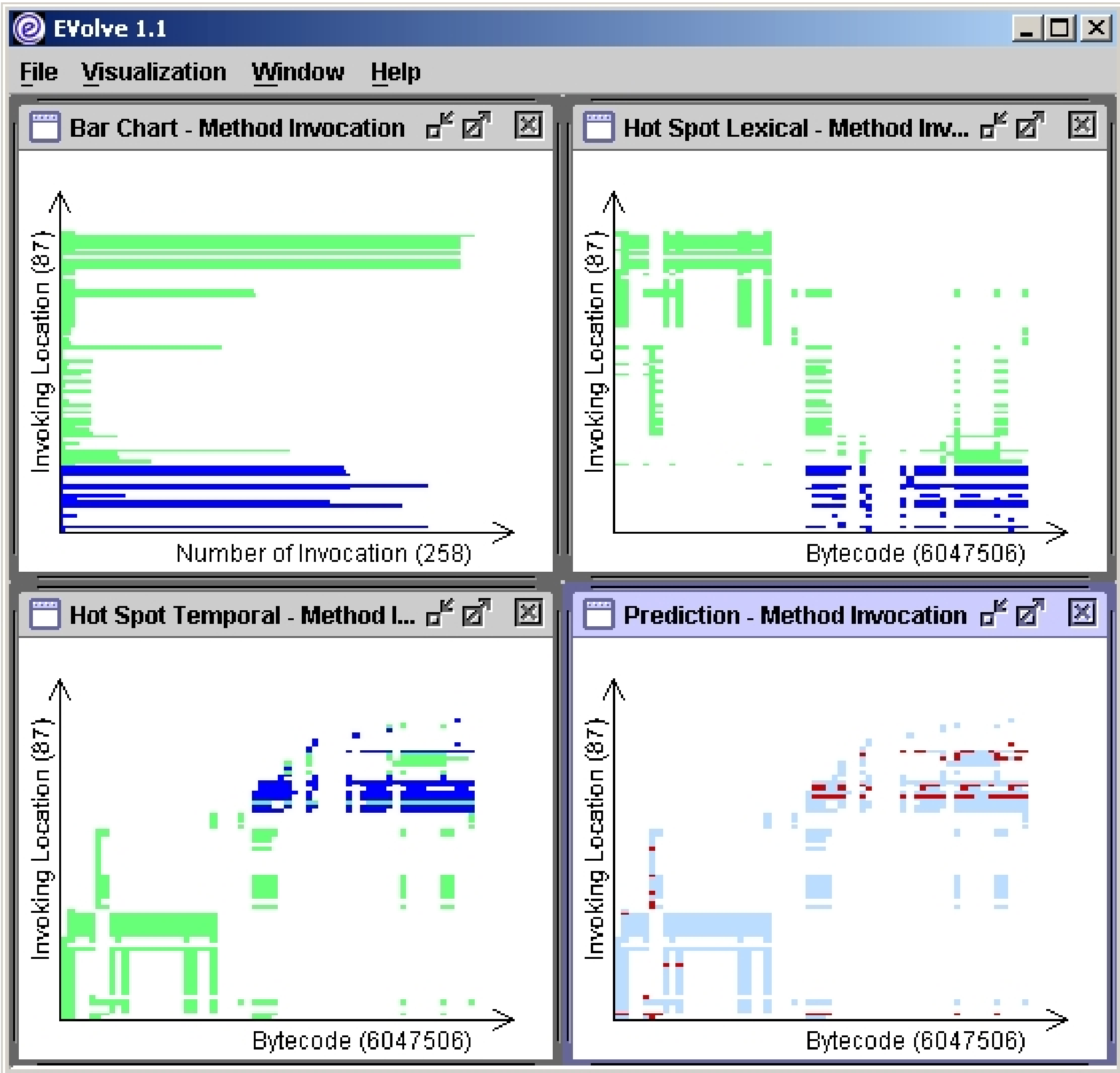


Figure 3: Screen Shot of EVolve.

However, EVolve was also designed to be extensible, and so the architecture of EVolve makes it simple to add new kinds of visualizations to the library.

In this section we first give an overview of a typical end-user interaction with EVolve, and then show how a profiler/visualizer developer can extend EVolve by adding new visualizations to the library.

## 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 in the library.

To create and modify each visualization, the end-user goes through the process shown in Figure 2. In the following sections we illustrate this process using the four visualizations in Figure 3. This example shows four different visualizations of method invocations from the Volano benchmark[4].

### 4.1.1 Bar Chart Visualization

Consider the upper left pane in Figure 3 which shows a bar chart of method invocations. Note that 87 method invocation locations were each executed up to 258 times.[1] The y-axis shows invocation site locations sorted in lexical order (`com.volano` classes in blue/black[2] at bottom, java libraries in green/grey on top). The length of the bar 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 invoke location over the entire program run. Users can find the name of the method being invoked by placing their mouse over the bars in the chart.

The end-user uses the following steps to produce this chart. First, the user indicates that he/she wants to create a bar chart; this corresponds to the box labelled create in Figure 2. Second, the end-user configures the bar chart by indicating which data should be displayed and which axes should be used. All such configurations are done via menus built by EVolve—only valid choices based on the current visualization type and the data source are provided to the user. In this case the end-user selects method invocations as the data to be displayed and maps the location of the invocation 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. Configuration is optional, and if no configuration is given by the 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 by EVolve. On the first visualization all of the data is displayed. However, the end-user may want to perform various data manipulations on the visualization (see the box labelled data manipulation in Figure 2). He/she may select only parts of the data (filter), assign colours 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, and then assigned this group the colour blue/black. All other visualizations that do not define their own colouring scheme share this colour selection. After specifying the modifications the end-user then indicates that the data should be visualized and the new visualization is computed and displayed. This process of specifying data manipulations and revisualizing 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 data or axes are assigned to the visualization.

### 4.1.2 Hot Spot Visualization

The top right pane in Figure 3 shows a lexical hot spot graph of the same 87 method invocations. It is produced in a similar manner to the bar chart, so we only mention differences from the bar chart visualization. The y-axis remains the same. The x-axis indicates time as number of bytecodes executed since start of program (a coordinate) for a total of 6,047,506 executed bytecodes. A hot spot graph shows when a particular invocation location is active. For example, from this 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 consists of program/JVM initialization and class loading, whereas the second half executes mostly `com.volano` code, with some calls to `java` libraries.

The bottom left pane illustrates a temporal hot spot graph. The x-axis remains the same as for the lexical hot spot, encoding time as number of byte codes executed. The y-axis shows the same 87 method invocation locations, but sorted by the time they are first activated. A temporal hot spot groups together invocations that execute together, emphasizing program phases. The blue/black `com.volano` invocations therefore appear clustered together high on the y-axis. EVolve provides two default sorting schemes (lexical and temporal), but is designed to facilitate painless integration of other sorting schemes.

[1] To make the example a reasonable size for the paper we are considering only 87 method invocation sites, the whole benchmark has significantly more sites.

[2] Electronic versions of this file include colour images.

#### 4.1.3 Prediction Visualization

Our first version of EVolve supported the bar chart, hot spot and other simple visualizations. After working with the system we realized that it would be very interesting to have another kind of visualization that helps us display the predictability of events.

The bottom right pane of Figure 3 shows this new visualization. In this case we are displaying predictor misses of virtual method invocations. The x-axis and y-axis are identical to temporal hot spot definitions, but the visualization provides its own colouring 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). 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 colours: 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. They should reduce the amount of red area in the graph. This visualization shows that `com.volano` exhibits a higher degree of polymorphism than method invocations in other packages. For example, in the startup phase, most invocations never change method targets.

### 4.2 Implementing a new visualization

As shown in Figure 2, implementing a visualization involves support for several operations such as creation, data manipulation, and data processing and drawing the visualization.

In order to simplify the task of building new visualizations, EVolve provides a visualization prototype (an abstract Java class) which implements methods that are common to all the visualizations and new visualizations are built by extending this prototype. Therefore, visualization providers can focus on implementing methods that are specific to their visualizations. The key methods support the following:

- Define the dimensions of the visualization and create a canvas to display the visual representation of the trace data.
- Receive elements from the visualization manager and generate the visual representation.
- Allow the end-user to make selections on the canvas and generate subsets of the data elements accordingly.
- If necessary, provide additional sorting schemes other than the default ones.

## 5 Experiences

EVolve has been used extensively by three classes of users: those interested in examining data using existing visualizations (end-users), providers of raw data who want to visualize a new data source (data providers) and designers of new visualizations (visualization providers). In this section we discuss some of these experiences.

### 5.1 End-Users

20 masters students in an upper level graduate course at McGill used EVolve to characterize the runtime behaviour of Java Programs. This experiment demonstrated the ease of use and clarity of EVolve for the end-user. None of the students had previous experience with visualization or characterization of runtime behaviour of object-oriented programs. They first ran a Java benchmark of their choice on an instrumented Kaffe JVM to produce execution traces as data sources, and then visualized executed bytecodes, allocated

objects and executed methods in temporal and lexical hot spot graphs (lower left and top right graphs in Figure 3).

This experience brought to light several aspects of EVolve which enhance program understanding:

- temporal hot spot graphs highlight different execution phases and are therefore useful in a preliminary exploration.
- a mouseover in EVolve shows the method / bytecode / object type pointed at, which allows a user to determine frequent occurrences within a program phase.
- using colour selection, those methods / bytecodes / object types are highlighted in other graphs, allowing a user to determine the active packages in a particular temporal hot spot, for example.
- by using the same x-axis in different graphs, different aspects of program behaviour can be compared. For example, object allocation can be compared with method invocations, allowing a user to determine the methods responsible for memory consumption in a program phase.

The following aspects were not present in EVolve during this class, but were added later as a result of this experience:

- we now show the cardinality of the x-axis and y-axis, both to provide feedback about resolution and in order to easily spot graphs that share the same x or y-axis.
- graphs are now aligned in panes, which also facilitates comparisons between graphs that share an axis.

We found that program understanding was enhanced by comparing two or more graphs, rather than by studying one graph in isolation.

## 5.2 Data Providers

As mentioned in Section 3, we developed the concept of element builders to simplify the process of developing a new front-end data source. Our experience has shown this to be an effective design choice. For example, an initial prototype of EVolve included a data source that read a simple text-based trace format. To apply the prototype system to real data sets it was necessary to create a new data source, one that read a different, binary trace format and presented the same events to the back-end system. This task was simplified by the use of the builder API to define and create the elements. In fact, the application of the builders was so straightforward that we are now considering ways to automate the process with a simple compiler—thus simplifying the job of a data source developer even further.

So far, EVolve has been used to visualize different bytecode execution traces generated by JVMPI (Java Virtual Machine Profiler Interface)[13], Kaffe (a Java virtual machine), and some internal formats.

## 5.3 Visualization Providers

By using the visualization prototype, we have implemented several visualizations that makes up the EVolve visualization library. Our experience shows that building new visualizations in EVolve is relatively simple and normally a full-fledged visualization only consists of about two hundred lines of Java code. Furthermore, the visualization prototype provides a standard routine for visualization providers, and this makes the task of implementing new visualizations very straightforward. For example, the prediction visualization (lower right graph in Figure 3) was added to EVolve in about three hours.

# 6 Related Work

Visualization tools are often used for performance tuning. Programs such as Jinsight[1, 7], JProbe[2] and OptimizeIt[3] are designed to help programmers optimize their programs by visualizing the runtime usage of system resources (CPU time, memory, etc). Unlike EVolve, these 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.

Visualization has also been applied to the fields of software understanding and reverse engineering. These tools, such as Rigi[12] (using SHriMP views[11]) and Moose[6] help developers understand 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.

Most software visualization tools are designed to visualize certain aspects of software systems, and so provide little to aid in the development of new visualizations. One of the few exceptions is BLOOM[8], which provides extensibility by using a visualization back-end that supports a variety of visualization strategies. In contrast, our approach to extensibility is to provide a framework that simplifies the task of designing new visualizations and connecting variant data sources to them.

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[9], 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.

# 7 Conclusion and Future Work

We presented the EVolve platform, a framework for visualizing the behaviour of programs. The architecture of EVolve is designed to facilitate the addition of new data sources as well as new visualization techniques. Both can be added independently, enabling a data provider to examine a new data source immediately using a wide range of visualizations, and allowing a visualization provider to test a new visualization technique on a variety of existing sources.

We described the use of three visualizations. These visualizations are part of EVolve's built-in library, and so are immediately available to end-users. Refinement of the framework was done through testing with novice users in the context of a project analyzing runtime behaviour of object-oriented programs. This experience resulted in various user interface improvements, now incorporated in the current version. We found that there is added value in being able to examine and compare a several different visualizations of the same data source at the same time.

Extensibility was demonstrated by adding a new, custom visualization. This was a straightforward process, requiring relatively little coding and minimal time (a few hours). Adding new data sources is similarly easy; we have used EVolve with traces generated from JVMPI, a customized Java virtual machine, and several obscure internal formats.

We plan to continue to extend EVolve's repertoire of visualization techniques, and test these on more data sources. Since extensibility is built-in, the core of the EVolve platform does not need to change. We are also investigating other user-interface and comparison techniques that may improve comprehension of the resulting visualizations.

# 8 Acknowledgments

This research is supported by grants from NSERC, FCAR, and the McGill Faculty of Graduate Studies and Research. The authors would also like to thank Bruno Dufour, David Eng, and John Jorgensen for their valuable input.

## References


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

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

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

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

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[6] M. Lanza and S. 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.

[7] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, pages 326–337, 1993.

[8] S. 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.

[9] S. F. Roth, P. Lucas, J. A. Senn, C. C. Gomberg, M. B. Burks, P. J. Stroffolino, J. A. Kolojejchick, and C. Dunmire. Visage: A user interface environment for exploring information. In *Proceedings of Information Visualization, IEEE*, pages 3–12, 1996.

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

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

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

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