SHADOW

A framework for creating high-level program analysis tools

Arie Gurfinkel

arie@aguga.net

Eugene Nudelman

eugnud@home.com

SHADOW: A framework for creating high-level program analysis tools

by Arie Gurfinkel and Eugene Nudelman

Published April 21, 2000

This report summarizes the work that has been done in the undergraduate project in computer science by Arie Gurfinkel, and Eugene Nudelman at the University of Toronto. The project was conducted under the supervision of Prof. Marsha Chechik.

There are currently a large number of tools available to analyze programs. These tools are very diverse and range from simple profilers to very complicated algorithm visualization and software verification packages. However, as it stands now, there is no general framework to aid in creation of such tools. These tools are usually hard to create and can be even harder to use. They are targeted towards large applications and tend to have a very steep learning curve. During this project we have addressed the needs of a program developer, who requires a set of simple tools to create a custom program analysis package. The result of this project is a framework called SHADOW. This report describes the framework and presents a number of case studies we have conducted to evaluate its potential.

Table of Contents

List of Tables

List of Figures

List of Examples

Chapter 1. Introduction

There exist a lot of tools that do some kind of software visualization, debugging, and verification or, in general, analysis. These tools are created in a different way and have a different internal architecture, even though a lot of things that they do internally are essentially the same. Hence, in general, each new tool from this family must be created practically from scratch, which makes creation of new program analyzers extremely hard. Nevertheless, it is often desirable, especially during the development of most moderately large software systems, to be able to create such tools on demand. SHADOW is a framework that aids in creation of program analyzers, debuggers, visualization and related tools. In itself it is neither a debugger nor a program analyzer, but rather a design guideline for a set of tools that allow one to rapidly create and/or customize program analysis software.

Tools that aid in analyzing programs serve a multitude of functions and are known by many different names, such as debuggers, profilers, program analyzers, visualizers and verifiers. For simplicity, we will use the term *program analyzer* to refer to all such tools. These tools range from a well-known C debugger `gdb`[2] [1] to a sophisticated algorithm animator package XTango[3]. The program analyzers can be divided into three categories based on their level of abstraction. new

1. **Machine-level.** Machine-level debuggers provide the machine view of a program. They achieve that by providing the machine or assembler code representation of the program. The user is then given the ability to examine the code and machine state as the execution of the program proceeds.

   These tools are useful to track low level program execution in order to locate a low level problem. For example, one can find out if a register overflow was a cause of the problem. However, since they loose most of the information about the high-level language in which the program was written, they are not a good choice to analyze high-level programs.

2. **Source-level.** Source-level debuggers provide the programming language view of a program. They present the source code of the program at hand and show what occurs during the program execution from the programming language point of view. Unlike the machine-level tools, they are aware of such language constructs as functions, classes, variables, etc.

   The source-level debuggers are very widely used and can be indispensable in the right circumstance. However, since they concentrate on the programming language view of the program, they do not attempt to represent the logical structure of the program as a programmer may see it.

3. **High-level.** High-level debuggers provide the programmer's view of a program. They try to represent the program at a higher level of granularity by concentrating on such things as message passing and data flow.

   Since there are many different logical representations of a given program, there is a wide variety of such tools. They range from profilers, software visualization packages, and other "ready-made" tools, to tools created by the programmer for a specific task, such as special print routines to illustrate the operation of the message passing mechanism. This category essentially includes all other tools that do not fit in the first two categories. The major problem with tools in this category is that there is no standard framework to create them. As a result, many of these tools do not allow any communication with each other or with external programs. It is usually impractical to use more than one of them at any given time.

1. For a detailed definition of this and other terms in this text refer to the bibliography and glossary sections at the end.

Figure 1-1. A high-level program analyzer.

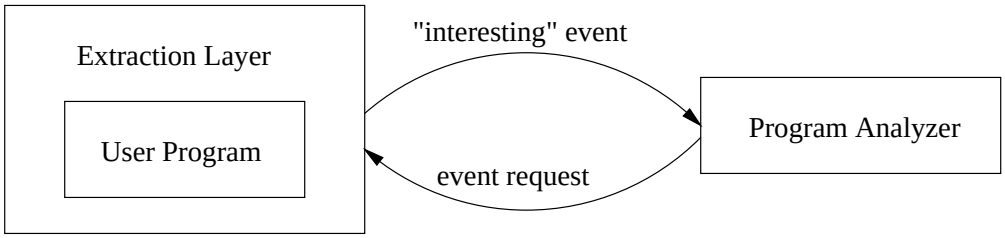
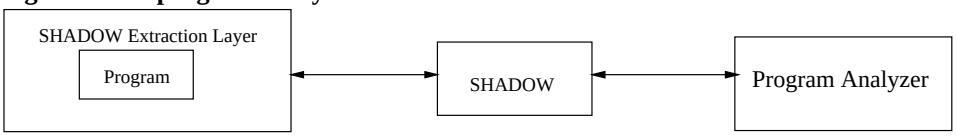


Figure 1-2. A program analyzer that uses the SHADOW framework.



Unlike high-level program analyzers, machine- and source-level program analyzers have well-defined goals and structure. Machine-level tools map the machine state during the program execution into standard terms such as machine instructions, registers and memory. The source-level tools map the same machine state into programming language concepts such as objects, functions, variables and statements. Therefore, the creation of such program analyzers is well standardized. For example, Java includes its own standard framework, JPDA[4], to construct such program analyzers. On the other hand, high-level tools map the machine state and the source code of the program into an abstract domain. It would seem infeasible to construct a framework to satisfy all of the requirements posed by all such tools. In fact, most of high-level program analyzers are constructed from ground up, essentially recreating many of the components of other similar tools. SHADOW is an attempt to create a framework to address this problem.

Essentially the construction of high-level program analyzers can be divided into two phases. First, the extraction layer is constructed. This layer is used to extract relevant information from the source code of the program, and then to inform the program analyzer about *interesting* events that occur during the program execution. The second phase is the actual creation of the program analyzer based on the information provided by the extraction layer. This is outlined in Figure 1-1. It then follows that no matter how different the high-level program analyzers are, most of them extract virtually the same information from the program to perform their analysis. However, the extraction layer is usually custom-made for the program analyzer at hand. This makes it difficult to use more than one such tool at the same time. It also makes the creation of such tools complex, even if the final result they produce is trivial. Since the extraction layer is custom made and is not the most important part of the tool, it tends to be customized to one specific language. This means that a completely generic tool, such as an algorithm animation package, becomes language dependent and can only be used with programs written in one specific language. Of course it is possible to use frameworks designed to create source-level program analyzers to construct the extraction layer. However, unlike the source-level program analyzers, high-level program analyzers require not just the information about the source code of the program and current state and changes to it during runtime, but they also need to know the structure of the program, such as dependency graphs and history of all changes, to perform an in-depth analysis. SHADOW is the framework that encapsulates this extraction layer by providing all of the information required by high-level program analyzers. It works by extracting the information from the underlying program and creating a *shadow* of this program to be used by program analyzers. The program analyzers then have a database-like access to this information, as well as an ability to register to be notified of changes to this information. The program analyzers essentially analyze the *shadow* of the program rather than the program itself. This simplifies the creation of such tools as well as enables to use more than one such tool at the same time. Since SHADOW defines a strict structure for this information it opens up new possibilities by allowing to use generic data analysis tools, such as

spreadsheets, to be used as high-level program analyzers.

In the rest of this chapter we will concentrate on describing the general requirements to the framework, as well as different implementation alternatives to satisfy these requirements.

We posed the following requirements on the framework in order to ensure its usability and extensibility.

a. **Language Independence.** If possible, the framework must not depend on any particular language. That is, it should be possible to analyze programs written in any combination of different languages. The following are the reasons to uphold this requirement:

   - many systems consists of *components* that are written in different languages. These components, however, constitute a *single* system. It is clearly highly desirable to analyze such systems in a uniform, and hence language - independent, manner.
   - there are many languages which are sufficiently close to each other. Moreover, software that performs similar tasks may be written in different languages. It is undesirable to have different tools for processing sufficiently similar things.

b. **Platform Independence [2] .** This requirement follows directly from the language independence requirement. The software that needs to be analyzed may run on many different platforms. Moreover, systems may be distributed over different platforms as well as languages. The locus of program execution must not affect the way in which it is processed.

c. **Ease of Extensibility.** While the above two requirements are highly desirable, it is impossible to create something that works without change in all situations. Therefore, the framework must be extremely flexible, and allow for extension and customization of its features. For example, it must define communication protocols for its modules, rather than the modules themselves, as those might have to be replaced. For instance, the framework requires some module to generate "interesting" events that occurs within the debugged program. There are many ways to achieve that - from instrumenting source code to using some debugging API. The rest of the framework must be insulated from the implementation of this module.

d. **Ease of Data Access.**
   - Since the primary purpose of SHADOW is to facilitate the creation of program processing tools, it must store and provide access to the information about the debugged program. In order to make the framework usable, data access must be as simple as possible.
   - There are many existing tools which are not designed for processing data about programs, but which nevertheless could be used for that purpose. Therefore, data access mechanism must facilitate data conversion to other formats.

The major goal of the framework is to provide an easy access to the information required to create a high-level debugger. Therefore, our major design concern was to create an extensible data structure to store that information. This design can be separated into two parts: designing the structure, and designing the possible implementation. The implementation of the data structure was the crucial part of this project. We did not believe that we could create a structure rich enough to accommodate every need, and therefore our goal was to implement the structure in such a way that it would be easily extendible. At the same time, the data structure had to be easy to use and had to represent the information in a *natural* way.

The information that is represented in the SHADOW framework comes from two sources: the information available at compile time (which comes ultimately from the source code of the program), and the execution

2. Note that this does not mean that the framework itself *must* be able to run on any platform.

state of the program. We will refer to the information that is obtained from the source code as *static*, and the information obtained from the execution state as *runtime*. It is customary to think about the source code of a program in a hierarchical way. In object-oriented languages the program is usually divided into namespaces, which are then subdivided into classes or modules, and then fields and methods. In non-object-oriented languages the same division occurs, albeit the terms are somewhat different. Therefore the most *natural* data structure to represent the *static* information is a tree. The *runtime* information can also be structured as a tree. In this case each runtime instance (of a class) can be thought of as a top-level parent node that has child nodes representing its fields and methods. A change in the state of a program during its execution results from either an addition of a new instance, a change of a field value, or an execution of a method. These changes can be recorded in the tree either by adding a new instance node, or by modifying a field or method node of an existing instance node. Therefore, the *natural* data structure to represent the *runtime* information is also a tree. So the basic building block for SHADOW's data structures is a tree. The only problem with using just a tree is that it may be necessary to record some information more than once. To solve this problem we introduce a notion of a *reference* node to a generic tree data structure. A *reference* node is a node that acts as a pointer to another node of the tree. This allows to reduce the size of the tree, by making it into a graph. However, we would still prefer to think about the structure as a tree, and assume that all *reference* nodes are expanded.

We have imposed the following conditions on the implementation of this data structure:

1. **Platform and Language Portability.** The data structure must be portable across languages and platforms. In order for the system to be useful, it is important that the data structure is platform- and language-independent. This will allow to use the framework in different environments. However, the greatest benefit is that it does not constrain the language for data analysis to the one in which the program is written or the framework is implemented.
2. **Disk representation.** The data structure must have a representation that is easily serializable to disk. This allows for greater flexibility, since then it is possible to use other tools, that cannot be augmented to access the framework directly, to analyze the data. For example, the stored data structure can be converted to the format supported by a spreadsheet program. The spreadsheet program can then be used to create charts and graphs, as well as to perform comparisons between different sets of this data.
3. **Ease of use.** The external API must be easy to learn and easy to use. If possible, the API must follow commonly accepted guidelines, since this will simplify its use and eliminate a sharp learning curve.

Now we will describe the alternatives that we have considered to implement the data structure. Since we have made a decision to implement the framework in Java, our first approach was to design the data structure as any other Java data structure. In this case, the data structure is implemented as a tree where each node of the tree is represented by a Java object. The advantage of this implementation is that it makes the data structure quite intuitive to use to anyone who is familiar with object-oriented languages. Since Java is platform independent, it would make the structure platform independent as well. However, it would also make it more complicated to access the structure from any language other than Java. This approach also does not define a disk representation for the structure. Of course it is possible to develop a proprietary disk representation. However, that would impact the ability to use tools that were not designed with our framework in mind to access the information.

In order to combat that dependence on Java without affecting the benefits of the above approach, it is possible to use CORBA[5] to implement the data structure. The only change that would have to be made is that the structure would have to be specified in an Interface Definition Language (IDL[6]) first, and then implemented in Java. This approach would eliminate the dependence on Java and allow for greater flexibility in the design of the data structure. This would also allow for the implementation of the framework to be done in a combination of CORBA-supported languages. However, this approach still has the same

drawbacks. It will require us to specify a custom set of API to access the data structure, and it still does not solve the problem of disk representation. In addition, it will require a dependence on CORBA, which is a heavyweight framework that is clearly not suitable for creation of small program analyzers.

An alternative approach is to concentrate on the disk representation first, and then create the runtime structure. We have decided to use XML[7] for this purpose. This means that first we devise an XML-based language to describe the required information, and then concentrate on how to work with documents written in this language. XML-based languages are very well suited to describe tree-like semi-structured information. There is also a wide variety of tools to work with XML documents. Therefore, using XML as the disk representation seems like a very good idea. Moreover, there is a standard set of APIs, called Document Object Model (DOM[8]), that can be used to access XML documents at runtime. These APIs have been designed to satisfy all of the goals we have outlined and are very well suited for our purposes. Thus using XML to implement the data structure gives us the following benefits: standard disk representation, well-known set of APIs, and an ability to implement the data structure in Java, as we intended. Even more so, the conformance to the DOM API does not mean that we cannot implement a more suitable custom set of APIs as well. In this case, our data structure will posses two sets of APIs: one that is designed to access XML documents and is well known and standardized, and another one, designed specifically for our data structure that provides a more convenient way to access it. XML also addresses the issue of extendibility of the data structure. Since the specifications of an XML language can be easily changed without affecting the implementation of the data structure, this makes it much easier to extend the data structure to contain more information than was originally planed.

The last alternative is the most promising one, and we have decided to implement SHADOW's data structures using this approach. The rest of this report describes the work that we have done on this project. It is divided into the following chapters.

Chapter 2. This chapter describes the design of the framework. It is not intended to be a design specification, but rather an outline of the design that we have done. It highlights the most interesting and difficult parts of the design. The complete design specification will be available on the SHADOW's web site (http://shadow.aguga.net).

Chapter 3. As part of our project, we have implemented a simple version of this framework in Java. This chapter describes the implementation. It outlines the difficulties we have and what changes we would like to make to the original design and the implementation in retrospect.

Chapter 4. This chapter concentrates on usage of the framework from the users point of view. It outlines a number of simple examples and describes them in a great detail. The purpose of this chapter is to provide a simple *user guide* for the framework.

Chapter 5. In order to evaluate the usability of the framework we have conducted a number of case studies. During this case studies we have implemented three different program analyzers. This chapter describes these program analyzers as well as evaluates our experience using the framework.

Chapter 6. This is the conclusion chapter.

Chapter 2. The SHADOW framework

Figure 2-1 outlines the major modules of the SHADOW framework, as well as their interactions with each other. This figure presents the general data flow within the runtime part of our framework. All of the rectangular blocks represent modules that we define in our design. The labels for the links describe the types of interaction and the data flow between the modules. The data flow starts in the `EventSource` module which generates `REvent` events that describe the "interesting" occurrences within the executing program. These events are then filtered by the `EventFilter` modules and enter the `EventSink` module for processing. Note that the two modules `StaticInfo` and `RuntimeInfo` are actually database modules (as indicated by vertical bars). In response to each event, `EventSink` queries these two databases and then updates the `RuntimeInfo` database based on the data from the event and the information returned by the queries. Whenever `RuntimeInfo` data structure is modified, events that caused this modification propagate through the data structure, notifying the user listeners. All of the modules and structures mentioned here are described in more detail later in this chapter.

The rest of this chapter is divided into two major parts. First, in Section 2.1, we talk about the data structures that we have decided upon for this framework. We describe three major data structures: static information, runtime information, and runtime events. The second part of this chapter, Section 2.2, describes the five major modules that constitute the framework, namely `EventSource`, `EventFilter`, `EventSink`, `StaticInfo`, and `RuntimeInfo`.

Overall, this chapter is meant to provide a general overview of the design of SHADOW. For complete design specifications see the SHADOW web page (http://shadow.aguga.net).

2.1. Data Structures

2.1.1. Database Structures Overview

The most important part of the SHADOW framework is the data structure in which the information about the program is stored. This data structure will be the interface between SHADOW and external programs that use SHADOW.

The data structure is represented on disk as an XML[7] document, and at runtime as a tree conforming to the DOM[8] interface. This poses a number of restrictions on the implementation of the data structure. An XML document is composed of XML tags that are nested in each other to form a tree. The DOM standard specifies that each tag in an XML document is represented by a node of type *Element*. These nodes are then connected to form a tree. For example, the XML document in Figure 2-2 would have a tree representation as shown in Figure 2-3. In Figure 2-2 the round boxes represent *Element* nodes that correspond to XML tags, and the solid lines show how these nodes are connected into a tree.

Each XML tag may also have one or more attributes. For example, the XML document in Figure 2-2 can be extended by associating an attribute *name* with each tag. The new document is shown in Figure 2-4. Each attribute is given by a key-value pair, and since XML documents are just text documents, both the name and the value of an attribute must be strings. The DOM tree represents the attributes of each XML tag by adding an *Attribute* node for each attribute to the corresponding *Element* node. The tree for the document in Figure 2-4 is given in Figure 2-5. In this Figure, the ovals represent the *Attribute* nodes. It is possible to specify the legal structure on an XML document using an XML Schema. This schema specifies what are the legal tags of a document, as well as their attributes. It also allows to specify the type

Figure 2-1. The SHADOW framework.

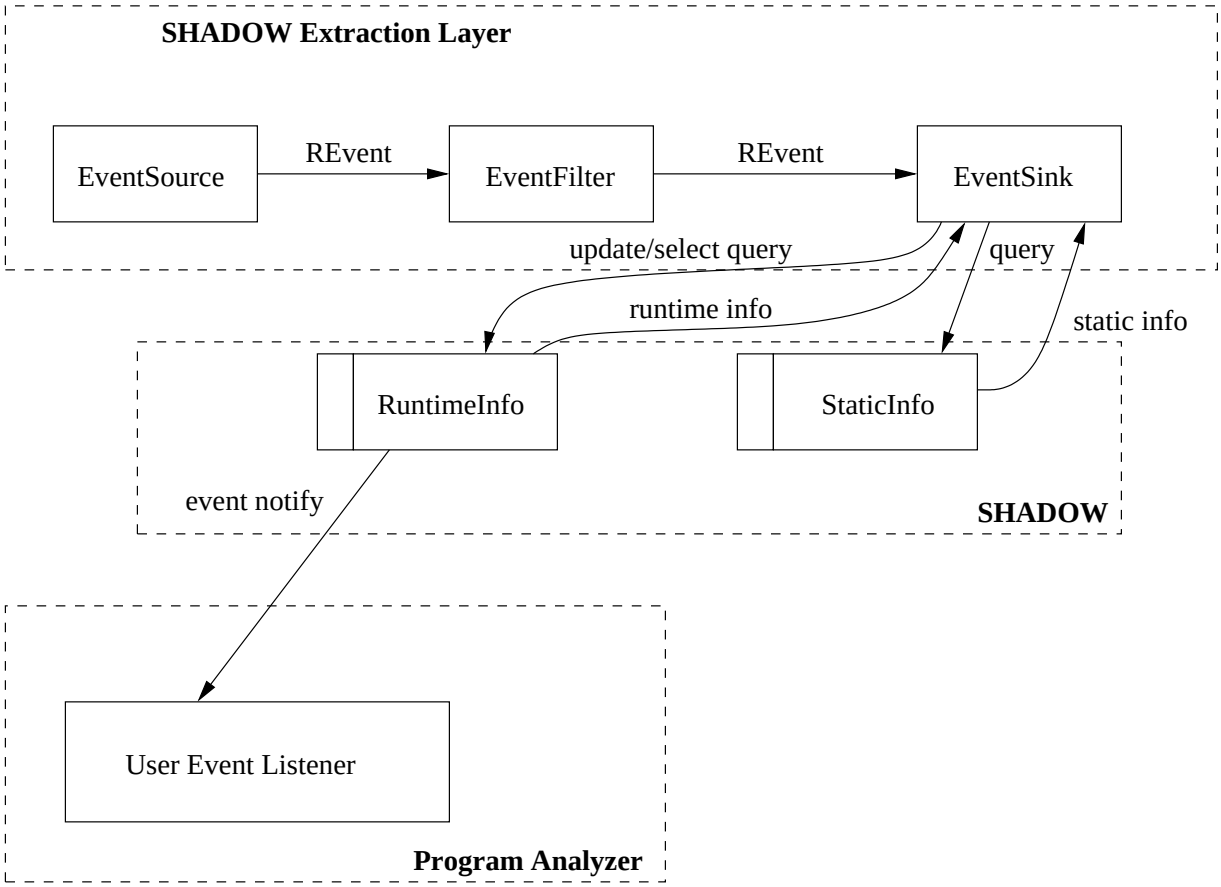


Figure 2-2. An XML document.

```
<class>
 <extends/>
 <field/>
 <method/>
</class>
```

Figure 2-3. The tree representation of the XML document in Figure 2-2.

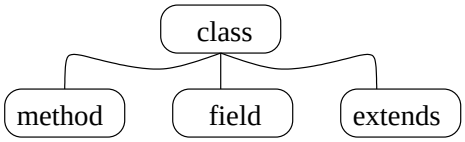


Figure 2-4. Attributes and XML tags.

```
<class name="MyClass">
 <extends name="MySyperClass"/>
 <field name="bar"/>
 <method name="foo"/>
</class>
```

Figure 2-5. DOM tree for Figure 2-4.

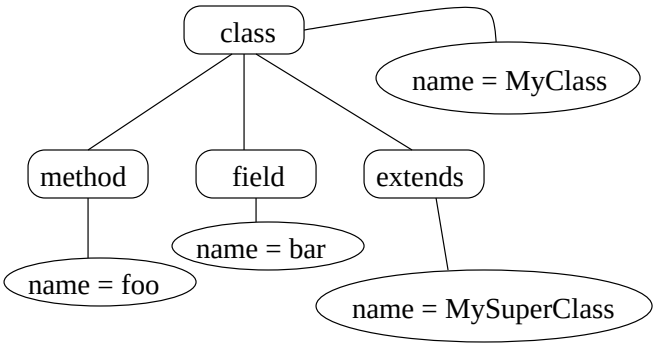


Figure 2-6. An XML view of a *reference* node.

```
<class name="java.lang.String">
  <extends name="java.lang.Object" href="Object.xml"/>
  ⋮
</class>
```

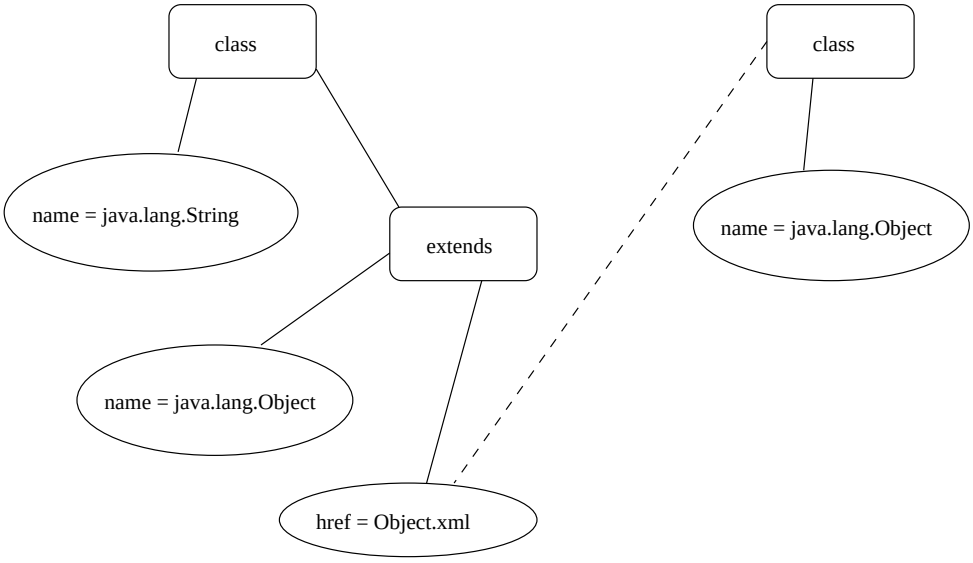
of each attribute, and the rules to convert between this type and its string representation. So although the fact that the attribute value must be a string may seem like a restriction on the data structure, in reality it is not. Therefore, in our further discussion, we will refer to the value of an attribute as being of a specific type, rather than mention that its value is actually the string representation of that value.

The data structure that we represent in XML specifies that the tree representing it must be able to contain nodes that point to other nodes in the same tree. The XML specifications do not specify a standard way to represent this in XML and therefore we have adopted the following convention. A node is a *reference* node if and only if it contains two attributes: *name* and *href*. The *name* attribute contains the name of the reference, and the *href* attribute contains the reference itself as a URI[9]. Notice that any node can be a *reference* node as long as it has the two required attributes. In the simplest case an implementation can use the URL form of the URI specification. Following this definition the *reference* node can be a pointer to the XML document containing it, but it also can be a pointer to some external entity such as another XML document, or even a part thereof. For example, to indicate that a class has a super class, we define a *reference* node called *extends* that points to the description of the super class. In this case the part of an XML document for Java's class `java.lang.String`, which outlines that its super class is the class `java.lang.Object`, would look like Figure 2-6 and its tree representation would be as in Figure 2-7. On that picture the dashed line represents the connection between the *reference* node and its target.

The data that SHADOW represents can be divided into two logical parts: static information and runtime information. The static information is the information obtained from the source code of the program. It contains the structure of the program along with all of its defined namespaces, classes, methods and fields (or functions and variables in case of a non-object-oriented language). The runtime information is the information about the changes in the runtime states of the program during its execution. It contains information about currently loaded classes and their instances. This information includes the values of all fields of each instance as well as the information about executing methods. It is convenient to have a separate data structure to represent each type of information.

2.1.2. Static information

Figure 2-7. Reference node.



The Table 2-1 summarizes the nodes that are used to describe static information.

Table 2-1. Static information nodes

Name	Attributes	Description
package	**name**	**Package descriptor**
packageRef	**name, href**	**Package reference**
class	**name**	**Class descriptor**
classRef	**name, href**	**Class reference**
extends	**name, href**	**Reference pointing to the super class**
modifier	**name**	**A modifier**
field	**name**	**Field descriptor**
fieldRef	**name, href**	**Field reference**
method	**name**	**Method descriptor**
methodRef	**name, href**	**Method reference**
typeRef	**name, href, array, primitive**	**A type reference**
returnValue	**name, href, array, primitive**	**A type reference for a return value of a method**
argument	**name**	**Method's argument descriptor**

Note: Currently the types of the nodes to describe static information are biased towards Java programs. The work is underway to expand them to other languages such as C++ and Python.

The nodes are divided into the following categories:

1. **Namespace nodes.** Namespace nodes describe a namespace as defined in the source language. In Java, namespaces are defined through the use of packages and classes. They are represented by the corresponding *package* and *class* nodes. Namespace nodes carry the least amount of information among all other nodes and act as dividers for the rest of the program. Namespace nodes can contain other namespace nodes depending on their definition. For example, in Java programs, *package* node

Figure 2-8. A simple Java class.

```
public class Simple extends java.lang.Object
{
  // - an int field
  private static int i;
  // - a field of type String
  String s;

  // - a method with arguments and return value
  public int add (int a, int b) throws Exception
  {
    if (a < 0)
      throw new Exception ("a is " + a);

    i = a + b;
    s = String.valueOf (i);
    return i;
  }
}
```

The following example illustrates the description of a simple Java class. Figure 2-8 is a description of a class in Java, Figure 2-9 is the same class represented in XML and Figure 2-10 is the representation of the same class as a tree. Note that Figure 2-10 is greatly simplified and does not show node attributes.

Most of the static information structure is self-explanatory. However, special care must be taken with *typeRef* and *returnType* nodes. As their name suggests, they are used to represent the type as it is defined in the source language. The types in a language are divided into two groups: primitive types, and non-primitive types. In our definition primitive types are those that do not have an XML representation in our framework, the others are non-primitive. The kind of referenced type is specified by the *primitive* attribute of the type node. If a type is primitive, and thus has no XML representation, then the *href* attribute of the type node must be empty. Otherwise, it must point to the XML description of that type. Type nodes also have another special attribute: *array*. This attribute can have any positive number and indicates if the given type is an array. A positive value of this attribute is taken to represent the dimensions of the array; a value of 0 indicates that the type is a scalar.

2.1.3. Runtime Information

The Table 2-2 summarizes the nodes that are used to describe runtime information.

Table 2-2. Runtime information nodes

Name	Attributes	Description
root	***none***	**The root of the runtime tree**
classes	***none***	**The root of instances sub-tree**
threads	***none***	**The root of threads sub-tree**
thread	**id**	**Thread descriptor**
threadMethod	**instanceId, methodId**	**Active method of a thread**

Name	Attributes	Description
rclass	**id**	**Class instance descriptor**
robject	**id**	**Object instance descriptor**
rfield	**id**	**Field descriptor**
values	***none***	**Container for field values**
value	**time, val**	**Describes a single field value**
rmethod	**id**	**Method descriptor**
threadInfo	**threadId**	**Container for thread-related info**
methodInfo	**started, finished**	**An entry in the *threadInfo list describing one method execution***
caller	**instanceId, methodId**	**The caller method**
returnValue	**val**	**Return value of a method**
argList	***none***	**Argument list**
argValue	**val**	**Argument value**
calledMethods	***none***	**List of called methods**
called	**methodId, instanceId, time**	**One entry in the *calledMethods list***

The hierarchical structure of the runtime information is outlined in Figure 2-11.

Runtime information structure is divided into two parts: classes and threads. Children of the *classes* node describe the runtime state of classes and their instances. Children of the *threads* node describe active threads. Currently the structure is designed to support only one thread, with a minimum support for multiple threads. The only information stored about each thread is its id and an id of its currently executing method. The rest of our description concentrates on the nodes that are descendants of the *classes* node. The sub-tree rooted at the *classes* node resembles the structure of the static information tree, with a few adjustments. The *rclass* nodes are not divided into packages as *class* nodes in the static information tree. There is also a new node, *robject*, that is similar to the *rclass* node, and represents an instance of a class at runtime. The structures of the sub-trees rooted at *rclass* and *robject* nodes are essentially the same. The only difference is that the *rclass* nodes can contain *robject* nodes as children, whereas *robject* nodes cannot.

The *rclass*, *robject*, *rmethod* and *rfield* nodes have an *id* attribute that identifies them uniquely in the tree. The *id* attribute can serve as a primary key to this tree. In addition to that, these nodes must contain one special reference node as their child. This node contains a reference to the static information about them. This is done in order to eliminate duplication of data from static information tree in the runtime information tree.

Runtime information structure contains the information about changes to the state of the program during its execution. It is required that the structure is able to store not just the current state of the program, but a historical information about how the program got to that state. In order to store this information in a structured way, we have introduced a notion of *time*. Each change to the runtime information is then indexed by the time in which it has occurred. In our definition, a unit of time is defined by one *atomic* change to the runtime information tree, where an atomic change is a change in the state of the program that resulted from one step of the program's execution. With multiple threads it is possible that

Figure 2-9. XML representation of a class in Figure 2-8.

```
<class name="Simple">
  <extends name="java.lang.Object" href="#/java/lang/Object"/>
  <modifier name="public"/>
  <field name="i">
   <typeRef primitive="true" array="0" name="int"/>
   <modifier name="private"/>
   <modifier name="static"/>
  </field>

  <field name="s">
   <typeRef primitive="false" array="0" name="java.lang.String"
            href="#/java/lang/String"/>
  </field>

  <method name="add">
   <modifier name="public"/>
   <returnType name="int" primitive="true" array="0"/>
   <exception name="java.lang.Exception" href="#/java/lang/Exception"/>
   <argument name="a">
    <typeRef name="int" primitive="true"/>
   </argument>
   <argument name="b">
    <typeRef name="int" primitive="true"/>
   </argument>
  </method>
</class>
```

Figure 2-10. Simplified tree view of the XML representation from Figure 2-9.

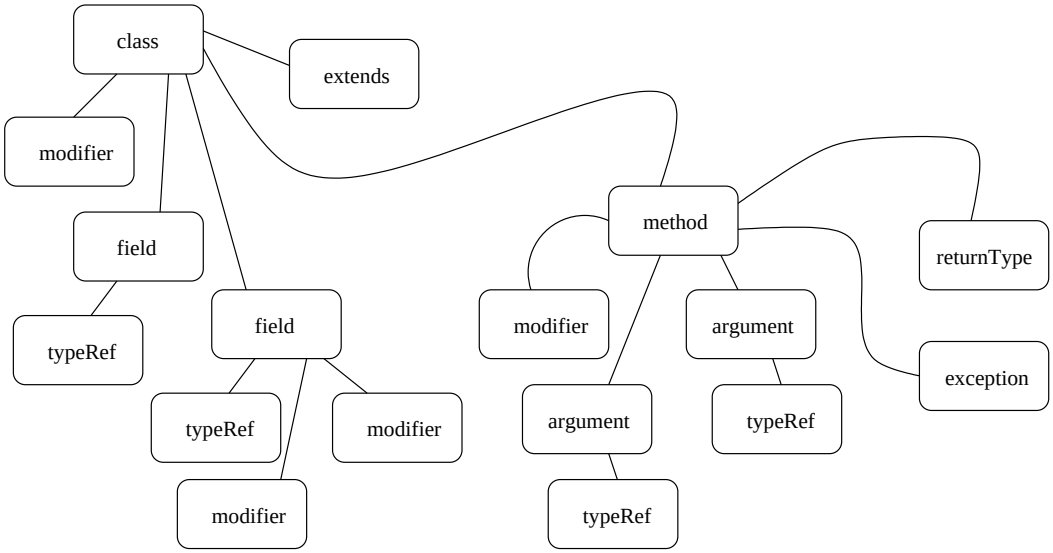


Figure 2-11. Hierarchical structure of the runtime information.

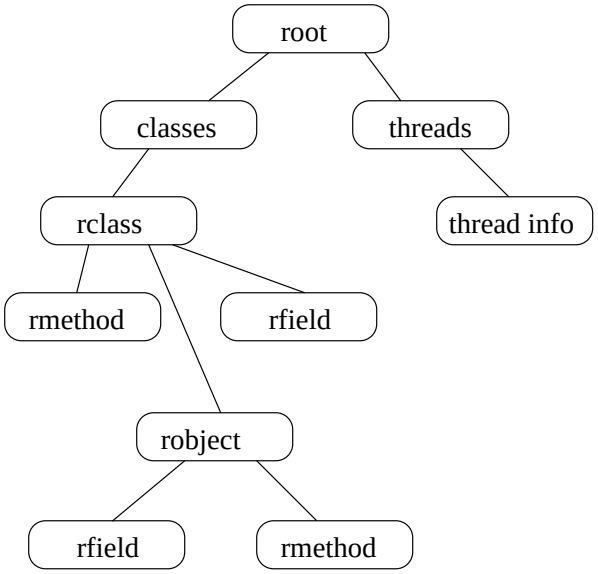
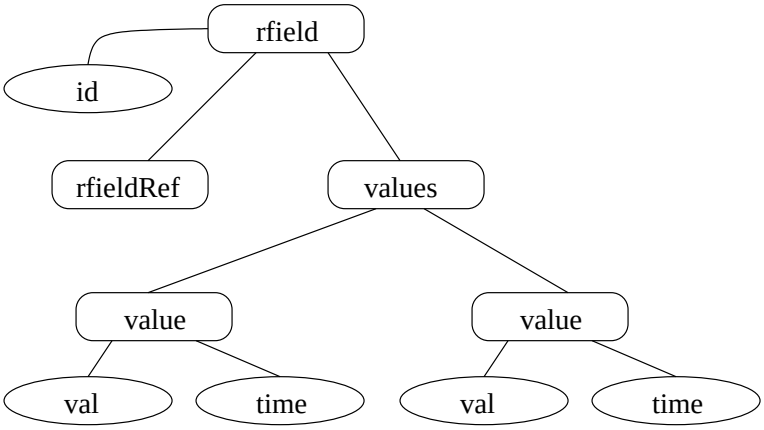


Figure 2-12. RField.



an atomic change results in more than just one modification of the runtime tree; in that case, the time is still incremented by one unit as expected.

The information about changes to the class or its instance is contained in the sub-trees rooted from *rmethod* and *rfield* nodes of this class or object. These trees have a special structure to provide an easy access to that information. The following is the detailed description of these sub-trees.

2.1.3.1. RField sub-tree

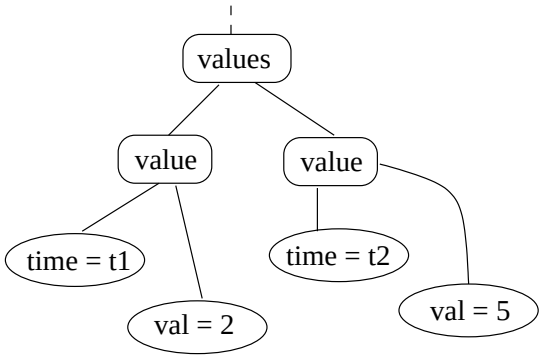
The sub-tree rooted at *rfield* node contains the information about the changes to a field of a class or an object during program's execution. Its general structure is given in the example in Figure 2-12.

The most interesting part of *rfield* is the sub-tree rooted at *values* node. This sub-tree contains the information about the changes to the value of that field. Each change is recorded in a corresponding *value* node. *Value* nodes have two attributes: *time* and *val*. The *time* attribute contains the time at which the field acquired this value. The *val* attribute contains the string representation of that value. If the type of the field is *primitive* then the string representation of the value is defined at the implementation stage, otherwise the value contains the *id* of the value that must be represented as a *rclass* or a *robject* node in the runtime

Figure 2-13. Another example program.

```
    public class Example
    {
      public static void main (String[] args)
      {
  5     Simple s = new Simple ();
        s.add (1, 1);
        s.add (2, 3);
      }
    }
 10
```

Figure 2-14. The values of field `i` after the execution of the program in Figure 2-13



information tree. Also it is important to remember that the tree only stores the changes that occurred to the field and the time at which the change took place. Therefore, to find what was the value of a field in a given time it is required to locate the *value* node with the closest *time* attribute from the left of the required time.

To demonstrate the above description, consider a trivial Java program in Figure 2-13 which will be used as the driver for our example. All this program does is trigger events associated with the class `Simple` from Figure 2-8. Now let us consider what happens to the field `i` of the instance `s` of the class `Simple` at the point where our program terminates. This field has been changed twice. At a certain time t_1 it has acquired the value *2*, and later, at time t_2, it has acquired the value *5*. Therefore, all of these values would be represented by a tree rooted at the *values* node as depicted in Figure 2-14.

2.1.3.2. RMethod sub-tree

The sub-tree rooted at *rmethod* node is the most complicated part of the runtime information structure. It contains all of the information about the runtime changes of a method, including the time of method entry, its arguments' values, methods called by it, and its return value. The structure of the sub-tree is outlined in Figure 2-15.

The *id* attribute of the *rmethod* node uniquely identifies this method from other methods of a class. In addition to this, *rmethod* node must contain exactly one child node of type *methodRef*, which points to the static information about the method. The rest of the information under the *rmethod* node is subdivided by the *threadInfo* nodes. This is done since each thread has its own call stack and its own runtime information about a method. Each *threadInfo* node contains a *threadId* attribute that uniquely identifies an execution thread. The *threadInfo* node contains multiple *methodInfo* nodes as children, each corresponding to the execution of the method by the given thread. Each *methodInfo* node has *started* and *finished* attributes,

Figure 2-15. RMethod sub-tree.

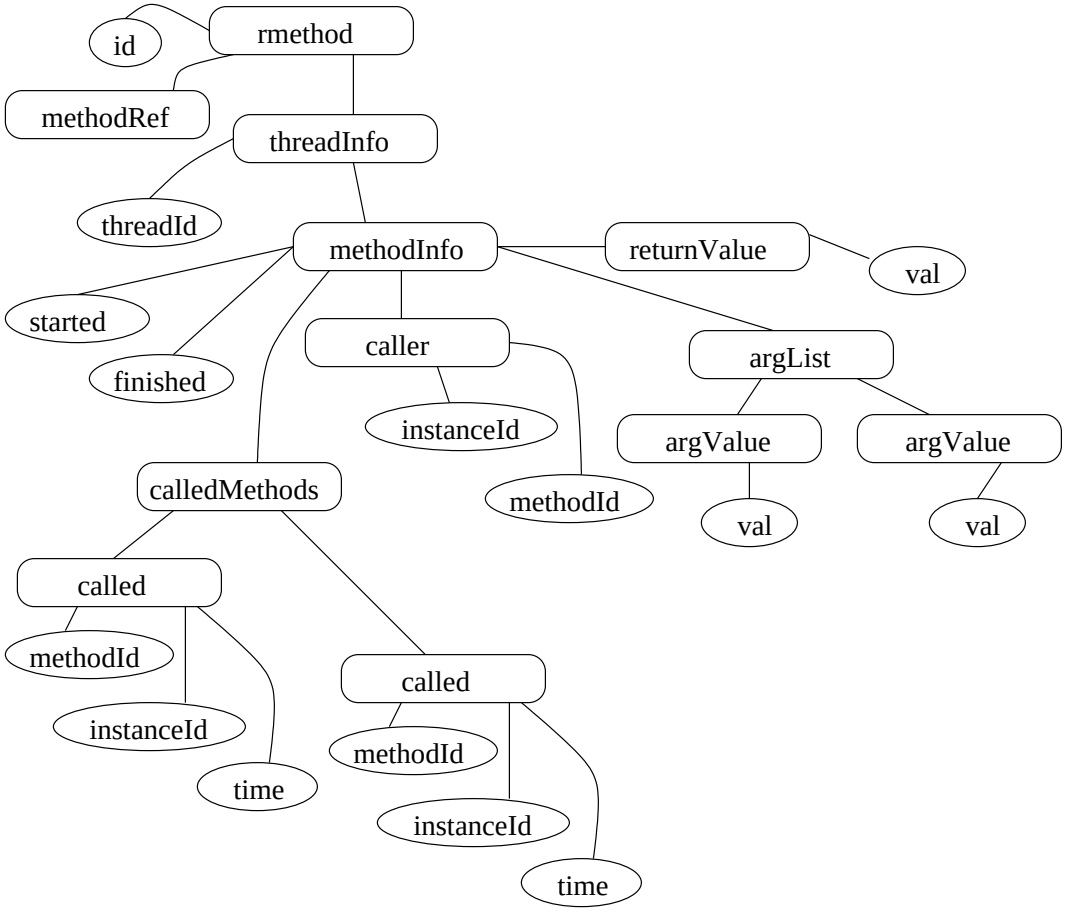
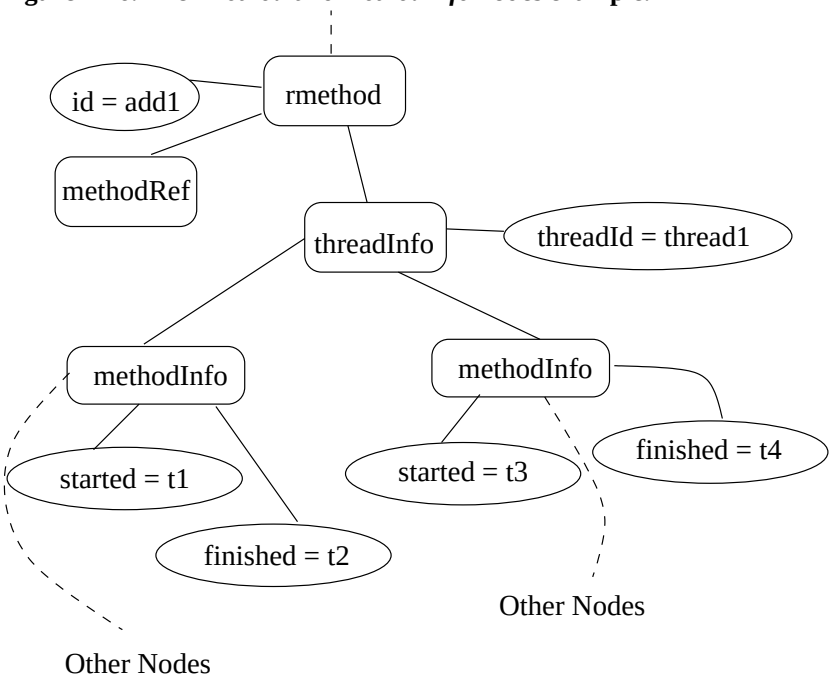


Figure 2-16. The *rmethod* and *methodInfo* nodes example.



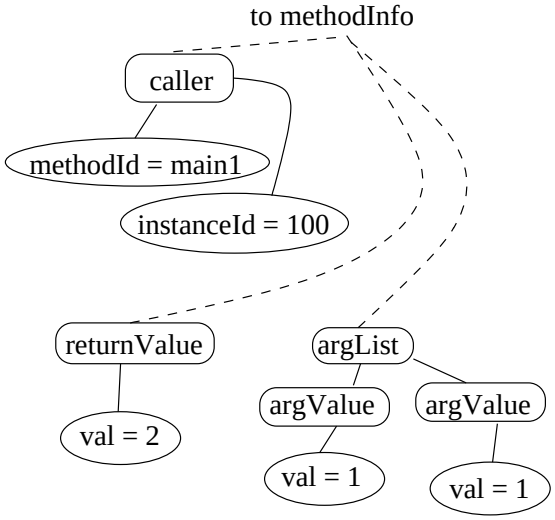
containing the time at which execution of this method begun and terminated, respectively. The caller method is identified by the *caller* node that contains the *instanceId* and *methodId* attributes that uniquely identify the caller method in the runtime information tree. A *methodInfo* node also contains a list of all methods called by the current method. This list is represented as a sub-tree rooted at *calledMethods* node. Each element of this list contains a unique id of the called method, as well as the time at which the call was made. This structure allows for an easy access to the information about any given method at any given time for any given thread. In order to obtain a call stack for a thread, one can to follow the *caller* reference to the caller method repeating that until a call stack of the required depth is constructed. This process will always terminate since there is always a *methodInfo* node that has no child *caller* node. This *methodInfo* node corresponds to the first method ever called by the given thread.

Consider again the program in Figure 2-13. At the end of its execution the method `add` of instance `s` of class `Simple` has been called twice. Since we do not know what has happened between the two calls, we will assume that its first execution has lasted from time t_1 to time t_2, and the second execution lasted from time t_3 to t_4. The *threadInfo* subtree corresponding to this method would then look as depicted in Figure 2-16. In this figure we assume that the method id of method `add` is *add1*, and that the id of the current thread is *thread1*. As the figure shows, the *threadInfo* node has two children of type *methodInfo*, each corresponding to a certain execution of method `add` by this thread. The content of the tree rooted at the *methodInfo* node corresponding to the first execution of method `add` is depicted by Figure 2-17.

2.1.3.3. Runtime Tree Events

In addition to representing the data, the runtime information structure defines a set of events to notify the user about the changes in the data. When a node of the runtime information structure is changed, or a new node is added, an appropriate event is generated at that node. The event is then propagated through the tree until it reaches the root node. This propagation of events is called *event bubbling*. While an event is *bubbled* to the top, it triggers the event handling routines registered at the nodes on its path. This is the mechanism by which the program that is using the SHADOW framework is notified about changes in the

Figure 2-17. The *returnValue*, *argList*, and *caller* nodes example.



program's execution state.

It is also important to notice that the nodes of the runtime tree are never deleted. This is done in order to guarantee that all of the information about the execution of the program is saved. This allows to analyze the runtime information tree *after* the program execution with the same result as analyzing it *during* the program execution. This can create a problem if the structure becomes too large to maintain. However, it is not an issue of the runtime information structure, but rather of the `EventSource` and `EventFilter` modules. These modules can control the number of events that are generated during the program execution. It is possible to restrict these events in such a way as to make the runtime information structure more manageable.

As described above, whenever the `RuntimeInfo` tree is modified, events are generated that propagate through the tree and trigger user's event listeners. The structure of these event objects is very simple. The most important field that they have is the *`sourceNode`* field, which indicates the point in the tree at which the event has originated. Each event also has a *`type`* field, which is the same as the *`type`* field of the events generated by `EventSource` (see Section 2.1.4). Using this field the user may find out exactly what kind of tree modification has occurred.

As described above, event bubbling is an important part of the tree event handling. It allows, for instance, to create some blanket event listeners at the root of the tree that handle all of the unprocessed events. However, sometimes it is convenient to terminate this bubbling, for example in order not to duplicate any work. Therefore, each event also has a flag field *`bubble`*, which is settable by the event handlers and which indicates whether the event should be propagated any further.

As an added convenience users are allowed to attach arbitrary data objects to each event.

2.1.4. Runtime Events

RuntimeEvent structures encapsulate information about various events that occur during the program execution.

As described in Section 2.2.1, events are always grouped into event *batches*. Each batch represents events that are either physically or logically simultaneous. However, for the purposes of processing, some (simul-

taneous) events may occur before others. Therefore, we require an event batch to be a *partially ordered* set of events.

Table 2-3 summarizes the types of events that we currently define and their fields. These events should be adequate for capturing almost all of the runtime information that occurs within the program. However, as with the rest of the framework, new event types may be added on demand.

Table 2-3. Runtime Events.

Event Type	Fields	Description
`ClassLoaded`	*`ClassName, UniqueID`*	**Generated whenever a new class is loaded by the system.**
`InstanceCreated`	*`UniqueID, ClassID`*	**Generated whenever a new object of some class is created**
`FieldModified`	*`UniqueID, FieldID, Value`*	**Generated whenever a static or instance field's value is changed.**
`MethodEntered`	*`UniqueID, MethodID, ThreadID, ArgumentList`*	**Generated whenever a method is called.**
`MethodExited`	*`UniqueID, MethodID, ThreadID, ReturnValue, ExceptionValue`*	**Generated whenever a method completes execution.**

For the most part the names of the fields should be self-explanatory. The most important field for all of the events is the *`UniqueID`* field. This field must uniquely identify the node in the data structure that reflects the object or the class to which the event pertains. For that reason the *`UniqueID`*, *`ClassID`* and *`ThreadID`* must be unique across the `RuntimeInfo` data structure. Note that the *`UniqueID`* field in the method and field events must identify an *robject* node in case of non-static fields and methods and an *rclass* node in case of the static class members.

2.2. Modules

2.2.1. `EventSource`

`EventSource` module is responsible for generating events that occur in the program during its execution. To generate these events it is necessary to extract information from a running program, which can be performed in a variety of ways. Moreover, the actual events that need to be generated might differ across languages, platforms, and applications. Therefore, it makes sense to encapsulate `EventSource` in a separate easily replaceable module.

We wanted SHADOW to be as unintrusive as possible. That is, we wanted the information flow to go *from* the analyzed program, but not towards it. This is particularly important because our goal was to decouple the extraction layer in order to allow several analyzers to be used simultaneously without interfering with each other. This constraint caused the following design decisions:

1. The `EventSource` module is the driver module for the whole framework. As such, it is the only active module, in a sense that it controls the data flow throughout the framework, by pushing appropriate

events forward.
2. `EventSource` and `EventFilter` modules are the only modules that allow user to affect behavior of our framework during run-time. The user has read-only access to the rest of the framework.
3. User may not directly request events to be generated or suppressed. However, the `EventFilter` allows for dynamic control of the number and set of generated events.

Events are always generated in *batches*, rather than individually. The main reason for this is that the data structure must always be left in a consistent state after events are processed. By our definition, each modification of the data structure corresponds to a time step within a program and vice versa. Hence whenever a set of events occurs simultaneously within a program, it must also be processed simultaneously within the framework. (Here simultaneity depends on the definition of an atomic occurrence). Therefore such set of events must be generated as a single unit.

2.2.2. `EventFilter`

`EventFilter` module accepts events from the `EventSource` module and passes them on to the `EventSink` module. In the process of doing so it may add, remove or modify events.

During a realistic program run, a multitude of events is generated. Often many of these events are unimportant for the user's current purposes. Moreover, the number of events may impose an unnecessary strain on the system performance. This module allows the user to alleviate these problems by specifying precisely which events are to be processed. The exact format of such a specification is not a part of our framework, as it depends greatly on each application and/or personal preferences.

It must be noted, however, that care must be taken whenever the event flow is restricted, as it is easy to restrict it too much. Enough events must be output for the `EventSink` to be able to process each event correctly.

We have decided not to specify the exact nature of interaction between `EventFilter` and `EventSource` modules, since it might differ greatly across applications. However, it is obviously beneficial for these modules to cooperate, as in many cases it might be possible not to generate events at all (rather than filter them out), which clearly can improve performance greatly.

2.2.3. `EventSink`

`EventSink` processes the events generated by the `EventSource` and updates runtime information data structure accordingly.

`RuntimeInfo` is a database that provides access to the general data structure for representing runtime information about the processed program. It is the job of the `EventSink` module to update this database. As with other databases, the update algorithms may be different across applications; hence they are encapsulated in this module. For example, different languages may behave differently under similar events (e.g.,Python and Javascript may require non-trivial processing for class members that can be added at runtime). Currently, our `EventSink` module processes events specific to Java and can be adopted to sufficiently similar languages.

`EventSink` must leave the data structure in a consistent state after each batch of events is processed. This generally means that all of the information contained in each batch must be somehow stored (and hence representable) inside the data structure, since this information may often be required for processing a subsequent batch of events. Thus this module must rely heavily on the `EventSource` and the `EventFilter` modules to provide the information sufficient for maintaining this consistency.

2.2.4. `StaticInfo` and `RuntimeInfo` modules

The `StaticInfo` and `RuntimeInfo` modules provide database access to the static and the runtime information data structures, respectively.

Since the data structure is in both cases tree-like, both of these modules should provide access to the root node of the corresponding tree. Once a root node is obtained, it is possible to get to any information stored in the database through a simple tree traversal. (This traversal may be performed via our custom API or the standard DOM API).

Reference nodes constitute a major part of both trees. While it is possible to resolve a reference manually by traversing the whole structure from root down, it is obviously not practical, especially since the format of the URI[9] is currently implementation dependent. Therefore, we require these two modules to also provide a way to resolve reference nodes efficiently.

Above describes a *necessary* and *sufficient* interface that makes these modules usable and all of the information accessible. However, this is clearly a minimal interface that does not take many common queries into account. Therefore as a future improvement it is possible to provide an interface that processes a generic query written in some specialized language. Currently good contenders for such a language are XPath[10], which allows one to specify a precise location of a node or attribute in an XML[7] document, or several languages that are developed in conjunction with specialized XML databases.

Chapter 3. The implementation

The reference implementation of the SHADOW framework is done in the Java programming language. It is currently targeted towards analyzing Java programs. However, we plan to extend this implementation to support other JVM languages such as JPython[11].

The implementation is divided into two parts: the extraction layer, and the static and runtime information database. The extraction layer is the part of the implementation that extracts the information from the executing program. This is the implementation of the `EventSource`, `EventFilter`, and `EventSink` modules. The database part of the implementation implements the actual database where the information extracted by the extraction layer is stored. This is the implementation of the `StaticInfo` and `RuntimeInfo` modules.

3.1. Extraction Layer implementation

3.1.1. Overview

The extraction layer is divided into two major parts, each corresponding to a major function of this layer. First part is responsible for implementing the `EventSource` and `EventFilter` modules. Its job is to actually extract/generate information about an executing program and present it to the rest of the framework in the form of events. The second part implements the `EventSink` module and is responsible for processing those events and updating the `RuntimeInfo` data structure appropriately.

3.1.2. `EventSource` and `EventFilter`

First we'll concentrate on the extraction mechanism proper. The main job of the extraction mechanism is to extract information from an executing program. It is impossible to accomplish such a thing in full generality, and, since we prefer to work in the Java environment, we have decided to concentrate on processing Java programs only. Information extraction can be accomplished in a variety of well-known and fully developed ways: from instrumenting the source code to creating a customized Java Virtual Machine. Our main goal was not to create one more way to extract such information, but rather to show how it can be used. Therefore, we have simply picked a tool that seemed to be the most promising and the easiest to use - the JPDA[4] framework (Java Platform Debugger Architecture), created by SUN. It was designed specifically for the purpose of debugging Java programs. In particular, this framework allows one to inspect a running program, and, more importantly for us, to be notified when some events occur during a program execution. Moreover, there is a pure Java interface (JDI[4] - Java Debug Interface) to this framework; therefore, it can be used without leaving the Java world. As an added benefit, JPDA is designed specifically for remote debugging, so enabling our framework to analyze programs running in a different virtual machine or even on a different network site.

JPDA turned out not to be a perfect match. There are two major reasons for this. First, it is a relatively new framework, that still has many bugs and design oversights. Consider, for example, the code fragment in Example 3-1. Once the last line is executed, JPDA will produce an event saying that `foo` has exited, but it is impossible to find out, from this event, that the return value of `foo` is 5. This is clearly the information that is highly desirable in many contexts, and in particular in our framework.

Example 3-1. Return value example.

```
          public int foo()
          {
           return 5;
          }
          ⋮
  5       ⋮
          x=foo();
```

Example 3-2. Array modification example.

```
     class foo
     {
       int     x;
       int[10] y;
  5  }
     ⋮
     foo obj = new foo ();
     ⋮
            obj.x = 5;
 10         obj.y[1] = 7;
```

As an example of another problem that we have encountered, consider the piece of code in Example 3-2. The problem with this example is that when line 9 is executed, JPDA generates an event indicating that the value of the field *x* has changed. However, when line 10 is executed, no events are generated. Again, a lack of such important information and this non-uniformity are highly inconvenient for most purposes. Thus, our framework currently doesn't handle return values and array modifications well. Since these are two quite important things, it is likely that they will be fixed in the future. As they are not essential to our research, we have postponed addressing these problems for now.

Despite such problems as illustrated above, JPDA is a usable framework with a relatively gentle learning curve and we were able to adapt it seamlessly for most of our purposes.

The set of "interesting" events generated by the JDI corresponds very well to those that we wanted to receive. However, there are two notable exceptions:

1. **New Instance Event.** The SHADOW framework defines that whenever a new instance of a class is created, an *InstanceCreated* event must be fired by the `EventSource` module. JDI specifications, on the other hand, do not define a similar event. Whenever a new instance is created, JDI signals this by firing *MethodEnter* event corresponding to the constructor method of this instance. This allows `EventSource` to generate two SHADOW events (*InstanceCreated* and *MethodEntered*) whenever a single JDI *MethodEnter* event is received. We take extra care since it is possible for more than one constructor to be executed for the same object. Thus two events are generated only for the first constructor of each object.

   This illustrates a useful approach that can be utilized in many implementations. Namely, one does not need a one to one correspondence between SHADOW events and information received from external entities in order to make the framework work.

2. **ExceptionEvent.** Whenever an exception occurs in a Java program, JDI generates an *ExceptionEvent* that contains information about the exception (its type, information contained in it, etc), where it has occured and where it was caught. However, although Java's stack is unwound when an exception occurs, no *MethodExit* events are generated in such a case. We, on the other hand,

would like to receive these events, since it then becomes much either to maintain the data structure consistency, and event handling becomes more uniform. Morever, in our design we have decided that *MethodExited* must contain information on whether the method completed normally or exited with an exception.

As with the previous case, this mapping of JDI events into SHADOW events is easy to implement. The reason for this is that our data structure contains a representation of the call stack at any given time. Hence when an *ExceptionEvent* is received, EventSource may do the following:

1. Query the RuntimeInfo database in order to obtain the portion of the current call stack between the methods that threw and caught the exception. All of this information is quite easy to get from our data structure.
2. For each method in the list returned in the previous step, generate an appropriate *MethodExited* event, containing the exception information.

The crucial point here, is that our data structure stores *all* of the information that has been available to us from the extraction mechanism in an accessible manner, and this allows various modules to deduce facts that are not directly obvious.

In our implementation, the EventSource and EventFilter modules are implemented together. The main reason for this is that JDI allows to specify exactly which events should be generated and which should not. Therefore, it is better to delegate filtering to JDI in the first place, rather than to filter a flood of events. The actual filter interface is quite rudimentary in our prototype implementation. Currently, the user must specify precisely which classes should be reflected in the framework. For those classes, all of the method and field events will be received. This interface is sufficient for the research purposes, but in the future we would like to look into improving it.

3.1.3. EventSink

The EventSink module modifies the RuntimeInfo data structure in response to the events generated by EventSource. Since the design of the runtime information structure specifies precisely what is the structure of the underlying tree, and since generated events provide all of the required information, implementing this module was quite a straightforward task.

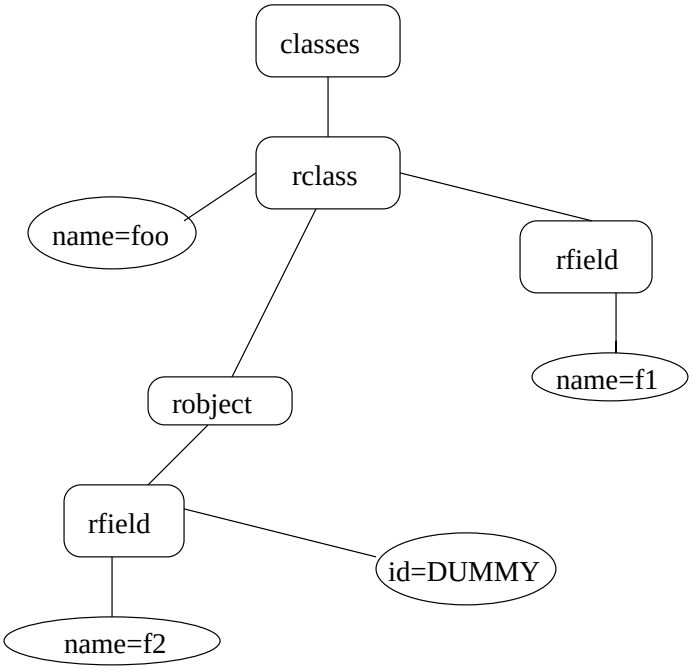
The mechanism for event batches was implemented in this module as follows. The EventSink module provides three interface functions: beginBatch(), processEvent(), and commitBatch(). The EventSource module first signals that a new batch is beginning by calling beginBatch method. It then passes each event in the batch using processEvent method, and then finally commits the whole batch to processing using commitBatch method. We have found this approach to be more convenient than to require from EventSource to package the events in batches and pass whole batches to EventSink. With our approach the EventSink may start processing events before all of the events of the batch have been acquired by the EventSource module.

One interesting optimization was employed for creating class and object nodes. As described in Chapter 2, these nodes always occur with whole subtrees representing member methods and fields of each object/class. The structure of these subtrees is stored in the StaticInfo database; therefore, it has to be queried whenever such a subtree is created. First query occurs when the *rclass* node for each type is created (i.e. whenever a class is loaded by the JVM). The same query should theoretically be done whenever a new instance of such a class is created later on. However, in order to save those queries we always create a dummy subtree corresponding to the instance of each class. Thus, later, when an object instance

Example 3-3. Class loading and instance creation example.

```
    public class foo
    {
     static int f1;
     int f2;
 5
     public static void main (String[] args)
      {
        foo obj = new foo ();
      }
10  }
```

Figure 3-1. `RuntimeInfo` after the *`ClassLoaded`* event.

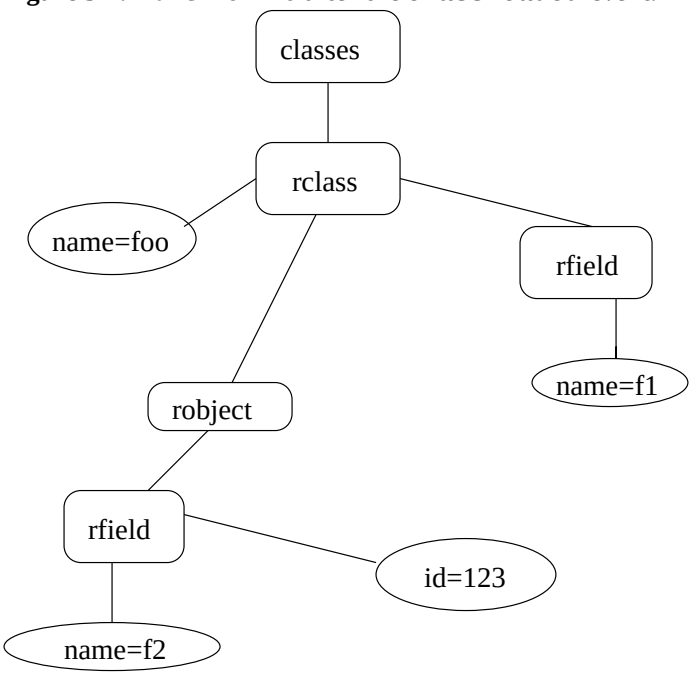


needs to be created, all we need to do is clone that subtree. At the same time. we avoid keeping duplicate information in the `RuntimeInfo` database and we do not pay extra price for querying that information from the `StaticInfo` database.

Consider, for example the piece of Java code in Example 3-3. In order to execute this code, JVM must first load the class `foo`. When this happens, `EventSource` generates a *`ClassLoaded`* event. This causes `EventSink` to modify our data structure as shown in Figure 3-1. Notice that even though no objects are yet created, the *rclass* node already has one child of the type *robject*. This node and its children were created in parallel with the *rclass* node, since in both cases the same query of the `StaticInfo` database on class `foo` must be performed.

Now consider what happens when the first line of the `main` method is executed. A new *`InstanceCreated`* event will be generated, containing an actual instance `id` of the new object. This will cause the tree to change as in Figure 3-2. Notice that for this change it is not necessary to query the `StaticInfo` database. All that is required is to update the `id` of the dummy *robject* node. If later another instance of `foo` is to be created, then the subtree rooted at this first *robject* node can simply be cloned in order to process the event.

Figure 3-2. `RuntimeInfo` after the *`ClassLoaded`* event.



3.1.4. Future Enhancements

Event Filtering. Currently event filtering is extremely rudimentary. User must specify precisely which classes they are interested in, and then *all* events pertaining to these classes will be generated. No finer granularity of filtering is allowed. One reason for this is that events are quite inter-related; therefore, a smart filter must be able to detect whenever not enough events are being generated and warn the user. For example, if class `foo` extends class `bar`, then if a new instance of class `foo` is created, a constructor for class `bar` must be called. Hence our framework must be aware of the existence of class `bar`, and therefore the user cannot request events for `foo` without also requesting events for `bar`.

On the other hand, we cannot allow all of the events to go through, since this is likely to overload the system even for moderately sized programs. It is far from obvious how to strike a balance, or whether it is possible to create a filter that is smarter than what we currently have. One approach is to let the user decide on an appropriate strategy in each case. However, we are planning to look closer into this issue and perhaps create a more reasonable filter interface.

Coping with the lack of information. As mentioned above, SHADOW should see enough events in order to enable handling for other events. For instance, if a *`MethodEntered`* event is generated for a method `foo.bar()`, we expect to have a node in our tree for the appropriate object of type `foo` and several nodes describing method `bar()` of that object. Also we expect to have full information in the `StaticInfo` database concerning class `foo`. This is important even if the user is not concerned with `foo` itself, since the method call stack (which is implicitely stored in the data structure) must go through `bar()`, and hence this event may directly affect handling of other events, that are of more interest to the user. Currently, whenever we encounter a situation where we lack information, the system produces an error message and terminates. The user may then restart it with a different event filtering and a more up to date static information database in order to proceed correctly. However, obviously in many cases it is possible to do better than that. For instance *`FieldModified`* events and to some extent method exit/enter events might often be ignored. Alternatively, it might sometimes be possible to add more nodes to the tree on the fly so that the tree remains consistent and still the new "offending" events can be processed. Also `StaticInfo` database may be updated dynamically. Therefore we plan to look into this issue and provide

a more resonable way to cope with the lack of information in the data structure.

Creating a Sink/Source protocol. This idea was suggested to us mainly by JPDA, which allows the debugged program to run in different JVMs, and at different network locations. This property turned out to be very convenient, and hence we'd like to have it as a feature of our system, although not all debugging architectures support this directly. We discovered another reason for such protocol during one of our case studies. We wanted to run our system mostly inside a web browser. However, JDI, being a relatively new and heavy-weight technology, did not run in that particular virtual machine. Since our `EventSource` implementation must use JDI, it couldn't run there as well. Moreover, due to the heavey-weight and low-level nature of these components, it would not be convenient to run them on the client side. Therefore, we realised that we really need some kind of networked protocol between `EventSink` and `EventSource`. We considered using some kind of RPC framework, such as CORBA[5] or RMI, but these tend to be too heavy-weight and incur too much overhead for our purposes.

One of our initial ideas when starting this project was to be able to record events from program execution on disk and later play them back through our framework. This entails two things: augmenting the current source implementation to store events to disk in some format and creating a new `EventSource` that is able to read those events and pass them on to `EventSink`. The protocol created for the purposes discussed above may also be used for this purpose of serialization of events, since then events can be recorded to disk and played back later using that protocol as the interchange format. Therefore, in the future we plan to create an implementation that allows `EventSource` and `EventSink` to talk to each other through means other than method calling.

Re-modulation of `EventSink`. In our project we have concentrated on Java programs. However, we have always been concerned with the ability to use SHADOW for other languages. We have realised that for each language runtime events and tree representation of a program may be slightly different. Therefore, our initial idea was to create a different `EventSink` module for each language. However, since then we have realised that many languages are so similar that `EventSink` might need only minor (if any) changes in order for it to work with different languages. For instance, rules of inheritance often differ accross languages, while method calling semantics remain more or less the same, as they are more tied up to the underlying platform. Hence, we plan to investigate the possibility of creating an even finer level of modularity for `EventSink` module in order to make adaption of SHADOW to new languages as easy as possible.

3.2. Database Implementation

SHADOW framework provides access to the stored information through the use of two databases. From the user's point of view, the databases appear to be two large XML[7] documents to which the user is granted access through the DOM[8] interface. However, the internal structure of these two databases is quite different. The static information database contains information obtained from the source code of the program. The source code of the program is a structured text document, and, therefore, it is quite trivial to convert it into an XML document. The static information database is a collection of such XML documents. The runtime information database represents the information obtained from the execution of the program. This information may not have a convenient text representation; moreover, it is convenient to provide access to the information in the format it was obtained. Also, while the static information is read-only and does not change during the execution of the program, the runtime information is constantly changing as the execution of the program proceeds. These differences forced us to implement two different database engines. The static information database engine is implemented as a straight DOM interface to which a custom set of APIs were added for the ease of use. The runtime information database is first implemented

as a custom data structure and then extended to support the DOM interface.

3.2.1. Static Information Database

3.2.1.1. User's view of the database

From the user's point of view the database is a large XML document. This document is structured as described in Section 2.1.2. The root of this document can be obtained by calling `getRoot` function of the `StaticInfo` module. The user then has two alternative approaches to search through this document. The first approach is to use the DOM APIs to traverse the tree. The second approach is to use our custom APIs. The first approach allows using XML tools such as stylesheet processors to access the database. The second approach is more favorable when the database is used from a programming language such as Java.

The custom APIs provide the following extensions to the DOM API.

1. Each node of the tree has a `getXXX` method corresponding to each legal attribute as defined by our XML Schema[14]. The `getXXX` method also converts the attribute's value from its string representation to its corresponding type.
2. Each reference node has a `resolve` method that "resolves" the reference node and returns the root node of the sub-tree pointed to by that reference.

When DOM APIs are used to access the database, the reference nodes can be resolved by passing their *href* attribute to the static method `resolve` of the `XmlLinkResolver` class. When traversing the database, the user is expected to resolve each reference node as required. This is done to simplify the underlying implementation of the database and contradicts the idea that the database is one large XML document.

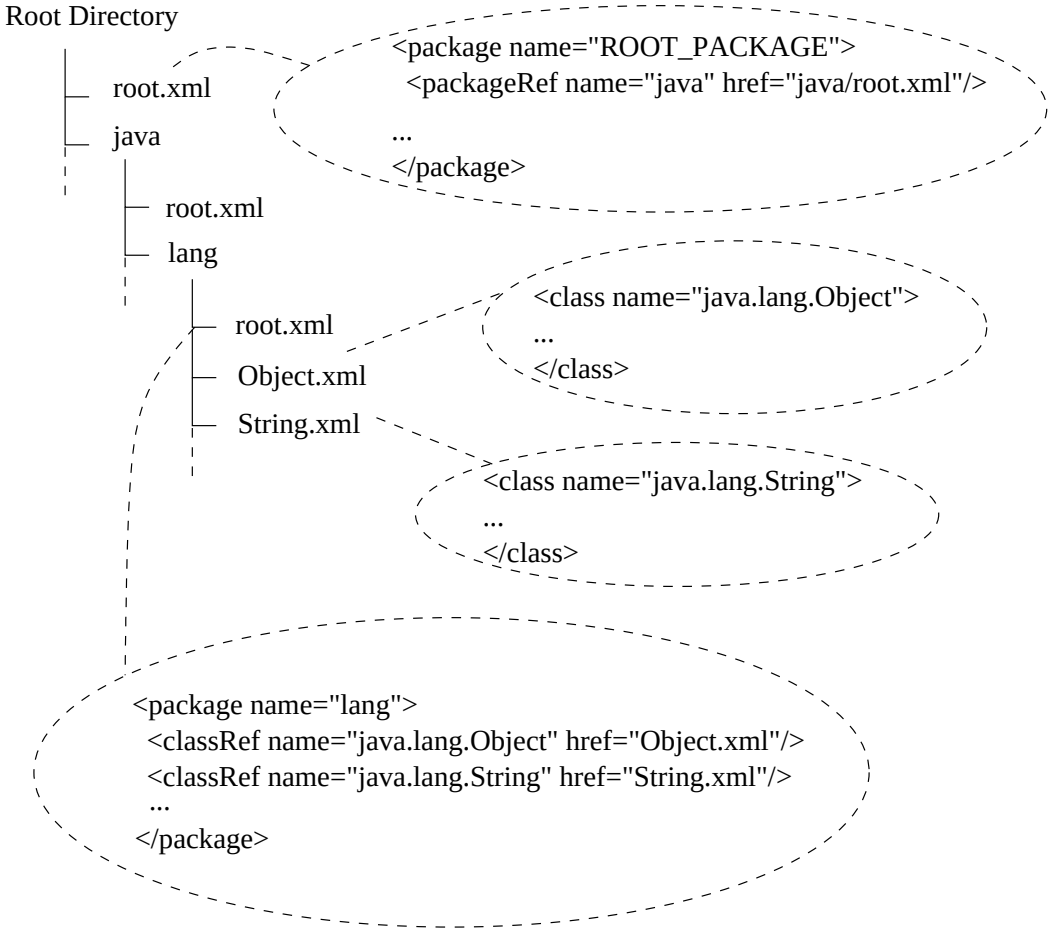
The custom APIs are implemented by a set of wrapper classes that extend the default `Element` implementation of the DOM interface provided by a given DOM implementation. These classes are generated directly from the XML Schema using `SchemaClassBuilder` tool created for this purpose. This allows recreating them easily each time the underlying XML Schema is modified.

3.2.1.2. Disk representation

Our first idea was to implement this database as one large XML document, or to use an actual XML database as the back-end. However, all DOM implementations that we have considered required the whole XML document to be loaded into memory before they would allow access to it. Since this document would contain information about all of the classes used by a program, the size of this document prevented us from allowing that. We also were unable to find any usable XML database, even though a number of them are being developed right now. Therefore, we had to implement a simple database engine ourselves. Since implementing an XML database was not our first priority, our implementation lacks many of the sophisticated features of such databases. The database is stored on disk as a set of XML files. For each Java class there is a corresponding XML document on disk. To simplify the management of these documents they are divided into directories, where a directory corresponds to a package of the class.

The Figure 3-3 shows the structure of the XML files on disk that correspond to the standard Java's package `java.lang`. In that figure the dashed ellipses indicate the content of each XML file. The *Root Directory* is the directory where the database is stored on disk. For each package there is a corresponding directory: `java` and `lang`. In each directory there is an index file: `root.xml`. This index file describes the package itself, and indicates what are its sub-packages and classes. Each class in a package is stored

Figure 3-3. Disk representation of static information database



in its own XML file. In this figure, these are `Object.xml` and `String.xml` files, which correspond to `java.lang.Object` and `java.lang.String` classes, respectively.

The database is populated by the `PackageToXml` utility. `PackageToXml` accepts a package name as its argument and generates an XML representation of this package, recursively descending into sub-packages. The information about each package and each class is obtained through the Java reflection mechanism. Thus, the only information we can currently store in the database is the interface of each class. The information such as comments and the actual source code of each method is not available through the reflection mechanism and thus cannot be obtained by `PackageToXml`. The benefit of using reflection is that the framework is not dependent on any proprietary parser library. However, we are considering changing this so that the information would be obtained from the actual source code files if they are available. This will increase the usability of this database since it will allow storing such things as javadoc comments in addition to raw interfaces.

3.2.1.3. Future Enhancements

Query Language and Indexes. The major problem with our database engine is the lack of a query language. The database engine also lacks the ability to specify indexes for faster access. One solution to this problem is to use a real XML database once it becomes available. Currently we are considering using Lore[15], which is an experimental XML database from the Stanford university. Another alternative is to use a query language such as XPath[10]. XPath is an XML query language in which queries are specified as a path through the XML tree. It is currently used in XSL[12] stylesheets to perform simple queries on XML documents. There are a number of XPath implementation that can work with a standard DOM tree, and therefore it is possible to use them to resolve XPath queries on our database.

Extentions to custom API. Currently our custom APIs only provide access to the attributes of each node. However, they do not provide a simplified access to the children of a node. This is sometimes inconvenient. For example, currently in order to determine if a class is *public*, the user has to traverse all of the immediate children of the node corresponding to that class looking for a *modifier* node with a value *public* in its *name* attribute. We plan to extend our APIs to provide easier access to such information.

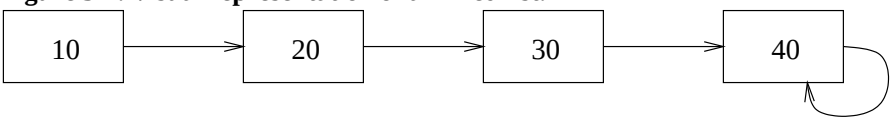
Automatic regeneration of the database. Since the database is populated by an external utility, it must be regenerated completely every time a class file is changed. This is a minor inconvenience for the reference implementation, but it can be a major drawback in making the system user friendly. In the future, we plan for the database to detect changes in the class files and regenerate appropriate parts of the database automatically.

3.2.2. Runtime Information Database

3.2.2.1. User's View of the Database

From the user's point of view, the runtime information database has exactly the same interface as the static information database. Its interface consists of DOM APIs and additional custom APIs. However, the implementation of the runtime information database is completely different from that of the static information database. The data stored in the static information database is derived from the source code of the program, and thus is textual in its form. On the other hand, the data in the runtime information database comes from the executing program, and the largest portion of this data consists of field values. It is usually easier to work with the binary representation of these values, rather than with their string representation. The binary representation is also much more compact. Therefore, in our implementation,

Figure 3-4. Visual representation of a linked list.



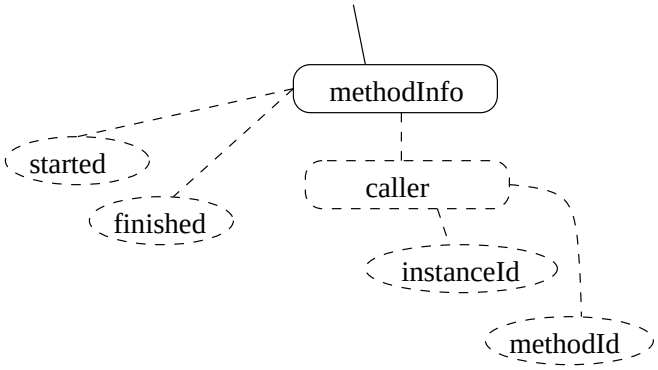
the data in the runtime information database is stored in its binary form. Since the DOM interfaces do not address the issue of the binary representation of the information, the implementation is done as a custom data structure that is extended to support DOM APIs. At a first glance it may seem quite inefficient since from the user's point of view the primary access to the information is the DOM APIs, and if the user does not want to learn our custom APIs then he or she must convert the information obtained from the DOM APIs back to its binary form. However, we think that this will not be a big problem in most cases. If the primary goal of the tool created with the framework is to visualize the data in some form, it is often the case that the string representation of the data is sufficient. Consider the following example. One of the structures in a program is a linked list, and the user wants to create a visual representation of this linked list as in Figure 3-4. In this case all of the values have to be displayed on the screen, and therefore must be converted to their string form. In this case the user will be perfectly satisfied using the DOM APIs to access the database. However, if the goal of the tool is to analyze the program, it may be more convenient to use our custom APIs. For example, if the program is as in the previous example, but the goal is not to visualize a linked list, but rather to have an analysis tool which ensures that at any point in time the sum of all values in this linked list is not greater than 100, then it is much more convenient to be able to obtain the binary representation of these values. In this case the user can use our custom APIs to do the job. We believe that if the goal is to create a very powerful analysis tool and if its performance is important, then the user must learn and use our custom APIs. However, it is always possible to use the more standard DOM APIs and convert the data to its binary form as needed, if the performance is not a major issue.

3.2.2.2. Implementation details

We have considered two alternative approaches to the implementation of the database. Both of these approaches will allow for the data in the database to be stored in the binary format, and will allow to use DOM APIs to access it. One was to implement a custom data structure and then implement the DOM interface for it. Another approach was to take a DOM implementation we liked and augment it to support our requirements. We have chosen the second approach since it did not require us to implement the DOM interface. The DOM implementation we have used was provided to us by Meteko Corporation.

The structure of the runtime information as described in Section 2.1.3 is naturally represented as a DOM tree. The addition we wanted to have was the ability to store some data in its binary form, and to minimize the number of nodes in the internal structure of the tree. The DOM interface requires that each node and each attribute of a node are represented by a Java object. In runtime information structure there are a lot of nodes that can be represented very compactly by one or two fields. For example, consider the *caller* node of *methodInfo* (see Figure 2-15 in Section 2.1.3). At runtime this node can be replaced by a single field in the class representing the *methodInfo* node, that contains a Java pointer to the appropriate *rmethod* node. However, there is no way to access this Java pointer through the DOM interface. In the case of attributes of a node, DOM interface dictates that each attribute must be represented by a Java object. In an example in Figure 2-15 in Section 2.1.3, it would imply that the *started* and *finished* attributes of the node *methodInfo* must be represented by two Java objects, when in fact it is much easier to represent them as two integer fields of the object corresponding to the *methodInfo* node. We wanted to have the best of the two worlds: the simplicity of DOM interface and the efficiency of a custom structure. Our solution was to introduce a notion of *virtual nodes* and *virtual attributes* to our DOM implementation.

Figure 3-5. RMethod with virtual nodes and attributes.



3.2.2.3. Virtual nodes and virtual attributes

Virtual nodes and *virtual attributes* are nodes and attributes of the DOM tree that are not represented as Java objects in the internal structure of the tree. Instead, the information needed to create them is stored directly in the class representing their respective parent node. Whenever a *virtual node* or a *virtual attribute* is requested through the DOM interface, a corresponding Java object is created on the fly. When the object is no longer in use, it is destroyed. By using Java's *weak* reference mechanism it is possible to find out how many references an object has. This allows us to find out when the object is not in use without introducing a custom garbage collection mechanism. This means that it is impossible to distinguish a *virtual node* or a *virtual attribute* from a real one through the use of DOM APIs. So, in example in Figure 2-15, it is possible to replace the *started* and *finished* attributes of the *methodInfo* node with *virtual attributes*, and replace the *caller* node by a *virtual node*, resulting in Figure 3-5. This Figure concentrates only on the parts that have been changed relative to Figure 2-15. Elements drawn with dashed lines are virtual, and do not exist as Java objects until explicitly requested through the DOM APIs. As can be seen from the Figure 3-5, the use of *virtual nodes* and *virtual attributes* allows to replace the six Java objects required to represent this part of the *methodInfo* node with a single object.

3.2.2.4. Other Enhancements

Our implementation also includes a number of other enhancements to improve performance and to minimize the memory size of the structure. As described in Section 2.1.3, the runtime information database must store all of the values of all of the fields in the examined program. It is expected that these values will constitute the major portion of the data in the database. It is thus imperative for the structure for these values to be as compact as possible. We also expect that most of the queries on this structure will benefit if it is indexed by the *time* [1] attribute. Therefore, we have tried to optimize the access time to this part of the structure, as well as to minimize the size of its internal representation. Internally, the values of a field are represented by a `HistoricalVariable` object. This object stores all of the values and their corresponding times in a *SkipList* structure indexed by *time*. This allows very efficient access to the information, while minimizing its size.

3.2.2.5. Indexes

1. See Section 2.1.3 for description of the nodes and attributes used to describe the values of a field.

Unlike the static information database, our implementation of runtime information database contains a number of indexes to improve efficiency of certain queries. There is an index based on the *id* attribute of each *runtime instance*. In Java programs, a runtime instance is either a *class* loaded in memory or an instance of a loaded class. These correspond to *rclass* and *robject* nodes in the tree. In addition to this, the *methodInfo* children of a *threadInfo* node are indexed by the value of their *started* attribute. One of the most important indexes is associated with *rfield* nodes. Since our data structure contains the history of all values for each field, we expect these values to constitute the bulk of data in the database. Hence it is imperative to store and index these values efficiently. As mentioned in Section 2.1.3, each value is stored together with the *time* at which it was attained. Hence the natural index for the values of each field is on this *time* attribute. These indexes allow efficient resolution of the queries we had to implement while experimenting with the framework.

3.2.2.6. Disk Representation

Currently the runtime information database is not stored on disk. Once the SHADOW framework is terminated, the information is destroyed. However, using the DOM interface it is quite simple to dump the data from the database into a large XML file. For the purposes of our project we did not need to store that information, but this is one of our future plans.

3.2.2.7. Future Enhancements

Query Language and Indexes. As in the case with the static information database engine, the runtime information database engine does not support any query languages. Although the runtime information database does support a limited number of indexes, there is no way for a user to define any new ones. All of the possible solutions for the static information database engine will be considered for this database engine as well.

Generation of wrapper classes. Currently the wrapper classes around the DOM implementation are done by hand. We want to investigate if it is possible to automate this process, and to generate these classes directly from the XML Schema as it is done in the static information database case. The wrapper classes for the runtime information structure are much more complicated, but if it is possible to generate the wrappers from the XML Schema, the benefits would be great.

Defining our dependence on Meteko's[13] DOM implementation. Our implementation of the runtime information database is currently very depended on the underlying DOM implementation. We want to identify these dependences and separate our structure from the underlying DOM implementation to allow future extensions using other available DOM implementations.

3.3. Overall Future Enhancements

The efficiency of the runtime information database engine is crucial to the overall performance of the system. In this initial implementation we have considered a number of optimization techniques that can improve the performance greatly. Due to the lack of time we have not implemented them thus far, but will evaluate and implement them in the future.

In general, for our reference implementation we were more concerned with creating a viable system rather than a user-friendly, usable, robust, and efficient product. Therefore, there are obviously a lot of

improvements that can and should be made to all parts of the system. We have already considered a number of ways to improve efficiency and reliability of the framework. However, the details of these improvements are to technical to be presented here.

Figure 4-1. `ShadowDriver` interface.

```
public interface ShadowDriver
{
  // - obtain access to static information
  StaticInfo getStaticInfo ();
  // - obtain access to runtime information
  RuntimeInfo getRuntimeInfo ();

  // - Start Shadow to collect information from a running process
  void startShadow (...);
}
```

Figure 4-2. Query Engine interface

```
public interface QueryEngine
{
  // - find a class given its full qualified name in the static info. db
  public Node findClass (String className, StaticInfo si);
}
```

Chapter 4. Using the SHADOW framework

In this chapter we will describe the use of the SHADOW framework from the user's point of view. The current implementation of the framework is not really meant to be user-friendly, and we have to assume a few minor details to simplify this explanation. First, we assume that there exists a module conforming to the `ShadowDriver` interface which is described in Figure 4-1. The `ShadowDriver` interface is the communication medium between the user's world and the SHADOW framework. It provides three essential methods: `getStaticInfo`, `getRuntimeInfo`, and `startShadow`. The `getStaticInfo` and `getRuntimeInfo` methods provide the access to the static and the runtime information database, respectively. The `startShadow` method is used to start the program to be analyzed and to instruct SHADOW framework to start collecting runtime events. The initialization of the `ShadowDriver` and the arguments of the `startShadow` method are not currently defined. However, it is understood that during this initialization the user will be required to at least provide the program to be analyzed, as well as to set up the required event filters.

We also assume that there exists a module conforming to the `QueryEngine` interface (see Figure 4-2). This interface provides a very simple way to locate a Java class in the static information database. It provides just one method, `findClass`, which finds the description of a class in the database given the fully qualified name of the class and the reference to the static information database.

The rest of this chapter is a set of example programs that use the SHADOW framework to obtain static and runtime information about the Java class outlined in Figure 4-3.

4.1. First example

Our first task is very simple: we want to obtain the information about the class `MyClass` from the static information database and print the names of its fields and methods. To accomplish this task, we will need to obtain the `StaticInfo` module first, and then to find the description of the class `mypackage.MyClass` in it. The description of this class is given to us as an XML subtree, and we have to traverse this subtree

Figure 4-3. Source code for the class `MyClass`.

```
   package mypackage;


   public class MyClass extends Object
 5 {
      int intField;
      MyClass objField;

      public MyClass ()
10    {
        super ();
        intField = 0;
        objField = null;
      }
15
      public void foo1 ()
      {
        ⋮
      }
20    public void foo2 ()
      {
        foo1 ();
        intField = 10 * intField;
        ⋮
25    }

      public void bar ()
      {
        objField = new MyClass ();
30      foo2 ();
        objField.foo2 ();
      }

      static MyClass application;
35
      public static void main (String[] args)
      {
        application = new MyClass ();
        application.foo1 ();
40      application.foo2 ();
        application.bar ();
      }
   }
```

printing the information we want in the process. The program that does just that is given in Figure 4-4. First, we need to initialize the SHADOW framework (line 10) and the query engine (line 12) that we want to use. Then, we obtain the static information module (line 14) using the `ShadowDriver`, and locate the class we want (line 16) using the `QueryEngine`. As the result, we obtain the root of the subtree that we want and store it in our local variable (line 16). Notice that the type of the result is an `SClass`. This is a wrapper class that extends the standard DOM's `Element` interface. This means that we can treat it as a standard DOM class. We use the wrapper class in this example because it is much easier to work with than a standard DOM interface.

The rest of the program is the traversal of the subtree we have obtained in the first step. First we call the `printSClass` method that prints the *name* attribute of the class node we have obtained. The program continues in the similar fashion traversing first the *field* nodes and then the *method* nodes of this subtree. Most of the code is pretty self explanatory, with the exception of the `getElementsByTagName` method. This method is a part of the DOM interface. It returns the list of all children of a node with a given tag name. It is used to obtain a list of all of the fields and all of the methods of the class that we are investigating.

As it can be seen from the previous example, it is quite easy to create a Java program that traverses the database. However, if the main goal of the program is to produce some printed, or graphical, output, this traversal can become quite annoying. Since our database is represented as an XML document, it is possible to use different XML processing techniques to simplify this task. One of these techniques is to use XSLT stylesheets. An XSLT stylesheet is a program that is *applied* by the XSLT processor to a given XML document. The XSLT stylesheet consists of a set of templates; each template contains a pattern which identifies to which part of an XML document it applies. Generally speaking, the XSLT processor walks the XML document in a depth-first manner applying the templates specified by the stylesheet. For a more detailed description of the XSLT stylesheets, see [19].

In order to apply the stylesheets to the database one requires an XSLT engine that works with the given implementation of the DOM interface. Most XSLT engines will require one to write a custom liaison to allow for this. For our case studies we have used Xalan from the Apache XML project, which was quite easy to customize in order for it to work with our DOM.

When using XSLT it is also possible to produce an HTML output, rather than just text output. This allows one to easily create a graphical front end without spending much time on writing the GUI. All that is required is to specify how does the XML representation of the interesting part of the database is to be converted into an HTML representation. It is then possible to combine the XSLT processor with an HTML browser or with a web server to create dynamic applications. In this case one would need to create a simple driver program that will obtain the `StaticInfo` module using the `ShadowDriver` interface, initialize the XSLT processor in use, and then instruct the processor to apply a given stylesheet to the static information database.

The XSLT stylesheet that produces the same result as the program in Figure 4-4 is presented in Figure 4-5. One of the major advantages of the stylesheet is that it is no longer necessary to write the tree traversal code. At the same time, the XSLT stylesheets provide a simple, yet very efficient, query language, XPath[10]. This significantly simplifies the extraction of the information from the database. Depending on the XSLT processor the stylesheets can be made as powerful as a Java program, since many XSLT processors allow to call external Java routines from the stylesheet.

4.2. Second Example

Figure 4-4. Obtaining static information using a Java program.

```
// - import all required packages first
import ...
public class ShadowEx1
{
  public static void main (String[] args)
  {
    // - obtain the Shadow driver from somewhere
    ShadowDriver driver = ...;
    // - obtain query engine
    QueryEngine query = ...;
    StaticInfo si = driver.getStaticInfo ();
    SClass clazz = (SClass)query.findClass ("mypackage.MyClass", si);
    printSClass (clazz);
  }
  static void printSClass (SClass clazz)
  {
    // - print the class name
    System.out.println ("class " + clazz.getName ());
    // - print Fields
    System.out.println ("Fields: ");
    printSClassFields (clazz);
    // - print Methods
    System.out.println ("Methods: ");
    printSClassMethods (clazz);
  }
  static void printSClassFields (SClass clazz)
  {
    // - get all children of type 'field'
    NodeList fields = clazz.getElementsByTagName ("field");
    // - get the length first
    int len = fields.getLength ();
    // - now loop through the fields
    for (int i = 0; i < len i++)
      printSClassField ((SField)fields.item (i));
  }
  static void printSClassMethods (SClass clazz)
  {
    NodeList methods = clazz.getElementsByTagName ("method");
    int len = methods.getLength ();
    for (int i = 0; i < len; i++)
      printSClassMethod ((SMethod)methods.item (i));
  }
  static void printSClassField (SField field)
  {
    // - just print the name, more complicated processing is
    // - is possible, just walk the subtree
    System.out.println ("Field: " + field.getName ());
  }
  static void printSClassMethod (SMethod method)
  {
    System.out.println ("Method: " + method.getName ());
  }
}
```

Figure 4-5. XSLTStylesheet to produce the same result as the program in Figure 4-4.

```
   <xsl:stylesheet>
    <xsl:template match="/package[@name='mypackage']/class[@name='MyClass'">
      <h1>class <xsl:value-of select="@name"/></h1>
      <h2>Fields</h2>
 5    <ul>
        <xsl:apply-templates match="field"/>
      </ul>
      <h2>Methods</h2>
      <ul>
10      <xsl:applyt-templates match="method"/>
      </ul>
    </xsl:template>

    <!- XSLT template for SField ->
15  <xsl:template match="field">
      <li>
       <xsl:apply-templates match="typeRef"/>
       <xsl:text> </xsl:text>
       <xsl:value-of select="@name"/>
20    </li>
    </xsl:tempalte>

    <xsl:tempalte match="method>
      <li>
25     <xsl:apply-templates match="returnType"/>
       <xsl:value-of select="@name"/>
      </li>
    </xsl:template>

30  <xsl:template match="returnType|typeRef">
      <xsl:if select="@primitive = 'false'">
        <xsl:element name="a">
          <xsl:attribute name="href">
     <xsl:call-template name="convert-href-to-html"/>
35        </xsl:attribute>
          <xsl:value-of select="@name"/>
        </xsl:element>
      </xsl:if>
      <xsl:if select="@primitive = 'true'">
40      <xsl:value-of select="@name"/>
      </xsl:if>
    </xsl:template>

    <xsl:template name="convert-href-to-html">
45    <!- custom template to convert 'href' attribute in the XML database
    into a href attribute in an HTML file goes here ->
    </xsl:template>
   </xsl:stylesheet>

50
```

We now turn to our second example in which we illustrate how to use the runtime information database to allow on-the-fly processing of the analyzed program. In this example, our task is to produce a print-out of events that occur in our trivial program given in Figure 4-3. This is accomplished by the program in Figure 4-6. In order to receive events from the runtime information database, one must first register an event listener with it. Our event listener is called `ClassEventPrinter`, and its source code is given in Figure 4-7. The program starts by creating an instance of the `ShadowDriver` module and registering the event listener with it. It then calls `startShadow` method to start the program to be analyzed. The `startShadow` method instructs SHADOW to start gathering events and waits until the program terminates. Therefore, when this method exits we can start the postmortem analysis of the information. Most of the work in this example is done in the event listener. Its job is to react to different events that occur and produce the printout we want. In order to receive the events, we must register the event listener on the part of the tree we are interested in; however, this part of the tree is not available before the class corresponding to `MyClass` program is loaded by the Java VM. The solution is to register our event listener with the root of the tree. Then, once it receives the *ClassLoaded* event corresponding to the class `MyClass`, it may move itself to the part of the tree corresponding to that class. This is possible because all events by default bubble up to the root, as described in Section 2.1.3.3.

Each event processed by the listener has a *source* attribute. This attribute contains the reference to the part of the tree that was changed. Therefore, the event listener can always find out what has actually happened and where. Most of the code for the event listener is self explanatory. Notice that we once again use our wrapper classes such as `RClass` and `RObject` instead of the straight DOM interfaces. These wrapper classes allow for a more efficient and more natural access to the information from a Java program. The DOM interface is still available and the program can be rewritten to use it instead. The DOM interface was not intended to be used directly by the programmer, but rather by more sophisticated processing tools, such as XSLT engines.

As the above examples show, SHADOW is quite easy to use. It gives the programmer the alternative of using a simpler and less efficient approach - XSLT stylesheets, while still allowing to use much more efficient Java APIs.

Figure 4-6. Using runtime information database.

```
   // - import all required packages
   import ...

   public class ShadowEx2
 5 {
     public static void main (String[] args)
     {
       // - obtain ShadowDriver
       ShadowDriver driver = ...;
10
       RuntimeInformation ri = driver.getRuntimeInformation ();

       // - create an event listener
       REventListener listener = new ClassEventPrinter ("mypackage.MyClass");
15
       // - add the event listener to the root of the tree
       ri.getRoot ().addREventListener (listener);

       // - start gathering events
20     driver.startShadow (...)

       // - once startShadow exits we can start post-mortem analysis here.

     }
25
   }
```

Chapter 5. Case studies

5.1. Introduction to the Demos

We have created three major case studies for our system, all reflecting various stages of the system's development. The information available statically at compilation time was the earliest information that our system could produce. Therefore, in order to utilize it we first created a simple class viewer that could display information about Java classes and packages in a web browser. This tool actually turned out to be quite useful to us in later stages of our project.

Later in our development, runtime information became available. At this time we created a tool that was very similar to the tool above, but which could display all of our runtime information in addition to the static data. These two tools do not do any complicated processing of our data; they simply output it in a formatted way. However, they do display all of the data that is available in our data structures.

The third tool we created actually performed some rudimentary processing and output runtime information in a structured way. This tool takes a Java class/object in an executing Java program and creates a pictorial representation of that object in JVM's memory. This representation is especially useful since it shows a clear relationship between each class and its superclasses. Some instructors at the University of Toronto have found such a representation to be a useful pedagogical tool.

5.2. How Applications Were Created

We have decided to create a web-based interface for all of our tools. The reason is that creating an interface may potentially be quite time-consuming. However, HTML provides a very easy and rapid way to create some very rich interfaces. In addition, HTML allows one to describe structured information in a text

Figure 4-7. `ClassEventPrinter`.

```
public class ClassEventPrinter implements REventListener
  {
    String className;
    public ClassEventPrinter (String _className)
      { className = _className; }
    public void processEvent (REvent event) {
      if (event.getType () == REvent.CLASS_LOAD_TYPE) {
        // - source of the class load event is the node that describes
        // - the loaded class
        RClass clazz = (RClass)event.getSource ();
        // - check if we are interested in this class
        if (className.equals (clazz.getSClass ().getName ()))
        {
           System.out.println ("Loaded: " + className);
           // - re-register on that class so that events
           // - will not have to bubble up to reach us
           clazz.addEventListener (this);
        }
        else if (event.getType () == REvent.MAKE_INSTANCE_TYPE) {
          // - source is the object that was created
          RObject robject = (RObject)event.getSource ();
          System.out.println ("Object of class: " +
               robject.getParentSClass ().getName () + " (with id: " +
               robject.getId () + ") was created");
          // - we can also use straight DOM calls to get to some information
          System.out.println ("Parent of an object with id " +
                          robject.getAttribute ("ID") + " is " +
                          robject.getParentNode ().getAttribute ("ID"));
        }
        else if (event.getType () == REvent.FIELD_CHANGE_TYPE) {
          RField field = (RField) event.getSource ();
          // - parent of RField is RObject, lets use DOM to get it
          RObject parent = (RObject) field.getParentNode ();
          System.out.println ("Field change on obj "+parent.getId());
          HistoricalVariable var = field.getVar ();
          // - obtain current (new) value
          HistoricalValue curValue = var.getCurrentValue ();
          // - obtain previous (old) value
          HistoricalValue prevValue =
              var.getValue (curValue ().getTime () - 1);
          String prefix =
             field.getSField ().isPrimitive () ? "" : "objId";
          System.out.println (parent.getParentSClass ().getName () + "." +
              field.getSField ().getName () + "changed from " + prefix +
              prevValue.toString () + " to " + prefix +
              curValue.toString ());
        }
        // - no one will need this event, so cancel bubbling
        event.setBubble (false);
      }
    }
  }
```

Example 5-1. A sample JSP program.

```
    <html>
    <body>
    <center>
    <%= String.valueOf(System.getCurrentTimeMillis()) %>
  5 </center>
    </body>
    </html>
```

extremely simple and consist of only a few lines of code each. Each servlet extracts parameters from the HTTP request (which is done almost automatically by the servlet framework). It then runs an XSLT processor (we use Xalan[20] from Apache) on an appropriate input and outputs whatever HTML is generated by the XSLT.

In terms of the actual experience, XSLT turned out to be very easy to learn and also very easy to use for our purposes. The actual coding part for all of our three demos took about a day or two total. The bulk of our work was in trying to create a nice HTML layout, as well as to pick out visually pleasing fonts and colors. Hence we have come to the conclusion that our framework is a useful and usable tool, and using XSLT and web interface is a very convenient and low-cost way to utilize the power of our framework.

5.3. How Applications Work

5.3.1. Static Class Browser

Our first case study was to create a simple class viewer that presents user with information that is contained in our `StaticInfo` database. As described previously in this report, this information must be generated and updated manually and is stored in a special file structure on disk. There are two basic views in this application. One view shows a contents of a single Java package (i.e. it shows a list of all packages and classes that are contained in it). For example, the Figure 5-1 presents a view of a part of the standard package called `java`. Notice the ToolTip below the `net` package (mouse pointer not shown). Whenever the user points to the name of a package or the class, a ToolTip pops up, displaying a fully qualified name of that package or class.

The second view in the static class browser displays actual class information. It shows a list of all of class's fields and methods, as well as a list of superclasses and interfaces. For example, Figure 5-2 illustrates what is seen for class `java.lang.Object`. Notice the ToolTip that corresponds to the `registerNatives()` method. Whenever the user points to the name of a method or a field, all of the Java modifiers for that field or method will be displayed. Also, all of the type names that correspond to Java classes and not to Java primitive types are actually hyperlinks that point to a similar page for the appropriate type.

5.3.2. Runtime Object Browser

For our second case study we wanted to present information contained in our `RuntimeInfo` database in a way similar to our original class browser. The first screen that the user sees contains a list of all loaded classes and all instances of those classes. For example, the screen in Figure 5-3 is what is seen when the system is run on one very simple program from Example 5-2. (This program presents a windows with a

Figure 5-1. `java` package in the Static Info display.

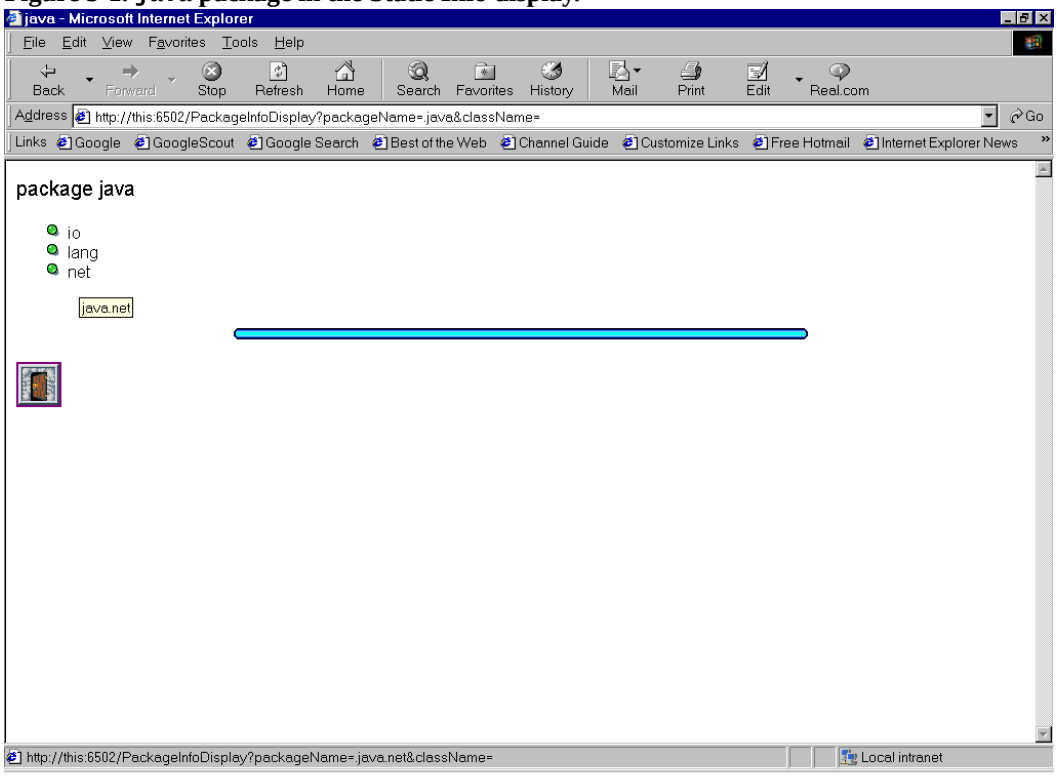
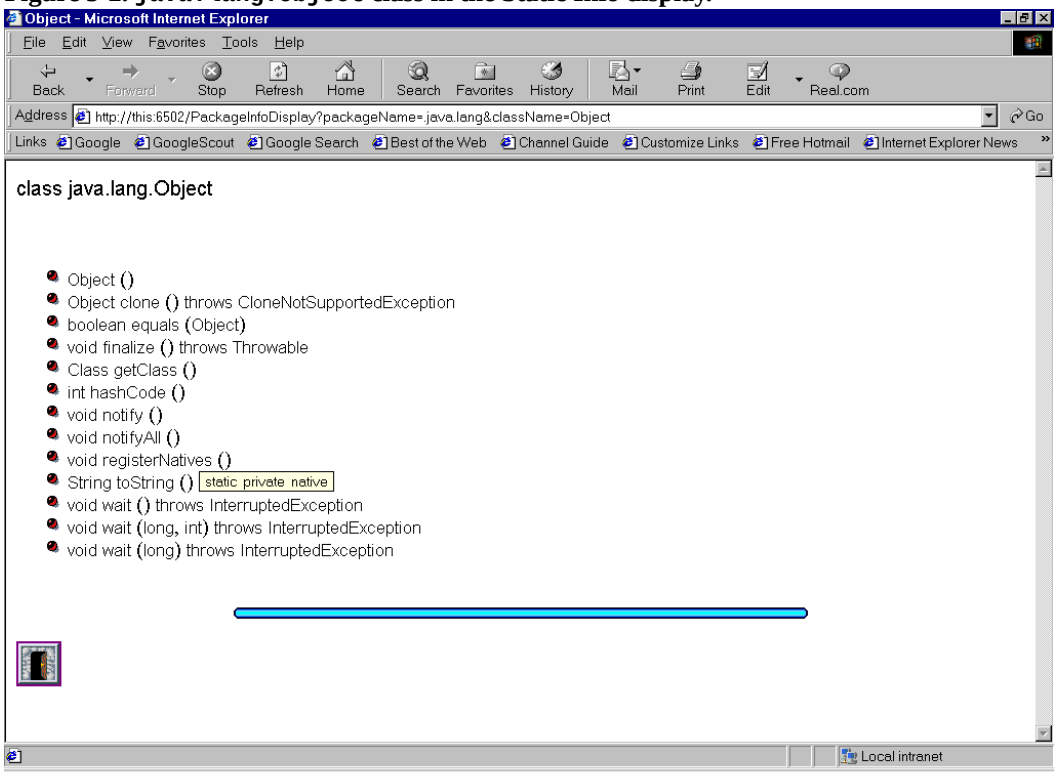


Figure 5-2. `java.lang.Object` class in the Static Info display.



button, and its events are controlled by that button. Hence it is possible to refresh a web-browser after each click in order to see the changes that occurs inside of the program). The user then can click on any class or any instance of each class to see a more detailed information about it. For example, the screen in Figure 5-4 shows what is seen when the user chooses to see object of type `experiments.debugee.GUIDebugee` whose object `ID` is `53`. This view shows all of the fields and methods belonging to this instance and class. Each field is shown with current value and system time at which the field was last modified. Notice the word `HISTORY` beside each field. When the user points to this word, a ToolTip is displayed containing all of the field's previous values and times when those values were attained.

On a color monitor it can be seen that the colors of the values of fields *f*, *l*, and *sm* are different from that of *x*. This is because these fields are not of primitive types, but are all some types of objects. The numbers that are seen are actual `ID`'s of these objects. Whenever a user clicks on one of these numbers, they will be taken to a similar display for that particular object. Note also that the user may click on the name of any field or method in order to go to the static class browser (previous case study) that describes that field or method and the whole class in a greater detail.

Figure 5-5 shows continuation of Figure 5-4. On Figure 5-5 we can see information about the method calls, particularly about the method `incr()` of this object. For each call we can see the time when it has started, as well as the time when it has finished. Also, whenever available, we can see information on who called this method and what methods were called. For example, at time 152 the method `main()` of object with `ID` `51` has called our `incr()`. Then, at time 166 `incr()` called method `setY()` of object with `ID` `417`. So it can be seen that the call stack at each particular time can be easily reconstructed. All of these object `ID`'s seen on this figure are actually hyperlinks that lead to the runtime information about their respective objects.

5.3.3. Java Memory Model

For our third case study we decided to do something more useful with information available to us. For each object and class we were able to create a diagram that displays the way that object is laid out in JVM's memory. Note that at the Figure 5-3 small paw icons appear beside each class and each object. These are actually links to our third application. When the user clicks on one of them, they'll see a screen similar to Figure 5-6. This screen also contains information on the instance of `GUIDebugee` with `ID` `53`, just as Figure 5-4, however it is presented in a different way. The nesting of the boxes indicates class inheritance. We can see immediately that `GUIDebugee` extends `java.lang.Object` and that this is essentially how the objects are laid out in memory. The number beside each field indicates its current values. As with the object browser, instance `ID`'s are actually hyperlinks that lead to displays for the their respective instances.

The most important features of this demo is the ability to display method inheritance. In the figure above the user points at the method `toString()` of class `Object`. (This can be seen by the ToolTip that appears beside that method). Note, however, that when the user does that, the method `toString()` of class `GUIDebugee` becomes highlighted in yellow. This means that this method has been overridden. Therefore whenever `toString()` is called on an instance `53`, the highlighted method is what will actually be executed.

Example 5-2. Sample Debugee Program

```
   import java.awt.*;
   import java.awt.event.*;

 5 class BtnListener implements ActionListener
   {
     GUIDebugee d=null;

     public BtnListener (GUIDebugee d)
10   {
       this.d=d;
     }

     public void actionPerformed(ActionEvent e)
15   {
       d.incr();
     }
   }

20 class FrameListener extends WindowAdapter
   {
     GUIDebugee d=null;

     public FrameListener (GUIDebugee d)
25   {
       this.d=d;
     }
     public void windowClosing(WindowEvent e)
     {
30     e.getWindow().dispose();
       System.exit(0);
     }
   }

35 class SomeClass
   {
     int y=0;

     public SomeClass ()
40   {
       y=0;
     }

     public void setY (int val)
45   {
       y=val;
     }

     public String toString ()
50   {
       return "y is " + y;
     }


55 }
```



```
   public class GUIDebugee
   {

60   public int x = 0;
```

Figure 5-3. Initial Screen in the Runtime Info display.

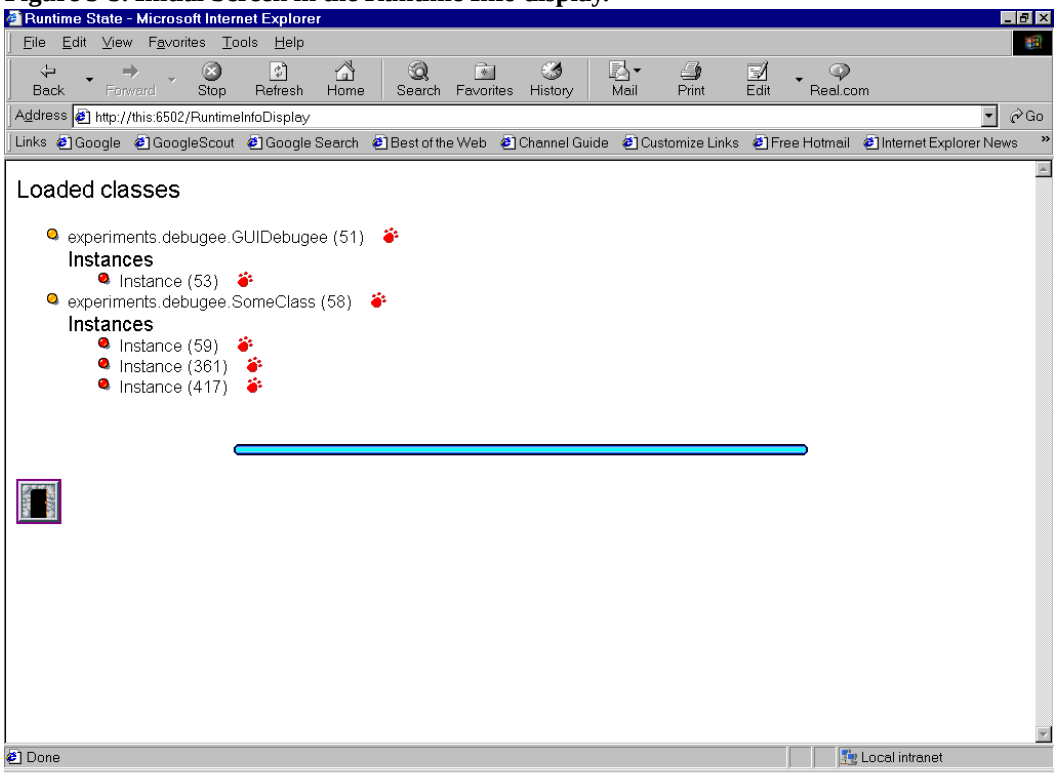


Figure 5-4. Instance 53 of `GUIDebugee`.

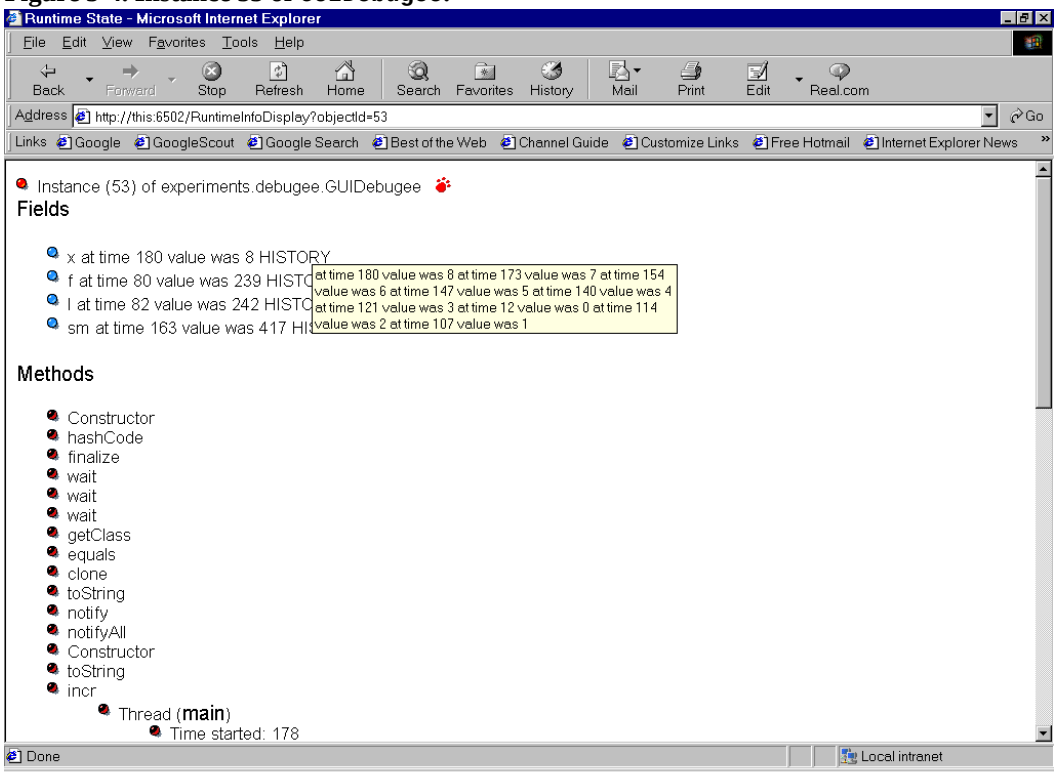


Figure 5-5. Instance 53 of `GUIDebugee` - continuation.

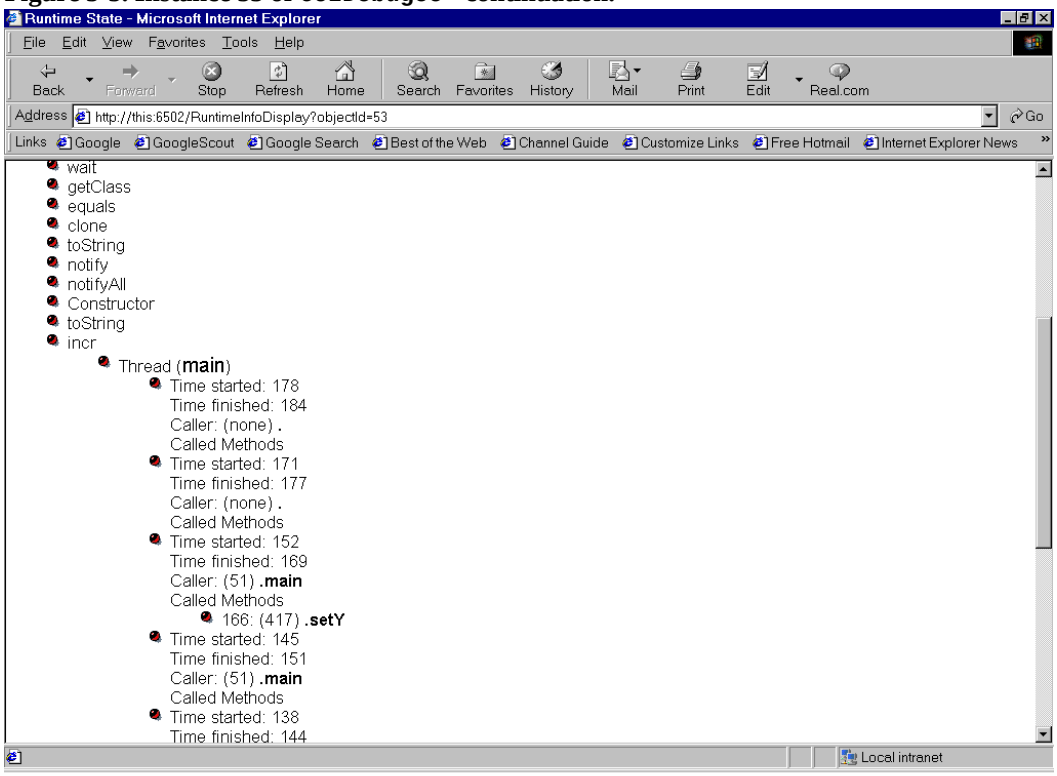
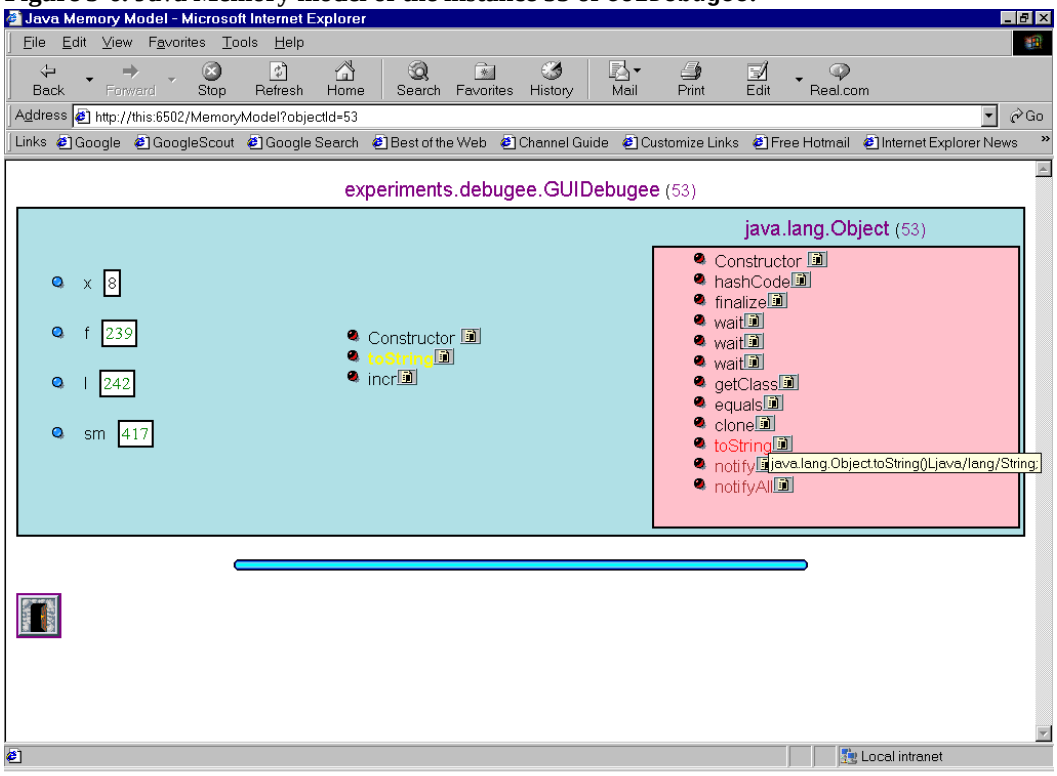


Figure 5-6. Java Memory model of the instance 53 of `GUIDebugee`.



Chapter 6. Summary

To summarize, we developed a framework that allows one to quickly and easily develop high-level program analyzers. This is achieved by decoupling information extraction and representation from the analyzing tool. We have developed an XML-based language specifically designed to represent information about compile-time and runtime features of a Java program.

We feel that our prototype implementation has successfully demonstrated the feasibility of our approach, and our future goals are to expand the framework into a more user-friendly and robust system, in particular, by solving the following shortcomings:

1. **No well-defined user interface.** Currently, no unified driver exists for the whole framework. It consists of a number of components that may work together if properly initialized and connected. There is no well-defined way or order of starting SHADOW, connecting it to the program that is to be analyzed, etc. There is also no nice way for the user to specify which events are to be filtered. Also, virtually no error-handling is performed by current modules. Therefore, it is next to impossible for someone to start up our system without being extremely familiar with all of its source code. Most of these issues are not difficult to solve and were simply postponed during our initial development.
2. **No custom APIs.** As described in this report, we plan to provide access to our data structures via DOM APIs and via our own custom APIs. We plan to generate wrapper classes that support our APIs directly from our XML Schema; currently these classes are partially generated, but partially written manually. Moreover, there is a project underway at SUN that attempts to do a similar thing. However, such an undertaking is a large project in itself, so the classes that we currently generate do not provide sufficient access to all of our data. Hence, for now, the only way to have access to all of the information is to use DOM APIs. Thus one of the future extensions is to provide a *standard* alternative set of APIs tailored specifically to our data.
3. **Concurrency.** Although we have thought about representing information about multi-threaded programs, we have not implemented any support for it yet. Moreover, we haven't yet finalized the structure of that information. Some things that we are concerned with include different threads, their status, object locks or semaphores and their wait queues. Representing this information is vital for analysis of most reasonably-sized programs. For instance, any Java programs that has a GUI is inherently multi-threaded.
4. **Efficiency and Robustness.** Our major concern now that we are ready to implement a usable and friendly system is with efficiency and robustness. We have ignored many issues such as scalability and error-recovery during our development, although not during our design. Now we are becoming more concerned with these issues. Many of them will only manifest themselves once we try to use SHADOW with reasonably large programs.

We believe that all of the issues listed here are solvable in a straightforward manner, and the bulk of the work has already been done. Also, despite this list of limitations, our framework can already be used to aid software developers, and with a little more work we hope it will become an indispensable tool for many programmers and researchers.

Bibliography

[1] *SHADOW design specification (http://shadow.aguga.net)* .

[2] *GDB: The GNU debugger (http://sourceware.cygnus.com/gdb/)* .

[3] *XTango algorithm animation system (http://www.cc.gatech.edu/gvu/softviz/algoanim/xtango.html)*.

[4] *Java Platform Debugger Architecture (http://java.sun.com/products/jpda/)*.

[5] *Object Management Group Home Page (http://www.omg.org)*.

[6] *Object Management Group Home Page (http://www.omg.org)*.

[7] *Extensible Markup Language XML (http://www.w3.org/XML/)*.

[8] *Document Object Model (DOM) (http://www.w3.org/DOM/)*.

[9] *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax (http://www.rfc-editor.org)*.

[10] *XML Path Language XPath (http://www.w3c.org/TR/xpath)*.

[11] *JPython Home (http://www.jpython.org)*.

[12] *Extensible Stylesheet Language (XSL) (http://www.w3.org/Style/XSL/)*.

[13] *Meteko (http://www.meteko.com)*.

[14] *XML Path Language (XPath) (http://www.w3.org/XML/)*.

[15] *Lore (http://www-db.stanford.edu/lore/)*.

[16] *Java Servlet API (http://java.sun.com/products/servlet/index.html)*.

[17] *JavaServer Pages technology (http://java.sun.com/products/jsp/index.html)*.

[18] *The Jakarta Project Subprojects - Tomcat (http://jakarta.apache.org/tomcat/index.html)*.

[19] *XSL Transformations (XSLT) (http://www.w3.org/TR/xslt)*.

[20] *Xalan-J Version 1.0.1 (http://xml.apache.org/xalan/index.html)*.

Glossary of Terms and Abbreviations

Disclaimer: Most of the definitions in this section were taken from the FOLDOC dictionary (http://foldoc.doc.ic.ac.uk), the Wĕbopēdia dictionary (http://webopedia.internet.com), the World Wide Web Consortium site (http://www.w3c.org) and the JavaSoft site (http://www.javasoft.com).

API

See: Application Program Interface

Application Program Interface

A set of routines, protocols, and tools for building software applications.

Batch

See: Event Batch

Bubbling

See: Event Bubbling

CGI

Common Gateway Interface, a standard for running external programs from a World-Wide Web HTTP server.

CORBA

Common Object Request Broker Architecture, an architecture that enables pieces of programs, called objects, to communicate with one another regardless of what programming language they were written in or what operating system they're running on.

Debugger

A program analyzer whose purpose it to aid in locating errors in a program.

See Also: Program Analyzer.

DOM

Document Object Model, is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

See Also: XML.

Event Batch

A partially ordered set of events that are logically simultaneous.

Event Bubbling

Propagation of events through the runtime information tree from the node where the event occurred to its parent.

Framework

A set of classes that embodies an abstract design for solutions to a number of related problems.

High-Level Program Analyzer

A program analyzer that is concerned with the *logical* structure (programmer's view) of the program.

See Also: Program Analyzer.

HTML

Hypertext Markup Language, the authoring language used to create documents on the World Wide Web.

HTTP

The client-server TCP/IP protocol used on the World-Wide Web for the exchange of HTML documents.

IDL

Interface Definition Language, a language used to specify interfaces for CORBA objects in a programming-language independent manner.

See Also: CORBA.

Java Servlets

Servlets are modules that extend request/response-oriented servers, such as Java-enabled web servers.

See Also: CGI.

JDI

Java Debug Interface, defines a high-level Java language interface to JPDA, which tool developers can easily use to write remote debugger applications.

See Also: JPDA.

JPDA

Java Platform Debugger Architecture is a multi-tiered debugging architecture that allows tools developers to easily create debugger applications which run portably across platforms, virtual machine (VM) implementations and SDK versions. It consists of 3 layers: Java Virtual Machine Debug Interface, Java Debug Wire Protocol, and the Java Debug Interface.

See Also: JDI.

JSP

Java Server Pages, technology for building applications containing dynamic web contents, achieved by mixing HTML and Java code.

JVM

Java Virtual Machine, a specification for software which interprets Java programs that have been compiled into byte-codes.

Machine-Level Program Analyzer

A program analyzer that is concerned with the machine view of the program.

See Also: Program Analyzer.

Modifier Node

A node in a SHADOW tree that contains any extra information about its parent, for use with language-dependent program analyzers.

Namespace Member Node

A node in a SHADOW tree that corresponds to a member element (such as a field or a method) of some namespace.

Namespace Node

A node in a SHADOW tree that serves as a divider between parts of the tree corresponding to various program namespaces.

Program Analyzer

A software tool that uses other programs or information about them as its input.

See Also: Debugger, Program Visualiser.

Program Visualiser

A program analyzer whose purpose it to display some information about the program in a user-friendly manner.

See Also: Program Analyzer.

Reference Node

A node in a SHADOW tree that points to another node, via its *href* and *name* attributes.

RMI

Remote Method Invocation, a set of protocols that enables Java objects to communicate remotely with other Java objects.

Runtime Information

Information about the states of an executing program.

See Also: Static Information.

Source-Level Program Analyzer

A program analyzer that is concerned with the source code view of the program.

See Also: Program Analyzer.

Static Information

Information about a program that is available during, or immediately after, the compilation.

See Also: Runtime Information.

Time

In the context of SHADOW framework, one atomic change to the runtime information tree.

Tomcat

A simple web server that can execute Java Servlets and JSP pages.

See Also: JSP, Java Servlets.

URI

Uniform Resource Identifier, the generic set of all names and addresses which are short strings which refer to objects (typically on the Internet).

See Also: URL, URN.

URL

Uniform Resource Locator, a standard way of specifying the location of an object, typically on the Internet. A kind of URI.

See Also: URI, URN.

URN

Uniform Resource Name, a location-transparent kind of URI.

See Also: URL, URI.

Virtual Node/Attribute

A node in an XML document for which a corresponding DOM node is only generated on demand.

See Also: XML, DOM.

Xalan

Apache's XSLT processor.

See Also: XSLT.

XML

Extensible Markup Language, a universal format for structured documents and data, developed by the World Wide Web Consortium.

XML Schema

An XML-based language that provides a way to describe and validate data in an XML environment by making relationships and types more explicit.

See Also: XML.

XPath

A language for addressing parts of an XML document, designed to be used, in particular, by XSLT.

See Also: XML, XSLT.

XSL

eXtensible Stylesheet Language, a language for specifying the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary.

See Also: XML, XPath, XSLT.

XSLT

Extensible Style Language Transformation, the language used in XSL style sheets to transform XML documents into other XML documents.

See Also: XML, XPath, XSL.