VICTORIA UNIVERSITY OF WELLINGTON

Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Understanding Frameworks through Visualisation

Kirk Jackson, Robert Biddle, Ewan Tempero
School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
{Kirk.Jackson | Robert.Biddle | Ewan.Tempero}@mcs.vuw.ac.nz

Technical Report CS-TR-00/02
November 2000

Abstract

Object-oriented frameworks can significantly improve productivity, but using to learn how to use them effectively is a known hard problem. We discuss the issues related to this problem, and describe the design and development of a prototype framework visualisation system intended to help solve the problem.

Publishing Information

A version of this paper appeared in the proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society, November 2000.

1 Introduction

> *"However, part of the folklore of frameworks is that nobody understands a framework until they have used it, so using it is more important than reading about the theory behind it" ..... "Thus, we need to explain how to use a framework without explaining how it works."* Ralph Johnson[2]

Object-oriented frameworks are bodies of reusable code that an advanced programmer can use in many situations to quickly produce a new application with less effort than starting from scratch. However, using a framework requires an intimate understanding of how it will interact with the extensions that the programmer will provide — knowledge that has to be learnt. To acquire knowledge of a framework's behaviour can take a great deal of effort, and increases the costs of development the first time a framework is used. This is seen as one of the major problems with frameworks [2]. In this paper, we present a tool, *Fire*, that uses *dynamic software visualisation* to help programmers understand how to use frameworks effectively.

Our paper is organised as follows. We begin by discussing the salient aspects of frameworks, in particular, the *framework usage scenarios*. These are the various ways that programmers use frameworks to build their applications. Section 3 discusses the kinds of visualisations that might be used to describe frameworks. We introduce a dynamic version of the UML "collaboration diagram", which we call an *instance* diagram. In section 4, we discuss the design and implementation of *Fire*. *Fire* provides visualisations of C++ programs compiled to use gdb, the Gnu Debugger. *Fire* can also display information relating to any frameworks used by the program, providing the relevant frameworks are described to it. Finally, we present our conclusions.

2 Object-Oriented Frameworks

A framework is a partial program that provides a skeleton for a family of object-oriented applications. Each member of the family can be constructed by just customising the framework as required. By reusing a common core of code, the development effort of constructing the whole family of applications is collectively lower, due to decreases in the quantity of code created, and benefits of reused expertise.

What makes a framework hard to use is the complex interaction between its parts, and the customisation done by the application programmer. Part of the complexity is due to the fact that object-oriented frameworks are typically are implemented with "inversion of control". The application developer provides classes to the framework, which are called at some point *by the framework*. Thus, it is the framework, rather than the application developer, that is controlling the collaboration of the program's classes.

The other part of the complex behaviour associated with frameworks is due to how the customisation is done. Wolfgang Pree coined the term "Hot Spot" in 1994 to describe the places in the framework that are changed when creating actual applications from the framework [6]. Hot spots allow code that may be varied to be grouped together, and "plugged-in" to a part of the framework, and so identify areas of the framework where customisation is intended [7]. There are a number of different ways that an application developer actually carries out the customisations. We have identified these *usage scenarios* below:

1. *Instantiating and combining supplied classes to form a basic application.* In this case, the framework developer has foreseen this use of the framework, and the only difficulty the application developer will have is in the *selection* of appropriate classes.

2. *Subclassing a concrete class, adding methods without overriding the supplied methods.*

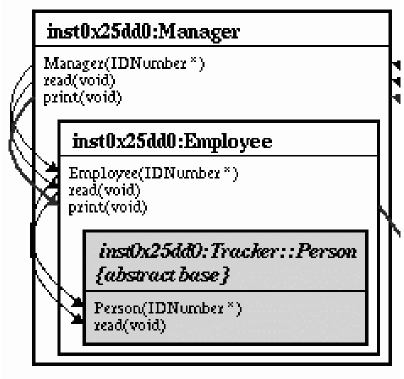

Figure 1: Instance Diagram object of an instance of the Manager class.

This occurs when the application developer is able to use the framework, but wants to add some extra functionality to the provided classes so they can be used elsewhere in the application. The internals of the class may need to be understood so that functionality provided by the framework is not affected by the alterations.

3. *Subclassing a concrete class, and overriding only one method in the subclass.* This situation is common, where a class in the framework performs almost all of the required behaviour, and the application developer needs only to change one method. The developer has to understand the responsibilities of the overridden method, and situations where it is used so that the new method can conform to those responsibilities.

4. *Subclassing a concrete class, and overriding several methods in the subclass.* This is an extension of case 3, but with more relationships to understand.

5. *Subclassing an abstract class, and creating all the methods required.* Where the framework has hot spots that will take instances derived from a base class, it is common for an application developer to create a completely new class to fill that hot spot. In this situation, the developer will need to look at the derived classes that have been provided by the framework, then try to determine the circumstances under which the methods are called by the framework, and the responsibilities of the classes.

6. *Subclassing a class that conforms to the hot spot's requirements, but also has additional functionality.* The difficulties here lie in preserving all current relationships (if the intention is for the new class to replace the base class), or to change the behaviour without the existing relationships of the base class interfering. Both difficulties are again overcome through intimate knowledge of the behaviour of the class that is being inherited from.

7. *Subclassing a framework class, and presenting an aggregate of other classes within the framework.* This is a less frequent usage pattern, but requires knowledge of the behaviour of the classes aggregated within the new class.

3 Dynamic Diagrams

A question that must be addressed with any effort involving visualisation is what kinds of pictures and pictures should be drawn. Our philosophy is to try to get as much as possible from "standard" diagrams before inventing new kinds. In this work, we have concentrated on producing diagrams that are either standard UML [8], or very close to it. The diagrams we have concentrated on are the

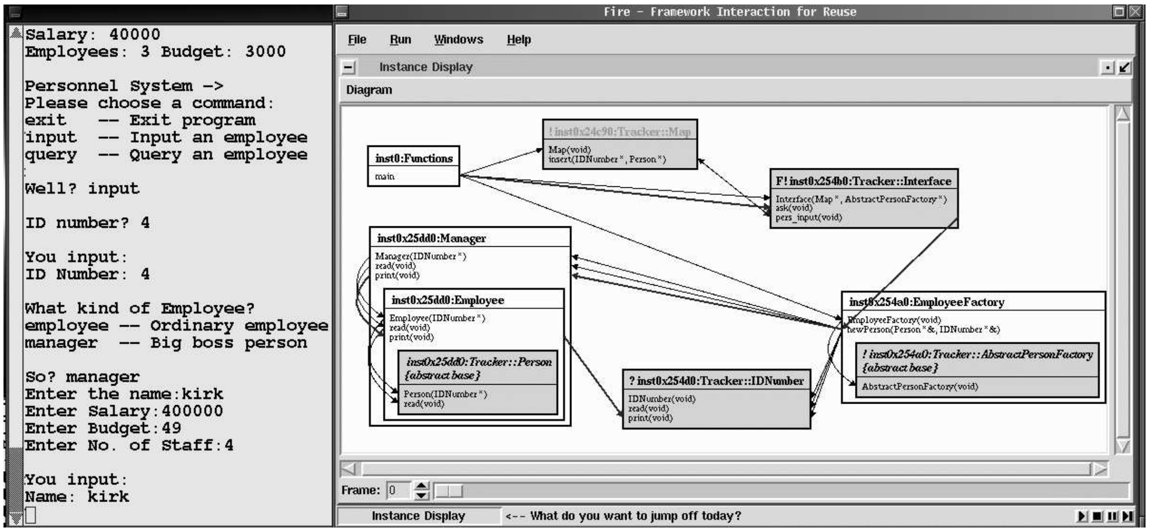

Figure 2: A complete instance diagram. The objects that come from classes in the framework are shaded. The calls on the current stack are shown in a different colour. The (separate) window on the left shows the text interface of the application being visualised.

sequence diagram, collaboration diagram, class diagram, and the object diagram. Object diagrams are a particular form of the class diagram, which shows individual instances and data values. The UML specification notes that the use of object diagrams is fairly limited, mainly to show snapshot examples of data structures.

The one change we have made to the UML diagrams we use is to add some degree of animation, hence "dynamic diagrams". Static diagrams are constrained by the lack of dimensions in which they can represent information. A diagram drawn in two dimensions has difficulty expressing both spatial and temporal information. A dynamic diagram can show extra information through movement and time.

Any object-oriented program can involve more than one instance of a class, and this is also true of programs based on frameworks. Class diagrams do not show the relationship between multiple objects when some of those objects are instances of the same class. Object diagrams do show this information, and, from the usage scenarios in section 2, we know that inheritance plays an important role in the use of frameworks, which is shown in class diagrams but not object diagrams. Furthermore, to understand how the inversion of control affects the behaviour of frameworks, it would be useful to see flow of control, which is shown only in sequence or collaboration diagrams. We have developed a variation of the collaboration diagram, called an *instance Diagram*, that has the aspects of each of the standard UML diagrams that we need to help understand frameworks.

The instance diagram shows the interactions between instances over time. It also shows some of the static structure, such as generalization (inheritance), aggregation and composition, and other associations.

Objects in an instance diagram are represented as a rectangle containing three portions, similar to the UML class diagram. The top section contains the instance and class name, the middle section the attributes and nested instances of strongly composed (aggregated) classes, and the bottom section the methods. The bottom section also shows inherited superclass "instances" nested inside. This allows showing the self-calls that many frameworks rely on.

Figure 1 shows an instance of a Manager class. The Manager class inherits from an Employee class, which in turn inherits from a Person class.

Instances can also have static relationships with other instances. Such relationships, such as the aggregation relationship, can be drawn in the same manner as in a UML class diagram, with lines between objects annotated to show type, -arity and direction.

Dynamic information can also be shown in an instance diagram. Instances and lines between instances, as well as annotations, may appear and vanish over the course of a program, a scenario, or part of a program's execution. This allows the viewer of an instance diagram to visualise the effects of the execution of the program.

When messages are passed between or within instances, an animated arrow from the source method to the destination method is drawn. The arrow remains highlighted while the method executes, and changes to unhighlighted when the next method executes in the program. Message arrows show the passing of messages between instances, and self calls within the instance. Figure 2 shows an instance diagram at a point in the execution of a program.

4 The Fire Prototype

In this section we outline a prototype design that will support the visualisation of framework interactions called Fire (Framework Interaction for REuse), which aids *the identification and understanding of the critical interactions between framework and user objects.* We are interested in supporting programmers learning to use frameworks. For the prototype, we chose students in a second year computer science course as a representative audience. Our prototype works with C++ frameworks, and provides diagrams based on UML, both of which are familiar to the students. The tool works with C++ programs compiled in the usual way, and allows a description of any framework involved by use of a framework information language, as described below. The prototype then provides dynamic visualisation of the running program, highlighting the role of frameworks.

4.1 A Framework Information Language

We have created a framework description language called *fwi* to add extra information to our visualisation — showing the context of different classes, and their suggested usages. A framework description language gives a concise, machine-parsable format that can be easily read into a visualisation program to increase its usefulness. Wilson and Wilson's *FDL* was a previous attempt at a textual framework description language, but their language had different intended uses, and is a less precise and powerful language for framework description our purposes.

The language we have developed captures much of the information traditionally represented in long textual documentation, such as whether it is better to inherit from a class, or invoke a method call to the class, and stores this information in a human and machine readable format. The *fwi* can use "+", "/", and "-" to represent public, protected, and private respectively, as well as "!" and "?" to represent "must" and "should". We devised the following list of information that a framework user needs to know when looking at a particular framework class, presented in the table below.

	Must !	Should ?	Public +	Protected /	Private -	Final
Class Instantiation	√	√				
Class Inheritance	√	√	√	√	√	√
Method Override	√	√				√
Method Call	√	√	√	√	√	
Member Variables			√	√	√	

- *Instantiation: Must* this class be instantiated?
  In some frameworks there are certain classes for which there must be an instance.
- *Inheritance*: Should this class be inherited from? Must it be inherited from? Is it indeed possible to inherit from this class?

It may be important not to subclass certain framework classes. Other situations may require that a class is subclassed, such as the case of an abstract base class in C++, which cannot be instantiated — but also a framework may require that non-abstract base classes be subclassed. For example, some classes may be tightly coupled, so that if one is subclassed, they all must be, to achieve the desired customisation.

- *Method Override:* Is it important that we override a method when inheriting from this class? Maybe it is just suggested, or perhaps we can't override the method.
  In many frameworks, base classes are provided for an application developer to subclass, and often the developer will only want to change the behaviour of the class slightly.

- *Method Calls*: Is the method public, protected or private? Is it suggested that we always call this method of the class, and must we?
  Some frameworks have a start method, which must be called for anything at all to happen. Apart the from general access public, private, and protected, we provide the "Must" and "Should" keywords in our fwi to show which methods are important to call.

- *Member Variables*: Can a field of a class be accessed? By subclasses?
  The usual public, protected, and private access specifiers indicate the access scope of variables, we have included them in the *fwi* for convenience.

This *fwi* format is designed to resemble C++, and is quickly understood by a human reader, and easily also parsed by program. Below is a portion of an *fwi* file for a small framework.

```
framework Tracker {
  public:
    abstract base class Person {
      protected:
        IDNumber *id;
        string name;
      public ?:
        Person(IDNumber *);
      public:
        should override virtual void read(void);
        must override   virtual void print(void);
    };
  public !:
    abstract base class AbstractPersonFactory {
      public ?:
        must override virtual
               int newPerson(Person *&, IDNumber *&);
    };
  .....
};
```

The framework name is `Tracker`, and it has two classes, `Person` and `AbstractPerson-Factory`. Both the `Person` and the `AbstractPersonFactory` are abstract base classes, and so must be subclassed. The `newPerson` method in the `AbstractPersonFactory` class must be overridden, and the `read` method in the `Person` class should be overridden. `AbstractPer-sonFactory` is a class that *must* be instantiated, as it has the `public !` access specifier. The `Person` constructor, and the `newPerson` methods *should* be called, as they have the `public ?` access specifiers.

4.2 Implementation Technology

Tcl/tk was chosen as the language to implement our prototype, as it provides a rich library of graphical and program manipulation commands. Also, it has already proven its worth as a rapid application development language[1]. Tcl/tk is a scripting language with a built in graphical toolkit, providing most of the elements necessary to build user interfaces[5]. The *[incr] tcl* and *[incr] tk* extensions to tcl/tk were used in *Fire* to allow an object-oriented design[9].

The GNU source level debugger *Gdb* was used to extract information about the running C++ program. *Gdb* allows us to capture information such as method calls, object creation, the inheritance hierarchy, the values of variables, whether calls are self calls or to other objects, and much more. *Gdb* has a very consistent output structure, meaning that it can be used in the same way on any platform. The *Expect* library[3] for tcl/tk has been used to wrap the execution of *Gdb*, sending commands to the running debugger, and capturing its output. *Expect* is useful, as *any* textual program may be wrapped in this way, and *Expect* allows an executing C++ program to continue unhindered while it is run through Gdb.

4.3 PMV

We have designed the *Fire* prototype to follow the PMV model of program visualisation, developed by Noble [4]. The model consists of three components:

- Program Component: Collects information about the program to be visualised, transforms it and sends it to the mapping component.
- Mapping Component: Translates the actions sent by the program component, and turns them into a format that can be used to update the visualisations.
- Visualisation Component: Receives directives from the mapping component, and turns them into a visual form, possibly updating existing pictures to reflect the changes that the program caused.

4.3.1 Program Component

In our situation, the target program is an executing C++ program supplied by someone who wants to help a prospective framework user understand the framework that they intend to use. The method of obtaining actions is through the direct execution of code. This is extremely useful in our situation, as it means that there is no need for source code annotation or compiler modification. This increases the audience of programs that can be visualised, as any program can be visualised by the prototype without modification.

The actions that are sent to the mapping component varies from instance creation to method call and return. When an action is detected in the running C++ program, execution is suspended while the action is sent to the mapping component. In *Fire*, the program component interacts directly with the mapping component, the *GdbInfo* class.

4.3.2 Mapping Component

The mapping component translates actions from program component, and does sensible filtering and summarisation of data before sending it on to the visualisation component. In *Fire*, there are two classes that perform the role of the mapping component. One class, the *GdbInfo* wraps around the execution of the running program, while the *FwiInfo* class gives additional information about the framework that cannot be gleaned from the running program.

The GdbInfo class generates events both at the time of selecting a program to run (static events), and while the program is running (runtime events). The different events are:

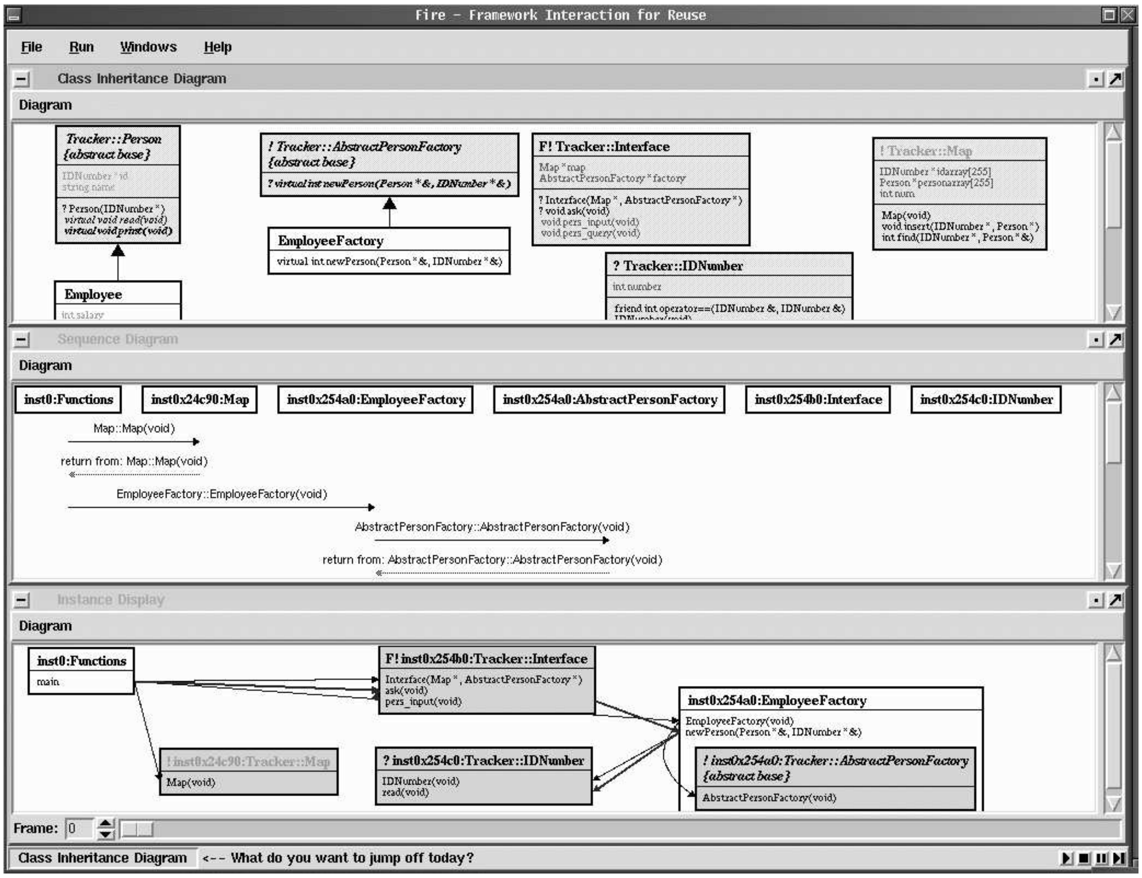

Figure 3: The main interface of the Fire prototype, showing a class diagram, a sequence diagram, and an instance diagram.

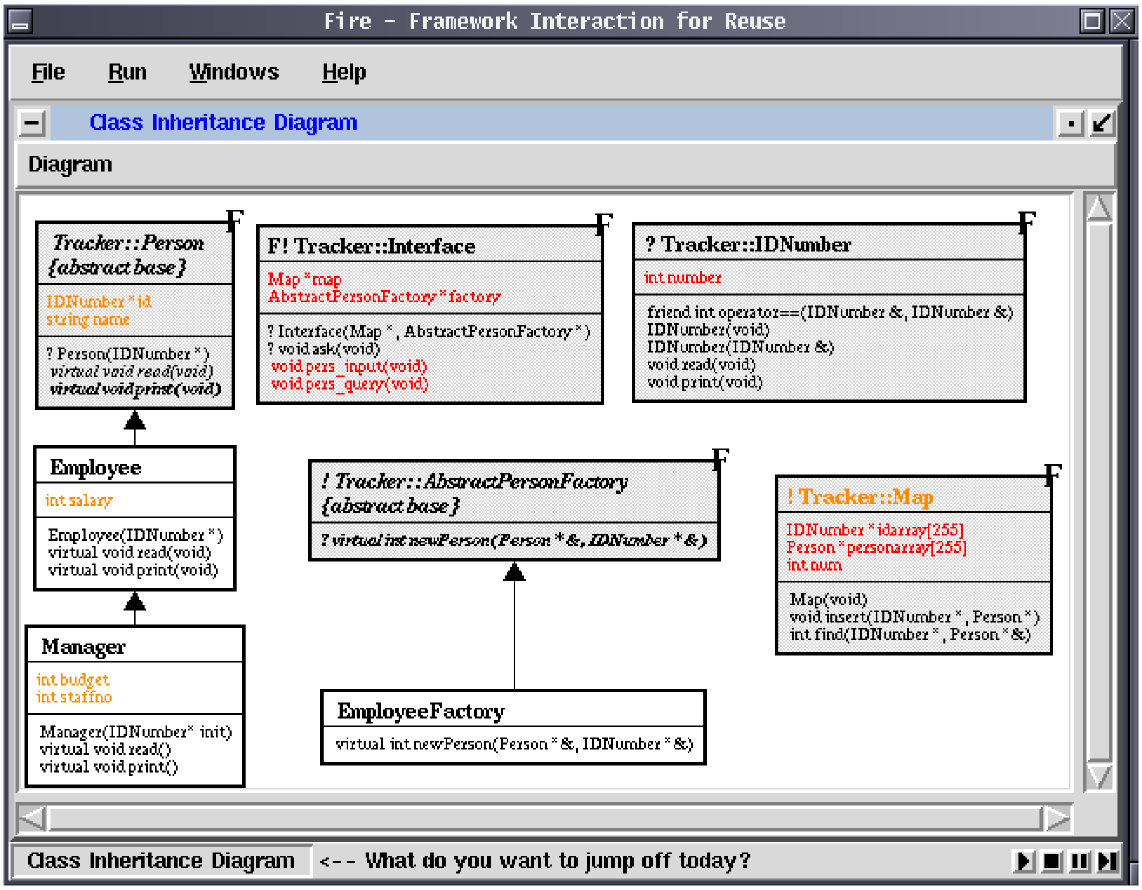

Figure 4: A class diagram displaying the inheritance in the program.

- **newtype** events are sent when the GdbInfo discovers a new classname that the visualisation component has not been notified of.
- **inherits** events are sent when the GdbInfo discovers that one class inherits from another, and sends the inherited class name, and its derived classes name to the visualisation component.
- **newinstance** events are sent when a new instance of a class is created, either as a local variable, a field, or a base class.
- **call / return** events are sent when a method is called between two instances.

The newinstance and the call / return events contain the name of the class that the instance was created from, as well as the memory address of the objects, which is used as an unique identifier.

The FwiInfo class is used to augment the runtime data that can be determined from the running program. It loads in a textual description of the framework (the fwi syntax in section 4.1) and extracts the information from the description, sending out a series of events, with class names, access specifiers, methods, and the other information stored in the fwi format. This allows the visualisation components to be improved with the additional information about the framework classes.

Communication between the program component classes and the visualisation classes occurs through event notification as in the *"Observer Pattern"*. The *GdbInfo* class is a subject, and the Mapping classes that wish to receive events attach themselves as observers to the class. The *FwiInfo* class behaves similarly, except the FwiInfo also sends events to the GdbInfo class as well as the mapping classes.

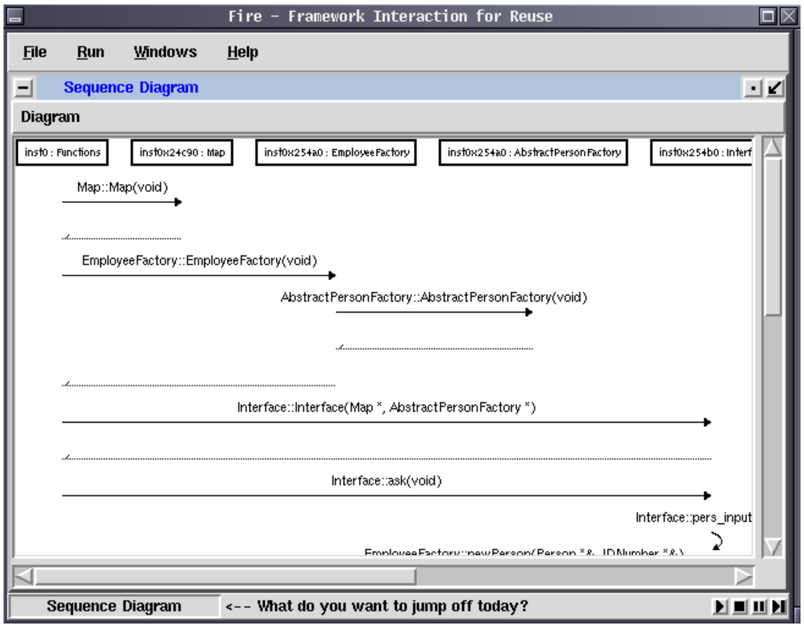

Figure 5: A sequence diagram from *Fire*.

- "**framework** frameworkname" events are sent when the fwi file is read, and a framework name is found.
- "**framework** . . . **class** ClassName" events are sent for each new class in the fwi file, along with the access specifiers for that class.
- "**framework** . . . class ClassName . . . methodsignature" events are sent for each method, along with the methods access specifiers.
- "**framework** . . . class ClassName . . . fieldsignature" events are sent for each fields, along with the fields access specifiers.

4.4 Visualisation Component

The visualisation component receives the changes from the mapping component, and updates the visualisations to reflect these changes. Our prototype is designed to visualise three different *views* of a running program: a static UML class diagram, a UML sequence diagram, and an instance diagram. The prototype allows simultaneous viewing of any of these, and also allows the user to enlarge or reduce their sizes as shown in figure 3. The visualisations are animated, and updated immediately when the events are received. They are also interactive, and the user can move objects around on the screen to customise the layout.

The purpose of the diagrams is to help users understand how the frameworks are used. They provide this help by using the fwi file information to distinguish classes and objects within the frameworks, and highlight their usage guidelines. The dynamic display allows the user to follow the flow of control, especially important to understanding frameworks because of the effects of inversion of control, and the polymorphic effects of subclasses and overridden methods. To allow control flow to be followed more carefully, the prototype also allows the user to pause program execution, and to hide irrelevant classes or objects.

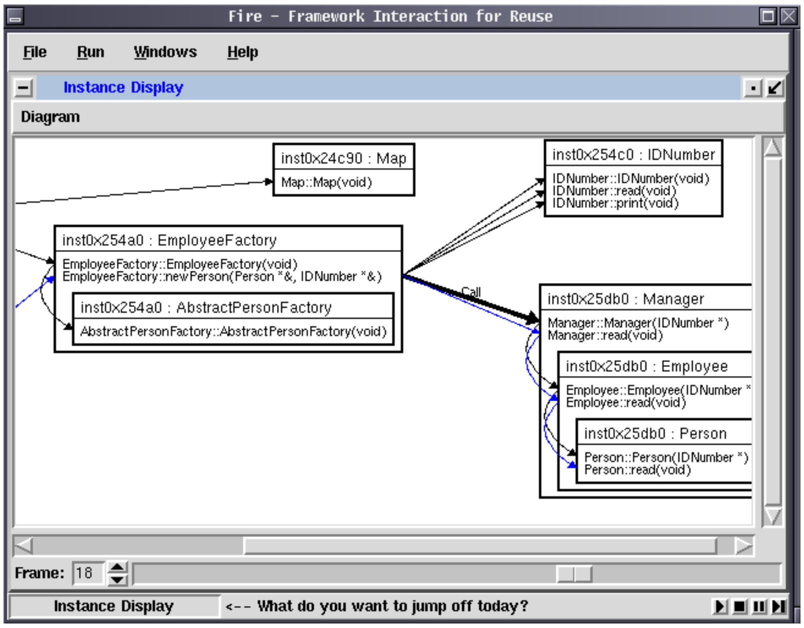

Figure 6: An instance diagram from *Fire*. The animation slider is at frame 18, so the 18th calling is highlighted.

4.4.1 ClassInheritanceView

The ClassInheritanceView responds to **newtype** and **inherits** events from the GdbInfo by drawing a UML class diagram showing classes and their inheritance relationships. The class diagram is also augmented with information sent by the FwiInfo class (such as the name of the framework). The resulting class diagram contains all the inheritance relationships within the program, but it does not display the associations between classes (figure 4). This is merely a constraint on the scope of the prototype. A full system would ideally produce a diagram showing all relations, not just the inheritance relationships.

4.4.2 SequenceView

The SequenceView responds to the message call events sent from GdbInfo by drawing a UML sequence diagram showing the methods being called by one instance to another over time. This diagram is generated solely on dynamic information, and is animated in real time as the program executes (figure 5). The SequenceView implements the full set of specifications for the UML sequence diagram, with calls between instances and self calls on an instance. It does not represent actors however, which are commonly drawn in sequence diagrams. With our prototype, the actor is the person running the prototype, so this is not considered to be a limitation.

4.4.3 InstanceView

The InstanceView class responds to all of the events passed by the GdbInfo and the FwiInfo objects, drawing an Instance Diagram of the messages within and between different instances. The instances displayed in the InstanceView classes show their inherited base classes, the method calls both to other instances, and within the current instance (in the case of self calls), and represent all the information stored in the fwi file and sent via the FwiInfo events (figures 2 and 6). The prototype includes display of nested derived classes, and animated links showing method calls and returns.

The instance diagram also allows the user to highlight specific method calls in order by scrolling forward or backward. This allows the user to carefully examine the precise control flow from object to object, with the role of the framework made clear. (See figure 6.)

5 Conclusion

Object-oriented frameworks are hard to learn. There are complex interactions and interdependencies between user and framework classes that a programmer must comprehend before being able to use the framework productively to make new applications.

We have attempted to support programmers understanding of framework code by providing animated visualisations of example programs interacting with the framework.

We created a textual representation of object-oriented framework information to supplement the information available by probing the running program, communicating to the viewer the intentions and the design decisions of the framework developers through visualisation.

We used the common forms of the Unified Modeling Language to represent the visual information, and augmented the static diagrams with the dynamic instance diagram. The instance diagram is a useful and compact way to display a large amount of behavioural information, and the ability to move through time and toggle the display of certain information helps reduce the cognitive overload on the viewer.

Our prototype tool was developed using relatively lightweight and non-invasive technology, allowing use without modification to programs or to system software. The general approach would be applicable to many programming languages and development platforms.

The goal of Fire is to help programmers understand frameworks more effectively than current methods. We have not yet established that it in fact does so. We have been working on the first step to determining its effectiveness, which is to determine its usability. We have explored the effectiveness of our prototypes using standard techniques for usability evaluation, including trials with test-users from our target audience. Evaluation has suggested improvements in diagram presentation and interaction, but indicates that our overall approach is indeed useful to framework learners.

References

[1] Robert Biddle, Stuart Marshall, John Miller-Williams, and Ewan Tempero. Reuse of debuggers for visualization of reuse. In *Proceedings of the Fifth Symposium on Software Reusability*, pages 92–100, 1999.

[2] R. Johnson. Documenting frameworks using patterns. In *Proceedings of OOPSLA'92*, pages 63–76, Vancouver, B.C., Canada, October 1992. ACM Press.

[3] Don Libes. expect: Curing those uncontrollable fits of interaction. In *Proceedings of the Usenix Summer 1990 Technical Conference*, pages 183–192, Berkeley, CA, USA, June 1990. Usenix Association.

[4] James Noble. *Abstract Program Visualisation.* PhD thesis, Victoria University of Wellington, 1996.

[5] John K. Ousterhout. The Tcl programming language. *Dr. Dobb's Journal of Software Tools*, 19, 1994.

[6] Wolfgang Pree and Hermann Sikora. Design patterns for object-oriented software development. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 663–664. ACM Press, 1997. Tutorial.

[7] Don Roberts and Ralph Johnson. Evolving frameworks: A pattern-language for developing object-oriented frameworks. In *PLoPD3*, 1996.

[8] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[9] Mark L. Ulferts. [incr Widgets] an object-oriented mega-widget set. In USENIX Association, editor, *Proceedings of the Tcl/Tk Workshop, July 6–8, 1995, Toronto, Ontario, Canada*, Proceedings of the Tcl Tk Workshop — USENIX Association 1995; 3rd, pages 61–76, Berkeley, CA, USA, July 1995. USENIX.