Essential Software Structure through Implicit Context

by

Robert James Walker

B.Sc. (Geophysics, Major Program), University of British Columbia, 1992
B.Sc. (Computer Science, Honours Program), University of British Columbia, 1994
M.Sc. (Computer Science), University of British Columbia, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

March 2003

© Robert James Walker, 2003
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date 7 March 2003
Abstract

Software reuse and evolution are problematic. Modules tend to express a great deal of knowledge about the external modules, large-scale structure, and behaviour of the systems in which they reside. As a result, the reusability of individual modules is decreased as they are too dependent on other modules. Likewise, the evolvability of systems is decreased as they are too dependent on the details of individual modules.

However, we must have some dependences between our modules, or they would not be able to operate together. Not all dependences are equally bad, then. Each module can be seen to have a minimal core, an essential structure, beyond which it makes no sense to reduce. This essential structure describes the basic responsibilities of that module, and its expectations of the responsibilities of external modules. The thesis of this dissertation is that expressing the essential structure of our software modules, through the use of implicit context, makes those modules easier to reuse and the systems containing those modules easier to evolve.

Implicit context revolves around the notion that we must abandon the need for all modules to be defined with respect to an absolute frame of reference. Instead, modules can be defined relative to a reference frame of convenience. In order for modules to work together, the inconsistencies in their independent world views must be reconciled when they attempt to communicate. Implicit context provides the novel mechanism of contextual dispatch to perform this reconciliation. When any communication passes into or out of a module, that communication may be rerouted, altered, replaced, or discarded altogether, as the situation dictates. This translation process sometimes requires access to information about the communication history of the system; objects previously passed can be retrieved, or the appropriate recipient of a message can be determined.

The dissertation describes a prototype tool that supports the application of implicit context to Java source code. The thesis is validated by applying this tool in two case studies: comparing the evolution of an implicit context-based implementation of an FTP server to an object-oriented implementation; and reusing the Outline View of the Eclipse integrated development environment in a different application, even though the Outline View was not designed for such reuse.
# Contents

Abstract ii  
Contents iii  
List of Tables vii  
List of Figures viii  
Acknowledgments xi  
Trademarks xiii  

1 Introduction  
1.1 Overview of Implicit Context ................................................... 2  
1.2 A Simple Example ...................................................................... 4  
1.3 A Brief Survey of Evolution and Reuse Techniques .................. 6  
  1.3.1 Ad Hoc Approaches ............................................................. 6  
  1.3.2 Product-, Architecture-, and Interface-Centric Approaches .... 8  
  1.3.3 Generative Approaches ......................................................... 9  
1.4 Organization of the Dissertation .............................................. 10  

2 Motivation  
2.1 Reusing the Outline View of Eclipse ........................................ 13  
2.2 Copy-and-Modify ....................................................................... 17  
2.3 Refactor ..................................................................................... 18  
  2.3.1 Dependences in makeContributions ........................................ 19  
  2.3.2 Refactoring without Changing the Class Hierarchy .............. 21  
  2.3.3 Refactoring through an Abstract Class .................................. 23  
2.4 Summary ................................................................................... 26
6.2 Evolution of a File Transfer Protocol Server .................................................. 117
  6.2.1 FTP Concepts .................................................. 117
  6.2.2 The Procedure .................................................. 118
  6.2.3 Results and Lessons Learned .................................................. 134
6.3 Summary .................................................. 141

7 Discussion .................................................. 143
  7.1 Local Execution .................................................. 143
    7.1.1 Event Traces .................................................. 145
    7.1.2 Reconciliation of Local and Global Execution .................................................. 149
  7.2 Communication History: Use and Efficient Support .................................................. 155
    7.2.1 Communication History Queries as an Expression of Minimal Dependencies .................................................. 155
    7.2.2 Communication History Queries as Pattern Matches .................................................. 156
    7.2.3 Efficient Support of Communication History Queries .................................................. 158
  7.3 Implicit Context and Aspect-Oriented Programming .................................................. 159
    7.3.1 Outline of Concepts in AspectJ .................................................. 159
    7.3.2 Describing AspectJ in Terms of Event Traces .................................................. 161
  7.4 Summary .................................................. 165

8 Related Work .................................................. 167
  8.1 Principles .................................................. 167
    8.1.1 Separation of Concerns .................................................. 168
    8.1.2 Dependences, Coupling, and Cohesion .................................................. 168
    8.1.3 Local Reasoning .................................................. 168
    8.1.4 Design-for-Change .................................................. 169
    8.1.5 Reflection .................................................. 170
  8.2 Ad Hoc Approaches .................................................. 171
    8.2.1 Copy-and-Modify .................................................. 171
    8.2.2 Refactoring .................................................. 171
  8.3 Interface-, Architecture-, and Product-Centric Approaches .................................................. 172
    8.3.1 Inheritance, Polymorphism, and Frameworks .................................................. 173
    8.3.2 Components and Middleware .................................................. 175
    8.3.3 Product Lines .................................................. 177
    8.3.4 Adaptors and Wrappers .................................................. 178
  8.4 Generative Approaches .................................................. 180
    8.4.1 Generic Programming .................................................. 181
    8.4.2 Generators .................................................. 182
    8.4.3 Collaboration-Based Design .................................................. 183
8.4.4 Aspect-Oriented Programming .............................................. 184
8.5 Other Related Topics .......................................................... 186
  8.5.1 Essential Structure ....................................................... 186
  8.5.2 Context .................................................................... 187
  8.5.3 Multiple World Views .................................................... 188
  8.5.4 Names ......................................................................... 188
  8.5.5 History and Traces .......................................................... 189
8.6 Summary .............................................................................. 190

9 Conclusion ............................................................................. 191
  9.1 Contributions ................................................................... 192
  9.2 Future Work ..................................................................... 193

Bibliography .............................................................................. 195

Appendix A A Brief Overview of Notation ........................................ 217
  A.1 Square Boxes .................................................................. 217
  A.2 Rounded Boxes ............................................................... 218
  A.3 Lines and Arrows ............................................................... 219
  A.4 Filled Circles .................................................................. 221
  A.5 Jagged Polygons ............................................................... 222

Appendix B A Brief Overview of the Abstract Factory Design Pattern ... 223

Appendix C Listing of makeContributions ........................................ 225

Appendix D The Application Programming Interface to Call History .... 228
  D.1 The History Class ........................................................... 228
  D.2 The Call Class ................................................................. 234

Appendix E Example Results of Applying the Prototype Tool ................ 236
  E.1 Before Applying the Tool .................................................. 236
    E.1.1 Contents of SomeClass.java ........................................ 236
    E.1.2 Contents of OtherClass.java ....................................... 236
    E.1.3 Contents of example.map ........................................... 237
    E.1.4 Contents of apply.app ............................................... 237
  E.2 After Applying the Tool ....................................................... 238
    E.2.1 Contents of icworkingdir/SomeClass.java ...................... 240
    E.2.2 Contents of icworkingdir/OtherClass.java ...................... 242
List of Tables

5.1 Interpretation and support for capturing communications, for \textit{in} event designators, by each kind of boundary .......................... 76
5.2 Interpretation and support for capturing communications, for \textit{out} event designators, by each kind of boundary ....................... 77
5.3 Interpretation and support for capturing communications, for \textit{gets} and \textit{sets} event designators, by each kind of boundary ..................... 77
5.4 The query methods defined on the \textit{History} class ............................... 87
5.5 The interface to the \textit{Call} class ........................................ 87
6.1 Statistics on Eclipse and reusing the Outline View ............................. 111
6.2 Statistics on synthesizing and evolving the FTP server ......................... 134
## List of Figures

1.1 A design utilizing the Abstract Factory design pattern .......................... 4  
1.2 Supporting independent world views in conjunction with the Abstract Factory design pattern example ................................... 7  

2.1 Screenshot of the Eclipse integrated development environment ................. 14  
2.2 Reusing the Outline View in another application ................................ 15  
2.3 Dependences of the JavaOutlinePage class ........................................ 16  

3.1 Supposed, actual, and ideal dependences involving extraneous identification 36  
3.2 An extraneous parameter arises due to the mismatch between minimal specifications and implementations ................................. 37  

4.1 Overlapping boundaries ................................................................. 47  
4.2 Nested boundaries ........................................................................... 47  
4.3 Performing contextual dispatch to use the Abstract Factory design pattern 49  
4.4 Performing contextual dispatch to retrieve state through an added method 51  
4.5 Retrieving state through the use of the communication history ............... 52  
4.6 An implementation of makeContributions expressing only its essential structure ................................................................. 55  
4.7 The original implementation of makeContributions ............................... 56  
4.8 Using communication history to retrieve discarded information ............ 59  
4.9 Altering an incoming message to hide the presence of EEK ..................... 61  

5.1 Using the Abstract Factory design pattern to isolate platform-specific dependencies ................................................................. 66  
5.2 The operation of boundary maps attached to CompoundWidget ............... 68  
5.3 Overview of the operation of the prototype tool .................................... 72  
5.4 BNF syntax for mapsets ...................................................................... 74  
5.5 BNF syntax for capture clauses ........................................................... 75  
5.6 BNF syntax for calls to proceed ......................................................... 82  
5.7 BNF syntax for modifiers .................................................................... 84
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8</td>
<td>A sample query method on the History class</td>
<td>88</td>
</tr>
<tr>
<td>5.9</td>
<td>An example communication history</td>
<td>90</td>
</tr>
<tr>
<td>5.10</td>
<td>BNF syntax for renaming maps</td>
<td>92</td>
</tr>
<tr>
<td>5.11</td>
<td>BNF syntax for tool directives</td>
<td>93</td>
</tr>
<tr>
<td>6.1</td>
<td>The Eclipse integrated development environment</td>
<td>105</td>
</tr>
<tr>
<td>6.2</td>
<td>The module resulting from the reuse of the Outline View</td>
<td>109</td>
</tr>
<tr>
<td>6.3</td>
<td>The Outline View reused as an interface to my prototype tool</td>
<td>110</td>
</tr>
<tr>
<td>6.4</td>
<td>An example mapset performing a communication history query, Part 1</td>
<td>115</td>
</tr>
<tr>
<td>6.5</td>
<td>An example mapset performing a communication history query, Part 2</td>
<td>116</td>
</tr>
<tr>
<td>6.6</td>
<td>Evolving versions of the FTP server</td>
<td>119</td>
</tr>
<tr>
<td>6.7</td>
<td>Initial design for the purely object-oriented version of the FTP server, Part 1</td>
<td>121</td>
</tr>
<tr>
<td>6.8</td>
<td>Initial design for the purely object-oriented version of the FTP server, Part 2</td>
<td>122</td>
</tr>
<tr>
<td>6.9</td>
<td>Initial design for the implicit context version of the FTP server, Part 1</td>
<td>124</td>
</tr>
<tr>
<td>6.10</td>
<td>Initial design for the implicit context version of the FTP server, Part 2</td>
<td>125</td>
</tr>
<tr>
<td>6.11</td>
<td>The FTP-specified state machine for authentication</td>
<td>127</td>
</tr>
<tr>
<td>6.12</td>
<td>The FTP-specified state machine for passive mode</td>
<td>127</td>
</tr>
<tr>
<td>6.13</td>
<td>Evolved design for the object-oriented version of the FTP server, Part 1</td>
<td>128</td>
</tr>
<tr>
<td>6.14</td>
<td>Evolved design for the object-oriented version of the FTP server, Part 2</td>
<td>129</td>
</tr>
<tr>
<td>6.15</td>
<td>Evolved design for the implicit context version of the FTP server, Part 1</td>
<td>130</td>
</tr>
<tr>
<td>6.16</td>
<td>Evolved design for the implicit context version of the FTP server, Part 2</td>
<td>131</td>
</tr>
<tr>
<td>6.17</td>
<td>Cross-evolved design of the FTP server</td>
<td>133</td>
</tr>
<tr>
<td>6.18</td>
<td>Using communication history to implement the Abstract Factory design pattern</td>
<td>137</td>
</tr>
<tr>
<td>6.19</td>
<td>Boundary map applied to command interpreters to alter their behaviour on the basis of whether the session has been authenticated, Part 1</td>
<td>139</td>
</tr>
<tr>
<td>6.20</td>
<td>Boundary map applied to command interpreters to alter their behaviour on the basis of whether the session has been authenticated, Part 2</td>
<td>140</td>
</tr>
<tr>
<td>A.1</td>
<td>Square boxes denoting types</td>
<td>217</td>
</tr>
<tr>
<td>A.2</td>
<td>Square boxes denoting instances and boundaries</td>
<td>218</td>
</tr>
<tr>
<td>A.3</td>
<td>Rounded boxes</td>
<td>218</td>
</tr>
<tr>
<td>A.4</td>
<td>Lines and arrows denoting attribute relationships</td>
<td>219</td>
</tr>
<tr>
<td>A.5</td>
<td>Lines and arrows denoting composition relationships</td>
<td>219</td>
</tr>
<tr>
<td>A.6</td>
<td>Lines and arrows denoting subtyping relationships</td>
<td>220</td>
</tr>
<tr>
<td>A.7</td>
<td>Lines and arrows denoting control and data flow</td>
<td>220</td>
</tr>
<tr>
<td>A.8</td>
<td>Lines and arrows denoting dependence relationships</td>
<td>221</td>
</tr>
<tr>
<td>A.9</td>
<td>Filled circles denoting message interception</td>
<td>221</td>
</tr>
<tr>
<td>A.10</td>
<td>Jagged polygon denoting call history</td>
<td>222</td>
</tr>
</tbody>
</table>
B.1 Static structure of the Abstract Factory design pattern. . . . . . . . . . . . . . . . 223
E.1 General structure of instrumentation for methods and constructors . . . . . 239
Acknowledgments

It was a dark and stormy night in the Imager Computer Graphics Laboratory. I haunted the corridors of a Counterstrike world, inflicting nightmarish vengeance upon those who opposed me. And then I thought of my thesis topic....

Or something like that. There were definitely nights, Imager, and Counterstrike involved. All of which were made more bearable by my friends and colleagues there, including David Bullock, Bill Gates, Rob Scharein, Lisa Streit, Valerie Summers, Roger Tam, and Marcelo Walter. Many interesting technical discussions. Much help and advice. But most of all: Good times.

Anne Lavergne, who also started as an Imagerite, taught me much: some good, some bad. I would take the same route over again. Things fall apart.

The second half of my grad student life began by haunting the dark corners of the basement of the CICSR building in what came to be known as the Sensory Deprivation Chamber. The Software Engineering Group being the newest and smallest group in the department, the battle for real estate was prolonged. In those days, the only other grad student around was Elisa Baniassad, another friend and colleague. Eventually, we eaked out a corner of the Distributed Systems Group Laboratory, and Martin "I hate that" Robillard joined our merry band. Ah, Martin, you can always be counted on to be up for beer no matter what corner of the planet we find ourselves in—as it should be.

At some point, I ran off for the summer to IBM T.J. Watson Research Center in Hawthorne, New York. There, I met Peri Tarr and Harold Ossher, whose work helped drive some of my thoughts on my thesis. I also met Siobhan Clarke, another summer intern, with whom a close friendship and work collaboration developed.

The Software Engineering Group grew into the Software Practices Laboratory, and new faces arrived on the scene. We were not as boisterous and close a group as the circle of friends in Imager, but the atmosphere was collegial, and much help was offered by all. Of particular note were Brian de Alwis, Kris De Volder, Gregor Kiczales, and Jonathan Sillito. Yvonne Coady joined us eventually, and we all purchased ear plugs. But deafness is a small price to pay for her energy and incite (not to mention her insight).

There are many others who have helped over the years in so many ways, personal and professional. You are not forgotten, but there is only so much into which I wish to delve.
And finally, there is Gail Murphy: the ideal mentor, a fun colleague, and a good friend. I would not be finishing this dissertation now, and especially not in software engineering, were it not for you. I cannot find the words to express my appreciation and admiration for you. Thank you for everything, but especially, for being human and showing me that there is nothing wrong with that.

Thank you all. A Ph.D. is a lonely road, but you were the music that brightened the way.

ROBERT JAMES WALKER

The University of British Columbia
March 2003
Trademarks

"POPL" and "Principles of Programming Languages" are registered trademarks of the Association for Computing Machinery. "LEGO" is a registered trademark of The LEGO Group. "Microsoft" and "Windows" are registered trademarks of Microsoft Corporation. "CORBA" is a registered trademark of Object Management Group; "Unified Modeling Language" is a trademark of Object Management Group. "Motif" and "UNIX" are registered trademarks of The Open Group. "AspectJ" is a trademark of Palo Alto Research Center Inc. "Java" is a trademark of Sun Microsystems, Inc.
Chapter 1

Introduction

Software systems evolve [Belady and Lehman, 1976; Lehman and Parr, 1976; Parnas, 1994; Svahnberg and Bosch, 1999; Eick et al., 2001]. The need to correct errors, to introduce new features, and to support different platforms all exert pressure upon our existing systems. Software is expensive to create; therefore, we would rather modify our systems to accommodate new requirements, rather than construct wholly new systems. Similarly, when pieces of existing systems provide functionality that is needed in new systems, there is a potential savings in reusing those pieces within the new systems [McIlroy, 1969; Lanergan and Grasso, 1984; Krueger, 1992; Sen, 1997].

Software systems are modularized in the hope that changes can be isolated to individual modules without affecting the remainder [Parnas, 1972; Korson and Vaishnavi, 1986; Sullivan et al., 2001]. Consider a particular module in a system: it will depend on other modules in that system and other modules in that system will depend upon it. When modules are too interdependent, too coupled, problems arise [Stevens et al., 1974]. We are limited in our ability to modify a module because of the dependences\(^1\) upon its interface and externally visible behaviour. We are also limited in our ability to reuse a module outside of its current system because of its dependences on the interfaces and externally visible behaviour of other modules there. However, we must have some dependences between our modules, or they would not be able to operate together.

Each module can be seen to have a minimal core, an \textit{essential structure}, beyond which it makes no sense to reduce. This essential structure describes the basic responsibilities of that module, and its expectations of the responsibilities of external modules. Essential structure is often abstract: a module may care that an external service exist and that it behave within certain constraints, but otherwise may be oblivious to the exact details of a concrete interface to such a service.

Dependences that are not essential to the structure of our modules, which I call \textit{extra-}\(^1\)“Dependence” is used synonymously with “dependency” in this dissertation [Barber, 2001].
neous embedded knowledge (EEK), occur in two ways. First, when an abstract description of an external service is made too concrete, we constrain the module in which that description occurs; we cannot reuse the module unless that exact concrete service is present. Second, systems often have large-scale concerns in addition to the small-scale concerns of our individual modules. Details of such large-scale concerns tend to get pushed into our small-scale modules. Once there, we cannot reuse the module in another system with different large-scale concerns. Once there, we cannot evolve our system to change its large-scale concerns without the modification of multiple modules, a time-consuming and error-prone process.

Ultimately, the presence within our modules of details of their contexts of operation cause us difficulty in evolving or reusing them. State-of-the-art programming mechanisms do not permit us the right level of abstraction to avoid such contextual details when implementing our modules. The thesis of this dissertation is that expressing the essential structure of our software modules, through the use of implicit context, makes those modules easier to reuse and the systems containing those modules easier to evolve.

In Section 1.1, I look in brief at implicit context, and consider how it can be used to express the essential structure of a simple example in Section 1.2. Section 1.3 gives a short survey of existing techniques for software reuse and evolution. The organization of the dissertation is described in Section 1.4.

1.1 Overview of Implicit Context

To ease software evolution and reuse, we must abandon the notion that all modules can be defined with respect to an absolute frame of reference. Such a reference frame would need to hold across all versions of a system and across all systems; otherwise, the constraints and names that are valid for one version of one system will not be valid for some other version or system.

Instead, I define modules relative to a reference frame of convenience. In traditional software development, this frame lies implicitly at the system-level: every module expresses only a subset of the names and constraints upon the system, but no conflict is permitted among these modules. Implicit context pushes the reference frame down to the level of modules: each module is permitted to express a set of names and constraints that are different from those expressed by every other module. With implicit context, each module is permitted an independent world view.

Since each module does not need to agree on a common frame of reference, it is free to express only those details of its operation that are of immediate importance from its local point-of-view—i.e., free to express its essential structure. Details of significance only to higher-level modules can be omitted, and dealt with separately.
When we compose modules to form a higher-level module, we must reconcile the differences in the world views of the lower-level modules to agree with the world view of the higher-level module. After all, the modules must ultimately work together to form a coherent system. But postponing the reconciliation as long as possible maximizes the flexibility of the modules. Each module only expresses what it must from its local perspective, and this expression is transformed to suit higher-level modules only when those higher-level modules are definitely to be used. Conceptually, the conflict in world views does not manifest itself until communication occurs between modules using different world views. Therefore, we can alter those communications to reconcile the differences as they cross between different world views.

Implicit context consists of three concepts: boundaries between conflicting world views, contextual dispatch which is used to alter communications, and communication history which is used to retrieve previous state when performing contextual dispatch.

Contextual dispatch is a generalization of object-oriented dispatch. Contextual dispatch may be performed on communications as they cross a boundary between conflicting world views. Rather than selecting the implementation of a method on the basis of the run-time type of a given object, we select the method or methods to execute on the basis of the context in which the communication occurs—both static properties and dynamic properties of the system. Contextual dispatch can be used to fill in concrete details in an abstract service request: a message can be rerouted to a concrete implementor of the service, with additional parameters filled-in.

Trouble can arise when the source of such additional parameters is not immediately obvious. Instead, communication history can be used. Communication history is, conceptually, a record of all communication that has occurred in the system, including all arguments passed and values returned. Through queries on communication history, we can retrieve state information of interest: the last passed argument of a particular type, for example. Communication history queries have the advantage that they can be expressed to minimize our assumptions about the system, as consistent with the principle of essential structure. Thus, communication history allows us to reduce the coupling in our systems, easing software evolution and reuse.

Let us now consider a simple example in which contextual dependences adversely affect the evolvability and reusability of a module within a system, and then see how essential structure through implicit context can remedy the situation.
1.2 A Simple Example

The Abstract Factory design pattern [Gamma et al., 1994] has become a standard means of reducing coupling in our systems in the hopes of easing evolution. Consider a system that involves the use of a graphical user interface (GUI); which concrete classes need instantiating to represent widgets (such as buttons, sliders, or text areas) depends on concrete details of the context in which the system exists, such as the operating system or hardware present. This design pattern allows some modules to be decoupled from knowledge about these concrete details of the context, in the hopes that the system can then be run on a wider variety of platforms; the specific design I describe is illustrated in Figure 1.1 using a notation based on the Unified Modeling Language [Object Management Group, 2000], but with extensions and liberties. See Appendix A for details of this notation, which is used throughout this dissertation.

Take a particular widget, such as a button. There are different concrete implementations of button for the Motif GUl library (which runs on UNIX platforms) and for Microsoft Windows. If we wish to isolate these differences from a module that just needs to create a button, we can create a small hierarchy of classes consisting of AbstractButton at the base. This abstract base class is then subclassed on the basis of which platform is supported: MotifButton implementing buttons for the Motif library and MSWindowsButton implementing buttons for Microsoft Windows.

A client module that needs to create buttons should be oblivious to which platform

---

2 See Appendix B for an overview of the Abstract Factory design pattern.
is in use, and thus, should express no knowledge of either MotifButton or MSWindowsButton. However, such a client cannot instantiate the abstract button base class to get an instance of the subclass appropriate for the current platform in use. Instead, this instantiation is done indirectly.

There is a hierarchy of classes parallel to the button hierarchy. At the base is an abstract class called AbstractFactory. This is subclassed on the basis of platform into MotifFactory and MSWindowsFactory. Some part of the system must know which platform-specific factory is appropriate for the current operating context; this part is then responsible for instantiating the correct factory subclass. AbstractFactory possesses an abstract method called createButton; this method is then implemented by each platform-specific factory to create the corresponding platform-specific version of button.

Now, a client module must possess a parameter of type AbstractFactory, through which is passed the concrete, platform-specific factory instantiated elsewhere. The createButton method is called on this factory, and a concrete, platform-specific instance of button is created—but without the client requiring knowledge of what platform is currently in use. The instance of button is then accessed through the interface defined by AbstractButton. This approach also allows the addition of subclasses supporting other platforms, without the need to modify any client modules.

Difficulties exist with this approach. Consider the nature of a client module using the Abstract Factory design pattern. In the absence of the pattern, the client would construct a button object through the call (using Java™ [Gosling et al., 2000] syntax):

```java
AbstractButton button = new AbstractButton();
```

But in the presence of the pattern, this call must instead become:

```java
AbstractButton button = factory.createButton();
```

where factory is a formal parameter of the module. If the client module is to use the Abstract Factory pattern, it is stuck with this constraint. Even if the client module is moved to a new system (through reuse or evolution) where the Abstract Factory pattern is not needed or inappropriate, it remains constrained to use it. This constraint is of importance to the system in which the client exists; the client module itself simply requires that a button be created. Thus, we again see the contextual dependences of a module reducing its reusability and evolvability.

The call above that directly instantiates AbstractButton represents the essential structure of this part of the client: the client must express that an instance of AbstractButton is to be created. It is only the system as a whole that must express that this instantiation is to take place via the Abstract Factory design pattern.
Implicit context allows for the separation of the concrete details of instantiation from the fact that an object must be created, as shown in Figure 1.2. There are two issues with which to deal when performing this separation. First, we want the module(s) calling the client to continue to see the same interface: these callers should still pass a factory for the client to use. But the client should be unaware that this information is being passed to it, because it should express no knowledge of the Abstract Factory pattern: the factory is effectively discarded. Second, when the attempt is made to instantiate AbstractButton, we can use contextual dispatch to replace this attempt with an actual call to the createButton method on the factory object that was passed earlier. This factory object is retrieved via a query to communication history.

Through the expression of essential structure via implicit context, I have made the client less dependent upon this particular system, allowing it to be reused in other versions or other systems where the Abstract Factory design pattern is not used. The client possesses a world view where Abstract Factory is not in place, but where concrete details of platforms remain unseen. The system as a whole possesses a different world view, one in which Abstract Factory is very much in use. Each world view is an expression of the essential structure appropriate at that level; implicit context allows their differences to be reconciled.

1.3 A Brief Survey of Evolution and Reuse Techniques

Many software reuse and software evolution techniques, approaches, and tools have been defined. These address a wide range of problems, including locating reusable modules of interest and re-engineering legacy systems; many of these problems, while important, lie outside the purview of this dissertation. Here, I assume that one has managed to locate software of interest, but requires the means to reuse or evolve it. There are three broad categories of prior work related to this thesis: ad hoc approaches; product-, architecture-, and interface-centric approaches; and generative approaches.

1.3.1 Ad Hoc Approaches

Ad hoc approaches are ones where manual intervention forms the basis for supporting change. Copy-and-modify and refactoring are representative examples. Copy-and-modify (also called scavenging) is likely the most common form of reuse, since it requires no special support from languages or tools [Krueger, 1992]; the programmer simply notices a similarity between an existing source code module and a desired one, makes a copy of it, and modifies the copy to the needs of the new situation. Refactoring [Opdyke, 1992; Fowler et al., 2000] is a slightly more sophisticated technique for restructuring a system, often used for object-oriented systems; example refactorings include moving methods between
Figure 1.2: Supporting independent world views in conjunction with the Abstract Factory design pattern example. In (a), the client attempts to instantiate AbstractButton directly. In (b), the rest of the system believes that the client uses the Abstract Factory to instantiate AbstractButton indirectly. These conflicting world views are combined and reconciled in (c). At a boundary surrounding Client, implicit context is used to intercept the attempt to directly instantiate AbstractButton, rerouting it to a particular instance of AbstractFactory. The net effect is to remove the dependence upon AbstractFactory from inside the client, placing this dependence externally. Now, the client can be reused in contexts that do not use the Abstract Factory pattern.
classes, adding a parameter to an overridden method, and collapsing a class hierarchy. One can avoid simple errors in refactoring through the use of tools.

Typically, ad hoc approaches suffer from three shortcomings: non-repeatability, code replication, and interface constraints. If the changes made to a module are not quite correct, reverting to an earlier version of the module is necessary. Many steps may have been needed to derive the new version from the old; these steps need to be partially repeated to derive a corrected new version. This can be onerous and error-prone when many steps exist between the old and new version. Tool support can mitigate this difficulty to an extent.

Code replication results from the need to maintain both the original version and modified version of a module. Consider what must occur if a problem is detected in one of these versions; it is often the case that identical or similar changes are needed in every copy. This requires one to locate all the copies and perform the same or similar steps on each to correct the problem. Such duplicated effort is tedious and error-prone [Burd and Munro, 1997].

Finally, one does not always have the luxury of having full control over every module that is dependent upon the ones requiring modification. Although the concept of encapsulation [Parnas, 1972] allows implementations of an interface to change, we are often constrained to maintain a published interface for the sake of clients—otherwise, those clients would all break. Refactoring can be used to change the structure of parts of a system without replicating those parts, but it cannot help us to overcome interface constraints. I will describe other difficulties with refactoring in Chapter 2.

1.3.2 Product-, Architecture-, and Interface-Centric Approaches

Product-, architecture-, and interface-centric approaches form a continuum that varies on the basis of the degree to which systems using them are constrained. At one end of this continuum are systems that possess a set of configurable options, allowing for a pre-defined family of products. At the other end are systems built from some pre-existing parts, where the interfaces to these parts constrain the way in which the rest of the system must be built.

Object-orientation provides for reuse and evolution through inheritance. Object-oriented frameworks [Johnson, 1997] supply a set of interacting classes, with the intent that some will be subclassed, specializing the framework to a given context. Middleware frameworks, such as CORBA® [Object Management Group, 1998], provide a standardized calling protocol and interface constraints to which new modules (i.e., components) are to adhere [Szyperski, 1997].

The trouble with all of these approaches are two-fold. First, no matter how standardized the interface provided by one of these approaches, it inevitably needs to evolve. The need to maintain backwards-compatibility to support existing client modules, and to update the interface to accommodate new ideas, new features result in pressure in opposite directions; predicting such changes ahead of time is rarely possible [Wegner, 1996]. As a
result, languages evolve, libraries evolve, and standards evolve—or else, they stagnate and are eventually abandoned.

Second, no single architecture is ideal, or even usable, for all systems. Expecting modules to always plug-in to any architecture like "LEGO® blocks" is unrealistic. Most modules place constraints upon their systems in the form of dependences on external modules. A given module may need support from its system for certain global properties, such as particular security or fault detection protocols; such support cannot be provided by an arbitrary framework [Ran, 1999].

All these approaches require a single, global perspective on the system, an unrealistic requirement for software change that is distributed over time and geographical space.

1.3.3 Generative Approaches

Generative approaches fall into a number of subcategories; Czarnecki and Eisenecker [2000] give a good survey. Implicit context is itself a generative approach; the most closely related generative approaches are generic programming and aspect-oriented programming.

Generic Programming

Generic programming [Musser, 1997; Musser and Nishanov, 1997; Jazayeri et al., 1998], most famously exemplified by the C++ Standard Template Library [Stepanov and Lee, 1995], is concerned with finding the most abstract representation of efficient algorithms. This goal is similar to that of essential structure, though more narrow in scope. Generic programming represents such algorithms in terms of generic parameters; concrete versions can be generated for concrete contexts by binding these generic parameters to concrete values or types.

However, generic parameters must be declared explicitly. This results in one of two requirements; either the most elaborated use of the generic module must be predicted, so that appropriate generic parameters may be provided, or the generic module must evolve. Even should such prediction be possible, a generic module will then tend to possess such a large number of generic parameters that it will be difficult to use in simpler, concrete situations. Evolving a generic module to add generic parameters would require that all instantiations of it be modified to bind values to these new parameters—often not possible.

Aspect-Oriented Programming

Aspect-oriented programming (AOP) [Kiczales et al., 1997] permits the separation and encapsulation of crosscutting concerns, concerns that affect the implementation of multiple modules within a system, such as synchronization, remote invocation, or fault tolerance.
By pulling out these concerns and placing them in their own, special modules (called aspects), the hope is that the core functionality of the modules will be less obscured and more reusable, and that the crosscutting functionality will be more evolvable, understandable, and reusable since it is localized. AOP is realized in a number of concrete programming languages, especially AspectJ™ [Kiczales et al., 2001], an aspect-oriented version of Java [Gosling et al., 2000].

More recently, the term "aspect-oriented programming" has come to be used for a number of approaches related to AOP as originally formulated [Elrad et al., 2001], including composition filters [Aksit et al., 1992] and subject-oriented programming (SOP) [Harrison and Ossher, 1993]. Composition filters provide a means for intercepting and rerouting messages at run-time. SOP allows separate class hierarchies to be composed; messages sent to an object from within one of these hierarchies can cause the invocation of behaviour within other hierarchies.

Some features of implicit context are similar to those of existing AOP techniques. However, its support for inconsistent, independent world views is unique. Such support permits reuse of modules without a need for pre-planning. It also allows modules to be oblivious to changes in the system around them, reducing the need to invasively modify many modules within a system undergoing change.

1.4 Organization of the Dissertation

I begin the meat of this dissertation in Chapter 2, considering a motivational example of the difficulties of software evolution and reuse. This example points the way to the concept of essential structure, discussed in Chapter 3, describing minimal specifications and extraneous embedded knowledge along the way.

In Chapter 4, I define the model of implicit context, and demonstrate how it can be used to realize essential structure. The model has been realized in a concrete, prototype tool for applying implicit context to Java source code. The tool is described in Chapter 5. It operates as a preprocessor to Java source code. The developer writes snippets of code, called boundary maps, that specify the details of the translation process for particular kinds of communications. The developer also specifies to which boundary (or boundaries) a given boundary map is to be applied; under the prototype, boundaries can be defined around only individual classes or individual methods. The original source code and boundary maps are fed into the prototype tool, which mechanically transforms the source accordingly.

In Chapter 6, I describe two case studies that I performed: one describing a case of unanticipated reuse of a feature from an industrially-developed system; and one describing a case of parallel development and evolution with and without implicit context. The case studies make use of the prototype tool. Taken together these two case studies validate the
thesis by providing evidence that essential structure through implicit context eases software evolution and software reuse.

Related work is described more thoroughly in Chapter 8. I discuss a number of outstanding issues and potential solutions in Chapter 7, especially in regards to making the technique practical. And I describe my contributions, future work, and conclusions in Chapter 9.
Chapter 2

Motivation

When we begin building a software system, we strive to design software modules\(^1\) that are simple and conceptually clean. When we finish building a version of the system, a different story has typically unfolded. An original vision of independent and cohesive modules that interact through a narrow interface is too often replaced with a reality in which there exists a larger than desired set of interactions between modules. Such additional interactions couple our modules, preventing them from being easily reused in new systems, and preventing our systems from being easily evolved.

Obviously, modules must communicate to provide system behaviour. Communication leads to interaction between modules. Sometimes we can point to interdependences between modules and say outright that these do not belong: the modularization of the system is poor, and as a result, our modules are strongly coupled or weakly cohesive. There are other times when we cannot make such a strong statement, when certain dependences are necessary. The problem resides in the fact that, at such times, a module ends up interacting with other modules for reasons not directly related to providing its core behaviour. At these times, it is not the mere existence of dependences that is negative, but their placement relative to the rest of the structure.

To investigate the sources of these troubles, I examine a motivational example in Section 2.1, where I attempt to reuse a feature, called the Outline View, that exists in an open-source integrated development environment, called Eclipse. The Outline View displays a tree-like view on the major declarations in a source file. This Outline View is highly dependent upon parts of the development environment that I am not interested in reusing. As evolution is an extreme form of reuse in which we are reusing most of one system in another (version), this example serves as a general motivation for the difficulties encountered in both software reuse and software evolution.

I discuss the two most common approaches to reuse in this chapter. First, I examine

\(^{1}\)I use the term *module* in a generic sense to mean any method, class, package, procedure, unit, component, etc.
the simple reuse technique of copy-and-modify in Section 2.2 and find it lacking for my purposes. Second, I consider the more sophisticated approach of refactoring in Section 2.3; while I find it too to be lacking, it brings us closer to the concept of essential structure described in Chapter 3. Less common approaches using special mechanisms are considered in Chapter 8.

2.1 Reusing the Outline View of Eclipse

Figure 2.1 shows a screenshot of an integrated development environment (IDE) called Eclipse. Eclipse is written in the Java programming language [Gosling et al., 2000], combining a number of standard programming tools such as a compiler, debugger, and text editor. It also supports an Outline View, shown on the right of the screenshot. The Outline View is an abstract, hierarchical view of the major programmatic declarations within the Java source file currently displayed in the text editor.

In the example screenshot, a simple class called Test is being edited, and the Outline View shows the outline of Test as a tree: the root of the tree represents Test itself, there are nodes for methods within Test (i.e., foo(), foo2(), and blah(int)), and so on. Each node in the outline tree has an icon representing a combination of its kind (i.e., class vs. field or method) and its modifiers for visibility, synchronization, etc. At the top of the Outline View window are four toggle buttons that can be used to modify the view of the programmatic declarations. When each toggle is depressed, respectively, the declarations are sorted lexically, fields are hidden (or “filtered”), static members are hidden, and non-public members are hidden.

The text editor and Outline View operate in unison: as the source file is modified, the Outline View is dynamically updated. When a declaration is selected in the Outline View, the text editor jumps to the corresponding location within the source text. For example, in the screenshot, the declaration for the inner class Inner within Test has been selected, and the text editor has moved to the corresponding location. Also available within the Outline View is a pop-up menu (not shown) that allows various tasks to be performed in a context-specific fashion on the selected declarations. For example, a set of declarations can be deleted from the source or various refactoring operations [Opdyke, 1992] can be performed, such as renaming a single declaration (all references to that declaration are altered correspondingly).

The Outline View thus possesses a non-trivial set of functionality. This functionality

---

2Eclipse was originally developed by Object Technology International, Inc.; it has since been released to the open-source community and may be downloaded from http://www.eclipse.org/

3Reiss [1996] provides a survey of software tools and development environments.
would be useful for a number of programming and analysis tools. For example, the Feature Extraction and Analysis Tool [Robillard and Murphy, 2002] makes use of a similar graphical user interface (GUI) to select sets of programmatic declarations and perform extraction and analysis tasks on these. The tool that is described in Chapter 5, for the application of implicit context, itself needs GUI support to make its use easier; the Outline View would be a good basis to build this support upon (see Figure 2.2).

Eclipse itself is large, consisting of approximately 800,000 lines of source code. Most of its functionality is not of interest to us in the context of these other tools. Were I to simply reuse the whole of Eclipse, the additional 800,000 lines of source code would be an unacceptable cost: there would be a cost in memory overhead, system maintenance, dynamic performance, and system comprehensibility. In short, our tools would become

4This was calculated using the UNIX wc tool, and thus includes blank lines and comment-only lines.
unnecessarily complex and thus fragile—even were we able to somehow hide the unwanted functionality, which is not a given. Therefore, I desire to “pull out” and reuse the Outline View, leaving behind the remainder of Eclipse.

Unfortunately, the Outline View was not designed to be reusable outside the context of Eclipse. To see the difficulties this causes us, let us consider the design of Eclipse. `JavaOutlinePage` is a class within Eclipse that forms a key part of the implementation of the Outline View. Along with its inner classes, an instance of `JavaOutlinePage` is responsible for maintaining the tree data structure for the outline of a particular source class, for displaying this tree, and for performing the various filtering operations available; multiple instances of `JavaOutlinePage` may exist at a time to permit quick switching between different source files in the Eclipse editor. Since `JavaOutlinePage` and its inner classes are so central to the implementation of the Outline View and I wish to reuse the Outline View, I need to reuse these classes. Figure 2.3 shows all of the classes upon which `JavaOutlinePage` and its inner classes (shown in boldface, and boxed) depend. These are simple dependences, consisting of explicitly named classes within the source text of `JavaOutlinePage`, any superclasses these possess, and any interfaces they implement; this diagram does not include less obvious, indirect dependences, should they exist. Nevertheless, even at this level, `JavaOutlinePage` possesses many dependences upon classes (e.g., `JavaPluginImages`, `JavaEditorMessages`) that I am not interested in reusing when I pull out the Outline View.

Eclipse is well-designed. It provides a type hierarchy with increasing abstraction towards its base. This permits clients to refer to the least concrete class that each needs; as a result, more concrete classes can be created to extend the system without the need to modify most clients. This is good object-oriented design [Booch, 1986; Rumbaugh et al., 1991;
Figure 2.3: The classes upon which JavaOutlinePage and its inner classes depend. Methods and fields are not shown. The inheritance relationships and aggregation relationships of these classes are shown. The other, individual dependency relationships are not shown explicitly. JavaOutlinePage and its inner classes are shown in boldface and boxed. The three superclasses that fall outside Eclipse (in the standard Java libraries) are shown in parentheses.
Zweben et al., 1995]. In many cases, Eclipse also provides a hierarchy of interfaces\(^5\) in parallel to its hierarchy of classes; this makes it harder for developers to ignore the conceptual coherence of their classes, since one cannot so easily overlook the abstract interface because of implementation details. Eclipse also provides well-integrated support for design patterns such as Observer [Gamma et al., 1994]: each class is designed taking into consideration what events it might provide that could be of interest to other parts of the system—and these events themselves are structured in a hierarchy of abstractions.

Though it is well-designed, Eclipse remains complex, with many interdependences between its classes. There are two traditional approaches to deal with such problematic dependences in reusing the Outline View:

1. Copy the classes implementing the Outline View, and heavily modify them either to create a generic, reusable widget, or to customize them on a case-by-case basis. I describe this in Section 2.2.

2. Refactor Eclipse to separate the problematic dependences. Section 2.3 looks at this approach.

2.2 Copy-and-Modify

A typical approach to source code reuse today is to copy the source code of interest and modify it to suit the new context in which it must exist [Krueger, 1992; Rosson and Carroll, 1996]. This approach tends to greatly increase the amount of source code: one extra copy for each new context. Each of these must then be maintained independently, which is troublesome since the bigger the system, the more work there is to debug it and modify it. Software reuse is intended to minimize duplication of effort, and copy-and-modify is a poor substitute for it [Krueger, 1992; Burd and Munro, 1997].

In a variant on this, one might be tempted to alter the reused code to structure it “correctly.” This assumes that this code was not structured correctly, in some sense, in the first place. There are two possibilities: either the code really was poorly structured initially, or what is desired is to create a version of the code explicitly supporting reuse, one that can then be tailored to specific situations. As I have argued above, Eclipse is well-designed, so the first possibility does not hold. The second possibility is equivalent to refactoring, and is examined in the following section.

\(^5\)In Java, an interface is a special class that defines abstract methods, but no method implementations. As a result, interfaces are not instantiable. However, multiple inheritance can be simulated through the use of interfaces, without as many concerns over ambiguities as to the recipient of an overridden method call.
2.3 Refactor

Krueger [1992: p. 133] identifies abstraction as "the essential feature in any reuse technique" and others agree that explicitly designing for reuse makes reuse easier [Johnson and Foote, 1988; Castano and De Antonellis, 1993; Parnas, 1994; Griss et al., 1995; Wasmund, 1995; Jacobson et al., 1997]. Therefore, the question is whether we can refactor [Opdyke, 1992] Eclipse to have the Outline View explicitly support reuse by improving its abstraction. A sufficient abstraction must permit the non-reusable, context-specific dependences within the Outline View to be separated from the parts I wish to reuse.

As an example, Section 2.3.1 describes the dependences arising on a particular method in the Outline View (specifically, in JavaOutlinePage) called makeContributions. This method is declared as an abstract method within Page, the superclass of JavaOutlinePage. The makeContributions method is called polymorphically by Eclipse upon the instance of Page that is about to be displayed in the Outline View. This call permits the Page an opportunity to add specialized functionality to the standard menu and status line of Eclipse, and to the standard tool bar of the Outline View. Such an interface only makes sense in a context where a standard menu, status line, and tool bar exist. Outside Eclipse, the standard menu and status line will not necessarily exist. Therefore, assuming that such a menu and status line exist tie the Outline View to Eclipse, preventing it from being reused in other contexts.

We can consider two variants of refactoring to attempt to eliminate the unwanted dependences present in this method: coarse-level refactoring, where the system decomposition is radically altered, and fine-level refactoring, where the basic class hierarchy is left as-is.

At the coarse-level, Eclipse is currently well-structured. Classes in Eclipse are conceptually coherent, with small sets of methods. Coarse refactoring, such as changing the class hierarchy, or moving methods between classes is unlikely both to maintain cohesion and to separate out the problematic dependences. Therefore, such coarse-level refactoring to reduce the dependences present in the Outline View would be at the cost of making other portions of Eclipse even harder to reuse or to evolve. Each new situation where we needed to reuse a different portion of Eclipse would require yet another convoluted refactoring. Thus, I abandon hope for the coarse-level refactoring approach.

At the fine-level, perhaps individual methods could be chopped up into smaller methods within the same class, separating out the unwanted dependences. Ideally, this would leave behind a highly-reusable core structure, while the context-dependent details could be manipulated in new contexts without the need to modify this core. I examine this approach further in Section 2.3.2, but find that the ideal is not met: the unwanted dependences cannot be so easily removed from makeContributions. As an alternative, the core structure could be placed in a reusable, abstract class while each set of context-dependent
details would be placed in a subclass of the abstract class. This alternative is discussed in Section 2.3.3. While it, too, is insufficient, it contains the seed of an approach we can use.

2.3.1 Dependences in makeContributions

Below is part of the source for a method on JavaOutlinePage called makeContributions. I begin by looking at its existing functionality, and identify its dependences upon the rest of the system.

This method is called by other parts of Eclipse when a JavaOutlinePage instance is made visible; it allows that instance the opportunity to add functionality to the standard menu bar, tool bar, and status line that are provided to all graphical views in Eclipse.

// Begin running example 2.1

public void makeContributions(IMenuManager menuManager, IToolBarManager toolBarManager, IStatusLineManager statusLineManager) {

The method begins by checking that an actual instance of IStatusLineManager has been passed and, if so, decorates it with an instance of StatusBarUpdater; StatusBarUpdater (potentially) writes messages on the status line when the user changes the set of declarations, on display in the Outline View, that are selected.

if (statusLineManager != null) {
    StatusBarUpdater updater =
        new StatusBarUpdater(statusLineManager);
    addSelectionChangedListener(updater);
}

Next, an instance of LexicalSortingAction (an inner class of JavaOutlinePage, and so, already customized for its purposes) is created and added to the tool bar; an action is essentially a callback function. The tool bar manager adds a new button to the tool bar according to the properties of the added action; when the user presses the button, the run method of the registered action is invoked. In the case of LexicalSortingAction, the action taken is to sort (or unsort) the declarations displayed in the Outline View.

6Throughout the thesis, there are lengthy examples in which explanatory text is interspersed with source text. These are introduced with a comment of // Begin running example and terminated with a comment of // End running example. The original source text of makeContributions is presented in Appendix C without the interspersed commentary.
Action action = new LexicalSortingAction();
toolBarManager.add(action);

Similar calls are made to add actions for the other three buttons on the tool bar of the Outline View. Shown below are those that create the button that hides or reveals (i.e., filters) field declarations; the calls for setting up the other buttons are similar and, thus, are not shown. FilterAction is also an inner class of JavaOutlinePage, but the actual kind of filter (e.g., for fields) and the images, descriptions, etc. appropriate thereto parameterize it, unlike in the case of LexicalSortingAction. As a result, some detailed parameters are exposed within this section of code.

    action =
        new FilterAction(new FieldFilter(),
            JavaEditorMessages.getString(
                "JavaOutlinePage.HideFields.label"),
            JavaEditorMessages.getString(
                "JavaOutlinePage.HideFields.description.checked"),
            JavaEditorMessages.getString(
                "JavaOutlinePage.HideFields.description.unchecked"),
            JavaEditorMessages.getString(
                "JavaOutlinePage.HideFields.tooltip.checked"),
            JavaEditorMessages.getString(
                "JavaOutlinePage.HideFields.tooltip.unchecked"),
            "HideFields.isChecked");

    JavaPluginImages.
        setImageDescriptors(action, "lcl16", "fields_co.gif");
toolBarManager.add(action);

    ...
• FilterAction, FieldFilter, and LexicalSortingAction exist, but these are inner classes within JavaOutlinePage, and so, do not trouble our task of reusing JavaOutlinePage.

• StatusBarUpdater can deal with events concerning a change in the items selected in the Outline View.

• StatusBarUpdater requires a parameter of type IStatusLineManager in its constructor, which apparently cannot be null.

• LexicalSortingAction and FilterAction are subclasses of Action.

• Actions are individually passed to IToolBarManager and not, say, buttons, and the add (Action) method is to be used for this purpose.

• FilterAction requires a particular, ordered set of parameters in its constructor, where the first is a (supertype of) FieldFilter and the rest are strings, each with a particular purpose.

• The strings appropriate as parameters to the constructor of FilterAction are returned by static calls on JavaEditorMessages, and these messages are obtainable by passing other, particular strings to it.

• The file location of the image to be associated with an action (which is drawn on its button) can be set through a cryptic-looking call on a class called JavaPluginImages.

Now, I examine how refactoring can or cannot help us eliminate the unwanted dependences in makeContributions.

### 2.3.2 Refactoring without Changing the Class Hierarchy

In reusing the Outline View, I choose a set of classes to pull out of Eclipse on the basis of their support for the functionality of interest. There will be no status line nor menu bar in my separated Outline View, so IMenuManager, IStatusLineManager, and StatusBarUpdater are not of interest. Likewise, I am not interested in classes supporting a Java editor and there will be no concept of a Java plugin external to Eclipse; thus, JavaEditorMessages and JavaPluginImages are not of interest either. To reuse the Outline View independent of the rest of Eclipse, I must eliminate mention of these classes from makeContributions.

First, I simplify the creation of the field filtering action by making an additional inner class within JavaOutlinePage, called FieldFilterAction, inheriting from FilterAction. Two additional inner classes are needed for the static filter action and public
visibility filter action (not shown in the code above). The replacement statement is similar to the statement instantiating LexicalSortingAction, and has the positive effect of making clearer the tasks performed by makeContributions. The downside of such a refactoring is that JavaOutlinePage as a whole still makes mention of JavaEditorMessages and JavaPluginImages. Also, it is not possible to move any inner classes outside of JavaOutlinePage without exposing private state information to the system at large—which would lead to structural degradation. But let us ignore this for the moment, and look at the next set of dependences.

Mention of StatusBarUpdater can be eliminated from makeContributions by replacing the lines mentioning it by a call to a hook, say

```java
if(statusLineManager != null) {
    contributeToStatusLine(statusLineManager);
}
```

and implementing contributeToStatusLine to make mention of StatusBarUpdater. But again, JavaOutlinePage as a whole still mentions StatusBarUpdater.

Dealing with IMenuManager and IStatusLineManager is more difficult, as the signature of makeContributions forms part of the published application programming interface (API) of Eclipse: I cannot simply change it on a whim, as client developers depend upon it. I can, however, reimplement it as follows:

```java
public void makeContributions(IMenuManager menuManager, IToolBarManager toolBarManager, IStatusLineManager statusLineManager) {
    contributeToStatusLine(statusLineManager);
    contributeToToolBar(toolBarManager);
}
```

where the implementation of contributeToToolBar ends up consisting solely of the construction and addition to the tool bar of the four actions provided by the Outline View, as described earlier.

```java
private void contributeToToolBar(IToolBarManager toolBarManager) {
    Action action = new LexicalSortingAction();
    toolBarManager.add(action);
    action = new FieldFilterAction();
    toolBarManager.add(action);
```
Now, contributeToToolBar is reusable in my new context, although large swathes of the remainder of JavaOutlinePage are not. I do not want merely to copy out the reusable portions of JavaOutlinePage, for the same reasons I rejected the copy-and-modify approach discussed in Section 2.2. Let us now consider how we might eliminate the dependences altogether from JavaOutlinePage without destroying the structure of Eclipse in the large.

2.3.3 Refactoring through an Abstract Class

As an alternative, we can modify the approach described above to split JavaOutlinePage into two classes: an abstract class containing the reusable elements, and a concrete subclass of this containing the Outline View dependences specific to Eclipse. I implement the abstract class much as I refactored JavaOutlinePage above.

```java
// Begin running example 2.2

public class AbstractJavaOutlinePage extends ... { 
    public void makeContributions(IMenuManager menuManager,
        IToolBarManager toolBarManager,
        IStatusLineManager statusLineManager) {
        contributeToStatusLine(statusLineManager);
        contributeToToolBar(toolBarManager);
    }

    protected void contributeToStatusLine(IStatusLineManager manager) {
        // subclasses to override this as needed
    }

    Also, rather than require the implementation of a new FieldFilterAction class, I invoke an abstract method createFieldFilterAction that is to be implemented by my concrete subclass.
```
protected void contributeToToolBar(IToolBarManager toolBarManager) {
    Action action = new LexicalSortingAction();
    toolBarManager.add(action);

    action = createFieldFilterAction();
    toolBarManager.add(action);

    ...
}

protected abstract Action createFieldFilterAction();

...

// End running example 2.2

I have thus created so-called hooks in my abstract class that concrete classes can then implement. The context-dependent details are implemented in a context-specific concrete class through the use of these hooks. Since this concrete class is intended to be specific to Eclipse and not reusable outside it, these details are no longer problematic from the perspective of reusing the Outline View.

public class EclipseJavaOutlinePage
    extends AbstractJavaOutlinePage {

    protected void contributeToStatusLine(IStatusLineManager manager) {
        if (manager != null) {
            StatusBarUpdater updater =
                new StatusBarUpdater(manager);  
            addSelectionChangedListener(updater);
        }
    }

    protected Action createFieldFilterAction() {
        Action action =
            new FilterAction(new FieldFilter(),
                JavaEditorMessages.getString("filterOptimizer_label");

    }

    public class FilterAction extends Action {
        private FieldFilter filter;

        public FilterAction() {
            super("filterOptimizer_label");
        }

        public void execute() {
            filter = new FieldFilter();
        }

    }

    private class FieldFilter extends Filter {
        private static final String label;

        public FieldFilter() {
            super(label);
        }

        public void setString(String value) {
            label = value;
        }

    }

    // End running example 2.2
Problems remain with this solution. AbstractJavaOutlinePage still explicitly mentions IMenuManager, IToolBarManager, and IStatusLineManager, even though in a given context each of these might not be reused. For example, the reused Outline View in Figure 2.2 does not possess a status line, and so, IStatusLineManager has no use there. However, since these are formal parameters required by the Eclipse API and the implementation of makeContributions is intended to override that of its superclass, mention of these classes cannot be removed by this approach.

More serious problems arise when one considers additional cases of reuse. It might be necessary to pull out other dependences, requiring a different abstract base class. This would necessitate one of two changes.

The first option requires the implementations of my AbstractJavaOutlinePage, EclipseJavaOutlinePage, and any other subclasses of AbstractJavaOutlinePage to change to increase the abstraction of AbstractJavaOutlinePage and transfer the concrete details to the subclasses. If we lack control over the source code in the original system, we would need to create and maintain a separate version of this source; this has the same drawbacks as copy-and-modify, described in Section 2.2. Furthermore, such changes are dangerous, potentially introducing errors every time we invasively modify the system.

The second option would introduce an additional abstract class as the superclass of AbstractJavaOutlinePage. This abstract superclass would have even fewer concrete details; most of the body of AbstractJavaOutlinePage would move to the
new abstract superclass, with only a few concrete implementation details remaining. This solution leads to structural degradation, as there come into being a hierarchy of abstract classes, most of which are little different from their superclasses; such a hierarchy would be difficult to maintain, as the conceptual cohesion of each would be weak—mere dribs and drabs of concrete details.

Many of the other dependences on page 20 remain after my attempts at refactoring. But there are two points to consider: not all of these dependences are problematic; and, not all knowledge can be eliminated if an algorithm fulfilling the role of makeContributions is to remain. There is the seed of an approach here. What I have approximated in refactoring makeContributions is its essence; I have stated in an abstract fashion the core of what this method is about, and allowed the details to be filled in externally. To reduce makeContributions much further would be to render it devoid of content; there would be no value in reusing it then. Thus, there is a hope that there exists a minimal, essential structure to makeContributions, a structure that possesses some quality of context-independence and, therefore, high reusability.

\section*{2.4 Summary}

Software reuse and software evolution are problematic because of the presence of dependences between the modules of our systems. I have examined a particular example, the reuse of the Outline View from the Eclipse integrated development environment, to investigate the nature of these dependences.

I have attempted two simple, traditional approaches to removing these dependences: copy-and-modify and refactoring. Copy-and-modify is found lacking due to the resulting need to maintain two parallel but closely related versions of the same code base, which would significantly reduce the benefits of reuse. Refactoring also possesses general drawbacks. It requires that interfaces be modified upon which client code depends; this is generally not possible since we rarely have control over such client code. It requires that all combinations of concrete details within our classes be separated into a complex assortment of abstract superclasses; such structural degradation is the anathema of software evolution and comprehensibility. Although more sophisticated techniques exist (and are examined in Chapter 8), none provides the sufficient means to cope with excess contextual dependences.

Refactoring has motivated the idea of essential structure: the minimal implementation, maximally abstract, that actually provides the functionality that forms the core of a given module. I examine this concept in more detail in the following chapter.
Chapter 3

Essential Structure

Software reuse and software evolution are made more difficult when our modules have assumptions built into them about the context in which they operate. In Chapter 2, we have seen how JavaOutlinePage expresses knowledge of the context in which it exists, namely Eclipse, by explicitly referring to the JavaEditorMessages and JavaPluginImages classes. The latter classes have no meaning outside of the particular context of Eclipse, which makes JavaOutlinePage difficult to reuse.

Our refactored version of the makeContributions method in JavaOutlinePage attempted to do away with these contextual dependences by abstracting out some of these details. But arbitrary abstraction is insufficient. For example, when a class participates in the Abstract Factory design pattern [Gamma et al., 1994] as a client, it must be aware of this participation; the abstract factory class must be explicitly named even though only the product classes managed by the factory are of interest to the client. As a result, the client class cannot be reused in a context where Abstract Factory is inappropriate: even in an attempt at flexibility, such built-in assumptions make software brittle.

This is not to say that there is no form of abstraction that suffices. To reuse a given module, it must represent some form of functionality that is of interest to us, or there is no point in reusing it; such functionality lies at the core of the purpose of that module. Abstracting away whatever this particular functionality is would remove the core of the purpose of the module—an operation that makes no sense.

This core may require some dependences upon the system outside the module. However, any details of the external context beyond this core functionality are constraints that will impede the reusability of the module. An overconstrained module is more likely to require change as the system evolves. These extra constraints are knowledge of the external context of a module that is extraneous from the perspective of the core purpose; I thus refer to such knowledge as extraneous embedded knowledge (EEK). Abstracting the implementation down to its inner core, which I term the essential structure, will maximize the number of contexts in which a module can be used, and will eliminate the presence of EEK within
it, thereby making that module less likely to change when the system around it changes.

More precisely, knowledge expressed within a given module is EEK relative to that module if that knowledge is not explicitly described by a minimal specification of the module's responsibilities. A module that contains no EEK expresses only its essential structure.

In Section 3.1, I discuss and define the concept of minimal specifications and, in Section 3.2, examine their use in defining EEK and identifying it in the makeContributions method example of Chapter 2. I then consider a useful categorization of some forms of EEK in Section 3.2.1. The general way in which each might be eliminated to realize essential structure is described in Section 3.3.

### 3.1 Minimal Specifications

A given module has a set of goals it must achieve. When a module must be implemented to meet a minimum set of constraints, those constraints are termed the requirements of the module. Requirements represent the set of constraints that are necessary for an implementation to meet. The process of implementation is one of elaboration of the requirements, to such an extent that a typical reaction to the fact that an implementation of a module contains many more details than its requirements is "Of course!" Thus, it is generally not the case that it is sufficient for the implementation of a module to only meet the constraints imposed by its requirements.

To judge whether a module has met its goals, we need some idea of what those goals are, some description of them. Such a description may be implicit, it may be explicit but informal, or it may be explicit and formal—regardless, there is a description possible. A specification can be considered such a description, since the specification of a module "should tell the user everything he must know to use the [module], and it should tell the implementer everything he must know about the [module] to implement it" [Lamport, 1989]. A requirements specification is a specification that is limited to a description of the necessary constraints on a module; these may or may not be sufficient. A specification in general can contain more details than arise from the requirements specification.

The more constraints there are upon a module, the fewer implementations that can be created to meet those constraints; the more constraints that a module places upon its system, the fewer systems in which the module can be used. We wish to maximize the number of contexts in which a module can potentially be reused; then, if the system in which the module occurs needs to change, this minimizes the likelihood that the module itself will need to evolve. Therefore, we must reduce the constraints upon our modules to those that are both necessary and sufficient. This is not a radical departure from current thought; the departure lies in the assumptions that are made about what is necessary and what is sufficient. I wish to describe the goals of our modules in as abstract fashion as
possible, and to implement our modules to express this description without the addition of
details. Then, we will have implementations that possess only the necessary and sufficient
constraints upon the behaviour of modules and the behaviour of the system containing these
modules. Let us look at an example.

An Example of the Problem

Consider one possible, informal specification of the makeContributions method of
JavaOutlinePage, judging by its implementation. It might read something like the
following.

- makeContributions takes three parameters, of types IMenuManager,
  IToolBarManager, and IStatusLineManager, and has a void result type.

- makeContributions checks whether the argument passed in the IStatus-
  LineManager parameter is null; if and only if the parameter is not null, it
  creates a new instance of StatusBarUpdater and adds this instance to Java-
  OutlinePage’s set of selection changed listeners.

- makeContributions creates a new instance of LexicalSortingAction
  and adds this to the argument passed in the IToolBarManager parameter.

The rest of the specification would continue in this fashion.

This specification is very detailed and concrete—to such an extent that many would
object to its use as a “specification” rather than an implementation. Even so, it makes no
mention that the IStatusLineManager must be passed to the constructor called on
StatusBarUpdater, nor that addSelectionChangedListener must be used to
add the new instance of StatusBarUpdater to JavaOutlinePage’s set of selection
changed listeners, and so on. The claim could be made that these details that have appeared
in the implementation were necessary to meet the specification, or even that there was no
other way to implement them. Such claims could be perfectly true—under the constraints
of the current system design and current programming mechanisms.

But note that these details might well not be valid in this implementation should the
system around makeContributions change. While the same specification might hold
for an implementation of makeContributions in a different system, or different ver-
ion of the same system, these details would prevent this particular implementation from
being usable in all such situations.

At first glance, the constraining nature of this detailed specification might seem a good
thing. After all, it tells us how to implement makeContributions in a fairly precise
way. One might then be tempted to over-specify makeContributions to ensure that all
its details are necessary and sufficient. However, Wing [1990: p. 11] states:
In practice, you must usually deal with incomplete specifications. Why? Specifiers may intentionally leave some things unspecified, giving the implementer some freedom to choose among different data structures and algorithms. Also, specifiers cannot realistically anticipate all possible scenarios in which a system will be run and thus, perhaps unwittingly, have left some things unspecified. Finally, specifiers develop specifications gradually and iteratively, perhaps in response to changing customer requirements, and hence work with unfinished products more often than finished ones.

We can observe three interdependent principles at work here.

1. Current programming techniques require details to be added to the implementation of a module, beyond those that are specified.
2. Modules tend to be under-specified to allow their implementations some flexibility.
3. Specifications tend to miss details because specifiers cannot predict all reuse or evolution scenarios in which a module must operate.

Therefore, feeling the need to create exacting specifications is misguided.

At some level, the concrete details that appear in the implementation of makeContributions for this particular version of Eclipse must be specified and implemented; after all, makeContributions needs to operate in that particular context. However, there is an inner core, an essential structure, that makes an implementation of makeContributions bona fide. If we could directly implement this essential structure, in a manner similar to the refactoring of Section 2.3.2, and layer on the concrete details, ideally the inner core would be highly reusable and the outer layers more easily changed.

Defining the Inner Core

To be able to define such an essential structure, then, requires us to describe the details that are necessary and sufficient for an implementation to possess. I call such a description a minimal specification; it is intrinsic to a module, whether or not it is made explicit.

Consider again the case of makeContributions. At its core, it must add some functionality to allow the status line to reflect selection changes in the Outline View, and to add four actions to the tool bar, one for sorting declarations lexically, one for hiding fields, etc. Thus, a minimal specification for makeContributions is:

- makeContributions must add functionality to the status line, if and only if there is one, to reflect selection changes in the Outline View.

Since the given specification is informal, there is room for ambiguity in its interpretation; however, such worries are secondary in defining the necessary and sufficient constraints.
• It must add four actions to the tool bar, one for sorting declarations lexically, one for hiding fields, etc.\(^2\)

The difference between this and the earlier specification lies in the degree of abstraction. In the latter specification, there are no references to concrete names, or specific protocols or algorithms, but merely to the concepts that are necessary to realize. It is a specification of sufficient detail since it gives all the abstract details that are significant for makeContributions. Attempting to reduce the specification beyond this point would be to alter the nature of makeContributions, and therefore, it is a specification of necessary detail. This latter specification is therefore minimal.

Definition 1 (Minimal specification) A minimal specification is a specification in which every detail present is necessary and sufficient to meet the goals of its module, in as abstract a fashion as possible.

This is not to say that there is no useful module that could be defined with a further reduced specification. In fact, we have seen in Section 2.3 how separating makeContributions into contributeToStatusLine and contributeToToolBar is useful. The minimal specification for contributeToStatusLine would be provided by reducing the minimal specification for makeContributions to the first item present there:

• contributeToStatusLine must add functionality to the status line, if there is one, to reflect selection changes in the Outline View.

Similarly, the minimal specification for contributeToToolBar would be provided by reducing the minimal specification for makeContributions to the second item present there. But neither of these single items by themselves is sufficient to specify makeContributions.

Now, I consider how implementations are overconstrained because they possess knowledge in addition to that described in their minimal specifications.

### 3.2 Extraneous Embedded Knowledge

In Chapter 2, we saw how dependences upon the external context coupled the parts of Eclipse that I wished to reuse, to the parts that I did not. The concept of a minimal specification allows us to identify those dependences that should not be present within a given module. Once these have been identified, we can hope to do something about them.

Consider the differences between the implementation of makeContributions (given in Section 2.3.1) and the minimal specification of makeContributions (given in

\(^2\)Note that there is an implicit assumption here that the tool bar exists.
Section 3.1). First, the minimal specification says nothing about a return value from makeContributions, so a result type of `void` seems appropriate. The minimal specification says nothing about menus, but does mention a tool bar and a status line; however, it does not explicitly mention the classes `IMenuManager`, `IToolBarManager`, nor `IStatusLineManager`. The minimal specification does not even indicate that the tool bar and status line are passed as arguments.

```java
// Begin running example 3.1

public void makeContributions(IMenuManager menuManager,
                               IToolBarManager toolBarManager,
                               IStatusLineManager statusLineManager) {

    The minimal specification indicates that the status line might not exist, but if it does, that
    functionality must be added to it to reflect selection changes in the Outline View. However,
    the minimal specification does not state that a `null` value indicates that the status line does
    not exist, nor does it state that creating an instance of `StatusBarUpdater` and adding
    it as a selection changed listener are necessary in exactly this form.

    if(statusLineManager != null) {
        StatusBarUpdater updater =
            new StatusBarUpdater(statusLineManager);
        addSelectionChangedListener(updater);
    }

    And finally, while the minimal specification mentions the addition of four actions to the
    tool bar, it makes no mention of the concrete, detailed interfaces and protocols for doing so,
    such as the classes `LexicalSortingAction`, `FilterAction`, `JavaEditorMessages`, or `JavaPluginImages`, or any of the methods thereon.

    Action action = new LexicalSortingAction();
    toolBarManager.add(action);

    action =
        new FilterAction(new FieldFilter(),
                        JavaEditorMessages.getString(
                            "JavaOutlinePage.HideFields.label"),
                        JavaEditorMessages.getString(
                            "JavaOutlinePage.HideFields.description.checked"),
                        JavaEditorMessages.getString(
                            "JavaOutlinePage.HideFields.description.unchecked"),
                        32
As pointed out by Wing (see page 30), specifications are often incomplete, with details needing to be added in the implementation. The fact that such extra details appear in the implementation of makeContributions should come as little surprise.

However, let us consider an ideal situation. Imagine that we develop a new system, decomposing it into various modules, for each of which we provide a minimal specification. Assuming that such minimal specifications are expressed in (or can be translated to) some canonical form, we seek out extant modules matching all or part of the minimal specifications we need. But here is the catch: even if we manage to locate such extant modules, there is no guarantee whatsoever that they will be reusable in our new system. They will likely possess extra dependences upon the larger context of the system; since these dependences are not present in the minimal specification, they are not likely to be satisfied in a different system. I thus consider such dependences, and the knowledge they imply, to be extraneous. This is not to suggest that it is not important knowledge within a larger piece of the system, or the system as a whole; merely, this knowledge does not belong in the local scope of the given module.

**Definition 2 (Extraneous embedded knowledge)** Any knowledge expressed by a module of its external context that does not explicitly appear in its minimal specification is extraneous embedded knowledge (EEK) with respect to that module.

Recall the example of the Abstract Factory design pattern from Chapter 1; EEK tends to be present when a class participates in this design pattern as a Client. The minimal specification of the class that acts as Client is itself unlikely to refer to the Abstract Factory design pattern; some larger module that contains this class is the likely source for the requirement that Abstract Factory be used. Therefore, knowledge of Abstract Factory is EEK with respect to Client. This EEK manifests itself through the explicit naming...
and use of the AbstractFactory class when the Client needs to instantiate an AbstractProduct class. Client would be just as effective at providing its core purpose were it possible to express it as directly instantiating AbstractProduct.

A more subtle version of EEK can be found in large-scale modules containing smaller-scale modules. Consider drawing toy robots represented by a Robot class. Robot possesses a draw method that delegates to a series of other methods for drawing the different parts of the robot: the torso, head, left arm, and so forth. Each of these methods, in turn, delegates to yet other methods for drawing smaller bits of the robot, primitive objects (such as rectangles), or both. In implementing these methods, we realize that we need a canvas on which to draw primitives. Because the robot may need to be drawn on different canvases, a canvas object must be passed to the Robot class; therefore, a parameter is added to the draw method to accept such an object. Since we need to have this canvas at the points where it is required to draw primitives, a parameter is added to all the methods that draw parts of the robot and each method dutifully passes the canvas on. Only the methods that draw primitives actually do anything with the canvas; the others simply pass it to the methods to which each delegates. Knowledge of the canvas can thus be seen to be EEK from the perspective of these methods: a minimal specification of each of these methods could be created that ignored the existence of any canvas object. The canvas object is only present within these methods because a canvas object needs to be explicitly used to draw primitives, and this object must travel from the clients of Robot, through the hierarchy of delegations, down to the primitive drawing calls being made.

To re-emphasize, categorization of knowledge as EEK can only occur relative to a particular module. For example, the fact that knowledge of the Abstract Factory design pattern is EEK within Client does not imply that it is EEK within a larger, parent module containing Client. This parent may indeed be very concerned with the fact that the Abstract Factory design pattern is being used in favour of some other means of flexibility. Likewise, primitive drawing methods need to refer to a canvas for their operation to be meaningful. Therefore, a canvas object is not likely to be EEK within such a primitive drawing method even if canvas objects are not mentioned in the minimal specifications of clients of these primitives.

3.2.1 Varieties of EEK

Extraneous embedded knowledge can be characterized on the basis of the kind of dependence it creates. Such a characterization is not intended as a partitioning of the space of all EEK, but rather as a convenient nomenclature, and as a means of insight into whether all EEK is equally problematic. Some of the categories into which EEK can be seen to fall include: extraneous identification, where specific types or methods are explicitly named; extraneous parameters, where particular ordered sequences of types are required of the ar-
guments for a given method to be called; extraneous pre-condition assumptions, where the caller must adhere to certain constraints prior to calling a method; extraneous post-condition assumptions, where a called method is assumed to return a value meeting some, possibly implicit, semantics; and extraneous protocol adherence, where sequences of calls must be made in particular orders, or where certain semantic properties are otherwise assumed to hold sway. I examine each of these categories in turn.

**Extraneous Identification**

The simplest form of EEK consists of the dependences a client module forms on particular names and signatures of external modules. To specify a module, we often resort to using external modules to provide partial functionality. Generally, it is the conceptual functionality of these external modules we are interested in, and not the specific interfaces provided by them. For example, a method may require some library method for determining the current time, without caring what the exact name of this method is, or which library it is in, or even the multitude of possible parameters it accepts (Figure 3.1). But with current mechanisms, it is difficult to avoid such knowledge. If any of the external names or signatures change, our client module will break. What should be important to the client is not which module will be providing desired external functionality, but rather what functionality is needed. Therefore, I refer to false dependence on names or signatures as extraneous identification.

For example, say we had a client that referred to the Vector class of the Java standard class libraries, but at some point Vector is renamed DynamicArray, thereby breaking our client. Our client cared about the Vector functionality, not the name; therefore, the specific name was EEK within the client.

Similarly, in makeContributions, our minimal specification mentions concepts such as status line and tool bar but our implementation mentions IStatusBarManager and IToolBarManager. The concepts would remain the same even should the names of the concrete classes be otherwise.

**Extraneous Parameters**

An implementation of a module can possess parameters that its minimal specification does not mention. For example, consider three methods m1, m2 and m3. Method m1 calls m2 and m2 calls m3. Method m3 requires some particular information param that is known by or needs to be created by m1, but m2 does not need to know about param for its internal computations. Nevertheless, for param to pass from m1 to m3, it is typically passed via m2 (Figure 3.2). In most such situations, this parameter is not present due to the minimal specification of m2; therefore, param is an extraneous parameter within the signature of m2. The argument passed to m2 from m1 may or may not be EEK there, depending on the
Figure 3.1: The caller method calls the callee method to utilize its service. This implies that (a) caller depends upon the behaviour of callee (represented by the simple, dashed arrow); however, (b) caller must access this behaviour by making explicit reference to the name “callee,” whether or not this name represents the behaviour of true interest, and thus this dependence is a case of extraneous identification. Ideally, (c) caller and callee each realizes only its minimal specification (represented by the dashed, rounded boxes); the minimal specification of caller depends upon the minimal specification of callee but not upon the abstract label “callee.” Since the only dependence upon callee from caller is indirect via the minimal specification in the ideal situation, the concrete entity realizing this minimal specification may be replaced or renamed.

minimal specification of m1: if it recognizes that the service it desires of m2 requires this argument, then it is not EEK; otherwise, it is.

We have seen the presence of an extraneous parameter in the implementation of makeContributions on JavaOutlinePage. There, the minimal specification made no mention of any menu bar concept (let alone, the actual IMenuManager type). Such a formal parameter appeared for the sake of parallel structure: implementations of makeContributions on other classes may need to use the menu bar in some way; in order to utilize polymorphic dispatch, every sibling class of JavaOutlinePage must implement makeContributions with an identical interface, including the formal parameter of type IMenuManager. In this case, the argument passed in the formal parameter may well end up going nowhere at all, but is passed “just in case” it is needed.

Extraneous parameters are burdensome for evolution. It often comes to pass [Lewis et al., 2000; Walker and Murphy, 2000] that, during the elaboration of a system, we require some additional data to be passed to a method deep in the control flow. When this data is only obtainable from much higher in the control flow, all the intervening methods must have an extraneous parameter added to them. Should the module requiring the extra data
Figure 3.2: An extraneous parameter arises due to the mismatch between minimal specifications and implementations. The minimal specification $m_1$ depends upon the minimal specification $m_2$ that, in turn, depends upon the minimal specification $m_3^*$. While the modules $m_1$ and $m_2$ realize $m_1$ and $m_2$ respectively, $m_3$ actually realizes the minimal specification $m_3$, which is more detailed than $m_3^*$ (though it must be similar enough to substitute). This extra information causes the introduction of an extraneous parameter (param) into $m_1$ and $m_2$.

be replaced with one that does not, this extraneous parameter remains in all the intervening methods even though it serves no purpose. Add the constraints of polymorphism across multiple classes, and the situation gets even more tangled.

Extraneous Pre- and Post-condition Assumptions

Consider a method client that calls another method, service, to perform some portion of its functionality. The minimal specification of client has a notion of the service provided by service, and therefore, what information should be passed to service that the service should act upon. However, the minimal specification of service may well have a different notion of what information is required.

For example, service may require a random number to complete its functionality. The minimal specification of service requires that its clients generate this random number and pass it in. Thus, for client to be able to call service, the pre-condition that client compute and pass a parameter, which is of significance only to service, must be satisfied. If the minimal specification of client fails to recognize the need for this random number, this pre-condition is extraneous with respect to client. The implementor
of client likely saw the presence of service, thought that the pre-condition was not very onerous, and proceeded to use it accordingly. This is a problem: client will always be constrained to use a service requiring this pre-condition—even though it did not require it itself.

Similarly, we can consider the case where the return value of service does not directly satisfy the needs of client. Instead, a post-condition on service must be explicitly recognized by client: the implementation of client must include additional computations to convert the return value.

The expression of pre- and post-conditions within client is structurally unsound. The definition of such conditions depends on the implementation of and interface to service, instead of the interface to the abstract, external service that is described by the minimal specification of client. The introduction of dependences is similar to the situation with extraneous parameters (see Figure 3.2) save that an additional computation is added as a result of the dependence, rather than an additional parameter.

In the case of the Outline View of Eclipse, it could be argued that the supposedly minimal specification of makeContributions (given on page 30) really could be reduced further while remaining a specification of makeContributions. Namely, it could become:

- makeContributions must add functionality to the status line to reflect selection changes in the Outline View.
- It must add four actions to the tool bar, one for sorting declarations lexically, one for hiding fields, etc.

where there is no mention that the status line might not exist. In such a case, the presence within the implementation of the test to determine whether the status line is null is not only EEK from the point of view that it states too concretely what it means for the status line not to exist, but also because it is performing a test that it should not know about.

**Extraneous Protocol Adherence**

The information contained in a client module regarding a service module that it calls is not necessarily limited to the syntactic interface to the service. The implementor of the client had access to knowledge regarding the semantics of each parameter and return value of every method of the service (from the interface definition of the service and its accompanying documentation) at the time the client was implemented. Since such detailed semantics are

---

3In reality, this is a minimal specification for a subtly different module than the one discussed earlier, rather than an alternative minimal specification.
unlikely to have been present within the minimal specification of the client module, their presence would be EEK within the implementation of the client.

For example, Java version 1.1 contained no Set class or interface in its standard class library\textsuperscript{4}; to emulate such a class, Hashtable may have been used instead. If our client module were using Hashtables to provide the functionality of Sets, the implementor of the client module may have assumed other properties of particular Hashtables such as lookup efficiencies; or, the Hashtable could be required as the type of an input parameter, spreading the need for the use of the Hashtable to other modules. However, the minimal specification of the client only required the functionality of an abstract Set; the other semantic properties of Hashtables formed extraneous constraints upon the implementation of our client.

In the implementation of the makeContributions method of JavaOutlinePage, we have similarly seen cases of extraneous protocol adherence. The clearest case involved the creation of the field filter action: the protocol required that makeContributions possessed knowledge: of the key strings to look up message bodies; that the getString method of JavaEditorMessages be used to locate the actual labels, descriptions, etc.; and that the setImageDescriptors method of the JavaPluginImages class be used to locate the images. The minimal specification requires only that makeContributions create a field filter action; this specific protocol ties it tightly to Eclipse, and furthermore to a particular version of Eclipse.

Having examined some varieties of extraneous embedded knowledge, I consider how minimal specifications might more closely be realized through the concept of essential structure.

3.3 Reduction to Essential Structure: An Abstract View

With current programming techniques, we implement each of our modules both to realize its minimal specification and to satisfy the minimal specification of any other modules that we decide to call or otherwise reference within that implementation (consider Figure 3.2). Consider the effect of adding details of the context of operation into our implementation of a module when those details are not present within the minimal specification for the module (i.e., the effect of adding EEK). Such EEK couples the implementation of our module to one particular context, making it difficult to change that context without changing that implementation. What we need is a means of implementing a module in such a way that it is not tied to a single context. We can implement our modules to be maximally reusable and

\textsuperscript{4}That the Set interface was added to version 1.2 of Java is indicative of the fact that even standard libraries evolve, so making assumptions about their behaviours and interfaces leads to evolutionary problems.
maximally unlikely to need to change if their system evolves, if they contain only essential structure.\textsuperscript{5}

**Definition 3 (Essential structure)** An implementation of a module that expresses no extraneous embedded knowledge possesses only essential structure.

The question arises of how the implementation of a module can be reduced to its essential structure. Clearly, some details can be removed through refactoring, as we saw in Section 2.3. But some EEK would not seem to be amenable to such simple solutions: a given implementation will always explicitly mention *some* names and make *some* assumptions about how its external context is communicating with it. Such assumptions are EEK if not described in a minimal specification of the module.

However, all is not lost. There are three key ideas that can still allow us to reduce our modules to their essential structure. The first idea is to do away with the assumption that there is a one-to-one correspondence between a name that appears in an implementation and the actual name of an external module that gets accessed via that name. If we accept that identifiers within an implementation are simply abstract labels for external modules that meet some minimal specifications, the dependence that those labels actually be the names of those modules disappears.

Permitting late binding of our identifiers and the like allows great flexibility: we can express only the essential structure of a given module, nothing more, nothing less. The essential structure of the entire system includes its concrete details, and so it must express them. But below this level, only the details that are consistent with the essential structure of our modules should be expressed: we should only express identifiers in ways that are meaningful within the implementations of our modules themselves; we should only declare parameters and pass arguments that are meaningful within those implementations; we should express only whatever pre- and post-conditions and protocols that are consistent with the essential structure of our modules.

The second idea is similar: rather than assuming that the concrete communication protocols expressed by our implementation must be exactly those that actually exist, let us consider our implementation to possess an abstract description of the communication with the external context. This abstract description is expressed in a manner that is convenient to the expression of the module without consideration of the larger, concrete system in which it resides. For example, a common communication style consists of method calls

\textsuperscript{5}Some might be tempted to think of an implementation that expresses only essential structure as being some sort of "pseudocode implementation." Unfortunately, the notion of what constitutes pseudocode varies too greatly to aid us here. While some uses of pseudocode parallel the goals of essential structure, other pseudocodes are effectively formal languages for which one could write a compiler (e.g., those of Lekkos and Peters [1978] or of Robillard [1986]). Programs written in the latter form consist of pseudocode only in the sense that the given language does not exist in reality.
that pass arguments. The presence of a method call within an implementation is likely to be a straightforward, abstract description of communication with that external method. However, the communication that appears within a module to be a method call might actually be reified as a remote procedure call.

Thus, rather than insisting that we explicitly express only the essential structure of our modules, retroactive abstraction permits us to utilize “emergent” essential structure. By reifying our retroactively abstract implementations in the right way, we can obtain the same effect as if the module really had been implemented to express only its essential structure.

The third idea deals with the fact that the EEK that we avoid or remove from our modules almost certainly has to be expressed somewhere in our systems; it is important that this knowledge be expressed in the appropriate place, where it is not extraneous. To permit this external expression of details, we need to compose these details into our modules to form higher-level modules.

What we need is a model that allows us to express the essential structure of our modules, either directly or in an emergent fashion. This model must also allow those modules to be composed in such a way that the essential structure of ever larger-scale modules can still be expressed. I consider such a model in Chapter 4.

3.4 Summary

Every module in our systems, regardless of scale, must be specified, if only implicitly. The more detailed and concrete a specification, the fewer contexts in which the specification will be applicable; hence, an implementation of a detailed and concrete specification will tend to be less reusable and more likely to need to evolve. Therefore, we should strive to specify only the details that are both necessary and sufficient for our modules to meet and do so as abstractly as possible; that is, we should give their minimal specifications.

Extraneous embedded knowledge (EEK) arises in our implementations when we express knowledge of the context external to a module that is not explicitly given in its minimal specification. EEK is problematic because it places dependences in our modules upon specific contexts. When those dependences are not enforced by a minimal specification, we have constrained our module so that it can only operate in a subset of the contexts where the minimal specification should otherwise satisfy the requirements. Such EEK includes:

extraneous identification extraneous assumptions about the names and interfaces of external modules;

extraneous parameters parameters that are not used at all, or that are not needed from a local perspective;
extraneous pre- and post-condition assumptions where our modules perform (or avoid) computations to meet pre- or post-conditions on the use of external interfaces because of extraneous assumptions about those interfaces; and

extraneous protocol adherence where our modules perform specific sequences of calls because of extraneous assumptions about the required protocols for external interfaces.

By eliminating such extraneous dependences, we can maximize the number of contexts in which a given module is reusable, and decrease the likelihood that a given module will need to evolve. But to eliminate all EEK requires that we treat the identifiers, method calls, etc. that appear in an implementation as abstract even when they may appear concrete. Considering these constructs to be abstract retroactively allows us to delay binding concrete programmatic elements to them as late as necessary prior to running our systems; in this fashion, we need not necessarily explicitly express only the essential structure of our modules, but such an essential structure can arise in an emergent fashion. To realize this level of abstraction requires a model for the expression of essential structure that allows us to compose our modules to express the essential structure of higher- and higher-level modules. I consider such a model, implicit context, in the following chapter.
Chapter 4

A Model for Essential Structure through Implicit Context

In previous chapters, we have seen how the concept of essential structure arises, in order to improve software reuse and to ease software evolution. But to benefit from the concept, we need a model of how essential structure may be realized. We have identified two points that such a model must meet: retroactive abstraction, where the apparently concrete identifiers and programmatic constructs within a module can be reinterpreted after the fact as abstract concepts to be reified; and composability of our abstract modules in such a way as to form a concrete system.

When a module expresses only its essential structure, it becomes as context-independent as possible. However, this is not to say that the context in which a module operates is irrelevant; quite the opposite is true in fact. Therefore, we must provide a means for a module to interact with its context while it does not explicitly make mention of that context (except as required by its essential structure). I call this means implicit context. Through implicit context, we provide retroactive abstraction within our modules in such a way that we may compose our modules to form fully concrete systems.

Implicit context utilizes two basic concepts. First, each module may have an independent world view of the context in which it operates. With today’s common practice, we expect all our modules to agree upon the global view of the system: certain classes exist with certain interfaces; each module may only have a partial view of the world, but this view agrees with all the others. This level of abstraction does not suffice. It is difficult enough to define a single, consistent global view for a particular version of a system; defining a single global view that will hold sway for all time as a system evolves is next to impossible. Thus, the process of reuse and evolution would potentially be made simpler were we freed from the constraints of conforming to a complex, volatile world view. Second, reconciliation of the differences between those world views must be performed when the modules
interact in order to form a coherent system. Reconciliation is performed by interpreting communications differently depending on the context in which they take place.

In Section 4.1, I examine the model for implicit context. I then demonstrate how implicit context can be used to express the essential structure of modules in Section 4.2. I postpone a description of implementing implicit context until Chapter 5.

4.1 Implicit Context

Before describing the model for implicit context, I give an analogy with human conversations as a motivational framework. Consider that in human conversation we do not spell out every concept we wish to communicate at every instant the understanding of those concepts is required. We expect much information to be understood from or altered by context. Such use of context takes two forms: omission, where words or details are left out to be filled-in from earlier details within a conversation or general knowledge, and alteration, where the words that are spoken or the way that they are interpreted depends upon the individuals who are speaking. “It spun wildly” could refer to a ride at the county fair, or to one’s impression of a room while experiencing extreme nausea; the details about “it” have been omitted, to be understood from what has previously been discussed. Likewise, one’s response to the question, “What is politics?” might be quite different depending on who is asking; the explanation given a young child is likely to be significantly different from that given an adult. Meaningful use of context can require that the participants in a conversation share a common world view when referencing knowledge outside the confines of the conversation. Stating that someone “acted the role of Cyrano” would be meaningless if the listener knew nothing of Cyrano de Bergerac.¹

Similarly, the history of communications within a system can be viewed as a conversation between modules, and so, we can leverage the concepts of omission, alteration, and world view. Rather than forcing modules to repeatedly give the same details in their communications, I wish to allow them to communicate while omitting details—details that would be extraneous embedded knowledge within those modules. At the same time, we must perform alteration of communications according to the implicit context in which they occur, including from where they are received or to whom they are sent. And finally, modules need to share a common world view, or apparently share one, so their implicit context can be correctly used.

With these concepts in hand, I proceed to discuss the three components of implicit context. In Section 4.1.1, I describe boundaries, which are the transition zones between differing world views. To reconcile the differing world views, contextual dispatch is per-

¹In human conversation, we also perform operations such as checking that we have understood what is being discussed. I am not attempting to provide such operations via implicit context.
formed at these boundaries; Section 4.1.2 details this. And finally, communication history (outlined in Section 4.1.3) is the conceptual record of all communications that have occurred within the system; it is used to provide details for performing contextual dispatch.

4.1.1 Boundaries

We need to define the program fragments for which differing world views are in force. When communication occurs between these program fragments, we must translate this communication according to the differences in these world views in order to get the program fragments to work together. I define the program fragments to be modules, and define there to be a boundary around each module that demarcates the end of one world view and the imposition of another.

Ultimately, we care about the communications between these world views because these are what must be translated. This means that, for some program fragment to be defined as a module, it must be possible (at a minimum) to differentiate which communications arise inside the module and which arise outside, with a boundary between them. This leads to our definition of modules as follows.

Definition 4 (Module) Any program fragment can be defined to be a module as long as two constraints are met. First, every communication in the program, potential or actual, can be unequivocally identified as arising from either inside or outside that module. Second, every communication in the program, potential or actual, can be unequivocally identified as travelling to the inside of that module or not.

This begs the question of what a communication is, of course. Obvious forms of communication include method invocations and field accesses. Under some circumstances, it can be useful to consider other program instructions to be communications, such as arithmetic operations, type casts, or assignment operations. In general, any program instruction could be viewed as a communication, possibly to another module within the program, or possibly to some virtual machine executing the program. In the implementation discussed in Chapter 5 of this dissertation, I view communications as being limited to method invocations, field accesses, and type references. Other implementations could make different choices.

A concept similar to boundaries has long been used to separate and encapsulate the details within modules that are to be hidden from the external world [Parnas, 1972]; traditionally, a module then exposes only a high-level interface to its functionality, allowing the implementation behind this interface to be changed. Boundaries differ from module interfaces in two ways. First, they encapsulate the external context of the module (the system outside the module), in addition to the internal details. Second, each boundary acts as a facade both for the module itself, and for the external context. As a result, the external context may perceive the interface to the module to be quite different from that perceived
by the module itself and, similarly, the module (the inside of the boundary) may perceive
the interface to the external context to be quite different from that perceived by the external
context itself. This allows for differing world views to hold sway within each boundary.

We can consider boundaries to exist around obvious program modules, such as classes,
but also around less obvious ones, such as individual methods, individual statements within
methods, groups of classes, groups of methods, etc. Even very complicated boundaries
could be defined, such as a boundary around a method from one class, several methods from
a second class, an entire third class, and a single statement within a fourth class. Whether
such complex boundaries are needed or useful is a separate issue; nothing prevents us from
conceiving of such a convoluted boundary.

In simple cases, boundaries around individual methods, boundaries around individual
classes, and boundaries around individual (Java) packages nest simply. As a result, com­
munication to and from these modules cross one boundary at a time. However, program
fragments may be defined to belong to multiple modules; thus, boundaries need not be
well-nested but may overlap in complicated ways in the most general case. Therefore, a
definition of boundaries is required that permits arbitrary overlap of modules.

Definition 5 (Boundary) Every communication from inside a module that references the
system external to that module must cross the boundary of that module. Likewise, every
communication from outside a module that references some part of that module must cross
the boundary of that module.

As a result, communications to or from a program fragment in the intersection of two mod­
ules must cross two boundaries (see Figure 4.1). When one module is strictly a subset of
another, outgoing communication is considered to cross the boundary of the inner module
first and incoming communication is considered to cross the boundary of the outer module
first (see Figure 4.2). In the general case of overlap, no natural order to these boundaries
exists, but it is convenient to impose an order on them so we can unequivocally determine
which is crossed first; imposing such an order is an issue with which an implementation of
the model must cope, so we delay further discussion of this topic to Chapter 5.

4.1.2 Contextual Dispatch

With boundaries, we can segregate differing world views into well-defined subsets of our
systems. But these subsets must work in conjunction to form a whole, functioning system.
The presence of differing world views implies a disagreement as to the resolution of names
and interpretation of communication. Therefore, this disagreement must be reconciled to
achieve a coherent system.

Contextual dispatch is a mechanism for reconciliation between differing world views.
It is a generalization of such concepts as object-oriented polymorphism [Strachey, 1967;
public class Simple {
    int doit() { ... }
    void print() { ... }
    public static void main(java.lang.String[] args) { ... }
}

Figure 4.1: Modules can be defined that overlap. Here, we see a module that consists of the non-public methods within the class Simple and another that consists of the methods with void result types; the boundary of the former is represented as a dashed line, the latter as a solid line. If the print method communicates with the doit method, the solid-line boundary must be crossed. If print communicates with main the dashed-line boundary must be crossed. But if print communicates with any other modules, both these boundaries must be crossed.

public class Simple {
    int doit() { ... }
    void print() { ... }
    public static void main(java.lang.String[] args) { ... }
}

Figure 4.2: Modules can be defined such that one is strictly a subset of the other. Here, we see a module that consists of the print method within the class Simple and another that consists of the entire class Simple itself; the boundary of the former is represented as a dashed line, the latter as a solid line. If the print method communicates with the doit method or the main method, the dashed-line boundary must be crossed. If print communicates with any other modules, both these boundaries must be crossed.
Cardelli and Wegner, 1985], multiple dispatch [Bobrow et al., 1986; Chambers, 1992], predicate dispatch [Chambers, 1993; Ernst et al., 1998], and subjectivity [Harrison and Ossher, 1993].\(^2\) These other mechanisms allow for determining the method implementation that is to receive a given message variously on the basis of the types and values of the arguments of that message, the type of the receiving object, or the type of the sending object. Contextual dispatch additionally allows the recipient implementation (or implementations) to be selected on the basis of the context from which the message originates; we can utilize both structural information about the system and knowledge of the current and previous system state to further discriminate between potential recipients.

Contextual dispatch also allows for the interception and alteration of communication. The implicit context model is generic with respect to what communication is: any information that flows across a boundary is subject to contextual dispatch. We often make casual reference to message passing when discussing contextual dispatch, but such discussion is applicable to any other form of information flow too, such as procedure calls, events, signals, field accesses, or thrown exceptions. Communications are intercepted and, depending upon the context, may be:

- rerouted to a different recipient or multiple recipients,
- delayed or ignored completely, and/or
- altered to add, remove, or reorder arguments.

Figure 4.3 illustrates an example of contextual dispatch. As described in Section 3.2, for a class to participate in the Abstract Factory design pattern [Gamma et al., 1994] as a Client, it must explicitly name the AbstractFactory class in order to create instances of an AbstractProduct class. Such explicit knowledge of the Abstract Factory design pattern will generally not be part of a minimal specification of the class acting as Client and, thus, will be extraneous embedded knowledge there. Instead, the essential structure of the Client would simply be to create the AbstractProduct:

```java
AbstractProduct product = new AbstractProduct();
```

The difficulty lies in the fact that I need the system to use the Abstract Factory design pattern, with this class acting as Client. To reconcile this difference, I first permit my Client class to contain the statement above. When the message to create this AbstractProduct crosses the boundary of the Client, contextual dispatch is performed to reroute the message to the appropriate AbstractFactory instance. This rerouting may be to different AbstractFactory instances at different times, depending upon the state of the system. This implies that an implementation of contextual dispatch must provide a means for testing and retrieving the state of the system.

\(^2\)I leave further discussion of these other mechanisms to Chapter 8.
The class SomeClass apparently sends the message "new" to the class AbstractProduct; however, this message is intercepted at the boundary of SomeClass, altered to the message "makeAbstractProduct," and rerouted to one of two instances of AbstractFactory depending on the current state of the system.

Contextual dispatch could utilize additional method calls or field accesses to determine the current system state. In the Abstract Factory design pattern example, I need to locate an instance of AbstractFactory that meets some set of criteria; perhaps it is the most recently created instance of AbstractFactory, or the instance that was passed from a particular module, indicating that it is the one to be currently used. However, such state information is not always immediately retrievable from a given module; this is where we can make use of the communication history of the system.

### 4.1.3 Communication History

Contextual dispatch needs to access the state of the system in order to add arguments to a message, or to determine to where a communication is to be sent. Sometimes the required state information is available directly, where one or more methods can be called to retrieve it. Other times the required state information is not retrievable through the interfaces provided by the modules of our system. At these times, communication history can be used to retrieve the information of interest.

Communication history\(^3\) is a dynamic record of all communication that has occurred within the system from the start of its execution to the current point in its execution. Communication history was originally termed call history [Walker and Murphy, 2000]; however, it can provide access to other forms of communication, so I have chosen to change the name.
munication history provides access to all the method calls with their arguments, target objects, return values, and thrown exceptions, and all field accesses. Information is stored implicitly within the communication history during the system execution. Information is retrieved explicitly from the communication history via *queries*. Each query defines a set of constraints, such as the type of an object of interest, the call in which an object was passed as an argument, the causal or temporal order of calls, etc. A query is fulfilled by finding the information recorded within the communication history that satisfies these constraints. To see the utility of communication history, consider again the example involving the Abstract Factory design pattern (Figure 4.3).

In our example, contextual dispatch must determine to which instance of Abstract-Factory the message *makeAbstractProduct* is to be sent. There could be a method called *getCurrentFactory* already available on some class (say, FactoryCreator) in the system from which the appropriate instance could be retrieved. If such a method does not exist, some other means must be provided to allow the retrieval of this state. There are two options, which I describe below.

In the first option, I can use contextual dispatch to effectively add the method *getCurrentFactory* to the interface of FactoryCreator, as shown in Figure 4.4. Here, I use contextual dispatch to intercept the instantiation of a particular subclass of AbstractFactory, storing the newly created instance. Next, I use contextual dispatch to simulate the presence of the method *getCurrentFactory*. The attempt to call this method on FactoryCreator is intercepted, returning the stored instance of AbstractFactory. This instance is then used to create the abstract product of interest to SomeClass.

In the second option (shown in Figure 4.5), communication history is used. The instance of AbstractFactory created by FactoryCreator is implicitly recorded within the communication history. Contextual dispatch, upon intercepting the attempt by SomeClass to instantiate AbstractProduct, makes a query against communication history to retrieve this instance of AbstractFactory. The retrieved instance is then used to create the desired abstract product.

Consider the minimal specification of the module containing SomeClass that needs it to use the Abstract Factory design pattern. This minimal specification might explicitly require that the abstract product be created via that instance of AbstractFactory obtained through a method called *getCurrentFactory* on FactoryCreator. In this case, this module could reasonably be implemented according to the first option. However, it will not be very reusable as it will be explicitly dependent upon FactoryCreator.

A more realistic minimal specification would have an abstract requirement concerning the instance of AbstractFactory that is to be used. For example, the minimal specification might require the use of the most recently created instance of AbstractFactory.
Figure 4.4: Performing contextual dispatch to retrieve state through an added method. The instance of ConcreteFactory created by FactoryCreator (1) is intercepted and stored. When SomeClass attempts to instantiate AbstractProduct (2), contextual dispatch intercepts this attempt; in its place, the method getCurrentFactory is invoked on FactoryCreator (2.1). This invocation attempt is itself intercepted through contextual dispatch and the stored instance of AbstractFactory is returned in its stead (return arrow not shown). This instance is then used to create the abstract product desired by SomeClass (2.2).
Figure 4.5: Retrieving state through the use of the communication history. The instance of ConcreteFactory created by FactoryCreator (1) is implicitly stored within the communication history (1.1). When SomeClass attempts to instantiate AbstractProduct (2), contextual dispatch intercepts this attempt; in its place, a query on communication history is made (2.1) thereby retrieving this instance of ConcreteFactory, viewed as an AbstractFactory (the return arrow from call 2.1 possessing this reference is not shown). This instance is then used to create the abstract product desired by SomeClass (2.1).
without reference to its creator. The option using communication history would then be preferable in this situation. To see why, we must consider the nature of communication history queries. Communication history allows us to perform a pattern match on the state of the system, where as much or as little of the pattern as needed can be defined. In the example above, we can literally ask for the most recently created instance of AbstractFactory without saying anything about its origins. Such a query thus allows us to decouple our module from FactoryCreator. We can express the minimal necessary knowledge of the state of the system via such a query.

It is important to emphasize two points about communication history. First, the model presented above is conceptual. Pragmatic constraints prevent any practical implementation of implicit context from naively conforming to this model; for example, there is a computational overhead associated with recording every method call that occurs in the execution of a system, and the longer a system runs, the larger the space required to record the communication history. However, a practical implementation of implicit context need not actually adhere to this model in order to give the appearance of doing so; static analysis can be performed to strip down the overhead to support communication history. I examine the opportunities for realizing a practical implementation in Chapters 5 and 7. Second, it is important that communication history queries be made with respect to the local world views of the modules in which contextual dispatch occurs. Otherwise, we still risk embedding global assumptions in our modules; they would merely be moved out by one layer of indirection.

With this outline of the implicit context model in hand, I proceed to describe how it can be used in order to achieve our goals: expressing essential structure in order to increase the reusability of our modules while making them less sensitive to the evolution of our systems.

4.2 Using Implicit Context to Express Essential Structure

Previous chapters have identified two features that must be provided to express essential structure: retroactive abstraction of our modules and their composition to form concrete, coherent systems. In this section, I examine two approaches to meeting these goals. To do this, I make use of the example from Chapters 2 and 3 of the makeContributions method of the JavaOutlinePage class of Eclipse. Recall that the minimal specification of makeContributions was described in Section 3.1 as:

- makeContributions must add functionality to the status line, if there is one, to reflect selection changes in the Outline View.
- It must add four actions to the tool bar, one for sorting declarations lexically, one for hiding fields, etc.
In the first approach, described in Section 4.2.1 (below), I provide an implementation of the module that expresses only its essential structure, and demonstrate how to have such a module interoperate with a complete system. I implement the minimal specification directly. The resulting, abstract implementation is then embedded in a concrete context. I use the implicit context model to translate between its local world view and the system-wide world view.

Realistically, we cannot always expect to correctly identify the minimal specification of a module. Even when we do, a given implementation may provide behaviour only some of which is of interest to us. Therefore, situations will arise in which a reused module contains what we consider to be extraneous embedded knowledge. Rather than refactor such a module or reimplement it, it might be preferable to somehow “cancel out” the EEK already present.

In the second approach, described in Section 4.2.2, I begin with an existing implementation that I wish to reuse. However, this implementation provides more than the essential structure of makeContributions—it contains EEK. I need to have this module operate as though it expressed only its essential structure, so I could then embed it in a concrete context in a fashion identical to the first approach. Therefore, I use implicit context in this second approach to capture any communication from my reused module that expresses EEK with respect to the minimal specification of makeContributions. The result is indistinguishable from the abstract implementation provided in the first approach.

4.2.1 Adding Contextual Details to an Abstract Module

To create an implementation of makeContributions that is highly reusable and unlikely to evolve, we can express its essential structure by adhering to its minimal specification as closely as possible. The result, using Java, is shown in Figure 4.6. Because I have chosen to write this module in Java, I have used several simple assumptions, including that the statusLine and toolBar are passed as arguments, that their types are Status-Line and ToolBar respectively, that toolBar has a method addAction, etc. Strictly speaking, all such assumptions are extraneous embedded knowledge. However, if we treat them as abstractions rather than as concrete expressions, they are not constraining and effectively do not act as EEK. I show how this is possible below, as I embed this implementation in a concrete context.

Embedding the Abstract Module in a Concrete Context

Consider taking the implementation of Figure 4.6 and using it in a concrete context. As an example, this implementation could be used within a larger implementation for Eclipse as a whole. Let us say that the higher-level module containing makeContributions
public void makeContributions(StatusLine statusLine,
    ToolBar toolBar) {
    if (statusLine.exists())
        reflectOutlineViewSelectionChanges(statusLine);

    toolBar.addAction(new LexicalSortingAction());
    toolBar.addAction(new FieldVisibilityFilterAction());
    ...
}

Figure 4.6: An implementation of makeContributions, in Java, expressing only its essential structure.

possesses a minimal specification that includes all the concrete details in the original implementation of makeContributions, repeated in Figure 4.7. We must use implicit context to add in these concrete details to the abstract module given in Figure 4.6, translating between the world view of Figure 4.6 and that of Figure 4.7. There are a series of translations that we need to make.

The first step of the translation lies in the differences between the signature of the abstract module and the signature expected by Eclipse as a whole. This means that any incoming messages (which would expect the signature of the original implementation in Figure 4.7) crossing the boundary of makeContributions must be intercepted and altered to the signature of our essential structure implementation (in Figure 4.6). Conceptually, this involves three operations:

• discarding the menuManager argument;
• swapping the toolBarManager and statusLineManager arguments; and
• replacing the IToolBarManager and IStatusLineManager types with ToolBar and StatusLine respectively.

The first two operations are straightforward. The third is too because, for these two types at least, we can define a bijective mapping between the world views of our essential structure implementation of makeContributions and the implementation of the higher-level module containing it; in other words, these types are simply different labels that are interchangeable.

In the second translation step, I intercept the outgoing message "statusLine.exists()" and replace it with: "statusLine != null." The third translation is similar, intercepting the message:
public void makeContributions(IMenuManager menuManager,
    IToolBarManager toolBarManager,
    IStatusLineManager statusLineManager) {
if (statusLineManager != null) {
    StatusBarUpdater updater =
        new StatusBarUpdater(statusLineManager);
    addSelectionChangedListener(updater);
}

Action action = new LexicalSortingAction();
toolBarManager.add(action);

action =
    new FilterAction(new FieldFilter(),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideFields.label"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideFields.description.checked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideFields.description.unchecked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideFields.tooltip.checked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideFields.tooltip.unchecked"),
        "HideFields.isChecked");
    JavaPluginImages.
        setImageDescriptors(action, "lcll6", "fields_co.gif");
toolBarManager.add(action);

...
reflectOutlineViewSelectionChanges(statusLine);
and replacing it with two messages:

StatusBarUpdater updater =
    new StatusBarUpdater(statusLine);
addSelectionChangedListener(updater);

Finally, the world view of Eclipse as a whole does not possess a class called Field-
VisibilityFilterAction; therefore, I must intercept the attempt at instantiating it,
and replace it with the sequence of operations:

FilterAction action =
    new FilterAction(new FieldFilter(),
    JavaEditorMessages.getString(  
        "JavaOutlinePage.HideFields.label"),
    JavaEditorMessages.getString(  
        "JavaOutlinePage.HideFields.description.checked"),
    JavaEditorMessages.getString(  
        "JavaOutlinePage.HideFields.description.unchecked"),
    JavaEditorMessages.getString(  
        "JavaOutlinePage.HideFields.tooltip.checked"),
    JavaEditorMessages.getString(  
        "JavaOutlinePage.HideFields.tooltip.unchecked"),
    "HideFields isChecked");
JavaPluginImages.
setImageDescriptors(action, "lcl16", "fields_co.gif");
returning the new instance of FilterAction instead.

After these translations, applied externally to the essential structure of makeCon-
tributions, we have a concrete implementation interoperating with the rest of Eclipse
that remains free from EEK.

An Alternative Interpretation of the Essential Structure

In the situation described above, the mapping between the world view of the essential struc-
ture implementation of makeContributions and the world view of Eclipse is straight-
forward, consisting of mere rearrangement, discarding of some information, and one-for-
one replacement. However, I could have chosen a slightly different implementation of the
essential structure of makeContributions. This is especially true since the informal
minimal specification of makeContributions is ambiguous: it is unclear whether we
should consider there to be a single status line and tool bar in existence (i.e., some sort
of global properties) or not. Let us consider how this ambiguity might affect the essential
structure and implementation of makeContributions.

The implementation I gave in Figure 4.6 takes the view that the status line and tool
bar parameterize makeContributions. One alternative interpretation could be imple­
mented thus:

```java
public void makeContributions() {
    if (statusLineExists())
        reflectOutlineViewSelectionChangesInStatusLine();

    addActionToToolBar(new LexicalSortingAction());
    addActionToToolBar(new FieldVisibilityFilterAction());
    ...
}
```

A second alternative interpretation treats statusLine and toolBar as fields or
globals:

```java
public void makeContributions() {
    if (statusLine.exists())
        reflectOutlineViewSelectionChanges(statusLine);

    toolBar.addAction(new LexicalSortingAction());
    toolBar.addAction(new FieldVisibilityFilterAction());
    ...
}
```

In both these alternative implementations, in order to correctly interpret the particular
status line and tool bar objects to which the messages are to be passed, I need to retrieve
the objects that were passed to makeContributions but discarded at the boundary.
Therefore, I can perform a communication history query when the outgoing messages are
intercepted, locate the appropriate status line and tool bar objects, and pass the messages
on to them. This is illustrated in Figure 4.8. In other words, the ambiguity in the minimal
specification can still be dealt with by the application of implicit context.

In all these situations, I have implemented makeContributions to express only
its essential structure, and used implicit context to add in the concrete details needed by
higher-level modules. I now consider an alternative scenario, where my implementation
contains EEK, but I still wish to reuse it.
Figure 4.8: Using communication history to retrieve discarded information. In (a), some external client calls `makeContributions`, passing three objects to it as arguments; this call is intercepted and the arguments are discarded. In (b), `makeContributions` attempts to call `addAction` on some object of type `ToolBar` that it believes to exist; this call is intercepted, the object of type `IToolBarManager` that was discarded in (a) is retrieved from communication history, and the `add` method is called upon it. (The argument list for `addAction` and `add` has been left out of the diagram.)
4.2.2 Cancelling Out EEK Already Present within a Module

Rather than implementing makeContributions from scratch, we may decide that some existing implementation almost does the job. However, this existing implementation likely expresses knowledge of its external context that is EEK relative to the minimal specification of interest. Therefore, we need a means to translate the existing implementation to simulate a module that expresses only the essential structure of interest; we must somehow "cancel out" the EEK.

Consider starting with the original implementation of makeContributions, shown in Figure 4.7. I want to translate this implementation, so that, from the perspective of an external viewer, it appears to provide only the essential structure of makeContributions. The implementation of makeContributions shown in Figure 4.6 is much closer to the essential structure. We can thus think of this cancellation of EEK as a translation from the original implementation to the essential structure implementation, consisting of a series of steps. This transformation changes the externally observable structure and behaviour of the module; however, I describe the transformation as though I am altering the internal implementation. From the external perspective, there is no way to see a difference between these two descriptions.

As a first step, I need to modify the signature of makeContributions to eliminate the first formal parameter, swap the other two, and alter the types from IStatusLineManager and IToolBarManager to StatusLine and ToolBar respectively. This alteration is illustrated in Figure 4.9. Now, the world view external to the boundary of makeContributions perceives it as having the signature required by the minimal specification, while internal to the boundary, it has the original signature. In this situation, when I translate an incoming message that lacks an argument corresponding to the IMenuManager formal parameter, I have contextual dispatch simply add null to the message in this position. If I were dealing with a similar situation in which adding a null argument did not suffice, I would need contextual dispatch to create a dummy instance or retrieve an instance from communication history, as appropriate.

The next bit of EEK present in the original implementation involves extraneous details involved with the status line. The minimal specification requires that the implementation add functionality to the status line, if it exists, to reflect selection changes in the Outline View. The original implementation of makeContributions simply checks if statusLineManager is null to determine if it exists, but this makes an extraneous assumption about what it means for the status line to exist. Therefore, this test should be altered somehow, but difficulty arises from the fact that one might not consider the expression

statusLineManager != null
Figure 4.9: Altering an incoming message to hide the presence of EEK within a module. Within the boundary of `makeContributions`, the instances of `StatusLine` and `ToolBar` are viewed as instances of `IStatusLineManager` and `IToolBarManager`; their positions are also swapped within the message. Since the original implementation of `makeContributions` expects an instance of `IMenuManager` as its first argument, `null` is added to the message in this position.

To be communication that flows across the boundary of `makeContributions`. With such an interpretation, this test cannot be intercepted and altered to the more general form of interest to us, namely "statusLineManager.exists()." I consider an alternative solution in the next subsection.

Since I wish to eliminate mention of `Action` and its attendant constraints that `LexicalSortingAction` and `FilterAction` be subclasses of it, I should also replace reference to `Action` with reference to `LexicalSortingAction`. Finally, the detailed set of calls to create and initialize the instance of `FilterAction` need to be captured and discarded, returning `null` instead. The second call to `addAction` is also captured, and its argument replaced with a call to instantiate `FieldVisibilityFilterAction` before it is allowed to continue.

I have thus successfully modified the original implementation of `makeContributions` to express only the essential structure (I discuss the difficulties of capturing the inequality test below). The resulting module can now be reused in any context where its minimal specification is needed.
Coping with Statements that Do Not Produce Communication

Above, I pointed out that certain statements and expressions, such as inequality tests, do not obviously produce communication. If we choose to consider them as producing communication, coping with them is straightforward.

The alternative, of not considering them as producing communication, requires two steps: (a) add the `if`-statement appearing at the start of the essential structure implementation of `makeContributions`; and (b) effectively delete the `if`-statement from the start of the original implementation.

Recall that the external world view expects that `makeContributions` will have the signature appearing in Figure 4.6. For step (a), I capture calls to the expected signature function `makeContributions(StatusLine, ToolBar)`

and use contextual dispatch to perform the functionality of the new `if`-statement and to invoke the actual signature of `makeContributions`, in that order. Thus, we can think of the change as though the call were to this facade:

```java
public void makeContributions(StatusLine s, ToolBar t) {
    if(statusLine.exists())
        reflectOutlineViewSelectionChanges(statusLine);
    makeContributions(null, (IToolBarManager)t,
                     (IStatusLineManager)s);
}
```

For step (b), presumably the `if`-statement at the start of Figure 4.7 would not be considered a communication either, so I must capture the individual calls within its body (to instantiate `StatusBarUpdater` and call `addSelectionChangedListener`) and discard them.

This alternative solution is not very satisfactory, since it effectively throws out several lines of code, calling into question the purpose to reusing this module in the first place. Indeed, the reduction to essential structure in this example is extreme and may indicate that doing so is not worthwhile. All in all, it would be better to treat the test for the `null` status line as an outgoing message, in order to avoid convoluted solutions. Implicit context does permit EEK already present within our modules to be cancelled out, even in such problematic situations.

Although it is possible to cancel out the EEK present in an existing implementation, it may require some convolutions to do so. Too much discarding and too many convolutions themselves will lead to structural degradation, the avoidance of which motivated the need for implicit context in part. This dissertation does not address at what point it becomes
more expensive (in whatever sense) to reuse an ill-fitting module than to implement one expressing the required essential structure.

4.3 Summary

Implicit context permits each module to have its own, independent view of the world in which it operates. These views must be reconciled when the modules within a system communicate, in order for those modules to be able to work together.

To this end, implicit context defines there to exist a boundary around each module; the world view within this boundary need not agree with the world view outside it. When communication crosses these boundaries, it is intercepted and translated according to the differences in world views on either side of the boundary; this operation is called contextual dispatch. It sometimes occurs that additional information must be added to the communication crossing a boundary or that the recipient of such communication depends on the state of the system. When such information cannot be simply retrieved via the interfaces provided by our modules, we can use the history of communication within the system to retrieve otherwise hidden information about the system state. Such use of the communication history can allow the definition of the system state to be expressed explicitly instead of, for example, assuming that an invariant exists between the system state and the value of an encoding of that state as recorded within some variable.

Implicit context allows for the expression of essential structure within our modules, and for the elimination of extraneous embedded knowledge already present there. A module, whose minimal specification is indifferent to contextual details of interest to a higher-level module, can have those details added through implicit context by intercepting incoming and outgoing communication and altering it appropriately. A module that possesses extraneous contextual details can have those details cancelled out through implicit context also by intercepting incoming and outgoing communication and altering it appropriately.

Chapter 5 describes how implicit context can be and has been implemented in a prototype tool; in Chapter 6, I give concrete examples of case studies I have performed using implicit context; and further optimizations and pragmatic considerations for better implementations are also discussed in Chapter 7.
Chapter 5

A Prototype Tool for Utilizing Implicit Context in Java

In Chapter 4, I presented a model of essential structure through implicit context. To realize this model, we must possess a mechanism for implementing it. Such a mechanism requires a means to define boundaries between independent world views; to capture information flowing across these boundaries; and to perform contextual dispatch on the captured information flow, including the use of communication history.

To evaluate the implicit context model, I have constructed a prototype tool that supports the main features of the model for use on source code written in the Java programming language [Gosling et al., 2000]. I chose to work with programs written in Java so as to be able to apply implicit context to large existing systems. Although the work in this chapter is devoted to tackling Java specifically, the concepts of essential structure and implicit context are more generally applicable. Developing a special purpose language tailored to implicit context is premature at this point.

The prototype tool operates as a preprocessor to Java source code. I chose to have the tool support boundaries only around individual classes and methods, largely because these declarations are easily named in Java source. To support interception and alteration of communication at these boundaries, I provide a construct called a boundary map; system developers must explicitly create boundary maps to perform contextual dispatch. Boundary maps are declared in files that are separate from the Java source code. System developers also explicitly declare, through tool directives, to which boundary or boundaries a given boundary map is to be attached.

I begin, in Section 5.1, by giving an overview of the syntax and operation of boundary maps through a simple example. I describe the general structure of the tool in Section 5.2 along with an overview of its usage, leaving the non-trivial details of its algorithms to the other sections of this chapter. I then describe the features of boundary maps as provided
by the tool in Section 5.3. Tool directives are described in Section 5.4. I consider, in Section 5.5, how boundary maps are sequentially applied so that multiple boundary maps can be composed. Finally, Section 5.6 describes some of the implementation issues of the tool.

5.1 A Simple Example

To examine the major features of my prototype tool, consider the following example that uses implicit context; this example was introduced in Section 1.2. We have a simple Java class, called CompoundWidget, for creating a particular kind of compound widget comprising a button and a scroll bar. Shown below is part of the declaration of CompoundWidget, including the declarations for two fields that hold its button and scroll bar objects and a constructor declaration.

```java
public class CompoundWidget {
    private Button button;
    private ScrollBar scrollBar;

    public CompoundWidget() {
        button = new Button();
        scrollBar = new ScrollBar();
        // Perform other setup
    }

    // Other methods
}
```

The constructor instantiates the Button and ScrollBar classes, and stores each resulting object in the appropriate field of the newly-created instance of CompoundWidget. This source code would occur in a file named CompoundWidget.java.

Now, imagine that we have different versions of Button and ScrollBar for different platforms, such as a version operating with the Motif library to run on Unix systems and a version to operate on Microsoft Windows systems. We decide to use the Abstract Factory design pattern [Gamma et al., 1994] to isolate these platform-specific differences from the rest of our system. The class structure at which we arrive is shown in Figure 5.1. Here, Button and ScrollBar are abstract base classes, each specialized into platform-specific versions. There is also a hierarchy of widget factories, consisting of an abstract base class called WidgetFactory and platform-specific subclasses. Each platform-specific widget factory creates instances of the corresponding platform-specific versions of Button and
Figure 5.1: Using the Abstract Factory design pattern to isolate platform-specific dependencies. We have separate subclasses of Button and ScrollBar for both Motif and Microsoft Windows systems.

ScrollBar. Clients can ignore platform differences by possessing an instance of either subclass of WidgetFactory but treating it as being of type WidgetFactory. Calling the methods on this instance causes the appropriate instances of Button or ScrollBar to be created polymorphically. One part of the system must still create an instance of the appropriate widget factory subclass and pass it to the client, and thus, this part will be aware of which platform is in use.

I (in my role as a particular developer of this example) wish to allow the interior of the CompoundWidget module to ignore the added complexity of Abstract Factory, while having the system as a whole utilize these platform-specific subclasses. The system as a whole needs to interact with CompoundWidget as though the Abstract Factory pattern were in use there. Thus, the declaration of CompoundWidget should be externally indistinguishable from the following declaration.

```java
public class CompoundWidget {
    private Button button;
    private ScrollBar scrollBar;

    public CompoundWidget(WidgetFactory fact) {
```


button = fact.createButton();
scrollBar = fact.createScrollBar();
// Perform other setup
}

// Other methods
}

The original declaration describes one world view, that which holds sway within CompoundWidget. This new declaration describes a different world view, that which holds sway in the system external to CompoundWidget. The differences between these two declarations are that an instance of WidgetFactory should be passed to the constructor, and that this instance should be used to build the individual pieces of the CompoundWidget. In fact, since Button and ScrollBar are abstract classes in the view of the system as a whole, they are not directly instantiable.

To translate between these differences, I begin by considering there to be a boundary around CompoundWidget between these independent world views. The prototype tool supports the use of boundaries around individual classes and individual methods; the fully qualified name of the class or method is used to refer to its boundary. Next, I must specify contextual dispatch at this boundary to resolve the conflict between the world views. I specify three boundary maps to do this: one that translates incoming attempts at constructing CompoundWidget instances parameterized by a WidgetFactory object; one that translates outgoing attempts at instantiating Button; and one that translates outgoing attempts at instantiating ScrollBar. The operation of these boundary maps is depicted in Figure 5.2. I walk through the boundary map code below.

I group these three boundary maps into a construct called a mapset. The mapset below is named widgetFactory to allude to its purpose. Mapsets are specified in files separate from source code files.

// Begin running example 5.1

mapset widgetFactory {

The first boundary map translates parameterized calls into the CompoundWidget constructor. Since no parameterized constructor exists within the boundary, such calls must be replaced by calls to the non-parameterized constructor. The communications to be captured by this boundary map are specified in the capture clause to the right of the colon on the line below; this boundary map intercepts in-bound messages to CompoundWidget parameterized by WidgetFactory so the capture clause uses the keyword in. Each boundary map must declare its result type, even when we are dealing with a constructor; the
public class CompoundWidget {
    private Button button;
    private ScrollBar scrollBar;

    public CompoundWidget() {
        button = new Button();
        scrollBar = new ScrollBar();
    }

    public CompoundWidget(WidgetFactory fact) {
        button = fact.createButton();
        scrollBar = fact.createScrollBar();
    }
}

Figure 5.2: The operation of boundary maps attached to CompoundWidget. In (a), messages from the external system that attempt to instantiate CompoundWidget via a call to a parameterized constructor (which does not actually exist) are rerouted to the non-parameterized constructor. In (b) and (c), attempts to instantiate Button and ScrollBar within the boundary are rerouted via an instance of a widget factory. (Note that the use of communication history, described in the main text, is not depicted here.)
result type here is CompoundWidget, declared at the beginning of the line below. Once a communication conforming to the description in the capture clause crosses the boundary, it is captured and the body of the boundary map (declared within a matching pair of braces) is executed in its place.

CompoundWidget map(): in(CompoundWidget(WidgetFactory)) {

The body of this boundary map simply indicates that the non-parameterized constructor should be run in place of the original call. The argument of type WidgetFactory passed in the original call is discarded.

this();
}

The second boundary map specifies the translation of requests to instantiate Button. Any communications passing out across the boundary that are to instantiate Button are captured by this boundary map, so it uses the keyword out (additional forms are available for capturing each access that gets or sets the value of a field). An object of type Button will be returned to the site that sent the captured communication.

Button map(): out(Button.new()) {

I need to replace the call to instantiate Button with the appropriate call on an instance of WidgetFactory, but no such instance is lying about for us to use. Therefore, I query communication history to retrieve an instance of WidgetFactory. Specifically, I find the instance that has most recently been passed in any communication. Communication history queries are implemented through a set of methods on the History class, provided by the tool. This particular query requires an instance of type Class and returns an instance of type Object; since we know that this must actually be an instance of type WidgetFactory, I downcast the returned object.

---

1In Java, one constructor on a given class may invoke another constructor on the same class. This is done via the syntax `this` followed by the argument list appropriate to the desired constructor. In this example, we are replacing the invocation of one constructor (which takes a single argument of type WidgetFactory) with another (which takes no arguments). The latter invocation is implicitly on the same class as the former.

2In Java, one instantiates classes through the syntax `new SomeClass(args)." To refer to these instantiation attempts, the syntax is changed a little (to "SomeClass.new(args)") for consistency with other message capture syntax.

3The selection of this particular instance of WidgetFactory is a design choice on the part of the developer who creates the boundary maps; other choices are possible using other communication history queries, default values, etc.

4In Java, one may access the instance of type Class that represents a particular class, say SomeClass, through the syntax "SomeClass.class." The resulting instance can be used in standard Java introspection.
WidgetFactory fact = (WidgetFactory)
    History.lastInstancePassed(WidgetFactory.class);

The first call to construct an instance of CompoundWidget may come before the first call
to construct an instance of WidgetFactory; the communication history query would
then return null. In such a case, I choose to create a default widget factory of the Motif
variety.\(^5\)

```java
if(fact == null)
    fact = new MotifWidgetFactory();
```

Finally, I use the retrieved (or created) widget factory to create and return a button object
that is platform-specific.

```java
    return fact.createButton();
}
```

The third and final boundary map is much like the second, the only differences being
that the communications captured are to instantiate ScrollBar and that the call to the
widget factory creates and returns an instance of ScrollBar.\(^6\)

```java
ScrollBar map(): out(ScrollBar.new()) {
    WidgetFactory fact = (WidgetFactory)
        History.lastInstancePassed(WidgetFactory.class);

    if(fact == null)
        fact = new MotifWidgetFactory();

    return fact.createScrollBar();
}
```

// End running example 5.1

Boundary maps can sometimes be specified in a way that permits their reuse in con­
junction with multiple maps. Therefore, we must indicate to the tool which mapsets should
be attached to which boundaries; this is done with tool directives. The following tool direc­
tive indicates that the mapset named widgetFactory should be attached to the boundary

---

\(^5\) Again, this is not the only choice available.

\(^6\) This boundary map replicates the core of the earlier one. Such replication could be reduced by
the support for mechanisms similar to C++ templates, for example. However, such extensions are
optimizations. They do not affect the key properties of the tool, so they remain a topic for future
work.
named CompoundWidget (the tool determines that there is a class of the same name, and therefore, it is the boundary around this class that is being referenced). Tool directives are specified in one or more files separate from those containing source code or boundary maps.

apply widgetFactory to CompoundWidget;

To apply implicit context in this situation, the prototype tool is run with the three files containing the tool directive, the widgetFactory mapset, and the CompoundWidget source code. The tool produces a new set of Java source code that is ready for compilation. This new source code is not intended to be human readable; if changes are needed, the tool is re-run with changed inputs. The resulting source code simulates the effects of the implicit context model: the CompoundWidget module is oblivious to the presence of the Abstract Factory design pattern, while the system as a whole sees this design pattern in use.

5.2 Tool Structure

Figure 5.3 shows an overview of the operation of the prototype tool. The system developer provides three kinds of inputs to the tool: the original Java source code on which to apply implicit context, the boundary maps describing the use of implicit context, and the tool directives indicating which boundary maps attach to which modules within the source code. Each kind of input is provided in one or more files separate from the other kinds of input.

The tool parses the original source code and boundary maps and forms abstract syntax trees (ASTs) of the code found there. The parser is generated by JJTree and JavaCC from an annotated grammar for the Java language. This grammar is a modified form of the one provided in the JavaCC distribution; it has been tweaked to make the resulting ASTs more explicitly represent such things as method invocations.

According to the tool directives given to the tool, the boundary maps are combined with the original source code. To do this, the tool replicates pieces of the boundary map AST and inserts them at particular points of the source code AST; further details of this process are provided in Section 5.6.1.

The combined AST is then traversed and modified in order to support communication history queries. Instructions are inserted at particular nodes within the AST to record the communication history; further details of instrumentation and communication history support are provided in Sections 5.3.5 and 5.6.2.

The final, instrumented AST is then translated back into Java source code. This new version of the source code is not intended for human consumption. If the developer requires changes to the resulting system, the inputs to the tool are altered and the tool re-run. In fact, the instrumentation and other transformations make the new source code difficult for a

---

7Both available from WebGain at the time of writing, http://www.webgain.com/
Figure 5.3: Overview of the operation of the prototype tool. See the main text for an explanation of the four steps involved.
human to read. The tool could as well compile the code as produce transformed source; I chose the source-level transformation approach in order to permit debugging of the tool.

5.3 Boundary Maps

For convenience, rather than naming each individual boundary map, boundary maps are collected into sets called mapsets. Each mapset possesses a name, unique among all mapsets, that can be used within tool directives. Below is a simple mapset called example:

```
mapset example {
    void map(): in(someMethod()) throws SomeException {
        System.err.println("Calling someMethod()" );
        CONTEXT.proceed();
    }
}
```

The example mapset contains a single boundary map. Usually there will be multiple boundary maps in a mapset, and each mapset can be applied to more than one boundary. Each boundary map consists of up to six parts: a capture clause, formal parameters, an optional throws clause, a body, a result type, and optional modifiers; mapsets conform to the Backus–Naur Form8 (BNF) syntax [Naur et al., 1960] shown in Figure 5.4.9

The capture clause specifies which communications should be intercepted that cross the boundary to which this boundary map is attached; the example specifies that communications should be intercepted that cross the boundary from the external context into the module that are bound for the someMethod method. I look at communication capture in Section 5.3.1. Boundary maps can allow the arguments of captured communications to be exposed to the body of the boundary map, although the communications captured here possess no arguments; I examine how this is done in Section 5.3.2.

The types of exceptions thrown within the body of the boundary map sometimes need to be explicitly declared within a throws clause, in identical fashion to the rules for Java methods [Gosling et al., 2000: §8.4.4]. The example above declares that the body can throw a SomeException. To translate from a world view where a call can result in an exception to one where it cannot, one declares a set of exceptions within the boundary

---

8Surrounding non-terminals with "<" and ">") has been dropped in favour of slanted text. Uppercase slanted text refers to non-terminals unchanged from their definition in the grammar for Java; lowercase refers to non-terminals that are new or redefined. Terminals are denoted by boldface, for keywords, or are surrounded by quotation marks, for special characters.

9This syntax is intentionally reminiscent of that of AspectJ [Kiczales et al., 2001], the connection with which is described in Chapter 8.
mapset ::=  
  mapset IDENTIFIER "{"  
  { boundary-map }  
  "}"  

boundary-map ::= standard-boundary-map | renaming-map  

standard-boundary-map ::=  
  [ modifier ] map FORMAL-PARAMETERS ":"  
  capture-clause [ THROWS-CLAUSE ]  
  CONSTRUCTOR-BODY

Figure 5.4: BNF syntax for mapsets. The production rules for the non-terminals IDENTIFIER, FORMAL-PARAMETERS, THROWS-CLAUSE, and CONSTRUCTOR-BODY are identical to those for identifiers, formal parameters, method throws, and constructor bodies in Java [Gosling et al., 2000: §3.8, §8.4.1, §8.4.4, & §8.8.5 respectively]. Other non-terminals are defined later in the chapter.

map's throws clause. To translate in the opposite direction, one must place the contextually dispatched call within a try block, and handle the exception in an associated catch block.

The body of the boundary map specifies how these captured communications should be altered; in general, each body can consist of an arbitrary block of Java source code, with a few additions and restrictions. In this case, the boundary map has been specified to invoke a print statement prior to the execution of someMethod. The body indicates that the original communication should be allowed to proceed on its way through a call to proceed, indicated by the statement\(^\text{10}\):

\[ \text{CONTEXT}.\text{proceed}(); \]

The call to proceed is used, rather than re-invoking the original method (i.e., calling someMethod again). Section 5.3.3 discusses the reasons for using this construct.

Boundary maps can be prefaced by certain modifiers just as constructor, method, and field declarations can. The permitted modifiers and their interaction with the modifiers of source code declarations are described in Section 5.3.4. This simple boundary map does not need to make use of communication history, but this is often not the case; I describe how communication history can be used in Section 5.3.5. An additional form of boundary map is provided by the tool, the renaming map; this is described in Section 5.3.6.

\(^{10}\)This syntax was chosen to reduce the likelihood that an existing name (such as proceed) in the input source would be treated as a keyword. Also, there was the possibility that other context references might be needed. The call to proceed is the only one defined by the prototype tool.
5.3.1 Capturing Communications

Communications may be captured as they cross boundaries. The capture clause of each boundary map specifies the kind of communications that are captured by that boundary map. While communication capture and alteration are concepts in the dynamic realm, the tool uses and modifies the static properties of the code it operates on to achieve the effect of dynamic capture and alteration. The implementation of this process is described in Section 5.6.1.

Boundary maps are differentiated on the basis of whether they map communications crossing the boundary from the outside to the inside (in-maps) or vice versa (out-maps). In addition, boundary maps can be defined to select communications that attempt to set the value of a field or to get the value of a field; these are special forms of out-maps. Each of these four options is denoted by a different event designator\(^1\) appearing in the capture clause: in, out, gets, or sets.

Capture clauses conform to the BNF syntax shown in Figure 5.5. Each event designator is parameterized by a communication description, consisting of the name of the method (or field) being called and a set of formal arguments. Each formal argument is either a type name or the name of a formal parameter; this latter kind is described further in Section 5.3.2.

Not all capture clauses can be used in conjunction with all kinds of boundaries, even

\(^1\)Strictly speaking, these are not keywords but identifiers treated in a context-sensitive way by the tool’s parser; they may be used elsewhere as ordinary identifiers in the boundary map code or the source code.
of those combinations for which a meaningful interpretation can be defined, not all are sus-
portable in Java. The combinations consist of three dimensions: kind of event designator,
communication description properties, and kind of boundary. Tables 5.1, 5.2, and 5.3 de-
scribe the treatment of each combination; the four communication kinds are represented as
*id* (bare identifier), *Q.id* (qualified name), *id(args)* (unqualified name with formal argu-
ments), and *Q.id(args)* (qualified name with formal arguments).

For **in** event designators, communication descriptions can have prefixes that are iden-
tical to the names of the boundaries on which these communications are being intercepted;
such prefixes are stripped from the communication description prior to considering how
to deal with it. This is where the column marked "—" comes from: if the communica-
tion description consists solely of the name of the boundary on which the communication
is being intercepted, I have found the final destination for the communication. For exam-
ple, in-mapping communications of the form *doit()* on the boundary around a method
named *doit()* indicates that the entry in the "—" column should be considered. Sim-

<table>
<thead>
<tr>
<th>Use of <strong>in</strong></th>
<th>—</th>
<th><em>id</em></th>
<th><em>Q.id</em></th>
<th><em>id(args)</em></th>
<th><em>Q.id(args)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>—</td>
<td><strong>field access</strong></td>
<td><strong>field access on nested type</strong></td>
<td>method or constructor invocation</td>
<td>method or constructor invocation on nested type</td>
</tr>
<tr>
<td>Interface</td>
<td>—</td>
<td><strong>field access</strong></td>
<td><strong>field access on nested type</strong></td>
<td>—</td>
<td>method or constructor invocation on nested type</td>
</tr>
<tr>
<td>Method</td>
<td><strong>method invocation</strong></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Constructor</td>
<td><strong>constructor invocation</strong></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Field</td>
<td><strong>field access</strong></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 5.1: Interpretation and support for capturing communications, for **in** event designa-
tors, by each kind of boundary. The symbol — indicates that there is no simple interpretation
for the occurrence of the given communication at the given boundary, and so the case causes
the tool to indicate an error. **Boldface** entries indicate that capture of the given communi-
cation kind is fully supported at the given boundary. **Italicized** entries indicate that capture
of the given communication kind is unsupported at the given boundary. **Roman entries**
indicate that capture of the given communication kind is partially supported at the given
boundary; see the main text for details.

---

76
<table>
<thead>
<tr>
<th>Use of <code>out</code></th>
<th><code>id</code></th>
<th><code>Q.id</code></th>
<th><code>id(args)</code></th>
<th><code>Q.id(args)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>field access</td>
<td>field access</td>
<td>method or constructor invocation</td>
<td>method or constructor invocation</td>
</tr>
<tr>
<td>Interface</td>
<td>field access</td>
<td>field access</td>
<td>method or constructor invocation</td>
<td>method or constructor invocation</td>
</tr>
<tr>
<td>Method</td>
<td>field access</td>
<td>field access</td>
<td>method or constructor invocation</td>
<td>method or constructor invocation</td>
</tr>
<tr>
<td>Constructor</td>
<td>field access</td>
<td>field access</td>
<td>method or constructor invocation</td>
<td>method or constructor invocation</td>
</tr>
<tr>
<td>Field</td>
<td>field access</td>
<td>field access</td>
<td>method or constructor invocation</td>
<td>method or constructor invocation</td>
</tr>
</tbody>
</table>

**Table 5.2:** Interpretation and support for capturing communications, for `out` event designators, by each kind of boundary. **Boldface** entries indicate that capture of the given communication kind is fully supported at the given boundary. Roman entries indicate that capture of the given communication kind is partially supported at the given boundary; see the main text for details.

<table>
<thead>
<tr>
<th>Use of <code>gets/sets</code></th>
<th><code>id</code></th>
<th><code>Q.id</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>field access</td>
<td>field access</td>
</tr>
<tr>
<td>Interface</td>
<td>field access</td>
<td>field access</td>
</tr>
<tr>
<td>Method</td>
<td>field access</td>
<td>field access</td>
</tr>
<tr>
<td>Constructor</td>
<td>field access</td>
<td>field access</td>
</tr>
<tr>
<td>Field</td>
<td>field access</td>
<td>field access</td>
</tr>
</tbody>
</table>

**Table 5.3:** Interpretation and support for capturing communications, for `gets` and `sets` event designators, by each kind of boundary. Formal arguments are not permitted in the communication descriptions for `gets` and `sets` event designators, so no columns are shown for those kinds. **Boldface** entries indicate that capture of the given communication kind is fully supported at the given boundary. Roman entries indicate that capture of the given communication kind is partially supported at the given boundary; see the main text for details.
ilarly, in-mapping communications of the form `SomeClass.doit()` on the boundary around a class named `SomeClass` results in “`SomeClass`” being stripped from the communication; thus, the entry in the column marked “`id(args)`” should be considered. However, if we in-map communications of the form `SomeClass.doit()` on the boundary around a method named `doit()`, no stripping occurs and the entry in the column marked “`Q.id(args)`” should be considered.

This stripping process does not have an analogue for out-mapping, where a fully-qualified name cannot be interpreted until the outermost boundary named is encountered. For example, a reference to `String` cannot be interpreted as equivalent to a reference to `java.lang.String` until import declarations make this interpretation unavoidable. Since import declarations are not found until a communication has passed an outer class boundary, there is no means to gradually resolve such a qualified name.

Any communication descriptions matching a table entry that is unsupported results in the tool signalling an error. Communications consisting of just a type name do not occur in Java, so they are unsupported; these correspond to the entries in the “—” column for classes and interfaces. Similarly, the only communications that should cross the boundary of a field, method, or constructor from the outside to the inside are those that name that field, method, or constructor, respectively; all others do not occur in Java. Field accesses cannot be in-mapped in Java, because the Java Virtual Machine does not permit code to be executed when a field is accessed [Lindholm and Yellin, 1999: §3.11.5 & §3.11.8].

The interception of field accesses and method or constructor invocations is partially supported (as indicated in Tables 5.2 and 5.3) because I require that such communications actually cross the boundary. For example, a communication that is sent directly by a method to itself is not considered to have crossed the boundary of that method. This is a subtle point: if a method `doit()` contains an invocation `doit()`, it is considered to be sending itself a communication directly. In general, a name reference is considered to cross a boundary if it does not resolve to a declaration within that boundary. The name resolution process is described below. The remaining forms of capturing communications are partially supported due to the way in which communication capture and alteration is implemented; this is discussed in Section 5.6.

**Name Resolution**

Declarations are referenced through names; however, a given name can refer to different declarations depending upon the scope in which the name occurs. Therefore, we need a set of rules for determining how to resolve and match names. The following process is used

---

12 In-mapping of a field can be simulated by out-mapping all accesses to that field (i.e., code can be added at every location where the field is referenced)—a potentially difficult prospect, since it requires that all source code making such accesses be available to the tool.
for resolving the names in boundary maps and source code.

Names are resolved with respect to a given boundary; only the declarations inside
the boundary are used to resolve the names there. The following code is an example of a
complete compilation unit\textsuperscript{13} that I will use to illustrate name resolution.

```java
package somepackage;

import java.util.Vector;

public class SomeClass extends AnotherClass {
    private Vector set = new java.util.Vector();

    public SomeClass() {}

    public Object doSomething(SomeClass other, SomeClass set) {
        if (other.set.elementAt(0) == this.set.elementAt(0))
            return set.elementAt(0);
        return null;
    }

    private Object elementAt(int index) {
        return set.elementAt(index);
    }
}
```

There are three sites in this source code that reference the set field; but each is quali-
ified differently: one is qualified by the local variable other, one by the keyword this,
and one is unqualified. Consider the following capture clauses:

1. out(set)

2. out(SomeClass.set)

The expressions that the tool determines as matching each of these capture clauses varies.

If a boundary map containing either of these capture clauses is attached to the bound-
ary of the entire class, none of the three sites in the source code is considered to match.
Each communication description resolves to a reference to the field declaration that occurs
within the boundary; therefore, these communications would not cross the class boundary.

\textsuperscript{13}A compilation unit consists of the contents of a single Java source file. In general each Java source file can contain the declaration of only one class. Some minor exceptions exist.
In the case of boundary maps attached to the boundary around the doSomething method, the sites captured by each of these capture clauses is different. The expression other.set resolves to SomeClass.set since other is declared as a formal parameter of type SomeClass within the method boundary. The expression this.set resolves to set, since this is no longer needed to disambiguate the field reference from the formal parameter reference. Therefore, capture clause #1 would match only the second expression in doSomething, this.set, while capture clause #2 would match only the first expression, other.set.

Similarly, capture clauses of the forms:

3. out(java.lang.Vector.elementAt(int))

4. out(Vector.elementAt(int))

would capture different expressions if contained in a boundary map attached to the class boundary. Capture clause #3 would match no expressions within this class; it is not until we encounter the import statement that we know that Vector resolves to java.lang.Vector and this happens outside the class boundary. Capture clause #4 matches three expressions, though: the two within the equality test of the if-statement, and the one within the body of the elementAt method.

### 5.3.2 Parameterized Boundary Maps

The body of a boundary map often needs to make use of the objects passed within the communications captured by it. To enable this, boundary maps can be specified with formal parameters; these formal parameters are then bound to arguments within the communication descriptions of the boundary map's capture clause. For example, for communications sent out of a module to the Vector.elementAt(int) method and captured, one might be interested in making use of the instance of Vector upon which the method is called, the value of the int argument passed in this call, or both or neither of these. All four of these options can be expressed by boundary maps:

Object map(Vector v): out(v.elementAt(int)) { ...}

Object map(int pos): out(Vector.elementAt(pos)) { ...}

Object map(Vector v, int pos): out(v.elementAt(pos)) { ...}

Object map(): out(Vector.elementAt(int)) { ...}
The body of the boundary map can then make use of these formal parameters identically to how a method uses its formal parameters [Gosling et al., 2000: §8.4.1].

The order of the formal parameters does not matter. Thus, the third example above could equivalently be:

```java
Object map(int pos, Vector v): out(v.elementAt(pos)) { ...
```

The tool currently requires that each formal parameter to a boundary map have its name occur exactly once within the capture clause.\(^\text{14}\)

The tool resolves names occurring in the capture clause as much as possible before matching it to communications occurring in the source code. Names are resolved against the formal parameters declared by the boundary map; thus, in all five examples listed above, the communication pattern that is resolved is `Vector.elementAt(int)`. The matching process then proceeds as described in Section 5.3.1.

### 5.3.3 Resuming Captured Communications

Boundary maps capture particular communications. Often, we wish to add functionality to the execution of the methods receiving these communications rather than replacing the original functionality outright. However, directly re-calling the method that has been captured causes a recursive call that is itself captured by the boundary map. Thus, we need a means to avoid the resulting infinite loop. This means is the `call to proceed`, indicated by the expression:

```java
CONTEXT.proceed(args)
```

The `args` are a comma-separated list of arguments of identical type and order as the formal parameters of the boundary map. The result type of the call to proceed is that of the captured communication had it not been captured. BNF syntax for calls to proceed is given in Figure 5.6.

For example, consider the following boundary map. It captures all attempts to read elements from Vectors, ensuring that a read lock is in place before the access happens and that the read lock is released after the access happens.

```java
Object map(): out(Vector.elementAt(int)) {
    LockManager.getReadLock();
    // Allow vector access to proceed
    Object obj = CONTEXT.proceed();
}
```

\(^{14}\)This constraint can be relaxed. If the name of a given formal parameter occurs two or more times within the event designator, it implies that the objects passed in these positions must be identical. This is a minor feature, and so I have left it for future work.
Figure 5.6: BNF syntax for calls to proceed. The Java syntax for primary expressions (indicated by the non-terminal PRIMARY) [Gosling et al., 2000: §15.12] has been extended to include calls to proceed. The other non-terminals in slanted text correspond to existing Java syntax.

```
PRIMARY ::= call-to-proceed
        | PRIMARY-NO-NEW-ARRAY
        | ARRAY-CREATION-EXPRESSION

call-to-proceed ::= CONTEXT "." proceed "(" [ARGUMENT-LIST] ")"
```

LockManager.releaseReadLock();
return obj;
}

The functionality of this particular boundary map is not concerned with the actual Vector instance on which the access is happening nor with the value of the int being passed to this method. To do this in the absence of a call to proceed would require a different boundary map, one that exposed these values to the body:

```
Object map(Vector v, int pos): out(v.elementAt(pos)) {
    LockManager.getReadLock();
    // Allow vector access to proceed
    Object obj = v.elementAt(pos);
    LockManager.releaseReadLock();
    return obj;
}
```

But this makes the code more complicated, runs the risk that we may inadvertently alter the values of v and pos, and causes an infinite loop because the call to Vector.elementAt within the boundary map body is recursive, and thus is captured by the boundary map as well.

If we do wish to replace the Vector object and int value passed in the call to elementAt, we could use the following boundary map.

```
Object map(Vector v, int pos): out(v.elementAt(pos)) {
    LockManager.getReadLock();
    Vector special = getSpecialVector();
    Object obj = CONTEXT.proceed(special, pos + 1);
```
In this example, I have changed the access attempt to have it made on a different vector than the one originally called, and the position at which the access is made is one greater than the original. The call to proceed must be provided with arguments of the same types and same order as occur in the formal parameters of the boundary map.

The use of the call to proceed places an onus on the system. Boundary maps can be used to effectively introduce methods to a class that does not already exist. Should such a boundary map make a call to proceed, there is no pre-existing implementation that should be executed. There are two possible options in such a situation: ignore the call to proceed, or signal an error. The tool does the latter.

5.3.4 Modifiers

Boundary maps translate communications between differing world views. Constructor, method, and field declarations can all possess modifiers. Therefore, to be a true translation, boundary maps need to cope with any difference in which modifiers are perceived to affect behaviour.

The boundary map should be declared as possessing the set of modifiers expected by the calling context, and should translate these into the set of modifiers provided by the called context. For example, say that the method doit is declared only with the modifier private. If the world view of the calling context expects that doit is public, static, and synchronized then the boundary map should be declared public, static, and synchronized while being implemented to translate this view to the reality that doit is only private. There are many issues in permitting or limiting this sort of power to change visibility that lie beyond this dissertation.

Currently, the prototype tool only supports the static modifier on boundary maps (see Figure 5.7 for BNF syntax); issues arising from their use are described in the subsection below. However, some visibility modifiers can be simulated with the tool; these are described in the subsection after next.

The static Modifier

Methods and fields in Java can be declared as operating either in the context of a particular instance, or without such a context. The former case is the default; the latter is indicated by 

\[ \text{LockManager.releaseReadLock();} \]
\[ \text{return obj;} \]

\[ \}

15 Other techniques and languages, such as AspectJ [Kiczales et al., 2001], support method introduction.
the presence of the `static` modifier on the declaration of the method or field. Likewise, boundary maps can be declared as `static`.

Specifying a boundary map without a `static` modifier implies two things. First, the communication being captured must possess an instance context; for an out-map, this is the instance context from which the communication was sent. Second, this captured instance context can be used within the body of the boundary map via the keyword `this` just as is the case with method bodies. Specifying a boundary map with a `static` modifier indicates that no instance context is required of the communications captured by the boundary map. Consequently, the keyword `this` cannot be used within the body of such a boundary map; its presence will cause a compile-time error.

A non-`static` boundary map cannot capture communications that do not provide an instance context. A `static` boundary map can capture communications that provide an instance context; this instance context will be discarded. This discarding procedure is consistent with the operation of Java itself [Gosling et al., 2000: §15.12.1].

**Simulating Visibility Modifiers**

The prototype tool ignores issues of the standard visibility modifiers. For example, it does not take into account the fact that calls to `private` members cannot cross class boundaries, so any in-mapped method is considered `public`.

Consider the following simple class:

```java
public class Simple {
    private void doit() {}
}
```

We may in-map calls to the `doit` method, with or without adding behaviour there. This effectively causes `doit` to appear to be `public` from the perspective of the external context. So, for example, we might specify a boundary map to be attached to `Simple` as:

```java
void map(): in(doit()) {
    CONTEXT.proceed();
}
```

The tool adds a method to `Simple` that invokes the original implementation of `doit`; thus, this added method can access all the `private` methods of `Simple`. The external context would then see the declaration of `Simple` as:
public class Simple {
    public void doit() {}
}

Translation of visibility modifiers can be mimicked to a degree. Such mimickry is useful when we have two independent world views where a given method is considered private in one world view but public in the other. If we wished to make the method doit visible only within the boundary, we could in-map calls to it in the following way:

    void map(): in(doit()) {
        throw new IllegalAccessError();
    }

Instances of the class IllegalAccessError are thrown by the Java Virtual Machine when an attempt is made to invoke a method from a context that is not permitted access to it; this implementation mimics that behaviour. This approach is not totally satisfactory, as modules containing calls to doit will still compile; only at run-time will they fail.

Out-mapping can similarly be used to alter effective interfaces. Imagine that we have a class Client that calls doit:

public class Client {
    public void someMethod() {
        Simple simple = new Simple();
        simple.doit();
    }
}

We can out-map this call to mimic the effect of a public modifier on doit:

    void map(): out(Simple.doit()) {
    }

If we wished to have this out-map make a call to proceed, though, problems would occur because of the way boundary maps are implemented by the tool. Boundary maps are implemented within the scope of the boundary to which they apply. The in-map is implemented within the scope of the Simple class and, thus, is able to access the doit method even though this is private. The out-map is implemented within the scope of the Client class and, thus, doit remains private from its perspective. This behaviour is not a deliberate design. Visibility control remains an issue for future work.
5.3.5 Communication History

Communication history is, conceptually, a complete record of every piece of information passed between modules. The prototype tool deals strictly with method and constructor invocations, though. These are recorded in two fashions: on a causal basis, in terms of a tree-structure consisting of calls and returns for each thread of control within an execution; and on a temporal basis, irrespective of causality. A means of querying communication history on either basis is supported by the prototype tool, through an interface on a specially provided class called History and its attendant helper, Call. For each invocation, the parameters passed can be retrieved through the appropriate queries. This is meant literally: in Java, objects are passed by reference; therefore, the reference that has been passed can be retrieved, not necessarily the original state behind that reference.

There are several methods defined on History for querying communication history, as listed in Table 5.4; these are all static methods. Most of these allow the thread, class, method name, and method formal parameter types to be optionally specified; null arguments are treated as though any value in the corresponding position matches. All these methods return an instance of Call if a valid match is found, or null otherwise. A sample method, lastCall, is shown and described in Figure 5.8. The methods defined on the Call class are shown in Table 5.5. The complete interface to the History and Call classes is described in Appendix D.

To examine the operation of communication history and its query interface, consider the following toy program.

```java
public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        test.printIt(1);
        test.printIt(2);
        aMethod();
        test.printIt(3);
    }

    public void printIt(int number) {
        System.out.println(number);
    }

    public static int aMethod() {
        Test otherTest = new Test();
        otherTest.printIt(2);
        return 0;
    }
}
```
Method | Search kind
-------|-------------
firstCall | either
lastCall | either
lastCallAnySubclass | either
lastCallInCFlow | causal
lastCallReturningClass | either
lastInstancePassed | temporal
mostRecentCall | temporal

Table 5.4: The query methods defined on the History class. Each method searches causally or temporally. Those methods marked as “either” have forms that can operate in either of these fashions. The interface is described in detail in Appendix D.

<table>
<thead>
<tr>
<th>Result type</th>
<th>Method name</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>getMethodName()</td>
</tr>
<tr>
<td>java.lang.Thread</td>
<td>getThread()</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>getReturnValue()</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>getTarget()</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>getTargetClass()</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>getParameter(int i)</td>
</tr>
<tr>
<td>Call</td>
<td>getPredecessor()</td>
</tr>
<tr>
<td>Call</td>
<td>getParent()</td>
</tr>
<tr>
<td>boolean</td>
<td>precedes(Call other)</td>
</tr>
</tbody>
</table>

Table 5.5: The interface to the Call class. The interface is described in detail in Appendix D.
lastCall

```java
static Call lastCall(Thread thread,
                      Class cls,
                      String mth,
                      String[] paramTypes,
                      Object returnVal,
                      Call relativeTo)
```

Searches for and returns the most recently occurring call conforming to the search parameters passed as arguments; returns `null` if no call can be found that matches these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>The thread of control whose tree should be searched, in the case of causal searches; or <code>null</code> if temporal searches should be performed</td>
<td>current thread</td>
</tr>
<tr>
<td>cls</td>
<td>The exact class on which the executed method was defined; or <code>null</code> if the class is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>mth</td>
<td>The name of the method on which the executed method was defined; or <code>null</code> if the method is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>paramTypes</td>
<td>An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or <code>null</code> if the parameter types are not important</td>
<td>—</td>
</tr>
<tr>
<td>returnVal</td>
<td>The object that must have been returned by the method execution; or <code>null</code> if the returned object is not of interest</td>
<td><code>null</code></td>
</tr>
<tr>
<td>relativeTo</td>
<td>The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters</td>
<td><code>mostRecentCall()</code></td>
</tr>
</tbody>
</table>

**Figure 5.8:** A sample query method on the `History` class. The slanted formal parameters are optional; that is, the `lastCall` method is overloaded with forms missing these parameters, in all combinations. If an argument is not provided that corresponds to an optional formal parameter, the default value indicated in the table is used.
Executing this program causes `Test.main` to be invoked initially. A series of numbers is printed to the standard output in the order: 1, 2, 2, 3. The communication history just before the program terminates is shown in Figure 5.9.

We can see the effects of communication history by applying the following mapset to the boundary of `main`:

```java
mapset findIt {
    static void map(): in(main(String[])) {
        CONTEXT.proceed();

        String[] params = new String[] { "int" };
        Call call =
            History.lastCall(Test.class, "printIt", params);
        call =
            History.lastCall(Test.class, "printIt", params,
                call.getPredecessor());
        call = call.getParent();

        System.out.println(call.getMethodName() + ": " +
            call.getReturnValue());
    }
}
apply findIt to Test.main(String[]);
```

This boundary map begins by calling to proceed with the original functionality of `main`. It then retrieves the most recent call to `printIt`, which can be seen from Figure 5.9 to be the lowermost call on the diagram. Tracing back over the sequence of dashed arrows, the boundary map finds the call that occurs most recently prior to this one, which is the rightmost call on the diagram. The parent of this second most recent call to `printIt` is retrieved by tracing back over the solid arrow. Finally, the boundary map prints the name of this parent and the result it returns, as:

```
aMethod: 0
```

At the time of developing the boundary map, we may know that the call we are interested in retrieving is the last call to `aMethod`. In this case, a simpler query to communication history could be used in the boundary map:
Figure 5.9: An example communication history. The diagram represents the actual data structure used by the tool, so even though, for example, it makes no sense to have a return value when the return type is `void`, there is always a slot for both. The solid arrows indicate the causal relationships between the calls; these arrows lead from the callers to the callees with later callees lower on the diagram. The dashed arrows indicate the temporal ordering of the calls, leading from earlier to later calls. Since the execution of `main` has not finished, its result is not recorded (represented as “in progress” in the diagram—a garbage value). Calls to constructors have been left off the diagram. The target objects are actually stored as references; the names `test` and `otherTest` are shown here for readability.
mapset findIt2 {
    static void map(): in(main(String[])) {
        CONTEXT.proceed();

        String[] params = new String[0];
        Call call =
            History.lastCall(Test.class, "aMethod", params);

        System.out.println("aMethod: " +
            call.getReturnValue());
    }
}

This locates the same, ultimate call of interest and produces the same output as the earlier version.

I chose to provide the communication history functionality in this class-based fashion for the sake of ease of development. It being initially unclear how communication history was to operate and be maintained, I decided that a class-based approach would allow maximum flexibility with minimum changes needed to the tool as the research progressed. The interface that has been implemented for communication history is, by no means, complete. For example, there are currently no query methods that allow one to specify that a call should be found that contains a particular object as an argument, and there is currently no method for determining the number of parameters on a returned call. Section 5.6.2 discusses the drawbacks of the current implementation approach, and Section 7.2 describes how an improved version of the tool should implement it to avoid such drawbacks.

5.3.6 Renaming Maps

The boundary maps described thus far are based upon the notion that communications are captured as they cross boundaries, and that an operational description of the alteration of these communications can be specified. The pragmatics of Java make the use of this model difficult for altering communications involving type references, such as static method invocations (e.g., capturing the reference to System within an invocation to System.loadLibrary). Type references are not first class entities in Java, so replacing one type reference with another is not possible procedurally. Although Java provides support for introspection, one cannot easily transform a static type reference embedded within a potentially complex expression into a different expression that uses introspection. Instead, the prototype tool provides a second kind of boundary map: renaming maps.

A renaming map specifies a target name to be matched and a replacement name that is inserted in its stead. For example, if we wished to replace all references to java.lang.
System, within a boundary, with a reference to SomeClass, we would use the following renaming map:

```
java.lang.System -> SomeClass;
```

This would replace any explicit occurrences of the string "java.lang.System," and hence, implicit references through the string "System" will not match. That is, name resolution does not occur.

In general, the tool allows any type name (including exception types) to be replaced with any other type name (although array types are not supported). However, Java syntax is ambiguous: type names and field accesses have identical syntax. If a name does not resolve at a given boundary, we often cannot tell whether it refers to a type or a field. Therefore, the tool considers any unresolved name to be a fair target of renaming maps. BNF syntax for renaming maps is provided in Figure 5.10.

Renaming maps do not, by themselves, ensure the type compatibility of all expressions upon which they act. A local variable whose type is altered by a renaming map may have assigned to it an expression whose type is not assignable to the replacement type. For example, consider the following source method and renaming map:

```
public void someMethod() {
    String s = "test string";
}
```

```
String -> mypackage.String;
```

The body of someMethod, after applying the renaming map, effectively becomes:

```
mypackage.String s = "test string";
```

However, this statement is illegal because an instance of `java.lang.String` is not assignable to a variable of type `mypackage.String` since the former is not a subtype of the latter. If the tool allowed assignment operations to be captured and altered (which it does not), the assignment could presumably be replaced; otherwise, the resulting module will not compile.
This behaviour is peculiar to java.lang.String since it is the only class for which literals are defined within Java. I did not consider this significant enough of a problem to deal with in the prototype. More generally, one would need to assure that the types of both sides of assignments were compatible. The tool does not currently flag such constraint violations, but allows the compiler (used after the tool has produced transformed source) to catch them. The presence of such incompatibilities means that the translation between world views internal and external to the module is incomplete.

5.4 Tool Directives

Tool directives are statements telling the tool which mapsets to attach to which boundaries, like the following:

```
apply someMapSet to SomeClass;
```

Each tool directive consists of two parts: the name of the mapset (someMapSet, in this case) to apply to a boundary, and the name of the boundary itself (SomeClass). BNF syntax for tool directives appears in Figure 5.11.

The prototype tool allows three kinds of boundaries to be named within tool directives: whole classes, individual methods and constructors, and individual fields. This is not to say that implicit context is fully supported at all these boundaries; see Section 5.3.1 for more detail. If the tool is unable to locate the named boundary within the files provided to it, it prints a warning to this effect.

Tool directives reside in the global namespace. This means that mapsets and boundaries must be referenced within tool directives via global names, i.e., they must be fully qualified. For example, the class JavaOutlinePage of Eclipse occurs in the package org.eclipse.jdt.internal.ui.javaeditor; thus, it must be referred to as org.eclipse.jdt.internal.ui.javaeditor.JavaOutlinePage within a tool directive.

Fields are referenced by appending their names to the fully qualified name of their defining class.

Methods also are referenced by appending their names to the fully qualified name of their defining class, but are more problematic because of the formal parameter list of each. To match a method, the tool lexically matches the types of the formal parameters as specified within the source and within the tool directives; no resolution of the types found
in either formal parameter list is performed prior to matching. For example, consider this simple example of Java source:

```java
package somepackage;

import otherpackage.SomeClass;

public class AnotherClass {
    public AnotherClass(SomeClass arg) {}
    public void doit(SomeClass arg) {}
}
```

and this tool directive:

```plaintext
apply someMapSet to somepackage.AnotherClass.
doit(otherpackage.SomeClass);
```

Although the tool would search within AnotherClass here to find the doit method, it would not find any matches since the lexical declaration of doit says that it takes a formal parameter of type SomeClass and not otherpackage.SomeClass. The tool only uses the information provided within the files passed to it, but it does not assume that the information contained there forms a closed universe. Therefore, I chose the parameter type matching rule to avoid the need for disambiguating so-called on-demand import statements, such as

```java
import otherpackage.*;
```

In the presence of this alternative statement, SomeClass could resolve to SomeClass, otherpackage.SomeClass, somepackage.SomeClass, or even java.lang.SomeClass—even though the import statement says that SomeClass resolves to otherpackage.SomeClass.

Constructors are handled similarly to methods. The name of a constructor is the same as the unqualified name of its class. For example, the constructor shown above for AnotherClass would be referenced in a tool directive as:

```plaintext
somepackage.AnotherClass.AnotherClass(SomeClass)
```

Since tool directives are separated from the modules they affect, I chose to place tool directives in the global namespace; placing them inside particular source modules seems inappropriate. This choice has additional consequences: not all boundaries are easily named from the global namespace. For example, Java possesses constructs such as local classes whose names are not well-defined outside their local scopes, and anonymous classes which
have no names at all. While we can define an arbitrary rule to construct their names (such as a numbering scheme based on their lexical order within a globally named scope), these resulting names tend to be unstable: if we merely reorder the source code, the names will change. Similar problems would arise should we choose to support sub-method-level boundaries, such as boundaries around individual statements. It is possible that specifying boundaries through operational means—defining properties or conditions that need to be met—would allow such problems to be overcome. However, it is not clear at this point whether separate boundaries around such difficult-to-name modules are needed.

5.5 Sequencing Boundary Maps

More than one boundary map can readily be defined that attempts to capture a given communication. In these situations, it is important to have an explicit sequence in which boundary maps are to apply. A total order of boundary maps is defined for the prototype tool in the following fashion: of greatest precedence is the order in which the files containing tool directives are specified to the tool, followed by the order of the tool directives within a particular file, followed by the order of boundary maps within a mapset named by a tool directive.

The tool blindly applies the boundary map of greatest precedence to the source before considering where or how the second most precedent boundary map is to apply. The source that is input to the tool for matching against the second boundary map is the output of the first.

For example, consider the following source, mapset, and tool directive.

```java
public class SomeClass {
    public void method1() {
        System.err.println("In method1()");
    }

    private void method2() {
        String s = "test string";
    }
}

mapset example {
    void map(): in(method1()) {
        System.err.println("Before method1()");
        CONTEXT.proceed();
    }
}
```
void map(): in(method1()) {
    System.err.println("Ignoring method1()"');
}

String -> mypackage.String;

String -> java.lang.String;
}

apply example to SomeClass;

The tool would apply the first in-map on method1 first, followed by the second in-map on method1. As a result, calls to method1 would result in the string "Ignoring method1()" being printed. Next would come the first renaming map, which would result in the type of the local variable s becoming mypackage.String. Finally, the second renaming map would be applied, but would not find any matches for "String" since the first renaming map has already replaced these.

5.6 Implementation Issues

The previous sections within this chapter have largely described the features of the prototype tool from the perspective of a system developer using it. However, this leaves the question of how the tool actually performs its tasks, a question that leads to subtle properties of behaviour to which I have, so far, merely alluded.

Recall that every boundary map possesses a body that can contain fairly arbitrary Java source code. Once a boundary map has captured a communication, its body indicates how that communication is to be altered. However, this is a model that must operate in the context of the realities of Java, where one cannot literally intercept communications (especially to non-existent methods) as they cross boundaries, and where it would be very inefficient to actually do so. Instead, as indicated in Section 5.2, the tool performs a multi-stage process before compile-time on the three kinds of inputs provided it.

I begin, in Section 5.6.1, by examining how the tool combines boundary maps and Java source to simulate the interception of communications crossing boundaries. Section 5.6.2 looks at instrumentation of the resulting source to support communication history, and the effects that sequencing multiple boundary maps has upon this support.
5.6.1 Combining Boundary Maps and Java Source

The tool combines specified boundary maps with the source within specified boundaries. The way it goes about this depends on two things: the kind of boundary and the kind of boundary map to be combined. I begin by looking at the combination process for in-maps, followed by the combination process for out-maps (including gets and sets event designators). Table 5.1 indicates the support provided by the tool for various combinations of kinds of communication descriptions and boundaries.

The in-mapping of method invocations results in two possible situations: either the method is already present, or it is not. If it is already present, I change the name of the method to hide it and any references to that method within the boundary are changed to the new name of the method. In both cases, the body of the boundary map is inserted as the body of a new method, with the original name of the replaced method. The result type and context-kind of this new method are taken from the result type and context-kind of the boundary map; after all, the purpose of the boundary map may be to alter these properties to the perspective of a different world view. The new method is inserted at the level of the closest enclosing class, even when the boundary map is applied to a method boundary.

In-mapping constructors is more complicated, because we are stuck with the name of the constructor. Instead, we can play with the number and types of parameters that the constructor takes; dummy parameters can be inserted to effectively hide the original constructor, with appropriate dummy arguments inserted in calls to it within the boundary. This has not been fully implemented in the tool.

Attempts to capture qualified method or constructor invocations can be problematic. When applied to the boundary of a type, they are interpreted as method or constructor invocations on nested types. This is done as in the non-qualified case as long as the nested types are already present. I deemed it too complicated to introduce new classes correctly for it to be worthwhile doing so in the prototype; it is not clear from whence some details about the new class would come.

Out-maps are generally more straightforward. Again, the body of the boundary map is inserted as a new method with an obfuscated name. All call sites (or field access sites) within the boundary matching the communication description are replaced by calls to the new method.

Out-mapping constructors has a few more twists to it. In Java, there are two components to an invocation of a constructor. First, space is allocated to the new instance via use

---

16The new name consists of the prefix “...CONTEXT$inner” followed by a number, which starts at 0 for the first changed method, 1 for the second, etc.

17One cannot embed a method directly inside of another method in Java—it is not clear what the semantics of such an embedding would be.

18Here, the name consists of the prefix “...CONTEXT$map” followed by a number.
of the new operator, then this space is initialized through the actual constructor invocation. If we were to only capture the latter component, we would need to constrain our out-map to ensure that it consisted of only a different constructor invocation. Instead, the tool allows for the capture of the new operator itself, through a special syntax. To capture constructor invocations of the form new somepackage.SomeClass(args), one describes the communication as somepackage.SomeClass.new(args). Each entire instantiation expression within the boundary is replaced in this situation.

If the boundary map body contains a call to proceed, this must be replaced with an invocation of the original method (via its new name, in the case of in-maps) or original constructor, or with an access of the original field, where the communication arguments that are exposed to the boundary map body are replaced with whatever new values are in place and the other arguments are unchanged. If no such original target exists, the tool signals an error.

The tool implements renaming maps by replacing all the occurrences of a name within a boundary with a different name in a straightforward fashion.

5.6.2 Instrumentation, Communication History, and Sequencing of Boundary Maps

In order to reflect upon the history of communications made within a system, we need both a means to record the communications made within that system, and a means to access this record. My proof-of-concept implementation of communication history for Java stores method calls and method returns within a data structure consisting of one tree per thread of control, and a doubly-linked list through the nodes in all these trees to indicate the temporal order of the events therein. Interfaces to this data structure are provided for the system developer to utilize communication history; see Section 5.3.5 for details.

To store calls and call returns in the tree, I defined two snippets of code to instrument the methods in a system, one that is executed at the start of each method and one that is executed at the end of each method. These snippets need to worry about operating correctly in the presence of exceptions, and thus, tend to be long and ugly; Appendix E shows a simple example in all its gory detail.

Every method in the classes operated on by the tool is instrumented. This means that every method invocation causes information to be stored in the communication history record, and it is stored for the entire duration of the execution of the system. This is obviously inefficient both in terms of execution speed and memory requirements. There are many cases in which it is fairly simple to detect when such information is not needed and thus does not need to be stored in the first place, or situations in which the information is no longer needed and can be discarded. These ideas are discussed further in Chapter 7.

More subtle problems arise from the fact that instrumentation does not occur until after
all boundary map transformations have occurred, and that only method entries and exits are
instrumented, not the call sites themselves. Consider the following source, mapset, and tool
directive.

```java
public class SomeClass {
    public static void main(String[] args) {
        OtherClass obj = new OtherClass();
        for(int i = 0; i < 3; i++)
            obj.method1();
    }
}

public class OtherClass {
    public void method1() {
        method2();
    }
    public void method2() {
        String s = "a string";
        System.err.println(s.length());
    }
}

mapset example {
    void map(): in(method1()) {
        Call call = History.lastCall(null, "method2", null);
        if(call == null)
            CONTEXT.proceed();
        else
            System.err.println("method2() called before");
        Call call2 = 
            History.lastCall(String.class, "length", null);
        System.err.println(call2 != null);
    }
    void map(): in(method2()) {
```
Implicit context is being used here (through the first boundary map) to change the behaviour of the system such that method2 is invoked at most once. The second boundary map is present simply to illustrate the problem discussed in the next paragraph; conceptually, its effect should be idempotent.

Before implicit context was applied, running SomeClass would have caused the following to be printed:

```
8
8
8
```

We would expect that, after applying implicit context, this would become:

```
8
true
method2() called before
true
method2() called before
true
```

However, this will not happen under the prototype tool. Instead, this will be printed:

```
8
false
8
false
8
false
```

Appendix E shows the code that results from the application of the tool on this example. There are two problems, as described below.

The in-map on method2 causes it to be renamed to `__CONTEXT$inner1` and the call site within method1 is changed to this new name. A new method is inserted in OtherClass called method2 that delegates to `__CONTEXT$inner1`, replacing the call to proceed within the boundary map. But when the resulting class is
instrumented, the method that is getting invoked each time is not called method2 but 
_CONTEXT$inner1—we have changed the name out from under our communication 
history query. This is why “method2() called before” is never printed. This prob-
lem is solved by having the tool differentiate between original and mapped names better. 
Which name to use in a given situation depends on the world view(s) under consideration, 
and the tool needs to take this into account.

The other problem arises from the fact that the call site within method2, invoking
String.length(), is not instrumented and the source for String itself has not been
fed into the tool. Therefore, the query to find the last call on String.length() always
fails, returning null. This problem is simple to fix: at a minimum, one could instrument
every call site. Better solutions exist (see Chapter 7).

All these difficulties arise from the simplistic implementation of communication his-
tory that I chose. This sufficed for my proof-of-concept investigations (see Chapter 6), but
does not suffice for a non-prototype version of the tool. A stronger implementation of com-
munication history will be based on the discussion in Chapter 7; such an implementation
would address the problems outlined above.

5.7 Summary

A system developer must describe four things in specifying and using my prototype tool:
(1) the Java source code on which implicit context is to operate; (2) to which boundary or
boundaries it must be attached; (3) which communications it is to intercept; and (4) how
it should alter the intercepted communications. Items 3 and 4 are provided by the bound-
dary map construct; multiple boundary maps are organized into named mapsets. Item 2 is
provided by tool directives, which name mapsets and the boundaries to which they are to
apply.

The prototype tool performs a multi-stage process before compile-time on the three
kinds of inputs provided it (namely, tool directives, boundary maps, and original Java
source):

1. The boundary maps and original Java source are parsed and stored as abstract syntax
trees (ASTs).

2. Each tool directive is parsed to identify the mapset and boundary named in it. These
names are matched against the ASTs.

3. Assuming that the mapset and boundary matching the names within the tool directive
are found, each boundary map within the mapset is applied to the boundary, in the
lexical order in which they occur. After each transformation, the transformed source
AST is used as the input for the next transformation.
4. After all the tool directives have been carried out, the transformed source AST is instrumented to support communication history queries.

5. The transformed source AST is unparsed to produce a new set of Java source, ready for compilation.

Boundary maps consist of: a capture clause, describing the set of communications to be intercepted at the boundary to which the boundary map is attached; a body, specifying how the intercepted communications are to be altered; a set of formal parameters, bound to parts of the intercepted communications and exposed for use within the body; and various other minor parts, such as a result type, and optional `throws` clause and/or `static` modifier.

Communication history can be used within boundary map bodies. It is supported through a set of methods on two special classes, called `History` and `Call`. Communication history is supported by instrumenting the transformed source code in order to record every method entry and exit. This simplistic approach has several drawbacks, including impact on the speed of execution, and ever-increasing memory demands the longer the execution continues. I consider solutions to these problems and to the other shortcomings of the tool in Chapter 7.
Chapter 6

Validation

I now consider the validation of the hypothesis that expressing essential structure through implicit context eases software evolution and software reuse. I performed a set of case studies to provide evidence in support of several properties involved in easing software evolution and reuse.

Good design can go a long way to easing software evolution and reuse. But “no matter how ambitious the intentions of the designers . . ., software designs tend to erode over time” [van Gurp and Bosch, 2002]. State-of-the-art design and implementation practices that do provide a measure of evolvability and reusability come at the cost of explicitly anticipating and planning for specific kinds of change. Implicit context attempts to allow us to do away with much of this cost, focusing on immediate system needs, not potential future ones.

The changes that are actually needed in our systems over time often differ from those that we anticipate. State-of-the-art design and implementation practices generally handle unanticipated change poorly, requiring invasive modifications of the details of our systems—a risky and difficult prospect. Implicit context helps us to cope with unanticipated change by specifying modifications external to our modules.

State-of-the-art design and implementation practices also tend to increase the complexity of individual modules for the sake of greater overall flexibility, as described in Chapter 2. Implicit context allows us to avoid this complexity in our local modules, moving it to more global scopes where it can be manipulated directly as the system evolves. In other words, implicit context allows us to express the local concerns of each module locally, separating the more global concerns.

At an early stage of the work on the ideas in this dissertation, Walker and Murphy [2000] performed a case study and published its results. It demonstrated that implicit context could be used to restructure a system to isolate a particular module from an unwanted feature, while continuing to have the system as a whole provide that feature. Specific-

1Invasive modifications involve making a change within the body of a module, as opposed to externally specifying a change.
cally, the pluggable look-and-feel feature of the Java Swing graphical user interface library [Fowler, 1998] was removed from the JButton class, which provides the functionality of a button widget. This case study addressed the concept of essential structure in only its most nascent form; as such, I do not discuss it further within this dissertation.

For this dissertation, I performed two case studies, each dealing with a subset of the properties discussed above; each case study involved systems written in the Java programming language [Gosling et al., 2000]. The first case study, described in Section 6.1, focused on whether unanticipated reuse can be handled more easily through the use of implicit context. I removed the Outline View of Eclipse (a large, industrially-developed integrated development environment) and reused it in another system, independent of the rest of Eclipse. Such reuse was not planned for in the design of Eclipse.

The second case study, described in Section 6.2, focused on whether unanticipated evolution can be handled more easily through the use of implicit context. I designed and implemented two versions of a server providing a subset of the file transfer protocol (FTP), one version using a purely object-oriented approach and one version using implicit context as well; these designs ignored the existence of the remainder of the FTP specification. Each version was then evolved to determine whether fewer invasive changes were necessary when implicit context was used. The purely object-oriented version was additionally evolved via implicit context; this demonstrates that implicit context can be used to evolve a system not constructed with it from scratch, while still providing benefit.

These two case studies described below concentrate on structural issues bearing on the evolvability of our systems and the reusability of our modules. For implicit context to be adoptable as an industrial development technique, additional factors must be addressed, such as the development process to be used, tool design, and run-time cost. While further research involving a non-prototype tool is needed, the case studies provide some insights into these factors.

### 6.1 Reusing the Outline View of the Eclipse IDE

This case study concentrated on demonstrating the use of implicit context in the arena of unanticipated reuse. I began with an existing, large, well-designed system: the Eclipse integrated development environment. I chose to reuse a feature of Eclipse called the Outline View (see Figure 6.1); the functionality provided by the Outline View is described in Section 2.1.

Eclipse was designed and implemented to be highly extensible, utilizing such state-of-the-art techniques as object-orientation [Booch, 1986; Rumbaugh et al., 1991], reflection [Smith, 1982], and design patterns [Gamma et al., 1994]. However, Eclipse was not designed in a way that planned for such reuse of the Outline View. The Outline View was
Figure 6.1: The Eclipse integrated development environment. The Outline View is in the sub-window on the right.

not modularized in a fashion amenable to reusing it, possessing too many dependences on classes within Eclipse that I did not want to reuse (see Section 2.1 for details). Therefore, state-of-the-art practices were unable to aid us in this task without requiring extensive, invasive modifications to Eclipse—the very thing I wanted to avoid.

Instead, I used implicit context to create a facade around the Outline View. From the perspective of the interior of the Outline View, it continued to operate within the rest of Eclipse. From the perspective of the external world, the Outline View was turned into a GUI widget (in a class called JavaViewer) independent of the remainder of Eclipse. I then used this GUI widget as the beginnings of an interface to the prototype tool described in Chapter 5; this interface currently allows one to browse through the declarations in a set of source classes and to generate very basic skeletons of boundary maps. The procedure that I followed in accomplishing this task is described in Section 6.1.1. Section 6.1.2 discusses the results of the case study.
6.1.1 The Procedure

The procedure that I followed in reusing the Outline View outside Eclipse consisted of the following, major steps.

1. Identify the classes implementing the Outline View.
2. Determine the dependences these classes have on the remainder of Eclipse.
3. Create the JavaViewer class to act as a facade to the Outline View, replacing the dependences the reused classes had on the remainder of Eclipse.
4. Utilize JavaViewer as a widget in a graphical user interface to my prototype tool.

I describe each of these steps in turn.

Identifying Classes for Reuse

Eclipse contains no class called "OutlineView"; therefore, some effort needed to be expended to determine the classes that implement the Outline View. I began by performing a search for any class whose name contained the words "Outline" or "View." Examining each of the classes that matched this search, I eliminated those that clearly had no relationship to the Outline View. Of the remainder, two stood out as appearing significant on the basis of their names and content: JavaOutlinePage and ContentOutline. I then determined the superclass hierarchies of each of these classes, and examined the comments contained within the source corresponding to all these classes.

Eclipse contains a number of "views" (such as editors, task lists, and the Outline View) that operate together to form the environment for system development. The ContentOutline class provides the Outline View, as evidenced by the comments contained in it, but only in coarse form. ContentOutline refers to the actual contents displayed within it in an abstract form; this allows the contents to be specialized, for example, on the basis of the programming language in which development is being done. This is the role of the JavaOutlinePage class.

This description is further complicated by the fact that each view really acts as a placeholder (with certain semantic properties) for GUI widgets. Thus, PageBookView, which is the superclass for ContentOutline, contains a PageBook, which is a kind of GUI widget used for displaying one of a set of pages at a time. In turn, PageBook holds instances of Page, which is a superclass of JavaOutlinePage.

I decided that the majority of the functionality of the Outline View is implemented in a total of seven classes within Eclipse: JavaOutlinePage, ContentOutline, PageBookView, ViewPart (the superclass of PageBookView), WorkbenchPart (the superclass of ViewPart), PageBook, and OverlayIcon (responsible for drawing
the icons shown for each node of the tree within the Outline View. In addition to these
were several classes that made small contributions to the functionality of the Outline View.

Eclipse is built on top of a custom graphics package, called the Standard Widget
Toolkit (SWT), instead of the graphics package provided by the standard Java class li-
braries, the Abstract Windowing Toolkit (AWT). Eclipse also makes heavy use of a GUI
library, called JFace, built on top of the SWT; the equivalent GUI library provided by the
standard Java class libraries is called Swing. While it should be possible to translate any
program written on top of SWT/JFace to instead use AWT/Swing, this would be a signifi-
cant undertaking. I felt that this translational process would require too much effort for too
little additional evidence for it to be worthwhile within the scope of this case study. There­
fore, I chose to reuse the SWT and JFace as libraries: i.e., they were not altered in any way,
either manually or by the tool.

Determining the Dependences on Eclipse

Determining the dependences that my reused classes had on the remainder of Eclipse was
an iterative process. I isolated the classes that I intended to reuse and attempted to com-
pile them. The compiler complained any time it could not resolve a reference to types and
their members that were not present. From the error messages generated by the compiler, I
created a set of types lacking any functionality so that the compiler could resolve these ref-
erences. Eventually, I refined this set of dummy types to the point where the compiler would
compile the classes I was reusing, implying that I had determined the set of dependences
on the interface to the remainder of Eclipse.²

Of course, mere compilation does not imply correct behaviour. I proceeded to de-
termine the more subtle dependences: which of the types and their members performed
functionality I considered useful in the context of my application, and what protocols were
expected to be followed to have the Outline View function. Determining these subtle de-
pendences was an ad hoc process involving two operations: running the compiled system
and seeing where it broke, and examining the original system to determine the protocols for
initializing and using the Outline View. I describe this more below.

Creating the JavaViewer Facade

To provide an interface for the external world to use the Outline View in isolation from the
remainder of Eclipse, I created a class called JavaViewer. JavaViewer operates in the
style of other widgets provided by SWT/JFace. Its public interface includes methods that

²This process is equivalent to forming declaratively complete hyperslices in the Hyper/J tool
(see http://www.research.ibm.com/hyperspace/ for details of the tool, and Tarr et al.
[1999] for details of the concepts it uses).
perform the following behaviour: creation of the actual widget controls that are displayed;
setting the input model that is represented by the tree within the Outline View; getting and
setting the title of the shell in which the Outline View is displayed; getting and setting the
set of nodes in the tree of the Outline View that are selected; and adding new action buttons
to the tool bar of the Outline View. To implement this behaviour, JavaViewer contains
an instance of JavaOutlinePage and an instance of ContentOutline; many of the
methods described above simply delegate to existing methods on these classes.

In order for the interface provided by JavaViewer to suffice, I had to replace all
of the dependences on the parts of Eclipse I decided against reusing. Most of these de­
pendences implemented functionality required by Eclipse overall but not required by my
application; therefore, contextual dispatch was used to "dead-end" these, i.e., intercepting
outgoing calls and having them immediately return, with trivial result values such as null.
The non-trivial dependences that remained fall into three categories: additional Outline
View functionality, translation of initialization protocols, and translation of the Java source
element representations. I discuss these below.

Recall that I decided to reuse a set of classes that implemented a majority of the func­
tionality of the Outline View; the remainder was scattered amongst other classes. Rather
than reuse these other classes, which would have pulled along much unwanted functionality,
I reimplemented the small remaining pieces in two classes: MemberPage and JavaEl­
ementLabelProvider. The names of these two classes are identical to existing classes
in Eclipse; since they fulfill the same roles as these existing classes, from the perspective of
the Outline View, this choice seemed reasonable.

Since the Outline View was never designed to be a widget, its implementation assumes
various details of initialization at run-time have already occurred prior to its instantiation.
These details are taken care of by parts of Eclipse not reused in my application and, hence,
had to be dealt with through a small set of boundary maps.

The most significant remaining dependences dealt with the representation of elements
in Java source code. The Outline View provides a tree view of the elements in Java source,
such as methods, fields, and classes. Eclipse provides a particular set of classes for repre­
senting these source elements. These classes are intimately tied-in with other functionality
of Eclipse, which deals with source editing and compilation. Thus, I decided that these
classes representing source elements were too specific to Eclipse for a general-purpose
Outline View widget. Instead, I created a separate, more generic hierarchy of classes for
representing source elements. The world view of the reused classes had to be translated to
make use of this new source element hierarchy.

At this point, I had obtained a module providing the functionality of the Outline View
that could be used independent from the remainder of Eclipse. Figure 6.2 shows the classes
in this module.
Figure 6.2: The module resulting from the reuse of the Outline View. The classes with names in boldface text are those that were reused verbatim from Eclipse. The classes with names in Roman text represent classes and interfaces I implemented to replace like-named ones in Eclipse. The classes with italicized names are new. Dependences denoted with dotted arrows are those that arise from the boundary maps attached to many of these classes. Note that dependences on SWT, JFace, and the standard Java class libraries are not shown.
Using JavaViewer as an Interface to the Tool

Under normal circumstances, I could use the Outline View module I created simply by instantiating JavaViewer and making the appropriate calls to its interface. However, the circumstances that I encountered in using the Outline View as an interface to my prototype tool were special.

My module, providing an isolated Outline View, makes use of a generic hierarchy representing Java source elements. My prototype tool produces abstract syntax trees (ASTs) the nodes of which also represent Java source elements. To use JavaViewer as an interface to my tool, one would need to dynamically translate these ASTs into a tree representation using the source element hierarchy provided by the JavaViewer. This should not be difficult, but it does seem wasteful.

Instead, I deemed it appropriate to use implicit context to alter the world view of the JavaViewer so that it directly used the source element hierarchy already provided by my tool. In this case, the tool would not need to perform a translation to and from its ASTs, but could hand these ASTs directly to the JavaViewer.

Figure 6.3 shows the results: a JavaViewer instance displaying the major source elements from an AST provided by the parser of my tool. I created a special feature for this JavaViewer through a new class called TemplateMapAction. The user of the interface can select a set of nodes in the tree; they can also bring up a pop-up menu by right-clicking in the JavaViewer window. TemplateMapAction adds an option to this pop-up menu that reads, “Create in-map templates.” Selecting this option causes a skeleton mapset to be written to a temporary file, and a text editor displaying this file to be started. The user can then add in their specifications for contextual dispatch to in-map the selected nodes. This is simple functionality, but a step towards interactive tool support for using implicit context.

![Figure 6.3](image-url)

*Figure 6.3:* The Outline View of Eclipse reused as an interface to my prototype tool (see Chapter 5). The contents of the window consists of a JavaViewer widget.
6.1.2 Results and Lessons Learned

I successfully reused the classes implementing the Outline View of Eclipse as a simple interface to my prototype tool. Table 6.1 shows statistics for the case study. Only a small fraction of Eclipse was reused (17% if we include the SWT and JFace in the calculation, or 0.3% if we do not) indicating that I reused the Outline View without pulling along much extraneous functionality.

Tool Limitations

Unfortunately, a very small number of invasive modifications were needed due to limitations in the prototype tool. There were three kinds of these, involving package and imports statements and implements clauses. In designing the prototype tool, I had deliberately chosen to ignore issues arising from these statements and clauses for the sake of limiting the scope of the project.

The package name for the reused modules needed to be commented out to avoid namespace collisions between the original, unmodified source and the output of the prototype tool. Similarly, import statements needed to be commented out when non-reused classes were imported by the original code. A better implementation of the tool should sup-

<table>
<thead>
<tr>
<th>Description</th>
<th>Source size (loc)</th>
<th>Boundary maps all (loc) (#)</th>
<th>Boundary maps non-trivial (loc) (#)</th>
<th>Boundary maps renaming (loc, #)</th>
<th>Tool dirs. (loc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse, total</td>
<td>Java: 775,314 C: 11,121</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWT &amp; JFace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new, facade</td>
<td>1,231</td>
<td>666</td>
<td>181</td>
<td>214</td>
<td>23</td>
</tr>
<tr>
<td>new, interface</td>
<td>128</td>
<td>466</td>
<td>53</td>
<td>442</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 6.1: Statistics on Eclipse and reusing the Outline View. The rows give statistics, respectively, for all of Eclipse, including for the SWT and JFace; the SWT and JFace only; the Outline View classes I reused; the new classes, etc. that I added in creating the Java-Viewer facade; and the additional classes, etc. that I added to the facade to use it as an interface to my prototype tool. Numbers represent either number of lines of source code (labelled “loc”), including blank and comment lines, or number of instances (labelled “#”). Statistics for non-trivial boundary maps and renaming maps are given in addition to the statistics for all boundary maps combined. Trivial boundary maps are ones that intercept messages and immediately return with the most obvious default result, such as null for reference types, false for boolean type, etc. Each renaming map occupies one line of code.
port management of the two "versions" of the source in a completely transparent manner.

In addition, when a reused class declares that it implements an interface that is not to be reused, I had two options: I could comment out the implements clause, or I could provide a dummy interface to replace the original. Neither option is very satisfactory. A better implementation of the tool would provide a means to ignore an unwanted implements clause.

Determining Classes to Reuse

A major difficulty I had in reusing the Outline View was simply to figure out which classes within Eclipse implement it. Because of the choices of class decomposition made within Eclipse, the functionality that we can point to on a display as being "the Outline View" is scattered amongst many classes. This was a perfectly reasonable decision on the part of the designers of Eclipse, since they were designing it to be extensible; the behavioural abstractions in place within this design serve these extensibility requirements. However, to ease software reuse through implicit context, we need to know what classes we wish to reuse in the first place; the model does not speak to how to determine this. Thus, going from the position of possessing no knowledge of Eclipse to the position of having sufficient knowledge to attempt to apply implicit context required approximately 6 weeks of effort. This time would have been shorter if the various documentation then available on Eclipse had spoken more to the existing design of Eclipse rather than to how one can use its extensibility features.

After the point of knowing enough about Eclipse to attempt to reuse some of its classes, I only required approximately 1 week to successfully apply implicit context. This includes the time required to locate and correct a few significant errors in my prototype tool.

Automated Tool Support

Some automated tool support for generating boundary maps would have aided in this case study. As pointed out in Table 6.1, a large number of the boundary maps that I created and applied were of a trivial nature. Trivial boundary maps are ones that intercept messages and immediately return with the most obvious default result, such as null for reference types, false for boolean type, etc. An automated tool that could generate trivial maps for any outgoing, unresolved dependences would have saved me considerable effort. Furthermore, simply providing automated tool support to determine what the unresolved dependences are for a given boundary would have helped enormously. Both of these forms of support would be straightforward to create, building upon the functionality that the prototype tool already possesses for resolving names at boundaries (see Section 5.3.1 for details of this process).
Organizing and Ordering Boundary Maps

Organizing and ordering boundary maps proved to be a significant challenge. Most boundary maps performed minor transformations; as such, it was difficult to have a clear mental picture of a given module before and after one of these small transformations.

I typically created one mapset for all of the dependences on a given class that was not being reused; hence, there was one mapset for eliminating references to IJavaElement, one for JavaEditor, and so forth. As described in Chapter 5, the prototype tool applies boundary maps in the order in which they are specified to the tool. Since each transformation is quite small, getting this order correct was difficult at best, and sometimes not possible. Consider that a given module might contain the following line of source:

\[ A.m1().m2(); \]

and that \texttt{A.m1} is not declared within that module. Let us also say that we need to non-trivially map calls to \texttt{A.m1} and \texttt{B.m2}. The details of and order in which we apply our boundary maps matters. If we were to attempt to remap calls to \texttt{B.m2} first, the line above would not resolve at all and the boundary map would not apply here. If we were to remap calls to \texttt{A.m1} first, through a boundary map where the result type were \texttt{B}, then the attempt to remap \texttt{B.m2} would match the second call here. Even more subtle effects arise when we alter the result types of messages through our boundary maps; some of these are described in Section 5.6.2.

Many of these troubles could be averted through two provisions. First, boundary maps that are applied "simultaneously," as opposed to being explicitly ordered, should not be arbitrarily applied but applied in the lexical order of the source that they affect. For example, in the statement shown above, boundary maps affecting \texttt{A.m1} should be applied prior to boundary maps affecting \texttt{B.m2}.

Second, specifications should be allowed that complete partial world views. In the example above, part of the trouble stems from the fact that, within the boundary, we cannot determine the type of \texttt{A.m1} until a boundary map is attached that affects this. Instead, we should permit the inclusion of statements specifying such type information, much as \texttt{import} statements do within Java currently.

Communication History

In this case study, I utilized a total of 11 queries on communication history—6 queries to construct the facade, and an additional 5 queries to construct the interface to the prototype tool. Four of the 11 queries were involved in translating calls on the Java source element hierarchy from those appropriate to Eclipse to those appropriate to the facade, such as checking whether a given method is \texttt{static} or a whether a given field is \texttt{private}. 

113
Two were involved in handling the initialization of the Outline View for the facade. One translated the Eclipse protocol for looking up the currently displayed page. And the remaining four were involved in translating calls on the Java source element hierarchy from those appropriate to the facade to those appropriate to the prototype tool.

The main problem encountered here was for the first four queries involved in translating between source element hierarchies. The prototype tool does not currently instrument call sites. This can lead to the apparent disappearance of calls that one is interested in retrieving from communication history, as described in Section 5.6.2. I dealt with this through a kludge involving the insertion of a dummy method. The call site is changed to call the dummy method, and the dummy method delegates to the method originally called. Communication history could then be queried for calls on this dummy method. Future implementations of the tool should deal with call sites directly.

An example mapset using communication history is shown in Figures 6.4 and 6.5; it is used for translating between the source element hierarchies of Eclipse and the facade. Java source element hierarchy classes in Eclipse, such as IMember, represent their modifiers (e.g., static) as a set of flags encoded in an int. This int could be retrieved via the getFlags method; the presence of individual modifiers encoded in this int could then be tested through static methods provided on the class Flags, such as isStatic. Within the facade, the encoding of modifiers was to be hidden and the tests for modifiers directly supported by the individual classes of the source element hierarchy, such as proto.hierarchy.Member.

Four boundary maps needed to be applied to any class testing for the presence of modifiers (namely, JavaOutlinePage). First, any attempts to retrieve the int encoding the modifiers had to be dead-ended, returning "0." But ultimately, one will need to query communication history regarding these attempted calls, so the kludge described above needs to be used: the call to getFlags is intercepted and replaced with a call to getFlags_dummy. This dummy method is introduced into the boundary through the second boundary map. The third boundary map replaces references to the type IMember with references to the type proto.hierarchy.Member. And the fourth boundary map changes the test for the static modifier via the Flags class to a direct call on the isStatic method on proto.hierarchy.Member. But this latter call invokes an instance method, meaning that the appropriate instance has to be found. This instance is retrieved via communication history, where the last call to getFlags must have been called on the instance of interest.

**Overall**

Overall, the case study demonstrates an ability to reuse a chunk of an existing system that was not designed to enable such reuse, while avoiding invasive modifications to the reused
// FlagsMaps
// Translate the Java source element hierarchy modifier
// flags (e.g., static, private) from the Eclipse format
// to the facade format.
// Eclipse returns an encoding of all flags via getFlags()
// in which we can test for individual flags, such as:
// int flags = someMember.getFlags();
// boolean isStatic = Flags.isStatic(flags);
// The facade provides methods to test for individual flags
// directly.
//
mapset FlagsMaps {
    // Delegate calls to getFlags() to a dummy method. The
    // returned int encodes the flags.
    static int map(IMember member): out(member.getFlags()) {
        return getFlags_dummy(member);
    }
}

...  

// Provide a dummy method to dead-end calls to getFlags();
// just return 0.
static int map():
    in(getFlags_dummy(IMember)) {
        return 0;
    }

...

(continued in next figure)

Figure 6.4: An example mapset performing a communication history query, Part 1 of 2. This mapset translates the tests for modifiers on the Java source element hierarchy from those appropriate to Eclipse to those appropriate to the facade.
// Rename IMember (from Eclipse) to proto.hierarchy.Member
// (from the facade).
IMember -> proto.hierarchy.Member;

...

// Replace attempts at testing for the static flag on the
// int returned by getFlags() with direct calls.
static boolean map(): out
(Flags.isStatic(int)) {
    // We need to find the call from whence the flags being
    // tested came. Do this by finding the last call to
    // getFlags[_dummy]().
    call flagsCall =
        History.lastCall(Thread.currentThread(), null,
            "getFlags_dummy", null);

    // The first parameter passed to this call will be the
    // source hierarchy member.
    proto.hierarchy.Member member =
        (proto.hierarchy.Member)flagsCall.getParameter(0);

    // Now perform the direct test on this member.
    return member.isStatic();
}

...
modules in the process (other than a few due to minor limitations in the prototype tool). There is no way we could have accomplished this kind of reuse through existing approaches, as discussed in Chapter 2.

The largest question remains that of when it becomes more expensive to reuse modules than to reimplement them. At some point, extant modules will be such a poor fit for the new context to which wish to move them that they would not truly meet our needs. Where this point lies is very sensitive to details of the process and the domain in which we work. In this case study, it was expensive to determine the classes I wished to reuse, but had I already possessed background knowledge of Eclipse, this cost would have been reduced. On the other hand, implementing a module with the functionality of the Outline View would have required a learning effort as well. Finding the correct tradeoffs awaits further research.

6.2 Evolution of a File Transfer Protocol Server

This case study concentrated on demonstrating the use of implicit context in the arena of unanticipated evolution. I constructed parallel versions of a File Transfer Protocol (FTP) [Postel and Reynolds, 1985] server using different approaches and evolved each in ways not catered for by the original design. I outline some high-level FTP concepts of importance in Section 6.2.1.

There were two stages to the case study. In the first stage, I selected a subset of the features described in the FTP specification. I designed and implemented two versions of an FTP server providing this subset of features—one using standard object-orientation, and the other using object-orientation plus implicit context. Both of these versions ignored the presence of the remainder of the FTP specification.

In the second stage, I selected a subset of the remaining features of the FTP specification, and evolved my two FTP server implementations to incorporate these new features. Each of the evolved versions continued to use the development approach it used for the initial stage. Additionally, I wished to demonstrate the efficacy of implicit context in easing the evolution of a system that was not initially constructed with it. Therefore, I took the initial version of my purely object-oriented FTP server and "cross-evolved" it, i.e., I evolved it using implicit context.

In Section 6.2.2, I describe in more detail the procedure that I followed in the case study. The results and lessons that I learned from the case study are discussed in Section 6.2.3.

6.2.1 FTP Concepts

The File Transfer Protocol uses two kinds of connections to communicate between client and server, control connections and data connections. The server listens at a given port on
a given machine for connection attempts by clients; successful attempts establish a control connection.

The client sends messages to the server consisting of a command and zero or more arguments as appropriate for the command. The server processes each message and responds with a series of one or more reply codes indicating the status of the requested operation along with any return data. Each control connection is stateful and some commands must be sent in an order prescribed by the protocol to complete successfully.

Some commands (notably RETR to retrieve files from and STOR to store files to the server) establish a data connection for transferring large amounts of data. This data connection is established by the server to a default port on the client; a PORT command can be issued to select a different client-side port. A server can be told to passively await the establishment of a data connection, rather than establish it itself, through the PASV command; this command returns the server-side port at which the server awaits the connection. A data connection may be transient, shutting down upon the completion of an operation; this depends upon the requests made and the implementation of the server. Both the control connection and data connection are closed at the successful reception of the QUIT command, if they remain open.

Other commands can be issued to modify the way the data passed in the data connection is interpreted. For example, the TYPE command is used to indicate whether data is to be interpreted as ASCII or binary (called “image” type), the MODE command is used to indicate whether the transmission mode is as a stream of bytes or in blocks, and the STRU command is used to indicate whether the data is structured as files or records.

Users can be identified via the USER command and authenticated via the PASS (password) command.

Within the FTP specification (RFC 959 [Postel and Reynolds, 1985]), there is a specification of the Minimum Implementation that a server must provide in order to be a bona fide FTP server: the commands USER, QUIT, PORT, TYPE, MODE, STRU, RETR, STOR, and NOOP (which does nothing) must be recognized. Note that PASS and PASV are not required by the Minimum Implementation. The protocol provides for additional commands not of importance here.

6.2.2 The Procedure

The procedure that I followed in developing and evolving the FTP server consisted of the following, major steps, as illustrated in Figure 6.6.

1. Design and implement two versions of the FTP server meeting the requirements for the Minimum Implementation of RFC 959, one using only object-oriented techniques, and one using object-orientation plus implicit context.
Figure 6.6: Evolving versions of the FTP server, from the initial, Minimum Implementation to include the authentication and passive mode features. The circles represent versions of the server; the arrows represent tracks of evolution.

2. Select a subset of the remaining features from RFC 959.

3. Evolve the designs and implementations of each version, using its original development approach, to add the new features.

4. Evolve the design and implementation of the purely object-oriented version of the server from Step 1 using implicit context to add the new features.

I describe each of these steps in turn.

Developing the Minimum Implementations

In developing the two versions of the Minimum Implementation, I was concerned that I provide a fair comparison between the two approaches. On the one hand, I did not want the design of the object-oriented version to be influenced by the ideas of implicit context. On the other hand, I did want the object-oriented version to be well-designed.

To deal with these concerns, I took two precautions. First, I completely designed, implemented, and tested the object-oriented version prior to developing the version utilizing implicit context. Second, I had several colleagues critique my design for the object-oriented version, altering it accordingly. Below, I first describe the design of the object-oriented version and then the design of the version using implicit context.
Object-oriented version. The UML class diagram for the object-oriented design is shown in Figures 6.7 and 6.8.

An instance of Server runs listening for connection attempts by clients on a particular port. When a client connects to the server, an instance of Session is created. In turn, the Session object creates one instance each of ControlConnection, DataConnection, and TransferContext. The Session object is used to provide a convenient handle on the current control connection, data connection, and transfer context.

Each ControlConnection runs in its own thread. A ControlConnection listens at its port for incoming request messages from the client. Received requests are sent through a chain of objects for parsing and action. The Interpreter encapsulates the knowledge that the initial three or four letters of the message specify the FTP command, while the remainder consists of command-specific arguments. The CommandFactory returns an instance of one of the Command subclasses as appropriate to the specified command. Each Command subclass encapsulates the knowledge of the syntax of the arguments specific to that command. Thus, Interpreter strips the command out of the message and asks CommandFactory for the corresponding Command instance. If such an instance exists, Interpreter hands it the remainder of the message for parsing and action. One or more responses are generated along this chain and handed to the ControlConnection to be sent to the client; each response is encoded as an instance of a subclass of Reply. Each Reply subclass encapsulates the knowledge of its corresponding reply code, used in formatting the response sent over the control connection.

The DataConnection object encapsulates the knowledge of the data connection protocol appropriate to the current session state, such as the file structure and transmission mode. For example, this protocol may require that the actual data connection be closed after each file transfer and this fact needs to be communicated to the client.

The TransferContext object contains the state specific to the session, such as the address and port of the client and representation type.

This system is reasonably well-designed. This design uses a number of design patterns [Gamma et al., 1994] including Interpreter, Command, Factory Method, and Singleton. The design encapsulates specific pieces of knowledge about the File Transfer Protocol (such as the encoding of reply codes, the argument syntax for particular commands, and the data format used across the data connection) in different classes, so that each might be changed in isolation from the rest of the system. Subclassing is used so that additional classes can be added (for any new commands, say) without the need for much of the rest of the system to change. For example, ControlConnection can treat any instance of Reply through the same, abstract interface without worrying about its specific subclass. Each class has a well-defined purpose.
Figure 6.7: Initial design for the purely object-oriented version of the FTP server, Part 1 of 2. This diagram shows only the static features of the design, and does not include dependencies other than those arising from subclassing and the existence of fields.
Figure 6.8: Initial design for the purely object-oriented version of the FTP server, Part 2 of 2.
Implicit context version. A modified UML class diagram of the design using implicit context is shown in Figures 6.9 and 6.10. Many of the classes are surrounded by dotted lines; these represent boundaries. Attached to these are mapsets, shown as rounded boxes.

An instance of Server runs listening for connection attempts by clients on a particular port. When a client connects to the server, an instance of Session is created, running in a separate thread.

Session provides the basic structure of the FTP server architecture: it instantiates ControlConnection and CommandInterpreter, and then runs in a continuous loop, reading lines from the ControlConnection and having the CommandInterpreter interpret them—at least, this is the world view within Session. In reality, the SessionArchitecture mapset ignores the attempt to instantiate CommandInterpreter and intercepts attempts at interpreting the received message, rerouting them to CommandFactory.

CommandFactory strips off the initial four characters of the received message (which encode the name of an FTP command) and locates the corresponding instance of a subclass of CommandInterpreter. The arguments to the command are then passed to this instance for further interpretation.3

Each ControlConnection is much simpler in this design than in the purely object-oriented one. A ControlConnection only reads from its socket or writes to its socket when it is explicitly told to do so. ControlConnection knows nothing of the rest of the system.

DataConnection is similarly simple, knowing only how to read or write files. Extra pieces of behaviour are added to it through the DataConnectionArchitecture mapset: the files being read and written must be filtered according to the current representation type; and, the data connection must be set up according to the remote address and data port for the client.

Each subclass of CommandInterpreter is responsible for parsing the arguments to one of the FTP commands; exceptions are thrown if the arguments do not conform to the correct syntax. The mapset attached to each command interpreter intercepts any thrown exceptions and sends replies in their place. It also intercepts other outgoing messages, inserting additional communication with the client as required by the Protocol for each command; for example, the RETR command requires that the client be notified when the data connection is about to be opened. The ReplyMarshal mapset around Session is

---

3The knowledge of the encoding of the command within the message, and the knowledge of the correspondence between a particular four-letter sequence and a given subclass of CommandInterpreter, are intimately connected but different. Therefore, CommandFactory could have been designed as a module containing two smaller modules, one for each of these two different pieces of knowledge. This situation is similar to the example of reducing makeContributions on page 31.
Figure 6.9: Initial design for the implicit context version of the FTP server, Part 1 of 2.
Figure 6.10: Initial design for the implicit context version of the FTP server, Part 2 of 2.
responsible for actually translating the Reply objects into messages and handing them to the ControlConnection.

Since the use of the Singleton design pattern is merely an optimization within the system, I chose to design all the classes to treat the Reply and RepresentationType subclasses as though they were not singletons. Instead, attempts to instantiate any Reply or RepresentationType subclass were intercepted and replaced with a call to the first-created instance of that subclass.

The remaining mapsets performed minor adjustments to allow us to do such things as deal with exceptions, small name adjustments, and the finer points of stream-based communication over network sockets, such as buffering for efficiency.

My implementation expressed the essential structure of the modules in this version of the FTP server. The most basic, and least likely to change, features of FTP were encapsulated in these modules. More flexible details of protocol have been separated and added externally.

**Selecting New Features**

RFC 959 describes various commands and features not required of a Minimum Implementation. Some of these are simple additions, accommodated by adding new subclasses to the hierarchy of commands; we would expect an object-oriented design to handle these with ease. Others are not so straightforward, requiring modifications to multiple parts of the system. I selected two of these more problematic features, authentication and passive mode.

Authentication involves ensuring that a user is who they claim to be. FTP provides for authentication through its PASS command, through which a user can provide a password. Most FTP commands require that the user be authenticated prior to their usage, the exceptions being USER, QUIT, and PASS itself. FTP implicitly specifies a state machine for authentication, shown in Figure 6.11.

The server is normally responsible for establishing the data connection. That is, the server creates a socket and connects to a particular port on a remote machine. In passive mode, the server waits, listening at a specific port for the client or other server to establish the data connection. Passive mode thus affects any functionality involving the data connection. The state machine involved in passive mode is shown in Figure 6.12.

**Evolving the Versions to Add the New Features**

I evolved my initial designs to add the authentication and passive mode features. I used purely object-oriented techniques to evolve the object-oriented version, and a combination of object-orientation and implicit context to evolve the other version. I describe, in turn, the
changes required to each.

**Object-oriented version.** The changes to the design of the purely object-oriented version, resulting from the addition of the authentication and passive mode features, are shown in Figures 6.13 and 6.14; new classes and altered classes are highlighted in grey. Five new subclasses of Reply were also added in a straightforward manner; thus, I do not show the Reply hierarchy for the evolved design.

Authentication required a number of additions to the TransferContext class. Previously, I had not been bothering to store the user name passed in the USER command because FTP did not require its use within the Minimum Implementation. In my evolved design, I added fields to record the user name and whether the user is logged in. I also added setter and getter methods for this state.

A PasswordCommand was added to the Command hierarchy to parse the arguments to the PASS command. A class called Authenticator was created to perform the actual
Figure 6.13: Evolved design for the object-oriented version of the FTP server, Part 1 of 2.
validity check on a username/password pair. PasswordCommand uses Authenticator to perform this validity check, storing the results in the current TransferContext.

The UserCommand had to be altered to store the arguments passed in the USER command into the current TransferContext object. Its reply to the client was also changed from “Command OK” to “Password needed.”

Every other command, save QUIT, must be authenticated prior to its operation. Therefore, every subclass in the Command hierarchy had to be altered to check that the user had been logged in. If not, a reply of “User not logged in” had to be sent to the client. For most commands, this involved the insertion of a few lines of code at the beginning of the corresponding perform method.

To add passive mode, the DataConnection class required extensive modifications. First, a method and a field had to be added to allow passive mode to be set. The passive-Socket field records the socket used to listen for attempts by the client or other servers to establish the data connection. The send and receive methods had to be modified to test for passive mode. If passive mode was set, the passive socket was listened at. Once the data connection was established, the passive socket was closed, causing the server to return to active mode.

I added a PassiveModeCommand to the Command hierarchy. This class parses the arguments to the PASV command (none are permitted) and tells the current DataConnection instance that passive mode is set.

CommandFactory was altered to store an instance of each of the two new subclasses of Command, just as it stores an instance of all the original subclasses of Command.
Implicit context version. The changes to the design of the version using implicit context, resulting from the addition of the authentication and passive mode features, are shown in Figures 6.15 and 6.16; new classes and altered classes are highlighted in grey. As with the object-oriented version, there were five subclasses of Reply also added that are not shown.

Authentication required the introduction of the PasswordCommandInterpreter class to parse the arguments passed for the PASS command, plus the attendant Password-Protocol mapset to return a standard reply to the client. Beyond this, authentication was enabled through the application of three mapsets: UserNameAuthentication, PasswordAuthentication and AuthenticationCheck. Each of these mapsets expressed a portion of the state machine shown in Figure 6.11 as described below.

UserNameAuthentication is the simplest of these mapsets. We can see from the state machine that any occurrence of a USER command results in the system transitioning to the state where it awaits a password. Therefore, this mapset merely intercepts the reply “Command OK” sent from the UserNameCommandInterpreter class and replaces it with “Password needed.”

The other two mapsets require us to determine which state the session is in. For the session to be in an authenticated state, two properties must hold. First, the most recent
Figure 6.16: Evolved design for the implicit context version of the FTP server, Part 2 of 2.
occurrence of the USER command must have been immediately followed by an occurrence of the PASS command. Second, the password sent as an argument to this PASS command must be valid for the user name sent as an argument to this USER command. Finding the most recent occurrence of the USER command requires a communication history query to find the most recent call for the current thread to the perform method on the UserNameCommandInterpreter. If the next call to a perform method on any subclass of CommandInterpreter happens to be to PasswordCommandInterpreter, we know that a PASS command immediately followed the USER command. The communication history queries can retrieve the user name and password arguments passed to these methods. The validity check is performed by a method on the Authenticator class, added for this purpose.

The PasswordAuthentication mapset captures the “Command OK” reply issuing from the PasswordCommandInterpreter, and follows the procedure above to determine if the session is authenticated. If it is, the captured reply is replaced with “User logged in”; if it is not, the captured reply is replaced with “Not logged in.”

The AuthenticationCheck mapset intercepts requests to perform any command except USER, PASS, or QUIT. It checks that the session is authenticated using the procedure above. If it is, the intercepted message is called to proceed; if it is not, the reply “Not logged in” is sent to the client.

The passive mode feature required the addition of a PassiveModeCommandInterpreter, to parse the arguments to the PASV command (none are permitted), with its attendant PassiveModeProtocol mapset to translate exceptions into the proper reply messages. The CommandAction mapset was modified to intercept the “Entering passive mode” reply sent from the PassiveModeCommandInterpreter. The real reply needs to contain the address and port number of the socket on which the server will be listening for the establishment of the data connection. This information is added to the reply through communication history queries.

Finally, the DataConnectionArchitecture mapset needed a boundary map added to it to intercept attempts to create a socket. If the session is in passive mode, the server listens for the data connection to be established; otherwise, it calls to proceed with the socket creation.

Cross-evolution: Adding Features to the OO Version through Implicit Context

Finally, I took the initial design and implementation of the version using only object-orientation and added the authentication and passive mode features to it with implicit context; the changes are shown in Figure 6.17.

Two new Command subclasses were added, PasswordCommand and PassiveModeCommand, and an Authenticator class to test username/password pairs for va-
Figure 6.17: Cross-evolved design of the FTP server. Much of the design that remains unchanged is not shown, nor are the five added Reply subclasses.
lidity, as in the evolved object-oriented version of the system. Likewise, CommandFactory was altered as before, and five subclasses of Reply were added that are not shown in the diagram.

The mapsets UserNameAuthentication, PasswordAuthentication and AuthenticationCheck were effectively the same as in the evolved implicit context version of the system.

The biggest difference between the cross-evolved version and implicit context version of the system lies in the use of the PassiveMode mapset. PassiveModeCommand was implemented as though it were communicating directly with the DataConnection. When DataConnection attempts to create a socket, this attempt is intercepted to see if the session is in passive mode. If it is, the DataConnection listens at a socket for the data connection to be established; if it is not, the message to create the socket is called to proceed.

6.2.3 Results and Lessons Learned

Table 6.2 shows some statistics for the case study. The total number of lines of code for the versions using implicit context tends to be slightly greater than the number of lines of code for the versions using only object-orientation. This is an artifact due to two limitations

<table>
<thead>
<tr>
<th>Approach</th>
<th>Source size (loc)</th>
<th>Boundary maps all (loc)</th>
<th>Boundary maps renaming (loc, #)</th>
<th>Tool dirs. (loc)</th>
<th>Invasive mods. (#)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial version</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pure OO</td>
<td>1,404</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OO + IC</td>
<td>848</td>
<td>909</td>
<td>120</td>
<td>39</td>
<td>29</td>
</tr>
<tr>
<td><strong>Evolved version</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pure OO</td>
<td>1,722</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OO + IC</td>
<td>941</td>
<td>1,169</td>
<td>133</td>
<td>57</td>
<td>29</td>
</tr>
<tr>
<td>Cross</td>
<td>1,602</td>
<td>196</td>
<td>34</td>
<td>10</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 6.2: Statistics on synthesizing and evolving the FTP server. Numbers represent one of: number of lines of source code (labelled “loc”), including blank and comment lines; number of instances (labelled “#”); number of queries to communication history (labelled “# CHQs”); or, number of classes that were modified (labelled “# cls”). Statistics for renaming maps are given in addition to the statistics for all boundary maps combined. Each renaming map occupies one line of code. The number of invasive modifications to the code that were required to evolve the server are shown in the rightmost column; each deletion or insertion of a consecutive set of lines of code is counted as one modification.
of the tool, a lack of wildcarding and a lack of direct support for subtyping; both these limitations lead to the need to duplicate a significant number of boundary maps and tool directives. When the numbers are modified to eliminate such duplicates, the total number of lines of code for each approach are roughly equal.

**Invasive Modifications**

More significant to this case study are the number of invasive modifications that needed to be made to each version to accommodate the new features. The purely object-oriented approach required a total of 16 invasive modifications among 11 classes. The two approaches that evolved the FTP server with implicit context required only a single instance of invasive modification.\(^4\) While 16 modifications may not sound like many, realize that this was for only one step of evolution. Systems undergo continual evolution throughout their existence [Lehman and Parr, 1976; Belady and Lehman, 1976]; each stage of changes would require further invasive modifications. Such invasive modifications tend to cause software structure to degrade, making each successive evolution step harder [Lehman, 1974, 1980; Eick et al., 2001].

**Avoiding Planning for Change**

One might be tempted to interpret the results of this case study in the wrong way. While I say that I wish to avoid having to plan for change, it is important to be clear about my meaning. Change happens and it is important to plan for it but problems occur when we assume that specific changes will happen. For example, in the object-oriented version of the FTP server, I created a class hierarchy for different commands. This hierarchy is a natural structure to use for the sake of abstraction within the initial version of the server; it also happens to be easy to add new subclasses to it to support one form of evolution. The initial version using object-orientation also provides a class hierarchy to support different structures of data being communicated over the data connection; however, there is only one kind of data structure actually supported. The hierarchy is present in the latter case to explicitly worry about the possibility of specific future extensions, i.e., new structures for data. Thus, I expended time and effort in an attempt to support future extensions that may never happen—a poor investment.

Therefore, one might think that it is not necessary to synthesize our modules without extraneous embedded knowledge, that trying to reduce our modules to their minimum would be an act of planning for change. After all, it is possible to have our modules express essential structure in emergent fashion (e.g., using implicit context to evolve the purely

---

\(^4\)This one was due to the fact that the prototype tool does not support the capture of initializer executions.
object-oriented version of the FTP server), providing a means for evolving existing systems. We can see from Table 6.2 that the evolved version of the FTP server using implicit context actually required the addition of more boundary maps than did the cross-evolved version, both in terms of lines of code (260 vs. 196) and number of non-renaming maps (13 vs. 12).

However, partially expressing essential structure is not ideal. For starters, the numbers above are slightly deceiving; 64 of those lines of code (constituting 4 boundary maps) that were added to the initial implicit context version are necessary only because of a lack of support for wildcarding in the tool. But more significantly, the modules of the cross-evolved version continue to express knowledge beyond their emergent essential structure. Further evolution steps would still need to cope with this remaining EEK, as would any attempt at reusing these modules. By building our modules to avoid EEK in the first place, we are not planning for specific changes, we are simply avoiding worrying about the future. Through implicit context, we can make each module do exactly what it should and nothing more.

**Communication History**

The initial version of the FTP server using implicit context made 39 queries to communication history. Of these, 13 were used in the Singleton mapset and 13 were used in the ReplyMarshal mapset. Each of these mapsets contained 13 boundary maps to deal with each of the different subclasses of Reply; hence, these 26 queries would be reduced to 2 if the tool dealt with subclasses explicitly and/or supported wildcarding. Most communication history queries here involved retrieving the current object (i.e., the most recently created object in the current thread) representing the state of the FTP session, such as the current DataConnection, current remote address for the client, or the current Session.

A few queries involve determining the current state of the system by testing for the order of a small set of calls. For example, a small state machine is specified by the boundary map in Figure 6.18. Here, the current representation type is defined to be ASCII non-print by default, unless the client has most recently explicitly specified that it should be image type. RepresentationTypeCommandInterpreter must know how the arguments to the TYPE command correspond to different representation types; however, it should not know what the system in which it exists does with the representation types. Therefore, it simply calls a non-existent method to set the representation type appropriate to the arguments passed with the TYPE command. This boundary map takes this event and translates it into an instantiation of the appropriate subtype.

In the evolved version were 18 additional queries to communication history: 2 of these were used to find the local port number on the socket used for passive mode so the client could be given this information; 4 were used for extensions to the ReplyMarshal
// Replace attempts at instantiating RepresentationType with
// an instantiation of one of its subclasses.
static RepresentationType map():
  out(RepresentationType.new()) {
    // Find the last call that attempted to set image
    // representation type, if such exists.
    Call imageCall =
      History.lastCall(null,
            "setImageRepresentationType",
            null);

    // Find the last call that attempted to set ASCII
    // representation type, if such exists.
    Call asciiCall =
      History.lastCall(null,
            "setASCIINonPrintRepresentationType",
            null);

    // By default, we use ASCII representation type. So, if
    // there has been no attempt at setting image type, we're
    // done. Otherwise, the attempt at setting image type
    // needs to be more recent than any attempt at setting
    // ASCII type for image type to be current.
    if(imageCall == null ||
        asciiCall != null && imageCall.precedes(asciiCall))
      return new ASCIINonPrintRepresentationType();

    return new ImageRepresentationType();
  }

Figure 6.18: Using communication history to implement the Abstract Factory design pattern. Attempts to instantiate the abstract class RepresentationType are intercepted and replaced with an instance of one of its subclasses. The choice of subclass is based upon which representation type the client most recently specified, as determined through the communication history queries.
mapset for the added subclasses of Reply\footnote{There were 5 subclasses of \texttt{Reply} added; I forgot to add a boundary map to \texttt{ReplyMarshal} for one of these—another good reason why the tool should deal directly with subclasses. Also, I did not turn the added subclasses into Singletons; this was a decision not to bother, as the system worked fine without the Singleton optimization.}; 6 were used for supporting the authentication protocol; and 6 were used to have the \texttt{DataConnection}'s attempt to create a socket create a passive mode socket when appropriate.

The boundary maps making use of the state machines for authentication and passive mode (Figures 6.11 and 6.12) were by far the most complicated. For example, Figures 6.19 and 6.20 show the boundary map applied to all command interpreters that must respond "Not logged in" if the session has not been authenticated; this boundary map operates as follows. The most recent occurrence of the \texttt{PASS} command is located as an execution of the \texttt{perform} method on \texttt{PasswordCommandInterpreter}. If none such is found, the session is definitely unauthenticated. If the client has sent the \texttt{PASS} command, two conditions must be met: (1) the password in it must have been validated and (2) the \texttt{PASS} command must be immediately preceded by the most recent occurrence of the \texttt{USER} command.

We determine that the password has been validated by finding the call to the \texttt{isValid} method of \texttt{Authenticator} and making sure it has returned \texttt{true} (which is recorded as its object equivalent, \texttt{Boolean.TRUE}). We locate the most recent occurrence of the \texttt{USER} command and the occurrence of any command that immediately precedes the \texttt{PASS} command located earlier. These must be one in the same for the session to be authenticated.

The boundary maps shown in Figures 6.18 and 6.19–6.20 are relatively complicated and hard to understand, but this is largely a product of clumsy syntax. All that I am really doing here are pattern matches on the communication history, plus a couple of specific tests on the calls that matched the pattern of interest. For the representation type, we see that the session is to use the \texttt{ImageRepresentationType} if and only if the communication history matches a pattern that is essentially: "image [\texttt{\textasciitilde ascii}]* $". For the session to be authenticated, the communication history must match a pattern that is essentially: "\texttt{USER PASS [\texttt{\textasciitilde USER, PASS}]* $}". In Chapter 7, I examine how a different style of communication history query can be supported that directly expresses such pattern matching.

**Boundary Map Organization**

I originally envisioned mapsets as constructs of convenience with little understanding of how boundary maps should be organized into them. The structure of the mapsets used for the FTP server reflected this lack. I did attempt some vague order on the basis of the shared purpose of a set of boundary maps, but it was insufficient. A mapset is a transformation
// Ensure that the session is authenticated before allowing
// a command to be performed.

void map(): in(perform(String)) {
    Thread thread = Thread.currentThread();

    // Find the most recent occurrence of processing the PASS
    // command.
    Call passCall =
        History.lastCall(thread,
            PasswordCommandInterpreter.class,
            "perform", null, null);

    // If the client has issued no PASS command, the session
    // is definitely unauthenticated.
    if (passCall != null) {
        // If the session has been authenticated, the isValid
        // method on Authenticator must have been called as a
        // result of the PASS command. Find it.
        Call lastTest =
            History.lastCallInCFlow(thread,
                Authenticator.class,
                "isValid", null, null,
                passCall);

        (continued in next figure)

Figure 6.19: Boundary map applied to command interpreters to alter their behaviour on
the basis of whether the session has been authenticated, Part 1 of 2. See main text for
discussion.
// If isValid was not called, the session is unauthenticated.
if (lastTest != null) {
    // Find the last occurrence of processing the USER command.
    Call userCall =
        History.lastCall(thread, 
            UserNameCommandInterpreter.class, 
            "perform", null, null);

    // Find the last command that was processed just before the PASS command found above.
    Call predecessorOfPass =
        History.
            lastCallAnySubclass(thread, 
                CommandInterpreter.class, 
                "perform", null, null, 
                passCall.getPredecessor());

    // For the session to be authenticated, the isValid test must be true, and the last USER command must have immediately preceded the last PASS command.
    if (lastTest.getReturnValue().equals(Boolean.TRUE) && userCall == predecessorOfPass) {
        CONTEXT.proceed();
        return;
    }
}

// Otherwise, the session is unauthenticated.
Context.send(new NotLoggedlnReply());
}

Figure 6.20: Boundary map applied to command interpreters to alter their behaviour on the basis of whether the session has been authenticated, Part 2 of 2.
from one (partial) world view to another. As such, one must have a clear picture of what the initial and final world views are; if one does not, it is hard to maintain a mental model of what the mapsets do. This is why it is easy, at least at an abstract level, to understand the purpose of the AuthenticationCheck mapset, but not that of CommandAction.

In the evolved version of the FTP server originally constructed with implicit context, I modified some of the existing mapsets for two reasons. Once again, I needed to compensate for the lack of call site instrumentation by the prototype tool. I also decided to add boundary maps directly into these existing mapsets. However, these additions should have—and could have—been added externally to the existing mapsets. This was an oversight at the time, as I failed to recognize the importance of avoiding invasive modification of all of the program.

Overall

I successfully evolved both versions of the FTP server to incorporate the "unanticipated" addition of the authentication and passive mode features. This is not surprising: put sufficient effort into modifying a system, even to the point of rewriting it in toto, and one can have that system accommodate arbitrarily complex changes. However, remember that we wish to minimize the number and scope of the changes we make, and on this score, implicit context aids us as seen in the results.

6.3 Summary

I conducted two case studies to provide evidence in support of the thesis of this dissertation.

In the first case study, I selected a feature in an existing system and reused it in a different system of markedly different character—without reusing parts of the original system in which I was not interested. This feature was the Outline View of the Eclipse integrated development environment, described in some detail in Chapter 2. By using implicit context to express the emergent essential structure of the Outline View, I was able to cancel out the dependences upon Eclipse that were created by the extraneous knowledge embedded there. The Outline View was not modularized in a fashion amenable to its reuse; the designers of Eclipse did not plan for it to be reused in this way. Nevertheless, implicit context allowed for its reuse—reuse that would not have been possible when limited to the state-of-the-art.

In the second case study, I designed and implemented parallel versions of a system, one version using only object-orientation (OO) and one version using object-orientation plus implicit context. I then evolved these versions, each using the same approach with which it was originally constructed. The original version consisted of the Minimum Implementation of the File Transfer Protocol (FTP) as defined by the FTP specification; my designs ignored the existence of the remainder of the FTP specification. I had colleagues vet the initial OO
design to ensure that it was reasonable. I selected two features from the remainder of the FTP specification, authentication and passive mode, and evolved the versions to include them. In addition, I "cross-evolved" the initial OO version using implicit context.

While all three evolved versions were able to incorporate the new features, the OO version required extensive invasive modifications. The versions evolved through implicit context were able to avoid all but one invasive modification, the latter due to a lack of support in my prototype tool. The cross-evolved version demonstrates that implicit context can be adopted after the initial construction of a system. However, only the modules of the version initially constructed with implicit context are maximally free of EEK. Future evolution of these systems, inevitable in the real world, would require more and more invasive modifications—at great difficulty and risk—of the versions not initially constructed with implicit context. The EEK contained therein fully remained in the evolved OO version: for example, an instance of Session still needs to be passed to each Command subclass, the entire system needs to know that the Reply subclasses are each Singletons, every Command subclass holds a piece of the authentication protocol, etc. EEK partially remained in the cross-evolved version (e.g., an instance of Session still needs to be passed to each Command subclass, and the entire system needs to know that the Reply subclasses are each Singletons) since I only cancelled out the minimum amount of EEK hindering the immediate evolution step being performed.

What remains is to make the use of implicit context fully practical. I have identified a number of shortcomings in the prototype tool through the case study, beyond those described earlier in Chapter 5. I now proceed to discuss how these may be addressed in future work.
Chapter 7
Discussion

I have alluded to a number of issues that revolve around the question of what a module represents in isolation from its context of operation. In this chapter, I begin in Section 7.1 by describing a representation for the “meaning” of modules in isolation from their systems; this representation is based on the idea of local execution, the specification of the behaviour defined by a module without reference to its context. This section goes on to explain how implicit context can be used to support such localized reasoning about modules.

Reasoning about modules through appeals to local execution is only useful if such reasoning can be transformed to the global perspective of a system. Communication history can be used to bind a module in a flexible way to a novel context of operation. This binding is a specification of minimal dependences between a module’s local execution and its context’s execution. Section 7.2 describes this property of communication history and how it and local execution can be used to efficiently support communication history queries.

Section 7.3 then points out how aspect-oriented programming can be described in terms of local execution and minimal dependences, and compares it to implicit context.

7.1 Local Execution

A program is a specification of an execution [Hehner, 1999]. The lexicon and syntax of a programming language are used to interpret a string (the program) as a set of instructions; the semantics of the language places constraints on how to perform each instruction and the order in which instructions are to be performed. A virtual machine reads each instruction, performs it, and selects the next instruction according to the semantic constraints imposed by the language.

A module is a program fragment; therefore, a module is an execution specification fragment. However, whether one can attribute any meaning to a module in isolation, as separated from the context of any particular system, depends upon the way one interprets this statement. There are two possibilities: either a module is a fragment of an execution
specification or a module is a specification of an execution fragment. Let us consider if that difference is significant.

If we regard a module as merely a fragment of a specification, we cannot expect it to necessarily have much meaning in isolation. For example, in the Abstract Factory design pattern, some class must act the role of Client; that Client class will possess instructions similar to the following:

```java
public void doit(AbstractFactory factory) {
    Product p = factory.makeProduct();
}
```

However, while the use of particular names here is intended as an aid to understanding what is going on in the system, this code is equivalent to the following where the names have simply been changed:

```java
public void a(A obj) {
    B b = obj.f();
}
```

It is not until we consider some higher-level module containing this Client class that the presence of the Abstract Factory design pattern, or the part that the code snippet above plays in it, becomes definite. In other words, we would have to consider the other specification fragments (modules) that our module depends upon to make significant definite statements about it.

But a module does say something in isolation about the execution of the program of which it is a part. A set of "arbitrary" calls (from the perspective of an isolated module) may indeed only possess a meaning that we would consider significant when those calls exist in a larger context. That we cannot ascribe greater significance to that set of calls may bring into question our choice of modularization. However, when we read the source text of such a module in isolation, it does not say anything more than that this set of "arbitrary" calls is to take place. The snippet of code from the Client class shown above simply makes a call on an object passed as an argument, and stores the result; if there is anything more going on (such as the presence of the Abstract Factory design pattern), the clues lie in the parts of the system not seen above.

Rather than disavowing any knowledge of the meaning of a module in isolation from any given system, we can consider the module in isolation to be a complete program written in a special-purpose language. There is a virtual machine that interprets the instructions in this language [Goldberg, 1973a,b]. It is defined by the context in which the module executes; that context consists of such items as the other modules present, the state of the system, higher-level virtual machines, and the platform on which the system is executing.
Some instructions may be invariant under a large set of contexts. For example, consider a module written using Java syntax that utilizes primitive arithmetic operators: the behaviour resulting from those operators is invariant under all contexts that adhere to standard Java semantics. However, if that module were to execute in a context that did not adhere to standard Java semantics, the behaviour of an arithmetic operator might be overridden, perhaps in the fashion of C++ [ISO/IEC, 1998: p. 231ff.]. The presence of an arithmetic operator is simply an instruction to the local virtual machine, to be interpreted as appropriate to the situation. It does not matter from the local module’s perspective how its instructions are implemented; only the data that are input to it (through formal parameters, global variables, or return values) will affect its local computation.

With this picture, we can consider a module to be a specification of a local view on a fragment of its program’s execution; that is, a module is a specification of local execution. This local execution is invariant for a given set of inputs to the module. The arithmetic operator will always be called at the same point in the module, but the actual behaviour caused by that call will vary depending on the context of the module.

This is not to say that the execution of a module will be invariant from a global perspective, though. To see how the local perspective maps to the global perspective, I use event traces. Section 7.1.1 introduces event traces and their use in representing local execution. Section 7.1.2 demonstrates how contextual dispatch permits the reconciliation of local and global execution.

7.1.1 Event Traces

I have described the execution of a module in terms of the instructions performed by a virtual machine that is defined by the context of the module. However, another representation exists with which to picture an execution: an event trace.

An event trace can be thought of as a ticker tape, where time monotonically increases in one direction along the tape. The events transcribed upon the tape are the instructions performed by a virtual machine in the order in which they are performed. Consider this simple Java program:

```java
public class Simple {
    public static void main(String[] args) {
        Simple s = new Simple();
        s.doit();
    }

    void print() {
        System.err.println("Hello world!");
    }
}
```
void doit() { 
    for (int i = 0; i < 2; i++) {
        print();
    }
}

An event trace for the execution of this program might look like the following,\(^1\) where time increases downwards. I have used indentation to help indicate the nesting of executions.

Enter: Simple.main(String[])
Enter: Simple()
Exit: Simple()
Enter: Simple.doit()
Enter: Simple.print()
Exit: Simple.print()
Enter: Simple.print()
Enter: Simple.print()
Exit: Simple.doit()
Exit: Simple.main(String[])

This is a very simple trace, showing only the entries and exits to methods within the class Simple. A more complex trace might show the calls to and returns from these methods, the execution of the instructions to print "Hello world!", the creation of the instance of Simple, assignments, comparisons, field accesses, garbage collection, class loads, etc.

Different event traces are possible for the same execution, with widely differing levels of detail. For example, the trace produced by one particular Java Virtual Machine (JVM)\(^2\) on the simple program shown above consists of nearly 9,000 lines of text. Even so, this trace is fairly high-level, displaying only such events as method entries and exits, object creations, garbage collection, class loads and unloads, and field accesses; other, lower-level events, such as the execution of an arithmetic operator or an assignment statement, are not shown.

The low-level events that do not appear in an event trace can sometimes be implied. For example, in order that a method entry occur, a call to that method must have happened;

\(^1\)Java supplies a default constructor for concrete classes where one has not been explicitly declared.

\(^2\)Sun Classic VM, version 1.2.2, build 1.2.2.006, green threads, nojit; executed as “java -Xrunhprof Simple” at the command line.
the trace produced by the JVM does not happen to record this call as a separate event. Thus, an alternative trace to the one I showed above is as follows, where method calls and returns have been added. I explain the significance of the boldface events below.

Call: Simple.main(java.lang.String[])  
Enter: Simple.main(String[])  
Call: Simple()  
Enter: Simple()  
Exit: Simple()  
Return: Simple()  
Call: Simple.doit()  
Enter: Simple.doit()  
Call: Simple.print()  
Enter: Simple.print()  
Call: System.err.println(java.lang.String)  
Return: System.err.println(java.lang.String)  
Exit: Simple.print()  
Return: Simple.print()  
Call: Simple.print()  
Enter: Simple.print()  
Call: System.err.println(java.lang.String)  
Return: System.err.println(java.lang.String)  
Exit: Simple.print()  
Return: Simple.print()  
Exit: Simple.doit()  
Return: Simple.doit()  
Exit: Simple.main(String[])  
Return: Simple.main(java.lang.String[])  

In Java, an event trace for an entire program and an event trace for an individual module would be related in a straightforward manner. The full program trace that I have just given contains the event trace for the execution of the individual modules within the program; these local executions form subsequences within the full trace. Above, we can see the event trace for the Simple.doit method embedded within the full trace; it consists of the lines in boldface.

This local execution is independent of the context in which Simple.doit exists. To see more clearly this independence of local execution from context, consider an extension to the simple program given earlier. Now, we have a second class, Subsimple, which is a subclass of Simple. This new class overrides the implementation for the print method.
public class Subsimple extends Simple {
    public static void main(String[] args) {
        Subsimple s = new Subsimple();
        s.doit();
    }

    void print() {
        System.err.println("Ciao world!");
    }
}

The full program event trace for the execution of "java Subsimple" would be as follows.

Call: Subsimple.main(java.lang.String[])
Enter: Subsimple.main(String[])
Call: Subsimple()
Enter: Subsimple()
Exit: Subsimple()
Return: Subsimple()
Call: Subsimple.doit()
Enter: Simple.doit()
Call: Simple.print()
Enter: Subsimple.print()
    Call: System.err.println(java.lang.String)
    Return: System.err.println(java.lang.String)
    Exit: Subsimple.print()
Return: Simple.print()
Call: Simple.print()
Enter: Subsimple.print()
    Call: System.err.println(java.lang.String)
    Return: System.err.println(java.lang.String)
    Exit: Subsimple.print()
Return: Simple.print()
Exit: Simple.doit()
Return: Subsimple.doit()
Exit: Subsimple.main(String[])
Return: Subsimple.main(java.lang.String[])

Note the difference between the methods being called and the methods being entered; this is simply object-oriented dispatch taking place. The effect of the calls made by
Simple.doit have changed in this new context; however, the local execution of Simple.doit remains unchanged: the same sequence of local events occurs. Thus, we can represent this local execution without the need to represent the remainder of the full program execution:

```
Enter: Simple.doit()
Call: Simple.print()
Return: Simple.print()
Call: Simple.print()
Return: Simple.print()
Exit: Simple.doit()
```

This representation begins to give us a flavour of independent world views. However, object-oriented dispatch is a limited means of expressing something akin to independent world views. After all, Java classes explicitly declare that they specialize other classes and they implicitly declare (through the absence of the final modifier) that they permit such specialization. To make the leap to truly independent world views and the reconciliation of local and global execution, we must consider a different example.

### 7.1.2 Reconciliation of Local and Global Execution

Once more, I consider the example of the Abstract Factory design pattern. We have a Client class that requires the creation of an instance of a Product class. We want this Client class to specify its essential structure; an invocation of the constructor of Product is as straightforward a specification of the creation of an instance as Java permits. The resulting Client class is shown below (for the sake of simplicity, I have reduced it to a minimum necessary to illustrate the point).

```java
public class Client {
    public void doit() {
        Product p = new Product();
    }
}
```

The local execution of this module can be represented by the following event trace.

```
Enter: Client.doit()
Call: Product()
Return: Product()
Exit: Client.doit()
```

Let us proceed to consider using Client in different systems. In one situation, we embed the Client class in a context where Product is a concrete class, as shown below.
public class Product {
}

public class Main {
    public static void main(String[] args) {
        Client c = new Client();
        c.doit();
    }
}

Here, the full program execution is represented as a straightforward event trace, with the local execution of Client.doit in boldface:

Call: Main.main(java.lang.String[])
Enter: Main.main(String[])
Call: Client()
Enter: Client()
Exit: Client()
Return: Client()
Call: Client.doit()
Enter: Client.doit()
Call: Product()
Enter: Product()
Exit: Product()
Return: Product()
Exit: Client.doit()
Return: Client.doit()
Exit: Main.main(String[])
Return: Main.main(java.lang.String[])
public abstract class AbstractFactory {
    public abstract Product makeProduct();
}

class MSWindowsFactory extends AbstractFactory {
    public Product makeProduct() {
        return new MSWindowsProduct();
    }
}

class MotifFactory extends AbstractFactory {
    public Product makeProduct() {
        return new MotifProduct();
    }
}

class Main {
    public static void main(String[] args) {
        AbstractFactory factory;

        String os = System.getProperty("os.name");
        if(os == "Windows")
            factory = new MSWindowsFactory();
        else
            factory = new MotifFactory();

        Client c = new Client();
        c.doit(factory);
    }
}

We would like this program to result in an execution that could be represented by the following event trace (for executions on UNIX platforms), where the local execution of Client.doit is boldface.

Call: Main.main(java.lang.String[])
Enter: Main.main(String[])
Call: System.getProperty(java.lang.String)
Return: System.getProperty(java.lang.String)
As in the object-oriented dispatch example described earlier, there are disconnects in this event trace between which methods are being called and which methods are being entered. In both examples, this disconnection cannot be explained through a simple examination of the event traces as shown; the event traces are effectively non-deterministic. In the object-oriented dispatch example, the means of mapping between the local execution trace and the methods to be executed was specified by annotations in the source code, namely extends clauses, and the run-time type of the target object (Simple versus Subsimple).

In the contextual dispatch example, this mapping must take place through the provision of a boundary map, such as the following.

```java
mapset AbstractFactoryMap {
    void map(Client c): in(c.doit(AbstractFactory)) {
        c.doit();
    }
}
Product map(): out(Product.new()) {
```
AbstractFactory factory = (AbstractFactory)
    History.lastInstancePassed(AbstractFactory.class);

    return factory.makeProduct();
}]

apply AbstractFactoryMap to Client;

There is one boundary map here for each of the two disconnects present in the event trace. The first boundary map translates from the call to Client.doit(AbstractFactory) to the method entry of Client.doit(); this is a translation from the event trace expressing the global perspective (that the Abstract Factory design pattern is being used) to the local perspective (that it is not). The second boundary map translates from the call to instantiate Product to the method entry of AbstractFactory.makeProduct; this is a translation from the local perspective back to the global perspective.

That is the nature of dispatch in general: to map from one perspective to another. Contextual dispatch permits these perspectives to be inconsistent. Thus, from the global perspective, what is "really" happening during the execution of this program is the following event trace, expressed in a fashion consistent with object-oriented dispatch.

Call: Main.main(java.lang.String[])
Enter: Main.main(String[])
Call: System.getProperty(java.lang.String)
    Return: System.getProperty(java.lang.String)
    Call: MotifFactory()
        Enter: MotifFactory()
        Exit: MotifFactory()
        Return: MotifFactory()
    Call: Client()
        Enter: Client()
        Exit: Client()
        Return: Client()
    Call: Client.doit(AbstractFactory)
        Enter: Client.doit(AbstractFactory)
        Call: MotifFactory.makeProduct()
            Enter: MotifFactory.makeProduct()
            Call: MotifProduct()
                Enter: MotifProduct()
                Exit: MotifProduct()
On the other hand, from the local perspective of Client, the global execution is consistent with the following event trace.

Call: Main.main(java.lang.String[])
Enter: Main.main(String[])
Call: Client()
Enter: Client()
Exit: Client()
Return: Client()
Call: Client.doit()
Enter: Client.doit()
Call: Product()
Enter: Product()
Exit: Product()
Return: Product()
Exit: Client.doit()
Return: Client.doit()
Exit: Main.main(String[])
Return: Main.main(java.lang.String[])

In this case, Client would not necessarily have anything to say about exactly what had happened throughout the entire execution, but this is equivalent to event traces in general: they always leave out some details of what has happened.

Neither perspective should be considered more "real" than the other. Both perspectives leave out details, and describe the execution in a way that is convenient to a given reference frame. Contextual dispatch permits a translation between these reference frames, so that one can reason about what is happening in a localized fashion. I now consider the connection between communication history and event traces.
7.2 Communication History: Use and Efficient Support

In Chapter 4, I described communication history as conceptually being a complete record of everything that has happened within an execution. Regardless of the fact that any particular event trace leaves out some information, we can select an event trace that provides sufficient details to answer particular questions about communication history. In Chapter 5, I described an implementation of communication history that was a literal realization of this model: every event within an execution was recorded and at the disposal of queries that occur in boundary maps.

However, there are two important points to be made about communication history. First, communication history should not be regarded as simply some novelty in a programmer’s “bag o’ tricks”—although it could certainly be misused in this fashion. Second, the naïve implementation provided by my prototype tool can be highly optimized.

7.2.1 Communication History Queries as an Expression of Minimal Dependencies

We want to maximize the number of contexts in which a module may be used. That is, we want to be able to both reuse a module in multiple systems and to make it insensitive to changes within a given system. Through the expression of essential structure, we can achieve this maximization. But the scalability of the technique will be limited if we have to take a given module and immediately tie it to concrete entities in a system. Those entities may well change, and by tying down our abstract dependences so immediately, we limit the flexibility of all but our lowest level modules. Instead, we should express the minimal dependences that our module has upon its context. For modules that approach the global level, these minimal dependences may indeed be concrete. But below this level, they need not be.

Look again at the Abstract Factory design pattern example described in Section 7.1.2. In performing the translation between the local and global executions there, we needed to express where the instance of \texttt{AbstractFactory} (used in the translation process) was to be located. One thing must be understood here: \textit{Where to locate this instance is a choice that is made on the basis of the system design.} That is, there is some protocol that the system follows that defines where the instance is to be found. In the example in Section 7.1.2 the choice of protocol made was simply whatever instance of \texttt{AbstractFactory} that was last passed in the system was the one to be found. That choice would not be appropriate in every system. In systems where that choice was not appropriate, the boundary maps would need to be replaced, and hence, only the inner module (i.e., \texttt{Client}) would be reusable.

In those systems where that choice was appropriate, the boundary map should be reusable in addition to the inner module. But to realize that potential reusability, the proto-
col needs to be declared as abstractly as possible—otherwise, the declaration of the protocol itself would contain extraneous embedded knowledge.

In the absence of communication history, we would need to take our choice of protocol and manually compile it. We would need to manually analyze where in our system the last instance of AbstractFactory was passed, insert some code there to record this instance, and have our boundary map access this record. But this manual compilation would need to be performed again any time we were to change the system, and any time we wanted to reuse Client. This would be tedious and error-prone.

### 7.2.2 Communication History Queries as Pattern Matches

A query to communication history allows us to specify an abstract description of the protocol to be followed. If the protocol is simply to find the last-passed instance of a given type, the appropriate communication history query describes this. If a more complex protocol is needed, a different communication history query can be made.

As long as the event trace we are translating from is deterministic, we can describe the necessary subsequence(s) that locate the information we need. An event trace can be thought of as a string: each event represents a “character” in a complex alphabet. Consequently, a communication history query is a pattern match against one of the event traces of the two independent world views undergoing translation.

For example, to locate the last passed instance of AbstractFactory, we would need to describe those “characters” representing method calls containing an argument of type AbstractFactory. Let us say that this set of call events is described as

\[
\text{call}(*(\ldots, \text{AbstractFactory}, \ldots))
\]

where `call` is a keyword. The argument in this expression gives the kind of calls within the set: the asterisk indicates that any method name is a valid match, and the formal parameter list says that one formal parameter has to be of type AbstractFactory but that its position in the list does not matter. I call each of the events in this set a primitive event because its description makes no mention of its position relative to other events.

To describe the placement of this pattern within the event trace, we might borrow some standard notation from regular expression tools, such as `grep` from UNIX:

\[
\text{event}^{*} \text{ \text{call}^{*}(\ldots, \text{AbstractFactory}, \ldots)} \text{ \text{call}^{*}(\ldots, \text{AbstractFactory}, \ldots)}^{*} \$
\]

The caret matches the start of the event trace. The keyword `event` matches any single event; the asterisk after it matches zero or more occurrences. Next comes the call passing an instance of AbstractFactory, followed by zero or more events which are not calls passing AbstractFactory. And finally, the dollar sign matches the (current) end of the
event trace. I call each event matched by this complete query a *compound event*; it consists of primitive events with ordering constraints attached.

The syntax I have just used in the example is intended to merely give a flavour of the pattern matching in communication history; I do not promote it as being elegant or even sufficient. In fact, we are generally interested in matching patterns within communication history that possess more structure. Specifically, we often want to pair up method entries and method exits, or calls and returns, that are nested in particular ways. Such pairing is equivalent to balanced parenthesis matching. Thus, regular expressions do not suffice to describe the patterns of interest, but more general context-free grammars are required [Chomsky, 1956, 1959].

Determining if an event trace matches a pattern of interest is equivalent to deciding if a string belongs to a language. Therefore, one might be tempted to point out that this could be an undecidable problem in the most general case. To this, I have two points to make.

First, while one could make arbitrarily complex queries, there is no reason to assume that such would be necessary in any real situation. The hope here is that, by making "reasonable" local choices in terms of system structure and queries to the communication history, that no bizarre queries would need to be made at higher levels as a result. It is true that this is a conjecture; my desire to address this conjecture in the context of "practical," "reasonable," and "real" situations make an analytic approach to its validation all the more difficult. But a communication history query that was more complex than the kind of system it was querying would be an indication of misuse.

Second, a communication history query does not need to be arbitrarily complex. An arbitrarily complex query would effectively be a complete computation independent of the rest of the system. In most practical situations, one could simply retrieve the information of interest through a communication history query, and perform a computation based upon this information separately. For example, in the boundary maps shown in this dissertation, that is precisely what happens: one or more communication history queries are made, retrieving particular information about the execution, and then computations are performed on the basis of this information.

A communication history query is simply a localized means of retrieving information from the execution of a system. This locality has two aspects. A communication history query is present at a given place in the source text. The question it answers does not depend on individual snippets of code that have to all work correctly and in concert. A communication history query also takes place at a specific location within the execution of the system. The question it answers does not depend on having asked various smaller questions previously during the execution, and having stored those answers away for later retrieval. And then, a variable that, for example, contains a flag simply contains a flag: one cannot be sure that its value expresses the property in which one is interested.
7.2.3 Efficient Support of Communication History Queries

Recall that the prototype tool that I described in Chapter 5 is quite inefficient in two ways: (1) it instruments every method entry and method exit, resulting in bloat in the size of the source text and in a degradation of dynamic performance due to the execution of all this instrumentation; and (2) every event is recorded once and forever, requiring ever increasing storage space, and ever slower responses to queries. With the picture I have outlined of communication history queries as pattern matches against event traces, I can now address how we might efficiently support communication history queries.

The basic idea is, rather than support every possible query that one might make on communication history, to support within a given system only those queries that are actually made. Thus, in the example used in the previous section, we would need to support only this query:

```
^ event* call(*(.., AbstractFactory, ..))
[^call*(.., AbstractFactory, ..)]*$
```

if no others occurred in the system.

There are three steps needed to reduce the support to the minimum necessary: (1) optimize the query expression; (2) instrument those points in the source text that represent the primitive events of interest; and (3) feed these primitive events into an automaton to identify if the compound event of interest has occurred.

In the example query, the subexpression

```
^ event*
```

means that we are only interested in a suffix of the event trace. This subexpression can be discarded, reducing the query to

```
call(*(.., AbstractFactory, ..))
[^call*(.., AbstractFactory, ..)]*$
```

The primitive events of interest here are partitioned into those that match

```
call(*(.., AbstractFactory, ..))
```

and those that do not. The pattern "a [^a]* $" for any subexpression a indicates that we want the most recent occurrence of a. Therefore, we only need to instrument those points in the source text that can be statically identified as potentially generating calls to methods with arguments of type AbstractFactory.

Finally, our automaton needs to record only the instance of AbstractFactory passed in the most recently generated primitive call event handed to it by the instrumentation. As soon as a new event of this kind occurs, the old instance stored by the automaton can be discarded in favour of the new.
Not all communication history queries will necessarily be so amenable to reduction. However, my prototype implementation of communication history provided only a small set of queries, and this set quickly reached a stable size through the course of practical use. This is an indication that, at worst, optimized support for this small set of queries can be provided.

For more general queries, parser generation technology is available that can cope with context-free grammars. This is not to say that the problem will necessarily be simple to solve, but there is a promising route forward. Previous work by Havelund and Roşu [2001] and by Giannakopoulou and Havelund [2001] may help here.

7.3 Implicit Context and Aspect-Oriented Programming

Implicit context bears resemblance to a group of techniques generally referred to as aspect-oriented programming (AOP) [Kiczales et al., 1997]. AOP was originally described as providing the means for the separation and encapsulation of concerns (such as synchronization or distribution) that "crosscut" the base modularity of a system. More recently the term has come to refer to any technique that permits functionality to be inserted into a system on the basis of predicates that describe the points of interest [Elrad et al., 2001]. Implicit context provides a superset of the former definition, but is an element of the latter definition.

Describing the fundamental differences between the various AOP techniques, rather than superficial differences of syntax or tool support, has been difficult for the community. Superficial differences can be overcome; fundamental differences are not so amenable to change. The model of local execution and event traces can be used as a means of comparison. I outline some of the features and concepts behind a particular AOP approach in Section 7.3.1, and translate this description to the model of event traces in Section 7.3.2. A more general description of AOP techniques can be found in Section 8.4.4.

7.3.1 Outline of Concepts in AspectJ

AspectJ [Kiczales et al., 2001] is an aspect-oriented extension to Java; it is the chief representative of a particular style of AOP, the one with the highest level of development (at the time of writing this dissertation). The AspectJ-style of AOP operates on the concept of join points. A join point is an event in the execution of a program where behaviour can be added, to either replace or supplement other behaviour. Join points are equivalent to primitive events as I have described them above. To add or replace functionality, one quantifies a set of join points through a predicate that matches certain properties of events of interest. These sets are called pointcuts, since they generally describe a set of join points that crosscuts the system's modularity. The pointcut predicates are constructed as expressions that
compose simpler keyword expressions called pointcut designators (PCDs). PCDs exist that capture method executions, field accesses, initializer executions, etc.

For example, a pointcut predicate declaration might read as:

```java
pointcut pc(String s): args(s) &&
  execution(public void *.meth(String));
```

This statement declares a pointcut predicate called `pc`; it captures join points (specifically, method executions) possessing certain properties. The argument to the `execution` PCD indicates that the method being executed must be named `meth`, that it must take exactly one argument of type `String`, that it must have a result type of `void`, and that it must possess `public` visibility. The asterisk indicates that this method may belong to any class; such wildcards may be used in place of much of the other information that we do not wish to describe with particular values.

The `args` PCD permits two things: it constrains the passed argument to be of a particular type (String in this case), and it can expose the object passed in that argument through the formal parameter to the predicate.

Pointcut predicates can be used in the declaration of functionality much as event designators are used in the declaration of boundary maps; this added functionality is called advice. That functionality is performed whenever the predicate evaluates to true. There are three basic kinds of advice: `before` advice executes just before an identified event occurs; `after` advice executes just after an identified event occurs; and `around` advice executes in place of an identified event.

For example, we can use the pointcut predicate `pc` to define the advice like the following:

```java
void around(String s): pc(s) {
  System.err.println("Discarding call to meth:" + s);
}
```

Now, whenever an execution of a method called `meth`, with the other properties mentioned above, is about to happen, this execution is discarded in favour of this advice. The `String` object that was to be passed to `meth` is captured and passed to the body of the advice, where it can be used like a formal parameter to a method.

An additional PCD of interest in my discussion is `cflow`. A pointcut predicate using this PCD can be declared as:

```java
pointcut pc2(int i, String s):
  cflow(execution(* *.*(int))) && args(i) && pc(s);
```

\[3\] In fact, that is where the syntax for boundary maps was derived from.
The pointcut described by pc2 is a restriction (i.e., a subset) of that described by pc. It requires that any execution described by pc also happen within the control-flow of a certain other method execution; that is, it must occur after a certain method entry, but before that method exits. This nesting method must possess an argument of type int; again, this argument is exposed so it can be used within added, boundary-map-like functionality.

7.3.2 Describing AspectJ in Terms of Event Traces

The semantics of AspectJ are based upon changes to the execution stack [Wand et al., 2002]. For example, execution selects stack changes involving the top frame on the stack (i.e., its addition or removal), while cflow selects those stack changes that happen when a given execution frame remains on the stack. This stack-based semantics is extended in a straightforward manner to permit the capture of events that do not truly involve the execution stack of the Java Virtual Machine, such as field accesses. Once a frame is removed from the stack, it ceases to be accessible.

It can be difficult to tell from all the details of syntax and semantics if there is any significant difference between the AspectJ-style of AOP and implicit context. However, Hannemann and Kiczales [2002] have indicated that AspectJ is not capable of separating the Abstract Factory design pattern in the manner shown above. Describing AspectJ semantics in terms of event traces allows a more general comparison, resulting in three key differences.

- Most pointcut predicates can only differentiate between primitive events.
- The cflow PCD permits the description of compound events, but only in a very limited way.\(^4\)
- Reconciliation between conflicting world views is not possible.

I examine these points below.

Pointcut Predicates and Primitive Events

Consider again the simple Java program shown in Section 7.1.1:

```java
public class Simple {
    public static void main(String[] args) {
        Simple s = new Simple();
        s.doit();
    }
}
```

\(^4\)This statement remains true for the minor variants of the cflow PCD, like cflowbelow.
void print() {
    System.err.println("Hello world!");
}

void doit() {
    for(int i = 0; i < 2; i++) {
        print();
    }
}

We can capture and replace the execution of print through the following pointcut and advice:

pointcut printPC(): execution(void Simple.print());

void around(): printPC() {
}

This would discard the execution of print without replacing it with anything. This has the effect of translating the original execution trace to the following one; events discarded from the original trace are struck out.

Call: Simple.main(java.lang.String[])
Enter: Simple.main(String[])
Call: Simple()
    Enter: Simple()
    Exit: Simple()
    Return: Simple()
Call: Simple.doit()
    Enter: Simple.doit()
    Call: Simple.print()
        Enter: Simple.print()
        Call: System.err.println(java.lang.String)
        Return: System.err.println(java.lang.String)
        Exit: Simple.print()
        Return: Simple.print()
    Call: Simple.print()
        Enter: Simple.print()
        Call: System.err.println(java.lang.String)
        Return: System.err.println(java.lang.String)
Exit: Simple.print()
Return: Simple.print()
Exit: Simple.doit()
Return: Simple.doit()
Exit: Simple.main(String[])
Return: Simple.main(java.lang.String[])

But note that this pointcut does not differentiate between the two calls to Simple.print. The only difference between them lies in their relative position within the trace. Thus, only a compound event could differentiate between them.

The cf low PCD and Compound Events

The cf low PCD can be used to specify compound events to a limited extent. Consider the following pointcut predicate:

pointcut doitCF(): cf low(execution(void Simple.doit()));

It would match every event that occurred while Simple.doit remained on the stack, as shown by the boldface lines below:

Call: Simple.main(java.lang.String[])
Enter: Simple.main(String[])
Call: Simple()
Enter: Simple()
Exit: Simple()
Return: Simple()
Call: Simple.doit()
Enter: Simple.doit()
Call: Simple.print()
Enter: Simple.print()
Call: System.err.println(java.lang.String)
Return: System.err.println(java.lang.String)
Exit: Simple.print()
Return: Simple.print()
Call: Simple.print()
Enter: Simple.print()
Call: System.err.println(java.lang.String)
Return: System.err.println(java.lang.String)
Exit: Simple.print()
Return: Simple.print()
Exit: Simple.doit()
Return: Simple.doit()
Exit: Simple.main(String[])
Return: Simple.main(java.lang.String[])

Let us say that we wanted to ensure that no executions of the constructor of Simple occur while Simple.doit executes; none do so in this example, but we might want to ensure this property even if the system changes later. We could capture any such events by the pointcut:

```java
pointcut constructor(): execution(Simple.new());
```

However, this constructor pointcut would also capture every other execution of this constructor. To differentiate between the constructor executions of interest and those not of interest, we could conjoin these pointcuts:

```java
pointcut unwanted(): constructor() && doitCF();
```

Advising the unwanted pointcut would allow us to signal an error should it occur. Thus, we can see that cf low allows us to specify some compound events.

But there is nothing obviously special about frames remaining on the stack, other than the fact that the flow of control ought to return to each frame eventually. One might be interested in some arbitrary primitive event in the past, irrespective of whether some representation of it remains on the stack. We have seen such use in previous chapters (e.g., in Figures 6.5, 6.18, 6.19—where a control-flow-based query is also used—and 6.20).

Communication history queries and pointcut predicates permit the partitioning of event traces into those of interest and those not of interest, at a given point in an execution. Without any reference to prior events, every event trace with the same final primitive event falls into the same equivalence class. The use of generalized communication history allows us to discriminate more finely between different event traces. Pointcut designators like cf low are an intermediate level of expressiveness, without an obvious reason as to why they are not either more general (permitting finer call trace differentiation) or less (treating primitive events as sufficient). In general, making unwarranted assumptions about the structure of the control flow is to introduce extraneous embedded knowledge; for

---

5It should be pointed out that both implicit context and AspectJ provide lexical means for discriminating between event traces of interest and not of interest. Implicit context tests for events relative to a boundary, which immediately separates event traces into two partitions. Similarly, AspectJ provides the means for limiting events of interest to within a lexically-identified scope (through the within and withincode PCDs) and aspect associations; neither of these exposes more of the communication history, but can be used in selecting modularizations. Implicit context does not impose any modularizations, and my prototype tool supporting it provides only simple ones compared to AspectJ; this is an area for future work.
example, when an assertion that a particular call has occurred at some previous time will suffice, an assertion used in its place that a very specific and complete sequence of calls has occurred is EEK.

**Reconciliation of Conflicting World Views**

Aside from the limited power of discrimination provided by its pointcut designators, AspectJ inherits from Java the constraint that a globally-consistent perspective of the system is necessary. For example, the Abstract Factory design pattern example from Section 7.1.2 is not supportable in AspectJ. In AspectJ, one could specify a pointcut via

```java
call(Product.new())
```

that would capture any attempt to instantiate the `Product` class, and replace this attempt with other functionality (such as calling a method on a Factory class); however, `Product` must be a concrete class. That is, given that a call to the constructor exists, a declaration of the constructor must also exist, even if the execution can never get there. Without the concept of local execution, as provided by implicit context, such separation of concerns is not possible.

### 7.4 Summary

The behaviour specified by a module in isolation remains the same regardless of the context in which that module is embedded; this behaviour is that module's local execution. The local execution of a module does not change if that module does not change.

We can represent an execution, local or otherwise, as an event trace. Such a trace records events of significance within an execution in the temporal order in which they occur. In simple situations, an event trace representing a local execution can be considered a subsequence of an event trace representing a global execution. Special forms of dispatch, such as object-oriented polymorphism or contextual dispatch, form more complex situations.

In the presence of these dispatch mechanisms, the local execution is a translation of the global execution. For object-oriented dispatch, events that occur in the global view, such as a call to execute a given method, may be replaced with a different event in the local view, such as the execution of a different method. For contextual dispatch, the translation process can be much more complex; it can depend upon earlier events that have occurred in the event trace.

The idea of event traces can be used to efficiently support queries on communication history. These queries are pattern matches against event traces. The primary principle involved is that of supporting only those queries that are actually made within the system.
The query expression can then be optimized for typically-encountered patterns; this is analogous to the process of optimizing typically-encountered regular expression patterns. Two parts are then required to support the optimized queries. First, the individual (primitive) events mentioned in the pattern can be statically matched in the source text. Second, an automaton is needed to which can be fed these primitive events of interest. The points in the source text that match the primitive events of interest are instrumented so that should they occur during the execution, they can feed information to the automaton. This efficient support for communication history queries remains to be implemented in a tool.

Finally, I have demonstrated some differences in the expressibility of implicit context and one particular approach to aspect-oriented programming (namely, AspectJ) through the use of event traces. Pointcut predicates in AspectJ are the equivalent of communication history queries; however, only relatively simple queries can be made due to the fact that AspectJ’s semantics are based upon the idea of the execution stack. Most pointcut predicates can only differentiate between primitive events, ones that do not involve ordering of multiple events. The cflow pointcut designator is an exception, but its use is limited to describing events still in existence upon the execution stack. And AspectJ is not capable of translating between global and local executions: a single, globally-consistent world view must exist. I have given a simple example where this is shown to be a limitation with negative consequences.
Chapter 8

Related Work

I have argued in earlier chapters for the ability of essential structure through implicit context to ease software evolution and software reuse. There has been much other work that addresses evolution and reuse. In addition, there have been various ideas that bear some resemblance to those of essential structure or implicit context without necessarily focusing on evolution or reuse. In this chapter, I consider the relationships of these two sets of work to this dissertation.

I begin by considering the relationships to some basic principles in Section 8.1. I then proceed to other approaches to software reuse and evolution; these fall into three major categories. Ad hoc approaches are considered in Section 8.2. Approaches that centre upon particular interfaces, architectures, or products are described in Section 8.3. And Section 8.4 expounds upon generative approaches. Sometimes, a given approach falls under multiple headings. Cross-references are given to other categories where needed.

Finally, I consider some additional topics that bear directly on the concepts of essential structure and implicit context, in Section 8.5.

8.1 Principles

In this section, I first consider two basic ideas that are applicable to the majority of approaches addressing software evolution or reuse: separation of concerns, in Section 8.1.1; and dependences, coupling, and cohesion, in Section 8.1.2. I then proceed to three other ideas: local reasoning, in Section 8.1.3; design-for-change, in Section 8.1.4; and reflection, in Section 8.1.5. There is less widespread agreement as to the importance of these latter ideas in addressing reuse or evolution.

1Run-time evolution lies outside this dissertation as a topic for future work; thus, we do not consider approaches to it here.

167
8.1.1 Separation of Concerns

Dijkstra [1976] is credited with having first coined the term separation of concerns to describe the idea that software should be structured in such a way that different "concerns" can be dealt with individually. The idea pervades modern thought on software structure, with its most extreme application in aspect-oriented approaches [Hürsch and Lopes, 1995] (see also Section 8.4.4). However, the question of what a "concern" is or is not still troubles us [Murphy et al., 2001].

While essential structure is clearly another attempt at separating concerns, it gives some specific guidance as to what is to be removed from our modules: extraneous embedded knowledge.

8.1.2 Dependences, Coupling, and Cohesion

Essential structure is an attempt to minimize the dependences that a module has on its system, and vice versa. Software tends to get more difficult to evolve as it grows older [Belady and Lehman, 1976; Parnas, 1994]; this can be attributed to the growth in the number of dependences in the system as its structure degrades [Stevens et al., 1974; Wang et al., 2001]. Stevens et al. described software structure and dependences in terms of coupling and cohesion. Coupling is a property describing the degree of dependence between modules; a highly coupled system is harder to evolve. Cohesion is a property that has been more elusive to describe. Roughly speaking, it is a property that describes how meaningful the relationship is among elements in a module. If a module consists of odds and ends that are thrown together, it has low cohesion; if the elements within a module are tightly bound such that they must work together for the module to function correctly, the module has high cohesion. High cohesion is thus considered preferable for good design.

Much work has tried to measure and assess the cohesion of systems [Eder et al., 1992; Ott and Bieman, 1992; Lakhotia, 1993; Briand et al., 1998; Kang and Bieman, 1998; Nandigam et al., 1999]; these measures effectively describe cohesion as intra-modular dependences. Thus, it should not surprise that some empirical work indicates that reusing a module is easier when it possesses low cohesion [Bieman and Kang, 1995], since reuse often requires adaptation or evolution of a module.

Coupling and cohesion tend to treat all dependences equally. None of this prior work permits the differentiation of necessary dependences from extraneous embedded knowledge; the dependences that represent EEK are the problematic ones.

8.1.3 Local Reasoning

There have been a number of efforts that have either aimed at or touched upon the idea that one needs to reason about systems on a local rather than global basis. Permitting such
local reasoning can ease the burden on a developer, requiring less effort to construct good mental models for the tasks they undertake [Edwards, 1995]. On the other hand, requiring global reasoning tends to bury the hapless developer in a "plethora of details" [Parnas and Clements, 1990].

Local reasoning and isolation of changes are complementary. Global variables were regarded as harmful early on precisely because of the necessity for global reasoning when using them [Wulf and Shaw, 1973]. Even objections to the use of local variables at the time were concerns about the ability for local reasoning [Fisher, 1983]; without strong scoping mechanisms in place, the particular "local" variable to which one refers can be quite malleable and difficult to determine.

Encapsulation of details behind an interface [Parnas, 1972] permits those details to be modified without affecting the rest of the system, to an extent. However, such encapsulation does not suffice to permit local reasoning, since the internals of a module can depend on the interfaces of many external modules.

The need for exception handling arose because of the impossibility of performing truly global reasoning [Parnas and Würges, 1976]. Even in a system where one could have all the source code in order to perform a formal verification, the system ultimately needs to interact with the real world: hardware breaks down, and humans can behave unpredictably.

To reason soundly at the local level, we need to have that reasoning translate well to the global level. Due to the shortcomings of the current state of the art, developers tend to be understandably dubious about foregoing the need to see all the details for an entire system all at once. Sitaraman et al. [2000] discuss reasoning about software component behaviour, both in isolation and as composed into a system. The properties of interest reasoned about locally must be maintained when moving to the global view. As long as we can somehow continue the reasoning at an inter-modular level without re-analyzing low-level details, local reasoning can be sound. In my future work on implicit context, I will need to demonstrate how the translation between world views also transforms the local reasoning into sound global reasoning. Such work awaits formalization of the approach. However, the local reasoning aspects of the work of Sitaraman et al. should be directly applicable.

8.1.4 Design-for-Change

It has long been held that good software design must consider the accommodation of change [Parnas, 1972, 1979]. Some design decisions are narrowly applicable to a single context and therefore are prone to volatility. Isolating these design decisions from the majority of a system (for example, by encapsulating them in modules behind more general interfaces) makes it easier to change these decisions. VanHilst and Notkin [1996a] explain the need for finer-grained modularization to isolate smaller design decisions, a recognition that modules can be highly complex entities in their own right and thus difficult to change. Essential
structure considers modules to exist at multiple levels of granularity.

This principle of design-for-change is often applied in the form of good coding practices or making design decisions explicit [Han, 1997]; for example, using a named constant whose value is defined at one place is preferable to having the value of that constant scattered throughout the system. However, on the larger scale, design-for-change amounts to predicting change and designing for it, rather than using a more malleable coding style. Indeed, some proponents would claim that evolution and, in particular, reuse are impractical unless one anticipates changes during the development of a system [Parnas, 1972; Castano and De Antonellis, 1993; Wasmund, 1995].

But one can rarely accurately predict the future. Without accurate prediction, there are two pitfalls. First, we waste effort adding complexity into our systems that ends up not being used, while making these systems less efficient. Our systems become harder to understand, since the purpose of the added complexity is not apparent. Second, our designs will not necessarily accommodate the changes that are actually needed.

Essential structure allows us to design our modules in a style that accommodates external change while minimizing the likelihood of internal change. It avoids the expense and omniscience required to predict change. However, in situations where one can get away with making assumptions about the structure of a system and the evolution thereof, techniques that take advantage of these assumptions are liable to be at an advantage. It remains to be shown what the cost–benefit tradeoffs are of essential structure through implicit context.

8.1.5 Reflection

Computational reflection [Smith, 1982] is “the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation” [Maes, 1987]. Two degrees of reflection are often distinguished: introspection involves reacting to the representations of the computation present in a given system, while metaprogramming involves the manipulation of those representations.

Introspective capabilities are provided by Java, for example, where one can specify that classes be dynamically loaded, ask about the methods and fields of classes, and cause those methods to be invoked if they prove of interest. One cannot, however, add or replace methods or types through this facility. Metaprogramming capabilities are provided by CLOS, for example [Kiczales et al., 1991]. Metaprogramming is a superset of introspection, adding facilities to dynamically create or replace methods or types.

Metaprogramming is a very powerful tool. It requires a great deal of discipline to use properly, rather than to resort to its use at every opportunity. Ad hoc application of metaprogramming tends to make programs harder to reason about, since it alters those programs dynamically in a way that can be opaque. Implicit context provides a mechanism
that is less powerful than metaprogramming: for example, one cannot dynamically create new classes with implicit context. Implicit context does permit reasoning about a system, and certain translations of dynamic events within a system (see Chapter 7); however, the hope is that the static specification of these events will permit the right balance between expressive power and comprehensibility. Further study is required to judge whether this hope is met.

8.2 Ad Hoc Approaches

Ad hoc approaches are those where manual intervention forms the basis for supporting change. These fall into two categories: copy-and-modify, described in Section 8.2.1; and refactoring, described in Section 8.2.2.

8.2.1 Copy-and-Modify

Copy-and-modify is likely the most common form of reuse, since it requires no special support from languages or tools. The programmer notices a similarity between an existing source code module and a desired one, makes a copy of the original, and modifies the copy to the needs of the situation. This idea goes by other names such as scavenging [Krueger, 1992], reclamation [Garnett and Mariani, 1990], or extraction [Lanubile and Visaggio, 1997]. Some such techniques use more sophisticated means for identifying the candidate modules to be copied, such as flow graph-based program slicing [Lanubile and Visaggio, 1997].

Copy-and-modify suffers from two shortcomings. First, after having been copied, the original module may eventually require changes itself due to error corrections. These changes cannot automatically propagate to the copies of the module. Therefore, the manual copy-and-modify operation must be repeated to propagate the changes, and since the process was manual, there is no guarantee that one will be able to repeat the process faithfully. Second, copies of the module may be scattered hither and yon wherever it was reused. Since there is no permanent connection between the original and any copies of it, locating each version in order to propagate a change to it is problematic. Empirical results support the contention that these shortcomings are a practical reality [Burd and Munro, 1997]; copy-and-modify, while lightweight, is error-prone.

8.2.2 Refactoring

A significant number of techniques centre about the idea of meaning-preserving transformations. This is often called refactoring [Opdyke, 1992; Tokuda and Batory, 1999; Fowler
et al., 2000; Beck, 2000; Butler and Xu, 2001; Baxter, 2002]; similar ideas have been pro-
moted without using this term [Feather, 1982; Burson et al., 1990; Griswold and Notkin,
1993]. The differences between these techniques lie in the kinds of restructuring supported,
or details of usage.

In short, refactorings are performed when one desires to restructure the functionality
in a system. Object-oriented refactorings might include such operations as moving the
implementation of a method from one class in a hierarchy to its children, adding a parameter
to an overridden method, or collapsing a class hierarchy. Tools are often applied to permit
refactoring to avoid simple errors.

Refactoring can either be performed in the modify step of copy-and-modify, or it can
be performed on the original code itself. In the former case, it suffers from the same short­
comings as does copy-and-modify: code must be replicated, and it is not a simple process
to propagate changes through the copies, even should those copies be located.

When refactoring is applied to the original source code of a module, the problems of
replication are avoided; however, new troubles arise. If one is working on a system the
source code of which is completely isolated from that of any other system, then one is
completely free to restructure it as desired. But it is more common that one must adhere
to published interfaces, either those provided by other developers (for example, interfaces
to libraries) or provided to other developers (for example, the interface to a framework); in
this situation, one is constrained as to what restructuring is possible without violating these
interfaces. Some empirical work indicates that it is best to avoid changing such interfaces
whenever possible [Svahnberg and Bosch, 1999].

Furthermore, a simple modularization is not always possible that will satisfy the needs
of every system in an ideal fashion, and thus, there is no corresponding refactoring that will
suffice; I elaborate on this point in Section 2.3. This is the reason for much of the work on
multiple world views (see Section 8.5.3).

8.3 Interface-, Architecture-, and Product-Centric Approaches

There exists a continuum of approaches that centre upon the idea of standardizing certain
commonalities between systems or modules. At one end of this continuum are systems that
allow for minor variation on the basis of configurable options. At the other end are sys­
tems that are built from pre-existing components, each of which adheres to some standard
interface.

Inheritance, polymorphism, and frameworks are the topics of Section 8.3.1. I describe
approaches that adapt or wrap existing modules to integrate them, in Section 8.3.4. In
Section 8.3.2, I discuss components and middleware architectures. And I finish with product
line approaches in Section 8.3.3.
8.3.1 Inheritance, Polymorphism, and Frameworks

Object-orientation is founded on three interacting mechanisms: encapsulation of details behind an interface [Parnas, 1972]; inheritance of the implementation of parts of that interface [Taivalsaari, 1996]; and polymorphic dispatch of a call on that interface to one of its alternative implementations [Strachey, 1967; Cardelli and Wegner, 1985]. This is a powerful combination, and its potential for easing synthesis, reuse, and evolution of software is no small part of why there has been so much research into and adoption of object-orientation.

Although object-orientation has been somewhat successful, particularly when compared to procedural development techniques [Kiran et al., 1997], it has not lived up to all the claims made about it in regards to its support for reuse and evolution [Bassett, 1997a: pp. 57–58]. Inheritance can be overused [Johnson and Foote, 1988]. It is not a general-purpose reuse mechanism, but requires conceptual similarities to exist between the superclass and subclass lest the design of the system erode. Some empirical studies indicate that inheritance is used fairly sparingly in real systems, with the average number of subclasses residing in the neighbourhood of 0.6 [Bieman and Zhao, 1995]. One might claim that this is simply an indication that there was little opportunity for reuse in those systems; however, the prevalence of code replication [Burd and Munro, 1997] and design replication [Gamma et al., 1994] makes this doubtful. That the depth of inheritance hierarchies is inversely related with their comprehensibility has been claimed [Daly et al., 1996], but remains contentious [Deligiannis et al., 2002].

Added complications such as multiple inheritance (i.e., inheritance from more than one superclass) or representation inheritance [Edwards, 1997] do not alleviate the lack of a general-purpose reuse mechanism.

The difficulty of typical approaches to inheritance is that subtyping becomes coupled to the reuse of source code. Some languages allow these concepts to be separated to a degree. On the one hand, interface inheritance [Hamilton and Radia, 1994] is straightforward; for example, Java permits the description of interfaces, essentially abstract classes that specify a protocol to which implementing classes must adhere. Implementation inheritance is also possible. C++ permits private inheritance, which can be used indirectly (i.e., via delegation to an inherited, private method) in the reuse of a set of implementations. However, this kind of implementation inheritance is still inheritance, requiring more conceptual continuity between classes than pure reuse of algorithms or source [Weihe, 1997, 2001]. Generative techniques (see Section 8.4) have been pursued to address this shortcoming.

Polymorphism is a mechanism allowing the decoupling of method calls from method implementations. Commonly used object-oriented languages like C++ and Java provide single dispatch: the method implementation that receives a message is selected on the basis of the run-time type of only one object; only the static types of additional parameters are
used to discriminate between overloaded methods. Other languages, such as CLOS [Bobrow et al., 1986] and Cecil [Chambers, 1992], provide for multiple dispatch: the run-time type of every parameter to a message can affect the selection of the method implementation to receive that message. Subjectivity [Harrison and Ossher, 1993; Harrison et al., 1994; Kristensen, 1998] permits method selection not only on the basis of the type of the receiver, but also of the type of the sender (see Section 8.4.4 for further discussion of subjectivity). Predicate dispatch permits selection of a recipient implementation on the basis of whether a given predicate evaluates to true or false [Chambers, 1993; Ernst et al., 1998]. Contextual dispatch is a generalization of these other dispatching mechanisms. The context in single dispatch is the run-time type of the object receiving the message; in predicate dispatch, it is the system state used to compute the predicate. Only contextual dispatch permits local frames of reference that conflict with one another. However, as these forms of dispatch increase in power and expressiveness, they become more complex to specify. How sharply this complexity increases, or whether it is outweighed by the benefits gained are matters requiring further study.

Metaphoric polymorphism purports to permit the identification of semantic correspondences between types, as independent from subtyping [Rinat and Magidor, 1996]. This concept was later made concrete (though renamed correspondence polymorphism) in a language called LCP [Rinat et al., 1999]. Their primary example is based upon the perceived similarity of an algorithm for computing powers of an integer with an algorithm for computing powers of a matrix. This idea is a limited form of retroactive abstraction (see Section 3.3): calls in the source text are reinterpreted so that they are sent to the appropriate implementation in the given context. The limitations lie in the fact that there can only be a one-for-one replacement of types and methods: no dynamic selection is possible, and there must still be a globally consistent frame of reference. Metaphoric/correspondence polymorphism is effectively retroactive genericity (see Section 8.4.1).

Another mechanism that can be considered a form of dispatch is implicit invocation (also called publish–subscribe) [Garlan and Notkin, 1991]. Here, a given module provides an interface whereby external modules can register an interest in specific kinds of events as defined by that interface; each external module provides a callback function when it registers. When an event occurs that the module publishes, each external module that has subscribed to this event has its callback function invoked. One can consider this event to be a call that gets dispatched to zero or more method implementations, though without any return values. Implicit invocation requires that the publisher module announce specific events. As such, it must be explicitly aware that these events—and no others—are of potential interest to external modules and announce when they occur; often, this explicit knowledge is EEK with respect to the publisher. On the other hand, implicit context permits the form of the events of interest to be specified externally to a module. Thus, the events published can be
tailored to meet the needs of the system without the need to invasively modify the module. As a result, from the system’s perspective, that module would be providing an implicit invocation interface; from the module’s perspective, it would not.

A framework is a collection of interacting classes, some of which are abstract [Johnson and Foote, 1988]. To reuse a framework, a developer subclasses the abstract classes as appropriate to the particular system being developed. As such, frameworks can be considered highly constrained middleware architectures where components must be added via inheritance (see Section 8.3.2) [Johnson, 1997]. Frameworks do not alleviate us from the difficulties of evolution, since the frameworks themselves need to evolve [van Gurp and Bosch, 2001]. Indeed, this evolution can be particularly difficult since, presumably, existing systems depend on the original interface provided by the framework (or there would be no reason to invest in evolving the framework in the first place).

In short, objects and frameworks are insufficient to support general software reuse or evolution by themselves [Griss et al., 1995]. Implicit context does not reject object-orientation, but can be used orthogonally to it, as has been shown in the case studies of Chapter 6.

8.3.2 Components and Middleware

The term software component [McIlroy, 1969] is used in a variety of ways. One possible definition offered by Szyperski [1997: p. 34] is: “A software component is a unit of composition with contractually specified interfaces and explicit context dependences only. A software component can be deployed independently and is subject to composition by third parties.” Thus, “component” tends to imply more than “module,” in that a module (as defined in this dissertation\(^2\)) can be quite tightly coupled to its system, and not intended for separate reuse. A component differs from a framework in that a framework allows custom modules to be plugged into it (via inheritance) while components are the modules getting composed [Johnson, 1997]. Through essential structure and implicit context every module has the potential to be treated as a component after the fact—consider the case of the Outline View of Eclipse (see Chapter 6).

Components can be made to interoperate in one of two fashions: through “glue” techniques, described in Section 8.3.4; or through standardized interfaces to an architecture that is then used to communicate with other components. These standardized architectures are called middleware.

A difficulty with components is indicated by Johnson [1997: p. 14]: “Each component makes assumptions about its environment. If components make different assumptions then it is hard to use them together.” Others agree [Berlin, 1990; Garlan et al., 1995]. This need to make consistent assumptions naturally leads to the desire for standard environments

\(^2\)See Section 4.1.1.
in which to compose components, and thence, to the standard middleware architectures, described below.

POLYLITH [Purtilo, 1994] provides a "software bus" that allows the specifications of a program's structure, its deployment onto nodes, and inter-module communication to be separated. Simple communication statements within components are adapted to conform to the needs of the actual deployment geometry and communication protocol needs of a heterogeneous architectural- and language-environment. While POLYLITH is an attempt at making modules more flexible by separating the concerns of distributed and inter-process communication, it cannot remove the same degree of EEK as implicit context since it does not permit the indirect kind of communication where, for example, parameters can be filled in from the communication history.

Microsoft’s Component Object Model (COM) [Williams and Kindel, 1994] permits components to communicate through universally-identified, immutable interfaces. When a component increases in functionality, rather than changing the definition of its interface, it defines an additional, new interface that encompasses the new functionality; this is intended to alleviate problems of version-control. Also, when a COM component A possesses an interface pointer to another COM component B, component A can query component B to see if it supports additional interfaces. COM assumes that component functionality never decreases. COM+ [Kirtland, 1997], which automates some of the specification processes of COM, permits the implementor of a class to specify attributes on COM components to express how to initialize system services, how to bind fields to other components, or hints on how to pass parameters. COM+ class specifications are riddled with EEK; such classes require intimate knowledge of COM+ to operate correctly. Neither COM nor COM+ provides a means to reflect upon system history.

The Common Object Request Broker Architecture (CORBA) [Object Management Group, 1998] permits components to request particular, registered services. The broker locates other components that supply these services, and directs the communications appropriately. CORBA IDL context clauses are a means of passing environment variables across a network connection [Baker, 1997: p. 65].

JavaBeans (self-contained, reusable software units that can be composed into composite components in a standardized fashion, and manipulated visually in a builder tool) [Hamilton, 1997] provide BeanContexts [Cable, 1998] to permit an abstraction for the context in which a JavaBean logically functions at run-time. A JavaBean may query this context to determine the availability of services and to subsequently use those services. JavaBeans will therefore be riddled with EEK in support of the BeanContext mechanism; JavaBeans and BeanContexts will also need to agree upon names, semantics, and protocols for services. Although partial system history is represented via the hierarchy of nested BeanContexts and JavaBeans, no other reflection upon system history or state is implicit in the
No matter how standardized the interface provided by one of these approaches, it inevitably needs to evolve. The need to maintain backwards-compatibility to support existing client modules, and to update the interface to accommodate new ideas and new features result in pressure in opposite directions; predicting such changes ahead of time is rarely possible [Wegner, 1996]. As a result, languages evolve, libraries evolve, and standards evolve—or else, they stagnate and are eventually abandoned. Additionally, no single architecture is ideal, or even usable, for all systems. Expecting modules to always plug-in to any architecture like “LEGO blocks” is unrealistic. Most modules place constraints upon their systems in the form of dependences on external modules. The need that a given module may have for certain global properties, such as particular security or fault detection protocols, to be supported by its system cannot be provided by an arbitrary framework [Ran, 1999].

In contrast, implicit context avoids the need for an overarching standard. Individual modules may disagree upon the global perspective, permitting differing interfaces and protocols to be specified for each. As a result, independently developed modules can be composed into a single system without the need for pre-planning.

### 8.3.3 Product Lines

Product lines are a realization of the idea that programs come in families [Parnas, 1976]; each member is a variation on a theme. Griss [2000b] defines a product line to be “a set of products that share a common set of requirements, but also exhibit significant variability in requirements.” Batory et al. [2000] describe product line architectures as one of the primary technologies that addresses software evolution.

The most basic approach to realizing product lines is through customization after the fact. An original product is taken and variants thereof produced. The variants can be created via copy-and-modify, or by restructuring the program to conform to one of the approaches discussed below.

Another basic approach is to configure the program, either via preprocessor statements in the source text [Ernst et al., 2002], or through selection of the particular modules to combine into the executable program [Karhinen et al., 1997]. That a tool as complex as mathematical concept analysis has been proposed simply as a means of understanding possible configurations [Snelting, 1996] is an indication that preprocessor statements are not truly a simple means of configuring product lines.

More sophisticated approaches to supporting product lines have been proposed including the use of aspects [Griss, 2000a,b] (see Section 8.4.4) or mixin layers [Batory et al., 2000] (see Section 8.4.3) to select and compose optional features.

However, the idea of a product line is a goal, not an approach in itself. Thus, a product line can suffer from the same difficulties of evolution as any non-product line program, as
demonstrated in the following quotation [Svahnberg and Bosch, 1999: p. 418]:

It is common that a component implements not only the domain functionality that it is supposed to support, but also a number of facilitating functionalities that are more focused on providing application structure and support. [...] The problem is that such functionality has a tendency to evolve more than the domain functionality [...] If this functionality is embedded in the same component implementations as the domain functionality, not only must all component implementations be altered for the new control functionality, but the maintainers must also manually find their way between domain and control functionality to find out where to make the changes.

Thus, separating the domain and control functionality (which is possible with implicit context) would, ideally, permit the latter to evolve while the former remained oblivious.

8.3.4 Adaptors and Wrappers

A variety of approaches attempt to decouple or reuse modules either through alteration of interfaces, or by adding additional functionality behind an apparently unchanged interface; these approaches tend to lie somewhere between middleware and generative approaches. Encapsulation of legacy software is one promoted application of such techniques [Sneed, 2000]; component interconnection is another Dellarocas [1997].

Interface definition languages, such as IDL [Lamb, 1987], permit the integration of modules through the specification of shared intermediate data representations. IDL permits the refinement of data structure specifications in such a way as to make abstract, generic specifications concrete and specific. For example, a type (known as a structure in IDL) that represents a tree could be described very abstractly in terms of leaf and non-leaf nodes; non-leaf nodes would then be described as recursively containing a sequence of trees. To make this representation concrete, a refinement of the structure could specify that each node contain an additional attribute representing depth, or that the sequence of trees in non-leaf nodes be represented as a linked-list. The resulting specifications are potentially similar to retroactive abstraction and contextual dispatch; the chief differences are that contextual dispatch is dynamic in nature, allowing the mapping of abstractions to be performed according to system state and history, and that IDL only permits refinement, not conflicting world views.

Bart [Beach, 1992] is another software bus. Components are connected via declarations in the glue language SGL, which are compiled to utilize Bart. Data is shared amongst components via implicit invocation (see Section 8.3.1). Although this system is intended to provide component independence, components must agree on the meaning and form of data. Furthermore, the troubles inherent with implicit invocation remain.
Many of the structural design patterns [Gamma et al., 1994] are typical attempts at this sort of wrapping. For example, the Adapter design pattern alters the interface of a class so that it may be used by clients expecting a different one. The Adapter design pattern has the disadvantage that, in its simplest form, it is not transparent to all clients since an adapted object no longer conforms to its original interface. Of course, if clients were to access the object only through the adapted interface, this would not be a problem. Otherwise, multi-way adaptation [Gamma et al., 1994: p. 143] can be used to get around this problem, but it introduces the need for clients to instantiate a different class from the one being adapted, i.e., the source code for the clients must be altered. The Decorator design pattern allows additional responsibilities to be attached to an object dynamically. Since the Decorator design pattern works via delegation, there ensues the notorious problem of ensuring that the object calls its own methods via the decorator—the so-called object schizophrenia problem\(^3\) [Sekharaih and Ram, 2002]. The Facade design pattern hides a complex interface behind a simpler front, the advantage being when a simplified interface will suffice. Facade only permits the removal of interface details without providing higher-level adaptation.

Type adaptation [Yellin and Strom, 1997] provides for a dynamic, protocol-based adaptation between two modules. This adaptation is based on the formal specification of the communication protocols of the two modules being composed. It can possess memory, but cannot make reference to the system at large (to obtain additional parameters, or to determine the recipient of a call), nor to explicit communication history. Implicit context permits a partial, incremental adaptation between multiple modules; unbound interactions can later be adapted through the addition of boundary maps. Communication history permits us to specify partial traces of behaviour; specifying complete state machines is tedious and error-prone, as noted by Yellin and Strom. Implicit context requires formalization on par with the type adaptation work.

Others have promoted the need for connections as first-class entities, existing and/or manipulatable at run-time [Ducasse and Richner, 1997]. However, the mere ability to specify or manipulate such connectors does not prescribe the design of the components.

Frame-based reuse [Bassett, 1997a,b] claims some of the highest rates of successful reuse in industry (e.g., it has been used successfully at Boeing [Illback, 1999]). The concept builds on object-orientation, defining three changes. First, frame hierarchies are organized to define how parts (called frames) combine to make objects; that is, the hierarchy is based on “has-a” relationships. Second, the same hierarchy defines both abstraction and composition relationships. Third, frames define their differences from the frames they abstract, rather than having to inherit entire methods as-is or to deal with massive redundancy. As a result, frame libraries scale more gracefully, since they do not need to have large numbers of classes with tiny differences. Essential structure and implicit context share these goals with

---

3The term appears to have originated from http://www.research.ibm.com/sop/sopcadap.htm.
frame-based reuse; they go further in that frames still require globally-consistent naming and a strict hierarchy. However, implicit context does not provide a simple mechanism by which to use a given module differently in multiple places within the same system; either replication or complicated boundary maps would be needed to mimic this ability of frames. Such a mechanism should be added to implicit context in future; there are no inherent difficulties in so doing.

Binary component adaptation [Keller and Hölzle, 1998] is a technique intended to patch together components that are incompatible due only to weak syntactic constraints, such as different naming schemes. Patching is defined with delta classes: specifications of methods, attributes, and interfaces (in the Java sense) to be added to or renamed in a class. These specifications can include code to be added to perform the patching. As a result, two methods can be matched together even if they have different numbers of parameters, with one important effective restriction: a missing parameter has to be filled in somehow. For example, to identify a method output(PrintStream) in class A with a method println() in class B requires println() to be added to A. Method A.println would then need to delegate to A.output by supplying a default parameter, such as System.out. Unlike with contextual dispatch, this technique provides no means to determine the parameter to be filled-in from the caller's context (i.e., the patch is not dynamic), nor does it automatically permit translation between differing world views.

Flexible packaging [DeLine, 2001] focuses on separating the details about a module's interaction from the module itself. Flexible packaging separates a module's functionality and its interactions, called its packaging, into distinct entities: a ware and a packager. A given ware can be packaged to work in different environments, such as a plug-in for a web browser or a command-line filter. A ware must explicitly define its interaction with packagers, although this interaction is generic. Thus, retroactive abstraction is not possible and a globally-consistent world view is needed.

The goals of implicit context are similar to those of its adaptor and wrapper predecessors in that it allows a component to provide one interface while its clients expect another; as a result, components are reusable without invasive change being needed. However, implicit context is a mechanism that allows greater dissimilarity between the expected and provided interfaces. With implicit context, we are not constrained by the information being directly passed or directly accepted: we may add additional information garnered from the communication history, or discard information not immediately required.

8.4 Generative Approaches

Generative approaches fall into a number of subcategories; surveys are given by Biggerstaff [1998] and Czarnecki and Eisenecker [1999, 2000]. Some approaches described earlier,
such as frame-based reuse (see Section 8.3.4), are also generative in nature. Generic pro-
gramming is considered in Section 8.4.1. I describe the heterogeneous topic of generator-
based approaches in Section 8.4.2. I cover collaboration-based design (in Section 8.4.3)
here, since many adherents to it utilize generic programming or generators. I finish my dis-
cussion of generative approaches with aspect-oriented programming and similar techniques
in Section 8.4.4. Implicit context is itself a generative approach; the most closely related
generative approaches are generic programming and aspect-oriented programming.

8.4.1 Generic Programming

Generic programming [Musser, 1997; Musser and Nishanov, 1997; Jazayeri et al., 1998],
most famously exemplified by the C++ Standard Template Library [Stepanov and Lee,
1995], is concerned with finding the most abstract representation of efficient algorithms.
This goal is similar to that of essential structure, though more narrow in scope. Generic pro-
gramming is realized concretely through templates: generic algorithms expressed in terms
of formal, generic parameters; concrete versions of these algorithms can be generated for
concrete contexts by binding these generic parameters to concrete values or types.

Parameterized programming [Goguen, 1984, 1996] is similar to the template realiza-
tion of generic programming, except that specific constraints (called theories) can be placed
upon the formal parameters. A view expresses explicitly how a given module satisfies a
given theory; it is the binding from one to the other. The work of Novak [1997] is very
similar, though less formal and more applicable to modules at varying granularities.

Generic programming and parameterized programming require pre-planning, since
formal parameters must be explicitly declared. The only means either approach offers to
add arbitrary functionality within a module is through such formals. Modules that are highly
parameterized tend to be difficult to use, since the purpose of most of the parameters is
unclear without a careful examination of the source text of the module, and the majority of
the hooks provided would be rarely needed. Implicit context permits retroactive genericity:
any communication crossing the boundary of module can be mapped as desired. The use of
essential structure ensures the absence of a bewildering array of formals whose purpose is
not apparent.

The functional programming paradigm offers genericity through parameterized types
and monads [Bird, 1998]. While some have promoted the use of functional programming
for reuse [Odersky and Wadler, 1998], it suffers from the same need for pre-planning as do
other genericity mechanisms.

Jiazzi [McDirmid et al., 2001] permits a degree of retroactive genericity by mapping
the names of (unbound) external classes and methods within a Java component to those
extant within a system. However, there is no dynamic mapping possible, no communication
history, and no ability to add arbitrary functionality within a pre-existing module.
8.4.2 Generators

There are a variety of techniques that operate on the principle that programs should be generated from a set of fragments that are composed according to script-based specifications.

A number of older techniques applied the ideas of top-down design and implementation [Wirth, 1971]. Abstract programs were successively refined to add implementation details. Balzer et al. [1976] and Cheatham [1989] describe the idea of using automated transformation to optimize specifications to arrive at concrete implementations. Feather [1989] describes a specific example of such a transformational environment based on a specification language called Gist. It is intended to be used in the fashion of refinement rather than compositionally, as in implicit context. As such, Gist does not suffice to solve the problems of reusing independently developed components, or of easing evolution.

In Draco [Neighbors, 1989] the world is divided into modelling domains (such as networking or databases), each of which is representable in terms of a domain-specific language. The intent is that many large programs are constructed that incorporate ideas from multiple domains; the program modules that deal with a particular domain are best described in terms specific to that domain. To get these modules to interoperate, they are transformed via refinements to lower-level, more common domains. Domains and their transformations are heavyweight constructs; someone must build the refinements for a new domain when needed. Furthermore, the modules written in each domain must maintain a single world view. The domain program fragments cannot overlap in their functionality in an aspect-oriented sense; they can only be interconnected in LEGO block fashion.

GenVoca [Batory and O'Malley, 1992] has been proffered as a technique for composing components into hierarchical software systems. GenVoca's equivalent to Draco's refinements are larger-grained; GenVoca may transform a large set of interdependencies simultaneously, instead of individually, as in Draco. Similarly, my implicit context prototype tool also transforms all the dependencies within a boundary simultaneously according to boundary map specifications. Retroactive abstraction is not possible in GenVoca, nor are non-hierarchical systems specifiable. GenVoca permits modules to be parameterized, so that they may be composed both "vertically" (in class-hierarchy fashion) and "horizontally" (in terms of collaborative behaviour between classes); these latter ideas have been elaborated fully in the mixin layers approach [Smaragdakis and Batory, 2002] and are discussed further in Section 8.4.3.

Generators are a step towards the automated composition of components. However, with only an ability to perform top-down refinement, we must continue to embed extraneous knowledge in our modules. A lack of retroactive abstraction requires some degree of prescience in the design of our components. While extant generator-based techniques have improved reuse, they have do not suffice, nor do they lend themselves to easing evolution.
8.4.3 Collaboration-Based Design

Functional decomposition of systems fell out of favour as a design paradigm with the advent of object-orientation. Nevertheless, the interaction of a set of objects to perform given functionality remains an important design consideration, the obscuration of which has been bemoaned [Berlin, 1990]. As a result, special attention has been paid to the particular collaborations between objects to achieve particular functions; this is known as collaboration-based or role-based design.

A contract [Helm et al., 1990] is a partial specification of interacting classes, their required interfaces and implementations, and any invariants they must maintain. Such contracts can then be refined to fill-in missing details or additional requirements to form new contracts, or sets of classes can be declared to conform to such contracts [Holland, 1992]. Contracts are formal specifications, rather than implementations.

Mixins [Bracha and Cook, 1990; Flatt et al., 1998] encourage code reuse through a form of multiple inheritance. A class can be described as extending a mixin class, thereby inheriting its methods and attributes as in extending non-mixin classes. However, if a class extends both a mixin M and a non-mixin S, then M is considered to subclass S in the context of this part of the hierarchy. As a result, methods of M may make calls to super-methods, and have these calls bind differently depending on where in the hierarchy they have been mixed-in. VanHilst and Notkin [1996b] showed how a set of collaborating mixins could be used to compose a collaborative design into a set of classes; the technique proved difficult to scale as the specification of how to compose the mixins required exceedingly long and cumbersome expressions.

Mixin layers [Smaragdakis and Batory, 1998, 2002] take a set of collaborating mixins and places them in a parameterized module. This is the “horizontal” dimension of composition first described in the GenVoca system [Batory and O’Malley, 1992]. The layers are then stacked vertically by instantiating each with the layer below it; within the definition of the mixin layer lie the more fine-grained bindings between the lower mixin layer (specified as a formal parameter) and the current one. Mixin layers simplified the composition expressions that were a cause of difficulty for VanHilst and Notkin.

These techniques require a high degree of preplanning and standardization. Mixins and mixin layers are dependent upon particular names being present within the vertical dimension of composition. In contrast, implicit context can be used to compose independently developed collaborations, and does not depend upon there being a strict hierarchy.

The idea of considering an object in terms of its individual roles in a system has been addressed (e.g., by Pernici [1990] and by Kristensen and Østerbye [1996]). Little of this work dealt with composing individually developed roles until Kendall [1999] applied aspect-oriented techniques to the problem. I now proceed to discuss these.
8.4.4 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [Kiczales et al., 1997] permits the separation and encapsulation of crosscutting concerns, concerns that affect the implementation of multiple modules within a system, such as synchronization, remote invocation, or fault tolerance. By pulling out these concerns and placing them in their own, special modules (called aspects), the hope is that the core functionality of the modules will be less obscured and more reusable, and that the crosscutting functionality will be more evolvable, understandable, and reusable since it is localized.

AOP is realized in a number of concrete programming languages; currently, the most highly developed of these is AspectJ [Kiczales et al., 2001], an aspect-oriented version of Java. The AspectJ-style of AOP is characterized by the need for a globally-consistent world view, and the ability to perform additional or replacement functionality when particular events occur. Only simple events can be specified. A detailed comparison between AspectJ and implicit context is given in Chapter 7. AspectJ possesses specific modularity constructs (i.e., aspects); the model of implicit context does not prescribe any, and my prototype tool has only weak support (i.e., mapsets). While there have been arguments that AspectJ’s modularity constructs are not sufficient (e.g., by Clarke and Walker [2002]), those of implicit context clearly are not. More work is needed on this issue.

More recently, the term “aspect-oriented programming” has come to be used for a number of approaches related to AOP as originally formulated [Elrad et al., 2001].

Composition filters [Akşit et al., 1992] provide a means for intercepting and rerouting messages at run-time. This permits functionality to be added or replaced when a dynamic event occurs. The original destination for a message must exist, however, in order for the message to be sent, even if it will ultimately be intercepted. Thus, the composition filters approach requires a globally-consistent frame of reference.

Adaptive object-oriented programming (typically shortened to “adaptive programming”) [Lieberherr et al., 1994] supports a particular form of essential structure. It permits modules to possess pattern-based knowledge of the class hierarchy in a system. Such patterns are used to specify where in the system to locate information of interest, while stating as little as possible about the system. Mezini and Lieberherr [1998] demonstrated how adaptive programming can be used to support collaboration-based design. Adaptive programming attempts to specify all support for crosscutting concerns in terms of traversals of the class hierarchy; that this is an effective and comprehensible means of such specification is unclear. It does not provide support for inconsistency in local world views.

Subject-oriented programming [Harrison and Ossher, 1993] allows separate class hierarchies (called subjects) to be composed; messages sent to an object from within one of these subjects can cause the invocation of behaviour within other subjects. This invocation can be in addition to or in place of the behaviour specified in the originating subject; the
choice is based on explicit rules and exceptions thereto that are specified by the developer [Ossher et al., 1996]. However, subjects are quite heavyweight modules. Each must define a self-consistent and complete world view, rather than the sorts of fragments available to implicit context. Also, the use of rules for specifying the composition proved difficult to maintain; an absence of a complete set of rules required the frequent implementation of new rules, which interacted unpredictably.

Theme/UML (earlier called subject-oriented design) [Clarke et al., 1999; Clarke and Walker, 2001] builds on the ideas of subject-oriented programming and generic programming. It permits mixin-like functionality to be defined, but without the need for a consistent world view. Constraints can be placed on the behaviour and interactions of the templated model elements. Clarke and Walker [2002] later showed the potential for such designs to be realized in aspect-oriented implementations, indicating some weaknesses of the AspectJ-style of AOP. Recently, Mezini and Ostermann [2002] have introduced an implementation mechanism that more closely provides Theme/UML-like support.

Implicit context provides a combination of these concepts. Implicit context provides for the separation and encapsulation of crosscutting concerns, although this is not realized in my prototype tool in as clean a fashion as in AspectJ. The AspectJ-style of AOP does not provide for independent world views; as a result, the Abstract Factory design pattern example (described in Chapters 1 and 7) cannot be encoded in it. Contextual dispatch is similar to composition filters, but the latter cannot deal with non-existent entities. Adaptive programming provides for only a limited form of essential structure. Subject-oriented programming provides only heavyweight independent world views and a poorly extensible means of composition.

Hyperspaces [Tarr et al., 1999] are an extension to the ideas of AOP in general, and of subject-oriented programming in particular. They are intended to permit "multidimensional separation of concerns." The idea centres around the proposition that no single modularization exists that will suit all purposes. Instead, sometimes one will desire a functional view of the system and other times one will desire a class-based view, for example. Hyperspaces have been partially realized in the Hyper/J tool: portions of existing systems (i.e., individual Java classes or methods) can be removed and composed together independent from the remainder. However, those pieces have to have been compiled in the first place, so Hyper/J does not support program fragments with independent world views in the manner of my prototype tool for implicit context; thus, it remains a relatively heavyweight solution. Thang and Katayama [2002] have argued that the hyperspace approach (see Section 8.4.4) yields an improvement over mixin layers for the expression of collaboration-based designs because of its ability to reconcile differing world views.

Essential structure goes beyond the separation of crosscutting concerns. A concern can only truly crosscut modules if there are multiple modules to analyze within a system.
If a given concern does not crosscut in one version of a system but does in a later version, it would need to be separated after the fact—refactored—with the attendant difficulties as described in Section 2.3. Essential structure defines what is core and what should be separated from a localized perspective; this locality gives our modules their greatest flexibility and least dependences.

8.5 Other Related Topics

There are a few odds-and-ends topics that are of relevance to this dissertation. In Section 8.5.1, I consider other ideas related to essential structure. Approaches to dealing with context are described in Section 8.5.2. I discuss the support for multiple world views in Section 8.5.3. Section 8.5.4 expounds upon the treatment of names. And finally, Section 8.5.5 looks into other uses of computational history and traces.

8.5.1 Essential Structure

Parnas [1979] describes how the excessive distribution of information makes for difficulties in the extension or contraction of programs. Many of the approaches described above attempt to limit the distribution of knowledge for this reason. Essential structure is a concept that intends to have modules express only the knowledge that absolutely must. Similar but less extreme ideas have been proposed previously.

The idea of specifying abstract modules and refining them is an old one (see Section 8.4.2 for example). Software architectures [Perry and Wolf, 1992; Garlan and Shaw, 1993] describe the highest level of abstraction (and structure) in a system. Implicit context allows for software architecture to be localized: a module describing the software architecture can be devised, and the (abstract) entities named in that module can be mapped to concrete, “lower-level” modules in the system. This is what I did in the implicit context design for the FTP server, described in Chapter 6.

Others have expressed a desire for finer-grained modularity [VanHilst and Notkin, 1996a; Jackson et al., 2001]. This is possible in the essential structure model, where a module can be a program fragment of fairly arbitrary properties. My prototype tool supports method-granularity modules, for example.

Most interestingly, Mätzel and Bischofberger [1997] proposed a model for design that could cope with unanticipated evolution. It consists of a sequence of wrappers around the most core features; the likelihood of evolution diminishes closer to the core. This is very similar, perhaps identical, to the model of essential structure. However, their model is elucidated no further, nor does it appear to have been implemented.
8.5.2 Context

Many approaches have attempted to deal with the context of operation of a system. Most of these consider that the system must actively react to its context, rather than having the context reinterpret the system’s actions. As a result, the context-sensitive modules must be far more complex, but must still evolve if the model of context needs to change.

Adaptive programming [Gouda and Herman, 1991] (not to be confused with the work of Lieberherr and colleagues) is a logic formalizing the changes undertaken by a program reacting to the current state of its environment.

Behaviourally adaptive objects [Lang and Lockemann, 1998] separate objects into two separate, interacting entities: crystals to represent the state of an object, receive messages, and select behaviour, and contexts to define operations. If more than one context is appropriate for the response to a message, the crystal must explicitly order the behaviours it selects and somehow resolve conflicts between them. Contexts are defined across sets of crystals too, tightly coupling them as a result. Thus, behaviourally adaptive objects are fraught with EEK—even more than other approaches due to the tight coupling of crystals.

Context-oriented programming [Gassanenko, 1993, 1998] introduces virtual method tables to bind names to the appropriate functionality in a given context. Similarly, context relations [Seiter et al., 1998] provide a language-based mechanism in support of the Strategy design pattern [Gamma et al., 1994] by allowing “context objects,” basically dispatch tables, to be dynamically attached to instances. Upon receipt of a message by an object, method selection is performed by the context object currently attached to the receiver object. Context reflection [Nakashima, 1992] allows interpretation of messages and knowledge in terms of a current context. This context is defined in terms of a set of context fragments, which are name–predicate pairs, that are explicitly added or removed from the scope, and used to interpret knowledge. None of these techniques directly allow for the kind of dynamic interpretation of events provided by implicit context, nor or any reflection upon the communication history.

Often, “context” is considered as the environment purely external to the system. Dey and Abowd [1999] discuss context in terms of mobile computing where that context is external to the entire system, and capable of rapid and continuous change at run-time. Wermelinger et al. [2002] propose a very specific four-layer architecture for the sake of easing evolution. The layers separate the basic computations, the interactions between the computations, the configurations of feature interactions, and the interactions with the context. Here, context represents actors (in the sense of use cases [Rumbaugh et al., 1991; Object Management Group, 2000]).
8.5.3 Multiple World Views

Other than the work on subject-oriented programming and hyperspaces described in Section 8.4.4, only a few approaches allow multiple world views in any sense at all.

Clovers [Stein and Zdonik, 1989] permit types to possess differing but co-existing “chunks” of state. Each chunk can be used to support different roles. The different roles must be explicitly selected via a coercion operation. This does not allow the level of transparency needed by the independent world views of implicit context.

Balzer [1991] investigated tolerating temporary inconsistency in database systems, formalizing the treatment and resolution of such problems. This is only vaguely related to independent world views, being more of a way to avoid excessive synchronization.

Implicit parameters [Lewis et al., 2000] allow one to explicitly designate when a set of intervening methods between a sender and a receiver do not need to be aware of a set of parameters: the parameters are transferred from the sender to the receiver without alteration of the source code for the intervening methods. The developer specifies within the source code for the sender and for the receiver that each parameter is to communicate via the implicit mechanism. Thus, the sender and receiver must be explicitly aware of the special form of interaction, resulting in extraneous embedded knowledge.

8.5.4 Names

There has been much work that recognizes the significance of names.

Link editors were an early mechanism for mapping names referenced in one module to the actual declarations provided in others [Hamlet, 1976; Gautron and Shapiro, 1987].

Radia [1989] defines a model of contexts and naming in which a context is explicitly selected to interpret names. Thus, globally consistent names are not required. Others have promoted similar systems [Browne et al., 1995].

Bowman et al. [1993] provide a formalism for reasoning about naming systems, specifically in the context of database querying. It is based upon the concept of resolution functions. There is a preference hierarchy of these functions; each is evaluated in turn against the properties of interest, until either a non-empty set of matches is found, or the hierarchy is exhausted. Multiple items may be found that match the query. This is similar to the way that communication history queries must navigate a sequence of traces expressed in terms of independent world views to locate all matching events. Thus, the work of Bowman et al. may be of use in formalizing implicit context.

Kaplan and Wileden [1995] demonstrate the importance and subtlety inherent in naming systems. Their primary example arises from the UNIX tool make. The way that make determines if a given program must be recompiled is independent of whether that program has been changed, because the temporal assertions expressed in make are controlled
by specifications of paths that are independent of the paths used by the compiler. Minor ontological mismatches arising from basic assumptions in independently developed, but interacting systems can lead to unpredictable results.

Implicit context is more than just changing names, since dynamic target selection and arbitrary functionality can result from an external name reference.

8.5.5 History and Traces

The concepts of communication history and event traces are fairly widespread. However, they have largely been used for the sake of specifications, rather than implementations.

Trace assertion [Bartussek and Parnas, 1978; Wang and Parnas, 1994; Janicki and Sekerinski, 2001] places constraints upon the observable behaviour of modules. Communication history queries are effectively trace assertions specified about the system from a localized (internal) perspective. Similarly, state abstraction [Hoffman and Strooper, 1995] has been promoted as a means of specifying behaviour.

Gist [Feather, 1989] is a specification language used in refinement-based transformations (see Section 8.4.2). Among other constructs, Gist supports historical references to extracting information from any preceding state in the computation. The support for the implementation of such historical references is proposed via a high-level description that is similar to the discussion in Chapter 7. Certain patterns of historical reference occur frequently in Gist specifications, just as they do in implicit context; the support for these patterns are optimized. Such historical references are used in other specification techniques (e.g., that of Broy and Stølen [2001]).

Traces [Kiczales, 1993] allow the interpretation of messages to be altered based upon a limited form of explicit context. A list of "ancestor classes," which are independent of the class hierarchy, may be explicitly built and attached to an object. Also attached to each object is a set describing patterns of ancestor classes. When a message is received by an object, the choice of method to invoke is determined by which pattern the ancestor list of that object matches. The ancestor lists of traces can be thought of as particular paths through the communication history tree, permitting a limited means of reflecting upon the system history.

Program annotations have been used to perform event sequence analysis [Howden and Shi, 1996]. This is used in verification, rather than implementation.

Essential structure and implicit context are intended to blur the boundaries between specification, design, and implementation. Similar to older approaches, implicit context permits refinement of details, but also their replacement and intermingling in a non-hierarchical fashion. Communication history queries are a convenient means of specifying abstract constraints on state with a minimum of dependences.
8.6 Summary

There has been much previous work that has either attempted to address the problems of software evolution and software reuse, or that touches directly upon the ideas used by essential structure and implicit context.

Essential structure is a specific kind of separation of concerns, minimizing the dependencies of a module on its external system. Essential structure promotes the principle of local reasoning. Implicit context supports the mapping of modules, produced via local reasoning, to the global level. That this mapping is (or can be made to be) sound in some sense remains to be demonstrated. Essential structure rejects the notion that it is necessary to predict change to a system or module; design-for-change that consists of good coding practices suffices.

Essential structure and implicit context allow us to avoid problems of code replication or constraints of published interfaces that occur in ad hoc approaches. Problems encountered in various approaches to standardizing interfaces and architectures, where the standards themselves must evolve or stagnate, are avoided through the provision of independent world views. However, implicit context can work in concert with such standards, permitting the localized use of those standards within particular modules.

Essential structure and implicit context have much in common with various adaptive and generative techniques. None of these provide the combination of avoidance of predicting change, retroactive genericity, communication history querying, and reconciliation of conflicting world views that I have argued is useful in easing evolution and reuse.

This is not to say that essential structure and implicit context should be adopted tomorrow. There are tradeoffs in applying any given technique. Exactly where the costs of using essential structure and implicit context, rather than some other technique, outweigh the benefits must be studied further.
Chapter 9

Conclusion

When dependences upon the context of operation creep into a module, the module becomes harder to evolve and harder to reuse. When these dependences are not strictly necessary, the module contains extraneous embedded knowledge. A module lacking extraneous embedded knowledge possesses only essential structure.

Implicit context is a model combining the concepts of boundaries, contextual dispatch, and communication history. Boundaries separate independent and mutually inconsistent world views. Contextual dispatch translates the communications passing between independent world views. Communication history is queried to locate information that is used in the translation process.

The thesis of this dissertation is that expressing the essential structure of our software modules, through the use of implicit context, makes those modules easier to reuse and the systems containing those modules easier to evolve. I conducted two case studies to provide evidence in support of this thesis.

In the first case study, I removed the Outline View of the Eclipse IDE and reused it as a generic widget in a different system of markedly different character—without reusing the remainder of Eclipse. By using implicit context to express the emergent essential structure of the Outline View, I was able to cancel out its dependences upon the rest of Eclipse; these dependences were extraneous embedded knowledge (EEK) within the Outline View. The Outline View was not modularized in a fashion amenable to its reuse; the designers of Eclipse did not plan for it to be reused in this way. Nevertheless, implicit context allowed for its disciplined reuse—reuse that would not have been possible when limited to the state-of-the-art, without ad hoc modification of the source text.

In the second case study, I designed and implemented parallel versions of a File Transfer Protocol (FTP) server, one version using only object-orientation (OO) and one version using object-orientation plus implicit context. I then evolved these versions, each using the same approach with which it was originally constructed. The original version consisted of a Minimum Implementation as defined by the FTP specification; my designs ignored the
existence of the remainder of the FTP specification. I had colleagues vet the initial OO design to ensure that it was reasonable. I selected two features from the remainder of the FTP specification, authentication and passive mode, and evolved the versions to include them. In addition, we “cross-evolved” the initial OO version using implicit context.

While all three evolved versions were able to incorporate the new features, the OO version required extensive invasive modifications. The versions evolved through implicit context were able to avoid all but one invasive modification, the latter due to a lack of support in my prototype tool. The cross-evolved version demonstrates that implicit context can be adopted after the initial construction of a system. However, only the modules of the version initially constructed with implicit context were maximally free of EEK. Future evolution of these systems, inevitable in the real world, would require more and more invasive modifications—at great difficulty and risk—of the versions not initially constructed with implicit context. Some EEK remained in the cross-evolved version since we only cancelled out the minimum amount of EEK that had been hindering the immediate evolution step being performed.

With essential structure through implicit context, these cases of software reuse and software evolution were possible and practical without the need for code replication, preplanning, or invasive modification. In other words, evolution and reuse were made easier.

9.1 Contributions

This dissertation makes three major contributions.

First, it has identified the concepts of essential structure and extraneous embedded knowledge (EEK). I have shown how the presence of EEK acts as a barrier to software evolution and software reuse.

Second, it has identified the concept of localized, independent world views for our modules. Localized, independent world views permit each module to express only its essential structure; as a result, independently developed modules can be composed even if they make conflicting assumptions about their external context.

Third, it has described the concrete design and use of a tool for expressing essential structure through implicit context. The combination of independent world views, contextual dispatch, and communication history is unique. This combination allows resolution of the conflicts between the assumptions made by the modules of a system. Thus, modules are made highly reusable even when they were not designed to be so. By using this combination to permit the expression of only essential structure within our modules, evolution of the system is more likely to proceed without the need to modify most modules.
9.2 Future Work

While I have demonstrated potential in the use of essential structure through implicit context, much work remains before I would consider promoting its industrial adoption. This future work falls under three topics to be addressed in the near-term: communication history, tool support, and formalization. In addition, a number of longer-term issues exist.

As I have indicated throughout this dissertation, the implementation of communication history that I have provided in my prototype tool is too simplistic for the general support of implicit context. Chapter 7 has described how the instrumentation to support communication history queries can be minimized. It will also be necessary to determine how one might practically differentiate, in a given boundary map, between the communication history as viewed locally within a boundary and as viewed externally to that boundary. Then, the appropriate queries can be specified in order to translate between the world views on either side of the boundary. Finally, an implementation providing cleaner syntax for the expression of communication history queries will be needed.

Following on this line of inquiry, better tool support is needed. The prototype tool only weakly provides the ability to transform one localized world view to another. The prototype also lacks the ability to reuse multiple versions of a given module without its explicit replication. A facility to define new modules as transformations of old ones—while maintaining the use of the original as well—is needed. Such a facility would also aid in ensuring the comprehensibility of boundary map transformations: a set of transformations could be applied to a given module to obtain a different and explicit module. Currently, the results of applying one boundary map at a time are implicit and difficult to understand. These ideas have been described and motivated in more detail in Sections 5.6, 6.1.2, and 6.2.3.

It would have been premature to formalize the ideas in this dissertation before this point, the concepts to be formalized were unknown at that time. With the advent of this dissertation, the concepts have been identified. Formalization could aid us in assuring consistency and completeness of the facilities to be provided by implicit context tools. In order for us to have confidence that local reasoning about modules is sound, we need to have a means of ensuring that important constraints are maintained as we map local views to the global view. As a first step, I am currently engaged in investigating a representation of dependences that differentiates between necessary dependences and extraneous embedded knowledge. Tying the results of that investigation into an existing approach to formal requirements specifications (that of Jackson and Jackson [1996] looks like a strong candidate) is a possible road towards formalization. In addition, work on formal verification (e.g., by Klarlund et al. [1995]), temporal logics (e.g., by Emerson [1996]), and flow expressions [Shaw, 1978] has potential applicability here.

A number of other, longer-term topics for research in this area exist. While this dissertation has concentrated on software evolution and software reuse, benefit is likely to
accrue from using essential structure in the initial synthesis of our systems. Demonstrating such benefit will require the development of better tools first, since existing techniques for software synthesis are stronger than their counterparts for evolution and reuse.

The queries one can make upon communication history are fairly arbitrary. This raises the question of the interaction of implicit context and security concerns. One might want to force certain modules to be pure black boxes, with only external behaviour visible. While this is strictly true of any module under the definitions of implicit context, one is free to define smaller grained modules; for example, if the behaviour of a class that is externally visible at its boundary is too coarse, one can define boundaries around individual methods within it. Limiting the sorts of queries that could be made would have the effect of limiting the sorts of transformations possible between world views. The significance of such limitations remains to be investigated.

Implicit context as presented in this dissertation depends on static analysis and optimization to be practical. It is not clear that these sort of optimizations will work as desired in the presence of systems that evolve at run-time, for example, ones that dynamically load libraries. Of course, it is not even clear whether the existence of essential structure and implicit context might alter the need for or nature of run-time evolution. This area remains a big unknown.

Essential structure through implicit context is a disruptive technology [Christensen, 1997]. It has the potential to change the ways we develop and modify software in fundamental ways. Evaluating its cost–benefit tradeoffs is, therefore, difficult. One cannot simply throw the technology at expert developers and expect to get meaningful results. Aside from the masking effects of the low usability of my prototype tool, the very disruptiveness of the technology greatly complicates attempts at measurement. When a technology changes the nature of the tasks we perform, the systems we operate upon, and our mental models, we are all novices. Therefore, determining if the use of that technology has positive effects requires time and experience. Qualitative results are a good first step.
Bibliography


195


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, United States: Addison–Wesley. 4, 17, 27, 48, 65, 104, 120, 173, 179, 187, 223


202


205


207


209
David R. Musser and Gor V. Nishanov (1997). A fast generic sequence matching algorithm. Technical Report 97-11, Computer Science Department, Rensselaer Polytechnic Institute, Troy, United States. 9, 181


D. L. Parnas (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12), 1053–1058. 1, 8, 45, 169, 170, 173


Pierre N. Robillard (1986). Schematic pseudocode for program constructs and its computer automation by SCHEMACODE. *Communications of the ACM* 29(11), 1072–1089. 40


Appendix A

A Brief Overview of Notation

Here, I describe the notation that appears in figures in the dissertation. It uses a modified version of the Unified Modeling Language™ [Object Management Group, 2000].

A.1 Square Boxes

Square boxes generally represent types, as in Figure A.1; the name of the type appears within the box. Attributes and methods of types can appear within the box representing the type, in which case, the box is divided into three sections: the top section contains the name of the type, the middle section contains attribute information, and the bottom section contains method information.

The visibility of attributes and methods can be indicated; a "+" means that the attribute or element is public, a "—" means that it is private, and "#" means that it is protected. Additional attributes can be shown as lines or arrows; see Section A.3 for details. Static methods and attributes are indicated by underlining their name.

![Square Boxes](image)

(a) (b) (c) (d)

**Figure A.1:** Square boxes denoting types. A type can either (a) provide no details of its properties or (b) describe its attributes and methods. A heavy outline and boldface name (c) represent an active class (a UML term denoting a class that runs in its own thread). A dotted outline (d) represents a referenced type that does not really exist.
Different box outlines are used to indicate different things. A heavy outline and boldface type name indicate that the type is an active class, i.e., it runs in its own thread of control. A dotted outline is used to indicate a type that is not really present—some module makes reference to it, but this reference is remapped to a different type.

I also use square boxes to represent instances, in which case, the name of the type is preceded by a colon and is underlined, as shown in Figure A.2. Occasionally, I use square boxes to also represent other kinds of modules, such as methods, in which case the name of the module appears just outside the edge of the box. And finally, boundaries are also represented as square boxes, with dashed lines, that surround other types. These are always shown with either a rounded box or circle breaking the line to visually suggest that they are a boundary.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig_a2.png}
\caption{Square boxes denoting instances and boundaries. An instance (a) is shown by prefacing the name of its type with a colon and underlining it. A boundary is shown as a dashed square box surrounding other modules with either (b) a circle or (c) a rounded box breaking its line.}
\end{figure}

### A.2 Rounded Boxes

I use rounded boxes, illustrated in Figure A.3, for two purposes. Mapsets (see page 67) are shown as a rounded box with a solid edge; this box is embedded in the edge of a dashed, square box representing a boundary. In a few places, I explicitly represent a minimal specification (see Section 3.1) as a rounded box with a dashed edge; the name of the module for which it is a minimal specification appears just outside the edge of the box.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig_a3.png}
\caption{Rounded boxes denote either (a) mapsets or (b) minimal specifications.}
\end{figure}
### A.3 Lines and Arrows

Lines and arrows of various sorts are used to represent relationships between elements.

A solid line without any arrowheads between two types denotes a generic relationship. Such a relationship can have certain properties associated with one or both ends; these adornments indicate attributes of one of the types in question. For example, consider Figure A.4. This diagram indicates that `SomeType` possesses an attribute called `otherType` of type `OtherType`, and that this attribute has private visibility. The multiplicity of the attribute can also be indicated; in this example, the asterisk indicates that many instances of `OtherType` are associated with each instance of `SomeType` through the `otherType` attribute.

![Figure A.4: Attribute relationships are indicated with solid lines lacking arrowheads.](image)

More specific relationships can be indicated between instances of a class and the parts of each, including aggregation (shown with an empty diamond) and composition (shown with a filled diamond); see the UML specification [Object Management Group, 2000: §3.46] for a discussion of the subtle difference between aggregation and composition. In Figure A.5, instances of `SomeType` compose instances of `OtherType`; these composed instances are accessible via the attribute `otherType`.

![Figure A.5: A composition relationship between classes is indicated with a filled diamond at the aggregate's end of the relationship.](image)

Subtyping and inheritance relationships can also be indicated between types, as shown in Figure A.6. Generalization of interfaces and classes is indicated as an arrow, with an unfilled triangular arrowhead, pointing from the more specialized to the more generalized type. When a class implements an interface, this is indicated by dashing the line of the arrow.
Figure A.6: Subtyping relationships are indicated with an unfilled triangular arrowhead pointing to the more general type. A subinterface or subclass (a) is indicated with a solid line for the arrow; a class that implements an interface (b) is indicated with a dashed line for the arrow.

Flow of control is represented as solid lines with filled triangular arrowheads, as shown in Figure A.7. Annotations can be added to these arrows to indicate message passing. The sequence of messages is indicated by numbering them. When the details do not matter for a particular example, I do not differentiate between messages sent from particular instances versus messages sent from a static scope. In some circumstances, I annotate messages with variable names to allude to message arguments; in other circumstances, I indicate dependences on instances to allude to message arguments.

Dependences of various forms are shown as dashed lines between the elements related by the dependence, as shown in Figure A.8; often, there is an arrowhead on this line pointed

Figure A.7: Control flow is indicated by solid lines with filled, triangular arrowheads. Specific messages (a) can be shown as annotations on these arrows; numbers indicate the sequencing of messages. I also indicate message arguments (b) as dependences between messages and instances.
Figure A.8: Dependence relationships are indicated with a dashed line between elements. An arrowhead (a) indicates a directional dependence. An annotation in braces (b) indicates a constraint. An annotation in guillemets (c) indicates a stereotyped dependence.

at the element upon which there is the dependence. Specific kinds of dependences are indicated by annotations on the relationship: constraints between elements with an annotation surrounded by braces, and stereotypes (such as «similar») are indicated as one or more words bracketed with guillemets.

A.4 Filled Circles

Filled circles are used to represent places where contextual dispatch (see Section 4.1.2) occurs, as shown in Figure A.9. These are always found embedded in the edge of a boundary.

Figure A.9: Filled circles denoting message interception. The original, supposed route of the message and its recipient are shown dotted.
A.5 Jagged Polygons

Polygons vaguely reminiscent of sunbursts are used to indicate call history (see Section 4.1.3), as shown in Figure A.10.

Figure A.10: Jagged polygon denoting call history.
Appendix B

A Brief Overview of the Abstract Factory Design Pattern

In this appendix, I describe the Abstract Factory design pattern in brief. This description will not cover all the details or aspects of this design pattern, merely the ones of importance to this dissertation. The reader who is interested in all the details should refer to the book by Gamma et al. [1994]; quotations are taken from that source.

The intent of the Abstract Factory design pattern is to “provide an interface for creating families of related or dependent objects without specifying their concrete classes.” Among other properties, this design pattern allows most modules in a system to be independent of the means of creation of particular “product” classes.

Figure B.1 shows the static structure of this design pattern. This structure is represen-

![Figure B.1: Static structure of the Abstract Factory design pattern.](image)
ative of the kind of structure present when Abstract Factory is in use; this exact structure is unlikely to be. There are five kinds of participant classes involved in this pattern.

AbstractFactory provides an abstract interface for creating instances of abstract products. Each ConcreteFactory is responsible for implementing the operations that create instances of concrete products. In the static structure shown, the sets of concrete products created by each concrete factory are disjoint; ConcreteFactory1 creates instances of ProductA1 and ProductB1 while ConcreteFactory2 creates instances of ProductA2 and ProductB2.

Each AbstractProduct provides an interface that hides the differences in implementation for a set of concrete products. Each ConcreteProduct implements its corresponding abstract product interface, and is instantiated by its corresponding concrete factory.

A Client may then be oblivious to the details of differences between the sets of concrete products. If it has been given an instance of a concrete factory, it may interact with this instance through the interface provided by AbstractFactory to create (abstract) products. The Client may then interact with the created instance of the concrete product via the appropriate AbstractProduct interface, oblivious to which particular subclass it is.
Appendix C

Listing of makeContributions

Below is the complete text of the makeContributions method from the JavaOutlinePage class of Eclipse. It has been pretty-printed. This code is discussed in Chapter 2 in depth.

```java
public void makeContributions(IMenuManager menuManager,
                               IToolBarManager toolBarManager,
                               IStatusLineManager statusLineManager) {
    if (statusLineManager != null) {
        StatusBarUpdater updater =
            new StatusBarUpdater(statusLineManager);
        addSelectionChangedListener(updater);
    }

    Action action = new LexicalSortingAction();
    toolBarManager.add(action);

    action =
        new FilterAction(new FieldFilter(),
                        JavaEditorMessages.getString(  
                            "JavaOutlinePage.HideFields.label"),
                        JavaEditorMessages.getString(  
                            "JavaOutlinePage.HideFields.description.checked"),
                        JavaEditorMessages.getString(  
                            "JavaOutlinePage.HideFields.description.unchecked"),
                        JavaEditorMessages.getString(  
                            "JavaOutlinePage.HideFields.tooltip.checked"),
                        "There were some uninformative comments of the form "/ /\$NON-NLS-1\$" in various places, presumably for supporting some development tool. I have removed these.

    String concatenation has been used to break very long string literals in a few places. None such appeared in the original text.
```

---

1 There were some uninformative comments of the form "// $NON-NLS-1$" in various places, presumably for supporting some development tool. I have removed these.

2 String concatenation has been used to break very long string literals in a few places. None such appeared in the original text.
JavaEditorMessages.getString(
    "JavaOutlinePage.HideFields.tooltip.unchecked"),
    "HideFields.isChecked";
JavaPluginImages.
setImageDescriptors(action, "lcl16", "fields_co.gif");
toolBarManager.add(action);

action =
    new FilterAction(
        new VisibilityFilter(VisibilityFilter.NOT_STATIC),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideStaticMembers.label"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideStaticMembers.description.checked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideStaticMembers.description." +
            "unchecked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideStaticMembers.tooltip.checked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideStaticMembers.tooltip." +
            "unchecked")
    );
JavaPluginImages.
setImageDescriptors(action,
    "lcl16",
    "static_co.gif");
toolBarManager.add(action);

action =
    new FilterAction(
        new VisibilityFilter(VisibilityFilter.PUBLIC),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideNonePublicMembers.label"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideNonePublicMembers.description." +
            "checked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideNonePublicMembers.description." +
            "unchecked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideNonePublicMembers.tooltip.checked"),
        JavaEditorMessages.getString(
            "JavaOutlinePage.HideNonePublicMembers.tooltip." +
            "unchecked")
    );
"unchecked"),
"HideNonePublicMembers.isCheckedException";
JavaPluginImages.
setImageDescriptors(action, "lcll16", "public_co.gif");
toolBarManager.add(action);
}
Appendix D

The Application Programming Interface to Call History

The implementation of the query interface to call history is described at a conceptual level in Section 5.3.5. Here, I describe its application programming interface (API). Section D.1 describes the API to the History class and Section D.2 describes the API to the Call class.

D.1 The History Class

The API for the History class consists of seven query methods. These are described on the following pages as indicated in the table below. All these methods are internally synchronized to operate in a multithreaded environment.

<table>
<thead>
<tr>
<th>Query method</th>
<th>page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstCall</td>
<td>229</td>
</tr>
<tr>
<td>lastCall</td>
<td>232</td>
</tr>
<tr>
<td>lastCallAnySubclass (2 forms)</td>
<td>230, 231</td>
</tr>
<tr>
<td>lastCallInCFlow</td>
<td>233</td>
</tr>
<tr>
<td>lastCallReturningClass</td>
<td>230</td>
</tr>
<tr>
<td>lastInstancePassed</td>
<td>230</td>
</tr>
<tr>
<td>mostRecentCall</td>
<td>230</td>
</tr>
</tbody>
</table>
firstCall

**public static Call firstCall(Thread thread,**
Class cls,**
String mth,**
String[] paramTypes,**
Object returnVal,**
Call relativeTo)**

Searches for and returns the earliest call conforming to the search parameters passed as arguments; returns **null** if no call can be found that matches these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>The thread of control whose tree should be searched, in the case of causal searches; or <strong>null</strong> if temporal searches should be performed</td>
<td>current thread</td>
</tr>
<tr>
<td>cls</td>
<td>The exact class on which the executed method was defined; or <strong>null</strong> if the class is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>mth</td>
<td>The name of the method that was executed; or <strong>null</strong> if the method is not of interest</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>paramTypes</td>
<td>An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or <strong>null</strong> if the parameter types are not important</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>returnVal</td>
<td>The object that must have been returned by the method execution; or <strong>null</strong> if the returned object is not of interest</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>relativeTo</td>
<td>The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters</td>
<td>mostRecentCall()</td>
</tr>
</tbody>
</table>
**mostRecentCall**

**public static Call mostRecentCall()**

Searches for and returns the most recently occurring call of any form.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Takes no parameters</td>
<td></td>
</tr>
</tbody>
</table>

**lastInstancePassed**

**public static Object lastInstancePassed(Class cls)**

Searches for and returns the object of type `cls` that was most recently passed as a target or argument to some call; returns `null` if no such call can be found.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cls</code></td>
<td>The class of which type to search for the most recently passed instance</td>
<td>—</td>
</tr>
</tbody>
</table>

**lastCallReturningClass**

**public static Call lastCallReturningClass(Class cls)**

Searches for and returns the most recently occurring call that returns an object of a type that is assignable to the class passed in `cls`; returns `null` if no call can be found that matches these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cls</code></td>
<td>The class to which the type of the returned object is assignable; or <code>null</code> if the class is not of interest</td>
<td>—</td>
</tr>
</tbody>
</table>

**lastCallAnySubclass**

**public static Call lastCallAnySubclass(Class cls)**

Searches for and returns the most recently occurring call on any class that is assignable to the class passed in `cls`; returns `null` if no call can be found that matches these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cls</code></td>
<td>Some superclass of the class on which the executed method was defined; or <code>null</code> if the class is not of interest</td>
<td>—</td>
</tr>
</tbody>
</table>
**lastCallAnySubclass**

**public static Call lastCallAnySubclass(Thread thread,**

`Class cls,`

`String mth,`

`String[] paramTypes,`

`Object returnVal,`

`Call relativeTo)`

Searches for and returns the most recently occurring call on any class that is assignable to the class passed in `cls`; returns `null` if no call can be found that matches these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>The thread of control whose tree should be searched, in the case of causal searches; or <code>null</code> if temporal searches should be performed</td>
<td>—</td>
</tr>
<tr>
<td>cls</td>
<td>Some superclass of the class on which the executed method was defined; or <code>null</code> if the class is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>mth</td>
<td>The name of the method that was executed; or <code>null</code> if the method is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>paramTypes</td>
<td>An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or <code>null</code> if the parameter types are not important</td>
<td>—</td>
</tr>
<tr>
<td>returnVal</td>
<td>The object that must have been returned by the method execution; or <code>null</code> if the returned object is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>relativeTo</td>
<td>The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters</td>
<td>—</td>
</tr>
</tbody>
</table>
public static Call lastCall(Thread thread,
        Class cls,
        String mth,
        String[] paramTypes,
        Object returnVal,
        Call relativeTo)

Searches for and returns the most recently occurring call conforming to the search parameters passed as arguments; returns null if no call can be found that matches these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>The thread of control whose tree should be searched, in the case of causal searches; or null if temporal searches should be performed</td>
<td>current thread</td>
</tr>
<tr>
<td>cls</td>
<td>The exact class on which the executed method was defined; or null if the class is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>mth</td>
<td>The name of the method that was executed; or null if the method is not of interest</td>
<td>—</td>
</tr>
<tr>
<td>paramTypes</td>
<td>An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or null if the parameter types are not important</td>
<td>—</td>
</tr>
<tr>
<td>returnVal</td>
<td>The object that must have been returned by the method execution; or null if the returned object is not of interest</td>
<td>null</td>
</tr>
<tr>
<td>relativeTo</td>
<td>The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters</td>
<td>mostRecentCall()</td>
</tr>
</tbody>
</table>
lastCallInCFlow

**public static Call lastCallInCFlow(Thread thread,**
Class cls,**
String mth,**
String[] paramTypes,**
Object returnVal,**
Call cflowOf)**

Searches for and returns the most recently occurring call that occurred as a child of the
call in cflowOf, also conforming to the other search parameters passed as arguments; 
returns **null** if no call can be found that matches these parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>The thread of control whose tree should be searched, in the case of causal searches; or <strong>null</strong> if temporal searches should be performed</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>cls</td>
<td>The exact class on which the executed method was defined; or <strong>null</strong> if the class is not of interest</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>mth</td>
<td>The name of the method that was executed; or <strong>null</strong> if the method is not of interest</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>paramTypes</td>
<td>An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or <strong>null</strong> if the parameter types are not important</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>returnVal</td>
<td>The object that must have been returned by the method execution; or <strong>null</strong> if the returned object is not of interest</td>
<td><strong>null</strong></td>
</tr>
<tr>
<td>cflowOf</td>
<td>The call whose children are to be searched for a match</td>
<td><strong>null</strong></td>
</tr>
</tbody>
</table>
D.2 The Call Class

The History class possesses a helper class, called Call, that provides an interface to the values related to specific calls.

getMethodName

public java.lang.String getMethodName()

Returns the name of the method that was invoked by this call, as determined by the tool's instrumentation (see Appendix E).

getThread

public java.lang.Thread getThread()

Returns the thread instance in which this call was invoked.

getReturnValue

public java.lang.Object getReturnValue()

Returns the result value returned by this call, as an Object (meaning that primitives are wrapped in instances of their corresponding classes), or null if the call has not returned or has a result type of void.

getTarget

public java.lang.Object getTarget()

Returns the instance upon which this call was invoked, or null if the invoked method was static.

getTargetClass

public java.lang.Class getTargetClass()

Returns the class upon which the method was declared that was invoked by this call.

getParameter

public java.lang.Object getParameter(int i)

Returns the i-th argument that was passed in this call, as an Object (meaning that primitives are wrapped in instances of their corresponding classes).

getPredecessor

public Call getPredecessor()

Returns the call that temporally preceded this one.
getParent

**public** Call getParent()

Returns the call that invoked the method that sent this call.

precedes

**public boolean** precedes(Call other)

Determines whether this call temporally precedes the call passed in other.
Appendix E

Example Results of Applying the Prototype Tool

This appendix illustrates the results of applying the prototype tool to the example described in Section 5.5. Section E.1 shows the inputs to the tool and Section E.2 shows its outputs.

E.1 Before Applying the Tool

The tool has four inputs, two Java source files (SomeClass.java and OtherClass.java), one mapset file (example.map), and one tool directive file (apply.app). The tool makes use of the three letter suffix of each filename in determining the kind of its contents.

E.1.1 Contents of SomeClass.java

SomeClass is merely a client of OtherClass, instantiating it, and calling its method1 three times.

```java
public class SomeClass {
    public static void main(String[] args) {
        OtherClass obj = new OtherClass();
        for(int i = 0; i < 3; i++)
            obj.method1();
    }
}
```

E.1.2 Contents of OtherClass.java

OtherClass possesses two methods: method2, which prints the length of a string; and method1, which simply delegates to method2.
public class OtherClass {
    public void method1() {
        method2();
    }

    private void method2() {
        String s = "a string";
        System.err.println(s.length());
    }
}

E.1.3 Contents of example.map

The example mapset contains two boundary maps. The first of these is responsible for causing method2 to be invoked at most once. If it finds that method2 has been called before, it prints a string to this effect. In addition, it retrieves the last call to String.length.

mapset example {
    void map() : in(method1()) {
        Call call = History.lastCall(null, "method2", null);
        if (call == null)
            CONTEXT.proceed();
        else
            System.err.println("method2() called before");

        Call call2 =
            History.lastCall(String.class, "length", null);
        System.err.println(call2 != null);
    }

    void map() : in(method2()) {
        CONTEXT.proceed();
    }
}

E.1.4 Contents of apply.app

The tool directive causes the example mapset to be attached to the boundary of OtherClass.

apply example to OtherClass;
E.2 After Applying the Tool

When the prototype tool is run on the four inputs in Section E.1, it produces two new Java source files, one for `SomeClass` and one for `OtherClass`; see Section 5.2 for an overview of the operation of the tool. So as not to overwrite the original source files, these are stored in a subdirectory of the current working directory called `icworkingdir`.

The code shown in the two subsections below has been cleaned up a bit in two ways for the sake of readability. It has been pretty printed, and the qualifiers for several packages have been elided, namely those for `ca.ubc.cs.se.context.instrument` (the package for the instrumentation classes, including `History` and `Call`) and `java.lang`.

The instrumentation uses a few classes not in the abstract programming interface for call history queries; information that remains the same for each invocation of a given method or constructor is stored in a single object that is reused for each invocation. `CallReceptor` is used for such recurring information, while `CallState` and `ReturnState` are invocation-specific. `CallReceptor` is used to represent the class, method, formal parameter types, and formal parameter names of each method. `CallState` records the object references and primitive values passed to an invocation. `ReturnState` records the result value, be it an actual object reference or primitive value that is returned, or be it an exception that is thrown. Both `CallState` and `ReturnState` record a reference to the target object on which the method is invoked.

`CallReceptor` instances are created and initialized in a `static` initializer for each class. An instance of `CallState` and `ReturnState` are created at the entry to each method or constructor. The entry to the method is then recorded in call history.

The general structure of the instrumentation is shown in Figure E.1. The original body of the method is placed within the body of a `try` statement, so that any exceptions it may throw can be recorded in the `ReturnState` instance for that invocation. The complex set of tests in the `catch` clause for each method is necessary because declared exception types must be differentiated from errors and exception types that do not need to be declared. The base class for all exceptions, `Throwable`, is unfortunately an exception type that must be declared; hence, an exception caught as a `Throwable` cannot be simply rethrown. A `finally` clause is used to ensure that the method exit is recorded in call history regardless of whether an exception has occurred. The original exception is rethrown without alteration.

When a boundary map affects an existing method, the original method body is moved to a new method named `__CONTEXT$inner` followed by a number. The body of the boundary map is placed in a new method named `__CONTEXT$map` followed by a number. The original body of the affected method is replaced by a delegating call to the boundary map method. Any calls to proceed within the boundary map are replaced with calls to the

---

1Just after any explicit constructor invocation, if present, in the latter case.
Original method or constructor header {
    [Original explicit constructor invocation, if present]
    CallState instantiation
    ReturnState instantiation

    Addition of target instance and arguments to CallState object
    Addition of target instance to ReturnState object
    History._callReceived(CallReceptor for this method,
                             CallState object);

    try {
        Original method body, save that return statements are instrumented
    }
    catch(Throwable __CONTEXT_exception) {
        Recording of exception in ReturnState object
        Testing for each exception type that this method declares it throws
        if(__CONTEXT_exception instanceof RuntimeException)
            throw (RuntimeException)__CONTEXT_exception;
        throw (Error)__CONTEXT_exception;
    }
    finally {
        History._returnFromCall(CallReceptor for this method,
                               ReturnState object);
    }
}

Figure E.1: General structure of instrumentation for methods and constructors.
new method containing the original method body. Any calls to the affected method that exist within the affected boundary are replaced with calls to the new method containing the original method body. Thus, an invocation of method\(1\) results in an eventual delegation to the original body of method\(2\) (now contained in \_CONTEXT$\_inner1\) and not the new body; this causes the first problem noted in Section 5.5.

Note that, aside from the static initializer, inner methods, and map methods, a default constructor has to be added if it is not explicitly declared, in order that it can be instrumented.

E.2.1 Contents of icworkingdir/SomeClass.java

```java
public class SomeClass {
    private static CallReceptor __CONTEXT_callReceptor0;  
    private static CallReceptor __CONTEXT_callReceptor1; 

    static {
        __CONTEXT_callReceptor1 =
            new CallReceptor(SomeClass.class,  
                "main",  
                new String[] {"String[]"},  
                new String[] {"args"});

        __CONTEXT_callReceptor0 =
            new CallReceptor(SomeClass.class,  
                "SomeClass",  
                new String[] {},  
                new String[] {});
    }

    SomeClass() {
        final CallState __CONTEXT_callState0 = new CallState(0);  
        final ReturnState __CONTEXT_returnState0 = new ReturnState();

        __CONTEXT_returnState0.setThis(this);
        History._callReceived(__CONTEXT_callReceptor0,  
            __CONTEXT_callState0);

        try {
            
        } catch(Throwable __CONTEXT_exception) {
            __CONTEXT_returnState0.setException(__CONTEXT_exception);
            if(__CONTEXT_exception instanceof RuntimeException)
                throw (RuntimeException).__CONTEXT_exception;
    }
}
```
throw (Error)__CONTEXT_exception;
}
finally {
    History._returnFromCall(__CONTEXT_callReceptor0,
                            __CONTEXT_returnState0);
}

public static void main(String[] args) {
    final CallState __CONTEXT_callStatel = new CallState(1);
    final ReturnState __CONTEXT_returnStatel = new ReturnState();

    __CONTEXT_callStatel.addParameter(args);
    History._callReceived(__CONTEXT_callReceptor1,
                           __CONTEXT_callStatel);

    try {
        OtherClass obj = new OtherClass();
        {
            int i = 0;
            for(; i < 3; i++)
                obj.method1();
        }
    }
    catch(Throwable __CONTEXT_exception) {
        __CONTEXT_returnStatel.setException(__CONTEXT_exception);
        if(__CONTEXT_exception instanceof RuntimeException)
            throw (RuntimeException)__CONTEXT_exception;
        throw (Error)__CONTEXT_exception;
    }
    finally {
        History._returnFromCall(__CONTEXT_callReceptor1,
                                __CONTEXT_returnStatel);
    }
}
E.2.2 Contents of icworkingdir/OtherClass.java

public class OtherClass {
    private static CallReceptor __CONTEXT_callReceptor0;
    private static CallReceptor __CONTEXT_callReceptor1;
    private static CallReceptor __CONTEXT_callReceptor2;
    private static CallReceptor __CONTEXT_callReceptor3;
    private static CallReceptor __CONTEXT_callReceptor4;
    private static CallReceptor __CONTEXT_callReceptor5;
    private static CallReceptor __CONTEXT_callReceptor6;

    static {
        __CONTEXT_callReceptor0 =
            new CallReceptor(OtherClass.class,
                             "__CONTEXT$map1",
                             new String[] { },
                             new String[] { });
        __CONTEXT_callReceptor1 =
            new CallReceptor(OtherClass.class,
                             "__CONTEXT$inner1",
                             new String[] { },
                             new String[] { });
        __CONTEXT_callReceptor2 =
            new CallReceptor(OtherClass.class,
                             "method2",
                             new String[] { },
                             new String[] { });
        __CONTEXT_callReceptor3 =
            new CallReceptor(OtherClass.class,
                             "__CONTEXT$inner0",
                             new String[] { },
                             new String[] { });
        __CONTEXT_callReceptor4 =
            new CallReceptor(OtherClass.class,
                             "__CONTEXT$map0",
                             new String[] { },
                             new String[] { });
        __CONTEXT_callReceptor5 =
            new CallReceptor(OtherClass.class,
                             "__CONTEXT$inner1",
                             new String[] { },
                             new String[] { });
        __CONTEXT_callReceptor6 =
            new CallReceptor(OtherClass.class,
                             "__CONTEXT$map1",
                             new String[] { },
                             new String[] { });
    }
}
new CallReceptor(OtherClass.class, "OtherClass", new String[] {}, new String[] {});

final CallState __CONTEXT_callState0 = new CallState(0);
final ReturnState __CONTEXT_returnState0 = new ReturnState();

__CONTEXT_returnState0.setThis(this);
History._callReceived(__CONTEXT_callReceptor0, __CONTEXT_callState0);

try{
    __CONTEXT_callStatel = new CallState(0);
    __CONTEXT_returnStatel = new ReturnState();
    __CONTEXT_callStatel.setThis(this);
    __CONTEXT_returnStatel.setThis(this);
    History._callReceived(__CONTEXT_callReceptor1, __CONTEXT_callState1);
    try {
        __CONTEXT$map0();
    }
}
catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState1.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
        throw (RuntimeException).__CONTEXT_exception;
    throw (Error).__CONTEXT_exception;
}
finally {
    History._returnFromCall(__CONTEXT.callReceptor1,
        __CONTEXT.returnState1);
}

public void method2() {
    final CallState __CONTEXT_callState2 = new CallState(0);
    final ReturnState __CONTEXT_returnState2 = new ReturnState();
    __CONTEXT_callState2.setThis(this);
    __CONTEXT_returnState2.setThis(this);
    History._callReceived(__CONTEXT.callReceptor2,
        __CONTEXT.callState2);
    try{
        __CONTEXT$map1();
    } catch(Throwable __CONTEXT_exception) {
        __CONTEXT_returnState2.setException(__CONTEXT_exception);
        if(__CONTEXT_exception instanceof RuntimeException)
            throw (RuntimeException).__CONTEXT_exception;
        throw (Error).__CONTEXT_exception;
    }
    finally {
        History._returnFromCall(__CONTEXT.callReceptor2,
            __CONTEXT.returnState2);
    }
}

private void __CONTEXT$inner0() {
    final CallState __CONTEXT_callState3 = new CallState(0);
    final ReturnState __CONTEXT_returnState3 = new ReturnState();
    __CONTEXT_callState3.setThis(this);
    __CONTEXT_returnState3.setThis(this);
try {
    _CONTEXT$inner1();
}

try {
    Call call = History.lastCall(null, "method2", null);
    if (call == null)
        _CONTEXT$inner0();
    else
        System.err.println("method2() called before");
    Call call2 =
        History.lastCall(String.class, "length", null);
    System.err.println(call2 != null);
} catch(Throwable __CONTEXT_exception) {
    _CONTEXT$mapO();
    __CONTEXT_returnState4.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
        throw (RuntimeException)__CONTEXT_exception;
    throw (Error)__CONTEXT_exception;
}
finally {

    try {
        Call call = History.lastCall(null, "method2", null);
        if (call == null)
            _CONTEXT$inner0();
        else
            System.err.println("method2() called before");
        Call call2 =
            History.lastCall(String.class, "length", null);
        System.err.println(call2 != null);
    } catch(Throwable __CONTEXT_exception) {
        _CONTEXT$mapO();
        __CONTEXT_returnState4.setException(__CONTEXT_exception);
        if(__CONTEXT_exception instanceof RuntimeException)
            throw (RuntimeException)__CONTEXT_exception;
    }
}
throw (Error)__CONTEXT_exception;
}
finally{
    History._returnFromCall(__CONTEXT_callReceptor4,
        __CONTEXT_returnState4);
}

private void __CONTEXT$inner1() {
    final CallState __CONTEXT_callState5 = new CallState(0);
    final ReturnState __CONTEXT_returnState5 = new ReturnState();

    __CONTEXT_callState5.setThis(this);
    __CONTEXT_returnState5.setThis(this);
    History._callReceived(__CONTEXT_callReceptor5,
        __CONTEXT_callState5);

    try {
        String s = "a string";
        System.err.println(s.length());
    }
    catch(Throwable __CONTEXT_exception) {
        __CONTEXT_returnState5.setException(__CONTEXT_exception);
        if(__CONTEXT_exception instanceof RuntimeException)
            throw (RuntimeException)__CONTEXT_exception;
        throw (Error)__CONTEXT_exception;
    }
    finally{
        History._returnFromCall(__CONTEXT_callReceptor5,
            __CONTEXT_returnState5);
    }
}

private void __CONTEXT$mapl() {
    final CallState __CONTEXT_callState6 = new CallState(0);
    final ReturnState __CONTEXT_returnState6 = new ReturnState();

    __CONTEXT_callState6.setThis(this);
    __CONTEXT_returnState6.setThis(this);
    History._callReceived(__CONTEXT_callReceptor6,
        __CONTEXT_callState6);
try {
    __CONTEXT$inner1();
}
catch( Throwable __CONTEXT_exception ) {
    __CONTEXT_returnState6.setException( __CONTEXT_exception );
    if( __CONTEXT_exception instanceof RuntimeException )
        throw ( RuntimeException ) __CONTEXT_exception ;
    throw ( Error ) __CONTEXT_exception ;
}
finally {
    History._returnFromCall( __CONTEXT_callReceptor6 ,
                            __CONTEXT_returnState6 );
}