SERIOUS Refactoring Handbook

Authors:
Aitor Aldazabal  European Software Institute
Matthias Rieger  University of Antwerp
Claudio Riva  Nokia Research Center
Bart Van Rompaey  University of Antwerp
Josetxo Vicedo  European Software Institute
Jan Zwanenburg  Philips Medical Systems

Editors:
Bart Van Rompaey and Matthias Rieger

July 30, 2008
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Introduction</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Target Audience</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Software Reengineering</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Book Format and related Work</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Reengineering Process</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Tools</td>
<td>7</td>
</tr>
<tr>
<td>1.6 Structure of the Book</td>
<td>7</td>
</tr>
<tr>
<td><strong>2</strong> Reducing Accidental Complexity</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Reducing Accidental Complexity</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Understanding Nested Control Structures</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Handling else-if chains</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Improving the Layout of Source Code</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Automatic Source Code Transformations</td>
<td>22</td>
</tr>
<tr>
<td><strong>3</strong> Refactoring State Machines</td>
<td>29</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>3.2 Reverse Engineering State Machines</td>
<td>31</td>
</tr>
<tr>
<td>3.3 Reengineering a State Machine</td>
<td>38</td>
</tr>
<tr>
<td><strong>4</strong> Refactoring for Performance</td>
<td>45</td>
</tr>
<tr>
<td><strong>5</strong> Testing Legacy Systems</td>
<td>51</td>
</tr>
<tr>
<td>5.1 Support refactoring with (Unit) Testing</td>
<td>53</td>
</tr>
<tr>
<td>5.1.1 Unit testing as a refactoring complement</td>
<td>53</td>
</tr>
<tr>
<td>5.1.2 Unit testing: objectives and guidelines</td>
<td>54</td>
</tr>
<tr>
<td>5.1.3 Use a testing framework</td>
<td>56</td>
</tr>
<tr>
<td>5.1.4 test Pragmatically</td>
<td>59</td>
</tr>
<tr>
<td>5.2 Test Effort Estimation</td>
<td>61</td>
</tr>
<tr>
<td>5.3 Introducing Test Suites</td>
<td>64</td>
</tr>
<tr>
<td>5.4 Test Suite Evolution and Reuse</td>
<td>74</td>
</tr>
<tr>
<td>Chapter/Section</td>
<td>Title</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>6</td>
<td>Architecture Level Restructuring</td>
</tr>
<tr>
<td>6.1</td>
<td>Architectural Analysis</td>
</tr>
<tr>
<td>6.2</td>
<td>Architectural Evolution</td>
</tr>
<tr>
<td>A</td>
<td>State Machines and their Representations</td>
</tr>
<tr>
<td>B</td>
<td>Script to extract Nesting Depth</td>
</tr>
<tr>
<td>C</td>
<td>Script to Extract Control Flow Skeleton</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Target Audience

This book is directed towards developers who are tasked with improving some part of a software system. The book aims to collect knowledge, methods, techniques, and tools that have been proven useful in typical reengineering tasks. Since every successful software engineer will be maintaining his systems over a longer period of time, implying continuous improvement of existing source code, the contents of this book will be interesting for anybody in the software development community.

1.2 Software Reengineering

It has been known for the last twenty years (Lehman’s Laws) that software must continue to be adapted to an ever changing requirement, or it will become progressively less useful. It is also known that software that evolves over time becomes more complex. The initial design of the software will no longer be able to accommodate the new requirements. All this makes the software harder to maintain, and if no additional effort is spent on reengineering the software, it might well be that it must be abandoned.

Software reengineering is any activity that:

(A) improves one’s understanding of software,
(B) prepares or improves the software itself, usually for increased
   (B.1) maintainability, evolvability
   (B.2) reusability

Reengineering has been named “controlled software evolution”, as it is used to try to hold against the inevitable deterioration of a system in the course of its life.
1.3 Book Format and related Work

The book presents reengineering knowledge in the form of patterns. The pattern as a format to transmit knowledge about best practices has been made popular in the field of software engineering with the seminal book Design Patterns: Elements of Reusable Object-Oriented Software [GHJV95]. A pattern is “a three-part rule which expresses a relation between a certain context, a problem and a solution.” (Christopher Alexander) The pattern does not only explain what solution may be used to resolve a problem, in what context the solution becomes relevant (and in which context the solution would be counter-productive), but it also tells us why that solution is needed.

In Object-Oriented Reengineering Patterns [DDN02] the pattern format has been used to describe knowledge about how to reengineer a system, from first contact to detailed refactoring scenarios for particular problems. The scope of the OORP book is broader than this handbook, but is certainly another good source of information.

The book Refactoring: Improving the Design of Existing Code [FBB+99] presents a number of simple refactorings. Each one is presented as a recipe, i.e. it describes the narrow, code centered context in which the refactoring is applicable. The patterns as used in this book aim at describing a wider context which also includes forces that may prevent the application of the pattern.

1.4 Reengineering Process

The activities that are performed during the reengineering of a system can be organized in a process that is illustrated with the so-called horseshoe model.

The activities are broken down into the following categories:

1. Model Capture: Reverse engineer from the source-code into a more abstract form, typically some form of a design model. How abstract depends on the kind of problem you want to solve.

2. Problem Detection: Identify design problems in that abstract model.

3. Problem Resolution: Propose an alternative design that will solve the identified problem.

4. Program Transformations: Make the necessary changes to the code, so that it adheres to the new design yet preserves all the required functionality. Here testing will play an important role.
Figure 1.1: The Reengineering Life-Cycle.

Note that the horseshoe model is a way to describe the various activities that take place during a project, but does not necessarily prescribe a strict order on when these activities must take place.

The ascending and the descending arm of the horseshoe can be described separately as Reverse and Forward Engineering.

**Reverse Engineering** During the reverse engineering phase we try to get an understanding of the system and its problems. The system is not altered, but we gather additional knowledge about the system. Two activities can be identified:

**Information Extraction:** Collecting information from different sources, organizing the knowledge.

**Information Abstraction:** Aggregation of information about subsystems and their relations, reconstructing the architecture of the system.

The reverse engineering phase ends with a model of the current architecture of the system. On this model we are able to search for design or architectural problems.

**Forward Engineering** Forward Engineering is the traditional software process of moving from high-level abstractions and logical implementation-independent designs to the physical implementation of a system.
With the knowledge about the problems that this architecture entails, we can devise a new target architecture that resolves the problems. We then get to map the new architecture to the implementation.

1.5 Tools

A reengineering project cannot be accomplished without tools. The tools we present in this book are mostly open source tools, some of them academic prototypes, some of them industrial strength. An important pillar of our tool set is the UNIX command line and collection of tools, consisting of, for example, `find`, `grep`, `sort` and `PERL`. For many of the small tasks that a reengineer is confronted with there is no out-of-the-box tool support, but an ad hoc tool can be assembled from these building blocks. Fluency in the language of the UNIX command line is a definite plus for every reengineer.

1.6 Structure of the Book

This book is structured around refactoring scenarios that are deployed and elaborated upon in individual book chapters. One or more patterns describe the steps to a generalized solution for the stated scenario goal. The scenarios that are discussed in the book are:

Reducing Accidental Complexity. This chapter is about source code components that have been growing in complexity over time up to a point where such a component can no longer be grasped as a whole by a human.

Refactoring State Machines. State machines are a well known mechanism to model a system. The patterns in this chapter help the reengineer to detect, reverse engineer and refactor a degraded state machine implementation to an explicit and extensible form.

Refactoring for Performance. The decomposition of a software system impacts quality attributes such as memory consumption and start-up time. This chapter enlists a pattern about refactoring a system’s sub-optimal data layout (termed Redundant Objectification).

Testing a Legacy System. Introducing automated tests for an existing system may be challenging due to missing test interfaces, high coupling, centralized complexity etc. The patterns in this chapter offer solutions to write efficient I/O tests, to incrementally refine integration tests to finer grained tests, etc. Furthermore, patterns that help in estimating
refactoring test effort and in test case reuse quantify and reduce the overhead of automated tests.

Architecture Level Restructurings. This chapter provides patterns to recover the structure of an application and study migration paths. Moreover, this chapter enlists patterns to migrate to new technology or to extract a component from a system.
Chapter 2

Reducing Accidental Complexity

This chapter is about source code components that have been growing in complexity over time up to a point where it can no longer be grasped as a whole by a human. Reducing Accidental Complexity is a meta-pattern that describes this problem context and delegates sub-problems and their solution to separate patterns.

2.1 Reducing Accidental Complexity

Problem

Accidental complexity is created when the structure of the source code does not respect the limited human capability to hold more than a few concepts in the mind at any given moment. This differs from the essential complexity of the problem the source code solves. Accidental complexity can be reduced by following some rules and guidelines when writing code. Essential complexity cannot be reduced.

One of the most apparent places where complexity manifests itself is in the nesting depth of function implementation. Without careful attention to this aspect, function or method implementations tend to get more complex over time as they fulfill more and more requirements. More general, readable code exhibits a number of characteristics:

- All code in the function contributes to the task that is described by the name of the function. The logic of the function is clear.
- The code maintains the same level of abstraction throughout the function. This means that you tend to implement the construction of a string message in a different function than the protocol handling the sending of the message.
• Variables are defined close to where they are used.
• Names of variable and functions express their intent clearly.

Some aspects characterizing a simple function can be specified with a threshold:
• Nesting depth is less than 4. Or better: The reader does not have to scroll vertically to read the code in the editor window.
• Ideally, a function can be seen in its entirety on a single screen. Reading is easier than recalling.

These thresholds should be considered rules of thumb, rather than the absolute truth. How many lines of code make a function too long depends on many different factors, among them psychological ones which differ from one programmer to the next. As mentioned above, difficult to understand code may also be caused by essentially difficult problems. It is not possible to make every piece of code digestible without effort on the part of the reader.

There are a number of indications that a function is not too long, even if it exceeds some of the thresholds mentioned above:

• The function handles a single concept. The logic of the function is still clear, despite its length.
• The function consists of a repeated fragment that cannot be extracted and unified, for example the initialization of a table of constants. It’s “cognitive length” is therefore only the length of one single fragment.
• The function does not have to be extended and maintained. An example is the implementation of a mathematical operations.
• The functions needs a lot of state, which would have to passed around to extracted functions. Of course, in an OO context, the “Replace Method with Method Object” refactoring can help with this problem.

The question is: How can we recognize accidental complexity which makes an implementation confusing, distinguish it from essential complexity, and how can we transform the confusing code into something that is maintenance-friendly?

Solution
We present a three step approach to reducing accidental complexity. These questions are to be considered in sequence to finally devise a conclusion and a way to actually reduce the complexity.

1 http://www.refactoring.com/catalog/replaceMethodWithMethodObject.html
Determine if a code fragment is **too complex**? How programmers perceive complexity to a large extent depends on the individual. We therefore cannot give absolute thresholds which determine that a piece of code is too complex.

A number of prerequisites can guide the judgment of the maintainer:

- You have to change/maintain/extend a piece of code.
- You are confused by the code.
- Writing a test for the function proves difficult as the function does too much, requiring a complicated test setup, and an elaborate control of the test execution results.

Determine if we can reduce complexity? We need to recognize essential complexity that will not yield to code structure improvements. Apart from the fact that the code tackles a complex problem domain, there are the following hints that restructuring attempts will not make the code easier to understand:

- The code adheres to the principles of conceptual unity (only a single task is performed) and common abstraction level (understanding a part does not distract from understanding the whole).
- The code remains difficult to understand even if the control flow is well understood.

Determine how we can reduce complexity? There are a number of different types of long and complex code fragments that can be handled each in their own manner. For each of these types a canonical idiom exists towards we can aim:

- Nested Control Structures. See Section 2.2
- Switch Statements implementing State Machines. See Chapter 3
- *if — else if — else* chains. See Section 2.3
- Needlessly complex implementations for which a simpler alternative exists. See Section 2.4
2.2 Understanding Nested Control Structures

Problem

Control structures make source code more easy to understand by forcing the programmer to express his intent using a small number of well-known constructs. However, nested control structures with a depth of more than 3 can have the inverse effect and require a high cognitive effort on the part of the maintainer. Unfortunately, even originally simple code tends to become more complex as more requirements are added over time. If no conscious decision is taken to reduce nesting depth in a function, the effort to understand the implementation will so become unreasonably high.

How can you work with code that exhibits complicated control flow?

Solution

Make the control structures explicit, emphasizing them over the rest of the code. You are then able to concentrate on this aspect and increase your understanding of the code undisturbed by less relevant details. Below we present some solutions to achieve this goal, distinguishing between solutions which make control flow information available in the normal editing environment (informally called high-tech), and solutions which present the information outside of the editors, to be studied separately (accordingly called low-tech for the purpose of this section).

High-Tech Solution: Advanced Editors

Advanced editors provide a number of features that help in dealing with nested control statements.

Control Structure Diagrams

A control structure diagram (CSD) clearly depicts control constructs, control paths, and the overall structure of each program unit, in a visual representation which is integrated with the source code. Figure 2.1 shows how classes, methods, loops, if- and return-statements are represented in a CSD.

Folding

Folding means to interactively hide from view the contents of the blocks from the source text. For example, a while-statement hides the body of the loop and only shows the condition. Using folding, one can control very specifically how much of the code one wants to see. Figure 2.2 shows...
Figure 2.1: A CSD diagram for a JAVA class.

the completely folded version of a 300 line switch-statement. This view transparently distinguishes all the cases of the switch. The folding must be interactively controllable: clicking on a square with a + sign will unfold the contained block.

Figure 2.2: A folded switch statement. Unfolded, the original block of this example comprised 300 lines.

Low-Tech Solution: Separate Views

If the control flow cannot be made visible within the editor, we can prepare special source code views and make them available to the maintainer in
parallel to the editing environment.

**Extract the Control Flow Skeleton**

When the editor does not offer the folding feature, the next best solution is to create a view of the source code which emphasizes the control flow over the non-control flow statements. In Listing 2.1 a function body is shown reduced to only control flow related keywords and expressions. This presentation of the source allows the maintainer to get a condensed overview of a code fragment which would, for example, normally span multiple pages. Such an overview can be used as like an index into the function, providing context without necessitating scrolling on behalf of the user.

Listing 2.1: The control flow skeleton of a C function.

```c
for (i = 0; str[i]; i++)
    switch (c){
    case 'a':
        break
    case 255:
        while (c && c != 32){
            if (c == 0){
                return 0
            } else{
                return 0
            }
        } break
    default:
    }
```

**Marking Code On Paper**

The low tech solution to understanding complex functions uses paper and color. We make a printout of only the function under study. We make the printout on one-sided paper because double-sided will force us to turn the pages and we will not be able to hold the entire function in view at the same time.

We then use differently colored text markers for the following tasks:

- Delimiting structures like cases in a `switch` statement or branches of an `if`-statement.
- Delimiting areas in the code which perform the same or similar functions (duplicated code).
• Read access to specific variables. Specifically: conditional tests which use specific variables.

• Write access to specific variables.

• `return` and `break` statements.

An example can be seen in Figure 2.3.

Figure 2.3: Source code which is highlighted with textmarkers.

Tools

• **JGRASP**[^2] is an editor which offers CSD and folding capabilities. JGRASP is an open source IDE running on platforms which have a JAVA runtime. It handles programs written in JAVA, and C++, among other languages.

• The script `controlFlowSkeleton.pl` in Appendix C extracts the statements and expressions pertaining to the control flow of a given block.

[^2]: [http://www.jgrasp.org](http://www.jgrasp.org)
2.3 Handling else-if chains

Intent: This pattern seeks to make complex code that is the result of else-if chains simpler.

Problem:

An else-if chain is an if-statement with more than two branches. For example this statement contains two additional if tests:

```c
if(isSCPchannel(channelId)) { ... }
else if(isSPchannel(channelId)) { ... }
else if(isCPchannel(channelId)) { ... }
else { ... }
```

An else-if chain allows for the same type of control flow dispatch as a switch statement: when mutually exclusive branches are to be executed.

Just as every other type of code, else-if chains tend to grow longer and can make the code hard to read: if you do not have the entire statement in view, it’s difficult to keep the overview. While trying to understand the intricacies of the complex actions within the branch, we lose track of why we arrived here in the first place, and what the other possibilities are.

Breaking an else-if chain apart like a long sequential function is however impossible: due to the mutual exclusiveness of its branches, an else-if chain is not easily broken apart.

How can we restructure a long else-if chain?

Solution

The means to keeping the else-if chain understandable is to unify the level of abstraction of the code. What we as the reader of the software want to know from the else-if statement are the number of alternatives, and an indication of what has to be done in each case. The knowledge about what is to be done is at a finer level of detail and should be abstracted away.

The solutions for the refactoring of an else-if chain depend mostly on the type of the if-conditions:

- The else-if chain is the normal solution for mutually exclusive branches which cannot be decided on by integer comparison. A canonical example for the use of the else-if chain is this one where commands in string form are interpreted:

```c
if (strcmp(cmd,"activate")==0) {
    activateCmd(str, idx);
}
else if (strcmp(cmd,"deactivate")==0) {
```
This example also illustrates well that if the code in the branches is kept to a minimum, the level of abstraction is uniform and the overview over the entire construct is not lost. This kind of else–if chains are the goal of any refactoring. Extracting the code in the branches into separate functions achieves this goal.

- Conditionals which can be decided using switch statements should be done so. This is the case for all comparison on the basis of integer values.

For example, this code using an else–if chain

```c
if (systemMode == E_residentialBridge) {
  lanMode = ResidentialBridge;
} else if (systemMode == E_crossConnect) {
  lanMode = CrossConnect;
} else {
  lanMode = QosAware;
}
```

is less clear than this code using a switch:

```c
switch (systemMode) {
  case E_residentialBridge: lanMode = ResidentialBridge;
  break;
  case E_crossConnect: lanMode = CrossConnect;
  break;
  default: lanMode = QosAware;
}
```

because the switch presents the situation more clear to the programmer:

- the decision depends only on the value of the switch expression, in the example on the variable systemMode.
- less duplication as the switch does not repeat the comparisons to systemMode.

- If the else–if chain is not followed by any more code, we can employ the “Return Not Else” idiom[Ber99]: instead of creating an additional branch, we return from the function.
For example, this else–if chain

```c
if (port->search(channel, vpn)) {
    refresh (server1, lan));
} else if (isLocal(lan)) {
    server1->attach(lan);
} else {
    server2->attach(lan);
}
return 0;
```

can be restructuring into this form:

```c
if (port->search(channel, vpn)) {
    refresh (server, member->lan));
    return 0;
}
if (isLocal(lan)) {
    server1->attach(lan);
    return 0;
}
server2->attach(lan);
return 0;
```

The second form is easier because the reader can dispense with things as he goes along, rather than having to keep them in mind until the end.

Note that some coding rules enforce functions with single-point of return, in which case this solution cannot be applied.

- If the conditions of the else–if chain are used to determine the type or kind of an entity, and especially if they are repeated multiple times in the program, the reengineering pattern “Replace conditional by polymorphism” [DDN02] should be considered.

If the conditionals are executing an action with side effects, e.g.

```c
if (Group::contains(item))
    // do something
} else if (Group::attach(item))
    // do something else
} else
```
the element of ordered sequence comes into play as well. However, side effects in conditions are a non-standard idiom.

Trade-offs

Pro’s:

• An else–if chain is usually an important enough element that it warrants to be understood on its own. This pattern helps to emphasize the selection represented by the else–if chain.

• A compact representation of the selection will be more easily recognizable as being similar to another selection. Refactoring opportunities will therefore be more apparent.

Con’s:

• For long else–if chains, even small fragments of action code can lead to overly long functions, thereby triggering the moving of the fragments to small functions. This can also mean a loss of overview, if no good names can be found for these functions.

Related Patterns

• Patterns for Selection (http://csis.pace.edu/~bergin/patterns/Patternsv4.html)

• Two Patterns for Polymorphism (http://csis.pace.edu/~bergin/patterns/polymorphism.html)

• Patterns about putting complex conditionals into a predicate function.

• State machines

2.4 Improving the Layout of Source Code

Source code should be written so that it can be read by humans. The compiler can make sense of anything that is syntactically correct. The real bottleneck is the human reader who must make sense of the code, once a need for adaptive or corrective maintenance arises.

There exist a number of characteristics of the human mind which constrain the abilities of the human code reader. Writing the code with the
reader in mind can reduce the cognitive load and therefore lead to more maintainable code.

This section lists a number of simple rules for improving code layout and reducing the cognitive load:

**Line up multiple uses of an Identifier**

If you use an identifier multiple times within a condition, line them up so we can quickly see that the same identifier is checked multiple times.

For example, this code checks if the variable `message_i.dstIPAdr` has one of two possible values:

```c
(message_i.dstIPAdr == cdvIGMQAddress_c || message_i.cdvfPkt.fGroupAddr == message_i.dstIPAdr)
```

This fragment can be made more readable if we switch the order in the second branch of the `or`:

```c
(message_i.dstIPAdr == cdvIGMQAddress_c || message_i.dstIPAdr == message_i.cdvfPkt.fGroupAddr)
```

We're exploiting the ability of the human eye to recognize patterns in nearby entities. We can better compare the two names if they are one above the other, than if they are far apart, especially if they are quite cryptic and not easily pronounceable.

**Combine all conditionals which do not start a new branch**

You are writing a couple of nested `if` statements.

```c
if (boardType.compare("XXX1") )
{
  if (appliqueType.compare("ABC") )
  {
    if (appliqueVariant.compare("AA") )
    {
      configureForFastTrack();
    }
    else
    {
      setUp();
    }
  }
}
```

⇓

20
if (boardType.compare("XXX1") && appliqueType.compare("ABC") )
{
  if (appliqueVariant.compare("AA") )
  {
    configureForFastTrack();
  }
  else
  {
    setUp();
  }
}

By reducing the number of nested blocks, we do not only save indentation space and prevent having to vertically scroll, we are also telling the reader upfront that the two conditions have the exact same scope. This means that the reader does not have to move his eyes down to check how the two block ends relate to each other.

**Short Case First**

You are writing an if-else-statement. One of the actions can be expressed simply in a statement or two. One is much longer. For some reason it is not desirable to apply Function for Complex Action.

You want your reader to be able to read and understand the code as simply as possible. You also want the reader to be able to easily determine if this is an if with an else or without an else.

Therefore, arrange the code so that the short case is written as the if (not the else) part.

```
if (someCondition())
  // aStatement
else
  ... // lots of statements
  ...
```

This will permit the reader to easily dispense with one case before forgetting the condition that is used to choose between cases.

---

3This pattern has been copied verbatim from http://csis.pace.edu/~bergin/patterns/PatternsV4.html#scf
Reverse conditionals if they are easier to understand

You are writing an if-else-statement. Both branches contain a similar amount of code. Formulate the condition in a positive way and arrange the if and the else branch accordingly.

The original code and the transformed code look like this:

```
if (!isSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge(quantity);
```

⇓

```
if (isSummer(date))
    charge = summerCharge(quantity);
else
    charge = winterCharge(quantity);
```

Removing the not from the condition requires one less step in the mind of the reader.

Related Patterns

Some of the examples presented in this section have been taken from the following pattern languages for basic coding:

- Coding at the Lowest Level. Coding Patterns for Java Beginners
- Patterns for Selection

2.5 Automatic Source Code Transformations

Intent: To effectuate a change on a number of locations in the source code, write a small transformation script based on regular expressions.

\[\text{Example comes from } \text{http://www.refactoring.com/catalog/reverseConditional.html}\]
\[http://cns2.uni.edu/~wallingf/patterns/elementary/papers/coding-at-the-lowest-level.pdf\]
\[http://csis.pace.edu/~bergin/patterns/Patternsv4.html\]
Problem

Certain types of refactoring require a textual transformation of many instances of a specific code idiom. The change is something that lies between the capabilities of the Find/Replace editor command, and the refactorings offered by a sophisticated refactoring engine. Since the change has to be applied to many instances, manual execution will be very tedious and time consuming. This problem is difficult because:

- Refactoring tools are either not available or are not able to perform the desired change.
- It is tedious, error-prone, and time-consuming to try to perform the change by hand.
- The to-be-replaced idiom contains slight variations, making an exact string match impossible.
- The to-be-replaced idiom may occur in many different contexts, not all of them will need to be changed. The decision to replace or not might involve a lot of contextual knowledge.

Yet, solving this problem is feasible because

- Regular expressions allow us to find very specific textual occurrences.
- Scripting languages enable full control over when to replace a match.
- The string manipulation that is possible in a full programming language allow us to perform arbitrarily complex transformations.
- Scripting languages like PERL excel in handling line-oriented data. Transformations that are contained within a single line are therefore handled with minimal effort.

Solution

Write a small script in a language with good text management facilities. The script reads the source code of a file, matches the occurrences of the idiom to be transformed, and applies the change to each match.

Steps:

1. Determine if the entities that you want to change can be found by a regular expressions. The following characteristics improve these chances:
• The entity can be described using a regular expression pattern, i.e. it does not depend on nested entities like expressions can be. The easiest is always a fixed string (like a keyword) that only exists at the locations that we want to treat.

2. Choose a programming language to write your transformation scripts in. Programming languages with good regular expression support are: PERL, RUBY, and PYTHON.

3. Write a script which reads all the lines of one file and applies the changes.

Example 1: The first example illustrates a replacement which is sufficiently complicated so that it cannot be done with the find/replace command of an editor.

The system being refactored in this example is written in RUBY. The program prints debug information directly to STDOUT, which results in lines like this found everywhere:

```ruby
if (debug) printf("actual side = BOTH");
if (fullDebug) printf("checkCurrentRelease () ");
```

Since we wanted to be able to redirect debugging output into a file, we implemented the methods debug(msg) and fullDebug(msg). Every instance of a line printing debug output should now be changed into something like the following code:

```ruby
depbug("actual side = BOTH")
fullDebug("checkCurrentRelease () ")
```

To perform this transformation we can use the following PERL script:

```perl
while(<>) {
  s/\s+if\(\(debug\)\)\s+printf\(\(\+7\)\);/debug($1);
  s/\s+if\(\(fullDebug\)\)\s+printf\(\(\+7\)\);/fullDebug($1);
  print $;
}
```

The script reads all lines from a file that is taken from the command line and writes the changed (or unchanged) lines back to the standard output.

24
Example 2: The second example is more complicated and involves a validation step. A program contains the following conditional idiom in many places:

\[
(\text{cl.side}==\text{SIDE\_BOTH} \ || \ \text{cl.side}==\text{actualSide})
\]

In order to make the code more compact and readable, we want to implement this idiom as a method `onActualSide`.

```cpp
int Class::onActualSide() {
    return (side==SIDE\_BOTH \ || \ side==actualSide);
}
```

and replace all occurrences of the idiom above with the call to the new method, e.g. `cl.onActualSide()`.

The following PERL-script takes care of the replacements:

```perl
$sidecheck = qr{(\w+)=(?:SIDE\_BOTH|actual\_Side)};
while(<>) {
    s/\s+/ $sidecheck \s+ \|| \s+ $sidecheck 
    /v a l i d a t e ($&,$1,$2) 
    /ex;
    print $;
}
# makes sure that the two comparisons in
# \# cl.side == SIDE\_BOTH \ || \ cl.side == actual\_Side
# # are performed on the same variable, e.g. 'cl'.
# # If this is true, the replacement string is created.
# # If false, then the original match is returned.
sub validate {
    my ($match, $varA, $varB) = @_; 
    return "$varA.onActualSide()" if ($varA eq $varB); 
    return $match
}
```

4. Validation checks can also involve user input. The user is presented with the original and the changed line and can individually accept or reject the change:

```perl
while(<>) {
    my $changed = $;
    if($changed =~ s/foo\((\+?)%\)/bar($1)/) {
        $&==$changed if(validate($&,$changed));
    }
    print $;
}
# ok the change with the user
sub validate {
    my($original,$changed) = map{s/\s+/\s+$/;$/}@_; 
    print STDERR "OK? $original $changed [Y/n]";
    my $ok = <STDIN>;
    # default is 'Yes'
    if($ok ne "\n") { $ok='y' }
    return ($ok eq 'y') ? $changed : $original;
}
```

A sample execution of this replacement script looks like this:
Ok?: foo(&cdrRec, 0, sizeof(CdrRecord));
bar(&cdrRec, 0, sizeof(CdrRecord));

[Y/n]: y
Ok?: foo(i,container);
bar(i,container);

[Y/n]: n

Example 3: The third example transforms code which extends over multiple lines.

The program contains the following idiom in many places:

```plaintext
trc_printf(traceId,"discard packet,%d",msg.channelID);
dbg_printf("discard packet,%d", msg.channelID);
```

To get rid of the duplication a single function `trc_and_dbg()` is created which takes care of both the tracing and the debugging. Now, the client code has to be changed to invoke this uniform function, i.e. for the above example the resulting code looks like this:

```
trc_and_dbg(traceId,"discard packet,%d",msg.channelID);
```

Matching code that extends over multiple lines can be done with the `s` modifier in PERL regular expressions. It requires however that the code is read in its entirety into memory before the regular expression is applied to it. The following transformation script takes care of this:

```perl
foreach $f (@ARGV) {
    open IN, $f or die "Could not open file '$f', abort";
    # undef the record separator to get entire file contents at once
    my $source = do { local $/; <IN> };
    close IN;

    $source =~ s/trc_printf\(".*\",(\.+\?)\); \n> dbg_printf\(".*\",(\.+\?)\);\n> trc_and_dbg\($1\);\n> /xsg;
    print $source
}
```

5. Generate a list of all the files that you want to apply your script to. This can be done with the UNIX utility `find`. For example, to perform an action on all *.hpp and *.cpp files below the root directory of a project, this shell script skeleton is useful:

```bash
for file in `find projectroot/ -name '.*[hc]pp'`
do
    # invoke the transformer script here
done
```
If you want to apply the changes only to a subset of the source files, select a more specific directory than `projectroot`.

6. Since you cannot be sure that your transformation works correctly from the get-go, you have to keep the original code around until you have checked the results. If you want to validate the results of the transformation manually, write the transformed code into a separate file like this:

```bash
target="$file.transformed"
transformer.pl $file > $target
# uncomment next line once transformer works flawlessly
#mv $target $file
```

Without the line `mv $target $file` this script is idempotent, i.e. you can rerun it again and again without changing the original code.

If you want to validate the output by compiling, keep the original code in a backup file and write the transformed code into the original file where it is found by the build system:

```bash
$backup="$file.original"
cp $file $backup
transformer.pl $backup > $file
```

Should the transformation be incorrect, restore the original code into its rightful place and redo with an improved `transformer.pl`.

7. Test the transformation for syntactic correctness by compiling the code, and for semantic correctness by running the tests.

**Trade-Offs**

**Pros:**

- The transformations are only constrained by the combined capabilities of regular expressions and a full fledged programming language. This gives the user an enormous control.

- The transformation is repeatable and fast. It is not prone to accidental mistypes common for repetitive tasks performed by humans. With proper handling of backups, we’re able to tune the transformation over multiple iterations until it does exactly what is needed.

**Cons:**

- Writing and testing a transformation script takes some time in itself. Doing the transformation by hand might be faster if there are only
a few occurrences. The effectiveness of the approach depends on the familiarity with regular expressions and the chosen programming language.

- The approach is only applicable to changes which do not require analysis on the level of a parser.

Difficulties:

- The greatest difficulty is to find a regular expressions which precisely matches the entities you want to change and nothing else. The regular expression might match an occurrence not targeted by the refactoring, e.g. in a string literal or a comment. Name clashes may occur if you want to match variables found in multiple places, or classes with the same name in different namespaces.

- It might not be easy to formulate in a programmatic fashion the conditions which determine if the transformation should take place or not.

- It is difficult to estimate the time it takes to write and test the transformation script versus the time it takes to perform the transformation manually.

An approach that works is to build up a library of transformation scripts that provides a template for many situations. The effort to adapt a given template to a concrete query is then negligible.
Chapter 3

Refactoring State Machines

State machines are a common way of representing a variety of problems in computer science. For certain implementations of state machines, e.g. switch-statements, unattended growth can lead to overly complex and un-maintainable code. This paper presents a small, code-level pattern language which assists a maintainer in the different tasks of identifying, recovering, and refactoring of state machine implementations with the goal of applying some of the well known idioms and patterns that have been developed for the procedural and the object-oriented paradigm.

3.1 Introduction

Code that grows over the course of multiple evolutionary steps tends to get more complex than what can be handled by someone not familiar with the code. When the cost of adding another feature and the risk of introducing errors in the course become too large to be carried, a dedicated effort to refactor first becomes necessary.

In such a legacy system one is typically confronted with code fragments that are too complex to directly integrate some required changes. To reduce the complexity via refactoring, you first have to understand the nature of the code. When identified as a state machine implementation, i.e. a set of states with transitions triggered by external input, specific solutions exist to make such fragment maintainable again.

Figure 3.1 contains an overview of a small pattern language for refactoring state machines that we present in this paper. Having some clues that the given code fragment encompasses some state machine implementation triggers the entrance point to the language, helping the maintainer to (i) recognize the state machines; (ii) extract its essence in terms of states

---

1this chapter is extracted from the paper "Refactoring State Machines" by Matthias Rieger, Bart Van Rompaey and Serge Demeyer. in Proceedings of the 6th Nordic conference on Pattern Language of Programs (VikingPloP), pp 39–57, September 2007.
and transitions; and (iii) subsequently transform them into a modular, well structured implementation under a variety of implementation choices.

Figure 3.1: Pattern Language for Refactoring State Machines (SM).

The first two patterns of the pattern language constitute the reverse engineering part and are described in Section 3.2.

A state machine is a well known representation of a specific class of problems. There exist multiple idioms and patterns which describe how to implement a state machine. Knowing that a given code fragment implements a state machine therefore gives us a clear goal towards which we can guide our refactoring effort. Therefore, in the first pattern, named DETECTING AN IMPLICIT STATE MACHINE, we need to gather evidence that helps to decide whether the given code fragment implements a state machine. A positive answer gives rise to applying the follow-up patterns.

The next pattern, RECOVER THE ESSENTIAL STATE MACHINE seeks to guide the developer in identifying the essential composition of the state machine in terms of states, transitions and action code. Having recovered this state machine model out of the source code, two more questions pop up. First, the developer needs to decide upon a new go/no go decision about an eventual refactoring. Secondly, the most favorable implementation for the case needs to be chosen.

Once we have extracted the structure of a state machine in terms of states and transitions, we can go over to the next step of rebuilding it using one of the well known idioms and patterns for state machine implementations. The intent is in all cases the same: re-implement an existing state machine so that its states and transitions become explicit. The preferred implementation technique depends largely upon the contextual programming paradigm, the desired level of abstractions and the representation and amount of states. In Section 3.3 we describe reengineering patterns for refactoring towards two typical state machine implementations. The pattern REFACTOR TO A PROCEDURAL STATE MACHINE proposes an implementation using a master function that dispatches control to state-representing
functions. Refactor to an Object-Oriented State Machine describes how to refactor a deteriorated state machine implementation towards the object–oriented state pattern by Gamma et al [GHJV95].

The pattern language in this paper is written for software developers working on evolving legacy systems, having to decide between quick hacks and from scratch implementations when coping with new requirements. The patterns here help to explore and carry out a path in the middle: refactoring the existing implementation first, facilitating future additions. In that sense, this work is similar to the reengineering patterns of [DDN02] and the refactoring guidelines of [Ker05]. Furthermore, a small appendix contains some basic information about state machine representations (Appendix A).

### 3.2 Reverse Engineering State Machines

Suppose a complex piece of code, containing lots of branches with complex conditions that you are required to understand and which you suspect of implementing a state machine. In order to perform a successful refactoring towards a clean implementation, we first need to decide if the code fragment in question really concerns a state machine. Secondly, we have to reverse engineer the state machine and extract all states, transitions, and associate the action code within the state machine correctly with the given states and transitions. The following forces play a role in determining the difficulty, required effort, scope and support required to proceed with reverse engineering the code fragment:

**Reverse Engineering Goal.** Will the concerned code be extended with new requirements? Are you tracking a bug, possibly involving test writing? Maybe you plan a rewrite or port of the code. In all cases you need to understand what the code is doing, although the level and scope of the desired understanding differs.

**Familiarity with the code.** Which artifacts are available except for the source code? The documentation may mention the presence of a state machine or even presents a (possibly outdated) specification. Are the original developers still around, or, if not, how experienced are the current maintainers with the code? Moreover, what is your own knowledge of state machines?

**Impact and involvement.** What is the role of this code in the system? What is the current change rate? Checking the involvement of other developers (possibly via the versioning system) as well as how the system relies on this piece through its interface gives a feeling for how difficult and important it is to reverse engineer.

**Quality of the code.** Several lightweight metrics give a first impression of the code’s quality. The readability can be verified via size metrics and identifier names. The number of bugs in recent months tell something about
the reliability. Determine changeability via versioning system queries as well as by asking the developers (how many of your colleagues avoid this piece of code at all cost?). The outcome helps to estimate how much effort is required.

Pattern: Detecting an Implicit State Machine

*Intent:* Determine whether a complex code fragment implements a State Machine using a code review checklist.

*Problem*

How can you identify a State Machine implementation in source code? This problem is difficult because:

- There exist many guidelines and patterns about how to implement a State Machine [R.B00a, p. 202]. Some implementations are strongly localized in a lengthy construct, others rely more on abstractions and dispatching. Clearly, each of such implementations bears its own characteristics, adding to an eventual checklist. Moreover, a State Machine can just as well be implemented in an arbitrary combination of conditions and abstractions.

Yet, solving this problem is feasible because:

- Developers typically use naming conventions to denote State Machine parts such as states or transitions. As a reviewer, you can make use of such indicators.

- Given system documentation, you may have encountered state diagrams or requirements that are likely to translate into a State Machine.

In order to perform a successful refactoring to a clean state machine, we first need to decide if the code fragment in question really implements a state machine. Secondly, we have to reverse engineer the state machine and extract all states and transitions, and associate the action code within the state machine correctly with the given states and transitions.
Solution

Detecting the Structure of a State Machine

A state machine is a closed program section that repeats itself, getting input from outside, and selecting on possible alternatives among a set, until it has reached an end-state, upon which the state machine is exited.

The basic structure of a state machine is therefore (in pseudo code):

```plaintext
state = initialState;
while (moreInput() && state != endState) {
    <select action code based upon state and input>
    <execute action code with input>
    <transition to new state>
}
```

Any code fragment that conforms to this schema can be viewed as a state machine.

The selection of the action code happens via a switch–statement or an else-if chain. In many case the state is represented by a numerical type. Therefore, a switch condition is able to dispatch control flow to the appropriate state code. The following code exhibits a canonical example for such an implementation:

```plaintext
switch ($state) {
    case 1:
        $return .= utf7[ord($char) >> 2];
        $residue = (ord($char) & 0x03) << 4;
        $state = 2;
        break;
    case 2:
        $return .= utf7[$residue | (ord($char) >> 4)];
        $residue = (ord($char) & 0x0F) << 2;
        $state = 3;
        break;
    case 3:
        $return .= utf7[$residue | (ord($char) >> 6)];
        $return .= utf7[ord($char) & 0x3F];
        $state = 1;
        break;
}
```
When the state is encoded in a string, however, an state machine implementation must use an else-if chain, as can be seen in the following example:

```c
if (pos == root_struct) {
    status = "body";
    root_struct.ns = msg["namespace"];
} elseif (name == "Body") {
    status = "envelope";
} elseif (name == "Header") {
    status = "envelope";
} elseif (name == "Envelope") {
    // nothing to do
}
```

It is also possible that states are encoded in the values, or ranges of values, of multiple variables. The following example shows how three states are defined as ranges of the variable `count`:

```c
if (count == 0)
    ... // action for empty collection
else if (0 < count && count < maxEntries)
    ... // action for normal usage
else if (maxEntries <= count)
    ... // action for exhausted capacity
```

Finally, the purpose of the code can be an indication that it is implementing a state machine. State machines are preferred solutions for a number of common tasks such as, for example, (stateful) protocols, lexical analyzers, (GUI) event handlers, etc. Oftentimes programmers note that the present code implements a state machine by means of accompanying comments.

Pattern: Recover the Essential State Machine

*Intent:* Recover the composition of a State Machine from the implementation.

*Problem*

How can you extract states, transitions and actions from a State Machine implementation?

This problem is difficult because:

- A State Machine may not have been implemented consistent with a particular implementation pattern. States or transitions may have been omitted.
Yet, solving this problem is feasible because:

- You can exploit the check list heuristics that made you decide that we are dealing with a State Machine implementation.

In order to perform a successful refactoring to a clean state machine, we need first to decide if the code fragment in question really implements a state machine. Secondly, we have to reverse engineer the state machine and extract all states, transitions, and associate the action code within the state machine correctly with the given states and transitions.

**Solution**

Once we know that the code fragment in question implements a state machine, we can extract its individual constituents, e.g. its *states, transitions*, and the actions which are tied to them. With the information about states, transitions and actions you can draw a representation of the state machine, e.g. a state diagram (see Appendix [A](#)). This serves as the basis for the reengineering of the state machine.

To be able to identify the states and transitions of the state machine better, consider a first round of refactorings[2] We have, for example, observed that separate cases in a complex switch–statement often contain duplicated pieces of code. Be aware, however, that moving code around before you have identified the states and transitions may make reverse engineering more difficult if the code contains statements pertaining to these concepts.

**Extracting the State Variables**

A state machine has a finite number of states, which are represented by either one, or a group of variables. At the same time we must identify the variable, called the *state variable* (*sv*), which holds the input values that trigger the transitions. Three hints can lead you here:

**Name:** Names ending in *status*, or *state*, are good indicators for the *sv*. Be aware, however, if the input variable comes from another state machine, for example from the settings of a radio button GUI element, it might still be called *state*. The name *status* is also often used for the resulting value of an invocation.

**Usage:** The *sv* will be assigned to whenever a state transition occurs. This normally happens towards the end of a case. The input variable, on

---

the other hand, will only be read from, since the assignment of new input values will happen outside of the state machine.

**Definition:** The state variable must be defined globally, whereas the input variable is a most likely a parameter to the function containing the switch-statement.

After having identified the state and the input variables, list all potential values that they can have (find all assignments to $sv$ or, if $sv$ is of an enumeration type, find its definition). The switch-statement should provide one case for each of the states and/or each of the input values explicitly named.

**Single or multiple state variables:** In the simplest and most common case, the current state is represented by the value of a single variable. But there are cases where the state is represented by a set of variables.

For example, the state here is defined by a set of flags, each one being true in the exclusion of the others.

```c
bool inTeX = false;
bool inComment = false;
bool inString = false;
bool inClause = false;
```

There may also be flags that encode state in addition to the main state variable, e.g.

```c
bool isTransitionToFirstActivation;
bool isCreateAndGo;
```

These flags may indicate an additional state that is not represented by a potential value of $sv$. If you find secondary state variables consider extending the range of the primary state variable (adding new primary states) and getting rid of the secondary ones, thus making the state machine more explicit.

**Extracting the Transitions**

A state machine, in the course of its execution, transitions between its states. For each state, determine the transitions that leave it. A transition is characterized by an assignment to (one of) the state variable(s) $sv$.

The value of $sv$ at this position indicates the new state of the state machine. If $sv$ was not assigned to before the transition is concluded, a self-transition has occurred, leaving the state machine in the same state.

A return from the function containing the switch-statement or else-if chain may indicate a transition to the end-state of the state machine. If
there are such direct returns from within the state machine, consider creating a separate 'STOP' case and transition to this state instead of returning directly. At the cost of an extra iteration through the machine you will have a more explicit model, and you will be able to more easily handle refactorings like extracting action code into separate functions. The action code of the 'STOP' case will also be the right place for cleanup actions that must be performed when the state machine exits.

Extracting Action and Transition Code

A state machine usually attaches some actions to either the states, the transitions, or both.

For each state determine precisely which code belongs to it. This code is the action code and it will also contain the transition code. The following example presents the difference between state code, which is executed each time the state is entered, and transition code, which is only executed for a specific transition.

```c

```case INIT:
  lnp = do_read_line(&curproc); // state action
  if (INSTR(lnp) == ps_pro) {
    state = AFTERPRO; // transition 1
  } else {
    state = NORMAL; // transition 2
  }

  headl = lnp; // transitional action
  lp = &lnp->l.next;
}
break;
```

Some `switch`-statements use the `fall-through` mechanism, e.g.

```c

```switch (state) {
  case stateA:
  case stateB : actionCode1();
    if (status == stateA) { break }
  case stateC : actionCode2();
    break;
}
```

A fall-through, especially a conditional fall-through, is usually harder to understand than if all cases conclude with a `break`. The solution below is therefore preferable:

```c

```switch (state) {
  case stateA : actionCode1(); break;
  case stateB : actionCode1(); actionCode2(); break;
  case stateC : actionCode2(); break;
}
```
This solution also shows the advantage of encapsulating the action code into a function: it can be invoked from multiple states.

### 3.3 Reengineering a State Machine

Multiple design guidelines exist for implementing a state machine. We identified the following forces to impact the decision for a particular solution:

* **Future changes.** How will the state machine change in the foreseeable future? How will that impact the states, transitions or actions of the state machine. Solutions differ in the effort that is required to make such changes and extent to which they scale.

* **Time of changes.** When are changes to be made, and how much time will be foreseen for the update? If high availability of the system is important, dynamically updating the state machine may be a requirement. A solution where states and transitions are easy to update and extend at runtime, such as a table-driven implementation, then becomes more favorable.

* **Experience.** The experience of the developers in either procedural or object-oriented techniques influences the decision in the sense that a learning curve has to be taken into account if the developers are not accustomed to the chosen solution.

Of the set of solutions, we present two patterns discussing refactoring a state machine implementation to the common solutions of using the object-oriented state pattern and a procedural, function-based approach.

#### Pattern: Refactor to an Object-Oriented State Machine

**Intent:** Refactor a complex state machine implementation to a state machine according to the object-oriented state pattern by introducing a subclass hierarchy representing the various states.

**Solution**

The object-oriented state pattern is a well known design pattern from [GHJV95]. The state pattern (see Figure 3.2) represents a state machine by the so-called context object which acts as the interface to the clients of the state machine. The states of the machine are all implemented in separate classes, each one inheriting from an abstract STATE class. The CONTEXT class holds a pointer to the current state and delegates handling of the input event to the state. When a transition occurs, an instance of the new state is created and replaces the old state in the context class reference.
The object–oriented solution exhibits the following benefits (see also the Consequences section of the state pattern in [GHJV95]):

- Polymorphism removes one level of `switch`–statement, that is needed in the procedural implementation. This makes the code more maintainable in light of potential changes in the number of states.

- In the OO solution, a `switch`–statement growing with the number of state is not needed since dispatch is done via polymorphism. If there are many states, getting rid of an unwieldy `switch`–statement improves the understandability.

- If the state is represented by a multitude of variables, a state object hides this fact. Since from the the viewpoint of the context object, a state transition is only a single assignment, state transitions are then always atomic.

- States are made explicit. A `CONCRETESTATE` class contains only code pertaining to a single state. This focuses the attention of the person reading the code and makes the code more understandable.

- If the state must be represented by the values of multiple variables, the danger of state changes that leave an inconsistent state arises. Encapsulating all state variables in an object will make state changes atomic again.

The following trade-offs play a role in preferring an alternative solution over the object–oriented pattern:

- Mechanisms like polymorphism are only available in an object–oriented language. Even with a suitable programming language, the success-
ful application of object–oriented design principles depends on the familiarity of the developers with the paradigm.

- Depending on the size of the state machine, an implementation using multiple classes can introduce too much boilerplate code, as the number of states corresponds to an equally large number of classes to be created. If there is only a small number of input events that are recognized in a certain state, the overhead of creating a new class for each state makes this solution inefficient. Additional classes also tend to pollute the namespace and design diagrams. As such, in the case of a state machine of limited size, a single switch–statement may be the right level of abstraction to make the code understandable.
Pattern: Refactor to a Procedural State Machine

*Intent:* Refactor a complex state machine implementation to a procedural design, with a set of functions representing the various states.

*Solution*

In the procedural implementation of a state machine, states are modeled by functions instead of classes. For each state we have one function which associates the actions with the given inputs. We also need an initial dispatch function, which, based on a variable that holds the current state, dispatches the control flow to the function that represents this state. The following example shows a procedural implementation with two states:

```c
void initialize() {
    currentState = idle;
}

void dispatch(int inputEvent) {
    // dispatch control flow to current state
    switch(currentState) {
        case idle: idleState(inputEvent); break;
        case busy: busyState(inputEvent); break;
    };
}

// handle 'idle' state
void idleState(int inputEvent) {
    if(inputEvent == START) currentState = busy;
}

// handle 'busy' state
void busyState(int inputEvent) {
    if(inputEvent == STOP) currentState = idle;
}
```

*Tradeoffs*

The procedural solution exhibits the following benefits:

- It can be applied in any programming language, i.e. it does not rely on object–oriented features.
- The procedural way provides a more compact solution. We are assuming here that the state machine is only a small part of the system and don't want to put too much emphasis on it, which a design with
multiple classes—as required by the state design pattern—would do. The entire state machine can be implemented in a single source file which reduces clutter in the source tree. As an example for the understandability of smaller switch-statements, see the following code which implements a comment remover for C/C++ programs.

```c
switch(state) {
    case INCODE: {
        switch (c) {
            case '/': c = peek();
                        if (c=='/') { nextline(); break; }
                else if (c=='*') { append(' '); advance();
                                      state=INCOMMENT;}
                else { copy_advance(); break; }
        break; }
    case INSTRING: {
        switch (c) {
            case '\\': copy_advance(); copy_advance(); break;
                case '"': copy_advance(); state=INCODE; break;
                default: copy_advance();
        break; }
    case INCOMMENT: {
        switch (c) {
            case '*': if (peek()=='/') { advance();
                                      state=INCODE; }
                            advance(); break;
                default: advance();
        break; }
    }
}
```

This state machine, consists of three states, and four transitions, and is implemented in about 25 lines of code (disregarding the helper functions). Since we can keep the entire state machine on as single page, the procedural implementation is preferable over the OO solution.

- A dispatch function which does not contain any action code is self-documenting in that it gives a good overview over all the existing
states.
• We are free to change the order of the two dispatches needed in the procedural solution. If there are many more inputs than states we just dispatch on the input first, and then on the state. This results in more but smaller methods.

The following liabilities may arise from the application of the procedural pattern:
• At least two switch–statements are needed to dispatch the control flow to the correct location.
• Since we must write at least one dispatch based on the value of the state variable $sv$, having a large number of states will make procedural code difficult to understand.
• The number of states determines the size of one switch–statement of the procedural solution.

Refactoring Steps
Once a decision has been made for either the object–oriented or the procedural implementation, the refactoring steps are similar for both cases.

1. Define a blank canvas for the new implementation. This can be a fresh function or a new class. The code will be moved gradually from the old to the new location. At the end, the old location is an empty shell which delegates to the new place.

2. Encapsulate all action code in separate methods. This makes the state-handling function more transparent, showing all different inputs that are handled at a glance. If possible, try to separate the action code from the transition code which again improves understandability.

3. Use naming conventions for the methods and variables. For example:
   • The method which dispatches on the current state has a name such as $dispatch()$, since it is the entry point into the state machine.
   • In the procedural case, every method representing a state should end in State.
   • The variable storing the state value should end in State or Status as well.
• The name of the variable containing the input value that determines the transition to the next state should contain the string command or action.

Naming conventions let you separate dispatch–methods from methods implementing the actions of the state machine. Separating these entities lets the maintainer selectively improve his understanding of what the system does. Moreover, it will isolate the impact of future changes (e.g. introduction of additional action code).

4. Test the state machine as before the refactoring. Since the behavior of the state machine is not changed, all existing tests should work without changing.

Related Patterns

• Embedded State Machine Implementation
• Replace State-Altering Conditionals with State [Ker05]

Chapter 4

Refactoring for Performance

Object-oriented implementation techniques are highly debated in the embedded world, as they are said to be memory and performance hungry. This chapter reports about a pattern to refactor a system containing redundant objects, i.e. data structures where the motivation for object modeling is weak, or does not weight against the performance implications. Such objects can be refactored towards primitive types complemented with supporting operations on these primitives.¹

Pattern: Refactor Redundant Objectification

Intent: Improve system performance by refactoring redundant objects to primitives.

Problem

In a protocol or data record driven system, hundreds of parameters can be stored in a neatly decomposed object hierarchy, corresponding with a standardized configuration hierarchy. Next to managing these parameters by offering standard access and modification operations, a limited amount of special operations are provided for all object types. When a large number of such objects high in the hierarchy are created, the system’s performance may suffer due to (i) the time required for the creation of these objects and (ii) the memory to store these objects. We call such a design ”Redundant Objectification”. In a context of limited resources, such a design may hinder the system’s deployment on its target, or prevent the implementation of new requirements.

The parameters under question are modeled as leaf objects of the system hierarchy, although they merely stand for a primitive value or a list of primitive values. We call them wrapped primitives, as the classes of these objects are instantiated from re-implement the operations that can be issued on primitive types (arithmetic, casting, ...). Only a couple of additional services (such as persistence, serialization, etc) are introduced via additional operators.

This problem is difficult as

- You need to be aware of the amount of wrapped primitive instance that are created at run-time to be able to compute the expected gain.
- By refactoring towards primitives, the uniformity of the services that are offered for all objects in the system is traded for increased performance.
- Modifying the interfaces of the (wrapped) primitives requires client code to be adapted everywhere the additional services are being used.

Yet, solving this problem is feasible because

- The expected gain can be calculated using the specifications, developers’ experience or manual inspection.
- Locations in client code that require modification can be searched for, thereby estimating both the risks and effort of the operation.

Solution

The solution contains the following steps:

- Identifying Improvement Opportunities.
- Assessing Impact and Risk of Changes.
- Risk of Changes

Identifying Improvement Opportunities

Use two criteria to identify data structure where the design can be improved:

- “No substantial new functionality” is associated with the data. Mere data holders that do not provide operations—beyond getters and setters—on their members will only introduce an additional level of indirection.
Since they do not make use of object-oriented features, remodeling them using basic types would only marginally affect the client code. Note that the presence of additional, common framework services is disregarded here. These need to be implemented in another manner, thereby sacrificing some of the system’s uniform interface to achieve the required performance enhancements.

- A large number of these objects must be present in the system. Only then will refactoring them contribute substantially to the reduction of the memory overhead.

Assessing Impact and Risk of Changes

We start with estimating how much effort it will cost to refactor the client code. As these primitive wrappers typically form the core of the system, the impact is large. Be aware of possible wrapped primitive derived classes.

In order to assess the effort needed to perform the refactoring, we need to know where these entities are being used. Standard UNIX commands such as find and grep come to help here. Search for declarations involving a primitive wrapper: class attributes, method parameters, local variables and the primitive wrapper class definition itself as well. This search localizes usages and indicates the scope as well.

Secondly, localize instance usage. Detecting such usage is less obvious as there is no reference to the name of the primitive wrapper, complicating the search query. Therefore, you may need to rely on naming conventions. Instance variables, local as well as temporary variables may be named according to a common scheme and notation (e.g. Hungarian notation). Skim variable declarations to verify the presence of such a scheme. The command in Listing 4.1 shows how to extract a number of object variables of the wrapped primitive type Identifier from the source code, knowing that instances are typically named groupId or lineId.

Listing 4.1: UNIX command pipe to extract occurrences of variable names groupId and lineId from source code.

```bash
rm -rf tmp.txt;
for codefile in `find . -name "*.cpp|c|cc|hpp|h"`;
   do egrep -n -H "(groupId|lineId)\." $codefile >> tmp.txt;
done
```

Another option consists of building up a model of the system and query for instances of such primitive wrappers.
**Risk of Changes**

Because the procedure to refactor wrapped primitives is equal for all of them, categorize changes and count frequencies for each type of change across all types. Using simple search commands, you can list occurrences of type declarations - member variables, local variables, includes, parameters, ... .

For each occurrence type, identify the corresponding refactoring operation. Next to the frequency for these operations, assess its corresponding risk as it influences e.g. the testing effort. Compose risk categories such as:

- **No Change.** Certain occurrences may remain as is. Assignment statements with a primitive wrapper instance as L-value remain unchanged, as the standard assignment operation on primitives will replace the overloaded assignment operator. Several programming languages offer a facility to make an alias for a type (e.g. `typedef` in C++), allowing you to make identify the wrapped primitive name with a primitive type.

- **Harmless Changes.** Removing or replace header file inclusion, removing unused variables, occurrences where operations supported by primitives were used on objects. Creation and deletion operations of wrapped primitives can be removed as well, as primitives are stored on the stack and removed whenever they go out of scope.

- **Trivial Changes.** When refactoring the primitive objects to primitive types, we lose all operations not being supported on primitives. These operations will be refactored into separate functions, e.g. a call to operation `opX` of object `obj` `obj.opX(par1, ..., parN)` needs to be refactored to a function `opX` with the primitive `prim` as first parameter `opX(prim, par1, ..., parN)`.

- **Dangerous Changes.** Certain operations may be dangerous to replace. For example, take C++ cast overloading. The original operation on the wrapped primitive might have a different implementation than the default operation on the primitive, yet the compiler may suggest that no change is required.

For both the harmless as well as the trivial changes, a succeeding build will provide satisfying confidence of a behavior preserving refactoring operation. The dangerous changes require a safety net in the form of a regression suite, to avoid that subtle implementation differences go unnoticed.
Refactoring Steps

Finally, perform the necessary refactorings. For each type,

- Introduce support functions for the non-default operations on the type.
- Replace the wrapped primitive by a primitive. Use aliasing if possible to keep the type name.
- Perform the changes in the harmless, trivial and dangerous categories.
- Rely on the compiler and regression test suite to confirm the behavior preserving nature of the refactoring operations.

Tradeoffs

Pro’s:

- Structured approach with estimation techniques for impact and risk.

Con’s:

- In system with dynamic configurations, estimating the amount of objects in at run-time may be unreliable. In such a case, the performance gain becomes hard to estimate. Dynamic analysis (e.g. via a debugger) helps to obtain a more accurate estimate, yet may be more expensive to obtain.

- The effort estimation may be jeopardized by the lack of instance naming heuristic. Moreover, the alternative, building up a model of the system, require tool support featuring fact extraction and a query mechanism.

Related Work

Chung et al. proposes a decomposition of performance in a soft-goal interdependency graph [CNYM00], in accordance with the characteristics of efficiency suggested by ISO 9126 [ISO01]. Yu et al. and Andreopoulos propose the use of such soft-goal interdependence graphs to guide refactoring towards improved performance without sacrificing simplicity [YMY+03, And04].
Smith and Williams have published a series of papers on the subject of Software Performance Anti-Patterns [SW01, SW02, SW03]. They enlist the originating problems and indicate the role of refactoring in the associated solutions. The *Redundant Objectification* anti-pattern fits well among this list.

For embedded systems with limited resources (e.g. battery-driven), Temmerman et al. illustrate a refactoring scenario that optimizes the energy consumption by removing redundant data and corresponding accesses as well as moving methods closer to the data [TDC+07].

Demeyer demonstrates that refactoring large conditionals into polymorphic calls may improve the performance in C++ programs [Dem05].

Overbey et al. suggest to automate hand optimizations common in supercomputer programming by a refactoring engine, and to defer their execution until build time, in order to preserve the maintainability of the original code base [OXJF05]. Examples of these refactorings are the unrolling of loops, and the optimization of data structures based on the machine’s cache size.

In the Flyweight pattern, Gamma et al. propose the sharing of objects to support large numbers of fine-grained objects efficiently [GHJV95]. This pattern may serve as an alternative to the primitive type solution for situations where groups of objects can be replaced by a limited set of shared objects.
Chapter 5
Testing Legacy Systems

This chapter hosts patterns that play about testing legacy systems. In particular, this chapter is subdivided in four sections Support Refactoring with (Unit) Tests, Test Effort Estimation, Introducing Test Suites and Test Suite Evolution and Reuse.

The first section, Support Refactoring with (Unit) Test, presents a general introduction to testing, with a strong focus on the creation of maintainable unit tests. The Test Effort Estimation section contains a pattern Estimating Refactoring Test Effort where a procedure to estimate the test writing effort for fine grained white box tests is explained.

To introduce tests in a legacy system, the third section Introducing Test Suites contains the patterns:

- Grow Tests Alongside Complex Refactorings, explaining how one can incrementally refactoring existing coarse grained tests towards small scale tests during system refactorings.
- Isolate Subsystems For Testing details how to break dependencies between a subsystem under test and the rest of the system by introducing stubs that replace neighboring subsystems.
- Introduce a Selective & Robust I/O Regression Suite describes how I/O tests, that are typically change sensitive tests, can be made more robust and targeted towards selective parts of a system.

Two more patterns fit in the context of Test Suite Evolution and Reuse:

- Testing a Code Generator specifies how an evolving code generator can be validated by means of I/O tests.
• **Reuse Existing Test Cases** explain how test cases can make reuse of each other, by looking for (i) similar test behavior under varying conditions; and (ii) tests for extended behavior.
5.1 Support refactoring with (Unit) Testing

During refactoring, a developer tries to improve a system’s non-functional aspects such as maintainability or performance. The functional behavior of the system however, must remain the same. How can you guarantee the latter?

The short answer is that you can’t fully ensure it. The case of verifying the correctness of refactoring falls under the same restrictions as software testing in general. To quote Dijkstra:

> Program testing can be used to show the presence of bugs, but never to show their absence [Dij72].

Still, you can assess the risks involved with refactoring and develop a testing strategy that attempts to minimize regressions, i.e. breaking previously working functionality.

In this section, we first summarize the ideas behind unit testing, and motivate why it forms suitable testing technique to safeguard refactoring operations. Furthermore, we introduce guidelines to reach the unit testing objectives of efficiency and maintainability. The examples in this section demonstrate how one can contribute to these objectives in practice.

Then, we introduce a couple of patterns describing best practice in testing during reengineering: “Grow tests alongside complex refactorings”, “Use stubs for isolation” and “Testing a Code Generator”.

5.1.1 Unit testing as a refactoring complement

The refactoring process is a bottom up approach, consisting of small incremental steps that are well understood and can be composed into larger refactoring operations.

The limited scope of every step makes it easy to complement refactoring with small-scoped testing. You end up with a development cycle which consists of applying a small refactoring step, recompiling the system and executing related tests. The early testing effort signals regressions immediately, preventing bugs from staying in the system until discovery at a later, more expensive testing phase.
The generally adopted term for such a testing technique is called unit testing.

**What is a “Unit” in Unit Testing?**

A unit of code denotes a small building block logically belonging together. In an object-oriented environment, one class or a group of related classes constitute a unit. In a procedural context, a unit is formed by a module or a set of functions gathered in a single source file.

5.1.2 Unit testing: objectives and guidelines

In this section, we introduce a series of quality characteristics a unit tester should pursue to reach the objectives of efficiency and maintainability:

**Automated.** This is objective number one. Unless your tests are automated and thus easy to launch, developer will not persist in executing them as frequent as after every refactoring step.

**Self-Checking.** Write tests in such a way that the expected test result is codified along with the test specification. This allows the testing framework to automatically verify and report the outcome of the test (the actual result). Moreover, by specifying the expected result together with the test specification, a test case explicitly shows the corresponding functional requirements. It makes it easier for developers to use test cases as documentation as well as during subsequent functional changes.

**Isolated.** Make your tests self-contained. Enclose all operation and data to conduct the test inside the test case, and isolate it from the environment when needed. Do not make tests dependent on each other, as you risk to introduce errors that propagate through the test code. Remove dependencies on external resources such as databases, file systems and network access.

**Repeatable.** Time after time, test runs on the same system should yield the same results. Do not leave data, traces or results from tests behind that can stop a next run from properly executing.

**Robust.** Related to the isolation objective, tests should be robust to external events such as parallel activities, shared resources, varying hardware environments, etc.

**Reporting.** The test execution environment should minimize output in case of a successful run, but report as precisely as possible on the error location in case of failures.
**Fast.** Unless the selected tests can be executed quickly, developers will not regularly execute them. You should be able to launch the desired tests with a single command. To avoid developers loosing their focus during test execution, the waiting time should only be counted in seconds. There must be some flexibility towards the set of test cases selected for a run, so that only the relevant ones are executed.

**Specific.** Each test has precisely one objective. It can verify whether the system reacts successfully in a positive scenario, or terminates as expected in an abnormal situation. A test can also document a yet to be resolved bug, or demonstrate that the bug is fixed.

**Necessary.** Avoid cluttering the set of test cases with duplicate or overly obvious tests.

**Consistent.** Specify all test cases in a similar, structured manner that makes the parts a test is composed of (such as setup, stimulate, verification phases) explicit and uniform across all test cases. Developers new to a certain set of test cases will understand tests faster; People new to the environment can be educated quickly.

**Composable.** Make sure that test cases are the essential building blocks which can be executed independent of each other. Executing all the tests all the time takes too much time, you should rather only run the relevant tests during certain refactoring operations. Compose logical groups of test cases into test suites.

**Coverage.** Be aware of what your tests cover, at different levels of granularity (see Test Coverage box).

**Maintainable.** Unit tests must be codified as well, resulting in additional source code that needs to be written and maintained. Therefore, keep test code concise and focused on one objective.
What does “Test Coverage” mean?

“Test coverage” is the degree to which a given test or set of tests addresses all specified system requirements. This degree measures both breadth (how much of a system is covered) as well as depth (at which granularity you exercise the code and check the results). Coverage can be expressed at various levels:

At feature level, you can express the number of scenarios that is exercised during software testing.

For an object-oriented system, you can talk about code coverage in terms of class coverage, method coverage, branch coverage, statement coverage, etc. Test coverage can be seen as a measure of the quality of your software tests.

It is considered a good practice to write a test for every bug that is discovered. The amount of bugs for which there exists a test can also be a test coverage measurement.

The following sections introduce some techniques that help in reaching the stated quality characteristics.

5.1.3 Use a testing framework

Testing frameworks take away the burden of having to develop a testing environment which executes test suites and reports the results. Testers are also forced into a rigid structure when specifying test cases.

We use the following terminology for object-oriented unit testing, originating from Beck [Bec94].

- a **Test Suite** is an aggregator for test cases, allowing developers to combine and execute related test cases.
- a **Test Case** groups a set of tests performed on the same unit under test.
- a **Test Command** is a container for a single tests. It is typically implemented as a method of a test case containing one or more setup-stimulate-verify cycles.
- a **Fixture** is the set of objects that compose the unit under test.
- a **Setup** initializes a test case’s fixture into the desired state for testing.
• a *Stimulus* is a message sent to the unit under test to provoke a certain reaction

• a *Check* fetches this reaction and checks it against the expected result. Assert statements compare the actual result with the expected one and report accordingly.

Beck’s initial SMALLTALK implementation was quickly followed by ported implementations for other programming languages. These implementations are referred to as the *xUnit* family of testing frameworks and are today’s *de facto* standard in unit testing [Ham04]. The family contains implementations of the framework for all major programming languages and environments are available: procedural languages such as FORTRAN and C, object-oriented environments such as C++, JAVA [HT04] and C# as well as scripting languages like PERL [3], PHP [4], PYTHON [Pei04], RUBY [5] etc.

An example test case written for CxxTest [6], a C++ *xUnit* family member, is given in Listing 5.1. You can distinguish the STCPPAPI test case, which is the subsystem test for the C++ API of a certain subsystem. In CxxTest, test cases inherit from the generic TESTSUITE class.

Listing 5.1: A sample CxxTest test case

```cpp
#include <cxxtest/TestSuite.h>
#include "SstCppApi.hpp"

class SstCppApi : public CxxTest::TestSuite
{
    // Fixture
    List<Person> persons;
    Person person;

    public:
    void testAddToList( void ) {
        // (1) Setup
        strcpy(person.name, "James");
        strcpy(person.surname, "Bond");
        person.age=38;

        // (2) Stimulus
        persons.saveObject(&person);
        persons.flushCache();
        Person first;
        res=persons.getFirst(&first);

        // (3) Check
        TSM_ASSERT("Person's age must remain after saving",
                   first.age==per.age);
        TS_ASSERT_EQUALS(0, strcmp(first.name, per.name));
        TS_ASSERT(0 == strcmp(first.surname, per.surname));
    }
};
```

The fixture consists of two objects, persons and person. There is no explicit setup method, but one should be introduced when multiple test commands would exercise the same unit under test. Furthermore, there is one test command, called testAddToList. During setup, the person object is initialized with some data. Next, this object is saved into the persons list, which is the stimulus sent to the unit under test. The check starts with retrieving this person object, followed by three assert statements checking the correctness of the retrieved object.

A successful execution of a suite of five test cases yields a minimal output, in order to directly conclude this success:

```
./subsystTest.exe
Running 5 tests.....OK!
```
In case there were failures in the test run, you want to be point to the exact line of the problematic test. In that case the output of CxxTest looks like the sample:

Listing 5.2: Example output of a CxxTest run with errors.

```bash
./subsysTest.exe
Running 5 tests...
In SstCppApiSuite::test_noDynamicMemoryAllocation:
  /vobs/.../home/sst/SstCppApiSuite.h:33:
    Error: Test failed: Person’s age must remain after saving
  /vobs/.../home/sst/SstCppApiSuite.h:33:
    Error: Assertion failed: first.age!=per.age
...
Failed 1 of 5 tests
Success rate: 80%
```

Coming back to the unit testing objectives, using a testing framework helps to achieve the following aspects:

**Automated.** Tests run automatically after issuing a single command.

**Reporting.** As shown in the listings, the output is indeed minimal in case of success, but explicit when failures occur.

**Composable.** xUnit knows the concept of a test suite, and even allows the developer to execute individual test cases.

**Consistent.** Tests are specified in terms of test cases, fixtures, setups, stimuli, test commands and checks. xUnit implementations impose certain naming conventions which make test commands, setups and checks recognizable by their name.

**Self-Checking.** The testing frameworks checks the results of every test, by comparing the actual outcome during test execution with the expected result specified in the test by the developer.

### 5.1.4 Test Pragmatically

Testing time is limited. After all, it does not directly yield value for the customer. Even the refactoring activity might be hard to sell to the project management as the benefits play in the long term and won’t be visible at first.
Therefore, within the available time, you should spent effort wisely. A software project carries a lot of information that helps you to decide where to test.

The following advise helps you to optimize test effectiveness [DDN02].

- **Grow your tests incrementally.** Write tests that are relevant for the next refactoring operation.

- **Bundle tests in regression suites.** Gather all the tests you develop over time into easy-to-execute regression suites. Run them periodically.

- **Concentrate on fragile parts.** Focus testing effort on fragile parts, which you identify by asking developers and studying the system’s history. You look for parts that are complex, frequently changing, poorly understood and error prone, and that will need to be changed in the future. Use the versioning system and the current snapshot to detect frequently changing and complex parts.

- **Test the interface, not the implementation.** If the test results of a black box testing approach against the interface are according to the specifications, it’s not worth investing time in testing the internals.
5.2 Test Effort Estimation

Pattern: Estimating Size of Functional Test Code

**Intent:** Given a system’s source code, estimate the size of the corresponding, required functional test code.

**Problem**

Alongside development, there are software tests to validate the system against the requirements, to reveal and document defects, etc. Several types of testing are encoded to automate their execution. Writing and extending the associated test cases can add considerably to the overall programming effort. Typically, this effort is either not taken into account, quantified by a default value or roughly estimated. Yet, given production code and the desired test coverage criteria, the size of the test code to be written can be calculated as an additional source for estimating testing effort.

This problem is difficult, because the test code size depends on system characteristics such as complexity or coupling, as well as on test code properties such as the use of a test framework, the size of units, the desired coverage, etc.

Yet, solving this problem if feasible because these characteristics can be taken into account as parameters in the calculation, by either measuring them from the source code or indicate test requirements.

**Solution**

Consider the function (an invokable entity in the system such as a function, method or procedure in the source code) as the unit of granularity for this calculation. The following steps then describe how to apply an estimation procedure for the amount of test code required to test a set of function to a certain path coverage level.

**Function Identification.** This subset of functions $F$ can be composed by collecting the functions of units under test, by aggregating the systems’ interfaces, by identifying individual functions, etc. As such, given such a set, the corresponding testing effort can be described in terms of a number of functions to be tested multiplied by a factor per function that describes the extent to which that function requires test code.

**Function Testability.** To estimate the test effort for an individual function, we rely on two internal metrics characterizing its testability:
• In order to test individual functions, a test harness needs to be put in place, isolating the function under test (FUT) from the environment to a certain extent. Multiple approaches exist between full isolation and no isolation at all. When the function is tested in complete isolation, the test harness must provide stubs for all external functions the FUT interacts with. In case of an integration test, no isolation is required. Another, common possibility consists of isolating the function from all production functions, but keeping the binding with library and system interfaces. The number of external functions a function interacts with is summarized by the FanOut metric. For each of these external, stubbed functions some code needs to be written. Typically, stub code is limited to the minimum that is required to imitate the actual external function's behavior in the context of the test. As such, an average stub size can be estimated and multiplied with the number of stubs to be created in order to calculate the contribution of stub writing. In a formula, we calculate the amount of stub code as $SCa \times FanOut_x$ with $SCa$ the average stub size and $FanOut_x$ the tailored FanOut measurement for function $x$.

• The test harness needs to provide the necessary means to invoke the function. We foresee a test case per function with a setup part where the environment is build up. In particular, during a setup phase instance variables satisfying each of the parameter types of the function are created. The number of parameter of the function (NOP) thus determines the amount of setup required. The average amount of code per parameter is encoded in the variable $PCa$ representing the limited number of lines required to construct the parameter type instance variable and initialization with some test data. Furthermore, we add an average number of lines to initialize the module or object of the function under test $UIa$ and some initialization code to put the stub in place $SIa$. Summarizing, for each function $x$ we add $PCa \times NOP_x + SIa \times FanOut_x + UIa$ setup code.

• Finally, we incorporate the complexity of the individual functions in the amount of actual testing code. The more complex a function is, i.e. the number of paths that can be followed throughout the function body, the more test code is required to cover a satisfying subset of all paths. The McCabe’s Cyclomatic Complexity (MVG) metric quantifies this complexity. Assuming an average testing line count $TCa$ needed to reconfigure some actual parameters, call the FUT (aka the stimulus), fetch the result and compare it with the actual, expected result (the verification) for a single path; and defining $COV$ as the desired path coverage percentage the amount of testing code for function $x$ in terms of its complexity is estimated at $COV \times TCa \times MVG_x$. 

62
Calculating Total. Finally, the overall test code size can be calculated by summing up the contributions of individual functions. Table 5.2 summarizes the calculations for each of the three test writing tasks stub writing, test case setup and actual testing code. The overall formula then appears as

\[ \sum_{x \in F} SCa \times FanOut_x + PCa \times NOP_x + SIa \times FanOut_x + UIa + COV \times TCa \times MVG_x. \]

Tradeoffs

Pro’s:

- This estimation technique is well suited for development methodologies with a larger coding/testing cycle. In particular, this approach is useful in a test-first high-risk, high-impact refactoring context, such as reducing coupling between subsystems, abstracting away recurrent code blocks or moving behavior closer to data.

- Lightweight technique that uses objective measurements to obtain an effort estimation.

- Individual estimation numbers can be tweaked to a particular environment.

Con’s:

- The kind of testing environment nor test implementation language are taken into account, although these factors logically have an impact on the estimations for setup as well as actual test code.

- Depending on the design of the system, especially the extent to which coupling and cohesion are taking into account as indicators for a good decomposition, isolating a function may prove to require orders of magnitude more effort. The estimation model does not measure this, abstraction the effort into a constant used for all affected functions.

- The estimation technique is only applicable to fully encoded tests that directly use the product code. The code size of I/O tests, command

Table 5.1: Contribution of function characteristics to test code parts.
language tests, generated tests, etc. can not be estimated using this procedure.

- The technique only reports size of the test code to be written, but does not say anything about the effort required to document, review, debug, archive or specify test cases.

**Difficulties**

- The estimation model requires measurements about individual function in the system, which is information that is hard and time-consuming to obtain manually. Moreover, tools that automate this are scarce and require a model of the system and a querying mechanism.

- Stubs can vary heavily in size, depending on (i) the operation that they replace and the (ii) responsibility of the stub. For example, a stub for an event observer may be a lot harder to construct than a stub for a database connection. Stubs that verify the input they receive and check the interaction patterns with the unit under test (called mock objects) also contain more code. As such, the average stub size may be hard to estimate.

### 5.3 Introducing Test Suites

**Pattern:** Grow tests alongside complex refactorings

**Intent:** To refactor complex code, grow and co-evolve a test suite alongside the refactoring steps.

**Problem**

You are about to refactor a piece of complex code (e.g. long function with high cyclomatic complexity, lots of duplication, extensive switch statements, etc) into properly abstracted, manageable chunks. The available tests are not very trustworthy as certain tests fail and the coverage of the targeted system part is poor. As you are afraid of introducing defects during the refactoring and therefore would like to extend the test suite first.

The current code is however hard to test, thereby requiring (i) an excessive amount of time to isolate components; and (ii) compose the necessary scenarios with associated input data and expect values. Testing the target design would be a lot more efficient, as the introduced abstractions are made to be tested in isolation.
You are faced with a dilemma: proceeding with the refactoring and paying the price of the introduced defects, or spending time writing tests thereby delaying the actual intent.

Solution

Refactoring is a stepwise activity, and so can testing be as well. Introduce white box tests for the parts that will be touched by the next refactoring step and gradually move to the target design, thereby refactoring both code and tests synchronously. By storing the intermediate versions, you have the possibility to go back to a previous version in case the tests indicate that defects were introduced.

The following steps describe the procedure to incrementally split up a monolithic code block into testable chunks.

- Initially, you can start with a couple of black box I/O tests that you may find among the existing test suite. Consider expanding them to some extent to (i) learn about the system and (ii) cover some important paths.

- When breaking apart big components, split up the tests as well. Catch the output of successful tests at break points and provide it as input for a new test focusing on the code after the break.

- Write tests for the abstractions that you introduce, even before the actual abstraction has been implemented. This will force you into thinking how to write isolated, testable components (this idea stems from test-driven development).

Tradeoffs

Pro’s:

- The test suite harnesses your refactoring operations are harnessed. Successfully ran tests after every refactoring step prove the correctness of that step, and give you the confidence to proceed.

- In case a test fails, you can go back to the previous step and repeating the last refactoring step differently based upon the acquired experience.

- The more you advance, the more control you obtain due to (i) increased program comprehension and (ii) increasing coverage due to the growing test suite.
• The tests specified during this refactoring operation can be reused in the regression suite during further system evolution.

Con's:
• As the initial complexity makes it difficult to get a grip on meaningful test scenarios that provide you with a reasonable coverage, you inevitable will start with an incomplete harness. However, you can keep it under control by balancing the measured code coverage with the time required to proceed to a higher coverage requirement.
• You will introduce test code that needs to be evolved as well, resulting in additional effort.

Difficulties
• Deciding where to introduce stubs to isolate the component from the environment is difficult as stubbing every external call is time-consuming. But, the less you isolate the less traceable the reason for a failing test will be, as you can also not rely on the correctness of these neighboring components in the first place. The extent to which these are covered can be an argument to decide for or against stubbing them.

Related Work
"Grow your tests incrementally" and "Use a testing framework" [DDN02].

Pattern: Isolate Components: Divide and Conquer

**Intent:** Setup a build architecture that allows the tester to isolate components during unit testing. In a test build, dependencies with other components are replaced by stub placeholders.

**Problem**

In a software system, components interact which each other to achieve certain functional goals. During unit testing, you want to verify the correctness of a single component. Therefore, the component is tested in isolation of its neighbors to facilitate the tester to bring the unit in the desired state,
to remove unwanted side effects during certain scenarios and to prevent errors residing in other units to propagate and mislead the tests.

This problem is difficult because:

- Without the components it depends on, a component can not be built.
- A component under test (CUT) expects a reply on calls to its dependencies.
- A build environment is hard to modify, especially towards intrusive changes such as reduced builds.
- Removing dependencies between tightly coupled components can prove to be time-intensive.

Yet, solving this problem is feasible because

- To obtain a working build, empty implementations for dependencies’ used interfaces serve as placeholders for the actual components.
- A minimal implementation returning a desired response can be put in the stub implementation.
- Rather than changing the existing build system, you can extend it by adding instructions to make a build composed of the component to be tested, the stubs corresponding to its dependencies as well as the test environment and test cases.
- You can chose, pragmatically, the scope of the units under test.

**Solution**

To test a component in isolation, without interference from the environment, a test build in which dependencies are replaced by stubs is made.

---

**What are Stubs?**

A **stub** is a dummy procedure used when linking a program with a run-time library. The stub routine need not contain any code and is only present to prevent "undefined label" errors at link time.

A **mock** is special case object that mimic real objects for testing and verifies the way it is used during testing [Fow07].
1. Separate a component’s interface from the implementation (Figure 5.1). This requires the extraction of the public members of the component into a separate entity (e.g. an interface in Java, a pure abstract class in C++, a set of function declarations in C, etc). In an object-oriented context, members should be called on the interface rather than on the actual type in order to relay calls to the actual implementation in either production or test via late binding.

2. For each component to be tested, create separate, stub implementations for the depending components’ public interfaces. These stubs contain the smallest possible implementation that can return a response as desired by the CUT.

3. Identify how the build system composes the system by building individual components which get assembled in the end. For each component, introduce separate build instructions that allow the developer to make an isolated build composed of (i) the component under test (CUT); (ii) stub implementations for all interfaces of components the CUT depends on; (iii) test cases associated with the CUT; and (iv) a testing framework as execution entry point (Figure 5.2).

Tradeoffs

Pro’s:

- This testing approach is non-intrusive, it keeps the unit under test free of test code which would otherwise clutter up the actual implementation via test handles.
Figure 5.2: Architecture for isolating components. Figure courtesy of Alcatel-Lucent.
• The isolated environment allows you to test the correctness of the unit under test without undesired side effects and without complex setups of connected components.

• An isolated, reduced test environment builds and runs faster, providing near instantaneous test feedback.

• Reuse. Stubs for a component C that serves as dependency for component A are reused in case component B also depends on C.

Con’s:

• Providing basic stub implementations requires both white box insight in the component as well as domain knowledge about what a standard response is. Moreover, after initial introduction these stubs will need to be co-evolved with their corresponding production component as well.

• Different test scenarios expect different responses from dependencies: one test verifies whether a default, positive scenario proceeds as should, while another test scenario might verify the component’s behavior in case of exceptional circumstances. This requires customizable, more complex stubs.

• You might struggle to setup an isolated environment in case of a decomposable system. In practice, there is often a discrepancy between the planned interface usage restrictions and the actual source code, where coupling is higher due to illegitimate calls to internal interfaces of other units or components. You will have to decide whether to remove (refactor) those additional dependencies first, or whether you will spend additional effort writing stubs for them.

Difficulties

• You might consider not to stub all components the unit under test is connected with. Certain components can prove to be too difficult to stub, are known to be well tested or are just convenient to include in an isolated environment.

• In an embedded systems environment, a software system often contains low-level components that deal with memory and data management or components that communicate with hardware. Such systems can also be developed on simulated host environments, in which case the lower level components already are stubs which you can reuse.
• Extract interfaces is an intrusive refactoring operation, as it requires replacing type declarations by interface declarations throughout the system. You want to "Grow test cases alongside complex refactorings" in case the existing test suite is not satisfactory.

Related Work

Feathers describes, via thorough examples, how to break dependencies and introduce isolated, testable units by means of dependency breaking techniques that he calls seams [Fea05].
Pattern: Introduce a Selective & Robust I/O Regression Suite

**Intent:** Introduce a batch of small, robust I/O tests to obtain a fast and maintainable regression suite.

**Problem**

Suppose a I/O intensive system, i.e. a system that can be directly linked with input data as well as expected output. There are no or only a limited amount of tests available for this system, thereby making a refactoring mission on this system a daunting task. However, you do have the deployed system and were successfully able to execute a couple of manual scenarios. As the system is relatively mature without major bugs in recent months, you are confident that the current output corresponds to the specifications. How can you build up a regression batch in a limited amount of time?

This problem is difficult as:

- As writing tests for every fine-grained feature is too costly, a selection of test cases needs to be made. This selection contains input data targeting components impacted by the refactoring.

This problem is feasible, because:

- Documentation for a standardized input format helps to identify atomic units of input.

- Writing test infrastructure for this approach is limited to aggregating individual system runs with test input data in a regression batch.

**Solution**

Apply the following steps to obtain the I/O regression suite.

**Test Case Composition** Compose (and select) I/O test cases that are useful in the context of the refactoring mission. There are multiple approaches to map input data onto system components. Talk to the developers to learn from their inside knowledge. Skim the documentation for data categories, decision points and boundary values. Speculate about design and compose sample input data for each test case that you feed to the system.

Keep the input data small and focused. This will help to identify the root cause of a test failure during refactoring, as you lack the explicit traces to the root cause that are typically present in white box tests.
**Store Expected Output.** Perform a run for each of the test cases and inspect the resulting output. Compare this result to the specification if available, else consider the system as an oracle. Store the resulting output as the expected test case output, to be used during the remainder of the refactoring case. Use a profiler to verify where the program flow passes through expected components. Learn from the output of an initial run of test cases to compose additional test cases.

**Build Test Framework** Compose and collect a couple of auxiliary functions supporting regression batch execution in a small test framework.

Introduce a mechanism that compares the actual outcome with the expected result and reports in a condensed way. A script to check line-per-line identify can be written in a couple of lines. Or, the output of a `diff` tool could be used. Next, build a small reporting engine. When a single test case passes, the output should be minimal. When a test case fails, the report should mention the test name as well as the difference. Finally, compose a script that collects all test cases in a suite that can be executed at once.

**Improve Test Robustness.** To improve the robustness of the I/O regression batch, the expected output can be further reduced by removing less relevant output aspects (including syntactical sugar in the output). For example, you can consider checking whether certain (parts or sequences of) lines appear in the output instead of storing the expected output as a full copy of the initial system output.

At one side, such a test approach reduces the defect detection power of the regression batch. Indeed, in case the tests have to cover a phase of pure refactoring in which output must remain the same to the letter, this kind of reduction in verification is not desired. At the other side, when refactoring is intertwined with forward development, new functionality can easily break the majority of tests, resulting in maintenance-prone tests. A separate test, with more typical, integrated input data can serve as verification mechanism for strict refactoring operations in that case.

**Build Extensions.** Additional extensions, such as a mechanism that allows you to specify which line sequences are not allowed in the output, are possible. They will however complicate the comparison mechanism of the test framework. The test suite can be divided over sub suites responsible covering subsystems, test goal, etc. E.g., some tests may exemplify a system bug and are allowed to fail until the bug is fixed. Such tests are not desired in a regression batch.
Selective Regression Testing. When composing a I/O regression batch, test run-time may soon become too long. Apply a selective regression testing approach, e.g. Retest Risky Use Cases, Retest by Profile or Retest Changed Segments [R.B00D].

Tradeoffs

Pro’s

• Writing I/O tests for an unfamiliar system is a form of Write tests to understand.

• Tests that succeed serve as regression suite that can incrementally grow. Developing an individual test case takes a minimal amount of time.

• This testing approach does not require time-consuming activities such as stubbing or test build integration that are typical to white box testing.

• During phases of forward development, these tests can help in communicating and proving progress.

Con’s

• I/O tests can’t pinpoint the reason of failures as they are black box.

• I/O tests run slower than e.g. white box tests or tests that are integrated in the development environment (i.e. without input/output needs).

• It may be more difficult to measure up the accumulated coverage of a batch of tests and as such to assess the value of the batch.

• Providing input data and expected outcomes may be difficult when the formats are not well specified.

5.4 Test Suite Evolution and Reuse

Pattern: Testing a Code Generator

Intent: During the evolution of a code generator and in absence of unit tests covering the generated code, we want to have a fast means to check that we did not break it.
Problem

Part of a system is being generated by a code generator, based on a number of specification files. The generator is to be changed and is to generate different code for some sub part of the specification. Most of the generated code will however be the same. It is expected that the code generator will have to be changed in radical ways, increasing the danger that the generation of the current code could be affected.

This is difficult because:

- Most likely, the generated part is not covered by unit tests. We therefore do not have a reliable measure which tells us that we did not break the system.
- The new behavior and the changed one are intertwined in the generated classes. We have to retain the old behavior and make sure the new one is correct as well.
- The generator creates a large amount of code, easily hundreds of classes. Confirming correctness of our changes will take time.

Yet, it is feasible to solve the problem because:

- The immediate product of the code generator is not a running system but a number of source files which can be compared with tools for detecting textual differences.
- The generated code will be very uniform (otherwise it would not be generated). Tracking changes in a uniform background is easier than finding them in a jungle of variation.

Solution

In absence of a test harness to ensure the behavior preservation of the generator, we have to write our own tests. During the evolution of the code generator there will be two types of changes applied to it:

- Behavior preserving refactorings: they will not change the output of the generator at all.
- Behavior altering changes: the output of the generator will exhibit changes that conform to a given pattern.

The easiest way of checking that the relevant behavior of the generator does not change is to compare after each step the current output—the generated code—with the previous output. In the first case, a `diff` can tell us
immediately if the refactorings did break the generator, in the second case we have to check for all differences whether they conform to the expected pattern.

To improve the sensitivity of the testing to unwanted changes (and to improve traceability in case of an error) we will also rely on the debug output produced by the running generator. It too represents the status of the generator, and can be compared to previous versions as well.

1. **Preparing the Code Generator:** To improve the traceability of our changes, the code generator should be able to produce copious amounts of debug information. If it does not already do so, it should be instrumented with debug statements at key points in the program. Points of notice

   - List of entities, e.g. lists of generated classes, should be logged as they exhibit state in a concise manner.
   - Since we are using `diff` to compare logs, we can reduce the amount of spurious differences by always using a fixed order when logging lists of entities. Figure 5.3 shows the difference between changing and keeping the order of elements in a list. On the left window where the change of order confuses `diff` we do not see that the effective difference between the two lists consists only in one element. The amount of spurious difference completely disguises the fact that the lists are almost equal. On the right hand window, we see that if order is kept, additions are immediately visible. The easiest way of getting order is to sort alphabetically on names. When a new element is introduced in the list, it will only disturb the comparison in a single position, instead of potentially more. Manual comparison is made easier.
   - The names of the files to which the generated code is written should be logged with their entire path.
   - Logged data that is to be picked up by the testing framework (see below) should be prefixed with unique text markers which can be recognized easily using regular expressions.

2. **Setting up the Testing Environment:** In order to be sure that you can safely evolve the code generator, you need to first determine all the modes in which the generator is run when the project is compiled. This requires you to look at the build system (makefile) for the project and extract all command line flags with which the generator is invoked. Each run that the generator does during project compilation is going to be replicated with a run during testing. Each test run will also be configured to produce the maximum amount of debug output. A test run consists in the following steps:
Figure 5.3: Keeping the same order in lists (see left half) improves on the ability to detect essential differences.

(a) Run the generator with a given set of command line flags.

(b) From the log file, read the names of all files that have been generated. Sort the list of names and concatenate their content into a single file. Insert name and path before the content of the file to enable traceability.

(c) Concatenate all the logs and write them in a different file.

(d) Compare the last version with the current output using a `diff` tool. Both types of output, the generated code and the log files, have to be controlled. Use `diff` only to determine if there are changes or not. This test result must be reported to the user.

(e) As another smoke test, the code produced by the code generator can also be compiled after each step. This requires however that the build system can be constrained to only a selected set of files, as in many cases it will take too long to compile the entire system.

These steps must be performed by a script so that tests can be run semi-automatically. The only human intervention should be necessary for the investigation of the differences between the old and the new output.

3. Investigating the Differences: If there are changes (expected or unexpected) in either the generated output or the logs, investigate incongruencies between the last and the current version with a graphical `diff` viewer. These tools offer the capability to jump from difference
to difference with a single mouse-click. The fact that all the output is in a single file also speeds up the process.

The editor PsPad [7] has good support for this investigation. The viewer KDiff [8] has an even finer-grained display, showing differences on the level of words and characters.

4. **Perform Refactorings and Evolutionary Changes:** During the actual work on the code generator, group the changes which are behavior preserving as much as possible. As they should not change the output of the generator, you will get an Ok from the testing framework for any number of successful refactoring steps.

Make sure to create a new version of the generator before an evolutionary change. Make small evolutionary steps, so that there is only one, or very few, difference patterns that you have to control with the graphical diff viewer.

5. **Recording the Refactoring Steps:** Any reengineering project that is larger than a single refactoring step will incur the risk of taking a wrong turn at one of the steps. In order to be able to undo the refactoring actions, it is advisable to record each step in a version control system. The regular version control system might of course be used for that purpose. We have to consider some issues however:

- We will want to store the output of the latest generator version, the generated code but also the debug output as well. This kind of material is normally not recorded.
- We will want to combine the check-in programmatically with a successful diff: Whenever the comparison of old with new does not show any difference, everything important is checked in automatically.

We are able to implement a simple version control system where we store the refactored file in a directory, adding increasing version numbers to its name (see Figure 5.5 and Listing 5.3). A commentary on what has been changed is stored alongside each version. The decisions taken along the way can then be related to the changes that are found when comparing different versions of the generator. Note that if a handmade solution is chosen for controlling the versions, backup must be organized as well.
Tradeoffs

Pro’s:

- The tests do not involve building the entire system and only depend on how fast the code generator runs.

Con’s:

- The tests require manual intervention, which is a slow and tedious process. Oversights can happen.
- There is no existing testing framework for these kind of tests. It has to be built up from the ground.

Difficulties:

If the evolutionary changes radically alter the output of the generated code, it is no longer feasible to perform visual checks with a graphical diff viewer. In these cases, building the system and using regular unit tests will be unavoidable.

Pattern: Reuse existing test cases

Intent: Make maximal reuse of existing test cases, yet keep tests focused and independent of each other.

Problem

The nature of test scenarios typically results in overlaps between test cases, e.g. fixture data that is partially or entirely the same, or a setup of the unit under test that is shared among test cases or extended upon. Therefore, we want to make use of existing test cases to (i) test under varying conditions and (ii) to test extended behavior.

This problem is difficult, because making reuse of test cases requires a certain interaction between test cases the tests of which have to remain independent. Moreover, reusable fixtures and setups requires the introduction of abstractions that can make it harder to understand the intent of a test.

Yet, solving this problem is feasible because
• Testing frameworks such as xUnit clearly separate, both by means of terminology as by means of test design, between a test case’s fixture, setup, stimuli and tear down.
• xUnit will make sure that tests remain independent.
• a tree of test cases, by means of inheritance, makes it possible to reuse and expand upon each others fixture and setup.

Solution

Identify reuse possibilities by (i) looking for similar test behavior under varying conditions; and (ii) tests for extended behavior.

Varying Conditions. For Test Behavior under varying conditions, reuse stimulate, verify and tear down phases of the test, refine setUp. To enable this reuse, encapsulate the recurrent stimulate, verify and tear down behavior in an abstract test case.

Figure 5.6 shows a sample test design that makes reuse of the specified test behavior. The system's behavior for adding a network node is expected to be the same, regardless of the node being either a Printer or a Workstation. Therefore, the test stimuli and verification statements are encapsulated in the abstract test case AddNodeToNetworkTestCase. The subclasses override the original setUp, initializing a Printer and Workstation instance respectively. Note that the introduction of varying conditions or variants is not constraint to the concept of class siblings in object-oriented programming. Changing data within instances of one type to verify boundary value behavior also fits within this reuse approach.

Extended Behaviour. To build up a chain of tests that gradually requires more test resources, test cases can expand and reuse test cases earlier in the chain. At the beginning of its (overriding) setUp, the subclass calls the setUp of the super class test case. At the end of tearDown, the subclass passes control to the tearDown of the super class.

To refine the test behaviour, new test commands can be written and inherited test commands overridden or neutralized (empty overriding placeholder).

The first test case in Figure 5.7 NetworkTestCase, verifies Network instances and their properties. The second test case, AddNodeTestCase, requires a configured network to stimulate and verify. To test printer functionality a deployed network containing a Printer has to be in place.

Tradeoffs
**Pro’s:**

- Extensive code reuse, reducing the share of test code.
- Differences between test cases becomes more clear.
- Reusing test data leads to familiarization, to typical test data that can be easier picked up and extended upon by outsiders.

**Con’s:**

- Additional abstraction is introduced, requiring testers to inspect multiple classes in multiple files.
- The test execution cycle is no longer isolated in a single test case with setup in setUp, stimulate and verify in a single test command and tear down in tearDown. Rather, these test concepts are scattered. The polymorphic behavior adds additional complexity.
Figure 5.4: A graphical diff view as provided by PsPAD (above) and KDiff (below).
Figure 5.5: Simple Version Control: a directory contains each step of the progressive refactoring of the generate.rb program.

Listing 5.3: A PERL script managing the versions of a script in a repository directory.

```perl
# creates the filename of the next version
# (including the repository-path)
sub makeNewVersionName {
    my $num = 1 + (getLastVersion($repository))[1];
    my $nextversion = $scriptname;
    $nextversion =~ s/\./\$num\r/;
    return "$repository/$nextversion"
}

# looks into the directory where the versions of the
# generate.rb script are located and returns the file
# and the number of the latest step.
# Note that if there are no previous refactoring steps
# then the file name returned is "", the number is 0.
sub getLastVersion {
    my ($dir) = @_; 
    opendir(DIR, $dir) || die "Can't opendir "$dir": $!";
    my $nextversion = "$\scriptname/$num\r/";
    chdir(DIR);
    my ($max, $maxFile) = (0, ' '); 
    foreach (@versions) {
        if (m/\d+\r/ && $1 > $max) {
            $max = $1;
            $maxFile = "$dir/$file";
        }
    }
    closedir(DIR);
    return ($maxFile, $max);
}
```
Figure 5.6:

Figure 5.7: Example of extending test behaviour.
Chapter 6

Architecture Level Restructuring

This chapter hosts patterns that play at a more architectural level. In particular, this chapter is subdivided in two sections Architectural Analysis and Architectural Evolution. The former contains two patterns that help in analyzing the architecture of an existing system:

- **Architecture Reconstruction**, describing the process of obtaining a documented architecture for an existing system by inferring the architectural information from the available evidence.

- **Analyzing System Merging Effort**, to quantify the difference between two forked, diverting systems with the aim of deciding about the subsequent evolution path.

The Architectural Evolution section consists of three patterns that describe two actual migration paths:

- **Create Reusable Component**, detailing the process of extracting a component from a software system to be reused in other contexts or scenarios.

- **Integration of a reusable component into a new system**, integrating a reusable component into a new system, by assembling, adapting and wiring.

- **Phased Migration of a Software Stack**, explaining how to gradually migrate a software stack from one technology to another.
6.1 Architectural Analysis

Pattern: View-based software architecture reconstruction

Intent: Software architecture reconstruction is the process of obtaining a documented architecture for an existing system by inferring the architectural information from the available evidence. The most reliable source of information is the system itself, either the source code or traces obtained from its execution. Other sources of information are the design documentation, the description of supported features, interviews with the system experts. The main goals are:

- To create an architectural model that makes the concrete architecture explicit from the implementation. By confronting the reconstructed architectural model with the conceptual models, the developers can increase their understanding of the system.
- To enable the architects to analyze the various structural dependencies among the software components.
- To enable the architects to check the architectural conformance of the implementation.
- To enforce the architectural rules as they are formulated by the architects.

Problem

The as-implemented software architecture is unclear or unknown.

Solution

The reconstruction process is an iterative incremental process that is divided in three phases: process design, view recovery and result interpretation. We describe the three phases below. Each phase consists of several activities that are detailed in the next section.

The first phase is the Process design. During this phase, the process designer elicits the architectural problems from the stakeholders and defines the source and target viewpoints for the reconstruction. The process designer is also responsible for defining the architecturally relevant concepts and their mappings to the implementation. The phase consists of two activities: problem definition and concept determination. This phase is a conceptual activity that is mainly conducted through workshops with the stakeholders and interviewing the system experts.
The second phase is the View Recovery. The reconstructor is responsible for recovering the target views that have been defined in the process design phase. The reconstructor has to carefully analyze the available artifacts in order to gather the required data for creating the target views. This phase can be partly automated with tools and require the help of the systems experts in order to be conducted efficiently.

The third phase is the Result Interpretation. During this phase, the reconstructor interprets the results of the reconstruction with the stakeholders. One typical activity is to check the conformance of the reconstructed views against the hypothetical views or against certain architectural rules that have to be satisfied in the implementation (architecture conformance checking). The results of the reconstruction are a reliable source of information for an assessment of the architecture against future requirements or quality attributes. The architectural views also used for re-documenting a system that had an outdated or not-existed architectural documentation.

1. **Problem definition** The goal is to define the problems that a reconstructed architecture description should solve (the goals of the reconstruction). The main input are the discussions with the stakeholders and system experts. The outcome is a memorandum of the problem statement and an approved reconstruction plan.

2. **Concept determination** The objective is to recover the architectural concepts of the system and to define the architectural viewpoints that will be recovered. The inputs are the existing documentation, interviews of the stakeholders and system experts, catalog of reference viewpoints and reference architecture for product families. The output is the definition of architectural concepts: source and target viewpoints and their mappings; hypothetical views.

3. **Data gathering** The objective is to gather the data from the implementation and create the source view. This is a reverse engineering activity that can be highly automated with tools. The inputs are source code, build/make files, configuration files, existing documentation and databases. The outcome is the populated repository containing the source view.

4. **Knowledge inference** The objective is to derive the target architectural views from the source view. This activity can be conducted manually or semi-automatically with approaches based on relational algebra or other formalisms. The inputs are the source view defined in the previous activity and domain knowledge. The outcome is the set of target architectural views.

5. **Presentation** The goal is to present the architectural views in textual or graphical format. The views can be presented with textual reports,
hierarchical relational graphs, hyperlinked web documents, UML format and CASE tools.

6. Architecture Conformance Checking The goal is to check the conformance of the target views against the architectural rules. The inputs are the reconstructed views. The outcome is the list of violations. Techniques are: manual conformance check, automatic checking with the rules specified in binary relational algebra or other formalisms (like UML).

7. Architecture Assessment The goal is to check that certain quality attributes are satisfied in the software architecture. The input are the reconstructed views. The outcome is an assessment report. Techniques for architecture assessment are SAAM and ATAM.

8. Re-documentation The goal is to update the existing architecture documentation for the system. The input are the target views. The output is the up to date architecture documentation. The re-documentation can be conducted manually by the architects or the reconstructed views can be used as the official documentation.

Pattern: Analyzing System Differences

Intent: Quantify the difference between two forked, diverting systems to determine a cost-effective (co-)evolution strategy.

Problem

A given software system has been evolving for some time. At a certain point, a new, similar product is being built, reusing components of the old system. A copy of the reused components has been made and put under version control (see Figure 6.1). In the flow of the further evolution both systems are drifting apart from each other, due to small differences in requirements or different features being implemented. Both products are known to continue to coexist for a couple more years and will undergo both similar as well as specific changes.

Maintaining two similar subsystems is an expensive task. Implementing a common feature in both systems costs twice. A defect that is detected and repaired in one component may also impact the other system.

As maintainer, you suspect that there may exist a better strategy to cope with these two systems. Basically, there are three options:

- Keeping both system separate as they are now. Maintenance costs will be duplicated to the extent the systems are similar.
Identifying similar parts to be subsequently refactored into generalized, common libraries.

Working towards a unified system.

Making up the case depends upon the respective costs of (i) generalizing and extracting and refactoring common parts and (ii) maintaining the resulting artifacts. The lifetime of the systems, the internal quality of the system(s), sufficient knowledge by the maintainers, the presence of good software tests, the conformity of extensions are all influencing factors next to the raw difference.

This problem is difficult because a system can be quantified in numerous ways. We do not know what the most suitable comparison method is. Yet, solving this problem if feasible because you can apply multiple measurements and compare their results. Moreover, you can opt for the more lightweight methods that are easy to obtain and automate.

This pattern focuses on quantifying the difference between two systems – at the code level – as part of the input required to pick the optimal strategy.

Solution

We describe a range of techniques to assess the difference between two similar components. We gradually refine the analysis by assessing the difference at increasingly detailed levels of granularity. For every approach, we mention how it helps the analysis, and which questions can be answered.
Physical source structure

We start with studying the directory hierarchy of the subsystem as stored in the versioning system. Deviations indicate particular changes being made to only one of both subsystems.

The system’s structure can be studied by listing the directory structure with nesting indentation. The following UNIX command produces a directory tree such as shown in Listing 6.1.

```
ls -R | grep ':$' | perl -lape 's;/\.\?//;s(\.+\?\//\( | )g;s/\b/\|/'
```

Listing 6.1: Fragment of directory tree output

```
|_export
|_generated
 | |_lt
 | |_nt
 | |_scripts
 | |_original
 | |_testruby
 | |_sources
 | |_testContainers
 |_stubs
 |_subs
 | |_common
 | | |_lt
 | | |_nt
 | |_tests
 | | | |_Int
 | | | |_linemap
...
```

A deviating structure reveals a structural redesign reflected in the physical layout. Local changes such as additional directories or source files may point to functionality only present in one component. The name of unique directories are an indicator for certain features one of both components has in addition.

A similar, visually more appealing approach consists of creating a treemap visualization of both directory structures (see Figure 6.2).

Sets of source files

Studying sets of source files helps in the following ways: (1) Quantify the amount of new files in both hierarchies against the shared files from before the split, and (2) Identify new files for further investigation.
Figure 6.2: Aggregated directory structure of two related systems. Common files are white, files found only in one or the other system are grey and black respectively.

Figure 6.3: Comparison of source files
Figure 6.3 indicates which sets of files to identify. At the left and right, you have sets of files unique to one subsystem. In the middle, you have shared files. Furthermore, you can subdivide the subsystems in functional or structural partitions to become more fine grained results.

Practically, you obtain lists of file names as well as the shared and unique sets via the UNIX tools `find` and `comm`. We applied some abstractions to unify previously renamed but actually matching files (mainly neutralizing prefixes).

**Line Count**

At this point, we can quantify the amount of unique files and map them onto certain functionality. In case of a merger, we do not expect too much change in these files, as they will be controlled and invoked by the older, shared part.

The more complex and thus time-consuming work resides in shared files being different, because those files either contain internal changes or are being modified to hook in the new functionality of external files. Therefore, in a next stage we want to investigate which shared source files have been diverting.

A relatively cheap approach to compute source file similarity consists of comparing file sizes, line per line. Applying the UNIX command `wc -l` to the unified shared source list provides results rapidly.

![Figure 6.4: Comparison of shared source file sizes.](image)

Figure 6.4 shows, for a particular partition of our case, the difference in file sizes. From this visualization, you can deduce in which directions both
systems have been diverting. Outliers at the far left and right of the chart indicate locations containing large extensions in functionality. The amount of perfect matches together with the absolute changes in line count again gives a similarity indication.

Although cheap to compute, this line count approach also carries major limitations. This approach tends to underestimate the amount of difference. First, changes within one line will remain unnoticed. Secondly, the amount of change is not computed starting from a common baseline, but rather indicates a difference in growth. In the worst case changes of the same amount of lines zero each other out. Therefore, all numbers should be interpreted as minima.

A Comparison of Classes

In order not to get lost in the details of changes per line of code, we compare the two systems on a higher level of abstraction. We extract a model from the source code, containing classes, methods, interfaces (public, private, protected) and inheritance. We then determine which entities are common, and which are unique to each system.

```
Class Xdsl_DebugHandler

Same public Interface:
- Xdsl_DebugHandler(), ~Xdsl_DebugHandler(), help(), processLine()

Differences in public Interface:
- Members only in Asam: Members only in Isam:
  - addTinyDebugCommand() -

Same private Interface:
- cleanCurrentCommand(), helpCmd(), otherCmd(), printClassesList(), readAndCheckBoardNumber(), readAndCheckClassName(), readAndCheckInstance(), readAndCheckCommand(), passParameters(), factory_m, boardId_m, lastBoardId_m, instanceId_m, mjrCommand_m

Differences in private Interface:
- Members only in Asam: Members only in Isam:
  - debugCommandList -
```

Figure 6.5: Comparison of the set of methods in the class interfaces.

In Figure 6.5 we see an exemplary format in which differences at the class level could be reported. The results can be aggregated over all classes to come up with an overall estimate on structural changes.
Using duplication detection

Another approach consists of applying a duplication detection tool on both systems, where the source lines of subsystem A are compared with those of subsystem B. An example is shown in Figure 6.6.

Figure 6.6: A dotplot can show where the source code differs. The important feature is the middle diagonal which is formed by code that is common on both sides.

To detect major differences, we check the diagonal for irregularities. A horizontal gap in the diagonal means that code has been inserted in the subsystem represented horizontally; vertical gaps indicate additions in the vertically represented subsystem.

To make this technique work effectively, we need to make the middle diagonal as complete as possible, so that differences show up clearly as gaps in the line. This is achieved if we align corresponding source files in the same order on both sides.

Note that the dotplot visualization is useful mostly if the systems have not diverted too much, when differences are the exception rather than the norm. Otherwise, the display becomes too crowded and analysis is hindered by too much noise.
Tradeoffs

Pro’s:

- Lightweight comparison techniques that are easy to obtain and automated.
- For systems with a clear mapping between functionality and/or design and the physical layout (including naming), the directory and file comparison approach allows the reengineer to speculate about feature diversification early on.
- Line counts can serve as input for actual effort estimation.

Con’s:

- Using the source code as basis for comparison implicates both opportunities and threats. The source code is an opportunity, as it offers an up to date view of the system. The methods also entail threats in the form of the heuristics and assumptions used (e.g. the compared entities might have been renamed). This implicates that some careful manual post-processing is required to interpret the results.
- The visual approaches are suited for aggregation of large system data and human understanding, but do not serve quantitative data.
- These quantification techniques do not take into account how the systems are being used to determine whether parts can be merged or generalized, nor are properties such as non-functional constraints being considered.

6.2 Architectural Evolution

Pattern: Extracting code from an existing system to create a reusable component

Intent: This pattern aims at extracting the code from a software system in order to create a component that will be reused in other contexts or scenarios. The extracted code is wrapped as a reusable component and will be part of a component platform. By component we understand a coherent package of software that can be independently developed and delivered as a unit and that can be assembled with other components in order to build
larger systems. It can be seen as a large-grained object that is composed of several objects.

This approach will offer us several benefits impacting the whole life-cycle of the systems that integrate it. At design and implementation time the productivity will increase because developers do not write all the software from scratch. Furthermore, the amount of testing required for verification of products will be smaller because the component has already been tested. Last, the maintenance effort for the software that includes this component is also reduced.

**Problem**

Duplication of the code among several systems by uncontrolled copy& paste can result in an increased maintenance effort, as bug fixes or enhancements to the code must be replicated among the different systems; good reuse implies using the same unaltered component in many contexts.

**Solution**

The development of the reusable component involves several tasks. First, functionality to be reused will be defined into a candidate reusable component. Secondly, dependencies of the candidate component with the rest of the system will be analyzed. If the extraction of the candidate component is feasible then the code is extracted and wrapped as a generic component that provides customization capabilities. Finally, the reusable component is stored for further usage in new scenarios.

The initial situation, precondition, is that we have an existing system that implements a functionality that we want to include in more contexts or scenarios. The suggested steps to be followed when creating the reusable component are:

1. **Identify and scope the functionality** that is already available on the original system and that we want to integrate in other contexts. This step will involve an initial definition of the reusable component by means of identifying the interesting function/s, class/es, subsystem/s, data structures (any piece of SW) from the original system that we would like to reuse. As a result, the size of the reusable component is defined.

   *E.g. We have a feature that renders a chart in a dialog w.r.t the evolution of a SW metric, LOC, and we want to reuse this chart rendering feature in another application where drawing a chart is a requirement but based on a different set of data, temperature values. We identify that the method render LOC history is the functionality to be reused.*
2. **Analyze the dependencies of the candidate reusable component.**

Analyze the behavior of the component by retrieving information about its dependencies with other parts of the original system. This involves identifying which are the SW pieces that depend on the set of operations or data offered by the candidate component, plus what pieces of SW the candidate component depends on. The dependencies will have to be solved either by including the dependent software pieces as part of the reusable component or by stating them as dependencies that should be solved by the system where it will be integrated (required interfaces).

This analysis could be supported by reverse engineering the code to UML diagrams. Class, sequence or collaboration diagrams might highlight the dependencies of the candidate component with the rest of the system.

If the candidate reusable component is too coupled to the system architecture then the effort to isolate it as a reusable component might be bigger than the effort required to develop it from scratch (see trade-offs). Coupling of the candidate reusable component might be measured in order to evaluate the effort. Possible metrics to use are: coupling between object classes or fan in/fan out.

*The following class diagram shows the structural dependencies of the class LOC_Chart_Renderer with the rest of the system. The reusable component should include the container and the labels, as these are required to render the chart; but the reusable component should become independent of the chart source data (LOC_History class).*

3. **Commonality / Variability analysis.** In order to reuse a component in different contexts it must be generic enough to capture the commonality across those contexts and it must also offer mechanisms so it can be specialized as needed. The requirements that satisfy the candidate reusable component and the requirements that are expected from systems where it will be reused are the inputs for this step. The component has to be refactored in a way that generic functionality is offered by the component and plug-points or extension points are provided in order to tailor or customize the component for specific contexts.
The generic component should be able to render data provided. The system A should be able to render LOC. The system B should be able to render temperature data.

4. **Collect requirements needed by system B not present in the candidate reusable component from system A.** At this stage, it must be evaluated how far the candidate reusable component is from the functionality expected. If many adaptations are specific to system B then it might be better to develop the functionality from scratch (see trade-offs).

5. **Build the reusable component.** Once the commonalities and the possible variation points have been identified the candidate component source code must be refactored as a generic component that can be tailored to specific contexts. In the OO paradigm the generalization of the component and the further specialization can be achieved, for example, by providing a skeleton with the common behavior and then by specializing through interfaces that should be implemented on the specific context where the component is deployed (see example). Regarding technologies for implementing the component, it might be delivered as a middleware component (OSGi, CORBA) or by using other component technologies such as COM or JavaBeans.

The following class diagram shows the candidate reusable component extracted. It provides two interfaces: IRenderer, an interface that is provided and allows rendering the chart and IRenderable, an interface that is required by the reusable component. This interface must be implemented by the user and it gives access to the data that should be rendered. Regarding dependencies, the component will have internal dependencies, like the java swing components and external dependencies, like the interface IRenderable that has to be implemented.
6. **Store reusable component** on a repository to ease further usage on new scenarios. The reusable component must be tagged properly in order to ease the future use. Some interesting tags to be stored along with the component could be:

- name
- version
- formal description of functionality it provides,
- provided and required interfaces,
- any other dependency such as run-time environment,
- QoS aspects

**Hints**

UML tools can help on retrieving system architecture of the software piece to be reused.

**Tradeoffs**

Building the reusable component might require a huge effort depending on two factors: a) the candidate reusable component is strongly coupled to the architecture of the original system and it is very hard to isolate and b) functionality offered by the candidate reusable component is valuable but requires many adaptations due to the requirements of the new system. Depending on these two factors, the effort needed to reuse the component might be higher than the effort required to develop it from scratch. Therefore the effort is worthwhile only if the component is to be reused at least four or more times.
Related Work

Reverse engineering patterns used for retrieving the architecture of a system. Integration of a reusable component into a new system.

Pattern: Integration of a reusable component into a new system

*Intent:* This pattern aims at integrating a platform component into a new system, by assembling, adapting and wiring this component. This integration should not suppose any modification to the internals of the component since adaptations to the new system requirements should always be made through extension points or parameters provided by the component. This pattern complements the previous one.

Problem

A component platform is available in the organization with generic components which have been developed for further reuse on more than one context. The pattern focuses on integration of a generic reusable component into a new system.

Solution

Steps:

1. **Component selection and evaluation.** The functionality that the component provides must be checked in order to validate that the requirements are met. Documentation provided with the component must be analyzed in order to decide whether to carry on the integration or not. Several concerns should be addressed:
   - Does the component impose any constraints (pre-conditions) that can not be met by the context where it will be integrated?
   - Is there any QoS aspect, e.g. any timing constraint that the component is unable to provide?
   - Does the component provides enough customization capabilities in order to tailor its behaviour to the new scenario or application?

2. **Assemble component.** Component required interfaces must be provided by the system and the interfaces that the component provides are called by the system in order to achieve its goals. This step might involve adding additional glue code, adapters.

   *The following class diagrams show how the new reusable component is integrated in both systems. On both applications the interface*
IRenderable is implemented in order to provide the needed data to the generic chart renderer.

Figure 6.10: System A (left) and B (right) after integration of the reusable component.

3. **Test.** That the integration of the generic component has been successful on all contexts. It must be validated that the original system, the system that has provided the generic component, is working as before and that the component has been successfully reused on the other contexts where it has been integrated.

**Hints**

This is a skeleton of a possible implementation on java of the generic reusable component and of the adaptations needed to integrate it into a specific context.

```java
// Generic component
class Chart_Renderer_Component {
  IRenderable irenderable;
  public void Render () {
    Data = irenderable.getChartSourceData();
    Renderer.render(Data);
  }
}

interface IRenderable {
}
```
Tradeoffs

See Extracting code from an existing system to create a reusable component

Related Work

Extracting code from an existing system to create a reusable component.

Pattern: Phased Migration of a Software Stack

*Intent:* The purpose of the pattern is to gradually migrate a software stack from one technology to another. This is mainly relevant when this software stack consists of much code (say more than 100K lines), and/or when the migration goes over product releases (in other words, products are based on partially migrated software stacks). The main forces that make this difficult are major releases during the migration and functional additions in these releases.
Problem

A software stack is built with a certain technology. For some reasons, this software stack needs to be migrated to an other technology. This technology is present throughout a large part of the software stack (and not limited to a small subset like a platform library).

The most obvious, brute force, approach is the following:

1. Either stop all normal development on the software stack, or let normal development proceed on an other software archive.
2. Refactor all code in the software stack by replacing use of the old technology by use of the new technology. This may take a considerable amount of time (months/years), in which the code of the software stack will not even compile.
3. Make the migrated software stack compilable again. This can be quite tricky (depends on the kind of technology change).
4. Test the compiled software stack, and solve the problems (which have inevitably been introduced by refactoring).
5. If normal development was not stopped in step 1, its results need to be merged with the migrated software stack. Typically, the changes done in normal development also need to be refactored.

This is a big bang approach. The drawbacks are:

- a long period of time with a software stack that is not even compilable, let alone runnable. Harder problem solving. The testing in step 4 may reveal problems which have been introduced a long time ago (say at the start of the refactoring), which makes these problems harder to solve.
- all-or-nothing. If some product requires some part of the new technology in some part of the software stack, this product can only be made when the whole refactoring has been completed.
- running behind the facts: in case normal development proceeds, it will be done on basis of the old technology. This is grueling, since it is already known that it needs to be refactored into using the new technology. So the total effort increases because the normal development cannot be done right away with the new technology.

Types of technology for which this pattern is applicable:
- libraries
- computer languages (as long as the old language and the new language are interoperable in some way)
Solution

**Background.** We assume the software stack comprises of two layers, namely a Base layer and a Top layer. The Top layer is dependent on the Base layer, but not the other way around.

The software stack uses the old technology all over the place. In particular, it is used in the interfaces provided by the components in the Base layer. We summarize this by saying that the Base layer offers an old style set of interfaces. Obviously, the Top components use these interfaces.

The solution scales up to software stacks of any complexity, but the simple stack assumed here is easier to present.

**Recipe.** The following diagram shows the phased migration from the old technology to the new technology. We show five different points in time, with four refactoring steps. Each of the stacks has the old and/or new technology, the Base and the Top layer, with the style of interfaces they support. We present the usage of the technology, and usage of the Base layer by the UML lollipop notation.

![Figure 6.11: Phased Migration from Old to New Technology.](image)

Explanation of the stacks:

Stack 1. This is the starting point. Both layers work with the old technology.

Stack 2. The new technology is introduced and put next to the old technology. In theory, this is no effort.

Stack 3. Let the Base layer use the new technology and provide new style interfaces next to the old style interfaces. So at the end of this step, all Base component provide two styles of interfaces; both the old style and new style. The step to realize this stack involves considerable effort.

Stack 4. The Top layer uses only the new style interfaces. So the top layer has been refactored to use the new style libraries. In practice, the Top
layer often consists of several independent subsystems, say Top1 and Top2. Each of these can be refactored independently, in serial or in parallel fashion. Typically, this is the step costing most effort, but it can span several releases. For example, by refactoring Top1 for the first major release, and Top2 for the second major release.

Stack 5. The Base layer does not offer the old style interfaces anymore, and removes the dependency on the old technology, and the transition to the new technology is complete. The step to get her may cost considerable effort, but can also span releases.

Tradeoffs

Pro’s

This solution has the following advantages over the brute force solution:

- The software stack stays compilable all the time.
- The software stack says functionally working all the time. This makes it possible to release products at any point in time during the refactoring process (although there may be performance considerations).
- The software stack can be tested at all times (manual and/or regression tests).
- Easier problem solving. The average time between changing code and hitting it in a test is much smaller here.
- Earlier leveraging of new technology. From stack 3 onwards, components in the top layer can use the new technology.
- Right-in-one-time. Normal development can write its code using the new technology quite soon; Base development from stack 2 onwards, and Top development from stack 3 onwards.

Con’s

- Extra effort for components to support two types of interfaces.
- Footprint of releases when both technologies are present.

Constraints

The old technology and the new technology should be able to live side-by-side on one system, and even be interoperable to a certain extent.
Variations

- The old technology is not removed from the software stack, so the refactoring ends at stack 4 in the diagram above. It is not always economical to make the last step and remove the old technology.
- The software stack has more than 2 layers. For example, there are 3 layers, called TopUpper, TopLower and Base, where TopUpper depends on TopLower, which depends on Base. Then this pattern can be applied recursively, as follows. First follow the recipe above, towards stack 3, seeing TopUpper and TopLower together as Top. Then the migration from stack 3 to stack 4 is realized by first making TopLower provide both old style and new style interfaces; then migrating TopUpper so that it only uses the new style interface; and then cleaning up TopLower so that it only uses the new style interfaces. This yields stack 4 of the diagram above.

Known Uses

It is quite common to have such a software stack, and since technologies change relatively frequent in software industry, migrations are common too. Known uses include:

- migrating from one library to another
- migrating from one computer language to another (as long as the old language and the new language are interoperable in some way)
- migrating from one internal set of classes/interfaces to another
- migrating an Operating System (which can be seen as software stack)

Example

The example worked out below is a software stack of 1.5 million lines of code, migrated from Java to .NET.

**Background.** The software stack is used for building medical systems like X-ray scanners, MRI-scanner, CT-scanner, UltraSound machines and so on. In this simplified presentation, the stack comprises of two layers, namely a Base layer and a Top layer. The Top layer is dependent on the Base layer, but not the other way around.

In this case, the old technology is Java and the Java libraries. The new technology is the Microsoft .NET framework. The special thing about this, is that the .NET framework includes the Java language (or actually a variant called J#), and the Java libraries, so that existing Java-based products can be run without too much effort on the .NET framework. Note, however that these Java libraries are not an integral part of the .NET framework. Instead, they are served as a side-dish. For example, the .NET framework has a JavaList class for managing list of elements, and next to that there is a
DotNetList class which is conceptually the same but has a slightly different interface.

Now this software stack uses lists all over the place. In particular, lists are used in many places in the interfaces provided by the components in the Base layer. We summarize this by saying that the Base layer offers a Java-style set of interfaces. Obviously, the Top components use these interfaces.

**Recipe.** The following diagram shows how the migration from Java to .NET was done. We show five different points in time, with four refactoring steps. Each of the stacks has the Java or .NET framework, the Base and the Top layer, with the style of interfaces they support.

![Figure 6.12: Phased Migration from Java to .NET Technology.](image)

Explanation of the stacks:

Stack 1. This is the starting point. For example, Base components work with JavaLists.

Stack 2. The Java platform has been replaced by the .NET platform. .NET supports Java-compatible libraries, and we have JavaList next to DotNetList. In theory, this is no effort.

Stack 3. Let the Base layer use the proper .NET interfaces and provide .NET-style interfaces next to the Java-style interface. So at the end of this step, all Base component provide two styles of interfaces; both the Java-style and the .NET style. Often, wrappers play a role in effectively realizing this. But still, the step to realize this stack involves considerable effort.

Stack 4. The Top layer uses only the .NET style interfaces. So the top layer has been refactored to use the .NET style libraries. In practice, the Top layer consisted of several independent subsystems, which each could be refactored independently. So it also costs considerable effort to realize this, but it can span several releases.
Stack 5. The Base layer does not offer the .NET style interface anymore, and removes the dependency on the Java libraries; the transition to the new technology is complete. The step to get here may cost considerable effort, but can also span releases.
Appendix A

State Machines and their Representations

A finite state machine (FSM) or finite automaton is a model of behavior composed of states, transitions and actions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that needs to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment. A transition is triggered by an input action or event. A sophisticated state machine may execute an action when a specific state is entered, and another action when a state is left. A simple state machine associates an action with each transition between a source and a target state.

To determine which action code to execute next, a state machine works on two parameters, the current state and the input. The normal view of a state machine as a certain state acting on the input leads to the idiom of first selecting the state and then take the input event to decide on the action to be executed.

State Machine Representations

State machines have two common representations: the state transition table and the state diagram.

State Transition Table

A state transition table is a way to present the transitions of a state machine in a clearly arranged manner. A commonly used layout shows the states on the vertical axis, and the input events on the horizontal axis. Each cell contains the state into which the machine transitions, given the state on the
vertical and the input on the vertical axis (an empty cell means that a state does not handle a given input).

<table>
<thead>
<tr>
<th>Events</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S1, S2</td>
</tr>
<tr>
<td>S2</td>
<td>S2, S1</td>
</tr>
</tbody>
</table>

Table A.1: State Transition Table. State $S_1$ transitions to $S_2$ if input 0 is recieved.

A table driven approach of designing a state machine does a good job in giving an oversight of all possible states, inputs, and transitions. It’s however difficult to include into the display the actions that have to be taken for each transition.

**State Diagram**

A graphic way to represent state machines is a diagram made out of bubbles and arrows. Each circle represents a state, and arrows represent transitions from state to state. The transitions are annotated with the input by which they are triggered.

Figure A.1: State Machine Diagram for the state machine of Table A.1. The initial state is $S_1$ as indicated by the double circle.

State diagrams are a good way to design a state machine, or to reverse engineer a state machine from the source code, because they can be drawn freely on a piece of paper. A new transition that is discovered during the reverse engineering phase can be added more easily to a free–form drawing that to the rigid structure of a table.
Appendix B

Script to extract Nesting Depth

The PERL script presented in this section extracts the nesting depth of C/C++/JAVA functions. It reads a list of file names from a text file and then examines the contents of each of these files, printing the maximum nesting depth per function.

Listing B.1: Script to extract maximal nesting depth from C++ functions.
$inExtern = 0;
next;
}

#-- marks end of a function block when current
# nesting depth is 1 (because externs and
# namespaces are filtered out)
# still, if could be a struct, but in that case
# there is no current function name, so don't print
if ($curND==1 & $curFuncName ne "") {
    print "$curFuncMaxND\t$curFuncName\t$filename\n";
    $curND--;
}

} close SOURCEFILE;
}
close INFILE;
Appendix C

Script to Extract Control Flow Skeleton

The PERL script presented in this section extracts the control flow skeleton of a C/C++/JAVA function. It reads the source code of a single function from a file and then removes everything that does not pertain to the control flow of the function.

Listing C.1: Script to extract control flow skeleton from C/C++/JAVA functions.

```
#!/usr/bin/perl -w -d
#
# The script extracts a control flow skeleton from a C/C++/Java block, e.g.
# a function body. Everything that does not belong to the control flow or the
# conditions which guide it is removed. Line--Numbers are prepended to the
# control flow to let the user see the length of the omitted code. The skeleton
# is indented to help the reader.
#
# The script does not understand function boundaries, i.e. it should be
# fed with the body of one function at each time. By giving an offset, the
# script extracts the outermost block found starting on the given line or
# after it. By giving the number of the line where a function start, the
# script will extract the control of the entire function body. Giving
# a maximal nesting depth, the script will suppress all deeper nested code.
# Giving a number of blocks, the script will extract this number of consecutive
# blocks.
#
# Invocation:
#
# controlFlowSkeleton.pl [<filename>] [-offs <number>]
# [-max <number>]
# [-blocks <number>]
#
# If no file name is given, reads input from STDIN.
#
# Author: Matthias Rieger, SERIOUS project, University of Antwerp/Alcatel-Lucent
# Created: March 31, 2007
# History: + April 1, 2007
#   - code is nicely formatted
#   + April 2, 2007
#     - blocks are extracted starting from a given offset
#   + April 3, 2007
#     - adding a constraint for maximal indentation level
#   + April 10, 2007
#     - correct indentation for arguments of function calls when no
#       parenthesis or braces occur on the line
#     - insertion of artificial braces around if/else/for/while--blocks
#       which consists of a single statement the programmer has not
```

113
puts braces around. This means that such code
if (!onlyUpdateHw)
bridgedPortInfoP->pVID = vlanId;
return true;

is now condensed to instead of
if (!onlyUpdateHw)
{
    if (!onlyUpdateHw)
        return true
}

############################################################################
# $Id: controlFlowSkeleton.pl,v 1.3 2007/04/02 15:52:25 rieger Exp $
#BeginDefaults and Constants
# the dummy file name indicating input comes from STDIN
$stdinfn = "-";
# formatting string for one indentation level
Sind = " ";
#EndDefaults and Constants
#End RegExes
Begin RegExes
# matches an expression in balanced parentheses
$expression = qr\{
    (?-m) (?-P noBacktrack)
    (?{ $expression })
    (?{$expression})
}\):
# matches text till the end of a statement or the line
$toEndOfStatement = qr(\.*(\?=(\?|\n)))*$:
#EndRegExes

sub readInput {
    my ($filename) = @ARGV;
    my $IN = 
    if ($filename eq $stdinfn)
        ($IN = STDIN);
    else {
        open IN, "$filename" or die "Could not open source file " $filename", abort:"
        $IN = 
    }
# undef the record separator to get the entire file contents at once
my $source = do { local $/; <$IN> }
    close IN if ($filename ne $stdinfn);
    return $source
}

sub removeCPPComments {
    my ($sourcecode) = @ARGV;
    # removing C++ comments, from
    # http://perl.org/perlfaq8.html#
    # How do I use a regular-expression-to-strip-C-style-comments-from-a-file?
    $sourcecode =~ s/\*(\*\*+|\*\*)|/\1/ ge; # defined $2
    return $sourcecode
}

sub replaceStringContents {
    my ($literalstring) = @ARGV;
    # replaceStringContents($2) : retainLineBreaks($2) if $2
    return $literalstring
}

sub defuseDelimitersAndKeywords {
    return $literalstring
}

sub replaceStringContents {
    my ($literalstring) = @ARGV;
    return $literalstring
}

# removes the delimiters which we use to identify blocks and expressions
# from strings. Also changes keywords so that they do no longer confuse our parser
sub defuseDelimitersAndKeywords {
    my($str) = @ARGV;
    114
# removes everything from the string, except for linebreaks
sub retainLineBreaks {
  my ($text) = @_;  
  $text = qr/\[\^\n\]+/g;  
  return $text;  
}

# add braces to if/for/while blocks which consist of a single statement
# that the programmer did not care to brace.
sub insertBraces {
  my ($code) = @_;  
  $code = qr/( ( ? : if | while | for ) \$expression )\{\} /s $1 { $2 }/ogs;
  # we have to brace loose 'else'--branches in a second regex.
  # because we want to be sure that something like
  #
  #    if ( cond ) do it ( )
  #
  # else {
  #
  #    if ( cond ) do it ( )
  #
  #}
  #
  # has already been matched with the 'if' in the first regex
  # and we do not get this faulty thing:
  #
  #
  #    else {
  #
  #}
  #
  # has already been matched with the 'else' in the first regex
  # and we do not get this faulty thing:
  #
  #
  #    else {
  #
  #}
  $code = qr/\{ (. )\{\}+? ( ? : ; | \} ) /$1 { $2 }/gs;
  return $code;
}

# extracts the block which starts after the given offset.
# The source code is expected to not contain any comments or literal strings.
# any more.
sub extractBlock {
  my ($source, $offset, $consecutiveBlocks) = @_;  
  my @lines = split /\n/, $source;  
  my ( $body, $lineCount, @blockStarts ) = (' ', 0);  
  foreach my $line (split /\n/, $source) {
    $lineCount++;  
    next if ($lineCount < $offset);  
    while ($line =~ m/$expression)/$1/ {  
      my ($before, $match, $after) = ($' , $& , $' );  
      $body .= "$before$match" ;  
      if ($match eq '\{') {  
        push @blockStarts , $lineCount ;  
      } elsif ( # $match is '}' ) {  
        if ( scalar @blockStarts ) {  
          pop @blockStarts ;  
          if ( $#blockStarts < 0 ) {  
            $consecutiveBlocks--;  
            return $body if ( $consecutiveBlocks < 1 ) ;  
          }  
          $line = $after ;  
          $body .= "$line\n" ;  
        }  
        print STDERR "No closing brace found for block starting on line \$line\n," ;  
        $blockStarts[-1] ;  
        return $body  
      }  
    }  
  }  
  return $body ;
}

# extracts the control flow entities from the source code.
# No comments or literal strings are expected to be left in the code.
sub controlFlowSkeleton {
  my ($source) = @_;  
  my ($condensed) = ( '.' );  
  while ($source =~ m/\{\{|\}\} \n\{|\} |\n\nswitch\$s\$expression |\n\ncase\b.*? |\n\ndefault\b\$s/:}  

\break\b
\bcontinue\b
| else\s+if\s+$expression
| (?<\#)\belse\b
| if\s+$expression
| while\s+$expression
| do\b
| for\s+$expression
| goto$toEndOfStatement
| return$toEndOfStatement
| bexit$toEndOfStatement

{ my ( $before , $match , $after ) = ( $' , $&,$' ) ;
$condensed . = retainLineBreaks($before) ;
# leave the embellishment of braces with white space intact ,
# to let them stand out visually
if ( $match =˜ m/\{|\}/ && $before = ˜ m/ +$ / )
{ $condensed . = $& ;
}$condensed . = $match ;
if ( $match =˜ m/\{|\}/ && $after = ˜ m/ ˆ + / )
{ $condensed . = $& ;
}$source = $after ;
return $condensed }

# removes all empty lines , indents the code , and prepends the line numbers.
# $offset is the number of the first line in the source
# $maxShownNDepth is the level of nesting that will still be shown
sub formatSourcecode
{
my ( $source , $offset , $maxShownNDepth ) = @ ;
$offset = $offset ? $offset − 1 : 0 ;
my ( $linecount , $formatted ) = ( $offset , ' ' ) ;
my $maxNestingDepth = 0 ;
local ( $nestingDepth , @parenStack ) = ( 0 ) ;
foreach my $line ( split /\n/ , $source ) {
$linecount ++ ;

# indent the code
my $nestingDepthDelta = 0 ;
my $indentation = undef ;
while ($line =˜ s/\s+ / / ;
my ( $before , $match ) = ( $' , $& ) ;
unless ( defined $indentation )
{ if ( $match eq ' ) && $before = ˜ m/ +$ / )
{ $nestingDepth −− ; next }
$indentation = computeIndentation() ;
}$nestingDepthDelta−−
if ( $match eq ' ) ;
$nestingDepthDelta++
if ( $match eq ' ) ;
push @parenStack , length ( $indentation ) + length ( $before )
if ( $match eq ' ) ;
pop @parenStack , if ( $match eq ' ) ;

if ( $nestingDepth < $nestingDepth+$nestingDepthDelta )
{ $maxNestingDepth = $nestingDepth+$nestingDepthDelta ;
$nestingDepthDelta = $nestingDepth+$nestingDepthDelta ;
}$maxNestingDepth = $nestingDepth+$nestingDepthDelta ;
return ( $formatted , $maxNestingDepth )
}
sub computeIndentation
{
return ( scalar @parenStack )
? ' \x' x ( @parenStack[−1]+1 )
: $ind x $nestingDepth ;
}
# summoned : 116
# Main Program

($inputfile, $offset, $maxNestingDepth, $consecutiveBlocks) = (undef, -1, 0, 1);
while (@ARGV) {
    $_ = shift @ARGV;
    if (/^-offs$/) { $offset = shift @ARGV; next }
    if (/^-max$/) { $maxNestingDepth = shift @ARGV; next }
    if (/^-blocks$/) { $consecutiveBlocks = shift @ARGV; next }
    if ($inputfile) { die "Unknown command line option '$_', abort\n" }
    $inputfile = $_
}

$source = removeCPPComments(readInput($inputfile));
$source = insertBraces($source);
if ($offset > 0) ($source = extractBlock($source, $offset, $consecutiveBlocks));
my $skeleton = controlFlowSkeleton($source);
($skeleton, $maxDepth) = formatSourcecode($skeleton, $offset < 0? 0 : $offset, $maxNestingDepth == 0?10000:$maxNestingDepth);
if ($inputfile ne $stdinfn) {
    my @msg = ();
    push @msg, "code read starting from line $offset" if ($offset > 0);
    push @msg, "maximal nesting depth is $maxDepth";
    push @msg, "maximal nesting depth shown is $maxNestingDepth" if ($maxNestingDepth > 0 &
        $maxNestingDepth < $maxDepth);
    print join(", " @msg), "\n";
}
print $skeleton
Bibliography


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Mass., 1995.


