Abstract

The common object-oriented languages suffer from the “tyranny of dominant decomposition”. As such, they are not able to cleanly modularize every concern. This means that crosscutting concerns arise, which are typically scattered over the software system, and tangled with other concerns. The principle of AOP is a proposed solution to the problem of crosscutting concerns. It strives to refactor the crosscutting concerns into aspects. In the light of AOP, there is a demand for aspect mining techniques that automatically mine a legacy system for all crosscutting concerns potentially to be turned into aspects. Such a legacy system is characteristically of large scale, complex and poorly documented; properties that make it practically infeasible to manually mine for crosscutting concerns.

In this dissertation, we investigate the extent to which cluster analysis can be used to perform aspect mining. Cluster analysis is an unsupervised learning technique. It is concerned with grouping objects that are similar, according to a user-provided distance measure. We propose five aspect mining techniques that use cluster analysis. The proposed techniques differ in the distance measures they use to group source code methods in the legacy system. We further compare these techniques to one another, and to aspect mining techniques authored by other researchers. This comparison is performed in terms of both aspect mining results and analysis properties.

Acknowledgements

I would have never finished this dissertation without the help of a lot of people. Therefore I would like to express my gratitude towards:

Prof. Dr. Theo D'Hondt for promoting this thesis.

Andy Kellens for being an excellent advisor. Not only did he come up with the subject of this dissertation, he also guided me throughout the whole process of creating this thesis, from the implementation to the writing stages. He continuously provided suggestions, ideas and feedback, in addition to proof-reading everything. Without his help, this dissertation would never have seen the light of day.

Prof. Dr. Kim Mens for his guidance in the early stages of this thesis.

The researchers at the Programming Technology Lab for their invaluable feedback and help throughout the past year.

The Vrije Universiteit Brussel and Departement Informatica for an excellent education.

Last but not least my parents, girlfriend, family and friends for their continuous support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Fundamental problem

Many existing programming languages, including object-oriented, procedural and functional languages, can be classified as generalized-procedure (GP) languages. Their means of abstraction and composition are founded on forms of generalized procedure (object, method, procedure). After many years of the practical use of these languages, they have shown a particular weakness: they are not able to cleanly encapsulate each concern in a generalized procedure of its own. (A concern should be understood to be any piece of interest or focus in a software system.) As a consequence, crosscutting concerns arise [KLM$^+$97]. A crosscutting concern is characterized by:

1. scattering, meaning that the concern is spread out over different generalized procedures in the software system, instead of being localized in one place in the source code;
2. tangling, which is the term used to indicate that a generalized procedure encapsulates more than one concern.

"The holy grail of software engineering" as it is sometimes called, separation of concerns is essential in the software development process. The aim of this principle is to break the software system into distinct modules that overlap in functionality as little as possible. The desired advantages are to [BK04]:

- facilitate software evolution;
- improve program comprehensibility;
- improve program adaptability;
- increase program maintainability;
- allow reuse of code.

It should be clear that crosscutting concerns violate the separation of concerns, and that they are thus of note to any programmer and programming language engineer.

1.1.1 A Solution: AOP

Aspect-Oriented Programming (AOP) [KLM$^+$97] is a relatively new programming paradigm. It strives to cleanly encapsulate the crosscutting concerns into aspects, which are a new kind of module. Novel languages for writing aspects were invented (e.g. AspectJ [KHH$^+$01]) that are complementary to a "deficient" GP language (Java in the case of AspectJ). The complemented language is called the base language, while the software system written in that language is called the base system. At run- or compile-time, the aspects are woven into the base system.

1.2 Aspect Mining

Commonly, AOP languages are used to build software from the ground up, or to extend an existing software system with new functionality. However, another scenario is the migration of an existing software system to being an AOP solution. This means that the crosscutting concerns in the system are refactored into aspects. In research, this migration is divided into two consecutive steps [KM05]:

1. **An aspect mining step** Aspect mining is the activity of discovering those crosscutting concerns that potentially could be turned into aspects, from the source code and/or run-time behaviour of a given software system.
2. **An aspect refactoring step** Aspect refactoring is the activity of actually transforming the identified aspect candidates into real aspects in the source code.

A legacy system is characteristically poorly documented, large and complex. Thus, in the case of a legacy system, it is particularly hard (if not unfeasible in practice) to carry out the aspect mining step by hand. That is why the main focus of aspect mining research lays on tools that aide the developer in mining for aspects. There exist dedicated browsers that assist the developer in tracing the scattered path of a crosscutting concern, using some form of intelligence or a query language. Additionally, there are those aspect mining techniques that aim to automatically uncover all aspect candidates in the legacy system. However, at the current time, a fully-automatic aspect mining technique seems unlikely. Our research is concerned with the second category of aspect mining approaches. More specifically: we investigate the extent to which cluster analysis can be used to automatically mine for aspects.

1.3 Cluster Analysis and Aspect Mining

Cluster analysis is often said to be a machine learning technique [Mit97]. The analysis requires two inputs: a set of objects, and a (binary) distance measure between these objects. The distance measure relates a degree of dissimilarity between the objects. Thus, indirectly, it also provides a notion of similarity. Cluster analysis aims to group objects into clusters, i.e. groups of objects, such as to maximize:

- cluster cohesion, which is the degree to which objects in the same cluster are similar to one another;
- cluster separation, which is the degree to which objects in different clusters are dissimilar to one another.

Aspect mining and cluster analysis have intersected twice before. The aspect mining technique of Shepherd et al. [SP05a] uses methods in the legacy system as the objects in cluster analysis. The deployed distance measure considers two methods to be similar, only if they have similar names. This technique effectively groups similarly named methods into the same cluster. The underlying hypothesis is that crosscutting concerns are often implemented through a rigorous use of naming conventions, and that they can thus be detected using lexical analysis techniques as this one.

The other aspect mining technique groups methods as well [LHH05], but in computing the distance between two methods, it analyzes the method bodies in which both the methods are called. The methods in a resulting cluster tend to be called together in method bodies. The idea is that a recurring pattern of method calls can hint at a crosscutting concern.

1.4 The Aim of the Dissertation

The purpose of this dissertation is to investigate the extent to which cluster analysis can be used to perform aspect mining. We consider the suitability of different cluster algorithms that implement cluster analysis. This assessment is made through implementation and experimentation work, as well as through (literature) study. As our main work, we propose several aspect mining techniques that use cluster analysis to automatically mine for all aspect candidates in the legacy system. (Although a manual post-processing step is still at order in the use of each technique.) Finally, we evaluate the proposed aspect mining techniques.

The aspect mining techniques we propose differ in the source code properties they analyze. Tree Clustering is a more structural analysis, because it considers parse trees of methods. Call Clustering is the most semantical, because of how it mines for recurring patterns of method calls. Ngram and Name Clustering are largely lexical analysis techniques, as they process strings extracted from method names and bodies, respectively. Type Clustering specifically analyzes typing information.

We take different approaches to evaluating our aspect mining techniques. On one hand, we discuss the individual mining results of the techniques, and compare the techniques through their individual results. We additionally compare our techniques to work by other researchers, in terms of the characteristics of the analyses. In another leg of the evaluation, we compare the mining results of our techniques with those of techniques by other researchers. This is done through a paper by Ceccato et al. [CMM$^+$05]. In this paper, the results of three aspect mining techniques are compared. We extend the comparison with our own results.

The benchmark used in our experiments (and those of Ceccato et al.) is JHotDraw [JHo], a relatively large-scale Java system. We present an extensive list of crosscutting concerns in JHotDraw, composed through literature study and manual investigation of the system. It is in the context of this list that we perform much of the evaluation work described above.

1.5 Outline of the Dissertation

In Chapter 2, we discuss the problem of crosscutting concerns, and how the AOP principle strives to leverage this problem. In Chapter 3, the edit distance between strings and trees is explained, as are approaches that compute this distance. (It is the edit distance between trees that is used in the Tree Clustering technique we propose.) Chapters 4 and 5 survey cluster analysis and aspect mining, respectively. Chapter 6 discusses JHotDraw, the benchmark used in our experiments. In Chapter 7, we present our aspect mining techniques, as well as our approach to cluster analysis. This chapter further includes a comparison of our work to existing aspect mining techniques, in terms of the properties of the analyses. All our experimentation work is outlined in Chapter 8, after which our aspect mining results are discussed and compared in Chapter 9. We conclude with Chapter 10, in which we discuss what we have learned through the work of this dissertation.

Chapter 2

Aspect-Oriented Programming

This chapter details Aspect-Oriented Programming (AOP), a relatively new principle that tries to leverage the problem of crosscutting concerns. Such concerns are poorly-modularized and therefore detrimental to the overall manageability of the software system. We first discuss this fundamental problem, before presenting the general AOP principle. As an example, we conclude with a quick introduction to AspectJ, the best-known realization of AOP.

2.1 Crosscutting Concerns

A key feature of programming languages is the manner in which they allow the developer to: (1) build abstractions, and (2) further combine the abstractions to compose new abstractions. Most traditional languages are generalized-procedure (GP) languages. Their means of abstraction and composition are based on forms of generalized procedure (e.g. class, method, procedure). Also, GP languages know one dominant dimension of decomposition (it is for instance the class hierarchy in object-orientation). In the context of GP languages, a concern to be implemented is [KLM$^+$97]:

- a **component**, if it can be cleanly modularized in a generalized procedure;
- an **aspect**, if it can not be encapsulated as such.

An aspect typically leads to two malicious symptoms in code. The first is that of *scattering.* This means that the aspect is implemented over several GP's, rather than being implemented in one locality in the source code. The second is that of *tangling.* A GP is tangled if it encapsulates more than one concern. In general, it is said that an aspect *crosscuts* the base composition – or, that it is a *crosscutting concern.*

Logging and error handling are prototypical crosscutting concerns. Let us give an example. In an object-oriented system, the same logging statement might be present in many methods. Furthermore, the primary roles of these methods might be entirely different concerns than logging. In that case, the logging concern is scattered over different methods, which are in turn tangled. Alternately, a special-purpose logging method might be implemented in several classes that are otherwise unrelated to logging. This puts the crosscutting at a different level of granularity than in the previous example.

The notion of "separation of concerns" is fundamental in the software development process. It signifies the goal to break the software system into distinct modules, in a manner that those modules overlap in functionality as little as possible. This idea is oft-compared to the Roman principle of "divide and conquer". It is more efficient to solve a general problem, if that problem is broken into subproblems, and each subproblem is then considered separately. In terms of a given software system, adhering to the separation of concerns is beneficial to [BK04]:

- software evolution, which is facilitated;
- the degree to which code can be reused;
- (system) comprehensibility;
- adaptability;
- maintainability.

The last three properties are also called the "ilities" of the software system. It should be clear that crosscutting concerns violate the separation of concerns. They hence pose a significant problem. This is especially true because crosscutting concerns are practically unavoidable in software systems of considerate scale. As a general statement, it can be said that GP languages suffer from "the tyranny of dominant decomposition" [TOHS99]. Especially in the context of this dissertation, it is important to note that the traditional object-oriented languages commonly suffer from this tyranny.

2.2 Aspect-Oriented Programming

The concept of Aspect-Oriented Programming (AOP) [KLM+97] is a proposed solution to the problem of crosscutting concerns. AOP languages strive to cleanly encapsulate the crosscutting concerns into *aspects*, which are a new kind of module. (From this point on in the text, we take "aspect" to denote the module in AOP – rather than taking "aspect" to be a synonym to "crosscutting concern", as in the previous section). To this end, an AOP language offers constructs to specify aspects. The aspects are complementary to a *base system*, which is written in the *base language*. At run-time or compile-time, a *weaver* integrates the aspects into the base system. An AOP language utilizes a *join point model* (JPM). Commonly, a JPM is specified as:

1. The places, or run-time events, in the base system where the aspects can be applied. These places (events), are called the *join points*.
2. The ways in which a condition can be specified through the quantification of join points. Such a condition is called a *pointcut*.
3. The ways in which behaviour can be specified as resulting from the event of a particular pointcut being met. Behaviour that is attached to a pointcut is called *advice*.

An AOP programmer writes aspects by specifying pointcuts and advice. Given a particular base system, the set of join points is determined by the JPM (thus AOP language). As an example, a basic AOP language could utilize a JPM where: (1) the join points are method calls as they occur at run-time, (2) pointcuts are specified as disjunctions of join points, (3) advice is specified as code, written in the base language, that must be executed when a particular pointcut is met. Using this AOP language, one could for example create an aspect that logs a message to the stream, each time one of selected methods is called. In the case of this aspect, the advice is the code that performs the logging, while the selection of methods is determined in the pointcut to which the advice is attached.

AOP is sometimes characterized as "quantification + obliviousness" [FF00]. Quantification is the manner in which part of the base system, or part of its behaviour, can be captured through join points and pointcuts. The above example of a basic AOP language is also basic in its quantification. There exist more intricate ways in which a pointcut can logically relate to a wide array of join points. For example, a set of join points can be specified through a single wildcard or string pattern. Pointcuts can commonly be composed out of other pointcuts, using different logical connectives.

Quantification is often considered to be dynamic or static. Dynamic quantification captures run-time behaviour of the base system, so that advice can be attached to the detection of this behaviour. Static quantification captures a subset of the base system in its static form, so that this subset can be modified or augmented (e.g. with new classes or methods).

Obliviousness relates how the base system should not contain a single trace of the aspects that apply to it. This means that the base system should not be annotated, or contain statements, such that there exists a physical link between the base system and its aspects. This principle benefits the separation of concerns.

2.2.1 Some Interesting AOP Approaches

Stateful Aspects In the next section, we present AspectJ, the most popular AOP language. Another interesting AOP approach is that of stateful aspects. An AOP language that implements this approach (e.g. EAOP[DFS04],

```
aspect Logging {

    // the pointcut
    pointcut move():
        call (void Figure.move(int))

    // the advice
    before (): move() {
        GlobalLogStream.println("about to move");
    }
}
```

Figure 2.1: A Logging aspect, illustrative of pointcuts and advice.

JAsCo[VSCF05]) has a JPM where a pointcut specifies a protocol.[1] In this context, a protocol is a pattern of join points (e.g. method calls) that could occur at run-time. Advice is specified as having to be executed at the run-time detection of a particular protocol. The idea behind stateful aspects is that a pointcut (protocol) can be specified using a specialized, declarative language. In an AOP language without a stateful aspects mechanism, the programmer typically needs to manually implement a state machine to detect protocols. This involves mixing advice code with control code that determines the applicability of an aspect, which is undesirable.

Adaptive Programming In an AOP language that implements the adaptive programming approach (e.g. JAsCo [VSV+05], DJ [OL01]), an aspect can specify a traversal through a data structure. Advice can be executed at the visiting of the objects in the structure. The idea behind adaptive programming is that a traversal can be specified using a specialized, declarative language. The common example in this context is that of the Visitor design pattern (Section 6.4.5 for an explanation of this design pattern). Without AOP, the traversal of the visitor must be hardcoded into the data structure itself. In AOP without adaptive programming, the traversal must still be completely specified – for lack of a specialized, declarative language.

2.2.2 AspectJ

AspectJ [KHH+01] is an AOP language. It allows to specify aspects that are complementary to a software system written in Java. AspectJ models aspects as modules that can contain advice, pointcuts and *inter-type declarations*.

Figure 2.2 illustrates an aspect in AspectJ. It is a logging aspect, for printing information to a stream whenever the `Figure.move` method is called. The aspect defines a `move` pointcut to capture the execution points at which the logging must be performed. The `call` construct inside the `move` pointcut is a pointcut itself; layers of abstraction can be built through composing pointcuts

[1]This is a somewhat simplified representation of the JPM that such a language implements.

```
aspect Logging {

    public interface Logger {}

    declare parents: Figure implements Logger;

    public LogStream Figure.logStream;

    public void Figure.log() { logStream.println("about␣to␣move"); }

     ...
}
```

Figure 2.2: A Logging aspect, illustrative of inter-type declarations.

out of pointcuts. The `call` pointcut is primitive, in that it is provided by AspectJ. It must be given a parameter value to capture the method call(s) of interest. In this case, the value is simply the name of the method. The aspect also defines advice. It is specified using the `before` keyword. This means that the advice is to be executed *before* the execution process designated by the pointcut, i.e. before the `Figure.move` method is called.

In AspectJ, pointcuts and advice provide a dynamic mechanism, which is dependent on run-time execution points. Conversely, inter-type declarations augment the static structure of the base system. Figure 2.2 illustrates what the inter-type mechanism can do. It is another Logging aspect, which imposes onto the Figure class:

- the Logger interface (Figure now implements Logger – also note how the aspect declares this interface itself);
- an instance variable `logStream`;
- a `log` method with a concrete implementation that writes to the stream.

Naturally, AspectJ allows to define more intricate aspects than the examples here presented. Pointcuts can be composed using logical connectives on pointcuts (e.g. ||, &&, !). Complementary to `before` advice, `after` and `around` advice exist (the latter wraps around an execution point). Run-time information can be retrieved from the base system, for use in an aspect. Naming patterns can be specified rather than exact names. For example,

```
call(void Figure.set*(..))
```

captures all the accessor methods in the Figure class, regardless of which variables the methods access, or what formal parameters they have.

This list of features is not exhaustive. For further reference regarding AspectJ, we point to [KHH+01].

2.3 Summary

We have explained what crosscutting concerns are, how they are detrimental to the overall manageability of the software system, and how they suffer from tangling and scattering. Our techniques (Chapter 7) belong to the research field of aspect mining (Chapter 5), which is concerned with detecting crosscutting concerns in a given software system. We have also discussed AOP, a proposed solution to the problem of crosscutting. We have characterized AOP through the notion of join point models, and through "quantification + obliviousness". The general field of AOP encompasses a multitude of concrete AOP languages, some of which we have outlined in this section (AspectJ, stateful aspects, adaptive programming).

Chapter 3

Edit Distance

In this chapter we introduce the principle of edit distance, between pairs of strings and pairs of trees, respectively. We show how this distance can be computed using dynamic programming.

3.1 String Edit Distance

The *edit distance* can be computed between an ordered pair of strings ([JP04] for an introduction). This distance relies on two concepts. Firstly, there are three *edit operations* that apply to a string of symbols. These operations are given below, where the notation in between parenthesis is the commonly used shorthand to denote the operation.

- **substitution** of a symbol x for a symbol y ($x \rightarrow y$)
- **deletion** of a symbol x ($x \rightarrow \lambda$)
- **insertion** of a symbol x ($\lambda \rightarrow x$)

The second fundamental concept is that of a *scoring function* σ, which assigns a cost to each edit operation. This function maps each $x \rightarrow y$, $x \rightarrow \lambda$ and $\lambda \rightarrow x$ to a positive integer. It considers the symbols x and y in doing so. Most commonly, the scoring function takes into account whether $x = y$ for a given $x \rightarrow y$. Alternatively, in some approaches a separate edit operation **match** is defined specifically for when $x = y$.

An *edit script* is a series of edit operations. Its cost is simply the sum of the costs of the operations it contains. Any string can be transformed into any other string using such a script. The edit distance between two strings s_1 and s_2 is the cost of a minimum cost edit script that transforms s_1 into s_2.

The concept of *string alignments* gives us another way of reasoning about edit distance. Such an alignment is given in Table 3.1. Ignoring all λ with which the strings are interspersed, it is the upper string ("ATCTGAT") we want to

A	T	λ	C	λ	T	G	A	T
λ	T	G	C	A	T	λ	A	λ

Table 3.1: An alignment between two strings of DNA.

transform into the lower one ("TGCATA"). An alignment also hands us an edit script. To retrieve this script, we read the columns from left to right. From each column, with top symbol x and bottom symbol y, we derive an edit operation $x \rightarrow y$ (note that x and y can be λ). A one-to-one correspondence exists between the edit scripts that transform a particular string into another one, and the alignments between those two strings. Hence it is sensible to talk about the cost of an alignment. Also because of this correspondence, the problem of computing the edit distance reduces to finding an appropriate alignment.

Suppose s_1 and s_2 are two strings we want to compute the edit distance for. They are of length n and m, respectively. Let $s_{i,j}$ denote the cost of a minimum cost alignment between prefixes $s_1[1 \rightarrow i]$ and $s_2[1 \rightarrow j]$. Hence, the edit distance between s_1 and s_2 is $s_{n,m}$. We have the following recurrence relation for computing $s_{n,m}$:

$$s_{i,j} = min \begin{cases} s_{i-1,j} + \sigma(s_1[i] \rightarrow \lambda) \\ s_{i,j-1} + \sigma(\lambda \rightarrow s_2[j]) \\ s_{i-1,j-1} + \sigma(s_1[i] \rightarrow s_2[j]) \end{cases} \tag{3.1}$$

One can implement a tree-recursive algorithm that starts from the initial problem $s_{n,m}$. The algorithm deploys the above recurrence relation to break the problem into (the combination of) three subproblems, which are then similarly solved in top-to-bottom fashion. However, this algorithm has exponential complexity. It is therefore not suitable to deploy in practice. This is where the principle of dynamic programming steps in. It is based on the following observation: a tree-recursive algorithm can recompute the same subproblem many times. In this case, there are many paths from the initial problem $s_{n,m}$ to the same subproblem $s_{u,v}$ (for several different values of $u, v \leq n, m$). Because of this, $s_{u,v}$ is repeatedly solved in the basic tree-recursive approach. In dynamic programming, the recursive computation strategy is reversed. That is to say, all subproblems are systematically solved in a bottom-up manner. The key idea is that any $s_{i,j}$ is computed from values $s_{x,y}$ $(x, y \leq i, j)$ which were computed and stored earlier. As such, computing any $s_{i,j}$ is performed in constant time. This gives rise to the algorithm for string edit distance in Table 3.2. The computational complexity of this algorithm is a comfortable $O(m \times n)$.

The algorithm in Table 3.2 stores "backtracking pointers" in matrix b. The "North", "West" and "Northwest" pointers each relate a direction within matrix b. Optimal alignments (thus edit scripts) for the strings s_1 and s_2 can be constructed through following these pointers, starting from $b_{n,m}$.

```
input: $s_1$ and $s_2$ to compute the edit distance for

Create an $(n \times m)$-matrix $b$.
for $i \leftarrow 0$ to $n$:
    $s_{i,0} \leftarrow 0$
for $j \leftarrow 1$ to $m$:
    $s_{0,j} \leftarrow 0$
for $i \leftarrow 1$ to $n$:
    for $j \leftarrow 1$ to $m$:
        Compute $s_{i,j}$ according to the recurrence relation.
        Store $s_{i,j}$ for later use.

        $b_{i,j} \leftarrow \begin{cases} \text{“}North\text{”} & if\ s_{i,j} = s_{i-1,j} + \sigma(s_1[i] \rightarrow \lambda) \\ \text{“}West\text{”} & if\ s_{i,j} = s_{i,j-1} + \sigma(\lambda \rightarrow s_2[j]) \\ \text{“}Northwest\text{”} & if\ s_{i,j} = s_{i-1,j-1} + \sigma(s_1[i] \rightarrow s_2[j]) \end{cases}$

return($s_{n,m}$,$b$)
```

Table 3.2: A dynamic programming algorithm for string edit distance.

We provide a generalization of the principle of backtracking pointers. Given a recurrence relation such as Equation 3.1, we call the left-hand side of the formula the “bigger problem”. Each clause on the right-hand side is formulated in terms of a “smaller problem” and (the cost of) an edit operation. When the recurrence is deployed, the optimal clause is selected with respect to the global goal of minimizing the cost of the edit script. The optimal clause encompasses the optimal smaller problem and the optimal edit operation. A backtracking pointer relates the bigger problem with the optimal smaller problem. Additionally, the optimal edit operation is associated with the pointer. In following the pointers from the initial problem to the smallest subproblems, the associated edit operations form an optimal edit script. For this process to be possible, a backtracking pointer needs to be associated with each subproblem.

The algorithm in Figure 3.3 precisely uses backtracking matrix b to print out an optimal edit script that transforms a given string into another.

3.2 Tree Edit Distance

We outline the algorithm of Shasha and Zhang [SZ97] to compute the edit distance between trees ([Bil05] for an additional introduction to tree edit distance). The algorithm imposes a number of conditions. More specifically the trees must be:

- rooted;
- labeled, i.e. each node is assigned a label;
- numbered in left-to-right post-order fashion.

```
**input:**
We want to compute the edit script that transforms $u$ in $v$.
$b$ is the backtracking table that is achieved as a side effect of
computing the edit distance between $u$ and $v$.
**PrintScript**($u$, $v$, $b$, $i$, $j$):
    if $i = 0 \vee j = 0$:
        return
    if $b_{i,j}$ = "$North$"
        print $u_i \rightarrow \lambda$
        **PrintScript**($u$, $v$, $b$, $i-1$, $j$)
    if $b_{i,j}$ = "$West$"
        print $\lambda \rightarrow v_j$
        **PrintScript**($u$, $v$, $b$, $i$, $j-1$)
    if $b_{i,j}$ = "$Northwest$"
        print $u_i \rightarrow v_j$
        **PrintScript**($u$, $v$, $b$, $i-1$, $j-1$)
```

Table 3.3: The algorithm for retrieving the optimal edit script that transforms one string into another.

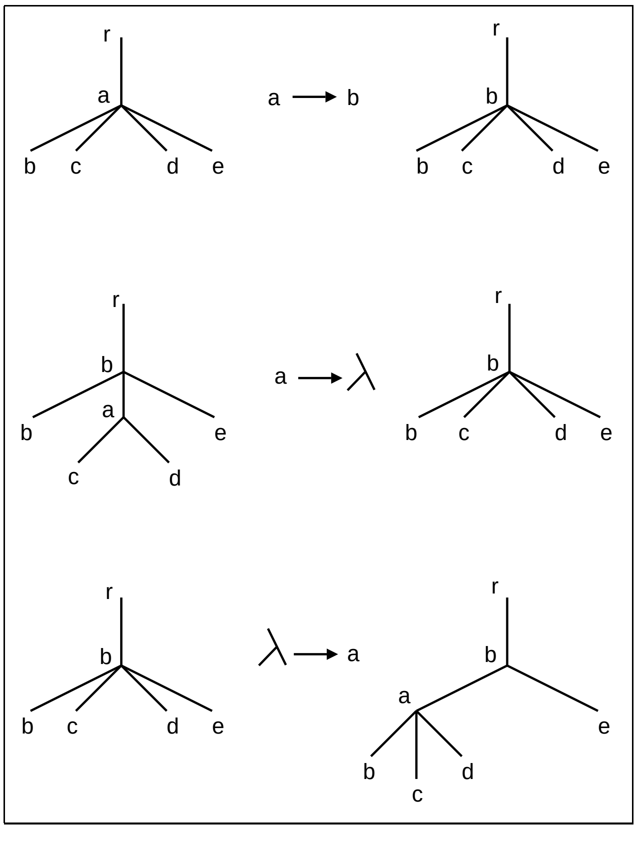


Figure 3.1: The three edit operations on trees.

Left-to-right post-order numbering is a depth-first numbering scheme. The traversal processes the children of a node from left to right. A node is only numbered after all its children have been assigned a number. (The two trees of Figure 3.2 are numbered in this fashion.)

Again we define edit distance in terms of edit operations. The operations for trees are displayed in Figure 3.1. They are respectively: substitution of one node for another, deletion of a node, and insertion of a node. To denote edit operations we use the arrow notation on node labels, unless the labels alone do not tell us which nodes are actually meant. In case a node n is deleted, its children are kept in the tree, where they are made children of the parent of n. In case a node n is inserted as a child of node p, a consecutive subsequence of the children of p becomes the children of n – without further specification as to which subsequence.

The concepts of a scoring function, an edit script and ultimately the edit distance itself, all carry over from edit distance for strings. That is to say, tree edit distance is the minimum accumulated cost to transform one tree into the other, using deletions, insertions and substitutions of nodes. However, the recurrence relation for trees is more complex. To be able to explain it, we must introduce some new concepts and notation:

- Θ is the empty tree.

For a given tree T_1:

- $t[i_1]$ is the node $\in T_1$ with number i_1;
- $T[i_1]$ is the subtree $\subset T_1$ rooted at $t[i_1]$
- $l(i_1)$ is the number of the left-most descendant of $t[i_1]$;
- $\mathit{forest}(l(i_1)..j_1)$ is the *forest* in T_1 induced by the nodes $t[l(i_1)]$ up to $t[j_1]$ – this forest is an ordered collection of trees, where $t[j_1]$ is the root of the right-most tree.

Note that for $i_1 = j_1$, $\mathit{forest}(l(i_1)..j_1)$ is the subtree $\subset T_1$ rooted at $t[j_1]$. This relation proves important throughout the rest of this text.

For an additional tree T_2, where $t[i_2]$ and $t[j_2] \in T_2$:

- $\mathit{treedist}(i_1, i_2)$ is the tree edit distance between the subtrees rooted at $t[i_1]$ and $t[i_2]$;
- $\mathit{forestdist}(l(i_1)..j_1, l(i_2)..j_2)$ is the edit distance between the given pair of forests.

The concept of an alignment is replaced by that of a *mapping* between two trees T_1 and T_2. Such a mapping is illustrated in Figure 3.2. It is a relation $M \subset T_1 \times T_2$ that draws lines between the nodes of the trees. Nodes in T_1

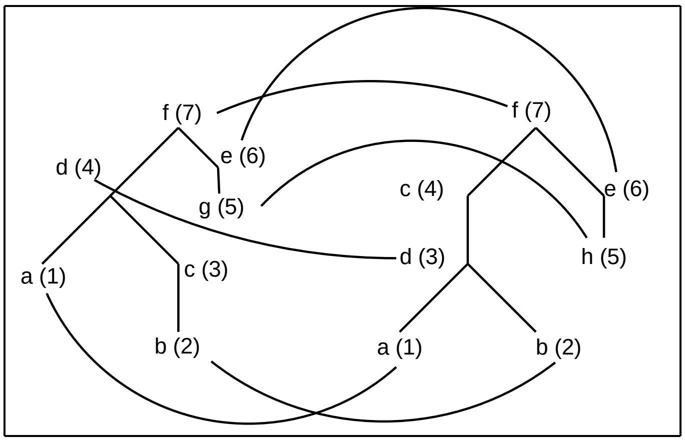


Figure 3.2: A mapping from one tree to another.

that are not touched by a line are regarded as deleted (from the first tree). Untouched nodes in T_2 are regarded as inserted (into the second). Furthermore, the following conditions must hold on any $(t_1[i_1], t_2[i_2])$ and $(t_1[j_1], t_2[j_2])$ in M:

$t_1[i_1]$ is to the left of $t_1[j_1]$ iff $t_2[i_2]$ is to the left of $t_2[j_2]$ (sibling order preserved)

$t_1[i_1]$ is an ancestor of $t_1[j_1]$ iff $t_2[i_2]$ is an ancestor of $t_2[j_2]$ (ancestor order preserved)

Intuitively, and as with alignments for strings, there exists a one-to-one correspondence between the edit scripts that transform a particular tree into another, and the mappings between those two trees. Mappings thus inherit the cost of the edit script they map to.

We now present the recurrence relation for tree edit distance, in the form of three lemmas (Table 3.4). These lemmas presume trees T_1 and T_2 of which $t_1[i_1]$ and $t_2[j_1]$ are the respective roots. We want to compute the edit distance between T_1 and T_2. This distance is $forestdist(l(i_1)..i_1, l(j_1)..j_1)$. We assume Lemma 1 to be self-explanatory. Likewise Lemma 3 follows directly from Lemma 2.

In Lemma 2 we seek a minimum-cost mapping between $forest(l(i_1)..i)$ and $forest(l(j_1)..j)$. With respect to the nodes $t_1[i]$ and $t_2[j]$, a mapping can impose the following relations:

1. $t_1[i]$ is not touched by a line in M.
2. $t_2[j]$ is not touched by a line in M.
3. $t_1[i]$ and $t_2[j]$ are both touched by lines in M.

Lemma 1 *Let $i \in desc(i_1)$ and $j \in desc(j_1)$. Then*

$$forestdist(\Theta, \Theta) = 0$$
$$forestdist(l(i_1)..i, \Theta) = forestdist(l(i_1)..i-1, \Theta) + \sigma(t_1[i] \to \lambda)$$
$$forestdist(\Theta, l(j_1)..j) = forestdist(\Theta, l(j_1)..j-1) + \sigma(\lambda \to t_2[j])$$

Lemma 2 *Let $i \in desc(i_1)$ and $j \in desc(j_1)$. Then*

$forestdist(l(i_1)..i, (j_1)..j) =$

$$min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \sigma(t_1[i] \to \lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \sigma(\lambda \to t_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ +forestdist(l(i)..i-1, l(j)..j-1) + \sigma(t_1[i] \to t_2[j]) \end{cases}$$

Lemma 3 *Let $i \in desc(i_1)$ and $j \in desc(j_1)$. Then*

(1) if $l(i) = l(i_1)$ and $l(j) = l(j_1)$

$forestdist(l(i_1)..i, (j_1)..j) =$

$$min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \sigma(t_1[i] \to \lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \sigma(\lambda \to t_2[j]) \\ forestdist(l(i_1)..i-1, l(j_1)..j-1) + \sigma(t_1[i] \to t_2[j]) \end{cases}$$

(2) if $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$

$forestdist(l(i_1)..i, (j_1)..j) =$

$$min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \sigma(t_1[i] \to \lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \sigma(\lambda \to t_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + treedist(i, j) \end{cases}$$

Table 3.4: The recurrence relation for tree edit distance.

It is obvious that this list is exhaustive. We now examine the respective consequences of the three cases:

1. $(i, \lambda) \in M$. This gives the first clause of the recurrence in Lemma 2.
2. $(\lambda, j) \in M$. This gives the second clause of the recurrence.
3. $(i, k) \wedge (h, j) \in M$. $t_2[k] \in forest(l(j_1)..j)$. $t_1[h] \in forest(l(i_1)..i)$. We can reflect on the orientation of node i with respect to node h. Suppose the following:

   (A) h is an ancestor of i. This is impossible because i is the root of the right-most tree in the forest.

   (B) h is to the right of i. Impossible for the same reason as above.

   (C) i is to the right of h. Through the sibling ordering condition on mappings, k must be to the right of j. This is impossible for the same reason as in (A).

   (D) i is an ancestor of h. Through the ancestor ordering condition on mappings, k must be an ancestor of j. This is impossible for the same reason as in (B).

   Hence $h = i$. Through an analogous proof $k = j$, and $(i, j) \in M$. Now, by the ancestor condition on mappings, any node in subtree $T_1[i]$ can only be touched by a node in subtree $T_2[j]$. This has the consequence that we can split the third clause of our recurrence into two independent subproblems, which we then combine. One subproblem is the distance between $T_1[i] - i$ and $T_2[j] - j$. The other is the distance between the nodes in $forest(l(i_1)..i)$ and $forest(l(j_1)..j)$ that remain.

The actual algorithm by Shasha and Zhang is based on a number of observations. Firstly, to compute any $treedist(i, j)$ it is necessary to compute almost all $treedist(i_s, j_s)$ where $i_s < i$ and $j_s < j$. This suggests a dynamic programming approach to limit the amount of computations, as in the case of string edit distance. The two `for`-loops in the pseudocode of the tree edit distance algorithm, in Table 3.5, realize this idea. However, when an algorithm straight-forwardly implements the dynamic programming approach, it has time complexity $O(|T_1|^2 \times |T_2|^2)$, where $|T|$ is the number of nodes in T. This is easily prohibitive in practice.

Secondly, when $l(i) = l(i_1)$ and $l(j) = l(j_1)$, the first recurrence in Lemma 3 is at hand. In that case, $forestdist(l(i_1)..i, (j_1)..j) = treedist(i, j)$. Thus, certain tree distances arise as side effects from computing the distance between other structures. That is why the algorithm in Table 3.5 computes only those tree distances that have not yet arisen as such side effects. Ultimately, the main procedure in Table 3.5 makes use of a subprocedure. This subprocedure similarly follows the dynamic programming approach to compute forest distances. The main procedure makes use of a permanent matrix to store tree distances.

```
input: $T_1$ and $T_2$ to compute the edit distance for

Create an $(|T_1| \times |T_2|)$-matrix $M$ for storing the tree distances.
for $i \leftarrow 1$ to $|T_1|$:
    for $j \leftarrow 1$ to $|T_2|$:
        Use a subprocedure to compute $treedist(i, j)$,
        if it has not been achieved as a side effect
        earlier in the computation process.
return $M(|T_1|, |T_1|)$
```

Table 3.5: A sketch of Zhang and Shasha's dynamic programming algorithm for tree edit distance.

The subprocedure creates a temporary matrix for forest distances, each time it is called. The overall algorithm of Shasha and Zhang has the following time complexity:

$$O(|T_1| \times |T_2| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2)))$$

The overall space requirements is $O(|T_1| \times |T_2|)$.

3.3 Summary

We have introduced the notion of edit distance. This distance is defined as the minimum accumulated cost to transform one entity (string or tree) into the other, using deletions, insertions and substitutions (of symbols or nodes, respectively). A scoring function assigns a cost to each edit operation. Tree-recursive algorithms that compute the edit distance are typically expensive, as they can repeatedly solve the same subproblem. The dynamic programming approach avoids this repetition by using a bottom-up strategy in computing the subproblems. However, while the dynamic program for string edit distance has quadratic time complexity, Zhang and Shasha's algorithm for tree edit distance is significantly more expensive. The technique of Tree Clustering we propose in Chapter 7 computes edit distance between parse trees of methods.

Chapter 4

Cluster Analysis

The aim of cluster analysis is to divide a set of data objects into clusters, such that the objects in a cluster are similar to each other, but different from objects in any other cluster. Numerous algorithms exist that perform cluster analysis. We discuss different categories of such algorithms, where each category is illustrated with at least one concrete algorithm. We conclude with an overview of the surveyed algorithms, and a section on cluster evaluation, which is concerned with evaluating the result of cluster analysis.

4.1 What is Cluster Analysis?

Cluster analysis aims to group similar objects. (See [HK00], [Ber02], [TSK05], and [And05] for general introductions to cluster analysis.) It is often said to be a machine learning technique. Sometimes, it is labeled a statistical technique. Throughout the past, cluster analysis has been used in various domains, from sociology to biology.

The input to cluster analysis is two-fold. Firstly, a set of data objects is required. Additionally, the user needs to provide a binary distance measure between pairs of these objects.[1] Given this input, cluster analysis aims to divide the data set into clusters, i.e. groups of objects, such as to maximize:

- **cluster cohesion**, which is the degree to which objects in the same cluster are similar to one another;
- **cluster separation**, which is the degree to which objects in different clusters are dissimilar to one another.

All notions of proximity between objects (such as similarity, distance and dissimilarity), are derived from the same user-provided distance measure. The result of cluster analysis is a *clustering* (hierarchical algorithms produce multiple

[1]Especially in this dissertation, it is important to note that the objects in cluster analysis are typically not objects in object-orientation.

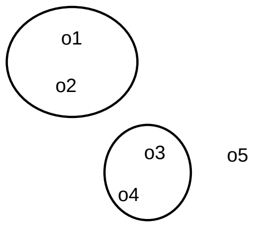


Figure 4.1: The diagram shows a clustering of 5 objects.

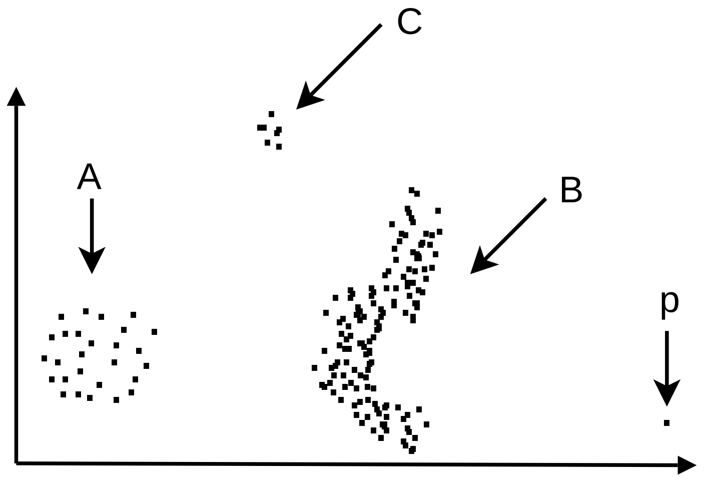


Figure 4.2: Three natural clusters and one outlier, in two-dimensional Euclidean space.

clusterings, see Section 4.5). Sometimes a diagram is used to illustrate a clustering, as in Figure 4.1. Cluster analysis is often regarded to be the search to capture the *natural* clustering in a data set. Typically, the clusters in a clustering are non-overlapping. Alternately, each object can be assigned to multiple clusters, where the membership of an object to a cluster is weighted. This approach is called *fuzzy clustering* [TSK05].

Often, the set of data objects has attributes. In that case, all objects share these attributes, but typically have different values for them. (If the objects are people, each person has an age, weight, etc.) Generally, an attribute is of a certain kind; it is *numerical* (i.e. assumes numerical values), or *categorical* (the values have no numerical interpretation). If a data set has n attributes, then the *dimensionality* of that set is n. (While we adhere to the principle of having shared attributes amongst all the objects, we commonly talk about the "attribute of an object"; similarly, we often say an object to have a dimensionality, while it is the same dimensionality for all the objects.) Alternately, when the objects have no attributes, cluster analysis performs strictly on the basis of the provided distance measure. In that case, cluster analysis works in a *distance space.*

Throughout textbooks and articles, principles of cluster analysis are illustrated in the simple case where the objects have two numerical attributes, and the distance measure between them is standard Euclidean. This is done due to the ease with which this case can be presented graphically. It is in this context that some characteristics of clusters are generally explained. Figure 4.2 gives such a graphic. Three natural clusters (labeled A, B, C) are visually apparent. We can note the following:

- cluster A is of lower *density* than cluster B;
- cluster A is of *globular shape*, while cluster B is *non-globular*;
- cluster A is of greater *size* than cluster C.

Some objects do not belong to any natural cluster. Such objects are called *outliers.* Object p in Figure 4.2 is an outlier. Special consideration needs to be given to outliers, to prevent them from skewing the result of cluster analysis. Often, a special filtering phase is installed, to do away with these oft-malicious objects.

A *cluster algorithm*, which implements cluster analysis, can be characterized by a number of properties. Specifically, the following questions can be asked regarding a cluster algorithm.

- How sensitive is the algorithm to outliers?
- Can it detect clusters of non-globular shape?
- How well can it handle a clustering in which the natural clusters are of diverse shapes, sizes, or densities?
- Does it support distance spaces?
- Does it support non-numerical attributes?
- How well does it perform in the face of objects with high dimensionality? (With more dimensions, it becomes less likely to distinguish well-separated clusters.)
- Is it scalable (in terms of both time and space requirements)?
- What input does it require from the user? (E.g. some algorithms require the user to provide the desired number of clusters, which can put a considerate problem into his, or her hands.)

There is a near uncountable amount of different cluster algorithms. Therefore, the algorithms are commonly categorized according to a taxonomy. There is always some overlap between categories, in that an algorithm can show characteristics of more than one category. Also, there is not one common taxonomy. We derive our own from both [TSK05] and [HK00]. Concretely, we take cluster algorithms to implement the single pass clustering scheme, or to be partitional, hierarchical, density-based, grid-based, model-based, or subspace-based.

4.2 Distance Measures

The process of cluster analysis relies on a distance measure. In some cases, this measure is domain-specific (e.g. it is the edit distance between strings as discussed in Chapter 3). However, typical distance measures exist that can be deployed in many domains. For instance, standard Euclidian distance can always be computed for objects with numerical attributes.

4.2.1 Jaccard Distance

An attribute can be binary, in that it either assumes value 0 or 1. When two objects have only binary attributes, similarity between them can be computed according to the *simple matching coefficient* (SMC):

$$SMC(object_1, object_2) = \frac{\text{number of matching attributes}}{\text{number of attributes}}$$

where an attribute matches only if it has the same value in both objects. The problem with this formula is that binary attributes can be *asymmetric*. This means that one value is regarded more important than the other. Commonly, the convention is adopted to let 1 signify the most important value, while 0 is then of lesser importance. In this context, a matching attribute is more significant if it has value 1 in both objects.

The usual example of asymmetric attributes is that of (binary) documents. Each attribute relates to a word. An attribute value is 1, if the document contains that word; otherwise it is 0. Typically, there are many more words (attributes) than a single document contains. Then, documents have many more 0-value attributes than positives. If SMC were applied in this case, two documents could be regarded similar on the basis of words neither of them contain. This is generally undesirable.

The Jaccard distance measure remedies the shortcoming that SMC exhibits in the light of asymetric attributes. This measure is computed as follows:

$$Jaccard(object_1, object_2) = \frac{\text{number of attr. that do not match}}{\text{number of attr. not involved in “}0-\text{matches”}}$$

A 0-match occurs when two objects have value 0 for the same attribute. A simple example can clarify the difference between SMC and Jaccard. Suppose that in the case of (binary) documents, the attributes (thus words) are the following:

(vehicle, automobile, benny, elita, alfredo, johnny, fred, monkey, ape)

Document d_1 is about cars, it contains the words “vehicle” and “automobile”. Thus, if we suppose that the above list of attributes imposes an ordering, the object for d_1 has the following attribute values:

$$(1, 1, 0, 0, 0, 0, 0, 0, 0)$$

Document d_2 is about primates, it contains the words "monkey" and "ape". The corresponding object has the following attribute values:

$$(0, 0, 0, 0, 0, 0, 0, 1, 1)$$

As such,

$$SMC(d_1, d_2) = \frac{5}{9}$$

$$Jaccard(d_1, d_2) = \frac{4}{4} = 1$$

Because the Jaccard is a distance, we convert it to a similarity through the formula

$$1 - Jaccard$$

which gives 0. Thus, while Jaccard (rightfully) recognizes no similarity between the two documents, SMC does see a similarity based on the 0-matches.

4.2.2 Cosine Similarity

Similarly to Jaccard distance, cosine similarity is tailored for asymetric attributes. In contrast, the cosine measure is not restricted to binary attributes; more general types of numerical attributes are supported. In this case, asymmetry signifies the importance of non-zero values over zero values. Thus, objects must be considered similar based on the number of attributes for which they both have a non-zero value.

Cosine similarity stems from the domain of document clustering [ZK02], in which it is the intent to group documents that have similar content. In this domain, an object is a document, while an attribute relates to a word. The value of an attribute signifies the frequency with which that word appears in the document. Since the attributes are ordered, the objects can be represented as *document vectors*. As an example of a document vector, consider:

$$(1, 6, 3, 7, 5)$$

where the numbers are the attribute values. Cosine similarity is then computed as follows:

$$cosine(o, p) = \frac{\sum_{k=1}^{n} o_k p_k}{\sqrt{\sum_{k=1}^{n} o_k^2}\ \sqrt{\sum_{k=1}^{n} p_k^2}}$$

where n is the dimensionality of the objects, and o_i is object o's value for attribute i. The denominator in the formula normalizes by the norms of the vectors. Hence, the magnitudes of the vectors is no factor in the computation of the cosine measure.

4.3 Single Pass Clustering

Single pass clustering is one of the most inexpensive techniques in cluster analysis. The technique relies on an ordering of the data objects. It traverses this ordering a single time. In doing the traversal, the technique maintains a set S of clusters that have been formed so far. When an object is visited, it is either merged with an existing cluster in S, or forms a new singleton cluster to be put in S. A merger occurs between an object and a cluster, only if the cluster is closest to the object of all clusters in S, and the distance between the two candidate components is below a certain user-provided threshold.

The time complexity of the single pass scheme is $O(n \times m)$, where n and m are the numbers of objects and clusters, respectively. Thus, the complexity is $O(n^2)$ in the worst case, but generally lower. The space complexity is bounded by a low $O(n)$. On the downside, the result of the single pass depends on an ordering of the objects, which is often taken to be arbitrary. Additionally, single pass clustering seeks to be light-weight in its complexity, rather than to provide a clustering of good quality. In fact, the technique has no characteristics that ensure any degree of quality in cluster analysis.

4.4 Partitional Clustering

A partitional cluster algorithm is iterative. It starts from an initial clustering, then iterates over that clustering until it is deemed to be a suitable solution. In each step of the iteration, some objects are relocated from one cluster to another. This is done as to increase, or decrease the value of an *objective function.* In some algorithms this function is implicit. Because the objective function is global (to every step of the analysis), the resulting clustering typically has some desirable property. As an example, the resulting clusters can be *center-based.* This means that every object is closer to the center of its cluster than to the center of another cluster. (The exact definition of the center depends on the specific algorithm used.) Partitional algorithms also have some potentially undesirable properties. The resulting clustering can be highly dependent on the initial clustering, which in turn is generally dependent on some input-parameters. When a technique religiously attempts to increase (or decrease) some global objective function, the technique can get stuck at a local maximum (minimum).

4.4.1 K-means

The well-known K-means technique [Har75] is a partitional algorithm that provides center-based clusters. In this case, the center of a cluster is its *centroid.* This is not necessarily one of the objects in the cluster. Rather, the notion of a centroid implies that it is computed, rather than selected. This has the consequence that K-means is only applicable to numerical attributes. The basic K-means algorithm is as follows:

```
1. Select $k$ objects as initial centroids.
2. repeat
     • Form $k$ clusters by assigning each object to its most similar centroid.
     • Recompute the centroid of each cluster.
3. until Centroids do not change
```

Not in all cases does the algorithm converge to a stationary positioning of the centroids. Therefore, the halting condition of the algorithm is sometimes altered (e.g. a fixed number of iterations is specified).

No objective function is apparent in the pseudocode of K-means. However, in specifying a concrete instantiation of the basic algorithm, the choice of (implicit) objective function is nonetheless important. This choice partly determines the convergence properties of K-means. A commonly-used objective function is the sum of squared error (SSE). It maps each clustering to a positive number, as follows:

$$SSE(clustering) = \sum_{i=1}^{K} \sum_{x \in C_i} dist(c_i, x)^2$$

where

C_i is cluster i in the clustering,

k is the number of clusters,

x is a data object,

c_i is the centroid of C_i,

$dist$ is the standard Euclidean distance.

SSE gives a measure of error between the objects and the centroids they are assigned to. Naturally, this is an objective function we want to minimize; a smaller SSE indicates a better fit of the objects to their centroids. One of the cases in which K-means is guaranteed to converge, is the case where:

1. SSE is the (implicit) objective function,
2. Euclidean distance is the distance measure between the objects,
3. the centroid of a cluster is its mean (where the mean of a group of objects constitutes the mean of each attribute).

Convergence to an optimal however, is not guaranteed in the above case. The other issues regarding K-means include, firstly, the choice of the initial centroids. The resulting clustering is highly sensitive to this initial setting. Sometimes the initial centroids are taken randomly, or they are taken so that they (roughly) lie as far apart as possible. Another option is to perform multiple runs of K-means, starting from a different initial setting each time. Alternately, a hierarchical cluster algorithm (Section 4.5) can determine the initial centroids in a pre-processing step. Secondly, the user needs to provide the parameter k, of the desired number of clusters. This puts a definite complication into his, or her hands.

Some problems are associated with the general approach of mining center-based clusters. Especially natural clusters of non-globular shape are hard to characterize using one central object. These clusters are generally not detected by K-means and similar algorithms. A measure like SSE is often minimized by (globular) clusters of equal size and density – while the natural clusters might not be that uniform. Additionally, outliers have a definite negative influence on the computation of centers; they "pull" the centers towards them, away from the natural centers.

On the upside, K-means is a simple and lightweight approach. Only the data objects and centroids need to be kept in memory. Convergence typically occurs in the early iterations.

4.4.2 K-medoids

The K-medoids algorithm [KR90] is very similar to K-means, but it does away with the restriction of numerical attributes. That is to say, it supports distance spaces. To this end, K-medoids replaces the concept of a centroid with that of a *medoid*. Whereas a centroid is computed, a medoid is selected. More specifically, the medoid of a cluster is its most centrally located object. In a particular cluster, the medoid is that object with the highest average similarity to all the other objects. As with K-means, we can compute the SSE measure of a clustering (with respect to its medoids). Concretely, K-medoids is based on the operation of swapping a medoid with a non-medoid. This is in terms of making the medoid a non-medoid, and vice versa. In each iteration, the algorithm considers which two objects it could swap. It performs the swap that results in the highest decrease in SSE. Naturally, K-medoids inherits all negative points of K-means. It even has an additional shortcoming, in that it is relatively expensive. Generally, the algorithm is only for use in the case of small data sets.

4.5 Hierarchical Clustering

A hierarchical cluster algorithm does not produce one clustering, as a partitional algorithm does. Rather, a hierarchy is built, of which each level is a distinct clustering. An agglomerative hierarchical cluster (AHC) algorithm starts with

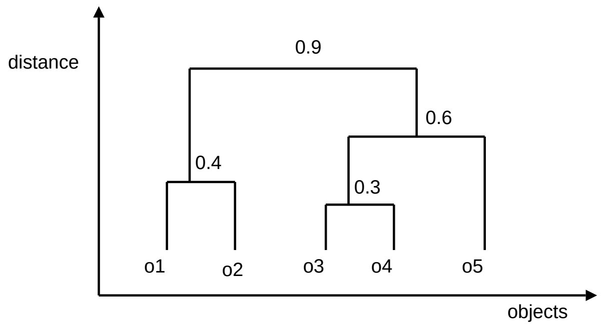


Figure 4.3: This dendrogram represents a hierarchy of five initial objects.

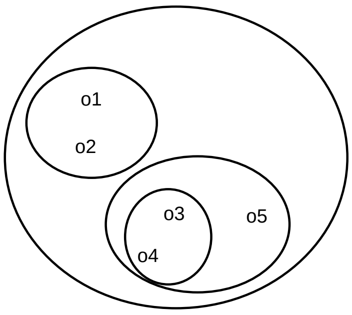


Figure 4.4: This diagram represents a hierarchy of five initial objects.

putting each object into a separate cluster. In each iteration, the two closest clusters are merged. As such, each iteration adds a new level (thus clustering) to the hierarchy. This process is halted when only one cluster remains, or some condition is met. The resulting hierarchy can be illustrated in a *dendrogram* (Figure 4.3), or a diagram (Figure 4.4). (Both figures illustrate the same hierarchy, in which the clustering of Figure 4.1 is contained.) A *composite cluster* is the result of a merger, while the *singleton clusters* are the one-object clusters the AHC process starts with. Each joint in a dendrogram represents a composite cluster; the numbers indicate the distances at which the clusters were merged. In the AHC process, distance between singleton clusters is the user-provided distance between objects. The most important element of a concrete AHC algorithm is precisely how distance between composite clusters is computed.

Divisive hierarchical algorithms also exist. These start from one cluster that encompasses all objects. In each iteration, a selected cluster is split, until only singleton clusters remain. From this point on in the text, we focus on the much more commonly-used AHC approach.

With respect to Partitional Clustering In the hierarchical approach, the user does not have to provide the desired number of clusters. Additionally, there is no initialisation problem as with algorithms like K-means. On the other hand, any merging (or splitting) decision is final; when two objects are put into the same cluster in AHC, they will not relocate to different clusters at any later time. When a mistake is made early in the agglomerative process, it can thus have a definite negative influence on the remainder of the process. A hierarchical algorithm also does not provide any guarantee as to optimize a global objective function; a local optimization problem is solved at each merger or split. Finally, a basic AHC algorithm with no scalability measures has a sometimes prohibitive time complexity of $O(m^2 log(m))$ and space complexity of $O(m^2)$, where m is the number of objects in the cluster analysis.

4.5.1 Lance-Williams

The Lance-Williams framework [LW69] actually encompasses several different AHC algorithms. All of these algorithms implement the AHC process in the most straight-forward way. The only difference between them lies in the manner in which they compute the distance between composite clusters. First, we introduce the algorithms separately.

- **MinLink** *The distance between two composites is the distance between their two closest objects.* This algorithm is good at handling non-globular shapes, but is sensitive to outliers.
- **MaxLink** *The distance between two composites is the distance between their two most distant objects.* This algorithm tends to break large clusters and favours globular shapes. It does handle outliers better.
- **GroupAverage** *The distance between two composites is the average of all distances between an object from one cluster, and an object from the other.* This algorithm is kind of an intermediate algorithm, in between Min-Link and Max-Link. It is therefore difficult to distinguish any particular weaknesses or strengths.
- **Ward** *The distance between two composites is the increase in SSE (see K-means, Section 4.4.1) that results from merging these two clusters together.*
- **Centroid** *The distance between two composites is the distance between their centroids (see K-means).*

All of the above AHC algorithms fit into the Lance-Williams update formula. Imagine we merge the clusters A and B, to form composite cluster R. For the AHC process to be able to continue, it requires the distances between R and each existing cluster Q. The Lance-Williams update formula precisely expresses $d(R, Q)$ (with d the distance function) as a linear function of $d(A, Q)$, $d(B, Q)$ and $d(A, B)$. It is given below:

$$d(R, Q) = \alpha_A d(A, Q) + \alpha_B d(B, Q) + \beta d(A, B) + \gamma |d(A, Q) - d(B, Q)|$$

Each of the 5 AHC algorithms outlined above can be implemented through a setting of the parameters $(\alpha_A, \alpha_B, \beta, \gamma)$ in the Lance-Williams formula. The Lance-Williams formula is typically implemented using a so-called *distance matrix*. This two-dimensional matrix contains the pair-wise distances, and is continuously updated in the agglomerative process. While it can be interesting to discuss the 5 separate algorithms, the unifying formula is particularly handy in implementation.

4.5.2 CURE

Another AHC algorithm is CURE (Clustering Using Representatives) [GRS98]. The main feature of this algorithm is the representation of a cluster through a limited number of representative objects. The first representative is chosen to be the object that is furthest away from the center of the cluster. The others are consecutively chosen to be as far away as possible from the already-determined representatives. The idea is that this model can easily adapt to the geometry of a natural cluster, whether it is globular or non-globular. Of course, because this approach has a particular interest in far away objects, outliers need special consideration; they can otherwise skew the representation of a cluster. Therefore, all representative objects are "shrunk" towards the center of the cluster, with a factor α. This factor is a constant, so that far away objects are shrunk more (from a discrete perspective). In the end, the distance between composite clusters is taken to be the distance between their closest representative objects.

CURE is often classified as a scalability algorithm, because it takes special measures to ensure that larger data sets can be handled. The first measure is that of *sampling*. Instead of engaging all objects into the AHC process, a sample of the complete set is taken. After the agglomerative clustering, each object outside of the sample is assigned to the cluster it is closest to. The second scalability measure is that of *partitioning*. The data set is partitioned into non-overlapping subsets. Each subset is clustered separately, after which a global pass clusters the different clusters from the subsets.

As a more general approach to eliminate outliers, it is noted that clusters involving outliers are typically small. This is because outliers are not close to many objects. CURE thus interrupts the AHC process at two occasions, to filter away the clusters that are deemed too small. This strategy is helped by the manner in which the process of sampling additionally isolates outliers.

While it may not be directly apparant, CURE handles only numerical attributes (e.g. consider the shrinking of the representatives). Due to some guarantees that must be imposed on the scalability measure of partitioning, CURE requires a choice of k, the desired number of clusters. Additional input parameters are the factor α, and the number of representatives in each cluster (the authors of CURE suggest a constant 10 or more for the latter).

4.5.3 ROCK

The authors of CURE present an algorithm that supports distance spaces, namely ROCK [GRS99]. It is similar to CURE in terms of scalability measures and outlier handling. Yet, a different approach to distance is taken. ROCK is based on notions that belong to graph theory. Two objects are *neighbours* when the similarity between them exceeds a certain treshold. Given two objects, there is a *link* between them for every common neighbour they have.

It is a common strategy to regard the distance between composites as the distance between two of their objects. A problem is associated with this kind of approach: two separate natural clusters can be unjustly merged if a few of their objects are neighbours. ROCK recognizes the natural border between the clusters, by observing the relatively low amount of links between them. More precisely, ROCK takes the distance between composites to be the total amount of links between them, divided by the expected number of links. This division is made as to prevent a bias towards merging larger clusters.

4.5.4 CHAMELEON

CURE and ROCK take a different approach in computing distance between composites. Because CURE considers the two closest representatives of a pair of clusters, it can be said that CURE focusses on the *closeness* of clusters. ROCK, on the other hand, computes the aggregate *inter-connectivity* of clusters. In that sense, MinLink is one algorithm similar to CURE, while GroupAverage is more akin to ROCK. Both approaches turn out to have weaknesses in detecting particular types of natural clusterings. Therefore, CHAMELEON [KN99] considers both closeness and inter-connectivity.

Recall that ROCK computes the number of expected links between two clusters, in order to perform some normalization of distance according to cluster size. But this is done according to a *global* model of inter-connectivity. As a consequence, the model generally fails to fit a natural clustering that shows different characteristics in terms of inter-connectivity. CHAMELEON does not have this shortcoming. In computing the closeness and inter-connectivity of two clusters, it takes into account the characteristics of the clusters themselves.

It is not possible to compute closeness and inter-connectivity for single-object clusters. Therefore, CHAMELEON relies on the construction of a so-called k-nearest neighbour graph to build an initial partitioning. This pre-processing step requires two input parameters: k to determine neighbourhood, and $MinSize$ to determine the granularity of the initial clustering.

4.6 Density-based Clustering

Density-based clustering algorithms have in common that they see a cluster as a region of high density, surrounded by a region of low density. Such a cluster is a *density-based* cluster. This notion of a density-based cluster differs from that of a center-based cluster, which is build from one central object inside of it (see Section 4.4). The important difference between these two definitions of a cluster lies in the shapes they can adapt to. Deploying the definition of a center-based cluster tends to result in globular clusters, whereas the definition of a density-based cluster does away with this restriction. That is to say, density-based clustering is better at detecting clusters of arbitrary shape. Additionally, a density-based approach is generally more resistant to outliers than an algorithm like K-means, which we discussed in Section 4.4.1.

4.6.1 DBSCAN

As the prototypical example of a density-based clustering algorithm, we present DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [EKSX96]. In the above introduction to density-based clustering, an important notion is left undefined: that of density itself. DBSCAN fills in this definition of density through a number of new concepts:

The collection of objects within a radius ϵ of a given object is called the **ϵ-neighbourhood** of that object.

An object is a **core object** if its ϵ-neighbourhood contains at least a minimum number $MinPts$ of objects.

An object p is **directly density-reachable** from an object q, if p is within the ϵ-neighbourhood of q, and q is a core object.

An object p is a **border object** if it is not a core object, but it is directly density-reachable from a core object. A **noise object** is neither a core or a border object.

An object p is **density-reachable** from an object q, if there is a chain of objects $p_1, p_2, p_3, ..., p_n$ with $p_1 = q$ and $p_n = p$, such that p_{i+1} is directly density-reachable from p_i.

An object p is **density-connected** to object q, if there is an object o such that both p and q are density-reachable from o.

DBSCAN is thus dependent on the parameters ϵ and $MinPts$. A cluster is defined as a maximal set of density-connected objects. Concretely, DBSCAN puts a link in between two core objects if they are directly density-reachable from one another. Clusters are formed by collecting maximum-size connected graphs of core objects. Border objects that are close enough to a core object are put into the same cluster as the core object (a form of resolution is required in this step, as a border object may be on the border of multiple clusters). Noise objects are naturally discarded.

Aside DBSCAN's desirable properties, the algorithm also knows a few shortcomings. The parameters ϵ and $MinPts$ must be chosen carefully by the user.

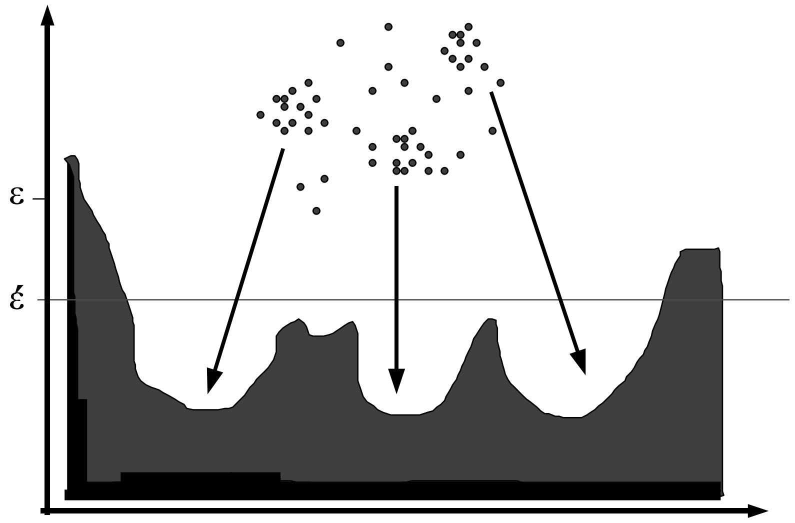


Image reproduced from [ABKS99].

Figure 4.5: A cluster visualization technique of OPTICS.

Different parameter settings can result in widely different clusterings. Additionally, these parameters are global to the algorithm. This makes it difficult for DBSCAN to handle a data set in which the natural clusters have different densities: each cluster might require a different parameter setting to be detected.

Due to the data structures and technicalities of the DBSCAN implementation, the algorithm is restricted to numerical data. (Although we imagine the general approach, as above-described, to be applicable in a distance space.) Finally, DBSCAN has trouble with higher-dimensionality objects, as density becomes more difficult to define in higher dimensions.

4.6.2 OPTICS

To overcome the difficulties associated with the parameters ϵ and $MinPts$ of DBSCAN, Ankerst et. al propose OPTICS (Ordering Points to Identify Clustering Structure) [ABKS99]. This algorithm provides a result that encompasses several runs of DBSCAN with a wide array of different parameter settings. OPTICS is based on the following observation regarding DBSCAN: higher density clusters (detected using a lower ϵ) are completely contained in lower density clusters (captured through a higher ϵ). To iterate through “several runs of DBSCAN”, OPTICS iteratively expands clusters with new objects (one object at a time). This gives a succession of clusterings. Concretely, what OPTICS does is not determine one clustering, but rather an ordering of the data objects. This ordering defines the order in which objects are added to clusters in the iterative expansion. The ordering is determined so that more dense clusters (with the lower ϵ) finish first. To generate the ordering, OPTICS still uses a global ϵ and $MinPts$, but the ϵ parameter can be regarded as an upper bound. To add more depth to this discussion, we introduce two new concepts (concepts from DBSCAN carry over):

The **core-distance** of an object p is the smallest ϵ' that makes p a core object, relative to the global $MinPts$. If p is not a core object according to the global ϵ and $MinPts$, its core distance is undefined.

The **reachability-distance** of an object q with respect to an object p is the greater value of the core-distance of p and the Euclidean distance between p and q. The reachability-distance between p and q is undefined if p is not a core object according to the global ϵ and $MinPts$. Reachability-distance is denoted $RD(p, q)$.

Intuitively, $RD(p, q)$ is the smallest setting of ϵ' that makes q directly density-reachable from p if p is a core object. That is why $RD(p, q)$ can not be smaller than the core distance of p; because in that case, p is not a core object and hence no object is directly density-reachable from it.

OPTICS performs a walk through the set of data objects. In doing this, it assigns each object p two labels: (1) its core-distance, and (2) the smallest reachability-distance with respect to an object considered before p in the walk. When a collection of objects is a density-based cluster according to an $\epsilon' < \epsilon$ and the global $MinPts$, this collection forms a subsequence of the walk. Naturally this walk determines the ordering OPTICS outputs.

A major element in OPTICS is its visualization techniques. One such visualization is given in Figure 4.5. The horizontal axis contains the data objects according to OPTICS' ordering. The vertical axis displays each object's assigned reachability-distance. Clusters are mapped to lower areas in the plot, while information specific to a setting of ϵ' can be read off the plot by setting a horizontal line at ϵ'.

4.7 Grid-based Clustering

Grid-based clustering algorithms are characterized by the grid data structure they impose on the data objects. The primary reason for doing this is an issue of efficiency: the performance of a cluster algorithm that takes the grid-based approach is generally independent of the number of input objects. Rather, the performance depends on the dimensionality of the data, and the number of cells in each dimension.

4.7.1 STING

One prototypical grid-based clustering approach is STING (STatistical INformation Grid) [WYM97]. We assume the data objects lie in a two-dimensional space (i.e. they have two attributes). This case can easily be generalized to more dimensions. The grid data structure exists out of layers of cells. The root

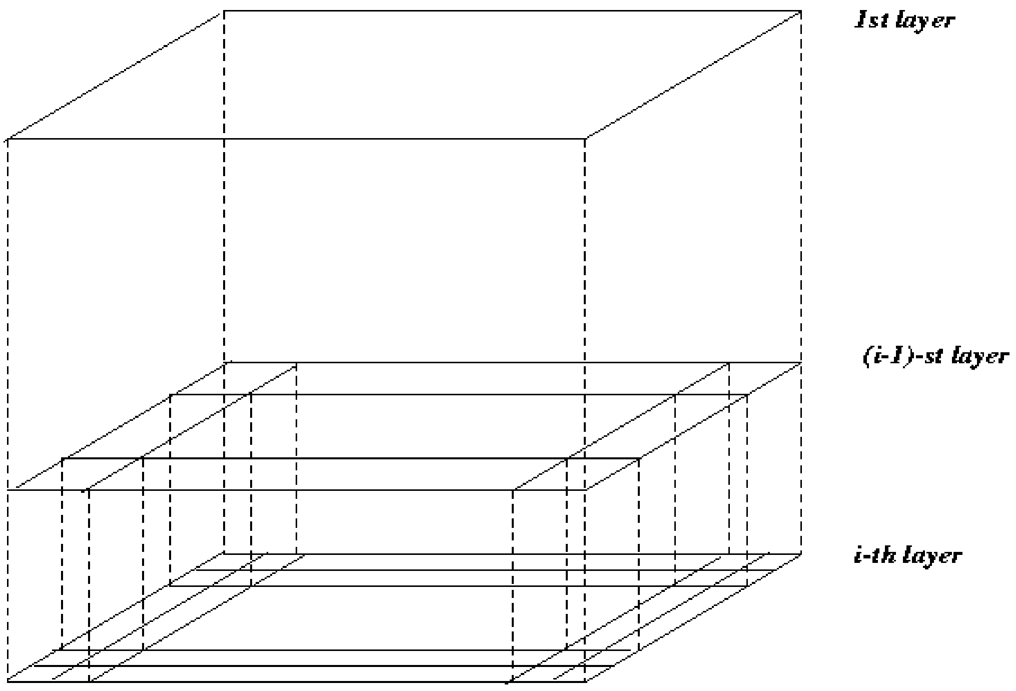


Image reproduced from [HK00].

Figure 4.6: A grid imposed by STING on a two-dimensional data space.

cell contains the entire data space. At the bottom lie atomic cells. The authors of STING, Wang et al., suggest the atomic cells to contain in between several dozens to several thousands of data objects. A cell in a particular layer consists out of a certain number of cells from the layer below it. (Wang et. al. suggest this number to be 4.) This grid structure is illustrated in Figure 4.6. Each cell is labeled with statistics concerning the data objects in that cell. These statistics include:

- the number of data objects.

The following statistics, which a cell is labeled with, relate to one particular attribute of the objects within that cell:

- the mean;
- the standard deviation;
- the minimum and maximum;
- the type of distribution (normal, uniform, exponential, or none if the distribution is not known).

For the atomic cells, these statistics can be derived directly from the data objects. (The “type of distribution” statistic can be set manually, or obtained through hypothesis tests such as χ^2). For the cells in a higher layer, these statistics can be derived from the layer below.

STING is actually a broader technique than cluster analysis. It allows the user to form "region oriented" queries in an SQL-like language. Such queries ask for regions of data objects that satisfy a certain condition, regarding density or otherwise. To answer a query, a starting layer is determined in the grid. Using confidence intervals, the cells (in the starting layer) that are relevant to the query are discriminated from those that are not. One layer below, only those cells are considered that are part of cells that have been proven relevant in the layer above. Confidences are again computed for the lower layer. This process is repeated to form a divisive recursion.

Ultimately, Wang et. al. do compare STING with the classic cluster algorithms. Of course, STING is query-independent while cluster algorithms only answer to the query of cluster analysis. But there is the additional issue of performance. In building the grid, STING makes a single pass over all data objects. Once this structure has been built, answering queries takes processing time $O(g)$, where g is the total number of cells at the lowest layer of the grid. This number g is usually much smaller than n, the number of data objects. Thus, the computational complexity of clustering is $O(n)$. This compares favourably with, e.g. the $O(n\ log(n))$ of DBSCAN and OPTICS (Sections 4.6.1 and 4.6.2, respectively).

4.8 Subspace-based Clustering

Sometimes it is worthwhile to consider only a subset of the attributes of the objects. This is the case when certain attributes do not provide any useful information for cluster analysis. For example, attributes might be duplicates of each other, or an attribute's values might be distributed uniformly. A cluster of documents might be characterized by only a few relevant words, out of the thousands of words that are the attributes of the documents. Noo natural clusters might exist in the space of all dimensions. For all these reasons, subspace-based clustering algorithms exist. These manage to automatically find clusters in a suitable dimensionality.

4.8.1 CLIQUE

CLIQUE [AGGR98] is at the same time a grid-based cluster algorithm, as it is a subspace-based cluster algorithm. But whereas STING (Section 4.7.1) follows a divisive approach, CLIQUE works bottom-up. In CLIQUE, the space of the data objects is divided into non-overlapping cells of equal volume. If the number of data objects in a cell is above a treshold T, the cell is *dense*. A cluster is then simply a collection of adjacent dense cells.

The bottom-up element of CLIQUE arises from the following observation: if a cell in n-dimensional space is dense, then so are its projections in $(n-1)$-dimensional space. Consider a two-dimensional space. Figure 4.7 illustrates

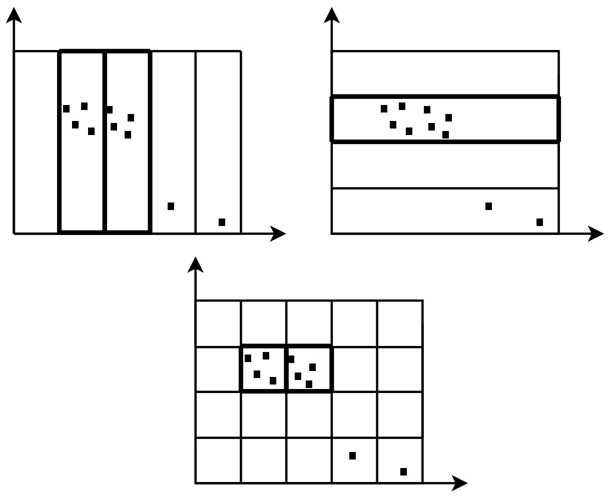


Figure 4.7: CLIQUE divides 1D and 2D into cells, in a two-dimensional data space. Dense cells are bold.

the case. We can partition one dimension into intervals. This gives rise to adjacent rectangles. After we do this for the other dimension as well, we can take the intersection of the dense rectangles in both dimensions, to collect the two-dimensional cells which could be dense. This is generally a subset of all $2D$ cells. (The dense cells in Figure 4.7 are given in bold.) CLIQUE's bottom-up recursion relies on precisely this process of taking the intersection of all dense cells in $(n-1)$-dimensional space, to collect the cells in n-dimensional space which could be dense.

The performance of CLIQUE is independent of the number of input objects. However, the run-time complexity is exponential in the number of dimensions. When too many dense cells are generated in the lower-dimensional spaces, CLIQUE runs into performance problems. On the other hand, CLIQUE searches a space that is generally much smaller than the original space of the objects. Again, as with DBSCAN (Section 4.6.1), the treshold for density is global; a data set with different-density natural clusters is problematic.

4.9 Model-Based Clustering

The idea of model-based clustering is to fit a mathematical model to the data. This generally provides a simpler representation of the data, in which clusters can be fully characterized by a limited set of parameters. An important question is that of which model to fit. Answering this question can be of significant difficulty. The model-based approach can also have trouble dealing with outliers and small-size clusters. While it can be desirable to have a strict mathematical foundation in cluster analysis, this approach does not necessarily do a better job at capturing the natural clusters, than techniques that have a purely intuitive basis.

4.9.1 EM

The better-known approach to model-based clustering is the EM algorithm (Expectation-Maximization, [Mit97] for an introduction). It assumes the objects were generated from a statistical process, and that each object was generated independently. The global model that EM fits to the data is a *mixture model.* This is a collection of several statistical distributions. Each distribution has a probability α_i associated with it, where α_i signifies the probability that any object x was generated from distribution i. That is to say, the probability of an object x in a mixture model is:

$$prob(x) = \sum_{i=1}^{k} \alpha_i prob_i(x)$$

where $1 \rightarrow k$ are the distributions in the mixture model, and $prob_i(x)$ is the probability of x in distribution i. Most often, $1 \rightarrow k$ are taken to be normal distributions, because they are simpler, easier to understand, and have proven to work well in the past.

What EM does is estimate the parameters (e.g. mean, covariance) of the distributions in the mixture model. Concretely, it computes a suitable parameter set Θ to fit the data. When a mixture model contains only one distribution, this can be done through the following *likelihood* function:

$$likelihood(\Theta) = \prod_{i=1}^{n} prob(x_i|\Theta)$$

where $x_{1\rightarrow n}$ are the data objects. The sought-for Θ is the parameter set with maximum likelihood. That is to say, the Θ that maximizes the probability of the data objects. When there is more than one distribution in the model, the following likelihood function holds:

$$likelihood(\Theta) = \prod_{i=1}^{n} \sum_{j=1}^{k} \alpha_j prob(x_i|\theta_j)$$

where Θ encompasses the parameter set of each distribution, i.e. $\Theta = (\theta_1, ..., \theta_k)$. However, it is not practical to maximize the above function through differentiation. To remedy this, another approach is taken. For a moment, we pretend to know the distribution z_i ($\in 1 \rightarrow k$) which object x_i was generated from – even though we don't. This gives rise to the following likelihood function:

$$likelihood(\Theta) = \prod_{i=1}^{n} \alpha_{z_i} prob(x_i|\theta_{z_i})$$

We can then calculate the expected likelihood $E(likelihood(\Theta))$ as follows:

$$E(likelihood(\Theta)) = \sum_{(all\ possible\ z)} (\prod_{i=1}^{n} \alpha_{z_i} prob(x_i|\theta_{z_i}))\ prob(z)$$

where $z = (z_1, ..., z_n)$ signifies a particular set of distribution choices. We can more fully specify this definition through the following equations:

$$prob(z) = \prod_{i=1}^{n} prob(z_i|x_i)$$

$$prob(z_i|x_i) = \frac{\alpha_{z_i} prob(x_i|z_i)}{\sum_{j=1}^{k} prob(x_i|\theta_j)}$$

A notable dependence arises. We want to compute every θ_i by maximizing the expected likelihood. But to compute this likelihood, we need probabilities $prob(z_i|x_i)$ of distributions relative to objects. In turn, to compute these probabilities, we need parameter settings θ_i. That is why EM is a two-phase approximation algorithm. It starts from a guess of Θ, the collection of all parameter settings of all distributions. From this guess, it calculates the probabilities $prob(z_i|x_i)$. This is the “Estimation” step of the algorithm. In the second step, which is the “Maximization” step, the maximal expected likelihood is calculated from the probabilities computed in the previous step. This results in a new guess of Θ, and the two steps of the algorithm are alternated until Θ converges, or a halting condition is met.

4.10 Cluster Evaluation

When a clustering has been produced, it still needs to be evaluated. A number of measures exist that grade a clustering. Typically, such a measure relates a degree of separation, cohesion, or both. Different approaches exist that rely on a certain model of clusters. For example, when center-based clusters were mined, cohesion can be computed off the distance between each object and its center; separation can be derived from the distances between the centers. One should be careful about using such center-based evaluation measures when the natural clusters are non-globular. Contrary to *unsupervised* measures, a *supervised* evaluation measure is computed relative to some externally provided structure. For example, an external source could provide a “correct” label for each object. In that case, correlation can be computed between this correct labeling, and the partitioning provided by the clustering. Additionally, a cluster can, if determined to “capture” a certain label, be evaluated in terms of:

- what percentage of all objects with that label are in the cluster (the cluster’s *recall*);
- what percentage of objects in the cluster have that label (its *precision*);

4.10.1 Silhouette Coefficients

The technique of silhouette coefficients [KR90] takes both cohesion and separation into account. It is computed as follows. With respect to a data object o, the average distance can be computed of o to all the other objects in its cluster. This is a_o. For a particular cluster C, such that $o \notin C$, the average distance can be computed of o to all the objects in C. The smallest such average distance, over all possible clusters save the one o is in, is b_o. The silhouette coefficient of o is then equal to:

$$(b_i - a_o)/max(a_o, b_o).$$

If the coefficient is negative, object o is (on average) closer to the objects in another cluster, than it is (on average) to the objects in its own cluster. When the coefficient is positive, the closer o is to the objects in its own cluster, the higher the coefficient is. The silhouette coefficient of a clustering can be computed by averaging the coefficients of the separate objects. Naturally, one wants the silhouette coefficient to be as high as possible.

4.11 Overview

We have surveyed several cluster algorithms. Furthermore, in Section 4.1 we have identified a number of properties that characterize cluster algorithms. Table 4.1 and 4.2 relate the cluster algorithms we have discussed with (some of) these properties. (The tables are partly due to Andritsos [And05], but we have extended them in parts.) Table 4.1 considers the input parameters that the algorithms require from the user. Table 4.2 considers the following criteria:

- **bias** Does the algorithm have a bias towards clusters of globular shape? Recall that algorithms with such a bias, which typically mine center-based clusers, face problems when confronted with non-globular natural clusters. Or can the algorithm detect arbitrary clusers? Grid-based clustering algorithms naturally bound the clusters according to vertical/horizontal axes.
- **outlier** Does the algorithm take special measures to deal with outliers?
- **complexity** What is the time complexity of the algorithm? Where
  - n is the number of objects to perform cluster analysis on;
  - k is the desired number of clusters;
  - i is the number of iterations in the algorithm;
  - m is the number of clusters in the initial clustering of CHAMELEON.
- **dist(ance)** Does the algorithm support distance spaces? Most commonly, when an algorithm requires the objects to have attributes, it further requires these attributes to be numerical.

K-means	number of clusters
K-medoids	number of clusters
Lance-Williams	none
CURE	number of clusters
ROCK	number of clusters
CHAMELEON	• α, the relative weight with which to consider closeness, with respect to inter-connectivity • $MinSize$, the granularity of the initial clustering • k, for use in determining neighbourhood
DBSCAN	ϵ, $MinPts$
OPTICS	ϵ, $MinPts$
STING	• number of objects in atomic cells • number of cells in a cell
CLIQUE	• treshold for dense cells • proportions of the grid
EM	• initial model parameters • convergence limit

Table 4.1: The discussed cluster algorithms and their input parameters.

	bias	**outlier**	**complexity**	**dist**
K-means	globular	no	$O(ikn)$	no
K-medoids	globular	no	$O(nik(n-k)^2)$	no
Lance-Williams	/	/	$O(n^2 \times log(n))$	/
CURE	arbitrary	yes	$O(n^2 \times log(n))$	no
ROCK	arbitrary	yes	$O(n^2 \times log(n))$	yes
CHAMELEON	arbitrary	yes	$O(nm + nlog(n) + m^2log(m))$	yes
DBSCAN	arbitrary	yes	$O(n \times log(n))$	no
OPTICS	arbitrary	yes	$O(n \times log(n))$	no
STING	vert/hor	yes	$O(n)$	no
CLIQUE	vert/hor	yes	$O(n)$	no
EM	globular	no	$O(n)$	no

Table 4.2: The discussed cluster algorithms and various properties.

In Table 4.2, some entries for the Lance-Williams framework are omitted. This is because the Lance-Williams formula encompasses several algorithms. We refer to Section 4.5.1, where we outlined the individual properties of the separate algorithms.

Chapter 5

Aspect Mining

This chapter discusses aspect mining, the research field in which it is the intent to uncover crosscutting concerns in legacy systems. Several approaches to aspect mining exist, which are surveyed in this chapter. We make the distinction between clone detection techniques and automated identification of aspect candidates.

5.1 What is Aspect Mining?

As discussed in Chapter 2, traditional object-oriented languages know an inherent short-coming, which leads to the presence of crosscutting concerns in object-oriented systems. The idea behind AOP approaches is that these ill-modularized concerns can be put into aspects. It is claimed that a software system built using the AOP paradigm is more understandable, maintainable and evolvable. Hence it is also desirable to refactor existing object-oriented systems to being aspect-oriented systems (i.e. with the crosscutting concerns put into aspects). In research, this migration is divided into two consecutive steps [KM05]:

1. **An aspect mining step** Aspect mining is the activity of discovering those crosscutting concerns that potentially could be turned into aspects, from the source code and/or run-time behaviour of a given software system. Such concerns are called *aspect candidates*.

2. **An aspect refactoring step** Aspect refactoring is the activity of actually transforming the identified aspect candidates into real aspects in the source code.

The research field of aspect mining focusses on mining so-called *legacy systems*. These are large, complex and poorly documented systems. In the case of a legacy system, carrying out the above two steps by hand is hard labour, if not unfeasible in practice. As a consequence, researchers have sought to facilitate this migration process through a wide array of different techniques. There are two general approaches that aide in the aspect mining step specifically. The first is that of *dedicated browsers*. These browsers assist the developer in manually tracing a crosscutting concern throughout the source code of the legacy system. Most commonly, the developer is to provide a “seed” of a concern (i.e. a locality in the source code), after which the browser attempts to provide hotspots in the code related to that concern. Additionally, some browsers offer a query language that the developer can use to explore the scattered path of a concern. Examples of dedicated browsers are Concern Graphs [RM02], Aspect Browser [Leo], JQuery [JD03] and SOUL [SOU].

The other main approach is that of techniques which aim to automatically uncover all aspect candidates in a given software system. Each of these techniques has a notion of what code or run-time symptoms are associated with crosscutting. However, to date no such technique can be called fully automatic: for now a manual post-processing of the mining results is at order, no matter what technique used. It is this category of techniques that is surveyed in the next sections.

5.1.1 Common Terms in Aspect Mining

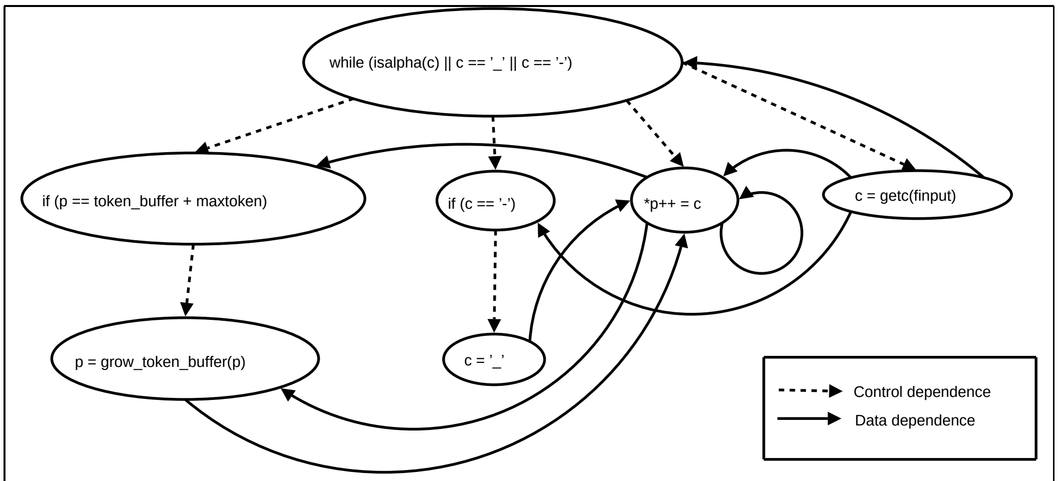
- **False positive** When an aspect mining technique wrongly identifies some data as constituting a crosscutting concern, that data forms a *false positive* in the output of the technique.
- **False negative** When a crosscutting concern is not detected by an aspect mining technique, it is a *false negative* in the output of the technique.

5.2 Clone Detection Techniques

In clone detection [vE05], a legacy system is mined for any piece of code that manifests itself repeatedly over different localities in the source code. The similar code fragments are called *clones*. Such clones are considered malicious much for the same reasons as crosscutting concerns: they are detrimental to the evolvability, understandability and maintainability of the software system. Detection of clones is no easy process, because often the clones are not precise copies of one another. Instead, the clones typically differ to a certain degree, e.g. the variables are named differently over the different clones, or the statements are reordered. As a result, various intricate clone detection techniques have been proposed. In this section, we give a small survey of selected clone detection techniques.

5.2.1 Relation to Aspect Mining

It is the belief of Bruntink et al. [vE05] that crosscutting concerns often lead to a high degree of code duplication. They see two reasons for this. Firstly, by definition a crosscutting concern is not modularized. Its implementation can



```
while (isalpha(c)  ||  c == ’_’  ||  c == ’−’) {
  if (p == token_buffer + maxtoken)
    p = grow_token_buffer(p);
  if (c == ’−’) c = ’_’;
    *p++ = c;
    c = getc(finput);
}
```

Table 5.1: Example code fragment and program dependence graph.

hence not be reused over different localities in the source code. As a result, the developer relies on copy-paste practice instead, to repeat the implementation where necessary. Secondly, a developer may use coding conventions and idioms to impose crosscutting functionality on different code entities, e.g. to add tracing or logging to methods. Clone detection techniques thus become valuable aspect mining techniques.

5.2.2 PDG-based Clone Detection

The program dependence graph (PDG) of a program is a graph that captures the data flow and control flow within that program. A vertex in the graph is either a control predicate (e.g. `if`, `loop`, `while`) or an assignment statement. The edges represent dependencies between these program pieces. An edge can be one of two types. It is either (1) a control dependence edge, where $a \rightarrow b$ means that whenever control predicate a evaluates to true, program piece b will be executed; or (2) it is a data dependence edge, where $a \rightarrow b$ means that program piece a assigns a value to a variable which may be used by program piece b. An example PDG is given in Table 5.1.

PDG's are commonly deployed in *program slicing* [Wei79]. Given the PDG of a program, this practice is the construction of a subgraph (i.e. slice) in that PDG, starting from a slicing *criterion*. This criterion is a locality in the program, and a vertex in the PDG. The idea is that the subgraph captures that specific subset of the program of which the run-time behaviour could affect, or could be affected by (the execution of) the slicing criterion. Slicing is thus a

`if (tmp->nbytes == -1)`	`if (tmp->nbytes == -1)`
`  {`	`  {`
`    error (0, errno, ”%s”, filename);`	`    error (0, errno, ”%s”, filename);`
`    errors = 1;`	`    errors = 1;`
`    free ((char *) tmp);`	`    free ((char *) tmp);`
`    goto free_lbuffers ;`	`    goto free_lbuffers ;`
`  }`	`  }`

Table 5.2: An example where backward slicing falls short.

reduction analysis that outputs the subset of the program that is relevant to a certain piece of code.

Slicing is often carried out as an iterative expansion. Initially, the slice constitutes the slicing criterion. In each iteration, edges are followed from the slice to vertices which are directly connected to the slice (i.e. connected through a single edge). Then, all (or some) of these vertices are added to the slice, and the next iteration considers the augmented slice. Recall that the edges in a PDG are directed. In *forward slicing*, only those edges are followed which leave from the slice. Forward slicing collects program pieces which could be affected by what happens at the slicing criterion. In *backward slicing*, only those edges are followed which arrive in the slice. Backward slicing collects program pieces which could affect what happens at the slicing criterion.

Clone Detection Using Traditional PDG's

The clone detection technique proposed by Komondoor and Horwitz [KH01] uses PDG's as above-described to find clones in source code. Komondoor and Jorwitz' algorithm requires two PDG's as input. Assume the provided PDG's are PDG_1 and PDG_2. The algorithm then looks for any pair of subgraphs (PDG_{sub1}, PDG_{sub2}) for which:

- $PDG_{sub1} \subset PDG_1$;
- $PDG_{sub2} \subset PDG_2$;
- $PDG_{sub1} \wedge PDG_{sub2}$ are isomorphic.

The idea is that (PDG_{sub1}, PDG_{sub2}) is in fact a pair of clones, i.e. *a clone pair*. The deployed sense of isomorphism is based on an equivalence relation between the vertices. Komondoor and Jorwitz' algorithm consists out of three main steps:

1. **Find clone pairs** First, the vertices of PDG_1 and PDG_2 are partitioned into equivalence classes. This is done based on the syntactic structure of the vertices. (Recall that the vertices are program pieces.) Thus, variable names and literal values are ignored in the equivalence. Vertices in the same equivalence class are called *matching* vertices. For each pair of matching vertices $(r_1, r_2) \in PDG_1 \times PDG_2$, two isomorphic subgraphs

are sought, such that one contains r_1 and the other contains r_2. The two subgraphs are then reported as one clone pair.

To find the isomorphic subgraphs, the algorithm performs backward slicing in both PDG_1 and PDG_2. It is performed in iterative expansion, as described above, and concurrently. Starting from r_1 and r_2 (and hence from two singleton slices), the algorithm slices backwards along any edge, to the direct predecessors of r_1 and r_2. If any two predecessors $(p_1, p_2) \in PDG_1 \times PDG_2$ match, they are added to their respective slices. Slices are extended in this manner until no further growth is possible. The two resulting slices are the isomorphic subgraphs.

2. **Remove subsumed clones** A clone pair (S_1^*, S_2^*) subsumes another clone pair (S_1, S_2), if $S_1 \subset S_1^*$ and $S_2 \subset S_2^*$. There is no need for the tool to report subsumed clones, and hence these pairs are filtered from the results of the previous step.

3. **Combine pairs of clones into larger groups** In some cases, clone pairs can be combined through transitive closure, eg. (S_1, S_2), (S_1, S_3) and (S_2, S_3) would be combined into the clone group (S_1, S_2, S_3).

In the end the resulting clone groups are presented to the user as clones. However, Komondoor and Jorwitz identified two complications in this basic algorithm. The first problem is that of clones that encompass the bodies of conditionals and loops, but can't be detected as a whole using backward slicing from a particular pair of vertices. This is illustrated in Table 5.2, where the cloning manifests itself clearly, but no backward slicing can group the statements in the bodies. To remedy this, forward slicing from control predicates was introduced. Also a mechanism was needed to prevent clones from crossing loops, i.e. to prevent them from comprising out of statements before and after a loop predicate, but not include the predicate itself. This loop-crossing problem otherwise complicates refactoring of clones into new procedures.

Clone Detection Using Fine-Grained PDG's

The clone detection technique developed by Krinke [Kri01] likewise uses PDG's, but operates on fine-grained graphs. A fine-grained PDG contains not only assignment statements and control predicates, but adapts the statement and expression nodes from the abstract syntax tree (AST) of the code to be modeled. There are also new types of edges. Next to the traditional control and data dependence edges, the following can hold:

- **an immediate (control) dependence edge** $a \to b$, if expression b is evaluated before expression a;
- **a value (data) dependence edge** $a \to b$, if the value computed at expression a is needed at expression b;

- **A reference (data) dependence edge** $a \rightarrow b$, if the value computed at expression b is stored into a variable in statement a.

Given these extensions to the PDG concept, the isomorphic subgraphs within two PDG's could be reported as clones (which is the approach followed by Komondoor and Jorwitz, as discussed above). However, Krinke's technique focusses on similarity rather than on isomorphism. As such, it is flexible enough to detect broader pairs of clones than just completely identical ones. Krinke defines similarity as:

> Two graphs G and $G^{'}$ are similar, if for particular vertices $v \in G$ and $v^{'} \in G^{'}$, the following holds:
>
> - For every path
>
> $$v_0, e_1, v_1, e_2, v_2, ..., e_n, v_n$$
>
> in G, there exists a path
>
> $$v_0^{'}, e_1^{'}, v_1^{'}, e_2^{'}, v_2^{'}, ..., e_n^{'}, v_n^{'}$$
>
> in $G^{'}$, and the attributes of the vertices and edges are identical if the two paths are mapped on each other. (Where $v_i^{(')}$ are vertices and $e_i^{(')}$ edges of their respective graphs.)
> - $v_0 = v$ and $v_0^{'} = v^{'}$. As such, v and $v^{'}$ are called the *starting vertices* of their respective graphs.

The attribute of an edge is its type (e.g. value dependence edge). The attributes of a vertex are:

- its general *class* (e.g. statement, expression, procedure call);
- its more specific *operator* (e.g. binary expression, constant);
- its most specific *value* (e.g. "+", a constant value, an identifier name).

Given this definition of similarity, Krinke proposes an inductive algorithm that takes as input two fine-grained PDG's and a chosen vertex in each. The algorithm outputs the two maximal-size similar subgraphs with the given vertices as their starting vertices. Due to performance reasons it is undesirable to run this algorithm with each possible pair of vertices as the starting vertices. Therefore, Krinke leaves it up to the user to make an application-specific selection of starting vertices. Finally, the constructed subgraphs are weighted according to the amount of data dependence edges they contain. Graphs with more such edges are regarded to be more valuable. This is done because structural information, which is what control dependence edges provide, can also be achieved through less expensive techniques (Section 5.2.3). In weighting the technique thus focusses more on semantical information, as provided by data dependence edges.

5.2.3 AST-based Clone Detection

The approach followed by Baxter et al. [BYM$^+$98] considers near-clones, i.e. clones that are identical save some slight differences. The proposed technique works on the abstract syntax tree (AST) of the code. Similar subtrees are reported as (near-)clones. For performance reasons, the amount of comparisons between subtrees is limited through hashing. An inexpensive hash function partitions the collection of all subtrees into buckets. Only those subtrees in the same bucket need to be fully compared with one another. Subtrees in different buckets are discarded as potential (near-)clones. The hash function is chosen so that subtrees that only differ in their leaves end up in the same bucket. These leaves represent literals in the code. The full comparison measure, to find the (near-)clones in a bucket, is the following formula:

Similarity(subtree1, subtree2) $= \frac{2\times S}{(2\times S)+L+R}$
where:
S = number of shared nodes
L = number of different nodes in subtree1
R = number of different nodes in subtree2

Further extensions to this basic algorithm cope with the facts that: (1) in many AST representations, sequences of statements are not strictly subtrees; (2) subsumed clones must not be reported; and (3) more general near-clones can be found by deploying the full comparison measure on the parents of subtrees already identified as (near-)clones.

5.2.4 Token-based Clone Detection

A key property of the CCFinder [KKI02] clone detection technique is that it deploys a reader on the source code, to transform the raw sequence of characters into a sequence of tokens. CCFinder thus works on an intermediate level between the original string representation of the code and its AST abstraction. The sequence of tokens is stripped of white spaces, comments and line breaks. Furthermore, a series of transformation rules is applied to the token sequence. In the context of the Java language, one such rule extends the token representation of a message send that has no explicit receiver: because the semantics of Java dictate that the run-time receiver will be the sender of the message, a token can be introduced representing that sender as the receiver. Another rule simply throws away the initialization lists with which Java Arrays can be initialized with values (e.g. `new int[] {1, 2, 3}`). Yet another rule throws away package names preceding class names. In general, the transformation rules can be classified in three categories, namely those that bring the code to:

1. a more uniform representation, to ease detection of clones that are semantically identical but expressed differently;

type	near-clones	non-contiguous	intertwined	out-of-order
PDG	+	+	+	+
TOKEN	+	-	-	-
AST	+	-	-	-

Table 5.3: Clone detection approaches and the obstacles they can overcome.

2. a more structural representation, as the technique is more interested in structural commonalities;
3. a more context-free representation, as clones can even occur over different applications (the interesting example of plagiarism is brought up).

After this transformation, identifiers in the code related to types, variables and constants are each replaced with the same special token, as to further lay the focus on structurally similar clones. In the end a suffix-tree matching algorithm [Gus97] is deployed to find the common subsequences in a pair of token sequences.

5.2.5 Discussion

We discriminate among clone detection approaches by considering the obstacles they can overcome. We identify the following such obstacles:

- A **non-contiguous** clone encloses a code fragment, but does not include every statement or expression within that fragment. There are gaps in the clone.
- A set of **intertwined** clones all enclose the same code fragment. It is to be understood that the intertwined clones do not in fact form one "clone group", i.e. a group of clones that are similar to *each other*. Rather, the intertwined clones belong to different clone groups. The most prototypical example of intertwined clones is a code fragment where the even statements belong to one clone group, and the uneven statements belong to another.
- A set of clones engage in an **out-of-order** relationship, if they contain the same statements (or expressions), but the ordering of the statements (expressions) differs over the different clones. Here, the clones do form one clone group.
- **Near-clones** belong to the same clone group but differ slightly, e.g. the used variables names and literals are different.

Most, if not all clone detection techniques consider near-clones. Typically this is done by withholding variable names and literals from the analysis, even though this strategy consequently ignores all data flows. Some techniques do not explicitly consider non-contiguous clones in their analysis. Often a technique tries to remedy this shortcoming via a post-analysis step in which clones are merged when they sit near each other in the source code. However, this is done regardless of whether the merged clones belong together logically. Table 5.3 gives an overview of which category of techniques (that we have discussed) is believed to overcome which obstacle. It appears the PDG-based approach surpasses all of its alternatives. However, at the current time all PDG-based clone detection techniques are comparatively very expensive and slow.

5.3 Automated Identification of Aspect Candidates

5.3.1 Detecting Unique Methods

The work of Gybels and Kellens [GK05] is based on the observation that in pre-AOP days, developers would often cleanly modularize the implementation of a crosscutting concern in a kind of "pseudo aspect". Such a pseudo aspect is in reality a regular class in object-orientation. Its actual use must thus be manually woven into the system, which leads to crosscutting. For example, given one method that is the entry point to such a pseudo aspect, the manual weaving strategy results in the scattering of multiple calls to this one method, where the calls are tangled with other concerns. Techniques based on clone detection have a hard time recognizing this recurring single-line pattern. Hence, a heuristic is at order that applies to a single method. Gybels and Kellens propose:

A unique method is:
"A method without a return value which implements a message implemented by no other method."

It is asked that a unique method has no return value, because it is not sensible to move method calls that have return values to advice; these calls are clearly part of the base functionality of the code. The unique-message condition stems from the prototypical crosscutting examples such as logging, which are often implemented through uniquely-named methods. The approach of Gybels and Kellens consists out of determining the unique methods in the legacy system. After filtering out getters and setters, as well as those methods that are called rarely (as an approximation of the degree of scattering), the remaining unique methods are left to manual inspection by the user. In the accompanying experiment, Gybels and Kellens did find typical aspects such as tracing.

5.3.2 Fan-in Analysis

Marin et al. [MvDM04] propose the use of the fan-in heuristic for aspect mining. The fan-in of a method is defined as the number of distinct method bodies that can invoke that method. Because of polymorphism, a method call to m contributes to the fan-in of: m itself, all methods refining m, and those refined by m. Fan-in analysis follows three consecutive steps:

1. Fan-in is computed for each method in the legacy system.
2. Three filters are applied. They serve to disqualify:
   - **Methods with a fan-in below a certain threshold.** Marin et al. use a treshold of 10, which tends to keep around 5% of all methods.
   - **Getters and setters.** In a first iteration these methods are detected through their signatures (names starting with get/set), in a second iteration through their implementations. Getters and setters on static fields are not eliminated because they can come in handy in the detection of the Singleton design pattern.
   - **Utility methods.** E.g. collection manipulation methods, type conversion methods (e.g. `toString()`).
3. The remaining methods are manually inspected for aspect candidates.

On one hand, this technique is bound to not detect some aspect candidates, as it favours those with a large footprint. It can for example miss a crosscutting design pattern, if that pattern is deployed in one isolated instance in the source code. On the other hand, the percentage of false positives in fan-in analysis is generally low. Statistical results, achieved from three case studies, show that more than one third of methods with high fan-in are seeds that lead to aspects. After filtering, almost only aspect seeds remain. There are three prototypical situations in which a high fan-in method (suppose m) can be linked to a crosscutting concern (suppose c).

1. m is a key element of c, e.g. it is the logging method called all over the software system.
2. c is spread over the system and relies on particular functionality; m is part of this functionality. Thus, c is implemented at the calling sites of m.
3. m is easily identifiable as belonging to a design pattern with known crosscutting nature, e.g. it is the `register` method of an Observer instance (see Section 6.3.1 for the crosscutting behaviour of Observer).

5.3.3 Natural Language Processing on Source Code

The technique of Shepherd et al. [SPT05] analyzes words found in the source code of the legacy system. The words that are analyzed are class fields, class names, method names, and words in comments of the source code. Method names are partitioned according to the occurrences of capital letters. Words that do not provide much information, like "with" and "the", are filtered away.

Rather than analyzing the words syntactically, the technique follows a semantical approach. To this end, Shepherd et al. have sought inspiration in the field of natural language processing. More precisely, their technique deploys lexical chaining. This means that the technique processes the words one by one. Each word is added to the most closely related chain of words. However, if no chain is close enough, a word remains a singleton chain on its own. In this manner, each resulting chain is a distinct group of closely related words. Lexical chaining relies on a distance measure between the words, and additionally a distance treshold to determine when a chain is close enough for a word to be added to it.

The semantical nature of the approach followed by Shepherd et al. stems from the used distance measure. Based on the WordNet database [Bud01], their technique derives a degree of semantical similarity from any two given words. For instance, “money” and “bills” are very similar, and so are “auction” and “sold”. Furthermore, word-sense disambiguation [IV98] is deployed, which derives additional meaning of words from the context in which they appear. In this manner, nouns can for instance be discriminated from verbs.

The idea is that each resulting chain represents a concern in the code. Shepherd et al. have identified three properties of a chain that are desirable in the light of aspect mining:

1. the concern represented by the chain is not already encapsulated in a class;
2. the chain contains more than one member (a singleton chain is likely a less significant concern, and already known to the user since it is expressed uniformly throughout the code - chains are sets, with no duplicates);
3. the members of the chain crosscut several classes in the code.

5.3.4 Analyzing Recurring Patterns of Execution Traces

Dynamic Analysis The Dynamit tool [BK04], developed by Breu and Krinke, detects recurring patterns in the execution trace of a Java program. To this end, the program under investigation is run. The entry, as well as exit, of any instance method is recorded. The execution trace is formed as a chronological sequence of these events, as illustrated in Table 5.4, where the entry of a method is designated by “{” and the exit by “}”.

Dynamit can detect four types of patterns in the execution trace. More specifically, two methods a and b can participate in the following four execution relations:

- **outside-before** ($a \rightarrow b$)
  if b is entered immediately after a is exited (immediacy is in the sense that no event, entry nor exit, takes place in between);
- **outside-after** ($b \leftarrow a$)
  if $b \rightarrow a$;
- **inside-first** ($a \in_{first} b$)
  if a is entered immediately after b is entered;

```
B() {
  C() {
    G() {}
    H() {}
  }
}
A() {}
B() {
  C() {}
}
```

Table 5.4: Example of an execution trace in Dynamit.

- **inside-last** ($a \in_{last} b$)
  if b is exited immediately after a is exited.

The set of all relations $a \diamond b$ that can be detected in the execution trace, with $\diamond \in \{\rightarrow, \leftarrow, \in_{first}, \in_{last}\}$, is labeled $S^{\diamond}$. However, the detection of these relations is not sufficient to conclude crosscutting. Therefore, Breu and Krinke impose two additional conditions. They are:

- **Uniformity** A relation $(a \diamond b) \in S^{\diamond}$ where $\diamond \in \{\rightarrow, \leftarrow, \in_{first}, \in_{last}\}$ is called uniform if $\forall (c \diamond b) \in S^{\diamond} : c = a$. An outside-before relation $a \rightarrow b$ is consequently uniform if each execution of b is preceded by an execution of a. An inside-first relation $a \in_{first} b$ is uniform if b never executes any other method than a as the first method.

- **Crosscutting** A relation $(a \diamond b) \in S^{\diamond}$ where $\diamond \in \{\rightarrow, \leftarrow, \in_{first}, \in_{last}\}$ is called crosscutting if $\exists (a \diamond c) \in S^{\diamond} : c \neq b$. That is, if the relation occurs in more than one calling context. The calling context of a relation is its second element.

The crosscutting condition is easy to motivate through the use of an example. Consider the following excerpt of code:

```
private void setMatrixSize(int n) {
  matrix = new int[n][n];
}
public void buildMatrix(int n){
  setMatrixSize(n);
  initMatrix ();
}
```

```
void Runner.doSth(A) {
  void A.a() {
  }
  void C1.c() {
  }
}
void Runner.doSth(A) {
  void A.a() {
  }
  void C2.c() {
  }
}
```

Table 5.5: A trace problematic to purely dynamic analysis of recurring patterns.

The following execution relations could occur in an execution trace of the above excerpt of code:

> `setMatrixSize` $\in_{\text{first}}$ `buildMatrix`,
> `setMatrixSize` $\rightarrow$ `initMatrix`,
> `initMatrix` $\leftarrow$ `setMatrixSize`.

Clearly, this example shows no crosscutting, and the crosscutting condition is thus at order to avoid patterns being falsely identified as aspect candidates. Ultimately, the Dynamit tool mines for execution relations that are uniform and crosscutting. It then presents them to the user as aspect candidates.

Static and Dynamic analysis Due to the dynamic nature of the analysis, Dynamit suffers from a particular problem. Java has the property of dynamic binding. Precisely which method is executed at run-time depends not on the static type of the variable holding the receiver, but on the run-time type of the receiver. This mechanism is typically responsible for a number of false positives in the results of Dynamit. In order to make this clear, one only has to consider the execution trace in Table 5.5. Consider the following execution relations which hold in that trace:

`A.a()` $\rightarrow$ `C1.c()` and `A.a()` $\rightarrow$ `C2.c()`

These relations appear uniformly, as every `C1.c()` (respectively `C2.c()`) is preceded by `A.a()`. The crosscutting condition also holds on these relations, as `A.a()` appears in two different calling contexts.

```
void Runner.doSth(A) {
  void A.a() {
  }
  void A.c() {
  }
}
void Runner.doSth(A) {
  void A.a() {
  }
  void A.c() {
  }
}
```

Table 5.6: An execution trace with static information.

However, while these are two separate execution relations, they are in reality both instances of the same pattern, namely the two-part sequence of method calls in the following method body:

```
static void doSth(A a) {
  a.a();
  a.c();
}
```

Because this pattern appears only once in the code, the two execution relations have thus been falsely recognised as aspect candidates. A simple technical solution to remedy this problem has been integrated into Dynamit [Bre04]. Since the tool uses AspectJ to trace the execution of a program, it was sufficient to have the tracing aspect use `call` pointcuts. These capture the static method being called, as opposed to `execute` pointcuts that apply to the result of dynamic binding. Ultimately the trace in Table 5.6 results, where the run-time types have been replaced by the original static declarations. As a consequence, the false positives described above are no longer reported by Dynamit, as the crosscutting condition no longer holds.

5.3.5 Clustering of Similar Method Names

Shepherd and Pollock [SP05b] focus on a particular type of crosscutting concern in Java systems: concerns which are, or should be, implemented as interfaces. They argue that these concerns are often inconsistently implemented across classes – even in the case of an explicit interface, as it does not bind the concern physically. The technique proposed by Shepherd and Pollock combines the concept of (semi-)automated aspect mining with the idea of a specialized view.

The automated aspect mining part is implemented through cluster analysis. The objects in the analysis are source code methods. The distance measure between the methods (objects) considers their names. The idea is that a resulting cluster constitutes methods that have similar names. Shepherd and Pollock work from the hypothesis that crosscutting concerns are often implemented through naming patterns. More specifically, the distance measure is as follows:

$$d(m_1, m_2) = \frac{1}{commonSubstringLength(m_1.name, m_2.name)}$$

The objects and distance measure are fed to an AHC algorithm, as discussed in Section 4.5. Whenever two clusters are merged in the AHC process, the composite cluster is labeled with the common substring of its children. In this manner, the above-described distance measure can be applied to composite clusters, rather than leaving the distance calculation between composites up to the AHC algorithm of choice. A distance treshold is installed that stops the AHC process if no two clusters meet it.

The specialized view consists out of three panes. The *cluster pane* displays all clusters found in the legacy system. Once a cluster has been selected in this pane, the *crosscutting pane* displays (the bodies of) its member methods. When a particular method is selected, its context (i.e. class file) is displayed in the *editor pane*. When a cluster was detected in the cluster analysis, the specialized view allows the user to easily, albeit manually, explore code related to that cluster, and detect:

- scattering and tangling in terms of the methods in the cluster being implemented over different classes;
- inconsistently applied design rules, scattering, tangling, duplication, throughout the methods in the cluster.

5.3.6 Clustering Based on Method Invocations

Lili He et al. [LHH05] combine cluster analysis with association rule mining to uncover potential aspects. In the cluster analysis, the objects and attributes are the source code methods in the legacy system. An attribute value is 1, if the attribute (method) calls the object (method); otherwise the value is 0. The Jaccard distance measure is deployed to perform the cluster analysis. (See Section 4.2.1 for an explanation of Jaccard.) Thus, two methods (objects) are similar based on the degree to which they appear together in method bodies. A cluster is meant to consist out of methods that are called together.[1]

An Example As an example of the cluster analysis performed by Lili He et al., consider the following three methods:

[1]The paper of Lili He et al. makes no mention of how their technique handles polymorphism, the principle through which it is impossible to determine beforehand which method is called as the result of a message send.

- $method_1()$, which calls $\{write(), flush(), alert()\}$
- $method_2()$, which calls $\{write(), flush(), open()\}$
- $method_3()$, which calls $\{write(), flush(), begin()\}$

Further assume that these three methods are the only methods in the legacy system that call any methods. Then, if we assume the following ordering of the methods in the legacy system (which are thus the attributes in cluster analysis):

$(method_1(), method_2(), method_3(), write(), flush(), alert(), open(), begin())$

we can represent the object corresponding to $write()$ as follows:

$(1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)$

while $flush()$ becomes:

$(1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)$

and $alert()$ becomes:

$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$.

We can then observe the distance between these objects (methods), through the Jaccard measure:

$$Jaccard(write(), flush()) = 0$$

$$Jaccard(write(), alert()) = \frac{2}{3}$$

Note that the deployed distance does not involve any notion of recurrence. When two methods are called once each, but both are called in the same method body, they have similarity 1 (i.e. perfect similarity). This is not a *recurring* pattern. Thus, a resulting cluster consists out of methods that are called together, often or not.

The cluster analysis is carried out using CHAMELEON (Section 4.5.4 for an outline of this algorithm). After the initial cluster analysis, a filtering step is applied where a cluster is discriminated against if a method it contains is not called often enough according to a certain treshold. This is done because only those clusters of which the methods are *frequently* called together are deemed valuable. The general idea is that the recurring pattern represented by a cluster can be extracted into advice.

The second leg of the technique uses association rule mining (ARM) [HK00] to extract information with which to model the pointcuts for the found aspect candidates. ARM is a database mining technique. It deals with a set of items I, and a set of transactions T, where each transaction is a subset of I. In the context of a shopping example, I could be the set of things one can buy, while a transaction could be the set of items resulting from one shopping trip. Given I and T, ARM aims to provide rules such as:

bread $\Rightarrow$ milk | 80%

`first()` $\Rightarrow$ `conn()` | 100%
`close()` $\Rightarrow$ `end()` | 100%

Table 5.7: Example association rules.

The above rule means that 80% of the time, when a person has bought bread, he or she has also bought milk. The given percentage is called the confidence factor. The goal of ARM is to automatically mine the database for rules with a high such factor.

Lili He et al. use ARM as follows: the items are the methods in the legacy system that remain after the above-described filtering, plus two special designators `first()` and `end()`. Conceptually, each method calls `first()` before its body, and `end()` after. For each method m in the legacy system, one transaction is formed of which the elements are the methods called by m. Hence, the ARM algorithm provides rules such as in Table 5.7. These indicate confidences with which the left-hand method is accompanied by the right-hand method in different method bodies. Some rules $A \Rightarrow B$ must be filtered away, namely those where B isn't immediately called after A in the different method bodies. The remaining rules guide the refactoring. An example: two methods `conn()` and `close()` have been identified as a cluster { `conn(), close()` } of methods that are called together. In case the association mining step results in the two rules of Table 5.7, then we can refactor `conn()` and `close()` into one around advice.

5.3.7 Formal Concept Analysis of Execution Traces

Formal Concept Analysis

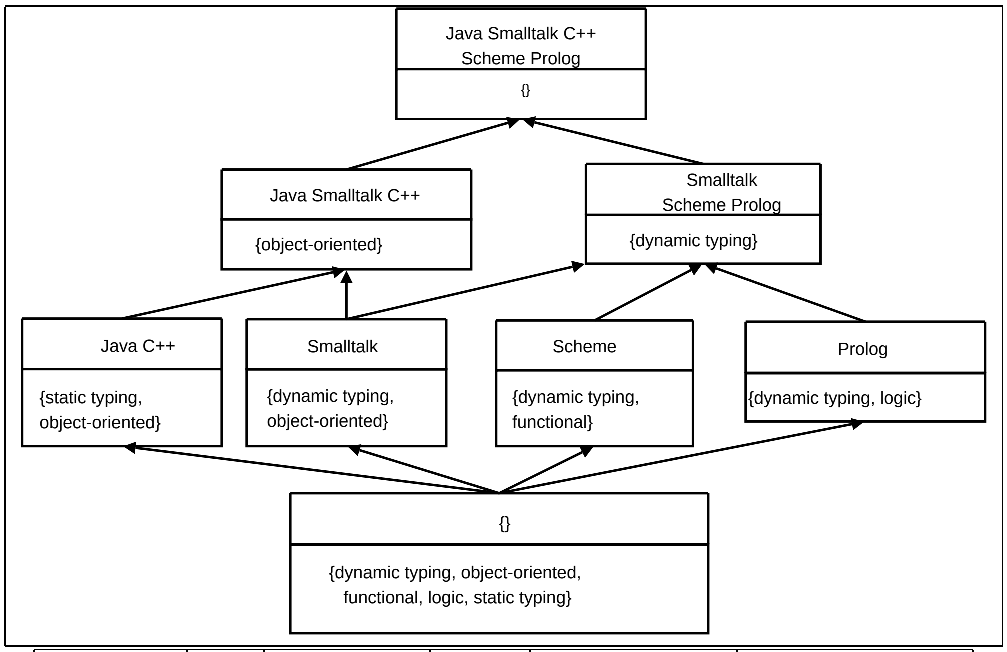
Formal concept analysis (FCA) [GW97] belongs to the domain of lattice theory. It serves to form maximal groups of objects that have common attributes. More specifically, a context (O, A, R) consists out of the set O of objects, the set A of attributes, and the binary relation $R = O \times A$ to indicate which object has which attributes. A concept is then defined as a pair of sets $(X \subset O, Y \subset A)$ such that:

$$X = \{o \in O | \forall a \in Y : (o, a) \in R\}$$

$$Y := \{a \in A | \forall o \in X : (o, a) \in R\}$$

The objects X thus have the attributes Y in common, such that these objects have no common attributes other than Y. Nor is there any object outside of X that has Y as its attributes. X is called the *extent* of the concept, while Y is the *intent*.

Given all concepts that can be computed from a context, a partial order between the concepts can be defined through the containment relationship on their extents. This gives rise to a lattice. Figure 5.1 illustrates such a lattice. It is


Java Smalltalk C++
Scheme Prolog
{}
Java Smalltalk C++
{object-oriented}
Smalltalk
Scheme Prolog
{dynamic typing}
Java C++
{static typing, object-oriented}
Smalltalk
{dynamic typing, object-oriented}
Scheme
{dynamic typing, functional}
Prolog
{dynamic typing, logic}
{}
{dynamic typing, object-oriented, functional, logic, static typing}

PL	OO	Functional	Logic	Static Typing	Dynamic typing
Java	X	-	-	X	-
Smalltalk	X	-	-	-	X
C++	X	-	-	X	-
Scheme	-	X	-	-	X
Prolog	-	-	X	-	X

Figure 5.1: An example lattice resulting from a Formal Concept Analysis.

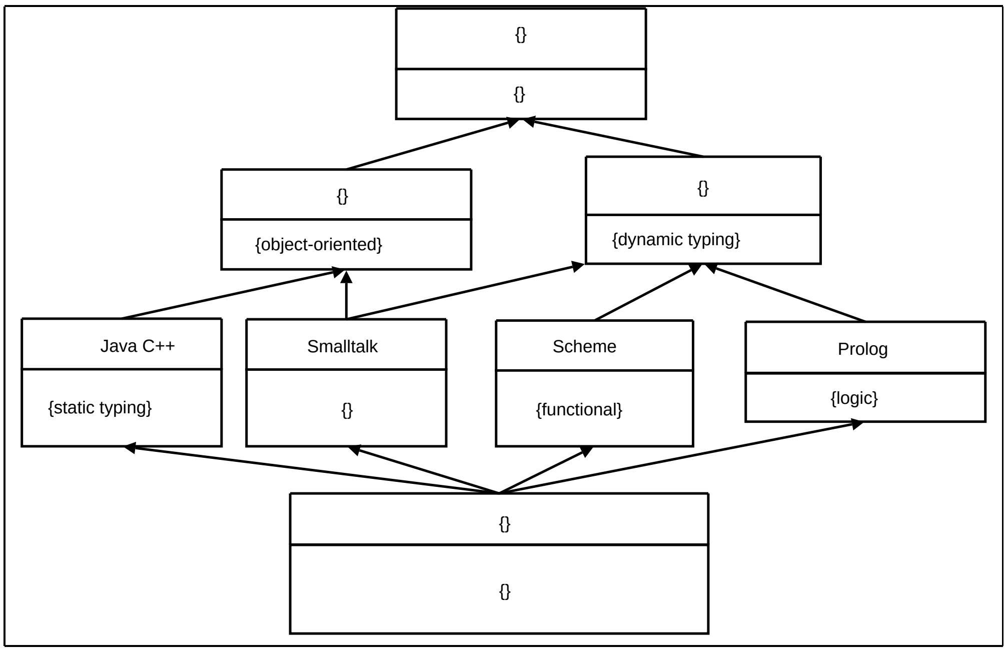


Figure 5.2: The sparse version of the lattice in Figure 5.1

based on programming languages and the properties such languages can have. The languages are the objects and their properties the attributes. Note that the same partial order can be defined based on the containment relationship between intents.

Concepts that sit higher in the lattice can be regarded as more general notions, with more objects and less applicable attributes than the more specific concepts lower in the lattice. A more compact and readable representation of an FCA lattice can be constructed through a sparser labeling of the concepts, as follows:

> A concept is labeled with:
>
> an attribute, if it is the most general concept carrying that attribute;
>
> an object, if it is the most specific concept carrying that object.

The sparse rendition of the lattice in Figure 5.1 is given in Figure 5.2. Given this representation, the extent of a concept can be read off the sparse lattice as the union of the objects with which more general concepts, plus the concept itself, are labeled. The manner in which intents can be read is analogous.

Formal Concept Analysis of Execution Traces Tonella and Ceccato [TC04] provide a dynamic analysis. Their technique considers a set of use cases, and generates an execution trace for each one. Each use case serves as an object in the subsequent FCA, while the attributes of an object (use case) are the methods in the corresponding execution trace. Given the resulting sparse concept lattice, the technique then attempts to associate each use case U with a concept that is specific of it, i.e. a concept whose extent only contains U. Note that this is not always possible for each use case. A use-case specific concept is then considered an aspect candidate, if the following crosscutting conditions hold:

- **Scattering** The attributes (methods) of a use-case specific concept belong to more than one class.

  The authors argue that this first condition is typically not sufficient to observe crosscutting, because a concern can be implemented over different modules as to simply delegate functionality, which is quite ordinary. Complementary to the first condition, a second phenomenon is to be observed:

- **Tangling** Different methods from the same class belong to more than one use-case specific concept.

Tonella and Ceccato's technique is semi-automated. It is still up to the aspect miner to define the use cases, and to instrument the legacy system so its execution can be traced. However, the construction of the lattice is automated, as well as the verification of the two above-described crosscutting conditions.

Methods in the Insertion execution trace.
`BinaryTree.BinaryTree()`
`BinaryTree.insert(BinaryTreeNode)`
`BinaryTreeNode.insert(BinaryTreeNode)`
`BinaryTreeNode.BinaryTreeNode(Comparable)`
Methods in the Search execution trace.
`BinaryTree.BinaryTree()`
`BinaryTree.search(Comparable)`
`BinaryTreeNode.search(Comparable)`

Table 5.8: The execution traces for the **Search** and **Insertion** use cases.

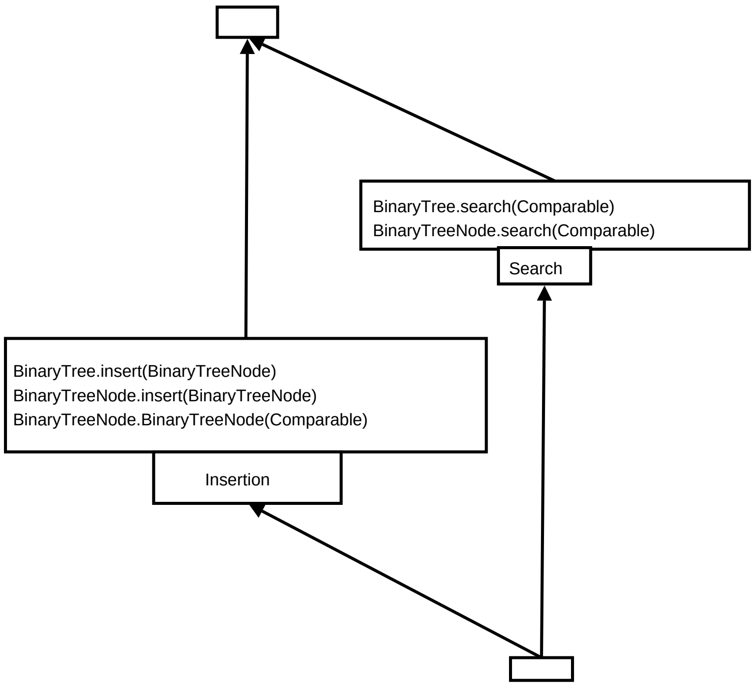

BinaryTree.search(Comparable)
BinaryTreeNode.search(Comparable)
Search
BinaryTree.insert(BinaryTreeNode)
BinaryTreeNode.insert(BinaryTreeNode)
BinaryTreeNode.BinaryTreeNode(Comparable)
Insertion

Figure 5.3: The resulting concept lattice for Table 5.8

As usual, manual post-processing of the mining results is required from the aspect miner.

As an example, consider a binary search tree application. We are interested in two use cases: insertion of a node, and the search for information in the tree. Consequently, we require two execution traces. They are illustrated in Table 5.8. Then, the FCA algorithm is run, resulting in the sparse lattice of Figure 5.3. Immediately it is clear that a degree of crosscutting is detected by the discussed technique: the concepts of insertion and search are both labeled with methods belonging to more than one class. Additionally, the BinaryTree and BinaryTreeNode classes are scattered over more than one concept.

5.3.8 Formal Concept Analysis of Identifiers

Tourwé and Mens [TM04] propose the use of FCA using lexical information from the source code. (FCA was discussed in Section 5.3.7.) Classes and methods are taken as the objects. Their names are split into substrings, according to the location of capital letters. For example, “QuotedCodeConstant” becomes “Quoted”, “Code” and “Constant”. Substrings with little meaning, like “with” and “in”, are ignored. Any remaining substring becomes an attribute to any object (class or method) whose name it appears in.

Tourwé and Mens report on an initial experiment in which they obtained 1212 concepts from 1446 objects and 516 attributes. Obviously, automatic filtering of concepts is at order in this technique. Therefore, the technique ignores the top concept in the lattice, if it has no attributes. Likewise, the bottom concept is ignored if it is void of any objects. In the next step of filtering, a concept is discriminated against if it contains only a single object. This is done because the singleton behaviour of the concept means that the contained object displays no relation to other objects, according to the chosen attributes.

The final experiment was carried out on the Smalltalk program SOUL as the legacy system. A number of interesting phenomena were detected:

- **Accessor Methods** In Smalltalk, a naming convention is applicable to methods that serve to reference or set an instance variable: they are simply named as the variable they apply to. In several instances a concept was found that groups such methods, bound by the name of the variable as the attribute.
- **Polymorphism** Classes implement polymorphism through naming their methods identically. Naturally these identically-named methods were assembled in the same concept.
- **Design Patterns** As an example of the detection of a design pattern, an instance of Visitor was found (see Section 6.4.5 for a discussion of this pattern). This instance was detected through (1) polymorphism (there is a hierarchy of Visitors in SOUL), and (2) the `visit` methods that are scattered over the data structure to be traversed.

- **Code Duplication** Not surprisingly, methods with similar names often have similar implementations.
- **Class-related Behaviour** Concepts were found that group one class together with methods that relate to that class. This type of concept can emerge from the Visitor, Builder and Abstract Factory design patterns: because the behaviour of a pattern instance depends on a choice of class, names of classes are embedded in the pattern instance.

Obviously, crosscutting is not inherent to some of these phenomena. Nonetheless, crosscutting concerns were detected with this technique (Visitor is one example).

5.3.9 Aspect Mining Based on PDG's

The aspect mining technique of Shepherd et al. [SGP04] uses PDG's.[2] (Section 5.2.2 for an explanation of PDG's.) The technique is particular, in that it discriminates against those crosscutting concerns that are more difficult to refactor. Concretely, the technique mines for `before` advice. In doing so, it looks for patterns (clones) that occur at the beginning of method bodies.

Given a pair of methods, the proposed technique uses information from both their AST's and (traditional) PDG's. It traverses through control dependence edges only. These edges give shape to a tree structure rooted at the entry of the PDG (method). Starting from the entries of the two methods, their PDG's are traversed breadth-first in lockstep, collecting the maximal common subtree. Of course, this process relies on a concept of equality between nodes in the PDG's. It is defined as follows. Given two nodes, their AST's are walked through in lockstep; any node in one AST must match its corresponding node in the other, where two nodes match when:

- they are variables and they are both of the same type, one is a subtype of the other, or they both share a parent type besides Object (the global root object in Java);
- they are literals of the same type (e.g. boolean);
- they are neither literal or variable but structurally identical (e.g. "if" matches "if").

The maximal common subtrees between method PDG's are then considered clones. Two filters are applied to the resulting set of clones, they are:

[2]We discuss this technique separately from the other techniques that use PDG's, because it is an actual aspect mining technique, rather than a clone detection technique.

- **Similar-data-dependence** Given two PDG subtrees representing clones, this filter ensures that the underlying data dependencies are the same in each subtree. These edges are only considered as a post-processing step, because this manner of operating is more time-efficient than the technique of Brinke (Section 5.2.2) which has exploding running times.
- **Outside-data-dependence** This filter eliminates any clone that has data dependencies to, or from a node outside of the clone. This is motivated by the AspectJ restriction that variables inside `before` advice can't be referenced in the adviced method. Hence this filter focusses on concerns that are difficult to refactor.

Ultimately, the remaining clone pairs are presented to the user as refactoring candidates, after they have been coalesced where possible.

5.4 Summary

We have discussed the research field of aspect mining, which is concerned with the detection of aspect candidates in legacy systems. Due to the scale, complexity and poor documentation of legacy systems, tools have been proposed that aim to aide the developer in mining for aspects candidates. We have surveyed those aspect mining techniques that strive to automatically identify all aspect candidates in the legacy system. We have additionally surveyed clone detection techniques; those are not tailored for the automated identification of aspect candidates, but can be used for this purpose. We will compare the surveyed aspect mining techniques with our own; in Chapter 7, this is done in terms of analysis properties, while in Chapter 9 the aspect mining results are compared.

Chapter 6

JHotDraw

JHotDraw [JHo] is a relatively large-scale Java program that serves as a framework for drawing applications.[1] We have used JHotDraw in our aspect mining experiments. This chapter discusses JHotDraw and crosscutting concerns that are present in the system. Additionally, attention is given to the position of design patterns in AOP, because many of the crosscutting concerns in JHotDraw are in fact design patterns. The list of concerns that are documented in this chapter is later used in the evaluation and comparison of our aspect mining techniques.

6.1 Motivation for our Choice of JHotDraw

JHotDraw was first devised as an exercise in design patterns. One of its authors is in fact co-author of the ground-laying Gang-of-Four book [GHJ95]. As such, JHotDraw has the reputation that it maximally exploits the means of abstraction and composition offered by traditional object-oriented languages. Consequently, JHotDraw makes for a particularly interesting case study in the field of aspect mining. Any crosscutting concern uncovered within JHotDraw can to a greater extent shed light on where object-orientation falls short. Adding to the motivation behind our choice of JHotDraw, is the high quantity of papers that discuss the system, and use it as a benchmark for aspect mining.

6.2 Architecture of JHotDraw

Figure 6.1 gives an overview of the core architecture of JHotDraw. The most important classes and interfaces are:

- **Drawing, Figure and DrawingView** These three interfaces form the core of JHotDraw. A Drawing represents a two-dimensional space. It further contains Figures to which it can delegate commands. Examples

[1]The version of JHotDraw under investigation here is 5.4b1.

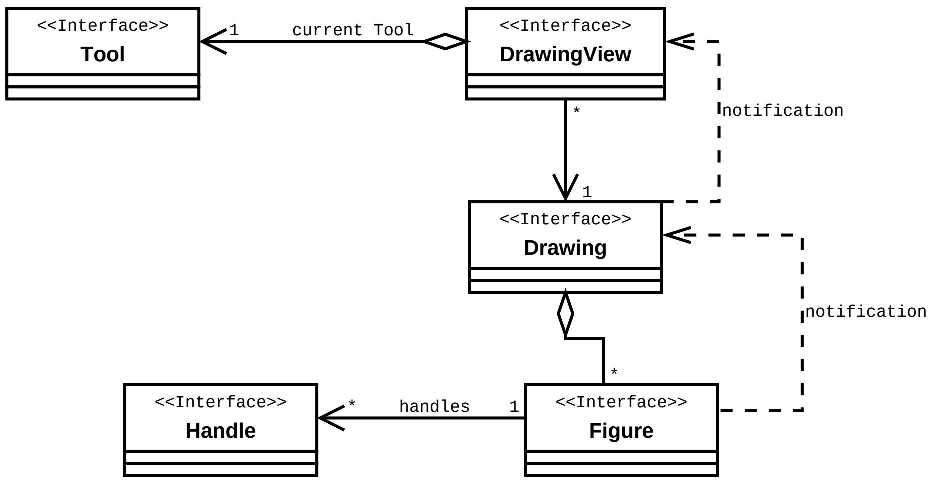


Figure 6.1: UML diagram of the core architecture of the JHotDraw framework.

of concrete Figures are: an ArrowTip, a TextFigure, an EllipseFigure. For one Drawing, multiple DrawingView can exist. Each DrawingView provides the screen a view of the Drawing it is dedicated to. The abstractive gap between Drawing and DrawingView maintains the separation between, respectively, JHotDraw's controller logic, and JHotDraw's underlying graphics system (which is Swing in the version under investigation).

- **Tool** A Tool defines a mode of a DrawingView. All input towards a DrawingView is delegated to its current Tool. Among the subtypes of Tool one finds classes with such clarifying names as DragNDropTool, ZoomTool, SelectionTool.

- **Handle** Direct manipulation of a Figure is done through a Handle that the Figure owns. A Handle knows its owning Figure and adapts it to a common interface.

- **Command (not on display in Figure 6.1)** A Command object encapsulates a request according to the Command design pattern. For further discussion regarding this design pattern within JHotDraw, we refer to Section 6.4.7.

6.3 Design Patterns and AOP

A design pattern is a generic description of an object-oriented solution. Design patterns have become a common vocabulary amongst software developers, improving mutual understanding and facilitating the maintenance of software systems. Seeing the quantity of design patterns that are present in JHotDraw, we deem it worthwhile to take a moment to discuss design patterns in the light of AOP.

	Locality	**Reusability**	**Composition Transparency**	**(Un)pluggability**
Adapter	yes	no	yes	yes
State	yes	no	n/a	yes
Decorator	yes	no	yes	yes
Visitor	yes	yes	yes	yes
Command	yes	yes	yes	yes
Singleton	yes	yes	n/a	yes
Observer	yes	yes	yes	yes

Table 6.1: AspectJ refactorings of design patterns and improvement in modularity properties.

While proven valuable, design patterns aren't infallible. Some elements of design patterns are hardly modularized and can be lost in the code, which is detrimental to the traceability of the patterns [CS95]. Adding or removing a design pattern from a software system is often an invasive modification [BFVY96]. Furthermore, even though design patterns draw strength from being reusable solutions, their concrete implementations typically don't allow for their reuse [CS95]. When design patterns are composed, this may lead to the pattern density problem, where the pattern implementations become entangled to the degree of deteriorating comprehensibility [DAAC05].

Consequently, researchers have sought to leverage the aforementioned problems, some through means of AOP. Denier et al. [DAAC05] mainly focus on the composition issue and propose aspect refactorings for Observer and Decorator (and their composition). The authoritative paper on "design patterns meet AOP" is due to Hanneman and Kiczales [HK02]. They make a case in favour of the AOP approach for design patterns, by showing that 17 of the 23 Gang-of-Four patterns enjoy modularity improvements in their AspectJ refactorings. Furthermore, Hanneman and Kiczales contribute a table, as (partly) seen in Table 6.1, where each row designates an aspect refactoring of a design pattern, while the columns show whether the refactorings enjoy improvement in four desirable modularity properties. These properties are: locality, reusability, composition transparency and (un)pluggability. In order to make these four concepts clear, we adapt the example given by Hanneman and Kiczales of the Observer design pattern.

6.3.1 Observer

A well-known pattern, Observer serves to define a dependency between objects: when a Subject changes state, all of its dependent Observers are notified. The pattern is illustrated in Figure 6.2. The Subject role is imposed on some classes, whilst the Observer role is imposed on dependent classes. The Subject maintains a collection of its dependent Observers. This collection is interfaced by a `register` method and an `unregister` method in the Subject, for adding and removing Observers. Completing the basic pattern is a `notify` method in

```
public abstract aspect ObserverProtocol {

    protected interface Subject { }
    protected interface Observer { }
    private WeakHashMap perSubjectObservers;

    protected List getObservers(Subject subject) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers = (List)perSubjectObservers.get(subject);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(subject, observers);
        }
        return observers;
    }

    public void addObserver(Subject subject,
        Observer observer) {
        getObservers(subject).add(observer);
    }

    public void removeObserver(Subject subject,
        Observer observer) {
        getObservers(subject).remove(observer);
    }

    protected abstract pointcut subjectChange(Subject s);

    after (Subject subject): subjectChange(subject) {
        Iterator iter = getObservers(subject).iterator ();
        while ( iter .hasNext() ) {
            updateObserver(subject, ((Observer)iter.next ()));
        }
    }

    protected abstract void updateObserver(Subject subject,
        Observer observer);
}
```

Table 6.2: AspectJ refactoring of the Observer design pattern.

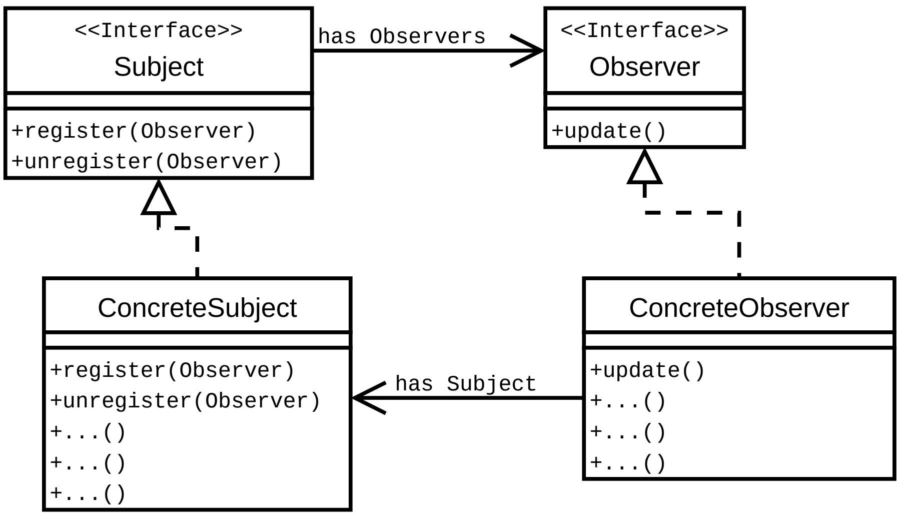


Figure 6.2: UML diagram of the Observer design pattern.

the Observer, so that the Subject can notify its dependents. (Often different `notify` methods exist for different types of events.)

The crosscutting behaviour is two-fold. Firstly, there is the matter of the superimposed roles, which are secondary roles to the participating classes. This results in tangling, where methods dedicated to different concerns are encapsulated in the same class. Also note the scattering of the Observer pattern over different classes. Secondly, in order for the Subject to be able to notify its Observers, the Subject's state-changing methods need to be crosscut with notification code.

The refactoring proposed by Hanneman and Kiczales is illustrated in Table 6.2. The aspect is abstract: any specific instance of the Observer pattern is to be realized through an aspect that derives from this abstract aspect. The abstract aspect defines two roles through means of two interfaces: one Observer interface, one Subject interface. Derived aspects are to impose these interfaces on the participating classes. An instance of a derived aspect keeps a data structure to map each Subject to its Observers. The `register` and `unregister` methods are part of the interface to the derived aspect. As such, the client needs to access the instance of the derived aspect. More specifically, in order to define the mapping between the Subjects and the Observers, the client is to use the following statements:

AnObserverAspect.aspectOf().addObserver(P, S)
AnObserverAspect.aspectOf().removeObserver(P, S)

Note that this violates the condition of obliviousness in AOP (Section 2.2). Pointcuts capture the join points at which the state of a Subject is changed. Advice attached to these pointcuts performs the actual notification of the Observers.

We can now revisit the four desirable modularity properties of aspect refactorings, and discuss them for the refactoring just described.

		Design patterns
Undo	WillChange/Changed	Observer
Drawing/Moving	Error Handling	Composite
Persistence	Serialization	Visitor
CheckDamage	Command Execute	Command
Cloning	Space	State
		Adapter
		Decorator
		Singleton

Table 6.3: The crosscutting concerns we have documented.

- **Locality (improved)** The Observer aspect and its concrete derivations are now the sole location for the implementation of the pattern. The participating classes have effectively been freed of the crosscutting roles and code, plus any coupling between them established by the pattern has vanished.
- **Reusability (improved)** The core logic of the Observer pattern is generically described in the abstract aspect. This logic can hence be reused by every instance of the Observer pattern: each instance is to derive one concrete aspect from the abstract aspect.
- **Composition Transparency (improved)** Because the implementation of a participating class has been decoupled from the Observer pattern, and each pattern instance sits in its own aspect, classes can participate in multiple Observer and Subject relationships without this composition being detrimental to comprehensibility.
- **(Un)pluggability** As Subjects and Observers have become oblivious to their roles, it is possible to switch to using or not using Observer in the software system.

6.4 Crosscutting Concerns in JHotDraw

Through literature study and manual investigation, we have compiled a list of crosscutting concerns in JHotDraw. We now discuss this list. The main focus in this discussion lies on the crosscutting nature of the different concerns. We work from the assumption that a design pattern constitutes a crosscutting concern, if Hanneman and Kiczales have determined AOP to improve the pattern in terms of its locality (Table 6.1). We further situate the presence of the crosscutting concerns – including design patterns – in JHotDraw.

In some cases, we detail the AspectJ refactoring of a concern, if this helps to illustrate where the software system can benefit from such an aspect refactoring. The AspectJ solutions for design patterns are due to Hanneman and Kiczales.

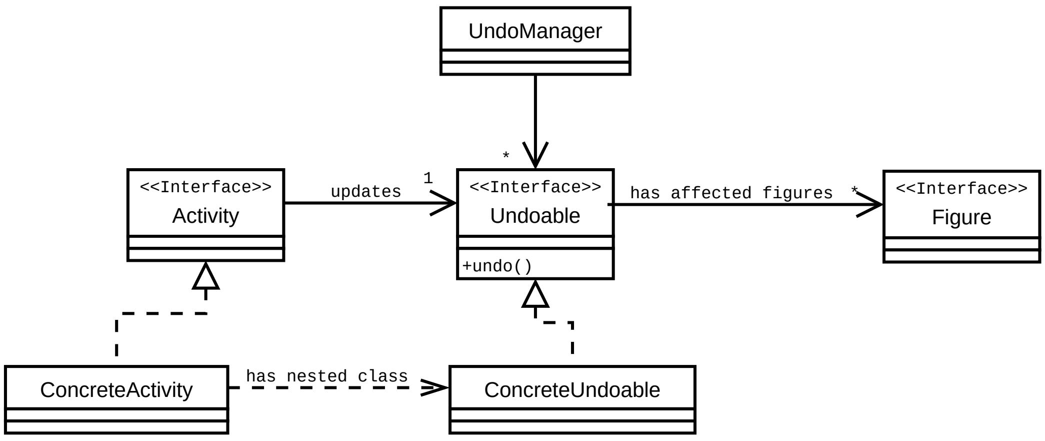


Figure 6.3: A simplified UML diagram of the Undo concern in JHotDraw.

However, while their AspectJ code is freely available[2], the authors often do not relate precisely how an aspect refactoring leads to improvement in the four modularity properties. That is to say, they provide little explanation as to the contents of Table 6.1. Therefore, we provide our own input on the matter.

We do not discuss each design pattern that we have found in JHotDraw. The patterns that we omit from the discussion *are* present in Table 6.3, which gives a global overview of *all* the crosscutting concerns that we have documented.

6.4.1 Undo

It is natural that the Undo concern, responsible for making actions in the drawing editor undoable, be a very pervasive concern throughout the software system. (Undo was discussed in [vDMM05].) A simplified representation of this concern is given in Figure 6.3. On one hand there is the Undoable interface. Its implementing classes are responsible for performing the actual undo operations. The instances of those classes are maintained on a stack by the UndoManager. It is the `undo` method in the Undoable interface that initiates the undo. On the other hand, there are the various Activities that can be undone. When it needs to be undoable, an Activity implements an Undoable as a static nested class. So that the undo may be done, an instance of the Undoable nested class keeps a (former) state of its enclosing Activity, and is notified each time the Activity's state changes. Furthermore, each Undoable instance maintains a list of affected figures that need be taken care of when an undo is initiated. (the simplification of this representation lays in the fact that there is no Activity interface as such; concrete activities are Tools, Handles and Commands, without a unifying interface).

[2]http://www.cs.ubc.ca/~jan/AODPs

The crosscutting behaviour manifests itself in the following three ways:

1. The nested Undoable, even though an encapsulation by nature, crosscuts its enclosing Activity.
2. The enclosing Activity has a factory method which returns an instance of its Undoable nested class.
3. Methods of the Activity that perform state changes are crosscut by code that serves to maintain the corresponding Undoable instance.

6.4.2 Error Handling

In Java's exception mechanism, an exception is either checked or unchecked. When a call is made to a method that could throw a checked exception (e.g. IOException), the calling method needs to do one of two things: (1) either declare "throws IOException" in its header to indicate that it could propagate a IOException, or (2) catch the exception in a `try-catch` statement. This policy is enforced by the compiler, which throws an error when the policy is violated. The use of unchecked exceptions does not impose such a policy: unchecked exceptions are caught in a `try-catch` statement at run-time, or crash the system at run-time. JHotDraw defines both custom checked and unchecked exceptions.

While error handling in itself is one of the prototypical crosscutting concerns, the policy described above adds to the crosscutting. When a method has no intent to produce or catch a particular checked exception, the method might still be forced to declare that it could propagate the exception. AspectJ allows to encapsulate `catch` clauses in aspects. Additionally, AspectJ knows a work-around to the crosscutting that originates from the policy regarding checked exceptions. This work-around is the "exception softening mechanism", in which checked exceptions are converted to unchecked exceptions.

6.4.3 Decorator

This design pattern serves to augment objects with new responsibilities. A Decorator object has the same interface as the object that it decorates and owns. Before redelegating an incoming request to this owned object, a Decorator object adds some functionality. One instance of this pattern in JHotDraw is quite prototypical (Figure 6.4). It is the case of adding a border to the graphical representation of a Figure. Concretely, the class BorderDecorator extends the abstract class DecoratorFigure, and as such implements the Figure interface. When a Figure object is passed as an argument to the constructor of DecoratorFigure, the Figure object is decorated with a border.

However, it is not clear to what extent Decorator is a crosscutting concern. Marin et al. [MMvD05] list this pattern, bundled together with its cousin Adapter, in their canonical list of elementary crosscutting concerns. (It is listed as the Redirection Layer concern). In doing so, they refer to Hanneman and

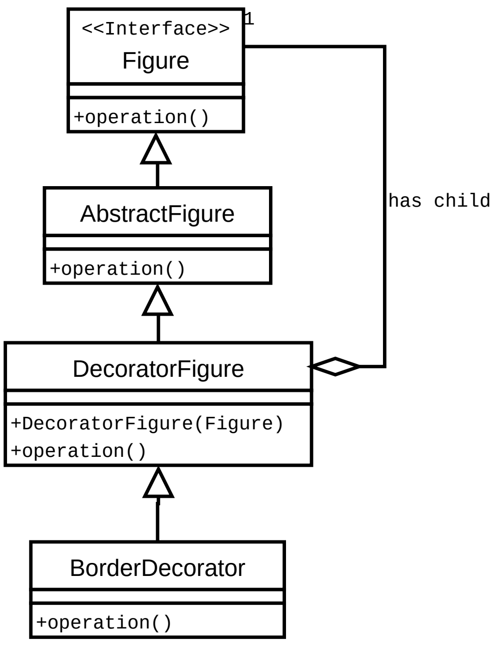


Figure 6.4: UML diagram of an instance of Decorator in JHotDraw.

Kiczales, who have come up with a refactoring for Decorator. According to Table 6.1, this refactoring shows improvements in three of the four modularity criteria.

On the other hand, van den Berg et al. [vdBCH06] propose an "operationalization of the definition of crosscutting", in order to aide the identification of crosscutting concerns. More specifically, they use matrices to model the relation between design elements, and then read a degree of crosscutting from these matrices. Applying this technique to an instance of the Adapter design pattern, they conclude that "this pattern does not require the utilization of AOP because there is no crosscutting" – we presume the same to hold for Decorator, seeing the similar nature of these patterns, both being redirection layers.

If we take a closer look at the refactoring proposed by Hanneman and Kiczales, we witness a simple mechanism. Each instance of the Decorator pattern has a corresponding aspect. Those methods of the decorated object to be added functionality to are captured through a pointcut and augmented with advice. Precedence of one Decorator over another is declared through aspect-precedence.

We presume that Hanneman and Kiczales evaluated this refactoring as follows:

- **Locality (improved)** The client is now free of the so-called "glue code" which manually wraps the Decorators around the target objects, as each Decorator aspect automatically applies to the target objects captured in its pointcut(s).
- **Reusability (no improvement)**

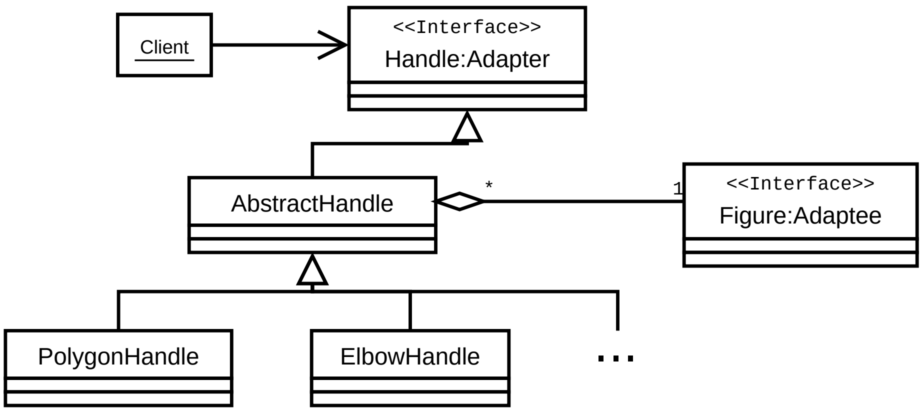


Figure 6.5: UML diagram of an instance of Adapter in JHotDraw.

- **Composition Transparency (improved)** The aspect precedence mechanism gives a more localized and clear specification of the precedence of different instances of Decorator. This contrasts with the object-oriented solution, in which this precedence is determined through the order in which the wrappings occur in the glue code. The glue code is often scattered and unstructured.
- **Unpluggability (improved)** Unplugging a decoration can now be done from within the corresponding Decorator aspect. This is easy, in comparison with having to get hold of the decorated object in the base system, to unwrap it of its decorations and pass it on stripped.

However, Hanneman and Kiczales' refactoring has raised some remarks [DAAC05]. The aspect elements are weaved into the base system at compile-time. This poses problems when it is required to dynamically reorder the precedence of Decorators, dynamically enable different combinations of Decorators, decorate only a subset of the instances of a targeted class, or decorate an object only during specific phases. The discussed refactoring could potentially not be sufficient for the above-discussed pattern instance in JHotDraw, as Figures are to be dynamically decorated with (and stripped of) borders.

6.4.4 Adapter

When a class proves difficult to plug into a software system because its interface is incompatible, the Adapter design pattern leverages the problem by adapting the problematic class to the desired interface. Adapter does this via a new class that does implement the desired interface, and of which each instance owns an instance of the class with the incompatible interface. The owning object translates any incoming request to a request for the owned object.

This design pattern clearly manifests itself in JHotDraw through the Handles mechanism (Figure 6.5). All the concrete Handles implement the same interface, but they differ in how incoming requests are translated before they are passed on

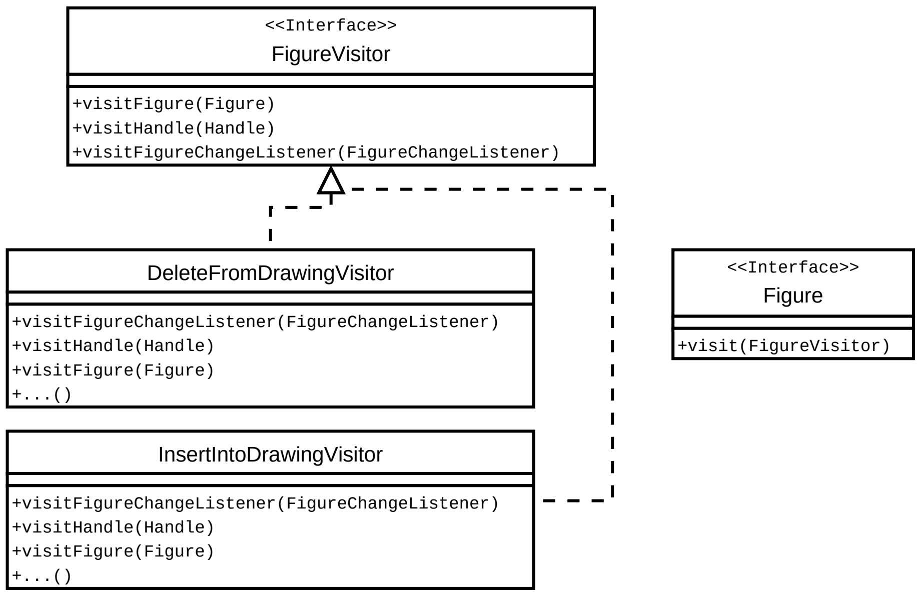


Figure 6.6: UML diagram of an instance of Visitor in JHotDraw.

to the owned Figure object. We believe the discussion regarding the crosscutting nature of Adapter to be similar to that in the previous section where Decorator was investigated. Both patterns are namely redirection layers of sorts.

6.4.5 Visitor

The Visitor design pattern allows to add new operations to the objects in an object structure, without modifying that structure. This allows for a cleaner separation between the structure and algorithms that use it.

On each class that participates in the object structure, the Visitor pattern imposes a `visit` method. When the object structure is to be augmented with new functionality, the client is to do nothing but encapsulate that functionality in a Visitor class. Each `visit` method takes a Visitor as its sole argument and automatically calls the appropriate functionality within the Visitor. The Visitor pattern is precisely crosscutting because of the `visit` methods it imposes throughout the object structure.

Visitor is used in JHotDraw over a hierarchy of Figures (Figure 6.6). There is a Visitor for deleting a hierarchy of Figures from a Drawing, and one for inserting a hierarchy into a Drawing. These run through the hierarchy of Figures and collect all Figures that are affected by the deletion, respectively insertion. Thus, it is the Figures that implement the `visit` methods.

6.4.6 Composite

In the Composite design pattern, of main interest to the client is the Component interface. Composite objects consist out of several Component objects. The

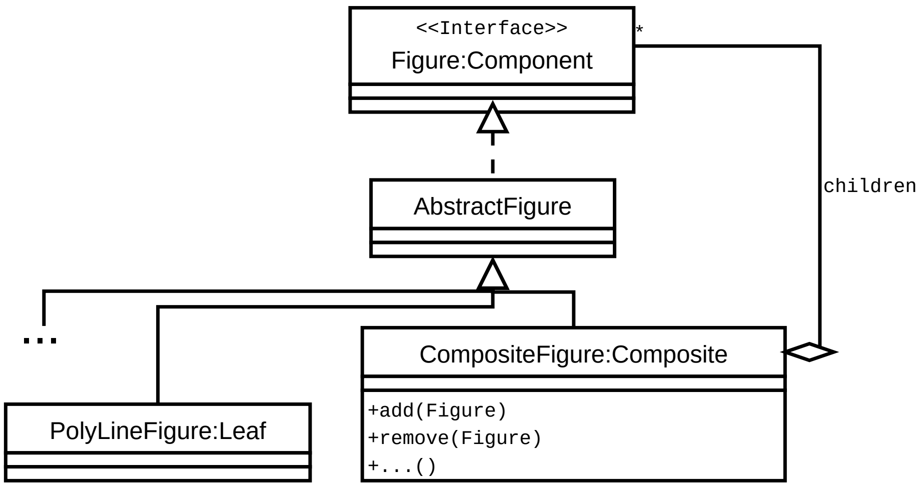


Figure 6.7: UML diagram of an instance of Composite in JHotDraw.

Composite pattern allows to manipulate a Composite as if it were a Component. To this end, the Composite class too implements the Component interface. However, the Composite class has to implement some additional methods that serve to manage the children of a Composite.

The most prominent instance of this pattern in JHotDraw is the class CompositeFigure. It implements the Figure interface whilst being a collection of several Figures (Figure 6.7).

The crosscutting behaviour lays with the "composite behaviour" of the Composite class. This becomes clear once one beholds the refactoring proposed by Hanneman and Kiczames. A designated aspect is created for each instance of the Composite design pattern. This aspect maintains a mapping between each Composite and its children. Additionally, the aspect encapsulates the utility methods to manage and access the children of a Composite. It is expected from the client that he, or she, gets hold of the instance of the aspect, in order to call these utility methods. A such, the Composite in the base system doesn't even need to keep track of its children, and is effectively freed of its "composite behaviour".

6.4.7 Command

Command is a design pattern that serves to represent a request by a Command object. For each kind of request there is a different concrete Command, whose (instance) fields can be set using all kinds of properties of the request (properties that would traditionally be passed along a method call as its arguments). Upon calling of its `execute` method, a Command brings the request it represents into practice, working with the Receiver whom the request is meant for.

Manual investigation of JHotDraw again shows a design pattern being used to bridge view and controller logic (Figure 6.8). While the Command hierarchy belongs to the controller logic, classes calling the Commands' `execute` methods

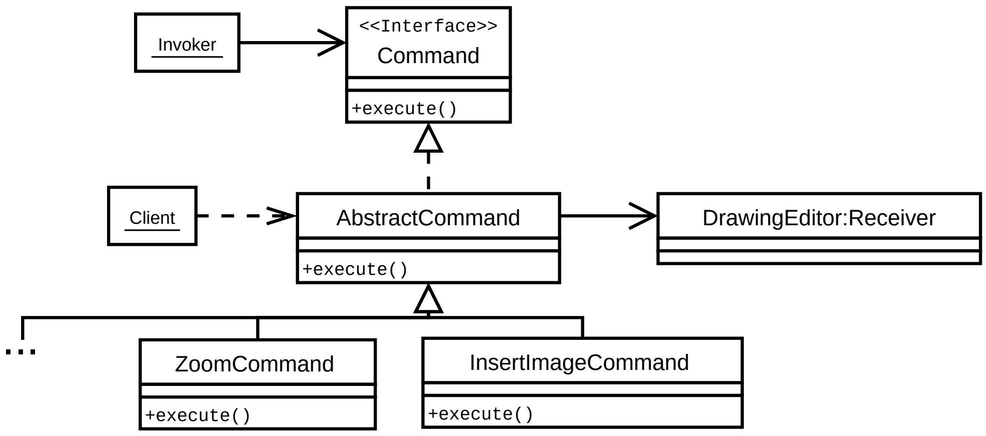


Figure 6.8: UML diagram of an instance of Command in JHotDraw.

derive from Swing components (recall that Swing is the underlying graphics system for JHotDraw).

Hanneman and Kiczales propose the use of an abstract aspect. This aspect declares three different roles for participating classes: (1) Command, (2) Invoker, i.e. the object that calls the `execute` method of a Command, and (3) Receiver. The selling point of this refactoring is how it frees the base system from:

- code that associates Invokers with Commands;
- code that associates Commands with Receivers;

The aspect keeps data structures to remember these associations. Additionally, pointcuts and advice are deployed to refactor the join points at which Invokers call `execute`. The intent is for the base system to be freed entirely from the Command pattern code.

6.4.8 Other Concerns

Here we present a series of crosscutting concerns that do not require to be as thoroughly discussed as the previous ones, because their crosscutting behaviour is more obvious. These concerns can further be divided into two groups, according to the applicable aspect refactoring: the first group consists of concerns that can be remodularized through inter-type declarations (to factor out and reimpose methods), while the second group requires pointcuts and advice.

Require Inter-type Declarations

- **Persistence** [vDMM05] This concern manifests itself through classes that implement the Storable interface, which consists out of the methods `write` and `read`. The `write` method serves to write a Storable object to a StorableOutput object, which is a specialized output stream. The `read` method serves to resurrect a Storable from a StorableInput object.

- **Serialization** The Java language allows an object to be converted into a sequence of bytes, with the possibility of later resurrecting a live object from this sequence. However, an object can be serialized in this manner only if it implements the Serializable interface. This interface declares no methods or fields, and serves only to identify serializable objects. If an object requires to be handled in a particular way during serialization, it needs to implement the `writeObject` and `readObject` methods.

- **Cloning** When a class implements the Clonable interface, this indicates that it is legal for the `Object.clone` method to make a field-by-field copy of an instance of the class. Clonable classes are also required to override this `clone` method.

- **Drawing/Moving Figures** [CMM+05] Some researchers have observed crosscutting in how Figures encapsulate some methods dedicated to drawing figures, and some methods that move figures. These can be regarded as different concerns.

Require Pointcuts and Advice

- **Command Execute** [vDMM05] Within the Command hierarchy, another crosscutting concern is present. The `AbstractCommand.execute` method does nothing but perform a check to see whether the DrawingView associated with the Command is not null. If this condition is not met, an exception is thrown. Methods that override this `execute` method are forced to do a super call to the method.

- **CheckDamage** Furthermore, most `execute` methods conclude with a call to `checkDamage`, to check whether the DrawingView has been changed.

- **WillChange/Changed** [SP05a] The `willChange` and `changed` methods inform a Figure that it respectively will be changed, or has changed. This pair of methods is often found enclosing the body of a Figure-changing method, thus making it a prime target for aspect refactoring.

- **Space** [MvDM04] The StorableOutput class implements a set of methods that write values to the stream. Each of these methods concludes with a call to the `StorableOutput.space` method, in order to write a space to the stream.

6.5 Summary

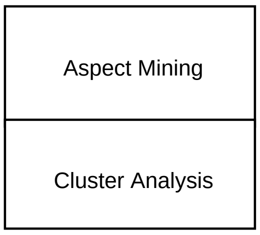
We have chosen JHotDraw as the benchmark for the evaluation and comparison of our aspect mining techniques. This choice is due to (1) the degree to which JHotDraw has already been deployed in aspect mining research, and (2) JHotDraw's reputation as a system that has already been modularized to the most of object-orientation's capabilities. We have discussed how design patterns are not, generally, infallible. In fact, a significant amount of the patterns can be regarded to have a crosscutting nature. Table 6.3 lists all crosscutting concerns that we have documented in JHotDraw, most of which were thoroughly discussed throughout this chapter. We deploy this list of crosscutting concerns in the evaluation and comparison of our aspect mining techniques.

Chapter 7

Five New Aspect Mining Techniques

The main contribution of our work is the design and implementation of five aspect mining techniques that use cluster analysis to automatically mine for all aspect candidates in the legacy system. A manual post-processing step is still at order in the use of each technique, in which the developer manually investigates the outputted clusters. At the heart of each technique lies a separate distance measure between the methods in the legacy system. Furthermore, we contribute a working definition of crosscutting. According to this definition, we tailor a mining technique to detect one specific category of crosscutting behaviour. We classify a mining technique as a Logic Clustering or a Method Clustering technique, according to whether it focusses on crosscutting in method bodies or classes, respectively. We discuss each of the five mining techniques according to this classification. First, we relate our approach to cluster analysis that underlies all the aspect mining techniques we propose. We conclude with a comparison of our work with aspect mining techniques authored by other researchers, in terms of the properties of the analyses.

7.1 Our Approach to Cluster Analysis



The principle of cluster analysis underlies each of the aspect mining techniques we propose. One can illustrate our overall aspect mining work as consisting out of two layers: an aspect mining layer, on top of a cluster analysis layer. There is a degree of independence between both layers, in that we can experiment with different cluster algorithms, while the ideas specific to aspect mining (e.g. the choice of distance measure) stay the same. Because cluster analysis is primarily an exploratory tool, we see fit to experiment with different approaches that perform cluster analysis (e.g. algorithms). In practice, we have implement the LanceWilliams framework for cluster analysis, and secondly propose *cluster feature* solutions, which deploy domain knowledge in performing cluster analysis.

7.1.1 A Choice of Cluster Algorithms

Considering which cluster algorithm(s) to use as the basis of our aspect mining techniques, we exercise a bias against algorithms that do not support distance spaces. Some of our distance measures compare the parse trees of methods, or strings. In those cases, it does not make sense to break down an object (e.g. parse tree) into separate attribute values. Grid-based and subspace-based clustering require a space formed by the object attributes (see Sections 4.7 and 4.8, respectively). The typical algorithms that implement density-based clustering (DBSCAN, OPTICS) require numerical attributes (Section 4.6). Hence, the grid-based, subspace-based, and density-based schemes are of lesser interest, because they do not support distance spaces.

The purpose of our work is to use one mining process (cluster analysis) to drive the other (aspect mining). We lay the emphasis on the overall aspect mining goal. In analyzing the aspect mining results, we do not wish the underlying cluster analysis to provide extra degrees of uncertainty. Therefore, we favour cluster algorithms that do not require parameters to be provided by the user. The typical partitional clustering algorithms (Section 4.4) do not satisfy this property. In the common case, they require the user to provide the desired number of clusters.

Regarding model-based clustering (Section 4.9), we have no particular interest in a mathematical model that captures the data. Especially if this approach provides no guarantees to generate a higher-quality clustering.

Single pass clustering (Section 4.3) has no characteristics that ensure any degree of quality in cluster analysis. However, the comfortable complexity of the technique, in both time and space, makes it interesting as a pre-processing step. In our case, we often perform a single pass to group near-identical objects, as a way of leveraging the size of the input for the main analysis. The idea is that the single pass is not likely to make wrong decisions in grouping the objects, precisely because it operates using a (very) strict sense of near-identity. The treshold at which to consider objects as near-identical is determined by experimentation and domain knowledge. For example, in grouping methods for similarity in their names, it is easy and rewarding to first group the identically-named methods (in a single pass), before considering different degrees of similarity (in a more expensive analysis). This pre-processing becomes especially interesting when near-identity, or identity, can be determined using a cheaper distance measure than the one used in the main analysis.

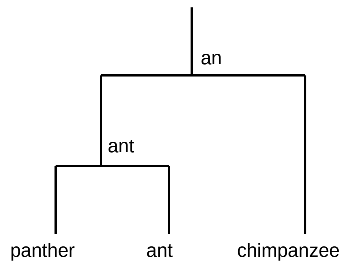


Figure 7.1: The AHC process, as determined by using the technique of cluster features.

By means of elimination, we arrive at hierarchical clustering algorithms. The algorithms of this kind that we presented in Section 4.5 are agglomerative hierarchical cluster (AHC) algorithms. They all support distance spaces, with the exception of CURE. Because hierarchical algorithms construct a hierarchy of several clusterings, they typically demand less input parameters. The algorithms CURE and ROCK do request some input values, while CHAMELEON has what we believe to be a significant amount of input parameters.

We have implemented the Lance-Williams framework for cluster analysis (Section 4.5.1). Through the parameters in the Lance-Williams formula, this framework allows to efficiently implement 5 distinct cluster algorithms: MaxLink, MinLink, GroupAverage, Ward and Centroid. While these 5 algorithms are conceptually simple, they nonetheless form a group that is representative of (certain) common AHC algorithms. Recall from our discussion on the CHAMELEON algorithm, that MinLink considers the closeness of clusters, as does CURE. Similarly, both GroupAverage and ROCK focus on the aggregate inter-connectivity of clusters. Because Ward and Centroid rely on centroids, which must be computed from numerical attributes, we have only implemented MaxLink, MinLink and GroupAverage.

7.1.2 Cluster Features

One AHC approach is commonly omitted from textbooks on cluster analysis. This does not surprise, as it is a domain-specific approach. More precisely, it is the idea to use knowledge from the problem domain to compute the distance between composite clusters. We have already seen an example of this. It is the aspect mining technique of Shepherd and Pollock (Section 5.3.5). This technique labels the objects with strings. The distance between the objects is a string dissimilarity measure applied to their labels. Additionally, composite clusters are given labels too. Shepherd and Pollock take the label of a composite to be the common substring amongst the two subclusters that form that composite. To create the AHC process, the same string dissimilarity measure is used on objects and composites alike. Figure 7.1 illustrates the AHC process in which this strategy results. The structure in the figure is similar to a dendrogram, with the objects at the bottom and a new composite at each joint. Instead of denoting each joint with a distance, we give the label of the composite cluster that is formed.

We suggest one such "cluster feature solution" for each of the five aspect mining techniques (distance measures) we have designed. The act of *feature derivation* is to label a composite cluster using the labels of the two subclusters that form it. We use the term *cluster feature* to denote the label that is given to each composite cluster and object. In a cluster feature solution, the idea is that the distance measure between objects strictly operates on the cluster features, and that we apply the same distance measure to composite clusters, in order to create the AHC process. Additionally, we always take the feature of a cluster to relate some information that holds for each object in that cluster.

Throughout the rest of this text, we explain each cluster feature solution by means of (1) what constitutes the cluster feature of an object, and (2) how feature derivation is performed. The used distance measure is implicitly taken from the general context (aspect mining technique) in which the cluster feature solution is proposed.

As an example, for the technique of Shepherd and Pollock, we specify:

- **Feature of an Object** The name of that method (which is a string)
- **Feature Derivation** Taking the common substring of two strings

7.1.3 Automatic Selection of a Clustering in the Hierarchy

We have implemented the technique of silhouette coefficients to evaluate the result of cluster analysis (Section 4.10.1 for a discussion of silhouette coefficients). This evaluation measure is the only (common) measure that considers both cluster separation and cohesion, whilst not having a bias towards particular types of clusters (e.g. center-based clusters).

The manner in which we use this measure is two-fold. One one hand, the outcomes of different cluster algorithms must be compared. On the other hand, an AHC algorithm produces a hierarchy of several clusterings. Such a structure can be a wealth, but it is nonetheless practical to have a starting point in the hierarchy, i.e. one particular clustering that seems most fit. We consider the clustering with the highest silhouette coefficient to be that fittest clustering. In fact, it is this fittest clustering that our post-filtering steps analyze, and that is ultimately presented to the aspect miner.

7.2 The Working Definition of Crosscutting

Each aspect mining technique is based on its own notion of what symptoms are associated with crosscutting. We remark that each technique relies on an implicit *working definition* of crosscutting. For example:

- to fan-in analysis (Section 5.3.2), a crosscutting concern is a method that can be called from many different places in the code, and that meets some additional conditions;
- to techniques based on clone detection (Section 5.2), a crosscutting concern is a set of clones with certain properties.

Note that the working definition of a particular technique is generally much narrower than what the authors of the technique believe to be crosscutting. That is why we call it a *working* definition: it is strictly meant for use in the actual mining process.

One contribution of our work is a new working definition of crosscutting. It is less restrictive than the implicit definitions that can be found in techniques such as fan-in analysis and clone detection techniques. However, we do not propose a mining technique that methodically looks for all concerns that meet our working definition. Rather, each of our aspect mining techniques relies on the definition to a certain degree. We have derived various pre- and post-filtering steps from the definition, which accompany the cluster analysis. In fact, we have precisely designed the definition because we identified a need for filters in earlier aspect mining experiments. The filters we have implemented are able to:

1. Reduce the input to the cluster analysis. On one hand this is beneficial to the time and space requirements of the (rather expensive) cluster analysis. On the other hand, the cluster analysis is more likely to produce an interesting aspect mining result, if the pre-filtering step does a good job at filtering uninteresting data.
2. Reduce the output of the cluster analysis. This limits the amount of post-processing work the user has to perform manually. (This manual work is generally tedious and error-prone.)

7.2.1 The Definition

Our working definition of crosscutting is strictly based on the case of mining a legacy system written in Java (although we imagine it can be adapted to certain other programming languages). We focus on code entities that crosscut code entities. In specifying what these entities are, we discriminate two cases:

- **Case 1** Logic that crosscuts methods (the tangling and scattering happen in the bodies of methods).
- **Case 2** Methods that crosscut classes (the tangling occurs in terms of methods that belong to different concerns being implemented in the same class, the scattering occurs in terms of the methods of a concern being implemented over different classes)

We deem methods and classes to be the primary candidates for being the crosscut modules in a legacy system. Hence, it is our belief that the above two cases cover most of what constitutes crosscutting in source code. Of course, a concern might exhibit both crosscutting cases. This is especially true for design patterns. E.g. the Observer pattern requires notification code in methods of the Subject (**Case 1**), while a Subject implements methods for registering Observers (**Case 2**). (See Section 6.3.1 for a discussion of this design pattern.) However, we hypothesize that both crosscutting cases can be mined for independently, without this distinction being significantly detrimental to the detection of the crosscutting concerns in the legacy system. In keeping with the Observer example, one technique might uncover the notification code, while another technique might point out the `register` and `unregister` methods.

If a technique is specific to one case of crosscutting behaviour, we can have it implement more specific measures (e.g. filters). The separate results can then be combined in a manual processing step, or using an additional technique. This strategy is the classic approach of "divide-and-conquer". A Logic Clustering technique is tailored to focus on instances of **Case 1**, while a Method Clustering technique is interested in instances of **Case 2**.

The Definition in Case 1

Imagine a concern C. For us to consider C as participating in an instance of crosscutting **Case 1**, it must meet certain conditions. We hereby specify them.

Let M denote the set of all methods in the legacy system, save the method declarations in interfaces. However, we do not entirely ignore these declarations. Rather, they help in shaping the binary *could-call* relation between methods in M. This relation holds between one method and another, if the first could call the second (taking into account the polymorphism mechanism of object-orientation in general and Java specifically).

We additionally define a distance between classes, namely the *hierarchy distance*. This measure relates how closely two classes sit in the legacy system. It is defined in Figure 7.2. (We take a "superclass" to mean a direct superclass, while an "ancestor" is a direct, or indirect superclass.) For example, two classes have hierarchy distance 1, only if one class is a superclass of the other, or they both have the same superclass. We also talk of the hierarchy distance between two methods, which denotes the hierarchy distance between the classes they are implemented in.

For us to consider a concern C as participating in an instance of crosscutting **Case 1**, the conditions in Table 7.1 must hold.

Imagine a concern that crosscuts only one method in the system (e.g. the execution of only one method is logged to the stream). Our working definition can fall short of capturing the concern, due to the **Scattering** condition. However, one can question the significance of this concern. Does it pose a great threat to the overall manageability of the legacy system? We thought not, which is why the **Scattering** condition is included in the first place. This case further

$dist(class_1, class_2) = 0$	iff $class_1 = class_2$
$dist(class_1, class_2) = 1$	iff one is the superclass of the other
$dist(class_1, class_2) = 2$	iff one is the superclass of the superclass of the other
$dist(class_1, class_2) = 3$	iff one is the superclass of the superclass of the superclass of the other
...	
$hdist(class_1, class_2) = x$	iff $dist(class_1, class_2) = x$, or $class_1$ and $class_2$ have a common ancestor $class_3$, such that $class_3 \neq$ java.lang.Object, and the greater value of $dist(class_1, class_3)$ and $dist(class_2, class_3)$ is x.
$hdist(class_1, class_2) = \infty$	iff there is no $x \in N$, such that $hdist(class_1, class_2) = x$
Where *hdist* denotes the hierarchy distance.	

Figure 7.2: The hierarchy distance between two classes.

For concern C:

- There is a set of *crosscutting methods* $(CM) \subset M$, which implement C.
- There is a set of *calling sites* $(CS) \subset M$, where each calling site could-call a subset of CM. The idea is that the calling sites are crosscut by the crosscutting methods.
- **Scattering** $|CS| > 1$
- **Redelegation**
  $\forall m_1 \in CS : \exists m_2 \in CM : (m_1 \text{ could-call } m_2) \wedge hdist(m_1, m_2) > \alpha$
- **Tangling**
  $\forall m_1 \in CS : \exists m_2 \in CM : \exists m_3 \notin CM :$
  $(m_1 \text{ could-call } m_2) \wedge (m_1 \text{ could-call } m_3) \wedge hdist(m_2, m_3) > \beta$
- **Polymorphism** The calling sites are not polymorphic methods.

Table 7.1: The working definition of crosscutting in **Case 1**.

illustrates how our definition is a *working* definition. It can perfectly well ignore types of crosscutting concerns that seem less significant, in order to produce a more manageable mining result. This trade-off becomes particularly important in the light of the manual post-processing phase that comes after each mining technique. The aspect miner should not be overwhelmed with results, as manually analyzing them can be tedious and error-prone.

The **Polymorphism** condition is included for a similar reason as the **Scattering** clause. Too many false positives in our mining techniques traced back to cases of polymorphism. This is most natural when methods are clustered for similarity in their names. But the problem occurred with each technique. Hence the added condition, even though it is perfectly possible for a concern to precisely crosscut polymorphic methods (see the Command Execute concern, Section 6.4.8). Thus, the trade-off lies between not detecting certain crosscutting concerns, and an aspect mining result with significantly less false positives.

The **Redelegation** clause discriminates against calling sites that are hierarchically close to the crosscutting methods they could-call. One can regard such a calling site as simply redelegating requests to methods in the same hierarchy (in calling the crosscutting methods). We do not believe this to constitute crosscutting. Similarly, the **Tangling** condition discriminates against a calling site, if the crosscutting methods that it could-call are hierarchically close to the other methods it could-call. Such a calling site can be regarded uniform, in that it only makes use of the hierarchy (or hierarchies) of concern C. Consequently, we argue that such a calling site does not display the tangling that is perhaps the primary symptom of crosscutting.

The working definition of crosscutting is parametrized. The parameters α and β determine the hierarchical distance at which to distinguish redelegation and tangling. In the case of mining JHotDraw, we take the values to be low (1 or 2). This is because JHotDraw is the composition of a limited amount of very large class hierarchies (e.g. the Figure, Handle, Command hierarchies). Some crosscutting concerns we have documented manifest themselves completely in only one of these hierarchies.

The above-outlined definition of crosscutting in **Case 1** does have a shortcoming. In the face of crosscutting instances of **Case 2**, i.e. methods being wrongly encapsulated in a class, the **Tangling** and **Delegation** conditions can fail.[1] This is because they discriminate against concerns on the basis of a low hierarchy distance between methods.

The Definition in Case 2

Imagine a concern C. Let M denote the set of all methods in the legacy system, save the method declarations in interfaces. For us to consider C as participating in an instance of crosscutting **Case 2**, it must meet the conditions in Table 7.2.

[1]Even though, hypothetically, one could deploy a **Case 2** technique to clear instances of that particular problem, after which a **Case 1** technique could use the **Tangling** and **Delegation** conditions in a more robust manner.

For concern C:

- There is a set of *crosscutting methods* $(CM) \subset M$, which implement C.
- There is a set of *implementing classes* $(IC) \subset M$, where each class implements a subset of CM. The idea is that the implementing classes are crosscut by the crosscutting methods.
- **Scattering** $\exists c_1 \in IC : \exists c_2 \in IC : hdist(c_1, c_2) > \alpha$
- **Tangling**
  $\exists c \in IC : \exists m_1 \in CM : \exists m_2 \notin CM :$
  $(c\ implements\ m_1) \wedge (c\ implements\ m_2)$
- **Polymorphism** The calling sites are not polymorphic methods.

Table 7.2: The working definition of crosscutting in **Case 2**.

The **Scattering** clause asks that the crosscutting methods are implemented in sufficiently different hierarchies. Again there is the same trade-off to consider: between a more manageable mining result, and some less pervasive crosscutting concerns not being detected. The parameter α should taken to be low, if one wants to detect concerns that completely occur in a larger hierarchy (e.g. Drawing/Moving in the Figure hierarchy in JHotDraw, see Section 6.4.8). The **Tangling** clause discriminates against an implementing class, if it only implements the crosscutting methods. Such a class can be regarded as uniformly dedicated to C, at which point there is no (**Case 2**) crosscutting to be observed in that class.

7.3 An Overview of the Five Aspect Mining Techniques

In order to use the above-given working definition of crosscutting, the sets of calling sites, crosscutting methods and implementing classes must be mined. To perform this mining, we use cluster analysis to group methods. This strategy seems natural in the light of the **Scattering** conditions. Furthermore, each of our aspect mining techniques adds some additional hypothesis to the working definition. We hereby specify the techniques in terms of these hypotheses.

With respect to a valid instantiation of **Case 1** of the working definition:

- **Call Clustering** [Is based on the additional hypothesis that] the crosscutting methods form one pattern, which then recurs in each calling site.

	(cluster interpretation) **A Cluster corresponds to ...**	(distance measure) **Two methods are similar when they ...**
Tree Clustering	the calling sites	are structurally similar
Ngram Clustering	the calling sites	are lexically similar
Type Clustering	the calling sites	are similar in the types they contain
Call Clustering	the crosscutting methods (**Case 1**)	form a recurring pattern to a certain degree
Name Clustering	the crosscutting methods (**Case 2**)	have similar names

Table 7.3: The aspect mining techniques and their characteristics in implementation.

- **Type Clustering** The calling sites are similar in the types they contain.
- **Ngram Clustering** The calling sites have lexically similar bodies.
- **Tree Clustering** The calling sites have structurally similar bodies.

With respect to a valid instantiation of **Case 2** of the working definition:

- **Name Clustering** The crosscutting methods have similar names.

We remind that every technique considers methods in the legacy system as the objects to perform cluster analysis on. Furthermore, the above hypotheses suggest two notions for each technique: a distinct distance measure, and an interpretation of a cluster. We first discuss the interpretations of clusters. With respect to a valid instantiation of **Case 1** of the working definition, a cluster in

Type Clustering,

NGram Clustering,

or Tree Clustering

is interpreted as the set of calling sites. A cluster in Call Clustering is the set of crosscutting methods. With respect to a valid instantiation of **Case 2** of the working definition, a cluster in Name Clustering is the set of crosscutting methods.

The distance measures of the techniques can now be straight-forwardly deducted. Table 7.3 gives the distance measures and different interpretations of clusters.

7.4 Implementation Notes

We have fully implemented the five aspect mining techniques we propose, the accompanying pre- and post-filtering steps, as well as the chosen and designed cluster algorithms. All implementation work was carried out in the Smalltalk programming language. We have deployed external tools in the code analysis, and in the presentation of the aspect mining results.

7.4.1 The Presentation of the Aspect Mining Results

The StarBrowser [Wuy], allows the user to browse through the Smalltalk environment, and drag any object he or she encounters into a classification in the StarBrowser. (Because everything in Smalltalk is an object, this effectively means that everything can be classified.) The user can further create custom classifications, and nest them at will.

We have integrated the StarBrowser in our implementation work. After the use of any of our aspect mining techniques, each resulting cluster is automatically presented to the user as a separate classification in the StarBrowser. This presentation offers the user an accomodating view of the aspect mining results, in which he or she can navigate through the clusters, the methods in a cluster, the classes those methods are implemented in, further properties of those classes, etc.

We believe such a browsable presentation of the aspect mining results to be invaluable in manually processing those results. In a view where only methods names are displayed, or indeed only their method bodies without further context, it can be hard to recognize crosscutting behaviour.

7.4.2 The Underlying Code Analysis Workbench

Our implementation work is founded on Irish [Fab]. This workbench allows to parse Java code, after which the code can be queried using a logic programming language. Irish operates on top of Smalltalk. Irish's built-in predicates allow to retrieve logic facts such as the superclass of a class, or the messages sent by a method. The more complex predicates we had to implement ourselves. Our custom predicates encompass, but are not limited to:

- the could-call relation between methods;
- the hierarchy-distance between classes;
- a check whether certain methods are polymorphic.

There are two remarks to be made. On one hand, a semantical code analysis technique is potentially expensive. Consider the query to compute the set of methods that a particular method could-call. This requires a lot of other relations to be involved in the computation, as the could-call relation takes polymorphism into account. For example, it might be necessary to trace back all the classes that implement a given interface. In finding these classes, it is necessary to consider all descendant interfaces that are direct, or indirect subinterfaces of the given interface. The classes that directly implement these descendant interfaces must be considered, as well as the descendant classes of those classes. A simple could-call query results in an intricate computation process. The logic program we constructed to implement the could-call predicate was inefficient. Therefore, we implemented it imperatively in the underlying Smalltalk base. Although, the point remains, the imperative implementation is still expensive.

The second remark concerns the accuracy of the predicates we have implemented. For example, it is possible to implement compile-time overloading resolution for Java (not overriding). However, we believe that (1) such implementation work would lead us too far from the real concerns of our work, (2) such code analysis would be computationally expensive. Therefore, some of the predicates we have implemented are approximation measures. For instance, methods are considered polymorphic when:

- they have the same name (where the name excludes the signature and return type);
- they are implemented in classes that have a common ancestor type (class or interface);
- the common ancestor also implements, or declares a method with the same name.

In ignoring signatures and return types, this measure is not entirely accurate, but remains relatively simple.

7.5 Logic Clustering

Of the five aspect mining techniques we have designed and implemented, four are tailored towards detecting instances of crosscutting **Case 1**. They are the Logic Clustering techniques. Before discussing the four techniques separately, we describe the pre- and post-filtering steps that precede and follow each technique. These steps have been implemented to perform fully automatically.

7.5.1 Pre-filtering

We do not propose a mining method that methodically looks for concerns that satisfy our working definition of crosscutting. Instead, we rely on this definition to derive several specific ideas. One of the ideas is a pre-filtering step that precedes all Logic Clustering techniques.

Concretely, the pre-filtering step investigates whether a particular method could be a calling site within a valid instantiation of the working definition. If a method could be a calling site (according to the pre-filtering step), it is a *qualified caller.* A method is a *qualified callee* only if a qualified caller could-call it.

Each of the Logic Clustering techniques uses the set of qualified callers as a starting point for the aspect mining process. Type, Ngram and Tree Clustering only consider qualified callers as the objects to perform cluster analysis on. This is natural, as those techniques mean a cluster to be a set of calling sites. The cluster analysis of Call Clustering only considers qualified callees as the objects.

In implementation, we consider a method m_1 to be a qualified caller when:

- It calls more then two methods
- **FRedelegation** $\exists m_2 \in M : (m_1 \text{ could-call } m_2) \wedge hist(m_1, m_2) > \alpha$
- **FTangling** $\exists m_2 \in M : \exists m_3 \in M : (m_1 \text{ could-call } m_2) \wedge (m_1 \text{ could-call } m_3) \wedge hdist(m_2, m_3) > \beta$

where M again signifies all the methods in the legacy system, save the declarations in interfaces.

The **FRedelegation** clause asks that m calls at least one method that sits in a sufficiently different hierarchy than m itself. The **FTangling** clause imposes the condition that m ought to call two methods that have a sufficient hierarchical distance between them. These two conditions work towards meeting their **Redelegation** and **Tangling** counter-parts in the working definition of crosscutting (in **Case 1**). The first condition in the filtering demands a qualified caller to call at least two methods. This condition might seem superfluous, in that the **FTangling** condition implicitly demands it. In implementation however, this first step proves an easy filter to leverage the work of the two more expensive measures that follow it.

7.5.2 Post-filtering

In cluster analysis, the resulting clustering typically contains objects that are not grouped with other objects. Such "singleton" objects were not determined to engage in a meaningful relationship with other objects. Hence, we filter them away. In the context of some of our aspect mining techniques, a cluster is interpreted as the group of calling sites in a valid instantiation of the working definition of crosscutting. In those techniques, the discrimination strategy against singleton clusters directly implements the **Scattering** condition. In the same techniques, we additionally filter any cluster that encompasses only polymorphic methods.

7.5.3 Call Clustering

The Call Clustering technique is based on the technique of Lili He et al. (Section 5.3.6). In terms of cluster analysis, both techniques are in fact identical in how they model objects with attributes. This means that the objects and attributes are methods. An attribute value is 1, if the attribute (method) could call the object (method); otherwise the value is 0. In our implementation, we base this assignment of the attribute values on the same could-call relation we have been discussing throughout this chapter. That is to say, it takes (Java) polymorphism into account. As in the technique of Lili He et al., cluster analysis is performed using the Jaccard distance measure for object dissimilarity. (And as such, we can refer to the example in Section 5.3.6 for an example of the cluster analysis performed by Call Clustering).

Thus, Call Clustering groups methods based on the degree to which they are called together in method bodies. Two methods have distance 0 (i.e. perfect similarity), only if they are only called *together*. In this scheme, a cluster typically consists of methods that form a recurring pattern. Call Clustering is the most semantical analysis technique we present. Rather than considering method bodies to be strings, or tree structures, the technique reasons in terms of execution paths that could occur at run-time.

In addition to only performing cluster analysis using qualified callees as the objects, Call Clustering takes the set of attributes to be the set of all qualified callers in the legacy system. (Qualified callers and callees are as determined in the pre-filtering step, Section 7.5.1.)

Additional Pre-filtering

The first step in the Call Clustering process is the construction of a dictionary. This dictionary maps each method to the methods that could-call it. Methods that could be called by fewer than 2 methods can not be part of a recurring pattern. There is hence no point in those methods participating in the rest of the analysis, since we are precisely interested in recurring patterns. Using the available dictionary, it is easy to efficiently filter away such uninteresting methods.

Cluster Feature

- **Feature of an Object** The set of attributes in cluster analysis for which the object has value 1
- **Feature Derivation** The intersection of two sets

Thus, the cluster feature of an object is the set of methods that could-call the object. Furthermore, each method in the feature of a particular cluster could-call every method in that cluster. We call the feature of a cluster its *callers*, while the methods in a cluster are its *callees*. Jaccard distance becomes

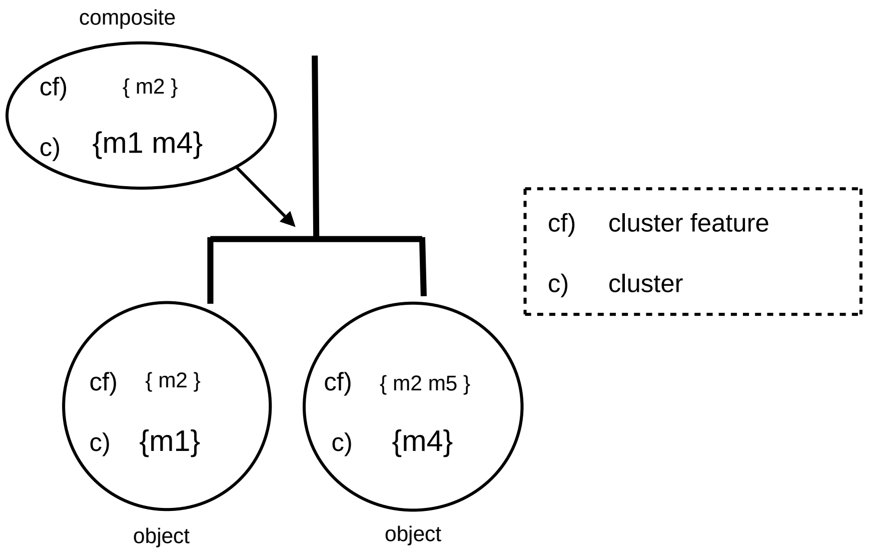


Figure 7.3: An illustration of Call Clustering's cluster feature solution.

applicable as the global distance measure between any two clusters, because composites now implicitly have attributes too. An attribute for any cluster has value 1, if the method corresponding to the attribute falls within the feature of that cluster. Otherwise the attribute has value 0. Figure 7.3 gives an illustration of Call Clustering's cluster feature solution.

Post-filtering

Call Clustering is distinct from the other mining techniques we present, in that it includes a more intricate post-filtering phase. This filtering step is based on the above specification of a cluster as two sets of methods: callers and callees. (See Call Clustering's cluster feature solution.) We hypothesize a correspondence between the callers of a cluster on one hand, and the calling sites in the working definition of crosscutting. From Section 7.3, we already interpret the callees (members) of a cluster to be the crosscutting methods. Based on this correspondence, we can apply **Case 1** of our working definition to its fullest extent. For example, we have implemented the full **Redelegation** condition: it is a check whether each calling site (caller of a cluster) is at least a certain hierarchy distance away from one of the crosscutting methods (callees of the same cluster). The conditions of **Scattering** and **Tangling** have similarly been implemented to their full extents. But rather than throw away any cluster that does not meet the three conditions, we allow some flexibility. If a calling site violates some condition, we retry to match the cluster to the working definition without that particular calling site.

7.5.4 Type Clustering

Type Clustering groups methods based on the static variable types their bodies contain. The methods in a resulting cluster typically have the same pattern of types in their bodies. In the cluster analysis, an attribute relates to a type in the legacy system. An attribute value is 1, if the object (method) makes use of that attribute (type) in its method body; otherwise the value is 0. The Jaccard distance is used to compute distance between objects; thus, two objects (methods) are similar based on the degree to which they have the same types in their bodies. (Section 4.2.1 for an explanation of Jaccard.)

In assigning the attribute values for a particular method, we only consider the static types of the variables to which that method sends messages. The restriction to message-receiving variables discriminates against Java native types such as `int` and `double`, of which the instances are not objects. We do not consider such native types as carrying much useful information regarding the modularity properties of a system. Additionally, we prohibit a method of being labeled as making use of the class that the method itself is implemented in. Without this measure, methods tend to be grouped simply on the basis of being implemented in the same class (methods often make enough `this` references for this to happen). Again, we do not think such clusters to be very interesting from an aspect mining perspective.

Cluster Feature

- **Feature of an Object** The set of attributes in cluster analysis for which the object has value 1
- **Feature Derivation** The intersection of two sets

This solution is at heart identical to that for Call Clustering. The cluster feature of an object is the set of (certain) static types the method contains. Each method in a particular cluster contains the types in the feature of that cluster. The Jaccard distance measure is again transposed to be the global distance between any two clusters. This transposition is done in an identical manner to the case of Call Clustering.

7.5.5 Ngram Clustering

The distance measure in Ngram Clustering analyzes the lexical properties of method bodies. The methods in a resulting cluster have lexically similar bodies.

An *Ngram* is a substring in a string of characters. At the basis of an Ngram lies a creation process that extracts many Ngrams from a given string. This process makes use of a *sliding window* of length n, where the parameter n specifies the length of each resulting Ngram. The sliding window is first positioned at the first character of the given string. This highlights the first n characters of that string, and so provides the first Ngram. The sliding window is then repositioned one character to the right, to give the second Ngram. In this manner, the sliding window traverses the entire string, to generate an Ngram at each character (with exception of the last $n-1$ characters). For example, when the given string is:

future events like these

Then, for $n = 5$, the Ngrams are:

futur, uture, turee, ureev, reeve, eeven, event, vents

Ngrams are sometimes used in the domain of document clustering. Recall from Section 4.2.2 that this domain is concerned with grouping documents that have similar content. Document clustering considers documents as the objects to perform cluster analysis on. In the Ngram approach to document clustering, each attribute in the cluster analysis relates to an Ngram. An attribute value signifies the frequency with which that Ngram appears in the document. When an ordering of the attributes is imposed, documents can assume the document vector representation. This allows cosine similarity to be computed between the documents (Section 4.2.2 for details).

Ngram Clustering follows the above-outlined approach to document clustering. For each of the participating methods, it builds a document vector based on the Ngrams in that method's body. Cosine similarity between the vectors is then used as object similarity.

To extract the Ngrams, the participating methods must be regarded as strings of characters. We take the characters to be the tokens in the method bodies. Thus, the Ngrams are strings (of length n) of tokens. The Java reader in Irish generates the tokens, which are pieces of text such as variable names, literals, etc. We prohibit less meaningful tokens from being part of any Ngram: parentheses, braces, dots, semicolons, etc. are ignored. Tokens pertain more semantical information than the atomic characters they are built from. Additionally, a technique that works at the granularity of atomic characters is generally expensive and requires elaborate tools to scale [KKI02].

Cluster Feature

- **Feature of an Object** The document vector of that method
- **Feature Derivation** The addition of two document vectors

We define the sum $o + p$ of two document vectors as s, where $\forall k : 1 \leq k \leq n$:

$$\begin{array}{ll} s_k = o_k + p_k & \text{if } o_k \neq 0 \wedge p_k \neq 0 \\ s_k = 0 & \text{otherwise} \end{array}$$

where n is the dimensionality of the objects in the cluster analysis, and x_i is object x's value for attribute i. The sum vector is of higher magnitude than its parts. However, cosine similarity has a normalization factor that makes the magnitude of a vector irrelevant.

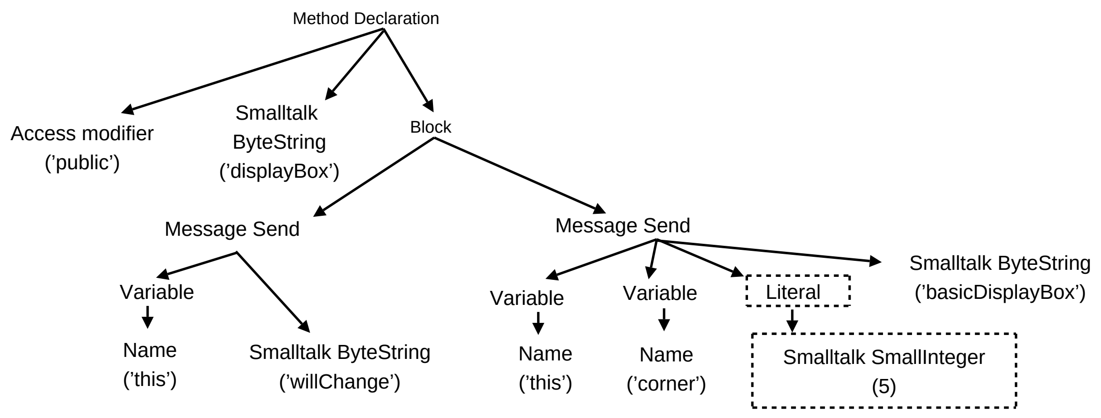

Method Declaration
Access modifier ('public')
Smalltalk ByteString ('displayBox')
Block
Message Send
Message Send
Variable
Name ('this')
Smalltalk ByteString ('willChange')
Variable
Name ('this')
Variable
Name ('corner')
Literal
Smalltalk SmallInteger (5)
Smalltalk ByteString ('basicDisplayBox')

Figure 7.4: The parse tree of the upper method in Figure 7.5.

```
public void displayBox() {
  this.willChange();
  this.basicDisplayBox(origin, 5);
}
```

```
public void displayBox() {
  this.willChange();
  this.basicDisplayBox(origin);
}
```

Figure 7.5: Two near-clones.

7.5.6 Tree Clustering

The distance measure in Tree Clustering analyzes the parse trees of methods. The methods in a resulting cluster have similar parse trees.

Consider two fragments of code that are identical, save some slight modifications to one of the fragments. As discussed in the context of clone detection techniques (Section 5.2), the two fragments are near-clones. While certain code modifications seem trivial to a developer, they can pose a significant problem to a mining method that wants to group near-clones. The oft-problematic cases are reorderings of statements, variables whose names are changed, etc.

Section 3.2 explains the edit distance between trees. It is the minimum accumulated cost to transform one tree into the other, using deletions, insertions and substitutions of nodes. When this distance is used between the parse trees of code fragments, it can leverage the problem of near-clones. Figure 7.5 illustrates this idea. The upper method differs from the lower method only because it passes an extra argument to the second method call. Figure 7.4 illustrates the parse tree that Irish generates for the upper method. In computing the edit distance between the parse trees of the upper method and the lower one, the boxed nodes in Figure 7.4 are labeled for deletion. The rest of the nodes match. This ratio suggests a high similarity between both parse trees. (As we will shortly see, this example does rely on some assumptions.)

Tree Clustering compares the parse trees of methods through edit distance, so as to group methods with similar parse trees. The edit distance is computed using our own implementation of the algorithm of Zhang and Shasha (Section 3.2 for an outline of the algorithm). Furthermore, the distance measure in Tree Clustering strictly focusses on structural information. This restriction allows for a more detailed investigation of what kind of benefits structural information has to offer, in the light of mining modularity properties in aspect mining.

The above example of edit distance relies on the scoring function we propose. Recall from Chapter 3 that the scoring function assigns a cost to each edit operation. We base this assignment of cost on an equality metric between the nodes. The nodes in the tree of Figure 7.4 are labeled with AST types (e.g. "variable", "literal"). Some nodes are denoted with a more specific AST value, which is put between parentheses (e.g. the string "willChange", the number 5). The equality metric we propose considers two nodes to be equal, only if they have the same AST type. Thus, the actual variable names and literal values are ignored, in favour of the more general labels. This strategy is commonly followed in clone detection techniques, precisely because near-clones often differ in variable names and literal values. Based on this equality metric, we define the scoring function as follows:

$$\sigma(a \rightarrow b) = \begin{cases} 1 & if \quad a = \lambda \quad (insertion) \\ 1 & if \quad b = \lambda \quad (deletion) \\ else & \\ 1 & if \quad a \neq b \quad (substitution) \\ 0 & if \quad a = b \quad (match) \end{cases} \tag{7.1}$$

Note that we extend the usual list of edit operations with an additional **match** operation, which is **substitution** in the case where the two nodes are equal. Thus, in computing the edit distance, a **match** is free, while any other edit operation comes at a price.

As illustrated in Figure 7.4, the leave nodes in an Irish parse tree are sometimes Smalltalk values. They denote literal values and names. We have already declared not to be particularly interested in that data. Therefore, we strip the parse trees of leaves. Because the tree edit distance algorithm we use is rather expensive, this trimming is also beneficial to the scalability of Tree Clustering. After the first pass over the leaves, we even perform a second pass to delete the new leaves. This fits into the idea that near-clones often interchange literals for variables, and vice versa. Additionally, typing information is omitted from the parse trees in Tree Clustering. In between this omission, the trimming of the leaves and the equality metric, the distance measure in Tree Clustering is indeed purely structural.

To use the edit distance in cluster analysis, it must be scaled with respect to the sizes of the trees. Otherwise, big near-identical trees can be regarded less similar than small but very different trees. Given the edit distance between two trees, we therefore divide it by the number of nodes in both trees. This denominator is the highest possible edit distance, which means that the scaled distance lies between 0 and 1 (this is often a requirement in cluster analysis techniques).

Scalability

The algorithm we use to compute the tree edit distance has high time and space requirements. It is not feasible to apply this algorithm to every pair of methods in a large-scale legacy system. Indeed, it took over a full week to compute all the tree edit distances in the JHotDraw case. Therefore, we propose a scalability technique that uses a lower bound to the edit distance. Given two methods in the legacy system, this bound returns a number that is lower than, or equal to, the edit distance between the methods.

Lower Bounds to Expensive Distance Measures In general, we propose two ways in which to use an inexpensive lower bound to an expensive distance measure. Both approaches limit the number of times the expensive measure is used in cluster analysis.

When the distance between two objects is to be computed, this can first be done so using the lower bound. When two objects are very far apart according to this bound, they will be at least as distant according to the expensive measure. Hence, rather than further computing the expensive distance measure between these objects, their distance is taken to be maximal (e.g. 1).

The second approach relies on single pass clustering to group objects that are near-identical according to the expensive measure. The resulting clustering provides the main AHC process with a smaller size input. (This idea has already been in discussed in Section 7.1.1). The lower bound is deployed to discriminate against objects that could not possibly be similar enough to be grouped in the single pass. After this filtering, if two objects are still candidates to be grouped in the single pass, they must be compared using the expensive measure. This is because the lower bound can be wildly optimistic (i.e. lower than the expensive distance).

Both scalability approaches require a treshold to compare the result of the lower bound against. A high treshold (distance) is required in the first approach, while the single pass strategy demands a low treshold (distance). It is true that global thresholds in cluster analysis often prove problematic, because different areas in the data set can have different properties (DBSCAN is the typical example of this issue, Section 4.6.1). However, the first approach asks for a treshold at which to consider two objects as not possibly belonging in the same cluster. We do not believe that setting this distance poses a threat to finding the natural clusters, if it is done through some domain knowledge and experience. The single pass approach still relies on the expensive measure to make any decision in cluster analysis.

When a lower bound is used to limit the use of a more expensive distance measure, the bound must be sensible. Otherwise, there is no benefit in terms of scalability. For example, consider a bound that consistently labels objects as being identical. This bound is a valid lower bound to any distance measure. However, it does not provide any useful idea as to how similar two objects really are. Therefore, some domain knowledge must be deployed in determining the bound.

A Lower Bound to Tree Edit Distance We propose a lower bound in the case of tree edit distance. Determining this bound is a matter of minimizing the estimated edit distance between two methods, such that: (1) the estimate is lower or equal than the actual edit distance, and (2) the estimate is informed enough to still be useful. The validity of our lower bound can be understood in the context of mappings between trees. Recall from Section 3.2 that a mapping draws lines between the nodes of two trees. The lower bound simply draws these lines as to minimize the estimated edit cost, whilst disregarding certain properties that the actual edit distance is restricted by.

The bound we propose relies on the node equality metric and scoring function that are used in the Tree Clustering technique. We first reduce the problem of finding the lower bound, to be the issue of finding an upper bound to the number of **match** operations in an edit script between the methods. In searching this upper bound, we ignore structural information and operate strictly on the basis of the node equality metric. Concretely, we take the upper bound to be the number of nodes in one tree that are equal to a node in the other tree, such that there is a one-to-one correspondence between the equal nodes.

In implementation, we deploy a hash function. This function maps AST types to integer indices. Given the AST tree of a method, and using the hash function, we construct a hash-like table that considers each AST type in the tree, and maps the type to the number of times it appears in the tree. (Since the values are counters, collisions are no issue.) Given such a hash-like table for each of the two methods, we can efficiently compute the upper bound to the number of **match** operations in an edit script between the methods. The hash function does not have to be perfect, in that it maps different AST types to different indices. The lower bound on the distance is allowed to be too optimistic.

Of all the nodes that do not participate in the upper bound to the amount of **match** operations, we involve as many as possible in **substitution** operations. This is done because a **substitution** operation involves two nodes in a single operation at a cost of 1. The remaining nodes engage in **deletion** and **insertion**.

In this manner we construct an estimated edit script that is, in cost, a lower bound to the actual edit script. Hence, we have a lower bound to the edit distance. Additionally, we are able to compute it in $O(n)$, where n is the number of nodes in the largest of the two trees.

Cluster Feature

- **Feature of an Object** The parse tree of that method
- **Feature Derivation** The derivation of a new tree from two trees

In the rest of this section, we relate our approach towards feature derivation. As with the other aspect mining techniques we propose, this derivation is based on information that is a by-product of deploying the distance measure. In computing the edit distance between trees t_1 and t_2, we can retrieve the necessary information to efficiently construct the optimal edit script that turns t_1 into t_2. Feature derivation is then performed as follows:

> Given two initial trees (cluster features), we compute the edit distance between them. Thereafter, we construct the optimal edit script that turns the first tree into the second. The derived tree (cluster feature) is one of the initial trees, minus those nodes that do not engage in a **match** operation.

We rely on the construction of the edit script because this considers the degree of structural similarity between the two initial trees. This is not the case for an approach that simply retains all nodes in one tree that are equal to a node in the other. Furthermore, it does not matter which of the two initial trees is stripped of non-matching nodes. The net results stays the same. This is because the derived tree appears identically in both initial trees, in terms of node labels as well as ancestor and sibling ordering. This is due to the ancestor and sibling ordering conditions on mappings.

The authors of the tree edit distance algorithm in Section 3.2 do not consider how to construct the optimal edit script in case of trees. Therefore, we adapt the approach that is followed in the string edit distance algorithm of Section 3.1. That is to say, we use backtracking pointers.

Applying the method of backtracking pointers is less straight-forward in the case of trees than in the case of strings. We need to keep a pointer for every separate subproblem in the computation of the edit distance. The subproblems are the distances between subtrees, and the distances between forests. Instead of a two-dimensional backtracking matrix as in the case of strings, we deploy the same data structures as in the tree edit distance algorithm (Table 3.5). To keep the backtracking pointers, there is one two-dimensional matrix M_s for the distances between subtrees, and there are several $2D$ matrices for the distances between forests ($M_{f,1} \dots M_{f,n}$). Distance between subtrees traces back to distance between forests. Therefore, M_s does not contain symbols, but links to $M_{f,1} \dots M_{f,n}$. Another issue arises in the second relation in Lemma 3, Table 3.4. The third clause of that relation expresses the bigger problem in terms of not one, but *two* smaller problems. Hence, two pointers could need to be associated with the bigger problem. At which point the process that constructs the edit script becomes a tree-recursive process.

7.6 Method Clustering

In the context of our working definition of crosscutting, Method Clustering techniques are tailored towards finding instances of crosscutting **Case 2**. We propose one such technique, namely that of Name Clustering.

7.6.1 Name Clustering

The technique of Name Clustering is based on the cluster-based aspect mining technique by Shepherd and Pollock (Section 5.3.5). Name Clustering groups similarly-named methods, using a distance measure that is defined as follows:

$$distance(m_1, m_2) = \frac{1}{commonSubstringLength(m_1.name, m_2.name)}$$

The paper of Shepherd and Pollock does not specify any concrete algorithm to find the common substring that is required for the distance computation. Therefore, we resort to computing the longest common substring according to a dynamic programming algorithm [Gus97]. This algorithm relies on a partitioning of the input strings into symbols. Rather than regarding the strings as being sequences of atomic characters such as letters, we partition the strings according to the occurrences of capital letters. For example, we do not partition "get" into "g", "e" and "t". We do partition the method name `moveFigureBy` into "move", "Figure" and "By". These are separate words that have distinct meanings. Additionally, the words cannot be divided into smaller, meaningful words (although this property does not hold for every string we perform the capital-letter partitioning on). Hence, the substring algorithm searches longest common substrings of such meaningful words, rather than of characters such as letters.

Post-filtering

As discussed in Section 7.3, we mean the member methods of a cluster in Name Clustering to be the crosscutting methods in a valid instantiation of the working definition of crosscutting (**Case 2**). We further take the classes those member methods are implemented in form the set of implementing classes. This allows us to implement **Case 2** to its full extent, in defining filters that discriminate against clusters.

Firstly, singleton clusters are filtered away because they do not meet the **Scattering** condition. Secondly, the **Tangling** clause is deployed to discriminate against clusters. Due to the near-overwhelming amount of clusters that Name Clustering returned in early experiments, we have realized a more stringent implementation of **Tangling**. Rather than considering a class to be tangled if the methods it implements don't all belong to the same cluster, a class is tangled if its methods belong to more than one *composite* cluster. Only those clusters that crosscut a tangled class are considered in the rest of the analysis. Next, any cluster of which the members are polymorphic methods is filtered away. Finally, we ignore a cluster if its member methods are implemented in classes that sit closely together in the same hierarchy (**Scattering**).

Scalability

Especially because Name Clustering analyzes the full set of methods in the legacy system, scalability measures are at order. A simple distance metric aides in this, namely that of string identity. It checks whether two strings are identical, by comparing their symbols in pairwise fashion. This metric, with its linear complexity, is less expensive than the similarity algorithm Name Clustering is based on (which is quadratic). Additionally, two strings can often be judged non-identical, on the basis of the first two symbols. Hence, we deploy the identity metric in a single pass clustering scheme, to group identically-named methods in a single, inexpensive pass over the objects. This idea has already been explored in Section 7.1.1.

Cluster Feature

- **Feature of an Object** The name of that method (which is a string)
- **Feature Derivation** Taking the common substring of two strings

(As discussed in Section 7.1.2.)

7.7 A Comparison with Existing Aspect Mining Techniques

In this section, we compare our work to other aspect mining techniques. We consider three criteria in this comparison: (1) whether an aspect mining technique is semantical, (2) whether it is dynamic or static, and (3) whether it considers tangling in its implementation. Table 7.4 gives a comparison table for these criteria; it includes our own aspect mining techniques, as well as those by other researchers.[2]

As throughout this whole dissertation, we adopt the stance that a structural technique, which analyzes the AST of the code, is not necessarily a semantical technique. This explains the '-' mark for AST-based clone detection. Most existing aspect mining techniques are semantical, in that they analyze method calls, control and data flows (in PDG-based clone detection), or execution traces. By this standard, one could claim our own Ngram, Tree and Name Clustering schemes to *not* be semantical. However, these techniques become semantical (to a certain degree) through the pre- and post-filtering steps that we derived from the working definition of crosscutting. What's more, these

[2]The comparison table was partly adopted from the aspect mining survey by Kellens et al. [KM05]

	dynamic?	semantical?	tangling?
Unique Methods (Section 5.3.1)	-	+	-
Fan-in Analysis (Section 5.3.2)	-	+	-
Natural Language Processing (Section 5.3.3)	-	-	-
Patterns of Execution Traces (Section 5.3.4)	±	+	-
AHC Method Names (Section 5.3.5)	-	-	-
AHC Method Invocations (Section 5.3.6)	-	+	-
FCA Execution Traces (Section 5.3.7)	+	+	+
FCA Identifiers (Section 5.3.8)	-	-	-
Clone Detection PDG (Sections 5.3.9 & 5.2.2)	-	+	-
Clone Detection AST/Token (Sections 5.2.3 & 5.2.4)	-	-	-
Call Clustering	-	+	+
Type Clustering	-	+	+
Ngram Clustering	-	±	+
Tree Clustering	-	±	+
Name Clustering	-	±	+

Table 7.4: A comparison of aspect mining techniques.

pre- and post-filtering steps explicitly discriminate against cases where no tangling is detected. This contrasts with the majority of existing aspect mining techniques, which are oblivious to the phenomenon of tangling. (The notable exception is the formal concept analysis of execution traces.)

A dynamic analysis requires run-time information, whereas a static analyis does not. Our techniques are static. In fact, most existing aspect mining techniques are static. This is perhaps due to the inherent limitations of a dynamic analysis. One needs to be able to compile and run the software system under investigation, while it may also be necessary to instrument that system so its execution can be traced. Dynamic analysis will also consider a subset of the software system. As a downside, some possible execution paths will not be considered. As an upside, infeasible execution paths will likewise be ignored. Another advantage to using a dynamic analysis, is that the analysis can benefit from more specific information than its static counterpart: given a particular use case, the resulting program execution is deterministic – it has only to be observed.

7.8 Summary

We have discussed our general approach to cluster analysis. This discussion includes an assessment of the suitability of existing cluster algorithms for their use in our aspect mining techniques. Additionally, we have introduced custom cluster algorithms.

We have introduced five new aspect mining techniques that use cluster analysis to automatically mine for crosscutting concerns in the legacy system. Each technique still requires a post-processing step in which the clusters must be manually investigated for crosscutting behaviour. The introduced techniques differ in the code properties they analyze. While Call Clustering traces method calls, Tree Clustering compares parse trees for structural similarity. Type Clustering looks for common types amongst method bodies, while Name and Ngram Clustering look for lexical similarity in method names and bodies, respectively.

We concluded this chapter with a comparison of our work with aspect mining techniques authored by other researchers. This comparison was carried out in terms of analysis properties. In the next chapter, we discuss the experiments we have carried out with the discussed aspect mining techniques and cluster algorithms.

Chapter 8

The Experiments

In this chapter we detail the experiments we have carried out. We have run these experiments on JHotDraw, the Java system discussed in Chapter 6. Our experimentation work is two-fold. In the previous chapter, we introduced five new aspect mining techniques. Furthermore, we presented the eight cluster algorithms that we have implemented. Each of the aspect mining techniques needs to be driven by a cluster algorithm. Therefore, we have run some tests on subsets of JHotDraw, in an effort to match each aspect mining technique to the best suited cluster algorithm. However, as the main experiments of this dissertation, we have deployed each aspect mining technique on the (near) entirety of JHotDraw. The purpose of these main experiments is to get an idea of the weaknesses and strengths of our aspect mining techniques, relative to one another and to work by other aspect mining researchers.

8.1 Matching Cluster Algorithms to Aspect Mining Techniques

In this section we relate how each aspect mining technique we designed was matched to a cluster algorithm. Concretely, for each of the aspect mining techniques, we:

1. chose a suitable subset of JHotDraw;
2. repeatedly deployed the technique on this subset, each time using a different cluster algorithm;
3. computed the fittest clustering in each resulting hierarchy, i.e. the clustering with the highest silhouette coefficient (see Section 7.1.3 for our motivation in using this coefficient);
4. compared the silhouette coefficients of the fittest clusterings.

We do not deem it undesirable that the silhouette coefficient does not include a notion of crosscutting, because this part of our experimentation work strictly compares cluster algorithms. Table 8.1 relates the aspect mining techniques and the silhouette coefficients in which they resulted. (A high silhouette coefficient indicates higher cohesion and separation.) Recall that MaxLink, MinLink and GroupAverage are instantiations of the Lance-Williams framework (Section 4.5.1), while a cluster feature solution is domain-specific to one of the aspect mining techniques (Section 7.1.2). Table 8.2 gives the subsets of JHotDraw the techniques were deployed on, together with a measure of size of those subsets. The subsets are sets of Java packages.

For each aspect mining technique, the highest silhouette coefficient in Table 8.1 is given in bold. Recall from Section 4.5.4 that GroupAverage is a scheme that considers the aggregate inter-connectivity of clusters. In the cases of Tree and Name Clustering, if it ever becomes desired to experiment with more cluster algorithms, the comparatively high scores of GroupAverage suggest that it could be wise to investigate those algorithms that similarly consider aggregate inter-connectivity. (An example of such an algorithm is ROCK, Section 4.5.3.)

Table 8.3 relates how each cluster feature solution scored, relative to the other cluster algorithms. We conclude that these solutions mostly scored comparatively well, with the major exception of Ngram Clustering.

Table 8.4 relates which cluster algorithm we matched to which aspect mining technique. Note that we did not always opt for the algorithm with the highest coefficient. This is because of a particular benefit of cluster features. The feature of a cluster is a common property of all elements in that cluster; it is a direct summary of the cluster. We believe this to be valuable in manually processing, and discussing aspect mining results, because we can link cluster features to concerns. For example, consider a cluster in Name Clustering. It consists out of different methods that are dedicated to adding, removing and notifying Observers (Section 6.3.1 for a discussion of this pattern). The feature of that cluster can be the common substring “Observer”, which immediately relates the “meaning” of the cluster. Obviously, the feature of a cluster can also be computed in a post-processing step, after the cluster analysis. However, when this step is carried out after another algorithm than the cluster feature solution, there is no guarantee that the computed cluster feature is meaningful.

8.2 The Aspect Mining Experiments

This section relates the experiments we have carried out to investigate the relative (dis)advantages of our aspect mining techniques, in terms of the crosscutting concerns they can, and can not detect. We have deployed each technique on the subset of JHotDraw outlined in Table 8.5. (Note that this subset constitutes the bulk of JHotDraw.) Additionally, we have manually classified the resulting clusters according to whether we thought them to uncover crosscutting. We discuss these results in the next chapter.

	MaxLink	MinLink	GroupAverage	cluster feature
Ngram Clustering	**0.295239**	0.263731	0.284559	0.254483
Tree Clustering	0.627397	0.629288	**0.629905**	0.629485
Call Clustering	0.664888	0.65918	0.658789	**0.668328**
Type Clustering	0.378424	0.377504	0.378144	**0.379659**
Name Clustering	0.298972	0.292045	**0.303215**	0.298972

Table 8.1: Comparing silhouette coefficients for each aspect technique.

	packages	number of methods
Ngram, Tree & Name Clustering	applet, application, figures, samples	551
Call & Type Clustering	framework, standard	871

Table 8.2: The subsets of JHotDraw that were used to evaluate the cluster algorithms.

	relative score of the cluster feature solution
Ngram Clustering	worst
Tree Clustering	second best
Call Clustering	best
Type Clustering	best
Name Clustering	second best

Table 8.3: The relative score of each cluster feature solution.

	cluster algorithm
Ngram Clustering	MaxLink
Tree Clustering	cluster feature
Call Clustering	cluster feature
Type Clustering	cluster feature
Name Clustering	cluster feature

Table 8.4: How we matched each technique to a cluster algorithm.

benchmark	packages	number of methods (excl. declarations in interfaces)
JHotDraw	applet, application, contrib, figures, framework, standard, util	± 2700

Table 8.5: The subset of JHotDraw mined for aspects.

Chapter 9

The Aspect Mining Results

In this chapter, we discuss the results of the main experiments we have carried out. In these experiments, which we outlined in Section 8.2, we have deployed our aspect mining techniques on the (near) entirety of JHotDraw. The objective of this discussion is to get an idea of the relative strengths and weaknesses of our techniques, specifically in terms of the crosscutting concerns they can, and can not detect. We carry out this discussion in the light of 23 selected crosscutting concerns. It would lead us too far to discuss how each of these concerns was handled by each technique. Therefore, we seek different perspectives to the results. We first focus on crosscutting concerns that were strictly detected by our techniques, in that we did not know about these concerns before carrying out the experiments. Secondly, we consider technique by technique in the light of the mining results. We then focus on crosscutting concerns that were detected by none of our techniques. Throughout this whole chapter, we continuously compare our techniques with one another in terms of their results. Finally, we compare our results with those of techniques by other researchers.

9.1 Overview of the Results

Tables 9.1 & 9.2 relate crosscutting concerns to each of the five aspect mining techniques we have implemented. The numbers indicate how many clusters we have manually classified as uncovering a particular concern, using a particular technique. (The section numbers refer to the places in Chapter 6 where we discussed the concerns.) Some clusters naturally uncover multiple concerns, and hence contribute to multiple numbers. It is important to note that some concerns in the tables, especially design patterns, encompass distinct instances of the same concern. For example, the table entry for the Observer design pattern denotes distinct instances of this pattern. Comparing the numbers in the tables is not as valuable as first might seem. This is because one technique might provide more clusters related to a single instance of a concern, while another technique might detect more distinct instances of the same concern.

	Section	Tree	Ngram	Call	Type	Name
Undo	6.4.1	3	1	8	8	10
Observer	6.3.1	3	2	3	3	14
Decorator	6.4.3	3	4	0	0	0
Adapter	6.4.4	0	3	4	1	1
Visitor	6.4.5	2	1	0	0	2
Composite	6.4.6	5	3	2	2	5
State	/	0	1	1	0	4
Command	6.4.7	0	0	0	0	4
Error Handling	6.4.2	0	0	0	0	2
Singleton	/	0	0	0	0	0
Persistence	6.4.8	2	0	2	1	4
Serialization	6.4.8	0	0	0	0	1
Cloning	6.4.8	1	0	0	0	2
Drawing/Moving	6.4.8	1	0	0	0	3
Command Execute	6.4.8	0	0	0	0	0
CheckDamage	6.4.8	0	0	0	0	0
WillChange/Changed	6.4.8	0	1	1	0	0
Space	6.4.8	0	0	0	0	0

Table 9.1: The number of clusters detected for each already familiar concern.

	Tree	Ngram	Call	Type	Name
Duplication	2	3	2	0	1
Condition Checking	3	1	1	0	0
Inheritance Gap	3	4	0	1	0
CollectionsFactory	1	0	1	3	0
Concurrency	0	0	0	0	5

Table 9.2: The number of clusters detected for the newly-discovered concerns.

	# clusters returned	avg cluster size	precision	relative recall
Tree	55	7	60%	52%
Ngram	38	3	53%	47%
Call	36	5	67%	43%
Type	53	3	65%	30%
Name	236	6	26%	60%

Table 9.3: Some additional statistics regarding the mined clusters.

Table 9.1 relates the concerns that we discussed in Chapter 6. Table 9.2 relates concerns that we did not know, or did not know the instances of, before we carried out the experiments. Classifying the clusters for crosscutting nature was a manual enterprise. In a sense, there was a natural bias towards (manually) detecting the concerns of Table 9.1. Therefore, we more thoroughly discuss the newly-discovered crosscutting concerns of Table 9.2.

Table 9.3 relates some statistics for each of the experiments we have carried out. One of these statistics is the number of clusters that were returned by a technique. This number includes those clusters that we discarded in manual post-processing, but excludes those clusters that were labeled uninteresting in the automatic post-filtering. Another of the statistics relates the average number of methods in these clusters (the average cluster size). Hence, these two measures relate the scale of the manual labour we had to perform.

Table 9.3 additionally contains two measures that are commonly used in aspect mining: *recall* and *precision*. The measure of recall captures the percentage of aspect candidates that were detected by a technique. *Absolute recall* is relative to the set of all aspect candidates in the legacy system. Since this set is currently not known for any software system of reasonable scale, we have to rely on *relative recall*. This is the recall relative to a selected set of aspect candidates. In our case, the selected set exists out of the concerns in Tables 9.1 & 9.2. The measure of precision is the percentage of outputs that pertain to a crosscutting concern, relative to all outputs. In our case, it is the percentage of clusters that we have manually classified as exposing crosscutting.

A mining technique that simply outputs the complete legacy system has best-case recall (i.e. 100%), as it is sure not to miss any crosscutting concerns. However, the same technique has worst-case precision; it effectively maximizes the scale of the post-processing that the aspect miner has to perform manually. Aspect mining techniques often try to strike a balance between recall and precision.

9.2 Newly-Discovered Crosscutting Concerns

This section gives an overview of the crosscutting concerns that we did not know, or did not know the instances of, before we carried out the aspect mining experiments.

9.2.1 Condition Checking

Of course, not every use of an if-statement classifies as a crosscutting concern. Instead, we are interested in the case where the same condition is checked in multiple methods, and the condition is a likely candidate to be put into an aspect. It is interesting to note the tie between this property of recurrence, and the manner in which cluster analysis also imposes a condition of multiplicity. Tree Clustering found more instances of the Condition Checking concern

public class GraphicalCompositeFigure

```
public Object getAttribute(String name) {
  if (this.getPresentationFigure() != null) {
      return this.getPresentationFigure().getAttribute(name);
  }
  else {
      return super.getAttribute(name);
  }
}
```

public class GraphicalCompositeFigure

```
public void setAttribute(FigureAttributeConstant attributeConstant,
                         Object value) {
  if (this.getPresentationFigure() != null) {
      this.getPresentationFigure().setAttribute(attributeConstant, value);
  }
  else {
      super.setAttribute(attributeConstant, value);
  }
}
```

public class GraphicalCompositeFigure

```
public Object getAttribute(FigureAttributeConstant attributeConstant) {...}
```

public class GraphicalCompositeFigure

```
public void setAttribute(String name, Object value) {...}
```

public class GraphicalCompositeFigure

```
protected void initialize()  {...}
```

Cluster 1

```
public HandleEnumeration handles() {
  List handles = CollectionsFactory.current().createList(fPoints.size());
   ...
}
```

Figure 9.1: The CollectionsFactory concern crosscuts the `handles` method.

than the other techniques. This could be explained by the fact that condition checking can always be traced back to if-like language constructs, which are structures. Cluster 1 is an example of a Condition Checking cluster detected by Tree Clustering. In this example, the condition is not implemented through a recurring pattern of more than one method, as could be detected by Call Clustering.

9.2.2 Duplication

This concern manifests itself through methods that display a degree of lexical similarity, in a manner that these methods indicate copy-paste practice. In classifying clusters as instances of this concern, we limit ourselves to cases where the lexically similar methods sit in apparently unrelated classes, or the software system could potentially benefit from some modularization to limit, or eliminate the duplication. Admittedly, several different concerns fall into this category, but from an aspect mining perspective, the important angle remains the symptom(s) through which the specific concerns manifest themselves. Thus, we see fit to create this more general category.

9.2.3 CollectionsFactory

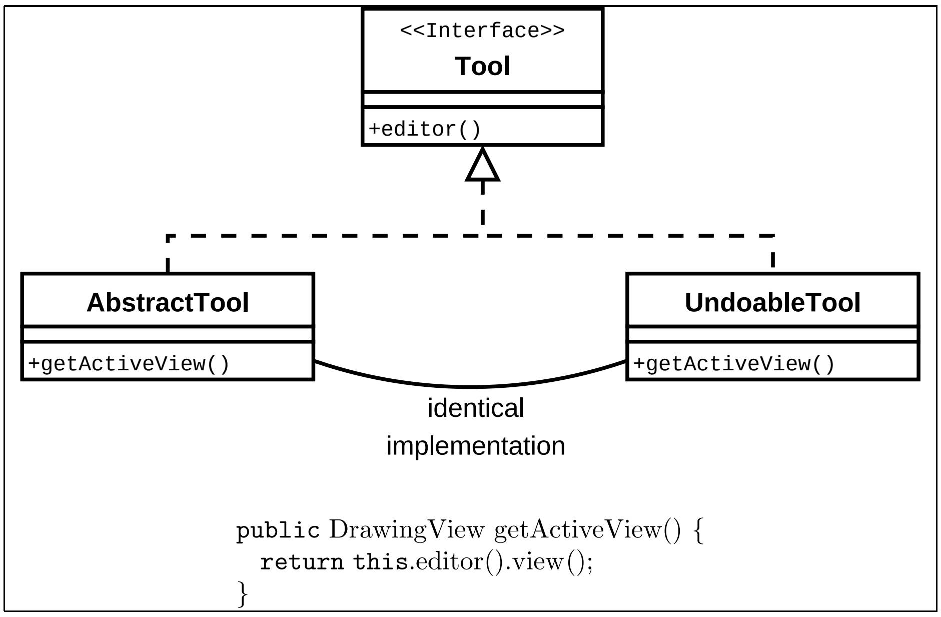
The CollectionsFactory concern is (what we believe to be) one of the more flagrant crosscutting concerns in JHotDraw. This makes it interesting to note how this concern has apparently not, to date, been discussed by other aspect mining researchers.

The CollectionsFactory abstract class maintains a static field `factory`, of type CollectionsFactory. The run-time value of this field is:

a CollectionsFactoryJDK12 if the JHotDraw code is deployed with the 1.2 version of the Java Development Kit.

a CollectionsFactoryJDK11 if the JHotDraw code is deployed with the 1.1 version of the JDK.

These two subclasses of CollectionsFactory take a different approach in creating basic data structures such as lists, maps and sets. Consequently, throughout the



```
public DrawingView getActiveView() {
  return this.editor().view();
}
```

Figure 9.2: An instance of Inheritance Gap: identical implementations lacking a common ancestor.

JHotDraw code, when some basic data structure is to be created, the following happens:

1. the `current` static method of the CollectionsFactory class is called – this is the accessor method for the `factory` field;
2. of the value that results in the previous step, the desired creation method is called (e.g. `createList()`, `createMap()`, `createSet()`).

An example of this pattern is given in Figure 9.1. Type Clustering detects several clusters related to CollectionsFactory. The feature of such a detected cluster contains two types: the CollectionsFactory type, bundled together with another type. Call Clustering detects clusters with two callees each; one of the callees is always `current`, called together with a creation method of CollectionsFactory (e.g. `createList`).

9.2.4 Inheritance Gap

By an inheritance gap, we (intuitively) mean a collection of similarly-named methods that have no coupling between them, in the sense that there is no shared declaration or common super-implementation; even though the similarly-named methods are implemented in the same hierarchy, and the software system could benefit from these methods being given such a common ancestor. One could argue that an inheritance gap constitutes a refactoring opportunity, rather than a real crosscutting concern.

Inheritance Gap poses itself in several instances in JHotDraw. An example is given in Figure 9.2. The AbstractTool and UndoableTool classes have identical implementations of the `getActiveView` method. Additionally, there is no common method (declaration or implementation) that both these methods derive from. That is to say, there is no physical coupling between them. The `editor` method called in the identical implementations is declared in the Tool interface, which AbstractTool and UndoableTool both implement. Considering this case, the question arises whether the duplication concerning `getActiveView` could be eliminated through a common superclass for AbstractTool and UndoableTool (which there currently isn't). If this solution is not feasible, at least a declaration of `getActiveView` in the Tool interface would provide a tighter coupling between both implementations. (AbstractTool and UndoableTool are the only classes that directly implement Tool, and the `getActiveView` method logically fits into this interface)

9.2.5 Concurrency

The concern of Concurrency manifests itself both through logic that crosscuts methods (e.g. a DrawingView being frozen upon certain mouse events), as through Concurrency-dedicated methods that crosscut general classes (e.g. the `lock` method in StandardDrawingView).

The results row for the Concurrency concern in Table 9.2 follows an interesting pattern: only with Name Clustering were clusters related to this concern detected. In classifying these clusters, Concurrency was clearly identifiable through the names of the methods that implement it. The keywords "lock", "Periodicity", and "Thread" gave this concern away instantly. In fact, we had no notion of there being a concurrency mechanism in JHotDraw, until the results of the Name Clustering experiment made it clear. We performed the Name Clustering experiment after all the other experiments. A natural question arises: did the other techniques really miss Concurrency, or did we miss it in manually classifying the clusters? The latter hypothesis appears not unlikely. Working back from the information provided by Name Clustering, we were able to recognize several methods as being part of Concurrency, where we formerly believed these methods to be uninteresting. This human oversight was due to the fact that we saw these methods being called, but they did not make their intent clear from their names alone.

We conclude that there is room for human error in manually inspecting clusters for crosscutting nature. This is especially true if methods do not immediately make clear, e.g. from their names, to what concern they belong. Furthermore, using several analysis techniques on the same case might bring interesting findings to the surface, that are otherwise missed if only a limited number of analyses is used.

public class ElbowHandle `private int` constrainX(`int` x) {...}
public class ElbowHandle `private int` constrainY(`int` y) {...}

Cluster 2

9.3 Logic Clustering

9.3.1 Ngram Clustering

With respect to the other cluster analysis techniques, more clusters in the Ngram experiment were classified as detections of the Duplication concern. Ngram Clustering is the most lexical Logic Clustering technique we have implemented, precisely grouping method bodies for lexical similarity. It is hence not surprising to see it excel at detecting cases of (lexical) code duplication.

However, a downside to the approach of Ngram Clustering can be seen in the false positives of the experiment. While these clusters uncover lexical duplication, we do not think the clustered methods to exhibit crosscutting behaviour. Cluster 2 shows such a false positive. It is clearly just the case of a class deciding to split a certain operation that is made on (x,y)-coordinates (the `private` modifier adds to the argument that the two methods are utility methods). It is interesting to note how each of the manually rejected clusters exists out of methods that are either implemented in the same class, or that sit closely together in the same class hierarchy. This is natural, as a reasonably well-modularized system will (try to) put similar code entities in the same module. On the other hand, a good amount of the concerns in Table 9.1 manifest themselves precisely in the bodies of methods that belong to the same class or hierarchy (e.g. the Space, Command Execute and WillChange/Changed concerns). There is hence no base for a strict in-or-out filter discriminating against methods that are hierarchically close to each other. Instead a weighted approach in filtering could be useful.

Another danger of Ngram Clustering is how a recurring pattern can be lost in code, if that pattern only constitutes a fraction of the method bodies it appears in. This is exemplified by the CheckDamage concern. It is a single method call occurring near the end of several different-purpose methods. Consequently, the similarity between these methods is generally minimal from a code duplication perspective. It is therefore difficult to group them together in cluster analysis. We presume the Command Execute and Space concerns to have fallen prey to the same effect, in the case of Ngram Clustering.

```
public static class TriangleRotationHandle.UndoActivity
↪ protected void setRotationAngle(double newRotationAngle) {...}
public class TriangleFigure
↪ public double getRotationAngle(...) {}
```

Cluster 3

9.3.2 Call Clustering

Call Clustering is the most semantical analysis technique of the ones we have implemented. Additionally, it applies our working definition of crosscutting to its fullest extent. A natural question arises: what is the trade-off concerning these properties? A first observation is that Call Clustering's precision is higher than that of any other cluster analysis technique. But this seems to come at a cost: the relative recall is lower.

According to Irish (the analysis workbench we deployed), 506 out of 2361 non-constructor methods in JHotDraw are never called. Hence they do not engage in the could-call relationship that Call Clustering is founded on, which means that they are ignored in the analysis. As a consequence, the same discussion is at hand as with dynamic vs. static analysis (see Section 7.7). Call Clustering has a better grasp on what constitutes feasible computation paths. However, a software system can "hide" a crosscutting pattern if it is not, or only partly deployed. This is especially of note for software systems that are frameworks, as is the case. We did not personally observe such "hiding" behaviour, but nonetheless keep in mind that it could have occurred.

With respect to Tree and Ngram Clustering, we see a higher amount of clusters related to the Undo concern. We thus see fit to discuss Call Clustering in terms of Undo. In Call Clustering, eight clusters were classified as pertaining to Undo. For six of these, it is the case that one callee belongs to an UndoActivity, while another callee belongs to another class (such as the corresponding Activity). One of these six clusters is Cluster 3.

Cluster 3 was not found in the Tree and Ngram experiments. The reason for this can be sought in the above-discussed shortcoming of techniques that trace code duplication (see the discussion of Ngram Clustering, Section 9.3.1). The methods in Cluster 3 encompass only a fraction of the method bodies that call them. The similarity between these bodies is minimal from a code duplication perspective. In Call Clustering on the other hand, the distance between methods is based on the degree to which they are called together. Hence, the link between the callees of a cluster can be strong even if the link between the callers isn't.

Another observation can be made regarding Call Clustering's Undo clusters. Five of them, of the eight total, exist out of getters and setters only. Another Undo cluster exists out of methods that implement the FigureEnumeration interface, which are clearly utility methods. We have noted a certain tendency in aspect miners to filter out getters, setters and utility methods (e.g. fan-in

public class EllipseFigureGeometricAdapter

```
public Shape getShape() {
  Rectangle rect = this.displayBox();
  Ellipse2D.Float  ellipse  = new Ellipse2D.Float (...);
  return ellipse;
}
```

public class DrawApplication

```
protected DrawingView createDrawingView(Drawing newDrawing) {
  Dimension d = this.getDrawingViewSize();
  DrawingView newDrawingView = new StandardDrawingView(...);
  newDrawingView.setDrawing(newDrawing);
  return newDrawingView;
}
```

Cluster 4

analysis, Section 5.3.2). This filtering is done for fear of such methods otherwise cluttering the mining results. However, in this case these methods constitute 7/8th of the clusters that cover the Undo concern.

9.3.3 Tree Clustering

The usual question of trade-off arises for this most structural analysis technique. Naturally, a structural commonality between methods does not necessarily imply any interesting relation between them. One false positive is illustrated in Cluster 4. The upper method is concerned with getting the shape of a geometric figure, while the lower is about a DrawApplication creating a new DrawingView. As a related problem, there is the greater possibility of an interesting cluster not being pure, i.e. the cluster is more likely to contain uninteresting methods as well. In a general sense, the distance measure deployed in Tree Clustering is “broader” than those of the other cluster analysis techniques. It more easily explores relations between code entities that are “far apart”, in that they belong to more different concerns.

Comparatively, Tree Clustering found a good amount of clusters related to such crosscutting concerns as Composite, Observer and Decorator. This suggests there to be a link between structures and concerns. Figure 9.3 shows the feature of Cluster 5 (a cluster which was found in Tree Clustering). This feature is the common tree structure amongst all the methods in Cluster 5. The different methods in this cluster all implement the notification code in a Subject of an Observer instance. They exhibit looping, casting and message sending behaviour in notifying the Observers. This shows that clusters in Tree Clustering can very well be pure. What’s more, the “broadness” property of

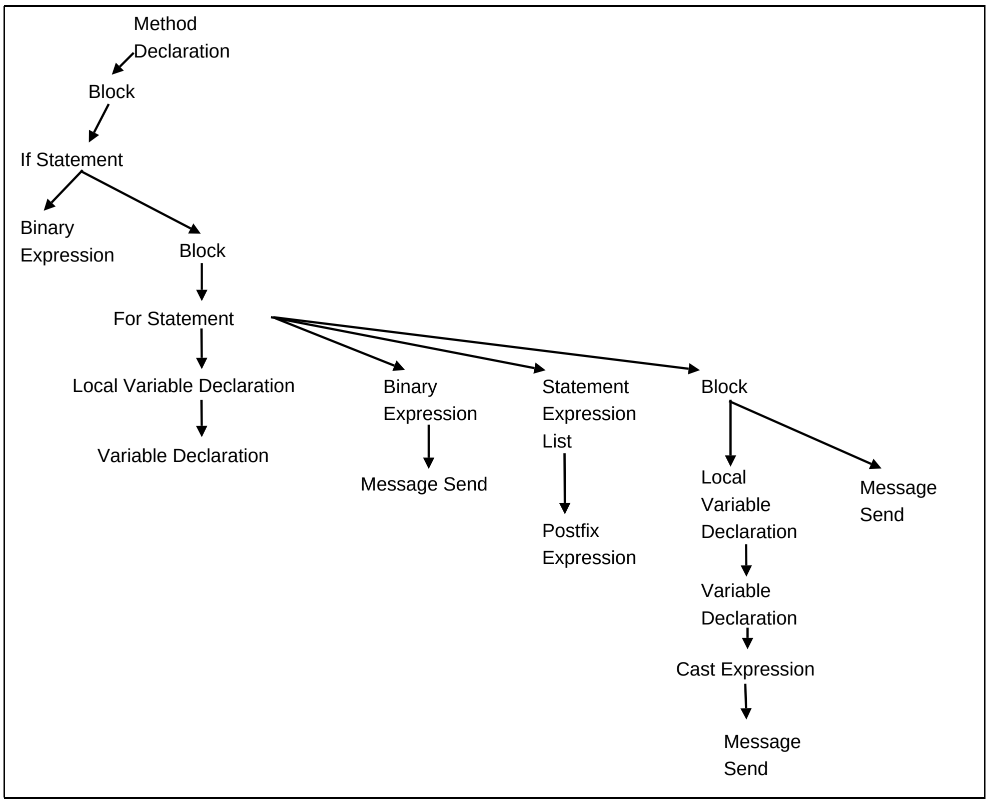


Figure 9.3: The common tree structure amongst the methods in Cluster 5.

public class StandardDrawingView

```
protected void fireSelectionChanged() {
  if ( fSelectionListeners != null) {
      for (int i = 0; i < fSelectionListeners . size (); i++) {
        FigureSelectionListener l =
          (FigureSelectionListener) fSelectionListeners .get(i );
        l .figureSelectionChanged(this);
      }
  }
}
```

public class StandardDrawing

```
public void figureInvalidated(FigureChangeEvent e) {
  if (fListeners != null) {
      for (int i = 0; i < fListeners. size (); i++) {
        DrawingChangeListener l =
          (DrawingChangeListener) fListeners.get(i);
        l .drawingInvalidated(new DrawingChangeEvent(this,
          e.getInvalidatedRectangle ()));
      }
  }
}
```

public class StandardDrawing

```
public void fireDrawingTitleChanged(...) {...}
```

public class StandardDrawing

```
public void figureRequestUpdate(FigureChangeEvent e) {...}
```

Cluster 5

public class ArrowTip

```
public void write(StorableOutput dw) {
  dw.writeDouble(this.getAngle());
  dw.writeDouble(this.getOuterRadius());
  dw.writeDouble(this.getInnerRadius());
  super.write(dw);
}
```

public class ArrowTip

```
public void read(StorableInput dr) throws IOException {
  this.setAngle(dr.readDouble());
  this.setOuterRadius(dr.readDouble());
  this.setInnerRadius(dr.readDouble());
  super.read(dr);
}
```

. . .

Cluster 6

Tree Clustering makes the technique more likely to group different instances of the same crosscutting pattern. Note how the first method in Cluster 5 is about a StandardDrawingView notifying FigureSelectionListeners, while the second method is about a StandardDrawing notifying DrawingChangeListeners.

Additionally, Tree Clustering groups methods that other cluster analysis techniques couldn't. Often, these methods are reverse actions of one another. This is clearly illustrated in the cluster which literally groups the method `action` with `reverseAction` (as a detection of Undo and Decorator). Similar couples of reversals, that Tree Clustering was also able to detect, are:

- **undo/redo** (as a detection of Undo)
- **add/orphan** (Composite)
- **removePointAt/insertPointAt** (Composite)
- **bringToFront/sendToBack** (false positive)
- **read/write** (Persistence)

Admittedly, precisely half of the mentioned reversals sit in clusters that contain additional, unrelated "noise" methods; those clusters are not pure. Let us discuss the last couple, of `read` and `write`. Cluster 6 illustrates the cluster within which this couple was found. The methods `read` and `write` do not share a single method call, nor are they called together, as could be exploited by Call Clustering. From the perspective of Type Clustering, there is only one common (static) type amongst the bodies of `read` and `write`, namely the type of the `super` keyword. (Remember that Type Clustering ignores the type of ArrowTip itself as an attribute for cluster analysis, because `read` and `write` are implemented in that class.) This single commonality in terms of types - in the context of `read` and `write` both containing other types – makes it more difficult for Type Clustering to group these methods together. Finally, the question remains whether Ngram Clustering manages to exploit the degree of lexical similarity that is undeniably present in both method bodies. The answer is negative. There are no common tokens in `read` and `write` other than "public", "void", "this" and "super". These are not very meaningful. If tokens were partitioned according to the occurrences of capital letters, more commonalities would arise: "Double", "Angle", "OuterRadius", etc. However, in Ngram Clustering we look for common Ngrams in a pair of method bodies. Two mismatching words break such a potentially common Ngram. For example, the "get" and "set" in front of "getInnerRadius" and "setInnerRadius" prohibit these two methods names of being the suffix of a common Ngram. Hence, for Ngram Clustering to detect `read` and `write` methods, we would have to adjust the "n" parameter which determines the length of the Ngrams. In fact, it would have to be so low the technique would no longer resemble an Ngram Clustering technique.

The relative recall of Tree Clustering can be seen as an added argument that a "broader" distance measure can be a valid perspective. At this measure, Tree Clustering comparatively scores second-highest.

9.3.4 Type Clustering

Type and Call Clustering can both be considered semantical analysis techniques. In the cases of all but one of the concerns in Tables 9.1 and 9.2, Type Clustering finds less related clusters than Call Clustering. Some results suggest this to be a matter of false negatives in Type Clustering. Especially the (comparatively) low relative recall of Type Clustering can be seen as an argument to this.

A pattern of static receiver types is more general than a pattern of methods. For example, it is likely that the WillChange/Changed concern was missed due to this property of generality; strip this pattern of the actual methods being called, and it becomes a pattern of the AbstractFigure type being used (mostly) in AbstractFigures – this is not a very distinguishable pattern.

Because Type Clustering is in a sense more general, it also finds clusters that Call Clustering couldn't. A good example of this is Cluster 7, which was found in Type Clustering. Each of the three methods in this cluster notifies ViewChangeListeners of a particular event. The only method that is called in all three of these methods is `getListenerList`. Because this is a single-call pattern, Call Clustering is not able to mine it. Type Clustering does recognize the link between the three methods, as it notices the two-type pattern of ViewChangeListener & EventListenerList.

public class DrawApplication

```
protected void fireViewCreatedEvent(DrawingView view) {
  final Object [] listeners = listenerList . getListenerList ();
  ViewChangeListener vsl = null;
  for (int i = listeners .length − 2; i >= 0; i −= 2) {
    if ( listeners [i] == ViewChangeListener.class) {
        vsl = (ViewChangeListener) listeners[i + 1];
        vsl . viewCreated(view);
    }
  }
}
```

public class DrawApplication

```
protected void fireViewDestroyingEvent(DrawingView view) {
  final Object [] listeners = listenerList . getListenerList ();
  ViewChangeListener vsl = null;
  for (int i = listeners .length − 2; i >= 0; i −= 2) {
    if ( listeners [i] == ViewChangeListener.class) {
        vsl = (ViewChangeListener) listeners[i + 1];
        vsl . viewDestroying(view);
    }
  }
}
```

public class DrawApplication

```
protected void fireViewSelectionChangedEvent(DrawingView oldView,
                                             DrawingView newView) {
  final Object [] listeners = listenerList . getListenerList ();
  ViewChangeListener vsl = null;
  for (int i = listeners .length − 2; i >= 0; i −= 2) {
    if ( listeners [i] == ViewChangeListener.class) {
        vsl = (ViewChangeListener) listeners[i + 1];
        vsl . viewSelectionChanged(oldView, newView);
    }
  }
}
```

Cluster 7

9.4 Name Clustering

Despite the post-filtering phase of the analysis, Name Clustering still returned 236 clusters total. This is a large amount of clusters to investigate manually. Especially since this work is generally quite tedious and error-prone. So in this respect, a higher degree of automatic filtering is desired. The (comparatively) very low precision of Name Clustering suggests that there is room to make such a correction. One might make an objection regarding the filtering that is already in place in Name Clustering. More precisely, it is perfectly possible that the WillChange/Changed concern was rejected as a cluster on the basis of these two methods being in the same class. However, Name Clustering was ultimately not tailored to find logic that crosscuts methods.

9.5 Selected Crosscutting Concerns

9.5.1 Error Handling

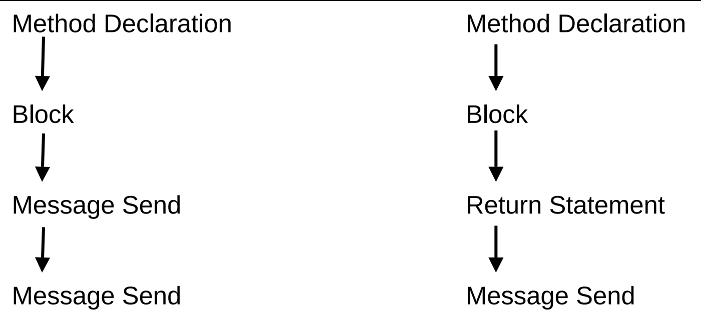
The Error Handling concern was only found through Name Clustering. This technique detected two naming patterns related to error handling: (1) the convention to prepend "JHotDraw" to the names of JHotDraw's custom exceptions, and (2) the "getNested" keyword that stems from how those custom exceptions are wrapped around an inner, nested exception.

The failure of the other techniques can be traced back to the pre-filtering step that precedes all Logic Clustering methods (Section 7.5.1). Error Handling in JHotDraw is based on the exception handling system of Java. Consequently, it is primarily an interplay between `try-catch` and `throw` clauses, where exceptions are created through the act of instance creation. This does not involve method calls. To determine whether a method body potentially displays tangling, the pre-filtering step examines only the method calls in that body. Hence, the filtering generally rejects methods that are only crosscut by the concern of Error Handling, because it does not "smell" the crosscutting.

9.5.2 Decorator

In aspect mining, the Decorator pattern can be particularly difficult to unearth. This is due to the nature of this design pattern: the decoration of base functionality goes under the disguise of that base functionality.

For example, the adding of a border to a Figure is taken care of in the class BorderDecorator. However, no variable in JHotDraw has BorderDecorator as its static type. Rather, run-time instances of BorderDecorator are held in variables of a broader static type, such as Figure. This is a great obstacle for Type Clustering, which clusters methods based on the static variable types their bodies contain. Likewise, Call Clustering is not likely to return clear instances of Decorator, as it relies on static variable types to draw the could-call relations. For example, consider any particular method of BorderDecorator.



```
public void moveBy(int x, int y) {
  this.getDecoratedFigure().moveBy(x, y);
}

public Object getAttribute(String name) {
  return this.getDecoratedFigure().getAttribute(name);
}
```

Figure 9.4: Tree structures associated with redelegation, with a matching code example for each.

public abstract class DecoratorFigure

```
public void draw(Graphics g) {
  this.getDecoratedFigure().draw(g);
}
```

public class UndoableHandle

```
public void draw(Graphics g) {
  this.getWrappedHandle().draw(g);
}
```

. . .

Cluster 8

Possibly many methods could call it, but generally these methods could call the same method in any other subtype of Figure. Consequently, there is no foundation for discriminating methods of concrete Decorators in the process of Call Clustering. Since a concrete Decorator in JHotDraw implements not many other - if any other - methods than those of the type it decorates (e.g. Figure), Name Clustering fails to distinguish Decorator due to the same problem of indistinguishability (there are after all many concrete Figures in the system).

On the other hand, Tree Clustering and Ngram Clustering both found an angle to approach the Decorator pattern. In Tree Clustering, we can associate two structures with the principle of redelegation. These are given in Figure 9.4, with a matching code example for each. Admittedly, these are very general structures. Nonetheless, Tree Clustering detected two clusters of which these structures are the features. In each cluster, 40% of the methods belongs to a Decorator. Furthermore, in either cluster 5 distinct instances of the Decorator pattern are uncovered.

In Ngram Clustering, the similar methods of different instances of Decorator were grouped. This is exemplified in Cluster 8 (found in Ngram Clustering), where DecoratorFigure and UndoableHandle are both instances of the Decorator pattern. In addition, Ngram clustering also uncovered the glue code that wraps the decoration of BorderDecorator around Figures. This glue code can be difficult to find using a semantical method that strictly considers method calls, as glue code is primarily a pattern of instance creation.

9.6 Undetected Crosscutting Concerns

Some concerns in Tables 9.1 demand our attention because they were detected by none of the cluster analysis techniques we experimented with. These concerns are the following:

- **Singleton**
- **Command Execute**
- **CheckDamage**
- **Space**

The above concerns constitute source code entities that stand in isolation. They are recurring patterns of only a single method call. It is hence difficult for cluster analysis techniques to detect them. Again, from a code duplication perspective, the similarity between the callers of these concerns is generally minimal. Had the Singleton pattern occured in many instances in the JHotDraw code, Name Clustering could have grouped the similarly-named accessor methods for the singleton objects (the authors of JHotDraw follow the convention of naming these methods `instance`). However, only one such accessor method exists in

	Fan-in	Identifier	Dynamic	ACA
Observer	+	+	+	+
Consistent Behaviour	+	-	-	±
Command Execute	+	+	-	-
BringToFront/ SendToBack	-	-	+	?
Adapter	-	+	+	+
Moving	+	+	+	+

Table 9.4: Aggregate Cluster Analysis compared to other aspect mining techniques.

JHotDraw. The above concerns thus stress the major shortcoming of cluster analysis: its condition of multiplicity.

On the other hand, CheckDamage is a crosscutting concerns that manifest themselves in the `execute` methods of the Command hierarchy. Clusters that strictly exist out of these `execute` methods – and could hence uncover CheckDamage - are filtered away in any case. This is due to the polymorphism clause that is part of the post-filtering to all our implemented mining techniques. Similarly, the Space concern does not in fact meet our working definition of crosscutting, because it only crosscuts methods that are implemented in the same class as the `space` method itself. This stresses the trade-off between different degrees of hardness of filtering. Relaxed filters can provide too much information for the researcher to manually inspect. Harsher filters are not always accurate.

9.7 With Respect to Existing Aspect Mining Techniques

In Section 7.7 we compared our techniques with (what we believe to be) all existing aspect mining techniques. This comparison was strictly performed in terms of analysis properties. In this section we discuss how the results of our experiments compare to those of other aspect mining techniques.

We rely on the comparison paper by Ceccato et al. [CMM$^+$05]. In that paper, three aspect mining techniques are compared with one another. This is done in the light of a selected group of representative crosscutting concerns. Ceccato et al. evaluate fan-in analysis (Section 5.3.2), the formal concept analysis of execution traces (Section 5.3.7), and the formal concept analysis of identifiers (Section 5.3.8). The latter two analyses we further call identifier analysis and dynamic analysis, respectively. Table 9.4 shows how we fit our aspect mining results into this comparison framework. We extend the comparison table of Ceccato et al. with a single column. To give a more global perspective, we bundle our different cluster analysis techniques to form Aggregate Cluster Analysis (ACA). As such, ACA is merely a quantifying term, rather than a real technique. Individual comparison measures for each cluster analysis technique can still be retrieved from the previous tables. The Moving concern should be understood to be half of the Drawing/Moving concern in Table 9.1. BringToFront/SendToBack is the concern of sending figures to the back or bringing them to the front of a DrawingView.

Fan-in analysis failed to detect the Adapter design pattern, of which the Handles mechanism is the primary manifestation in JHotDraw (Section 6.4.4). Ceccato et al. explain this failure as follows: the Handles mechanism does not involve one method being called in many different places. Instead, there is a large collection of similar methods. Note how the notions of multiplicity and similarity are central to the domain of cluster analysis. Indeed, all ACA techniques, save Tree Clustering, managed to uncover Adapter. On the other hand, ACA failed to detect Command Execute because this concern constitutes an isolated source code method (Section 9.6). Fan-in analysis does not impose a condition of multiplicity, and it did detect Command Execute. This suggests fan-in analysis and ACA to be highly complementary.

Table 9.4 contains one very general category of crosscutting behaviour: Consistent Behaviour. This category denotes any crosscutting concern that imposes specific functionality on a set of methods, where the set of methods can be captured through a pointcut. When the same functionality is super-imposed on various methods, this often involves the same method being called in many different contexts. Hence, fan-in analysis generally scores well at detecting Consistent Behaviour. Because the super-imposed functionality is often called in every execution scenario, dynamic analysis is typically not able to distinguish Consistent Behaviour by any specific use case. The methods that implement the super-imposed functionality do not necessarily follow a naming convention. Hence the negative mark for identifier analysis.

Many concerns in Tables 9.1 and 9.2 are instances of Consistent Behaviour, e.g. CheckDamage, WillChange/Changed, Space. These concerns were not necessarily handled similarly in our aspect mining experiments (i.e. different concerns were handled differently). Each instance of Consistent Behaviour was detected by at least one cluster analysis technique, unless that instance constitutes an isolated source code entity (as discussed in Section 9.6). This explains the '±' mark for ACA. In a sense, Ceccato et al. admit to Consistent Behaviour not being particularly suitable as an "atomic" concern type: they discuss Command Execute separately, to illustrate that Consistent Behaviour instances *can* be detected by identifier analysis.

Ceccato et al. suggest that researchers first look at any aspect candidate that is reported by a number of different aspect mining techniques. The reasoning is that such candidates are more likely to be real, or significant crosscutting concerns. In that sense, the results of ACA attest to the crosscutting nature of Observer and Moving. However, Ceccato et al. also focus on Moving as a "controversial" concern. Not all contributors to the comparison paper agree on whether it is crosscutting. Some see Moving as part of a Figure's core logic. Similarly, we only consider BringToFront/SendToBack to be crosscutting in that both operations are Commands in the Command hierarchy. Thus, if a cluster uncovered BringToFront/SendToBack, we generally dismissed it as a false positive. Hence the question mark in Table 9.4.

9.8 Conclusions

In Section 9.7, we introduced Aggregate Cluster Analysis (ACA), as the collection of our different aspect mining techniques. As such, ACA is only a quantifying term for use in discussion, rather than a real technique. From the list of crosscutting concerns that we composed before the experiments, ACA failed to detect four concerns. These four concerns constitute source code entities that stand in isolation (Section 9.6). They can hence difficultly be detected using an analysis that groups objects. On the other hand, ACA detected several concerns that we did not know about before carrying out the experiments (despite an extensive literature study and manual investigation of JHotDraw). We outlined the newly-discovered crosscutting in Section 9.2; some of them seem significant (e.g. Concurrency in Section 9.2.5), or flagrant (e.g. Collections-Factory in Section 9.2.3). In the comparison to results of techniques by other researchers, ACA and fan-in analysis seem highly complementary (Section 9.7). This is natural; the major shortcoming of ACA is that it relies on a condition of multiplicity in grouping the methods, while fan-in analysis evaluates each method separately for its value in aspect mining. Conversely, fan-in analysis can be seen to lack in that it does not explore relations between methods.

Our different aspect mining techniques have different strengths. In fact, none of the analyses seems particularly "useless". We hereby provide some selected examples that illustrate this. Tree Clustering found more instances of Condition Checking (Section 9.2.1) and provided particularly many reversals (Section 9.3.3). (Reversals are reverse actions, such as undo/redo). Call Clustering can mine a pattern that constitutes only a fraction of the method bodies it appears in (Section 9.3.2). Type Clustering contrasts with Call Clustering, in that it finds more general patterns (of receiver types, rather than methods). As a consequence, Type Clustering detected crosscutting concerns that Call Clustering couldn't (Section 9.3.4). Ngram Clustering detected more instances of Duplication (Section 9.2.2). Name Clustering is the only technique to have detected Concurrency (Section 9.2.5) and Error Handling (Section 9.5.1).

Conversely, we observed that our aspect mining techniques have different weaknesses. We again provide some selected examples that illustrate this. There tends to be more noise in the clusters of Tree Clustering (Section 9.3.3). Ngram Clustering has the risk of a pattern being lost in code, when it is small compared to the method bodies it appears in (Section 9.3.1). Name Clustering missed Decorator because of the problem of indistinguishability (Section 9.5.2). Call Clustering missed an Observer instance because a pattern of method calls can be too specific (Section 9.3.4). Type Clustering missed the WillChange/Changed concern precisely because it considers more general patterns (Section 9.3.4).

We observed that there is room for error in the manual classification of clusters. While a cluster might technically uncover a crosscutting concern, the aspect miner might not recognize the cluster as doing so. This oversight occurs when the used aspect mining technique does not provide an adequate perspective. For example, only with Name Clustering were we able to detect Concurrency (Section 9.2.5), likely because the analysis highlights easily recognizable keywords, rather than (in the case of Concurrency) muddier information such as method calls or tree structures. This suggests that different perspectives to the legacy system are useful, in terms of the use of several aspect mining techniques.

Chapter 10

Conclusions

10.1 Summary

The common object-oriented programming languages are based on one dominant dimension of decomposition: the class hierarchy. This dimension does not suffice to cleanly encapsulate every concern. As a consequence, crosscutting concerns arise that can not be cleanly encapsulated into modules. Crosscutting concerns are a problem because they violate the separation of concerns. This principle of separation is the idea that the software system be broken down into distinct modules, such that the modules overlap in functionality as little as possible. A software system that adheres to this principle is more comprehensible, adaptable and maintainable. The concept of Aspect-Oriented Programming (AOP) is a proposed solution to the problem of crosscutting concerns. The intent of AOP is to allow for a crosscutting concern to be cleanly encapsulated in an aspect. To this end, AOP languages offer new language constructs to model aspects.

In the light of AOP, there is a demand for aspect mining techniques that automatically mine a legacy system for all crosscutting concerns potentially to be turned into aspects. Legacy systems are characteristically poorly-documented, complex and of large scale. In such systems, it is hard (if not practically unfeasible) to manually mine for aspects. However, at the current time, a fully-automatic aspect mining technique seems unlikely.

Cluster analysis is a technique that aims to group similar objects into the same cluster. The underlying notion of similarity is based on a user-provided (binary) distance measure between the objects. Different cluster algorithms exist that perform cluster analysis.

In this dissertation, we investigated the extent to which cluster analysis can be used to perform aspect mining. We proposed 5 distinct aspect mining techniques that use cluster analysis. At the heart of each technique lies a separate distance measure between the methods in the legacy system. This means that each technique forms clusters of methods. The methods in a cluster are similar in a sense that is particular to the aspect mining technique. The five techniques and their distance measures are:

- **Call Clustering** [Two methods are similar when] they are called together to a certain degree.
- **Type Clustering** They are similar in the types they contain.
- **Ngram Clustering** They have lexically similar bodies.
- **Tree Clustering** They are structurally similar, in that they have similar parse trees.
- **Name Clustering** They have lexically similar names.

It is important to note that each of the techniques still requires a post-processing step of the user, in which he or she manually investigates the clusters.

It was necessary to make a selection among the wide array of existing cluster algorithms. We opted to implement algorithms that do not require the objects to have attributes (numerical or otherwise), as it does not necessarily make sense to partition a parse tree or string into separate attribute values. We further implemented algorithms that do not require input parameters, as they can add extra degrees of uncertainty to the analysis of aspect mining results.

Concretely, we implemented the Lance-Williams framework, which allows to efficiently realize 5 distinct cluster algorithms (MaxLink, MinLink, GroupAverage, Ward and Centroid). Furthermore, the algorithms are to a certain degree representative of other hierarchical algorithms. Additionally, we designed and implemented a domain-specific cluster algorithm for each of our aspect mining techniques. We call such an algorithm a “cluster feature solution”.

We chose JHotDraw [JHo] as the benchmark to evaluate our aspect mining techniques. JHotDraw is a relatively large-scale Java system of around 2800 methods and 18K lines of code. The motivation for our choice is based on two observations: (1) JHotDraw has the reputation that it maximally exploits the means of abstraction and composition that are offered by object-orientation (Java), and (2) JHotDraw has already been used in the evaluation, and comparison of aspect mining techniques.

The set of experiments we carried out is two-fold. Using subsets of JHotDraw, we did some tests as to get an idea which cluster algorithm is most suited for which aspect mining technique. The cluster feature solutions we introduced mostly scored comparatively well. As the main experiments of this dissertation, we then deployed our different aspect mining techniques on the (near) entirety of JHotDraw.

The results of our main experiments suggested a number of conclusions. We showed our aspect mining techniques to have individual strengths and weaknesses, relative to one other. When the different techniques detected the same concern, they often did so in different ways. These observations suggested that it could be worthwhile to consider the different techniques from a non-competitive angle. Thus, we also considered Aggregate Cluster Analysis (ACA), which is the term we use to denote the collection of our aspect mining techniques.

Before carrying out the experiments, we compiled a list of crosscutting concerns in JHotDraw. ACA was able to detect clear crosscutting concerns that are not on the list (i.e. concerns that we were not able to detect through manual investigation, or through the writings of other researchers). Furthermore, of the 18 listed concerns, ACA only missed 4. The undetected concerns have a common property: they constitute source code entities that stand in isolation. (For example, the concern is a recurring pattern of one method call.) Therefore, these concerns are difficult to mine using an analysis that groups objects.

We compared our own aspect mining techniques with those of other researchers, both in terms of analysis properties and results. One distinguishable feature of our work is the degree to which our techniques take tangling into account. Only the formal concept analysis of execution traces [TC04] also considers tangling in its implementation. Due to the paper of Ceccato et al. [CMM$^+$05], we were able to compare our results with those of fan-in analysis [MvDM04] and the two techniques that deploy formal concept analysis [TC04, TM04]. We concluded that ACA seems highly complementary with fan-in analysis. This is natural; the major shortcoming of ACA is that it relies on a condition of multiplicity in grouping the methods, while fan-in analysis evaluates each method separately for its value in aspect mining. Conversely, fan-in analysis can be seen to lack in that it does not explore relations between methods.

10.2 Contributions

We have proposed and implemented different aspect mining techniques. In the implementation, Irish [Fab] serves as the underlying code analysis workbench. Tree Clustering uses tree edit distance to compare the parse trees of methods. To this end, we have implemented a tree edit distance algorithm, from the pseudocode of Zhang and Shasha [SZ97]. Ngram Clustering takes advantage of Irish's reader to extract Ngrams (substrings) of tokens from method bodies, and then uses the Ngrams in a document clustering scheme. Name Clustering compares method names through a dynamic program that finds the longest common substring in a given pair of strings. Type and Call Clustering assume objects and attributes in the cluster analysis, and then use Jaccard distance to group methods based on the degree to which they, respectively, have common types in their bodies, and could be called by the same methods.

Irish allows to query Java code using logic programming queries. We have extended this workbench with more advanced predicates than those that are native to Irish. The most intricate predicate we have implemented determines whether a method could call another, taking into account polymorphism in the inheritance of classes and interfaces. However, due to the cost of such code analysis, in terms of both implementation work and run-time complexity, some of the custom predicates are really approximation measures.

In early experiments, we identified a need to filter the output of the cluster analysis, to get an overall higher-quality aspect mining result. To this end, we have introduced a working definition of crosscutting, from which we have derived various automatic pre- and post-filtering steps that accompany the cluster analysis. The working definition and derived filtering steps consider notions such as tangling, scattering and redelegation. The working definition further specifies a divide-and-conquer approach to aspect mining: when an aspect mining technique is tailored towards one case of crosscutting, more specific filters can be implemented for it.

Due to the cost of the deployed code analysis and clustering schemes, we have implemented several scalability measures. We have devised a general strategy in which the use of an expensive distance measure is limited through the use of a lower bound to that measure. We have designed one such lower bound, to the tree edit distance. Furthermore, we have designed several scalability approaches in which single pass clustering is deployed as a pre-processing step to the main cluster analysis.

Besides our implementation of several existing cluster algorithms, we have designed our own cluster feature solutions. To realize the cluster feature solution to Tree Clustering, we have designed an extension to the tree edit distance algorithm of Zhang and Shasha. This extension constructs the optimal edit script between trees.

We have integrated the StarBrowser [Wuy] in our implementation work, in order to automatically present the user with a browsable presentation of the aspect mining results. This presentation provides quick access to context information regarding the clusters that were formed.

Another of our contributions is a list (and a discussion) of the crosscutting concerns in JHotDraw. This list was compiled through an extensive literature study and manual investigation.

We have made an assessment of the suitability of numerous cluster algorithms for their use in our aspect mining techniques. We have done this through literature study and experimentation work.

Ultimately, we have performed experimentation work to compare our aspect mining techniques for their relative strengths and weaknesses. Through literature study, we have compared our aspect mining techniques with those authored by other researchers, in terms of mining results and analysis properties.

10.3 Possibilities for Future Work

Aggregate Cluster Analysis is merely a term to denote a collection of aspect mining techniques. It should be interesting to implement an actual combination of the different techniques. However, it is difficult to imagine an approach that would preserve the individual strengths of the techniques, without completely preserving the techniques themselves.

Naturally, there is room for experimentation with other distance measures. For instance, points-to analysis [Ryd03] could be deployed. This analysis links a variable reference to the "potential" run-time objects it could refer to. Laying these links involves notions of uncertainty and potentiality, because points-to analysis is in fact a *static* analysis. That is to say, it does not require run-time information. In terms of mining aspects using clusters, we imagine a scheme where two (source code) methods are grouped on the basis of the "potential" objects they could both send messages to.

Additionally, the aspect mining filters in our own implementation work can be modified or extended. In the Ngram experiment, every cluster that was falsely labeled to be crosscutting existed out of methods that are implemented closely in the same hierarchy. This suggests a new filter, although there are arguments against a strict filtering of "same-hierarchy" clusters.

We have already remarked how fan-analysis is complementary to ACA, precisely because it evaluates each method separately for its value in aspect mining. The technique of unique methods [GK05] is similarly incremental (i.e. it also follows the method-by-method strategy to aspect mining). As such, it should be worthwhile to combine ACA with either of the incremental techniques. Additionally, the incremental techniques could easily be extended with our pre-filtering steps, to give these rather basic techniques more focus.

In Chapter 4, we briefly touched on fuzzy clustering, the scheme in which an object can be assigned to multiple clusters. In the light of fuzzy clustering, a simple idea arises: if the clusters are taken to be concerns, then the (source code) methods that belong to more than one cluster (concern) could indicate cases of tangling. Because tangling is perhaps the primary symptom of crosscutting, it seems somewhat limiting that our techniques only provide non-overlapping clusters.

We have implemented custom predicates for querying the code of the legacy system. As noted above, some of these predicates are approximation measures. It could be rewarding to move towards a more precise code analysis. Although, in carrying out our experiments, we encountered very few cases where data was wrongly formed, or classified, due to the rough edges around our own code analysis work.

Bibliography

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 49–60. ACM Press, 1999.

[AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In Haas and Tiwary [HT98], pages 94–105.

[And05] Periklis Andritsos. A survey of aspect mining tools and techniques. *Tech. Report CSRG-443, U. of Toronto, Dep. of Computer Science*, march 2005.

[Ber02] Pavel Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[BFVY96] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[Bil05] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[BK04] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 310–315, Washington, DC, USA, 2004. IEEE Computer Society.

[Bre04] Silvia Breu. Towards hybrid aspect mining: Static extensions to dynamic aspect mining. In Tom Tourwé, Magiel Bruntink, Marius Marin, and David Shepherd, editors, *Workshop on Aspect Reverse Engineering (WARE)*, November 2004.

[Bud01] A. Budanitsky. Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures, 2001.

[BYM$^+$98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[CMM$^+$05] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[CS95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern languages of program design.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[DAAC05] Simon Denier, Hervé Albin-Amiot, and Pierre Cointe. Expression and composition of design patterns with aspects. In Lionel Seinturier, editor, *2nd French Workshop on Aspect-Oriented Software Development (JFDLPA 2005)*, September 2005.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, March 2004.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[Fab] Johan Fabry. Irish homepage. http://prog.vub.ac.be/˜jfabry/irish/index.html.

[FF00] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[GHJ95] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex.

[GK05] Kris Gybels and Andy Kellens. Experiences with identifying aspects in Smalltalk using unique methods. In Tourwé et al. [TKCS05].

[GRS98] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. In Haas and Tiwary [HT98], pages 73–84.

[GRS99] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Rock: A robust clustering algorithm for categorical attributes. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 512–521. IEEE Computer Society, 1999.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, January 1997.

[GW97] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. Translator-C. Franzke.

[Har75] John A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, Inc., New York, NY, 1975.

[HK00] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[HT98] Laura M. Haas and Ashutosh Tiwary, editors. *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.* ACM Press, 1998.

[IV98] Nancy Ide and Jean Veronis. Word sense disambiguation: The state of the art. In *Computational Linguistics*, volume 24, 1998.

[JD03] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 178–187. ACM Press, March 2003.

[JHo] JHotDraw homepage. http://www.jhotdraw.org/.

[JP04] Neil C. Jones and Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms (Computational Molecular Biology).* The MIT Press, August 2004.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[KM05] Andy Kellens and Kim Mens. A survey of aspect mining tools and techniques. *INGI Technical Report 2005-07, UCL, Belgium.*, 2005.

[KN99] George Karypis, , and Vipin K. News. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[KR90] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley, 1990.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[Leo] W. Leong. The Aspect Browser web site. http://www.cs.ucsd.edu/users/wgg/Software/AB.

[LHH05] J. Zhang L. He, H. Bai and C. Hu. Amuca algorithm for aspect mining. In *Proceedings of SEKE 2005*, 2005.

[LW69] C. Lance and W. Williams. A general theory of classification sorting strategies: Ii. clustering systems. *Computer Journal*, 1969.

[Mit97] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[MMvD05] Marius Marin, Leon Moonen, and Arie van Deursen. A classification of crosscutting concerns. In *ICSM*, pages 673–676. IEEE Computer Society, 2005.

[MvDM04] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

[OL01] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192*, pages 73–80. Springer-Verlag, September 2001.

[RM02] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Int'l Conf. Software Engineering (ICSE)*, pages 406–416, 2002.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Görel Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2003.

[SGP04] David Shepherd, Emily Gibson, and Lori L. Pollock. Design and evaluation of an automated aspect mining tool. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 601–607. CSREA Press, 2004.

[SOU] SOUL homepage.
http://prog.vub.ac.be/research/DMP/soul/.

[SP05a] David Shepherd and Lori Pollock. Aspects, views, and interfaces. In *Workshop on Linking Aspect Technology and Evolution at the International Conference on Aspect Oriented Software Development*, March 2005.

[SP05b] David Shepherd and Lori Pollock. Interfaces, aspects, and views. In Tourwé et al. [TKCS05].

[SPT05] David Shepherd, Lori Pollock, and Tom Tourwé. Using language clues to discover crosscutting concerns. In Martin Robillard, editor, *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–6. ACM Press, May 2005.

[SZ97] Dennis Shasha and Kaizhong Zhang. Approximate tree pattern matching. In *Pattern Matching Algorithms*, pages 341–371. Oxford University Press, 1997.

[TC04] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society.

[TKCS05] Tom Tourwé, Andy Kellens, Mariano Ceccato, and David Shepherd, editors. *Linking Aspect Technology and Evolution*, March 2005.

[TM04] Tom Tourwe and Kim Mens. Mining aspectual views using formal concept analysis. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04)*, pages 97–106, Washington, DC, USA, 2004. IEEE Computer Society.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. 21st Int'l Conf. Software Engineering (ICSE'1999)*, pages 107 – 119. IEEE Computer Society Press, May 1999.

[TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[vdBCH06] Klaas van den Berg, Josè Marìa Conejero, and Juan Hernàndez. Identification of crosscutting in software design. In *8th International Workshop on Aspect-Oriented Modeling*, Bonn, Germany, 2006. March 21, 2006.

[vDMM05] Arie van Deursen, Marius Marin, and Leon Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to jhotdraw, 2005.

[vE05] Remco van Engelen. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10):804–818, 2005. Student Member-Magiel Bruntink and Member-Arie van Deursen and Member-Tom Tourwe.

[VSCF05] Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful aspects in jasco. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2005.

[VSV$^+$05] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in JAsCo. In Peri Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 75–86. ACM Press, March 2005.

[Wei79] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979.

[Wuy] R. Wuyts. The starbrowser homepage. http://homepages.ulb.ac.be/˜rowuyts/StarBrowser/.

[WYM97] Wei Wang, Jiong Yang, and Richard R. Muntz. Sting: A statistical information grid approach to spatial data mining. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97,* *Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 186–195. Morgan Kaufmann, 1997.

[ZK02] Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 515–524, New York, NY, USA, 2002. ACM Press.