Finding Refactorings via Change Metrics

Serge Demeyer
LORE - University of Antwerp
Universiteitsplein 1
B-2610 WILRIJK (Belgium)
sdemey@uia.ua.ac.be
http://win-www.uia.ac.be/u/sdemey/

Stéphane Ducasse
SCG - University of Berne
Neubrückstrasse 10
CH-3012 BERNE (Switzerland)
ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

Oscar Nierstrasz
SCG - University of Berne
Neubrückstrasse 10
CH-3012 BERNE (Switzerland)
oscar@iam.unibe.ch
http://www.iam.unibe.ch/~oscar/

ABSTRACT

Reverse engineering is the process of uncovering the design and the design rationale from a functioning software system. Reverse engineering is an integral part of any successful software system, because changing requirements lead to implementations that drift from their original design. In contrast to traditional reverse engineering techniques —which analyse a single snapshot of a system— we focus the reverse engineering effort by determining where the implementation has changed. Since changes of object-oriented software are often phrased in terms of refactorings, we propose a set of heuristics for detecting refactorings by applying lightweight, object-oriented metrics to successive versions of a software system. We validate our approach with three separate case studies of mature object-oriented software systems for which multiple versions are available. The case studies suggest that the heuristics support the reverse engineering process by focusing attention on the relevant parts of a software system.

Keywords

Reverse engineering, metrics, refactoring, software evolution, object-oriented frameworks.

1. INTRODUCTION

Reverse engineering is an ongoing process in the development of any successful software system as it evolves in response to changing requirements. Therefore, system developers may benefit a lot from reverse engineering techniques that keep track of the design as it drifts from that of earlier versions. Integrated development environments and so-called 'round-trip' engineering tools help to keep design views consistent with the actual software. However, such tools (i) are applicable to small software systems only; (ii) require an initial design model to be matched against the source code; (iii) present classes in isolation rather than interaction between classes. Consequently, these tools do not really contribute to the understanding of how and why a software system evolves.

©2000 ACM

ACCEPTED version 4.1 of Thursday, July 13, 2000

Proceedings OOPSLA'2000, ACM Press

Electronic version available at
http://win-www.uia.ac.be/u/sdemey/Pubs/Deme00a/

We claim that to really understand evolving software, the changes themselves are the critical factor. Certainly in object-oriented development processes with their emphasis on iterative development (see among others [Gold95a]) change is an essential ingredient of system design. Consequently, a reverse engineering technique should exploit whatever information is available concerning the changes of a piece of code. Such change based reverse engineering (i) would be more scaleable, because we focus reverse engineering efforts on changing pieces of code instead of the full code base; (ii) would not need an initial design model, because the changes dictate what appears in the model; (iii) would give insight into class collaborations, because redistribution of functionality reveals how classes interact with each other.

In object-oriented development, changes are accomplished by means of so-called refactorings [Opdy92b], [Fowl99a]. Refactoring operations —such as provided by the Refactoring Browser [Robe97a]— reorganise a class hierarchy by shifting responsibilities between classes, and redistributing instance variables and methods. Thus, to reverse engineer object-oriented software, we must identify which refactorings have been applied and this will help us understand how and —to a certain extent why— the system has evolved.

Clearly, discovering refactorings by textual comparisons of versions of source code would be imaginable but extremely tedious, since the nature of the refactorings would be revealed only by manual examination. Instead, we *measure* each of the available versions of a software system, and compare the results in order to identify the presence and the nature of the refactorings. That is, instead of an algorithmic approach for finding refactorings, we adopt a set of heuristics.

In this paper we examine four heuristics for identifying refactorings and investigate how this knowledge helps in program understanding. Each heuristic is defined as a combination of change metrics which reveals refactorings of a certain kind. One heuristic may occasionally miss refactorings, or misclassify them, but such mistakes are typically corrected by one of the other heuristics. We experimentally validate the applicability of these four heuristics by testing them on three case studies. Each case study is representative, in the sense that each is a successful software system which has undergone successive refactorings to address changing requirements. Our results indicate that our approach can be successfully applied to discover which parts of a system have changed, how they have changed, and, to some extent, why the designers chose to change them.

The paper is structured like an empirical study. It starts with formulating the research hypothesis to guide the selection of metrics and case studies (section 2), continues with a specification of the four heuristics (section 3) and proceeds with a discussion of the experimental results (section 4). The latter includes both an overview of the empirical data on the performance of the heuristics, as well as anecdotal evidence on how these heuristics support reverse engineering. After an overview of the future plans (section 0), the paper compares our work with related work on refactorings, metrics and other object-oriented reverse engineering techniques (section 6), to finally summarise the contribution (section 7).

2. EXPERIMENTAL SET-UP

This section introduces the experimental validation of the heuristics for identifying refactorings. It starts with describing the research assumptions that lead to the selection of the metrics (i.e., the GQM paradigm), follows with a brief description of the tools we used to collect the necessary measurements and ends with an overview of the three case studies.

2.1 The Goal-Question-Metric Paradigm

The decision which metrics to apply is made using the Goal-Question-Metric paradigm [Fent97a]. The goal and question underlying the experiment are defined in Table 1.

Table 1: The Goal and Question for the experiment.

Goal	Identify where the design of an object-oriented software system is evolving and deduce corresponding design drifts.
Question	When comparing one version of a software system with another, what characteristics can be used as symptoms of refactorings ?

We are particularly interested in finding symptoms for three categories of refactorings because they correspond —in our experience— with generic design evolutions that occur frequently in object-oriented software systems.

1. *Creation of template methods.* This category of refactorings split methods into smaller chunks to separate the common behaviour from the specialised parts so that subclasses can override. Design intentions for such refactorings may be the improvement of reusability and the removal of duplicated functionality.
2. *Incorporation of object composition relationships.* These kind of refactorings move functionality to (newly created) sibling classes. They are usually applied to reduce coupling and often mark migration towards black-box frameworks.
3. *Optimisation of class hierarchies.* A category of refactorings that insert or remove classes within a class hierarchy and redistribute the functionality accordingly. Such refactorings are applied to increase cohesion, simplify interfaces and remove duplicated functionality.

The question 'How to find symptoms of refactorings when comparing one version of a software system with a subsequent version' leads to the research assumptions listed in Table 2. Each research assumption corresponds with a characteristic that may serve as a symptom for a refactoring.

There are two criteria that must be taken into account when selecting the actual metrics for each of the research assumptions. First of all, the metrics should be derivable from object-oriented source-code, because the source code is the most reliable source of information when comparing different versions of the same software system. Second, the metrics should be cheap to collect, because with different versions of a software system there is quite a lot of data to analyse.

Table 2: Overview of the research assumptions.

Method Size	A decrease in method size is a symptom of a method split.
Class Size	A change in class size is a symptom for a shift of functionality to sibling classes (i.e., incorporate object composition). Also, it is part of a symptom for the redistribution of instance variables and methods within the hierarchy (i.e., optimisation of class hierarchy).
Inheritance	A change in the class inheritance is a symptom for the optimisation of a class hierarchy.

The actual metrics selected for our experiment are listed in Table 3 and are defined by Chidamber & Kemerer [Chid94a] and by Lorenz & Kidd [Lore94a]. The table includes an abbreviation, a reference to the definition and a short description of the metric.

2.2 Data Collection and Case Studies

To gather the necessary measurements on these case studies, we have developed our own metrics tool named Moose/Metrics. Moose is a re-engineering environment developed as part of the FAMOOS project[1]; a project whose goal is to produce a set of reengineering techniques and tools to support the development of object-oriented frameworks. Metrics are examined as an important part of framework reengineering, which explains the motivation behind this work. The Moose/metrics tool extracts classes, methods and attributes from VisualWorks/Smalltalk source code, computes the necessary measurements and generates a few tab separated ASCII files subsequently loaded in Microsoft/Access and Microsoft/Excel for post processing.

We validate the applicability of the heuristics via an experiment involving three case studies: the VisualWorks framework ([Haun95a], [Howa95a]), the HotDraw framework ([Beck94a], [John92a]) and finally the Refactoring Browser ([Robe97a]). These case studies have been selected because they satisfy the following criteria.

- *Accessible*. The source code for the different versions of these frameworks are publicly accessible, thus other researchers can reproduce or falsify our results.
- *Representative*. Each of the three case studies is a successful software system which has undergone successive refactorings to address changing requirements.
- *Independent*. All frameworks were developed independently of our work, which implies that our experiment has not influenced the development process.
- *Documented*. The features that changed between the different versions are documented, making it possible to validate some experimental findings.

All of these case studies are implemented in Smalltalk. Nevertheless, each of these case studies represents a different kind of software system.

- *VisualWorks*. Is the only industrial system in the experiment. VisualWorks is a framework for visual composition of user-interfaces independent of the 'Look and feel' of the deployment platform. It is a typical black-box framework, in

[1] `http://www.iam.unibe.ch/~famoos/`

Table 3: Overview of the metrics selected.

Abbreviation	**Reference**	**Description**
Method Size (computed for each method)		
Mthd-MSG	[Lore94a]	Number of message sends in method body. In [Lore94a] this metric was abbreviated NOM, but we renamed it MSG to avoid name collision with the NOM metric (number of methods).
Mthd-NOS	[Lore94a]	Number of statements in method body.
Mthd-LOC	[Lore94a]	Lines of code in method body.
Class Size (computed for each class)		
NOM	[Chid94a]	Number of methods in class. This is the Weighted Method Count (WMC) metric where the weight of each method is 1.
NIV, NCV	[Lore94a]	Number of instance variables, number of class variables defined by class.
Inheritance (computed for each class)		
HNL (DIT)	[Lore94a], [Chid94a]	Hierarchy nesting level [Lore94a] or the depth of inheritance tree [Chid94a]. Number of classes in superclass chain of class; in case of multiple inheritance, number of classes in longest chain.
NOC	[Chid94a]	Number of immediate children of class.
NMI	[Lore94a]	Number of inherited methods, i.e. defined in a superclass and inherited unmodified.
NMO	[Lore94a]	Number of overridden methods, i.e. defined in a superclass but redefined in subclass. We do not distinguish between plain overriding, overriding of abstract methods and extensions of super methods.

the sense that programmers using the framework are supposed to customise its behaviour by (visually) configuring objects. The framework has quite a large customer base, so the framework designers must consider backward compatibility issues when releasing new versions.
The versions we measured during the experiment are the versions that have been released during the time span of 1992-1996.

- *HotDraw*. Is a framework for building 2-dimensional graphical editors. HotDraw is one of the better known framework experiments, among others because of its pattern style documentation. It's a typical white-box framework, as users of the framework are supposed to subclass framework classes in order to reuse the HotDraw design.
The HotDraw framework itself was implemented in VisualWorks/Smalltalk. Therefore, we numbered the versions according to their corresponding VisualWorks release.
- *Refactoring Browser*. This case study is the only software system which is not a framework. Nevertheless, it is interesting because it is a good example of an iterative development process.
The refactoring Browser is released about once each month with only small increments. Rather than measuring every single increment, we compared a first stable version with the latest version that was available at the time of our experiment. Our first version corresponds to the 2.0 release (April '97) while the second version is the 2.1 release (March '98).

Table 4 provides an overview of the changes for each of the three

Table 4: The transitions between versions the case studies.
(Ini = # initial version; Rmv = # removed; Add = # added; Ret = #retained (same name); Fnl = # final version)

	# classes					**# methods**				
	Ini	**Rmv**	**Add**	**Ret**	**Fnl**	**Ini**	**Rmv**	**Add**	**Ret**	**Fnl**
Visualworks										
1.0 -> 2.0	593	71	212	522	734	5282	1067	3569	4215	7784
2.0 -> 2.5	734	38	63	696	759	7784	337	858	7447	8305
2.5 -> 3.0	759	8	214	751	965	8305	207	1287	8098	9386
HotDraw										
1.0 -> 2.5	197	2	4	195	199	1425	153	72	1272	1344
2.5 -> 3.0	199	151	67	48	115	1344	1221	707	123	830
Refactoring Browser										
2.0 -> 2.1	277	2	52	275	327	2121	273	786	1848	2634

case studies involved in the experiment. The leftmost column lists the different transitions between the versions at our disposal, while the other columns give an impression of the size and impact of the changes. We see that VisualWorks is the largest software system (starting with 593 classes and 5282 methods in version 1.0 and ending with 965 classes and 9386 methods in version 3.0). We also see that the transition from VisualWorks 1.0 to 2.0 entailed most of the changes (71 classes and 1067 methods were removed; 212 classes and 3569 methods were added), with the one from VisualWorks 2.5 to 3.0 and from HotDraw 2.5 to 3.0 following as second and third. Here it is important to note that we compare versions via names, thus that we cannot detect renaming and instead count a 'remove' and an 'add'. Due to the large number of removals and additions, we assumed that the large number of changes between HotDraw 2.5 and 3.0 actually corresponds with renaming of classes and by checking the source code we have confirmed this assumption. Finally, note the small decrease in number of methods between HotDraw 1.0 and 2.5 and the large decrease in number of classes and methods between HotDraw 2.5 and 3.0. This confirms with what is reported in the documentation, because HotDraw is a system where the design has been expanded a little while moving from 1.0 to 2.5, while it has been reworked drastically during the transition from 2.5 to 3.0.

3. HEURISTICS FOR FINDING REFACTORINGS

Given the research assumptions in Table 2 and the list of metrics Table 3, we are now in the position to specify which heuristics we use to find refactorings. In the following subsections we define a heuristic as a combination of change metrics that identify a category of refactorings and name the heuristic according to its category. With each heuristic we specify a recipe that explains how to assess whether the result does correspond with a refactoring and a description of the false negatives and false positives. A *false positive* for a heuristic is a case where the heuristic answered 'yes' while it should have been 'no'; thus where it erroneously reported the presence of a refactoring. Conversely, a *false negative* is a case where the heuristic answered 'no' while it should have been 'yes', thus where it failed to identify the refactoring.

Note that with the expression 'compute the change in metric-X on component-Y' we mean 'subtract the metric-X measurement of component-Y in one version from metric-X measurement of component-Y in the subsequent version'. Since it occurs so often, we denote it with a shorthand 'δ X'. As an example, 'compute the change in HNL on class View' corresponds to the following function:

$$\delta\, HNL(View) = HNL(View, version\ n + 1) - HNL(View, version\ n)$$

3.1 Split into Superclass / Merge with Superclass

This heuristic searches for refactorings that optimise the class hierarchy by splitting functionality from a class into a newly created superclass, or that merge a superclass into one or more of its subclasses. That is, we look for the creation or removal of a superclass, together with a number of pull-ups or push-downs of methods and attributes.

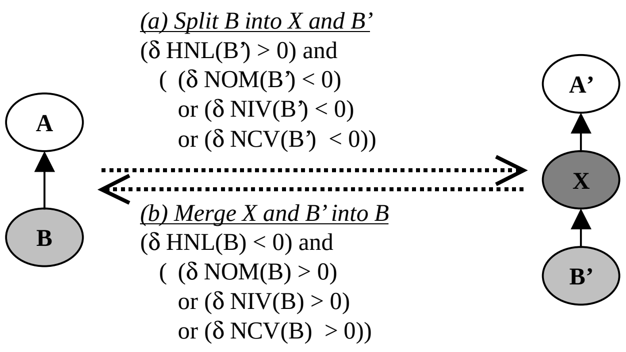

Figure 1: Heuristic for 'Split into Superclass / Merge with Superclass'.

The following recipe is summarised in Figure 1[2].

1. By inspecting changes in the inheritance hierarchy (via a change in HNL), we identify those classes where a superclass has been added or removed.
2. Next, we combine this information with changes in number of methods (NOM) and number of instance- and class variables (NIV, NCV) to see whether functionality has been pulled up or pushed down.
3. We also include the number of inherited and overridden methods (NMI and NMO) in this analysis to see the effect of the added or removed class on the overall protocol of the subclass.
4. Finally, we browse the source code of the corresponding classes to check whether the new class hierarchy corresponds with a split/merge and to see which pull-ups and push-downs actually occurred.

- *False negatives.* The heuristic may fail to detect a split or a merge, when the change in HNL is countered by an inverse change higher up in the hierarchy, or when the pull-up/push-down is countered by an equal addition or removal of new functionality. The former of these cases will show up as false positives of 'Move to Other Class' (section 3.3).
- *False positives.* The heuristic may discover situations that do not correspond to a split or a merge, if part of the class functionality has been rearranged, superimposed by an unrelated addition or removal of a superclass. It is worthwhile to further analyse such situation, as the rearrangement of class functionality may denote a false negative of 'Move to Other Class' (section 3.3). Moreover, the change in the superclass may represent an interesting optimisation of the class hierarchy anyway. Therefore, we include the number of inherited and overidden methods (NMI, NMO) in the heuristic, because they let us assess whether the operation optimises the class hierarchy.

[2] In all figures we use the following conventions. The same components in different versions are distinguished via a quote character (') which does not necessarily imply any change. Components that change in the refactoring are coloured in light-grey, while new or removed components are coloured in dark-grey.

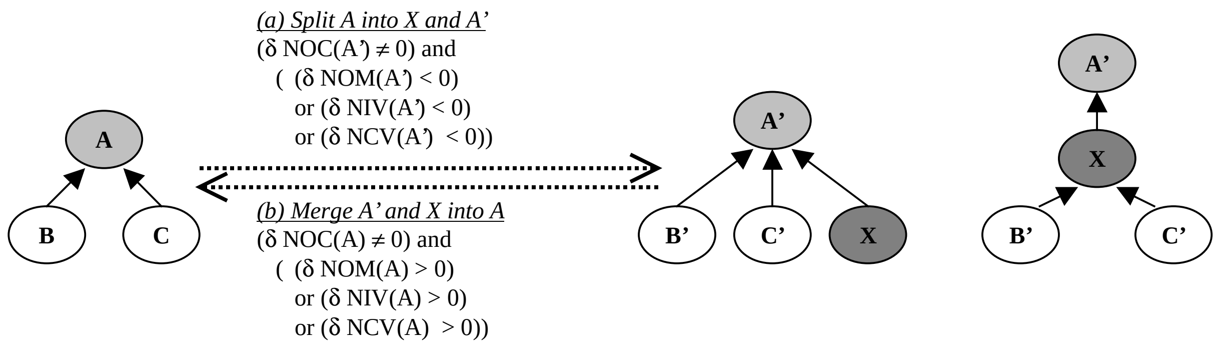

Figure 2: Heuristic for ‘Split into Subclass / Merge with Subclass’.

3.2 Split into Subclass / Merge with Subclass

This heuristic —like the one in 3.1— searches for refactorings that optimise the class hierarchy. However, it takes the viewpoint of the superclass (using changes in the number of children as the main symptom), while the other provides the perspective of the subclass (triggering on changes in the length of the inheritance chain). Thus the refactorings we are looking for here split functionality from a class into a newly created subclass, or merge a subclass with one or more of its subclasses. That is, we look for the creation or removal of a subclass, together with a number of pull-ups or push-downs of methods and attributes.

The following recipe is summarised in Figure 2.

1. By inspecting changes in the inheritance hierarchy (via a change in NOC), we identify those classes where a subclass has been added or removed.
2. Next, we combine this information with changes in number of methods (NOM) and number of instance and class variables (NIV, NCV) to see whether functionality has been pulled up or pushed down.
3. Finally, we browse the source code of the corresponding classes to check whether the new class hierarchy corresponds with a split/merge and to see which pull-ups and push-downs actually occurred.

- *False negatives.* The heuristic may fail to detect a split or a merge, when the refactoring did not involve a change in NOC, or when the pull-up/push-down is countered by an equal addition or removal of new functionality. Sometimes, these cases will show up as false positives of ‘Move to Other Class’ (section 3.3).
- *False positives.* The heuristic may discover classes that are not split or merged. Most often this is because some class functionality has been added, moved or removed together with an unrelated addition or removal of subclasses. Thus, sometimes the false positive does correspond with a false negative of ‘Move to Other Class’ (section 3.3). Also, the false positive may denote some other rearrangement of the class hierarchy which is worthwhile to analyse further as it represents another class hierarchy refactoring.

3.3 Move to Other Class (Superclass, Subclass or Sibling Class)

This heuristic searches for refactorings that move functionality from one class to another. This other class may either be a subclass, a superclass, or a sibling class (i.e., a class which does not participate in an inheritance relationship with the target class, although it usually has a common ancestor). That is, we look for removal of methods, instance variables or class variables and use browsing to identify where this functionality is moved to. Note that ‘Split into superclass’ (section 3.1) and ‘Split into subclass’ (section 3.2) are special cases of the more general move of functionality.

The following recipe is summarised in Figure 3.

1. By inspecting decreases in the number of instance variables (NIV), class variables (NCV) or methods of a class (NOM), we identify those classes where functionality has been removed.
2. We combine this with HNL and NOC measurements, to rule out those cases that correspond with splitting of classes (see sections 3.1-3.2).
3. Next, we browse the source and check for each method removed those methods that used to invoke it to see where the functionality has been moved to, or to assess whether the functionality has been removed.

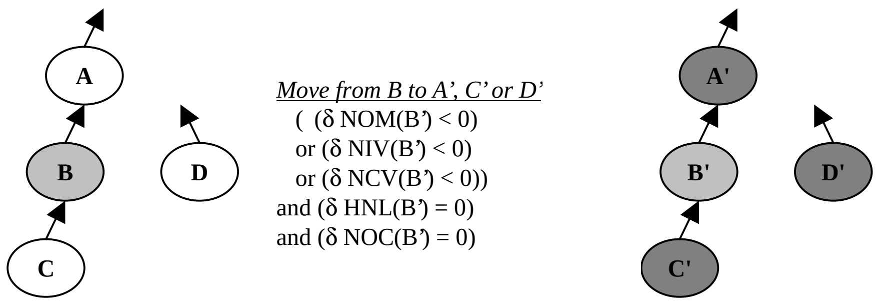

Figure 3: Heuristic for ‘Move to Superclass, Subclass or Sibling Class’.

4. Similarly, for each attribute removed, we check the methods that used to access it to see where the attribute has been moved to, or to assess whether it has been removed.

- *False negatives.* The heuristic may fail to detect a move, when it was countered by an equal addition or removal of functionality. The case when the move is accompanied by an addition or removal of a superclass or an immediate subclass, is detected anyway, as it is a false positive of either ‘Split into Superclass / Merge with Superclass’ (section 3.1) or ‘Split into Subclass / Merge with Subclass’ (section 3.2).
- *False positives.* The heuristic may discover cases that do not correspond to a move of functionality but then it detects symptoms of other refactorings. The functionality may be removed instead of moved, or the functionality may be replaced by code that requires less methods or attributes, thus a refactoring in either case. Especially in the case where attributes are factored into sibling classes it is rarely an ordinary move, but almost always partly a movement and partly a replacement of functionality.

3.4 Split Method / Factor Out Common Functionality

This heuristic searches for refactorings that split methods into one or more methods defined on the same class. That is, we look for decreases in method size and try to identify where that code has been moved to.

The following recipe is summarised in Figure 4.

1. By looking for decreases in the number of message sends (Mthd_MSG) within a method, we identify those methods where functionality has been removed. We include a threshold in the heuristic, because the large amounts of methods returned by the heuristic required to fine-tune the amount of functionality decrease we are interested in. In our experiment, we applied a threshold of 2, but reverse engineers are free to adapt this number according to their purpose.
2. A combination with decreases in number of statements (Mthd_NOS) and lines of code (Mthd_LOC) provides insight into the kind of removals that actually took place.
3. Sort the resulting list according to class name to identify other methods defined on the same class and presenting a similar decrease, because that’s a symptom of common functionality that has been factored out.
4. Finally, browsing the source code is necessary to see where the functionality has been moved to. That is, each removed message-send must be checked to see whether (new) methods defined on the same class perform that same message-send.

- *False negatives.* The heuristic may fail to detect a split, when it was countered by an equal addition of new functionality, or when the decrease in functionality was smaller than the threshold value.
- *False positives.* The heuristic may discover methods where functionality simply has been removed instead of split into another method. Also, sometimes the functionality that has been split off may be moved to another class which is difficult to recover by mere browsing. In either case it is worthwhile to further analyse the situation as both cases correspond to other interesting refactorings.

4. RESULTS

This section reports on the results obtained from applying the four heuristics defined in section 3 to the three case studies specified in section 2. We start with a summary of the empirical data collected from the experiment and an analysis of the advantages and potential drawbacks of the heuristics, and end with some anecdotal evidence of interesting design shifts we have identified using the heuristics.

4.1 Advantages and Potential Drawbacks

Table 5 provides a summary of the empirical data we gathered from the case studies. For each of the different transitions (leftmost column) and for each of the heuristics we show:

- *Focus*. The number of classes/methods returned by the heuristic. Since the heuristics are supposed to be complemented by manual browsing, this number should be small.
- *Focus %.* The focus in percentage. Is equal to the focus divided by the total number of classes/methods retained (columns ‘Ret’ in Table 4). This number allows us to compare the focus of the heuristic independent of the case study.

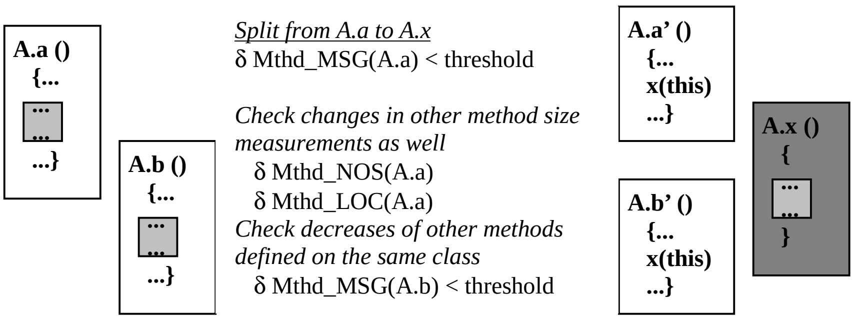
A.a () {... ... ...}
A.b () {... ... ...}
Split from A.a to A.x
δ Mthd_MSG(A.a) < threshold
Check changes in other method size measurements as well
δ Mthd_NOS(A.a)
δ Mthd_LOC(A.a)
Check decreases of other methods defined on the same class
δ Mthd_MSG(A.b) < threshold
A.a’ () {... x(this) ...}
A.b’ () {... x(this) ...}
A.x () { ... ... }
Figure 4: Heuristic for ‘Split Method / Factor Out Common Functionality’.

Table 5: Overview of the results
(The maximum Focus and Focus % values are marked with double underlining)

	Split or merge / superclass				**Split or merge / subclass**			
	Focus %	**Focus**	**#FP**	**#IFP**	**Focus %**	**Focus**	**#FP**	**#IFP**
Visualworks								
1.0 -> 2.0	3.64%	19	3	1	8.24%	43	41	3
2.0 -> 2.5	0.00%	0			1.58%	11	11	0
2.5 -> 3.0	0.15%	1	1	0	3.65%	24	24	2
HotDraw								
1.0 -> 2.5	0.00%	0			0.00%	0		
2.5 -> 3.0	10.42%	5	3	3	35.42%	17	16	10
Ref. Brwsr.								
1.0->2.0	1.82%	5	0	0	2.18%	6	4	4
	Move to other class				**Split method**			
	Focus %	**Focus**	**#FP**	**#IFP**	**Focus %**	**Focus**	**#FP**	**#IFP**
Visualworks								
1.0 -> 2.0	6.32%	33	19	6	2.49%	105	70	13
2.0 -> 2.5	1.72%	12	9	2	1.03%	77	63	9
2.5 -> 3.0	4.57%	30	26	3	0.44%	36	12	
HotDraw								
1.0 -> 2.5	14.36%	28	27	3	1.26%	16	9	4
2.5 -> 3.0	8.33%	4	2	0	14.63%	18	11	3
Ref. Brwsr.								
2.0 -> 2.1	4.00%	11	8	1	1.30%	24	18	6

- *# FP (false positives).* The number of false positives (refactorings that are misclassified) as assessed by manual browsing of the source code.[3] The lower this number, the less noise contained in the results and the more reliable the heuristic is.
- # IFP (interesting false positive). The number of misclassified refactorings that corresponded to another category of refactorings. This number calibrates the number of false positives, in the sense that reverse engineering can afford misclassifications but should not miss too many refactorings. Ideally, this number should be equal to the number of false positives.

The empirical data and our experience with the case studies gave us some insight into the advantages and potential disadvantages of using the heuristics for identifying design shifts.

[3] Note that we could not count the number of false negatives, as we did not have any list of refactorings as they have actually been applied. However, we used the documentation for each of the systems to check whether we covered all the documented design changes.

Advantages

- *Good Focus.* The heuristics focus on a small part of the software system. That is, the amount of classes and methods returned is always small enough to allow for further manual examination. For instance, the split method heuristic selects between 5 and 105 of the methods (threshold = -2). The number of classes selected lies between 5 and 43.
- *Reliable.* The heuristics are quite good at identifying which refactorings occurred during the process. That is, the number of false positives is low and a considerable amount of false positives is interesting anyway. This is largely due to the partial overlap between the heuristics: some of the false negatives of one heuristic are covered by false positives of another one.
  Due to the lack of empirical data on false negatives, it is too early to make any definite statements on the reliability. However, our experience with the case studies showed that we identified symptoms for all of the documented design shifts and even some undocumented ones. Therefore, we can safely assert that the heuristics may have missed some of the

refactorings, but they usually have identified the important design changes.

- *Road map*. The focus (both in absolute number as well as percentage) varies considerably between the heuristics. This may seem a negative effect, but we have experienced that it actually helps the reverse engineering process because it is a guidance for determining which places to browse first. That is, by first applying all the heuristics, and then manually browsing the results in 'best focus first' order, we could rely on the better focus of the prior heuristic to better assess items in the larger lists.
- *Class Interaction*. Since the heuristics select those places where the system design is changing, there is a good chance to focus on those parts of the design that are expanding or consolidating, which is a key factor in understanding why a design is adapted. Moreover, the heuristics filter out clusters of classes and methods, highlighting the way they interact with each other. Again, this helps to understand a design, because it is precisely the interaction between the various methods in different classes that determines how to reuse an object-oriented design.
- *Unbiased*. The heuristics do not require human knowledge to select the interesting pieces of code. Rather, it is the system itself that dictates what is worthwhile to investigate. Consequently, our heuristics may be used prior to reverse engineering techniques that require a reverse engineer to formulate a model of what is expected, like with reflexion models [Murp97a], concept assignment [Bigg94a], or round-trip engineering tools.

Potential Drawbacks

- *Vulnerable to renaming*. Measuring changes on a piece of code requires that one is capable of identifying the same piece of code in a different version. The most obvious approach —the one also used in our experiment— is using names to anchor pieces of code. However, with name anchors, rename operations dissolve to reappear as removals and additions. This phenomenon partly explains the large focus and large number of false positives in the transition from VisualWorks 2.5 to 3.0.
  Nevertheless, any reverse engineering technique that relies on source code must be conscious about names as they usually bear a lot of the domain semantics. Our heuristics are not an exception to that rule.
- *Imprecise for many changes*. Both focus and reliability depend a lot on the amount of changes. When too many changes have been applied on the same piece of code, the picture gets blurred and the technique becomes imprecise.
  However, this imprecision is only with respect to the identification of the refactorings, because the heuristics still identify important design changes. It only becomes more difficult to deduce the intentions behind these changes.
- *Requires human experience.* Once the interesting pieces of code have been identified, a reverse engineer must check the two versions of the source code to deduce which refactorings actually have been applied. This requires experience with refactorings and with the implementation language, in the sense that the reverse engineer must be well aware of how the refactorings interact with the coding idioms in the particular implementation language.
  Consequently, the heuristics should be applied in a semi-automatic setting, where a tool signals potential refactorings but where a human expert takes the actual decision. Semi-automatic tools represent a reasonable trade-off, as reverse engineering is typically a human centric activity [Bigg94a].
- *Considerable resources.* The metrics underlying our heuristics are selected to require 'cheap' parsing technology. For instance, we have been able to collect the measurements from the Smalltalk system via a straightforward use of the meta-object protocol and some rudimentary text processing to count the number of lines. Similar technology could be used to collect the measurements for Java systems, while for C++ we can do with incomplete parsers that only collect the class definitions and method signatures.
  Nevertheless, reverse engineering based on our heuristics requires quite a lot of resources: full access to different versions of the source code, plus a metrics tool that is able to measure changes (or at least feed measurements into another tool performing additional computations) plus a browsing tool that is able to navigate from a method invocation to the corresponding method definition. Yet, the amount of resources is comparable to other reverse engineering techniques. For instance, round-trip engineering tools require full parsing, program visualisation tools require complex layout algorithms, query languages require databases.

4.2 From Refactorings to Understanding Design Drift

While the empirical data in section 4.1 shows that the heuristics are effective in identifying which refactorings have been applied when going from one version to another, this by itself does not imply that we can actually deduce how and why the implementation has drifted from its original design. It is beyond the scope of this paper to analyse the intentions for the hundreds of refactorings we have identified in the experiment. Therefore, this section makes a selection by describing some of the more interesting refactorings including our interpretation of its design intention.

- *Reduce Code Duplication (Split into Superclass).* Some of the new operations added to the Refactoring Browser were refactorings to add or remove a parameter on a method. Looking at the implementation, the 2.0 release included a very flat hierarchy with an abstract root class 'Refactoring' and below a list of subclasses for each refactoring operation. When adding the subclasses for manipulating parameters, the developers noticed that this introduced duplicated behaviour with the former 'RenameMethod' refactoring and consequently they inserted a new abstract superclass 'MethodRefactoring' with subclasses for all refactorings concerning methods. Inspecting the methods that have been redistributed between the former 'RenameMethod' class and the new 'MethodRefactoring' class we could easily infer the obligations for new subclasses of 'MethodRefactoring'.
- *Optimize for platform independence (Merge With Superclass).* The VisualWorks framework provides a platform independent 'look and feel'. Therefore, the frameworks has abstract superclasses for each Widget with concrete subclasses for the different UI platforms supported by the framework (Motif, Macintosh, Windows, ...). In the transition from 1.0 to 2.0 they have reorganised this widget hierarchy, as is depicted in Figure 5. As such, they diminish

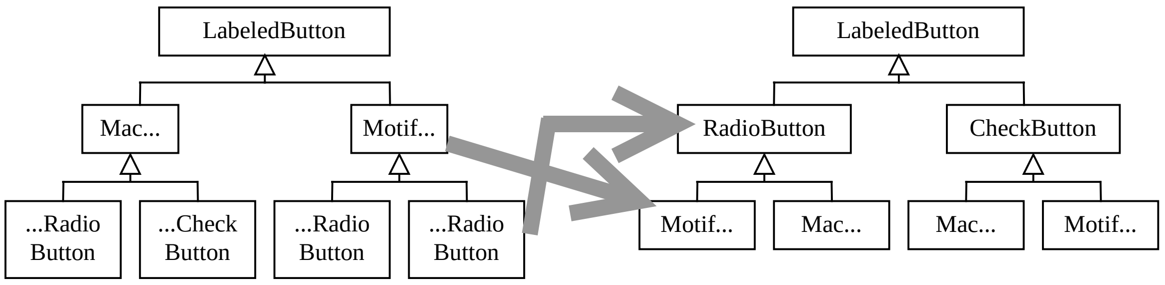

Figure 5: Optimize for platform independence.

the work when a new look-and feel is included because less concrete subclasses must be implemented. Moreover, the way the existing methods where redistributed over the class hierarchy, gave us insight into the protocol between the abstract and the concrete classes.

- *Introducing Layers (Move to Sibling)*. The Refactoring Browser provides multiple windows depending on the selected view on the software system, each window consisting of various panes. It is the 'RefactoringBrowser' class that implements the aggregation of panes, one of these panes being the 'BrowserNavigator'. In the 2.0 release, there was a complicated interaction between these two classes depending on various state changes. With the 2.1 release, this has been simplified, in such a way that the BrowserNavigator (= the component) ceases to invoke methods directly on the RefactoringBrowser (= the aggregation). This way, the designers have been able to introduce a layer into their system design (see Figure 6).

To summarise, we conclude that these four heuristics reliably and cost effectively identify which refactorings have been applied in going from one version to another. From there it is feasible to deduce how and why an implementation drifts from its original design. Although the heuristics have some disadvantages, most of them are inherent to reverse engineering techniques. Consequently, we claim that the proposed heuristics form the basis for a reliable reverse engineering technique, given that it is used by experts, on iterative projects with small increments.

5. OPEN QUESTIONS AND FUTURE WORK

Given the empirical results in section 4, the obvious question is whether they generalise. Especially because all case studies are successful Smalltalk systems it is interesting to ask (i) how these heuristics will behave on other implementation languages and (ii) how they will behave on systems that are 'in trouble'. Also, since there are other metrics that could have been used to measure changes between different versions than the ones listed in Table 3, another relevant question is (iii) how much these results depend on the particular metrics applied in the experiment. Finally, given the fact that the number of documented refactorings is large and keeps growing, another important question is (iv) whether the same technique can be applied to identify other refactorings.

As far as other implementation languages are concerned, we are quite confident that the heuristics will prove valuable. To confirm this feeling, we plan further experiments with C++ and Java systems obtainable from publicly available code and from the industrial partners in the FAMOOS project.

As far as troublesome systems concerns, the expected behaviour depends largely on what is implied by 'in trouble'. One thing we suspect, is that change metrics may be used to identify places where the design keeps on changing without actually improving. Here as well we plan more experiments to verify this assumption.

As far as other metrics concerns, the answer depends on the kind of metrics. For the method size and class size metrics we think that it is best to stick to these simple metrics, as it is often reported that code size metrics correlate with each other [Fent97a]. For the inheritance metrics we are less certain: the simple metrics were sufficient for our purpose, but it is possible that other metrics are better discriminators for distinguishing interesting refactorings. Finally, there is a category of metrics that we deliberately did not include in our experiment. Coupling/cohesion metrics are typically more expensive to collect and on top of that there is a strong disagreement in the literature about what constitutes good object-oriented coupling/cohesion metrics. Nevertheless, we assume that they may serve as good indicators for refactorings that introduce object composition and consequently might be used to mark transition from white-box to black-box reuse. In the future, we plan to define our own set of cheap coupling/cohesion metrics to see whether they offer any help.

As far as other refactorings concerns, the answer depends on the granularity of what you want to reverse engineer and the purpose why you are doing it. The refactorings selected in our experiment are coarse grained compared to the elementary refactorings provided by the refactoring browser, because we

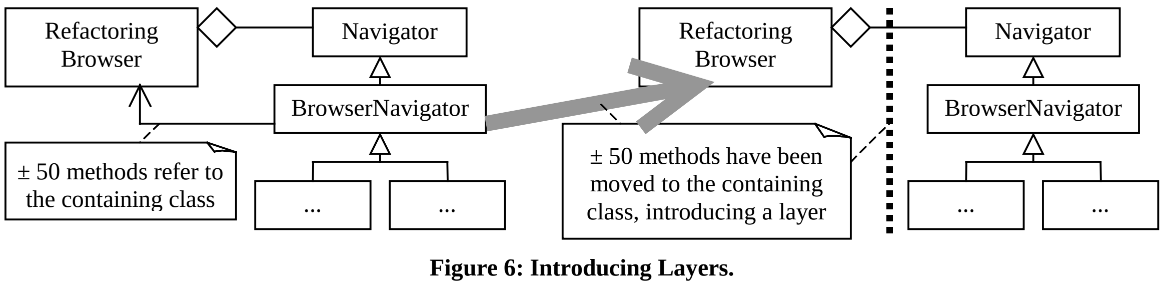
Refactoring Browser
Navigator
BrowserNavigator
± 50 methods refer to the containing class
...
...
Refactoring Browser
Navigator
BrowserNavigator
± 50 methods have been moved to the containing class, introducing a layer
...
...

Figure 6: Introducing Layers.

needed to deal with the issue of scale — recovering every single change would have provided too much noise to suit our reverse engineering purposes. On the other hand we do not cover the composite refactorings listed in Fowler's catalogue [Fowl99a], mainly because these also include the motivation for applying a refactoring — the motivation part of such refactorings must be recovered by humans. Based on our experience, we suppose that it is feasible to define heuristics for all possible refactorings (even if it entails defining special purpose metrics), although this is probably not worth the effort. Hence, to devise a catalogue of heuristics for identifying the most important refactorings, empirical studies are needed to investigate which refactorings are commonly used during reengineering.

Future Work

Besides continuing the validation of the heuristics against other software systems and experimenting with other metrics (in particular coupling and cohesion) we want to integrate these heuristics with other reverse engineering techniques. This is necessary because we agree with the viewpoint that a reverse engineer is much like a detective that solves a mystery from the scarce clues that are available [Will96b]. The heuristics proposed here are not an exception to that rule so should not be used in isolation but rather complement other reverse engineering techniques.

Consequently, we will work on better integration with other tools in the Smalltalk programming environment. A first path to explore is better integration with the Refactoring Browser —since this tool maintains a log file of the refactorings that have been applied it is one way of verifying the number of false negatives. Second on our list is integration with a Smalltalk configuration management tool (such as Envy) to exploit the version management features provided there. We also plan integration with Duploc and CodeCrawler, which are two visualisation tools under development in our lab. Duploc provides a visual representation of duplicated source code which is also well suited for detecting portions of source code that have changed [Duca99b]. CodeCrawler visualises predefined metrics combinations with simple graph layouts [Deme99c]. A promising research direction might be to include run-time information to analyse the effect of refactorings at the object-level [Rich99a]. In the longer run, we want to investigate tool integration for object-oriented languages besides Smalltalk; our work on an information exchange model for reengineering tools is a first step in that direction [Deme99d].

6. RELATED WORK

Refactoring as a technique to reorganise object-oriented constructs has been investigated for quite some years now. The Ph.D. work of Opdyke resulted in a number of papers describing incremental redesign performed by humans supported by refactoring tools [Opdy92b], [Opdy93a], [John93b]. This line of work resulted in the Refactoring Browser, a tool that represents the state of the art in the field [Robe97a]. In contrast, both Casais and Moore report on tools that optimise class hierarchies without human intervention [Casa92a], [Moor96a]. More recently, Schulz et al. report on refactoring to introduce design patterns in C++ systems [Schu98a], while [Toku99a] show that a typical system evolution can be reproduced significantly faster and cheaper via refactoring. Finally, [Fowl99a] summarises his experience with refactorings in industrial projects. In the refactoring literature we did not find any reference to techniques for reconstructing the refactorings that have been applied, although [Toku99a] shows that a manual approach is feasible on a small scale.

Quality metrics are often used as a problem detection technique and in that sense closely related to our work. An overview of the work in Object-Oriented Metrics —including references to quality metrics— is presented in [Hend96a]. To detect problems with quality metrics, one combines the metrics with thresholds and whenever a particular measurement exceeds a certain threshold value, the corresponding item needs further examination. Both [Chid94a] and [Lore94a] mention the use of thresholds to identify design anomalies, while [Mari98a] and [Kohl98a] report on the practical application thereof. Other uses of metrics as problem detection tools exist as well: for instance, both [Kont97a] and [Lagu97a] use metrics for detecting patterns of duplicated code. With these problem detection tools, we share the idea of using metrics to focus in large amounts of data, but differ in the elements we are interested in (design anomalies vs. refactorings).

The reverse and reengineering of object-oriented systems has gained considerable interest now that the object-oriented paradigm has reached industrial maturity [Casa97a]. Especially with the advent of UML, there are lots of CASE tool vendors working on so-called round-trip engineering as a way to iterate between modelling, generating code, changing that code and mapping this code back to the original model. Note however that reverse and reengineering is studied intensively in other branches of the software engineering community [Arno92a], [Wate94a], [Will96a]. Note as well that other researchers already exploited program changes in the context of reverse engineering: for instance, Thomas Ball et. al. annotate code views with colours showing code age [Ball96a], while Mehdi Jazayeri et. al use a three-dimensional visual representation for examining a system’s software release history [Jaza99a]. Restricting ourselves to the object-oriented literature, we see that reverse engineering research concentrates on combinations of program visualisation, run-time analysis, (logic) query languages and design (pattern) extraction. For instance, De Pauw et al. report on design extraction based on visual representations of run-time information [Pauw93a]. Lange and Nakamura describe how a Prolog fact base (containing static and dynamic information) linked to visual representations can help to capture the design of a framework [Lang95a]. Brown shows that it is possible to detect some structural patterns in Smalltalk source code [Brow96c], while Keller et. al. did larger scale experiments for C++ [Kell99a]. Florijn reports on tool support for patterns in reverse and forward engineering [Flor97a]. Wuyts advocates the use of a logic meta-language to express and extract structural relationships in object-oriented systems [Wuyt98a]. De Hondt reports on advanced browsing technology as a way to incrementally capture framework design in the form of reuse contracts [Hond98a]. Except for [Hond98a], we did not find any references to the exploitation of the iterative nature of object-oriented development for reverse engineering purposes.

7. CONCLUSIONS

We have presented an approach to understand how object-oriented systems have evolved by discovering which refactoring operations have been applied from one version of the software to the next. The main features of our approach are:

- It concentrates on the *relevant parts*, because the refactorings point us to those places where the design is expanding or consolidating.

- It provides an *unbiased view* of the system, as the reverse engineer does not have to formulate models of what is expected in the software.
- It gives an insight in the way *classes interact*, because the refactorings reveal how functionality is redistributed among classes.

These features are extremely useful in a reverse engineering process because it reveals where, how —and sometimes even why— an implementation has drifted from its original design.

We have demonstrated through three case studies that the technique is applicable in practice, and can effectively focus attention on interesting aspects of a software system. Our results are preliminary, in the sense that three case studies are by no means statistically significant. Nevertheless, it is clear that the approach warrants further study and experimentation.

8. ACKNOWLEDGEMENTS

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975.

We want to thank people who have reviewed earlier drafts of this paper (anonymous OOPSLA'98, ECOOP'99 and OOPSLA'2000 reviewers, Joachim Weisbrod, Radu Marinescu, Simon Moser, Robb Nebbe, Franz Acherman, Tamar Richner, Matthias Rieger, Michele Lanza and Sander Tichelaar), who have encouraged our work (Kent Beck, Don Roberts, John Brant, Wilfried Verachtert and Patrick Steyaert) or who have provided fruitful partnership (the Software Composition Group, the participants in the FAMOOS project and the Programming Technology Lab in Brussels).

9. REFERENCES

[Arno92a] Robert S. Arnold, Software Reengineering, IEEE Computer Society Press, 1992.

[Ball96a] Thomas Ball and Stephen G. Eick, "Software Visualization in the Large, " IEEE Computer, Vol. 29(4), April 1996.

[Beck94a] Kent Beck and Ralph Johnson, "Patterns Generate Architectures," Proceedings ECOOP'94, M. Tokoro, R. Pareschi (Ed.), LNCS 821, Springer-Verlag, 1994.

[Bigg94a] Ted Biggerstaff, Bharat Mitbander and Dallas Webster, "Program Understanding and the Concept Assignment Problem," Communications of the ACM, Vol. 37(5), May 1994.

[Brow96c] Kyle Brown, "Design Reverse-engineering and Automated Design Pattern Detection in Smalltalk," Ph.D. dissertation. North Carolina State University, 1996. [http://www.ksccary.com/kbrown.htm, http://hillside.net/patterns/papers/].

[Casa92a] Eduardo Casais, "An Incremental Class Reorganization Approach," Proceedings ECOOP'92, O. Lehrmann Madsen (Ed.), LNCS 615, Springer-Verlag, 1992.

[Casa97a] Eduoardo Casais and Antero Taivalsaari, "Object-Oriented Software Evolution and Re-engineering (Special Issue)," Journal of Theory and Practice of Object Systems (TAPOS), Vol. 3(4), 1997.

[Chid94a] Shyam R. Chidamber and Chris F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, Vol. 20(6), June 1994.

[Deme99c] Serge Demeyer, Stéphane Ducasse and Michele Lanza, "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization," Proceedings WCRE'99, Francoise Balmas, Mike Blaha and Spencer Rugaber (Ed.), IEEE Computer Society Press, 1999.

[Deme99d] Serge Demeyer, Stéphane Ducasse and Sander Tichelaar, "Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering," Proceedings UML'99, Bernhard Rumpe (Ed.), LNCS 1723, Springer-Verlag, Fort Collins, Colorado, 1999.

[Duca99b] Stéphane Ducasse, Matthias Rieger and Serge Demeyer, "A Language Independent Approach for Detecting Duplicated Code," Proceedings ICSM'99, Hongji Yang and Lee White (Ed.), IEEE Computer Society Press, 1999.

[Fent97a] Norman Fenton and Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, Second edition, International Thomson Computer Press, London, UK, 1997.

[Flor97a] Gert Florijn, Marco Meijers and Pieter van Winsen, "Tool Support for Object-Oriented Patterns," Proceedings ECOOP'97, Mehmet Aksit and Satoshi Matsuoka (Ed.), LNCS 1241, Springer-Verlag, 1997.

[Fowl99a] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[Gold95a] Adele Goldberg and Kenneth S. Rubin, Succeeding With Objects: Decision Frameworks for Project Management, Addison-Wesley, Reading, Mass., 1995.

[Haun95a] Jim Haungs, "A technical overview of VisualWorks 2.0," The Smalltalk Report, January 1995.

[Hend96a] Brian Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity, Prentice-Hall, 1996.

[Hond98a] Koen De Hondt, A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems, Vrije Universiteit Brussel, Brussels - Belgium, December, 1998, Ph.D. dissertation [ftp://progftp.vub.ac.be/dissertation/].

[Howa95a] Tim Howard, The Smalltalk Developer's Guide to VisualWorks, SIGS Books, 1995.

[Jaza99a] Mehdi Jazayeri, Claudio Riva and Harald Gall, "Visualizing Software Release Histories: The Use of Color and Third Dimension, " Proceedings ICSM'99, Hongji Yang and Lee White (Ed.), IEEE Computer Society Press, 1999.

[John92a] Ralph E. Johnson, "Documenting Frameworks using Patterns," Proceedings OOPSLA '92, ACM Press, 1992.

[John93b] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation," Object Technologies for Advanced Software - First JSSST International Symposium, Lecture Notes in Computer Science, Vol. 742, Springer-Verlag, 1993.

[Kell99a] Rudolf Keller, Reinhard Schaur, Sébastien Robitaille, Patrick Pagé, "Pattern-Based Reverse-Engineering of Design Components," Proceedings ICSE'99, IEEE Computer Society Press, 1999.

[Kohl98a] Gerd Kohler, Heinrich Rust and Frank Simon, "Assessment of Large Object-Oriented Software Systems: A Metrics Based Process," Object-Oriented Technology (ECOOP'98 Workshop Reader), Serge Demeyer and Jan Bosch (Ed.), LNCS 1543, Springer-Verlag, 1998.

[Kont97a] Kostas Kontogiannis, "Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics," Proceedings WCRE'97, Ira Baxter , Alex Quilici and Chris Verhoef (Ed.), IEEE Computer Society Press, 1997.

[Lagu97a] B. Lague and , D. Proulx and , E. Merlo and , J. Mayrand and and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," Proceedings ICSM'97, IEEE Computer Society Press, 1997.

[Lang95a] Danny B. Lange and Yuichi Nakamura, "Interactive Visualization of Design Patterns can help in Framework Understanding," Proceedings OOPSLA'95, ACM Press, 1995.

[Lore94a] Mark Lorenz and Jeff Kidd, Object-Oriented Software Metrics: A Practical Approach, Prentice-Hall, 1994.

[Mari98a] Radu Marinescu, "Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems," Object-Oriented Technology (ECOOP'98 Workshop Reader), Serge Demeyer and Jan Bosch (Ed.), LNCS 1543, Springer-Verlag, 1998.

[Moor96a] Ivan Moore, "Automatic Inheritance Hierarchy Restructuring and Method Refactoring," Proceedings OOPSLA '96, ACM Press, 1996.

[Murp97a] Gail Murphy and David Notkin, "Reengineering with Reflexion Models: A Case Study," IEEE Computer, Vol. 30(8), August 1997.

[Opdy92b] William F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois, 1992. [ftp://st.cs.uiuc.edu/pub/papers/refactoring/]

[Opdy93a] William F. Opdyke and Ralph E. Johnson, "Creating Abstract Superclasses by Refactoring," Proceedings CSC'93, ACM Press, 1993.

[Pauw93a] Wim De Pauw, Richard Helm, Doug Kimelman and John Vlissides, "Visualizing the Behavior of Object-Oriented Systems," Proceedings OOPSLA '93, ACM Press.

[Rich99a] Tamar Richner and Stéphane Ducasse, "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information," Proceedings ICSM'99, Hongji Yang and Lee White (Ed.), IEEE Computer Society Press, 1999.

[Robe97a] Don Roberts, John Brant and Ralph E. Johnson, "A Refactoring Tool for Smalltalk," Journal of Theory and Practice of Object Systems (TAPOS), Vol. 3(4), 1997.

[Schu98a] Benedikt Schulz, Thomas Genssler, Berthold Mohr and Walter Zimmer, "On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems.," Proceedings TOOLS 27 - Asia '98, IEEE Computer Society Press, 1998.

[Toku99a] Lance Tokuda and Don Batory, "Evolving Object-Oriented Designs with Refactorings," Proceedings ASE'99, IEEE Computer Society Press, 1999

[Wate94a] Richard C. Waters and Elliot Chikofsky, "Reverse Engineering: Progress Along Many Dimensions (Special Issue)," Communications of the ACM, Vol. 37(5), May 1994.

[Will96a] Linda Wills and Philip Newcomb, "Reverse Engineering (Special Issue)," Automated Software Engineering, Vol. 3(1-2), June 1996.

[Will96b] Linda Wills and James H. Cross, "Recent Trends and Open Issues in Reverse Engineering," Automated Software Engineering, Vol. 3(1-2), June, 1996.

[Wuyt98a] Roel Wuyts, "Class-management using Logical Queries, Application of a Reflective User Interface Builder," Proceedings TOOLS 26 - USA '98, IEEE Computer Society Press, 1998.