A Pattern Language for Reverse Engineering

v0.8 — July 12, 2000 11:26 am

Alpha-release of a part of forthcoming book
"Object-Oriented Reengineering, a Pattern-based Approach"
http://www.iam.unibe.ch/~famoos/patterns/

Serge Demeyer[(*)], Stéphane Ducasse[(+)], Oscar Nierstrasz[(+)]
[(*)] University of Antwerp - LORE - http://win-www.uia.ac.be/u/sdemey/
[(+)] University of Berne - SCG - http://www.iam.unibe.ch/~scg/

Abstract. Since object-oriented programming is usually associated with iterative development, reverse engineering must be considered an essential facet of the object-oriented paradigm. The reverse engineering pattern language presented here summarises the reverse engineering experience gathered as part of the FAMOOS project, a project with the explicit goal of investigating reverse and reengineering techniques in an object-oriented context.

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 program Project no. 21975 (FAMOOS).

Chapter 1

Reverse Engineering Patterns

-- ***REVIEWER NOTE:***
This chapter of the pattern language is pretty stable. Some minor details that might be changed:
- introduction must glue better with what comes before and after
- overview table and figure must be adapted
- pattern form must conform with the actual patterns

1. Introduction

This pattern language describes how to reverse engineer an object-oriented software system. Reverse engineering might seem a bit strange in the context of object-oriented development, as this term is usually associated with "legacy" systems written in languages like COBOL and Fortran. Yet, reverse engineering is very relevant in the context of object-oriented development as well, because the only way to achieve a good object-oriented design is recognized to be iterative development (see [Booc94a], [Gold95a], [Jaco97a], [Reen96a]). Iterative development involves refactoring existing designs and consequently, reverse engineering is an essential facet of any object-oriented development process.

The patterns have been developed and applied during the FAMOOS project (http://www.iam.unibe.ch/~famoos/); a project with had the explicit goal to produce a set of reengineering techniques and tools to support the development of object-oriented frameworks. Many if not all of the patterns have been applied on software systems provided by the industrial partners in the project (i.e., Nokia and Daimler-Chrysler). These systems ranged from 50.000 lines of C++ up until 2,5 million lines of Ada. Where appropriate, we refer to other known uses we were aware of while writing.

Acknowledgments. We would like to thank our EuroPLoP shepherds Mary Lynn Manns (2000), Kyle Brown (1999), Kent Beck and Charles Weir (1998) and all participants of the writers workshops where parts of this language has been discussed. Of course there is also Tim Cox, our contact person with the publisher: thanks for your patience —we hope we will not disappoint you. Next, we thank all participants of the FAMOOS project for providing such a fruitful working context. And finally, we thank our colleagues in Berne, both in and outside the FAMOOS team: by workshopping earlier versions of this pattern language you have greatly improved this manuscript.

2. Clusters of Patterns

The pattern language has been divided into *clusters* where each cluster groups a number of patterns addressing a similar reverse engineering situation. The clusters correspond roughly to the

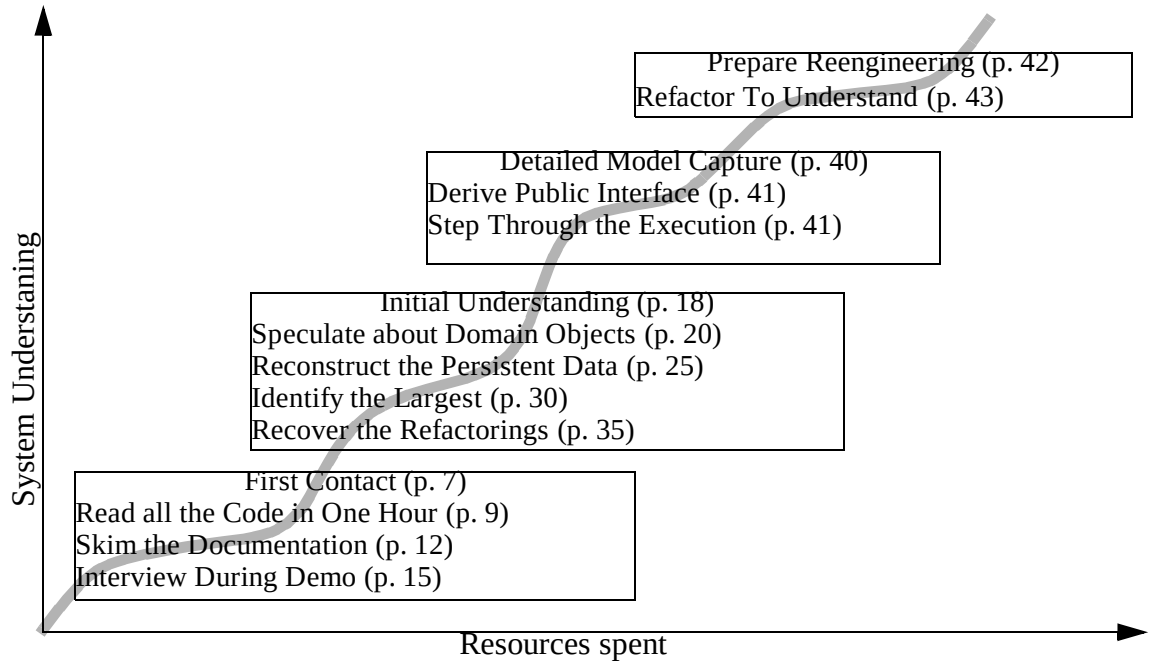


Figure 1 Overview of the pattern language using clusters.
Illustrating how the understanding gradually increases with the amount of resources you spend

different phases one encounters when reverse engineering a large software system. Below is a short description for each of the clusters, while figure 1 provides a road map.

- First Contact (p. 7)**.** This cluster groups patterns telling you what to do when you have your very first contact with a software system.
- Initial Understanding (p. 18)**.** Here, the patterns tell you how to obtain an initial understanding of a software system, mainly documented in the form of class diagrams.
- Detailed Model Capture (p. 40)**.** The patterns in this cluster describe how to get a detailed understanding of a particular component in your software system.
- Prepare Reengineering (p. 42)**.** Since reverse engineering often goes together with reengineering, this cluster includes some patterns that help you prepare subsequent reengineering steps.

3. Risk Factors

Reverse engineering projects include a lot of uncertainty, and to control these uncertainties some form of *risk management* is necessary. Consequently, with each phase of the reverse engineering process, you should identify potential risks of project delays and take appropriate actions to reduce these risks[Boeh89a].

To evaluate the applicability of a pattern from a risk management perspective, we introduce a number of *risk factors*. We present these risk factors at the beginning of each cluster to describe the most important threats that may jeopardize your reverse engineering project. We also use these risk factors as a way to compare all patterns by showing how each pattern reduces the corresponding risk factor (see Table 1).

	Limited Resources	Techniques and Tools	Reliable Information	Accurate Abstraction	Sceptical Colleagues
First Contact (p. 7)					
Read all the Code in One Hour (p. 9)	++	++	+	-	+
Skim the Documentation (p. 12)	++	++	-	+	-
Interview During Demo (p. 15)	++	+	/	+	-
Initial Understanding (p. 18)					
Speculate about Domain Objects (p. 20)	-	+	+	++	+
Reconstruct the Persistent Data (p. 25)	-	-	+	+	++
Identify the Largest (p. 30)	+	+	-	-	-
Recover the Refactorings (p. 35)	-	-	+	+	++
Detailed Model Capture (p. 40)					
Derive Public Interface (p. 41)	-	-	++	+	++
Step Through the Execution (p. 41)	-	-	++	+	++
Prepare Reengineering (p. 42)					
Refactor To Understand (p. 43)	--	--	++	++	+

Table 1: How each pattern reduces the risk.
Very well: ++, Well: +, Neutral: /, Rather Well: -, Not Well: --.
Limited Resources: The less resources you need to apply, the better.
Techniques and Tools: The less techniques and tools required, the better.
Reliable Information: The more reliable the information you get, the better.
Accurate Abstraction: The more abstract the information obtained, the better.
Sceptical Colleagues: The more credibility you gain, the better.

- **Limited Resources.** Because your resources are limited you must be selective in which parts of the system to reverse engineer. However, if you select the wrong parts, you will have wasted some of your precious resources. Therefore, in general *the less resources you need to apply, the smaller the risk of wasting them*.
- **Techniques and Tools.** For reverse engineering large scale systems, you need to apply techniques probably accompanied with tools. However, techniques and tools shape your thoughts and good reverse engineering, requires an unbiased opinion. Also, techniques and tools do require resources which you might not be willing to spend. In general, the *less techniques and tools required, the smaller the risk of biased information*.

- **Reliable Information.** A reverse engineer is much like a detective that solves a mystery from the scarce clues that are available [Will96b]. As with all good detective stories, the different clues and testimonies contradict each other and often key information is missing. Your challenge is to assess which information is reliable to deduce the missing pieces and dismiss the inconsistencies. In general, *the more reliable the information you start from, the smaller the risk to make the wrong deductions*.
- **Accurate Abstraction.** The whole idea of understanding the inner complexities of a software system is to construct mental models of portions of it, thus a process of abstraction. However, the more abstract you get, the easier it is to omit an essential piece of data. Only when you are able to separate the essential from the incidental, you can claim to understand the system. Therefore, *the more data you can discard without losing accuracy, the smaller the risk of misunderstanding*.
- **Sceptical Colleagues.** As a reverse engineer, you must deal with three kinds of colleagues. The first category are the faithful, the people who believe that reverse engineering is necessary and who thrust that you are able to do it. The second is the category of the sceptic, who believe this reverse engineering of yours is just a waste of time and that its better to start the whole project from scratch. The third category is the category of the fence sitters, who do not have a strong opinion on whether this reverse engineering will pay off, so they just wait and see what happens. To save your reverse engineering from ending up in the waste bag, you must keep convincing the faithful, gain credit with the fence sitters and be wary of the sceptic. In general, *the more credibility you gain, the smaller the risk you're project will be cancelled*.

Table 1 shows an overview of how the different patterns reduce the risks. This view is especially important because it helps you choosing the most appropriate pattern. For instance, it shows that Read all the Code in One Hour (p. 9) and Skim the Documentation (p. 12) take about the same amount of resources and also require about the same amount of techniques and tools (very little, hence the ++), yet score differently on the reliability and abstraction level of the resulting information. On the other hand, Speculate about Domain Objects (p. 20) requires more resources, techniques and tools then the previous two (i.e., -), but achieves better results in terms of reliable and abstract information.

4. Format of a Reverse Engineering Pattern

All the reverse engineering pattern presented make use of the following format.

- **Name.** Names the pattern after the solution it proposes. The pattern names are verb phrases to stress the action implied in applying them.
- **Intent.** Provides a thumbnail summary of the pattern.
- **Problem.** Describes the problem the pattern is solving. The section starts with a single sentence summarizing the heart of the problem addressed by the pattern. The section continues with a *context* section, which presents the context in which the pattern is supposed to be applied and which should be read as the prerequisites that should be satisfied before considering the pattern. The problem may also include a *Symptoms* sections, which lists some indications you may use to know when the problem occurs.

- **Solution.** Proposes a solution to the problem that is applicable in the given context. The section starts with a few sentences summarizing the process one is following while applying the pattern. This section may include a *Steps* or a list of *Hints* to be taken in account when applying the solution.
- **Trade-offs.** Discusses the issues that should be considered when applying the pattern, e.g. what is the impact, what makes it difficult and when should it *not* be applied.
- **Example.** Provides a realistic example of when and how to apply the pattern.
- **Rationale.** Provides a justification of the problem and the solution, plus some technical discussion of why the solution solves the problem. May also include a discussion on the *Typical Causes* of the problem, as a way of reassuring people that the problem in itself does not necessary come from incompetence.
- **Known Uses.** Presents the known uses of this pattern. Note that all patterns in this pattern language have been developed and applied in the context of the FAMOOS project. However, this section also presents other reported uses of the pattern we were aware of while writing the pattern.
- **Related Patterns.** Links the pattern in a web of other patterns, explaining how the patterns work together to achieve the global goal of reverse engineering. The section includes a *What Next* section which tells you how you may use the output of this pattern as input for another one.

Chapter 2

First Contact

-- ***REVIEWER NOTE:***
This chapter of the pattern language is pretty stable. Some major issues that will change:
- add examples for all patterns (the idea is that there will be a single "running example" for all reverse engineering patterns)
- add a section in the introduction explaining in which order the patterns should be applied

Some minor details that will change:
- include "why is it difficult" in problem
- structure the trade-offs according to pros, cons, difficulties
- - adapt the "related patterns" to the list of all patterns in the book, plus some others from the various pattern catalogues
- find "known uses" if possible

All the reverse engineering patterns in this cluster are applicable in the very early stage of a reverse engineering project. In such a situation, you need an initial assessment of the software system to get a feeling for what you might expect later on during the project. Accomplishing a good initial assessment is difficult however, because you need results quickly and accurately.

Risk Reduction

The list below is sorted according to the impact the risk factor will have on later reverse engineering activities. To reduce the risk we define some generic principles that we emphasised in the patterns.

- Limited Resources**.** Wasting time early on in the project has severe consequences later on. *Consequently, consider time as your most precious resource and therefore defer all time-consuming activities until later.* This is especially relevant because in the beginning of a project you feel a bit uncertain and then it is tempting to start an activity that will keep you busy for a while, instead of something that confronts you immediately with the problems to address.
- Techniques and Tools**.** In the beginning of a reverse engineering project, you are in a bootstrap situation: you must decide which techniques and tools to apply but you lack a solid basis to make such a decision. *Consequently, choose for very simple techniques and very basic tools.*

- Reliable Information**.** Because you are unfamiliar with the system, it is difficult to assess which information is reliable. *Consequently, keep track of what information is known to be correct and what information remains to be verified.*
- Accurate Abstraction**.** At the beginning of the project you can not afford to be overwhelmed by too many details. *Consequently, favour techniques and tools that provide you with a general overview.*
- Sceptical Colleagues**.** This risk is often reinforced in the beginning of a reverse engineering project, because as a reverse engineer there is a good chance that you are a newcomer in a project team. *Consequently, pay attention to the way you communicate with your colleagues.*

Which one first?

The patterns in this cluster tell you how to exploit information resources like source code — Read all the Code in One Hour (p. 9), documentation — Skim the Documentation (p. 12) and system experts — Interview During Demo (p. 15). Afterwards you will probably want to Confer with Colleagues (p. 47) and then proceed with the patterns in Initial Understanding (p. 18).

-- *REVIEWER NOTE:*
Here should come a discussion on the order in which too apply the various patterns in this cluster.
Basic idea is that the order depends on whatever takes the least time to start (since according to risk management, time is the most critical resource).
Second idea is that all patterns together should take you no more than a week:

1 day reading code
+ 1 day reading documentation
+ 2 days talking to people
+ 1 day drawing conclusions

Read all the Code in One Hour

Intent

Make an initial evaluation of the condition of a software system by walking through its source code in a limited amount of time.

Problem

You need an initial assessment of the internal condition of a software system to plan further reverse engineering efforts.

Context

This problem is difficult because:

- The internal condition will vary quite a lot, depending on the people that have been involved in developing and maintaining the system.
- The system is large, so there is too much data to inspect for an accurate assessment.
- You're unfamiliar with the software system, so you do not know how to filter out what's relevant.

Yet, solving this problem is feasible because:

- You have the source code at your disposal, so you have reliable data.
- You have reasonable expertise with the implementation language being used, thus you can recognize programming idioms.

Solution

Take the source code to a room where you can work undisturbed (no telephones, no noisy colleagues). Grant yourself a reasonably short amount of study time (i.e., approximately one hour) to walk through the source code. Take notes sparingly to maximize the contact with the code.

After this reading time, take about the same time to produce a report about your findings, including list of

1. the important entities (i.e., classes, packages, ...)
2. the coding idioms applied (i.e., C++ [Copl92a], [Meye98a], [Meye96a]; Smalltalk [Beck97a]); and
3. the suspicious coding styles discovered (i.e., "code smells" [Fowl99a]). Keep this report short, and name the entities like they are mentioned in the source code.

Hints

The fact that you are limited in time should force you to think how you can extract the most useful information. Below are some hints for things to look out for.

- Some development teams apply *coding styles* and if they did, it is good to be aware of them. Especially naming conventions are crucial to scan code quickly.
- *Functional tests and unit tests* convey important information about the functionality of a software system.
- *Abstract classes and methods* reveal design intentions.
- Classes *high in the hierarchy* often define domain abstractions; their subclasses introduce variations on a theme.
- Occurrences of the *Singleton pattern* [Gamm95a] may represent information that is constant for the entire execution of a system.
- Surprisingly *large structures* often specify important chunks of functionality that should be executed sequentially.

Trade-offs

Pros

- **Efficient.** Reading the code in a short amount of time is very efficient as a starter. Indeed, by limiting the time and yet forcing yourself to look at all the code, you mainly use your brain and coding expertise to filter out what seems important. This is a lot more efficient than extracting human readable representations or organizing a meeting with all the programmers involved.
- **Unbiased.** By reading the code directly you get an unbiased view of the software system including a sense for the details and a glimpse on the kind of problems you are facing. Because the source code describes the functionality of the system --no more, no less-- it is the only accurate source of information.
- **Developers vocabulary.** Acquiring the vocabulary used inside the software system is essential to understand it and communicate about it with other developers. This pattern helps to acquire such a vocabulary.

Cons

- **Low abstraction.** Via this pattern, you will get some insight in the solution domain, but only very little on how these map onto problem domain concepts.

Difficulties

- **Misleading Comments.** Be careful with comments in the code. Comment can help you in understanding what a piece of software is supposed to do. However, just like other kinds of documentation, comments can be outdated, obsolete or simply wrong.

Example

...

-- Here will follow a "running example" (= same example throughout the whole reverse engineering patterns)

Known Uses

While writing this pattern, one of our team members applied it to reverse engineer the Refactoring Browser [Robe97a]. The person was not familiar with Smalltalk, yet was able to identify code smells such as "Large Constructs" and "Duplicated Code". Even without Smalltalk experience it was possible to get a feel for the system structure by a mere inspection of class interfaces. Also, a special hierarchy browser did help to identify some of the main classes and the comments provided some useful hints to what parts of the code were supposed to do. Applying the pattern took a bit more than an hour, which seemed enough for a relatively small system and slow progress due to the unfamiliarity with Smalltalk.

The original pattern was suggested by Kent Beck, who stated that it is one of the techniques he always applies when starting consultancy on an existing project. Since then, other people have acknowledged that it is one of their common practices.

Related Patterns

If possible, Read all the Code in One Hour (p. 9) in conjunction with Skim the Documentation (p. 12) to maximize your chances of getting a coherent view of the system. To guide your reading, you may precede this pattern with Interview During Demo (p. 15), but then you should be aware that this will bias your opinion.

What Next

This pattern results in a list of (i) the important entities (i.e., classes, packages, ...); (ii) the presence of standard coding idioms and (iii) the suspicious coding styles discovered. This is enough to start Speculate about Domain Objects (p. 20) and Reconstruct the Persistent Data (p. 25) to improve the list of important entities. Depending on whether you want to wait for the results of Skim the Documentation (p. 12), you should consider to Confer with Colleagues (p. 47).

Skim the Documentation

Intent

Make an initial guess at the functionality of a software system by reading its documentation in a limited amount of time.

Problem

You need an initial idea of the functionality provided by the software system in order to plan further reverse engineering efforts.

Context

This problem is difficult because:

- The functionality will have changed over time and many these changes will be undocumented.
- The system is large, so there is too many data to inspect for an accurate assessment.
- You're unfamiliar with the software system, so you do not know how to filter out what's relevant.

Yet, solving this problem is feasible because:

- You have the documentation at your disposal, so you have at least an accurate description of how the system behaved in the past.
- You are able to interpret the formal (i.e., state-charts) and semi-formal (i.e., use-cases) specifications contained within the documentation, so you are able to understand their implication.

Solution

Take the documentation to a room where you can work undisturbed (no telephones, no noisy colleagues). Grant yourself a reasonably short amount of study time (i.e., approximately one hour) to scan through the documentation. Take notes sparingly to maximize the contact with the documentation.

After this reading time, take about the same time to produce a report about your findings, including a list of

1. the important requirements;
2. the important features;
3. the important constraints;
4. references to relevant design information.

Include your opinion on how reliable and useful each of these are. Keep this report as short as possible and avoid redundancy at all cost (among others, use references to sections and/or page numbers in the documentation).

Depending on the goal of the reverse engineering project and the kind of documentation you have at your disposal, you may steer the reading process to match your main interest. For instance, if you want insight into the original system requirements then you should look inside the analysis documentation, while knowledge about which features are actually implemented should be collected from the end-user manual or tutorial notes. If you have the luxury of choice, avoid spending too much time to understand the design documentation (i.e., class diagrams, database schema's, ...): rather record the presence and reliability of such documents as this will be of great help in later stages of the reverse engineering.

Check whether the documentation is outdated with respect to the actual system. Always compare version dates with the date of delivery of the system and make note of those parts that you suspect unreliable.

Avoid to read the documentation electronically if you are not sure to gain significant browsing functionality (e.g., hypertext links in HTML or PDF). This way you will not spend your time mastering the tool instead of actually reading the documentation..

Hints

The fact that you are limited in time should force you to think how you can extract the most useful information. Below are some hints for things to look out for.

- A *table of contents* gives you a quick overview of the structure and the information presented.
- *Version numbers and dates* tell you how up to date the documentation is.
- *Figures* are a always a good means to communicate information. A list of figures, if present, may provide a quick access path to certain parts of the documentation.
- *Screen-dumps, sample print-outs, sample reports, command descriptions,* reveal a lot about the functionality provided by the system.
- *Formal specifications*, if present, usually correspond with crucial functionality.
- An *index*, if present contains the terms the author considers significant.

Trade-offs

Pros

...

Cons

...

Difficulties

...

Example

...

-- Here will follow a "running example" (= same example throughout the whole reverse engineering patterns)

Rationale

Knowing what functionality is provided by the system is essential for reverse engineering. Documentation provides an excellent means to get an external description of this functionality.

However, documentation is either written before or after implementation, thus likely to be out of sync with respect to the actual software system. Therefore, it is necessary to record the reliability. Moreover, documentation comes in different kinds, i.e. requirement documents, technical documentation, end-user manuals, tutorial notes. Depending on the goal of your reengineering project, you will record the usability of each of these documents. Finally, documentation may contain large volumes of information thus reading is time consuming. By limiting the time you spend on it, you force yourself to classify the pieces of information into the essential and the less important.

Known Uses

...

Related Patterns

You may or may not want to Skim the Documentation (p. 12) before Read all the Code in One Hour (p. 9) depending on whether you want to keep your mind free or whether you want some subjective input before reading the code. Interview During Demo (p. 15) can help you to collect a list of entities you want to read about in the documentation.

What Next

This pattern results in a list of (i) the important requirements; (ii) the important features (iii); the important constraints; (iv) references to relevant design information plus an opinion on how reliable and useful each of these are. Together with the result of Read all the Code in One Hour (p. 9) and Interview During Demo (p. 15) this is a good basis to Confer with Colleagues (p. 47) and then proceed with Initial Understanding (p. 18).

Interview During Demo

Intent

Obtain an initial feeling for the functionality of a software system by seeing a demo and interviewing the person giving the demo.

Problem

You need an idea of the typical usage scenarios and the main features of a software system in order to plan further reverse engineering efforts.

Context

This problem is difficult because:

- Typical usage scenarios vary quite a lot depending on the type of user.
- If you ask the users, they have a tendency to complain about what's wrong, while for reverse engineering purposes you're mainly interested in what's valuable.
- The system is large, so there is too many data to inspect for an accurate assessment.
- You're unfamiliar with the software system, so you do not know how to filter out what's relevant.

Yet, solving this problem is feasible because:

- You have access to some key persons (both users, managers and maintainers) in the organisation around the software system which can demonstrate and explain its usage.

Solution

Observe the system in operation by seeing a demo and interviewing the person who is demonstrating. Note that the interviewing part is at least as enlightening as the demo.

After this demo, take about the same time to produce a report about your findings, including

1. some typical usage scenarios;
2. the main features offered by the system and whether they are appreciated or not;
3. the system components and their responsibilities;
4. bizarre anecdotes that reveal the folklore around using the system.

Hints

The person who is giving the demo is crucial to the outcome of this pattern so take care when selecting the person. Therefore, consider this pattern several times with different persons giving the demo. This way you will see variations in what people find important and you will hear different opinions about the value of the software system. Always be wary of enthusiastic supporters or fervent opponents: although they will certainly provide relevant information, you must spend extra time to look for complementary opinions in order to avoid prejudices.

Below are some hints concerning people you should be looking for, what kind of information you may expect from them and what kind of questions you should ask them.

- An *end-user* should tell you how the system looks like from the outside and explain you some detailed usage scenarios based on the daily working practices. Ask about the situation in the company before the software system was introduced to assess the scope of the software system within the business processes. Probe for the relationship with the computer department to divulge bizarre anecdotes.
- A person from the *maintenance/development team* should clarify the main requirements and architecture of a system. Inquire how the system has evolved since delivery to reveal some of the knowledge that is passed on orally between the maintainers. Ask for samples of bug reports and change requests to assess the thoroughness of the maintenance process.
- A *manager* should inform you how the system fits within the rest of the business domain. Ask about the business processes around the system to check for unspoken motives concerning your reverse engineering project. This is important as reverse engineering is rarely a goal in itself, it is just a means to achieve another goal.

Trade-offs

Pros

...

Cons

...

Difficulties

...

Example

...

-- Here will follow a "running example" (= same example throughout the whole reverse engineering patterns)

Rationale

Interviewing people working with a software system is essential to get a handle on the important functionality and the typical usage scenario's. However, asking pre-defined questions does not work, because in the initial phases of reverse engineering you do not know what to ask. Merely asking what people like about a system will result in vague or meaningless answers. On top of that, you risk getting a very negative picture because people have a tendency to complain.

Therefore, hand over the initiative to the user by requesting for a demo. First of all, a demo allows users to tell the story in their own words, yet is comprehensible for you because the demo

imposes some kind of tangible structure. Second, because users must start from a running system, they will adopt a more positive attitude explaining you what works. Finally, during the course of the demo, you can ask lots of precise questions, getting lots of precise answers, this way digging out the expert knowledge about the system's usage.

Known Uses

One anecdote from the very beginning of the FAMOOS project provides a very good example for the potential of this pattern. For one of the case studies ---a typical example of a 3-tiered application with a database layer, domain objects layer and user-interface layer--- we were asked 'to get the business objects out'. Two separate individuals were set to that task, one took a source code browser and a CASE tool and extracted some class diagrams that represented those business objects. The other installed the system on his local PC and spent about an hour playing around with the user interface (that is, he demonstrated the system to himself) to come up with a list of ten questions about some strange observations he made. Afterwards, a meeting was organized with the chief analyst-designer of the system and the two individuals that tried to reverse engineer the system. When the analyst-designer was confronted with the class-diagrams he confirmed that these covered part of his design, but he couldn't tell us whether there was something missing, nor did he tell us anything about the rationale behind his design. It was only when we asked him the ten questions that he launched off into a very enthusiastic and very detailed explanation of the problems he was facing during the design --- he even pointed to our class diagrams during his story! After having listened to the analyst-designer, the first reaction of the person that extracted the class diagrams from the source code was 'Gee, I never read that in the source code'.

Related Patterns

For optimum results, you should perform several attempts of Interview During Demo (p. 15) with different kinds of people. Depending on your taste, you may perform these attempts before, after or interwoven with Read all the Code in One Hour (p. 9) and Skim the Documentation (p. 12).

What Next

This pattern results in (i) some typical usage scenarios; (ii) the main features offered by the system and whether they are appreciated or not; (iii) the system components and their responsibilities; (iv) bizarre anecdotes that reveal the folklore around using the system. Together with the results of Read all the Code in One Hour (p. 9) and Skim the Documentation (p. 12)this is a good basis to Confer with Colleagues (p. 47) and then move on to Initial Understanding (p. 18).

Chapter 3

Initial Understanding

-- ***REVIEWER NOTE:***
This chapter of the pattern language has been sent to EuroPLOP2000; comments from the writers workshops have not yet been included
Some major issues that will change:
- add examples for all patterns (the idea is that there will be a single "running example" for all reverse engineering patterns)
- add a section in the introduction explaining in which order the patterns should be applied
Some minor details that will change:
- include "why is it difficult" in problem
- structure the trade-offs according to pros, cons, difficulties (in some cases already done)
- adapt the "related patterns" to the list of all patterns in the book, plus some others from the various pattern catalogues
- find "known uses" if possible

The patterns in First Contact (p. 7) should have helped you getting some first ideas about the software system. Now is the right time to refine those ideas into an initial understanding and to document that understanding in order to support further reverse engineering activities. The main priority in this stage of reverse engineering is to get an accurate understanding without spending too much time on the hairy details.

The patterns in this cluster tell you:

- How to extract a domain model from source code (Speculate about Domain Objects (p. 20)), with one variant concerning pattern extraction (Speculate about Patterns (p. 23)) and another concerning process architecture extraction (Speculate about Process Architecture (p. 23)).
- How to extract a class model from a database (Reconstruct the Persistent Data (p. 25)).
- How to identify important chunks of functionality (Identify the Largest (p. 30)).
- How to recognize which refactorings have been applied in the past (Recover the Refactorings (p. 35)).

With this information you will probably want to proceed with Detailed Model Capture (p. 40).

Risk Reduction

The list below is sorted according to the impact the risk factor will have on later reverse engineering activities. To reduce the risk we define some generic principles that we emphasised in the patterns.

- Reliable Information**.** Since the initial understanding will influence the rest of your project, accuracy is the single most important aspect. *Consequently, take special precautions to make the extracted models as reliable as possible.* In particular, plan for an incremental approach, where you will improve your initial understanding during later activities.
- Limited Resources**.** Documenting the initial understanding is crucial as all subsequent reverse engineering activities will benefit from it. *Consequently, consider* Initial Understanding (p. 18) *a very important activity and therefore plan a substantial amount of your resources here.* However, via an incremental approach you can stretch your resources in time, i.e. you will not allocate all your resources early in the project but rather some of the resources allocated later should improve the understanding (and corresponding models) acquired early.
- Techniques and Tools**.** While obtaining an initial understanding, you can afford the time and money to apply some heavyweight techniques and purchase some expensive tools. *Yet —because accuracy is so important— never rely exclusively on techniques and tools and always make a critical assessment of their output.*
- Accurate Abstraction**.** Understanding means building mental models and models are meant to strip away details. Yet, details are crucial to the overall system [Broo87a]. *Consequently, favour different models where each emphasizes a different perspective and choose the most appropriate ones when the situation calls for it.*
- Sceptical Colleagues**.** Good models of a software system help a lot because they greatly improve the communication within a team. However, since they strip away details, you risk to offend those people who spend their time on these details. Also, certain notations and diagrams may be new to people, and then your diagrams will just be ignored. *Consequently, take care in choosing which models to produce and which notations to use — they should be helpful to all members of the team.*

Speculate about Domain Objects

AKA: Map business objects onto classes

Progressively refine a domain model against source code, by defining hypotheses about which objects should be represented in the system and checking these hypotheses against the source code.

Problem

You do not know how concepts from the problem domain are mapped onto classes in the source-code.

Context

This problem is difficult because:

- There are many problem domain concepts and there is a myriad of ways to represent them in the programming language used.
- Lots of source-code won't have anything to do with representing the problem domain but rather with implementing solution domain issues (user-interface, database, ...).

Yet, solving this problem is feasible because:

- You have a rough understanding of the system's functionality, thus an initial idea of what the problem domain should represent.
- You have development expertise, so you can imagine how you would model the problem domain yourself.
- You are somewhat familiar with the main structure of the source code and you have the necessary tools to browse it, so that you can find your way around.

Solution

Use your development expertise to conceive a hypothetical class model representing the problem domain. Refine that model by inspecting whether the names in the class model occur in the source code and by adapting the model accordingly. Repeat the process until you're class model stabilizes.

Steps

1. With your understanding of the requirements and usage scenarios, develop a class model that serves as your initial hypothesis of what to expect in the source code. For the names of the classes, operations and attributes make a guess based on your experience and potential naming conventions (see Skim the Documentation (p. 12)).
2. Enumerate the names in the class model (that is, names of classes, attributes and operations) and try to find them in the source code, using whatever tools you have available. Take care as names inside the source-code do not always match with the con-

cepts they represent.[1] To counter this effect, you may rank the names according to the likelihood that they appear in the source code.

3. Keep track of the names which appear in source code (confirm your hypotheses) and the names which do not match with identifiers in the source code (contradict your hypothesis). Note that mismatches are positive, as these will trigger the learning process that you must go through when understanding the system.
4. Adapt the class model based on the mismatches. Such adaptation may involve
   (a) *renaming*, when you discover that the names chosen in the source code do not match with your hypothesis;
   (b) *remodelling* **(@refactoring ?@)**, when you find out that the source-code representation of the problem domain concept does not correspond with what you have in your model. For instance, you may transform an operation into a class, or an attribute into an operation.
   (c) *extending*, when you detect important elements in the source-code that do not appear in your class diagram;
   (d) *seeking alternatives*, when you do not find the problem domain concept in the source-code. This may entail trying synonyms when there are few mismatches but may also entail defining a completely different class model when there are a lot of mismatches.
5. Repeat from step 2 until you obtain a class model that is satisfactory.

Hints

The most difficult step while applying this pattern is the development of an initial hypotheses. Below are some hints that may help you to come up with a first class model.

- The usage scenarios that you get out of Interview During Demo (p. 15) may serve to define some use cases that in turn help to find out which objects fulfil which roles. (See [Jaco92a] for use cases and [Reen96a] for role modelling.)
- Use the noun phrases in the requirements as the initial class names and the verb phrases as the initial method names, as suggested in responsibility-driven design (See [Wirf90b] for an in depth treatment.)

Trade-offs

Pros

- **Scale.** Speculating about what you'll find in the source code is a technique that scales up well. This is especially important because for large object-oriented programs (over a 100 classes) it quickly becomes impractical to apply the inverse process, which is building a complete class model from source code and afterwards condensing it by removing the noise. Besides being impractical, the latter approach does not bring a lot of understanding, because you are forced to focus on the irrelevant noise instead of the important concepts.

1. In one particular reverse engineering experience, we were facing source code that was a mixture of English and German. As you may expect, this complicates matters a lot.

- **Applicability**. The pattern is applicable in all situations where you have the source code available.
- **Return on Investment**. The technique is quite cheap in terms of resources and tools, definitely when considering the amount of understanding one obtains.

Cons

- **Requires Implementation Expertise.** A large repertoire of knowledge about idioms, patterns, algorithms, techniques is necessary to recognize what you see in the source code. As such, the pattern should preferably be applied by experts in the implementation language.

Difficulties

- **Consistency.** You should plan to keep the class model up to date while your reverse engineering project progresses and your understanding of the software system grows. Otherwise your efforts will be wasted. If your team makes use of a version control system, make sure that the class model is controlled by that system too.

Example

...

-- Here will follow a "running example" (= same example throughout the whole reverse engineering patterns)

Rationale

If you Speculate about Domain Objects (p. 20), you go through a learning process which gains a true understanding. In that sense, the contradictions of your hypotheses are as important as the confirmations, because mismatches force you to consider alternative solutions and assess the pros and cons of these.

Known Uses

In [Murp97a], there is a report of an experiment where a software engineer at Microsoft applied this pattern (it is called 'the Reflexion Model' in the paper) to reverse engineer the C-code of Microsoft Excel. One of the nice sides of the story is that the software engineer was a newcomer to that part of the system and that his colleagues could not spend too much time to explain him about it. Yet, after a brief discussion he could come up with an initial hypothesis and then use the source code to gradually refine his understanding. Note that the paper also includes a description of a lightweight tool to help specifying the model, the mapping from the model to the source code and the checking of the code against the model.

The article [Bigg94a] reports several successful uses of this pattern (it is called the 'concept assignment problem' in the paper). The authors describe a special tool DESIRE, which includes advanced browsing facilities, program slicing, prolog-based query language, ...

Related Patterns

All the patterns in the First Contact (p. 7) cluster are meant to help you in building the initial hypothesis now to be refined via Speculate about Domain Objects (p. 20). Afterwards, some of the patterns in Detailed Model Capture (p. 40) (in particular, Step Through the Execution (p. 41)) may help you to improve this hypothesis.

What Next

After this pattern, you will have a class model representing the problem domain concepts. Other patterns will help you deriving other views on the system, for instance Reconstruct the Persistent Data (p. 25) when you want to learn about the valuable data inside a system, or Identify the Largest (p. 30) when you want to identify the important functionality, or Recover the Refactorings (p. 35) when you want to reconstruct the evolution process.

Consider to Confer with Colleagues (p. 47) after you did Speculate about Domain Objects (p. 20), in order to confirm you results with other findings.

Speculate about Patterns

Like Speculate about Domain Objects (p. 20)*, except that you build and refine a hypothesis about occurrences of architectural, analysis or design patterns.*

While having Read all the Code in One Hour (p. 9), you might have noticed some symptoms of patterns. Knowing which patterns have been applied in the system design may help a lot in understanding it: for instance a Singleton pattern may point to important system-wide services. You can use a variant of Speculate about Domain Objects (p. 20) to refine this knowledge. See the better known pattern catalogues [Gamm95a], [Busc96a], [Fowl97b] for patterns to watch out for. See also [Brow96c] for a discussion on tool support for detecting patterns.

Example

You are facing a 500 K lines C++ program, implementing a software system to display multimedia information in real time. Your boss asks you to look at how much of the source code can be resurrected for another project. After having Read all the Code in One Hour (p. 9), you noticed an interesting piece of code concerning the reading of the signals on the external video channel. You suspect that the original software designers have applied some form of observer pattern, and you want to learn more about the way the observer is notified of events. You will read the source code and trace interesting paths, this way gradually refining your assumption that the class "VideoChannel" is the subject being observed.

Speculate about Process Architecture

Like Speculate about Domain Objects (p. 20)*, except that you build and refine a hypothesis about the interacting processes in a distributed system.*

The object-oriented paradigm is often applied in the context of distributed systems with multiple cooperating processes. A variant of Speculate about Domain Objects (p. 20) may be applied to infer which processes exist, how they are launched, how they get terminated and how they interact. (See [Lea96a] for some typical patterns and idioms that may be applied in concurrent programming.)

Reconstruct the Persistent Data

Recover objects that are so valuable that they are stored in a database system.

Problem

You do not know which objects are critical for the functioning of the system, i.e. objects so vital that they must persist across different executions of your system and require special care in terms of back-up procedures and concurrency control.

Context

This problem is difficult because:

- Objects are run-time entities while most system descriptions are static.
- Run-time traces quickly generate huge amounts of data.

Yet, solving this problem is feasible because:

- The software system employs some form of a database to make its data persistent, thus there exists some form of database schema providing a static description of the data inside the database.
- You have some expertise with mapping data-structures from your implementation language onto a database schema, enough to apply the reverse process (i.e., reconstruct a class model from the database schema).
- The database comes with the necessary tools to inspect the actual objects inside the database, so you can exploit the presence of legacy data to fine-tune the reconstructed model.

Solution

Check the entities that are stored in the database, as these most likely represent valuable objects. Derive a class model representing those entities to document that knowledge for the rest of the team.

Steps

The steps below assume you start with a *relational database*, which is quite a typical situation with object-oriented systems. If you have another kind of database system, some of these steps may still be applicable.

Note that steps 1-3 are quite mechanical and can be automated quite easily.

1. Collect all table names and build a class model, where each table name corresponds to a class name.
2. For each table, collect all column names and add these as attributes to the corresponding class.

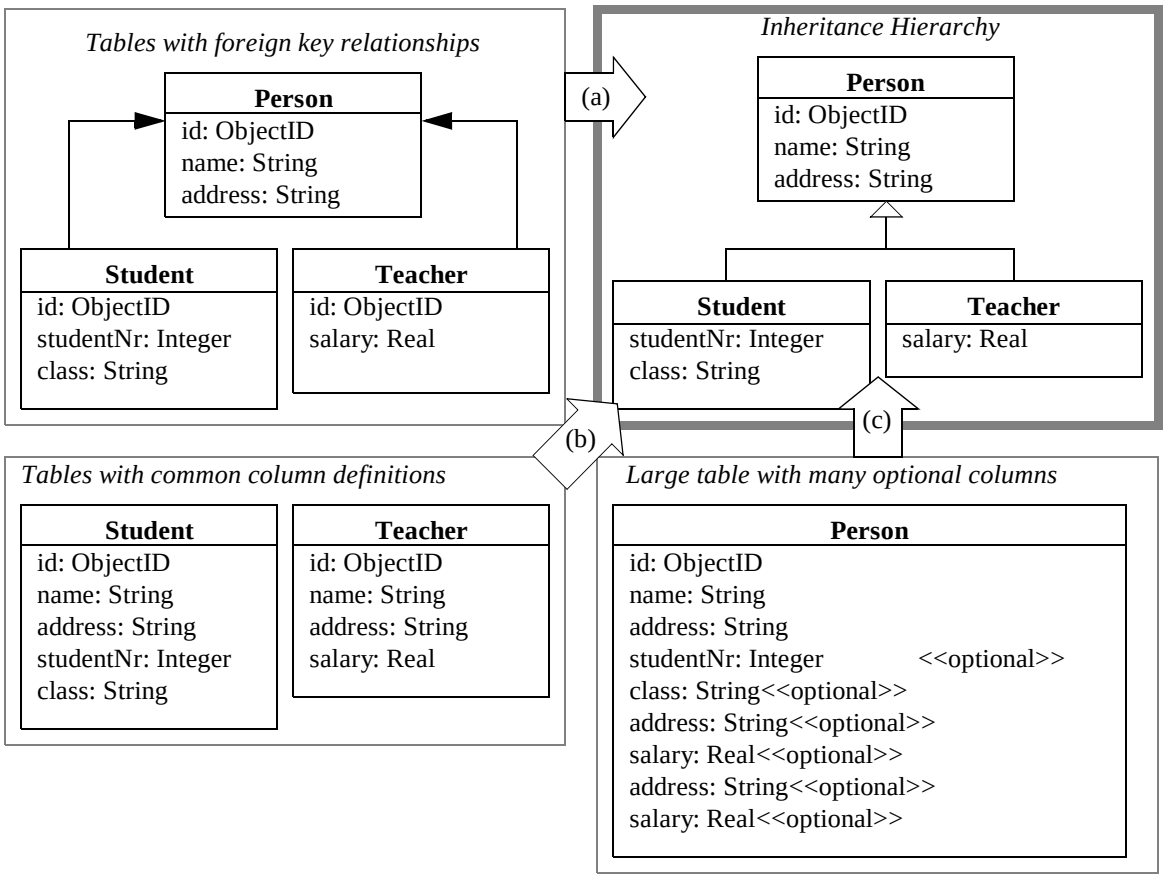


Figure 2 Mapping a series of relational tables onto an inheritance hierarchy.
(a) one to one; (b) rolled down; (c) rolled up

3. Collect all foreign keys relationships between tables and draw an association between the corresponding classes. (If the foreign key relationships are not maintained explicitly in the database schema, then you may infer these from column types and naming conventions.)

After the above steps, you will have a class model that represents the entities being stored in the relational database. However, because relational databases cannot represent inheritance relationships, there is still some cleaning up to do. (The terminology for the three representations of inheritance relations in steps 4-6 stems from [Fros94a].)

4. Check tables where the primary key also serves as a foreign key to another table, as this may be a "one to one" representation of an inheritance relationship inside a relational database. Examine the SELECT statements that are executed against these tables to see whether they usually involve a join over this foreign key. If this is the case, transform the association that corresponds with the foreign key into an inheritance relationship. (see figure 2 (a)).

5. Check tables with common sets of column definitions, as these probably indicate a situation where the class hierarchy is "rolled down" into several tables, each table representing one concrete class. Define a common superclass for each cluster of duplicated

column definitions and move the corresponding attributes inside the new class. To name the newly created classes, you can use your imagination, or better, check the source code for an applicable name. (see figure 2 (b))

6. Check tables with many columns and lots of optional attributes as these may indicate a situation where a complete class hierarchy is "rolled up" in a single table. If you have found such a table, examine all the SELECT statements that are executed against this table. If these SELECT statements explicitly request for subsets of the columns, then you may break this one class into several classes depending on the subsets requested (see figure 2 (c))

When you have incorporated the inheritance relationships, consider to improve the class model exploiting the presence of the legacy system as a source of information. In particular you can ...

-- **say something about data sampling and run-time inspection**

-- **say something about locating mapping code in the system itself**

Trade-offs

Pros

- **Team communication.** By capturing the database schema you will improve the communication within the reverse engineering team and with other developers associated with the project (in particular the maintenance team). Moreover, many if not all of the people associated with the project will be reassured by the fact that the data schema is present, because lots of development methodologies stress the importance of the data.
- **Model of critical information**. The database usually contains the critical data, hence the need to model it because whatever future steps you take you should guarantee that this critical data is maintained.

Cons

- **Limited Scope.** Although the database is crucial in many of today's software systems, it involves but a fraction of the complete system. As such, you cannot rely on this pattern alone to gain a complete view of the system.
- **Requires Database Expertise.** The pattern requires a good deal of knowledge about he underlying database plus structures to map the database schema into the implementation language. As such, the pattern should preferably be applied by people having expertise in mappings from the chosen database to the implementation language.

Difficulties

- **Polluted Database Schema.** The database schema itself is not always the best source of information to reconstruct a class model for the valuable objects. Many projects must optimise database access and as such often sacrifice a clean database schema. Also, the database schema itself evolves over time, and as such will slowly detoriate. Therefore, its is quite important to refine the class model using data sampling and run-time inspection.

Example

...

-- Here will follow a "running example" (= same example throughout the whole reverse engineering patterns)

Rationale

Having a well-defined central database schema is a common practice in larger software projects that deal with persistent data. Not only does it specify common rules on how to access certain data structures, it is also a great aid in dividing the work between team members. Therefore, it is a good idea to extract an accurate data model before proceeding with other reverse engineering activities.

Known Uses

The reverse engineering and reengineering of database systems is a well-explored area of research (see among others [Hain96a], [Prem94a], [Jahn97b]). Note the recurring remark that the database schema alone is too weak a basis and that data sampling and run-time inspection must be included for successful reconstruction of the data model.

- **Data sampling.** Database schemas only specify the constraints allowed by the underlying database system and model. However, the problem domain may involve other constraints not expressed in the schema. By inspecting samples of the actual data stored in the database you can infer other constraints.
- **Run-time inspection.** Tables in a relational database schema are linked via foreign keys. However, it is sometimes the case that some tables are always accessed together, even if there is no explicit foreign key. Therefore, it is a good idea to check at run-time which queries are executed against the database engine.

Related Patterns

Reconstruct the Persistent Data (p. 25) requires an initial understanding of the system functionality, as obtained by applying patterns in the cluster First Contact (p. 7).

There are some idioms, patterns and pattern languages that describe various ways to map object-oriented data structures on relational database counterparts. See among others [Kell98a], [Cold99a].

What Next

Reconstruct the Persistent Data (p. 25) results in a class model for the persistent data in your software system. Such a data model is quite rough, but it may serve as an ideal initial hypotheses to be further refined by applying Speculate about Domain Objects (p. 20). The data model should also be used as a collective knowledge that comes in handy when doing further reverse engineering efforts, for instance like in the clusters Detailed Model Capture (p. 40) and Prepare Reengineering

(p. 42). Consequently, consider to Confer with Colleagues (p. 47) after Reconstruct the Persistent Data (p. 25).

Identify the Largest

Identify important code by using a metrics tool and inspecting the largest entities.

Problem

You do not know where the important functionality is implemented in the million lines of source code you are facing.

Context

This problem is difficult because:

- There is no easy way to discern important from less important code.
- The system is large, so there is too much data to inspect for an accurate assessment.

Yet, solving this problem is feasible because:

- You have a metrics tool at your disposal, so you can quantify the size of entities in the source-code.
- You have the necessary tools to browse the source-code, so you can verify manually whether certain entities are indeed important.

Solution

Use the metrics tool to collect a limited set of measurements concerning the entities inside the software system (i.e., the inheritance hierarchy, the packages, the classes and the methods). Display the results in such a way that you can easily assess different measurements for the same entity. Browse the source code for the large or exceptional entities to determine whether the entity represents important functionality.

Steps

The following steps provide some heuristics to identify important functionality using metrics.

1. Identify large inheritance hierarchies.

   As inheritance is the most commonly used modelling concept in object-oriented systems it is a good idea to identify the largest subtree in the inheritance hierarchy as potential candidates for providing important functionality. To do this, compile a list of classes with the metrics Number of Descendant Classes — NDC (p. 50) and Hierarchy Nesting Level — HNL (p. 49) as the main indicators, and Number of Methods for Class — NOM (p. 48) plus Number of Attributes for Class — NOA (p. 48) as secondary indicators.

Sort the list according the main indicators to identify those classes at the root or at the bottom of the large inheritance hierarchies (see Table 2).

	NDC	HNL	NOM, NOA
(a) root of large inheritance hierarchy	large	small (~= 0)	Large values indicate a lot of impact on the subclasses.
(b) leaves of large inheritance hierarchy	small (~= 0)	large	Small values indicate a lot of impact from the parent classes.

Table 2: Identify large inheritance hierarchies.

2. Classes.

   Classes represent the unit of encapsulation in an object-oriented system, hence it is worthwhile to identify the most important ones. To do this, compile a list of classes with the metric Lines of Code for Class — WNOM (LOC) (p. 49) as main indicator and Number of Methods for Class — NOM (p. 48) plus Number of Attributes for Class — NOA (p. 48) as secondary indicator. Sort the list according to each of the criteria and inspect to top ten of each of them. Also, look for classes where the measurements do not correlate like the other classes in the system, they represent classes with exceptionally high or low values and are probably worthwhile to investigate further (see Table 3).

	WNOM(LOC)	NOM	NOA
(a) large code size	large	Uncorrelated with WNOM(LOC)	
(b) many methods	Uncorrelated with NOM	large	Uncorrelated with NOM
(c) many attributes	Uncorrelated with NOA	Uncorrelated with NOA	large

Table 3: Identify large classes.

3. Methods.

   ...

Hints

Identifying important pieces of functionality in a software system via measurements is a delicate activity which requires expertise in both data collection and interpretation. Below are some hints you might consider to get the best out of your data.

- **Which metrics to collect?** In general, it is better to stick to the simple metrics, as the more complex ones involve more computation, yet will not perform better for the identification of large entities.

  For instance, to identify large methods it is sufficient to count the lines by counting all carriage returns or new-lines. Most other method size metrics require some form of parsing and this effort is usually not worth the gain.
- **Which metric variants to use?** Usually, it does not make a lot of difference which metric variant is chosen, as long as the choice is clearly stated and applied consistently. Here as well, it is preferable to choose the most simple variant, unless you have a good reason

to do otherwise.
For instance, while counting the lines of code, you should decide whether to include or exclude comment lines, or whether you count the lines after the source code has been normalised via pretty printing. However, when looking for the largest structures it usually does not pay off to do the extra effort of excluding comment lines or normalizing the source code.

- **What about coupling metrics?** Part of what makes a piece of code important is how it is used by other parts of the system. Such external usage may be revealed by applying coupling metrics. However, coupling metrics are usually quite complicated, thus go against our principle of choosing simple metrics. Moreover, there is no consensus in the literature on what constitute "good" coupling metrics. Therefore, we suggest not to rely on coupling metrics. If your metrics tool does not include any coupling metrics you can safely ignore them. Otherwise it is better to calculate them after you have identified some large entities.
- **Which thresholds to apply?** Due to the need for reliability, it is better *not* to apply thresholds.[1] First of all, because selecting threshold values must be done based on the coding standards applied in the development team and these you do not necessarily have access to. Second, because "large" is a relative notion and thresholds will distort your perspective of what constitutes "large" within the system as you will not know how many "small" entities there are.

  Note that many metric tools include some visualisation features to help you scan large volumes of measurements and this is usually a better way to quickly focus on important entities.
- **How to interpret the results?** Large is not necessarily the same as important, so care must be taken when interpreting the measurement data. To assess whether an entity is indeed important, it is a good idea to simultaneously inspect different measurements for the same entity. For instance, combine the size of the class with the number of subclasses, because large classes that appear high in a class hierarchy are usually important.

  However, formulas that combine different measurements in a single number should be avoided as you loose the sense for the constituting elements. Therefore it is better to present the results in a table, where the first column shows the name of the entity, and the remaining columns show the different measurement data. Sorting these tables according to the different measurement columns will help you to identify extreme values.
- **Should I browse the code afterwards?** Measurements alone cannot determine whether a entity is truly important: some human assessment is always necessary. However, metrics are a great aid in quickly identifying entities that are potentially important and code browsing is necessary for the actual evaluation. Note that large entities are usually quite complicated, thus understanding the corresponding source code may prove to be difficult.
- **What about small entities?** Small entities may be far more important than the large ones, because good designers tend to distribute important functionality over a number of

1. Most metric tools allow you to focus on special entities by specifying some threshold interval and then only displaying those entities where the measurements fall into that interval.

highly reusable and thus smaller components. Conversely, large entities are quite often irrelevant as truly important code would have been refactored into smaller pieces. Still, different larger entities will share the important smaller entities, thus via the larger entities you are likely to identify some important smaller entities too. Anyway, you should be aware that you are only applying a heuristic: there will be important pieces of code that you will not identify via this pattern.

Example

...

-- Here will follow a "running example" (= same example throughout the whole reverse engineering patterns)

Trade-offs

Pros

- **Scale**. The technique is readily applicable to large scale systems, mainly because the metrics tool typically returns 20% of the entities for further investigation. When different metrics are combined properly (preferably using some form of visualisation) one can deduce quite rapidly which parts of the system represent important chunks of functionality.

Cons

- **Inaccurate**. Quite a lot of the entities will turn out not to be important and this you will only know after you analysed the source code. Moreover, there is a good chance that you will miss important functionality.

Difficulties

- **Interpretation of data**. To really assess the importance of a code entity, you must collect several measurements about it. Interpreting and comparing such multi-valued tuples is quite difficult and requires quite a lot of experience.

Rationale

The main reason why size metrics are often applied during reverse engineering is because they provide a good focus (between 10 to 20/% of the software entities) for a relatively low investment. The results are somewhat unreliable, but this can easily be compensated via code browsing.

Known Uses

In several places in the literature it is mentioned that looking for large object entities helps in program understanding (see among others, [Mayr96a], [Kont97a], [Fior98a], [Fior98b], [Mari98a], [Lewe98a], [Nesi98a]). Unfortunately, none of these incorporated an experiment to count how much important functionality remains undiscovered. As such it is impossible to assess the reliability of size metrics for reverse engineering.

Note that some metric tools visualise information via typical algorithms for statistical data, such as histograms and Kiviat diagrams. Visualisation may help to analyse the collected data. Datrix [Mayr96a], TAC++ [Fior98a], [Fior98b], and Crocodile [Lewe98a] are tools that exhibit such visualisation features.

Related Patterns

Looking at large entities requires little preparation but the results are a bit unreliable. By investing more in the preparation you may improve the reliability of the results. For instance, if you invest in program visualisation techniques you can study more aspects of the system in parallel, thereby increasing the quality of the outcome. Also, you can Recover the Refactorings (p. 35) to focus on those parts of the system that change, thereby increasing the likelihood of identifying interesting entities and focusing on the way entities work together.

What Next

By applying this pattern, you will have identified some entities representing important functionality. Some other patterns may help you to further analyse these entities. For instance, if you ..., you will obtain other perspectives and probably other insights as well. Also, if you Step Through the Execution (p. 41) you will get a better perception of the run-time behaviour. Finally, in the case of a object-oriented code, you can Derive Public Interface (p. 41) to find out how a class is related to other classes.

Even if the results have to be analysed with care, some of the larger entities can be candidates for further reengineering: large methods may be split into smaller ones (see [Fowl99a]), just like big classes may be cases of a *God Class*.

Recover the Refactorings

Reconstruct the iterative design process by comparing subsequent releases and measuring decreases in size, as such recovering refactorings like they have been applied in the past.

Problem

You want to recover what the original developers learned during an iterative development process.

Context

This problem is difficult because:

- The system has been released in several versions, and comparing successive releases is quite cumbersome.
- Even when the changes have been identified, it is difficult to reconstruct the learning process.

Yet, solving this problem is feasible because:

- You have a metrics tool at your disposal, so you can quantify the size of entities in the source-code and compare these.
- You have a source-code browser that allows you to query which methods invoke a given operation (even for polymorphic operations), so you can find out dependencies between classes.
- You have considerable expertise with the implementation language being used, so you can reconstruct refactorings from their effects on source-code.
- You have considerable development expertise, so you can envisage why a certain refactoring has been applied.

Solution

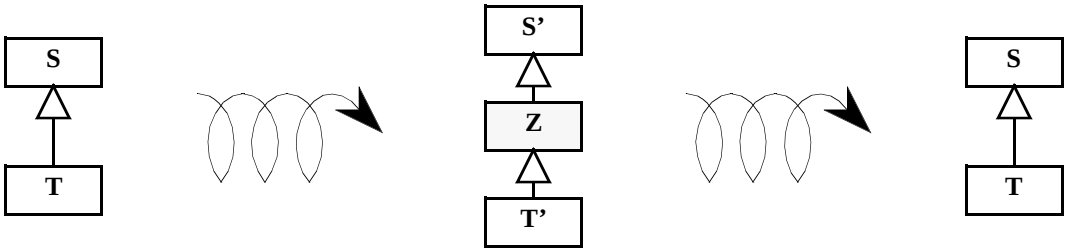
Use the metrics tool to compare the measurements of two subsequent releases and find entities that decrease in size, thus where functionality has been *removed*. Find out whether this functionality has been moved to another location, and as such recover the refactorings that have been applied. For each refactoring, put yourself in the role of the original developer and ask yourself what the change is about and why it was necessary.

Hints

We can recommend three heuristics to help you identifying the following refactorings.

- **Split into superclass / merge with superclass.** Look for the creation or removal of a superclass (change in Hierarchy Nesting Level — HNL (p. 49)), together with a number of

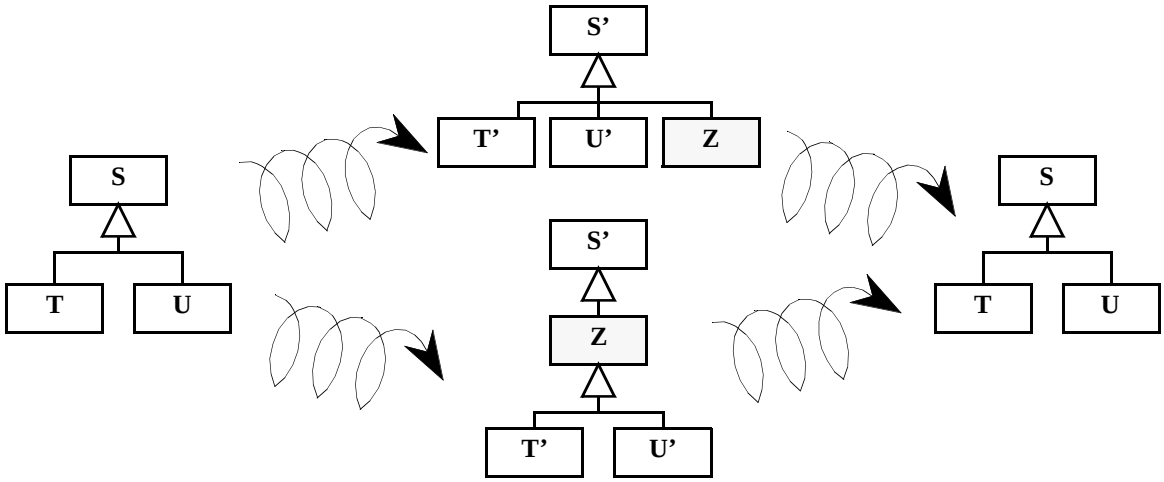
pull-ups or push-downs of methods and attributes (changes in Number of Methods for Class — NOM (p. 48) and Number of Attributes for Class — NOA (p. 48)).



Split T into Z and T'
(delta_HNL(T') > 0) and
((delta_NOM(T') < 0) or (delta_NOA(T') < 0))

Merge Z and T' into T
(delta_HNL(T) < 0) and
((delta_NOM(T) > 0) or (delta_NOA(T) > 0))

- **Split into subclass / merge with subclass.** Look for the creation or removal of a subclass (change in Number of Immediate Subclasses — NIS (p. 50)), together with a number of pull-ups or push-downs of methods and attributes (changes in Number of Methods for Class — NOM (p. 48) and Number of Attributes for Class — NOA (p. 48)).

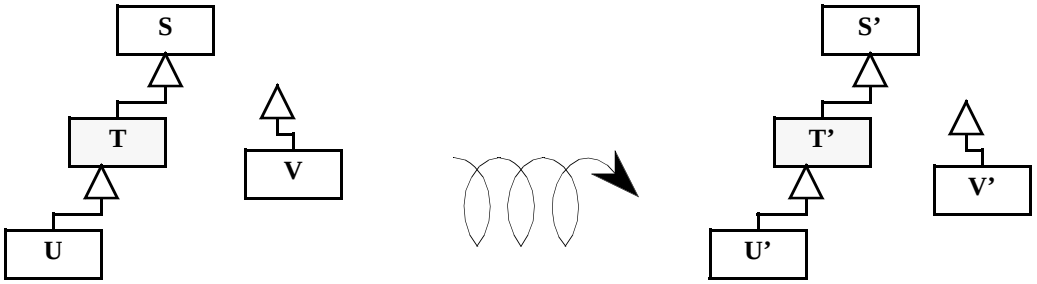


Split S into Z and S'
(delta_NIS(S') <> 0) and
((delta_NOM(S') < 0) or (delta_NOA(S') < 0))

Merge Z and S' into S
(delta_NIS(S) <> 0) and
((delta_NOM(S) > 0) or (delta_NOA(S) > 0))

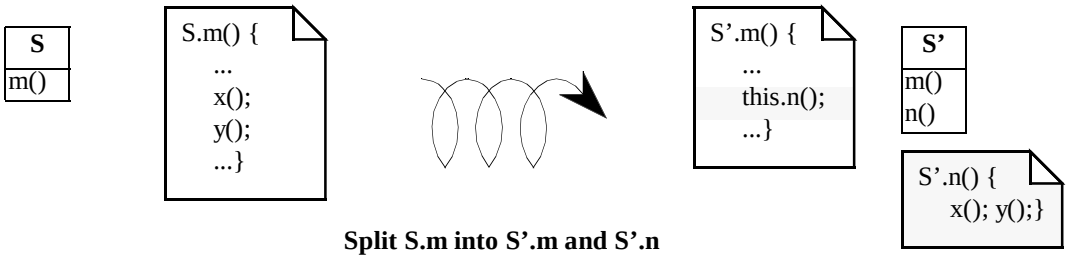
- **Move functionality to superclass, subclass or sibling class.** Look for removal of methods and attributes (decreases in Number of Methods for Class — NOM (p. 48) and

Number of Attributes for Class — NOA (p. 48)) and use code browsing to identify where this functionality is moved to.



Move from T to S, U or V
((delta_NOM(T') < 0) or (delta_NOA(T') < 0))
and (delta_HNL(T') = 0) and (delta_NIS(T') = 0)

- **Split method / factor out common functionality.** Look for decreases in method size (via Lines of Code for Method —LOC (p. 49), or Number of Invocations for Method — NOI (p. 49), or Number of Statements for Method — NOS (p. 49)) and try to identify where that code has been moved to.



Split S.m into S'.m and S'.n
(delta_LOC(S'.m) < threshold)

Trade-offs

Pros

- **Concentrates on relevant parts**, because the changes point you to those places where the design is expanding or consolidating and this in turn provides insight in the underlying design intentions.
- **Provides an unbiased view** of the system, because you do not have to formulate assumptions of what to expect in the software (this is in contrast to Speculate about Domain Objects (p. 20) and Reconstruct the Persistent Data (p. 25))
- **Extracts the interaction protocol**, because finding redistributed functionality involves inspection of method invocations (this is in contrast to Derive Public Interface (p. 41)).

Cons

- **Requires considerable experience**, 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.

- **Considerable tool support** is required, especially (a) a metrics tool that is able to compare different releases or otherwise export its measurements to a separate comparison tool; (b) a code browsers that is able to inspect polymorphic method invocations.

Difficulties

- **Imprecise for many changes**, because when too many changes have been applied on the same piece of code, it becomes difficult to reconstruct the refactorings.
- **Sensitive to renaming** if one identifies classes and methods via their name. Then rename operations will show up as removals and additions which makes interpreting the data more difficult.

Example

...

-- Here will follow a "running example" (= same example throughout the whole reverse engineering patterns)

Rationale

Many object-oriented systems came into being via a combination of iterative and incremental development (see [Booc94a], [Gold95a], [Jaco97a], [Reen96a]). That is, the original development team recognised their lack of problem domain expertise and therefore invested in a learning process where each learning phase resulted in a new system release. It is worthwhile to reconstruct that learning process because it will help us to understand the intentions embodied in the system design.

One way to reconstruct the learning process is to recover its primitive steps. In object-oriented parlance, these steps are called refactorings and consequently this pattern tells you how to recover refactorings like they have been applied in the past. The technique itself compares two subsequent releases of the source code identifying entities that decrease in size, because that's the typical symptom of functionality that has been moved elsewhere.

Known Uses

We ran an experiment on three medium sized systems implemented in Smalltalk. As reported in [Deme00a], these case studies suggest that the heuristics support the reverse engineering process by focusing attention on the relevant parts of a software system.

Related Patterns

Inspecting changes is a costly but very accurate way of identifying areas of interest in a system. If you Identify the Largest (p. 30) you will get less accurate results for a lower amount of resources.

What Next

By applying this pattern, you will have identified some parts in the design that played a key role during the system's evolution. Some other patterns may help you to further analyse these entities. For instance, if you ... you will obtain other perspectives and probably other insights as well. Also, if you Step Through the Execution (p. 41) you will get a better perception of the runtime behaviour. Finally, in the case of a class, you can Derive Public Interface (p. 41) to find out how this class is related to other classes.

Chapter 4

Detailed Model Capture

-- ***REVIEWER NOTE:***
This chapter of the pattern language is currently empty, except fro the pattern names and intents.

The patterns in First Contact (p. 7) should have helped you getting acquainted with the software system, while the ones in Initial Understanding (p. 18) should have provided you with an overall understanding of the system structure. The main priority now is to get detailed knowledge about a particular part of the system.

This cluster tells you *how* you might obtain such detailed knowledge. The patterns involve quite a lot of tools and rely on substantial technical knowledge, hence are applicable in the later stages of a reverse engineering project only. Indeed, only then can you afford to spend the resources obtaining detailed information since only then you have the necessary expertise to know that your investment will pay off.

Derive Public Interface (p. 41) recommends to check invocations of both constructor and overridden methods. Step Through the Execution (p. 41) which explains how to take advantage of your debugger.

Risks

The risks below are sorted according to the importance they have during this phase.

- Techniques and Tools**.** To obtain the required details from a software system you must pay the price in terms of technical expertise and tools. This is the most important risk during this stage of reverse engineering and *consequently, make sure your reverse engineering team possesses the necessary skills and tools*.
- Limited Resources**.** These patterns are applicable during the later stages of a reverse engineering project, thus resources are less scarce as you can be quite sure that your investment will pay off. On the other hand, the activities you apply require more resources. *Consequently, engage in detailed reverse engineering only when you are certain that you need to know the details about that part of a system*. To obtain that knowledge, consider the patterns in the previous clusters.
- Accurate Abstraction**.** All patterns in this cluster have in common that they extract detailed information, at an intermediate level of abstraction (i.e., between source code and design). Yet, detailed knowledge is necessary because in software engineering ---and this is in contrast with many other engineering disciplines--- details are very important [Broo87a]. So, even during fine-grained reverse engineering, there are little details that seem so obvious, yet may obstruct the understanding of the system if you failed to state them.[1] *Consequently, when working on intermediate abstraction levels, make sure you*

provide enough context so that the relationship with both higher and lower levels is clear.

- Reliable Information**.** As details are so important, you should be confident in the obtained results. *Consequently, favour extracting information from the trustworthy information sources.* Fortunately, because you're in the later stages of reverse engineering, you know which information sources are reliable and which ones are not.
- Sceptical Colleagues**.** You would not have arrived this far without the support of some colleagues, so at least you still have the support of the faithful. Moreover, you probably did satisfy the expectations, otherwise the sceptic would have succeeded to cancel your project. And if you did really well, you might even have won some fence sitters over into the camp of the faithful. At this stage, you will not achieve more support from your colleagues. *Consequently, keep on delivering the necessary results to avoid providing reasons for the sceptics to cancel your project.*

Derive Public Interface

Find out how a class is related to other classes by checking the invocations of key methods in the interface of that class. Two examples of key methods that are easy to recognise are constructors and overridden methods.

Step Through the Execution

Obtain a detailed understanding of the run-time behaviour of a piece of code by stepping through its execution.

1. A typical example of such a possibly harmful detail is the use of private/protected in a UML diagram. Depending on the favourite programming language of the author of the diagram, the interpretation is quite different and readers of the diagram should be made aware of this. That is, with a C++ background the interpretation is class based, thus instances of the same class may access each other's private attributes. On the other hand, with a Smalltalk background, the interpretation is instance based, thus it is only the object itself that is allowed to access its attributes. Finally, in Java there is yet another interpretation as a protected attribute may also be accessed by classes in the same package as opposed to subclasses only in C++.

Chapter 5

Prepare Reengineering

-- ***REVIEWER NOTE:***
This chapter of the pattern language is currently empty, except fro the pattern names and intents
+ the pattern "Refactor to understand"

The reverse engineering patterns in this cluster are only applicable when your reverse engineering activities are part of a larger reengineering project. That is, your goal is not only understanding what's inside the source code of a software system, but also rewriting parts of it. Therefore, the patterns in this cluster will take advantage of the fact that you will change the source code anyway.

Write the Tests

--

Record your knowledge about how a component reacts to a given input in a number of black box tests, this way preparing future changes to the system.

Build a Prototype

--

Extract the design of a critical but cryptic component via the construction of a prototype which later may provide the basis for a replacement.

Refactor To Understand

Obtain better understanding of a specific piece of code by iterative refactoring and renaming.

Problem

A particular piece of code seems important but is quite cryptic, hence you cannot fully understand it.

Context

This problem is difficult because:

- Cryptic code is difficult to read, thus to understand.
- Changing cryptic code may cause unexpected side-effects.

Yet, solving this problem is feasible because:

- The piece of code is relatively small and has clearly defined boundaries.
- Your development tools allow for rapid edit-compile cycles, so you can make some small changes and check whether you're still able to compile the source-code.
- You have a source-code browser that allows you to query dependencies between source-code entities (i.e., which methods invoke a given operation, which methods access a given attribute, ...), so that you can infer its purpose.

Symptoms

- Attribute names are reduced to cryptic acronyms.
- Method bodies are long.
- Methods contain comments separating parts of the methods.
- Methods have names that do not reveal their intent.
- Names of the classes are not conveying their purpose.

Solution

Iteratively rename and refactor the code to introduce meaningful names and to make sure the structure of the code reflects what the system is actually doing. Compile often to check whether your changes make sense.

As opposed to normal refactoring it is not required to run regression tests after each change, since your improvements will probably not make it into the actual system.

Steps

The typical refactorings applied during this iterative restructuring are Rename Attributes, Rename Methods, and Extract Methods ([Robe97a], [Fowl99a]). It certain cases the decomposition will not compile and then you may need to use Inline Self Sends on certain methods and subsequently apply Extract Methods.

Following guidelines will help you to find out where and how to apply small scale refactorings to improve the readability of the code. They can be applied in any order, typically in a bottom-up fashion.

- **Remove duplicated code.** If you identify duplicated code, try to refactor it into a single location. As such, you will identify slight differences that you probably would not have noticed before refactoring and that are likely to reveal some subtle design issues.
- **Replace condition branches by methods.** If you encounter big conditions with large branches, extract the leaves as new (private) methods and give them names that are based on the condition until you know more about the system to rename them with an intention revealing name.
- **Method bodies define same level of abstraction.** Long method bodies with comments separating blocks of code violate the rule of the thumb that all statements in a single method body should have the same level of abstraction [Beck97]. Refactor such code by introducing a new (private) method for each separated block of code; name the method after the intent recorded in the comment.
- **Rename attributes to convey roles.** Focus on attributes with cryptic names. To find out about their roles, look at all the attribute accesses (including invocations of accessors methods). Afterwards, rename the attribute and its accessors according to its role, update all references and re-compile the system.
- **Rename methods to convey intent.** Look for method names that do not have a straightforward relationship with the functionality in the piece of code. To retrieve their intent, investigate all invocations and deduce the method's responsibility. Afterwards, rename the method according to its intent, update all invocations and re-compile the system. (see also [Beck97])
- **Rename classes to convey purpose.** Retrieve class names where its not immediately clear how that class might contribute to the overall functionality. To capture their purpose, investigate clients of the class by examining who is invoking its operations. Afterwards, rename the class according to its purpose, update all references and re-compile the system.

Trade-offs

-- ***Reviewer's note.***
These trade-offs should be rewritten in "Pros / Cons / Difficulties/ style.

- **Static vs Iterative Understanding.** As an alternative solution, you could print the code on paper and with some coloured pens try to understand the code. However such an approach is *static*. It is difficult to have several iterations. By applying Refactor To Understand (p. 43) your understanding will grow over the iteration. Every steps will fertilize the next step of understanding.
- **Continuous Validation of Changes.** During your understanding you are elaborating hypotheses about the functionality of the code, you should be able to validate them by checking if the code is running.Moreover, while reading it you may notice some aspects that you would like to rename or refactor. Only printing the code does not support it. By applying Refactor To Understand (p. 43) you will be able to validate your changes. Having

unit tests or regression tests can strenghten your belief in your changes. **@--Remove the reference to unit- or regression testing--@**

- **Knowing What vs How.** You could apply Step Through the Code, however this will provide a view based on a flow of execution whereas what you really want to understand is the logic of the code to be able to integrate new functionality. By applying Refactor To Understand (p. 43) you focus on understanding what does the code.
- **Limiting Impact and Change Integration.** Refactor To Understand (p. 43) can lead you to make a lot of changes in the system that you are trying to understand. You certainly want to limit the impact of your changes. There is different ways to limit the impact:

  You may work on a separate copy of the part that you want to understand and never re-introduce the final result into the system. However, you or other members of your team may lose some really important benefits for future changes. You or other may have to redo the same work in the future.

  You may want to keep the resulting code. In such a case the part of the system on which you are applying Refactor To Understand (p. 43) should be: small (one to a couple of classes), not heavily connected to all the parts of your system or possess an interface that you should keep as a front end between this part and the rest of the system (Check Perdita Pattern).
- **Acceptance of Changes.** Refactoring your own code is always easier than changing code that somebody else wrote for a lot of technical reasons but also because of human communication reasons. Indeed you do not have problem to tell to yourself that your code was not good but this may be different if somebody else would tell that your code was wrong. That's why while applying Refactor To Understand (p. 43) you should always keep in mind that the original developer of the code may have problems to accept your changes. You should consider this dimension when thinking about the integration of your changes into the system.

  Alan Sneed [Sneed at WCRE99] reports that he was refactoring Cobol code and removing in particular goto statements in all the code he was reengineering. However, due to the pressure of the developers he was forced to reintroduced them because they did not accept these changes.
- **Error vs Code Quality Improvement.** The less you change the code, the less chances you have to introduce errors, so the listing approach is safer than renaming and refactoring the code. There are two ways to limit the risk. One way is to have unit tests and run them systematically. The other way is that you can apply the pattern on code that you will not integrate to the system you are working on. This way you can gain an understanding and know how to introduce new functionality while limiting the changes of the system. However, you will lose the possibility to improve the code and reduce its communicability to other possible programmers.
- **When to stop.** It is often difficult to stop changing code when you identify problems in the code. However depending of the time you have for your task you should pay attention not to tend to change code for the sake of its purity. Under severe time constraints a rule is just stop as soon as the new functionality can be introduced.
- **When Not to Apply.** If the code your code looks like spaghetti code and that you cannot identify an already structured piece of code, you may problems to limit the impact of the

changes. Moreover, if you chose not to introduce the resulting code in the application you may have problems to do a clear mapping between the elements of the original code and the refactored code.

Rationale

This pattern is based on the fact that (1) Refactorings help to improve software implementation and design quality [Opdy92b], [Robe97a], [Fowl99a], (2) we understand more easily the code we are writing, and (3) most of the time our understanding does not come in one shot but implies an iterative process where the previous understanding is the base for the next iteration.

Known Uses

John Brant and Don Roberts presented at ESUG'97 and Smalltalk Solution'97 an example of the application of this pattern. They show how they understood an algorithm by renaming and refactoring its code. During the several iterations of the pattern, the code slowly started to get more and more sense and the understanding gradually improved.

This pattern has been applied on a FAMOOS case study. We have to understand a huge method of 3000 lines of C++. We extracted all the conditional branch leaves as methods that we named them depending of the condition. Then we iterated and discovered that this huge method was in fact a complete parser for a command language.

A well defined part of the Moose application, its model extractor, needed to be extended to take into account namespaces. However, the main functionality was only composed by a couple of big methods containing a lot of duplication.This pattern has been applied on the particular class which big public interface methods containing a lot of copy and paste functionality where recomposed into public interfaces methods calling elementary functionality.

Related Patterns

To help to understand the functionality you may apply Step Through the Execution (p. 41). To keep your questions and annotations you can apply @Tie Code and Questions.@

If you do consider to replace the cryptic code with the improved version, make sure you have regression tests, for instance via Write the Tests (p. 42).

What Next

The main result is that you gain an intimate understanding of the part of a system that you refactored. The second result is that you may have a better designed piece of code with intention revealing name. However in the decision to integrate the resulting code into the legacy applications you should take into account that if you do not have regression tests you may introduce unexpected bugs.

Chapter 6

Miscellaneous

-- ***REVIEWER NOTE:***
This chapter of the pattern language is currently empty, except fro the pattern names and intents.

It is yet unknown how we will deal with what's here: currently its one pattern of our own ("Confer with colleagues") and thumbnails of patterns in other catalogues.

Confer with Colleagues

Share the information obtained during each reverse engineering activity to boost the collective understanding about the software system.

God Class

--

... (see [Brow98a])

Chapter 7

List of Metrics

-- ***REVIEWER NOTE:***
This chapter is currently just a condensed list of metrics that are useful during reverse- and reengineering.
The idea is that each metric will be described in approximately 1 page, including info on how to extract, references to the literature, "also known as", ... [Lore00x] is more or less what we have in mind,
Things that might change:
- currently we are inventing our own acronyms for some well-known metrics. We might go back to the more popular names, but then we will have problems with (a) UML terminology; (b) some acronyms are overloaded.

5. Class Size Metrics

Number of Methods for Class — NOM

Count the number of methods in a class.

Variants

- Include or not include private, protected, public
- Include or not the methods defined on class level instead of object level (i.e. static methods in C++, Jave; class methods in Smalltalk)
- Include or not the constructors

Number of Attributes for Class — NOA

Count the number of methods in a class.

Variants

- Include or not include private, protected, public

Lines of Code for Class — WNOM (LOC)

Count the lines of code for the complete class definition.

The abbreviation WNOM (LOC) stems from "Weighted Number of Methods, summing Lines of Code per Method".

Variants

- Before or after formatting
- Including or exclusion comment-lines
- Including the class definition itself, or just the sum of all lines of code per method (as suggested by the abbreviation)

6. Method Size Metrics

Number of Invocations for Method — NOI

Count the number of methods invoked in a method body.

Variants

- Include or exclude special invocations, such as operators, procedure calls

Lines of Code for Method —LOC

Count the lines of code in the method body.

Variants

- Before or after formatting
- Including or exclusion comment-lines

Number of Statements for Method — NOS

Count the number of statements in the method body.

Variants

- Before or after formatting
- Including or exclusion comment-lines

7. Inheritance Metrics

Hierarchy Nesting Level — HNL

Number of superclasses in the longest superclass chain.

Variants

- Include or exclude default roots (i.e., Object in Smalltalk, ...)

Number of Immediate Subclasses — NIS

Number of immediate subclasses.

Variants

- Include or exclude private/protected subclasses

Number of Descendant Classes — NDC

Number of descendant classes, thus total number of all subclasses for a class.

Variants

- Include or exclude private/protected subclasses

Chapter 8

References

[Beck97]

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

[Booc94a] Grady Booch, Object Oriented Analysis and Design with Applications (2nd edition), The Benjamin Cummings Publishing Co. Inc., 1994.

[Brow96c] Kyle Brown, "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk," Ph.D. thesis, North Carolina State University, 1996.

[Brow98a] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III and Thomas J. Mowbray, "AntiPatterns," 1998.

[Busc96a] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stad, Pattern-Oriented Software Architecture — A System of Patterns, John Wiley, 1996.

[Cold99a] Jens Coldewey, Wolfgang Keller and Klaus Renzel, Architectural Patterns for Business Information Systems, Publisher Unknown, 1999, To Appear.

[Deme00a] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, "Finding Refactorings via Change Metrics," OOPSLA'2000 Proceedings, to appear

[Fior98a]

[Fior98b]

[Fowl97b] Martin Fowler, Analysis Patterns: Reusable Objects Models, Addison-Wesley, 1997.

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

[Fros94a] Stuart Frost, "Modeling for the RDBMS legacy", Object Magazine, September 1994, pp.43-51.

[Gamm95a] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns, Addison Wesley, Reading, MA, 1995.

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

[Hain96a] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick and D. Roland, "Database reverse Engineering: From requirements to CARE Tools," Automated Software Engineering, vol. 3, no. 1-2, June 1996.

[Jaco92a] Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard, Object-Oriented Software Engineering — A Use Case Driven Approach, Addison-Wesley/ACM Press, Reading, Mass., 1992.

[Jaco97a] Ivar Jacobson, Martin Griss and Patrik Jonsson, Software Reuse, Addison-Wesley/ACM Press, 1997.

[Jahn97b] Jens. H. Jahnke, Wilhelm. Schäfer and Albert. Zündorf, "Generic Fuzzy Reasoning Nets as a Basis ofr Reverse Engineering Relational Database Applications," Proceedings of ESEC/FSE'97, LNCS, no. 1301, 1997, pp. 193-210.

[Kell98a] Wolfgang Keller and Jens Coldewey, “ Accessing Relational Databases: A Pattern Language,” Pattern Languages of Program Design 3, Robert Martin, Dirk Riehle and Frank Bushmann (Eds.), pp. 313-343, Addison-Wesley, 1998.

[Kont97a]

[Lea96a] Doug Lea, Concurrent Programming in Java, Design Principles and Patterns, Addison-Wesley, The Java Series, 1996.

[Lewe98a]

[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 (Eds.), LNCS 1543, Springer-Verlag, 1998, pp. 252-253.

[Mayr96a]

[Murp97a] Gail Murphy and David Notkin, “Reengineering with Reflexion Models: A Case Study,” IEEE Computer, vol. 8, 1997, pp. 29-36.

[Nesi98a]

[Prem94a] William J. Premerlani and Michael R. Blaha, “An Approach for Reverse Engineering of Relational Databases,” Communications of the ACM, vol. 37, no. 5, May 1994, pp. 42-49.

[Reen96a] Trygve Reenskaug, Working with Objects: The OOram Software Engineering Method, Manning Publications, 1996.

[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, no. 4, 1997, pp. 253-263.

[Wirf90b] Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener, Designing Object-Oriented Software, Prentice Hall, 1990.