A Pattern Language for Reverse Engineering

v0.4-- March 3, 2000 4:30 pm

http://win-www.uia.ac.be/u/sdemey/Pubs/Deme00n/

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. Due to limitations on EuroPLOP submissions, only part of the full pattern language is presented, namely the patterns describing how to gain an initial understanding of a software system and one pattern preparing subsequent reengineering.

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

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 which goal is to produce a set of re-engineering 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.

We welcome any feedback that would help us do that. We are especially interested in coarse grained comments ---does the structure work? is the set of risks complete? is the naming OK?--- rather than detailed comments on punctuation, spelling and layout.

Acknowledgments. We would like to thank our EuroPLoP'99 shepherd Kyle Brown: his comments were so good we considered including him as a co-author. We also want to thank both Kent Beck and Charles Weir who shepherded a very rough draft of what you hold right now. 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 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. This cluster groups patterns telling you what to do when you have your very first contact with a software system.

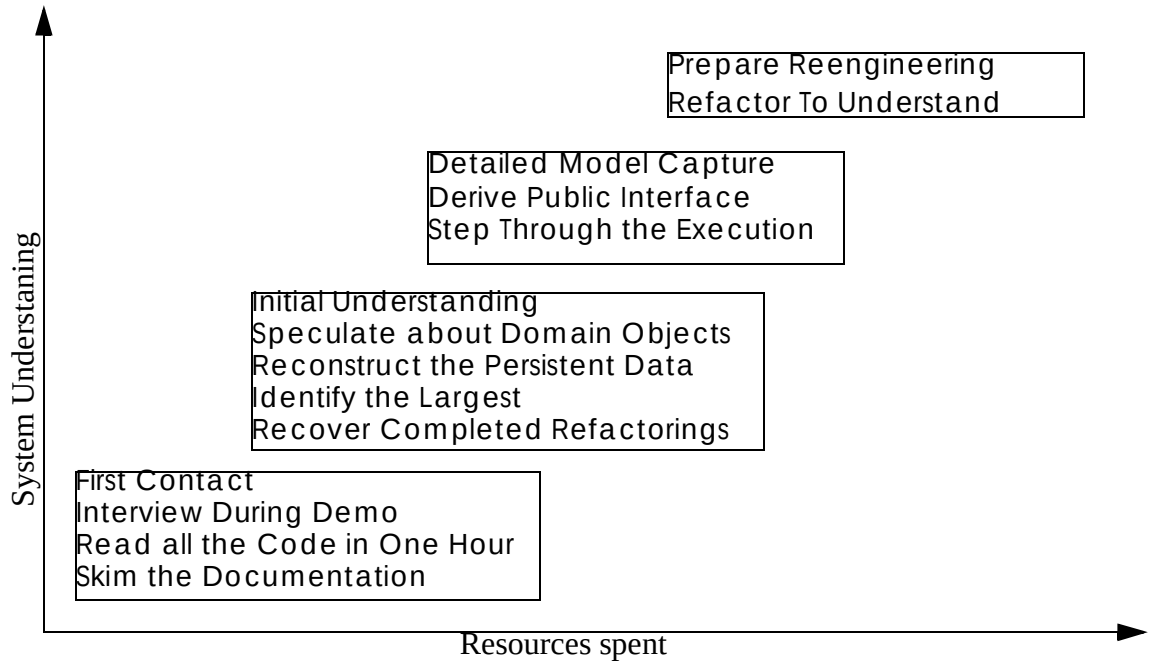

Figure 1 Overview of the pattern language using clusters.

- Initial Understanding**.** Here, the patterns tell you how to obtain an initial understanding of a software system, often documented in the form of class models.
- Detailed Model Capture**.** The patterns in this cluster describe how to get a detailed understanding of a particular component in your software system.
- Prepare Reengineering**.** 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.** 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.*

	Limited Resources	Techniques and Tools	Reliable Information	Abstraction	Sceptical Colleagues
First Contact					
Read all the Code in One Hour	++	++	+	-	+
Skim the Documentation	++	++	-	+	-
Interview During Demo	++	+	/	+	-
Initial Understanding					
Speculate about Domain Objects	-	-	++	++	+
Reconstruct the Persistent Data	-	-	++	+	++
Identify the Largest					
Recover Completed Refactorings					
Detailed Model Capture					
Derive Public Interface					
Step Through the Execution					
Miscellaneous					
Confer with Colleagues					

Table 1: How each pattern reduces the risk.
Very good: ++, Good: +, Neutral: /, Rather Bad: -, Very bad: --.
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.
Abstraction: The more abstract the information obtained, the better.
Sceptical Colleagues: The more credibility you gain, the better.

- **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*.
- **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, thus your challenge is to assess which information is reliable and solve the mystery by coming up with the most plausible scenario. In general, *the more reliable the information you get, the smaller the risk*.
- **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. For in-

stance, the reengineering taxonomy of Chikofsky and Cross [Chik90a], defines reverse engineering as "the process of analyzing a subject system to [...] create representations of the system [...] at a higher level of abstraction". Once you are able to strip away portions of a system as being irrelevant for your future needs, you have acquired the necessary understanding to proceed. Moreover, the more details you can confidently strip away, the higher your understanding and thus the Therefore, *the more abstract the information obtained, the smaller the risk.*

- **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.*

Table 1 shows an overview of how the different patterns reduce the risks. This view is especially important because it emphasises the different trade-offs implied by the patterns. For instance, it shows that Read all the Code in One Hour and Skim the Documentation 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 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.
- **Tradeoffs.** 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

-- **Due to limitations on EuroPLOP submissions, only part of the full pattern language is presented. Therefore, only the intent sections of the patterns in this cluster are preserved. We refer the reader the our EuroPLOP99 submission for the full version of these patterns.**

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.

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.

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.

Chapter 3

Initial Understanding

The patterns in First Contact 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), how to extract a class model from a database (Reconstruct the Persistent Data), how to identify important chunks of functionality (Identify the Largest), how to recognize which refactorings have been applied in the past (Recover Completed Refactorings). With this information you will probably want to proceed with Detailed Model Capture.

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 *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.*
- Abstraction**.** Understanding means building mental models and models are meant to strip away details. Yet, details are crucial to the overall system [Broo87a]. *Consequently, favor 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 de-

tails, 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

Intent

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

You are in the early stages of reverse engineering a software system: you have a rough understanding of its functionality and you are somewhat familiar with the main structure of its source code. You have on-line access to the source code of the software system and the necessary tools to browse it (i.e., from an elementary `grep` to a full-fledged code browser). You have reasonable expertise with the implementation language(s) being used.

Solution

Use your expertise to develop 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 untill 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).
2. Enumerate the names in the class model (that is, names of classes, attributes and operations) and try to find them in the the source code, using whatever tools you have available. Take care as names inside the source-code do not always match with the concepts 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 synomyms 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 may serve to define some use cases that in turn help to find out which objects fulfill which roles. (See [Jaco92a] for use cases and [Reen96a] for role modeling.)
- 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.)

Tradeoffs

- **Impact.** 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.
- **Requires Implementation Expertise.** In itself, the pattern does not require a lot of reverse engineering expertise. However, a large repertoire of knowledge about idioms, patterns, algorithms, techniques is necessary to recognize what you see. As such, the pattern should preferably be applied by experts in the implementation language.

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.

Example

Rationale

If you Speculate about Domain Objects, 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.

Moreover, the pattern scales up. This is especially important as 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, such an approach does not bring a lot of understanding, because you are forced to focus on the irrelevant noise instead of the important concepts.

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.

Related Patterns

All the patterns in the First Contact cluster are meant to help you in building the initial hypothesis now to be refined via Speculate about Domain Objects. Afterwards, some of the patterns in Detailed Model Capture (in particular, Step Through the Execution) 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 when you want to learn about the valuable data of an application, or **@references to remaining patterns here@.**

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

Speculate about Patterns

Like Speculate about Domain Objects*, except that you build and refine a hypothesis about occurances of architectural, analysis or design patterns.*

While having Read all the Code in One Hour, 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 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, 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, *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 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

Intent

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

Problem

You do not know which objects are valuable for the functioning of the system, i.e. that are so crucial that they require special care in terms of back-up procedures, concurrency control.

Context

You are in the early stages of reverse engineering a software system, having a rough understanding of its functionality. The software system employs some form of a database to make its data persistent.

You have access to the database and the proper tools to inspect its schema and obtain samples of data. Besides, you have some expertise with databases and knowledge of how data-structures from your implementation language are mapped onto the data-structures of the underlying database.

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.
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 is the "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

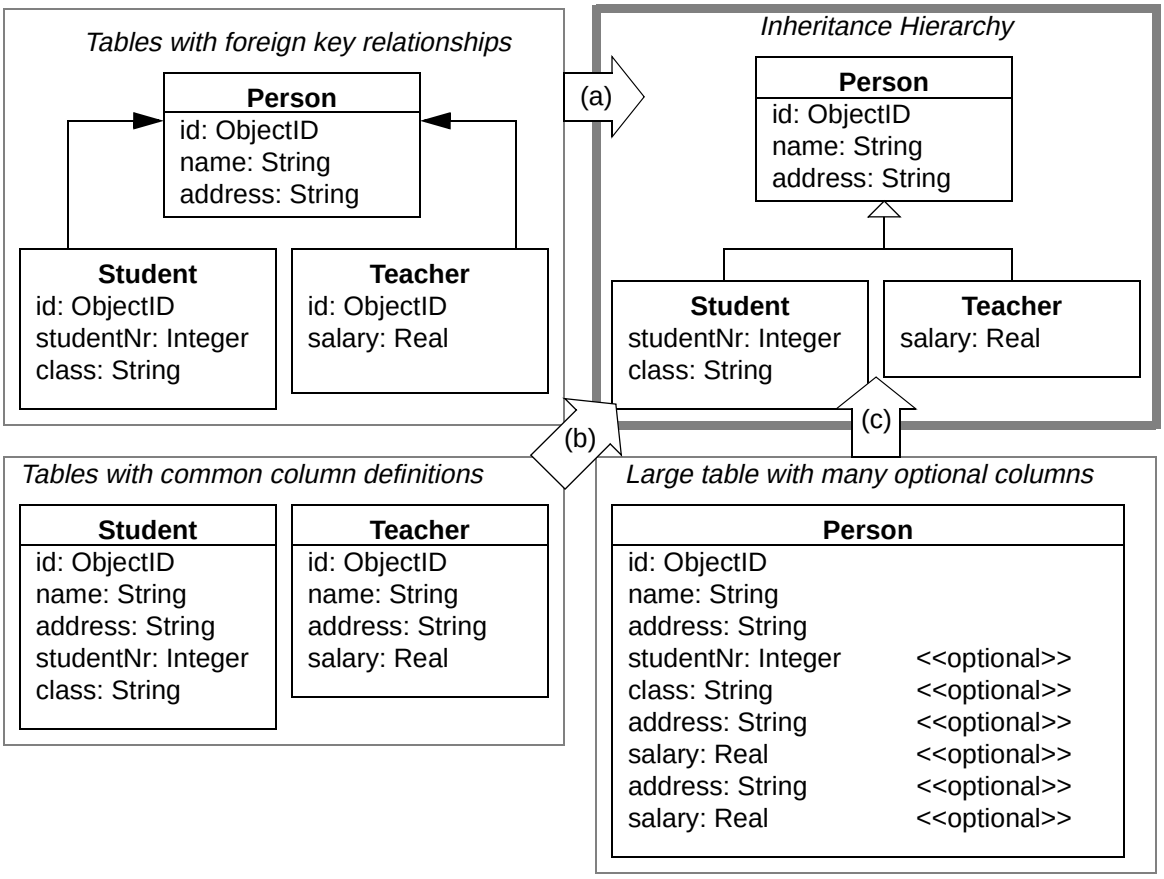

Figure 2 Mapping a series of relational tables onto an inheritance hierarchy.
(a) one to one; (b) rolled down; (c) rolled up

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**

Tradeoffs

- **Impact.** 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.

--

- **Requires Database Expertise.** In itself, the pattern does not require a lot of reverse engineering expertise. However, a good deal of knowledge about he underlying database plus constructs to map the database schema into the implementation language is necessary to recognize what you see. As such, the pattern should preferably be applied by people having expertise in mappings from the chosen database to the implementation language.
- **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.
- **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 optimize 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

You are asked to extend an existing database application so that it will be accessible via the world-wide web. The initial software system manipulates the business objects (implemented in C++) stored inside a relational database. You will reconstruct the data model underlying your business objects by mapping the table definitions in the database on the corresponding C++ classes.

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 requires an initial understanding of the system functionality, as obtained by applying patterns in the cluster First Contact.

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

What Next

Reconstruct the Persistent Data 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. 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 and Prepare Reengineering. Consequently, consider to Confer with Colleagues after Reconstruct the Persistent Data.

Identify the Largest

Intent

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

Problem

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

Context

You are in the early stages of reverse engineering an object-oriented software system, having a rough understanding of its functionality. You have a metrics tool and a code browser at your disposal.

Solution

Use the metrics tool to collect a limited set of measurements concerning the constructs 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 construct. Browse the source code for the large or exceptional constructs to determine whether the construct represents important functionality.

Steps

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

1. The inheritance hierarchy.

   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 metric Total Number of Subclasses as the main indicator, and Number of Methods for Class plus Number of Attributes for Class as secondary indicators.

   Look for large inheritance hierarchies that are very deep and shallow, are very broad and flat, and where the classes have a well distributed size. These ususally correspond with well-structured part of the inheritance hierarchy and as such are likely to be important.
2. Packages.

   ...
3. Classes.

   Look for large classes that appear at the top or at the bottom of the hierarchy, or that stand on themselves.
4. Methods.

   ...

Example

You are facing an object-oriented system and you want to find out which classes do the bulk of the work. You will produce a list of all classes where the number of methods exceeds the average number of methods per class, sort the list and inspect the largest classes manually.

Tradeoffs

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 trade offs that you should 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 constructs.

  For instance, to identify large methods it is sufficient to count the lines by counting all carriage returns or newlines. 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 metreics. 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 constructs.
- **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" constructs 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 constructs.
- **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 a construct is indeed important, it is a good idea to simultaneously inspect different measurements for the same construct. 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 construct, 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 construct is truly important: some human assessment is always necessary. However, metrics are a great aid in quickly identifying constructs that are potentially important and code browsing is necessary for the actual evaluation. Note that large constructs are usu-

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

ally quite complicated, thus understanding the corresponding source code may prove to be difficult.

- **What about small constructs?** Small constructs may be far more important than the large ones, because good designers tend to distribute important functionality over a number of highly reusable and thus smaller components. Conversely, large constructs are quite often irrelevant as truly important code would have been refactored into smaller pieces. Still, different larger constructs will share the important smaller constructs, thus via the larger constructs you are likely to identify some important smaller constructs 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.

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 constructs) 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 constructs 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 constructs 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 Completed Refactorings to focus on those parts of the system that change, thereby increasing the likelihood of identifying interesting constructs and focussing on the way constructs work together.

What Next

By applying this pattern, you will have identified some constructs representing important functionality. Some other patterns may help you to further analyse these constructs. For instance, if you ..., you will obtain other perspectives and probably other insights as well. Also, if you Step Through the Execution 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 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 constructs 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 Completed Refactorings

Intent

Recover refactorings that have been applied by identifying where functionality has been removed and finding out if it has been moved to another location.

Problem

You do not know how ---and to a certain extent why--- the design of a system has evolved.

Context

You are in a later stage of reverse engineering an evolving software system. You have an overall understanding of its functionality and you know the main structure of its source code. You have several releases of the source code at your disposal plus a a metrics tool to detect the differences between the releases.

Solution

Gather a number of size metrics for two subsequent releases and find constructs that decrease in size, thus where functionality has been removed. Find out where this functionality has been moved to 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 that help 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), together with a number of pull-ups or push-downs of methods and attributes (changes in number of methods - NOM and number of attributes - NOA).
- **Split into subclass / merge with subclass.** Look for the creation or removal of a subclass (change in number of immediate subclasses- NOC), together with a number of pull-ups or push-downs of methods and attributes (changes in number of methods - NOM and number of attributes - NOA).

- **Move functionality to superclass, subclass or sibling class.** Look for removal of methods and attributes (decreases in number of methods - NOM and number of attributes - NOA) and use code browsing to identify where this functionality is moved to.
- **Split method / factor out common functionality.** Look for decreases in method size (via lines of code - LOC, or number of message sends - MSG, or number of statements - NOS) and try to identify where that code has been moved to.

Example

You must understand an object-oriented framework that has been adapted several times as the developers gained insight into the problem domain. You will filter out all classes where the number of methods and attributes has decreased significantly and find out where that functionality has been moved to. With that knowledge, you will make a guess at the design rationale underlying this redistribution of functionality.

Rationale

Because you focus on constructs that decrease in size, you are likely to identify places where functionality has been moved to other locations. Such moving of functionality is always relevant for reverse engineering, as it reveals design intentions from the original developers.

Satisfying the prerequisite of having different releases of the source code plus the necessary tools to assess the differences, the main advantages of looking at changes are the following. (i) It concentrates on relevant parts, because the changes point you to those places where the design is expanding or consolidating. (ii) It 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 and ...) (iii) It gives an insight in the way components interact, because the changes reveal how functionality is redistributed among constructs (this is in contrast to Derive Public Interface).

Known Uses

We ran an experiment on three medium sized systems implemented in Smalltalk. As reported in [Deme99z], these case studies suggest that the heuristics support the reverse engineering process by focussing 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 ... or Derive Public Interface 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 constructs. For instance, if you ... you will obtain other perspectives and probably other insights as well. Also, if you Step Through the Execution you will get a better perception of the run-time

behaviour. Finally, in the case of a class, you can Derive Public Interface to find out how this class is related to other classes.

Chapter 4

Detailed Model Capture

-- **Due to limitations on EuroPLOP submissions, only part of the full pattern language is presented. Therefore, only the intent sections of the patterns in this cluster are preserved. The full versions of these patterns will appear later.**

Derive Public Interface

Intent

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

Intent

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

Chapter 5

Prepare Reengineering

-- **Due to limitations on EuroPLOP submissions, only part of the full pattern language is presented. Therefore, only one of the patterns is fully expanded and for the remaining ones only the intent sections are preserved. The full versions of these patterns will appear later.**

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

Intent

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

Intent

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

Wrap the Unimportant

Intent

Wrap the parts you consider unnecessary for the future reengineering in a black box component.

Refactor To Understand

Intent

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

Problem

The code is so cryptic and its layout is such that it is difficult to read ---hence to understand--- and difficult to know how to add new functionality.

Context

You need to be able to compile the code or at least make a copy of the source. You should also be able to adapt client code. The code you need to understand is well defined and small (at the maximum a couple of classes).

Symptoms

- Methods are long.
- Methods contain comments separating parts of the methods.
- Methods have cryptic and not intention revealing names [Beck97].
- Names of the classes are not conveying their responsibilities or roles in the application.
- Attribute names are reduced to cryptic symbols or are not meaningful.

Solution

Iteratively rename and refactor the code to introduce meaningful names and a better structure that reflects what the system is actually doing.

Hints

- **Bottom up.** Start by focusing on the attributes with unclear names and give them names that convey their roles. Start by looking at all the attribute accesses or the accessors calls and infer a type and/or the purpose of the attributes. Indeed, methods are abstractions that are built on top of the object intrinsic structure represented by the object attributes. So understanding and giving meaningful names to the attributes helps to gain an understanding of the method purpose.
- **Top Down.** If you have an external view of the class functionality via the analysis of tests, you can start to give meaningful names to the public interface of the class. If unit tests exist read them because they represent some uses of the code you want to understand. The methods used in unit tests are certainly part of the public interface of the object. As this may help you to understand the inputs and the outputs of the methods, you will understand the intention of the methods.
- **Leaf of Condition Branches.** If you encounter big conditions with large branches and leaves, extract the leaves as new methods and give them names that are based on the condition until you know more about the system and you rename them.

- **Same Level of Abstraction.** As methods represent the elementary unit of reuse in object-oriented languagues, try to refactor the code so that methods represent meaningful pieces of abstractions with one clear responsibility.A good rule of the thumb is to have the same level of abstraction in method body [Beck97].
- **Remove Duplicated Code.** If you identify duplicated code, try to refactor it first. Indeed, by focusing on the same piece of code, you will be able to identify slight differences that you probably would not have been able to notice when the code was duplicated. The questions that may rise concerning these differences may reveal some subtle design points.
- **Refactorings Applied.** Most of the time you will Rename Attributes, Rename Methods, and Extract Methods ([RobeThesis, RefactoringPaper,Fowler99]). Code to be extracted is often separated by comments indicating in the method body the different functionalities.It may also be the case that the decomposition of the class in methods is not suitable for supporting your refactorings. In such a case you may need to use Inline Self Sends on certain methods and subsequently apply Extract Methods.

Tradeoffs

- **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 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 you will be able to validate your changes. Having unit tests or regression tests can strenghten your belief in your changes [XPBook].
- **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 you focus on understanding what does the code.
- **Limiting Impact and Change Integration.** Refactor To Understand 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 reintroduce 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 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 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 [Opdy92, Robe98, Fow99], (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 patter, the code slowly started to get more and more sense and the understanding went growing.

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 system 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 Code. To keep your questions and annotations you can apply Tie Code and Questions.

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

Confer with Colleagues

Intent

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

5. Class Size Metrics

Number of Methods for Class

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

Count the number of methods in a class

Variants

- Include or not include private, protected, public

Lines of Code for Class

Count the lines of code for the complete class definition

Variants

- Before or after formatting
- Including or exclusing comment-lines
- Including the class definition itself, or just the sum of all lines of code per method

6. Method Size Metrics

Number of Invocations

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

Count the lines of code in the method body of a class

Variants

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

7. Inheritance Metrics

Depth of Inheritance Tree

Number of superclasses in the longest superclass chain

Variants

- Including or exclude defaulr roots (i.e., Object in Smalltalk, ...)

Immediate Number of Subclasses

Number of immediate subclasses

Variants

- Include or exclude private/protected subclasses

Total Number of Subclasses

Total number of subclasses for a class

Variants

- Include or exclude private/protected subclasses