Part II

Reverse Engineering

Chapter 3

Reverse Engineering Patterns

3.1 Patterns for Reverse Engineering

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 truly reusable object-oriented design is recognized to be iterative development (see [BOOC 94], [GOLD 95], [JACO 97], [REEN 96]). 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 whose 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.

In its current state, the pattern language can still be improved and we welcome all kinds of feedback that would help us do that. We are especially interested in course grained comments —does the structure work? is the set of forces complete? is the naming OK ?— rather than detailed comments on punctuation, spelling and lay-out.

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. Finally, we must thank all participants of the FAMOOS project for providing such fruitful working context.

3.2 Clusters of Patterns

The pattern language itself 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. Figure 3.1 provides a road map and below you will find a short description for each of the clusters.

First Contact (p. 109) This cluster groups patterns telling you what to do when you have your very first contact with the software system.

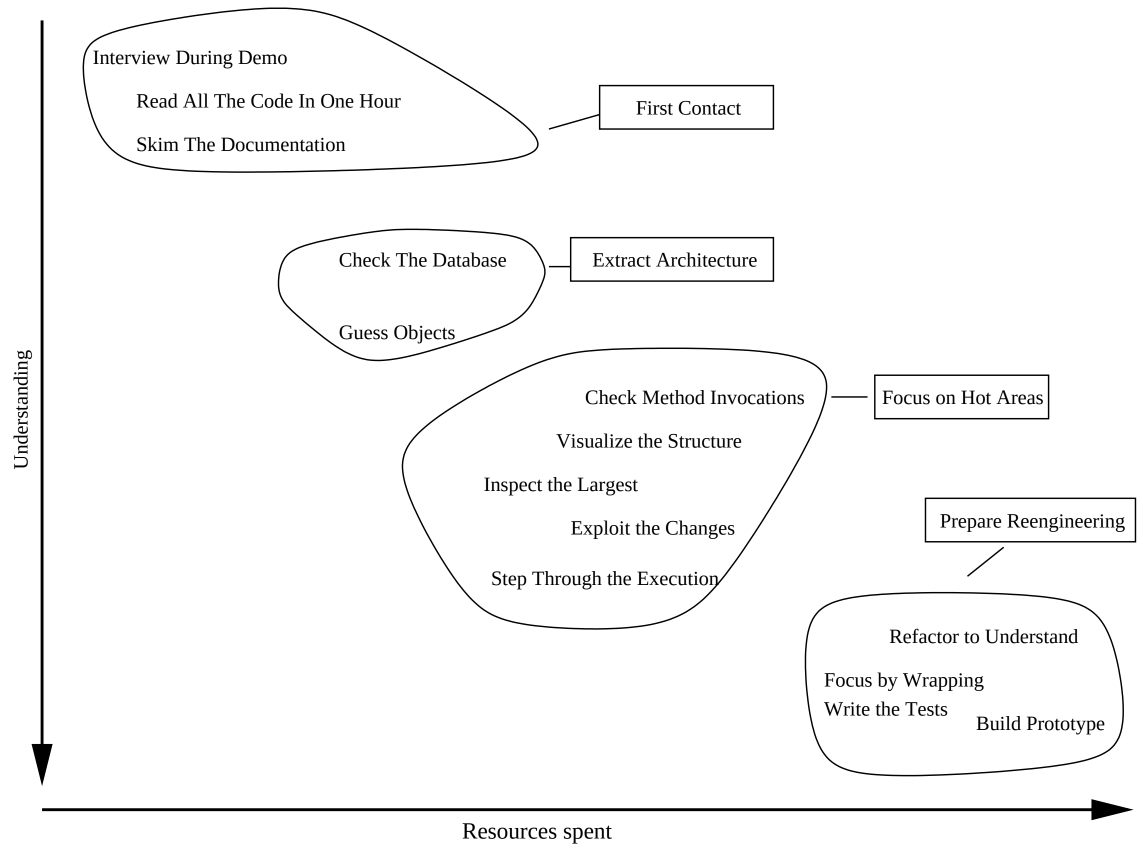

Figure 3.1: Overview of the pattern language using clusters.

Extract Architecture (p. 121) Here, the patterns tell you how to get to the architecture out of a system. This knowledge will serve as a blueprint for the rest of the reverse engineering project.

Focus on Hot Areas (p. 129) The patterns in this cluster describe how to get a detailed understanding of a particular component in your software system.

Prepare Reengineering (p. 145) Since reverse engineering often goes together with reengineering, this cluster includes some patterns that help you prepare subsequent reengineering steps.

3.3 Overview of Forces

All the patterns in this pattern language tell you how to address a typical reverse engineering problem. To evaluate the situation before and after applying the pattern we introduce a number of *forces*. The forces are meant to help you assessing whether the pattern is appropriate for your particular situation.

Limited Resources. Because your resources are limited you must select which parts of the system to reverse engineer first. However, if you select the wrong parts, you will have wasted some of your precious resources. In general the *less resources you need to apply, the better*.

Tools and Techniques. 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 better.*

Reliable Info. A reverse engineer is much like a detective that solves a mystery from the scarce clues that are available [WILL 96b]. 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 better*.

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. Consequently, the reengineering taxonomy of Chikofsky and Cross [CHIK 90], defines reverse engineering as ”the process of analyzing a subject system to [...] create representations of the system [...] at a higher level of abstraction”. Of course, the target level of abstraction for your particular reverse engineering step depends very much on the subsequent demands and so you don’t want to get too abstract. Still in general, the *more abstract the information obtained, the better*.

Sceptic 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 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 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 better*.

3.4 Resolution of Forces

Table 3.1 shows an overview of how the different patterns resolve the forces. 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 double plusses), yet score differently on the reliability and abstraction level of the resulting information. On the other hand, GUESS OBJECTS requires more resources, techniques and tools then the previous two (one minus), but achieves better results in terms of reliable and abstract information.

3.5 Format of a Reverse Engineering Pattern

The pattern presented hereafter have 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. Summarizes the purpose of the pattern, including a clarifying *Example* on when and how to apply the pattern.

Context. Presents the context in which the pattern is supposed to be applied. You may read this section as the prerequisites that should be satisfied before applying the pattern.

Problem. Describes the problem the pattern is supposed to solve. Note that the prerequisites defined in the ’Context’ section are supposed to narrow the scope of the problem, so readers are encouraged to read both sections together.

Solution. Proposes a solution to the problem that is applicable in the given context. This section may include a *Recipe* or a list of *Hints* and *Variations* to be taken in account when applying the solution.

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
FIRST CONTACT					
READ ALL THE CODE IN ONE HOUR	++	++	+	-	+
SKIM THE DOCUMENTATION	++	++	-	+	-
INTERVIEW DURING DEMO	++	+	0	+	-
EXTRACT ARCHITECTURE					
GUESS OBJECTS	-	-	++	++	+
CHECK THE DATABASE	-	-	++	+	++
FOCUS ON HOT AREAS					
INSPECT THE LARGEST	0	-	0	0	0
VISUALIZE THE STRUCTURE	-	- -	+	+	+
CHECK METHOD INVOCATIONS	-	-	+	+	0
EXPLOIT THE CHANGES	- -	- -	++	+	++
STEP THROUGH THE EXECUTION	-	0	++	-	+
PREPARE REENGINEERING					
WRITE THE TESTS	- -	-	++	++	0
REFACTOR TO UNDERSTAND	- -	-	0	+	0
BUILD A PROTOTYPE	- -	-	+	- -	++
FOCUS BY WRAPPING	- -	0	0	0	0

Table 3.1: How each pattern resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -. Limited Resources: The less resources you need to apply, the better. Tools and Techniques: The less techniques and tools required, the better. Reliable Info: The more reliable the information you get, the better. Abstraction: The more abstract the information obtained, the better. Sceptic Colleagues: The more credibility you gain, the better.

Forces Resolved. Describes the situation after applying the pattern. This description is done in terms of the forces.

Rationale. Explains the technical background of the pattern, i.e. why it works.

Known Uses. Presents the know uses of this pattern. Note that all patterns in this pattern language have been developed and applied in the context of the FAMOOS project. Yet, this section 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 *Resulting Context* which tells you how you may use the output of this pattern as input for another one.

Chapter 4

Cluster: First Contact

All the reverse engineering patterns in this cluster are applicable in the very early stage of a reverse engineering project when you are largely unfamiliar with the software system. Before tackling such a project, you need an initial assessment of the software system. However, accomplishing a good initial assessment is difficult because you need a quick and accurate result.

The patterns in this cluster tell you how to optimally exploit information resources like source code (READ ALL THE CODE IN ONE HOUR (p. 111)), documentation (SKIM THE DOCUMENTATION (p. 114)) and system experts (INTERVIEW DURING DEMO (p. 117)). The order in which you apply them depends mainly on your project and we refer you to the *"Related Patterns"* section in each pattern for a discussion on the trade-offs involved. Afterwards you will probably want to CONFER WITH COLLEAGUES (p. 152) and then proceed with EXTRACT ARCHITECTURE (p. 121).

Forces Revisited

Limited Resources. Wasting time early on in the project has severe consequences later on. *Consequently, time is the most precious resource in the beginning of a reverse engineering project.* 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.

Tools and Techniques. In the beginning of a reverse engineering project, you are in a bootstrapping situation: you must decide which techniques and tools to apply but you lack a profound basis to make such a decision. *Consequently, choose very simple techniques and very basic tools, deferring complex but time consuming activities until later.*

Reliable Info. Because you are unfamiliar with the system, it is difficult to assess which information is reliable. *Consequently, base your opinion on certified information but complement it using supplementary but less reliable information sources.*

Abstraction. At the beginning of the project you can not afford to be overwhelmed by too many details. *Consequently, favor techniques and tools that provide you with a general overview.*

Sceptic Colleagues. This force is often reinforced in the beginning of a reverse engineering project, because as a reverse engineer –or worse, a consultant– 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 as the first impression will have dire consequences later.*

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
READ ALL THE CODE IN ONE HOUR	++	++	+	-	+
SKIM THE DOCUMENTATION	++	++	-	+	-
INTERVIEW DURING DEMO	++	+	0	+	-

Table 4.1: How each pattern of FIRST CONTACT resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

READ ALL THE CODE IN ONE HOUR

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

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

> ***Example.*** *You are facing a 500 K lines C++ program, implementing a software system to display multi-media information in real time. Your boss asks you to look how much of the source code can be resurrected for another project. Before actually identifying what may be reused, you will leaf through the source code to get a feeling for its condition.*

Context

You are starting a reverse engineering project of a large and unfamiliar software system. You have the source code at your disposal and you have reasonable expertise with the implementation language being used.

Problem

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

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 (i) the important entities (i.e., classes, packages, ...); (ii) the coding idioms applied (i.e., C++ [COPL 92], [MEYE 98], [MEYE 96]; Smalltalk [BECK 97]); and (iii) the suspicious coding styles discovered (i.e., “code smells” [FOWL 99]). 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.

- Functional tests or units 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 [GAMM 95] may represent information that is constant for every complete execution of a system.
- Surprisingly large constructs often specify important chunks of functionality that should be executed sequentially.
- 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.

Forces Resolved

Limited Resources. By applying this pattern, you spend 1/2 a day (plus the time to collect the source code) to end up with a short list that is a reasonable basis for planning further reengineering efforts.

Tools and Techniques. Good source code browsers can speed you up and inheritance hierarchy browsers can give you a feel for the structure of a software system. However, be wary of fancy tools as they quickly overwhelm you with too much unnecessary information and may require a lot of time to configure correctly. Printing out the source code and reading a paper version may serve just as well.

Reliable Info. The concentrated contact with the code –and code is the only testimony you are sure is correct[1]– provides you with a rather unbiased view to start with. Moreover, by applying this pattern –especially in combination with SKIM THE DOCUMENTATION (p. 114)– you may already have encountered some contradicting pieces of information, which is definitely worthwhile to explore in further depth.

Abstraction. The information you get out is fairly close to the source code, consequently the abstraction level is quite low. However the fact that you work under time pressure forces you to skip details driving you towards an abstract view of the software system.

Sceptic Colleagues. The mere effect of asking quite precise questions after only 1/2 a day of effort raises your credit tremendously, usually enough for being allowed to continue your attempts.

Rationale

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.

Moreover, 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 reliable source of information. Be careful though 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.

Finally, 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.

[1]Remember the old Swiss saying: ”If the map and the terrain disagree, trust the terrain”

Known Uses

While writing this pattern, one of our team members applied it to reverse engineer the Refactoring Browser [ROBE 97a]. 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. 111) in conjunction with SKIM THE DOCUMENTATION (p. 114) 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. 117), but then you should be aware that this will bias your opinion.

Resulting Context. 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 GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to improve the list of important entities. Depending on whether you want to wait for the results of SKIM THE DOCUMENTATION (p. 114), you should consider to CONFER WITH COLLEAGUES (p. 152).

SKIM THE DOCUMENTATION

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

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

> ***Example.*** *You must develop a geographical information system. Your company has once been involved in a similar project, and your boss asks you to check if some of the design of this previous project can be reused. Before doing any design extraction on the source code, you will skim the documentation to see how close this other system is to what you are expected to deliver.*

Context

You are starting a reverse engineering project of a large and unfamiliar software system. You have the documentation at your disposal and you are able to interpret the diagrams and formal specifications contained within.

> ***Example.*** *If the documentation relies on use cases (see [*JACO *97]) for recording scenarios or formal languages for describing protocols, you should be able to understand the implications of such specifications.*

Problem

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

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 (i) the important requirements; (ii) the important features (iii); the important constraints; (iv) 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.

Be aware for documentation that 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 times with versions, file format and platform issues that certain word processors and CASE tools do not succeed to address.

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.
- References to other parts of the documentation convey chronological dependencies.
- Figures are a always a good means to communicate information. A list of figures, if present, may provide a quick path through the documentation.
- Screen-dumps, sample print-outs, sample reports, command descriptions, reveal a lot about the functionality provided in the system.
- Formal specifications, if present, usually correspond with crucial functionality.
- An index, if present contains the terms the author deems significant.

Forces Resolved

Limited Resources. By applying this pattern, you spend 1/2 a day (plus the time to collect the documentation) to end up with a short list that is a reasonable basis for planning further reengineering efforts.

Tools and Techniques. As reading the documentation only requires the physical document, the tool interference is really low. Yet, when CASE tools have been applied, it may be necessary to consult the documentation on line. Note that CASE tools often enforce some documentation conventions so be sure to be aware of them.

No special techniques are necessary to apply this pattern, unless formal specification or special diagrams are used.

Reliable Info. The success of this pattern depends heavily on the quality of the documentation. Applying this pattern (especially combined with READ ALL THE CODE IN ONE HOUR (p. 111)), you may have encountered some contradicting pieces of information, which is definitely worthwhile to explore in further depth.

Abstraction. The abstraction level you get out depends largely on the abstraction level of the available documentation, but is usually quite high because documentation is supposed to be written at a certain abstraction level.

Sceptic Colleagues. Unless good documentation is available, sceptics will almost certainly consider this activity a waste of time and you will probably loose some credibility with the faithful and fence sitters. This is a negative effect, so reduce its potential impact by limiting the time spend here.

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.

Related Patterns

You may or may not want to SKIM THE DOCUMENTATION (p. 114) before READ ALL THE CODE IN ONE HOUR (p. 111) 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. 117) can help you to collect a list of entities you want to read about in the documentation.

Resulting Context. 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. 111) and SKIM THE DOCUMENTATION (p. 114) this is a good basis to CONFER WITH COLLEAGUES (p. 152) and then proceed with GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126).

INTERVIEW DURING DEMO

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

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

> ***Example.*** *You are asked to extend an existing database application so that it is now accessible via the world-wide web. To understand how the end-users interact with the application, you will ask one of the current users to show you the application and use that opportunity to chat about the systems user-interface. And to understand some of the technical constraints, you will also ask one of the system maintainers to give you a demo and discuss about the application architecture.*

Context

You are starting a reverse engineering project of a large and unfamiliar software system. You have found somebody to demonstrate the system and explain its usage.

Problem

You need an idea of the typical usage scenario's plus the main features of a software system in order to plan further reverse engineering efforts.

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 (i) some typical usage scenarios or use cases; (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.

Hints. The person who is giving the demo is crucial to the outcome of this pattern so care is demanded when selecting the person. Therefore, consider to apply this pattern several times with different kinds of persons giving the demo. This way you will see variances 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.

Forces Resolved

Limited Resources. By applying this pattern, you spend 1/2 a day (plus the time to set-up the demo) to end up with a short list that is a reasonable basis for planning further reengineering efforts.

Tools and Techniques. Except for the equipment necessary to run the software system –which should be readily available– this pattern does not require anything special. The interviewing technique to apply requires a special listening ear though.

Reliable Info. A demo is a reliable means to dig out what features are considered important, but you cannot trust on it to omit irrelevant features. Of course the reliability of the information obtained depends largely on the person who is giving the demo. Therefore, if possibly cross-check any information against other more reliable sources (requirements, progress and delivery reports, source code, log files, ...).

Abstraction. The abstraction level achieved by seeing a demo is quite abstract, though it depends on the person who is giving a demo.

Sceptic Colleagues. The users and maintainers of a software system are usually quite eager to show you the system and tell you what they like and dislike about it. If you have a good listening ear this is a good way to boost your credibility.

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 predefined 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 spend about an hour playing around with the user interface to came 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 what was 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. 117) 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. 111) and SKIM THE DOCUMENTATION (p. 114).

Resulting Context. This pattern results in (i) some typical usage scenarios or use cases; (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 result of READ ALL THE CODE IN ONE HOUR (p. 111) and SKIM THE DOCUMENTATION (p. 114) this is a good basis to CONFER WITH COLLEAGUES (p. 152) and then move on to GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126).

Chapter 5

Cluster: Extract Architecture

The patterns in FIRST CONTACT (p. 109) should have helped you getting an initial feeling of the software system. Now is the right time to draw some blueprints of the complete system that will serve as a roadmap during the rest of the reverse engineering project. The main priority in this stage of reverse engineering is to get an accurate picture without spending too much time on the hairy details.

The patterns in this cluster tell you how to derive a system blueprint from source code (GUESS OBJECTS (p. 123)) and from a database schema (CHECK THE DATABASE (p. 126)). With these blueprints you will probably want to proceed with FOCUS ON HOT AREAS (p. 129).

Forces Revisited

Reliable Info. Since the blueprints resulting from these activities will influence the rest of your project, accuracy is the single most important aspect. *Consequently, take special precautions to make the extracted blueprints as reliable as possible.* In particular, plan for an incremental approach where you gradually improve the blueprints while you gain a better understanding of the system.

Limited Resources. Results coming from this stage of reverse engineering are always worthwhile. *Consequently, consider* EXTRACT ARCHITECTURE *a very important activity and plan to spend a considerable amount of your resources here.* However, via an incremental approach you can stretch your resources in time.

Tools and Techniques. While extracting an architecture, you can afford the time and money to apply some heavyweight techniques and purchase some expensive tools. *Yet —because accuracy is so important— do never rely on techniques and tools and always make a conscious assessment of their output.*

Abstraction. Architectural blueprints are meant to strip away the details. Yet, computer science has this strange phenomenon that details are crucial to the overall system [BROO 87]. *Consequently, favor different blueprints that emphasize one perspective and choose the most appropriate ones when necessary.* Adapt the notation to the kind of blueprint you are making ([DAVI 95] – principle 21).

Sceptic Colleagues. Good blueprints 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 blueprints to produce and which notations to use — they should be helpful to all members of the team.*

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
GUESS OBJECTS	-	-	++	++	+
CHECK THE DATABASE	-	-	++	+	++

Table 5.1: How each pattern of FIRST CONTACT resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

GUESS OBJECTS

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Progressively refine a model of a software system, by defining hypotheses about what should be in the code and checking these hypotheses against the source code

> ***Example.*** *You are facing a 500 K lines C++ program, implementing a software system to display multi-media information in real time. Your boss asks you to look how much of the source code can be resurrected from another project. After having* READ ALL THE CODE IN ONE HOUR *(p. 111), 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 gradually refine your assumption that the class* VIDEOCHANNEL *is the subject being observed by reading its source code and tracing interesting paths.*

Context

You are in the early stages of reverse engineering a software system: you have an initial understanding of its functionality and you are somewhat familiar with the main structure of its source code. Due to this, you have identified a certain aspect of the system as especially important. You have on-line access to the source code of the software system and the necessary tools to manipulate it (i.e., from an elementary grep to a professional browser). You have reasonable expertise with the implementation language being used.

Problem

You must gain an overall understanding of the internal structure of a software system and report this knowledge to your colleagues so that they will use it as a kind of roadmap for later activities.

Solution

Take a notepad and/or sketchpad (not necessarily as an electronic tool). Based on your experience, and the little you already understand from the system, devise a model that serves as your initial hypotheses of what to expect in the source code. Check these hypotheses against the source code, using whatever tools you have available. Consciously keep track of which parts of the source code confirm and which parts contradict your hypotheses. Based on the latter, refine the initial model, recheck the hypotheses and rework the list of confirmations and contradictions. Do this until you obtain a more or less stable model.

Note that it is a good idea to sort the entities in your hypotheses models according to the probability of appearance in source-code. This is especially useful as names inside the source-code do not always match with the concepts they represent. This may be due to particular coding conventions or compiler restrictions (identifiers cannot exceed a certain length), or because of the native language of the original programmer.[1]

Afterwards, sit down to produce a boxes- and arrows diagram describing your findings. As a rule of the thumb, make sure your diagram fits on one page. It is better to have two distinct diagrams, where each

[1] In one particular reverse engineering experience, we were facing source code that was a mixture of English and German. As you can imagine, grep is not a very good tool to check occurrences of English terms in German texts.

provides a clean perspective on the system than one messy diagram with too many details too read and memorize. People should be able to redraw the diagram from memory after they have seen it once; it is only then that your diagram will really serve as a roadmap.

Variations. The pattern itself is quite broad and thus widely applicable. Below are some suggestions of possible variants.

- *Guess Patterns.* While having READ ALL THE CODE IN ONE HOUR (p. 111), you might have seen some symptoms of patterns. You can use a variant of GUESS OBJECTS to refine this knowledge. (See the better known pattern catalogues [GAMM 95], [BUSC 96], [FOWL 97b] for patterns to watch out for. See also [BROW 96] for a discussion on tool support for detecting patterns.)
- *Guess Object Responsibilities.* Based on the requirements resulting from SKIM THE DOCUMENTATION (p. 114), you can try to assign object responsibilities and check the resulting design against the source code. (To assign object responsibilities, use the noun phrases in the requirements as the initial objects and the verb phrases as the initial responsibilities. Derive a design by mapping objects on class hierarchies and responsibilities on operations. See [WIRF 90] for an in depth treatment on responsibility-driven design.)
- *Guess Object Roles.* The usage scenarios that you get out of INTERVIEW DURING DEMO (p. 117) may serve to define some use cases that in turn help to find out which objects fulfill which roles. (See [JACO 92] for use cases and [REEN 96] for role modeling.)
- *Guess Process Architecture.* The object-oriented paradigm is often applied in the context of distributed systems with multiple cooperating processes. A variant of GUESS OBJECTS may be applied to infer which processes exist, how they are launched, how they get terminated and how they interact. (See [LEA 96] for some typical patterns and idioms that may be applied in concurrent programming.)

Forces Resolved

Limited Resources. The amount of resources you invest in this pattern depends mainly on the level of detail and accuracy that you want to achieve. Be wary of the hairy details though, as this pattern tends to have an exponential effort/gain curve. For detailed information, consider switching to STEP THROUGH THE EXECUTION (p. 142) instead.

Tools and Techniques. Applying this pattern does not require a lot of tools: a a simple grep may be sufficient and otherwise a good code browser will do. Probably you will also need a tool for producing the final blueprint, as it is likely that someone will have to update the blueprint later on in the project. However, choose a a simple drawing tool rather then a special purpose CASE tool, as you will need a lot of freedom to express what you found.

In itself, the pattern does not require a lot of techniques. 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, yet lots of this expertise may be acquired on the job.

Reliable Info. The blueprints you extract by applying this pattern are quite reliable because of the gradual refinement of the hypotheses and confirmation against source code. Yet, be sure to keep the blueprint up to date while your reverse engineering project progresses and your understanding of the software system grows.

Abstraction. If applied well, the different blueprints you achieve by means of GUESS OBJECTS provide the ideal abstraction level. That is, each blueprint provides a unique perspective on the software system that highlights the important facts and strips the unimportant details. Yet, navigating between the various blueprints provides you all the necessary perspectives to really understand the system.

Sceptic Colleagues. The results of GUESS OBJECTS pattern should drastically increase the confidence of your team in the success of the reverse engineering project. This is because the members of the team will normally experience an "*aha erlebness*", where the little pieces of knowledge they have fit the larger whole.

Rationale

Clear and concise descriptions of a system are a necessary ingredient to plan team activities. However, being clear and concise is for humans to decide, thus creating them requires human efforts. On the other hand, they must accurately reflect what's inside the system, so somehow the source-code should be incorporated in the creation process as well. GUESS OBJECTS addresses this tension by using a mental model (i.e., the hypotheses) as the primary target, yet progressively refines that model by checking it against source code. Moreover, conciseness implies loss of detail, hence the reason to extract multiple blueprints offereing alternative perspectives.

Known Uses

In [MURP 97], 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 hypotheses 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 (p. 109) cluster are meant to help you building the initial hypotheses to be refined via GUESS OBJECTS (p. 123). Next, some of the patterns in FOCUS ON HOT AREAS (p. 129) may help you to refine this hypothesis.

Resulting Context. After this pattern, you will have a series of blueprints where each contains one perspective on the whole system. These blueprints will help you during later reverse engineering steps, in particular the ones in FOCUS ON HOT AREAS (p. 129) and PREPARE REENGINEERING (p. 145). Consequently, consider applying CONFER WITH COLLEAGUES (p. 152) after applying GUESS OBJECTS (p. 123).

CHECK THE DATABASE

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Get a feeling for the data model inside a software system by checking the database schema.

> ***Example.*** *You are asked to extend an existing database application so that it is now 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.*

Context

You are in the early stages of reverse engineering a software system, having an initial 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. Or even better, you have samples of data inside that database and maybe you are even able to spy on the database queries during the execution of the system. Finally, 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.

Problem

You want to derive a data model for the persistent data in a software system in order to guide further reverse engineering efforts.

Solution

Check the database schema to reconstruct at least the persistent part of the data model. Use your knowledge of how constructs in the implementation language are mapped onto database constructs to reverse engineer the real data model. Make samples of data inside the database to refine the data-model.

Forces Resolved

Limited Resources. Reconstructing the data model from the database schema takes considerable resources, although it depends largely on the underlying technology. Factors that affect this force in a positive way are the quality of the database schema (is it in normal form?), the correspondence between the database paradigm and the implementation language paradigm (inheritance hierarchies do not map directly to relational tables), the expressiveness of the database schema (does it include foreign keys ?). On the other hand, the reverse engineering of database schemas may include techniques like data sampling and run-time inspection, which takes even more resources.

Tools and Techniques. This pattern can do without a lot of tool support: a dump of the database schema and some samples of data inside the tables is something all database systems can provide. However, there are some tools available to support you in recovering object models (see [HAIN 96], [PREM 94], [JAHN 97]).

This pattern requires substantial technical expertise, because it requires knowledge of ways to manipulate data structures in both the implementation language and the database, plus ways to map one onto the other.

Reliable Info. Because the pattern is based on analyzing persistent data, the reliability of the reconstructed data model is usually quite high. However, if the database system is manipulated by different software systems and if each of these software systems is build with different implementation technologies (CASE tools, 4GL, ...), the reliability of the data model tends to decrease because the database schema provides the most common denominator of all implementation technologies involved. Data sampling is a good way to cope with this problem though.

Abstraction. The abstraction level of the reconstructed data model tends to be low, as it is closer to the underlying database schema than it is to the implementation language. However, this depends largely on the amount of resources spent. For instance, with data sampling and run-time inspection one can drastically improve the abstraction level.

Sceptic Colleagues. If applied well, this pattern increases your credibility considerably, because a well defined data model is normally considered a collective source of knowledge which greatly improves the communication within a team. Moreover, almost all software engineers will have experience with data models and will appreciate their presence.

Rationale

Having a well defined central data model is a common practice in larger software projects that deal with persistent data. Not only, it specifies common rules on how to access certain data structures, it is also a great aid in assigning development tasks. 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-known problem, drawing certain attention in the literature (see [HAIN 96], [PREM 94], [JAHN 97]). 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.

Related Patterns

CHECK THE DATABASE requires an initial understanding of the system functionality, like obtained by applying patterns in the cluster FIRST CONTACT (p. 109).

There are some patterns that describe various ways to map object-oriented data constructs on relational database counterparts. See among others [KELL 98], [COLD 99].

Resulting Context. CHECK THE DATABASE results in a data 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 GUESS OBJECTS (p. 123). 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 FOCUS ON HOT AREAS (p. 129) and PREPARE REENGINEERING (p. 145). Consequently, consider to CONFER WITH COLLEAGUES (p. 152) after CHECK THE DATABASE.

Chapter 6

Cluster: Focus on Hot Areas

The patterns in FIRST CONTACT (p. 109) should have helped you getting an initial feeling of the software system, while the ones in EXTRACT ARCHITECTURE (p. 121) should have aided you deriving some blueprints of the overall system structure. The main priority now is to get detailed knowledge about a particular part of the system.

This cluster tell you *how*, and to some degree *where* 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 you can afford to spend the resources obtaining detailed information as only then you have the necessary expertise to know that your investment will pay off.

There are two patterns that explain you *where* to focus your attention: INSPECT THE LARGEST (p. 131) suggests to look at large object objects, while EXPLOIT THE CHANGES (p. 135) advises to look at the places where programmers have been changing the system. (Of course, no technique or tool will replace the human mind, hence to know where to focus your attention, be sure to CONFER WITH COLLEAGUES (p. 152) as well). Then, there are two patterns that inform you *where and how* to study program structures: VISUALIZE THE STRUCTURE (p. 138) tells about program visualisation techniques, while CHECK METHOD INVOCATIONS (p. 140) recommends to check invocations of both constructor and overridden methods. Finally, there is one pattern describing you *how* to investigate programs, namely STEP THROUGH THE EXECUTION (p. 142) which explains how to take advantage of your debugger.

Many reverse engineering projects prepare for a subsequent reengineering phase. If you're in such a situation, you might consider the patterns in PREPARE REENGINEERING (p. 145) as your next step. If you're not, then these patterns are the last ones we have to offer for helping you.

Forces Revisited

Tools and Techniques. 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 force during this stage of reverse engineering: *consequently, make sure your reverse engineering team does possess 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*. The patterns in the previous clusters should have helped you obtaining that knowledge.

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 [BROO 87]. 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 Info. 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.

Sceptic 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.*

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
INSPECT THE LARGEST	0	-	0	0	0
VISUALIZE THE STRUCTURE	-	- -	+	+	+
CHECK OVERRIDDEN METHODS	-	-	+	+	0
EXPLOIT THE CHANGES	- -	- -	++	+	++
STEP THROUGH THE EXECUTION	-	0	++	-	+

Table 6.1: How each pattern of FIRST CONTACT resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

[1] A typical example of such a 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 members. 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 members. Finally, in Java a protected member may also be accessed by classes in the same package.

INSPECT THE LARGEST

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Identify important functionality by looking at large constructs.

> ***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.*

Context

You are in a later stage of reverse engineering a software system. You have an overall understanding of its functionality and you know the main structure of its source code. You have a metrics tool at your disposal plus a code browser to inspect the source code. The metrics tool is configured in such a way that it provides you with a number of measurements of source code constructs you are feeding into it. Moreover, the metrics are defined in such a way that they have a high correlation with the amount of functionality implemented in the construct.

Problem

You must identify those places in the source code that correspond with important chunks of functionality.

Solution

Use the metrics tool to collect a limited set of measurements for all the constructs in the system. Sort the resulting list according to these measurements. Browse the source code for the largest among those constructs in order to understand how these constructs work together with other related constructs. Produce a list of all the constructs that appear important, including a description of how they should be used (i.e. external interface).

Hints. Identifying important pieces of functionality in a software system via measurements is a tricky business which requires expertise in both data collection and interpretation. Below are some hints that might help you getting 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 will not perform better for the identification of large constructs. This experience is backed up by empirical evidence, as it has been reported in the literature that size metrics have a high correlation (see among others [FENT 97]).

  For identifying important functionality in object-oriented source code, look at methods and classes. For methods you may restrict yourself to counting the lines of code and if available the number of other methods invoked.[2] For classes, you should count the number of methods and the number of

[2]Counting the lines of code can be done very efficiently without parsing, just by counting all occurrences of the <CR> character.

attributes defined on that class, plus the depth of the inheritance tree and probably also the lines of code (i.e., the sum of all the lines of code of all the methods of a class). (See the chapter on Metrics — p.22 for a more precise definition of each of the metrics and a list of other metrics you might collect).

- *Which variants to use ?* Usually, it does not make a lot of difference which 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. In such a case, do not exclude comment lines nor normalise the source code as the extra effort will not pay of. Another example of an alternative definition is the case of counting the number of methods, where one must decide how to deal with 'special' methods like class methods (i.e., the C++ static methods). In this case it is a good idea to count class methods separately as they represent a different kind of functionality.
- *Which thresholds to apply ?* It is better not to apply thresholds to filter out those constructs which measurements fall into a given threshold interval. Indeed, 'large' is a relative notion and thresholds will distort your perspective of what constitutes large within the system.
- *How to interpret the results ?* Do not only look for the largest construct while analysing the data. Before actually browsing the source code, check the distribution of measurements to see whether the 80/20 rule is satisfied. Also, gather several measurements in different columns one beside another and then look for unusual rows.

  (Note that the 80/20 rule is a more formal expression of the rule of the thumb that most constructs in source code will be small, and only a few exceptional cases will be large. To be precise, the rule states that 80% of the constructs will be smaller than 20% of the size of the largest construct.)

Forces Resolved

Limited Resources. Once the metric tool is configured for the particular language of the software system, collecting the necessary data is not that resource consuming; in the worst case it can be done via batch jobs during the night. However, analysing the data and browsing the selected source code constructs requires a lot of resources depending on the desired level of detail. You can neutralise this effect to some extent by limiting the set of metrics.

Tools and Techniques. To apply this pattern, you require a tool which should be able to collect the necessary measurements. However, since you can restrict yourself to simple counting metrics such a tool should be quite easy to obtain.

Analysing and interpreting the data however, requires a certain amount of knowledge. Some of this knowledge is summarised in our list of metric definitions (see the chapter on Metrics — p.22) and the rest you can learn on the job.

Reliable Info. It is not because a software construct is large that it is important, neither is it true that a small construct is always irrelevant. Therefore, the results contain quite a lot of noise, hence are somewhat unreliable. Still, given the amount of resources required, this pattern usually provides a good return on investment, especially since the large constructs will often point you to other more important but smaller constructs.

Abstraction. The abstraction level of this pattern results mainly from browsing the source code, not so much from measuring the constructs in the system. Therefore, the abstraction level of the results should be considered quite low.

Sceptic Colleagues. Metrics are often associated with process and quality control, therefore some programmers may believe that you will use the metrics to examine their productivity. Be careful if you have such programmers among your faithful as it may be a way to turn them into sceptics. In particular, do not blindly deduce that large constructs are bad and should be rewritten.

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 low investment, even though the results are somewhat unreliable. With such a good focus, you can afford some erroneous results which you will compensate anyway via code browsing.

The results are a bit unreliable because ’large’ is not necessarily the same as ’important’. Quite often large constructs are irrelevant as they would have been refactored into smaller pieces if they were important. Conversely, 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. Still, different larger constructs may share the same smaller construct, so via the larger constructs you may be able to identify some important smaller constructs too.

The main disadvantage of the pattern is that it forces you to look at the largest constructs first. Large constructs are usually the most complicated ones, therefore understanding the corresponding source code may prove to be difficult. Another disadvantage is that the analysis of the metrics data results in a list of raw software constructs. For program understanding, it is usually more important to know how these constructs work together with other ones, something which must be revealed by code browsing.

Note that by restricting yourself to a limited set of simple metrics you already avoid one of the most common pitfalls. Indeed, metrics tools usually offer you a wide range of metrics and since collecting data is so easy, you may be tempted to apply all metrics that are available. However, the more data you collect, the more data you must analyse and the amount of numbers will quickly overwhelm you. Moreover, some metrics require substantial parsing effort, which in turn requires the configuration of the parser to your software system, which can be painstaking and time-consuming. By limiting the amount a metrics and keeping the metrics simple, you circumvent these problems.

Known Uses

In several places in the literature it is mentioned that looking for large object constructs helps in program understanding (see among others, [MAYR 96a], [KONT 97], [FIOR 98a], [FIOR 98b], [MARI 98], [LEWE 98a], [NESI 88]). 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 [MAYR 96a], TAC++ [FIOR 98a], [FIOR 98b], and Crocodile [LEWE 98a] 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 VISUALIZE THE STRUCTURE (p. 138) you invest in program visualisation techniques to study more aspects of the system in parallel, thereby increasing the quality of the outcome. Also, you can EXPLOIT THE CHANGES (p. 135) 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.

Resulting Context. 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 VISUALIZE THE STRUCTURE (p. 138) you will obtain other perspectives and probably other insights as well. Also, if you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the runtime behaviour. Finally, in the case of a object-oriented code, you can CHECK METHOD INVOCATIONS (p. 140) 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 [FOWL 99]), just like big classes may be cases of a GOD CLASS (see [BROW 98]).

EXPLOIT THE CHANGES

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Recover design issues by asking where, how and why the developers have been changing the implementation.

> ***Example.*** *You must understand an old but evolving software system, where the evolution is controlled through a configuration management system. You will filter out those modules that have been changed most often and find out what these changes where about and why they were necessary.*

> ***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.*

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 way to detect the differences between the releases, i.e. a configuration management system and/or a metrics tool.

Problem

You must identify those parts in the design that played a key role during the system's evolution.

Solution

Use whatever means at your disposal to compile a list of targets of important/frequent changes. For each target, put yourself in the role of the original developer and ask yourself what the change is about and why it was necessary. With this insight, produce a list of crucial system parts, including a description of the design issues that makes them important.

Variations. The pattern comes in two variants corresponding to the way the targets of changes are identified.

- *The configuration database variant* requires that all changes to the system were done via a configuration management system which logs all changes in the configuration database. In that case you can take advantage of the query facilities provided by the configuration database to produce a list of components that have been changed. Sort the list according to the frequency of changes and inspect the corresponding source code plus the comments in the configuration database to find out how and why this component has changed.

- *The change metrics variant* identifies changes by comparing subsequent releases and measuring differences in size. With the change metrics variant, the first step is to measure the size of named constructs in two subsequent releases. Afterwards, you compile a list with three columns: the name of the construct and both measurements. Sort the list according to the largest decrease in size. For each decrease in size, ask yourself where this functionality has been moved to and then deduce how and why this construct has changed.

Hints. If you consider applying the change metrics variant on object-oriented source code, we can recommend three heuristics that help identifying the following changes.

- *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- 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).
- *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.

Rationale

A configuration management tool maintains and controls the different versions of the components that constitute the entire software system. If such a tool has been used for the software system you are reverse engineering, its database contains a wealth of information about where, how and why the software system has evolved. As a reverse engineer, you should exploit the presence of this database.

But even without a configuration management system, it is feasible to identify where, how and to some degree why a system has evolved by comparing subsequent releases and measuring changes. With change metrics, the results are less accurate than it is the case with the configuration database variant, mainly because the rationale for the change is not recorded thus must be deduced. On the other hand, 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 GUESS OBJECTS (p. 123) and VISUALIZE THE STRUCTURE (p. 138)) (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 INSPECT THE LARGEST (p. 131)).

Known Uses

There is a company called MediaGeniX, which incorporates a scaled down version of the configuration database variant into their development process and tools. It is based on the so-called tagging tool, which automatically updates one comment line in a method body each time this method is modified. The comment line records information like the date of the change, the name of the programmer and a reference into their configuration management system. The reference reveals the nature of the change (i.e., bug fix or a new feature) and via consultation of the actual configuration management system even what this change was about. Afterwards, they run queries to identify which features are localised to a few modules and which features cross-cut a large number of modules to identify where they may improve the design of the framework. Also, they have identified methods that are modified a lot when bug fixing, and used this information as input for their code reviewing. They have even identified cycles in the bug fixing, in the sense that the modification of one method fixed a bug but immediately introduced another bug and then the repair of the newly introduced bug again introduced the older bug. More information about the usage of the tagging tool in the context of reverse engineering can be found in [HOND 98].

Besides the tagging tool, we are aware of two other projects where people have been exploiting the version control system for reverse engineering purposes. First, there is the SeeSoft tool, developed at Bell Labs, which visualises source code changes and has been used successfully for reverse engineering purposes [BALL 96]. Second, there is the ARES project (see http://www.infosys.tuwien.ac.at/Projects/ARES/) which also experimented with visualisation of changes using the 3DSoftVis tool [JAZA 99].

Finally, concerning the change metrics variant, we ran an experiment on three medium sized systems implemented in Smalltalk. As reported in [DEME 99c], 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 VISUALIZE THE STRUCTURE (p. 138) or INSPECT THE LARGEST (p. 131) you will get less accurate results for a lower amount of resources.

Resulting Context. 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 VISUALIZE THE STRUCTURE (p. 138) you will obtain other perspectives and probably other insights as well. Also, if you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the run-time behaviour. Finally, in the case of a class, you can CHECK METHOD INVOCATIONS (p. 140) to find out how this class is related to other classes.

VISUALIZE THE STRUCTURE

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Obtain insight in the software system's structure —including potential design anomalies— by means of well-known visualisations.

> ***Example.*** *You want to understand an object-oriented class structure in order to improve it. In particular, you would like to redistribute responsibilities, by splitting large superclasses and hooking the subclasses underneath the appropriate ancestor. To analyse the situation, you will display the inheritance hierarchies, paying special attention to large classes high up in the hierarchy. Afterwards, for classes identified that way, you will display a graph showing which method accesses which attributes to analyse the class' cohesion and find out whether a split is feasible.*

Context

You are in a later stage of reverse engineering a software system. You have an overall understanding of the system's functionality and based on that understanding, you have selected part of the software system for further inspection. You have a program visualisation tool at your disposal plus a code browser to inspect the source code.

Problem

You want to obtain insight in the structure of a selected part of a software system, including knowledge about potential design anomalies.

Solution

Instruct the program visualisation tool to show you a series of graphical layouts of the program structure. Based on these graphical layouts, formulate yourself some assumptions and use the code browser to check whether your assumptions are correct. Afterwards, produce a list of correct assumptions, classifying the items in one of two categories: (i) helps program understanding, or (ii) potential design anomaly.

Hints. Obtaining insight in the structure of a software system via visualisation tools is difficult, especially when searching for potential design anomalies. We have included our expertise with program visualisation in a separate chapter and we refer the interested reader to the chapter on Visualisation — p.31 for further details.

Rationale

Program visualisation is often applied in reverse engineering because good visual displays allow the human brain to study multiple aspects of a problem in parallel. This is often phrased as ”one picture conveys a thousand words", but then of course the problem is which words they convey, thus which program visualisations to apply and how to interpret them. For the program visualisations listed in the the chapter on Visualisation — p.31 we describe both when to apply them and how to interpret the results. For other visualisations you will have to experiment to find out when and how to use them.

Related Patterns

If your program visualisation tool scales enough to accommodate the system your facing, then you can start to VISUALIZE THE STRUCTURE right away. However, since program visualisations rarely scale well, it is preferable to first filter out which parts of the source code are relevant for further analysis. Therefore, consider to INSPECT THE LARGEST (p. 131) or to EXPLOIT THE CHANGES (p. 135) before you VISUALIZE THE STRUCTURE.

Resulting Context. By applying this pattern, you will have obtained an overview of the structure of a selected part of a software system, including potential design anomalies. Some other patterns may help you to further analyse these constructs. For instance, if you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the run-time behaviour and if you CHECK METHOD INVOCATIONS (p. 140) you can find out how a class is related to other classes. If you have identified design anomalies, you should consider to refactor them (see [FOWL 99]). Some typical design anomalies including the way to refactor them can be found in the part on Reengineering — p.163.

CHECK METHOD INVOCATIONS

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

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.

> ***Example.*** *You have identified a number of classes that represent part of the domain model. You want to learn about the aggregation relationships between these classes and therefore, you will inspect for all constructor methods which methods are invoking them.*

> ***Example.*** *You have identified a part of a class hierarchy where the designers relied on template methods to customise the design. To learn how the subclasses interact with their superclasses, you will retrieve all methods overriding another one, and inspect who is invoking these methods.*

Context

You are in a later stage of reverse engineering a software system implemented in an object-oriented language. You have an overall understanding of the system's functionality and based on that understanding, you have selected a part of the class hierarchy for further inspection. You have a code browser at your disposal that allows you to jump from a method invocation to the places where the corresponding method is defined.[3]

Problem

You want to find out how a class is related to the other classes in the system.

Solution

Select key methods in the interface of the class and inspect who is invoking these methods.

Variations. The pattern has two variants depending on the selected methods in the public interface.

- *The constructor method variant* suggests you to look at invocations of constructor methods to reveal aggregation relationships between classes.
- *The template method variant* recommends you to select methods that are overridden in a subclass plus the methods invoking them to infer template methods [GAMM 95].

[3]Note that your code browser should take polymorphism into account. Polymorphism implies that one invocation has several candidates for being the defining method. Because the actual target can only be resolved at run-time, your browser must show all candidates.

Hints. If you consider applying the above variants, following suggestions may help you getting the best out of your efforts.

- For *the constructor method variant*, you must trace the chain of invocations until the result of the constructor is stored into an attribute. The class defining this attribute is the aggregation. Also, look out for invocations of constructor methods where the invoking object is passing itself as an argument and where this argument is stored into an attribute of the constructor class. In that case, the constructor class is the aggregation.
- *The template method variant*, explicitly states that you should look a methods that are overridden and not methods that are declared abstractly. The reason is that not all template methods distinguish the hook method via an abstract method, but that often a concrete method is used to specify the default behaviour. By looking for overridden methods, you are certain that you will cover the latter case as well.

Rationale

If the object-oriented paradigm is applied well, state should be encapsulated behind the interface of a class (see [MEYE 97] and [BECK 97] among others). Therefore, to understand how a class is related to other classes, method invocations are more reliable than attribute declarations. Yet, because the amount of method invocations is large you must choose which invocations to analyse. This pattern helps you in this choice by suggesting two specific kinds of methods that are easy to identify and result in well-known class relationships.

Known Uses

In [DEME 98a] we report on a case study where we applied the template method variant.

Related Patterns

Checking method invocations of classes is quite tedious, thus it is best to start with a small amount of classes. Therefore, consider to INSPECT THE LARGEST (p. 131) or EXPLOIT THE CHANGES (p. 135) or VISUALIZE THE STRUCTURE (p. 138) to limit the amount of classes to inspect.

Resulting Context. By applying this pattern, you will know how a class is related to the other classes in the system. If you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the run-time behaviour of these relationships.

STEP THROUGH THE EXECUTION

Intent

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

> ***Example.*** *You have a piece of code that implements a graph layout algorithm and you must understand it in order to rewrite it. You will feed a graph into the program and use the debugger to follow how the algorithm behaves.*

Context

You are in a later stage of reverse engineering a software system. You have an overall understanding of the system's functionality and based on that understanding, you have selected part of the software system for further inspection. You have a debugger at your disposal that allows you to inspect data structures and to interactively follow the step by step execution of a piece of code. You know a set of representative input data to feed into that piece of code to launch a normal operation sequence.

Problem

You want to obtain a detailed understanding of the run-time behaviour of a piece of code.

Solution

Feed representative input data in the piece of code to launch a normal operation sequence. Use the debugger to follow the step by step execution and to inspect the internal state of the piece of code.

Hints.

- Test programs usually provide representative input data in their initialisation code.
- Usage scenarios, like the ones resulting from INTERVIEW DURING DEMO (p. 117), may give clues on what is a normal operation sequence.

Forces Resolved

Limited Resources Once you know what you really want to understand, this pattern works well in a limited resource context. However, stepping through the code can be highly inefficient whitout a clear focus.

Tools and Techniques The success of this pattern relies on the ability to use a good interactive debugger.

Reliable Info By following the step by step execution of a program, you get a very reliable view of a piece of code. However, beware that the input data is indeed representative for a normal operation sequence.

Abstraction The abstraction level is quite low, unless you can tie the step by step execution to a typical usage scenario.

Sceptic Colleagues Neutral.

Rationale

In [MEYE 97], object-oriented programming is defined as “designing a system around the functionality it offers rather then the data structures it operates upon”. Hence, understanding the run-time behaviour is crucial to understand an object-oriented program. And the best way to get a view on the run-time behaviour is to see the events as they actually occur in a real execution, a view which is provided by interactive debugging tools.

Known Uses

In [ROCH 93] you can find some interesting debugging techniques applicable in the context of Smalltalk. Many of them will generalise to other programming environments as well.

Related Patterns

Stepping through program executions is quite tedious, thus it is best to focus on a small piece of code. Consider to INSPECT THE LARGEST (p. 131) or to EXPLOIT THE CHANGES (p. 135) or to VISUALIZE THE STRUCTURE (p. 138) to obtain such a focus. Also, you need some typical usage scenarios which may be provided by INTERVIEW DURING DEMO (p. 117).

Resulting Context. By applying this pattern, you will have a detailed understanding of the run-time behaviour of a piece of code. This may be necessary to apply patterns in PREPARE REENGINEERING (p. 145).

Chapter 7

Cluster: Prepare Reengineering

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.

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
WRITE THE TESTS	- -	-	++	++	0
REFACTOR TO UNDERSTAND	- -	-	0	+	0
BUILD A PROTOTYPE	- -	-	+	- -	++
FOCUS BY WRAPPING	- -	0	0	0	0

Table 7.1: How each pattern of PREPARE REENGINEERING resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

WRITE THE TESTS

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

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.

> ***Example.*** *You are asked to extend a parser for a command language so that it is able to parse two additional commands. Before actually changing the of parser, you will write a number of test programs that check whether the parser accepts all valid command sequences and rejects some typical erronous ones.*

Context

You are at the final stages of reverse engineering a software system, just before you will start to reengineer a part of that system. You have sufficient knowledge about that part to predict its output for a given input.

Problem

Before starting to reengineer the system, you want to make sure that all what used to work keeps on working.

Solution

Write a number of black box tests that record your knowledge about the input/output behaviour.

REFACTOR TO UNDERSTAND

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Obtain better readable —thus more understandable— and better organised —thus more extensible— code via renaming and refactoring.

> ***Example.*** *You are asked to extend a parser for a command language so that it is able to parse two additional commands. Before actually extending the parser, you will improve the readability of the source code. Among others, you will rename key methods and classes to reflect your understanding of a parser and you will split long and complex methods into smaller ones. As an example of the former, you will rename the class StreamIntf into Scanner and the method rdnxt into nextToken. An example of the latter would be to split the nextToken method, so that it becomes a large case statement, where each branch immediately invokes another method.)*

Context

You are at the final stages of reverse engineering a software system, just before you will start to add new functionality to that system. You have a good programming environment that allows you to rename things easily and that operates on top of a version control system.

Problem

The shape of the code is such that it is difficult to read —hence to understand— and difficult to add the new functionality.

Solution

Reorganise the code so that its structure reflects better what the system is supposed to do.

BUILD A PROTOTYPE

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

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.

> ***Example.*** *You have a piece of code that implements a graph layout algorithm. You have an idea on how the algorithm works, but the code is too cryptic to map your knowledge of the algorithm onto the code. You will write a prototype that implements your understanding of the algorithm and map pieces of your code onto the existing code.*

Context

Problem

Solution

Focus by Wrapping

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

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

> ***Example.*** *You have to migrate a graph manipulation program from a Unix to Macintosh user-interface platform. The original program is well designed and has separated out most of the platform specific operations into a separate layer. You will clean up this layering by moving* ***all*** *platform specific behaviour into a separate layer, this way wrapping the obsolete part into a separate component.*

Context

Problem

Solution

Chapter 8

Cluster: misc

CONFER WITH COLLEAGUES

Intent

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

> ***Example.*** *Your team has to reverse engineer a workflow system containing lots of complex rules on how tasks get transferred. Each team member investigates a part of the system and as such the knowledge about the workflow rules is distributed over the team. To increase the overall understanding, you will devote 15 minutes of the weekly team meeting to discuss reverse engineering results made during the last week.*

Context

You are a member of a software development team and part of the job assigned to your team is the reverse engineering of a software system. Different members of the team perform different reverse engineering activities and consequently the knowledge about the system is scattered throughout the team.

Problem

How do you ensure that every team member contributes to the overall understanding of the software system.

Solution

Use whatever means at your disposal (meetings, e-mail, intra-nets, ...) to ensure that whenever any team member finishes a reverse engineering step, the obtained information is shared with the rest of the team.

Hints.

- To avoid information overload, choose the communication channels in such a way that sharing the information fits well with the culture within your team. For instance, do not organise a special team meeting devoted to reverse engineering results; rather use an existing meeting as a vehicle for applying this pattern.

Rationale

Reverse engineering is sometimes compared with solving a puzzle [Will96b]. If team members keep some pieces of the puzzle for themselves it will never be possible to finish the puzzle. Consequently, it is imperative that a reverse engineering team is organised in such a way that information may be shared among the various team members.

Chapter 9

Pattern Overview

The followin tables summarize the patterns for reference purposes.

The first series of tables lists the patterns together with their problem and their solution, this way aiding reverse engineers to identify which patterns may be applicable to their problem.

The second series of tables show how all the patterns work together to tackle an overall reverse engineering project. For each pattern, the tables list the context and prerequisites plus the pattern results and how these results may serve as input for other patterns.

First Contact

Pattern	Problem	Solution
Read All The Code in One Hour (p. 111)	You need an initial assessment of the internal state a software system to plan further reverse engineering efforts.	Grant yourself a reasonably short amount of study time to walk through the source code. Afterwards produce a report including a list of (i) the important entities; (ii) the coding idioms applied ; (iii) the suspicious coding styles discovered
Skim The Documentation (p. 114)	You need an initial idea of the functionality provided by the software system in order to plan further reverse engineering efforts.	Grant yourself a reasonably short amount of study time to scan through the documentation. Afterwards produce a report including a list of (i) the important requirements; (ii) the important features (iii); the important constraints; (iv) references to relevant design information.
Interview during Demo (p. 117)	You need an idea of the typical usage scenario's plus the main features of a software system in order to plan further reverse engineering efforts.	Observe the system in operation by seeing a demo and interviewing the person who is demonstrating. Afterwards produce a report including a list of (i) some typical usage scenarios or use cases; (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.

Extract Architecture

Pattern	Problem	Solution
Guess Objects (p. 123)	You must gain an overall understanding of the internal structure of a software system and report this knowledge to your collegues so that they will use it as a kind of roadmap for later activities.	Based on your experience, and the little you already understand from the system, devise a model that serves as your initial hypotheses of what to expect in the source code. Check these hypotheses against the source code, refine the initial model and recheck the hypotheses. Afterwards, produce a boxes- and arrows diagram describing your findings.
Check the Database (p. 126)	You want to derive a data model for the persistent data in a software system in order to guide further reverse engineering efforts.	Check the database schema to reconstruct at least the persistent part of the data model. Use your knowledge of how constructs in the implementation language are mapped onto database constructs to reverse engineer the real data model. Use the samples of data inside the database to refine the data-model.

FOCUS ON HOT AREAS

Pattern	Problem	Solution
INSPECT THE LARGEST (p. 131)	You must identify those places in the source code that correspond with important chunks of functionality.	Use a metrics tool to collect a limited set of measurements for all the constructs in the system. Sort the resulting list according to these measurements. Browse the source code for the largest among those constructs in order to understand how these constructs work together with other related constructs. Produce a list of all the constructs that appear important, including a description of how they should be used (i.e. external interface).
EXPLOIT THE CHANGES (p. 135)	You must identify those parts in the design that played a key role during the system's evolution.	Use whatever means at your disposal to compile a list of targets of important/frequent changes. For each target, put yourself in the role of the original developer and ask yourself what the change is about and why it was necessary. With this insight, produce a list of crucial system parts, including a description of the design issues that makes them important.
VISUALIZE THE STRUCTURE (p. 138)	You want to obtain insight in the structure of a selected part of a software system, including knowledge about potential design anomalies.	Instruct the program visualisation tool to show you a series of graphical layouts of the program structure. Based on these graphical layouts, formulate yourself some assumptions and use the code browser to check whether your assumptions are correct. Afterwards, produce a list of correct assumptions, classifying the items in one of two categories: (i) helps program understanding, or (ii) potential design anomaly.
CHECK METHOD INVOCATIONS (p. 140)	You want to find out how a class is related to the other classes in the system.	Select key methods in the interface of the class and inspect who is invoking these methods. Two examples of key methods that are easy to recognise are constructors and overridden methods.
STEP THROUGH THE EXECUTION (p. 142)	You want to obtain a detailed understanding of the run-time behaviour of a piece of code.	Feed representative input data in the piece of code to launch a normal operation sequence. Use the debugger to follow the step by step execution and to inspect the internal state of the piece of code.

FIRST CONTACT

Context: You are starting a reverse engineering project of a large and unfamiliar software system.

Pattern	*Prerequisites*	*Result*	*What next ?*
READ ALL THE CODE IN ONE HOUR (p. 111)	• source code • expertise with the implementation language	• the important entities (i.e., classes, packages, ...) • the coding idioms applied • the suspicious coding styles discovered	• SKIM THE DOCUMENTATION (p. 114) and INTERVIEW DURING DEMO (p. 117) to get alternative views • CONFER WITH COLLEAGUES (p. 152) to report findings • GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to refine the list of important entities
SKIM THE DOCUMENTATION (p. 114)	• documentation • you are able to interpret the diagrams and formal specifications contained within	• important requirements • important features • important constraints • references to relevant design information. + an assessment of the reliability and usefulness for each of the above.	• READ ALL THE CODE IN ONE HOUR (p. 111) and INTERVIEW DURING DEMO (p. 117) to get alternative views • CONFER WITH COLLEAGUES (p. 152) to report findings • GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to map the information on an overall system blueprint.
INTERVIEW DURING DEMO (p. 117)	• running system • somebody who can demonstrate how to use the system	• typical usage scenarios or use cases • the main features offered by the system and whether they are appreciated or not • the system components and their responsibilities • bizarre anecdotes that reveal the folklore around using the system	• READ ALL THE CODE IN ONE HOUR (p. 111) and SKIM THE DOCUMENTATION (p. 114) to get alternative views • CONFER WITH COLLEAGUES (p. 152) to report findings • GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to map the information on an overall system blueprint.

EXTRACT ARCHITECTURE

Context: You are in the early stages of reverse engineering a software system. You have an initial understanding of its functionality and you are somewhat familiar with the main structure of its source code. (This initial understanding might have been obtained by the patterns in FIRST CONTACT (p. 109)).

Pattern	*Prerequisites*	*Result*	*What next ?*
GUESS OBJECTS (p. 123) (variants: guess patterns, guess object responsabilities, guess object roles, guess process architecture)	• knowledge of important aspects of a software system • on-line access to the source code plus the necessary tools to manipulate it • reasonable expertise with the implementation language being used	• a series of blueprints, each one containing a perspective on the whole system	• CHECK THE DATABASE (p. 126) if you are interested in the data model • all patterns in FOCUS ON HOT AREAS (p. 129) if you want to refine the blueprints
CHECK THE DATABASE (p. 126)	• software system employs some form of a database • you have access to the database, including the proper tools to inspect its schema and samples of the data • knowledge of how data-structures from your implementation language are mapped onto the data-structures of the underlying database	• a data model of the persistent part of your system	• GUESS OBJECTS (p. 123) if you need to obtain other overall bluebrints of the system • all patterns in FOCUS ON HOT AREAS (p. 129) if you want to refine the datamodel

FOCUS ON HOT AREAS

Context: You are in a later stage of reverse engineering a software system. You have an overall understanding of its functionality and you are fairly familiar with the main structure of its source code.

Pattern	*Prerequisites*	*Result*	*What next ?*
INSPECT THE LARGEST (p. 131)	• a code browser • The metrics tool is configured with a number of size metrics	• a list of constructs representing important functionality	• VISUALIZE THE STRUCTURE (p. 138) to obtain other perspectives on those constructs. • STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour. • (in the case of object-oriented source code) CHECK METHOD INVOCATIONS (p. 140) to find out how classes are related to eachother • refactoring if you want to split some of these larger contructs into smaller ones
EXPLOIT THE CHANGES (p. 135) (variants:configuration database, change metrics)	• several releases of the source code • a configuration management system and/or a metrics tool	• a list of design parts that played a key role during the system's evolution	• VISUALIZE THE STRUCTURE (p. 138) to obtain other perspectives on those constructs. • STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour. • (in the case of object-oriented source code) CHECK METHOD INVOCATIONS (p. 140) to find out how classes are related to eachother
VISUALIZE THE STRUCTURE (p. 138)	• a part of the software system • a program visualisation tool • a code browser	• insight in the selected part • list of potential design anomalies	• STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour. • (in the case of object-oriented source code) CHECK METHOD INVOCATIONS (p. 140) to find out how classes are related to eachother • refactoring if you want to split some of these larger contructs into smaller ones

CHECK METHOD INVOCATIONS (p. 140) (variants: contructor methods, overridden methods)	• a part of the software system • a program visualisation tool • a code browser that allows you to jump from a method invocation to the places where the corresponding method is defined	• a list of classes and the relationships between them	• STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour.
STEP THROUGH THE EXECUTION (p. 142)	• a part of the software system • an interactive debugger • a set of representative input data	• insight into the run-time behaviour of a piece of code	• PREPARE REENGINEERING (p. 145) if you need to reengineer that piece of code