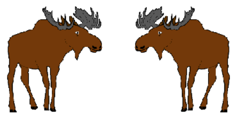

A Pattern Language for Reverse Engineering[1]

Serge Demeyer[(*)], Stéphane Ducasse[(+)], Oscar Nierstrasz[(+)]

[(*)] University of Antwerp - LORE - http://win-www.uia.ac.be/u/sdemey/

[(+)] University of Berne - SCG - http://www.iam.unibe.ch/~scg/

Abstract. Since object-oriented programming is usually associated with iterative development, reverse engineering must be considered an essential facet of the object-oriented paradigm. The reverse engineering pattern language presented here summarises the reverse engineering experience gathered as part of the FAMOOS project, a project with the explicit goal of investigating reverse and reengineering techniques in an object-oriented context. Due to limitations on EuroPLOP submissions, only part of the full pattern language is presented, namely the patterns describing how to gain an initial understanding of a software system.

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

A revised version of these patterns is published in: Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002. www.iam.unibe.ch/~scg/OORP

1. In Proceedings of EuroPLoP'2000, UVK GmbH, pp. 189-208.

Chapter 1

Reverse Engineering Patterns

1. Introduction

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

The patterns have been developed and applied during the FAMOOS project (http://www.iam.unibe.ch/~famoos/); a project with had the explicit goal 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.

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

2. Clusters of Patterns

The pattern language has been divided into *clusters* where each cluster groups a number of patterns addressing a similar reverse engineering situation. The clusters correspond roughly to the different phases one encounters when reverse engineering a large software system. Below is a short description for each of the clusters, while figure 1 provides a road map.

- **First Contact.** This cluster groups patterns telling you what to do when you have your very first contact with a software system.
- **Initial Understanding.** Here, the patterns tell you how to obtain an initial understanding of a software system, mainly documented in the form of class diagrams.
- **Detailed Model Capture.** The patterns in this cluster describe how to get a detailed understanding of a particular component in your software system.

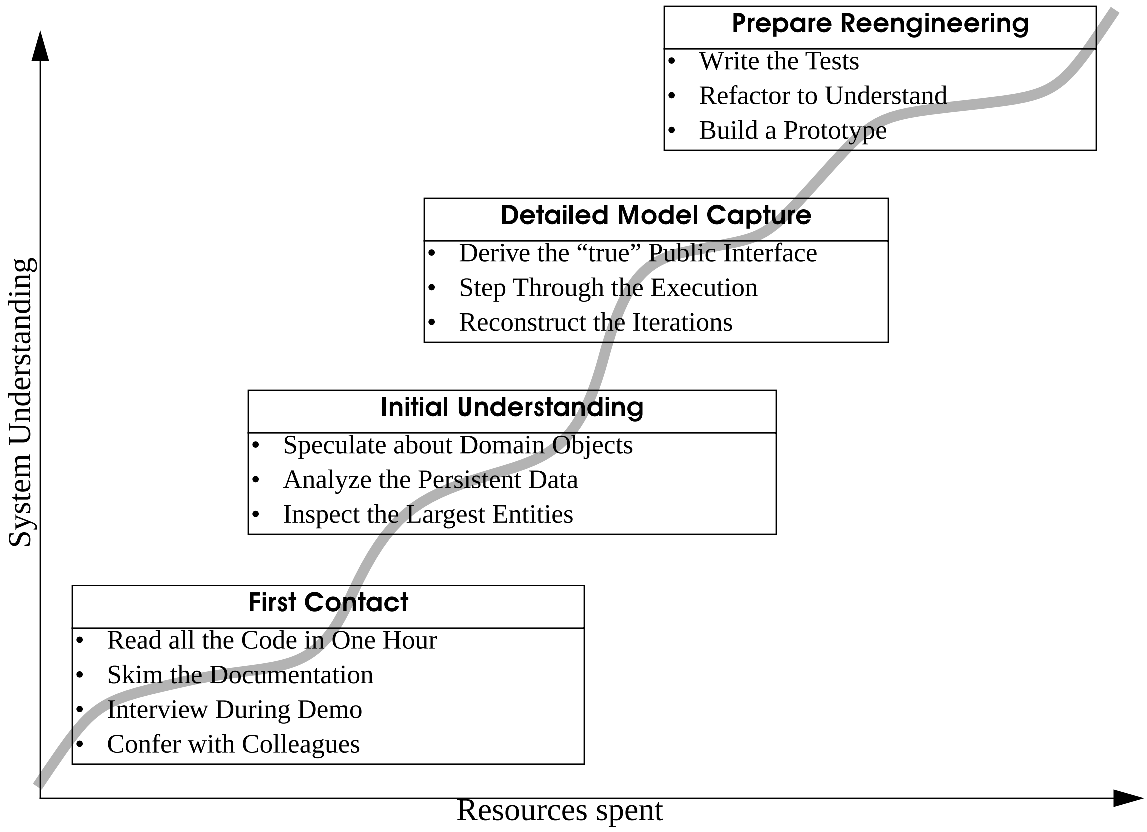

Figure 1: Overview of the four clusters in the pattern language.
Illustrating how understanding gradually increases with the amount of resources you spend

- **Prepare Reengineering.** Since reverse engineering often goes together with reengineering, this cluster includes some patterns that help you prepare subsequent reengineering steps.

Chapter 2

Initial Understanding

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

The patterns in this cluster tell you:

- How to extract a domain model from source code (Speculate about Domain Objects), with one variant concerning pattern extraction (Speculate about Patterns) and another concerning process architecture extraction (Speculate about the Architecture).
- How to extract a class model from a database (Analyze the Persistent Data).
- How to identify important chunks of functionality (Inspect the Largest Entities).

With this information you will probably want to proceed with Detailed Model Capture.

Speculate about Domain Objects

AKA: Map business objects onto classes

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

Problem

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

This problem is difficult because:

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

Yet, solving this problem is feasible because:

- You have a *rough understanding* of the system's functionality (for example obtained via Skim the Documentation and Interview During Demo), thus an initial idea of what aspects of the problem domain should represented.
- You have *development expertise*, so you can imagine how you would model the problem domain yourself.
- You are *somewhat familiar* with the main structure of the source code (for example obtained by Read all the Code in One Hour) and you have the necessary tools to browse it, so that you can find your way around.

Solution

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

Steps

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

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

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

Hints

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

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

Tradeoffs

Pros

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

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

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

Cons

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

Difficulties

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

Rationale

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

Known Uses

In [Murp97a], there is a report of an experiment where a software engineer at Microsoft applied this pattern (it is called 'the Reflection 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 then newcomer could come up with an initial hypothesis and then use the source code to gradually refine his understanding. Note that the paper also includes a description of a lightweight tool to help specifying the model, the mapping from the model to the source code and the checking of the code against the model.

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

Related Patterns

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

What Next

After this pattern, you will have a class model representing the problem domain concepts. Other patterns will help you deriving other views on the system, for instance Analyze the Persistent Data when you want to learn about the valuable data inside a system, or Inspect the Largest Entities when you want to identify the important functionality.

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

Speculate about Patterns

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

Description

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

Example

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

Speculate about the Architecture

Intent: Like Speculate about Domain Objects*, except that you build and refine a hypothesis about the architecture of a system. Especially useful in a distributed setting, where you build a hypothesis for the interacting processes in a distributed system.*

Description

“A software architecture is a description of the subsystem and components of a software system and the relationships between them” [Busc96a] (a.k.a. Components and Connectors

[Shaw96a]). The software architecture is typically associated with the coarse level design of a system and as such it is crucial in understanding the overall structure.

Moreover, the object-oriented paradigm is often applied in the context of distributed systems with multiple cooperating processes. To understand the inner workings of such a system, you must know about its architecture as this will help you mapping parts of the code to the corresponding process.

Therefore, a variant of Speculate about Domain Objects may be applied to infer which components and connectors exist, or in the context of a distributed system, which processes exist, how they are launched, how they get terminated and how they interact. (See [Busc96a] for a catalogue of architectural patterns and [Shaw96a] for a list of well-known architectural styles. See [Lea96a] for some typical patterns and idioms that may be applied in concurrent programming and [Schm00a] for architectural patterns in distributed systems.)

Analyze the Persistent Data

Intent: Learn about objects that are so valuable that they are stored in a database system.

Problem

You do not know which objects are valuable for the functioning of the system.

This problem is difficult because:

- "Valuable" is a subjective property, depending on which functionality is considered important for your reverse engineering project.
- Objects are run-time entities while most system descriptions are static. Run-time traces quickly generate huge amounts of data.

Yet, solving this problem is feasible because:

- The software system employs some form of a *database* to make its data persistent. Thus there exists some form of database schema providing a static description of the data inside the database.
- The database comes with the *necessary tools* to inspect the actual objects inside the database, so you can exploit the presence of legacy data to fine-tune your findings.
- You have some *expertise* with mapping data-structures from your implementation language onto a database schema, enough to reconstruct a class model from the database schema.
- You have a *rough understanding* of the system's functionality (for example obtained via First Contact), so you can put additional information in context.

Solution

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

Steps

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

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

1. Collect all table names and build a class model, where each table name corresponds to a class name.
2. For each table, collect all column names and add these as attributes to the corresponding class.
3. Collect all foreign keys relationships between tables and draw an association between the corresponding classes. (If the foreign key relationships are not maintained explic-

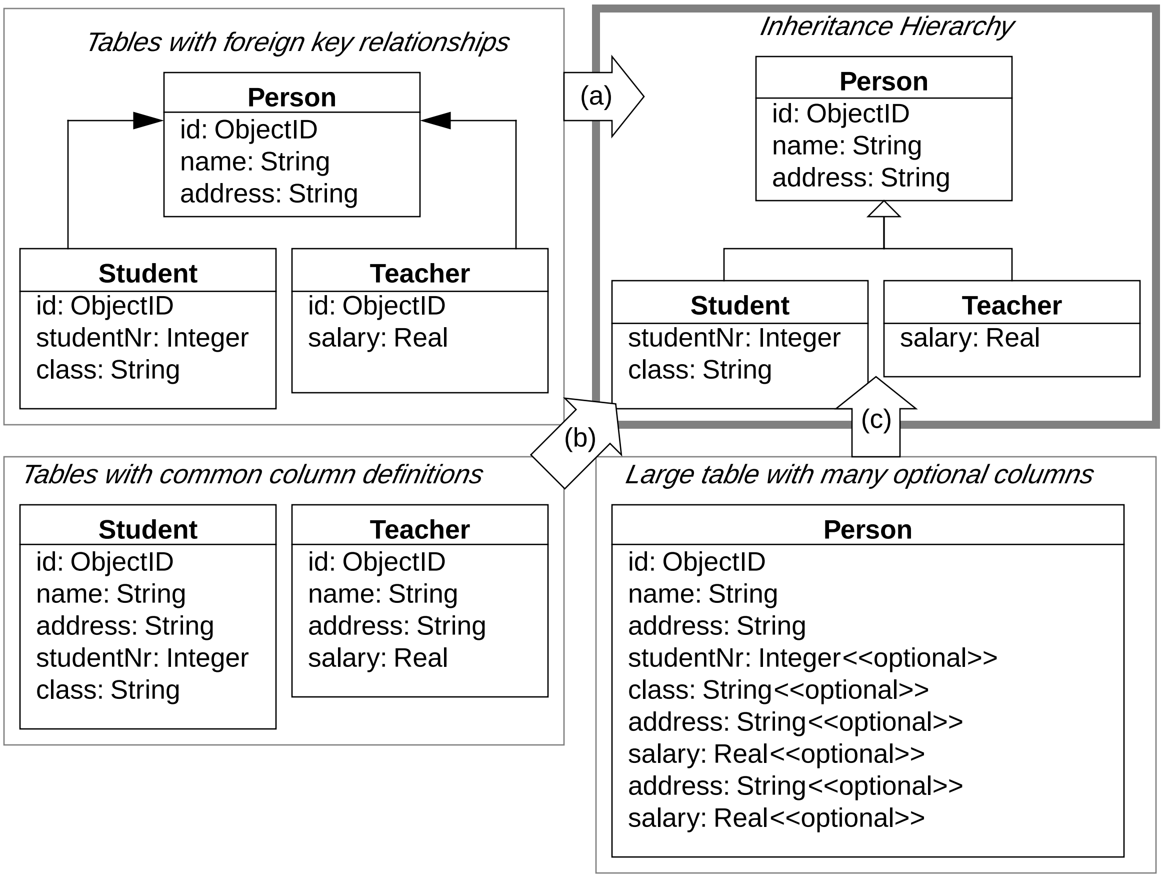

Figure 2: Mapping a series of relational tables onto an inheritance hierarchy. (a) one to one; (b) rolled down; (c) rolled up

itly in the database schema, then you may infer these from column types and naming conventions.)

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

4. Check tables where the primary key also serves as a foreign key to another table, as this may be a “one to one” representation of an inheritance relationship inside a relational database. Examine the SELECT statements that are executed against these tables to see whether they usually involve a join over this foreign key. If this is the case, transform the association that corresponds with the foreign key into an inheritance relationship. (see figure 2 (a)).
5. Check tables with common sets of column definitions, as these probably indicate a situation where the class hierarchy is “rolled down” into several tables, each table representing one concrete class. Define a common superclass for each cluster of duplicated column definitions and move the corresponding attributes inside the new class. To name the newly created classes, you can use your imagination, or better, check the source code for an applicable name. (see figure 2 (b))

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

When you have incorporated the inheritance relationships, consider to improve the class model exploiting the presence of the legacy system as a source of information. In particular you should inspect data samples to check for missing constraints and you should check at which queries are executed against the database engine to infer missing foreign keys.

Tradeoffs

Pros

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

Cons

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

Difficulties

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

Rationale

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

Known Uses

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

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

Related Patterns

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

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

What Next

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

Inspect the Largest Entities

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

Problem

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

This problem is difficult because:

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

Yet, solving this problem is feasible because:

- You have a *metrics tool* at your disposal, so you can quantify the size of entities in the source-code.
- You have a *rough understanding* of the system's functionality (for example obtained via First Contact), so you can put additional information in context.
- You have the necessary *tools to browse* the source-code, so you can verify manually whether certain entities are indeed important.

Solution

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

Steps

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

1. Identify large inheritance hierarchies.

   As inheritance is the most commonly used modeling concept in object-oriented systems it is a good idea to identify the largest subtree in the inheritance hierarchy as potential candidates for providing important functionality. To do this, compile a list of classes with the metrics "Number of Descendant Classes" and "Hierarchy Nesting Level" as the main indicators, and "Number of Methods for Class" plus "Number of Attributes for Class" as secondary indicators. Sort the list according the main indicators to identify those classes at the root or at the bottom of the large inheritance hierarchies (see Table 1).

	Number of Descendant Classes	Hierarchy Nesting Level	Number of Methods & Attributes for Class
(a) root of large inheritance hierarchy	large	small (~= 0)	Large values indicate a lot of impact on the subclasses.
(b) leaves of large inheritance hierarchy	small (~= 0)	large	Small values indicate a lot of impact from the parent classes.

Table 1: Identify large inheritance hierarchies.

2. Classes.

   Classes represent the unit of encapsulation in an object-oriented system, hence it is worthwhile to identify the most important ones. To do this, compile a list of classes with the metric "Lines of Code for Class" as main indicator and "Number of Methods for Class" plus "Number of Attributes for Class" as secondary indicator. Sort the list according to each of the criteria and inspect to top ten of each of them. Also, look for classes where the measurements do not correlate like the other classes in the system, they represent classes with exceptionally high or low values and are probably worthwhile to investigate further (see Table 2).

	Lines of Code for Class	Number of Methods for Class	Number of Attributes for Class
(a) large code size	large	Uncorrelated	
(b) many methods	Uncorrelated	large	Uncorrelated
(c) many attributes	Uncorrelated	Uncorrelated	large

Table 2: Identify large classes.

Hints

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

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

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

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

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

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

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

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

different larger entities will share the important smaller entities, thus via the larger entities you are likely to identify some important smaller entities too. Anyway, you should be aware that you are only applying a heuristic: there will be important pieces of code that you will not identify via this pattern.

Tradeoffs

Pros

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

Cons

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

Difficulties

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

Rationale

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

Known Uses

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

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

Related Patterns

What Next

By applying this pattern, you will have identified some entities representing important functionality. Some other patterns may help you to further analyze these entities. For instance, if you Step Through the Execution you will get a better perception of the run-time behavior. Finally, in the case of a object-oriented code, you can Derive the "true" Public Interface to find out how a class is related to other classes.

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

Chapter 3

References

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

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

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

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

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

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

[Deme00a] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, “Finding Refactorings via Change Metrics,” OOPSLA’2000 Proceedings, ACM Press, 2000.

[Fior98a] F. Fioravanti, P. Nesi, and S. Perli, "Assessment of System Evolution through Characterization," ICSE’1998 Proceedings, IEEE Press, 1998.

[Fior98b] F. Fioravanti, P. Nesi, and S. Perli, "A Tool for Process and Product Assessment of C++ Applications," CSMR’1998 Proceedings, IEEE Press, 1998.

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

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

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

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

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

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

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

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

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

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

[Kont97a] K. Konogiannis, "Evaluation Experiments on the Detection of Programming Patterns Using Software MEtrics," In WCRE’1997 Proceedings, IEEE Press, 1997.

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

[Lewe98a] C. Lewerentz and F. Simon, "A Product Metrics Tool Integrated into a Software Development Environment," ECOOP’98 Workshop Reader, Lecture Notes in Computer Science 1543, Springer-Verlag 1998.

[Mari98a] Radu Marinescu, “Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems,” Object-Oriented Technology (ECOOP'98 Workshop Reader), Serge Demeyer and Jan Bosch (Eds.), LNCS 1543, Springer-Verlag, 1998, pp. 252-253.

[Mayr96a] J. Mayrand, C. Leblanc and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", ICSM 1996 Proceedings, IEEE Press 1996.

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

[Nesi98a] P. Nesi, "Managing OO Projects Better", IEEE Software, July/August, pp.50-60, 1998.

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

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

[Schm00a]Douglas C. Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons 2000.

[Shaw96a]Mary Shaw and David Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.

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