Book Report

Object-Oriented Reengineering Patterns

Songfeng Wu (sw24)

The book first discusses the motivation of doing reengineering. Then the content is split into two main parts. One is reverse engineering while the other one is reengineering. I will follow the book's structure to discuss what the book introduces and what I learned from reading it.

The book begins with a traditional question why we need reengineering when we aim at developing a perfect system. The answer follows two development laws. Externally, software has to be changed to be compatible with the fast-changing environment. Internally, as software gets more and more complex while its functionalities being increased progressively, a simple structure and its extra content must be maintained and organized well respectively. And what motivates the authors to write the book is that the most valuable reengineering techniques and patterns are definitely nontrivial. The main idea and techniques are from the authors real-life software development experience.

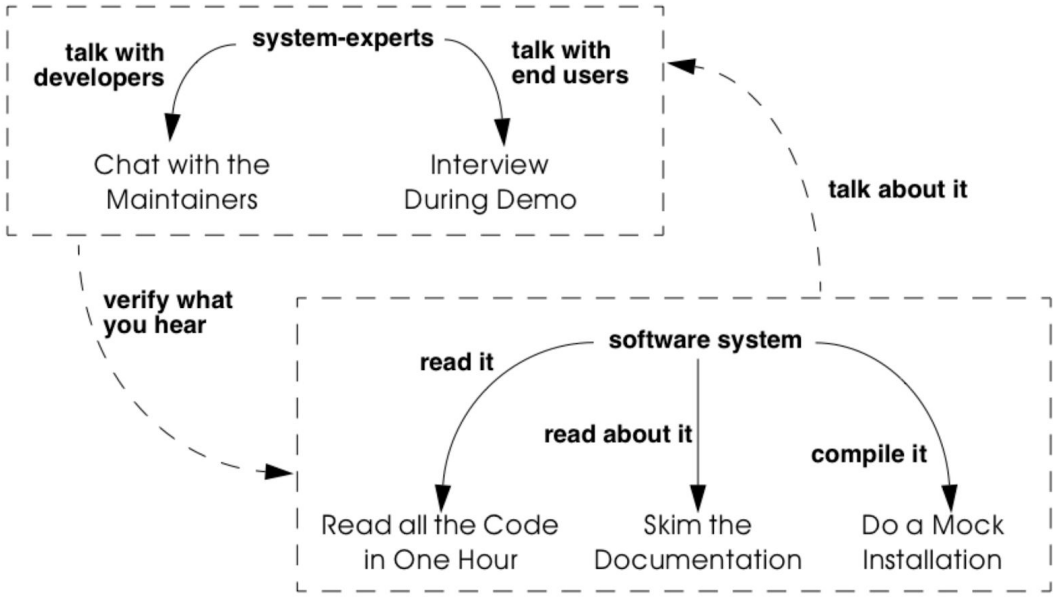
Essentially, reengineering simply is that we need to utilize resources we have to transform a legacy system which is as close to the "perfect system" as possible. Authors give out the concept of the Object-oriented legacy system as a successful object-oriented systems whose architecture and design no longer respond to changing requirements. Further, the book introduces the basic idea of pattern and design pattern as we learned in the class. The pattern is a recurring motif or structure that we see cyclicly during developing, while a design pattern is a generic solution that came up by people to resolve those recurring problems. Formally, the book gives the definition of design pattern languages, "a set of related patterns that can be used in combination to solve a set of complex problems." Based on the fact that patterns can be closely combined to solve a higher level problem which is relatively hard to fit with a single pattern. The structure of the book can then be figured. Each chapter gives some design patterns based on real-life experience and a guide on how to closely combine or composite patterns into one cluster. Based on the authors' experience, there are several reasons that we might want to apply reengineering strategy to a software. They are listed below.
(1) Unbundle a monolithic system so that the individual parts can be more easily marketed separately or combined in different ways.
(2) Improve performance
(3) Port the system to a new platform
(4) Extract the design as a first step to a new implementation
(5) Exploit new technology
(6) Reduce human dependencies by documenting knowledge about the system and making it easier to maintain.

During the early stage of one of my single full-stack project, I was trying to implement a function which was already implemented by an open source-project that I used for reference. The project was extremely huge and it was quite hard to unbundle the specific part that I would like to use on mine. Before I learned any reengineering techniques, I followed a naive approach that is doing it from scratch while following the idea of the open-source project. It turned out that the work is very time consuming and influence the project timeline. Even when I finally successfully implemented the function I was intended to, there was still a huge problem in front of me. As I wanted to add some new features, I found the most straightforward modification was either quite complicated or harmful to the existing functionalities, which implies that the system needs

reengineering as introduced in the book. The book concludes several signs that indicate a system needing reengineering. There are some main indicators that I also found helpful in my project. They can be listed as the following points.
(1) Obsolete or no documentation.
(2) Missing tests.
(3) Too long to turn things over to production.
(4) Too much time to make simple changes.
(5) Needs for constant bug fixes, new bugs pup up while a bug is fixed.
(6) Existing code smells.
Identifying such problems during the development can prevent further mistake which might be stacked on the previous incorrect implementation or idea.

Emerging from the above experience of implementing new functionality, an important idea is brought up by the book, which is “time is scarce”. The book introduces the idea while discussing the concept of reverse engineering. The main reason that the time is scarce is that wasting time during the early stage will result in severe consequences later on if the system is meant to be finished within a specific time period. A bad practice is that dive into the project before figuring out what the root problem of the project is.

Before starting coding work, it is also crucial to make sure that proper preparation has been made, such as setting up a navigator, setting up the periodically round table meeting, confirming the order of the tasks, identifying the exact problems, acquiring knowledge of the system and etc. Overall speaking, The preparation can be mainly divided into two parts. One is communicating with the experts who understand the systems well. The other one is understanding the system by ourselves. The graph below briefly shows how the process works.



Based on my own experience, I think holding the weekly round table meeting is the most important and efficient way to control the overall progress of the teamwork. One thing which needs to be addressed is that an individual may easily misses a point or making a conceptual

mistake. At the very beginning of my previous group project, we did not set up a firm plan on weekly meetings. Since the three of us all have different schedules, the meetings did not help make more progress. For example, when we had proper time for a meeting, some of us did not have the proper work done on time due to the random time meeting. We spent a lot of time helping each other catching up. The wasted time heavily slower our project progress. We had to put off the task which was supposed to be done at the time. Hence, at the end of the project, we had to work very intensively to reach what was expected at the beginning. Holding weekly meetings will not merely help the team member state on the same page. Moreover, it guarantees proper time management. In each meeting, the team can keep everything up to date which helps adjust the project schedule. As I mentioned, the round table meeting is that team members may have different agendas. People may also have different ideas on reengineering, which is seen as a minor problem of it. The book suggests convincing the most team members in believing reengineering which will greatly increase the efficiency of the teamwork as early as possible to avoid such a problem.

To keep the validation of the reengineering, having knowledge of the whole system is also quite important. The book recommends a pattern which is to view the code in a fairly short amount of time, generally one hour. Reviewing the code also helps ensure the code quality. One may make a checklist that covers the main idea of the legacy system. In our CS427 course project, before making any changes to the code, I always spend some time reading the test files and other files that tightly connect to the unimplemented part. Although the key point is to implement some new functionality, modifying the existing code is also required.

The book mentions that understanding the system by scanning the documentation, speculate the test cases, refine the designs, learn the information from data and etc. The pattern that reading the documentation is quite handy and is used in our current course project. Although as the book states, the documentation may contain a lot of non-useful information which may waste the reader's time, the advantage of doing so provides the reader with a high abstract understanding. I admit that simply reading the documentation may not be enough to fully understand the system. Combine reading documentation and speculating test cases may maintain the idea while minimizing the side effects. In our project, I usually prefer reading the use stories based documentation and exploring the test cases. It helps me quickly pick up the core idea that I need to know for a specific implementation. Also, It allows me to clearly figure out the structure of the partial system. Though I can try to minimize the possible side effect, just like what the book addresses, the first understanding might also be incorrect due to insufficient background knowledge of the system, which did occur based on my own experience. As the book emphasizes, "We get things wrong before we get things right!" Another unexpected influence is that following my strategy may consume a lot of time which violates the idea that one needs to strictly follow the time plan.

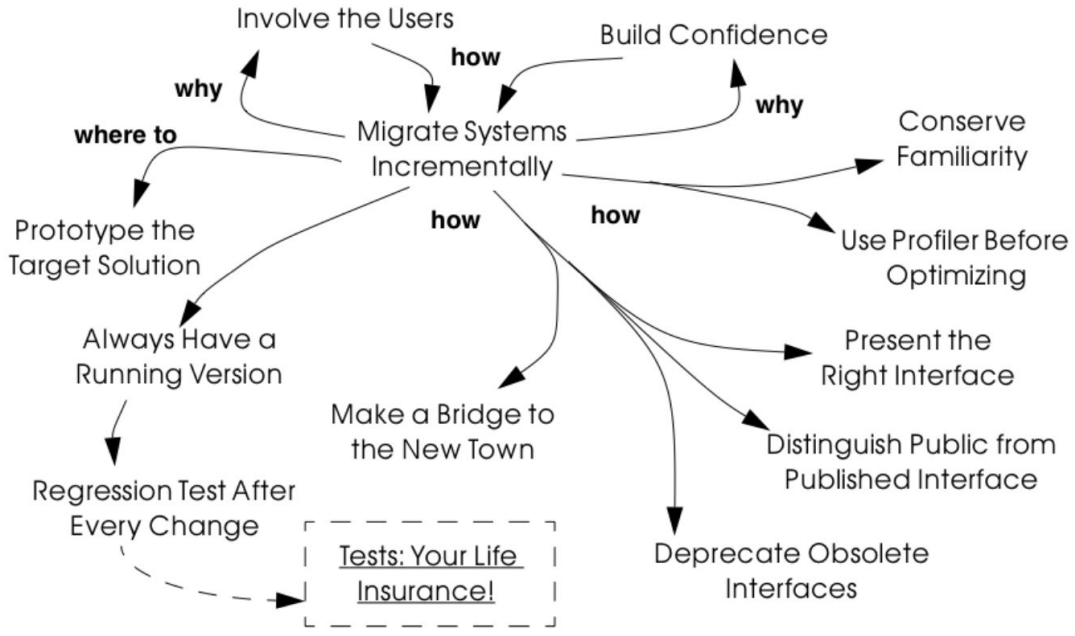
Then, the book states in detail how an engineer identifies potential design problems in a large system by collecting and evaluating measurements since most of the. The key idea of performing a detailed mode capture is to improve the understanding of the whole system through refactoring. For example, the book suggests writing comment-based or method based annotations. In our course project, I generally write comments for what I have changed to each function. And I keep

it up-to-date along I do the refactoring. The main advantage of doing it is that it let my teammates understand the code quickly. Besides that, the book addresses two more points that it also helps to minimize the context description and keep natural synchronization. Moreover, the book introduces how to have a better understanding by refactoring code partially and learn how classes collaborate by studying how clients use them.

In the second part, the book discusses the reengineering. To begin with, it emphasizes the importance of testing by claiming that testing is one's "life insurance" which is what I strongly agree with. There are two main reasons why we need tests, migrating the system incrementally and always having a running system. Another natural reason is that no customer accepts a buggy system. Besides those points, I learned that writing tests helps me to understand the functionality of the system by following the Test Driven Development in our course project. The book also emphasizes that one may record his or her understanding of a piece of code in the form of tests. By writing the tests first and then the functions that let test pass, we always have a running system that matches the principle mentioned in the book. At the same time, the book suggests developing incrementally, since some advanced tests may not help with the work at the time. This option will save some time by developing the tests we just need. Before applying the strategy, a good understanding of the system is needed since it is relatively hard to decide the boundaries of each component when the system is large. There are several key points that we have to test as mentioned in the book, testing fuzzy features, testing old bugs and retesting persistent problems. In order to write tests easily, efficiently and correctly, the book further recommends using a testing framework, such as the JUnit test we used in our project. The advantage of using test framework is that it simplifies the formulation of tests. When I use the JUnit test, given the sufficient online sources and the elaborate documentation, it greatly simplifies the process of writing test. Also, I do not need to put much effort into how to set up the JUnit test environment every time writing it. In one word, a testing framework makes it easier to organize and run tests. Although writing tests has a lot of advantages, the book warns the readers that writing tests is quite time consuming and one may want to follow the schedule strictly.

The book then introduces how one may migrate the old system to the new legacy system while the old system being used at the same time. Simply reengineer a legacy system and then deploy usually fail, claimed by the book. Moreover, a big Water-Fall project in new territories often fails in general. One must need to introduce the new changes to the system progressively, which is a nontrivial task to complete. It first requires one to engage customers, which is to let the user engaged in the migration in every step. Having feedback from users constantly will reduce the need for reengineering later on, though users may have problems evaluating system design. I usually ask a friend, who may be a potential user to my application, to give me feedback when I partially migrated some functionalities. In the beginning, it did take some time for him to catch up with what I was doing and what I wanted to reach. As he became more and more familiar with the system, such a process tended to be simpler and more efficient gradually. A user can evaluate a system from a different aspect that may not be thoroughly considered by the programmer. It quickly helped me address the key feature that the user wants the most. Communicating with the user can be considered in two parts. One is discussing with the user about the system designs as I mentioned above. The other one is that demonstrating users the

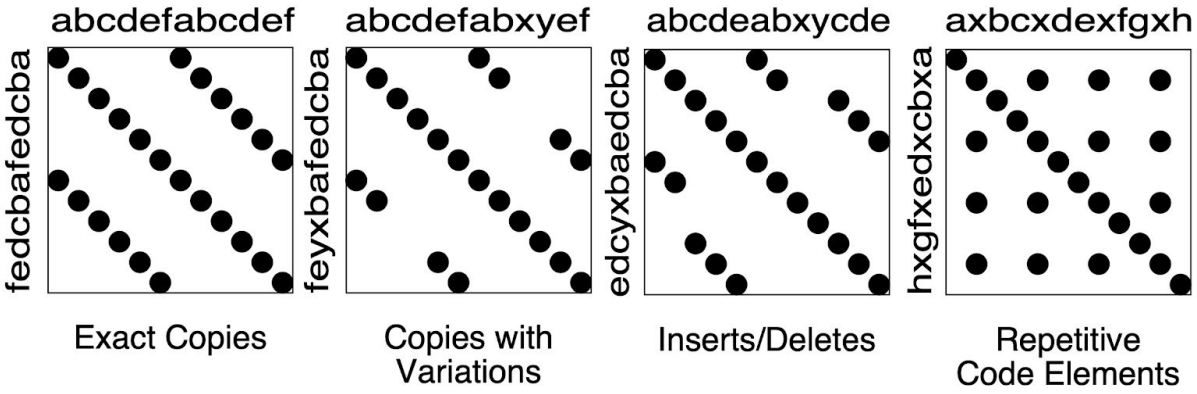
interface which requires one has a running version of the system always. The later one more directly conveys the information of how the system functions to the users. Although the system is changed incrementally and a running version is maintained, writing tests is an indispensable part of the migration. The main effect of testing is that it minimizes the unnecessary reengineering brought by the migration steps. Here, the rule of writing tests still applies, such as writing tests before implementing new functionalities, run those tests that might be affected by the changes, grow tests incrementally, strictly control the time spent on testing and etc. The following graph shows how to complete a migration of a system successfully.



In chapter 8, the book particularly emphasizes the importance of detecting and refactoring duplicate codes. It treats the duplicate code as the top one code smell which needs to be eliminated. Although not writing duplicate code sounds something easy to avoid, mistakenly introducing the duplicate code into a system is quite common, stated in the book. Before I took CS427, I was not aware of the harm of implementing duplicate code. Instead, I originally thought that writing duplicate code saves me some time when I found the existing code can be used for new tasks. When the duplicate code belongs to the different applications, the harm is minimal, the book states. Note that there are two main reasons that the duplicate code has to be eliminated. They are, duplicate code hampers the introduction of new changes since multiple parts of the system need to be modified and make the system more difficult to understand since one must identify the duplicates parts before trying to understand the system.

But according to the book, sometimes the duplicate problem is hard to be avoided. Consider the situation where two or three developers have to complete a project with a million level lines of code in eight months. It is just impossible for them without copying and pasting any existing code. Then the book gives some techniques about how to deal with delicate code. First, in order to make it easier to detect duplicate code, one may normalize the code with a specific format. Then one can hash each line and compare the hash values. This strategy is fairly easy to

implement whereas it may fail when the duplicate code is modified or there are simply too many duplicates. To enhance it, the book comes up with a solution called dot diagram. Let each axis of the matrix represent elements of the normalized files, the line of code for example. Place a dot for a match of two lines in two files in the matrix. Then the diagonal represents the duplicate code. The following picture indicates the multiple types of duplicate code. “Exact copies” means that diagonals of dots indicate copied sequences of source code. “The copies of variation” means that a portion of a copied sequences has been changed. Insert or delete means that a portion of code has been inserted or deleted. “Repetitive code elements” means that the same code is occurring periodically.



Although the dot diagram is quite simple, it works well when a large amount of unknown code is given. Also, the diagram is quite handy and easy to generate. Not surprisingly, there is some issue that may prevent one to get a good dot diagram. Such that a simple naive approach may not scale well to a large system. The developer may want to tune and optimize it to compromise simplicity. The interpretation of the dot diagram needs to be careful and subtle.

The book then discusses the responsibilities redistribution One may find that relatively modest changes to the system requires a long time of planning, testing and debugging due to weak encapsulation of data when he or she wants to migrate the system to make it more robust and adapt to possible future changes. It follows that redistributing responsibility is quite important. There are two main problems that come with reengineering. One problem is that data container (objects that just provide access to data, but no own behavior) is hard to be changed when may applications depend on them. The other one is that a developer may easily create god classes (a class that is superior to a lot of other classes) when he tries to complete any functional decomposition by following a fixed habit and style. In order to solve the first problem, the book suggests that strengthen encapsulation by moving behavior from indirect clients to the class containing the data it operates on under specific circumstances such as the functionality represents a responsibility of the provider, the functionality accesses the attributes of the provider and the functionality is defined by multiple clients. Further, the book introduces eliminating the navigation code and splitting up the god classes. Alone with identifying the difficulties and possible solutions of eliminating navigation code it strongly emphasizes that one should avoid defining accessors at the client class level which access the attributes of the provider attribute values. Next, the book discusses how to split the god classes. It gives an iterative solution as

redistributing responsibilities to data containers, or by spawning off new classes, until there is nothing left but a facade, and second by removing the facade.

In the last chapter named Transform Conditionals to Polymorphism, the book highlights the usefulness of eliminating a large number of case statements, since case statements make the code much more fragile in the long term according to authors' experience. The book introduces six patterns to solve the problem. They are introducing null objects, factoring out strategy, factoring out state transforming client type checks, transforming self type checks and transforming conditionals into the registration. The first pattern allows the system to encapsulate the null behavior as a provider class so that the client class does not have to perform a null test. It is a special case for transforming client type checks, mentioned in the book, which Implements a new method in each subclass of the provider hierarchy and replaces the entire conditional in the client by a function call to the new method. The following two patterns can be seen as two special cases of transforming self type checks. It Identifies or introduces subclasses according to the cases of the conditional and implements a hook method in each subclass for corresponding subclasses. Then it just simply replaces the conditional with a method call to the hook in the corresponding subclass. The difference between the two is that factoring out the state introduces function calls of new state object (A state allows an object to alter its behavior when its internal state changes. The object will appear to change its class.) to handle the cases in which the type information that is being tested may change dynamically while the other one introduces dynamically changed algorithm instead of state. According to the book, a strategy defines a family of algorithms, encapsulate each one in a separate class, and define each class with the same interface so they can be interchangeable. Strategy lets the algorithm vary independently from clients that use it. The transforming conditionals into the registration pattern defines a plug-in manager to manage all plug-in objects that will be queried by the tool clients to check the presence of the tools, where a tool client contains a lot of cases conditional. For each case of conditional, plug-in object should be created and registered automatically when the tool it represents is loaded, and it should be unregistered if and when the tool becomes unavailable. By querying the plug-in manager class, which returns a tool associated with the query and invoke it to access the wished functionality, the cases conditional in the tool client can be successfully removed.

To conclude, the book starts with definitions and background of reengineering that introduce the motivation and importance of applying reengineering techniques. Then it follows a structure that parallelly demonstrating how to understand the system as much as possible and how to perform reengineering on a system successfully. The way that the book introduces a design pattern always follows a specific format. It first introduce what the problem is. Then the book explains the possible difficulties and why it is feasible to solve. Further, it gives out some solutions and the rationale while analyzing the pros and cons. Finally, the book brings some examples to show how the strategy will work. Overall, the authors involve plenty of examples from their real-life experience of doing projects which, at least I think, is a great way to give readers a taste of reengineering. From the vivid examples, I quickly learned that some of the code in my current project can surely be enhanced. And the overview pictures greatly help me understand the workflows and processes. I also added some of the pictures included in the book report to help me introduce a concept. The main lesson that the book teaches me is how to thoroughly prepare, understand, improve and produce a system in a project. I think it covers the part I am mostly

interested in. And it serves as a good compensation to our course materials. I believe that I am able to apply the strategies that I learned from the book to my future projects.