

Strategic Systems Solutions

Allowing Systems to Evolve

A primer for reengineering and refactoring software

A Business Development Group White Paper





Contents

Introduction	3
Reengineering	4
Why Reengineer Software?	4
What Is Reengineering?	4
Reengineering Approaches	4
The Enterprise Solution	5
Software Refactoring	6
What is Software Refactoring?	6
Refactoring vs. Optimisation	6
Maintaining Correctness	6
Software Anti-Patterns	7
Summary	10
Supporting Material	11



Introduction

Organisations everywhere are experiencing tremendous pressure to evolve their systems so they can better respond to the needs of the marketplace. This pressure can be driven from a number of directions, typically escalating customer expectations, changes in enterprise standards, enhanced features, improved performance, integration with other systems, hardware/software obsolescence, and so on.

This document outlines some of the ideas, definitions, activities and principles of Software Reengineering and Refactoring, providing insight for developing a synergistic set of management and technical practices to achieve a disciplined approach to system evolution.



Reengineering

Why Reengineer Software?

Many organisations are faced with maintaining and developing software systems that suffer from the disorganisation that results from prolonged maintenance. As software ages, the task of changing it becomes more complex and more expensive. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor code quality, which in turn affects understanding.

Time-to-market is often used as an excuse in the sacrificing of things like supportability, maintainability, simplicity and even performance. Organisations soon discover that all these factors are equally important and that cutting corners to meet a more aggressive delivery schedule will, in the end, greatly reduce your time-to-market in subsequent phases.

Reengineering and/or refactoring a software project or suite of systems will not only improve it, but also allow it to evolve naturally, with a greater return on investment than could be obtained through a totally new development effort.

What Is Reengineering?

The Software Engineering Institute (SEI) defines “Reengineering” as:

“The systematic transformation of an existing system into a new form to realise quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer.”

Reengineering utilises a number of technical disciplines to allow program understanding and system design recovery from legacy artefacts such as source code. This is a significant problem in a wide array of application areas, where maintenance practices have failed to keep design and architectural documents (where they even exist in the first place) synchronous with changes to the production code.

The act of reengineering is not just one of maintenance, but actually system evolution. Software maintenance leaves the structure of the system relatively constant and the changes produce few economic and strategic benefits. System evolution is a structural form of change that makes the software qualitatively easier to maintain. Evolution allows the system to comply with new requirements and gain new capabilities, as well as increasing the strategic and economic value by making it easier to integrate with other systems. This can turn a project from a liability to an asset.

Reengineering Approaches

The approach chosen to evolve software-intensive systems depends on the organisation, the system, and the technology. There are basically two high-level approaches that can be adopted when reengineering a project.

1. Top-Down

The top-down approach treats the software more as “black boxes” that can be reformulated for integration with other systems. Sometimes it is more sensible and economical to reengineer systems by treating their components as black boxes and re-interfacing than it is to fully understand what is inside these boxes.

The various top-down reengineering methods available to the developer (use cases, goal



theory, interface/wrapping technology, etc.) are well documented and will not be covered in any further detail by this document.

2. Bottom-Up

From a bottom-up perspective, it is possible to start with a detailed code review and build up a new structure and a new form of documentation so that the system is qualitatively easier to maintain. Architectural extraction techniques may also start with source code analysis. This can be a good starting point when design documentation is inadequate or out of date.

The bottom-up reengineering options available to the developer are less well known, particularly in the field of Software Refactoring which is the topic of our next chapter.

The Enterprise Solution

System reengineering must not take place in a vacuum. Many attempts at evolution and migration fail because they concentrate on a narrow set of issues without considering the broader set of management and technical concerns. A focus on the technical problems to the exclusion of the enterprise problems is a recipe for disaster.

In addition to the software engineering and technology considerations, the enterprise approach addresses the needs of the customer, the organisation's strategic goals and objectives, the operational context of the enterprise, as well as the current legacy systems and their operational environment.

Before proceeding with system reengineering, it is useful to ask the following questions:

- *Have the benefits of evolving the system been determined?*
- *Has sufficient time been allowed for thorough systems analysis?*
- *How will work be prioritised and co-ordinated throughout the enterprise?*
- *How will lessons learned be captured and communicated, thus improving future practices?*



Software Refactoring

What is Software Refactoring?

Software Refactoring is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance. In most cases the goal is to transform code without impacting correctness.

As has been mentioned previously in this document, good software structure is essential for system extension and maintenance. Software development is a chaotic activity; the implemented structure of systems tends to stray from the planned structure as determined by architecture, analysis and design.

Refactoring vs. Optimisation

Refactoring is strongly recommended prior to performance optimisation. Optimisations often involve compromises to program structure. Ideally, optimisations affect only small portions of a system or program. Bottlenecks traditionally tend to be server/database access. Refactoring before optimisation can help partition the code that needs to be tweaked from the majority of the software.

Obviously there are times when this is not possible, such as when optimisation has been built into the code as it was being written. This is generally bad practice. Pareto's Law, a.k.a. the 80/20 Rule, states that 80 percent of a program's runtime is spent in 20 percent of the code. This rule is the dominating force driving the argument that premature optimisation is wasted effort.

Maintaining Correctness

There are various formal refactoring procedures whose implementations can be proven not to affect program correctness. Common techniques include superclass abstraction, conditional elimination and aggregate abstraction. These are large-grain transformations, dependant upon a multitude of small-grain program transformations familiar to virtually all programmers. Examples of small-grain transformations include renaming entities, creating new entities, migrating functionality, redirecting references, etc.



Software Anti-Patterns

Software Anti-Patterns are a way to recognise prevalent and recurring road-blocks to successful software evolution and maintenance. They efficiently map a general situation to a specific class of solutions using real world experience.

The Anti-Patterns outlined below focus on software, although there are solutions to many architectural and project management problems which will not be covered by this document. The solutions presented utilise various formal and informal refactoring approaches.

1 Spaghetti Code

Spaghetti Code is a classic and famous Anti-Pattern. It appears as a program or system with very little structure, where coding and progressive extensions have compromised it to such an extent that the structure lacks clarity, even to the original developer.

Symptoms:

- Quick demonstration code that became operational.
- Obsolete or scanty documentation.
- Minimal relationships between objects.
- Methods with no parameters.

Consequences:

- Very difficult to maintain and extend the system.
- Code is difficult to reuse, and when it is, it is often through cloning.
- Performance optimisation is almost impossible.

Refactored Solution:

1. Create abstract access to member variables using accessor functions.
2. Convert code segments into functions for reuse in future maintenance and refactoring efforts.
3. Reorder function arguments for consistency.
4. Remove obsolete portions of code.
5. Rename classes, functions and data types to conform to standard.

2 Cut and Paste Programming

Also known as “Software Cloning”. Cut-and Paste Programming is a very common, but degenerate form of software reuse. It creates maintenance nightmares and comes from the notion that it is easier to modify existing software than program from scratch.

Symptoms:

- Code is considered self-documenting.
- Lines of code increase without adding to overall productivity.
- The same bug reoccurs in the software despite many local fixes.
- Code reviews and inspections are needlessly extended.

Consequences:

- Excessive software maintenance costs.
- Defects are replicated, making it difficult to locate and fix every instance of a mistake.
- Reusable assets are not converted into an easily reusable and documented form.



Refactored Solution:

1. Mine the code to identify multiple versions of the same software segment.
2. Amend the code to create reusable components (black boxes).

3 The Blob

Also known as “The God Class”. The Blob is found in designs where one class monopolises the processing in a certain area. The Blob is generally a procedural design even though it may be represented using object notations and implemented in an object-oriented language.

Symptoms:

- Single class with a large number of attributes, operations, or both (typically 60 or more).
- A disparate collection of unrelated attributes and operations in a single class.
- A Single controller class with associated simple, data-object classes.
- A lack of OO design.
- A migrated legacy design which has not been properly refactored into an OO architecture.

Consequences:

- Too complex for reuse
- Difficult to test.
- Loss of OO advantage.
- May be expensive to load into memory for simple operations.

Refactored Solution:

1. Identify or categorise related attributes and operations according to contracts.
2. Find "natural homes" for these contract-based collections of functionality and migrate them there.
3. Remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments.

4 Functional Decomposition

Also known as “No OO”. Functional Decomposition is the result of experienced, non object-oriented developers who design and implement an application in an object-oriented language. The resultant code can be incredibly complex, as smart procedural developers devise very clever ways to replicate their time-tested methods in an object-oriented architecture.

Symptoms:

- Classes with “function” names (such as “DisplayTable” or “CalculateInterest”).
- All class attributes are private.
- Classes with a single action.

Consequences:

- No leveraging of OO principles (reuse, polymorphism, inheritance).
- Extremely expensive to maintain.
- No way to clearly document (or even explain) how the system works.
- Frustration and hopelessness on the part of the testers.

Refactored Solution:

1. Try to model a single method class as part of another existing class.
2. Combine several classes into a new class that satisfies a design objective.



3. If the class contains no state information, consider rewriting it as a function.

5	Poltergeists
---	--------------

Also known as “Proliferation of Classes”. Poltergeists are classes of limited responsibilities and roles to play in the system, therefore their effective lifecycle is quite brief. They clutter software designs, creating unnecessary abstractions and are excessively complex, hard to understand and difficult to maintain.

Symptoms:

- Redundant navigation paths.
- Transient associations.
- Stateless classes.
- Temporary, short-duration objects and classes.
- Classes with “control-like” operation names (such as “StartProcessAlpha”)

Consequences:

- A waste of resources every time they “appear”.
- Inefficient, as they utilise several redundant navigation paths.
- Needlessly clutter the design.

Refactored Solution:

1. Remove the controlling actions into the related classes that they invoke.



Summary

- One of the main goals of modern software is to move towards a paradigm of evolutionary systems. A system that evolves, rather than one that is repeatedly maintained, has the ability to be economically extended to incorporate extra functionality.
- Reengineering offers an approach to migrating an existing system to an evolvable system in a disciplined manner, using real-world engineering principles.
- Significant progress has been made in system understanding and object technology which makes it possible to extract architectural information from existing software system implementations. Once extracted, this “new” information allows top-down redesign and reengineering.
- Software Refactoring using AntiPatterns is a relatively new research area, which is a derivative of design patterns. It effectively provides a migration from negative solutions to positive solutions, resulting in the bottom-up modification of code to improve software structure in support of extension and long-term maintenance.
- Successfully evolving a system is not just a matter of addressing the technical problems. When performing reengineering work, there is a tendency to focus on a narrow set of technical issues without considering the broader system issues, such as the increased needs of the customer, the strategic goals and objectives of the organisation and the business operations of the enterprise.



Supporting Material

The following material will provide extra detail to the topics mentioned in this document:

- “The Unified Software Development Process” by Booch, Rumbaugh and Jacobson, published by Addison-Wesley.
- [The Software Engineering Institute](#), an on-line, federally funded research and development centre sponsored by the U.S. Department of Defence.
- [Software Development Magazine](#), an on-line, distributed development resource (requires registration for certain articles).
- [Reengineering Reference Library](#), a variety of downloadable papers that address reengineering issues.
- “Anti-Patterns” by Brown, Malveau, McCormick and Mowbray, published by Wiley.