Software Development AntiPatterns

Development AntiPatterns utilize various formal and informal refactoring approaches. The following summaries provide an overview of the Development AntiPatterns found in this chapter and focus on the development AntiPattern problem. Included are descriptions of both development and mini-AntiPatterns. The refactored solutions appear in the appropriate AntiPattern templates that follow the summaries.

- The Blob
  Procedural-style design leads to one object with a lion's share of the responsibilities, while most other objects only hold data or execute simple processes. The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes.
- Continuous Obsolescence
  Technology is changing so rapidly that developers often have trouble keeping up with current versions of software and finding combinations of product releases that work together. Given that every commercial product line evolves through new releases, the situation is becoming more difficult for developers to cope with. Finding compatible releases of products that successfully interoperate is even harder.
- Lava Flow
  Dead code and forgotten design information is frozen in an ever-changing design. This is analogous to a Lava Flow with hardening globules of rocky material. The refactored solution includes a configuration management process that eliminates dead code and evolves or refactors design toward increasing quality.
- Ambiguous Viewpoint
  Object-oriented analysis and design (OOA&D) models are often presented without clarifying the viewpoint represented by the model. By default, OOA&D models denote an implementation viewpoint that is potentially the least useful. Mixed viewpoints don't allow the fundamental separation of interfaces from implementation details, which is one of the primary benefits of the object-oriented paradigm.
- Functional Decomposition
  This AntiPattern is the output of experienced, nonobject-oriented developers who design and implement an application in an object-oriented language. The resulting code resembles a structural language (Pascal, FORTRAN) in class structure. It can be incredibly complex as smart procedural developers devise very "clever" ways to replicate their time-tested methods in an object-oriented architecture.
- Poltergeists
  Poltergeists are classes with very limited roles and effective life cycles. They often start processes for other objects. The refactored solution includes a reallocation of responsibilities to longer-lived objects that eliminate the Poltergeists.
- Boat Anchor
  A Boat Anchor is a piece of software or hardware that serves no useful purpose on the current project. Often, the Boat Anchor is a costly acquisition, which makes the purchase even more ironic.
- Golden Hammer
  A Golden Hammer is a familiar technology or concept applied obsessively to many software problems. The solution involves expanding the knowledge of developers through education, training, and book study groups to expose developers to alternative technologies and approaches.
- Dead End
  A Dead End is reached by modifying a reusable component if the modified component is no longer maintained and supported by the supplier. When these modifications are made, the

support burden transfers to the application system developers and maintainers. Improvements in the reusable component are not easily integrated, and support problems can be blamed upon the modification.

- Spaghetti Code
  Ad hoc software structure makes it difficult to extend and optimize code. Frequent code refactoring can improve software structure, support software maintenance, and enable iterative development.
- Input Kludge
  Software that fails straightforward behavioral tests may be an example of an input kludge, which occurs when ad hoc algorithms are employed for handling program input.
- Walking through a Minefield
  Using today's software technology is analogous to walking through a high-tech mine field. Numerous bugs are found in released software products; in fact, experts estimate that original source code contains two to five bugs per line of code.
- Cut-and-Paste Programming
  Code reused by copying source statements leads to significant maintenance problems. Alternative forms of reuse, including black-box reuse, reduce maintenance issues by having common source code, testing, and documentation.
- Mushroom Management
  In some architecture and management circles, there is an explicit policy to keep system developers isolated from the system's end users. Requirements are passed second-hand through intermediaries, including architects, managers, or requirements analysts.

SPAGHETTI CODE

- **AntiPattern Name:** Spaghetti Code
- **Most Applicable Scale:** Application
- **Refactored Solution Name:** Software Refactoring, Code Cleanup
- **Refactored Solution Type:** Software
- **Root Causes:** Ignorance, Sloth
- **Unbalanced Forces:** Management of Complexity, Change
- **Anecdotal Evidence:** "Ugh! What a mess!" "You *do* realize that the language supports more than one function, right?" "It's easier to rewrite this code than to attempt to modify it." "Software engineers don't write spaghetti code." "The quality of your software structure is an investment for future modification and extension."

Background

The Spaghetti Code AntiPattern is the classic and most famous AntiPattern; it has existed in one form or another since the invention of programming languages. Nonobject-oriented languages appear to be more susceptible to this AntiPattern, but it is fairly common among developers who have yet to fully master the advanced concepts underlying object orientation.

General Form

Spaghetti Code appears as a program or system that contains very little software structure. Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity, even to the original developer, if he or she is away from the software for any length of time. If developed using an object-oriented language, the software may include a small number of objects that contain methods with very large implementations that invoke a single, multistage process flow.

Furthermore, the object methods are invoked in a very predictable manner, and there is a negligible degree of dynamic interaction between the objects in the system. The system is very difficult to maintain and extend, and there is no opportunity to reuse the objects and modules in other similar systems.

Symptoms And Consequences

- After code mining, only parts of object and methods seem suitable for reuse. Mining Spaghetti Code can often be a poor return on investment; this should be taken into account before a decision to mine is made.
- Methods are very process-oriented; frequently, in fact, objects are named as processes.
- The flow of execution is dictated by object implementation, not by the clients of the objects.
- Minimal relationships exist between objects.
- Many object methods have no parameters, and utilize class or global variables for processing.
- The pattern of use of objects is very predictable.
- Code is difficult to reuse, and when it is, it is often through cloning. In many cases, however, code is never considered for reuse.
- Object-oriented talent from industry is difficult to retain.
- Benefits of object orientation are lost; inheritance is not used to extend the system;

polymorphism is not used.

- Follow-on maintenance efforts contribute to the problem.
- Software quickly reaches a point of diminishing returns; the effort involved in maintaining an existing code base is greater than the cost of developing a new solution from the ground up.

Typical Causes

- Inexperience with object-oriented design technologies.
- No mentoring in place; ineffective code reviews.
- No design prior to implementation.
- Frequently the result of developers working in isolation.

Known Exceptions

The Spaghetti Code AntiPattern is reasonably acceptable when the interfaces are coherent and only the implementation is spaghetti. This is somewhat like wrapping a nonobject-oriented piece of code. If the lifetime of the component is short and cleanly isolated from the rest of the system, then some amount of poor code may be tolerable.

The reality of the software industry is that software concerns usually are subservient to business concerns, and, on occasion, business success is contingent on delivering a software product as rapidly as possible. If the domain is not familiar to the software architects and developers, it may be better to develop products to gain an understanding of the domain with the intention of designing products with an improved architecture at some later date.

Refactored Solution

Software refactoring (or code cleanup) is an essential part of software development [Opdyke 92]. Seventy percent or more of software cost is due to extensions, so it is critical to maintain a coherent software structure that supports extension. When the structure becomes compromised through supporting unanticipated requirements, the ability of the code to support extensions becomes limited, and eventually, nonexistent. Unfortunately, the term“code cleanup” does not appeal to pointy−haired managers, so it may be best to discuss this issue using an alternative term such as “software investment.” After all, in a very real sense, code cleanup is the maintenance of software investment. Well−structured code will have a longer life cycle and be better able to support changes in the business and underlying technology.

Ideally, code cleanup should be a natural part of the development process. As each feature (or group of features) is added to the code, code cleanup should follow what restores or improves the code structure. This can occur on an hourly or daily basis, depending on the frequency of the addition of new features.

Code cleanup also supports performance enhancement. Typically, performance optimization follows the 90/10 rule, where only 10 percent of the code needs modification in order to achieve 90 percent of the optimal performance. For single−subsystem or application programming, performance optimization often involves compromises to code structure. The first goal is to achieve a satisfactory structure; the second is to determine by measurement where the performance−critical code exists; the third is to carefully introduce necessary structure compromises to enhance performance. It is sometimes necessary to reverse the performance enhancement changes in software to provide for essential system extensions. Such areas merit additional documentation, in order to preserve the software structure in future releases.

The best way to resolve the Spaghetti Code AntiPattern is through prevention; that is, to think, then develop a plan of action before writing. If, however, the code base has already degenerated to the point that it is unmaintainable, and if reengineering the software is not an option, there are still steps that can be taken to avoid compounding the problem. First, in the maintenance process, whenever new features are added to the Spaghetti Code code base, do not modify the Spaghetti Code simply by adding code in a similar style to minimally meet the new requirement. Instead, always spend time refactoring the existing software into a more maintainable form. Refactoring the software includes performing the following operations on the existing code:

1. Gain abstract access to member variables of a class using accessor functions. Write new and refactored code to use the accessor functions.
2. Convert a code segment into a function that can be reused in future maintenance and refactoring efforts. It is vital to resist implementing the Cut−and−Paste AntiPattern (discussed next). Instead, use the Cut−and−Paste refactored solution to repair prior implementations of the Cut−and−Paste AntiPattern.
3. Reorder function arguments to achieve greater consistency throughout the code base. Even consistently bad Spaghetti Code is easier to maintain than inconsistent Spaghetti Code.
4. Remove portions of the code that may become, or are already, inaccessible. Repeated failure to identify and remove obsolete portions of code is one of the major contributors to the Lava Flow AntiPattern.
5. Rename classes, functions, or data types to conform to an enterprise or industry standard and/or maintainable entity. Most software tools provide support for global renaming.

In short, commit to actively refactoring and improving Spaghetti Code to as great an extent as resources allow whenever the code base needs to be modified. It's extremely useful to apply unit and system testing tools and applications to ascertain that refactoring does not immediately introduce any new defects into the code base. Empirical evidence suggests that the benefits of refactoring the software greatly outweigh the risk that the extra modifications may generate new defects.

If prevention of Spaghetti Code is an option, or if you have the luxury of fully engineering a Spaghetti Code application, the following preventative measures may be taken:

1. Insist on a proper object−oriented analysis process to create the domain model, regardless of how well the domain is understood. It is crucial that any moderate−or large−size project develop a domain model as the basis of design and development. If the domain is fully understood to the point that a domain model is not needed, counter with "If that's true, then the time to develop one would be negligible." If it actually is, then politely admit you were mistaken. Otherwise, the time that it takes justifies how badly it was needed.
2. After developing a domain model that explains the system requirements and the range of variability that must be addressed, develop a separate design model. Though it is valid to use the domain model as a starting point for design, the domain model must be maintained as such in order to retain useful information that would otherwise be lost if permitted to evolve directly into a design model. The purpose of the design model is to extract the commonality between domain objects and abstract in order to clarify the necessary objects and relationships in the system. Properly performed, it establishes the bounds for software implementation. Implementation should be performed only in order to satisfy system requirements, either explicitly indicated by the domain model or anticipated by the system architect or senior developers.
3. In the development of the design model, it is important to ensure that objects are

decomposed to a level where they are fully understood by developers. It is the developers, not the designers, who must believe the software modules are easy to implement.

4. Once a first pass has been made at both the domain and design model, begin implementation based upon the plan established by the design. The design does not have to be complete; the goal is that the implementation of software components should always be according to some predefined plan. Once development begins, proceed to incrementally examine other parts of the domain model and design other parts of the system. Over time, the domain model and the design model will be refined to accommodate discoveries in the requirements gathering, design decisions, and to cope with implementation issues. Again, Spaghetti Code is far less likely to occur if there is an overall software process in which the requirements and design are specified in advance of the implementation, instead of occurring concurrently.

Example

This is a frequent problem demonstrated by people new to object−oriented development, who map system requirements directly to functions, using objects as a place to group related functions. Each function contains an entire process flow that completely implements a particular task. For example, the code segment that follows contains functions such as `initMenus()`, `getConnection()`, and `executeQuery()`, which completely execute the specified operation. Each object method contains a single process flow that performs all of the steps in sequence needed to perform the task. The object retains little or no state information between successive invocations; rather, the class variables are temporary storage locations to handle intermediate results of a single process flow.