AntiPatterns

**What Is an AntiPattern?**

AntiPatterns, like their design pattern counterparts, define an industry vocabulary for the common defective processes and implementations within organizations. A higher-level vocabulary simplifies communication between software practitioners and enables concise description of higher-level concepts.

**An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences. The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context.**

AntiPatterns provide real-world experience in recognizing recurring problems in the software industry and provide a detailed remedy for the most common predicaments. AntiPatterns highlight the most common problems that face the software industry and provide the tools to enable you to recognize these problems and to determine their underlying causes.

Furthermore, AntiPatterns present a detailed plan for reversing these underlying causes and implementing productive solutions. AntiPatterns effectively describe the measures that can be taken at several levels to improve the developing of applications, the designing of software systems, and the effective management of software projects.

**Why AntiPatterns?**

AntiPatterns are useful in several ways:

* They provide a common vocabulary for known dysfunctional software designs and solutions, as every AntiPattern has a short and descriptive name like *The Blob*, *Poltergeist* or *Golden Hammer*.
* They help detecting problems in the code, the architecture and the management of software projects.
* They describe both, preventive measures as well as *refactored solutions*, which can save software projects in trouble.

**Classification of AntiPatterns**

* Software Development AntiPatterns

A key goal of development AntiPatterns is to describe useful forms of 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.

* Software Architecture AntiPatterns

Architecture AntiPatterns focus on the system-level and enterprise-level structure of applications and components. Although the engineering discipline of software architecture is relatively immature, what has been determined repeatedly by software research and experience is the overarching importance of architecture in software development.

* Software Project Management AntiPatterns

In the modern engineering profession, more than half of the job involves human communication and resolving people issues. The management AntiPatterns identify some of the key scenarios in which these issues are destructive to software processes.

**Root causes of AntiPatterns**

Root Causes, also referred to as the ”seven deadly sins”, are software development mistakes which lead to negative consequences like cost overruns or the cancellation of projects.

The following Root Causes can be identiﬁed:

* **Haste**

Hasteleads to compromises in software quality. Due to tight schedules and deadlines, important activities such as unit testing and documenting source code are often neglected in favor of ”getting things done”.

* **Apathy**

Apathy is even worse, in that it’s understood by the developers that something needs to be improved, but they are unwilling to actually do it. This attitude of not caring about solving known problems can have very negative consequences for the software project.

* **Narrow-Mindedness**

Narrow-Mindedness is the basic resistance of developers against proven and working solutions. For example, they might refuse to learn and use a unit-testing framework, which would help to reduce the number of bugs in the code.

* **Sloth**

Sloth is characterized by decisions which always favor the most simple ”solution” or answer to a problem. The consequence is improper and obscure software design, which ultimately forces the developers to perform system discovery (trying to ﬁnd out how the software works) instead of writing or improving code.

* **Avarice or greed**

Avarice or greed can produce wrong design decisions. Especially architectural avarice can result in very complex and thus hard to maintain software. Proper abstractions are missing due to modelling excessive details of the system.

* **Ignorance**

Ignorance is the absence of motivation to understand things. This ”intellectual sloth” usually leads to long-term problems such as massive implementation dependencies, which make extending the software very hard.

* **Pride**

Pride is also known as the ”not-invented-here syndrome”. Failure to reuse existing software packages, components or libraries consumes more time and money for (unnecessary) designing, implementing, debugging, testing and documenting the new software components.

**Primal Forces**

Primal Forcesare pervasive, horizontal software design forces. They are – by definition – applicable across multiple problem domains.

* **Management of Functionality** must ensure that the software fulfills the requirements of the end-user.
* **Management of Performance** pays tribute to the fact that the system must not only meet functional requirements, but also performance requirements.
* **Management of Complexity** is another important force. Especially large software projects can only be handled by using sufficient and effective abstractions, otherwise very complex and almost unmanageable systems evolve.
* **Management of Change** is equally important. A lot of thought has to be put in the decision about *which* components of the system are designed to be adaptable and *how*.
* **Management of IT Resources** is mostly a task of the upper-level management of a company. Often, a very complex and heterogeneous hardware- and software landscape exists in large-scale companies, which needs to be handled appropriately. The same is true for the human resources of the company – skilled developers are the foundation of the success of the business. Time and money should be invested in training and education of employees.
* **Management of Technology Transfer** is concerned with the use and transfer of technologies and IT trends within the company as well as in relation with external enterprises and organizations.

Software Development AntiPatterns

Good software structure is essential for system extension and maintenance. Software development is a chaotic activity, therefore the implemented structure of systems tends to stray from the planned structure as determined by architecture, analysis, and design.

Software refactoring is an effective approach for improving software structure.  
The resulting structure does not have to resemble the original planned structure.

The structure changes because programmers learn constraints and approaches that alter the context of the coded solutions. When used properly, refactoring is a natural activity in the programming process.

For example, the solution for the Spaghetti Code AntiPattern defines a software development process that incorporates refactoring. Refactoring is strongly recommended prior to performance optimization. Optimizations often involve compromises to program structure. Ideally, optimizations affect only small portions of a program. Prior refactoring helps partition optimized code from the majority of the software.  
  
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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/walking-through-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**](http://sourcemaking.com/antipatterns/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**](http://sourcemaking.com/antipatterns/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.

**Golden Hammer**

**AntiPattern Name:** Golden Hammer

**Also Known As:** Old Yeller, Head-in-the sand

**Most Applicable Scale:** Application

**Refactored Solution Name:** Expand your horizons

**Refactored Solution Type:** Process

**Root Causes:** Ignorance, Pride, Narrow-Mindedness

**Unbalanced Forces:** Management of Technology Transfer

**Anecdotal Evidence:**

“I have a hammer and everything else is a nail.”

“Our database is our architecture.”

“Maybe we shouldn’t have used Excel macros for this job after all.”

Background

This is one of the most common AntiPatterns in the industry. Frequently, a vendor, specifically a database vendor, will advocate using its growing product suite as a solution to most of the needs of an organization. Given the initial expense of adopting a specific database solution, such a vendor often provides extensions to the technology that are proven to work well with its deployed products at greatly reduced prices.

General Form

A software development team has gained a high level of competence in a particular solution or vendor product, referred to here as the Golden Hammer. As a result, every new product or development effort is viewed as something that is best solved with it. In many cases, the Golden Hammer is a mismatch for the problem, but minimal effort is devoted to exploring alternative solutions.

This AntiPattern results in the misapplication of a favored tool or concept. Developers and managers are comfortable with an existing approach and unwilling to learn and apply one that is better suited. This is typified by the common “our database is our architecture” mind-set, particularly common in the world’s banking community.

Frequently, an advocate will propose the Golden Hammer and its associated product suite as a solution to most of the needs of an organization. Given the initial expense of adopting a specific solution, Golden Hammer advocates will argue that future extensions to the technology, which are designed to work with their existing products, will minimize risk and cost.

Symptoms And Consequences

Identical tools and products are used for wide array of conceptually diverse products.

Solutions have inferior performance, scalability, and so on when compared to other solutions in the industry.

System architecture is best described by a particular product, application suite, or vendor tool set.

Developers debate system requirements with system analysts and end users, frequently advocating requirements that are easily accommodated by a particular tool and steering them away from areas where the adopted tool is insufficient.

Developers become isolated from the industry. They demonstrate a lack of knowledge and experience with alternative approaches.

Requirements are not fully met, in an attempt to leverage existing investment.

Existing products dictate design and system architecture.

New development relies heavily on a specific vendor product or technology.

Typical Causes

Several successes have used a particular approach.

Large investment has been made in training and/or gaining experience in a product or technology.

Group is isolated from industry, other companies.

Reliance on proprietary product features that aren’t readily available in other industry products.

“Corncob” proposing the solution (see Corncob AntiPattern).

Known Exceptions

The Golden Hammer AntiPattern sometimes works:

If the product that defines the architectural constraints is the intended strategic solution for the long term; for example, using an Oracle database for persistent storage and wrapped stored procedures for secure access to data.

If the product is part of a vendor suite that provides for most of the software needs.

Refactored Solution

This solution involves a philosophical aspect as well as a change in the development process. Philosophically, an organization needs to develop a commitment to an exploration of new technologies.

Without such a commitment, the lurking danger of overreliance on a specific technology or vendor tool set exists. This solution requires a two-pronged approach: A greater commitment by management in the professional development of their developers, along with a development strategy that requires explicit software boundaries to enable technology migration.

Software systems need to be designed and developed with well-defined boundaries that facilitate the replaceability of individual software components. A component should insulate the system from proprietary features in its implementation.

If the system is developed using explicit software boundaries, the interfaces that make up the boundaries become points at which the software used in the implementation may be replaced with a new implementation, without affecting the other components in the system. An industry standard, such as the OMG IDL specification, is an invaluable tool for incorporating rigid software boundaries between components.

In addition, software developers need to be up to date on technology trends, both within the organization’s domain and in the software industry at large. This can be accomplished through several activities that encourage the interchange of technical ideas. For example, developers can establish groups to discuss technical developments (design patterns, emerging standards, new products) that may impact the organization in the future.

They can also form book study clubs to track and discuss new publications that describe innovative approaches to software development. In practice, we have found the book study club paradigm to be a very effective way to exchange ideas and new approaches.

Even without full management buyin, developers can establish informal networks of technology-minded people to investigate and track new technologies and solutions. Industry conferences are also a great way to contact people and vendors and stay informed as to where the industries are headed and what new solutions are available to developers.

On the management side, another useful step is to adopt a commitment to open systems and architectures. Without it, developers often acquire the attitude that achieving short-term results by any means necessary is acceptable. Though this may be desirable in the short term, future results become problematic because rather than building upon a solid foundation of past successes, effort is expended reworking past software to conform to new challenges.

Flexible, reusable code requires an investment in its initial development, otherwise long-term benefits will not be achieved Also, the danger of overreliance on a specific technology or vendor tool set is a potential risk in the product or project development. In-house research programs that develop proof-of-concept prototypes are effective for testing the feasibility of incorporating less risky open technologies into a development effort.

Another management-level way of eliminating or avoiding the Golden Hammer AntiPattern is to encourage the hiring of people from different areas and from different backgrounds. Teams benefit from having a broader experience base to draw upon in developing solutions. Hiring a database team whose members all have experience in using the same database product greatly limits the likely solution space, in comparison to a similar team whose experience encompasses a wide range of database technology solutions.

Finally, management must actively invest in the professional development of software developers, as well as reward developers who take initiative in improving their own work.

Variations

A common variation of Golden Hammer occurs when a developer uses a favorite software concept obsessively. For example, some developers learn one or two of the GoF patterns and apply them to all phases of software analysis, design, and implementation.

Discussions about intent or purpose are insufficient to sway them from recognizing the applicability of the design pattern’s structure and force-fitting its use throughout the entire development process. Education and mentoring is required to help people become aware of other available approaches to software system construction.

Example

A common example of the Golden Hammer AntiPattern is a database-centric environment with no additional architecture except that which is provided by the database vendor. In such an environment, the use of a particular database is assumed even before object-oriented analysis has begun. As such, the software life cycle frequently begins with the creation of an entity-relationship (E-R) diagram that is produced as a requirements document with the customer.

This is frequently destructive, because the E-R diagram ultimately is used to specify the database requirements; and detailing the structure of a subsystem before understanding and modeling the system presumes that the impact of the actual customer requirements on the system design is minimal.

Requirements gathering should enable system developers to understand the user needs to the extent that the external behavior of the solution system is understood by the user as a black box Conceivably, many systems are built to satisfy user requirements without utilizing a database at all. However, with the Golden Hammer AntiPattern in force, such possibilities are discounted up front, leading to every problem incorporating a database element.

Over time, the organization may develop several database-centric products that could have been implemented as independent systems. The database evolves into the basis for interconnectivity between applications, and it manages distribution and shared access to data. In addition, many implementation problems are addressed through using database proprietary features that commit future migrations to parallel the development of a technology of the implementation database.

At some point, it may be necessary to interoperate with systems that either do not share the same database-centric architecture or are implemented using a different database that may not permit unrestricted access to their information. Suddenly, development becomes extremely expensive as unique, special-purpose connections are built to “bridge” between unique systems. If, however, some thought is given to the problem before the situation gets too far out of hand, a common framework can be developed, where products chosen for particular areas are selected based on standard interface specifications, such as CORBA, DCOM, or TCP/IP.

Another example is an insurance company with several stovepipe legacy systems that decided in its move to client/server that Microsoft Access should be the key part of the solution for persistence. The entire front end of the call-center system was architected around an early version of this product. Thereafter, the system’s future was fully constrained by the development path of the database product because of a bad architecture decision. Needless to say, the system lasted less than six months.

Related Solutions

*Lava Flow.*

This AntiPattern results when the Golden Hammer AntiPattern is applied over the course of several years and many projects. Typically, older sections based on earlier versions of the Golden Hammer are delegated to remote, seldom-used parts of the overall application. Developers become reluctant to modify these sections, which build up over time and add to the overall size of the application while implementing functions that are seldom, if ever, used by the customer.

*Vendor Lock-In.* Vendor lock-in is when developers actively receive vendor support and encouragement in applying the Golden Hammer AntiPattern. A software project is actively committed to relying upon a single vendor’s approach in designing and implementing an object-oriented system.

**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 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.

**Kill Spaghetti Code AntiPattern through prevention**

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:

Gain abstract access to member variables of a class using accessor functions. Write new and refactored code to use the accessor functions.

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. Instead, use the Cut-and-Paste refactored solution to repair prior implementations of the Cut-and-Paste AntiPattern.

Reorder function arguments to achieve greater consistency throughout the code base. Even consistently bad Spaghetti Code is easier to maintain than inconsistent Spaghetti Code.

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.

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.

**Other prevention methods**

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:

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.

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.

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.

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.

**Related Solutions**

Analysis Paralysis. This AntiPattern is the result of taking the solution to its logical extreme. Rather than developing code ad hoc without a design to guide the overall structure of the code, Analysis Paralysis produces a detailed design without ever reaching a point at which implementation can commence.

Lava Flow. This AntiPattern frequently contains several examples of Spaghetti Code that discourage the refactoring of the existing code base. With Lava Flow, the code base had a logical purpose at some point in its life cycle, but portions became obsolescent, yet remained as part of the code base.

**Cut-And-Paste Programming**

**AntiPattern Name:** Cut-and-Paste Programming

**Also Known As:** Clipboard Coding, Software Cloning, Software Propagation

**Most Applicable Scale:** Application

**Refactored Solution Name:** Black Box Reuse

**Refactored Solution Type:** Software

**Root Causes:** Sloth

**Unbalanced Forces:** Management of Resources, Technology Transfer

**Anecdotal Evidence:**

“Hey, I thought you fixed that bug already, so why is it doing this again?” “Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!”

**Background**

Cut-and-Paste Programming is a very common, but degenerate form of software reuse which creates maintenance nightmares. It comes from the notion that it’s easier to modify existing software than program from scratch. This is usually true and represents good software instincts. However, the technique can be easily over used.

**General Form**

This AntiPattern is identified by the presence of several similar segments of code interspersed throughout the software project. Usually, the project contains many programmers who are learning how to develop software by following the examples of more experienced developers.

However, they are learning by modifying code that has been proven to work in similar situations, and potentially customizing it to support new data types or slightly customized behavior. This creates code duplication, which may have positive short-term consequences such as boosting line count metrics, which may be used in performance evaluations.

Furthermore, it’s easy to extend the code as the developer has full control over the code used in his or her application and can quickly meet short-term modifications to satisfy new requirements.

**Symptoms And Consequences**

The same software bug reoccurs throughout software despite many local fixes.

Lines of code increase without adding to overall productivity.

Code reviews and inspections are needlessly extended.

It becomes difficult to locate and fix all instances of a particular mistake.

Code is considered self-documenting.

Code can be reused with a minimum of effort.

This AntiPattern leads to excessive software maintenance costs.

Software defects are replicated through the system.

Reusable assets are not converted into an easily reusable and documented form.

Developers create multiple unique fixes for bugs with no method of resolving the variations into a standard fix.

Cut-and-Paste Programming form of reuse deceptively inflates the number of lines of code developed without the expected reduction in maintenance costs associated with other forms of reuse.

**Typical Causes**

It takes a great deal of effort to create reusable code, and organization emphasizes short-term payoff more than long-term investment.

The context or intent behind a software module is not preserved along with the code.

The organization does not advocate or reward reusable components, and development speed overshadows all other evaluation factors.

There is a lack of abstraction among developers, often accompanied by a poor understanding of inheritance, composition, and other development strategies.

The organization insists that code must be a perfect match to the new task to allow it to be reused. Code is duplicated to address perceived inadequacies in meeting what is thought to be a unique problem set.

Reusable components, once created, are not sufficiently documented or made readily available to developers.

A “not-invented-here” syndrome is in operation in the development environment.

There is a lack of forethought or forward thinking among the development teams.

Cut-and-Paste AntiPattern is likely to occur when people are unfamiliar with new technology or tools; as a result, they take a working example and modify it, adapting it to their specific needs.

**Known Exceptions**

The Cut-and-Paste Programming AntiPattern is acceptable when the sole aim is to get the code out of the door as quickly as possible. However, the price paid is one of increased maintenance.

**Refactored Solution**

Cloning frequently occurs in environments where white-box reuse is the predominant form of system extension. In white-box reuse, developers extend systems primarily though inheritance. Certainly, inheritance is an essential part of object-oriented development, but it has several drawbacks in large systems.

First, subclassing and extending an object requires some knowledge of how the object is implemented, such as the intended constraints and patterns of use indicated by the inherited base classes. Most object-oriented languages impose very few restrictions, types of extensions can be implemented in a derived class and lead to nonoptimal use of subclassing.

In addition, typically, white-box reuse is possible only at application compile time (for compiled languages), as all subclasses must be fully defined before an application is generated.

On the other hand, black-box reuse has a different set of advantages and limitations and is frequently a better option for object extension in moderate and large systems. With black-box reuse, an object is used as-is, through its specified interface. The client is not allowed to alter how the object interface is implemented.

The key benefit of black-box reuse is that, with the support of tools, such as interface definition languages, the implementation of an object can be made independent of the object’s interface. This enables a developer to take advantage of late binding by mapping an interface to a specific implementation at run time. Clients can be written to a static object interface yet benefit over time by more advanced services that support the identical object interface.

Of course, the drawback is that the supported services are limited to those supported by the same interface. Changes to the interface typically must be made at compile time, similar to interface or implementation changes in white-box reuse.

The distinction between white-box and black-box reuse mirrors the difference between object-oriented programming (OOP) and component-oriented programming (COP), where the white-box subclassing is the traditional signature of OOP and the dynamic late binding of interface to implementation is a staple in COP.

Restructuring software to reduce or eliminate cloning requires modifying code to emphasize black-box reuse of duplicated software segments. In the case where Cut-and-Paste Programming has been used extensively throughout the lifetime of a software project, the most effective method of recovering your investment is to refactor the code base into reusable libraries or components that focus on black-box reuse of functionality.

If performed as a single project, this refactoring process is typically difficult, long, and costly, and requires a strong system architect to oversee and execute the process and to mediate discussions on the merits and limitations of the various extended versions of software modules.

Effective refactoring to eliminate multiple versions involves three stages: code mining, refactoring, and configuration management. Code mining is the systematic identification of multiple versions of the same software segment. The refactoring process involves developing a standard version of the code segment and reinserting it into the code base.

Configuration management is a set of policies drawn up to aid in the prevention of future occurrences of Cut-and-Paste Programming. For the most part, this requires monitoring and detection policies (code inspections, reviews, validation), in addition to educational efforts. Management buy-in is essential to ensure funding and support throughout all three stages.

**Example**

There is one piece of code that we suspect has been cloned repeatedly throughout several organizations and probably is still cloned today. This piece of code has been observed several hundred times across dozens of organizations. It is a code file that implements a linked-list class without the use of templates or macros.

Instead, the data structure stored by the linked list is defined in a header file, so each linked list is customized to operate only on the specified data structure. Unfortunately, the original author of the code (rumor has it he was originally a LISP programmer) introduced a flaw in the linked-list code: It failed to free the memory of an item when it was deleted; instead, it just rearranged the pointers.

On occasion, this code has been modified to fix this defect; however, more often than not, it still exists. It’s clearly the same code set; the variable names, the instructions, and even the formatting is exactly the same in each and every case. Even the file is typically named <prefix>link.c, where the prefix is one or two letters that cryptically refer to the data structure managed by the list.

**Related Solutions**

Spaghetti Code often contains several instances of the Cut-and-Paste Programming AntiPattern. Because Spaghetti Code is not structured for easy component reuse, in many cases, Cut-and-Paste Programming is the only means available for reusing existing segments of code.

Of course, this leads to unnecessary code bloat and a maintenance nightmare, but empirical evidence suggests that Spaghetti Code without Cut-and-Paste Programming is typically an even worse mess than instances that make use of Cut-and-Paste Programming.

Cloning can be minimized in new development through the implementation of a software reuse process or organization Some degree of cloning is inevitable in large software development; however, when it occurs, there must be a formalized process for merging clones into a common baseline

**Functional Decomposition**

**AntiPattern Name:** Functional Decomposition

**Also Known As:** No Object-Oriented AntiPattern “No OO”

**Most Frequent Scale:** Application

**Refactored Solution Name:** Object-Oriented Reengineering

**Refactored Solution Type:** Process

**Root Causes:** Avarice, Greed, Sloth

**Unbalanced Forces:** Management of Complexity, Change

**Anecdotal Evidence:**

“This is our ‘main’ routine, here in the class called LISTENER.”

**Background**

Functional Decomposition is good in a procedural programming environment. It’s even useful for understanding the modular nature of a larger-scale application.

Unfortunately, it doesn’t translate directly into a class hierarchy, and this is where the problem begins. In defining this AntiPattern, the authors started with Michael Akroyd’s original thoughts on this topic. We have reformatted it to fit in with our template, and extended it somewhat with explanations and diagrams.

**General Form**

This AntiPattern is the result of experienced, nonobject-oriented developers who design and implement an application in an object-oriented language. When developers are comfortable with a “main” routine that calls numerous subroutines, they may tend to make every subroutine a class, ignoring class hierarchy altogether (and pretty much ignoring object orientation entirely).

The resulting code resembles a structural language such as Pascal or 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.



You will most likely encounter this AntiPattern in a C shop that has recently gone to C++, or has tried to incorporate CORBA interfaces, or has just implemented some kind of object tool that is supposed to help them. It’s usually cheaper in the long run to spend the money on object-oriented training or just hire new programmers who think in objects.

**Symptoms And Consequences**

Classes with “function” names such as Calculate\_Interest or Display\_Table may indicate the existence of this AntiPattern.

All class attributes are private and used only inside the class.

Classes with a single action such as a function.

An incredibly degenerate architecture that completely misses the point of object-oriented architecture.

Absolutely no leveraging of object-oriented principles such as inheritance and polymorphism. This can be extremely expensive to maintain (if it ever worked in the first place; but never underestimate the ingenuity of an old programmer who’s slowly losing the race to technology).

No way to clearly document (or even explain) how the system works. Class models make absolutely no sense.

No hope of ever obtaining software reuse.

Frustration and hopelessness on the part of testers.

**Typical Causes**

*Lack of object-oriented understanding.* The implementers didn’t “get it.” This is fairly common when developers switch from programming in a nonobject-oriented programming language to an object-oriented programming language. Because there are architecture, design, and implementation paradigm changes, object-orientation can take up to three years for a company to fully achieve.

*Lack of architecture enforcement.* When the implementers are clueless about object orientation, it doesn’t matter how well the architecture has been designed; they simply won’t understand what they’re doing. And without the right supervision, they will usually find a way to fudge something using the techniques they do know.

*Specified disaster.* Sometimes, those who generate specifications and requirements don’t necessarily have real experience with object-oriented systems. If the system they specify makes architectural commitments prior to requirements analysis, it can and often does lead to AntiPatterns such as Functional Decomposition.

**Known Exceptions**

The Functional Decomposition AntiPattern is fine when an object-oriented solution is not required. This exception can be extended to deal with solutions that are purely functional in nature but wrapped to provide an object-oriented interface to the implementation code.

**Refactored Solution**

If it is still possible to ascertain what the basic requirements are for the software, define an analysis model for the software, to explain the critical features of the software from the user’s point of view. This is essential for discovering the underlying motivation for many of the software constructs in a particular code base, which have been lost over time. For all of the steps in the Functional Decomposition AntiPattern solution, provide detailed documentation of the processes used as the basis for future maintenance efforts.

Next, formulate a design model that incorporates the essential pieces of the existing system. Do not focus on improving the model but on establishing a basis for explaining as much of the system as possible.

Ideally, the design model will justify, or at least rationalize, most of the software modules. Developing a design model for an existing code base is enlightening; it provides insight as to how the overall system fits together. It is reasonable to expect that several parts of the system exist for reasons no longer known and for which no reasonable speculation can be attempted.

For classes that fall outside of the design model, use the following guidelines:

If the class has a single method, try to better model it as part of an existing class. Frequently, classes designed as helper classes to another class are better off being combined into the base class they assist.

Attempt to combine several classes into a new class that satisfies a design objective. The goal is to consolidate the functionality of several types into a single class that captures a broader domain concept than the previous finer-grained classes. For example, rather than have classes to manage device access, to filter information to and from the devices, and to control the device, combine them into a single device controller object with methods that perform the activities previously spread out among several classes.

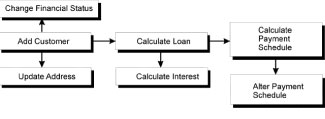
If the class does not contain state information of any kind, consider rewriting it as a function. Potentially, some parts of the system may be best modeled as functions that can be accessed throughout various parts of the system without restriction.

Examine the design and find similar subsystems. These are reuse candidates. As part of program maintenance, engage in refactoring of the code base to reuse code between similar subsystems (see the Spaghetti Code solution for a detailed description of software refactoring).

**Example**

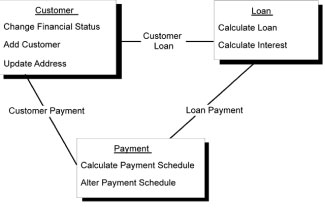
Functional Decomposition is based upon discrete functions for the purpose of data manipulation, for example, the use of Jackson Structured Programming. Functions are often methods within an object-oriented environment. The partitioning of functions is based upon a different paradigm, which leads to a different grouping of functions and associated data.

The simple example in figure below shows a functional version of a customer loan scenario:



* Adding a new customer.
* Updating a customer address.
* Calculating a loan to a customer.
* Calculating the interest on a loan.
* Calculating a payment schedule for a customer loan.
* Altering a payment schedule.

Next figure then shows the object-oriented view of a customer loan application. The previous functions map to object methods.



**Related Solutions**

If too much work has already been invested in a system plagued by Functional Decomposition, you may be able to salvage things by taking an approach similar to the alternative approach addressed in the Blob AntiPattern.

Instead of a bottom-up refactoring of the whole class hierarchy, you may be able to extend the “main routine” class to a “coordinator” class that manages all or most of the system’s functionality.

Function classes can then be “massaged” into quasi-object-oriented classes by combining them and beefing them up to carry out some of their own processing at the direction of the modified “coordinator” class. This process may result in a class hierarchy that is more workable

**Applicability To Other Viewpoints And Scales**

Both architectural and managerial viewpoints play key roles in either initial prevention or ongoing policing against the Functional Decomposition AntiPattern. If a correct object-oriented architecture was initially planned and the problem occurred in the development stages, then it is a management challenge to enforce the initial architecture.

Likewise, if the cause was a general lack of incorrect architecture initially, then it is still a management challenge to recognize this, put the brakes on, and get architectural help—the sooner the cheaper.

# Software Architecture AntiPatterns

Architecture AntiPatterns focus on the system-level and enterprise-level structure of applications and components. Although the engineering discipline of software architecture is relatively immature, what has been determined repeatedly by software research and experience is the overarching importance of architecture in software development:

1. Good architecture is a critical factor in the success of the system development.
2. Architecture-driven software development is the most effective approach to building systems. Architecture-driven approaches are superior to requirements-driven, document-driven, and methodology-driven approaches. Projects often succeed in spite of methodology, not because of it.

Software architecture is a subset of the overall system architecture, which includes all design and implementation aspects, including hardware and technology selection. Important principles of architecture include the following:

1. Architecture provides a view of the whole system. This distinguishes architecture from other analysis and design models that focus on parts of a system.
2. An effective way to model whole systems is through multiple viewpoints. The viewpoints correlate to various stakeholders and technical experts in the system-development process.

The following AntiPatterns focus on some common problems and mistakes in the creation, implementation, and management of architecture.

* [Autogenerated Stovepipe](http://sourcemaking.com/antipatterns/autogenerated-stovepipe)  
  This AntiPattern occurs when migrating an existing software system to a distributed infrastructure. An Autogenerated Stovepipe arises when converting the existing software interfaces to distributed interfaces. If the same design is used for distributed computing, a number of problems emerge.
* [Stovepipe Enterprise](http://sourcemaking.com/antipatterns/stovepipe-enterprise)  
  A Stovepipe System is characterized by a software structure that inhibits change. The refactored solution describes how to abstract subsystem and components to achieve an improved system structure. The Stovepipe Enterprise AntiPattern is characterized by a lack of coordination and planning across a set of systems.
* [Jumble](http://sourcemaking.com/antipatterns/jumble)  
  When horizontal and vertical design elements are intermixed, an unstable architecture results. The intermingling of horizontal and vertical design elements limits the reusability and robustness of the architecture and the system software components.
* [Stovepipe System](http://sourcemaking.com/antipatterns/stovepipe-system)  
  Subsystems are integrated in an ad hoc manner using multiple integration strategies and mechanisms, and all are integrated point to point. The integration approach for each pair of subsystems is not easily leveraged toward that of other subsystems. The Stovepipe System AntiPattern is the single-system analogy of Stovepipe Enterprise, and is concerned with how the subsystems are coordinated within a single system.
* [Cover Your Assets](http://sourcemaking.com/antipatterns/cover-your-assets)  
  Document-driven software processes often produce less-than-useful requirements and specifications because the authors evade making important decisions. In order to avoid making a mistake, the authors take a safer course and elaborate upon alternatives.
* [Vendor Lock-In](http://sourcemaking.com/antipatterns/vendor-lock-in)  
  Vendor Lock-In occurs in systems that are highly dependent upon proprietary architectures. The use of architectural isolation layers can provide independence from vendor-specific solutions.
* [Wolf Ticket](http://sourcemaking.com/antipatterns/wolf-ticket)  
  A Wolf Ticket is a product that claims openness and conformance to standards that have no enforceable meaning. The products are delivered with proprietary interfaces that may vary significantly from the published standard.
* [Architecture by Implication](http://sourcemaking.com/antipatterns/architecture-by-implication)  
  Management of risk in follow-on system development is often overlooked due to overconfidence and recent system successes. A general architecture approach that is tailored to each application system can help identify unique requirements and risk areas.
* [Warm Bodies](http://sourcemaking.com/antipatterns/warm-bodies)  
  Software projects are often staffed with programmers with widely varying skills and productivity levels. Many of these people may be assigned to meet staff size objectives (so-called “warm bodies”). Skilled programmers are essential to the success of a software project. So-called heroic programmers are exceptionally productive, but as few as 1 in 20 have this talent. They produce an order of magnitude more working software than an average programmer.
* [Design by Committee](http://sourcemaking.com/antipatterns/design-by-committee)  
  The classic AntiPattern from standards bodies, Design by Committee creates overly complex architectures that lack coherence. Clarification of architectural roles and improved process facilitation can refactor bad meeting processes into highly productive events.
* [Swiss Army Knife](http://sourcemaking.com/antipatterns/swiss-army-knife)  
  A Swiss Army Knife is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. In the attempt, he or she adds a large number of interface signatures in a futile attempt to meet all possible needs.
* [Reinvent the Wheel](http://sourcemaking.com/antipatterns/reinvent-the-wheel)  
  The pervasive lack of technology transfer between software projects leads to substantial reinvention. Design knowledge buried in legacy assets can be leveraged to reduce time-to-market, cost, and risk.
* [The Grand Old Duke of York](http://sourcemaking.com/antipatterns/the-grand-old-duke-of-york)  
  Egalitarian software processes often ignore people’s talents to the detriment of the project. Programming skill does not equate to skill in defining abstractions. There appear to be two distinct groups involved in software development: abstractionists and their counterparts the implementationists.

**Architecture By Implication**

* AntiPattern Name: Architecture by Implication
* Also Known As: Wherefore art thou architecture?
* Most Frequent Scale: System
* Refactored Solution Name: Goal Question Architecture
* Refactored Solution Type: Documentation
* Root Causes: Pride, Sloth
* Unbalanced Forces: Management of Complexity, Change, and Risk
* Anecdotal Evidence:

“We’ve done systems like this before!” “There is no risk; we know what we’re doing!”

**Background**

Dwight Eisenhower said that planning is essential, but plans are inconsequential. Another soldier said that no plans survive first contact with the enemy. The planning culture in modern management owes some credit to Robert McNamara, founder of the RAND Corporation.

In McNamara’s approach, plans are generated for speculative purposes, to investigate the potential benefits and consequences of different courses of action. Given the large number of unknowns in systems development, planning for IT systems must be more pragmatic and iterative.

One professional planner said that 20 percent of an engineer’s time should be devoted to planning. As we gain experience, our belief in this assertion increases. Productivity and efficiency can be greatly amplified when the work is well organized through planning.

The unfortunate consequence is that many organizations attempt to formalize too much of the planning. Planning is most effective when it is personally motivated and utilized. Time management experts teach that a key element of stress reduction is planning to balance life’s overall priorities. The form and use of time-management systems becomes increasingly personalized as the practice matures.

A group of CEOs from DoD Systems integration firms was formed to answer the question, “Wherefore art thou architecture?” The goal was to reflect on the changing nature of systems development, which has evolved into the reuse of existing legacy components and commercial software, and away from greenfield, custom code development (see the Reinvent the Wheel AntiPattern).

**General Form**

This AntiPattern is characterized by the lack of architecture specifications for a system under development. Usually, the architects responsible for the project have experience with previous system construction, and therefore assume that documentation is unnecessary.

This overconfidence leads to exacerbated risks in key areas that affect system success. Architecture definitions are often missing from one or more of these areas:

* Software architecture and specifications that include language use, library use, coding standards, memory management, and so forth.
* Hardware architecture that includes client and service configurations.
* Communications architecture that includes networking protocols and devices.
* Persistence architecture that includes databases and file-handling mechanisms.
* Application security architecture that includes thread models and trusted system base.
* Systems management architecture.

**Symptoms And Consequences**

* Lack of architecture planning and specification; insufficient definition of architecture for software, hardware, communications, persistence, security, and systems management.
* Hidden risks caused by scale, domain knowledge, technology, and complexity, all of which emerge as the project progresses.
* Impending project failure or unsuccessful system due to inadequate performance, excess complexity, misunderstood requirements, usability, and other system characteristics. For example, approximately 1 of 3 systems encounter serious performance problems during development and operations.
* Ignorance of new technologies.
* Absence of technical backup and contingency plans.

**Typical Causes**

* No risk management.
* Overconfidence of managers, architects, and/or developers.
* Reliance on previous experience, which may differ in critical areas.
* Implicit and unresolved architecture issues caused by gaps in systems engineering.

**Known Exceptions**

The Architecture by Implication AntiPattern is acceptable for a repeated solution, where there are minor differences in the code, such as installation scripts. This AntiPattern may also be useful in a new project domain as an exploratory effort to determine whether existing techniques are transferable into a new area.

**Refactored Solution**

The refactored solution to the Architecture by Implication AntiPattern entails an organized approach to systems architecture definition, and relies on multiple views of the system. Each view models the system from the perspective of a system stakeholder, who may be real or imaginary, individual or aggregate. Each stakeholder is responsible for a high-priority set of questions and issues, and each view represents the entire information system and answers these key questions and issues.

The views comprising a set of diagrams, tables, or specifications, are linked for consistency. Generally, a view is a lightweight specification. The purpose of the architecture documentation is to communicate architecture decisions and other issues resolutions. The documentation should be easy to understand and inexpensive to maintain.

That said, the only people who can define and implement a successful architecture are those who already fully understand it. Unfortunately, this is often not the case, as many projects adopt some new technology that is not well understood. Therefore, developing a good architecture from scratch is an iterative process and should be recognized as such.

An initial reference architecture should have strong strategies that can be implemented within the time period of the first product development. Thereafter, it will have to be incrementally refined by future versions of the reference architecture, and driven by new versions of the first product or new products.

The steps to define a system architecture using viewpoints are as follows :

Define the architecture goals. What must this architecture achieve? Which stakeholders, real and imaginary, must be satisfied with the design and implementation? What is the vision for the system? Where are we now and where are we going?

Define the questions. What are the specific questions that must be addressed to satisfy the stakeholder issues? Prioritize the questions to support view selection.

Select the views. Each view will represent a blueprint of the system architecture.

Analyze each view. Detail the architecture definition from each viewpoint. Create the system blueprints.

Integrate the blueprints. Verify that the views present a consistent architecture definition.

Trace views to needs. The views should address the known questions and issues to discover any gaps not addressed by the architecture specifications. Validate the architecture with respect to formal requirements. Prioritize the outstanding issues.

Iterate the blueprints. Refine the views until all questions, issues, and gaps are resolved. Utilize review processes to surface any remaining issues. If a significant number of unresolved issues remain, consider creating additional views.

Promote the architecture. Make an explicit effort to communicate the architecture to key stakeholders, particularly the system developers. Create lasting documents (such as a video tutorial) to provide valuable information throughout the development and maintenance life cycle.

Validate the implementation. The blueprints should represent an “as-built” design. Determine any deltas between the blueprints and the system implementation. Decide whether these differences should result in system modifications of updates to the blueprints. Upgrade the documentation for consistency.

We refer to this method as the goal-question architecture (GQA), analogous to the goal-question metric approach in software metrics

**Variations**

A number of approaches consider the system architecture using viewpoints; in some, the viewpoints are predefined. Most of these approaches are open-ended, in that one can select additional viewpoints as described.

The Reference Model for Open Distributed Process (RM-ODP) is a popular, useful standard for distributed architectures. RM-ODP defines five standard viewpoints: enterprise, information, computational, engineering, and technology It also defines a useful set of transparency properties for distributed infrastructure through the engineering viewpoint.

Another approach, the Zachman Framework, analyzes system architectures from the perspectives of data, function, and network Within each perspective are multiple levels of abstraction, corresponding to the planning needs of various groups of stakeholders. Enterprise Architecture Planning is an approach based upon the Zachman Framework for large-scale systems Neither of these approaches is tailored to object-oriented systems development.

A third approach, the Command, Communication, Control, Computer, Intelligence, Surveillance, and Reconnaissance Architecture Framework (C4ISR-AF), is used to define various command and control system architectures. A version of C4ISR-AF is used for other types of civilian systems. This approach has been very beneficial in enabling communications between architects across disparate domains

A fourth, the 4 + 1 Model View, is a viewpoint-based architecture approach supported by software engineering tools, such as Rational Rose The viewpoints include logical, use-case, process, implementation, and deployment. Finally, GQA is a generalization of the underlying method used in several of these architecture approaches

**Example**

A common but bad practice is object-oriented modeling without defining the viewpoint. In most modeling approaches, there is a blurring of the viewpoints. Many of the modeling constructs contain implementation detail, and the default practice is to intermingle implementation and specification constructs.

Three fundamental viewpoints are: conceptual, specification, and implementation The conceptual viewpoint defines the system from the perspective of the user. This is typically referred to as an analysis model. The distinction between what is automated and what is not is usually not represented in the model; rather, the model is drawn so that a user can explain and defend it to his or her peers.

The specification viewpoint concerns only interfaces. ISO IDL is one important notation that is strictly limited to defining interface information and excludes implementation specifics. The separation of interfaces from implementations enables the realization of many important object technology benefits, such as reuse, system extension, variation, substitutability, polymorphism, and distributed object computing. The final viewpoint, implementation, is best represented by the source code.

Complex implementation structures are beneficially augmented with object-oriented design models to help current and future developers and maintainers understand the code.

Another example of the Architecture by Implication AntiPattern is the following, where the key stakeholders did not have collective experience in what was built. The project was intended to deliver a Microsoft Distributed Common Object Model (DCOM)-based solution to extract legacy mainframe data, filter it based on business rule, and display it on Web pages.

However, the manager was a good software engineer with no distributed object technology (DOT) experience and the architect was a “dyed-in-the-wool” CORBA addict who helped the OMG derive its Object Management Architecture. To compound the problem, the project had few DCOM-aware staff; less than 10 percent.

In addition, the architecture and subsequent design were based on the OMA view of the DOT world, rather than DCOM. This led to an attempt to deliver CORBA services under a DCOM architecture. The resulting product suffered from a set of components that had no DOT consistency and were poor performers. Also, SIs found it very difficult to use, due to lack of a standardized approach. Finally, it failed in the marketplace.

**Related Solutions**

Architecture by Implication AntiPattern differs from the Stovepipe Systems AntiPattern in scope; the latter focuses on deficiencies in computational architecture. In particular, it identifies how improper abstraction of subsystem APIs leads to brittle architecture solutions. In contrast, the Architecture by Implication AntiPattern involves planning gaps constituted of multiple architecture viewpoints.

**Applicability To Other Viewpoints And Scales**

This AntiPattern significantly increases risk for managers, who defer important decisions until failures occur; often, it is too late to recover. Developers suffer from a lack of guidance for system implementation.

They are given de facto responsibility for key architectural decisions for which they may not have the necessary architectural perspective. Systemwide consequences of interface design decisions should be considered; in particular: system adaptability, consistent interface abstractions, metadata availability, and management of complexity.

Another important result of this AntiPattern is the deferment of resource allocation. The essential tools and technology components may not be available when needed due to lack of planning.

# Design By Committee

* **AntiPattern Name:** Design by Committee
* **Also Known As:** Gold Plating, Standards Disease, Make Everybody Happy, Political Party
* **Most Frequent Scale:** Global
* **Refactored Solution Name:** Meeting Facilitation
* **Refactored Solution Type:** Process
* **Root Causes:** Pride, Avarice
* **Unbalanced Forces:** Management of Functionality, Complexity, and Resources
* **Anecdotal Evidence:**

“A camel is a horse designed by a committee.”

“Too many cooks spoil the broth.”

### Background

Object orientation is often described as a two-generation technology. Data-centric object analysis is characteristic of the first generation, and design patterns are characteristic of the second.

The first generation espoused a philosophy that “objects are things you can touch.” A consequence of this is that virtually all designs are uniquely vertical. In the first generation, people believed a number of assumptions that were unsubstantiated by practice.



One of these was that project teams should be egalitarian; in other words, that everyone should have an equal say and that decisions are democratic. This leads to Design by Committee. Given that only a small number of object developers can define good abstractions, the majority rule invariably leads to a diffusion of abstraction and excess complexity.

### General Form

A complex software design is the product of a committee process. It has so many features and variations that it is infeasible for any group of developers to realize the specifications in a reasonable time frame.

Even if the designs were possible, it would not be possible to test the full design due to excessive complexity, ambiguities, overconstraint, and other specification defects. The design would lack conceptual clarity because so many people contributed to it and extended it during its creation.

### Symptoms And Consequences

* The design documentation is overly complex, unreadable, incoherent, or is excessively defective.
* The design documentation is voluminous (hundreds or thousands of pages).
* Convergence and stability are missing from the requirements and the design.
* The committee meetings resemble bull sessions during which substantive issues are rarely discussed and progress is painstakingly slow. People talk and work serially; that is, there is a single thread of discussion, and most people are unproductive most of the time.
* The environment is politically changed, and few decisions and actions can be undertaken outside of the meeting; and the meeting process precludes reaching timely decisions.
* There is no prioritization of the design features, and no ready answer to the questions: Which features are essential? Which features will be implemented in the initial delivery?
* Architects and developers have conflicting interpretations of the design.
* Design development becomes significantly over-budget and overdue.
* It becomes necessary to employ specialists to interpret, develop, and manage the specifications. In other words, dealing with each specification that was designed by committee becomes a full-time job.

### Typical Causes

* No designated project architect.
* A degenerate or ineffective software process.
* Bad meeting processes, marked by lack of facilitation or ineffective facilitation. The meetings are bull sessions; the loudest people win and the level of discourse is the lowest common denominator of sophistication.
* Gold plating—that is, features are added to the specification based on proprietary interests. This can happen for many reasons: marketability, the existence of proprietary technologies already containing these features, or speculative placement of features in the specification for potential future work.
* The attempt to make everybody happy, to satisfy all of the committee participants by incorporating their ideas. Unfortunately, it’s impossible to accept every idea and still manage complexity.
* Design and editing is attempted during meetings with more than five people.
* Explicit priorities and a software-value system are undetermined
* Concerns are not separated, and reference models are not used.

### Known Exceptions

There are few exceptions to the Design by Committee AntiPattern, and they occur when the committee is small: approximately 6 to 10 people; more than that and consensus becomes unlikely; fewer than six people and the breadth of understanding and experience become insufficient.

Also, committees should often be tiger teams, a small group of “experts” in a particular problem domain, organized for the duration of the solution for a specific problem or issue.

### Refactored Solution

The essence of the solution to Design by Committee is to reform the meeting process. It’s fair to say that most people are accustomed to enduring bad meetings, most of the time.

Thus, even simple alterations in the meeting process can yield substantial productivity improvements. With improved productivity, there is an opportunity for enhanced quality and more sophisticated solutions. Typical gains for software optimizations are less than an order of magnitude (2 to 10 times). Meeting productivity gains are much more dramatic, with several orders of magnitude typical (100 times), and we have seen productivity gains over 100,000:1.

First, most meeting rooms do not have a clock on the wall, particularly in hotel facilities. Time awareness is essential to meeting progress. Participants should be coached to manage the time allotted efficiently; participants should start their comments with a “25-words-or-less” summary, and add details only if requested. Posting the meeting goals, an agenda, and a clock where they are visible to all participants can improve meetings dramatically.

Second, in all meetings it’s important that group members answer two questions: “Why are we here?” and “What outcomes do we want?” When no meeting plan is prepared, it is particularly important that the group start with these two questions, and then work on generating the desired outcomes.

Another important reform is to assign explicit roles in the software process: owner, facilitator, architect, developers, testers, and domain experts. The owner is the manager responsible for the software development. He or she makes strategic decisions about the overall software process, and invites and organizes the participants.

At the beginning of the meeting, the process owner sets the goals and establishes expectations regarding the handling of the outcomes. For example, the decisions made at the meeting may be regarded simply as advice or be implemented exactly as discussed.

The facilitator is in charge of running the meeting. He or she is responsible for the process; other participants are responsible for technical content. The facilitator turns to the process owner if there are any key decisions to be made regarding the process.

The architect is the senior technical lead for the software project. He or she controls the editing of the architecture documentation, and may be in charge of key, system-level boundaries, such as the subsystem application program interfaces.

Each developer is typically responsible for a single subsystem and unit testing. Testers are responsible for monitoring specification quality and value-added testing such as integration, portability, and stress tests. The domain experts input key requirements to the process, but may not be involved in all aspects of development.

There are three categories of meeting processes: divergent, convergent, and information sharing. In a divergent process, ideas are generated for later utilization. In a convergent process, a selection or decision is made that represents a consensus. Information sharing can involve presentation, teaching, writing, and review.

The number of people involved in each meeting process is managed by the facilitator. Creative processes that require writing, highlighting, or drawing should be limited to breakout teams of five people or fewer.

Groups larger than five are less effective at making progress in creative tasks, although they are successful at reviewing and integrating results after a creative process. Highly productive meetings involve many parallel processes, and frequent iterations of breakout and review groups. Encouraging people to make a paradigm shift between single-threaded discussion and parallel work is a key challenge for the facilitator.

The primary purpose of most meetings is problem solving. A general problem-solving approach begins with a convergent process: The problem is defined and scoped for the group to resolve.

A divergent process is used to identify alternative solutions. Information sharing may be needed to explore the details and consequences of selected alternatives. Finally, a convergent process is used to select among the options.

One highly effective meeting process is called Spitwads It’s a general-purpose procedure that we have used on many occasions with excellent results.

1. Ask the question. The facilitator poses a question for the group to brainstorm. The question is written down on a viewgraph or flipchart) to avoid misunderstandings. The group is asked whether there are any suggested changes to the question before the brainstorming starts. Typical questions are: “What are ways that we can improve the performance of the system?” and “What is the most important requirement that has not been addressed?”
2. Write silently. The participants write responses on identical slips of paper. Each response is written on a separate piece of paper, and is limited to short phrases.
3. Toss spitwads. As the participants complete each idea, the facilitator instructs them to wad up the paper and toss it across the room into a receptacle—a cardboard box or a basket works best. This is done basketball style, and the facilitator encourages the participants to have fun with this exercise.
4. Conduct spitwad roll call. The “spitwads” are distributed randomly back to the participants, who one at a time read the ideas out loud to be recorded on a flipchart. Two flipchart recorders can be used simultaneously to speed up the process. The flipcharts are posted on the wall for all participants to see.
5. Reach common understanding. The ideas on the flipcharts are numbered. The facilitator then asks the group if there are any ideas that they do not understand. If so, the members of the group are encouraged to offer definitions. If an idea cannot be defined, it is safe to eliminate it.
6. Eliminate duplicates. The facilitator asks the group to identify any ideas that are duplicates or that should be combined. Participants identify the ideas by number to suggest changes. If there is an objection, the changes are overruled. (This is a common facilitation approach for editing: If there are objections to a proposed change, then the change is not accepted.)
7. Prioritize. The group is directed to silently select the best ideas on the list by number. They can pick more than one. The facilitator walks through the list tabulating votes (raised hands, no discussion).
8. Discuss. The exercise is complete. The group discusses the highest-priority selections and suggests what the follow-up actions will be.

### Variations

The Railroad (also known as Rubber Stamp) AntiPattern is a variation of Design by Committee, whereby a political coalition dominates the process and forces the adoption of designs with serious defects. The Railroad is often motivated by mutual business interests of many members of the committee. By adopting incomplete and defective specifications, details of the technology can be hidden effectively in the software.

In this way, software from a coalition developer becomes the de facto standard, as opposed to the written specifications. Some developers outside the coalition will even attempt to implement the “misfeatures,” resulting in wasted time and money.

### Example

Two classic examples of Design by Committee come from the domain of software standardization: the Structured Query Language (SQL) and Common Object Request Broker Architecture (CORBA).

#### Sql

The Structured Query Language (SQL) became an international standard in 1989. The original, SQL89, was a small document—115 pages—that represented an efficient, minimal design for the technology Virtually all relational database products implemented the full specification.

In 1992, the second version of SQL was standardized with significant extensions that resulted in a 580-page document. The SQL92 specification was implemented with a unique dialect in every product; few products implemented the entire specification. The next version of SQL, called SQL3, may well be thousands of pages in length.

The standards committee responsible for the design is adding a smorgasbord of new features that extend the concept well beyond the original intent. Some of the new features include object-orientation extensions, geospatial extensions, and temporal-logic extensions.

It’s unlikely that any product will ever fully implement SQL3, nor is it likely that any two products will implement the same subset in a portable manner. In this classic Design by Committee, the SQL standard has become a dumping ground for advanced database features.

An interesting solution to the problems of SQL convergence is presented by the technologies: Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). Each defined a standard application program interface for database access based on dynamic queries, query statements that are submitted and parsed at run time.

Because ODBC and JDBC define the query interface and query language for the clients, they provide isolation from product-specific database features. Clients can access multiple database products transparently. ODBC became an industry standard through the SQL Access Group (SAG), a software industry consortium.

Microsoft developed the ODBC specification independently of SAG, and then proposed it to the group. The specification was adopted rapidly by SAG and became a de facto industry standard. Vendors that promote proprietary solutions have had difficulty replacing this highly effective technology, which is universally supported by database vendors and database tool developers.

**Related Solutions, Patterns, and AntiPatterns**

Kyle Brown posted a version of the Design by Committee AntiPattern on the Portland Patterns Repository site That pattern uses a different template, which focuses entirely on describing the problematic solution and not upon the refactored solution. In contrast, this book includes a refactored solution with each AntiPattern, noting that Socrates was put to death for exposing society’s contradictions without offering any constructive suggestions.

**Applicability to Other Viewpoints and Scales**

The impact of the Design by Committee AntiPattern on developers is that they are expected to implement a highly complex and ambiguous design, which is a stressful situation. They may find the need to quietly subvert the demands of the committee with more realistic design approaches.

Managers suffer from this AntiPattern through a dramatic increase in project risk caused by the excessive complexity. Correspondingly, the schedule and budget of the project are likely to increase dramatically as the consequences of the design are discovered in the laboratory.

At the system level, it might be reasonable to deliver a system based on a Design by Committee specification, if no variations (multiple configurations) are required and the proposed implementation schedule is extended by 30 percent or more. Most developers can manage only a few variations, in platforms, databases, and feature sets.

# Reinvent the Wheel

* **AntiPattern Name:** Reinvent the Wheel
* **Also Known As:** Design in a Vacuum, Greenfield System
* **Most Frequent Scale:** System
* **Refactored Solution Name:** Architecture Mining
* **Refactored Solution Type:** Process
* **Root Causes:** Pride, Ignorance
* **Unbalanced Forces:** Management of Change, Technology Transfer
* **Anecdotal Evidence:**

“Our problem is unique.” Software developers generally have minimal knowledge of each other’s code. Even widely used software packages available in source code rarely have more than one experienced developer for each program.

Virtually all systems development is done in isolation of projects and systems with overlapping functionality. Reuse is rare in most software organizations. In a recent study of more than 32 object-oriented software projects, the researchers found virtually no evidence of successful reuse

### Background

Software and design reuse are significantly different paradigms. Software reuse involves the creation of a library of reusable components, the retrieval of those components, and the integration of the components with a software system. The typical result is a modest amount of reuse around the periphery of the system and additional software development to integrate the components.

Design reuse involves the reuse of architecture and software interfaces in multiple application systems. It requires the identification of horizontal components with uses across multiple application systems.

Design reuse also supports software reuse of the horizontal components without additional development for integration, thus it is a much more effective approach, in that a larger portion of the software system can be leveraged from reusable components.

The term greenfield (in Greenfield System, an alias of Reinvent the Wheel) originates from the construction industry. It refers to a new construction site where there are no legacy buildings to introduce constraints on the new building’s architecture.

### General Form

Custom software systems are built from the ground up, even though several systems with overlapping functionality exist. The software process assumes “greenfield” (build from scratch) development of a single system. Because top-down analysis and design lead to new architectures and custom software, software reuse is limited and interoperability is accommodated after the fact.

Most current software methods assume that developers are building custom software from scratch, and that they are building a single system in isolation. These are called greenfield system assumptions.

Greenfield systems inevitably become stovepipes that lack potential for interoperability, extension, and reuse. Greenfield assumptions are mismatched to most real-world software development problems, where legacy systems exist, and interoperation with them is an important requirement for many new systems. Greenfield assumptions also ignore significant reusable software assets in the form of Internet freeware and commercially available software.

### Symptoms And Consequences

* Closed system architectures—architectures and software—that are designed for one system at a time without provision for reuse and interoperability.
* Replication of commercial software functions.
* Immature and unstable architectures and requirements.
* Inadequate support for change management and interoperability.
* Extended development cycles involving failed and dead-end prototypes before the architecture is mature enough to support long-term system development and maintenance.
* Poor management of risks and costs, leading to schedule and budget overruns.
* Inability to deliver the desired features to the end user; extensive effort to replicate the functionality already operational in existing systems.

### Typical Causes

* No communication and technology transfer between software development projects.
* Absence of an explicit architecture process that includes architecture mining and domain engineering.
* Assumption of greenfield development; in other words, the process assumes that the system will be built from scratch.
* Lack of enterprise management of the computational viewpoint, leading to unique software interfaces in each system.

### Known Exceptions

The Reinvent the Wheel AntiPattern is suitable for a research environment and in general software development to minimize coordination costs where developers with different skills work at logistically remote sites.

### Refactored Solution

Architecture mining is a way to quickly create successful object-oriented architectures that are robust, product-independent, reusable, and extensible. Most object-oriented design approaches assume that design information is invented as the process proceeds.

In a top-down process, design information is generated from requirements, which may be represented as use cases and object-oriented analysis models. Requirements-driven architecture design is calledarchitecture farming. In a spiral process, design information is invented during each iteration. As the spiral process proceeds, architects derive new design information as they learn more about the application problem.

It’s fair to say that these approaches reinvent much of their design information.

Precursor designs exist for most information systems applications and problems. These designs are in the form of legacy systems, commercial products, standards, prototypes, and design patterns.

Experience proves it is not difficult to identify a half-dozen or more precursor designs for any given application problem. Valuable information is buried in preexisting designs, information that enabled earlier architects to build useful systems. Extracting this information for use in object-oriented architectures is called architecture mining.

Mining may be applicable at the application level for certain complex design problems. In some cases, it may be less expensive and risky to exploit existing expertise than to create new code without exposure to preexisting solutions. Mining is applicable at enterprise levels, but less so at global levels, given the reduced access to information resources.

Mining is a bottom-up design approach, incorporating design knowledge from working implementations. Mining can incorporate design input from top-down design processes, too, so that there can be both top-down traceability and bottom-up realism.

Before mining starts, it is necessary to identify a set of representative technologies that are relevant to the design problem. Technology identification can be done by various means, such as searching literature, interviewing experts, attending technical conferences, and surfing the Net. All available resources should be pursued.

The first mining step is to model each representative technology, to produce specifications of relevant software interfaces. We recommend using OMG IDL as the interface notation because it is concise and free from implementation detail.

OMG IDL is also a good design notation for the target architecture because it is language-independent, platform-neutral, and distribution-transparent. Modeling everything in the same notation creates a good basis for design comparison and trade-off.

While modeling, it is important to describe the as-built system, not the intended or desired design. Frequently, relevant design information is not documented as software interfaces. For example, some of the sought-after functionality may be accessible only through the user interface. Other key design lessons may be undocumented, and it is useful to capture this information, too.

In the second mining step, the designs are generalized to create a common interface specification. This step entails more art than science, as the goal is to create an initial “strawman” specification for the target architecture interfaces.

It is usually not sufficient to generate a lowest-common denominator design from the representative technology. The generalized interfaces should resemble a best-of-breed solution that captures the common functionality, as well as some unique aspects inspired by particular systems.

Unique aspects should be included when they create valuable features in the target architecture or represent areas of known system evolution. A robust assortment of representative technologies will contain indicators of likely areas of target system evolution.

At this point, it is appropriate to factor in the top-down design information as one of the inputs. Top-down information is usually at a much higher level of abstraction than bottom-up information. Reconciliation of these differences involves some important architecture trade-offs.

The final step in the mining process is to refine the design. Refinements can be driven by the architect’s judgment, informal walkthroughs, review processes, new requirements, or additional mining studies.

### Variations

Within an organization, software reuse is difficult to achieve. In a survey of several dozen object-oriented application projects, Goldberg and Rubin found no significant reuse Even if successful, the cost benefits of internal reuse are usually less than 15 percent Industry experience indicates that the primary role of internal reuse is as a investment in software for resale. Where large volumes make the potential savings significant, reuse can shorten time-to-market and support product customization.

On the other hand, we claim that reuse is prevalent, but in different forms: reuse of commercially available software and reuse of freeware. Because of larger user bases, commercial software and freeware often have significantly higher quality than custom-developed software. For infrastructure components upon which much application software depends, this enhanced quality can be critical to project success.

Commercial software and freeware can reduce maintenance costs when the software is used without modification and can be readily replaced with upgraded versions.

### Related Solutions

The impact upon management of complexity of architecture mining and the generalization to common interfaces is analyzed in Mowbray 95. Architecture mining is a recurring solution that addresses many of the problems caused by Stovepipe Systems. It is also one of the approaches for defining domain-specific component architectures.

### Applicability To Other Viewpoints And Scales

The Reinvent the Wheel AntiPattern puts managers at higher risk, in the form of increased time-to-market and a lower degree of functionality than that expected of the end users. Potential savings from reuse range from 15 to 75 percent of development cost, reduction of 2 to 5 times in time-to-market, and reduction of defects from 5 to 10 times