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.

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.

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**
  Procedural-style design leads to one object with a lion’ s share of the responsibilities, while most other objects only hold data or execute simple processes. The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes.
- **Continuous Obsolescence**
  Technology is changing so rapidly that developers often have trouble keeping up with current versions of software and finding combinations of product releases that work together. Given that every commercial product line evolves through new releases, the situation is becoming more difficult for developers to cope with. Finding compatible releases of products that successfully interoperate is even harder.

- **Lava Flow**
  Dead code and forgotten design information is frozen in an ever-changing design. This is analogous to a Lava Flow with hardening globules of rocky material. The refactored solution includes a configuration management process that eliminates dead code and evolves or refactors design toward increasing quality.
- **Ambiguous Viewpoint**
  Object-oriented analysis and design (OOA&D) models are often presented without clarifying the viewpoint represented by the model. By default, OOA&D models denote an implementation viewpoint that is potentially the least useful. Mixed viewpoints don’t allow the fundamental separation of interfaces from implementation details, which is one of the primary benefits of the object-oriented paradigm.
- **Functional Decomposition**
  This AntiPattern is the output of experienced, nonobject-oriented developers who design and implement an application in an object-oriented language. The resulting code resembles a structural language (Pascal, FORTRAN) in class structure. It can be incredibly complex as smart procedural developers devise very “clever” ways to replicate their time-tested methods in an object-oriented architecture.
- **Poltergeists**
  Poltergeists are classes with very limited roles and effective life cycles. They often start processes for other objects. The refactored solution includes a reallocation of responsibilities to longer-lived objects that eliminate the Poltergeists.
- **Boat Anchor**
  A Boat Anchor is a piece of software or hardware that serves no useful purpose on the current project. Often, the Boat Anchor is a costly acquisition, which makes the purchase even more ironic.
- **Golden Hammer**
  A Golden Hammer is a familiar technology or concept applied obsessively to many software problems. The solution involves expanding the knowledge of developers through education, training, and book study groups to expose developers to alternative technologies and approaches.
- **Dead End**
  A Dead End is reached by modifying a reusable component if the modified component is no longer maintained and supported by the supplier. When these modifications are made, the support burden transfers to the application system developers and maintainers. Improvements in the reusable component are not easily integrated, and support problems can be blamed upon the modification.
- **Spaghetti Code**
  Ad hoc software structure makes it difficult to extend and optimize code. Frequent code refactoring can improve software structure, support software maintenance, and enable iterative development.
- **Input Kludge**
  Software that fails straightforward behavioral tests may be an example of an input

kludge, which occurs when ad hoc algorithms are employed for handling program input.

- **Walking through a Minefield**
  Using today’ s software technology is analogous to walking through a high-tech mine field. Numerous bugs are found in released software products; in fact, experts estimate that original source code contains two to five bugs per line of code.
- **Cut-and-Paste Programming**
  Code reused by copying source statements leads to significant maintenance problems. Alternative forms of reuse, including black-box reuse, reduce maintenance issues by having common source code, testing, and documentation.
- **Mushroom Management**
  In some architecture and management circles, there is an explicit policy to keep system developers isolated from the system’ s end users. Requirements are passed second-hand through intermediaries, including architects, managers, or requirements analysts.

The Blob

- **AntiPattern Name:** The Blob
- **Also Known As:** Winnebago and The God Class
- **Most Frequent Scale:** Application
- **Refactored Solution Name:** Refactoring of Responsibilities
- **Refactored Solution Type:** Software
- **Root Causes:** Sloth, Haste
- **Unbalanced Forces:** Management of Functionality, Performance, Complexity
- **Anecdotal Evidence:** "This is the class that is really the *heart* of our architecture."

Background

Do you remember the original black-and-white movie The Blob? Perhaps you saw only the recent remake. In either case, the story line was almost the same: A drip-sized, jellylike alien life form from outer space somehow makes it to Earth.

Whenever the jelly thing eats (usually unsuspecting earthlings), it grows. Meanwhile, incredulous earthlings panic and ignore the one crazy scientist who knows what's happening. Many more people are eaten before they come to their senses. Eventually, the Blob grows so large that it threatens to wipe out the entire planet.

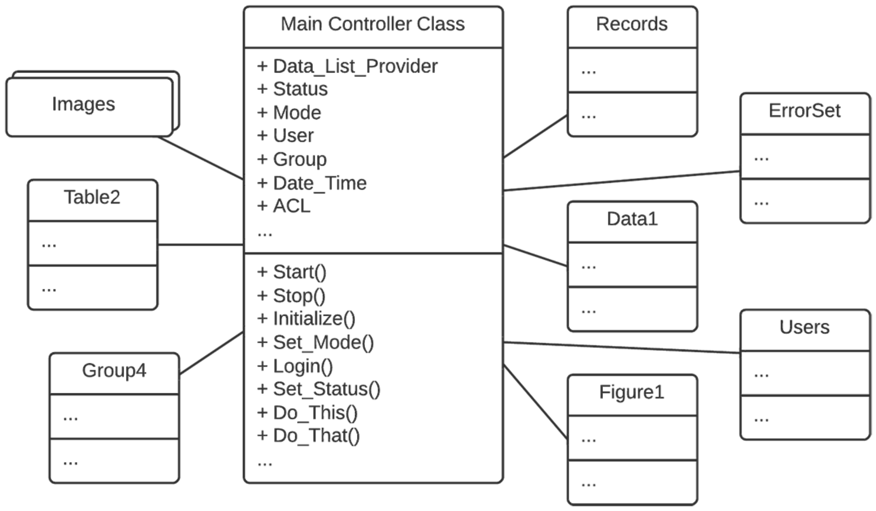
The movie is a good analogy for the Blob AntiPattern, which has been known to consume entire object-oriented architectures.

General Form

The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes. The key problem here is that the majority of the responsibilities are allocated to a single class.

In general, the Blob is a procedural design even though it may be represented using object notations and implemented in object-oriented languages. A procedural design separates process from data, whereas an object-oriented design merges process and data models, along with partitions.

The Blob contains the majority of the process, and the other objects contain the data. Architectures with the Blob have separated process from data; in other words, they are procedural-style rather than object-oriented architectures.



The Blob can be the result of inappropriate requirements allocation. For example, the Blob may be a software module that is given responsibilities that overlap most other parts of the system for system control or system management.

The Blob is also frequently a result of iterative development where proof-of-concept code evolves over time into a prototype, and eventually, a production system. This is often exacerbated by the use of primarily GUI-centric programming languages, such as Visual Basic, that allow a simple form to evolve its functionality, and therefore purpose, during incremental development or prototyping.

The allocation of responsibilities is not repartitioned during system evolution, so that one module becomes predominant. The Blob is often accompanied by unnecessary code, making it hard to differentiate between the useful functionality of the Blob Class and no-longer-used code (see the Lava Flow AntiPattern).

Symptoms And Consequences

- Single class with a large number of attributes, operations, or both. A class with 60 or more attributes and operations usually indicates the presence of the Blob
- A disparate collection of unrelated attributes and operations encapsulated in a single class. An overall lack of cohesiveness of the attributes and operations is typical of the Blob.
- A single controller class with associated simple, data-object classes.
- An absence of object-oriented design. A program main loop inside the Blob class associated with relatively passive data objects. The single controller class often nearly encapsulates the applications entire functionality, much like a procedural main program.
- A migrated legacy design that has not been properly refactored into an object-oriented architecture.

- The Blob compromises the inherent advantages of an object-oriented design. For example, The Blob limits the ability to modify the system without affecting the functionality of other encapsulated objects. Modifications to the Blob affect the extensive software within the Blob's encapsulation. Modifications to other objects in the system are also likely to have impact on the Blob's software.
- The Blob Class is typically too complex for reuse and testing. It may be inefficient, or introduce excessive complexity to reuse the Blob for subsets of its functionality.
- The Blob Class may be expensive to load into memory, using excessive resources, even for simple operations.

Typical Causes

- *Lack of an object-oriented architecture.* The designers may not have an adequate understanding of object-oriented principles. Alternatively, the team may lack appropriate abstraction skills.
- *Lack of (any) architecture.* The absence of definition of the system components, their interactions, and the specific use of the selected programming languages. This allows programs to evolve in an ad hoc fashion because the programming languages are used for other than their intended purposes.
- *Lack of architecture enforcement.* Sometimes this AntiPattern grows accidentally, even after a reasonable architecture was planned. This may be the result of inadequate architectural review as development takes place. This is especially prevalent with development teams new to object orientation.
- *Too limited intervention.* In iterative projects, developers tend to add little pieces of functionality to existing working classes, rather than add new classes, or revise the class hierarchy for more effective allocation of responsibilities.
- *Specified disaster.* Sometimes the Blob results from the way requirements are specified. If the requirements dictate a procedural solution, then architectural commitments may be made during requirements analysis that are difficult to change. Defining system architecture as part of requirements analysis is usually inappropriate, and often leads to the Blob AntiPattern, or worse.

Known Exceptions

The Blob AntiPattern is acceptable when wrapping legacy systems. There is no software partitioning required, just a final layer of code to make the legacy system more accessible.

Refactored Solution

As with most of the AntiPatterns in this section, the solution involves a form of refactoring. The key is to move behavior away from the Blob. It may be appropriate to reallocate behavior to some of the encapsulated data objects in a way that makes these objects more capable and the Blob less complex. The method for refactoring responsibilities is described as follows:

1. Identify or categorize related attributes and operations according to contracts. These contracts should be cohesive in that they all directly relate to a common focus, behavior, or function within the overall system. For example, a library system architecture diagram is represented with a potential Blob class called LIBRARY.
   In the example shown in figure 1, the LIBRARY class encapsulates the sum total of all the system's functionality. Therefore, the first step is to identify cohesive sets of operations and attributes that represent contracts. In this case, we could gather operations related to catalog management, like Sort_Catalog and Search_Catalog.
   We could also identify all operations and attributes related to individual items, such as Print_Item, Delete_Item, and so on.

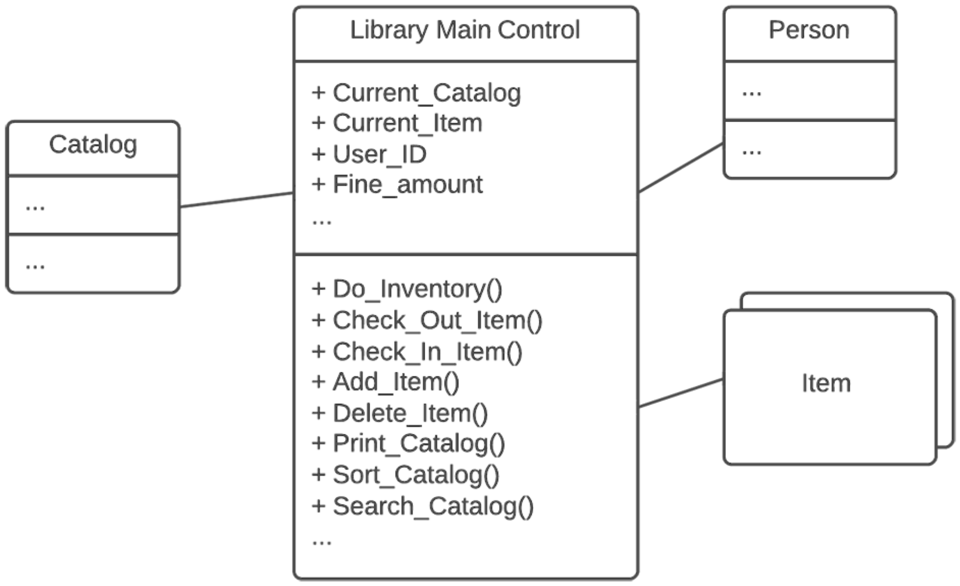


Figure 1

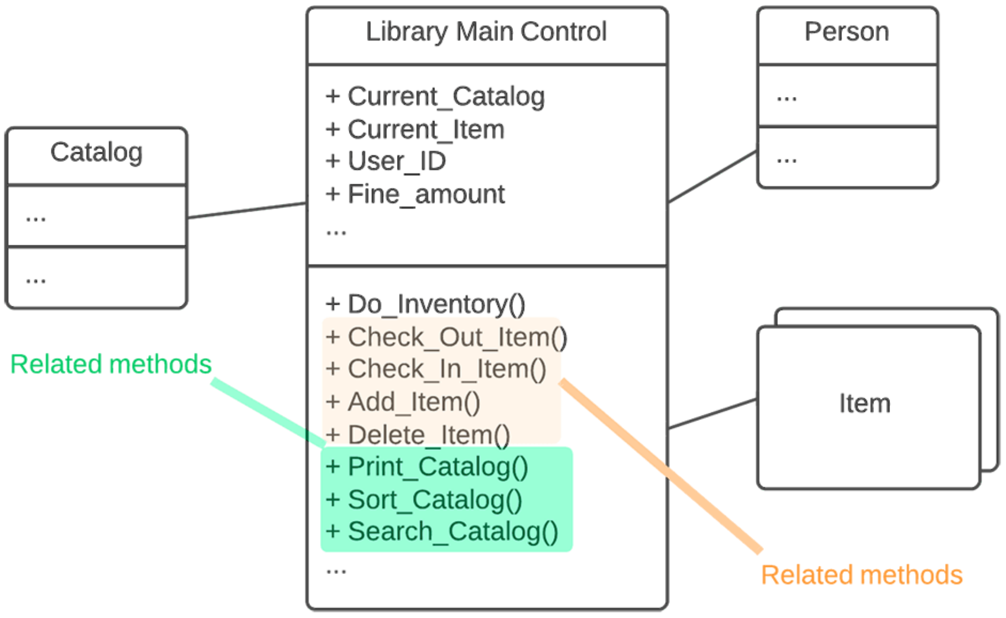


Figure 2

2. The second step is to look for "natural homes" for these contract-based collections of functionality and then migrate them there. In this example, we gather operations related to catalogs and migrate them from the LIBRARY class and move them to the CATALOG class.
   We do the same with operations and attributes related to items, moving them to the ITEM class. This both simplifies the LIBRARY class and makes the ITEM and CATALOG classes more than simple encapsulated data tables. The result is a better object-oriented design.

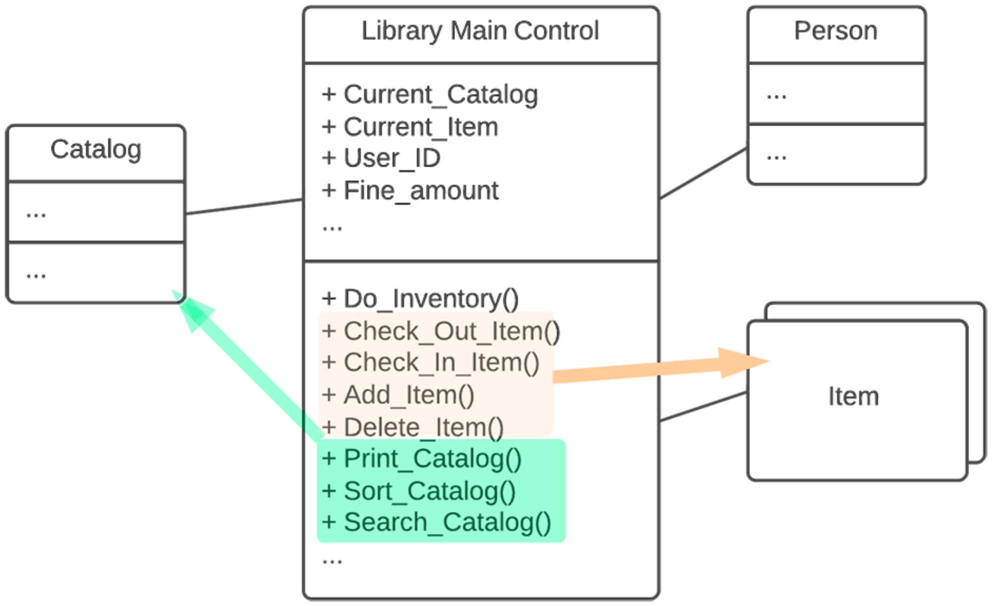


Figure 3

3. The third step is to remove all "far-coupled," or redundant, indirect associations. In the example, the ITEM class is initially far-coupled to the LIBRARY class in that each item really belongs to a CATALOG, which in turn belongs to a LIBRARY.
4. Next, where appropriate, we migrate associates to derived classes to a common base class. In the example, once the far-coupling has been removed between the LIBRARY and ITEM classes, we need to migrate ITEMs to CATALOGs, as shown in figure 4.

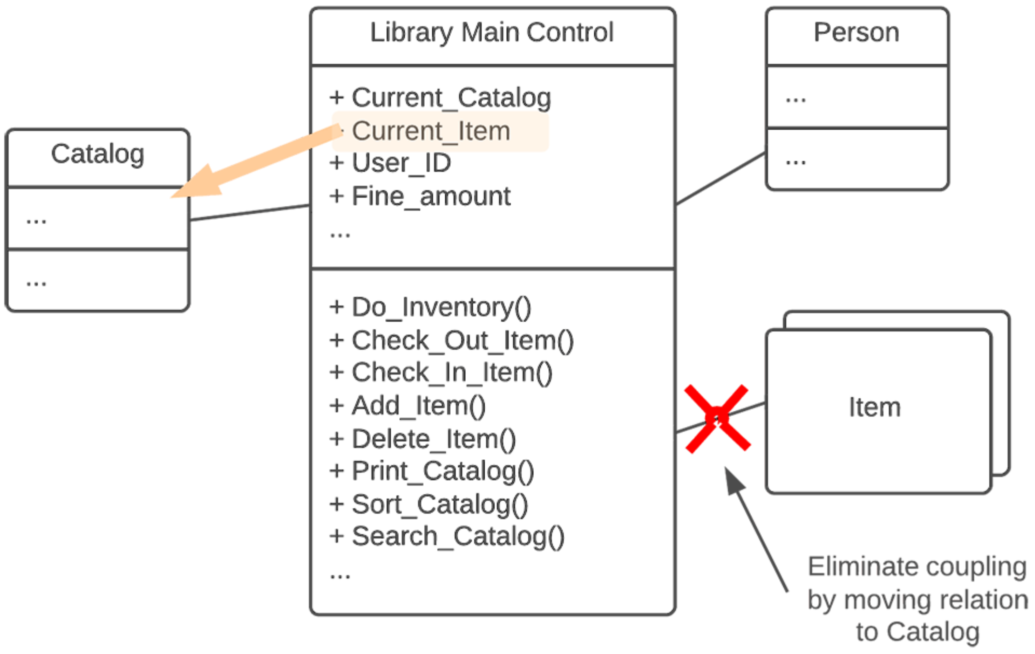


Figure 4

5. Finally, we remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments.
In our example, a Check_Out_Item or a Search_For_Item would be a transient process, and could be moved into a separate transient class with local attributes that establish the specific location or search criteria for a specific instance of a check-out or search.

Variations

Sometimes, with a system composed of the Blob class and its supporting data objects, too much work has been invested to enable a refactoring of the class architecture. An alternative approach may be available that provides an "80%" solution.

Instead of a bottom-up refactoring of the entire class hierarchy, it may be possible to reduce the Blob class from a controller to a coordinator class. The original Blob class manages the system's functionality; the data classes are extended with some of their own processing.

The data classes operate at the direction of the modified coordinator class. This process may allow the retention of the original class hierarchy, except for the migrations of processing functionality from the Blob class to some of the encapsulated data classes.

Riel identifies two major forms of the Blob AntiPattern. He calls these two forms God Classes: *Behavioral Form* and *Data Form*

The Behavioral Form is an object that contains a centralized process that interacts with most other parts of the system. The Data Form is an object that contains shared data used by most other objects in the system. Riel introduces a number of object-oriented heuristics for detect ing and refactoring God Class designs.

Applicability To Other Viewpoints And Scales

Both architectural and managerial viewpoints play key roles in the initial prevention of the Blob AntiPattern. Avoidance of the Blob may require ongoing policing of the architecture to assure adequate distribution of responsibilities.

It is through an architectural viewpoint that an emerging Blob is recognized. With a mature object-oriented analysis and design process, and an alert manager who understands the design, developers can prevent the cultivation of a Blob.

The most important factor is that, in most cases, it's much less expensive to create appropriate design than to rework design after implementation. Up-front investment in good architecture and team education can ensure a project against the Blob and most other AntiPatterns.

Ask any insurance salesperson, and he or she may tell you that most insurance is purchased *after* it was needed by people who are poorer but wiser.

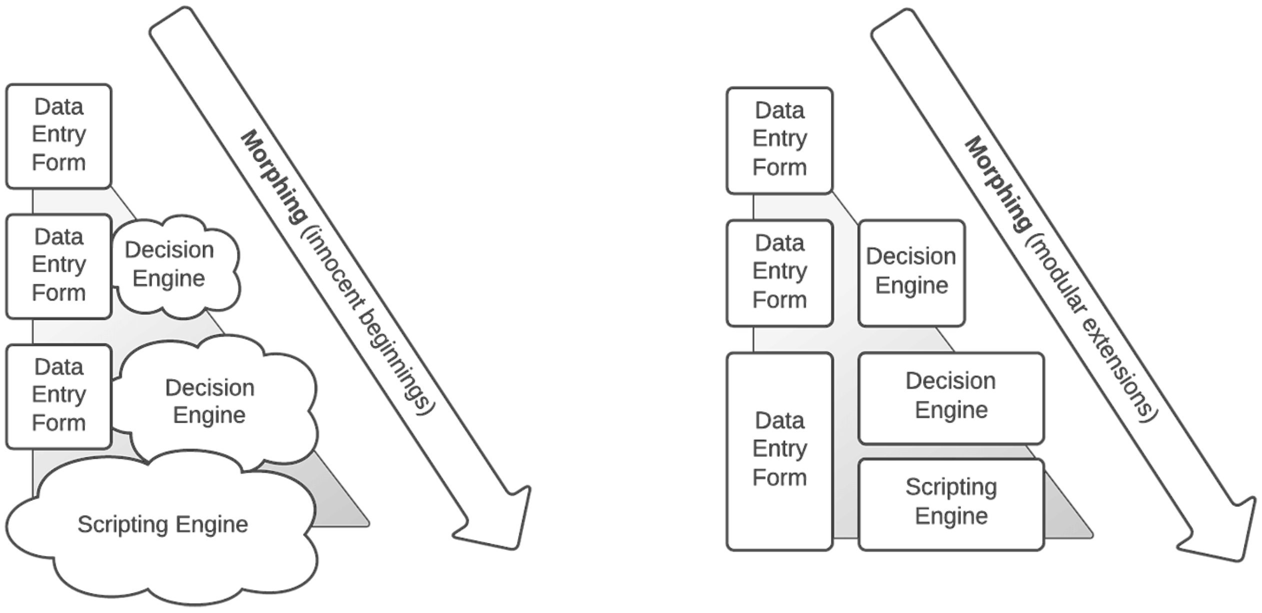
Example

A GUI module that is intended to interface to a processing module gradually takes on the processing functionality of background-processing modules. An example of this is a PowerBuilder screen for customer data entry/retrieval. The screen can:

1. Display data.
2. Edit data.
3. Perform simple type validation. The developer then adds functionality to what was intended to be the decision engine:
    - Complex validation.
    - Algorithms that use the validated data to assess next actions.
4. The developer then gets new requirements to:
    - Extend the GUI to three forms.
    - Make it script-driven (including the development of a script engine).

- ○ Add new algorithms to the decision engine.

The developer extends the current module to incorporate all of this functionality. So instead of developing several modules, a single module is developed. If the intended application is architected and designed, it is easier to maintain and extend.



Continuous Obsolescence

AntiPattern Problem

Technology is changing so rapidly that developers have trouble keeping up with the current versions of software and finding combinations of product releases that work together.

Given that every commercial product line evolves through new product releases, the situation has become increasingly difficult for developers to cope with. Finding compatible releases of products that successfully interoperate is even harder.

Java is a well-known example of this phenomenom, with new versions coming out every few months. For example, by the time a book on Java 1.X goes to press, a new Java Development Kit obsoletes the information. Java is not alone; many other technologies also participate in Continuous Obsolescence.

The most flagrant examples are products that embed the year in their brand names, such as Product98. In this way, these products flaunt the progression of their obsolescence. Another example is the progression of Microsoft dynamic technologies:

- DDE
- OLE 1.0
- OLE 2.0
- COM
- ActiveX
- DCOM
- COM?

From the technology marketers' perspective, there are two key factors: *mindshare* and *marketshare.* Rapid innovation requires the dedicated attention of consumers to stay current with the latest product features, announcements, and terminology.

For those following the technology, rapid innovation contributes to mindshare; in other words, there is always new news about technology X. Once a dominant marketshare is obtained, the suppliers' primary income is through obsolescence and replacement of earlier product releases. The more quickly technologies obsolesce (or are perceived as obsolete), the higher the income.

Refactored Solution

An important stabilizing factor in the technology market is *open systems standards.* A consortium standard is the product of an industry concensus that requires time and investment.

Joint marketing initiatives build user awareness and acceptance as the technologies move into the mainstream. There is an inherent inertia in this process that benefits consumers, for once a vendor product is conformant to a standard, the manufacturer is unlikely to change the conformant features of the product.

The advantages of a rapidly obsolescing technology are transitive. Architects and developers should depend upon interfaces that are stable or that they control. Open systems standards give a measure of stability to an otherwise chaotic technology market.

Variations

The Wolf Ticket Mini-AntiPattern describes various approaches that consumers can use to influence product direction toward improved product quality.

Lava Flow

- **AntiPattern Name:** Lava Flow
- **Also Known As:** Dead Code
- **Most Frequent Scale:** Application
- **Refactored Solution Name:** Architectural Configuration Management
- **Refactored Solution Type:** Process
- **Root Causes:** Avarice, Greed, Sloth
- **Unbalanced Forces:** Management of Functionality, Performance, Complexity
- **Anecdotal Evidence:** "Oh *that!* Well Ray and Emil (they're no longer with the company) wrote that routine back when Jim (who left last month) was trying a workaround for Irene's input processing code (she's in another department now, too). I don't think it's used anywhere now, but I'm not really sure. Irene didn't really document it very clearly, so we figured we would just leave well enough alone for now. After all, the bloomin' thing works doesn't it?!"

Background

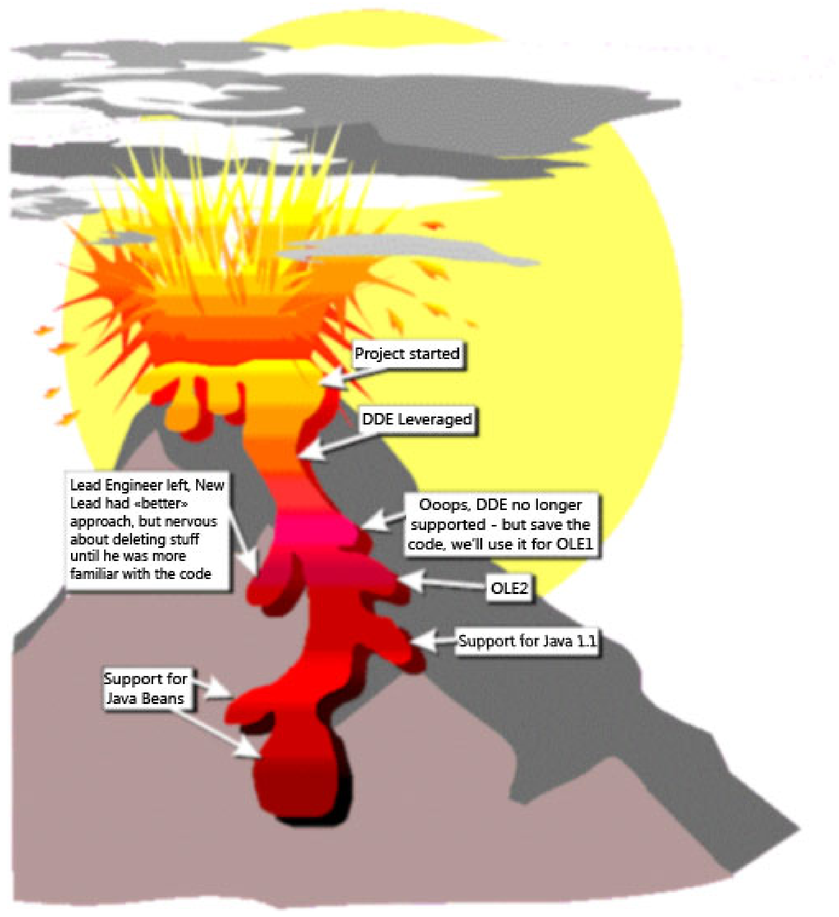
In a data-mining expedition, we began looking for insight into developing a standard interface for a particular kind of system. The system we were mining was very similar to those we hoped would eventually support the standard we were working on. It was also a research-originated system and highly complex. As we delved into it, we interviewed many of the developers concerning certain components of the massive number of pages of code printed out for us.

Over and over, we got the same answer: "I don't know what that class is for; it was written before I got here." We gradually realized that between 30 and 50 percent of the actual code that comprised this complex system was not understood or documented by any one currently working on it.

Furthermore, as we analyzed it, we learned that the questionable code really served no purpose in the current system; rather, it was there from previous attempts or approaches by long-gone developers. The current staff, while very bright, was loath to modify or delete code that they didn't write or didn't know the purpose of, for fear of breaking something and not knowing why or how to fix it.

At this point, we began calling these blobs of code "lava," referring to the fluid nature in which they originated as compared to the basaltlike hardness and difficulty in removing it once it had solidified. Suddenly, it dawned on us that we had identified a potential AntiPattern.

Nearly a year later, and after several more data-mining expeditions and interface design efforts, we had encountered the same pattern so frequently that we were routinely referring to Lava Flow throughout the department.


Project started
DDE Leveraged
Lead Engineer left, New Lead had «better» approach, but nervous about deleting stuff until he was more familiar with the code
Ooops, DDE no longer supported - but save the code, we'll use it for OLE1
OLE2
Support for Java 1.1
Support for Java Beans

General Form

The Lava Flow AntiPattern is commonly found in systems that originated as research but ended up in production. It is characterized by the lavalike "flows" of previous developmental versions strewn about the code landscape, which have now hardened into a basaltlike, immovable, generally useless mass of code that no one can remember much, if anything, about.

This is the result of earlier (perhaps Jurassic) developmental times when, while in a research mode, developers tried out several ways of accomplishing things, typically in a rush to deliver some kind of demonstration, thereby casting sound design practices to the winds and sacrificing documentation.

```
// This class was written by someone earlier (Alex?) to manager the indexing
// or something (maybe). It's probably important. Don't delete. I don't think it's
// used anywhere - at least not in the new MacroINdexer module which may
// actually replace whatever this was used for.
class IndexFrame extends Frame {
  // IndexFrame constructor
  // ---------------------------
  public IndexFrame(String index_parameter_1)
  {
    // Note: need to add additional stuff here...
    super (str);
  }
  // ---------------------------
```

The result is several fragments of code, wayward variable classes, and procedures that are not clearly related to the overall system. In fact, these flows are often so complicated in appearance and spaghettilike that they seem important, but no one can really explain what they do or why they exist.

Sometimes, an old, gray-haired hermit developer can remember certain details, but typically, everyone has decided to "leave well enough alone" since the code in question "doesn't really cause any harm, and might actually be critical, and we just don't have time to mess with it."

Though it can be fun to dissect these flows and study their anthropology, there is usually not enough time in the schedule for such meanderings. Instead, developers usually take the expedient route and neatly work around them.

This AntiPattern is, however, incredibly common in innovative design shops where proof-of-concept or prototype code rapidly moves into production. It is poor design, for several key reasons:

- Lava Flows are expensive to analyze, verify, and test. All such effort is expended entirely in vain and is an absolute waste. In practice, verification and test are rarely possible.
- Lava Flow code can be expensive to load into memory, wasting important resources and impacting performance.
- As with many AntiPatterns, you lose many of the inherent advantages of an object-oriented design. In this case, you lose the ability to leverage modularization and reuse without further proliferating the Lava Flow globules.

Symptoms And Consequences

- Frequent unjustifiable variables and code fragments in the system.
- Undocumented complex, important-looking functions, classes, or segments that don't clearly relate to the system architecture.
- Very loose, "evolving" system architecture.
- Whole blocks of commented-out code with no explanation or documentation.
- Lots of "in flux" or "to be replaced" code areas.
- Unused (dead) code, just left in.
- Unused, inexplicable, or obsolete interfaces located in header files.
- If existing Lava Flow code is not removed, it can continue to proliferate as code is reused in other areas.
- If the process that leads to Lava Flow is not checked, there can be exponential growth as succeeding developers, too rushed or intimidated to analyze the original flows,

continue to produce new, secondary flows as they try to work around the original ones, this compounds the problem.
- As the flows compound and harden, it rapidly becomes impossible to document the code or understand its architecture enough to make improvements.

Typical Causes

- R&D code placed into production without thought toward configuration management.
- Uncontrolled distribution of unfinished code. Implementation of several trial approaches toward implementing some functionality.
- Single-developer (lone wolf) written code.
- Lack of configuration management or compliance with process management policies.
- Lack of architecture, or non-architecture-driven development. This is especially prevalent with highly transient development teams.
- Repetitive development process. Often, the goals of the software project are unclear or change repeatedly. To cope with the changes, the project must rework, backtrack, and develop prototypes. In response to demonstration deadlines, there is a tendency to make hasty changes to code on the fly to deal with immediate problems. The code is never cleaned up, leaving architectural consideration and documentation postponed indefinitely.
- Architectural scars. Sometimes, architectural commitments that are made during requirements analysis are found not to work after some amount of development. The system architecture may be reconfigured, but these inline mistakes are seldom removed. It may not even be feasible to comment-out unnecessary code, especially in modern development environments where hundreds of individual files comprise the code of a system. "Who's going to look in all those files? Just link em in!"

Known Exceptions

Small-scale, throwaway prototypes in an R&D environment are ideally suited for implementing the Lava Flow AntiPattern. It is essential to deliver rapidly, and the result is not required to be sustainable.

Refactored Solution

There is only one sure-fire way to prevent the Lava Flow AntiPattern: Ensure that sound architecture precedes production code development. This architecture must be backed up by a configuration management process that ensures architectural compliance and accommodates "mission creep" (changing requirements).

If architectural consideration is shortchanged up front, ultimately, code is developed that is not a part of the target architecture, and is therefore redundant or dead. Over time, dead code becomes problematic for analysis, testing, and revision.

In cases where Lava Flow already exists, the cure can be painful. An important principle is to avoid architecture changes during active development. In particular, this applies to computational architecture, the software interfaces defining the systems integration solution. Management must postpone development until a clear architecture has been defined and disseminated to developers.

Defining the architecture may require one or more system discovery activities. System discovery is required to locate the components that are really used and necessary to the system. System discovery also identifies those lines of code that can be safely deleted. This activity is tedious; it can require the investigative skills of an experienced software detective.

As suspected dead code is eliminated, bugs are introduced. When this happens, resist the urge to immediately fix the symptoms without fully understanding the cause of the error. Study the dependencies. This will help you to better define the target architecture.

To avoid Lava Flow, it is important to establish system-level software interfaces that are stable, well-defined, and clearly documented. Investment up front in quality software interfaces can produce big dividends in the long run compared to the cost of jackhammering away hardened globules of Lava Flow code.

Tools such as the Source-Code Control System (SCCS) assist in configuration management. SCCS is bundled with most Unix environments and provides a basic capability to record histories of updates to configuration-controlled files.

Example

We recently participated in a data-mining expedition site where we attempted to identify evolutionary interfaces that resulted from preliminary interface architectures that we originated and were in the process of updating.

The system we mined was targeted because the developers had utilized our initial architecture in a unique way that fascinated us: Essentially, they constructed a quasi-event service out of our generic interapplication framework.

As we studied their system, we encountered large segments of code that baffled us. These segments didn't seem to contribute to the overall architecture that we had expected to find. They were somewhat incohesive and only very sparsely documented, if at all.

When we asked the current developers about some of these segments, the reply was, "Oh that? Well we're not using that approach anymore. Reggie was trying something, but we came up with a better way. I guess some of Reggie's other code may depend on that stuff though, so we didn't delete anything." As we looked deeper into the matter, we learned that

Reggie was no longer even at the site, and hadn't been there for some time, so the segments of code were several months old.

After two days of code examination, we realized that the majority of the code that comprised the system was most likely similar to that code that we already examined: completely Lava Flow in nature.

We gleaned very little that helped us articulate how their architecture actually was constructed; therefore, it was nearly impossible to mine. At this point, we essentially gave up trying to mine the code and instead focused on the current developer's explanations of what was "really" going on, hoping to somehow codify their work into interface extensions that we could incorporate into our upcoming revisions to our generic interapplication framework.

One solution was to isolate the single, key person who best understood the system they had developed, and then to jointly write IDL with that person. On the surface, the purpose of the IDL we were jointly writing was to support a crisis demonstration that was weeks away.

By utilizing the Fire Drill Mini-AntiPattern, we were able to get the systems developers to validate our IDL by using it to rapidly build a CORBA wrapper for their product for the demonstration. Many people lost a lot of sleep, but the demonstration went well. There was, of course, one side effect to this solution: We ended up with the interface, in IDL, which we had set out to discover in the first place.

Related Solutions

In today's competitive world, it is often desirable to minimize the time delay between R&D and production. In many industries, this is critical to a company's survival. Where this is the case, inoculation against Lava Flow can sometimes be found in a customized configuration-management (CM) process that puts certain limiting controls in place at the prototyping stage, similar to "hooks" into a real, production-class develop ment without the full restraining impact on the experimental nature of R&D.

Where possible, automation can play a big role here, but the key lies in the customization of a quasi-CM process that can be readily scaled into a full-blown CM control system once the product moves into a production environment. The issue is one of balance between the costs of CM in hampering the creative process and the cost of rapidly gaining CM control of the development once that creative process has birthed something useful and marketable.

This approach can be facilitated by periodic mapping of a prototyping system into an updated system architecture, including limited, but standardized inline documentation of the code.

Applicability To Other Viewpoints And Scales

The architectural viewpoint plays a key role in preventing Lava Flows initially. Managers can also play a role in early identification of Lava Flows or the circumstances that can lead to Lava Flows. These managers must also have the authority to put the brakes on when Lava Flow is first identified, postponing further development until a clear architecture can be defined and disseminated.

As with most AntiPatterns, prevention is always cheaper than correction, so up-front investment in good architecture and team education can typically ensure a project against this and most other AntiPatterns. While this initial cost does not show immediate returns, it is certainly a good investment.

Ambiguous Viewpoint

AntiPattern Problem

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 are one of the primary benefits of the object-oriented paradigm.

Refactored Solution

There are three fundamental viewpoints for OOA&D models: the business viewpoint, the specification viewpoint, and the implementation viewpoint. The business viewpoint defines the user's model of the information and processes. This is a model that domain experts can defend and explain (commonly called an analysis model). Analysis models are some of the most stable models of the information system and are worthwhile to maintain.

Models can be less useful if they don't focus on the required perspective(s). A perspective applies filters to the information. For example, defining a class model for a telephone exchange system will vary significantly depending upon the focus provided by the following perspectives:

- Telephone user, who cares about the ease of making calls and receiving itemized bills.
- Telephone operator, who cares about connecting users to required numbers.
- Telephone accounting department, which cares about the formulae for billing and records of all calls made by users.

Some of the same classes will be identified, but not many; where there are, the methods will not be the same.

The specification viewpoint focuses on software interfaces. Because objects (as abstract data types) are intended to hide implementation details behind interfaces, the specification viewpoint defines the exposed abstractions and behaviors in the object system. The specification viewpoint defines the software boundaries between objects in the system.

The implementation viewpoint defines the internal details of the objects. Implementation models are often called design models in practice. To be an accurate model of the software, design models must be maintained continuously as the software is developed and modified. Since an out-of-date model is useless, only selected design models are pertinent to maintain; in particular, those design models that depict complex aspects of the system.

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:

1. 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.
2. 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

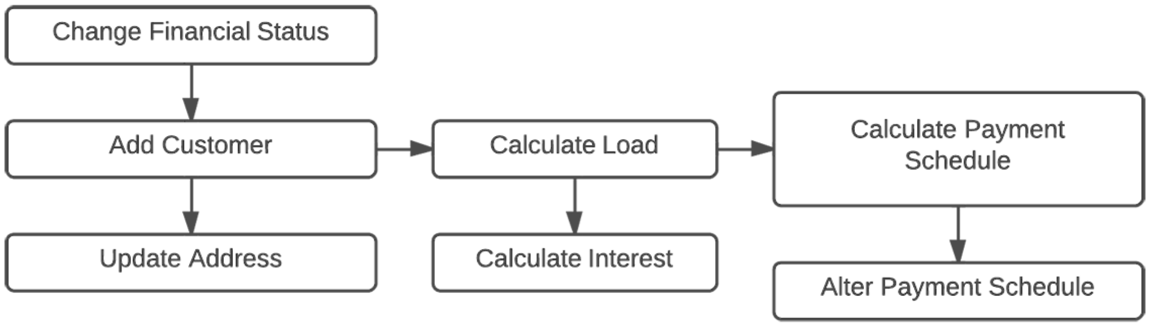
controller object with methods that perform the activities previously spread out among several classes.
3. If the class does not contain state information of any kind, consider rewriting it as a function. Potentially, some parts of the system may be best modeled as functions that can be accessed throughout 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

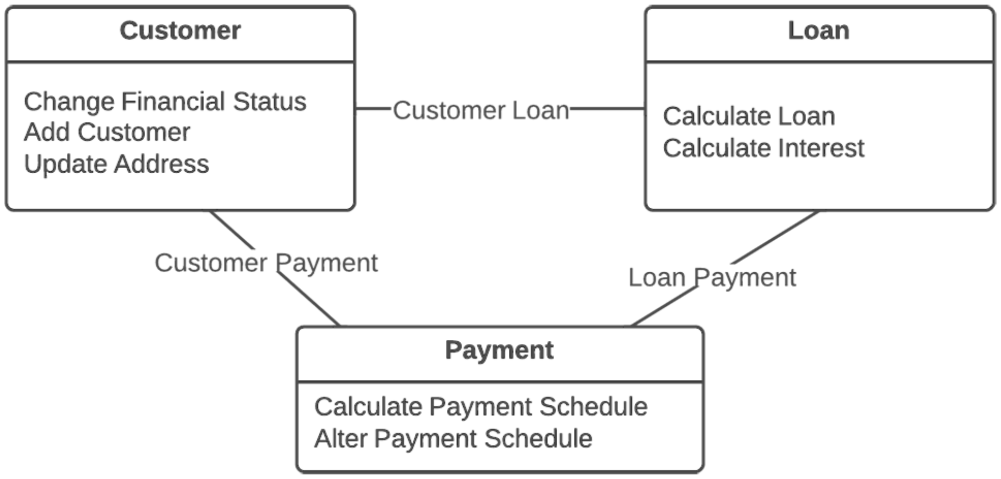
Functional Decomposition is based upon discrete functions for the purpose of data manipulation, for example, the use of Jackson Structured Programming. Functions are often methods within an object-oriented environment. The partitioning of functions is based upon a different paradigm, which leads to a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

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



1. Adding a new customer.
2. Updating a customer address.
3. Calculating a loan to a customer.
4. Calculating the interest on a loan.
5. Calculating a payment schedule for a customer loan.
6. 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.