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.

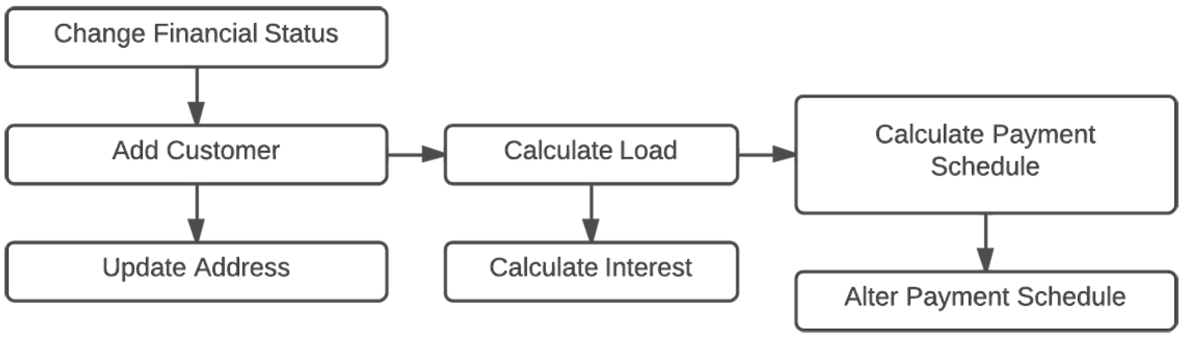
3. If the class does not contain state information of any kind, consider rewriting it as a function. Potentially, some parts of the system may be best modeled as functions that can be accessed throughout various parts of the system without restriction.

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

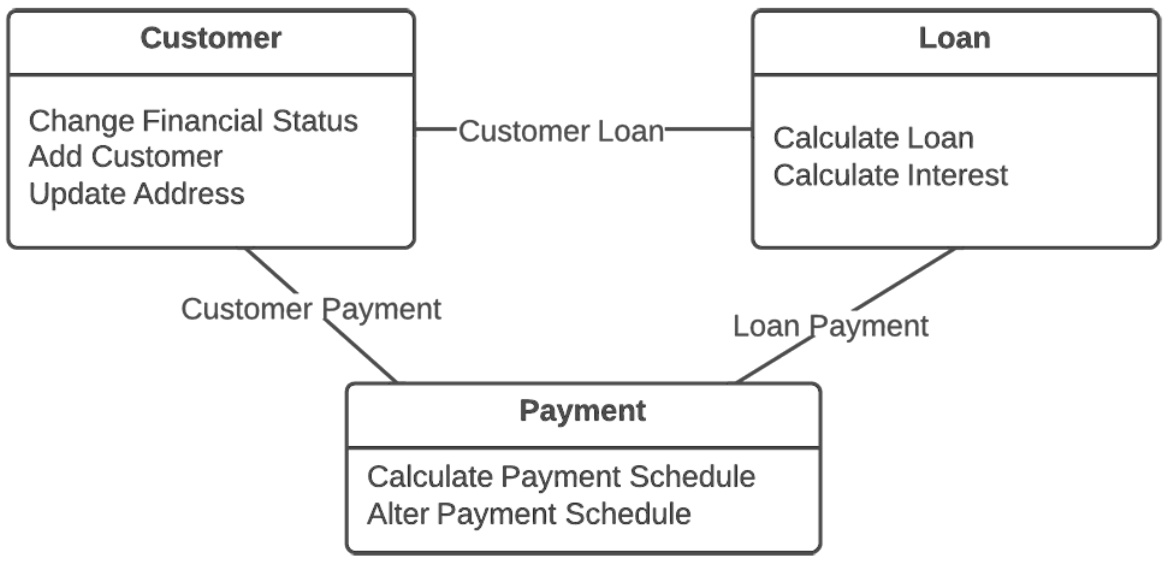
Functional Decomposition is based upon discrete functions for the purpose of data manipulation, for example, the use of Jackson Structured Programming. Functions are often methods within an object-oriented environment. The partitioning of functions is based upon a different paradigm, which leads to a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

The simple example in 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.