Advanced Topics of Software Engineering (ASE)

Chapter 2. From requirements to system design

Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

From requirements to system design

2.1. Software architecture

2.2. Antipatterns in software engineering

2.3. Reuse

2.4. Testability

2.5. Safety

2.6. Information security

Antipatterns

"If one does not know how to solve a problem, it may nevertheless be useful to know about likely blind alleys.

This is particularly true when something appears at first to be a solution but further analysis proves it is not..."

[Andrew Koenig (coined the term Antipattern in 1995)]

"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: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Antipatterns are solutions that don't work.

Antipatterns

The "ugly parking garage" antipattern

Antipatterns

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

> "The point isn't as much to say "don't do this" as it is to say "you probably don't even realize that you're doing this, but it doesn't work" ….

[from http://c2.com/cgi/wiki?AntiPattern]

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

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.
- Cut-and-paste programming, Spaghetti code, Continuous obsolescence…..

Software architecture antipatterns

- Architecture antipatterns focus on the system-level and enterprise-level structure of applications and components.
- Swiss army knife, Design by committee, Vendor lock-in….

[cf. https://sourcemaking.com/antipatterns/ for more examples]

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Root causes for antipatterns

Common mistakes in software project management and development:

- Insufficient communication with the client
- Unfulfilled requirements
- Insufficient testing
- Cost overruns and schedule slips

Reason for these mistakes:
"The 7 deadly sins"

Pieter Bruegel the Elder: Pride (Superbia), from the series The Seven Deadly Sins

The "7 deadly sins" in software practice (1)

Haste

- Solutions based on hasty decisions ("*time is most important*") lead to compromises in software quality.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The "7 deadly sins" in software practice (2)

Apathy

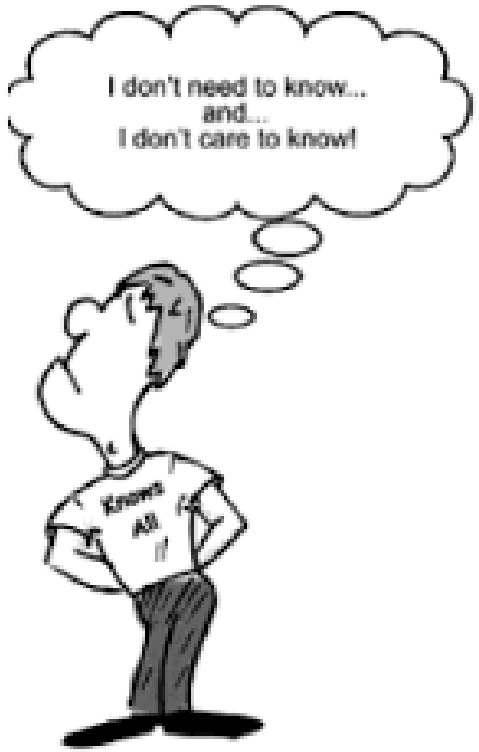
- Not caring about a problem, followed by unwillingness to attempt a solution.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The "7 deadly sins" in software practice (3)

Narrow-mindedness

- The refusal to use solutions that are widely known ("Why reuse? I only have to solve one problem").



["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The "7 deadly sins" in software practice (4)

Sloth

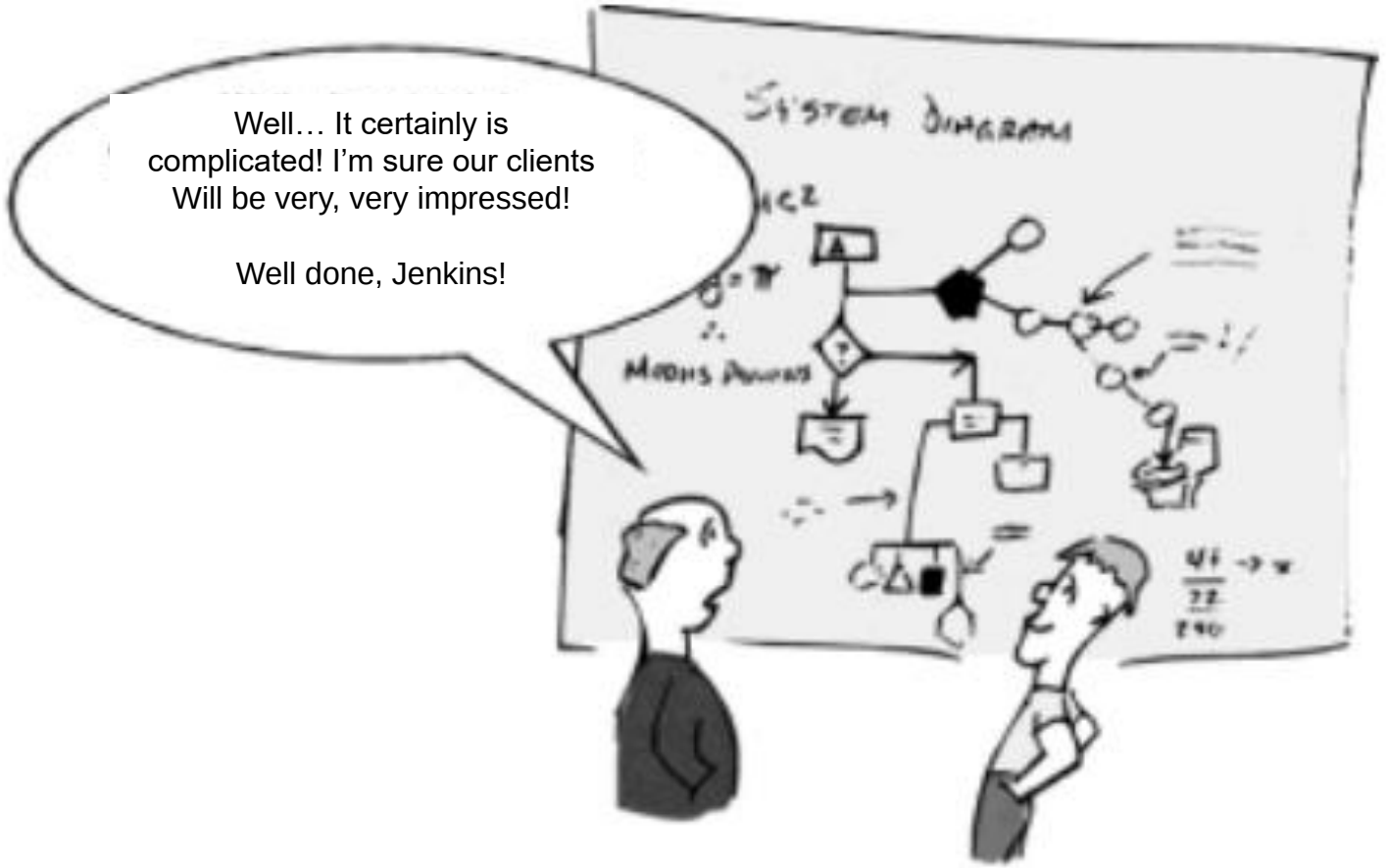
- Making poor decisions based on "*easy*" answers.
- Sloth usually ends up in sudden clarity.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The "7 deadly sins" in software practice (5)

Avarice (Excessive complexity)

- No use of abstractions, excessive modeling of details
- Addicted to complexity



["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The "7 deadly sins" in software practice (6)

Ignorance

- Failure to seek understanding.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The "7 deadly sins" in software practice (7)

Pride (Hubris)

- Not invented here (NIH): Not willing to adopt anything from the outside.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The blob - antipattern

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."

The Blob is found in designs where one class monopolizes the processing.

- Other classes primarily encapsulate data. class diagram composed of a single complex controller class
- Surrounded by simple data classes.
- Majority of the responsibilities are allocated to a single class.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

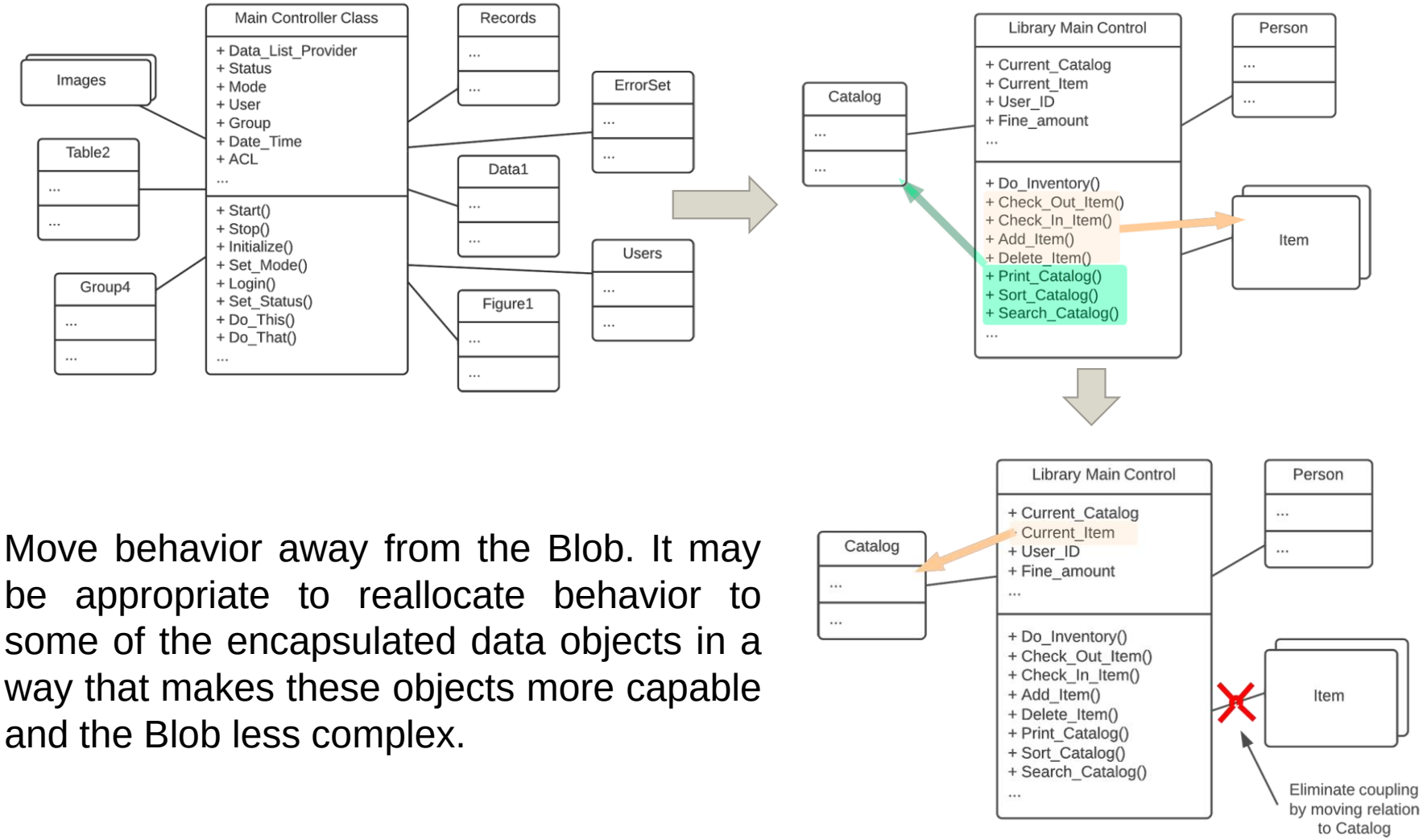
The blob - antipattern

Typical Causes

- **Lack of an object-oriented architecture.** The designers may not have an adequate understanding of object-oriented principles.
- **Lack of (any) architecture.** The absence of definition of the system components, their interactions, and the specific use of the selected programming languages.
- **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.
- **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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

The blob - antipattern



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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Functional decomposition - antipattern

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."

Result of experienced, non-object-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.

- Code resembles a structural language such as Pascal or FORTRAN in class structure.
- Can be incredibly complex, as smart procedural developers devise very clever ways to replicate their time-tested methods in an object-oriented architecture.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Functional decomposition - antipattern

Typical Causes

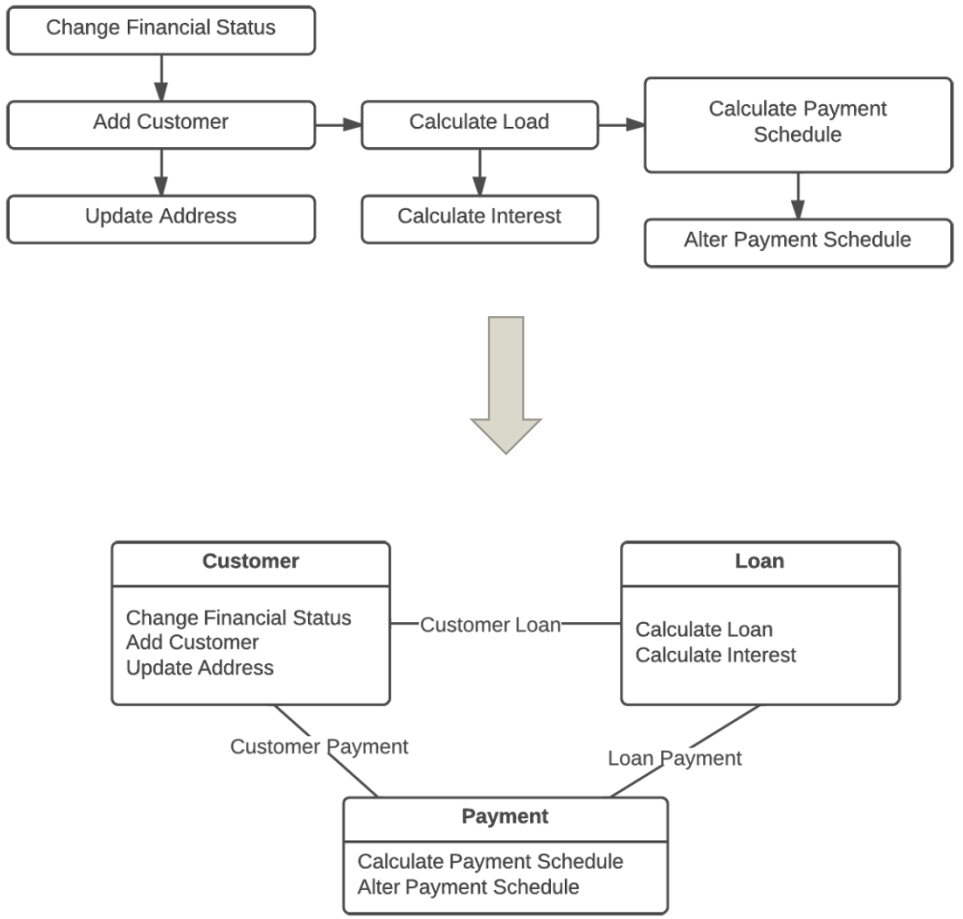
- **Lack of object-oriented understanding**. The implementers didn't "get it." This is fairly common when developers switch from programming in a non-object-oriented programming language to an object-oriented programming language.
- **Lack of architecture enforcement**. When the implementers are clueless 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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Functional decomposition - antipattern

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.

- Model will justify, or at least rationalize, most of the software modules.
- Provides insight as to how the overall system fits together.
- Several parts of the system exist for reasons no longer known and for which no reasonable speculation can be attempted.



["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Auto-generated stovepipe

Occurs when migrating an existing software system to a distributed infrastructure. An auto-generated 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.

- Existing interfaces may be using fine-grained operations to transfer information that may be inefficient in a distributed environment.
- Preexisting interfaces are usually *implementation-specific* and will cause subsystem interdependencies when used in a larger-scale distributed system.
- Local operations often make various assumptions about location, including address space and access to the local file system.
- Excess complexity can arise when multiple existing interfaces are exposed across a larger-scale distributed system.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Auto-generated stovepipe

- When designing distributed interfaces for existing software, the interfaces should be reengineered.
- A separate, larger-grained object model should be considered for the distributed interfaces.
- The interoperability functionality used by multiple subsystems should be the design center of the new interfaces.
- The stability of the new interfaces is very important, given that separately compiled software will be dependent upon these new designs.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

More antipatterns

… and there are many more antipatterns with a clear undesired influence on architecture but not directly related to architecture:

- Golden hammer
- Design-by-committee

Golden hammer - antipattern

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."

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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Golden hammer - antipattern

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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Golden hammer - antipattern

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.

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.

- Commitment by management in the professional development of their developers,
- Component should insulate the system from proprietary features in its implementation.
- Developers need to be up to date on technology trends.
- On the management side, adopt a commitment to open systems and architectures.
- Reusable code requires an investment in its initial development encourage the hiring of people from different areas and from different backgrounds.
- Actively invest in the professional development of software developers, as well as reward developers who take initiative in improving their own work.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Design by committee - antipattern

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.“

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, over constraint, and other specification defects. The design would lack conceptual clarity because so many people contributed to it and extended it during its creation.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Design by committee - antipattern

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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Design by committee - antipattern

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. Meeting productivity gains are dramatic, with several orders of magnitude typical (100 times), sometimes productivity gains over 100,000:1.

- Time awareness is essential to meeting progress. Participants should be coached to manage the time allotted efficiently.
- Posting the meeting goals, an agenda, and a clock where they are visible to all participants can improve meetings dramatically.
- "Why are we here?" and "What outcomes do we want?"
- Assign explicit roles in the software process: owner, facilitator, architect, developers, testers, and domain experts.
- 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.
- 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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Design by committee - antipattern

A short anecdote

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.

["AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." The Upstart Gang of Four (1998)]

Upcoming …

2.1. Software architecture

2.2. Antipatterns in software engineering

2.3. Reuse

2.4. Testability

2.5. Safety

2.6. Information security