Object Oriented Analysis and Design with Java

UE19CS353

Dr. Geetha D
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

PES UNIVERSITY ONLINE

UE19CS353: Object Oriented Analysis and Design with Java

Anti-Patterns - Architecture and Design Anti-Patterns

Department of Computer Science and Engineering

Definitions

- Pattern: Good ideas
- Antipattern: Bad ideas
- Refactoring: Better ideas

What is an Antipattern?

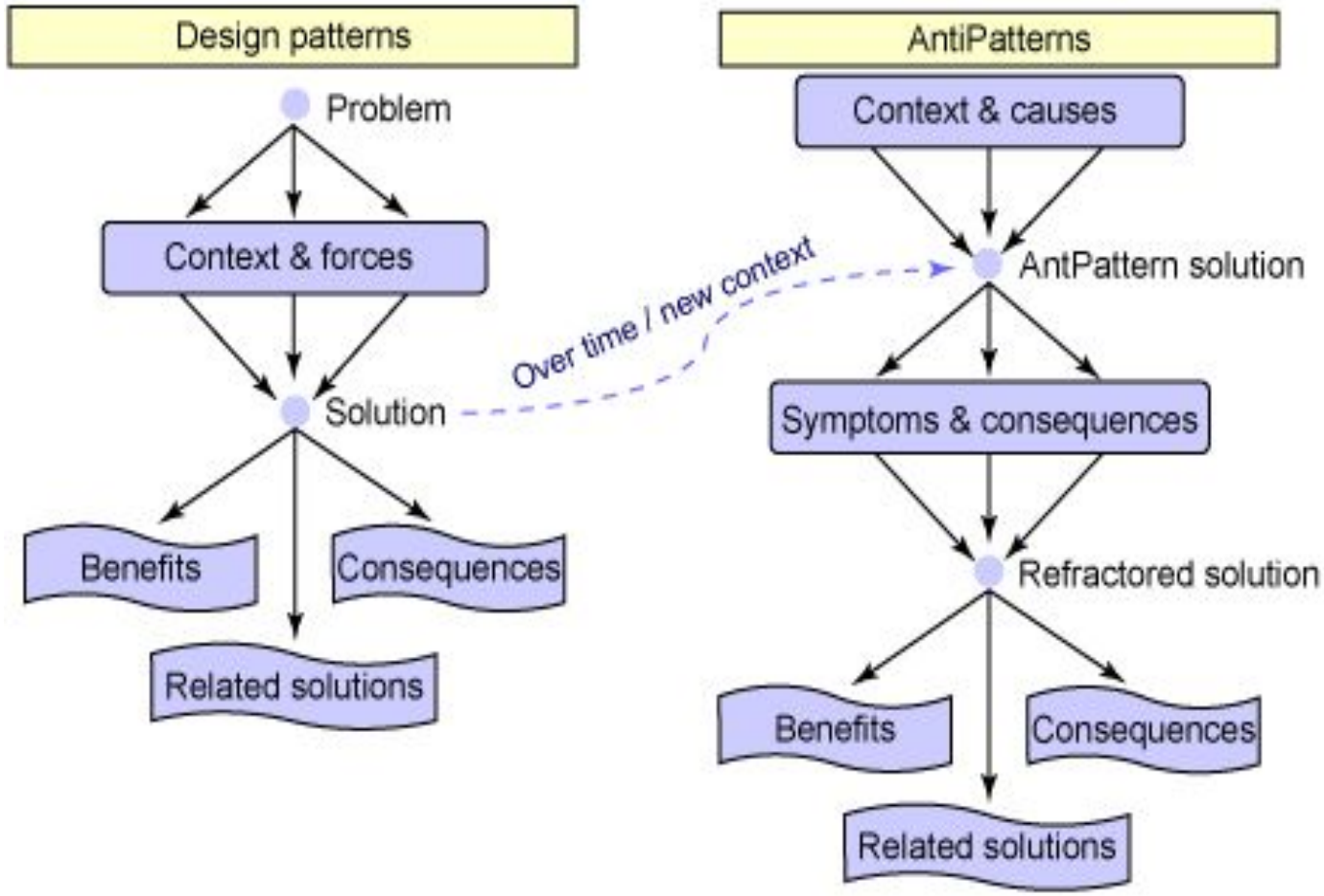
- An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.
- The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type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

What are AntiPatterns?

- “Negative Solutions,” or solutions that present more problems than they address.
- Natural extensions to design patterns.
- Bridge the gap between architectural concepts and real-world implementations.
- Provide Knowledge to prevent and recover from common Mistakes.
- Presents a detailed plan for reversing the underlying causes and implementing productive solutions.
- 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.

Antipatterns Vs. Design Patterns

- **Design Pattern-** a general repeatable solution to a commonly occurring problem
- **Antipattern-** Such a solution which is recognized as a poor way to solve the problem, and a refactored solution



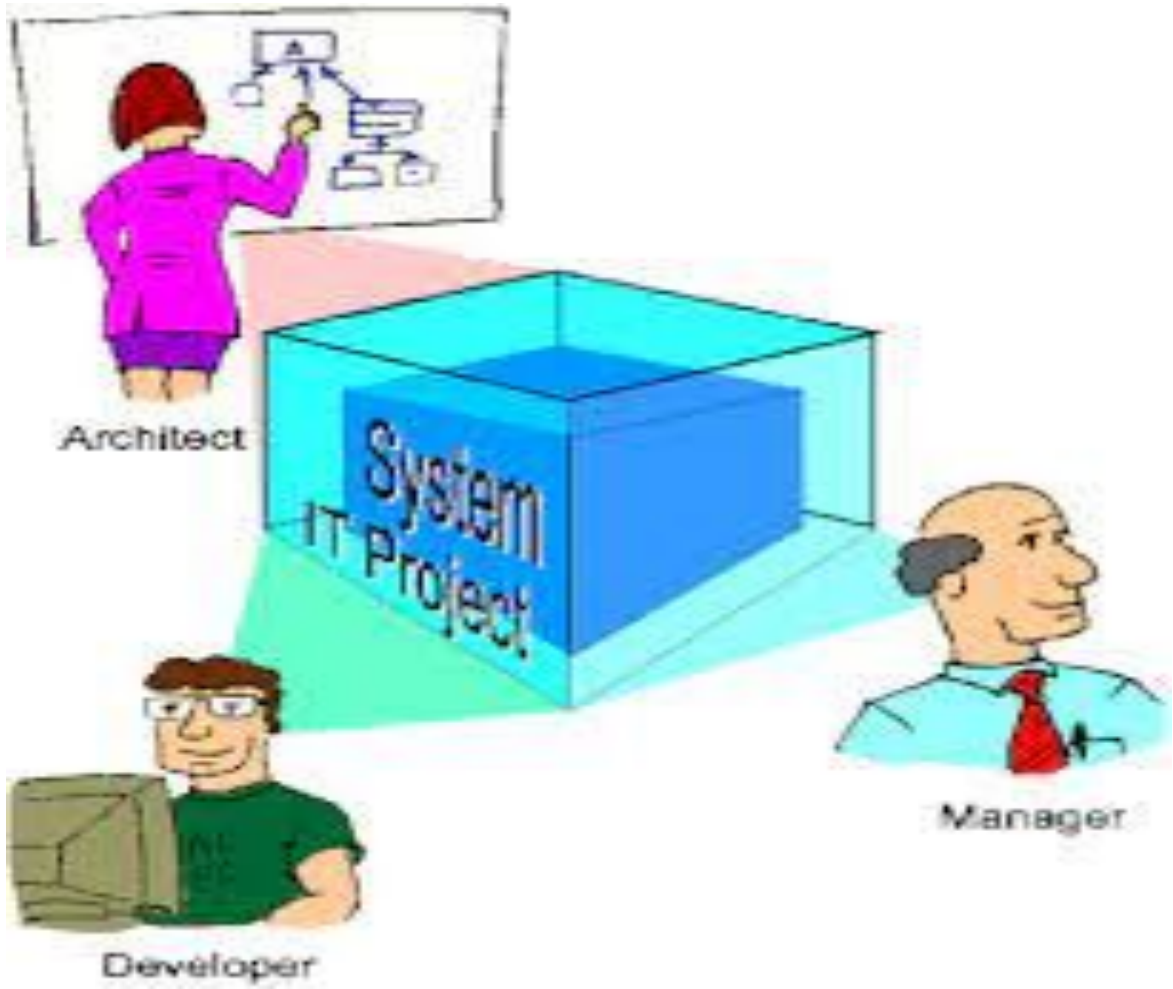
Why study Antipatterns?

- Antipatterns provide easily identifiable templates for common problems, as well as a path of action to rectify these problems.
- Antipatterns provide real world experience in recognizing recurring problems in the software industry providing a detailed remedy for the most common ones.
- Antipatterns provide a common vocabulary for identifying problems and discussing solutions.
- Antipatterns provide stress release in the form of shared misery.
- Antipatterns ensure common problems are not continually repeated within an organization

Viewpoints

AntiPatterns from three major viewpoints:

- The software developer,
- The software architect,
- The software manager



Principal AntiPattern viewpoints

Viewpoints

Development AntiPatterns describe situations encountered by the programmer when solving programming problems.

Architectural AntiPatterns focus on common problems in system structure, their consequences, and solutions. Many of the most serious unresolved problems in software systems occur from this perspective.

Management AntiPatterns describe common problems and solutions due to the software organization. It affects people in all software roles, and their solutions directly affect the technical success of the project.

Reference Model

A reference model for terminology common to all three viewpoints. The reference model is based upon three topics that introduce the key concepts of AntiPatterns:

- Root causes
  - Provide fundamental context for the AntiPattern
- Primal forces
  - are the key motivators for decision making
- Software design-level model (SDLM)
  - Define architectural scale

Reference Model – Root Causes

Root Causes

- Root causes are mistakes in software development that result in failed projects, cost overruns, schedule slips, and unfulfilled business needs.
- The root causes are based upon the "seven deadly sins," a popular analogy that has been used successfully to identify ineffective practices.

Haste: Tight deadlines often lead to neglecting important activities

Apathy: The attitude of not caring about solving known problems.

Narrow-Mindedness: Refusal of developers to learn proven solutions.

Root Causes (Contd.,)

Sloth: adaptation of the most simple “solution”.

Avarice: Greed in creating a system can result in very complex, and difficult to maintain software.

Ignorance: The lack of motivation to understand things.

Pride: The Failure to reuse existing software packages because they were not invented by a specific company

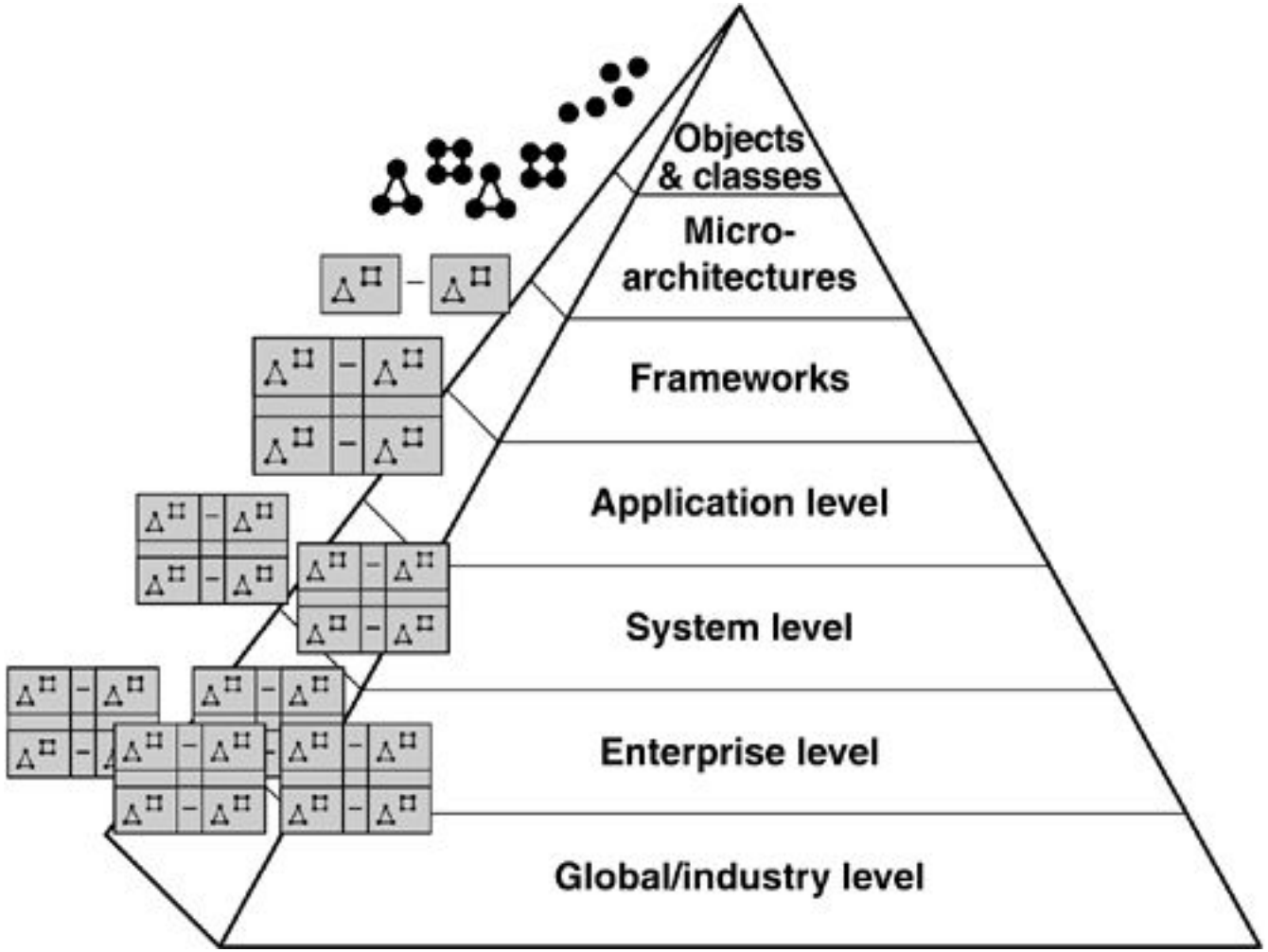
Primal Forces

Forces are concerns or issues that exist within a decision–making context. In a design solution, forces that are successfully addressed (or resolved) lead to benefits, and forces that are unresolved lead to consequences.

The primal forces include:

- Management of Functionality - Meeting the requirements
- Management of Performance - Meeting required speed and operation
- Management of Complexity - Defining abstractions
- Management of Change - Controlling the evolution of software
- Management of IT Resources - Managing people and IT artifacts
- Management of Technology Transfer - Controlling technology evolution

Software Design – Level Model



Software Design – Level Model (Contd.,)

- **The global level** contains the design issues that are globally applicable across all systems. This level is concerned with coordination across all organizations, which participate in cross–organizational communications and information sharing.
- **The enterprise level** is focused upon coordination and communication across a single organization. The organization can be distributed across many locations and heterogeneous hardware and software systems.
- **The system level** deals with communications and coordination across applications and sets of applications.
- **The application level** is focused upon the organization of applications developed to meet a set of user requirements. The macro–component levels are focused on the organization and development of application frameworks.

Software Design – Level Model (Contd.,)

- **The micro−component level** is centered on the development of software components that solve recurring software problems. Each solution is relatively self−contained and often solves just part of an even larger problem.
- **The object level** is concerned with the development of reusable objects and classes. The object level is more concerned with code reuse than design reuse. Each of the levels is discussed in detail along with an overview of the patterns documented at each level.

Anti-Pattern Template

Name:

The formal name of the AntiPattern

Also Known As:

Other popular, descriptive, or humorous names for the AntiPattern.

Most Frequent Scale:

Where the AntiPattern fits into the SDLM Model

Refactored Solution Name:

The name of the pattern that acts as the proper Refactored solution.

Refactored Solution Type:

This will identify the type of improvement that results from applying the AntiPattern solution. Such improvements include: Software, Technology, Process, and Role improvements.

Anti-Pattern Template (Contd.,)

Root Causes:

One or more key root causes that result in the AntiPattern.

Unbalanced Forces:

Identifies the Primal Forces that are ignored, misused, or overused in the AntiPattern.

Anecdotal Evidence:

Common phrases and humorous anecdotes that describe the problem.

Background:

Sets the Scene for the AntiPattern and introduces the problem under discussion.

General Form:

General characteristics of the AntiPattern are identified, and an overview of the nature of the problem is presented.

Anti-Pattern Template (Contd.,)

Symptoms and Consequences:

A list of symptoms and related consequences resulting from this AntiPattern.

Typical Causes:

A list of the unique causes of an AntiPattern, which should relate to corresponding symptoms and consequences where possible.

Known Exceptions:

Specific occasions when AntiPattern behavior and processes may not always be wrong.

Refactored Solutions:

Resolves the unbalanced forces, causes, symptoms, and consequences of the AntiPattern.

Anti-Pattern Template (Contd.,)

Variations:

Lists known variations of the AntiPattern,

Example:

An example of the AntiPattern based on real-world experience

Related Solutions:

Identifies and lists any cross-references to other Antipatterns which are closely related.

Applicability to Other Viewpoints and Scales:

Describes the impact of the AntiPattern to other applicable SDLM scales.

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 Development Antipatterns – The Blob

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.

Template:

- 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."

Software Development Antipatterns – The Blob

Background:

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

Software Development Antipatterns – The Blob

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.
- 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.

Software Development Antipatterns – The Blob

Typical Causes

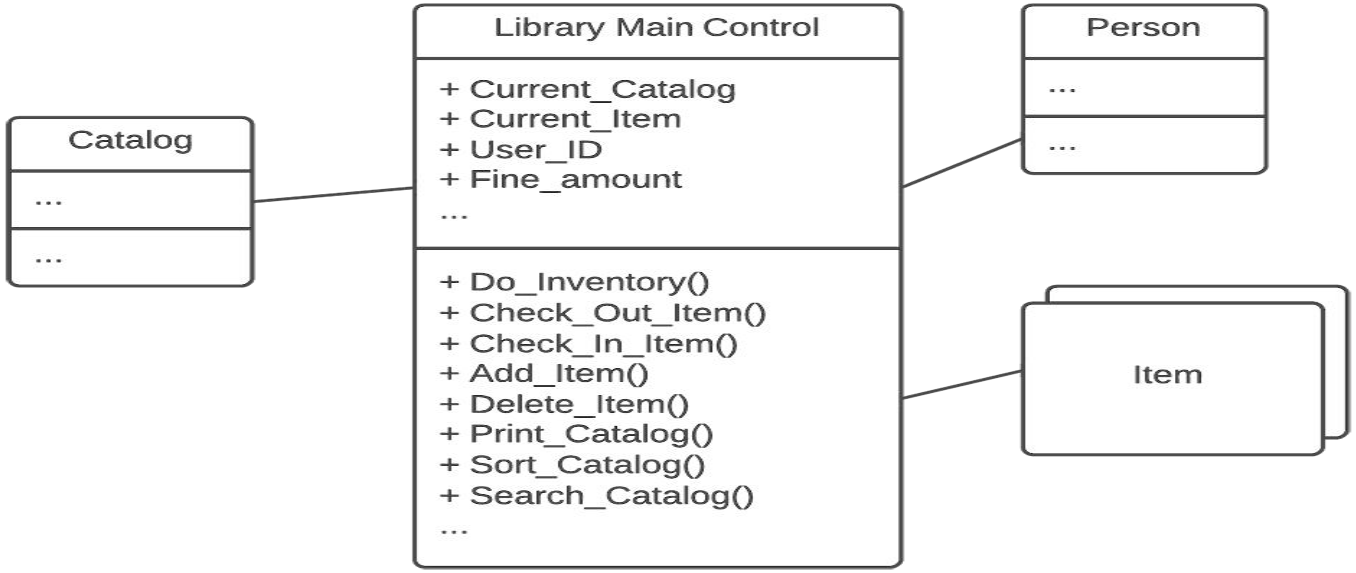
- Lack of an object-oriented architecture.
- Lack of (any) architecture.
- Lack of architecture enforcement.
- Too limited intervention.
- Specified disaster.

Software Development Antipatterns – The Blob

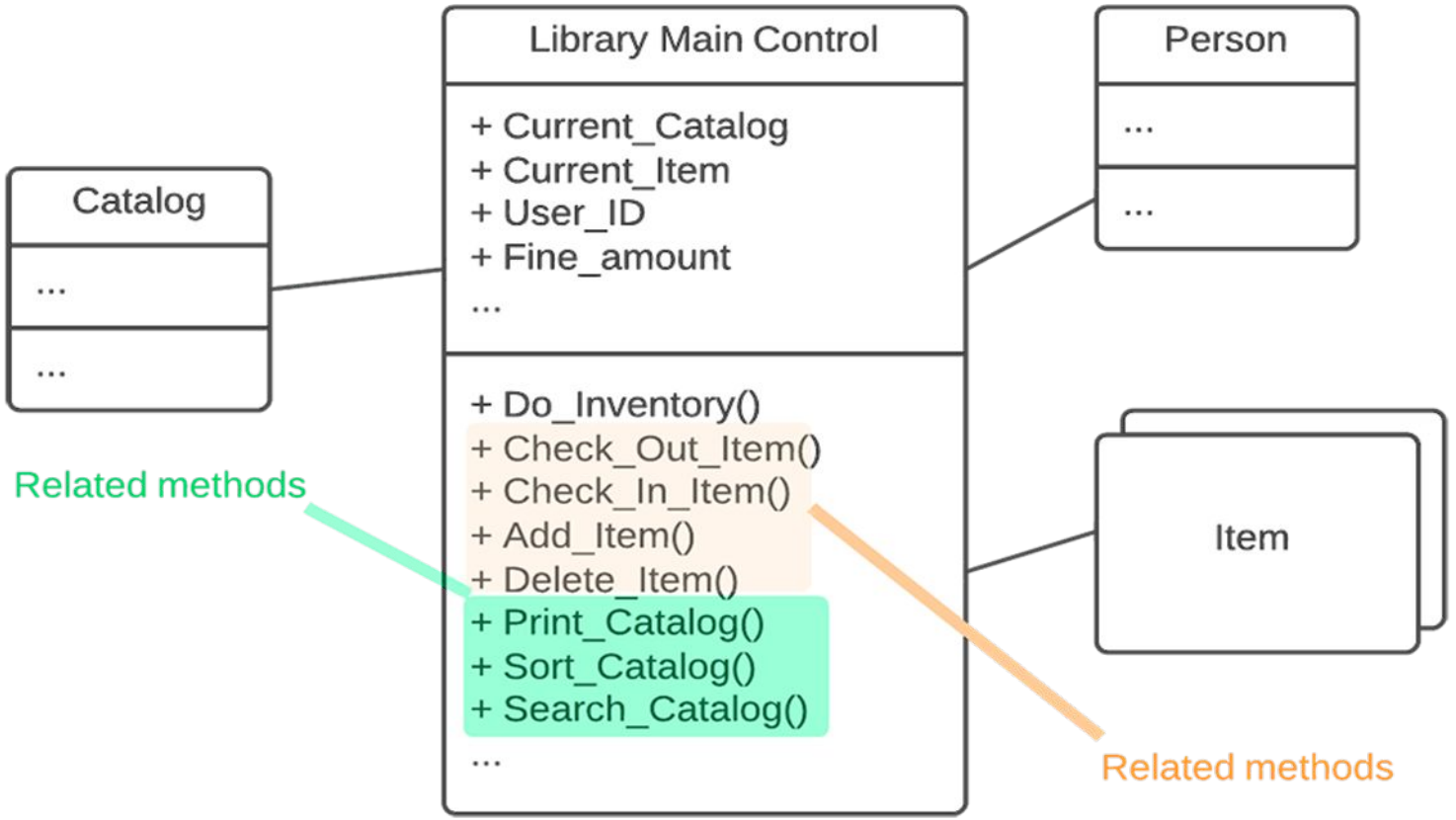
Refactored Solution

Step 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.



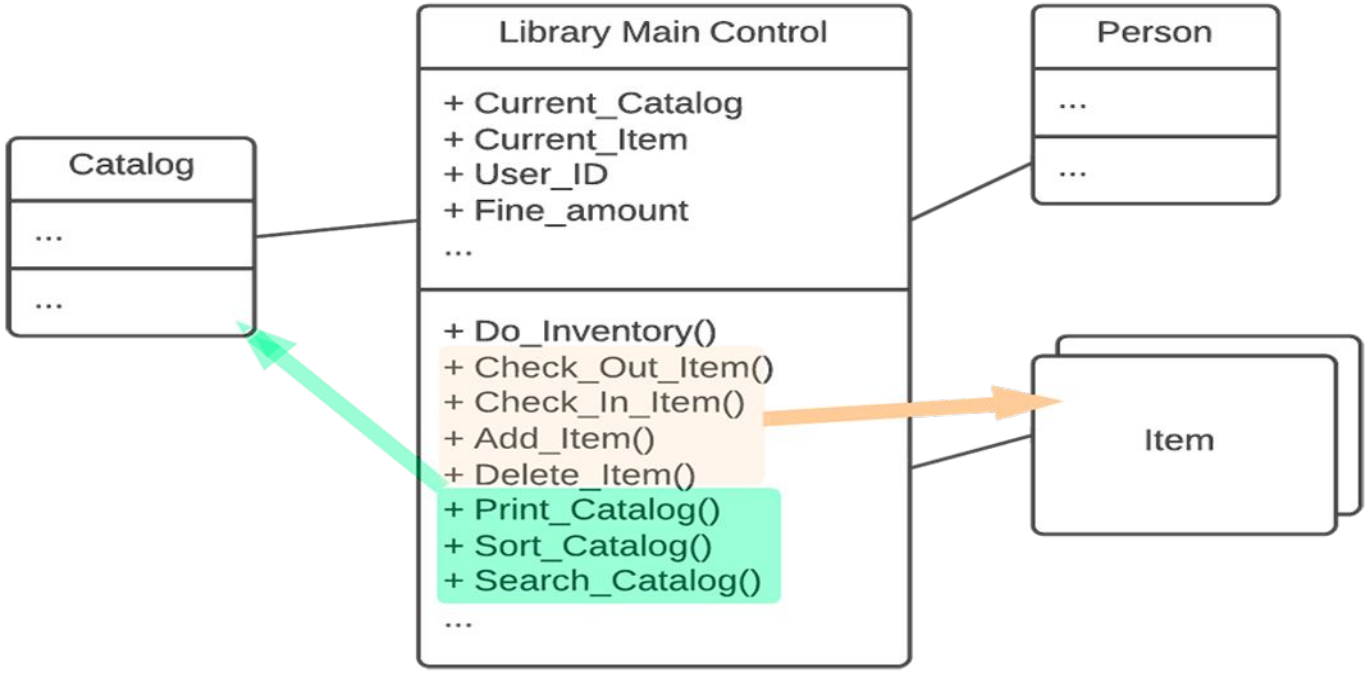
Software Development Antipatterns – The Blob



Software Development Antipatterns – The Blob

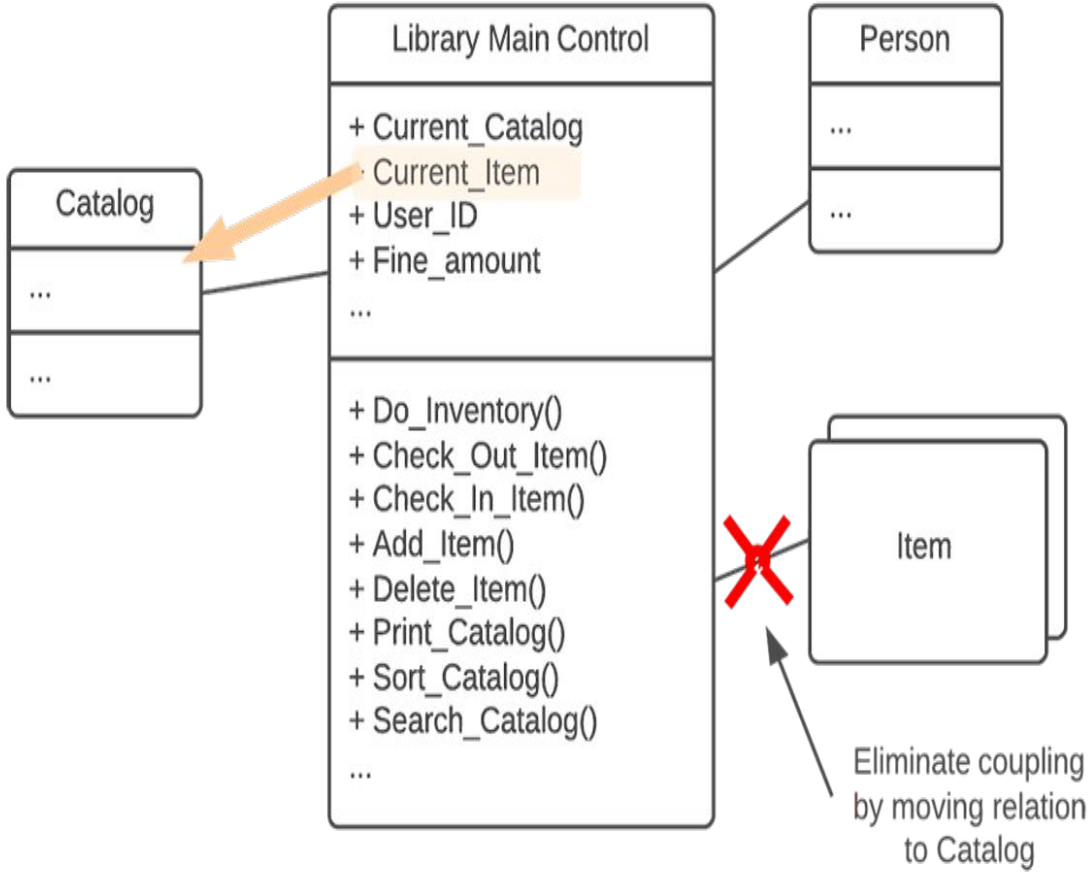
Step 2:

- 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.



Software Development Antipatterns – The Blob

- 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.
- 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,
- Finally, we remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments.



Software Development Antipatterns – 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?!"

Software Development Antipatterns – Lava Flow

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 basalt-like, immovable, generally useless mass of code that no one can remember much, if anything, about.

Software Development Antipatterns – Lava Flow

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.

Software Development Antipatterns – Lava Flow

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. Architectural scars.

Software Development Antipatterns – Lava Flow

Refactored Solution

- Ensure that sound architecture precedes production code development.
- In cases where Lava Flow already exists, the cure can be painful. An important principle is to avoid architecture changes during active development.
- 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.
- To avoid Lava Flow, it is important to establish system-level software interfaces that are stable, well-defined, and clearly documented.

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.

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

Software Architecture Antipatterns – Stovepipe Enterprise

Stovepipe Enterprise - Stovepipe is a popular term used to describe software systems with ad hoc architectures.

Template:

- AntiPattern Name: Stovepipe Enterprise
- Also Known As: Islands of Automation
- Most Frequent Scale: Enterprise
- Refactored Solution Name: Enterprise Architecture Planning
- Refactored Solution Type: Process
- Root Causes: Haste, Apathy, Narrow-Mindedness
- Unbalanced Forces: Management of Change, Resources, Technology Transfer
- Anecdotal Evidence: "Can I have my own island (of automation)?"

Software Architecture Antipatterns – Stovepipe Enterprise

Symptoms and Consequences

- Incompatible terminology, approaches, and technology between enterprise systems.
- Brittle, monolithic system architectures and undocumented architectures.
- Inability to extend systems to support business needs.
- Incorrect use of a technology standard.
- Lack of software reuse between enterprise systems.
- Lack of interoperability between enterprise systems.
- Inability of systems to interoperate even when using the same standards.
- Excessive maintenance costs due to changing business requirements; the need to extend the system to incorporate new products and technologies.
- Employee turnover, which causes project discontinuity and maintenance problems.

Software Architecture Antipatterns – Stovepipe Enterprise

Typical Causes

- Lack of an enterprise technology strategy, specifically:
- Lack of a standard reference model
- Lack of system profiles
- Lack of incentive for cooperation across system developments
- Lack of communication between system development projects.
- Lack of knowledge of the technology standard being used.
- Absence of horizontal interfaces in system integration solutions.

Software Architecture Antipatterns – Stovepipe Enterprise

Refactored Solution

Coordination of technologies at several levels is essential to avoid a Stovepipe Enterprise. Initially, the selection of standards can be coordinated through the definition of a standards reference model.

The standards reference model defines,

- The common standards and a migration direction for enterprise systems.
- The establishment of a common operating environment coordinates the selection of products and controls the configuration of product versions.
- Defining system profiles that coordinate the utilization of products and standards is essential to assure standards benefits, reuse, and interoperability.
- At least one system profile should define usage conventions across systems.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In

Template

- AntiPattern Name: Vendor Lock-In
- Also Known As: Product-Dependent Architecture, Bondage and Submission, Connector Conspiracy
- Most Frequent Scale: System
- Refactored Solution Name: Isolation Layer
- Refactored Solution Type: Software
- Root Causes: Sloth, Apathy, Pride/Ignorance (Gullibility)
- Unbalanced Forces: Management of Technology Transfer, Management of Change

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In

- A software project adopts a product technology and becomes completely dependent upon the vendor's implementation. When upgrades are done, software changes and interoperability problems occur, and continuous maintenance is required to keep the system running.

- In addition, expected new product features are often delayed, causing schedule slips and an inability to complete desired application software features.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In – Symptoms and Consequences

- Commercial product upgrades drive the application software maintenance cycle.
- Promised product features are delayed or never delivered, subsequently, causing failure to deliver application updates.
- The product varies significantly from the advertised open systems standard.
- If a product upgrade is missed entirely, a product repurchase and reintegration is often necessary.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In – Typical Causes

- The product varies from published open system standards because there is no effective conformance process for the standard.
- The product is selected based entirely upon marketing and sales information, and not upon more detailed technical inspection.
- There is no technical approach for isolating application software from direct dependency upon the product.
- Application programming requires in-depth product knowledge.
- The complexity and generality of the product technology greatly exceeds that of the application needs; direct dependence upon the product results in failure to manage the complexity of the application system architecture.

Software Architecture Antipatterns – Vendor Lock-In

Vendor Lock-In – Refactored Solution

The refactored solution to the Vendor Lock-In AntiPattern is called *isolation layer.* An isolation layer separates software packages and technology. This solution is applicable when one or more of the following conditions apply:

- *Isolation of application software from lower-level infrastructure.* This infrastructure may include middleware, operating systems, security mechanisms, or other low-level mechanisms.
- *Changes to the underlying infrastructure are anticipated within the life cycle of the affected software;* for example, new product releases or planned migration to new infrastructure.
- *A more convenient programming interface is useful or necessary.* The level of abstraction provided by the infrastructure is either too primitive or too flexible for the intended applications and systems.
- *There a need for consistent handling of the infrastructure across many systems.* Some heavyweight conventions for default handling of infrastructure interfaces must be instituted.
- Multiple infrastructures must be supported, either during the life cycle or concurrently.

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.

The areas where managers play vital role,

1. Software process management
2. Resource management (human & IT infrastructure)
3. External relationship management (e.g., customers, development partners)

Project Management AntiPatterns – Analysis Paralysis

Template:

- AntiPattern Name: Analysis Paralysis
- Also Known As: Waterfall, Process Mismatch
- Most Frequent Scale: System
- Refactored Solution Name: Iterative-Incremental Development
- Refactored Solution Type: Software
- Root Causes: Pride, Narrow-Mindedness
- Unbalanced Forces: Management of Complexity
- Anecdotal Evidence: "We need to redo this analysis to make it more object-oriented, and use much more inheritance to get lots of reuse." "We need to complete object-oriented analysis, and design before we can begin any coding."

Project Management AntiPatterns – Analysis Paralysis

Symptoms and Consequences:

- There are multiple project restarts and much model rework, due to personnel changes or changes in project direction.
- Design and implementation issues are continually reintroduced in the analysis phase.
- Cost of analysis exceeds expectation without a predictable end point.
- The analysis phase no longer involves user interaction. Much of the analysis performed is speculative.
- The complexity of the analysis models results in intricate implementations, making the system difficult to develop, document, and test.
- Design and implementation decisions such as those used in the Gang of Four design patterns are made in the analysis phase.

Project Management AntiPatterns – Analysis Paralysis

Typical Causes:

- The management process assumes a waterfall progression of phases. In reality, virtually all systems are built incrementally even if not acknowledged in the formal process.
- Management has more confidence in their ability to analyze and decompose the problem than to design and implement.
- Management insists on completing all analysis before the design phase begins.
- Goals in the analysis phase are not well defined.
- Planning or leadership lapses when moving past the analysis phase.
- Management is unwilling to make firm decisions about when parts of the domain are sufficiently described.
- The project vision and focus on the goal/deliverable to customer is diffused. Analysis goes beyond providing meaningful value.

Project Management AntiPatterns – Analysis Paralysis

Refactored Solution:

- Key to the success of object-oriented development is incremental development. Whereas a waterfall process assumes a priori knowledge of the problem, incremental development processes assume that details of the problem and its solution will be learned in the course of the development process.

- There are two kinds of increments: internal and external. An internal increment builds software that is essential to the infrastructure of the implementation. For example, a third-tier database and data-access layer would comprise an internal increment. Internal increments build a common infrastructure that is utilized by multiple use cases. In general, internal increments minimize rework. An external increment comprises user-visible functionality.

Project Management AntiPatterns – Death by Planning

Template:

- AntiPattern Name: Death by Planning
- Also Known As: Glass Case Plan, Detailitis Plan
- Most Frequent Scale: Enterprise
- Refactored Solution Name: Rational Planning
- Refactored Solution Type: Process
- Root Causes: Avarice, Ignorance, Haste
- Unbalanced Forces: Management of Complexity
- Anecdotal Evidence: "We can't get started until we have a complete program plan." "The plan is the only thing that will ensure our success."

Project Management AntiPatterns – Death by Planning

Symptoms and Consequences:

Glass Case Plan:

- The symptoms usually include at least one of the following:
- Inability to plan at a pragmatic level.
- Focus on costs rather than delivery.
- Enough greed to commit to any detail as long as the project is funded.
- The consequences are incremental:
- Ignorance of the status of the project's development. The plan has no meaning, and control of delivery lessens as time goes on. The project may be well ahead or behind the intended deliverable state and no one would know.
- Failure to deliver a critical deliverable (the final consequence).

Project Management AntiPatterns – Death by Planning

Symptoms and Consequences:

Detailitis Plan:

- The symptoms are a superset of the Glass Case Plan:
- Inability to plan at a pragmatic level.
- Focus on costs rather than delivery.
- Spending more time planning, detailing progress, and replanning than on delivering software:
- Project manager plans the project's activities.
- Team leaders plan the team's activities and the developers' activities.
- Project developers break down their activities into tasks.

Project Management AntiPatterns – Death by Planning

Typical Causes:

Glass Case Plan

- No up-to-date project plan that shows the software component deliverables and their dates.
- Ignorance of basic project-management principles.
- Overzealous initial planning to attempt to enforce absolute control of development.
- A sales aid for contract acquisition.

Detailitis Plan

- Overzealous continual planning to attempt to enforce absolute control of development.
- Planning as the primary project activity.
- Forced customer compliance.
- Forced executive management compliance.

Project Management AntiPatterns – Death by Planning

Refactored Solution:

The solution is the same for both the Glass Case and Detailitis Plans. A project plan should show primarily deliverables (regardless of how many teams are working on the project). Deliverables should be identified at two levels:

- *Product(s)*. Those artifacts sold to a customer, which include the internal corporate lines of business that use them.
- *Components (within products)*. Basic technology artifacts required to support a business service.

Deliverables include such things as:

- Business requirements statement
- Technical description
- Measurable acceptance criteria
- Product usage scenarios
- Component use cases

References

- https://sourcemaking.com/antipatterns/
- Antipatterns-Refactoring-Software-Architectures-and-Proj.pdf

THANK YOU

Department of Computer Science and Engineering