AntiPatterns

4010-362

Engineering of Software Subsystems

Reference

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis
William J Brown, Raphael C Malveau, Hays W "Skip" McCormick III, Thomas J Mowbray
John Wiley & Sons, 1998

AntiPatterns

- ✓**A pattern of practice that is commonly found in use**
- ✓**A pattern which when practiced usually results in *negative* consequences**
- ✓**Patterns defined in several categories of software development**
  - ***Design***
  - ***Architecture***
  - ***Project Management***

Purpose for AntiPatterns

- ✓Identify problems
- ✓Develop and implement strategies to fix
  - Work incrementally
  - Many alternatives to consider
  - Beware of the cure being worse than the disease

Forces Creating Anti-Patterns

- ✓**Management of**
  - *Functionality*
  - *Performance*
  - *Complexity*
  - *Change*
  - *IT resources*
  - *Technology transfer*

Pattern vs. AntiPattern

- ✓**Patterns**
  - *Usually bottom up*
  - *Begin with recurring solution*
  - *Then the forces and context*
  - *Usually leads to one solution*
- ✓**AntiPatterns**
  - *Top down*
  - *Begin with commonly recurring practice*
  - *Obvious negative consequences*
  - *Symptoms are past and present; consequences go into the future*

Software Design AntiPatterns

- ✓ **AntiPatterns**
  - *The Blob*
  - *Lava Flow*
  - *Functional Decomposition*
  - *Poltergeists*
  - *Golden Hammer*
  - *Spaghetti Code*
  - *Copy-and-Paste Programming*
- ✓ **Mini-AntiPatterns**
  - *Continuous Obsolescence*
  - *Ambiguous Viewpoint*
  - *Boat Anchor*
  - *Dead End*
  - *Input Kludge*
  - *Walking through a Minefield*
  - *Mushroom Management*

Refactoring – Preview

- ✓Design AntiPatterns are solved by refactoring
- ✓AntiPattern provides a useful refactoring
- ✓Refactoring
  - *Natural activity*
  - *Places structure back into the system*
  - *Do before performance optimization*
    - Often compromises structure
    - Refactoring limits to small portion

The Blob

- ✓**AKA**
  - ***Winnebago, The God Class, Kitchen Sink Class***
- ✓**Anecdotal Evidence:**
  - "This class is the heart of our system."

The Blob (2)

- ✓Like the blob in the movie can consume entire strucutres, i.e. your O-O architecture
- ✓Symptoms
  - *Single controller class, multiple simple data classes*
  - *No object-oriented design, i.e. all in main*
  - *Start with a legacy design*
- ✓Problems
  - *Too complex to test or reuse*
  - *Expensive to load into system*
- ✓Procedural design → separates process from data
  - *OO design merges process and data*

Causes

- ✓Lack of OO architecture
- ✓Lack of any architecture
- ✓Lack of architecture enforcement
- ✓Limited refactoring intervention
- ✓Iterative development
  - *Proof-of-concept to prototype to production*
  - *Allocation of responsibilities not repartitioned*

Solution

- ✓Identify or categorize related attributes and operations
- ✓Migrate functionality to data classes

Lava Flow

- ✓**AKA**
  - ***Dead Code***
- ✓**Anecdotal Evidence**
  - "Oh that! I don't think it's used anywhere now, but I'm not really sure. It is really not documented clearly, so we figured we would just leave well enough alone for now. After all, it works."
- ✓**Code, like lava, is fluid when it starts life then becomes hard and immovable later**

Symptoms and Consequences

- ✓Unjustifiable variables and code fragments
- ✓Undocumented complex, important-looking functions, classes
- ✓Large commented-out code with no explanations
- ✓Lot's of "to be replaced" code
- ✓Obsolete interfaces in header files
- ✓Proliferates as code is reused

Causes

- ✓Research code moved into production
- ✓Uncontrolled distribution of unfinished code
- ✓No configuration management in place
- ✓Lack of architecture

Solution

- ✓Don’t get to that point
- ✓Have stable, well-defined interfaces
- ✓Slowly remove dead code; gain a full understanding of any bugs introduced
- ✓Strong architecture moving forward

Functional Decompostion

- ✓**AKA**
  - *No OO*
- ✓**Anecdotal Evidence**
  - "This is our 'main' routine, here in the class called Listener."

Symptoms and Consequences

- ✓Non-OO programmers make each subroutine a class
- ✓Classes with functional names
  - *Calculate_Interest*
  - *Display_Table*
- ✓Classes with single method
- ✓No leveraging of OO principles
- ✓No hope of reuse

Causes

- ✓Lack of OO understanding
- ✓Lack of architecture enforcement

Solution

- ✓Perform analysis
- ✓Develop design model that incorporates as much of the system as possible
- ✓For classes outside model:
  - *Single method: find home in existing class where the data resides*
  - *Combine classes*
  - *No state: static function*

Poltergeists

- ✓**AKA**
  - ***Gypsy, Proliferation of Classes***
- ✓**Anecdotal Evidence**
  - "I'm not exactly sure what this class does, but it sure is important."

Symptoms and Consequences

- ✓Transient associations that go “bump-in-the-night”
- ✓Short-lived, stateless classes
- ✓Classes that begin operations but do nothing else
- ✓Classes with control-like names or suffixed with *manager* or *controller*. Only invoke methods in other classes.

Causes

- ✓**Lack of OO experience**
- ✓**Maybe OO is incorrect tool for the job.**
  - "There is no right way to do the wrong thing."

Solution

- ✓Remove Poltergeist altogether
- ✓Move controlling actions to related classes

Copy-and-Paste Programming

- ✓**AKA**
  - ***Clipboard Coding***
- ✓**Anecdotal Evidence**
  - "Hey, I thought you fixed that bug already, so why is it doing this again?"
  - "Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!"

Symptoms and Consequences

- ✓Same software bug reoccurs
- ✓Code can be reused with a minimum of effort
- ✓Causes excessive maintenance costs
- ✓Multiple unique bug fixes develop
- ✓Inflates LOC without reducing maintenance costs

Causes

- ✓Requires effort to create reusable code; must reward for long-term investment
- ✓Development speed overshadows all other factors
- ✓“Not-invented-here” reduces reuse
- ✓People unfamiliar with new technology or tools just modify a working example

Solution

- ✓Code mining to find duplicate sections of code
- ✓Refactoring to develop standard version
- ✓Configuration management to assist in prevention of future occurrence

Golden Hammer

- ✓**AKA**
  - ***Old Yeller***
- ✓**Anecdotal Evidence**
  - "Our database is our architecture"
  - "Maybe we shouldn't have used Excel macros for this job after all."

Symptoms and Consequences

- ✓**Identical tools for conceptually diverse problems.**
  - "When your only tool is a hammer everything looks like a nail."
- ✓**Solutions have inferior performance, scalability and other 'ilities' compared to other solutions in the industry.**
- ✓**Architecture is described by the tool set.**
- ✓**Requirements tailored to what tool set does well.**

Causes

- ✓Development team is highly proficient with one toolset.
- ✓Several successes with tool set.
- ✓Large investment in tool set.
- ✓Development team is out of touch with industry.

Solution

- ✓Organization must commit to exploration of new technologies
- ✓Commitment to professional development of staff
- ✓Defined software boundaries to ease replacement of subsystems
- ✓Staff hired with different backgrounds and from different areas
- ✓Use open systems and architectures