AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis
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
✓ Develop and implement strategies to fix via refactoring when a design AntiPattern
   – Work incrementally
   – Many alternatives to consider
   – Beware of the cure being worse than the disease
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 structures, i.e. your O-O architecture
✓ Symptoms
  • Single controller class, multiple simple data classes
  • No object-oriented design, i.e. all in main
✓ Problems
  • Too complex to test or reuse
  • Expensive to load into system
✓ Procedural design → separates process from data
  • OO design merges process and data
Solution

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

☑ Same software bug reoccurs
☑ Code can be reused with a minimum of effort
☑ Causes excessive maintenance costs
☑ Multiple unique bug fixes develop
☑ Maintenance costs are not reduced even though code is being “reused”
☑ 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
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

✓ 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
✓ Research code moved into production
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
Poltergeists

✓ AKA
  • *Gypsy, Proliferation of Classes*

✓ Anecdotal Evidence
  • “I’m not exactly sure what this class does, but it sure is important.”
Symptoms

✓ 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.
Solution

✔ Maybe OO is incorrect tool for the job.
  • “There is no right way to do the wrong thing.”
✔ Remove Poltergeist altogether
✔ Move controlling actions to related classes
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

✓ 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.
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
Functional Decomposition

✓ AKA
  • No OO
✓ Anecdotal Evidence
  • “This is our ‘main’ routine, here in the class called Listener.”
Symptoms

✓ 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
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