Lecture 18: AntiPatterns and Refactoring
Definitions

• Pattern: good ideas
• AntiPatterns: bad ideas
• Refactoring: better ideas
Kinds of AntiPatterns

  √ Software Development
  √ Software Architecture
  – Software Project Management
Software Development Anti-Patterns

- Control Freak
- Old Baggage
- Functional Fixation
- Gadget Clutter
- Golden Hammer
- Clipboard Coding
Control Freak

• Description
  – large object with all the control, which accesses simple data classes
    • Blob, Winnebago, God Class
  – essentially a procedural design

• Consequences
  – difficult to modify functionality
  – not based on domain model
  – difficult to test and reuse
Old Baggage

• Description
  – system contains many classes whose purpose is not known
    • Lava Flow, Dead Code
  – much of the code is left over from previous ideas and no longer has a purpose
    • was once fluid and useful, now is solid lava that you are afraid to remove

• Consequences
  – difficult to maintain, just gets worse
Functional Fixation

• **Description**
  – main top level routine
  – main calls other routines that have been made into objects/classes
  – top down development
    • Functional Decomposition

• **Consequences**
  – class models make no sense
  – no O/O benefits such as inheritance and polymorphism
Gadget Clutter

• **Description**
  – transient, ghost-like (no state) classes whose objects are created to perform some temporary responsibility
    • Proliferation of classes, poltergeists, popups
  – unnecessary abstractions that clutter the design

• **Consequences**
  – clutter up the design
  – non-object oriented classes with names like “start up counter”, “initialize data base”
Golden Hammer

• Description
  – a particular technology or product is used for everything because a development team has gained proficiency in it.

• Consequences
  – may leave out awkward by necessary features
  – system architecture fits the preordained solution rather than the problem domain
Spaghetti

• **Description**
  – well known bad detailed design feature for functional code – rats nest of control flows
  – O/O: objects are basically functions and there is little interaction. Single process control weaving its way in and out of objects

• **Consequences**
  – difficult to maintain and understand, lack of code re-use opportunities, does not have an underlying domain model
Clipboard Coding

• Description
  – re-use of code within a system
  – re-use of code from other systems
  – generally a good idea but can be over-used, and without due caution

• Consequences
  – bug duplication
  – incomplete required modification of re-used code
  – failure to identify and create re-usable classes, etc.
Software Architecture
Anti-Patterns

• Coincidental Architecture
• Cover Your Assets
• Bondage and Submission
• Assumed Architecture
• Platypus
• I Did it My Way
Coincidental Architecture

- Terminology and variations
  - combining things in a way that is driven by their properties rather than by a comprehensive idea for an architecture
  - needs constant repairs, patches and adhoc pieces
    - Stovepipe
- Types of:
  - Migration: moving to distributed system
  - System: subsystems integration
  - Enterprise: multiple non-matching, layered systems
Coincidental Architecture and Migration to Client Server

• Description
  – Re-use of interfaces when creating a distributed, client-server system from a single component system

• Consequences
  – interfaces too fine-grained
  – distribution of services determined by stand-alone systems architecture
Coincidental Architecture and Integration

• Description
  – lack of common subsystem abstractions
  – hodgepodge of pairs of subsystem dependencies; complete graph
  – pair-wise integration-driven implementation (as opposed to documented) architecture
    • Legacy, Uncle Sam special, Ad Hoc integration

• Consequences
  – very difficult to understand and maintain
Coincidental Architecture and Enterprise Systems

- Description
  - Enterprise: multiple layered systems
  - Independently designed subsystems at all levels
  - lack of standard reference model across lower layers
    - Islands of automation

- Consequences:
  - brittle, picky integration of subsystems
  - Architecture driven by interfacing heterogeneous system standards and infrastructure
## Enterprise Coincidental Illustration

<table>
<thead>
<tr>
<th></th>
<th>Enterprise System – Version 1</th>
<th>Enterprise System – Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sub A</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sub B</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sub C</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application specific services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value added functional services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic services infrastructure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standards and conventions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Copyright W. Howden*
Cover Your Assets

• Description
  – avoid making decisions
  – list all the alternatives
    • paralysis by analysis
  – course of action not clear

• Consequences
  – not clear what the present and future architecture will be
  – documents will not be used
Bondage and Submission

• Descriptions
  – Architecture based on a product line such as Visual Studio
    • Vendor lock-in

• Consequences
  – lack of control over future product upgrades
  – mixed product versions
  – commercial product updates drive maintenance
Bondage and Submission - Avoidance

• Isolation
  – isolation layer
  – isolation wrapper for preserving standard interface
Assumed Architecture

• Description
  – no specified architecture
    • wherefore art thou architecture?
  – usually similar to experience on previous systems
  – turns out to be too big/complex to do informally

• Consequences
  – hidden risks, misunderstood requirements, problems with backup and contingency plans
Platypus

• Description
  – designed by a committee
    • design by committee, political party, standards disease
  – everyone had to be accommodated
  – poorly run meetings

• Consequences
  – overly verbose, incoherent, no prioritization
I Did it My Way

• Description
  – Lack of design re-use, i.e. not the same as not re-using the wheel
    • re-inventing the wheel, design in a vacuum
  – Failure to carry out domain analysis to find essential features
  – Lack of legacy design documents

• Consequences
  – replication of commercial software
  – Immature and unstable architectures
Refactoring

• Periodically need to clean up the design and the code
• Related to idea of developing code for base increment, and then modifying it for subsequent phases
• Refactoring goals
  – improve cohesion, reduce coupling
• Reference: Martin Fowler, Refactoring, Addison-Wesley, 2000
Refactoring Processes

• Strategic
  – general design

• Tactical
  – detailed design

• Transformations
  – “mechanical” steps
Strategic and Tactical Refactoring Examples

• Strategic
  – Convert Procedural Design to Objects
  – Separate Domain From Presentation

• Tactical
  – Tease Apart Inheritance
  – Extract Hierarchy
Strategic Refactoring: Convert Procedural Design to Objects

• Functional/Procedural Design
  – data is passed from one function to the next, which operates on it and returns data

• Approach
  – convert data entities into objects
  – decompose behavior from functions
  – move behavior to objects
DS Procedural Example
DS Procedural Data (Bottom Up)

- Data Base (implied global)
- GetADate and SetMemberData request data
- Add and Delete Member Data Requests
- Admin request data
- Member request data
- Dating system request data
DS InputOutput

• Input data display
  – input for option choices
  – input for getADate, SetMemberData

• Output data display
  – output from getDate
  – output from setMemberData
  – output from add/delete Member
DS Procedural Objects and Behavior/Methods

• Data Base
  – elementary access function methods

• Requests (GetDate, SetMemberDate, Add/Delete member)
  – Object animation and associated procedural behavior methods
DS GUI Objects and Behavior/Methods

• GUI Frame/Dialog objects and data items

• Behavior and methods
  – frame/dialog construction code for input output data items assigned to initiating frame
  – event/input code assigned to responding object
  – presentation logic behavior assigned to initiating and responding objects
Introduce Architecture

• Identify general design metaphor (e.g. 3 tier/layer)
  – identify possible layers from procedural model
• Create interfaces
• Distribute objects/classes
• Edit and revise as necessary
Strategic Refactoring: Separate Domain From Presentation

• Need to separate domain logic from presentation logic (e.g. going from 2 to 3 tier system)

• Starting point
  – create domain objects for each GUI window
  – move or duplicate domain oriented data in/to new domain objects
  – move domain oriented logic to new domain objects
Domain Presentation Separation and System Architecture

• GUI will make calls to logic moved to domain object

• Domain interface needs to be determined
  – calls may be made to facade for whole domain or subset of domain objects that support domain interfaces
DS Domain Separation Example

• Assume 2 tier system
  – all domain logic in GUI
  – GUI interacts with Data Base subsystem
• Sample domain data moved to Domain
  – name of administrator that is used to determine user type
• Sample domain logic moved to Domain
  – move all Database calls or SQL to domain logic
DS – Initial Domain Separation Objects

• GUI/Presentation windows

⇒ Domain Logic Objects:

  GUI, MemberCommandsDialog, AdminCommandsDialog,
  SelectDaterPreferencesDialog,
  EnterMemberDataDialog,
  SelectedDateeDialog, Messages, LogOnDialog,

• Architecture

  – Domain Logic facade may contain logic also
DS GUI Domain Logic
⇒ Domain Logic Layer

“Check against special name and user data base to determine user type” ⇒ LogOnDialog domain object

“Search through data base to find a match against specified datee properties” ⇒ MemberCommandsDialog domain object

“Add Member with given name to Data Base” ⇒ AdminCommandsDialog domain object
DS Domain Logic Cleanup

- Initial Domain object creation guidelines are rough
  - names need to be changed, logic associated with more meaningful object from GUI, logic transferred to Domain facade interface. e.g.
    (i) LogOnDialog ⇒ LogOn
    (ii) logic for finding date transferred from MemberCommandsDialog to DateRequest (renamed SelectDaterPreferencesDialog)
    (iii) logic for adding a member transferred to DomainLogic subsystem facade from AdminCommandsDialog
Tactical Refactoring: Tease Apart Inheritance

• 2 or more inheritance hierarchies involved
• merged hierarchies have 2 different purposes
• approach
  – identify major hierarchy
  – extract subordinate hierarchy
  – create instance variable in common superclass
    • point to subordinate hierarchy
DS Tease Apart Example

• Superclass: DateSearchResult
• Hierarchy subtypes
  – date found or not found
  – dater discovered to be frequent dater or not
• Cohesion problems: lowest level classes mix two kinds of behavior/information
• Main hierarchy
  – date found or not found
DS – Initial Hierarchy

DateSearchResult

NoDateFound

NoDate-Frequent

NoDate-NonFrequent

DateeProps

DateeProps-Frequent

DateeProps-NonFrequent
DS – Subordinate Extraction
DS - Simplification

DateSearchResult
  - daterFrequency

NoDateFound

DateeProps

DaterFrequency
Tactical Refactoring:
Extract Hierarchy

• Class grows in size and complexity as new responsibilities are added
• Starts to look like a representative of the Swiss Army Knife Anti-Pattern
• Cohesion is bad
• Usage: create an instance and call one of its methods
Extract Hierarchy Steps

• Create subclasses of the super class for each of the different kinds of behavior

• Create a factory method for the superclass that returns an instance of the appropriate subclass

• Change calls to the superclass constructor to the superclass factory method
Extract Hierarchy and Factory Methods

• Standard use of factory methods
  – factory method in a superclass is defined to return an object defined by an interface
  – when superclass is subtyped, the factory method is overridden to return a concrete class instance, which implements the interface

• Emphasis in this kind of use is on subclass selection/definition at code writing time or system start up (via properties object)
Factory Methods and Extract Hierarchy Re-factoring

• Emphasis is on selection of subclass behavior at run time

• Techniques
  – subtype argument code and switch statement used to create appropriate subclass instance
  – subtype argument is name of class. Use “(type) Class.forName(name).newInstance()” to return instance of selected subtype
• Suppose DomainLogic has a Command class with getDate(), setMemberData(), addMember(), and deleteMember() methods. Command() constructor takes user-type code which it sets and which is then referenced by the methods in their conditional logic.

• DomainLogic facade or other client creates an instance of Command and invokes the appropriate method.
DS – Extract Hierarchy 2

• Create Command subclasses for MemberCommands and AdminCommands
• Create a factory method in Command that references its user-type class variable to create and return the appropriate subclass instance
• Change DomainLogic facade or other clients to call the factory method instead of the constructor. Same methods are called in the subclass instances.
Constructor sets userType attribute variable which is checked by methods to determine behavior.
Refactoring Steps - Categories

- Composing Methods
- Moving Features Between Objects
- Organizing Data
- Simplifying Conditional Expressions
- Making Method Calls Simpler
- Dealing with Generalization
Composing Methods - Examples

• Method creation
  – extract code and make a method from it
    • issues: local variables, parameter references
  – if there are many local variables, turn the method into an object whose class has a compute method
    • class variables for each local variables
    • now it will be easier to extract a new method

• Inlining code
  – taking a small piece of code and removing its method envelope
  – inlining an expression in place of the variable whose value was derived from it
Moving Features
Between Objects - Examples

- Move Method
- Move Attribute
- Extract Class
- InLine Class
- Hide Class Delegate
- Remove Middleman
- Introduce Foreign Method
- Introduce Local Extension
Refactoring Tools

• Simple edit-like tools
  – e.g. changing method names
    – find all possible references
    – change names

• E.g. IntelliJ IDE