ADVANCED SOFTWARE DESIGN
LECTURE 8
DESIGN EVALUATION AND IMPROVEMENT

Dave Clarke
TODAY’S GOALS

Learn techniques for

Design evaluation

General methodology

Anti-patterns and bad code smells

Design improvement

Refactoring
NOTE ON THE SLIDES

These slides list many available anti-patterns and refactorings. Some will be covered in detail, but not all will be covered. The aim of listing so many is to help you find what is relevant for your project.
IMPORTANCE

Designs are never right the first time:

Assume that your design is imperfect.
Techniques are required to analyse designs and improve them in an incremental fashion.

Just as best design practice can be catalogued into patterns, so can design analysis and improvement.

Anti-patterns and Refactoring
BAD ARCHITECTURE

Clients

Information Analysis Client

Document Processing Client

Information Retrieval Client

Services

Ricoh Colour Scanner

HP High-Res Scanner

Kodak Photo Printer

Xerox Colour Printer

Oracle DB

Sybase DB
IMPROVED DESIGN

The diagram illustrates the improved design with the following elements:

- **Trader**
- **Client**
- **Input Device**
- **Output Device**
- **Information Source**

The relationships show that the **Trader** uses the **Client**, and the **Client** uses the **Input Device**, **Output Device**, and **Information Source**. The **Information Source** exchanges with the **Client**.
DESIGN EVALUATION:
WHEN IS A DESIGN GOOD? BAD? UGLY?
HOW TO PINPOINT WEAKNESSES?
GOOD DESIGN

Good design is not an absolute!

Good design is design that copes with change.
THEMIS – A TITANESS

Blindfold – impartiality, no influence from outside factors

Scales – weighing evidence carefully, open process based on explained reasons

Sword – power to make decision
ROLE OF DESIGN EVALUATION

Provide advice for potential implementors – if done after design.

Improve design – if incorporated into design process.

Design evaluation is not an official step in a typical development life-cycle.
APPROACHES

Metrics – class width, hierarchy depth

Heuristics

Checks to see whether design matches analysis

  Use-case walkthroughs, especially with role-play

Checks of pattern usage

Design critiques, similar to art criticism or reviews
USE-CASE WALKTHROUGHS
USE CASE WALKTHROUGH

Walkthrough design artefacts simulating execution at the abstract level of design

Simulate, role-play, prototype.

Checks whether all use cases are covered by design – catch less important corner cases.

Look out for: Incompleteness, Inconsistency, Ambiguity
EVALUATING CHANGE
POSSIBLE FUTURE REQUIREMENTS

Future may bring changes to requirements – challenge: future is unknown

Good design will allow the system to be changed to meet those requirements

Imagine what changes may be required – perhaps suggested by requirements.

After initial identification, prioritise change cases, considering both most likely and most serious.

Test how well design can handle changes – follow through implications of modifications.

small local changes only = good design
EVALUATING PATTERN USAGE
EVALUATING PATTERN USAGE

Pattern design does not need to be evaluated – they represent best practice in their specified contexts.

Pattern bingo – avoid patterns for sake of using patterns.

Evaluate the match between the domain and the pattern.

Determine whether domain needs a well-known pattern.

Determine whether right pattern is being used.

Determine whether pattern is being used right.

Are well-known patterns for domain missing?
ANTI-PATTERNS
ANTI-PATTERNS
ANTI-PATTERNS

A pattern used in social or business operations or software engineering that may be commonly used but is ineffective and/or counterproductive in practice [wikipedia].

http://c2.com/cgi/wiki?AntiPatternsCatalog
CAN YOU THINK OF POSSIBLE ANTI-PATTERNS?
ANTI-PATTERN: THE BLOB
ANTI-PATTERN: THE BLOB

A single class monopolises most of the functionality.

Large class, unrelated attributes and functionality.

Single controller + simple data-object classes.

Migrated (unrefactored) legacy application
THE BLOB: CAUSES

Lack of object-oriented architecture

Lack of any architecture

Lack of architecture enforcement – The Blob grew accidentally, uncontrolled by architectural review

Too limited intervention – adding functionality to existing hierarchy rather than adding new classes and refactoring

Specified disaster – requirements dictated a procedural solution
REFACTORING THE BLOB

LibraryMainControl
- currentCatalog
- currentItem
- userId
- fineAmount
- etc.
  ...
- doInventory
- checkOutItem(Item)
- checkInItem(Item)
- addItem(Item)
- deleteItem(Item)
- printCatalog
- sortCatalog
- searchCatalog(Params)
- editItem
- findItem
- print
- openLibrary
- listCatalogs
- issueLibraryCard
- archiveCatalogs
- calculateLateFine
  ...

Person
- name
- userID
- itemsOut
- fines

Item
- title
- ISBN
- author
- publisher
- cost
- dateIn
- quantity
- etc.
  ...

related methods

related methods

Catalog
- topic
- inventory
  ...

Monday 18 November 13
THE BLOB REFAC'TORED

Person

name
userID
itemsOut
fines

LibraryMainControl

currentCatalog
currentItem
userID
fineAmount
etc
...
doInventory
print
openLibrary
issueLibraryCard
archiveCatalogs
calculateLateFine
...

Item

title
ISBN
author
publisher
cost
dateIn
quantity
etc.
...
checkOutItem
checkInItem
addItem
deleteItem
editItem
findOne

Catalog

topic
inventory
...
printCatalog
sortCatalog
searchCatalog(Params)
listCatalogs
archiveCatalogs
Factor out transient associations (checked out status)
ANTI-PATTERN: FUNCTIONAL DECOMPOSITION

Classes represent functions – a single action.

Misses point of OO architecture, OO principles.

Resembles procedural/functional program.
ANTI-PATTERN: FUNCTIONAL DECOMPOSITION

Move functionality to appropriate class (*noun*) – information expert!

Combine several classes into one.

If class has no state information, move functionality to another class.
ANTI-PATTERN: SWISS ARMY KNIFE

Excessively complex class interface.

Designer attempted to provide functionality to meet all possible needs.

Obscures intended class use.

Define useful sub-interfaces.

Use default parameters – by convention or via language itself.
ORGANISATIONAL ANTI-PATTERNS

Analysis paralysis – devoting disproportionate high effort to the analysis phase of a project

Design by committee – the result of having many contributors to a design, but no unifying vision

Vendor lock-in – making a system excessively dependent on an externally supplied component
PROJECT MANAGEMENT

ANTI-PATTERNS

Avalanche – an inappropriate mashup of Waterfall model and Agile Development techniques

Groupthink – during groupthink, members of the group avoid promoting viewpoints outside the comfort zone of consensus thinking

Over-engineering – spending resources making a project more robust and complex than is needed

Smoke and mirrors – demonstrating unimplemented functions as if they were already implemented

Software bloat – allowing successive versions of a system to demand ever more resources
Bystander apathy – when a requirement or design decision is wrong, but the people who notice this do nothing because it affects a different group of people.
SOFTWARE DESIGN

ANTI-PATTERNS

**Big ball of mud** – a system with no recognisable structure

**Gold plating** – continuing to work on a task or project well past the point at which extra effort is adding value

**Inner-platform effect** – a system so customisable as to become a poor replica of the software development platform

**Interface bloat** – making an interface so powerful that it is extremely difficult to implement

**Magic pushbutton** – coding implementation logic directly within interface code, without using abstraction
PROGRAMMING ANTI-PATTERNS

Cargo cult programming – using patterns and methods without understanding why

Spaghetti code – programs whose structure is barely comprehensible, especially because of misuse of code structures

Lasagna code – programs whose structure consists of too many layers
Base Bean – inheriting functionality from a utility class rather than delegating to it

God object – concentrating too many functions in a single part of the design

Object orgy – failing to properly encapsulate objects permitted unrestricted access to their private parts

Yo-yo problem – a structure that is hard to understand due to excessive fragmentation
METHODOLOGICAL ANTI-PATTERNS

Copy and paste programming – copying and modifying existing code rather than creating generic solutions

Golden hammer/Silver bullet – assuming favourite solution is universally applicable

Not Invented Here – tendency towards reinventing the wheel

Invented Here – due to lack of confidence, tends to dismiss any non-trivial in-house code

Premature Optimisation – coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency

Reinventing the square wheel – failing to adopt an existing solution and instead develop a poor quality custom one
BAD CODE SMELLS
BAD CODE SMELLS

Symptom in the source code that possibly indicates a deeper problem.

Not bugs, per se, but indicate weaknesses in the design.

Bad code smells can be a driving force for refactoring.
BAD CODE SMELL EXAMPLES

Duplicated code.

Long method.

Large class.

Too many parameters.

Übercallback — a callback that is trying to do everything

Complex conditionals — branches that check lots of unrelated conditions and edge cases that don't seem to capture the meaning of a block of code.
Feature envy – a class that excessively uses methods of another class.

Inappropriate intimacy – a class that has dependencies on implementation details of another class.

Freeloader – a class that does too little.
Contrived complexity – forced usage of overly complicated design patterns where simpler design would suffice.

Excessively long identifiers.

Excessively short identifiers.

Excessive use of literals – use named constants or external resource files.
OTHER DESIGN HEURISTICS
DESIGN HEURISTICS I

Distribute system intelligence

Beware of too many accessor methods

Beware of too much non-communicating behaviour

Model the real world whenever possible

Eliminate classes that are outside the system

A class should not depend on its clients

A class should capture one and only one key abstraction – cohesion
DESIGN HEURISTICS II

Keep related data and behaviour in one place – encapsulation

A class must know what it contains, but it should never know who contains it

Inheritance should be used only to model a specialisation hierarchy

Subclasses must have knowledge of their superclass, but superclasses should not know anything about their subclasses

Factor the commonality of data, behaviour and/or interface as high as possible in the inheritance hierarchy

When given a choice in an object-oriented design between a containment hierarchy and an association relationship, choose containment hierarchy
REFACTORING

Disciplined technique for restructuring internal code structure without changing its external behaviour in order to improve its non-functional aspects. e.g., code readability, reduced complexity to improve maintainability, more expressive internal architecture or object model to improve extensibility.

Contrast with practice – little refactoring and more attention to expediently adding new features.

Good, continual refactoring practice makes code easier to extend and maintain.
THE ACT OF REFACTORING

Done by performing a series of basic *micro-refactorings*

- small modifications to program’s source code
- often automated by IDEs

Not all refactorings are improvements. Just tools for making improvement.

http://sourcemaking.com/refactoring
REFACTORING CATEGORIES

Composing methods – package code properly.

Moving features between objects

Organising data

Simplifying conditional expressions

Making method calls simpler

Dealing with generalisation

Big refactorings
REFACTORING COMPOSING METHODS

Frequently, problems come from methods that are too long.

Long methods are troublesome because they often contain lots of information, which gets buried by the complex logic that usually gets dragged in.

Extract Method – extracts a method for a given chunk of code, setting up parameters and bindings

Inline Method – inlines a method body in place, ensuring parameter passing and return are handled properly.
One of the most fundamental, if not the fundamental, decision in object design is deciding where to put responsibilities.

This set of refactorings is all about object’s responsibilities.

**Extract Class** – create a new class and move the relevant fields and methods from the old class into the new class.

**Move Method** – create a new method with a similar body in a class that uses it most.
Refactorings that make working with data easier.

Change Bidirectional Association to Unidirectional – drop the unneeded end of an association

Replace Magic Number with Symbolic Constant – create a constant, name it after the meaning, and replace the number with it.
ENCAPSULATE COLLECTION

Method returns a collection.

Make return a read-only view and provide add/remove methods.
Conditional logic has a way of getting tricky, so here are a number of refactorings you can use to simplify it.

Replace Conditional with Polymorphism – move each leg of a conditional to an overriding method in a subclass. Make the original method abstract.
You have repeated checks for a null value.

```java
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

Replace null value with a null object.
Objects are all about interfaces. Coming up with interfaces that are easy to understand and use is a key skill in developing good object-oriented software.

Add Parameter – add a parameter for an object that can pass on missing information.

Introduce Parameter Object – a group of parameters naturally go together; replace them with an object.

Replace Constructor with Factory Method – you want to do more than simple construction when creating an object.

Replace Error Code with Exception – a method returns a special code that indicates an error; throw an exception instead.
Generalization produces its own batch of refactorings, mostly dealing with moving methods around a hierarchy of inheritance.

**Extract Interface/Subclass/Superclass** – move similar chunks of code together; introduce interfaces for common behaviour.

**Pull Up/Push Down Method** – move method up or down hierarchy
REPLACE DELEGATION WITH INHERITANCE

You are making too many simple calls to a delegate.

Make delegating class a subclass of the delegate.
Previous refactorings involve the individual “moves” of refactoring. What is missing is a sense of the whole “game.”

You are refactoring to some purpose. What does the whole game look like?

**Convert Procedural Design to Objects** – turn data records into objects, break up the behaviour, and move behaviour to the objects.

**Separate Domain from Presentation** – separate the domain logic into separate domain classes.
You have a class that is doing too much work, at least in part through many conditional statements.
CREATE HIERARCHY

Create a hierarchy of subclasses in which each subclass represents a special case
TEASE APART INHERITANCE

You have an inheritance hierarchy that is doing two jobs at once.
TEASE APART INHERITANCE

Create two hierarchies and use delegation to invoke one from the other.