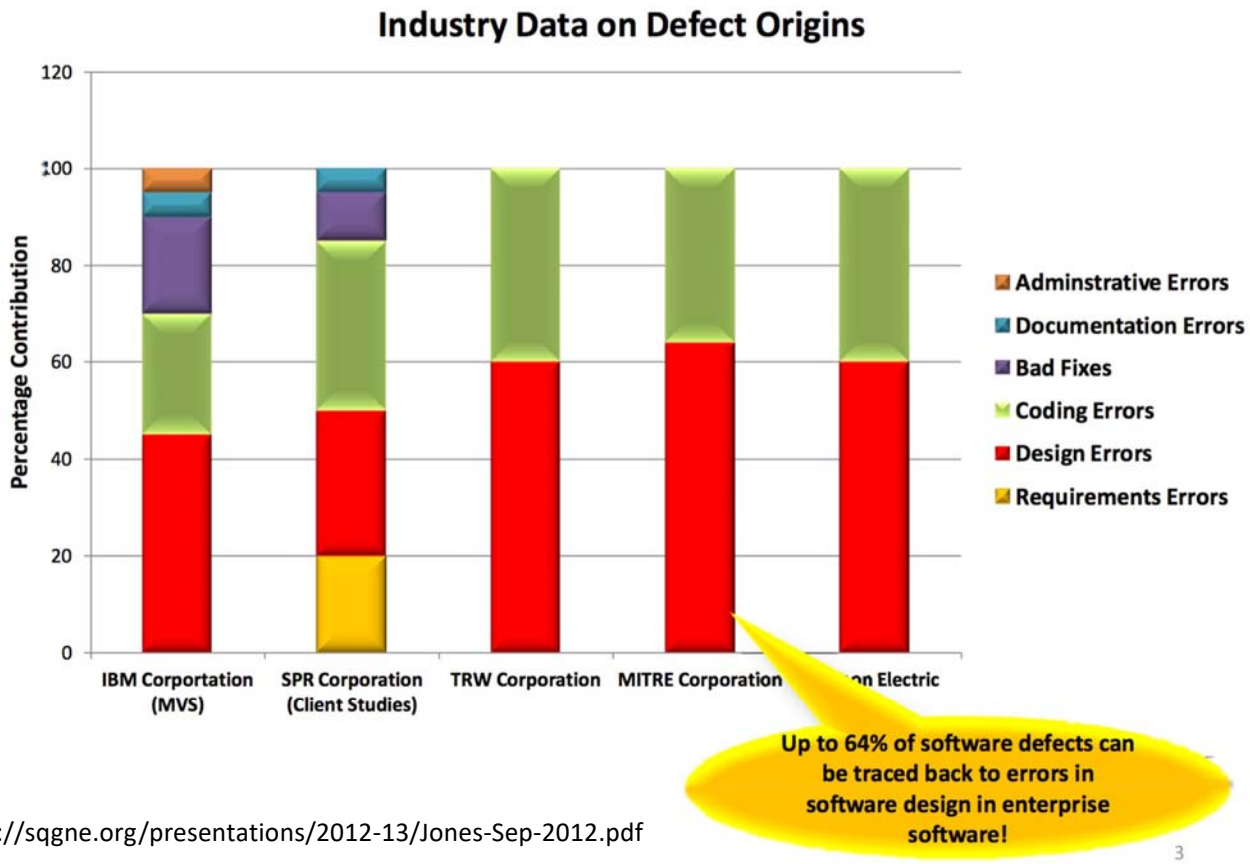
Smells and Refactoring

Code & Design Smells

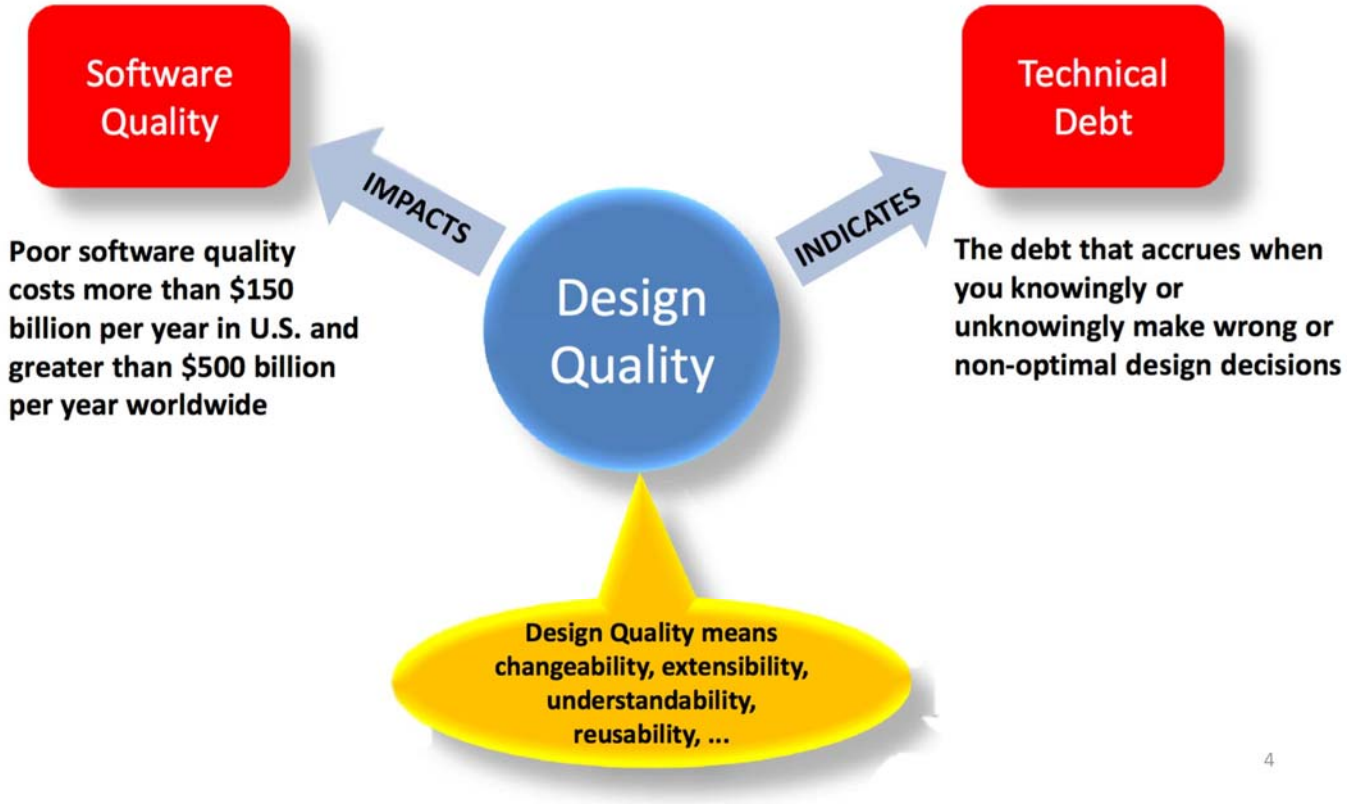
- "Design smells are the odors of rotting software."
- "Code and design smells are poor solutions to recurring implementation and design problems."
- "Smells are certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring."

Design smells indicate the accumulated design debt (one of technical debt).

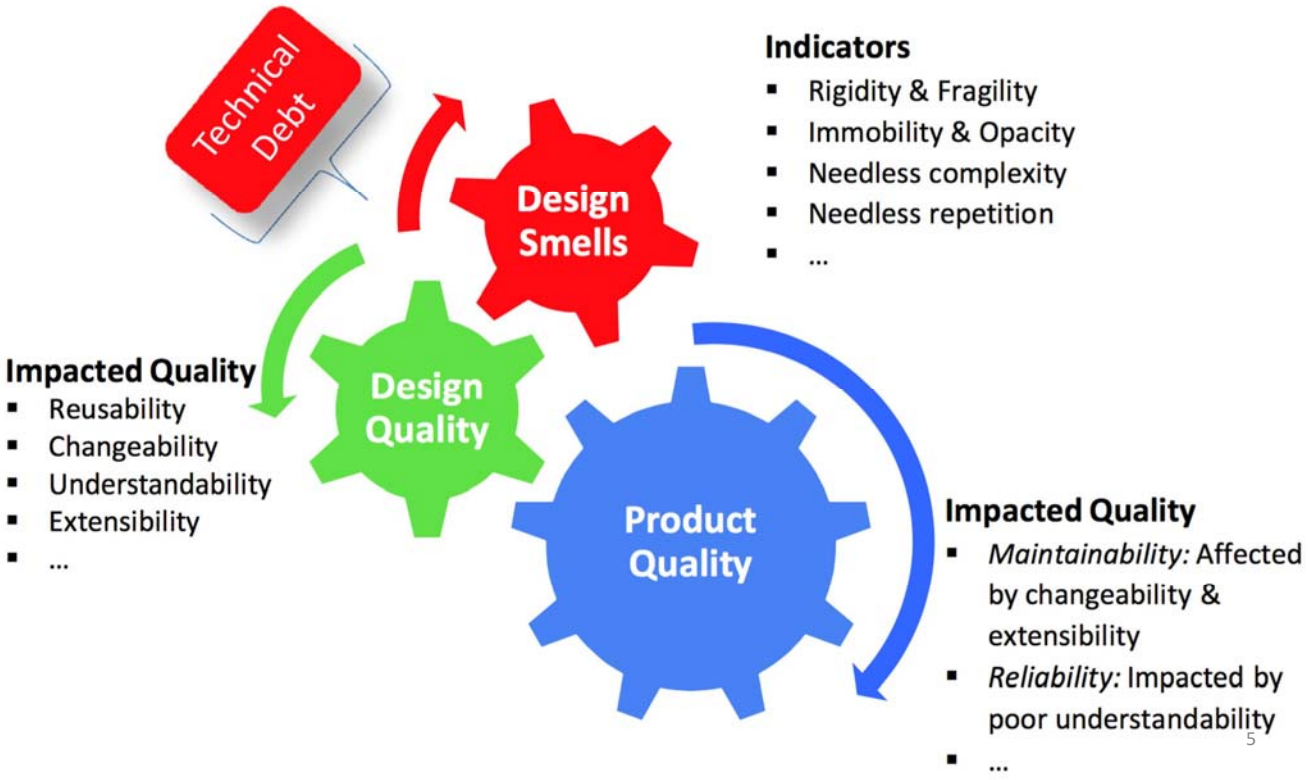


http://sqgne.org/presentations/2012-13/Jones-Sep-2012.pdf

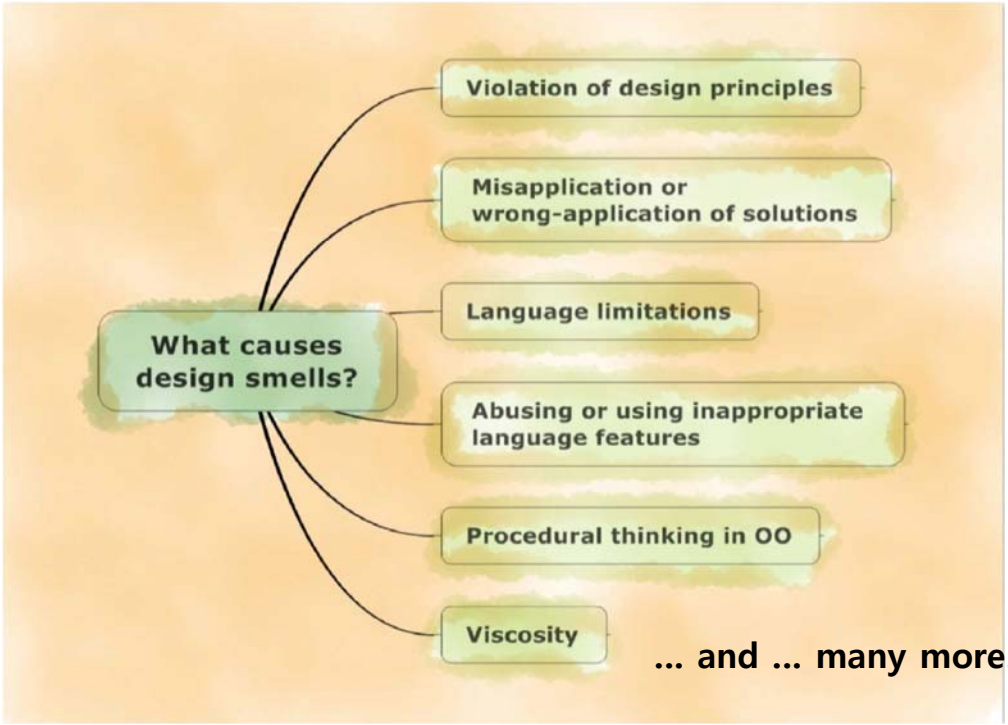
Why Care About Design Quality?



Why Care About Smells?



What Causes Design Smells?



Why Focus on Smells?

- A good designer is one who knows the design solution.
- A GREAT designer is one who understands the impact of design smells and knows how to address them.

7 Deadly Sins of Design

1. Rigidity – make it hard to change
2. Fragility – make it easy to break
3. Immobility – make it hard to reuse
4. Viscosity – make it hard to do the right thing
5. Needless Complexity – over design
6. Needless Repetition – error prone

- What is the 7^{th} sin?

Refactoring: Improving the Design of Existing Code
- Martin Fowler et.al (1999)

CODE SMELLS
&
REFACTORINGS

Code Smells

- Bloaters
- Object-Orientation Abusers
- Change Preventers
- Dispensables
- Couplers

Bloaters

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with.

Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- **Example**
  - Long Method
  - Large Class
  - Primitive Obsession
  - Long parameter List
  - Data Clumps

Long Method

- A method contains too many lines of code
- **Treatment**: *Extract Method*
  - Find parts of the method that seem to perform a single task, and make them into a new method
- Potential problem: You may end up with lots of parameters and temporary variables
  - Temporaries: Consider *Replace Temp With Query*
  - Parameters: Try *Introduce Parameter Object* and *Preserve Whole Object*
  - If all else fails, use *Replace Method With Method Object*

Large Class

- Classes can get overly large
  - Too many instance variables
  - More than a couple dozen methods
  - Seemingly unrelated methods in the same class
- **Treatment**: *Extract Class* and *Extract Subclass*
- A related refactoring, *Extract Interface,* can be helpful in determining how to break up a large class

Primitive Obsession

- Many programmers are reluctant to introduce “little” classes that represent things easily represented by primitives
  - Telephone numbers, zip codes, money amounts, ranges (variables with upper and lower bounds)
  - Use of constants for coding information (such as a constant USER_ADMIN_ROLE = 1)
- **Treatment**: If your primitive needs any additional data or behavior, consider turning it into a class

Long Parameter List

- Long parameter lists are difficult to understand, difficult to remember, and make it harder to format your code nicely
- More than three or four parameters for a method.
- **Treatment**: *Introduce Parameter Object*
- Another solution is called *Replace Parameter With Method*
  - The idea is that you shouldn't pass a parameter into a method if the method has enough information to compute the parameter for itself

Data Clump

- Sometimes different parts of the code contain identical groups of variables (such as parameters for connecting to a database). These clumps should be turned into their own classes.
- Often these data groups are due to poor program structure or "copypasta programming".
- **Treatment**:
  - *Extract Class*
  - *Introduce Parameter Object*
  - *Preserve Whole Object*

Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming and design principles.

- **Examples**
  - Switch Statements
  - Temporary Field
  - Refused Bequest
  - Alternative Classes with Different Interfaces

Switch Statements

- You have a complex switch operator or sequence of if statements
- A switch statement is used to do different things based on some type flag
  - Perhaps the types should be subclasses
- **Treatment**:
  - Use *Extract Method* to isolate the switch statement
  - Use *Move Method* to put it into the class that needs these types
  - Use *Replace Type Code with Subclasses*

Temporary Field

- A variable is in the class scope, when it should be in a method
- We expect an object to use all its fields, but almost always empty
  - It's confusing when an instance variable is used only in certain cases (e.g., to avoid long list of parameters between methods)
- **Treatment**:
  - Use *Extract Class* to create a home for these variables, together with their related behaviors, , achieving the same result as if you would perform **Replace Method with Method Object**.
  - Eliminate conditional code with *Introduce Null Object*

Refused Bequest

- Subclasses may inherit unwanted methods from their superclasses
  - This suggests that the hierarchy may be wrong
  - The purpose of inheritance may be code reuse, not the interface reuse
- Common symptoms
  - Throws an exception like UnsupportedOperationException
- **Treatment**: Create a new subclass and use *Push Down Method* and *Push Down Field* on the unused methods

Alternative Classes with Different Interfaces

- Two classes perform identical functions but have different method names
  - e.g., Java's Enumeration and Iterator classes
- Treatment:
  - *Rename Method* to make them identical
  - *Move Method*, *Add Parameter* and *Parameterize Method* to make the signature and implementation of methods the same
- Extract Only Identical Parts using *Extract Superclass*

Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.

Program development becomes much more complicated and expensive as a result.

- **Example**
  - Divergent Change
  - Shotgun Surgery
  - Parallel Inheritance Hierarchies

Divergent Change

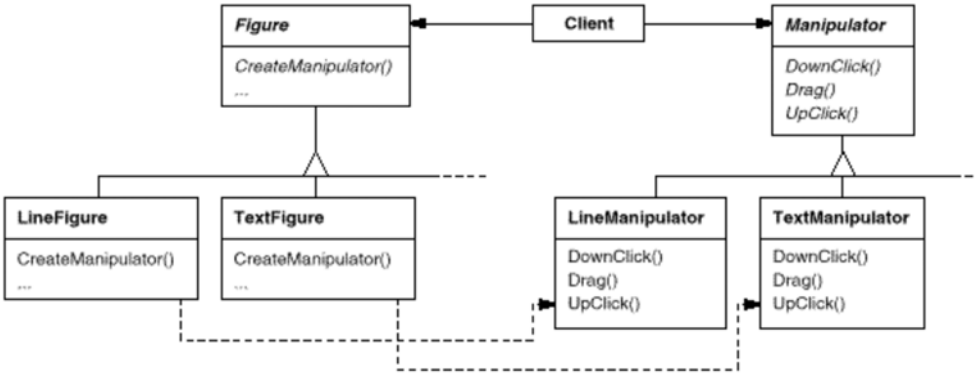
- One class keeps being changed in different ways for different reasons
- Often these divergent modifications are due to poor program structure or "copypasta programming".
- **Treatment**:
  - Identify each cause of change, and use *Extract Class* to isolate the required changes for that cause

Shotgun Surgery

- Every time you make a change, you have to make a lot of little changes in various classes
- **Treatment**:
  - Consolidate Responsibility in a Single Class
    - Use **Move Method** and **Move Field** to move existing class behaviors into a single class. If there is no class appropriate for this, create a new one.
  - Remove Redundant Classes
    - If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes via **Inline Class**.

Parallel Inheritance Hierarchy

- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.
- **Treatment**:
  - Merge Class Hierarchies (If Possible)



Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

- **Example**
  - Duplicate Code
  - Lazy Class
  - Data Class
  - Dead Code
  - Speculative Generality
  - (Comments)

Duplicate Code

- Martin Fowler refers to duplicated code as "**Number one in the stink parade**"
- **Treatment**:
  - The usual solution is to apply *Extract Method:* Create a single method from the repeated code, and use it wherever needed
  - This adds the overhead of method calls, thus the code gets a bit slower
    - Is this a problem?

Lazy Class

- Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted
- **Treatment**:
  - Components that are near-useless should be given the *Inline Class* treatment
  - For subclasses with few functions, try *Collapse Hierarchy*

Data Class

- A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters)
- **Treatment**:
  - *Encapsulate Field* (provide getter/setter)
  - *Encapsulate Collection*
    - Make the method return a *read-only* view (java.util.Collections supplies methods such as unmodifiableSet(Set) and unmodifiableMap(Map))
    - Provide add and remove methods as appropriate
  - Move Client Code Closer to the Data using *Move Method* and *Extract Method*
  - Get Rid of Old Data Access Method using *Remove Setting Method* and *Hide Method*

Dead Code

- A variable, parameter, field, part of conditionals, method or class is no longer used (usually because it is obsolete)
- Nobody had time to clean up the old code
- **Treatment**:
  - The quickest way to find dead code is to use a good IDE
  - Delete unused code and unneeded files
  - *Inline Class* or *Collapse Hierarchy*
  - *Remove Parameter*

Speculative Generality

- There is an unused class, method, field or parameter.
- Sometimes code is created "just in case" to support anticipated future features that never get implemented
- **Treatment**:
  - *Collapse Hierarchy*, *Inline Class*, *Inline Method*, *Remove Parameter*
  - In case of unused fields, simply remove them

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

- **Example**
  - Feature Envy
  - Inappropriate Intimacy
  - Message Chains
  - Middle Man
  - Incomplete Library Class

Feature Envy

- "Feature envy" is when a method makes heavy use of data and methods from another class
- Sometimes only part of the method makes heavy use of the features of another class
- **Treatment**:
  - If a method clearly should be moved to another place, use *Move Method*
  - If only part of a method accesses the data of another object, use *Extract Method* to move the part in question

Inappropriate Intimacy

- Classes may make too much use of each other's fields and methods
- **Treatment**:
  - Use *Move Method* and *Move Field* to reduce the association
  - Try to *Change Bidirectional Association to Unidirectional*
  - If the classes have common needs, try *Extract Class*
  - Use *Hide Delegate* to let another class act as a middle man
  - If a subclass knows too much about its superclass, use *Replace Inheritance With Delegation*

Message Chains

- A message chain occurs when a client requests another object, that object requests yet another one, and so on.
- These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client

```
class C {
    public int f(A a) {
        return a.getB().getC().getD().getValue() ;
    }
}
```

- **Treatment**: *Hide Delegate*

Middle Man

- Delegation—providing methods to call methods in another class—is often useful for hiding internal details

```
void makeLineItem() { sale.makeLineItem(); }
```

  However, too much delegation isn't good
- **Treatment**:
  - Use *Remove Middle Man* and talk to the object that really knows what is going on
  - Use *Inline Method* to absorb a few small methods into the caller

Incomplete Library Class

- Library classes (such as those supplied by Sun) don't always do everything we want them to do
  - It's usually impossible to modify these library classes
- **Treatment**: Use *Introduce Foreign Method:*
  - Write the method you want, as if it were in the library class

```
private static Date nextDay(Date arg) {
     // foreign method, should be in Date
     return new Date(arg.getYear(), arg.getMonth(),
                                arg getDate() + 1);
}
```


Code Refactorings

1. Composing Methods
2. Moving Features Between Objects
3. Organizing Data
4. Simplifying Control Expressions
5. Simplifying Method Calls
6. Dealing with Generalization

Composing Methods

- Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand – and even harder to change.
- The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.
- **Example**
  - Extract Method, Inline Method, Extract Variable, Inline Temp, Replace Temp with Query, Split Temporary Variable, Remove Assignments to Parameters, Replace Method with Method Object, Substitute Algorithm

Moving Features Between Objects

- Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope.
- These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.
- **Example**
  - Move Method, Move Field, Extract Class, Inline Class, Hide Delegate, Remove Middle Man, Introduce Foreign Method, Introduce Local Extension

Organizing Data

- These refactoring techniques help with data handling, replacing primitives with rich class functionality.
- Another important result is untangling of class associations, which makes classes more portable and reusable.
- **Example**
  - Self Encapsulate Field, Replace Data Value with Object, Change Value to Reference, Change Reference to Value, Replace Array with Object, Duplicate Observed Data, Change Unidirectional Association to Bidirectional, Change Bidirectional Association to Unidirectional, Replace Magic Number with Symbolic Constant, Encapsulate Field, Encapsulate Collection, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Subclass with Fields

Simplifying Control Expressions

- Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.
- **Example**
  - Decompose Conditional, Consolidate Conditional Expression, Consolidate Duplicate Conditional Fragments, Remove Control Flag, Replace Nested Conditional with Guard Clauses, Replace Conditional with Polymorphism, Introduce Null Object, Introduce Assertion

Simplifying Method Calls

- These techniques make method calls simpler and easier to understand.
- This, in turn, simplifies the interfaces for interaction between classes.
- **Example**
  - Rename Method, Add Parameter, Remove Parameter, Separate Query from Modifier, Parameterize Method, Replace Parameter with Explicit Methods, Preserve Whole Object, Replace Parameter with Method Call, Introduce Parameter Object, Remove Setting Method, Hide Method, Replace Constructor with Factory Method, Replace Error Code with Exception, Replace Exception with Test

Dealing with Generalization

- Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.
- **Example**
  - Pull Up Field, Pull Up Method, Pull Up Constructor Body, Push Down Method, Push Down Field, Extract Subclass, Extract Superclass, Extract Interface, Collapse Hierarchy, Form Template Method, Replace Inheritance with Delegation, Replace Delegation with Inheritance

List of Refactorings [F] (1/2)

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Expression
- Convert Procedural Design to Objects
- Decompose Conditional
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Hierarchy
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method
- Inline Temp
- Introduce Assertion
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Introduce Null Object
- Introduce Parameter Object
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object

List of Refactorings [F] (2/2)

- Pull Up Constructor Body Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Remove Assignment to Parameters
- Remove Control Flag
- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Rename Method
- Replace Array with Object
- Replace Conditional with Polymorphism
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation with Inheritance
- Replace Error Code with Exception
- Replace Exception with Test
- Replace Inheritance with Delegation
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
- Replace Nested Conditional with Guard Clause
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Record with Data Class
- Replace Subclass with Fields
- Replace Temp with Query
- Replace Type Code with Class
- Replace Type Code with State/Strategy
- Replace Type Code with Subclasses
- Self Encapsulate Field
- Separate Domain from Presentation
- Split Temporary Variable
- Substitute Algorithm
- Tease Apart Inheritance

Smells & Refactorings

Smell	Common Refactorings
Alternative Classes with Different Interfaces	Rename Method, Move Method
Comments	Extract Method, Introduce Assertion
Data Class	Move Method, Encapsulate Field, Encapsulate Collection
Data Clumps	Extract Class, Introduce Parameter Object, Preserve Whole Object
Divergent Change	Extract Class
Duplicated Code	Extract Method, Extract Class, Pull Up Method, Form Template Method
Feature Envy	Move Method, Move Field, Extract Method
Inappropriate Intimacy	Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation, Hide Delegate
Incomplete Library Class	Introduce Foreign Method, Introduce Local Extension
Large Class	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Lazy Class	Inline Class, Collapse Hierarchy
Long Method	Extract Method, Replace Temp with Query, Replace Method with Method Object, Decompose Conditional
Long Parameter List	Replace Parameter with Method, Introduce Parameter Object, Preserve Whole Object
Message Chains	Hide Delegate
Middle Man	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Parallel Inheritance Hierarchies	Move Method, Move Field
Primitive Obsession	Replace Data Value with Object, Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy
Refused Bequest	Replace Inheritance with Delegation
Shotgun Surgery	Move Method, Move Field, Inline Class
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method
Switch Statements	Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object
Temporary Field	Extract Class, Introduce Null Object

Refactoring to Patterns, Joshua Krievsky (2005)

{CODE, DESIGN}
SMELLS &
REFACTORINGS

Code Smells

- 22 Code Smells by Martin Fowler and Kent Beck
- 12 Smells by Joshua Kerievsky
  - 7 from Fowler + 5 by Kerievsky

Smells

1. Duplicated Code [F]
2. Long Method [F]
3. Conditional Complexity
4. Primitive Obsession [F]
5. Indecent Exposure
6. Solution Sprawl
7. Alternative Classes with Different Interfaces [F]
8. Lazy Class [F]
9. Large Class [F]
10. Switch Statements [F]
11. Combinatorial Explosion
12. Oddball Solution

Duplicated Code [F]

The most pervasive and pungent smell in software. It tends to be either explicit or subtle.

Treatment

- Chain Constructors (340)
- Replace One/Many Distinctions with Composite (224)
- Extract Composite (214)
- Unify Interfaces with Adapter (247)
- Introduce Null Object (301)

Long Method [F]

Short methods are
- superior in sharing logic
- easier to understand and contain less duplication
- easier to extend and maintain

Treatment
- Compose Method (123) w/ Extract Method [F]
- Move Accumulation to Collecting Parameter (313)
- Replace Conditional Dispatcher with Command (191)
- Move Accumulation to Visitor (320).
- Replace Conditional Logic with Strategy (129)

Conditional Complexity

Adding several new features over time may cause conditional logic to becomes complicated and expansive.

Treatment
- Replace Conditional Logic with Strategy (129)
- Move Embellishment to Decorator (144)
- Replace State-Altering Conditionals with State (166)
- Introduce Null Object (301)

Primitive Obsession [F]

Code relies too much on primitives. Primitives are generic while classes are more specific and natural to model reality.

Treatment

- Replace Type Code with Class (286)
- Replace State-Altering Conditionals with State (166)
- Replace Conditional Logic with Strategy (129)
- Replace Implicit Tree with Composite (178)
- Replace Implicit Language with Interpreter (269)
- Move Embellishment to Decorator (144)
- Encapsulate Composite with Builder (96)

Indecent Exposure

Methods or classes that ought not be visible to clients are publicly visible to them.

Treatment

- Encapsulate Classes with Factory (80)

Solution Sprawl

Code and/or data used to perform a responsibility becomes sprawled across numerous classes.

Identical twin brother of Shotgun Surgery [F]

Causes

- Hasty addition of features to a system without discretion

Treatment

- Move Creation Knowledge to Factory (68)

Alternative Classes with Different Interfaces [F]

The interfaces of two classes are different and yet the classes are quite similar.

Treatment

- Make them share a common interface.
- Unify Interfaces with Adapter (247)

Lazy Class [F]

A class that isn't doing enough to pay for itself should be eliminated.

Singletons may be costing you something by making your design too dependent on what amounts to global data.

Treatment

- Inline Singleton (114)

Large Class [F]

In general, large classes typically contain too many responsibilities and/or instance variables.

Treatment

- Extract Class [F] and Extract Subclass [F], etc.
- Replace Conditional Dispatcher with Command (191)
- Replace State-Altering Conditionals with State (166)
- Replace Implicit Language with Interpreter (269)

Switch Statements [F]

Switch statements (or nested-ifs) become bad only when they make your design more complicated or rigid than it needs to be.

Treatment

- Replace Conditional Dispatcher with Command (191)
- Move Accumulation to Visitor (320)

Combinatorial Explosion

Explosion of methods to handle the many ways of handling different kinds or quantities of data or objects.

Treatment

- Replace Implicit Language with Interpreter (269)

Oddball Solution

The same problem is solved in different ways.

- Subtly duplicated code

Causes

- Inconsistent interfaces

Treatment

- Unify Interfaces with Adapter (248)
- Substitute Algorithm [F]

Classification of Refactoring Patterns

- Creation (6)
- Simplification (6)
- Generalization (7)
- Protection (3)
- Accumulation (2)
- Utilities (3)

Total: 27

Creation

- While every OO system creates objects or object structures, the creation code is not always free of duplication, simple, intuitive, or as loosely coupled to client code as it could be.

Creation (Cont'd)

- **Replace Constructors with Creation Methods** (57) can clarify the intention of the constructors.
- **Polymorphic Creation with Factory Method** (88) can remove duplicate code If classes in a hierarchy implement a method similarly, except for an object creation step.
- **Move Creation Knowledge to Factory** (68) will reduce creational sprawl by consolidating creation code and data under a single Factory.
- **Encapsulate Classes with Factory** (80) ensures that clients communicate with objects via a common interface and reduce client knowledge of classes while making objects accessible via a Factory.
- **Encapsulate Composite with Builder** (96) shows how a Builder can provide a simpler, less error-prone way to construct a Composite [DP].
- **Inline Singleton** (114) shows how to remove a Singleton from code.

Simplification

- Much of the code we write doesn't start out being simple. To make it simple, we must reflect on what isn't simple about it and continually ask, "How could it be simpler?"
- The refactorings in this category present different solutions for simplifying methods, state transitions, and tree structures.

Simplification (Cont'd)

- **Compose Method** (123) produces Composed Method methods that efficiently communicate what they do and how they do what they do. A Composed Method consists of calls to well-named methods that are all at the same level of detail.
- **Replace Conditional Logic with Strategy** (129) shows how to simplify algorithms by breaking them up into separate classes.
- **Move Embellishment to Decorator** (144) shows how to separate embellishments from the core responsibility of a class.
- **Replace State-Altering Conditionals with State** (166) describes how to drastically simplify complex state transition logic.
- **Replace Implicit Tree with Composite** (178) shows how a Composite [DP] can simplify a client's creation and interaction with a tree structure.
- **Replace Conditional Dispatcher with Command** (191) shows how completely simplify a switch statement that controls which chunk of behavior to execute.

Generalization

- Generalization is the transformation of specific code into general-purpose code.
- The common motivation is
  - to remove duplicated code (foremost)
  - to simplify or clarify code (secondary)

Generalization (Cont'd)

- **Form Template Method** (205) helps remove duplication in similar methods of subclasses in a hierarchy.
- **Extract Composite** (214) applies Extract Superclass [F] to extract a Composite to a superclass so that the subclasses share one generic implementation.
- **Replace One/Many Distinctions with Composite** (224) allows us to handle one or many objects without distinguishing between the two.
- **Replace Hard-Coded Notifications with Observer** (236) allows instances of other classes to be notified, using Observer [DP].
- **Unify Interfaces with Adapter** (247) makes clients to interact with similar classes using a generic interface.
- **Extract Adapter** (258) produces classes that implement a common interface and adapt a single version of some code.
- **Replace Implicit Language with Interpreter** (269), targets code that would be better designed were it to use an explicit language.

Protection

Motivation may be to improve protection, or to reduce duplication or to simplify or clarify code.

- **Replace Type Code with Class** (286) helps protect a field from assignments to incorrect or unsafe values. (Used during **Replace State-Altering Conditionals with State** (166))
- **Limit Instantiation with Singleton** (296) is useful when you want to control how many instances of a class can be instantiated.
  - Use to improve memory usages or performance, not as a global access point.
- **Introduce Null Object** (301) is a refactoring that helps transform how code is protected from null values.
  - To remove lots of null value check duplication

Accumulation

Target the improvement of code that accumulates information within an object or across several objects.

- A **Collecting Parameter** is an object that visits methods in order to accumulate information.
- **Move Accumulation to Collecting Parameter** (313)
  - *Visited methods write to a Collecting Parameter.*
  - Often used with **Compose Method** (123)
- **Move Accumulation to Visitor** (320)
  - *Visitor gathers data from visited objects.*
  - Only useful for accumulating data from many objects, not one.
  - A Visitor can visit an object structure and add objects to that structure during its journey.

Utilities

Low-level refactorings used by the higher-level refactorings.

- **Chain Constructors** (340) is about removing duplication in constructors by having them call each other.
- **Unify Interfaces** (343) is useful when you need a superclass and/or interface to share the same interface as a subclass. The usual motivation for applying this refactoring is to make it possible to treat objects polymorphically
- **Extract Parameter** (346) is useful when a field is assigned to a locally instantiated value and you'd rather have that value supplied by a parameter.

12 Code Smells and Refactorings

Smell[a]	Refactoring
Duplicated Code (39) [F]	*Form Template Method (205)* *Introduce Polymorphic Creation with Factory Method (88)* *Chain Constructors (340)* *Replace One/Many Distinctions with Composite (224)* *Extract Composite (214)* *Unify Interfaces with Adapter (247)* *Introduce Null Object (301)*
Long Method (40) [F]	*Compose Method (123)* *Move Accumulation to Collecting Parameter (313)* *Replace Conditional Dispatcher with Command (191)* *Move Accumulation to Visitor (320)* *Replace Conditional Logic with Strategy (129)*
Conditional Complexity (41)	*Replace Conditional Logic with Strategy (129)* *Move Embellishment to Decorator (144)* *Replace State-Altering Conditionals with State (166)* *Introduce Null Object (301)*

12 Code Smells and Refactorings

Smell[a]	Refactoring
Primitive Obsession (*41*) [F]	*Replace Type Code with Class (286)* *Replace State-Altering Conditionals with State (166)* *Replace Conditional Logic with Strategy (129)* *Replace Implicit Tree with Composite (178)* *Replace Implicit Language with Interpreter (269)* *Move Embellishment to Decorator (144)* *Encapsulate Composite with Builder (96)*
Indecent Exposure (*42*)	*Encapsulate Classes with Factory (80)*
Solution Sprawl (*43*)	*Move Creation Knowledge to Factory (68)*
Alternative Classes with Different Interfaces (*43*) [F]	*Unify Interfaces with Adapter (247)*

12 Code Smells and Refactorings

Smell[a]	Refactoring
Lazy Class (*43*) [F]	*Inline Singleton (114)*
Large Class (*44*) [F]	*Replace Conditional Dispatcher with Command (191)* *Replace State-Altering Conditionals with State (166)* *Replace Implicit Language with Interpreter (269)*
Switch Statements (*44*) [F]	*Replace Conditional Dispatcher with Command (191)* *Move Accumulation to Visitor (320)*
Combinatorial Explosion (*45*)	*Replace Implicit Language with Interpreter (269)*
Oddball Solution (*45*)	*Unify Interfaces with Adapter (247)*

Pattern	To	Towards	Away
Adapter	*Extract Adapter (258), Unify Interfaces with Adapter (247)*	*Unify Interfaces with Adapter (247)*	
Builder	*Encapsulate Composite with Builder (96)*		
Collecting Parameter	*Move Accumulation to Collecting Parameter (313)*		
Command	*Replace Conditional Dispatcher with Command (191)*	*Replace Conditional Dispatcher with Command (191)*	
Composed Method	*Compose Method (123)*		
Composite	*Replace One/Many Distinctions with Composite (224), Extract Composite (214), Replace Implicit Tree with Composite (178)*		*Encapsulate Composite with Builder (96)*
Creation Method	*Replace Constructors with Creation Methods (57)*		
Decorator	*Move Embellishment to Decorator (144)*	*Move Embellishment to Decorator (144)*	

Pattern	To	Towards	Away
Factory	*Move Creation Knowledge to Factory (68), Encapsulate Classes with Factory (80)*		
Factory Method	*Introduce Polymorphic Creation with Factory Method (88)*		
Interpreter	*Replace Implicit Language with Interpreter (269)*		
Iterator			*Move Accumulation to Visitor (320)*
Null Object	*Introduce Null Object (301)*		
Observer	*Replace Hard-Coded Notifications with Observer (236)*	*Replace Hard-Coded Notifications with Observer (236)*	
Singleton	*Limit Instantiation with Singleton (296)*		*Inline Singleton (114)*
State	*Replace State-Altering Conditionals with State (166)*	*Replace State-Altering Conditionals with State (166)*	
Strategy	*Replace Conditional Logic with Strategy (129)*	*Replace Conditional Logic with Strategy (129)*	
Template Method	*Form Template Method (205)*		
Visitor	*Move Accumulation to Visitor (320)*	*Move Accumulation to Visitor (320)*	

Agile Software Architecture, Chapter3: Refactoring Software Architectures (2014)

ARCHITECTURAL SMELLS & REFACTORINGS

Architectural Smells

1. Breaking the DRY rule
2. Unclear roles of entities
3. Inexpressive or complex architecture
4. Everything centralized
5. “Not invented here” syndrome
6. Over-generic design
7. Asymmetric structure or behavior
8. Dependency Cycles
9. Design violations (such as relaxed layering)
10. Inadequate partitioning
11. Unnecessary dependencies
12. Missing orthogonality

Architectural Refactorings

1. Rename Entities
2. Remove Duplicates
3. Introduce Abstraction Hierarchies
4. Remove Unncessary Abstractions
5. Substitute Mediation with Adaptation
6. Break Dependency Cycles
7. Inject Dependencies
8. Insert Transparency Layer
9. Reduce Dependencies with Facades
10. Merge Subsystems
11. Split Subsystems
12. Enforce Strict Layering
13. Move Entities
14. Add Strategies
15. Enforce Symmetry
16. Extract Interface
17. Enforce Contract
18. Provide Extension Interfaces
19. Substitute Inheritance with Delegation
20. Provide Interoperability Layers
21. Aspectify
22. Integrate DSLs
23. Add Uniform Support to Runtime Aspects
24. Add Configuration Subsystem
25. Introduce the Open/Close Principle
26. Optimize with Caching
27. Replace Singleton
28. Separate Synchronous and Asynchronous Processing
29. Replace Remote Methods with Messages
30. Add Object Manager
31. Change Unidirectional Association to Bidirectional

Architecture & Code Refactorings

- Architecture Refactorings may trigger appropriate Code Refactorings.
  - Architecture refactoring **Rename Entity** will cause appropriate code refactorings such as **Rename Class/Method.**
  - Thus, it is recommendable to document impact of architecture refactoring by specifying required code refactorings.
- Refactorings may overlap. Hence, all refactorings should be processed, one by one, ordered by priorities.
  - Architecture refactorings before code refactorings
  - Priorization of requirements suggests priorization of refactorings
  - Strategic refactorings before tactical refactorings
  - Note: there is even overlap between some architecture and code refactoring patterns.