Refactoring Existing Java Code

Ethan Strominger

NEJUG, 12-Apr-2018

Agenda

- What, Why, When, Why not? Alternatives
- Refactoring Steps
- Creating Automated Tests
- Refactoring Code

What, Why, When, When Not?

What Is Refactoring?

The process of restructuring existing computer code—changing the factoring—**without changing its external behavior**. Refactoring improves nonfunctional attributes of the software. - wikipedia

Types of Refactoring

- Refactoring as you go (part of test driven development)
- Refactoring good code that has not degenerated to make it easier to implement a new requirement
- Refactoring existing “legacy code”

Quiz: Why/When

Why Refactor

- Incrementally improve
  - Understandability
  - Maintainability
  - Extensibility
- Make SOLID (consider Design Patterns)
- Ensure DRY

When Refactor

- New feature needs to be added
- Bug needs to be fixed
- Code review
- Not on critical path
- Boss gives you time!!

How Do I Convince My Boss?

- Include refactoring in all estimates going forward
- "Covert"
  - Evaluate if quicker, take the risk
- Overt
  - Estimate time spent now because of crappy code
  - Estimate how much time will be saved
  - Create 2 or 3 concise slides

When Not to Refactor

- Imminent deadline
- Product is being obsoleted
- Technical debt is too large
  - Too many bugs
  - Code too complex
  - Creating automated tests too difficult

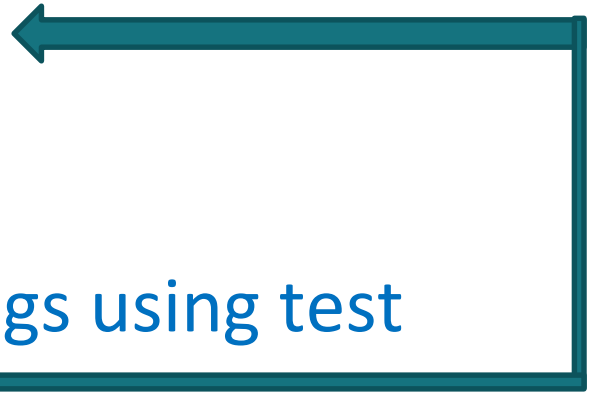
Quiz: Alternatives to Refactoring

Alternatives to Refactoring

Solution	Advantages	Disadvantages
Quick fix / kludge	• Quick	• Unintended consequences • Technical debt continues
Rewrite (Greenfield)	• Implement new approach • Gets around problems of old system	• Risky • Time consuming • Need to support existing customers
Refactor (Brownfield)	• Incremental improvements • Reduce complexity • Improve understandability • Improve maintainability • Improve extensibility	• Allocate time • Balance with need to support customers and deadlines

Refactoring Steps

1. Define strategy
2. Evaluate static code analysis
3. Evaluate and write tests
4. Check code coverage
5. Refactor
6. Add feature and/or fix bugs using test driven development



1. Define Strategy

Strategy Questions

- What factors to use when deciding where to start?
- What is the testing strategy?
- What patterns or approaches will you take to refactoring the code?
- How will you allocate time?

Quiz

Factors for deciding what to change

Factors for Deciding What to Change

Isolatability

Buggggniess

Functional Value

Testability

Frag ility

Change Frequency

Time Allocation

- On going, part of TDD
- Percent time allocation
- Large scale focused refactoring with dedicated time period

Testing Straregy

- AUTOMATE!!!
- What testing tools are you going to use?
- What areas needs beefing up?
- What is your target code coverage?
- What is your target functional testing coverage?

2. Static Code Analysis

Static Code Analysis

- Analyze code for issues without running any tests
- Used for
  - standards enforcement
  - warnings / refactoring suggestions
  - bugs
  - security
- Key features
  - suppress specific findings or categories of findings
  - customizable
  - speed
  - ease of use

Static Code Analysis

- Popular tools for Java: PMD, Checklist, Findbug, SonarJava

- IntelliJ and Eclipse provide formatting, refactoring suggestions (warnings), refactoring tools

IntelliJ vs Eclipse

- IntelliJ Ultimate (Community lacks code coverage)
  - Parameter names displayed in calling function

```
Item[] items = new Item[]{new Item( name: "foo",  sellIn: 0,  quality: 0)};
```

  - Better refactoring
- Eclipse
  - Free
  - Multiple projects in same window
  - Personal preference

3. Evaluate and Write Tests

Testing Pyramid

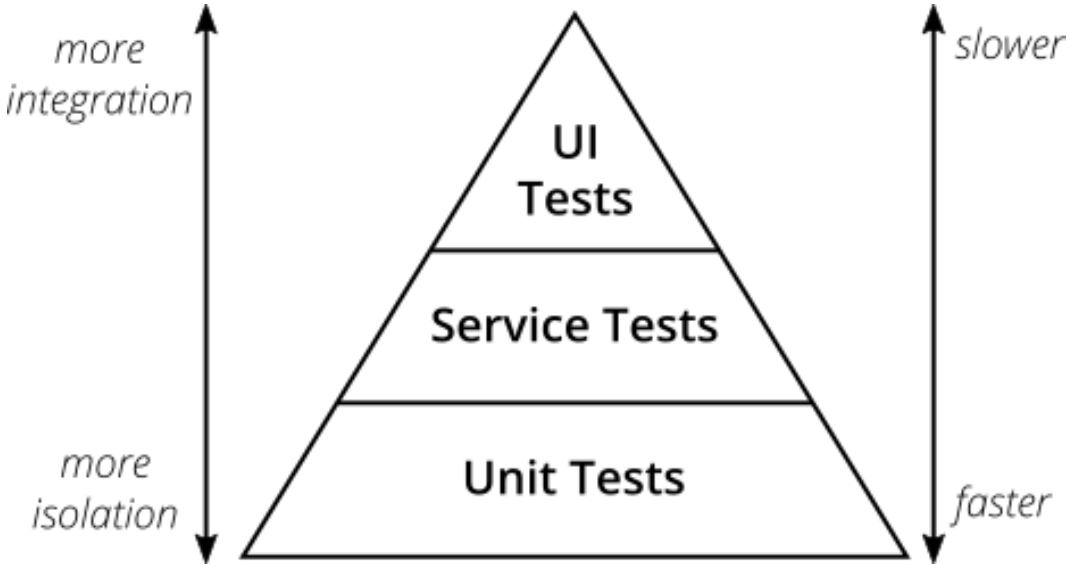


Image derived from https://martinfowler.com/articles/practical-test-pyramid.html
Concept originated from Mike Cohn and Jason Higgins.

Testing Pyramid

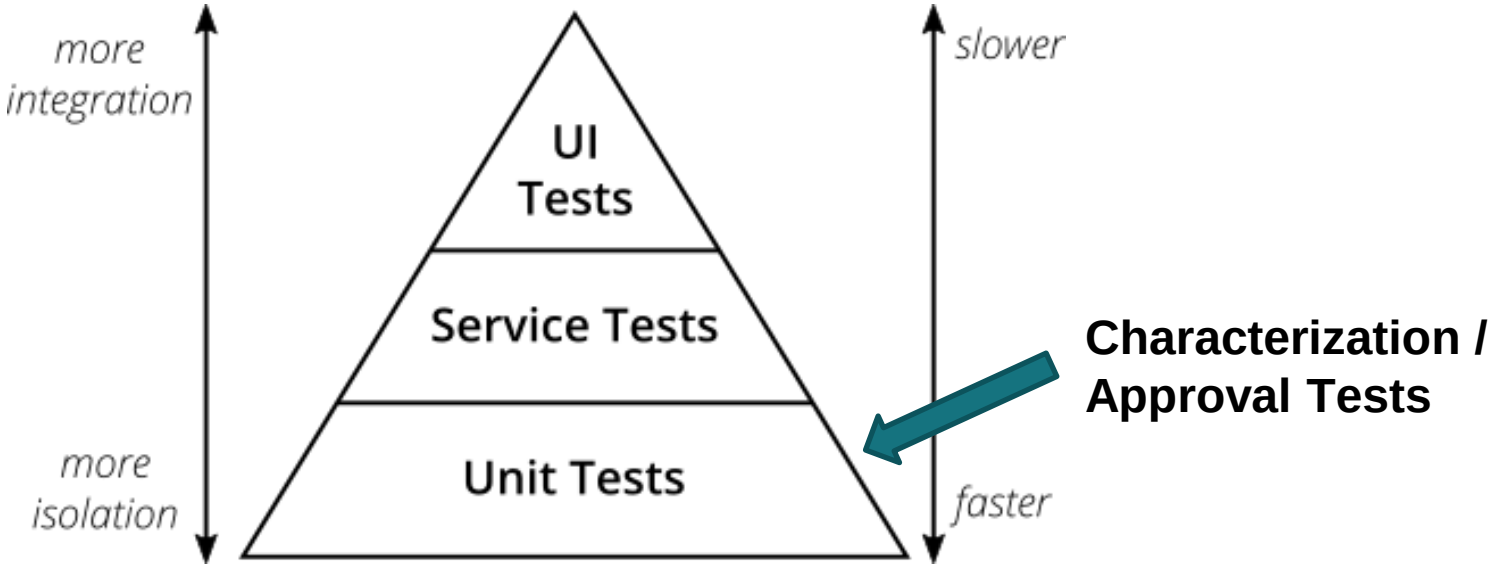
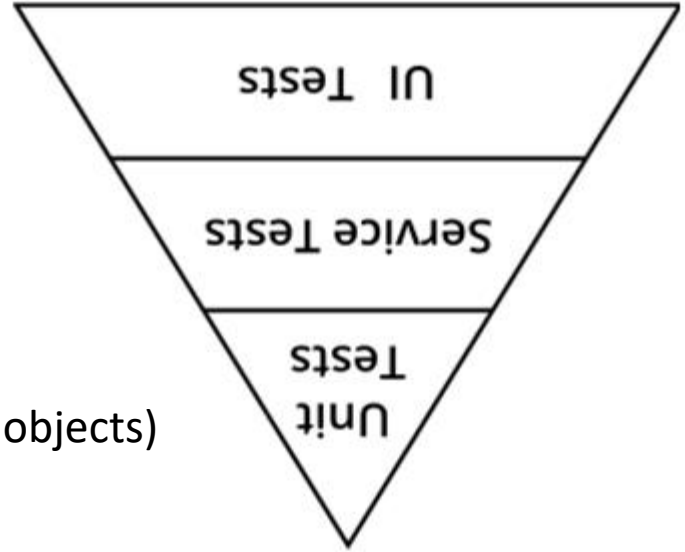


Image derived from https://martinfowler.com/articles/practical-test-pyramid.html
Concept originated from Mike Cohn and Jason Higgins.

Testing Pyramid – Anti-pattern



Contributors:

- Legacy code
- Non OO Design:
  - Low cohesion (inter-dependent code or objects)
  - No APIs
  - Unit tests will not be sufficient => order of calls important

Characterization Tests

Characterization tests are designed to reveal how system currently behaves.

Term coined by Michael Feathers

Approval Tests

Approval tests are characterization tests written in a way you can approve the results of the test automatically.

Term coined by Llewellyn Falco

Unit Test Case Strategies

- One test case for each scenario
- Parameterized tests (list of values for pre-defined test provided)
- Approval testing (capture results of the tests and approve the results)

Gilded Rose Inn Kata

- Inn sells Aged Brie, Sulfaras, Backstage Passes, and other items
- Legacy application tracks daysToExpire and quality, which change every day
  - daysToExpire decreases by 1
  - quality decreases according to next page
- Assignment is to add Conjured Item, which decrements twice as fast as other items

Gilded Rose Requirements

Product	Value of Days to Expire	Change in Quality
Aged Brie	>0 <=0	1 2
Backstage Passes	>10 >5 >0 <=0	1 2 3 Quality goes to 0
Sulfaras	N/A	N/A – always 80
Other	>0 <=0	-1 -2

Quality always between 0 and 50, except for Sulfaras

Demo

- Demo single and parameter tests: show code
- Demo approval tests
  - Run tests, show how results approved
  - Show code for all combinations are tested
  - Show code for sulfaras, with different combos

Refactoring Code

Refactoring

- Definitions
- Example Refactorings
- Literature

Refactoring Definition

The process of restructuring existing computer code—changing the factoring—**without changing its external behavior**. Refactoring improves nonfunctional attributes of the software. - wikipedia

Code Smell Definition

A code smell is a surface indication that usually corresponds to a deeper problem in the system.

- Martin Fowler, https://martinfowler.com/bliki/CodeSmell.html

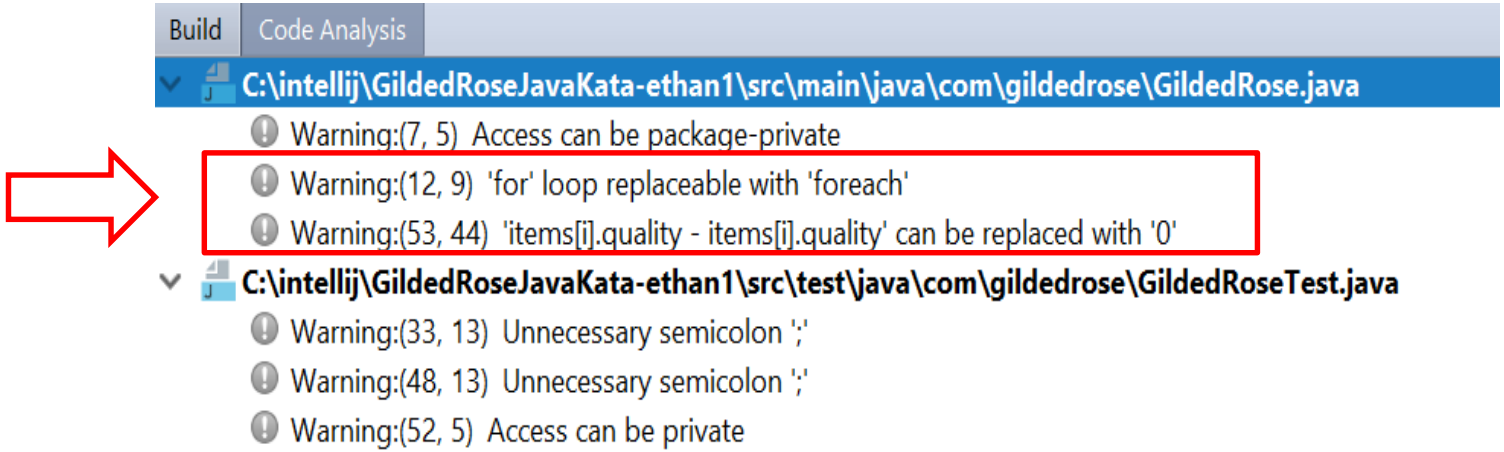
Quiz: Code Smells and Refactorings

Code Smells and Refactoring Examples

Code Smell	Refactoring
Unclear intention	Rename Method, Rename Variable
Long Method	Extract Method
Complex If	Decompose Conditional
Nested If	Replace Nested Conditional with Guard Clause, Reverse Conditions
Magic Numbers	Replace Magic Numbers with Constant

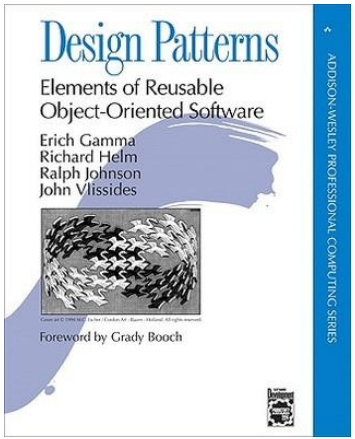
Examples

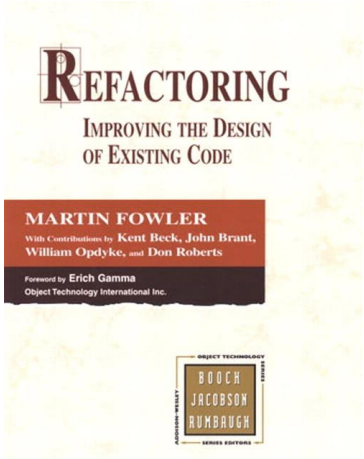
Review Warnings

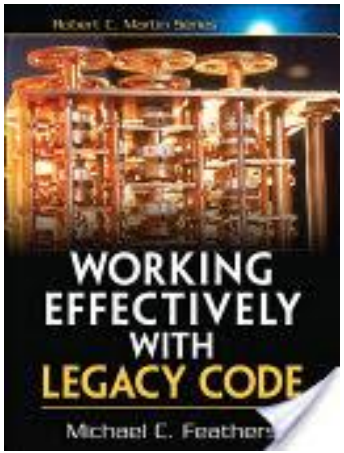


Literature

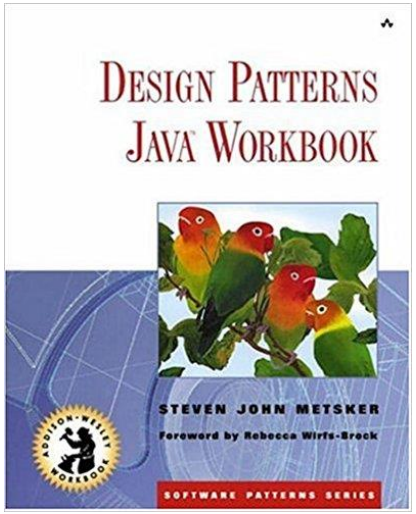
Literature: Smells, Refactorings, and Patterns

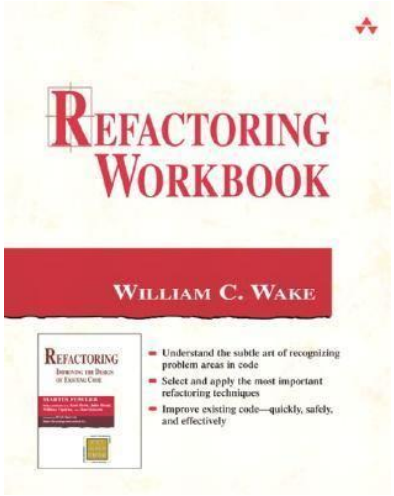


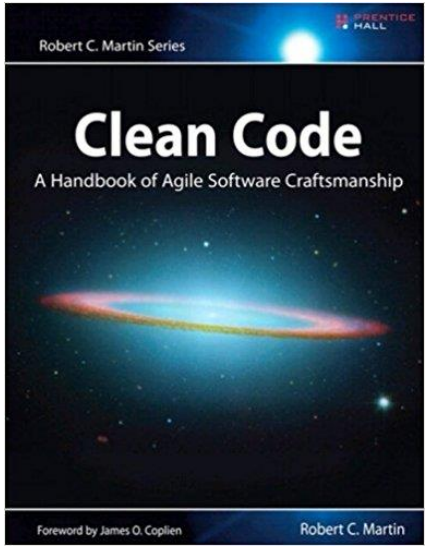




Practice Books







Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)		Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Table 1.1: Design pattern space

Refactoring Catalog

Composing Methods	*Moving Features Between Objects*	*Simplifying Conditional Expressions*
Extract Method	Move Method	Decompose Conditional
Inline Method	Move Field	Consolidate Conditional Expression
Inline Temp	Extract Class	Consolidate Duplicate Conditional Fragments
Replace Temp with Query	Inline Class	Remove Control Flag
Introduce Explaining Variable	Hide Delegate	Replace Nested Conditional with Guard Clauses
Split Temporary Variable	Remove Middle Man	Replace Conditional with Polymorphism
Remove Assignments to Parameters	Introduce Foreign Method	Introduce Null Object
Replace Method with Method Object	Introduce Local Extension	Introduce Assertion
Substitute Algorithm		
Organizing Data	*Making Method Calls Simpler*	*Dealing with Generalization*
Self Encapsulate Field	Rename Method	Pull Up Field
Replace Data Value with Object	Add Parameter	Pull Up Method
Change Value to Reference	Remove Parameter	Pull Up Constructor Body
Change Reference to Value	Separate Query from Modifier	Push Down Method
Replace Array with Object	Parameterize Method	Push Down Field
Duplicate Observed Data	Replace Parameter with Explicit Methods	Extract Subclass
Change Unidirectional Association to Bidirectional	Preserve Whole Object	Extract Superclass
Change Bidirectional Association to Unidirectional	Replace Parameter with Method	Extract Interface
Replace Magic Number with Symbolic Constant	Introduce Parameter Object	Collapse Hierarchy
Encapsulate Field	Remove Setting Method	Form Template Method
Encapsulate Collection	Hide Method	Replace Inheritance with Delegation
Replace Record with Data Class	Replace Constructor with Factory Method	Replace Delegation with Inheritance
Replace Type Code with Class	Encapsulate Downcast	
Replace Type Code with Subclasses	Replace Error Code with Exception	
Replace Type Code with State/Strategy	Replace Exception with Test	
Replace Subclass with Fields		

Code Smells and 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

Code Smells

Smells to Refactorings Quick Reference Guide

Smell	Refactoring
Alternative Classes with Different Interfaces: occurs when the interfaces of two classes are different and yet the classes are quite similar. If you can find the similarities between the two classes, you can often refactor the classes to make them share a common interface [F 85, K 43]	Unify Interfaces with Adapter [K 247]
	Rename Method [F 273]
	Move Method [F 142]
Combinatorial Explosion: A subtle form of duplication, this smell exists when numerous pieces of code do the same thing using different combinations of data or behavior. [K 45]	Replace Implicit Language with Interpreter [K 269]
Comments (a.k.a. Deodorant): When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous" [F 87]	Rename Method [F 273]
	Extract Method [F 110]
	Introduce Assertion [F 267]
Conditional Complexity: Conditional logic is innocent in its infancy, when it's simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well. You implement several new features and suddenly your conditional logic becomes complicated and expansive. [K 41]	Introduce Null Object [F 260, K 301]
	Move Embellishment to Decorator [K 144]
	Replace Conditional Logic with Strategy [K 129]
	Replace State-Altering Conditionals with State [K 166]
Data Class: Classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. [F 86]	Move Method [F 142]
	Encapsulate Field [F 206]
	Encapsulate Collection [F 208]
Data Clumps: Bunches of data that that hang around together really ought to be made into their own object. A good test is to consider deleting one of the data values: if you did this, would the others make any sense? If they don't, it's a sure sign that you have an object that's dying to be born. [F 81]	Extract Class [F 149]
	Preserve Whole Object [F 288]
	Introduce Parameter Object [F 295]

www.industriallogic.com/blog/smells-to-refactorings-cheatsheet

Code Smell Categories

The Bloaters	-Long Method -Large Class -Primitive Obsession -Long Parameter List -DataClumps
The Object-Orientation Abusers	-Switch Statements -Temporary Field -Refused Bequest -Alternative Classes with Different Interfaces
The Change Preventers	-Divergent Change -Shotgun Surgery -Parallel Inheritance Hierarchies

The Dispensables	-Lazy class -Data class -Duplicate Code -Dead Code, -Speculative Generality
The Couplers	-Feature Envy -Inappropriate Intimacy -Message Chains -Middle Man

Mäntylä, M. V. and Lassenius, C. **"Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study"**. Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431.

Resources

- ethan@agiletechskills.com
- https://agiletechskills.com/resources
  - – Books, videos, links
- https://agiletechskills.com/events
  - -Weekly coding dojo
- https://agiletechskills.com/services
  - – Consulting, pro bono session at your company

Conclusion

- Decide why, when, and what you are refactoring
- Define testing strategy
- Use static analysis tools
- Identify smells, refactorings, and design patterns
- Refactor using tool
- Add feature or fix bugs using test driven development

Questions??