Refactoring

Let’s have more Kool-Aid

Code Smells

Refactorings

Refactoring to Patterns

Technical Debt

Agile Coding Principles

Collective Code Ownership in practice:

- We cannot check in our code until:
  - All tests are green.
  - All duplication has been removed
  - The code is as expressive as we can make it.
  - The code is as simple as we can make it.
- *Note: nothing in here says you wrote the original code!*

The "code is the documentation"

- Comments should be minimized
  - They often lie.
  - They are an admission that we could not make our code express our ideas.
  - There are many things you can do to make software expressive.

Refactoring

"One man with courage makes a majority"
-- Andrew Jackson

Kool-Aid:

- XP refactors *mercilessly*.
- Refactoring is aided by *relentless* testing and continuous integration.
- Refactoring takes *courage*!

Defined:

- The act of improving code design "in place" without modifying the functional behavior of the code
- Increase code quality
- In theory, you should not have to change unit tests
- A goal of refactoring is to move toward a design pattern
  - *If design is good, then everybody will do it everyday!*

Adapted from presentation by J.Turpin & J.Morris, 2002

Code Smells

How do you identify code in need of refactoring?

"If it stinks, change it."

Types of smells:

- *Within classes*: Smells within the structure and responsibilities of a single class
- *Between classes*: Smells arising from the struc collaborations of many classes

Examples:

Duplicate code

Switch statements

Long method

Large class

Data class ("data bag")

Long parameter list

Primitive obsession

Temporary field

…and many more

Adapted from presentation by W..Wake, 2000

Examples: Smells *Within* Classes

Large Class

- Smell: Keep adding one more thing to the class
  - The class loses its sense of identity – you "smell" that nobody can figure out why it was written in the 1st place!
- Solution: *Extract a new class* - find features that reflect unique concepts and delegate responsibility to the new class

Long Parameter List

- Smell: Methods grows occasionally with new parameters or has lots of polymorphic variants
- Solution #1: *Preserve object* - pass the entire object instead of just a single value
- Solution #2: *Introduce parameter object* - combine parameters into a new (immutable) object

Examples: Smells *Within* Classes

Speculative Generality

- Smell: Guessing that flexibility will be needed in future
  - depends on what part system; lower level is probably OK (framework)
- *Solution*: Remove general code, method, class, etc.
  - OK when a framework or testing/debugging code

Duplicate Code

- Smell: Can be nearly identical or conceptually identical
- *Solution #1*: If inside same class, extract common code to private method
- *Solution #2*: If unrelated classes, refactor to a new class

Examples: Smells *Between* Classes

Primitive Obsession

- *Smell*: Not enough objects - only primitives, suggests a lack of strong domain modeling.
- *Solution:* Combine related primitives into objects
  - Ex: Pixels (not x,y), Colors (not RGB), Addresses (not street, city, zip)

Feature Envy

- *Smell*: Class manipulates data of another class
- *Solution*: Either move the behavior to the other class or refactor into a shared class.

Divergent Change

- *Smell*: Class picks up unrelated features over time
- *Solution:* Extract related features to new class(es)

Refactorings

Fowler's book has a catalog

- There are many more
- Often reversible

Principles:

- Pay attention to the mechanics
- Let the compiler tell you
- *Small steps* with *constant tests* (XP philosophy)

Unit Testing and Refactoring

- By having unit tests available, you can verify a refactoring produces *functional equivalence*
- XP's approach is "test-code-refactor" ("lather-rinse-repeat")

Adapted from presentation by W..Wake, 2000

Refactoring

Refactoring actions rewrite source code

- Within a single source file
- Across multiple interrelated source files

Refactoring actions preserve program semantics

- Does not alter what program does, just affects the way it does it
- Ideally should not change a class interface

Refactoring strengths

- Encourages exploratory programming
- Facilitates Agile programming techniques
- The resulting code should be of higher quality
  - This should be represented in your static analysis tools and your metrics

Ex Refactoring: Introduce Null Object

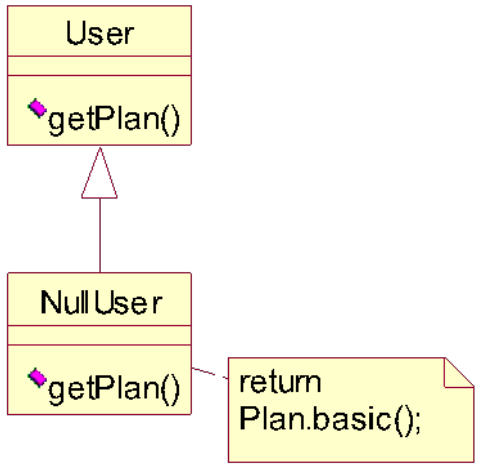
Smell: Your code does repeated checks for null return values

Solution: Null subclass type encapsulates default behaviors you might have when null is returned.

BEFORE

```
if (user == null)
    plan = Plan.basic();
else
    plan = user.getPlan();
```

AFTER



Adapted from presentation by W..Wake, 2000

Refactoring: Split Loop

BEFORE:

```
void printValues()  {
    double averageAge = 0;
    double totalSalary = 0;
    for (int i=0; i<people.length; i++) {
        averageAge += people[i].age;
        totalSalary += people[i].salary;
    }
    averageAge = averageAge / people.length;
    System.out.println(averageAge);
    System.out.println(totalSalary);
}
```

AFTER:

```
void printValues()  {
    double totalSalary = 0.0;
    for (int i=0; i<people.length; i++)
        totalSalary += people[i].salary;

    double averageAge = 0.0;
    for (int i=0; i<people.length; i++)
        averageAge += people[i].age;
    averageAge = averageAge / people.length;

    System.out.println(averageAge);
    System.out.println(totalSalary);
}
```

Smell: "There is a loop that is doing two things." (e.g., accumulating both average age and total salary for all employees.)

Solution: Split into 2 loops.

- Makes each thing easier to understand.
- Opens the door to further refactoring/optimization (e.g., one loop may be replaced by a Collection aggregate function).

Adapted from Scach, Object-oriented and Classical Software Engineering

Refactoring: Split Temporary Variable

Smell: "You have a temporary variable assigned to in more than one place, but is not a loop variable nor an accumulating/collecting temporary variable."

Solution: Make a separate temp var for each assignment.

Before:

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);
```

After:

```
final double perimeter = 2 * (height + width);
System.out.println(perimeter);
final double area = height * width;
System.out.println(area);
```

Adapted from Scach, Object-oriented and Classical Software Engineering

Ex Refactoring to Pattern: Singleton

Problem: You have an object representing a physical resource (say, a USB device).

- You want to make sure you only have one actual object in memory to present the one physical object
- You want to manage that object's lifecycle

Solution: Refactor Constructor to a Singleton

```
// BEFORE: Client creates USB
public byte readUSB(int comPort) {
  // you would probably have some
  // init code here (baud, xon/off)
  USB u = new USB(settings);
  return u.read();
}
// what happens if 2 threads call
// this? Or no GC happens before
// next call?
```

```
// AFTER: Apply a Singleton
public byte readUSB(int comPort) {
  USB u = USB.getUSB(settings);
  return u.read();
}
// I can internally manage
Class USB {
  private static _usb = null;
  private USB() { // init here }
  public static USB getUSB() {
    if (_usb == null) _usb = new USB();
      return _usb;
} }
```

Refactoring - Motivation

It is not always clear to management why you refactor

> "*I don't see any functional change in the code for all this work you are doing*"

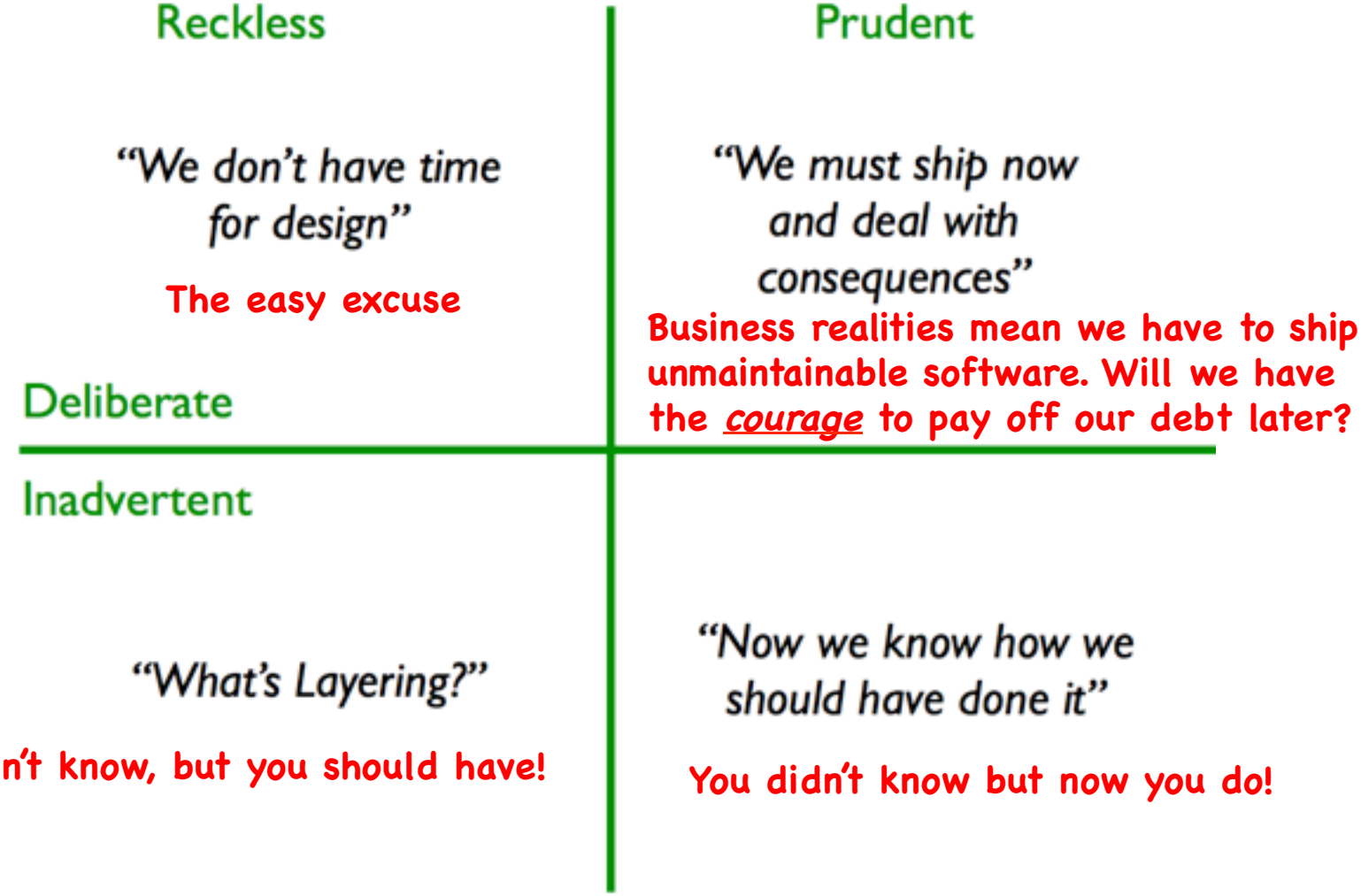
They need to be educated on *Technical Debt*

- When it is not done right the 1st time, you incur a debt in the software that must be paid off in the future.
- Refactoring basically pays down part of that debt
- Like most debt (e.g. credit cards), the longer the debt remains, the more it will cost (or eventually bankrupt) you!

So again, "Refactoring takes *courage*!"

- Development needs the courage to make a near-term investment for the better long term quality of the code,
- The Technical Debt Quadrant explains some smells!

The Technical Debt Quadrant



Annotations by Kgary, quadrant figure from Mfowler http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Refactoring Summary

Refactoring is sometimes thought of as “in-place design”, or “just-in-time design”

- I don’t agree with this. Refactoring may improve the low-level design of code, but in places where full design is needed, do it!

Refactoring is an expression of *Collective Code Ownership* and *Courage* agile values

- The code is the responsibility of everyone
- You should not go into a code module, look at it and think “geez what a mess” and not do anything about it!
- You also need the courage to alter schedule expectations in the name of quality
  - The *technical debt* doesn’t magically go away, and it only accrues interest!

Refactoring is only of limited use in isolation

- Real power is in embedding in a test-code-refactor cycle
- Target your Design Patterns!
- Use unit tests and metrics to diagnose, measure, and inject quality!