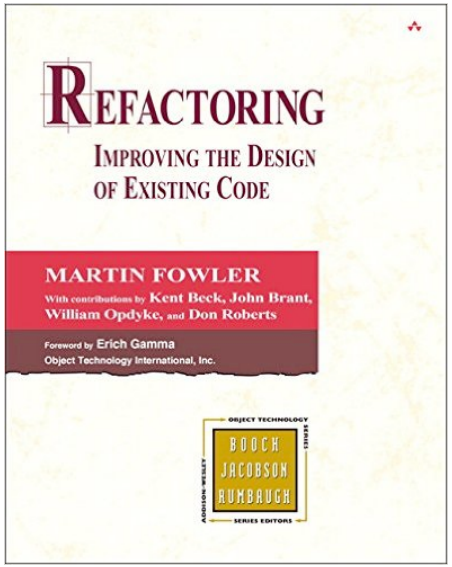
SSW-555: Agile Methods for Software Development

Refactoring

Prof. Jim Rowland
Software Engineering
School of Systems and Enterprises

Acknowledgements

Refactoring: Improving the Design of Existing Code by Martin Fowler



https://martinfowler.com/articles/workflowsOfRefactoring/#final

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

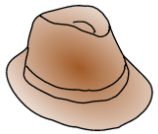
-- Martin Fowler

Today's topics

Who, what, when, why, and how of refactoring

Technical debt

Two hats of software development



Bad smells

Refactoring workflows

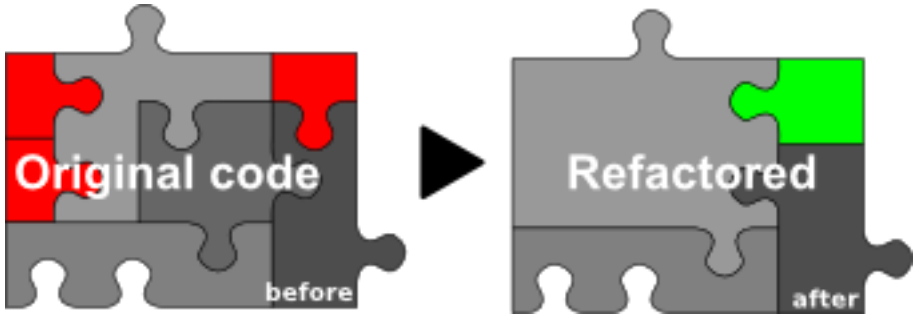
Eclipse and PyCharm support for refactoring

If it ain't broke don't fix it…

… may ***not*** be the best strategy for enhancing and maintaining software, especially software with bad smells...

What is refactoring?

Refactoring: changing the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior



http://www.moniro.com/my_pictures/product-software-refactoring.png

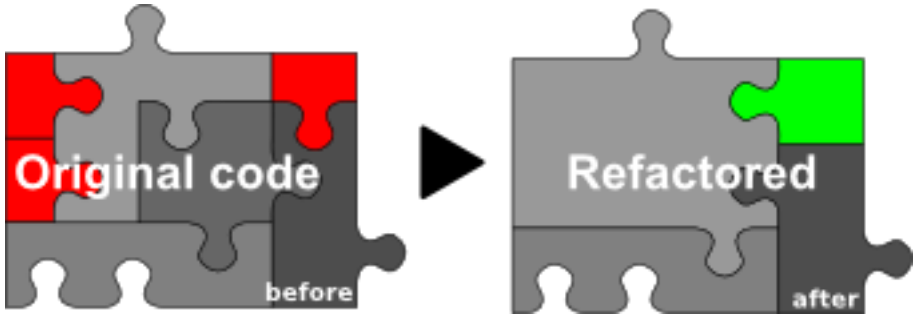
What is refactoring?

Users should not notice that the code has been refactored

Refactoring should improve readability and reduce complexity

Refactoring is not about performance optimization

Refactoring is critical for code that changes frequently



http://www.moniro.com/my_pictures/product-software-refactoring.png

Why refactor?

We start with a good design and write good code to implement that design

- BUT, over time the code changes to meet changes in the requirements
- BUT, the design may not be updated to optimize those changes

Refactoring refreshes the design and the code to reflect an optimal solution for our latest understanding of the problem and solution

Why: Benefits of refactoring

Refactoring improves the design of software

Refactoring makes software easier to understand and change

Refactoring helps you to find bugs

Refactoring helps you to program faster

Refactoring is needed to pay off "technical debt"

Technical debt

Additional development, testing, and maintenance effort

Caused by

- Bad design
- Taking shortcuts
- Not implementing the "right" solution throughout the lifecycle

Quick hacks add technical debt

Technical debt accumulates "interest" and makes changes even harder later on

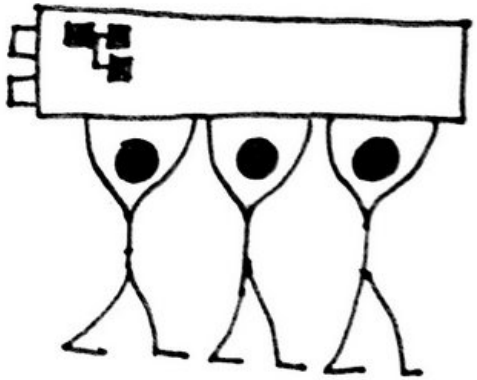
Refactoring helps to pay off "technical debt"

Whose code are we refactoring?

XP supports ***collective code ownership***

- Every team member collectively owns all code
- Every team member is responsible for all code
- Individuals don't own files or features

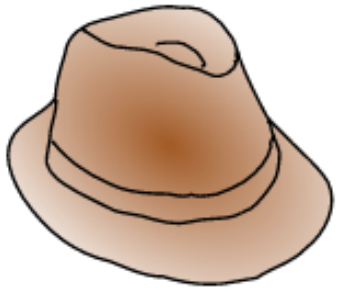
Anyone may refactor anyone else's code !?!?



Software development hats (Kent Beck)

1. Adding functionality to the system
   - Not changing existing code
   - Adding code
   - Adding tests (may break existing tests)

2. Refactoring
   - Not adding new functionality
   - Not adding tests
   - Not changing tests (unless necessary)
   - Small, quick, behavior-preserving changes



Fowler suggests that developers should switch hats frequently but don't mix the modes

https://martinfowler.com/articles/workflowsOfRefactoring

Good test cases are critical

Refactoring replaces working code with new code

Research shows that new code is more likely to contain bugs than older code

Automated regression testing is critical, e.g. xUnit

Regression testing verifies:

- The new code replicates the behavior of the old code
- The new code doesn't add any new bugs

When to refactor?

Rule of three

1. First time, just do it
2. Second time, wince at duplication
3. Third time refactor

When you add functionality

When you need to fix a bug

As you do a code review

Bad smells

As you read code you realize that it needs to be restructured (refactored)

Bad smells aren't necessarily bugs, but instead is something that isn't optimal and may suggest a bigger problem

- Design flaws
- Opportunity for improvement

Experience guides you when and how to refactor

Fowler has a catalog of "bad smells" that suggest refactoring

- Anti-patterns that you shouldn't do

Examples of bad smells

Duplicated code

Unnecessary complexity

Bad smells in classes

- Very large class
- Classes whose implementation depends on the implementation of another class
- Cyclomatic complexity/many different execution flows

Bad smells in Methods

- Too many parameters
- Long method with too much code
- Bad method or variable names
- Returning too much data from the method

Bad Smells: Duplicated Code

Observe: duplicated code

Refactoring techniques:

- Extract method – create a new method
- Pull up field – move the field to the superclass
- Form template method – move the method to the superclass and use polymorphism
- Substitute algorithm
- Extract new class

Refactoring: Extract method

1. You have a code fragment that can be grouped together
2. Turn the fragment into a method
3. Replace the fragment with a call to the method

Refactoring: Extract method

1. You have a code fragment that can be grouped together
2. Turn the fragment into a method

```
void printOwing() {
    printBanner();
    // print details
    System.out.println("name:", name);
    System.out.println("amount:", amt);
    }
```

```
void printOwing() {
    printBanner();
    // print details
    System.out.println("name:", name);
    System.out.println("amount:", amt);
    }
void printDetails(String name,
                  double amt) {
    System.out.println("name:", name);
    System.out.println("amount:", amt);
    }
```

Create a new method with the code fragment

Refactoring: Extract method

1. You have a code fragment that can be grouped together
2. Turn the fragment into a method
3. Replace the fragment with a call to the method

```
void printOwing() {
    printBanner();
    // print details
    System.out.println("name:", name);
    System.out.println("amount:", amt);
    }
```

```
void printOwing() {
    printBanner();
    printDetails(name, amt);
    }
void printDetails(String name,
                  double amt) {
    System.out.println("name:", name);
    System.out.println("amount:", amt);
    }
```

Call the new method

Bad smells: Long method

Observe: method is unreasonably long

Refactoring techniques:

- Extract Method
- Replace Temp with Query – replace temp with method
- Introduce Parameter Object – replace many parameters with an object
- Preserve Whole Object – send an object rather than the parts
- Decompose Conditional – replace conditional with methods

Comments may help identify clumps of code to extract into a new method

Bad Smells: Large class

Observe: too many instance variables in a class

Refactoring techniques:

- Extract Class
- Extract Subclass
- Extract Interface

Common prefixes or suffixes in variable names suggest groups of variables to extract into a new class

How: GEDCOM refactoring example

Scenario: Dates and date calculations are common in the GEDCOM project

Goal: Implement user story 'Less than 150 years old'

```
def us07_too_old(bd):
    ''' Return True if the specified birthdate is within the last 150 years
            bd:  a datetime representing the person's birth date
    '''
    return bd <= (datetime.today() - timedelta(days=365.25 * 150))
```

How: GEDCOM refactoring example

Scenario: Dates and date calculations are common in the GEDCOM project

Goal: Implement user story 'List recent births'

```
def US35_recent_birth(bd):
    ''' Return True if the specified birthdate is within the last 30 days
            bd:  a datetime representing the person's birth date
    '''
    return bd <= (datetime.today() - timedelta(days=30))
```

Similar to US17_too_old() but days instead of years.

How: GEDCOM refactoring example

Goal: Implement user story 'Siblings 9 months apart'

```
def US43_siblings_9_months_apart(bd1, bd2):
    ''' Return True if bd1 is less than 9 months from bd2
            bd1:  a datetime representing a person's birth date
            bd2:  a datetime representing a person's birth date
    '''
    return abs((bd1 - bd2).days / 30.4) <= 9
```

Similar to US17_too_old() & US35_recent_birth but months instead of days or years.

We've seen date comparisons of days, months, years…

Refactor to simplify these cases and futures cases…

How: GEDCOM refactoring example

```
def dates_within(dt1, dt2, limit, units):
    ''' return True if dt1 and dt2 are within limit units, where:
            dt1, dt2 are instances of datetime
            limit is a number
            units is a string in ('days', 'months', 'years')
    '''
    if units == 'days':
        return abs((dt1 - dt2).days) <= limit
    elif units == 'months':
        return (abs((dt1 - dt2).days) / 30.4) <= limit
    elif units == 'years':
        return (abs((dt1 - dt2).days) / 365.25) <= limit

def us07_too_old(bd):
    return not dates_within(bd, datetime.today(), 150, 'years')

def US35_recent_birth (bd):
    return dates_within(bd, datetime.today(), 30, 'days')

def US43_siblings_9_months_apart (bd1, bd2):
    return dates_within(bd1, bd2, 9, 'months')
```

One reusable function with simpler semantics

Test once, reuse many times

Revised functions must pass existing automated tests

Add new user stories with similar semantics easily

How: GEDCOM refactoring example

We can do better! Refactor again!

```
def dates_within(dt1, dt2, limit, units):
    ''' return True if dt1 and dt2 are within limit units, where:
            dt1, dt2 are instances of datetime
            limit is a number
            units is a string in ('days', 'months', 'years')
    '''
    conversion = {'days': 1, 'months': 30.4, 'years': 365.25}
    return (abs((dt1 - dt2).days) / conversion[units]) <= limit

def us07_too_old(bd):
    return not dates_within(bd, datetime.today(), 150, 'years')

def US35_recent_birth (bd):
    return dates_within(bd, datetime.today(), 30, 'days')

def US43_siblings_9_months_apart (bd1, bd2):
    return dates_within(bd1, bd2, 9, 'months')
```

Replace the if/elif with extensible data driven solution

No changes needed

Soap Box: Comments

"When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous" – Martin Fowler, Refactoring, pp. 88

Comments are not a replacement for clean, elegant code but judicious, concise comments are important and valuable for long term code maintenance.

```
def dates_within(dt1, dt2, limit, units):
    ''' return True if dt1 and dt2 are within limit units, where:
            dt1, dt2 are instances of datetime
            limit is a number
            units is a string in ('days', 'months', 'years')
    '''
    conversion = {'days': 1, 'months': 30.4, 'years': 365.25}
    return (abs((dt1 - dt2).days) / conversion[units]) <= limit

def dates_within(dt1, dt2, limit, units):
    conversion = {'days': 1, 'months': 30.4, 'years': 365.25}
    return (abs((dt1 - dt2).days) / conversion[units]) <= limit
```

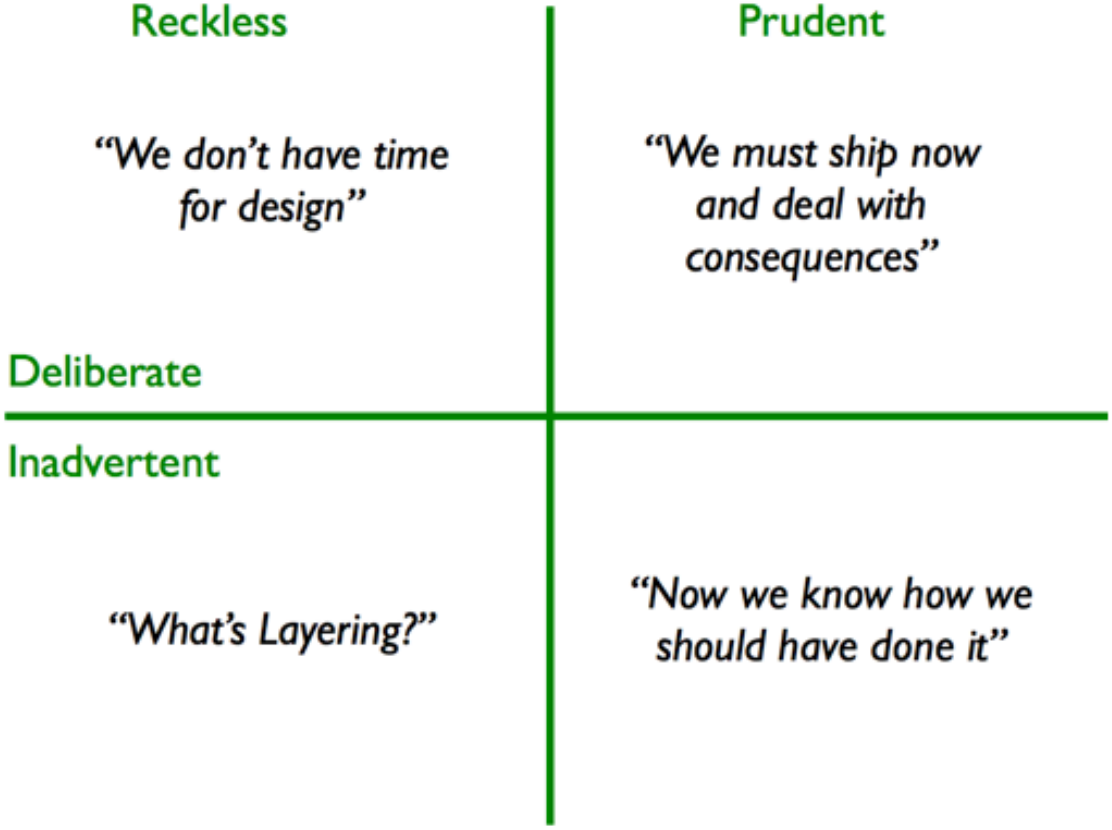
Which would you rather maintain for several years?

More bad smells and refactoring techniques

See Fowler's *Refactoring: Improving the Design of Existing Code* for more examples of bad smells and remedies

refactoring.com has a comprehensive set of examples

When: Technical Debt Quadrant

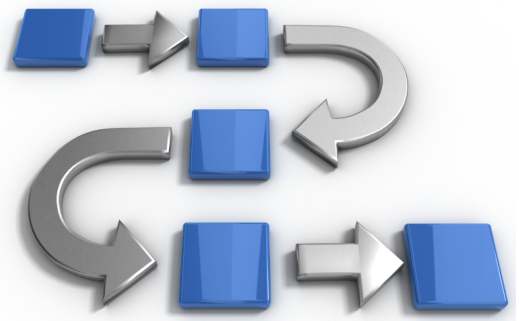


https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Refactoring workflows

When to refactor? To solve which problem?

- TDD
- Litter pickup
- Comprehension refactoring
- Preparatory refactoring
- Planned refactoring
- Long term refactoring



TDD refactoring workflow

Scenario: Writing new code

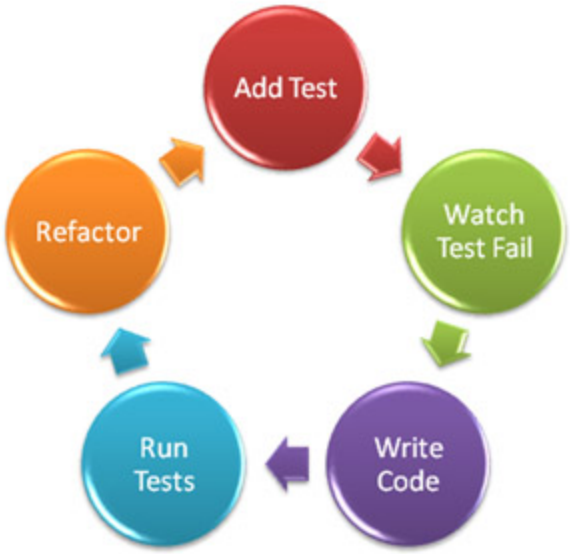
Write tests first

Run the tests (they will probably fail)

Write some code

Rerun the tests

Debug until the tests pass



- Relatively little untested code at any one time so bugs are likely to be in the most recent code

Litter pickup refactoring workflow

Scenario: Reading code and discover some bad code

- Less capable developer
- Capable developer in a hurry
- Didn't understand solution

Replace bad code with an elegant solution

"Leave the code cleaner than you found it"

– Robert "Uncle Bob" Martin

Opportunistic refactoring

Comprehension refactoring workflow

Scenario: reading code and gain insights into how the code works

Refactor the code to help the next reader benefit (possibly you!) from your experience and effort

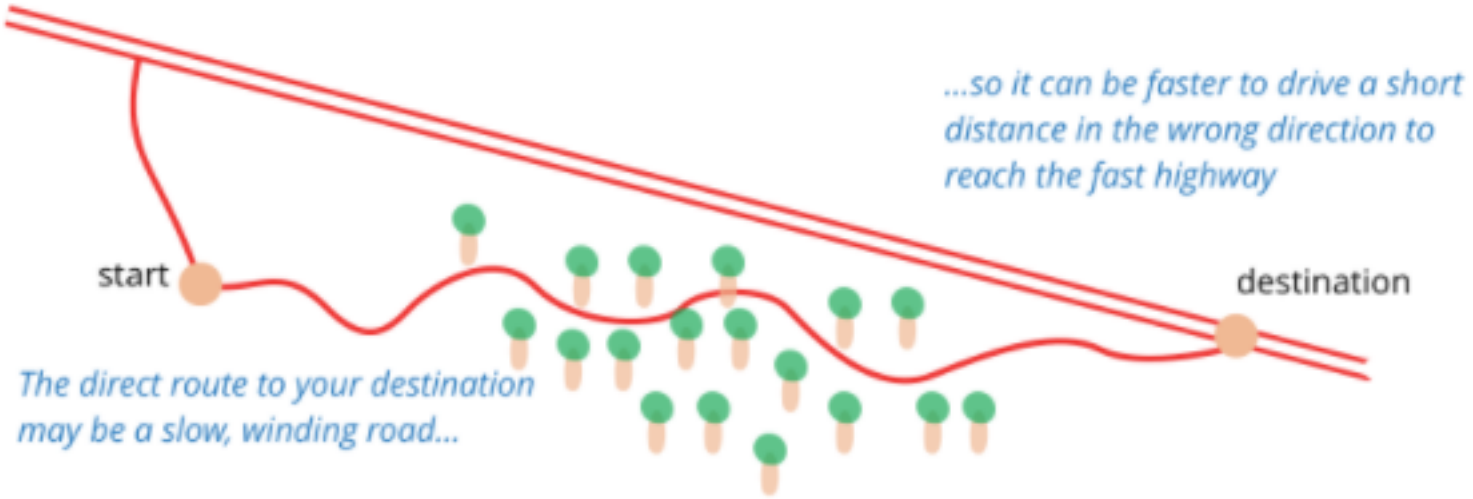
E.g. rename functions or variables to improve names to reflect what's happening in the code

Opportunistic: similar to litter pick up

Preparatory refactoring workflow

Scenario: need to add a new feature but it's easier to add the new feature after making changes to existing code

Sometimes it's better to go backward to go forward



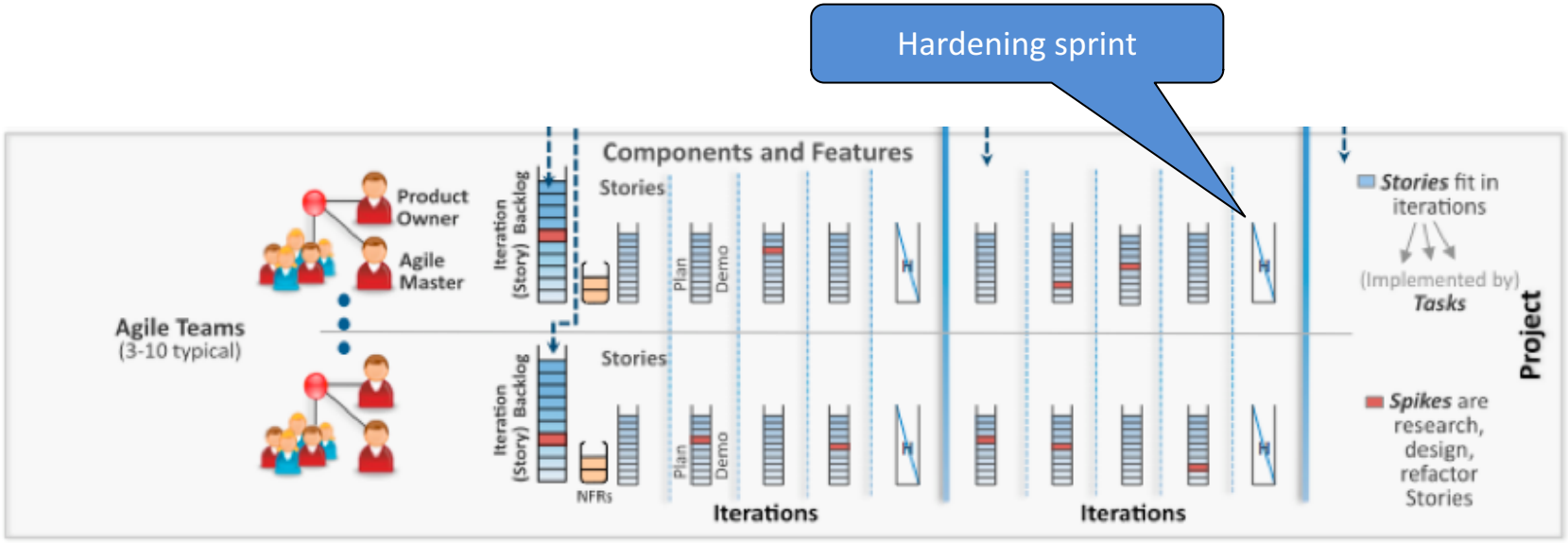
Jessica Kerr -- https://martinfowler.com/articles/preparatory-refactoring-example.html

Planned refactoring workflow

Scenario: Include time in the project schedule time for refactoring

E.g. Scaled Agile Framework allocates "hardening" sprints

This may suggest that you're waiting too long to refactor



Long term refactoring workflow

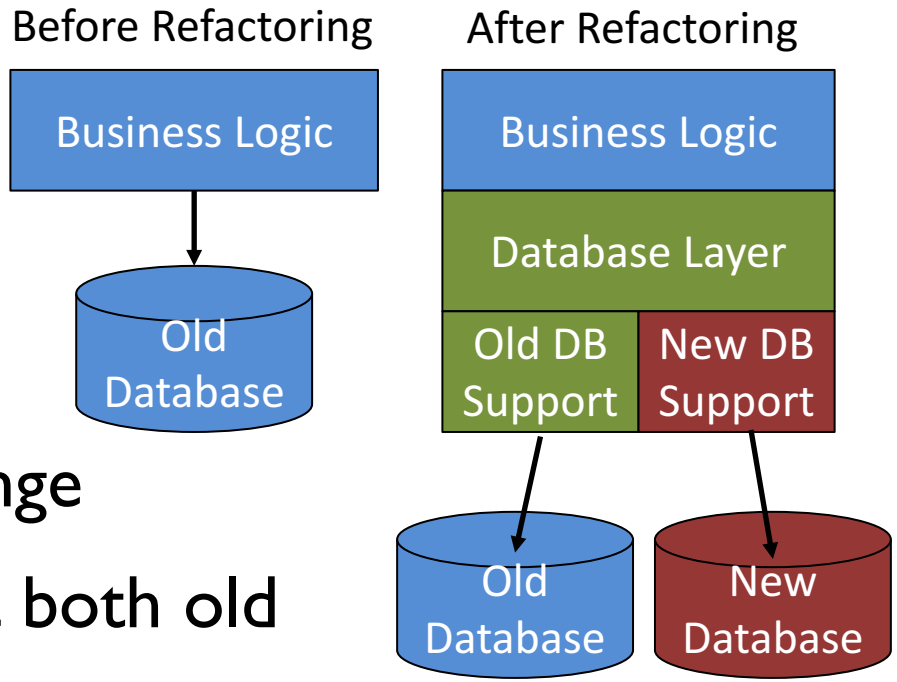
Scenario: Need a complex feature that that can't be added in a single sprint

New code to simplify change

E.g. support a new database solution or replace existing database technology

Add new code to simplify change

Add interface layer to support both old and new database technologies



Eclipse support for Refactoring

Eclipse has some built-in refactorings:

- Rename {field, method, class, package}
- Move {field, method, class}
- Extract method
- Extract local variable
- Inline local variable
- Reorder method parameters
- ...

Eclipse also supports user-defined refactorings

PyCharm support for Refactoring

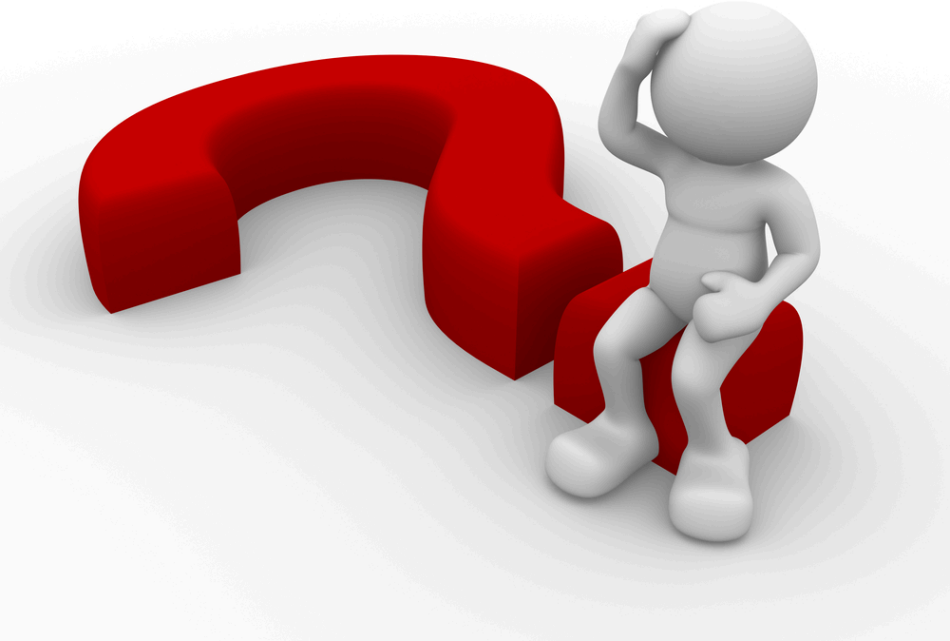
JetBrains PyCharm tool supports refactoring for Python

Examples

- Change signature
- Convert to Python Package/Module
- Inline
- Invert Boolean
- …

See https://www.jetbrains.com/help/pycharm/2016.1/refactoring-source-code.html

Questions?



Why refactor?

"Cities grow, cities evolve, cities have parts that simply die while other flourish; each city has to be renewed to meet the needs of the populace… Software-intensive systems are like that. They grow, they evolve, sometimes they wither away and sometimes they flourish… Users suffer the consequences of capricious complexity, delayed improvements and insufficient incremental change; the developers who evolve such systems suffer the slings and arrows of never being able to write quality code because they are always trying to catch up." – Grady Booch

Refactoring for Software Design Smells: Managing Technical Debt
By Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma