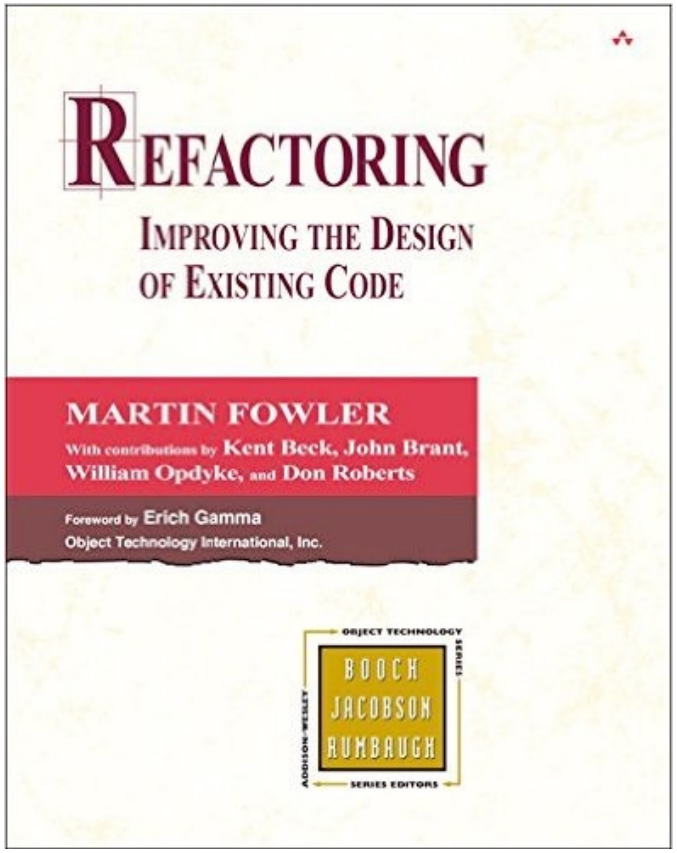
STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW-555: Agile Methods for Software Development

Refactoring
Week 6

Acknowledgements

- *Refactoring: Improving the Design of Existing Code* by Martin Fowler



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

https://refactoring.com/

https://sourcemaking.com/refactoring

```
                                                                  char
                                                              _3141592654[3141
        ),__3141[3141];_314159[31415],_3141[31415];main(){register char*
     _3_141,*_3_1415, *_3__1415; register int _314,_31415,__31415,*_31,
   _3_14159,__3_1415;*_3141592654=__31415=2,_3141592654[0][_3141592654
  -1]=1[__3141]=5;__3_1415=1;do{_3_14159=_314=0,__31415++;for( _31415
  =0;_31415<(3,14-6)*__31415;_31415++)_3141[_3141]=_314159[_31415]= -
1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
__3_1415     +__3141;for                {_31415 = 3141-
            __3_1415  ;                 _31415; 31415--
           ,_3_141 ++,                  _3_1415++){_314
           +=_314<<2 ;                  _314<<=1;_314+=
          *_3_1415;_31                   =_314159+_314;
          if(!(*_31+1)                   )* _31 =_314 /
          __31415,_314                   [_3141]=_314 %
          __31415 ;* {                   _3__1415=_3_141
         }+= *_3_1415                     = *_31;while(*
         _3__1415 >=                      31415/3141 ) *
         _3__1415+= -                     10,(*--_3__1415
        )++; 314= 314                     [_3141]; if ( !
        _3_14159 && *                     _3_1415)_3_14159
        =1,__3_1415 =                     3141-_31415;}if(
        _314+(__31415                      >>1)>=__31415 )
        while ( ++ *                      _3_141==3141/314
       )*_3_141---0                       ;}while(_3_14159
       ) ; { char *                       __3_14= "3.1415";
       write((3,1),                       (--*__3_14,__3_14
       },(_3_14159                         ++,++_3_14159))+
      3.1415926; }                         for ( _31415 = 1;
      _31415<3141-                         1;_31415++)write(
    31415% 314-(                           3,14),_3141592654(
  _31415     ) +                           "0123456789","314"
  [ 3]+1)-_314;                            puts((*_3141592654=0
,_3141592654))                              ;_314= *"3.141592";}
```

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

Martin Fowler

Today's Topics

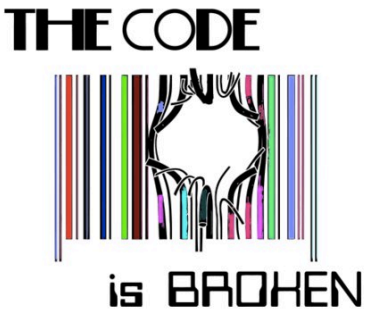
- Refactoring Overview
- Technical debt
- Bad smells
- Two hats of software development
- Refactoring workflows
- Eclipse and PyCharm support for refactoring

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...

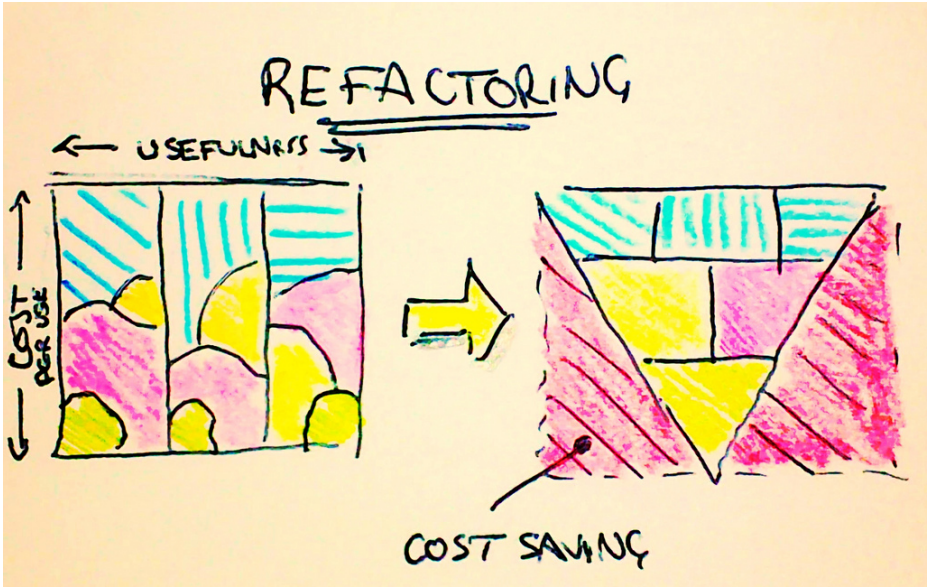
Why fix a part that isn't broken?

- Each part of your system's code has 3 purposes.
  - Execute the functionality
  - To allow change
  - To communicate well to developers who read it
- If the code does not do one or more of these, it is "**broken**"!!



What is refactoring?

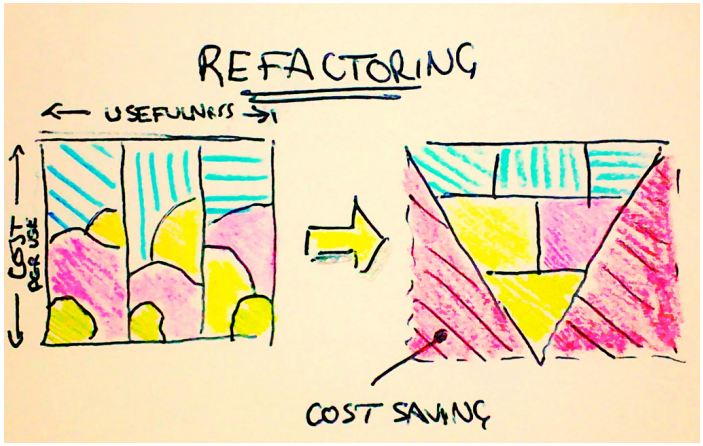
Refactoring: changing the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior



http://nonprofits.agileventures.org/2016/12/05/refactoring_and_pairing/

What is refactoring?

- Internal: Improved
  - Refactoring should improve readability and reduce complexity



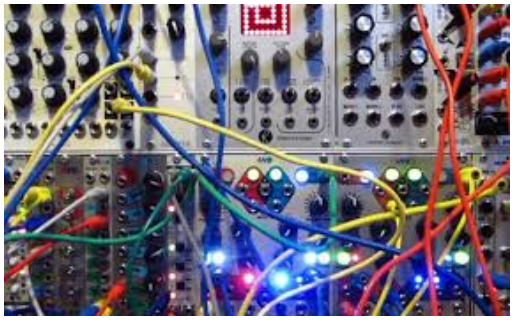
http://nonprofits.agileventures.org/2016/12/05/refactoring_and_pairing/

- External: Not changed
  - Users should not notice that the code has been refactored
  - Refactoring is not about performance optimization

Refactoring is critical for code that changes frequently!!

Why refactor?

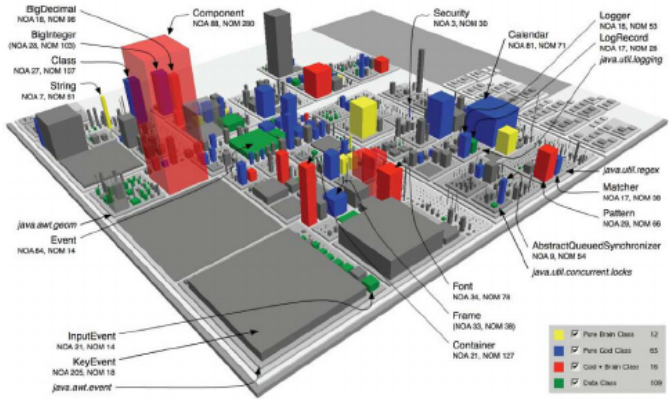
- We start with a good design and write good code to implement that design
  - BUT, over time the code changes to meet changes in the requirements
  - BUT, the design may not be updated to optimize those changes

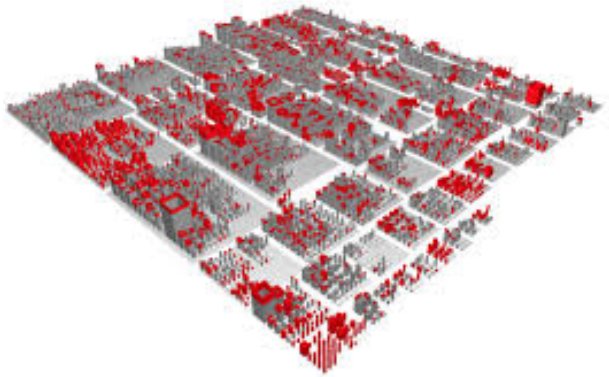


Why refactor?

"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."

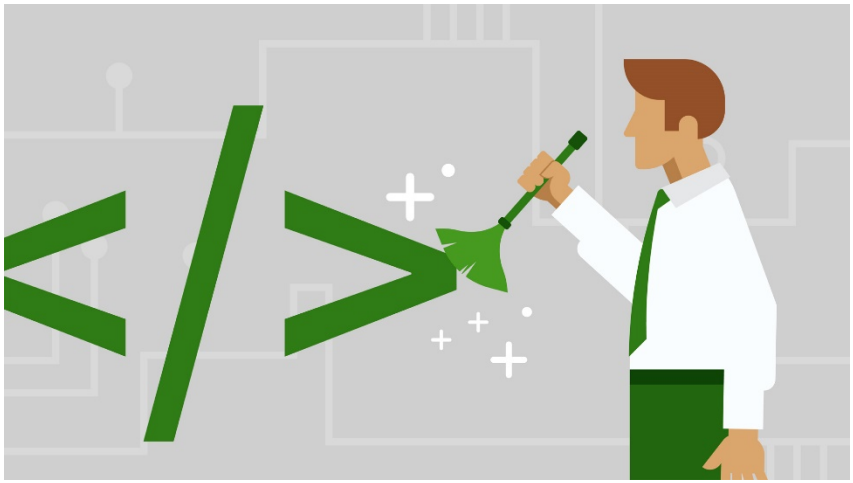
– Grady Booch





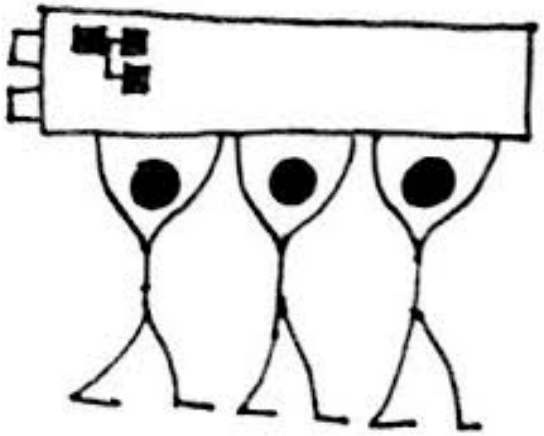
Why refactor? – The Benefits

- Refactoring refreshes the design and the code
  - Improves the design
  - Make code easier to understand and change
  - Helps you to program faster
  - Pay off "technical debt”



Who should do the 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 !?!?



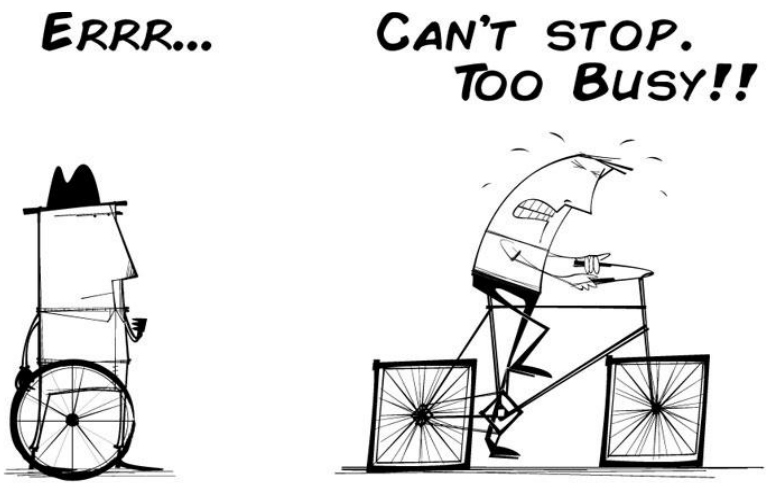
When to refactor?

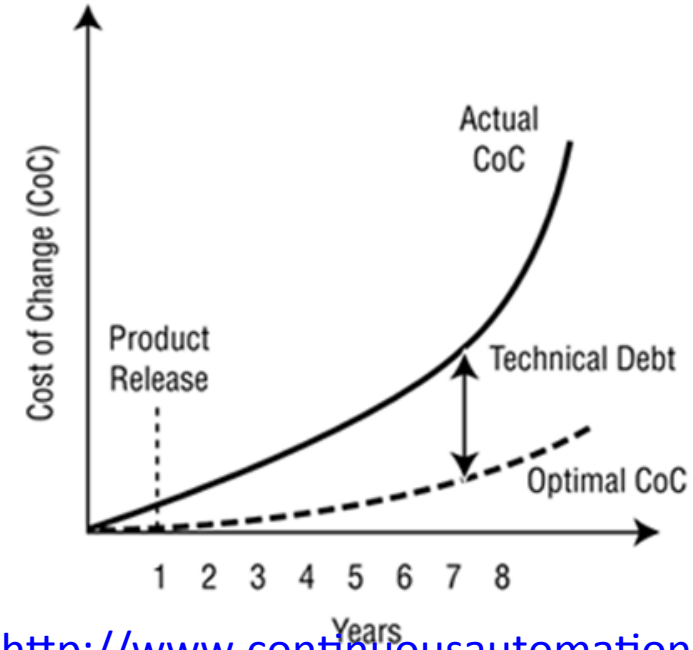
- If your code base:
  - Is difficult to understand
  - Hard to modify
  - Slow you down
  - Accumulate technical "debt"
  - Have bad "smells"

- That's a sign that you need to do refactoring in order to improve your efficiency.

Technical Debt

- Additional development, testing, and maintenance effort due to bad design, taking shortcuts, e.g. quick hacks, not implementing the "right" solution throughout the lifecycle…



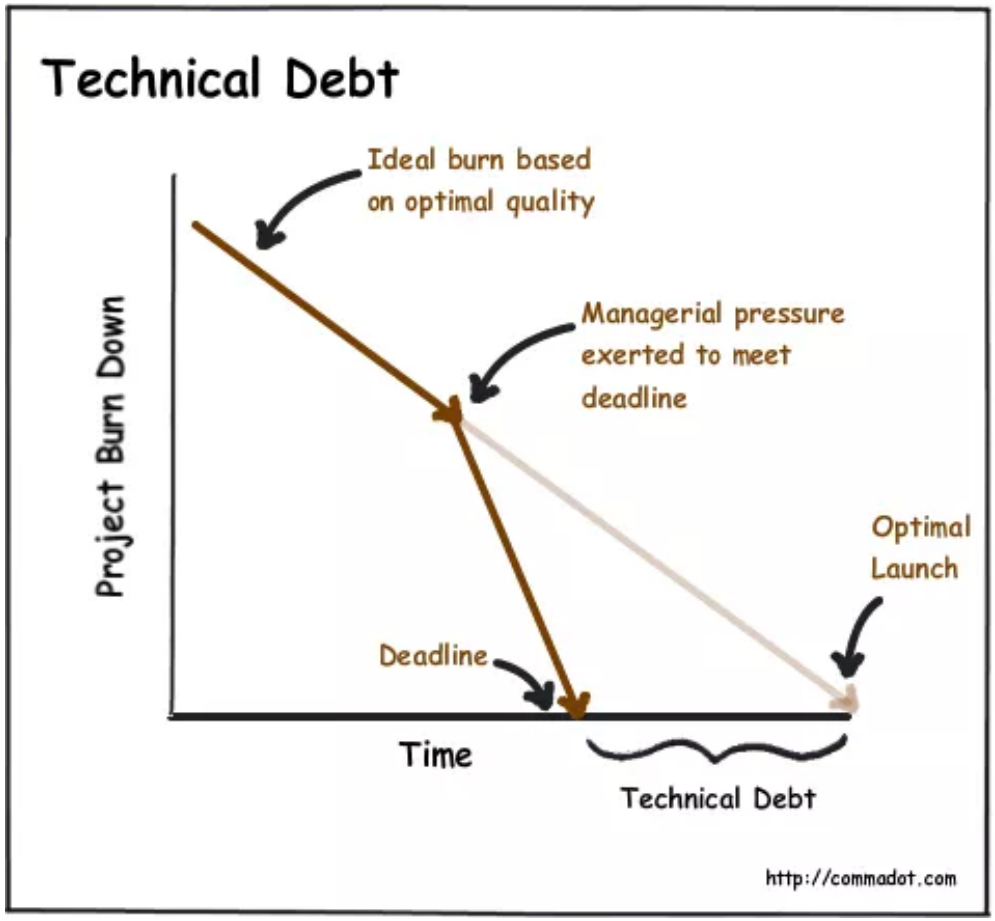


https://www.activestate.com/blog/2016/06/technical-debt-high-tech-ceos-perspective

http://www.continuousautomation.com/technical-debt-a-threat-to-business/

Technical Debt

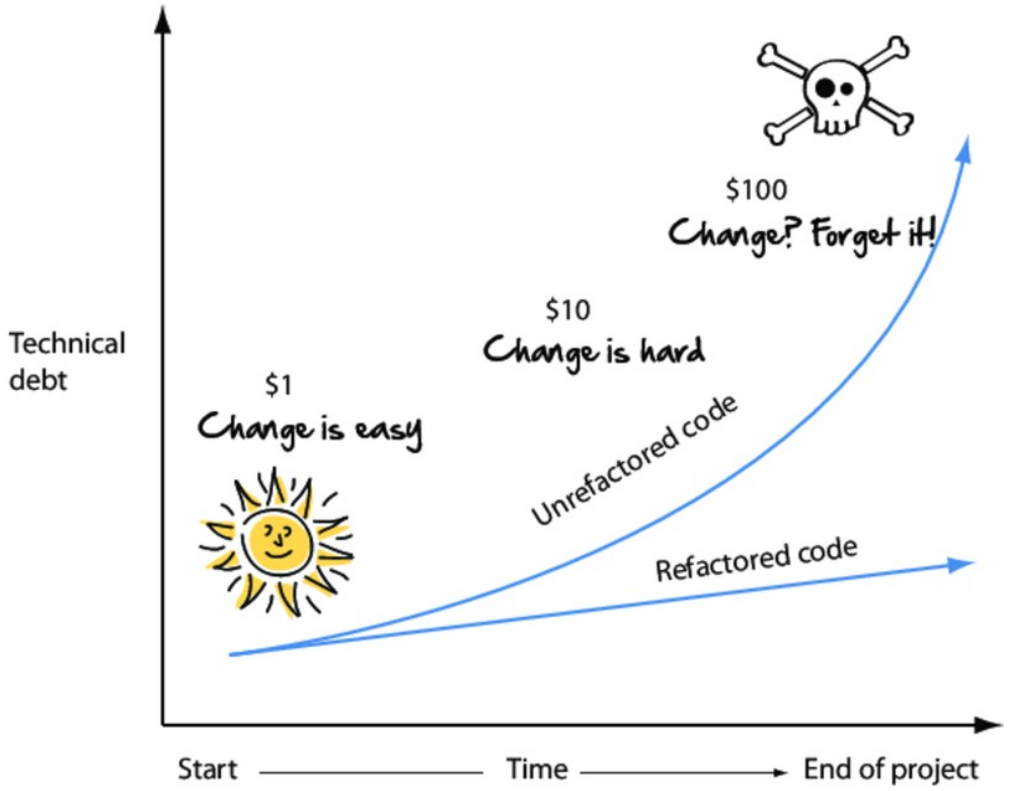
- Reflect on your burn-down chart: accumulates "interest"
  - Difficulties to make changes
  - Slower progress
  - Missing deadline…



http://commadot.com/ux-and-technical-debt/

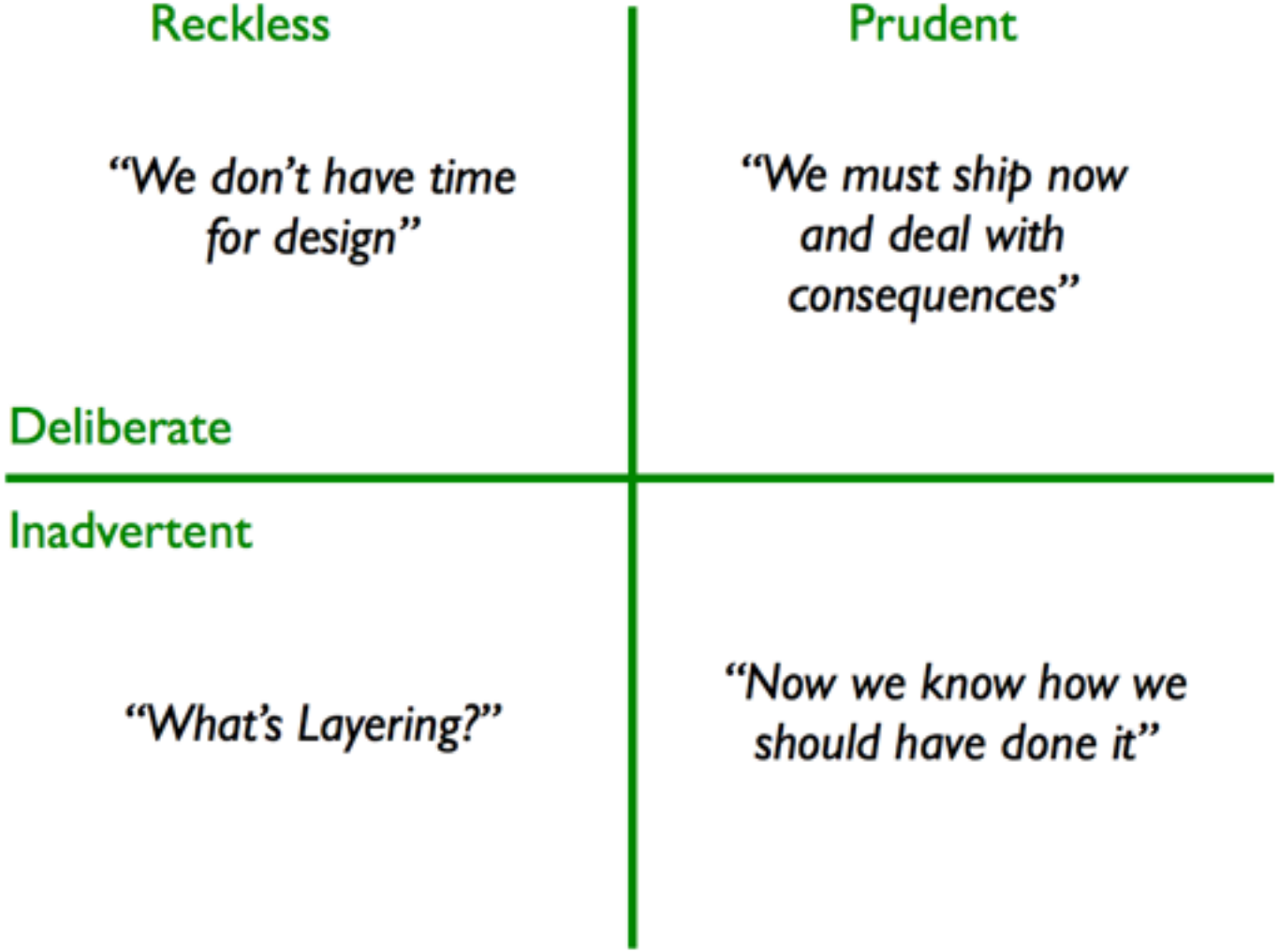
Technical Debt

- Solutions: Refactoring to pay off "technical debt"



https://www.safaribooksonline.com/library/view/the-agile-samurai/9781680500066/f_0091.html

Technical Debt Quadrant



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

Code "Smells"

```
41 @    static MappedField validateQuery(final Class clazz, final Mapper mapper, final StringBuilder origProp, final FilterOperator op, final
42          MappedField mf = null;
43          final String prop = origProp.toString();
44          boolean hasTranslations = false;
45          if (!origProp.substring(0, 1).equals("$")) {
46              final String[] parts = prop.split( regex: "\\.");
47              if (clazz == null) { return null; }
48              MappedClass mc = mapper.getMappedClass(clazz);
49              //CHECKSTYLE:OFF
50              for (int i = 0; ; ) {
51                  //CHECKSTYLE:ON
52                  final String part = parts[i];
53                  boolean fieldIsArrayOperator = part.equals("$");
54                  mf = mc.getMappedField(part);
55                  //translate from java field name to stored field name
56                  if (mf == null && !fieldIsArrayOperator) {
57                      mf = mc.getMappedFieldByJavaField(part);
58                      if (validateNames && mf == null) {
59                          throw new ValidationException(format("The field '%s' could not be found in '%s' while validating
60                      }
61                      hasTranslations = true;
62                      if (mf != null) {
63                          parts[1] = mf.getNameToStore();
64                      }
65                  }
66                  i++;
67                  if (mf != null && mf.isMap()) {
68                      //skip the map key validation, and move to the next part
69                      i++;
70                  }
71                  if (i >= parts.length) {
72                      break;
73                  }
74                  if (!fieldIsArrayOperator) {
75                      //catch people trying to search/update into @Reference/@Serialized fields
76                      if (validateNames && !canQueryPast(mf)) {
77                          throw new ValidationException(format("Cannot use dot-notation past '%s' in '%s'; found while vali
78                      }
79                      if (mf == null && mc.isInterface()) {
80                          break;
81                      } else if (mf == null) {
82                          throw new ValidationException(format("The field '%s' could not be found in '%s'", prop, mc.getClazz().getName()));
83                      }
84                      //get the next MappedClass for the next field validation
85                      mc = mapper.getMappedClass((mf.isSingleValue()) ? mf.getType() : mf.getSubClass());
86                  }
87              }
88
89              //record new property string if there has been a translation to any part
90              if (hasTranslations) {
91                  origProp.setLength(0); // clear existing content
92                  origProp.append(parts[0]);
93                  for (int i = 1; i < parts.length; i++) {
```

What's a prop?

What's a part?

Eeek!

Why all the null checks?

Control the loop

Comments, because code is unclear

Parameter mutation!

https://blog.jetbrains.com/idea/2017/09/code-smells-too-many-problems/

Code "Smells"

- According to Martin Fowler, "a code smell is a surface indication that usually corresponds to a deeper problem in the system".
  - Not bugs: Increase risk of bugs
  - Weaknesses of design: Slow down development
  - Bad practices: Factors contribute to technical debt

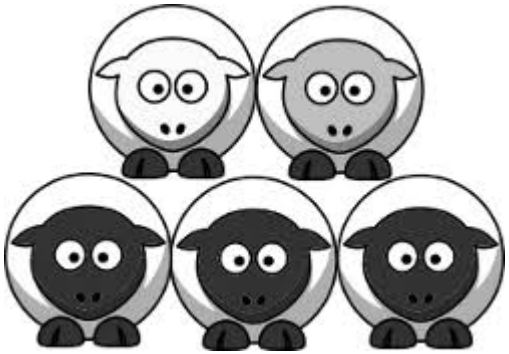
https://en.wikipedia.org/wiki/Code_smell

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 (Clone)

Observe: duplicated code



- Consequences:
  - Bugs are duplicated
  - Changes are more expensive
- 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

Duplicated Code Example

```
extern int array_a[];
extern int array_b[];

int sum_a = 0;

for (int i = 0; i < 4; i++)
        sum_a += array_a[i];

int average_a = sum_a / 4;

int sum_b = 0;

for (int i = 0; i < 4; i++)
        sum_b += array_b[i];

int average_b = sum_b / 4;
```

https://en.wikipedia.org/wiki/Duplicate_code

Refactoring: Extract method

```
extern int array_a[];
extern int array_b[];

int sum_a = 0;

for (int i = 0; i < 4; i++)
        sum_a += array_a[i];

int average_a = sum_a / 4;

int sum_b = 0;

for (int i = 0; i < 4; i++)
        sum_b += array_b[i];

int average_b = sum_b / 4;
```

Step 1: Identify code fragment that can be grouped together

Step 2: Turn the fragment into a method

Step 3: Replace the fragment with a call to the method

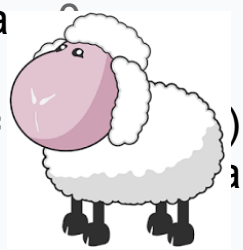
https://en.wikipedia.org/wiki/Duplicate_code

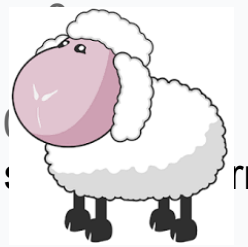
Step 1: Identify

```
extern int array_a[];
extern int array_b[];

int sum_a
for (int i =
                 array_a[i];
int average_a = sum_a / 4;

int sum_b
for (int i =
                 rray_b[i];
int average_b = sum_b / 4;
```





https://en.wikipedia.org/wiki/Duplicate_code

Step 2: Extract

```
extern int array_a[];
extern int array_b[];

int sum_a = 0;

for (int i = 0; i < 4; i++)
        sum_a += array_a[i];

int average_a = sum_a / 4;
```

```
int sum_b = 0;

for (int i = 0; i < 4; i++)
        sum_b += array_b[i];

int average_b = sum_b / 4;
```

```
int calc_average_of_four(int* array)
{
        int sum = 0;
        for (int i = 0; i < 4; i++)
                sum += array[i];

        return sum / 4;
}
```

https://en.wikipedia.org/wiki/Duplicate_code

Step 3: Replace

```
extern int array_a[];
extern int array_b[];
```

```
int sum_a = 0;

for (int i = 0; i < 4; i++)
        sum_a += array_a[i];

int average_a = sum_a / 4;
```

```
int sum_b = 0;

for (int i = 0; i < 4; i++)
        sum_b += array_b[i];

int average_b = sum_b / 4;
```

```
int calc_average_of_four(int* array)
{
        int sum = 0;
        for (int i = 0; i < 4; i++)
                sum += array[i];

        return sum / 4;
}
```

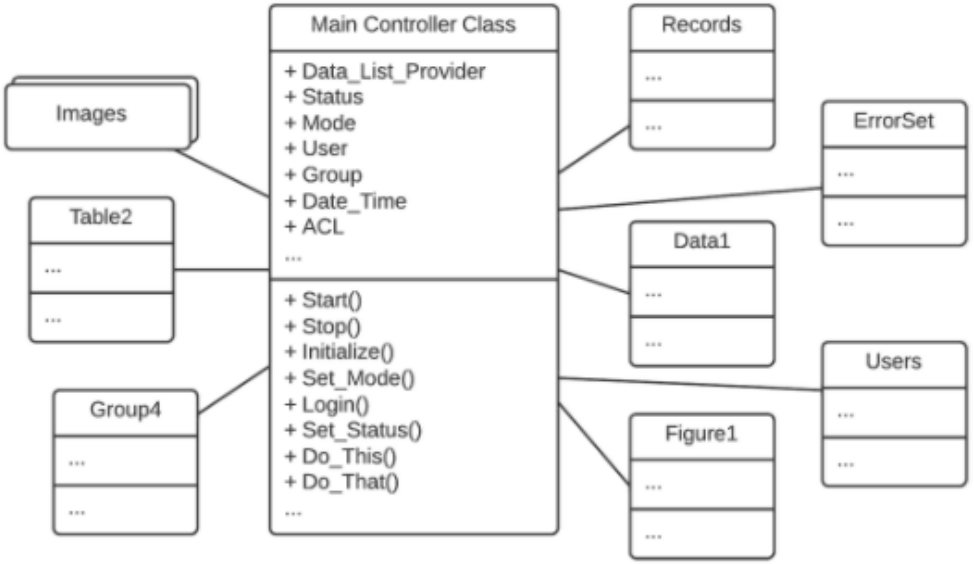
```
//After Refactoring
extern int array1[];
extern int array2[];
int avg1 = calc_average_of_four(array1);
int avg2 = calc_average_of_four(array2);
```

https://en.wikipedia.org/wiki/Duplicate_code

Bad Smells: Large class (God class)

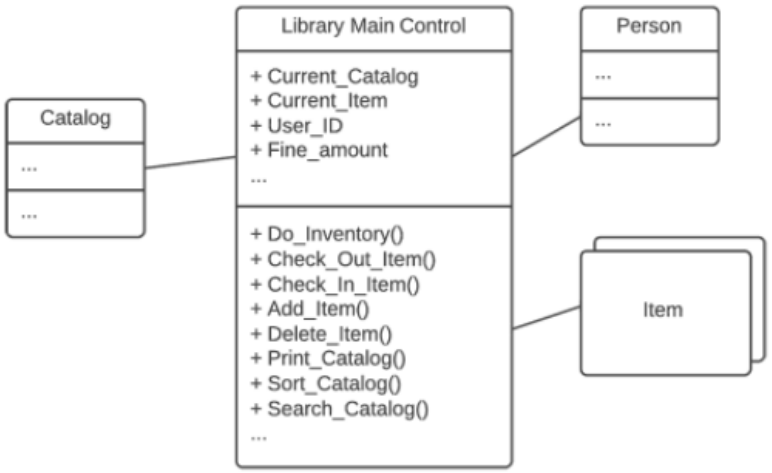
- Observe:
  - Too many variables
  - Too many lines of code
  - Too many functions

Some recommend that 200 lines is a good limit for a class



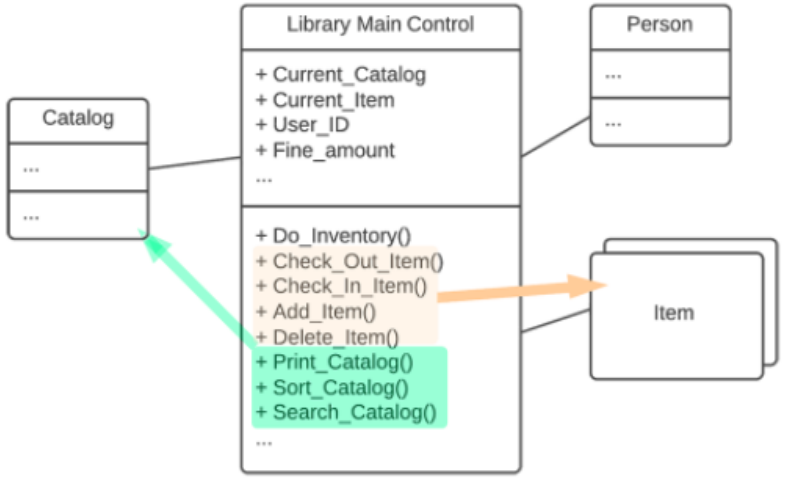
Bad Smells: Large class (God class)

- Refactoring techniques:
  - Extract Class
  - Extract Subclass
  - Extract Interface



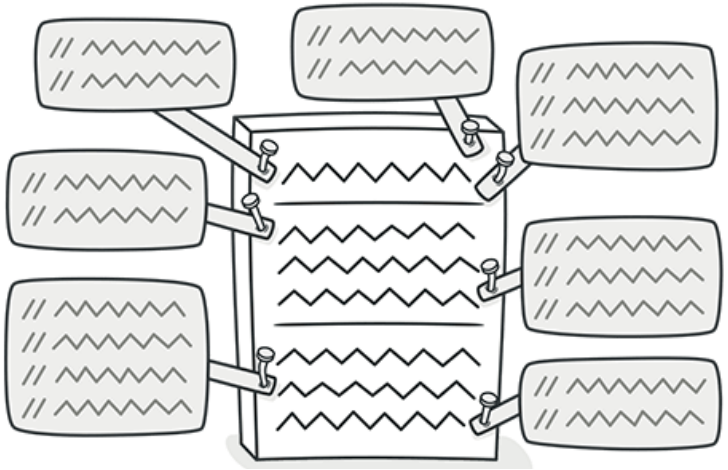
Tip!

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



Bad Smells: Too many comments

- Observe: code is filled with explanatory comments
- Comments are necessary:
  - Why something is implemented in a particular way
  - Complex algorithm



Bad Smells: Too many comments

- Don't comment bad code---rewrite it!

  —Brian W. Kernighan, *The Elements of Programming Style*

The best comment is a good name for a method or class

```
var bd = '1988-7-6'; // birth date
var md = '2015-6-3'; // married date
```

```
int a = 30; // age
int myVar = 17; // id of an individual
```

Bad Smells: Too many comments

- Don't comment bad code---rewrite it!

  —Brian W. Kernighan, *The Elements of Programming Style*

The best comment is a good name for a method or class

```
var bd = '1988-7-6'; // birth date
var md = '2015-6-3'; // married date
```

```
var birthDate = '1988-7-6';
var marriedDate= '2015-6-3';
```

```
int a = 30; // age
int myVar = 17; // id of an individual
```

```
int age = 30;
int individualID = 17;
```

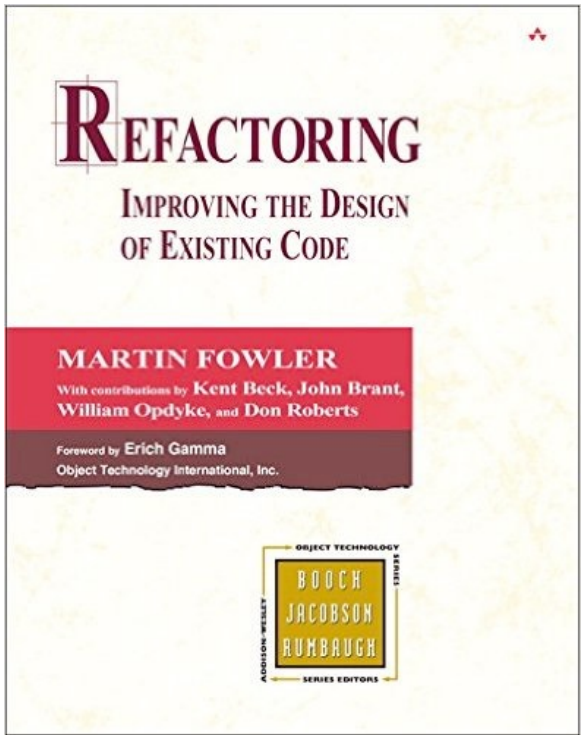
Bad Smells: Too many comments

- Refactoring techniques:
  - Extract Variables: If a comment is intended to explain a complex expression, the expression should be split into understandable subexpressions
  - Extract Method: If a comment explains a section of code, this section can be turned into a separate method.
  - Rename Method/Variable: Give a method/ variable a self-explanatory name.
  - Introduce Assertion: If you need to assert rules about a state that is necessary for the system to work.

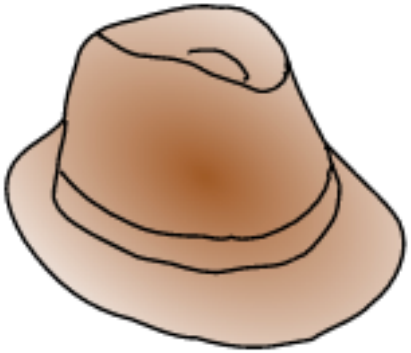
More bad smells and refactoring techniques

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

And see this website: https://sourcemaking.com/refactoring



The Metaphor of Two Hats

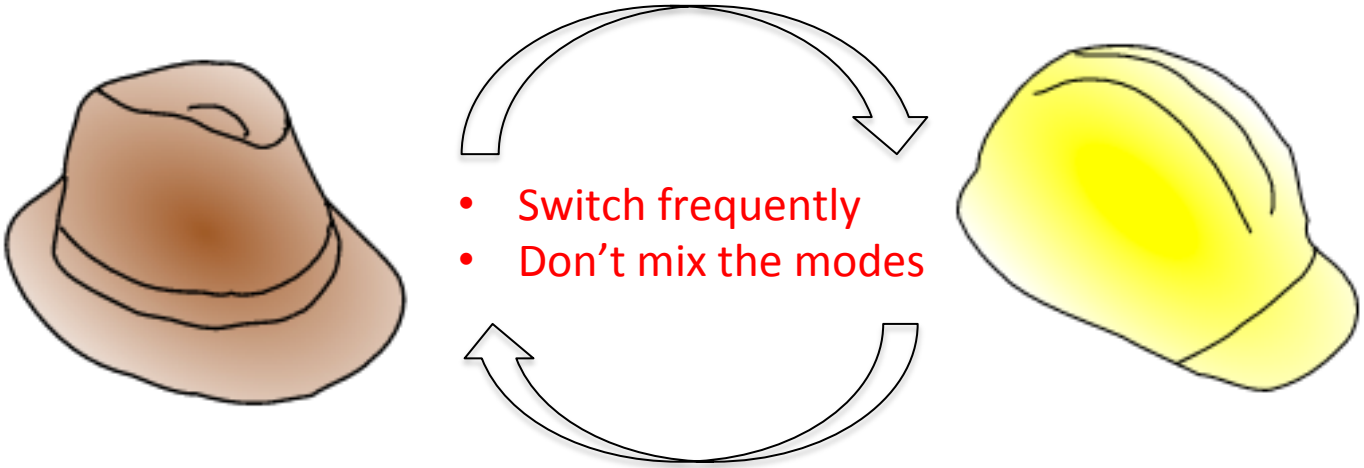


Refactoring
- Not adding new functions
- Not adding tests
- Not changing tests (unless necessary)

Adding Function
- Change code to add functions
- New tests, break existing tests
- Wise to keep change small

https://martinfowler.com/articles/workflowsOfRefactoring

The Metaphor of Two Hats



Refactoring
- Not adding new functions
- Not adding tests
- Not changing tests (unless necessary)

Adding Function
- Change code to add functions
- New tests, break existing tests
- Wise to keep change small

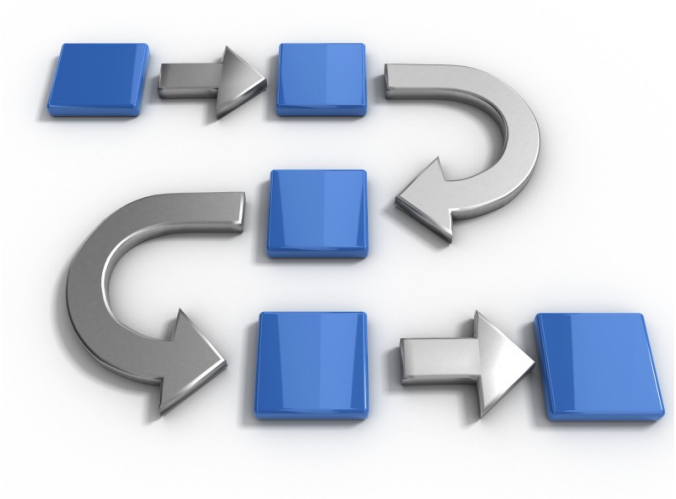
https://martinfowler.com/articles/workflowsOfRefactoring

Good test cases are critical

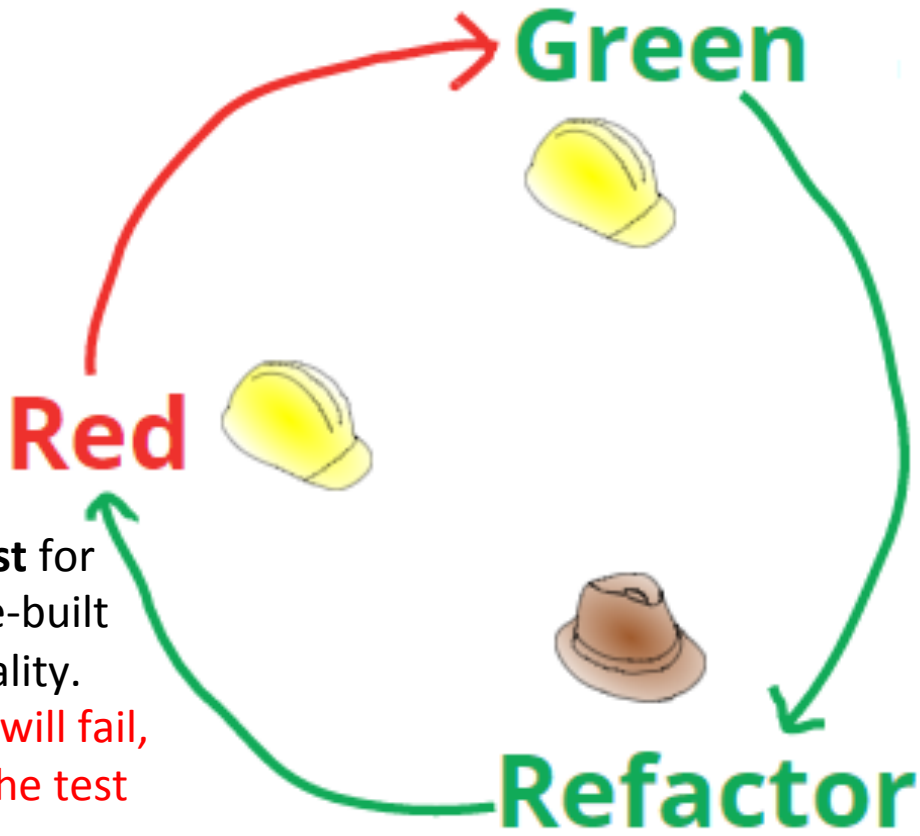
- Refactoring replaces working code with new code
- Research shows that new code is more likely to contain bugs than older code
- Automated regression testing is critical, e.g. xUnit
- Regression testing verifies:
  - The new code replicates the behavior of the old code
  - The new code doesn't add new bugs

Refactoring Workflows

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



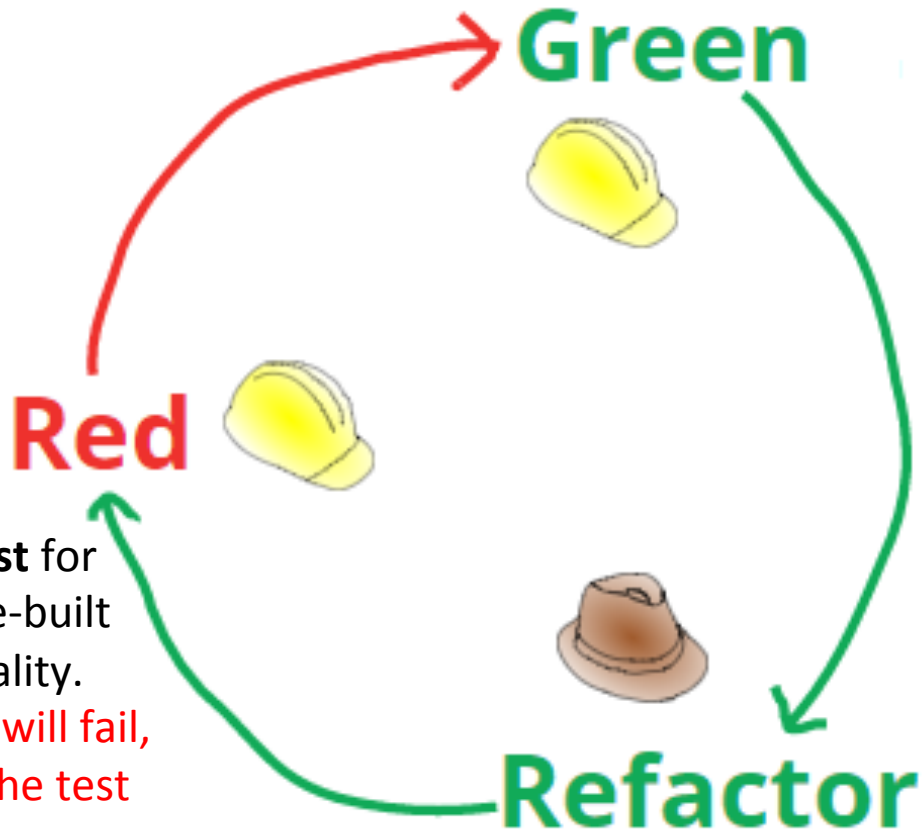
Two Hats in TDD



1. **Add a Test** for yet-to-be-built functionality. This test will fail, making the test suite red.

https://martinfowler.com/articles/workflowsOfRefactoring

Two Hats in TDD

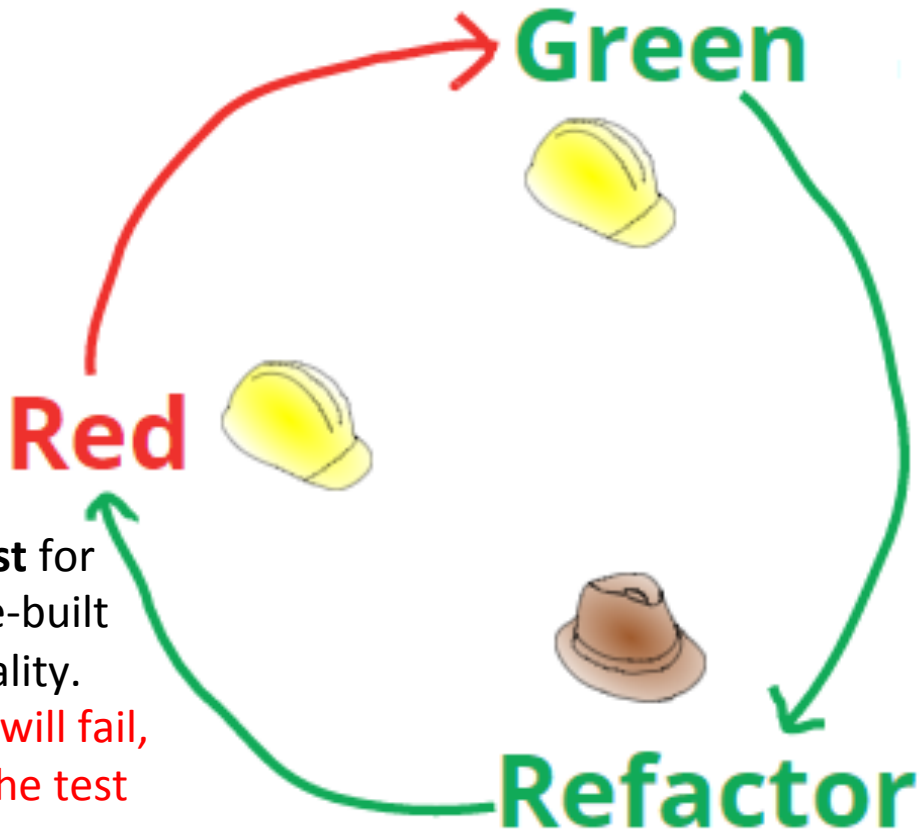


2. Make it work: implement the necessary functionality simply but crudely. Make the test pass (go green).

1. **Add a Test** for yet-to-be-built functionality. This test will fail, making the test suite red.

https://martinfowler.com/articles/workflowsOfRefactoring

Two Hats in TDD



1. **Add a Test** for yet-to-be-built functionality. This test will fail, making the test suite red.

2. Make it work: implement the necessary functionality simply but crudely. Make the test pass (go green).

3. Make it clean: Use refactoring to ensure the overall code base is as clean and well-design as possible. Make sure the test still pass.

https://martinfowler.com/articles/workflowsOfRefactoring

Litter-picking

- **Scenario**: A messy area in the code is found
  - Less capable developer
  - Capable developer in a hurry
  - Didn't understand solution

- Replace bad code with an elegant solution
- Opportunistic refactoring:

  "Leave the code cleaner than you found it"

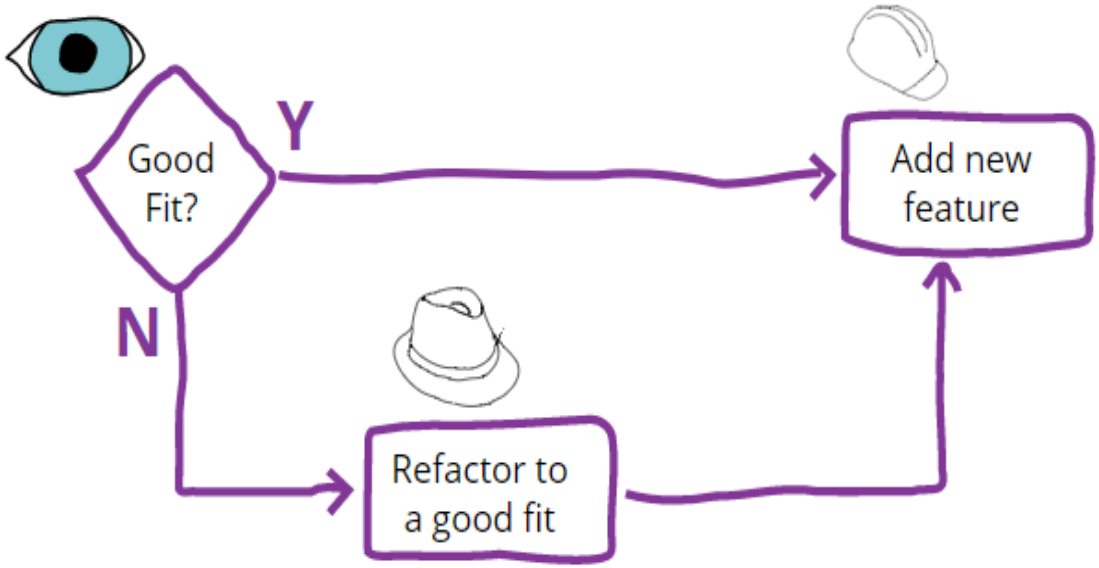
  – Robert "Uncle Bob" Martin

Comprehension Refactoring

- **Scenario**: Reading code and finding it hard to understand
- Refactor to move your understanding of the design out of your head and into the code:
  - Help the next reader benefit (possibly you!) from your experience and effort
- Opportunistic: similar to litter pick up (but focus on making code understandable).

Preparatory Refactoring

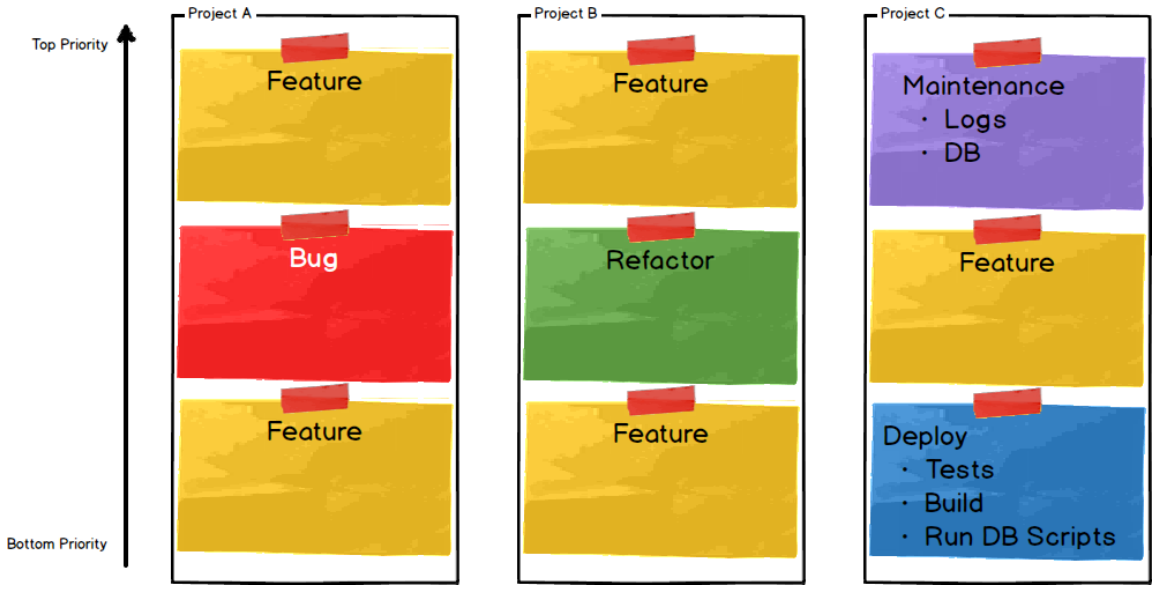
- **Scenario**: need to add a new feature
- But it's easier to add the new feature after making changes to existing code
- Provide immediate benefit!!



https://martinfowler.com/articles/workflowsOfRefactoring/#prep-task

Planned Refactoring

- **Scenario**: plan refactoring in the project schedule
  - E.g. Put refactoring tasks in your backlog…
- This may suggest that you're waiting too long to refactor (best to avoid this situation)



Long Term Refactoring

- **Scenario**: restructuring requires bigger changes than can be done in a single development episode, by one or two developers
  - Replacing a large module
  - Changing your database framework
  - Untangling some dependencies

- Refactoring technique: Branch By Abstraction
  - Use an abstraction layer that supports the current and a replacement implementation. See: https://martinfowler.com/bliki/BranchByAbstraction.html

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
- …

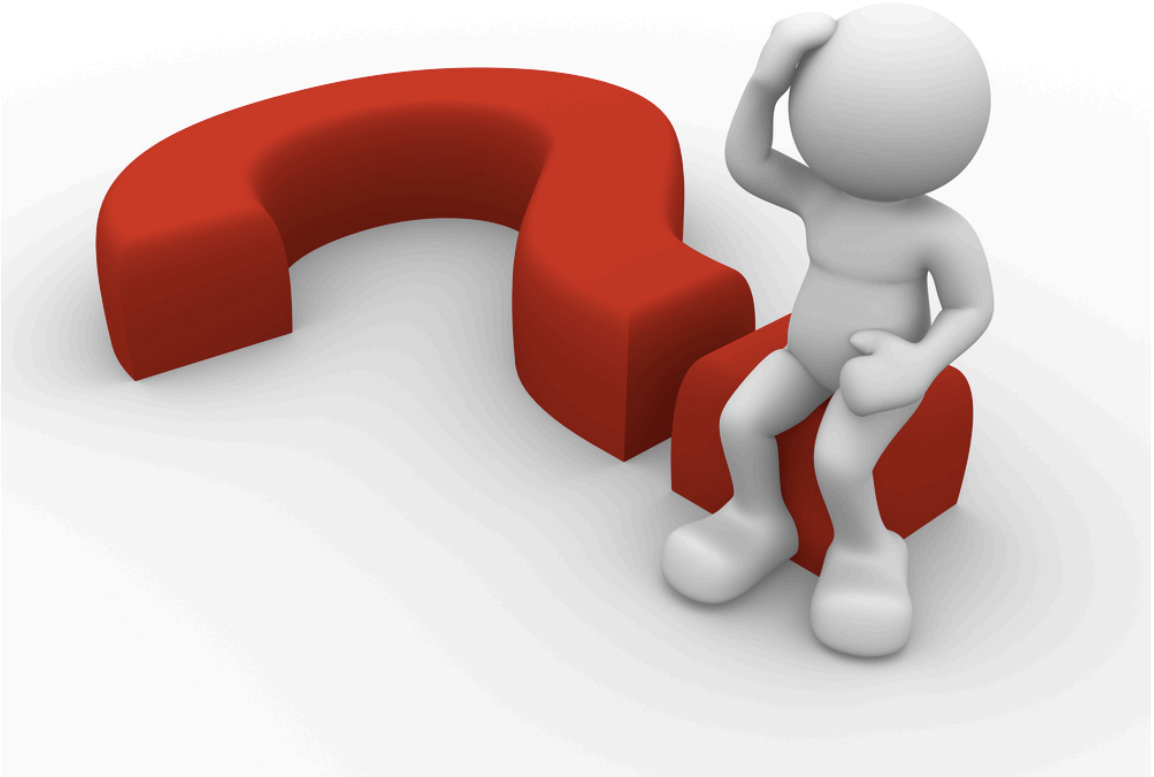
See
https://www.jetbrains.com/help/pycharm/2016.1/refactoring-source-code.html

Refactoring at Google

"At Google, refactoring is very important and necessary/inevitable for any code base. If you're writing a new app quickly and adding lots of features, your initial design will not be perfect. Ideally, do small refactoring tasks early and often, as soon as there is a sign of a problem. Refactoring is unglamorous because it does not add features. At many companies, people don't refactor because you don't get promoted for it, and their code turns into hacky beasts." Google feels refactoring is so important that there are company-wide initiatives to make sure it is encouraged and rewarded. Common reasons not to do it are incorrect: 'Don't have time; features more important' -- You will pay more cost, time in adding features (because it's painful in current design), fixing bugs (because bad code is easy to add bugs into), ramping up others on code base (because bad code is hard to read), and adding tests (because bad code is hard to test), etc. 'We might break something' -- Sign of a poor design from the beginning, where you didn't have good tests. For same reasons as above, you should fix your testing situation and code. 'I want to get promoted and companies don't recognize refactoring work' -- This is a common problem. Solution varies depending on company. Seek buy-in from your team, gather data about regressions and flaws in the design, and encourage them to buy-in to code quality. An important line of defense against introducing new bugs in a refactor is having solid unit tests (before the refactor)."

-- Victoria Kirst, Software Engineer, Google

Questions?



STEVENS

INSTITUTE *of* TECHNOLOGY

THE INNOVATION UNIVERSITY®

stevens.edu

Thank You