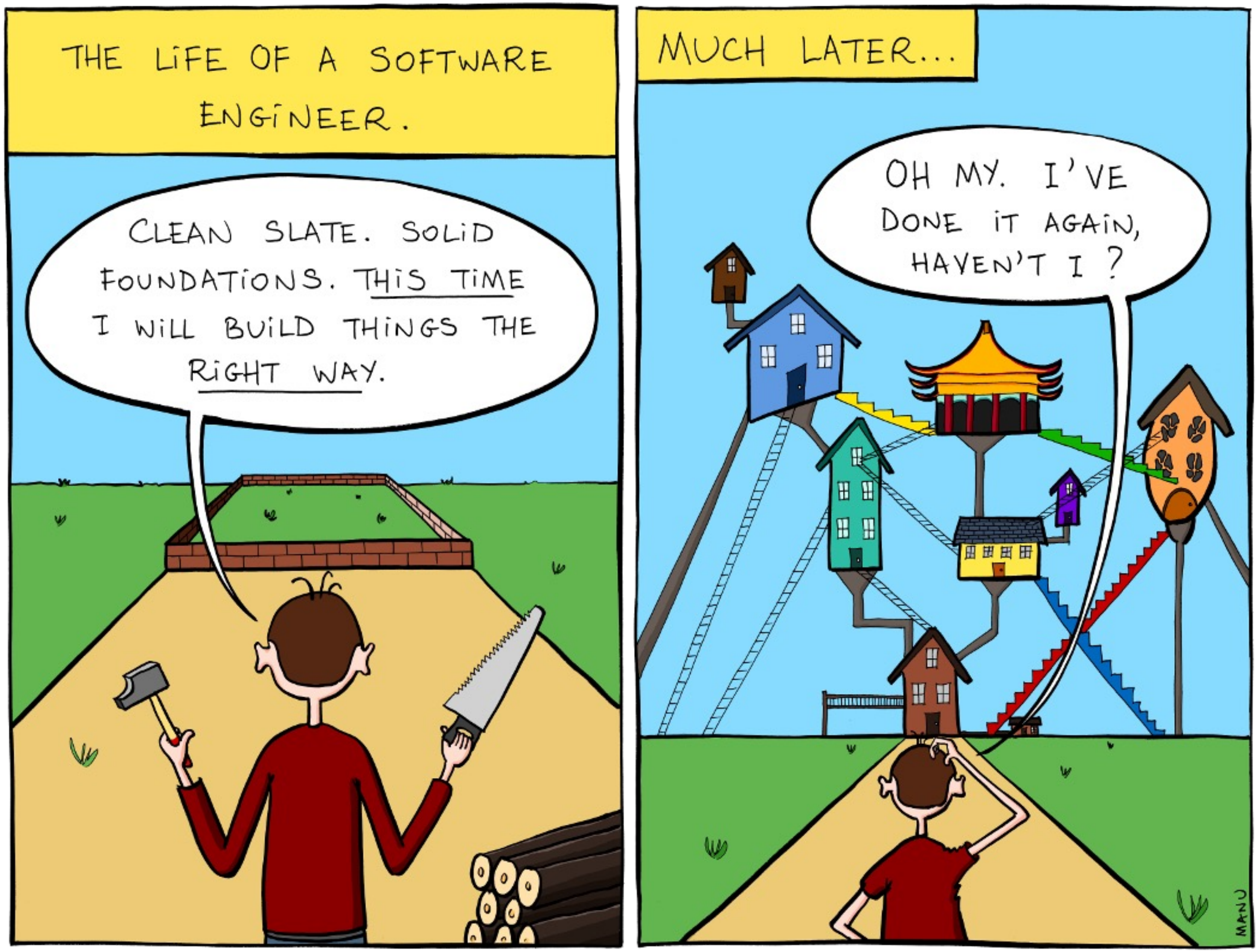

THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

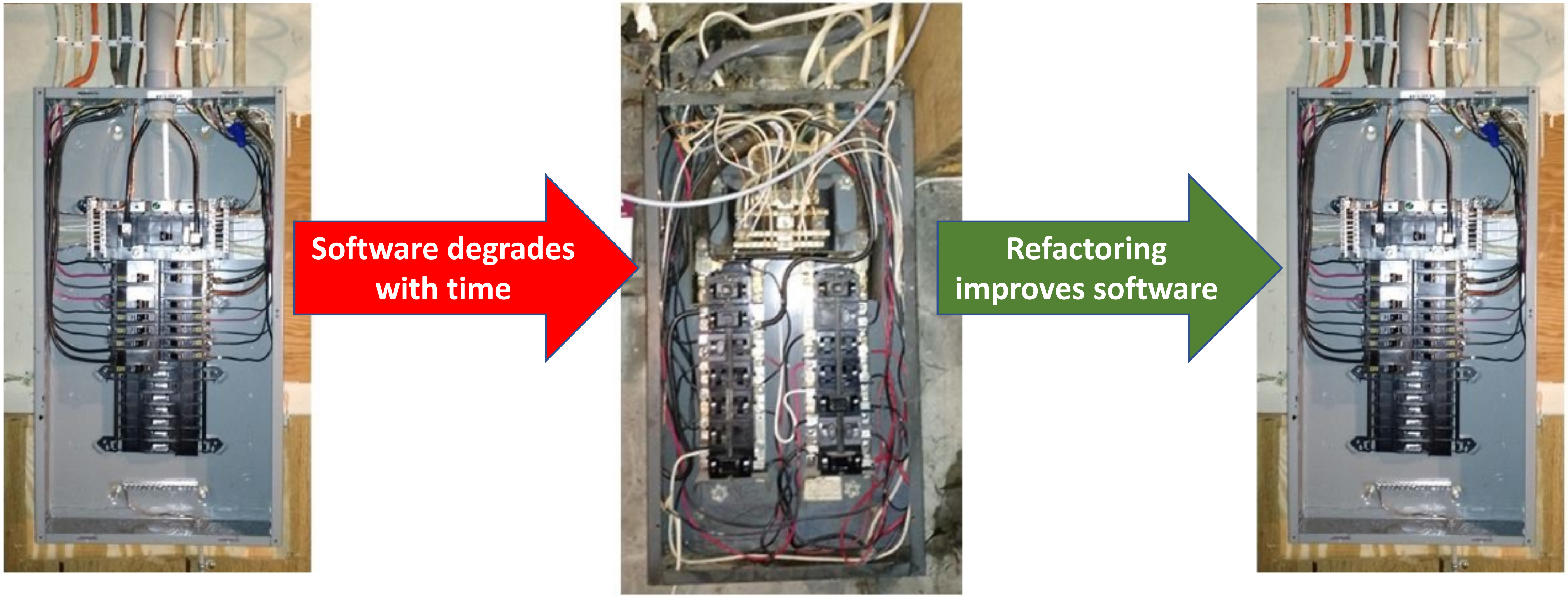
Refactoring

CSCI 2134: Software Development

Agenda

- Lecture Contents
  - Motivation for Refactoring
  - Types of Refactoring
  - Refactoring Safely
- Brightspace Quiz
- Readings:
  - This Lecture: Chapter 24
  - Next Lecture: Chapter 3

What’s the Problem?



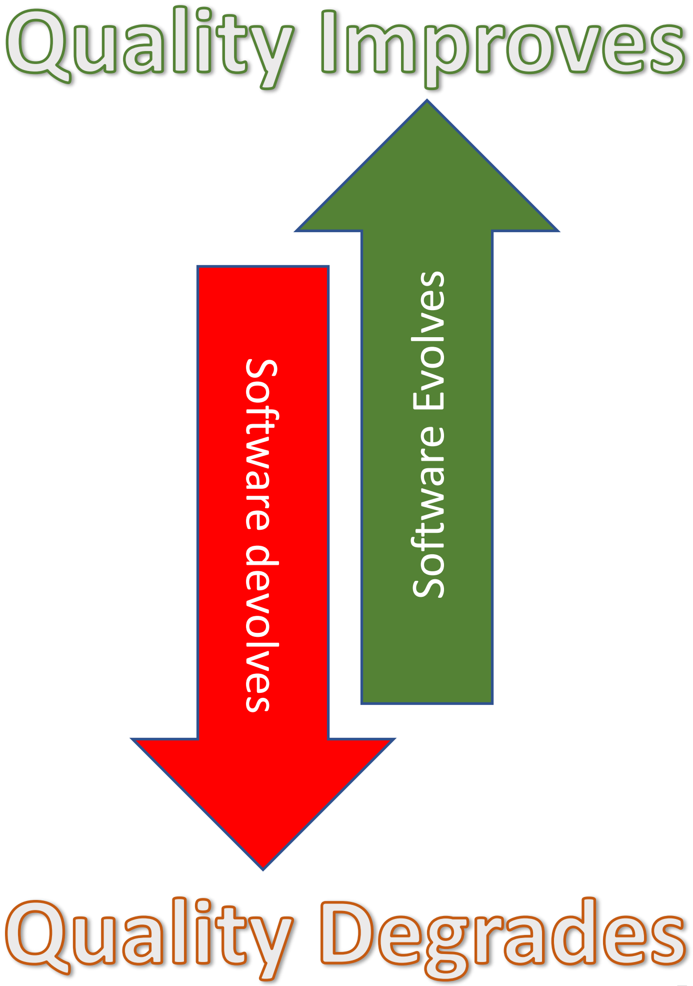
Software degrades with time

Refactoring improves software

Software Changes During Development

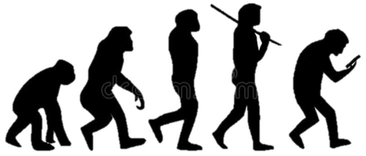
- Requirements change over the course of the project
  - Clients change their minds
  - Purpose of software changes
  - Etc.
- Design changes over the course of a project
  - Obstacles appear that need work-arounds
  - Better solutions and approaches are discovered
  - Constraints are changed or added
- Implementation changes over the course of a project
  - Design flaws are fixed
  - Work-arounds created
  - Short-cuts taken
  - Bugs are fixed
- **Why?**

  We never get it right the first time!



Software Evolves (or Devolves)

Software Evolution



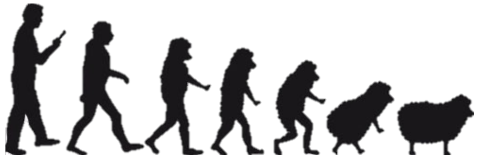
Software quality improves

- Bugs are fixed
- Useless code removed
- Bad code improved
- Complexity is reduced

Software Evolves

Software evolution should result in better quality code

Software Devolution



- Band-aid fixes
- Short-cuts
- Poorly thought-out design
- Undocumented changes
- Hacks
- Etc.

Software devolves

Software quality degradation should be avoided!

Why Should I Care?

- Software evolution must be consciously managed

  Many developers assume software evolution happens naturally.

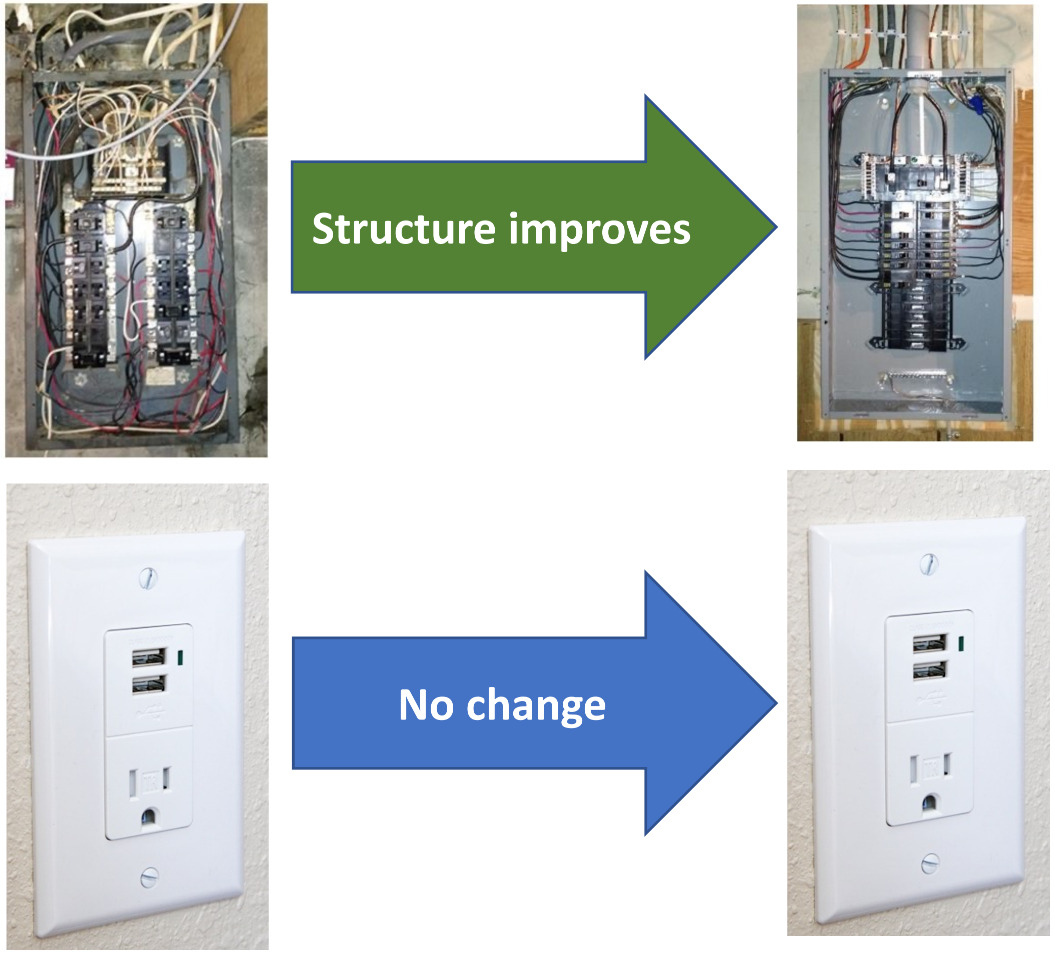
- **Key Idea**: Unmanaged software evolution results in devolution
  - Code becomes more complex, less maintainable, more brittle, etc.
  - **Technical debt:** the implied “cost” incurred when we do not fix problems that will affect the software in the future

- **Key Idea**: Managed software evolution treats every change as an opportunity to improve the software quality
  - One important strategy is **refactoring**

Refactoring

- **Definition**: Refactoring is “a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior” (Fowler 1999)

- **Alternative Definition**: Improving the code without changing the function.



Structure improves

No change

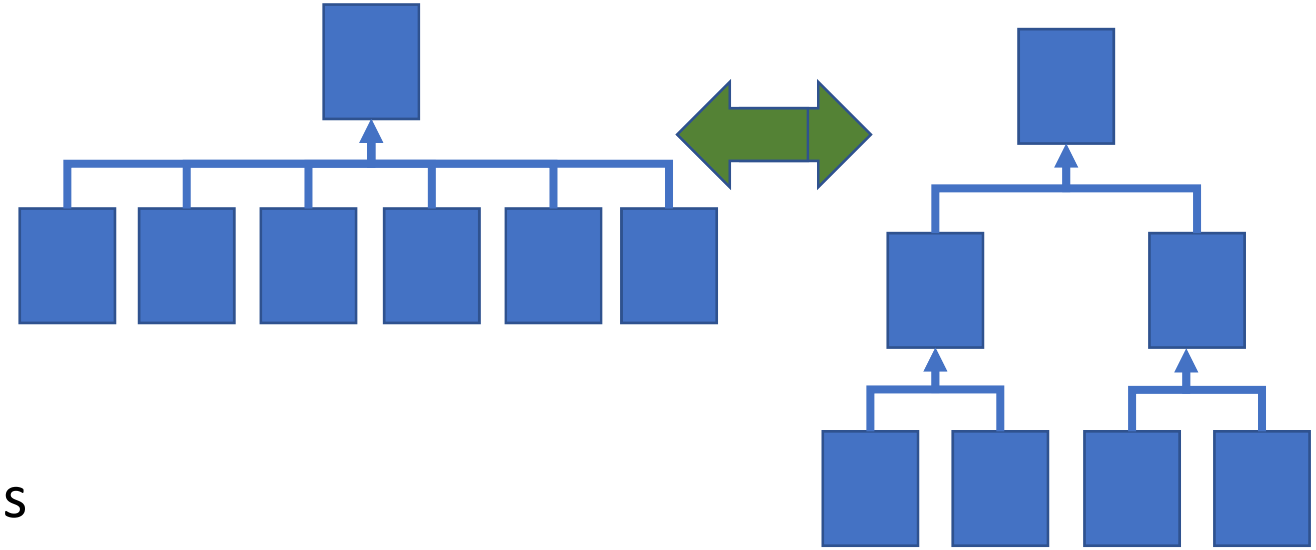
Examples of Refactoring

- Improving readability and maintainability
  - Removing magic constants
  - Making variable names more informative
  - Decomposing overly long methods

`2000 → AngryBot.NO_BOT_SCORE`

`i → districtIndex`

- Improving understandability
  - Reorganizing the class hierarchy
  - Removing dead or unused code
- Improving flexibility
  - Improving interfaces or APIs
  - Improving base and super classes



What is Refactoring?

- Refactoring is **not** bug fixing
- Refactoring is **not** adding features
- Refactoring is **not** design changes

- Why?
  - All the above **change functionality**
  - Refactoring improves code, not functionality

How Do I Know That Refactoring is Needed?

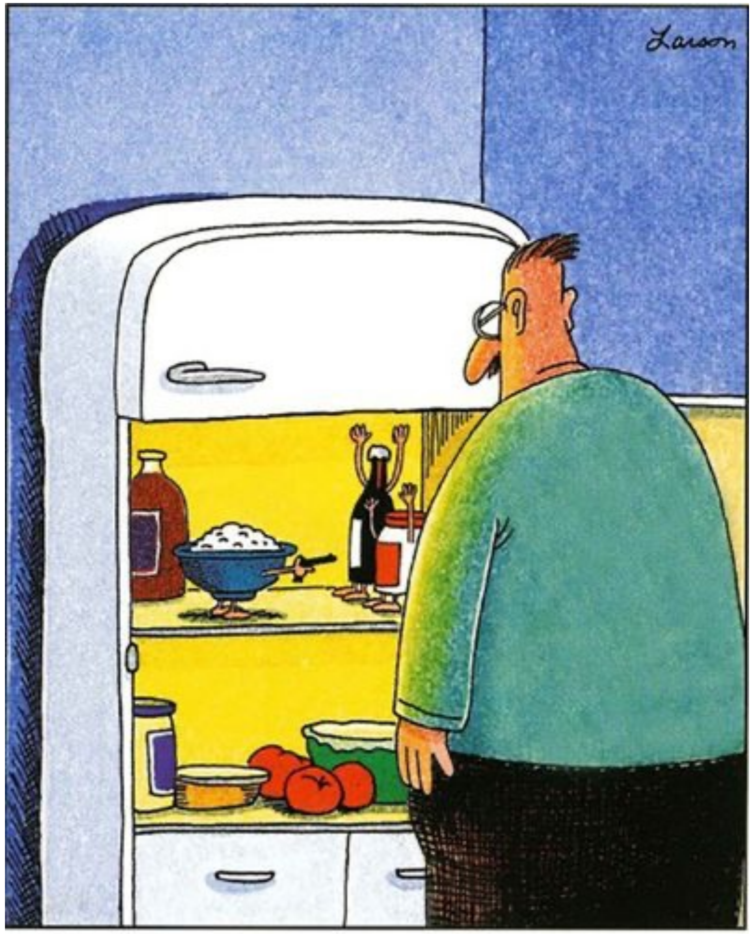
- How do we know when food goes bad?

  It starts to *smell*

- **Idea**: Bad smells tip our brain off that something is wrong, and we need to be careful

- **Idea**: “Code smells” are indicators of bad code.

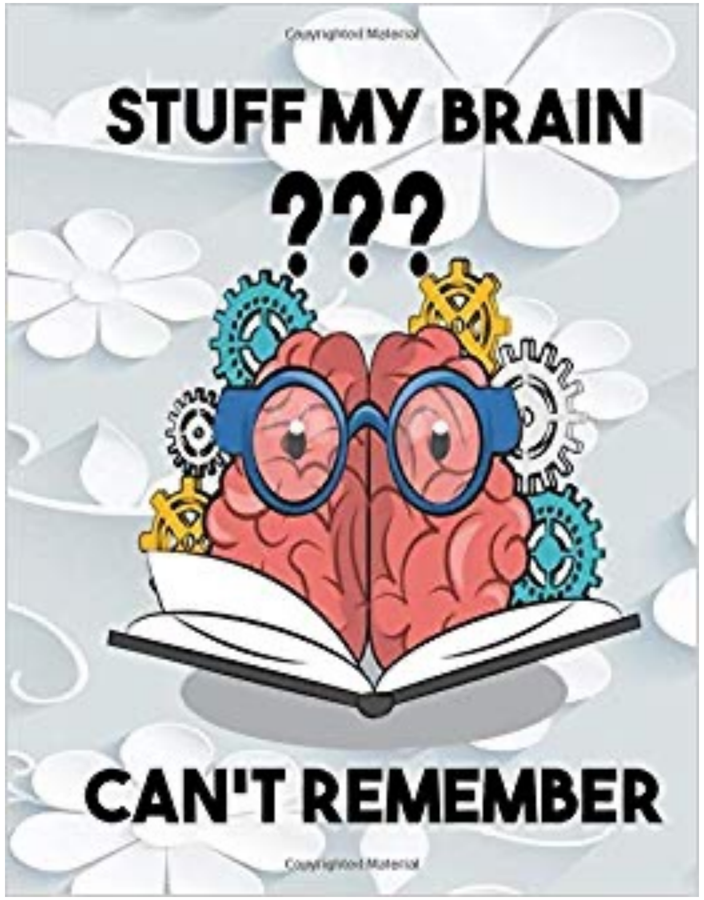
- We say that code *smells* if it has degenerated and needs to be fixed and/or refactored.



When potato salad goes bad

Code Smells: Refactoring May Be Needed

- Code is duplicated
- A method is too long
- A loop is too long or too deeply nested
- A method has a lot of parameters
- A method uses more features of another class than of its own class
- Poor naming of variables or methods
- Data members are public
- Comments are used to explain difficult code
- Global variables are used



Code Smells: Refactoring May Be Needed

- A class has poor cohesion
- A class interface does not provide a consistent level of abstraction
- Changes within a class tend to be compartmentalized
- Changes require parallel modifications to multiple classes
- Inheritance hierarchies have to be modified in parallel
- Related data items that are used together are not organized into classes
- A class doesn’t do very much
- A “middleman” object isn’t doing anything
- One class is overly intimate with another

Types of Refactoring

- Data-Level refactoring

  Improve use of variables and data

- Statement-level refactoring

  Improve use of individual statements

- Routine-level refactoring

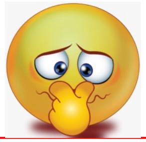
  Improve code at the routine/method level

- Class-implementation refactoring
- Class-interface refactoring
- System-level refactoring

Data-Level Refactoring

- Introduce an intermediate variable

```
totalGrade = 0.1 * quizzes + 0.2 * (assn1 + assn2 + assn3 + assn4) / 4 +
             0.2 * (lab1 + lab2 + lab3 + lab4 + lab5 + lab6 + lab7 +
             lab8 + lab9) / 9 + 0.2 * (midterm1 + midterm2) / 2 + 0.5 * final;
```



```
assignmentMark = (assn1 + assn2 + assn3 + assn4) / 4;
labMark = (lab1 + lab2 + lab3 + lab4 + lab5 + lab6 + lab7 + lab8 + lab9) / 9;
midtermMark = (midterm1 + midterm2) / 2;
totalGrade = 0.1 * quizzes + 0.2 * assignmentMark + 0.2 * labMark +
             0.2 * midtermMark + 0.5 * final;
```

Data-Level Refactoring

- Convert a multiuse variable to multiple single-use variables
- Use a local variable for local purposes rather than a parameter
- Replace a magic number with a named constant 🎦
- Rename a variable with a clearer or more informative name 🎦

Data-Level Refactoring:
Don’t Use Parameters Like Local Variables

- Poor practice

```
boolean contains(T item, Node head) {
  while (head != null && head.item != item) {
    head = head.next;
  }
  return head != null;
}
```

- Better practice

```
boolean contains(T item, Node head) {
  Node tmp = head;
  while (tmp != null && tmp.item != item) {
    tmp = tmp.next;
  }
  return tmp != null;
}
```

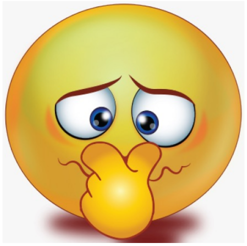
Why is this a problem?
Local variables are supposed to have one purpose, We assume that head refers to start of list. But this ceases to be true here. ☹

In some languages, modifications to parameters are seen by the calling function as well.

Statement-Level Refactoring

- Decompose complex Boolean expressions
- Move a complex Boolean expression into a well-named Boolean function

```
 if ((res == null) || (b == null) ||
     (b.getHeight() != height) || (b.getWidth() != width) ||
     (res.getHeight() != height) || (res.getWidth() != width)) {
   return null;
}
```

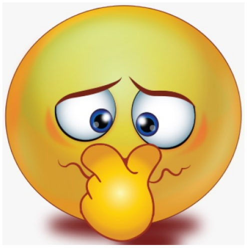


```
paramsAreNull = (res == null) || (b == null);
bIsNotSameSize = (b.getHeight() != height) || (b.getWidth() != width);
resIsNotSameSize = (res.getHeight() != height) || (res.getWidth() != width);
if (paramsAreNull || bIsNotSameSize || resIsNotSameSize) {
  return null;
}
```

Statement-Level Refactoring (cont.)

- Consolidate fragments that are duplicated within different parts of a conditional

```
if (head == null) {
  head = node;
  tail = node;
} else {
  tail.next = node;
  tail = node;
}
```

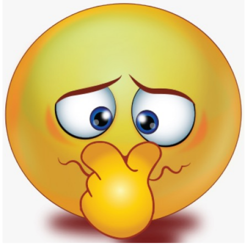


```
if (head == null) {
  head = node;
} else {
  tail.next = node;
}
tail = node;
```

Statement-Level Refactoring

- Use break or return instead of a loop control variable

```
found = false;
while (!found || node != null) {
  if (key == node.key) {
    found = true;
  } else {
    node = node.next;
  }
}
```

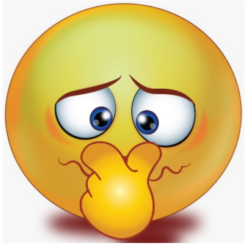


```
while (node != null) {
  if (key == node.key) {
    break;
  }
  node = node.next;
}
```

Statement-Level Refactoring

- Return as soon as you know the answer instead of assigning a return value within nested if-then-else statements

```
public Item get(int key) {
  Item found = null;
  for (Item item : list) {
    if (key == item.key) {
      found = item;
      break;
    }
  }
  return found;
}
```

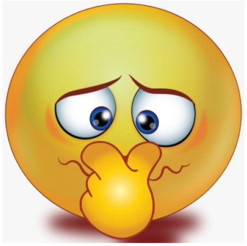


```
public Item get(int key) {
  for (Item item : list) {
    if (key == item.key) {
      return item;
    }
  }
  return null;
}
```

Routine-Level Refactoring

- Extract routine/extract method

```
public boolean hasDuplicates() {
  for (Item item : list) {
    int count = 0;
    for (Item item2 : list) {
      if (item.equals(item2)) {
        count++;
      }
    }
    if (count > 1) {
      return true;
    }
  }
  return false;
}
```



```
public boolean hasDuplicates() {
  for (Item item : list) {
    if (count(item) > 1) {
      return true;
    }
  }
  return false;
}

public boolean count(Item item) {
  int count = 0;
  for (Item item2 : list) {
    if (item.equals(item2)) {
      count++;
    }
  }
  return count;
}
```

Routine-Level Refactoring

- Remove a parameter
  - If a method does not use a parameter, it should be removed
  - Exception: methods that override other methods
    - But this itself is a code smell for the class hierarchy itself

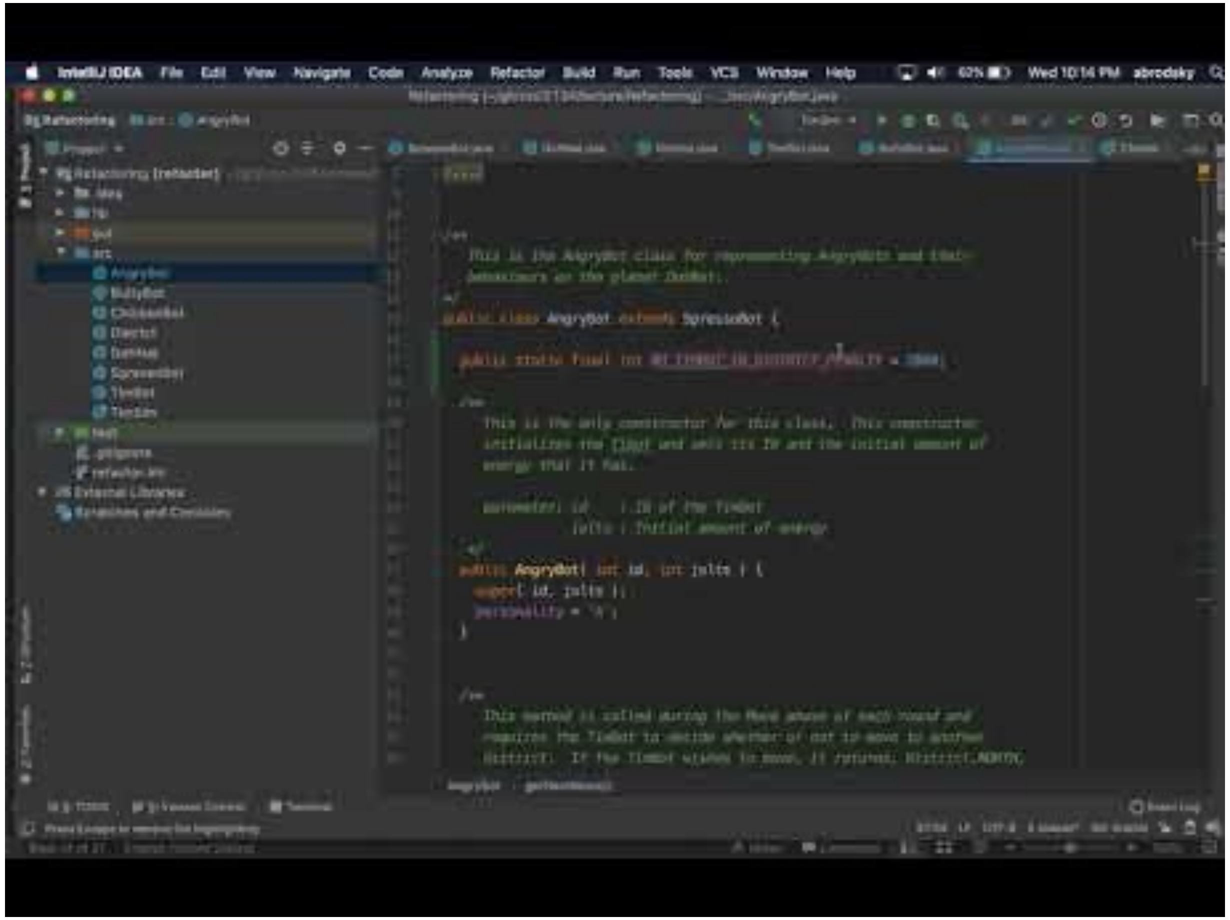
- Separate query operations from modification operations
  - Getters should not modify object state.
    - (In most cases) performing the same get operation should return the same value
  - Setters modify an object but typically do not return object properties

- Combine similar routines by parameterizing them

Class-Level and Class Interface Refactoring

- We will discuss class-level and interface level refactoring when we talk about design principles.
- One common modification is class renaming, which can be done easily in IntelliJ 🎦

Using IntelliJ to Refactor

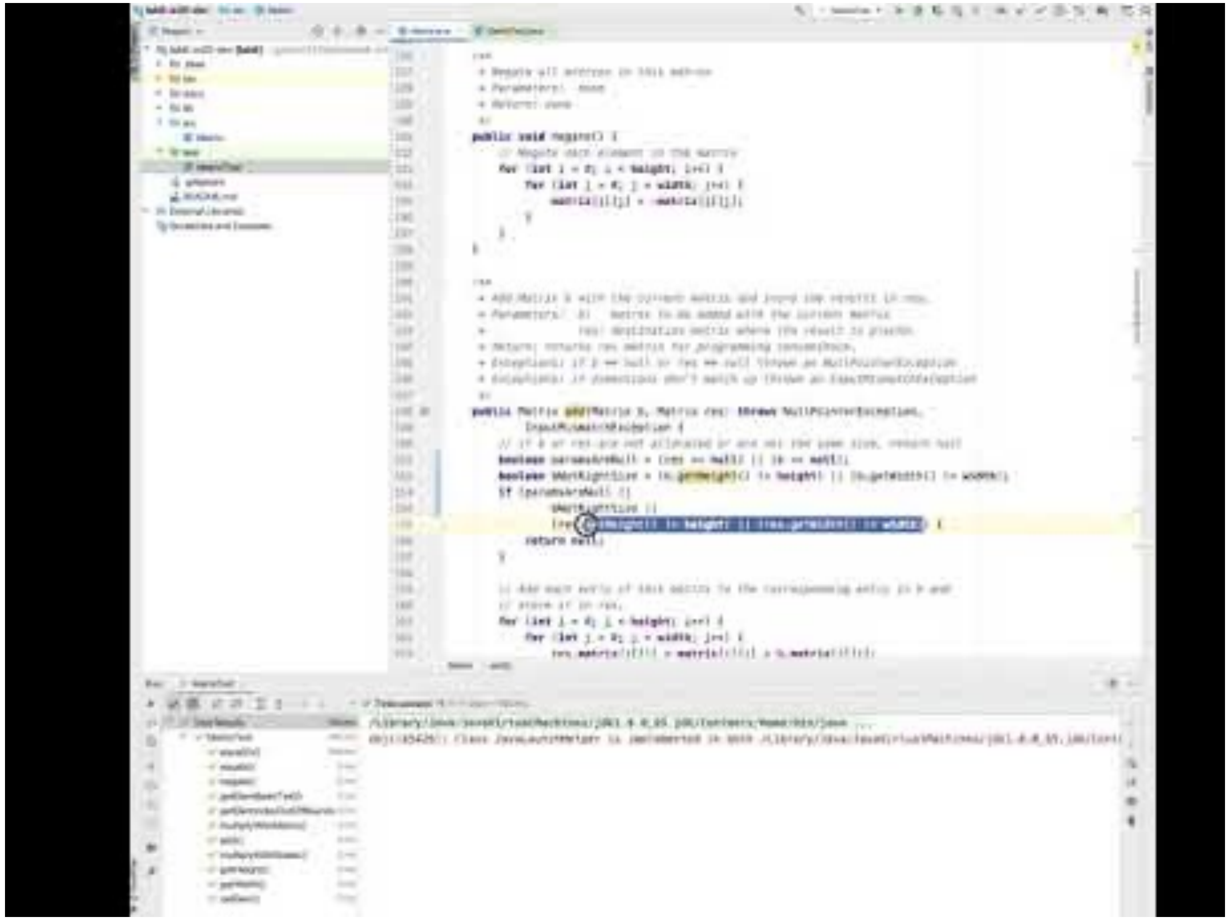


IntelliJ IDEA File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Refactoring
AngryBot
BullyBot
District
TimBot
External Libraries
Scratches and Consoles

```
/**
  This is the AngryBot class for representing AngryBots and their
  behaviours on the planet DohBot.
*/
public class AngryBot extends SpressoBot {

  public static final int [illegible] = [illegible];

  /**
    This is the only constructor for this class.  This constructor
    initializes the TimBot and sets its ID and the initial amount of
    energy that it has.

    parameter: id    : ID of the TimBot
               jolts : Initial amount of energy
   */
  public AngryBot( int id, int jolts ) {
    super( id, jolts );
    personality = 'A';
  }


  /**
    This method is called during the Move phase of each round and
    requires the TimBot to decide whether or not to move to another
    District.  If the TimBot wishes to move, it returns  District.NORTH,
```

https://youtu.be/FLrQnUlsjM4

Extracting Expressions with IntelliJ

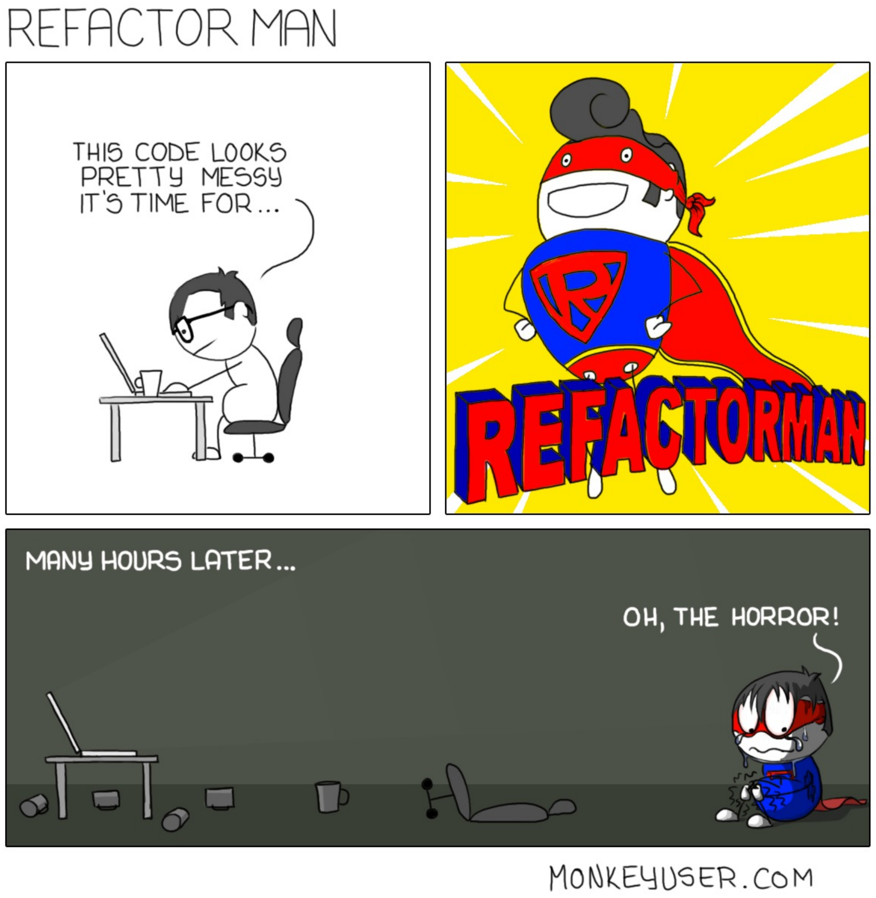


https://youtu.be/z4kZjun_jPo

Refactoring Safely (Work with a Net)

(McConnel, CC2 2004)

- **Save the code**: Be sure the current version is committed (and pushed)
- **Keep refactorings small**: While some refactorings can be big, keep each refactoring to as few code-changes as possible
- **Do refactorings one at a time**: It's much easier to undo one refactoring if you break something, than to undo many
- **Make a list of steps you intend to take**: Some refactorings need to be done in order
- **Keep a back-log:** Make a list of refactorings that you wish to do in the future. Don't start a second refactoring before completing the first
- **Commit often**: Commit your code after each refactoring to allow you to easily back-out of a refactoring



Refactoring Safely (Work with a Net)

(McConnel, CC2 2004)

- **Remember to follow all warnings**: Changing code can lead to errors, so be careful
- **Retest**: Perform **regression tests** after each refactoring is completed
- **Add test cases**: Add test cases if white-box testing is being used.
- **Review the changes**: more intense refactoring often results in more errors
- **Adjust approach depending on size of** refactorings. Larger refactors are more likely to create a defect

Refactoring Strategies

When should one refactor?

- Implementing everything and *then* refactoring everything is a bad idea that will require too much effort!

Perform incremental refactoring alongside implementation.

- Refactor when you add a method or a class
- Refactor when you fix a defect
- Target error-prone or high complexity modules
- During maintenance, improve the parts you touch

Key Points

- Software changes as it is developed
- Software can either evolve (improve) or devolve (degrade)
- Software can be improved through occasional refactoring
- Refactoring involves making changes to improve quality without changing functionality
- Refactoring can be done at the data, statement, method, and class levels
- Refactoring should be done safely by ensuring any changes can be reverted if necessary

Image References

Retrieved January 29, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- https://thumbs.dreamstime.com/t/human-evolution-sheep-30702287.jpg
- https://media.istockphoto.com/vectors/evolution-of-the-texting-human-vector-id529774419?k=6&m=529774419&s=612x612&w=0&h=0Vs3LtD_2Tc7ESx6erKrX9s14HNWMtT4TDnPOkl3SzQ=
- https://images-na.ssl-images-amazon.com/images/I/51q89Uuf7ML._SX258_BO1,204,203,200_.jpg

Retrieved October 30, 2020

- https://bonkersworld.net/building-software