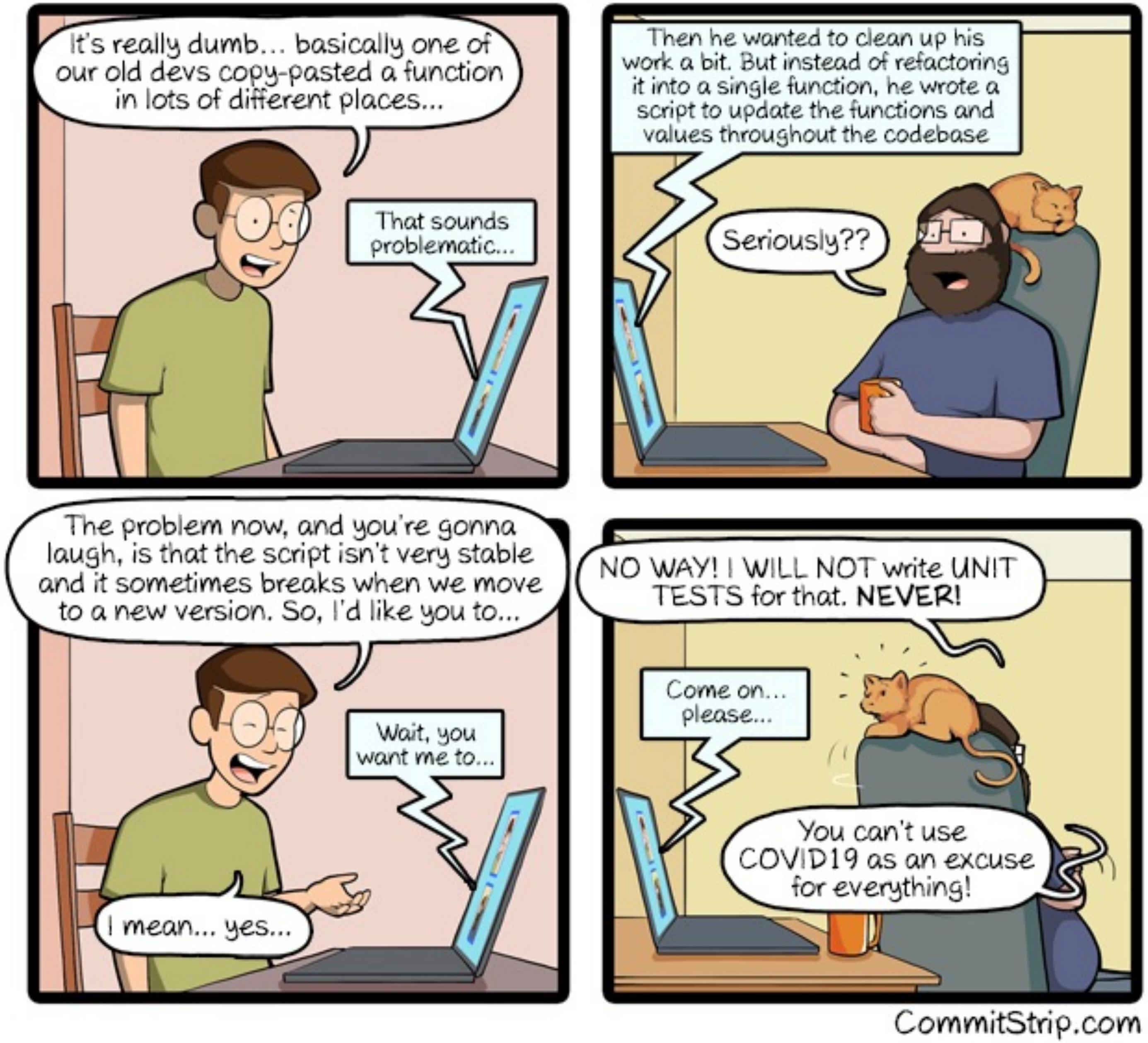

It's really dumb… basically one of our old devs copy-pasted a function in lots of different places...
That sounds problematic...
Then he wanted to clean up his work a bit. But instead of refactoring it into a single function, he wrote a script to update the functions and values throughout the codebase
Seriously??
The problem now, and you're gonna laugh, is that the script isn't very stable and it sometimes breaks when we move to a new version. So, I'd like you to...
Wait, you want me to...
I mean... yes...
NO WAY! I WILL NOT write UNIT TESTS for that. NEVER!
Come on... please...
You can't use COVID19 as an excuse for everything!
CommitStrip.com

Class-Level Refactoring

CSCI 2134: Software Development

Agenda

- Lecture Contents
  - Class-Level Refactoring
  - Class Implementation Refactoring
  - Class Interface Refactoring
- Brightspace Quiz

Readings:

- This Lecture: Chapter 5
- Next Lecture: Chapter 5

Student Learning Experience Questionnaires (SLEQs)

✓ Course and program (re) design.

✓ Evaluation of teaching effectiveness.

✓ Promotion, tenure, awards, and grants for instructors and teaching staff

✓ **Quality assurance** processes in the review and restructure of institutional, faculty, department and program goals.

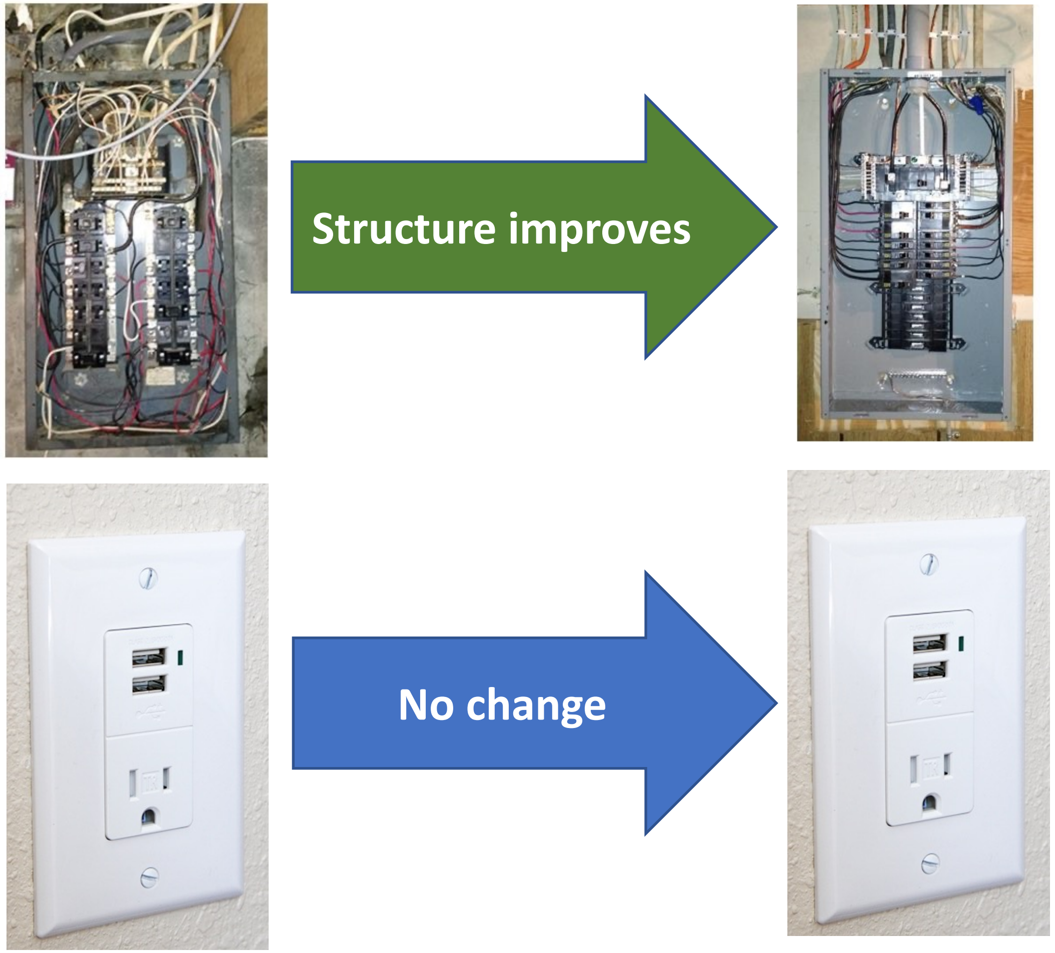
Complete SLEQs by:

- Checking your emails and following the link!
- Logging in to Brightspace and choosing “SLEQ” from the home page.

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

Types of Refactoring

- Data-Level refactoring

  Improve use of variables and data
- Statement-level refactoring

  Improve use of individual statements
- Routine-level refactoring

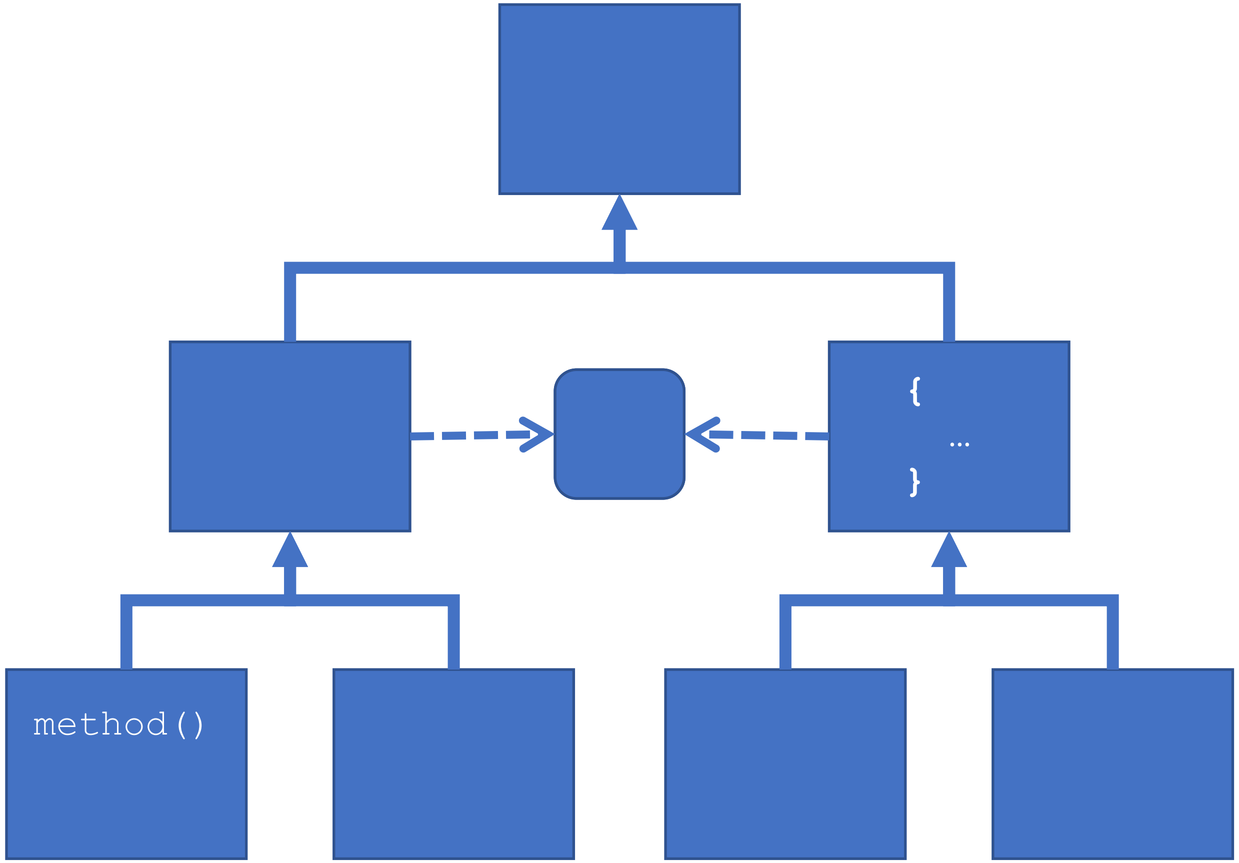
  Improve code at the routine/method level

- Class-implementation refactoring
  - Improve the code in the class
- Class-interface refactoring
  - Improve the class's interface (application of SOLID)
- System-level refactoring

Class Implementation Refactoring

Four general types of refactoring

- Replace virtual routines (method overrides) with data initialization
- Change member routine or data placement in the class hierarchy
- Change location of code in the class hierarchy
- Change code to use references or values (shallow vs deep copy)



Replace Method Overrides with Data Initialization

```
public abstract class Danger {
  …
  public abstract int doDamage();
}

public class TarPit extends Danger {
  …
  public int doDamage() {
    return TARPIT_DAMAGE;
  }
}

public class CoffeeAddict extends Danger {
  …
  public int doDamage() {
    return COFFEE_ADDICT_DAMAGE;
  }
}
```

Overriding methods

```
public abstract class Danger {
  private int damage;
  …
  public int doDamage() {
    return damage;
  }
}

public class TarPit extends Danger {
  public TarPit() {
    setDamage(TARPIT_DAMAGE);
  }
  …
}

public class CoffeeAddict extends Danger {
  public CoffeeAddict() {
    setDamage(COFFEE_ADDICT_DAMAGE);
  }
  …
}
```

data

Data initialization

Refactoring Action: Change Member Routine or Data Placement

These changes are normally performed to eliminate duplication in derived classes:

- Pull a routine up into its superclass.
- Pull a field up into its superclass.
- Pull a constructor body up into its superclass.

Several other changes are normally made to support specialization in derived classes:

- Push a routine down into its derived classes.
- Push a field down into its derived classes.
- Push a constructor body down into its derived classes.

Pull a method and variable into the superclass

```
public abstract class Shape {
  …
}

public class Triangle extends Shape {
  private Color color;
  …
  public void changeColor(Color color) {
    …
  }
}

public class Circle extends Shape {
  private Color color;
  …
  public void changeColor(Color color) {
    …
  }
}
```

```
public abstract class Shape {
 private Color color;
  …
  public void changeColor(Color color) {
   …
  }
}

public class Triangle extends Shape {
  …
}

public class Circle extends Shape {
  …
}
```

Silly Question

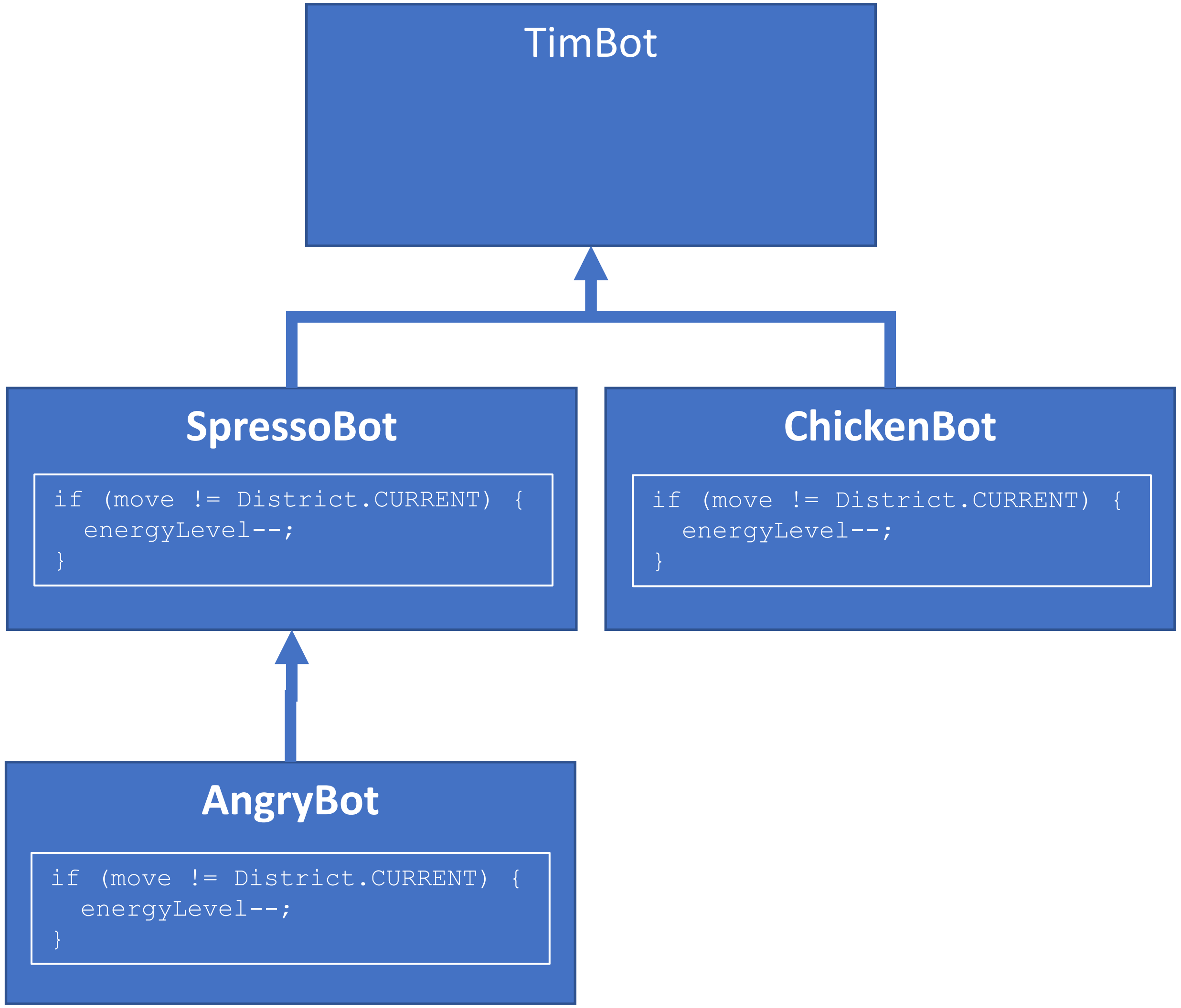
- Question: Why did we not put “Color” into shape from the start?

- Answer:
  - This example is very simple
  - Code **evolves and devolves:**
    - Initially only triangles may have had a color
    - Circles may have had color added later
  - Incremental additions may not foresee generalities or specialization of classes
  - Refactoring fixes these oversights and improves the code

Refactoring Action: Change Location of Code

Actions

- Combine similar code into a superclass
- Extract specialized code into a subclass



TimBot

SpressoBot

```
if (move != District.CURRENT) {
  energyLevel--;
}
```

ChickenBot

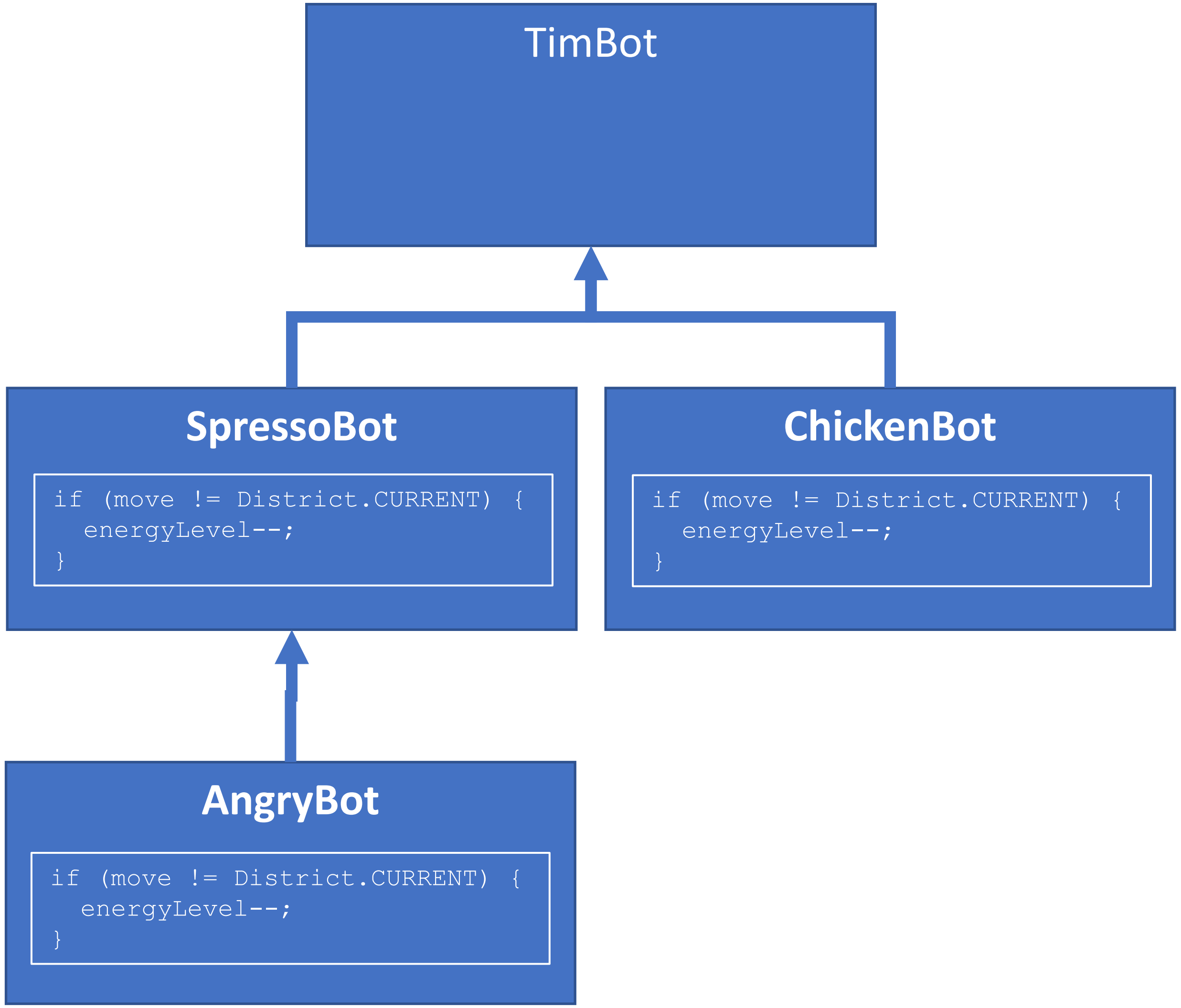
```
if (move != District.CURRENT) {
  energyLevel--;
}
```

AngryBot

```
if (move != District.CURRENT) {
  energyLevel--;
}
```

Example: Combine similar code in the superclass

```
public class TimBot {
   ...
}

public class SpressoBot extends TimBot {
   ...

   public int getNextMove() {
      ...
      // If move is not to stay here,
      // decrement energy level.
      if (move != District.CURRENT) {
        energyLevel--;
      }
      ...
      return move;
   }
}
```



```
if (move != District.CURRENT) {
  energyLevel--;
}
```

```
if (move != District.CURRENT) {
  energyLevel--;
}
```

```
if (move != District.CURRENT) {
  energyLevel--;
}
```

Example: Combine similar code in the superclass

```
public class TimBot {
  ...
}

public class SpressoBot extends TimBot {
  ...
  public int getNextMove() {
    ...
    // If move is not to stay here,
    // decrement energy level.
    if (move != District.CURRENT) {
      energyLevel--;
    }
    ...
    return move;
  }
}
```

```
public class TimBot {
  ...
  protected useMoveEnergy(int move) {
    if (move != District.CURRENT) {
      energyLevel--;
    }
  }
  ...
}

public class SpressoBot extends TimBot {
  public int getNextMove() {
    ...
    useMoveEnergy(move);
    ...
    return move;
  }
}
```

Other Refactoring Action
Change code to use references or values

Action

- Change value objects to reference objects
- Change reference objects to value objects

- In Java this is typically a decision to go from shallow to deep copying or reverse
  - Java does not have value objects

- Use
  - **Deep copy** If copies of objects are going to be modified differently
  - **Shallow copy** if all copies of the object will be modified in the same way

Class Interface Refactoring

• Move a routine/method to another class • Convert one class to two • Eliminate a class • Collapse a superclass and subclass if their implementations are very similar	Single Responsibility Principle
• Introduce an extension class	Open/Close Principle
• Replace inheritance with aggregation/delegation • Replace aggregation/delegation with inheritance	Liskov Substitution Principle
• Remove setters for fields that cannot be changed • Encapsulate unused routines /methods	Interface Segregation Principle
• Hide routines that are not intended to be used outside the class • Encapsulate an exposed member variable	Dependency Inversion Principle

Refactoring Actions to Promote the Single Responsibility Principle

- Move a method to another class
  - If a method does not support the class' responsibility it should moved
- Convert one class to two
  - If a class has multiple responsibilities (as we saw in the example for SRP) either
    - Split the class into two distinct classes
    - Split the class into a subclass and superclass
- Eliminate a class
  - If a class has no responsibility or the responsibility is not needed
- Collapse a superclass and subclass if their implementations are very similar
  - If two classes have the same responsibility, one of them is not needed

Refactoring Actions to Promote the Open/Close Principle

- Introduce an extension class
    - If a class needs additional functionality, extend it instead of modifying it.

- Classes that look like they may need to be modified in the future because they are too specific should be fixed
    - Use polymorphism instead of conditional statements
    - Use the most general supertype possible in the interface, e.g., *Shape* instead of *Rectangle*
    - We saw examples of this when we discussed OCP.

Example: Refactoring for OCP

Extend instead of modify

Commit 1:

```
public class Shape {
  private int xPos;
  private int yPos;
}
```

Commit 2:

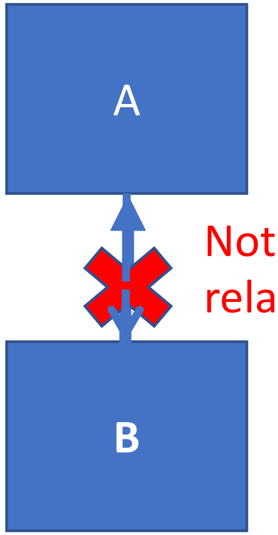
```
public class Shape {
  private int xPos;
  private int yPos;
  private Color color;
}
```

Refactor:

```
public class Shape {
  private int xPos;
  private int yPos;
}

public class ColoredShape extends Shape {
  private Color color;
}
```

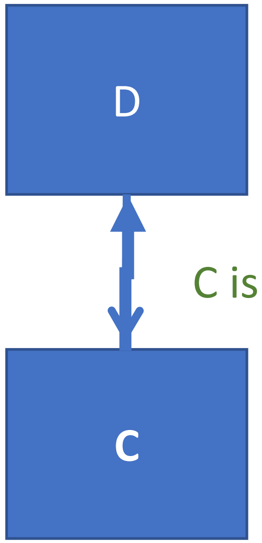
Refactoring Actions to Promote the **Liskov Substitution Principle**

- Replace inheritance with **delegation** (dependency/aggregation)
  - Inheritance should only be used if
    - There is a true **“is a”** relationship between the subclass and superclass
    - The subclass can be used in all instances that call for the superclass
  - If this is not the case, then inheritance should not be used
- Replace delegation with inheritance if
  - The public interface of the “used” class (C) is the same as the “using” class (D)
  - The “is a” relationship holds (C is a D)



B delegates to A

Not “Is a” relationship



C is a D

Example: Refactoring for LSP

Replace inheritance with delegation

```
public class Vector {
  public void insert(int index, int val);
  public int remove(int index);
}

public class Stack extends Vector {
  public void prepend(int val) {
    super.insert(0, val);
  }
  public int removeFrist() {
    return super.remove(0);
  }
}
```

```
public class Stack {
  private Vector vals;

  public void prepend(int val) {
    vals.insert(0, val)
  }

  public int removeFirst() {
    return vals.remove(0);
  }
}
```

Refactoring Actions to Promote the Interface Segregation Principle

- Remove setters for fields that cannot be changed
  - Make the interface as narrow as possible
  - If a field should only be changed by the class’ methods, do not provide setters
  - If subclasses of the class may need to set the field use protected instead of public!
- Encapsulate unused methods
  - Unless those methods are provided specifically for utility or extensibility by the user.

Example: Refactoring for ISP

Provide a minimal interface; Do not let setters make an invalid object

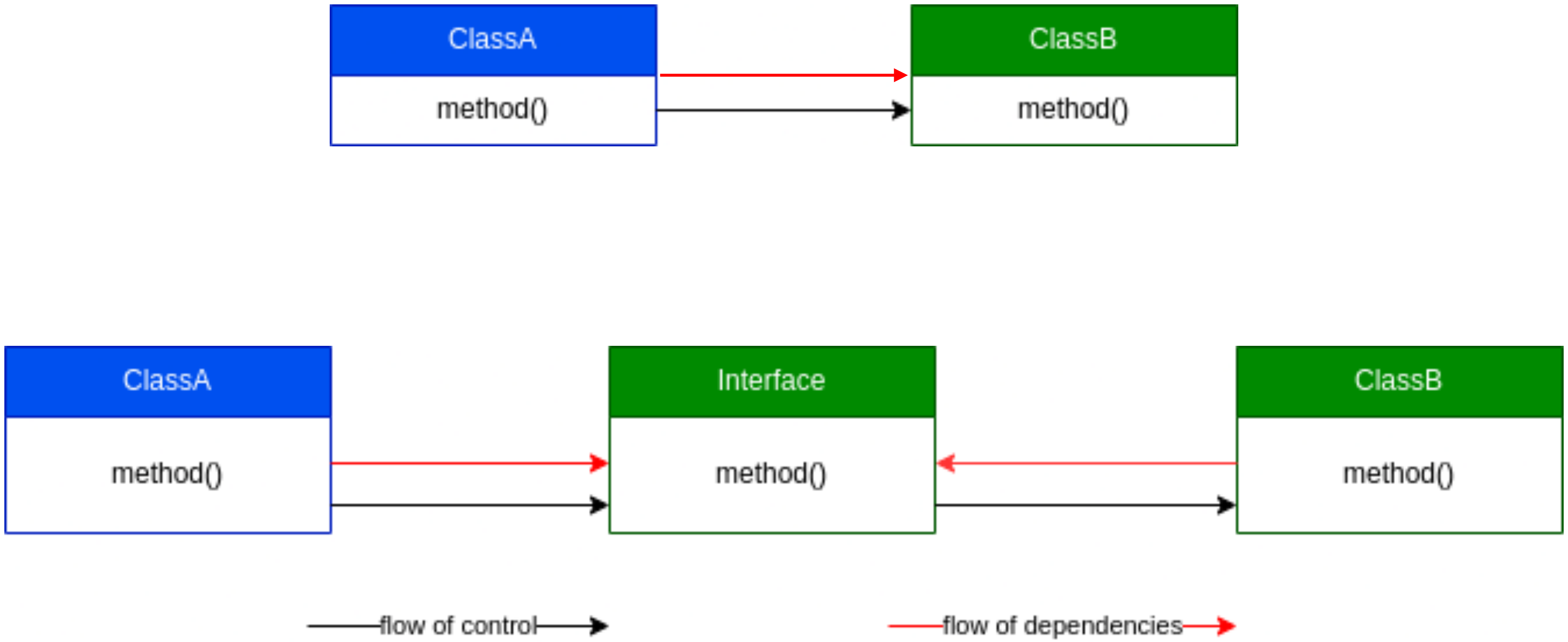
```
public class Triangle {
  private float side1;
  private float side2;
  private float side3;

  public void setSideLengths(float val1, float val2, float val3);


  public void setFirstSideLength(float val) {
    side1 = val;
  }
  public void setSecondSideLength(float val) {...}
  public void setThirdSideLength(float val) {...}
}
```

Refactoring Actions to Promote the Dependency Inversion Principle

- Hide routines that are not intended to be used outside the class
- Encapsulate all exposed member variables
- Any exposed methods and variables provide more information about a class's implementation **and lazy users will take advantage of this!**
  - Exposed methods/variables create opportunities for coupling.
  - **Only make public what is absolutely necessary necessary!!**

- **Prototypical refactoring:**
  - Replace dependency on concrete class with dependency on interface or abstract class

Refactoring for DIP: Rely on abstractions, not concreations



Polymorphism enables dependency inversion!

Example: Refactoring for DIP

```
public class Player {
  public int health;
  public void gameOver();
}

public class Enemy {
  private int attackPower;

  public void attack(Player p) {
    p.health -= attackPower;
    if (p.health <= 0) {
      p.gameOver();
    }
  }
}
```

```
public class Player {
  private int health;
  private void gameOver();

  public void takeDamage(int val) {
    health -= val;
    if (health <= 0) {
      this.gameOver();
    }
  }
}

public class Enemy {
  private int attackPower;

  public void attack(Player p) {
    p.takeDamage(attackPower)
  }
}
```

Key Points

- Refactoring classes improves both implementation and interfaces

- Class implementation refactoring involves determining the best location of functionality in the hierarchy
  - changes implementation details, not class hierarchy structure.

- Class interface refactoring involves determining which SOLID principles are violated and fixing them
  - changes class design class and structure.

Image References

Retrieved January 29, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- Image from StackOverflow, attributing it to https://www.coursera.org/lecture/object-oriented-design/1-3-1-coupling-and-cohesion-q8wGt
- https://library.kissclipart.com/20181002/cae/kissclipart-printer-icon-clipart-printer-computer-icons-clip-a-4355e69e2147b9a3.png
- http://clipart-library.com/printer-cliparts.html
- https://lh3.googleusercontent.com/proxy/Bln9JbfTJUZLqFBN6I5cBSMl6ww_z31qieplSeIlFzI3y7tMvFUIyPtWt-2HGgx3DGSYRC6lD-PmfiFz7Qc1XodvqTTmArCdcg
- https://lh3.googleusercontent.com/proxy/EjmkwGKjVndigDIQa4BpI6eQHDucnKJJ47EVJCLlHgP0c3SX16aHum4kGVL0Q6uEQrc1gaGh6jD7lpPMYm2GQSoAT3L_RQfQ6_nw
- https://cdn3.vectorstock.com/i/1000x1000/56/72/car-rental-car-for-rent-word-on-red-ribbon-vector-26835672.jpg

Retrieved November 27, 2020

- https://www.commitstrip.com/en/2020/04/16/a-story-of-duplicate-code/?

Retrieved November 23, 2023

- https://www.baeldung.com/cs/dip