CS 5004

LECTURE 8

ART OF DESIGN

SOLID, DESIGN PRACTICES

KEITH BAGLEY

SPRING 2022

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

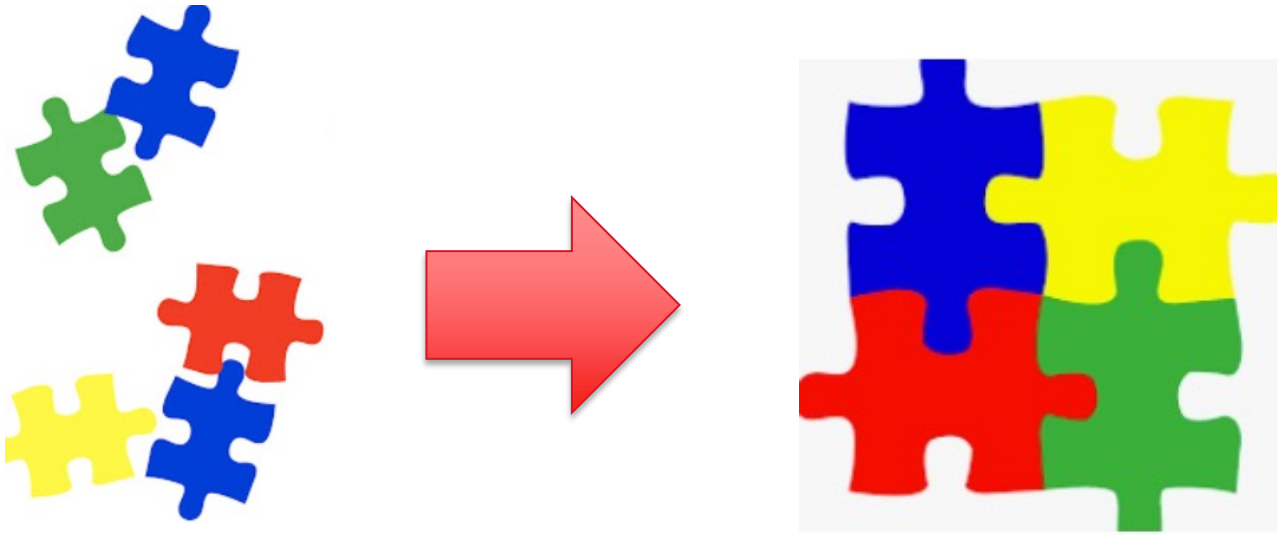
AGENDA

- Midterm Retrospective
- Pivot to Design
- Managing Complexity on a Smaller Scale
  - Code-jutsu: Double Dispatch
  - Code-jutsu: Using Iterators for Collections
  - Code-jutsu: Writing “good” tests
  - Code-jutsu: Private constructors & Factory Patterns
- SOLID Design Practices
- Q & A

LET'S LOOK AT THE MIDTERM

PIVOT TO DESIGN

- For the first half of the course, you've learned techniques and approaches
- The second half of the course pivots toward applying those techniques and principles for good design and implementation

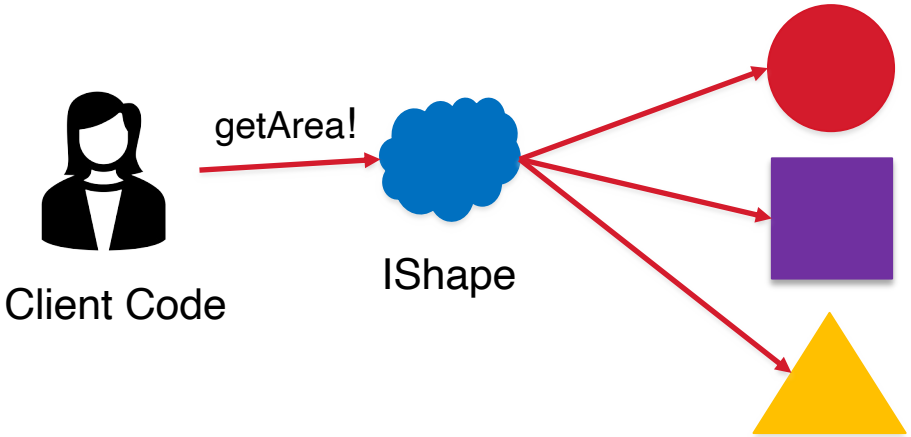


CODE-JUTSU

- We've covered the foundation of object-oriented programming, and *some* big-picture design
- Now, we'll cover a few tactical solutions to situations you might encounter

CODE-JUTSU: DOUBLE DISPATCH VS. SINGLE DISPATCH

- We understand how dynamic dispatch (runtime polymorphism) works
  - Messages sent to the runtime object invoke the correct method regardless of the compile-time supertype reference we use to point to the object
  - Code below is single-dispatch polymorphism




```
public static void main(String [] args ) {
    List<IShape> shapes = new ArrayList<>();
    shapes.add(new Circle( radius: 2));
    shapes.add(new Square( length: 2));
    shapes.add(new Triangle( height: 2,  base: 3));
    shapes.forEach(each -> System.out.println(each.getArea()));
}
```

DOUBLE DISPATCH VS. SINGLE DISPATCH

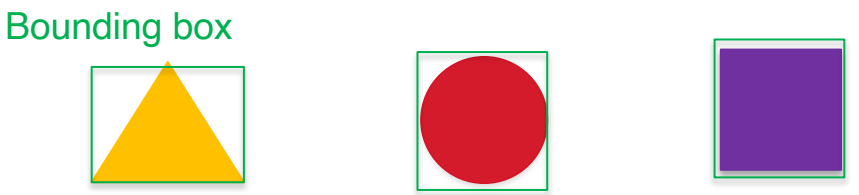
- The code works as expected because:
  - The compile-time check verifies that the variable we use (shapes) adheres to a protocol that allows the getArea() method to be called
    - Or, “the objects can respond to the getArea() message”
  - The runtime dynamic binding (runtime polymorphism) finds the correct method, based on the actual object type, not the compile type of the reference we use to access the object

```
public static void main(String [] args ) {
  List<IShape> shapes = new ArrayList<>();
  shapes.add(new Circle( radius: 2));
  shapes.add(new Square( length: 2));
  shapes.add(new Triangle( height: 2,  base: 3));
  shapes.forEach(each -> System.out.println(each.getArea()));
}
```

But what if we need to make the runtime decision based on two (or more) objects?

DD – TWO OBJECTS DETERMINE BEHAVIOR

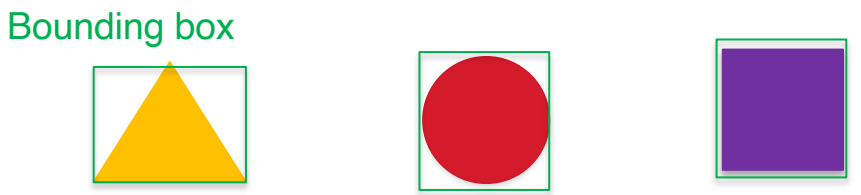
- Consider this situation:
  - We have the Shapes we have worked with this semester
  - We create a few Canvas classes (one scrolls, the other does not)
  - We wish to place our shapes on a canvas, and the transparency/fill is determined by the bounding box of the shape AND the type of canvas we're using
  - Canvas' know how to place() shapes




```
public static void singleDispatch() {
  Canvas canvas = new ScrollCanvas();
  List<IShape> shapes = new ArrayList<>()
  shapes.add( new Circle( radius: 2));
  shapes.add(new Square( length: 3));
  for (IShape shape : shapes) {
    canvas.place(shape);
  }
```

DD – TWO OBJECTS DETERMINE BEHAVIOR

- For this example, we'll implement 2 concrete Canvas classes and use println() to show the variation
- Each Canvas places shapes differently
- Each Canvas overloads place() to handle different shapes




```
public class FramedCanvas implements Canvas {
  @Override
  public void place(Circle circle) {
    System.out.println("Framing circle with bounding box of area " + circle.getArea());
  }

  @Override
  public void place(Square square) {
    System.out.println("Framing square with bounding box of area " + square.getArea());
  }
```

Let's write this code and run it to see what happens. Behavior is not quite what we want!

IMPLEMENT SINGLE-DISPATCH CODE

```java
public class FramedCanvas implements Canvas {
  @Override
  public void place(Circle circle) {
    System.out.println("Framing circle with bounding box of area " + circle.getArea());
  }

  @Override
  public void place(Square square) {
    System.out.println("Framing square with bounding box of area " + square.getArea());
  }
```

Etc.

SINGLE DISPATCH DIDN'T QUITE WORK

- Outcome wasn't quite what we wanted, but is well-defined
  - Concrete Canvas instance is target of message send
  - Single dispatch polymorphism identifies runtime object and calls correct method
    - But at compile time that method checks the type that is being used. We're passing a reference to IShape and cannot infer at compile time what the actual object might be. Thus the choices at runtime are: which version of place(IShape) do we want?

```
/Users/keithbagley/Library/Java/JavaVirtualMachines/openjdk-15.0.2/Co
Single Dispatch
SCROLLING unknown shape with bounding box of area 12.566370614359172
SCROLLING unknown shape with bounding box of area 9.0
```

DOUBLE DISPATCH

- From the previous example: the outcome with single dispatch did not match our intentions
- We want a runtime selection for the proper Canvas place() AND a runtime selection of the place() method that matches the current Shape object
- We need double-dispatch to handle polymorphic selection for both items
  - NB: Double Dispatch (2 dispatches) is a specialization of “multi-dispatch” (2 or more).

CODE-JUTSU: DOUBLE DISPATCH

- We'll implement two polymorphic calls.
  - One for the Shape and one for the Canvas
    - In that order. We'll "flip" the call sequence and let one polymorphic call trigger the other to bypass compile-time binding
    - Double Dispatch is used extensively in the Visitor design pattern (we'll come back to that later)

CODE-JUTSU: DOUBLE DISPATCH

- We will add a new method to the Shape interface. Consequence: Violates Open/Closed principle
- We'll call the placedBy() method to have the second dispatch bound properly
- And each Shape subtype will implement placedBy() such that the compile-time binding will force it to that type's concrete implementation

```
public interface IShape {
  double getPerimeter();
  double getArea();
  default void placedBy(Canvas canvas) {
    System.out.println("Not sure how to place myself");
  }
}
```

Feature expansion for Shape Interface

```
22    public static void doubleDispatch() {
23      Canvas canvas = new FramedCanvas();
24      List<IShape> shapes = new ArrayList<>();
25      shapes.add( new Circle( radius: 2));
26      shapes.add(new Square( length: 3));
27      Iterator<IShape> iterator = shapes.iterator();
28
29      while(iterator.hasNext()) {
30        iterator.next().placedBy(canvas);
31      }
32      // canvas.place(new Circle(2));
33    }
```

Client code triggers DD

```
@Override
public void placedBy(Canvas canvas) {
  canvas.place( square: this);
}
```

Square's placedBy()

DOUBLE DISPATCH OUTCOME

- Polymorphic behavior as intended for two objects
  - Receiver of the message (Shape)
  - Argument that is being operated on (Canvas)
- Violates Open/Closed Principle
  - Need to have control over the code if the original version was not designed for double dispatch
  - Can easily add new subtypes once design is stable

CODE-JUTSU: ITERATORS

- Iterators allow for traversing data structures in a uniform way without exposing details on the underlying representation of the data
- Thus far, we've used for-loops, enhanced for-loops and collection forEach to traverse the data structures
  - Nothing wrong with that approach, but essentially an "internal iterator" that requires some knowledge of the representation and does not allow multiple client access at the same time.
- We'll explore the Iterator Design Pattern later; for now we'll focus on the concrete iterators provided by Java collections

CODE-JUTSU: ITERATORS

- Java collections are Iterable. Each collection defines an iterator() method that returns an iterator that allows us to traverse the collection in a type-safe manner
- Three important methods for iterators:
  - next() returns the next element from the collection, if there is one. Throws a NoSuchElement exception if no more elements exist
  - hasNext() returns true if there are more elements to retrieve, false otherwise
  - remove() removes the last element retrieved by the iterator

CODE-JUTSU: ITERATORS

```
public static void showIterator() {
  Collection<String> cities = new ArrayList<>();
  cities.add("New York");
  cities.add("Beijing");
  cities.add("Mumbai");
  cities.add("London");
  cities.add("Nairobi");

  Iterator<String> iterator = cities.iterator();
  Iterator<String> otherIterator = cities.iterator(); // can have > 1

  otherIterator.next();
  otherIterator.next();
  while(iterator.hasNext()) {
    System.out.print(iterator.next().toUpperCase() + " - ");
  }
  System.out.println("\n\nNow show where the other iterator is at: ");
  System.out.println(otherIterator.next());
}
```

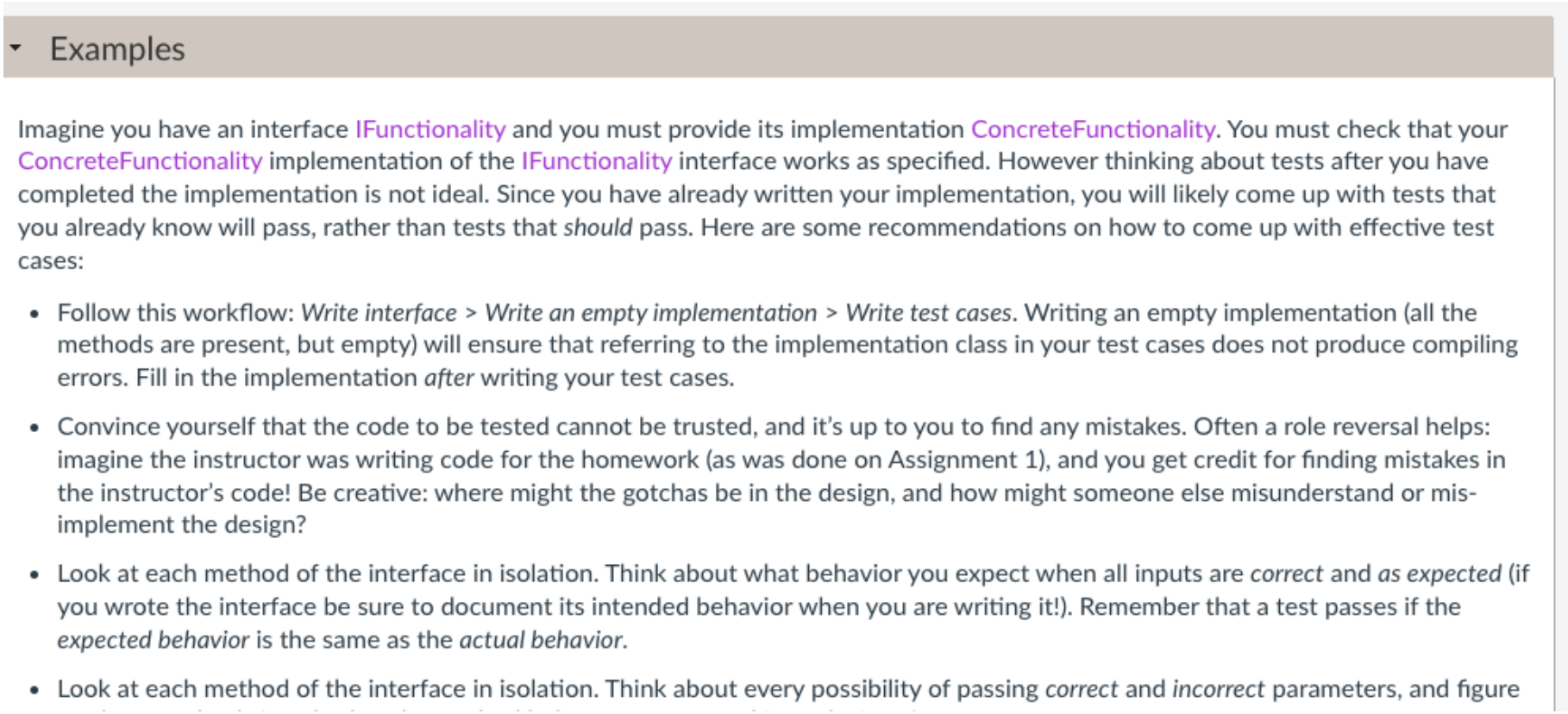
```
NEW YORK - BEIJING - MUMBAI - LONDON - NAIROBI -

Now show where the other iterator is at:
Mumbai
```

CODE-JUTSU: WRITING "GOOD" TESTS

- Writing "good" tests is not easy
  - BTW: We're talking Unit Tests here. The entire Software Quality Process is much larger than what you'll be doing as an individual developer writing code to test your own code
- Tips:
  - Use TDD to reduce bias. Plan and write your tests first
  - Put on your "SQA Hat" and actively try to break your code. Your job is not to prove the code works. Your job is to prove the code has issues. More than the "happy day path".
    - What are the edge cases & boundary conditions?
    - Positive & Negative tests?
    - What is illegal input that "should never" happen (but will)?
  - If the operational space is small enough, can you do naive fuzz tests (random input) or exhaustive tests (e.g. a chessboard has 8 rows & columns – checking bounds by brute force is not out of the question)

CODE-JUTSU: WRITING "GOOD" TESTS

- More tips in Module 0



Examples

Imagine you have an interface IFunctionality and you must provide its implementation ConcreteFunctionality. You must check that your ConcreteFunctionality implementation of the IFunctionality interface works as specified. However thinking about tests after you have completed the implementation is not ideal. Since you have already written your implementation, you will likely come up with tests that you already know will pass, rather than tests that *should* pass. Here are some recommendations on how to come up with effective test cases:

- Follow this workflow: *Write interface* > *Write an empty implementation* > *Write test cases*. Writing an empty implementation (all the methods are present, but empty) will ensure that referring to the implementation class in your test cases does not produce compiling errors. Fill in the implementation *after* writing your test cases.
- Convince yourself that the code to be tested cannot be trusted, and it's up to you to find any mistakes. Often a role reversal helps: imagine the instructor was writing code for the homework (as was done on Assignment 1), and you get credit for finding mistakes in the instructor's code! Be creative: where might the gotchas be in the design, and how might someone else misunderstand or mis-implement the design?
- Look at each method of the interface in isolation. Think about what behavior you expect when all inputs are *correct* and *as expected* (if you wrote the interface be sure to document its intended behavior when you are writing it!). Remember that a test passes if the *expected behavior* is the same as the *actual behavior*.
- Look at each method of the interface in isolation. Think about every possibility of passing *correct* and *incorrect* parameters, and figure

CODE-JUTSU: WRITING "GOOD" TESTS

- Some examples with chess pieces

```
76      /**
77       * Test constructor does not accept columns > 7.
78       */
79      @Test(expected = IllegalArgumentException.class)
80      @GradedTest(name = "Test constructor does not accept columns > 7", max_score = 1)
81      public void testConstructorException1() {
82          new King(0, 8, Color.BLACK);
83      }
84
85      /**
86       * Test constructor does not accept columns < 0.
87       */
88      @Test(expected = IllegalArgumentException.class)
89      @GradedTest(name = "Test constructor does not accept columns < 0", max_score = 1)
90      public void testConstructorException2() {
91          new Queen(0, -1, Color.BLACK);
92      }
```

CODE-JUTSU: WRITING "GOOD" TESTS

- Some examples with chess pieces

```
112     /**
113      * Test Pawn constructor does not accept rows < 1 for White Pawns.
114      */
115     @Test(expected = IllegalArgumentException.class)
116     @GradedTest(name = "Test Pawn constructor does not accept rows < 1 for White Pawns", max_score = 1)
117     public void testPawnConstructorException1() {
118         new Pawn(0, 7, Color.WHITE);
119     }
120
121     /**
122      * Test Pawn constructor does not accept rows > 6 for Black Pawns.
123      */
124     @Test(expected = IllegalArgumentException.class)
125     @GradedTest(name = "Test Pawn constructor does not accept rows > 6 for Black Pawns", max_score = 1)
126     public void testPawnConstructorException2() {
127         new Pawn(7, 7, Color.BLACK);
128     }
```

CODE-JUTSU: WRITING "GOOD" TESTS

- Some examples with chess pieces

```
170     /**
171      * Test bishops can move to correct locations.
172      */
173     @Test
174     @GradedTest(name = "Test bishops can move to correct locations", max_score = 4)
175     public void testBishopCanMove() {
176         assertTrue(this.whiteBishop1.canMove(2, 4));
177         assertTrue(this.whiteBishop1.canMove(5, 7));
178         assertTrue(this.blackBishop1.canMove(4, 2));
179 
180         assertFalse(this.whiteBishop1.canMove(7, 6));
181         assertFalse(this.blackBishop1.canMove(2, 3));
182         assertFalse(this.blackBishop1.canMove(6, 5));
183     }
```

CODE-JUTSU: WRITING "GOOD" TESTS

- Some examples with chess pieces

```
262     /**
263      * Test that staying in the same cell is not considered movement.
264      */
265     @Test
266     @GradedTest(name = "Test that staying in the same cell is not considered movement", max_score = 4)
267     public void testCanMoveIsSameCell() {
268         assertFalse("This is the same location. There is no movement.",
269                 this.blackRook1.canMove(this.blackRook1.getRow(), this.blackRook1.getColumn()));
270         assertFalse("This is the same location. There is no movement.",
271                 this.whiteBishop1.canMove(this.whiteBishop1.getRow(), this.whiteBishop1.getColumn()));
272         assertFalse("This is the same location. There is no movement.",
273                 this.blackQueen.canMove(this.blackQueen.getRow(), this.blackQueen.getColumn()));
```

```
282     /**
283      * Test that chess pieces can and can't kill (except pawns).
284      */
285     @Test
286     @GradedTest(name = "Test that chess pieces can and can't kill (except pawns)", max_score = 3)
287     public void testCanKill() {
288         assertTrue(this.whiteRook1.canKill(this.blackRook1));
289         assertTrue(this.blackQueen.canKill(this.whiteQueen));
290         assertTrue("Testing knight (5,2) taking queen (7,3)", this.whiteKnight1.canKill(this.blackQueen));
291
292         assertFalse(this.whiteRook1.canKill(this.whiteQueen));
293         assertFalse(this.whiteRook1.canKill(this.whiteBishop1));
294         assertFalse(this.blackQueen.canKill(this.blackKing));
295     }
```

CODE-JUTSU: PRIVATE CONSTRUCTORS?

- Why would we ever want to have a private or protected constructor?
  - Perhaps to limit the types of clients with respect to their ability to create certain objects, or **how** they create those objects
  - Enforce a “factory” pattern of creation
  - Enforce a “singleton” pattern

- Not explicitly mandated for HW6 PriorityQueue, it was possible to use that type of private/protected access for creation

- `PriorityQueue createEmpty()`: Creates and returns an empty PQ.

  Special Note: the createEmpty method should be a public class method (static) in your concrete class. It is NOT part of the PQ interface. In a future lecture, you'll learn that we call these "factory methods" but for now ensure you supply a static method in your concrete class that returns an empty PQ.

CODE-JUTSU: SINGLETON DESIGN PATTERN

- In some cases, we want to limit the number of instances of a class to a single object
  - For example, a “service hub” might store all well-known service addresses. Using a hub-and-spoke model, there is never more than one hub
  - We can enforce this constraint with the Singleton design pattern.
  - In most OO languages, including Java, the implementation is straightforward:

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

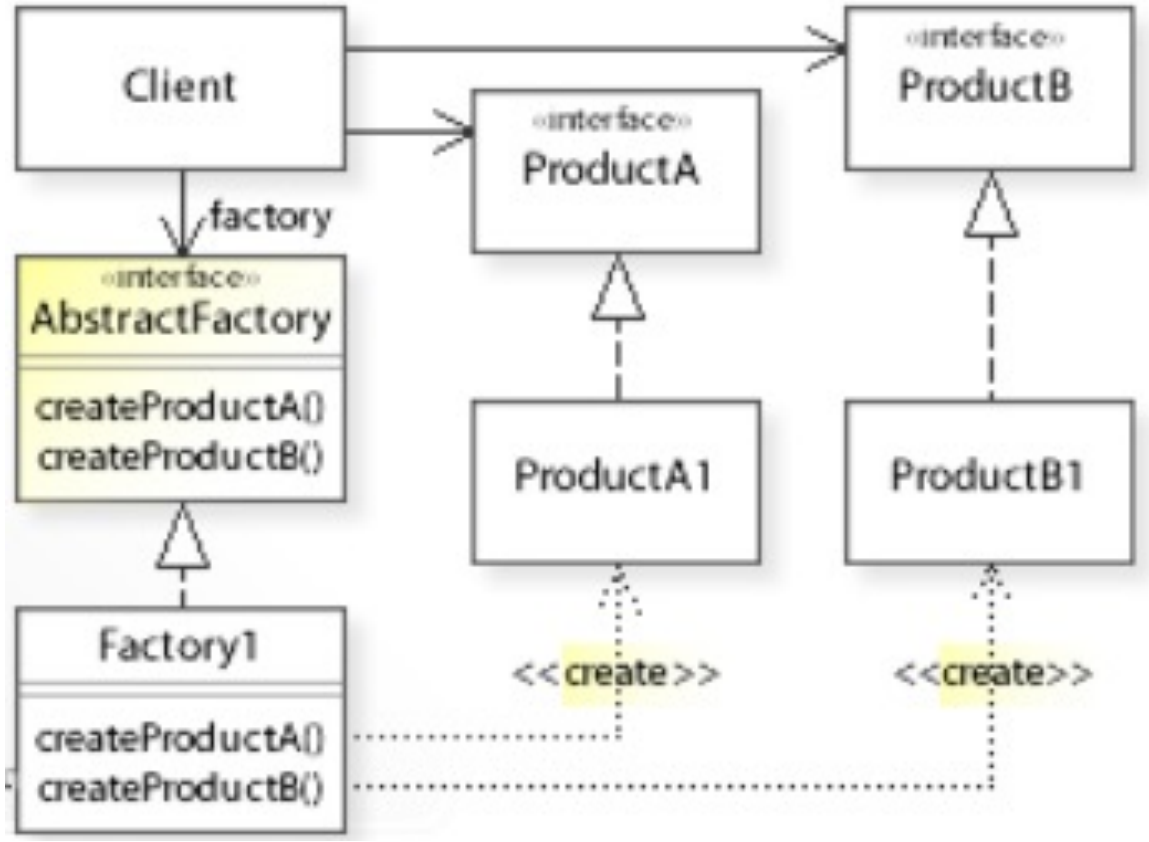
CODE-JUTSU: SINGLETON DESIGN PATTERN

- Making the instance a class attribute ensures only ONE object is created
- Making the constructor private ensures no one outside the class can construct an instance
- Using the getInstance() class method ensures the only instance ever retrieved is the one-and-only one we've instantiated.

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

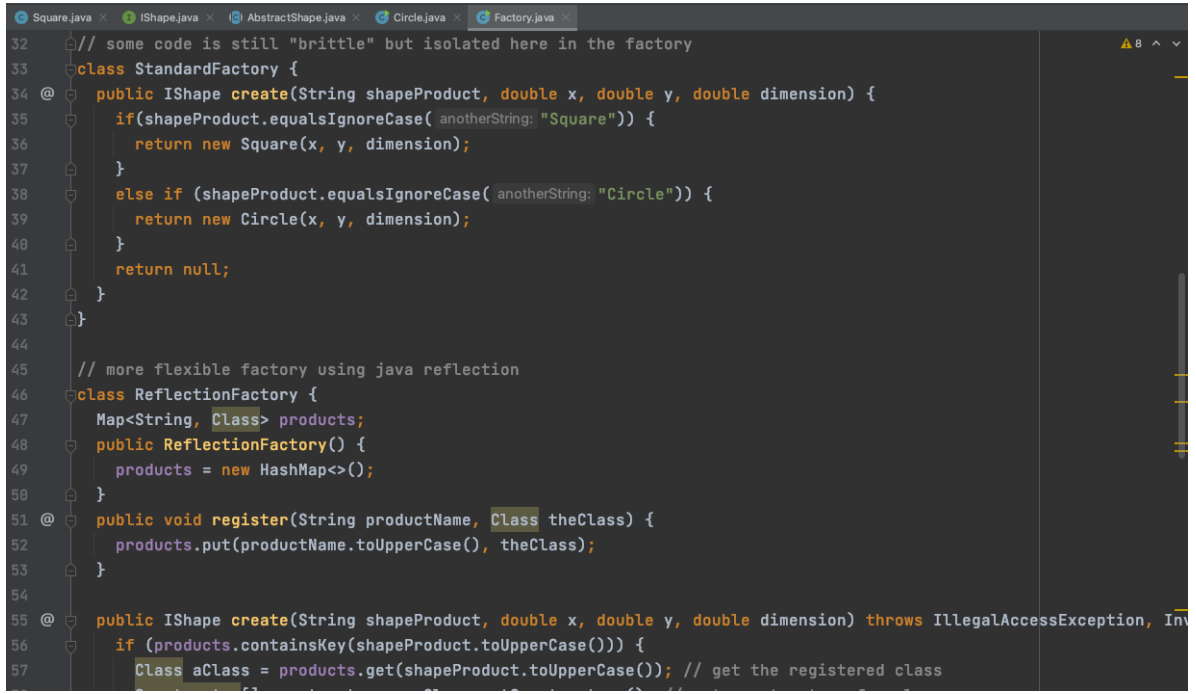
CODE-JUTSU: FACTORY PATTERNS

- Abstract Factory & Factory Method
- Provide an approach for creating families of related or dependent objects without specifying their concrete classes
  - The static createEmpty() for PQ was the beginning pieces of a “factory method”
  - A larger-scale Abstract Factory is an entire class focused on producing objects



ABSTRACT FACTORY

- Different approaches can be taken
  - Simple switch/if statement if “products” are stable an unlikely to change or be updated
  - Java Reflection for “as the system is running” modifications & updates. Also decouples instances from class names encoded in the solution
  - Exemplars, similar to the prototype language SELF



This is a more complex approach – I’ll show you the code, but we’ll write a simpler factory-method approach next

CODE-JUTSU: FACTORY METHODS & PRIVATE CONSTRUCTORS

- Limit creation by using private/protected constructors
  - Also possible: allow access to public factory to only those clients who are allowed creation rights via extra “origination” key or flag as a parameter
- Register factory methods with a “well known” service
  - Create instances on-demand

```
private Square(double length) {
  super( x: 0, y: 0);
  this.length = length;
}
```

```
public static IShape create(double value) { return new Square(value); }
```

```
// create our map of factory methods
Map<String, Function<Double, IShape>> factoryMethods = new HashMap<>();
factoryMethods.put("SQUARE", Square::create);
factoryMethods.put("CIRCLE", Circle::create);
```

SOLID

GOOD/BEST PRACTICES

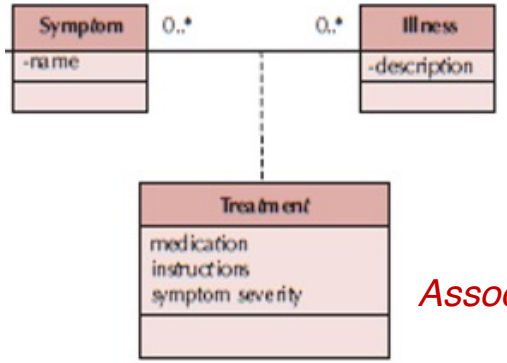
- Most disciplines collect and codify “best practices”
  - Repeatable (good) practices leads to repeatably (hopefully good) results
- CS (and in particular, the sub-discipline of software engineering) has collected sets of principles for good design & development
- Goals – Designs should
  - Tolerate & adapt to change
  - Be (relatively) easy to understand
  - Can be building blocks for many software systems (reuse)

QUALITIES OF WELL-DESIGNED SYSTEMS

- Highly Cohesive
  - Components serve 1 purpose
  - Well defined components
- Low Coupling
  - Coupling measures interdependency. Want to minimize threads of interdependency (spaghetti code) and reduce “ripple effect” when things change
  - Low coupling assists with overall system understandability as well

EXAMPLE: ASSOCIATION CLASSES

- As you've started to create more complex designs, the question arises: "where does some of the behavior **really** belong?"
- Common problem when we have many-to-many relationships
  - One approach is to use "association classes" when attributes about a relationship between two classes needs to be recorded
    - Students are related to Courses; Grade is an association class between them
    - Illness is related to Symptoms; a Treatment class provides an attribute to describe the relationship

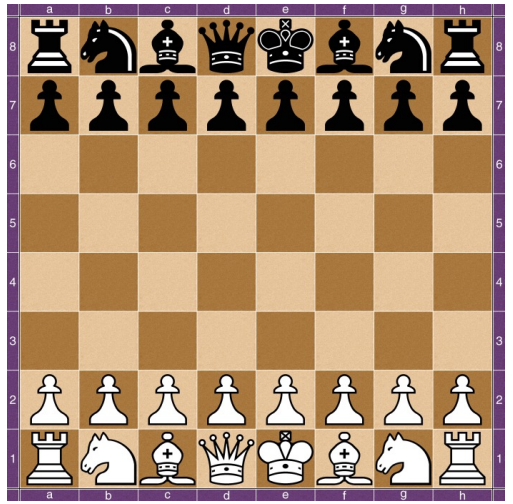


Association Class

Example from Dennis, Tegarden & Wixom

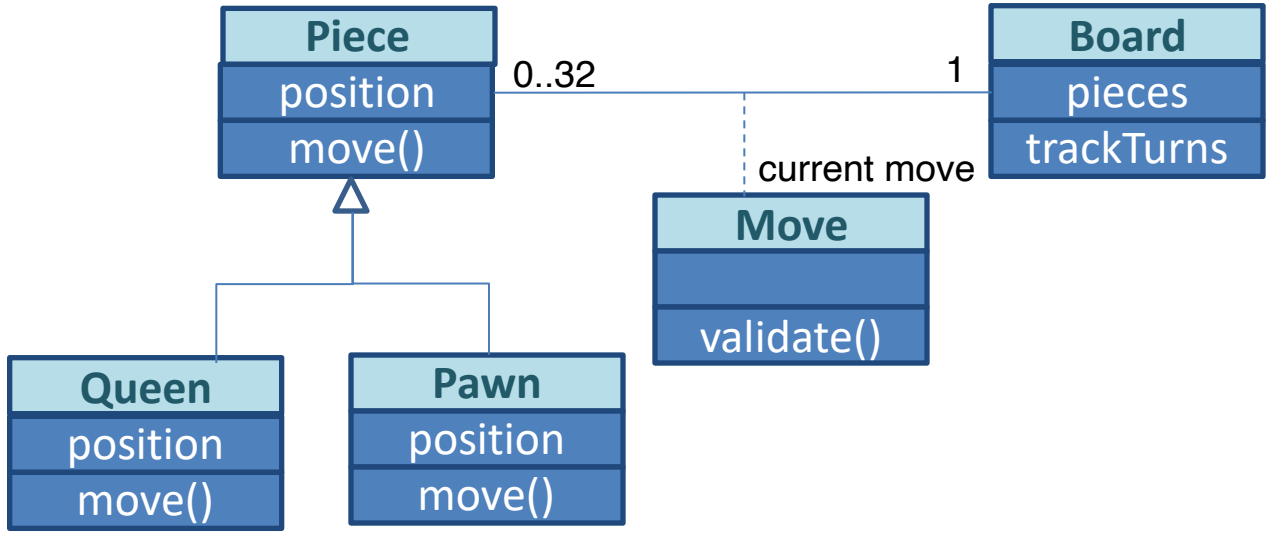
HOW MIGHT WE APPLY ASSOCIATION CLASSES?

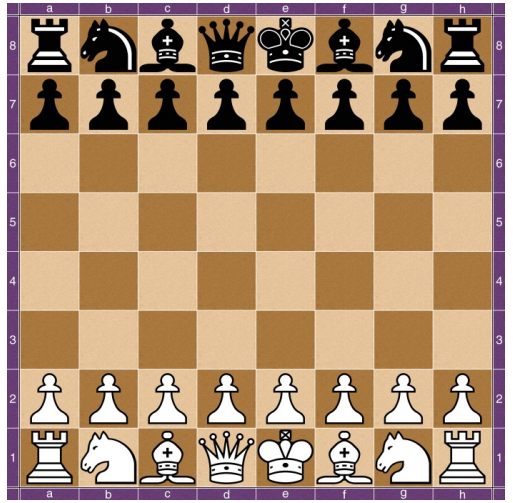
- Consider HW3 – Chess pieces: What if we wanted to build an actual chess game?
- We have handled the mechanics of moving pieces
  - The goal of the homework was to explore code reuse and polymorphism
- But what about the mechanics of the game itself?
  - Board, Moves, Rules (Checkmate, Castle, etc.)?
  - From our homework, pieces know how to move is that sufficient?



ASSOCIATION CLASSES?

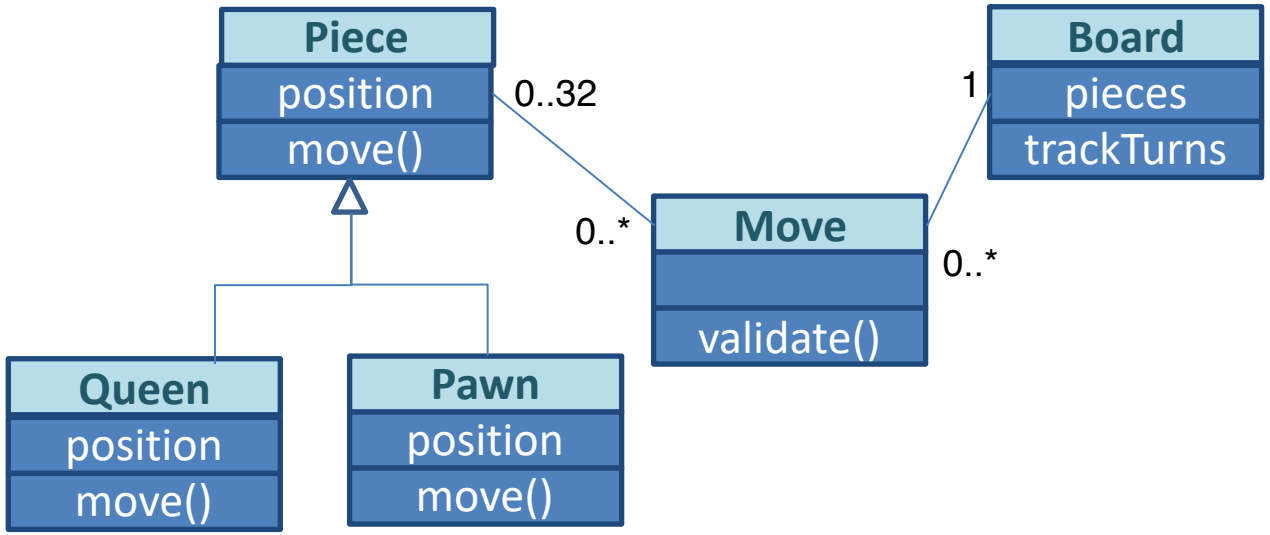
- There are different options. What are possible benefits to an association class?
  - At analysis time, we can envision this for discussion & understanding:

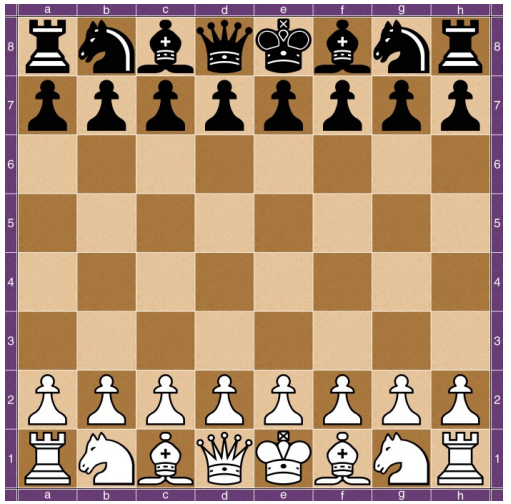




ASSOCIATION CLASSES -> DESIGN

- At design time, we need to convert the previous version
  - Also: the realization that there are likely reasons to track more than current move





DESIGN PRINCIPLES

- The previous example illustrates there is not a “one size fits all” correct answer when *designing* for most problems
- Design is truly about trade-off decisions
  - Some designs are superior, some designs are worse. Some designs are truly *poor/bad*
    - However, it’s difficult to say a design is “wrong”
- Given that, let’s explore some principles that lead us to “better” and “superior” Object Oriented designs that we can implement regardless of programming language

SOLID PRINCIPLES

- **S**RP: Single Responsibility Principle
- **O**CP: Open/Closed Principle
- **L**SP: Liskov Substitution Principle
- **I**SP: Interface Segregation Principle
- **D**IP: Dependency Inversion Principle

SINGLE RESPONSIBILITY

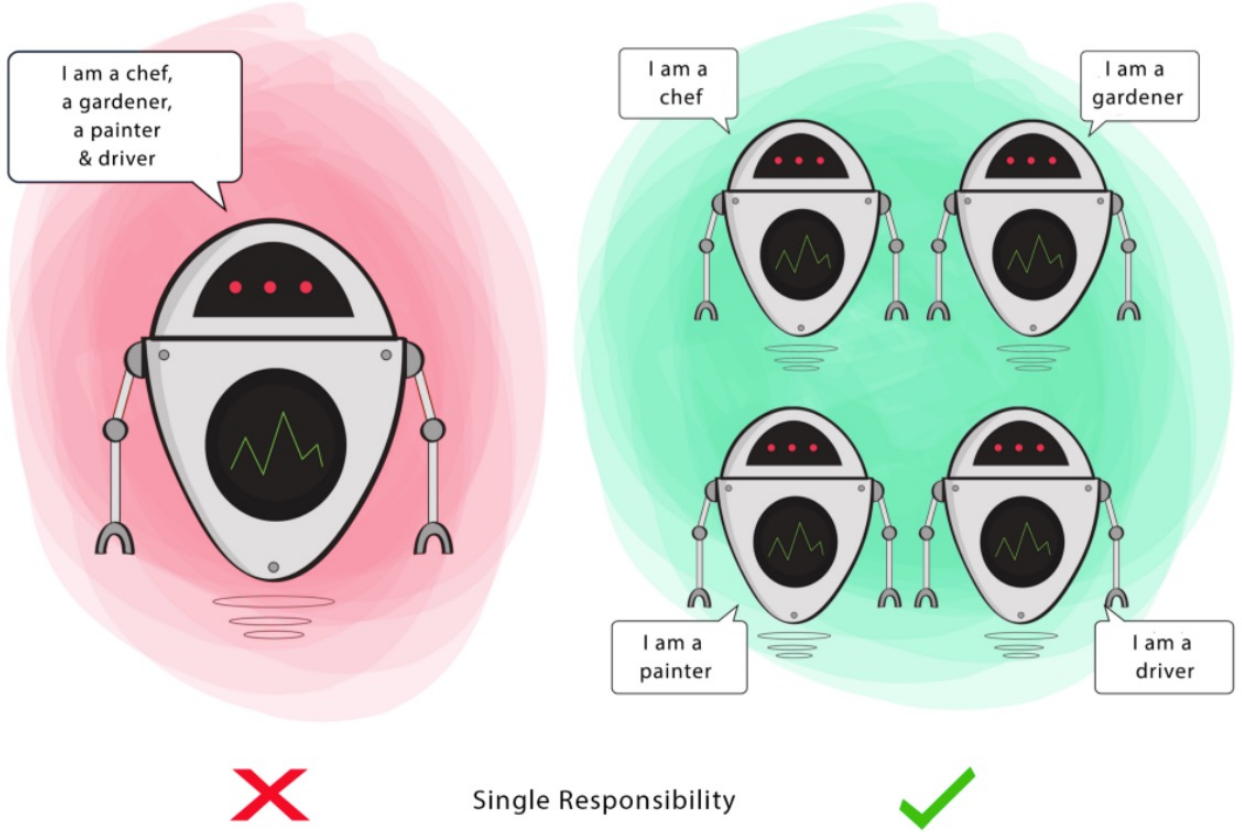
- **S**RP: Single Responsibility Principle
  - Each software component/module/class/function has a single purpose (or single responsibility). If the system requirements change, the component has only 1 reason to change
  - *Anti-principle*: The more responsibilities your component has, the more likely it will need to change, regardless of the requirements. And, the ripple effect of changing the component may negatively impact coverage of other requirements/responsibilities

SINGLE RESPONSIBILITY

- What does “single responsibility” mean?

SINGLE RESPONSIBILITY

- **S**RP: Single Responsibility Principle



Images from https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898
Copyright Ugonna Thelma

OPEN/CLOSED PRINCIPLE

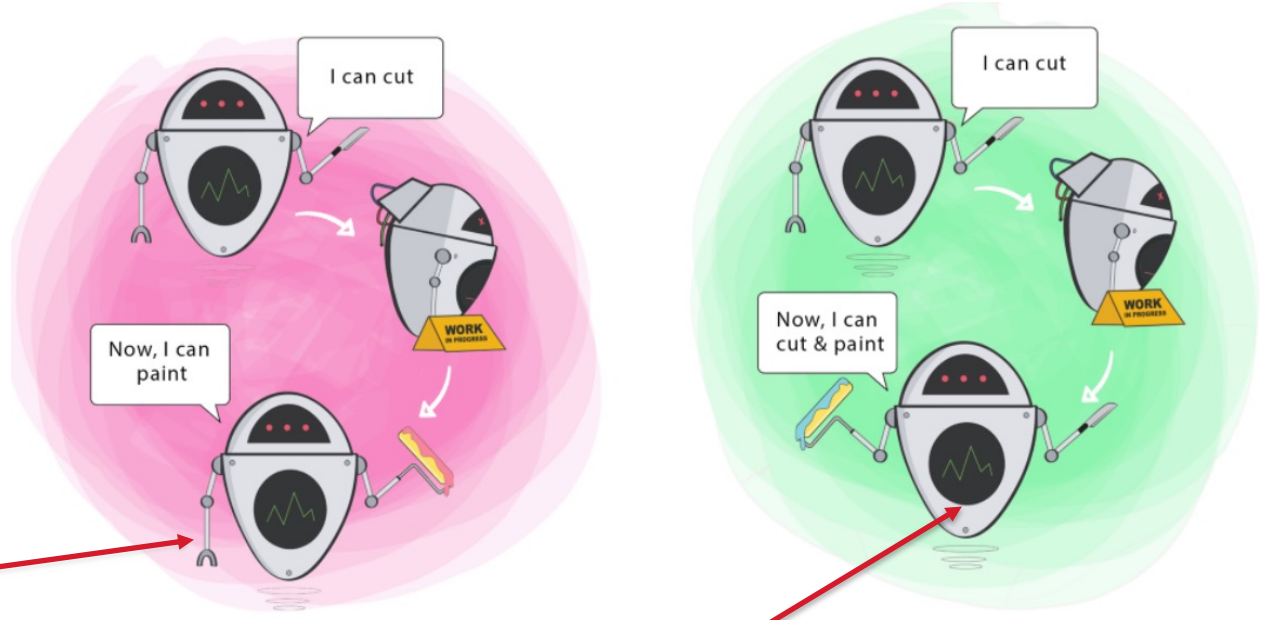
- Open for what?
- Closed when?

OPEN/CLOSED PRINCIPLE

- Open Closed Principle
  - Codified by Bertrand Meyer (creator of the Eiffel Language and Design By Contract)
  - Components should be Open for extension, Closed for modification. In other words, Components must be designed to be changed by adding new code rather than changing existing code.
  - *Anti-principle*: Allowing changes to existing code might “break” other code (customers, co-suppliers, other teams) that depend on the existing code & its current implementation

OPEN/CLOSED PRINCIPLE

- Open to extension
- Closed to modification



Modified component

Derived component
(subtype, subclass, composition, etc.)

Images from https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898
Copyright Ugonna Thelma

LISKOV SUBSTITUTION PRINCIPLE

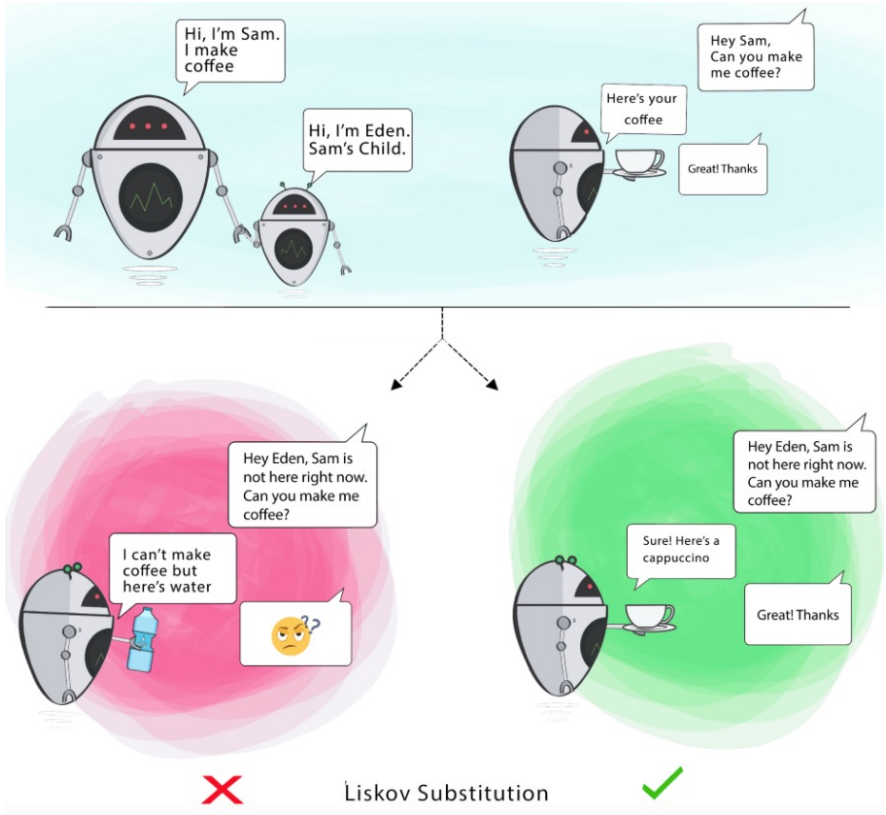
- Liss what?
- Substitute

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Liskov Substitution Principle
  - Codified by Barbara Liskov (MIT Professor) – supertype/subtype theory
  - To build systems from interchangeable parts, those parts must adhere to a contract (protocol) that allows parts to be substituted.
    "If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of expected functionality"
    - *Also allows for backwards compatibility*
  - *Anti-principle*: Create entirely new types that have no supertype-subtype relationship *(NB: not necessarily superclass/subclass)* and then use multiple if/switch statements to create brittle code that checks for the type of the component

LISKOV SUBSTITUTION PRINCIPLE

- Subtypes S can be used wherever supertype T is expected, without breaking code



Images from https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898
Copyright Ugonna Thelma

INTERFACE SEGREGATION PRINCIPLE

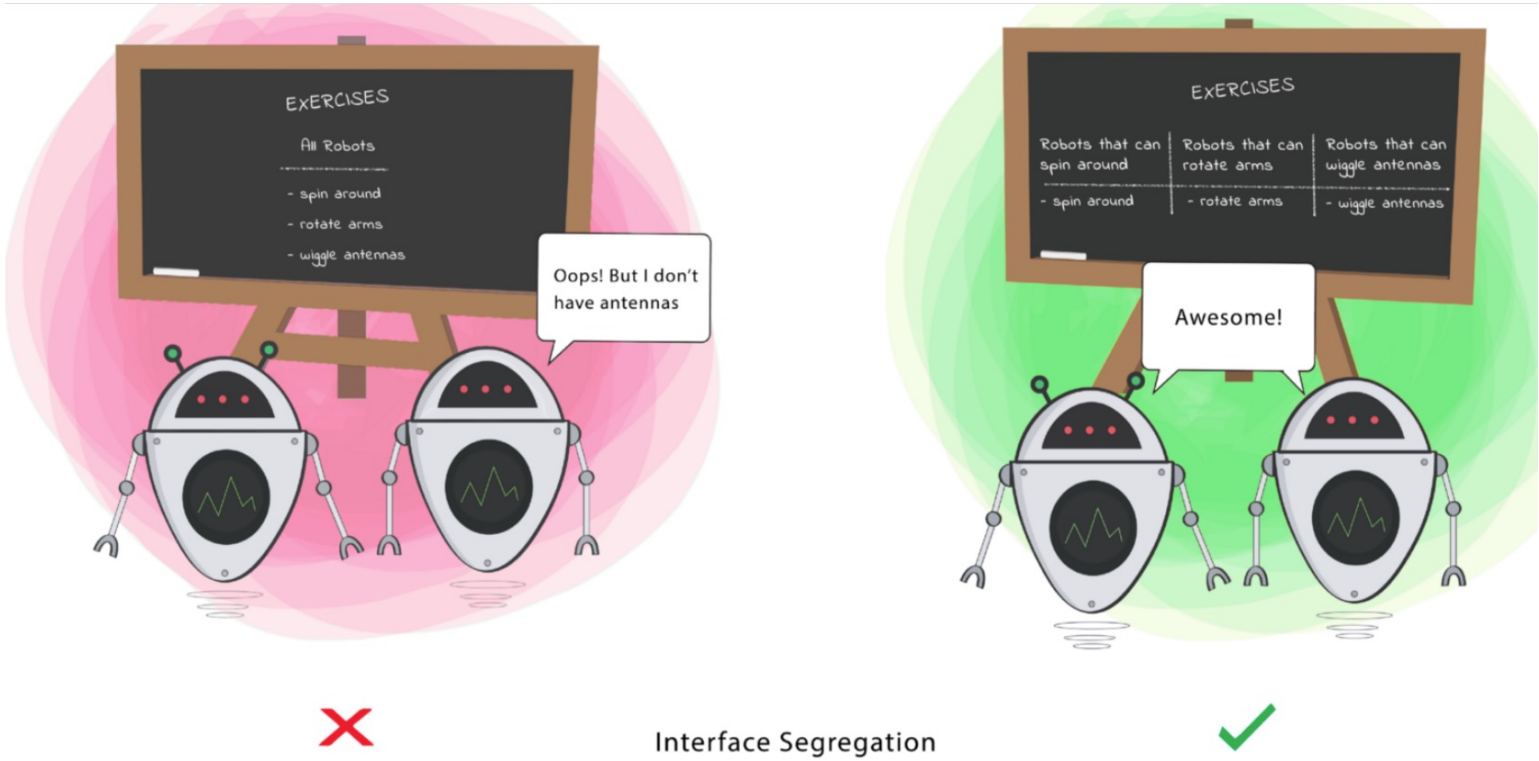
- Segregate -> set apart from the rest; separate
- Interfaces -> protocol
- What?

INTERFACE SEGREGATION PRINCIPLE

- Interface Segregation Principle
  - Designers should avoid depending on things they do not use. No client code should be forced to depend on methods it does not use.
  - If a multitude of clients are using only a subset of the methods in an interface (protocol), this interface should be decomposed and split into two or more smaller interfaces
  - *Anti-principle*: Kitchen-sink interfaces offering more than any client will ever need/want (this sometimes overlaps with single responsibility principle)

INTERFACE SEGREGATION PRINCIPLE

- Clients should not be forced to depend on methods they do not use



Images from https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898
Copyright Ugonna Thelma

DEPENDENCY INVERSION PRINCIPLE

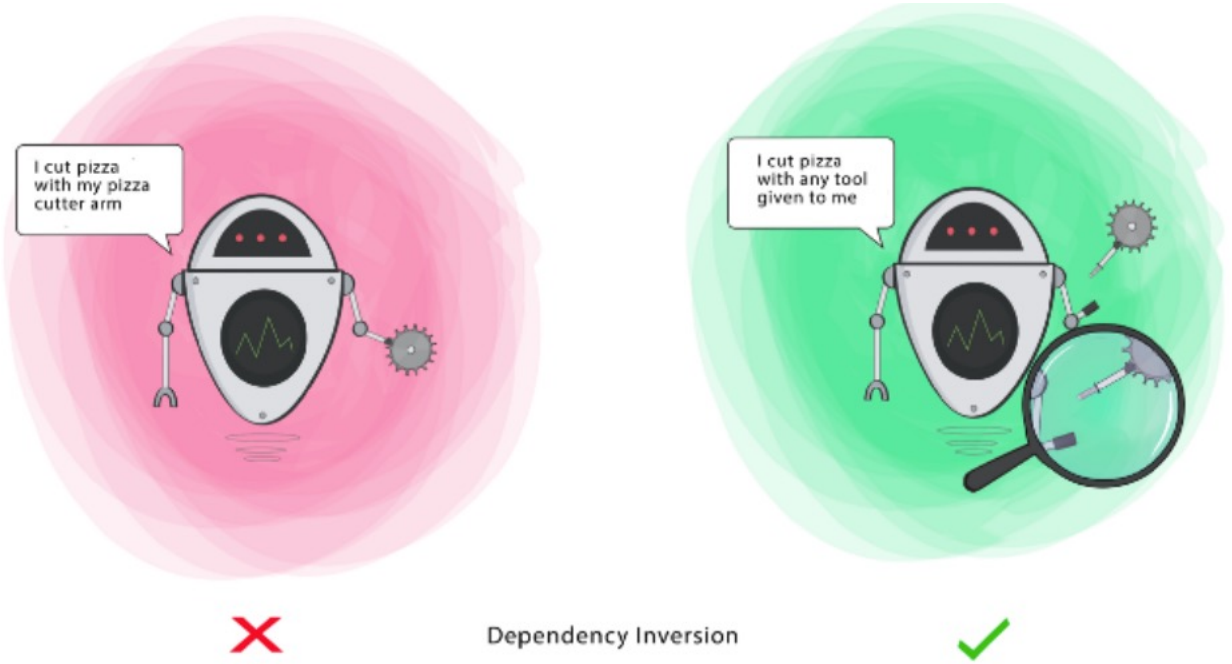
- Dependency – one thing relies on another thing
- MBAs & PMs in the room: Gannt chart task dependencies?
- What is inversion?

DEPENDENCY INVERSION PRINCIPLE

- Dependency Inversion Principle
  - Abstractions should not depend on details. Details should depend on abstractions.
    - Code that implements high-level policy should not depend on the code that implements low-level details. Low level details should depend on the policies
  - *Anti-principle*: Abstractions are dependent on low level details of implementation

DEPENDENCY INVERSION PRINCIPLE

- Abstractions should not depend on details. Details should depend on abstractions



Images from https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898
Copyright Ugonna Thelma

SOLID QUESTION

```java
// examples from https://www.baeldung.com/solid-principles
public class Book {

  private String name;
  private String author;
  private String text;

  //constructor, getters and setters omitted

  // methods that directly relate to the book properties
  public String replaceWordInText(String word){
    return text.replaceAll(word, text);
  }

  public boolean isWordInText(String word){
    return text.contains(word);
  }

  // What principle does this violate if we add the following? What's wrong with this?
  void printTextToConsole(){
    // our code for formatting and printing the text
  }
}
```

Assume we have a Book class as shown. We get a request from management to support a console application. What is the problem with adding the method outlined in red?

Content from:
https://www.baeldung.com/solid-principles

SINGLE RESPONSIBILITY!

```java
// examples from https://www.baeldung.com/solid-principles
public class Book {

  private String name;
  private String author;
  private String text;

  //constructor, getters and setters omitted

  // methods that directly relate to the book properties
  public String replaceWordInText(String word){
    return text.replaceAll(word, text);
  }

  public boolean isWordInText(String word){
    return text.contains(word);
  }

  // What principle does this violate if we add the following? What's wrong with this?
  void printTextToConsole(){
    // our code for formatting and printing the text
  }
}
```

Book is a domain class that manages the concept of what “books” do. Printing to the console goes beyond its “single responsibility”. Remember: classes should focus on doing ONE thing well.

Content from:
https://www.baeldung.com/solid-principles

SOLID QUESTION

```java
public interface Car {

    void turnOnEngine();
    void accelerate();
}

public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}

public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

Assume we have we define a simple Car interface with a couple of methods that all cars should be able to fulfill (the contract)
– turning on the engine, and accelerating forward

We like Tesla & their electric vehicles. What is the problem, and which principle is violated?

Content from:
https://www.baeldung.com/solid-principles

LISKOV SUBSTITUTION PRINCIPLE!

```java
public interface Car {

    void turnOnEngine();
    void accelerate();
}

public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}

public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

Basically, if ElectricCar is a subtype of Car (and it is) we should be able to replace ElectricCar wherever a Car is expected and not “break” things or disrupt the behavior of the system.

Obviously, that’s not true here since we’re throwing an exception when we call an advertised public method. We’re also not meeting the contract for the protocol.

Content from:
https://www.baeldung.com/solid-principles

SOLID QUESTION

```java
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}


public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

Before starting the Align program, you had a Summer job as a zookeeper in a Bear enclosure.
As avid zookeepers, you were more than happy to wash and feed our beloved bears. However, you were all too aware of the dangers of petting them. The first interface was large and had you doing things you really didn't want to do (and weren't qualified for).

Do you see the problem?

What principle did you follow to refactor the code for the second set of interfaces?

Content from:
https://www.baeldung.com/solid-principles

INTERFACE SEGREGATION!

```java
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}


public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

We split the large interface (with no-overlapping features) into 3 smaller ones.

We now have the opportunity to apply only the interfaces relevant to us as zookeepers. OR we can also extend any of the separate interfaces more easily to add more features:

```java
public interface BearGroomer extends BearCleaner
{
    void cutTheBearsHair()
}
```

Content from:
https://www.baeldung.com/solid-principles

Q & A

THANKS!

- Stay safe, be encouraged, & see you next week!