Single Responsibility Principle (SRP)

Just because you can, doesn't mean you should

SOLID Design

CSCI 2134: Software Development

Agenda

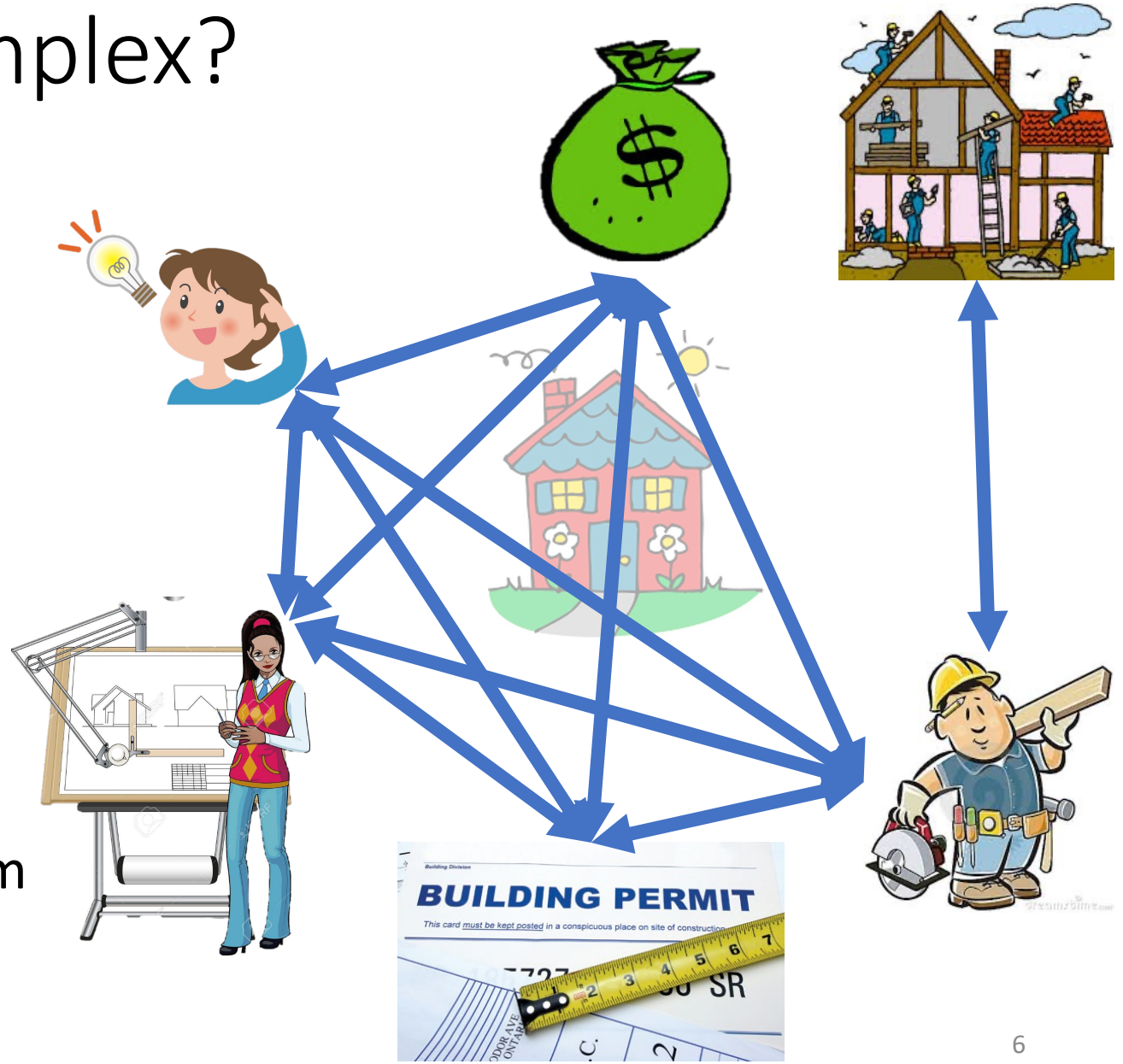
- Lecture Contents
  - Goal of good design
  - Characteristics of good design
  - SOLID Principles
- Brightspace Quiz
- Readings:
  - This Lecture: Chapter 5
  - Next Lecture: Chapter 5

Goals of Good Design

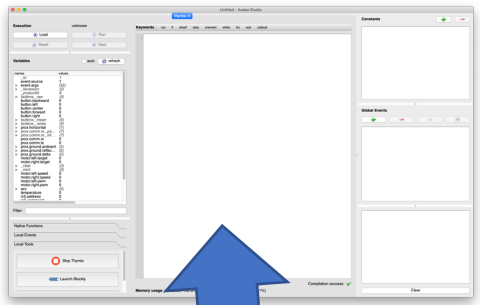
- Satisfy requirements
- Minimize complexity
  - The primary problem of software development is managing complexity
  - A good design reduces complexity through techniques such as
    - Abstraction
    - Decomposition
    - Encapsulation
  - Complexity cannot be eliminated, but it can be managed
- Maximize flexibility
  - Requirements change over the course of the project
  - A good design is flexible, accommodates change, and reduces the amount of changes in implementation as well
- Maximize maintainability
  - Understandability
  - Readability

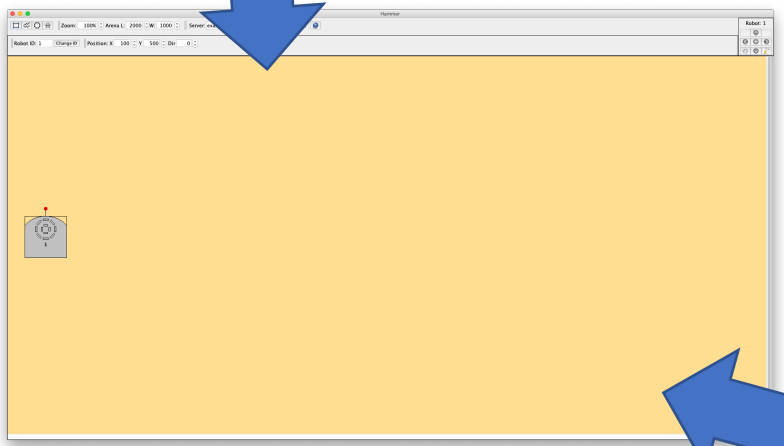
Why is Software Complex?

- Software models the real world
- The real world is complex
- Large software systems have to:
  - Interact with a variety of users
  - Interact with a variety of systems
  - Interact with a complex world
  - Meet a variety of different requirements
- Large software systems are built without ever completely understanding the entire problem



Example: Hammer.jar





Characteristics of Good Design

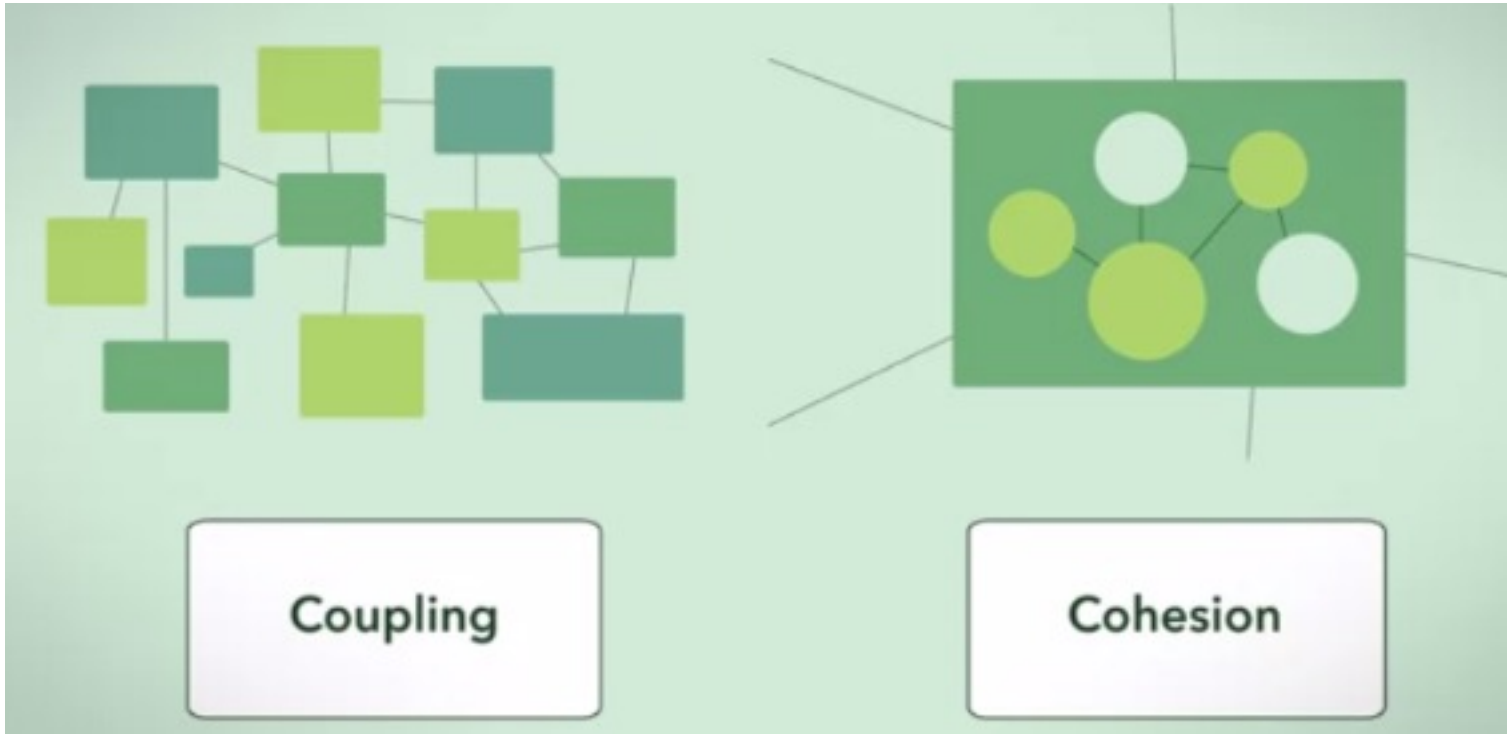
- Minimal complexity
  - Loose (low) coupling
  - Low-to-medium fan-out: single class does not use too many other classes
  - Leanness: minimum amount of code
  - Stratification: layering
  - Standard techniques
  - Reusability
  - High fan-in: single class is used by many classes)

- Flexibility
  - Adaptability
  - Extensibility
  - Portability

- Maintainability
  - Understandability
  - Readability

Two Common Design Criteria

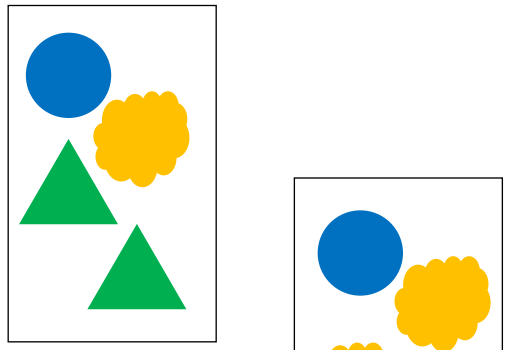
- **Cohesion:** a measure of relatedness to a single idea or responsibility of a class
- **Coupling:** a measure of dependence between classes or packages

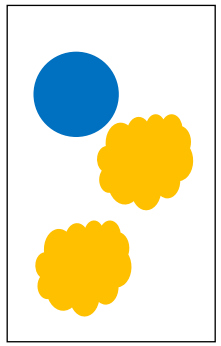


Cohesion

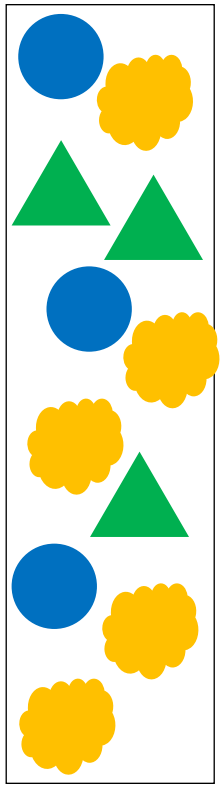
- Cohesion is a measure of relatedness to a single idea or responsibility of a class
  - A measure of how well everything sticks together
  - Aim for high cohesion
- Low cohesion means that either
  - Too many methods or classes are needed to complete a simple task (because the functionality is fragmented) or
  - One method or class has multiple functions, which makes the code difficult to understand
- **Key Ideas**:
  - Each class should represent a single concept
  - All methods of the class should be directly applicable to the concept
- Classes that are multi-concept or have unrelated methods reduce cohesion and indicate improvements are needed to the design.

Cohesion

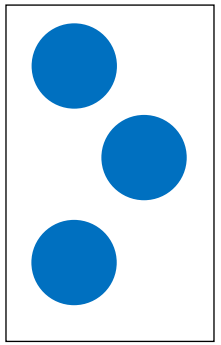


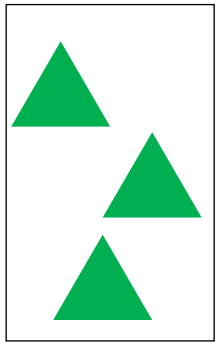


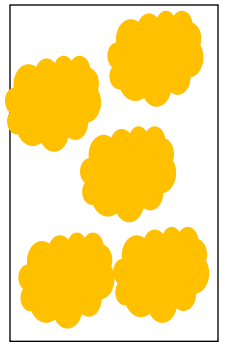
Bad cohesion
Too fragmented



Bad cohesion
Overloaded







Good cohesion

Example:
Bad vs Good Cohesion

```
public class Car {
  int engineSize;
  int gasTankSize;
  boolean diesel;
  boolean autoTransmission;
  …
}
```

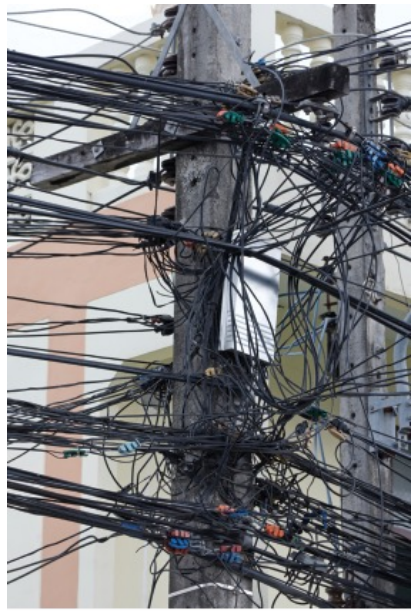
```
public class Car {
  Engine engine;
  Transmission transmission;
  FuelTank fuelTank;
  …
}
```

```
public class Engine {
  int size;
  boolean diesel;
  …
}
```

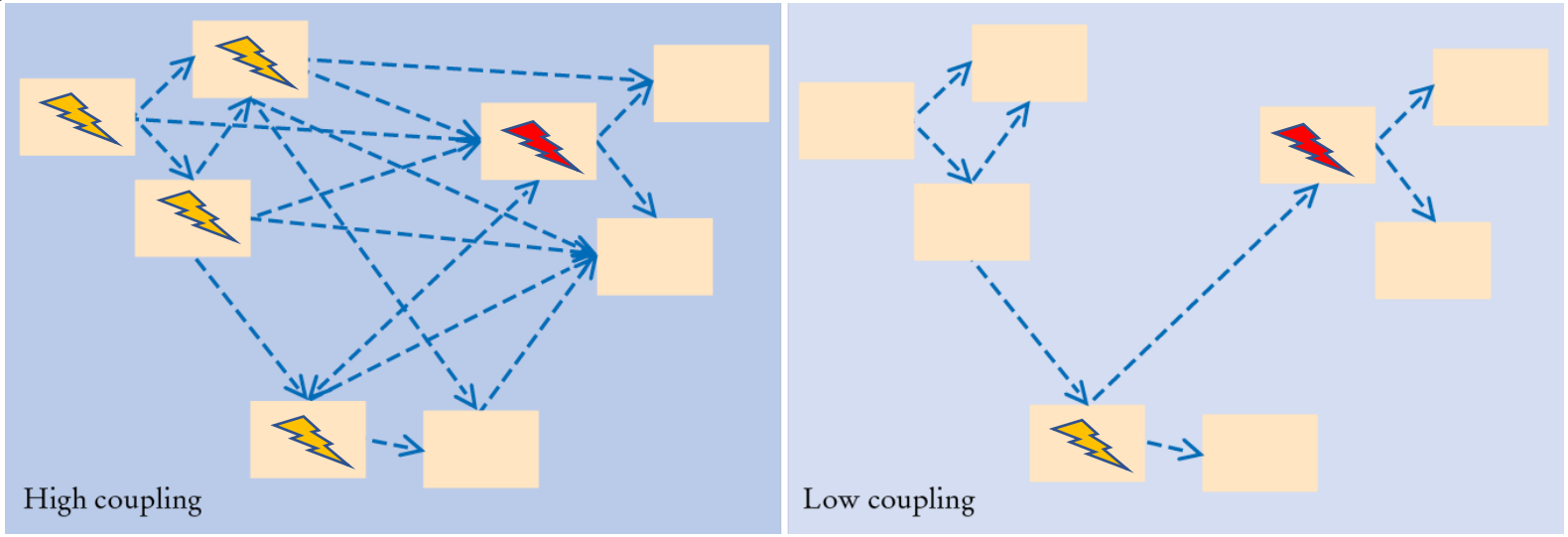
```
public class Transmission {
  boolean automatic;
  …
}
```

```
public class FuelTank {
  int size
  …
}
```

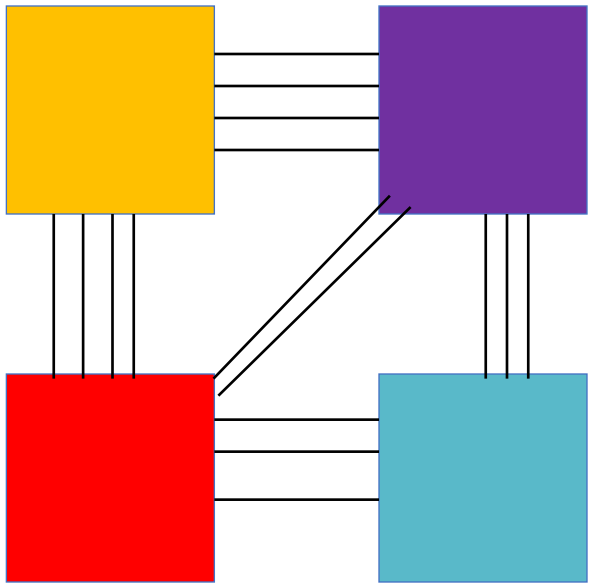
Coupling


© visual7/iStockphoto.

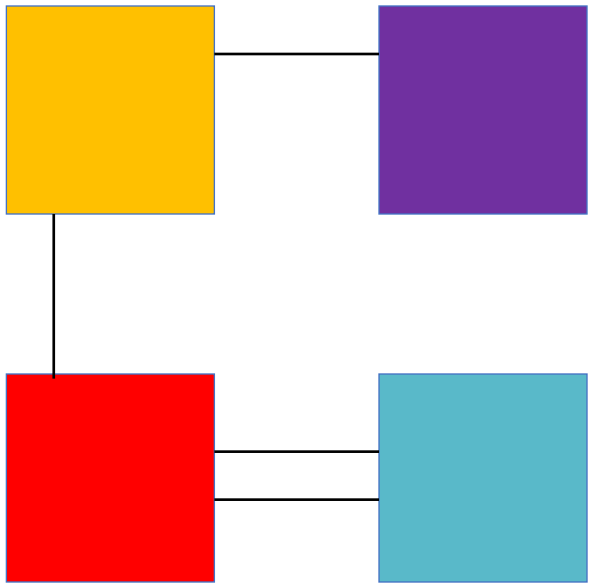
- Coupling is the density of dependencies among classes
  - High coupling means there are many dependencies
  - Low coupling means there are few dependencies
- Low coupling is preferred
- If a class changes and there is high coupling, many other classes will need to change as well



Coupling



High Coupling /
Tight Coupling



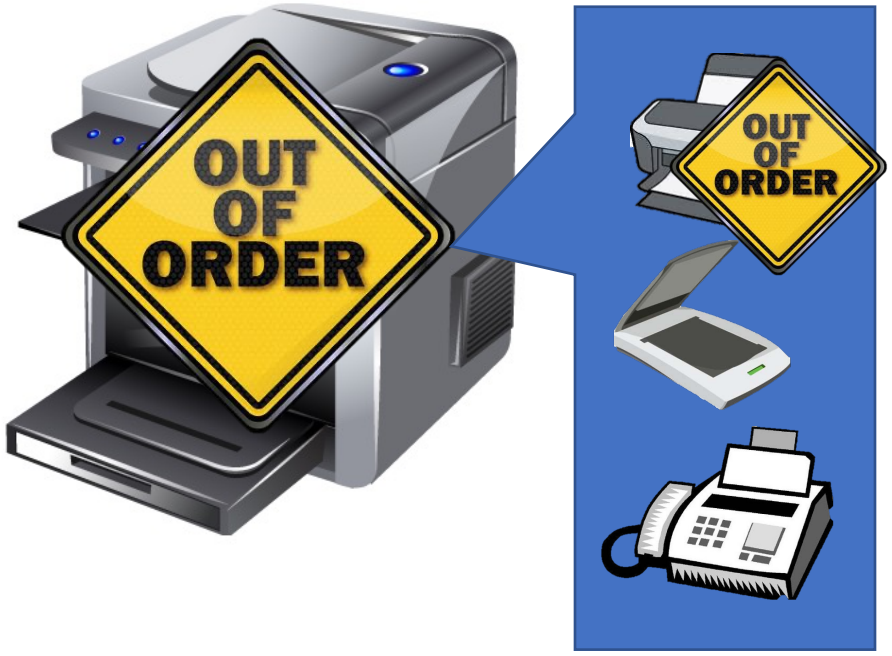
Low Coupling /
Loose Coupling

SOLID: Maximize Cohesion & Reduce Coupling

- SOLID is a set of basic object-oriented design principles for better design
- **Uses**:
  - Use SOLID principles to create code that is flexible and minimizes complexity
  - Use class-level refactoring to apply SOLID principles to classes that violate these principles
- SOLID stands for:
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen / Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DSP)

SOLID - Single Responsibility Principle (SRP)

Multiple Responsibilities



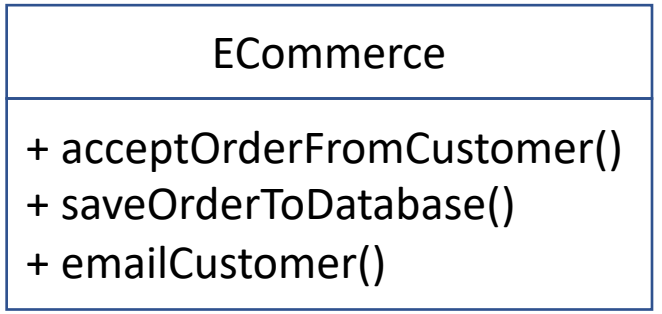
Single Responsibilities

SOLID - Single Responsibility Principle (SRP)

- Principle: Each class represents a single concept, role, or function
  - “Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.” - Wikipedia
  - “A class should have only one reason to change.” - Robert C. Martin (Uncle Bob)
- Purpose: Improves cohesion and reduces coupling
- Notes:
  - A class may have multiple methods, but these methods are all related to the same concept
  - If there are two unrelated methods in the same class, this is a sure sign that the SRP has been violated

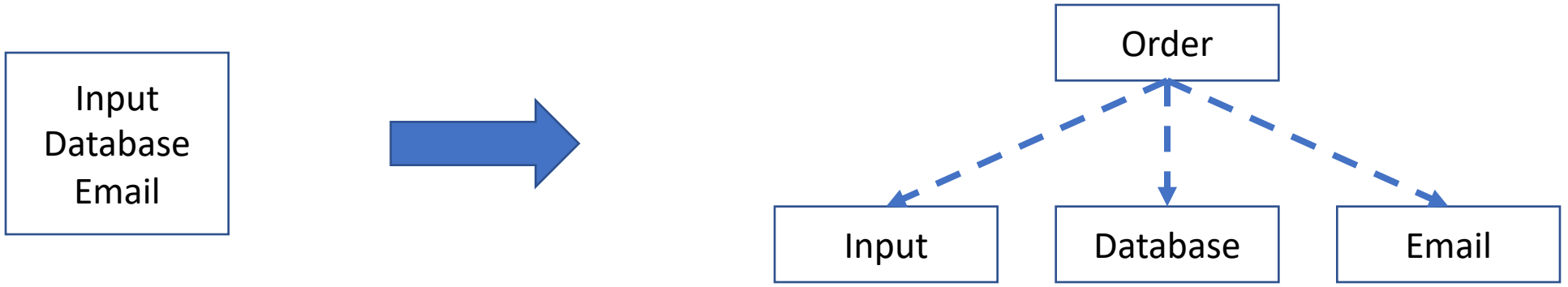
Example of SRP Violation:

- Scenario: Single class in an E-commerce system that
  - Accepts orders from a web page
  - Saves the order to the database
  - Emails the customer
- Code smell:
  - Three different functions in one class
  - Three unrelated dependencies
    - Input source
    - Database API
    - How email is sent
- Bad design choice because:
  - Multiple functionality is coupled together
  - Changing one part of the class could break another part



Fixing an SRP Violation

- To fix a single responsibility violation we typically split the offending class into multiple classes, each with a single responsibility



Another Example of SRP Violation

```
public class User {
  private int id;
  private String firstName;
  private String lastName;
  private String email;

  public User(String firstName, String lastName, String email) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }

  public boolean isValid(){
    return (firstName != "" && firstName != null &&
         lastName != "" && lastName != null &&
         email != "" && email != null);
  }

  public String displayName() {
    if (isValid()) {
      return "<H1>" + firstName + " " + lastName
          + "</H1><BR/><H2>" + email + "</H2>";
    } else {
      return "<BLINK>INVALID!</BLINK>";
    }
  }
}
```

Primary function:
Store a user

Secondary function
Output HTML

SRP Fix

```
public class User {
  private int id;
  private String firstName;
  private String lastName;
  private String email;

  public User(String firstName, String lastName, String email) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }

  public boolean isValid(){
    return (firstName != "" && firstName != null &&
         lastName != "" && lastName != null &&
         email != "" && email != null);
  }
}
```

Primary function:
Store a user

```
public class HTMLUser extends User {
  public HTMLUser(String firstName, String lastName, String email) {
    super(firstName, lastName, email);
  }

  public String displayName() {
    if (isValid()) {
      return "<H1>" + firstName + " " + lastName + "</H1><BR/><H2>" + email + "</H2>";
    } else {
      return "<BLINK>INVALID!</BLINK>";
    }
  }
}
```

Secondary function
Output HTML

SOLID – Open/Close Principle (OCP)

- Principle: Classes should be extendable but not modifiable.
  - “Software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification” - Object-Oriented Software Construction, Bertrand Meyer
  - Write once – Use many
  - There should be no need to modify a class to extend functionality.
- Definitions:
  - A class is **open**, if fields can be added, data structure changed, or internal functions modified
  - A class is **closed**, If it can be safely used by other classes because it will not be modified

SOLID – Open/Close Principle (OCP)

- **Purpose: reduce coupling**

- If class A uses class B, this creates coupling.
  - If class B is modified, this forces class A to be modified.
  - If class B is closed, the coupling is less dangerous

- Goal is to design classes that, once completed and tested, should never need to be modified again.
  - All extensions are done through subclasses

SOLID – Open/Close Principle (OCP)

- Example of an OCP violation:
  - An insurance company has a number of different policies
  - The business rules are encoded in a the *InsurancePolicy* class
  - Every time a new kind of insurance product is created, the InsurancePolicy class needs to be modified

- Code Smell:
  - This class may need to be modified in the future for other reasons than defect fixing!
  - Class cannot be extended without modifying it.

Example of OCP Violations

Example from http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/

```
// This class violates the Open/Closed principle.
// Why?

public class AreaCalculator {
  public static float Area(Rectangle[] shapes) {
    float area = 0.0f;
    for (int i = 0; i < shapes.length; i++) {
      area += shapes[i].getHeight() *
           shapes[i].getWidth();
    }
    return area;
  }
}
```

Are there other shapes beyond rectangles?

YES!

```
public class Rectangle {
  private float height;
  private float width;

  public Rectangle(float height, float width) {
    this.height = height;
    this.width = width;
  }

  public float getHeight() {
    return height;
  }

  public float getWidth() {
    return width;
  }
}
```

Example of OCP Violations

Example from http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/

```
// This class violates the Open/Closed principle.
// Why?

public class AreaCalculator {

  public static float Area(Object[] shapes) {
    float area = 0.0f;

    for (int i = 0; i < shapes.length; i++) {
      if (shapes[i] instanceof Rectangle) {
        Rectangle rect = (Rectangle) shapes[i];
        area += rect.getHeight() *
            rect.getWidth();
      } else if (shapes[i] instanceof Circle) {
        Circle circle = (Circle) shapes[i];
        area += circle.getRadius() *
            circle.getRadius() * Math.PI;
      }
    }

    return area;
  }
}
```

Danger Danger!

```
public class Rectangle {
  private float height;
  private float width;

  …
}
```

```
public class Circle {
  private float radius;

  public Circle(float radius) {
    this.radius = radius;
  }

  public float getRadius() {
    return radius;
  }
}
```

Are we deciding how to treat an object using an **`if`** or a **`switch`**?

YES!

Example of Fixing the OCP Violations

Example from http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/

```
// This class adheres to the Open/Closed principle.
// It never needs to change, for the rest of time
// this class can calculate the total area of any
// objects passed to it, so long as those objects
// implement the area() method of the IShape
// interface contract.

public class AreaCalculator {
  public static float Area(IShape[] shapes) {
    float area = 0.0f;
    for (int i = 0; i < shapes.length; i++) {
      area += shapes[i].area();
    }
    return area;
  }
}
```

All shapes that implement the IShape interface

```
public class Rectangle implements IShape {
  private float height;
  private float width;

  …

  public float area() {
    return height * width;
  }
}
```

```
public class Circle implements Ishape {
  private float radius;

  …

  public float area() {
    return radius * radius * Math.PI;
  }
}
```

```
public interface IShape {
  public float area();
}
```

As long as a *Shape* implements the IShape interface, AreaCalculator will never change!

Key 

Points

- Good design reduces complexity and improves flexibility and maintainability
- Coupling and cohesion are two measures of good design, with low coupling and high cohesion leading to lower complexity and better flexibility and maintainability
- SOLID is a set of object-oriented design principles intended to reduce complexity
- The Single Responsibility Principle states that each class should have a single responsibility
- The Open/Closed Principle states that classes should be designed to be extendable (open) but not modifiable (closed)

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 18, 2020

- https://images.slideplayer.com/25/8104238/slides/slide_7.jpg