COUPLING AND COHESION

Chandan R. Rupakheti

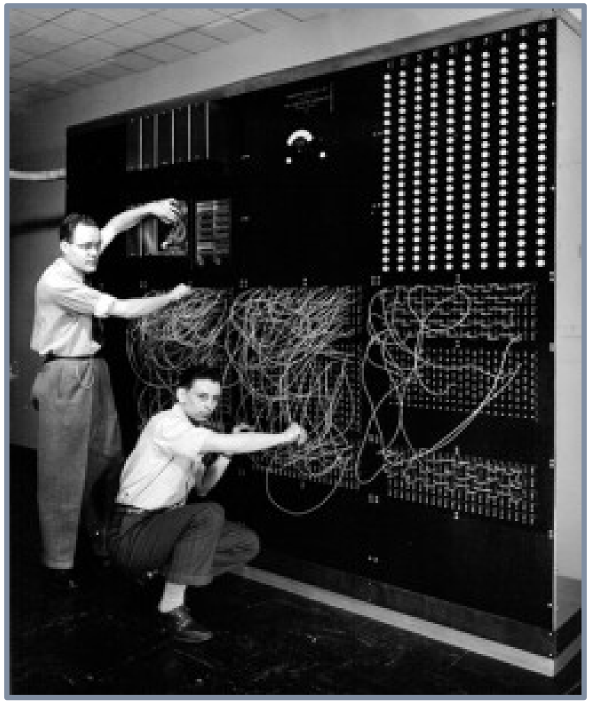
Week 2-2

Today

- Coupling
- Cohesion
- Command and Query Separation Principle

Coupling or Dependency

Coupling is the degree to which a software component relies on other software components to achieve its purpose



Cohesion

Cohesion refers to the degree to which elements of a software component belong together

The Classic SE Problem

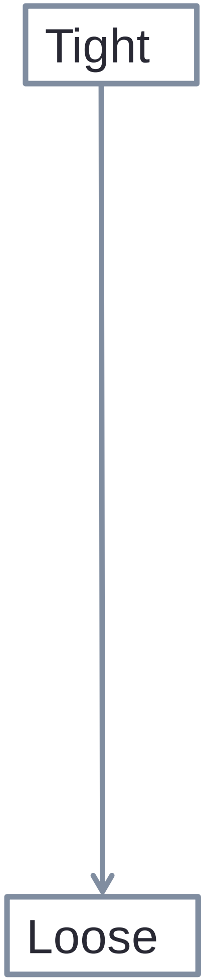
You **cannot** have no coupling and total cohesion at the same time

A good design is all about maintaining the right balance

COUPLING

The necessary evil …

Types of Coupling



Tight

Loose

1. Content
2. Common
3. External
4. Control
5. Stamp
6. Data
7. Message
8. No Coupling

Content Coupling

Module A has access to local data of **Module B**

```java
class Department {
 private List<Student> students;

 public List<Student> getStudents() {
  return students;
 }
}

class School {
 private List<Department> departments;

 public void addStudent(Student s) {
  for(Department d: departments) {
   d.getStudents().add(s);
  }
 }
}
```

Common Coupling

Global variables shared between modules

```
class GameState {
 public static int score;
}

class GameEngine {
 public void addScore(int score) {
  GameState.score += score;
 }
}

class Player {
 public void resetScore() {
  GameState.score = 0;
 }
}
```

External Coupling

Share an externally imposed data format or communication protocol

```
// <<external_software>>
class Serializer {
 public static byte[] serialize(String s) {
  byte[] ser = s.getBytes();
  for(int i = 0; i < ser.length; ++i) {
   ser[i] = (byte)(ser[i] & 0x1F00);
  }
  return ser;
 }
}
```

```
class Player {
 String name;
 public void save() throws Exception {
  byte[] ser = name.getBytes();
  for(int i = 0; i < ser.length; ++i) {
   ser[i] = (byte)(ser[i] & 0x1F00);
  }
  FileOutputStream f = new FileOutputStream("player.dat");
  f.write(ser);
  f.close();
 }
}
```

Control Coupling

Module A controls the control flow of **Module B** through flags

```
class Register {
 private Sale s;
 public void transact(float amount) {
  int stateCode = System.in.read();
  s.process(amount, stateCode);
 }
}

class Sale {
 private int total;
 public void process(float amount, int c) {
  if(c == 1)
   total += amount + 0.06 * amount;
  else
   total += amount + 0.08 * amount;
 }
}
```

Stamp Coupling

Modules share a composite data structure but uses only a part of it

```
class Student {
 private int id;
 private String name;
 private String location;

 public int getId() {
  return id;
 }
}

class Registrar {
 Map<Integer, Float> idToGPA;

 public float checkGPA(Student s) {
  return idToGPA.get(s.getId());
 }
}
```

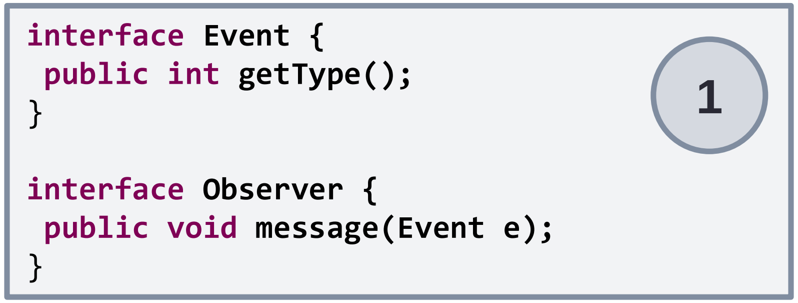
Data Coupling

Modules share data through parameters

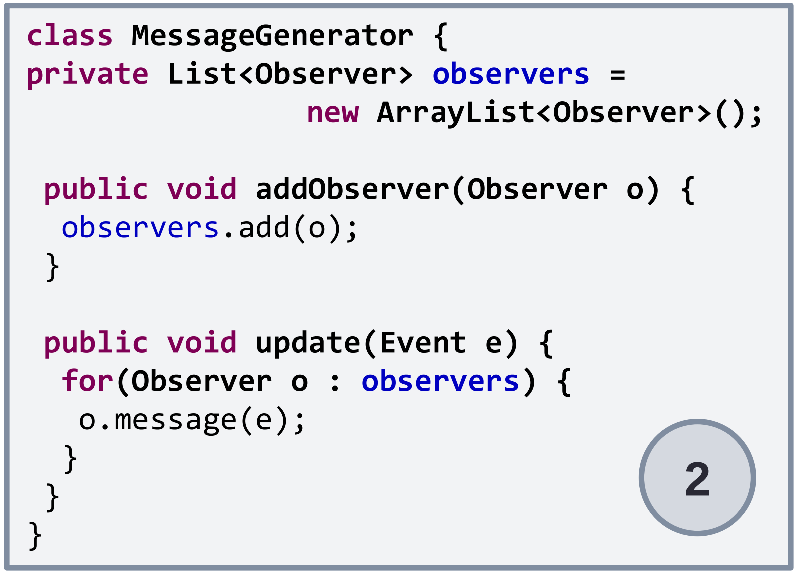
```
class GameEngine {
 List<Sprite> sprites;

 public void animate() {
  for(Sprite s : sprites) {
   s.move(5, -5);
  }
  // ...
 }
}

class Sprite {
 private int x,y;

 public void move(int dx, int dy) {
  x += dx;
  y += dy;
 }
}
```

Message Coupling

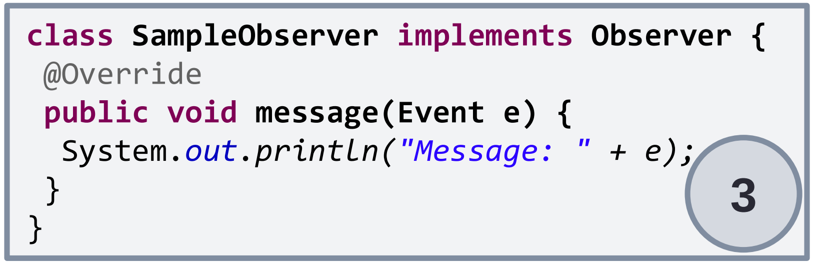
Modules communicate via message passing (e.g. Observer)



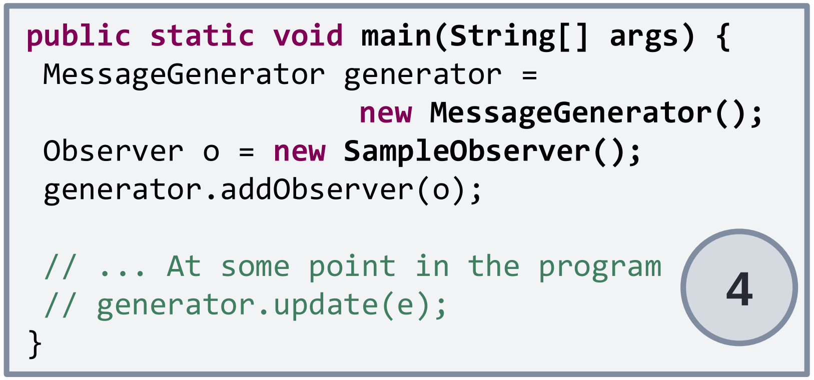
```
interface Event {
 public int getType();
}

interface Observer {
 public void message(Event e);
}
```



```
class MessageGenerator {
private List<Observer> observers =
                new ArrayList<Observer>();

 public void addObserver(Observer o) {
  observers.add(o);
 }

 public void update(Event e) {
  for(Observer o : observers) {
   o.message(e);
  }
 }
}
```



```
class SampleObserver implements Observer {
 @Override
 public void message(Event e) {
  System.out.println("Message: " + e);
 }
}
```



```
public static void main(String[] args) {
 MessageGenerator generator =
                  new MessageGenerator();
 Observer o = new SampleObserver();
 generator.addObserver(o);

 // ... At some point in the program
 // generator.update(e);
}
```

No Coupling

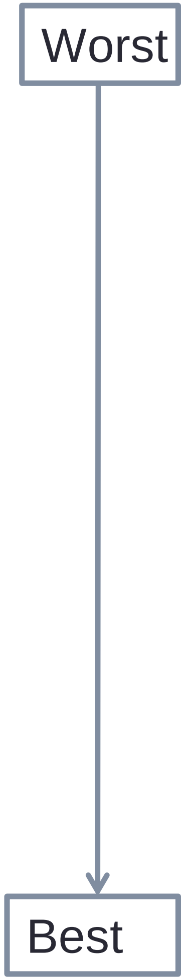
Only possible in the dream world!

Components need to talk with each other to achieve complex functionality!

COHESION

A distant dream of software engineers …

Types of Cohesion



1. Coincidental
2. Logical
3. Temporal
4. Procedural
5. Communicational
6. Sequential
7. Functional

Coincidental Cohesion

When parts of a module is grouped arbitrarily, e.g. Utilities class

```
public class Utilities {
 public static void saveUserPrefs(String prefs) {
  // ...
 }

 public static Connection connect(String server,
         String user, String password) {
  // ...
  return null;
 }

 public static String serializeToXML(Object o) {
  // ...
  return null;
 }
 // ...
}
```

Logical Cohesion

Grouped because they are logically categorized to do the same thing, e.g. MouseListener

```
public interface MouseListener
                 extends EventListener {

 public void mouseClicked(MouseEvent e);
 public void mousePressed(MouseEvent e);
 public void mouseReleased(MouseEvent e);

 public void mouseEntered(MouseEvent e);
 public void mouseExited(MouseEvent e);
}
```

Temporal Cohesion

Components grouped at runtime, e.g. exception processing function that does multiple task, such as close file, log error, and notify users

```java
public class InputProcessor {
 public void readInput(String file) {
  FileInputStream fIn = null;
  try {
   fIn = new FileInputStream(file);
   // Process file ... fIn.read();
  }
  catch(Exception e) {
   Utilities.handleError(e, fIn);
  }
 }
}

class Utilities {
 public static void handleError(Exception e,
                     FileInputStream file) {
  // Handle exception
  // Close the file
  // ...
 }
}
```

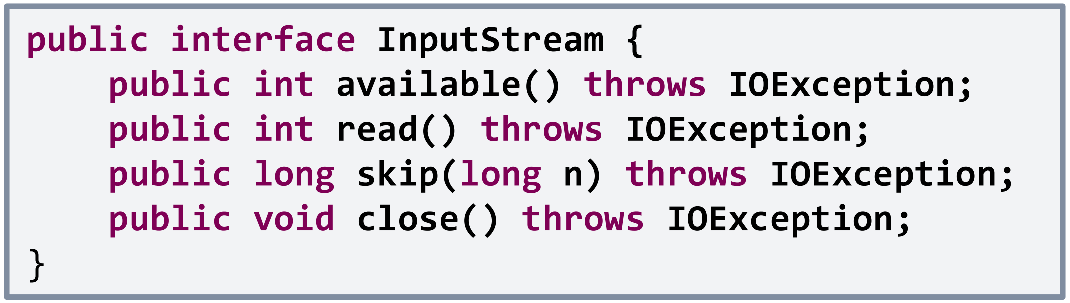
Procedural Cohesion

Grouped because they follow a certain sequence of execution, e.g. function which checks file permission and opens it

```
public interface InputStream {
    public int available() throws IOException;
    public int read() throws IOException;
    public long skip(long n) throws IOException;
    public void close() throws IOException;
}
```

Communicational Cohesion

Grouped because they operate on the same data



```
public interface InputStream {
    public int available() throws IOException;
    public int read() throws IOException;
    public long skip(long n) throws IOException;
    public void close() throws IOException;
}
```

Sequential Cohesion

Grouped because the output from one part is the input to another part

```
abstract class RGBTranformer {

 protected abstract Image tranformRed(Image image, int r);
 protected abstract Image tranformGreen(Image image, int g);
 protected abstract Image tranformBlue(Image image, int b);

 public Image transform(Image image, int r, int g, int b) {
  Image transformed = this.tranformRed(image, r);
  transformed = this.tranformGreen(transformed, g);
  transformed = this.tranformBlue(transformed, b);
  return transformed;
 }
}
```

Functional Cohesion

Grouped because they all contribute to a single well-defined task of the module

```java
abstract class RGBTranformer {

 protected abstract Image tranformRed(Image image, int r);
 protected abstract Image tranformGreen(Image image, int g);
 protected abstract Image tranformBlue(Image image, int b);

 public Image transform(Image image, int r, int g, int b) {
  Image transformed = this.tranformRed(image, r);
  transformed = this.tranformGreen(transformed, g);
  transformed = this.tranformBlue(transformed, b);
  return transformed;
 }
}
```

Command-Query Separation Principle

- Each method should be either a **command** or a **query**
- **Command Method**
  - Performs an action, typically with side effects, but has no return value
- **Query Method**
  - Returns data but has no side effects

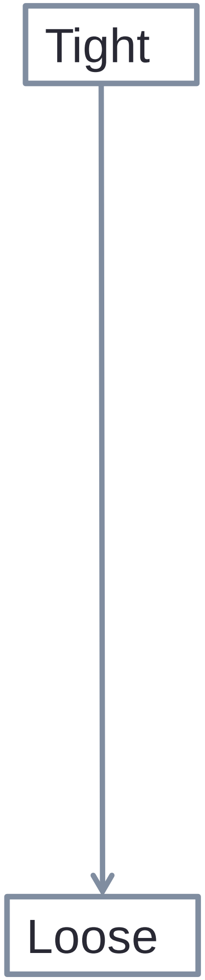
Why is Command-Query Important?

- **Principle of least surprise**, side effects only happen in “void” methods
- Provides for most flexible interface, e.g. a value can be queried multiple times without changing it

Recap

Design Principle:

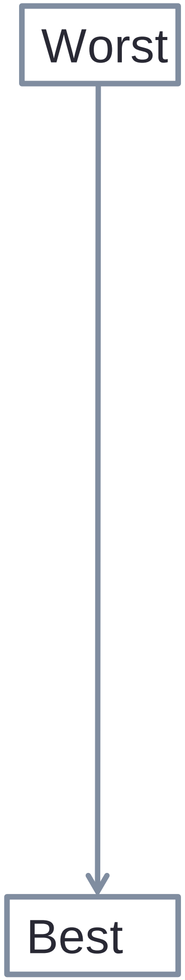
Each method should be either a command or a query but not both.



Tight

Loose

1. Content
2. Common
3. External
4. Control
5. Stamp
6. Data
7. Message
8. No Coupling



Worst

Best

1. Coincidental
2. Logical
3. Temporal
4. Procedural
5. Communicational
6. Sequential
7. Functional

Next Week

- Things Due
  - Client meeting during class (Lead Group?)
  - **Sprint 5** due in class
  - **Homework 2** by Tuesday, 8:00 am
  - **Exam 1** by Friday 5:10 pm
- Concepts
  - The Factory Method Pattern
  - The Dependency Inversion Principle
  - The Abstract Factory Pattern
- Exam 1 (Two Parts)
  - **In-Class** – Open Book / Open Note / Open Moodle - Online Quiz (20%)
  - **Take Home** – Design / Implementation / Testing Problems (80%)