D.R.Y., Cohesion, & Coupling

Don't Repeat Yourself (D.R.Y.)

Any information in your program should have a single place where it's kept, held, and represented.

"information" can mean primitive data (some `int`), invariants (`Account` objects can't have negative balances), or even comments & documentation.

Repeated Data

You might need to use suits of cards in both the `Card` and the `Hand` class.

```
public class Card {
    private static final String[] SUITS = {"S", "H", "D", "C"};
    ...// the rest
}

public class Hand {
    private static final String[] SUITS = {"S", "H", "D", "C"};
    ...// the rest
}
```

DON'T Repeat Data

Every card has a suit, sure. But why bother storing the array of suits twice?

```
public class Card {
    private static final String[] SUITS = {"S", "H", "D", "C"};
    ...// the rest
}
public class Hand {
    public double scoreHand() {
        ...
        for (String suit : Card.SUITS) {
            ...
        }
    }
}
```

Repeatedly Checking Invariants

```java
public class Account {
    public void deposit(double amt) {
        if (amt < 0) {
            throw new IllegalArgumentException("Can't deposit negative amount");
        } else {
            balance += amt;
        }
    }
    public double withdraw(double amt) {
        if (amt < 0) {
            throw new IllegalArgumentException("Can't withdraw negative amount.");
        } else {
            balance -= amt;
        }
    }

}
```

Simplify Checking Invariants

```java
public class Account {
    private void validateTransaction(double amt) {
        if (amt < 0) {
            throw new IllegalArgumentException("Invalid transaction amount");
        }
    }
    public void deposit(double amt) {
        validateTransaction(amt)
        balance += amt;
    }
    public double withdraw(double amt) {
        validateTransaction(amt)
        balance -= amt;
    }
}
```

Rules of D.R.Y.

It's better to avoid repeating yourself

- less typing
- fewer changes to repeat in case you have to make a change
    - Maybe negative deposits shouldn't throw an exception--without our changes, we'd have to change a bunch of conditionals instead of just changing our single function.
    -

It's also better to keep things simple whenever possible. ("AHA")

Cohesion and Coupling

How to write high-quality classes.

Reminders:

- A class should represent a single concept
  - Remember--classes are usually **singular**, not plural.
    - `Card`, not `Cards`
- The public methods and constants that a class contains make up its **public interface**
  - This is the set of ways in which your object can be used by other objects.
- If a class refers to too many concepts, it's time to break it up
  - Back to CRC: too many responsibilities? Make another card.

Cohesion

The degree to which all features of a public interface are related to the *single* concept that the class represents.

Example: Employee Payment System

Employees receive their biweekly paychecks. They are paid their hourly rates for each hour worked; howeever, if they worked more than 40 hours per week, they are paid overtime at 150% of their regular wage.

A First Attempt: One Class

```java
public class PaymentSystem {

    public void setRate(String empName, double wage) {...}
    public double getRate(String empName) {...}

    public void addHoursWorked(String empName, double hours) {...}

    public void payEmployees() {...}
    private void payEmployee(String empName) {...}
}
```

What concept(s) is/are being represented here?

A First Attempt: One Class

This implementation really has (at least!) two concepts built in:

- There is a Payment System, which should be able to dispense a certain amount of money to each Employee
- There are Employees, who possess information about their wages and their hours worked in a week.

Maybe more?

New Idea 1

Add a second class to represent the Employee separately.

```
public class PaymentSystem {
    public void payEmployees() {...}
    public void payEmployee(Employee e) {...}
}

public class Employee {
    public void setRate(double wage) {...}
    public double getRate() {...}

    public void addHoursWorked(double hours) {...}
}
```

New Idea 2

Add a third class that manages all timekeeping.

```
public class PaymentSystem {
    private TimeEntryTable timeEntry;
    public void payEmployees() {...}
}

public class TimeEntryTable {
    public void payEmployee(Employee e) {...}
    public void addHoursWorked(Employee e, double hours) {...}
}

public class Employee {
    public void setRate(double wage) {...}
    public double getRate() {...}
}
```

Pair Activity: Evaluate the Implementations

Work with a partner.

- Do the classes in Idea 1 each represent just one concept?
- Do the classes in Idea 2 each represent just one concept?
- In which implementation do the public interfaces have the most *cohesion*?
    - That is, which Idea makes it so that the methods within each class are accomplishing related tasks?

Cohesion Summary

Ultimately, we're trying to make sure that classes have clear responsibilities. This is a dual relationship.

- A class should be separated into multiple classes if it has too many disparate responsibilities
- Each individual class should have its own clearly defined responsibility.

Another Example

A cash register!

```
public class CashRegister {
    public void enterPayment(int dollars, in quarters, int dimes, int nickels, int pennies) {...}
    ...
    public static final double NICKEL_VALUE = 0.05;
    public static final double DIME_VALUE = 0.10;
    public static final double QUARTER_VALUE = 0.25;
    ...
}
```

There are two concepts here. What are they, and how can we break this class up?

Another Example, Take Two

```
public class CashRegister {
    public void enterPayment(Coin[] payment) {...}
}

public class Coin {
    public Coin(double value, String name) {...}
    public double getValue() {...}
}
```

- The CashRegister is responsible for storing money.
- The Coin is responsible for knowing how much it's worth.

Evaluating the Change

The classes now have their own clearly defined responsibilities, which is good.

Have we introduced any *dependencies*, or classes that need to use another class in order to function? (Think back to *collaboration* from CRC.)

Dependencies

A `CashRegister` needs the `Coin` class in order to function! This is a new dependency.

This dependency is highly assymmetrical: the `Coin` doesn't know the first thing about the `CashRegister`.

```
public class Coin {
    public Coin(double value, String name) {...}
    public double getValue() {...}
}
```

Dependencies vs. Collaboration

We learned about collaboration when talking about CRC cards.

- CRC is highly useful for deciding which classes might be needed to implement a project.
- Collaboration as a symmetric relationship makes it hard to soundly reason about whether some classes are too tightly connected or not.
- Dependency as a potentially one-way relationship can be easier to evaluate.

Coupling

It's good that the `CashRegister` depends on the `Coin`, but that the `Coin` doesn't depend on the register!

When we need to change the register, we need to reference the `Coin` class but change ONLY the `CashRegister` class.

If we change `Coin`, then we might need to propagate those changes back to the `CashRegister` class too.

Coupling

When many classes of a program depend on each other, then we have **high coupling** among our classes.

If there are relatively few dependencies, then we have **low coupling.**

Low Coupling vs. High Coupling

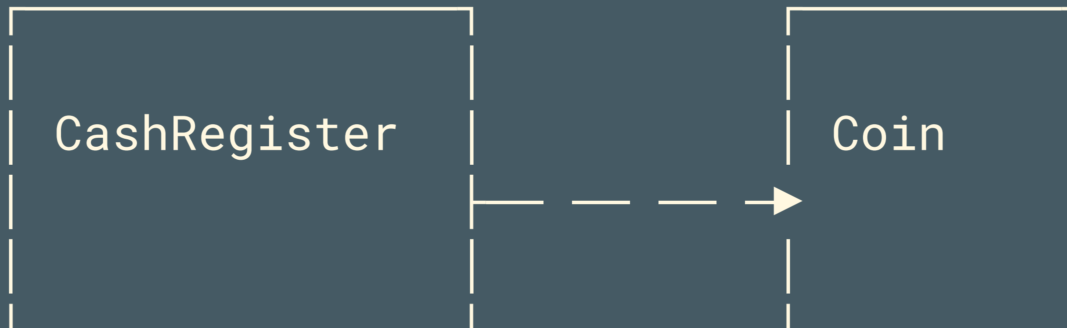
Low coupling is preferable to high coupling.

- If a depended-on class needs to change, all classes that depend on it might need updates.
- If we want to reuse a class between projects, then you'll also need to take along all of the classes on which that class depends.

Diagramming Coupling

"Unified Modeling Language", or UML, is used for visualizing relationships among objects. There are a lot of details to UML, but we'll just borrow two for now.

- Represent a class as a box
- Represent a dependency by drawing a dashed arrow pointing to the dependent class.



Dependency Diagram Practice

```
public class PaymentSystem {
    public void payEmployees() {...}
    public void payEmployee(Employee e) {...}
}

public class Employee {
    public void setRate(double wage) {...}
    public double getRate() {...}

    public void addHoursWorked(double hours) {...}
}
```

Dependency Diagram Practice

```
public class PaymentSystem {
    private TimeEntryTable timeEntry;
    public void payEmployees() {...}
}

public class TimeEntryTable {
    public void payEmployee(Employee e) {...}
    public void addHoursWorked(Employee e, double hours) {...}
}

public class Employee {
    public void setRate(double wage) {...}
    public double getRate() {...}
}
```

Questions

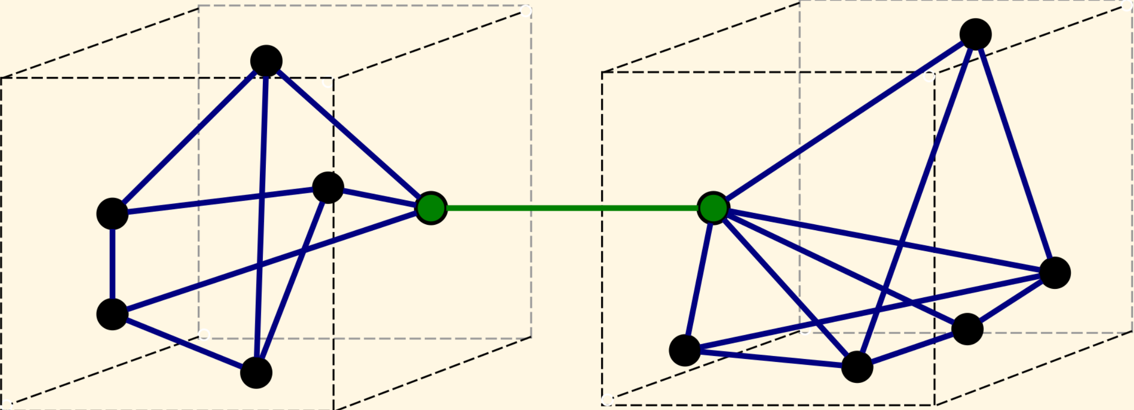
- If we change `Employee` dramatically in example one, which classes might later have to get updated? Example two?
- Based on the number of dependencies, which solution has higher coupling?

Cohesion and Coupling

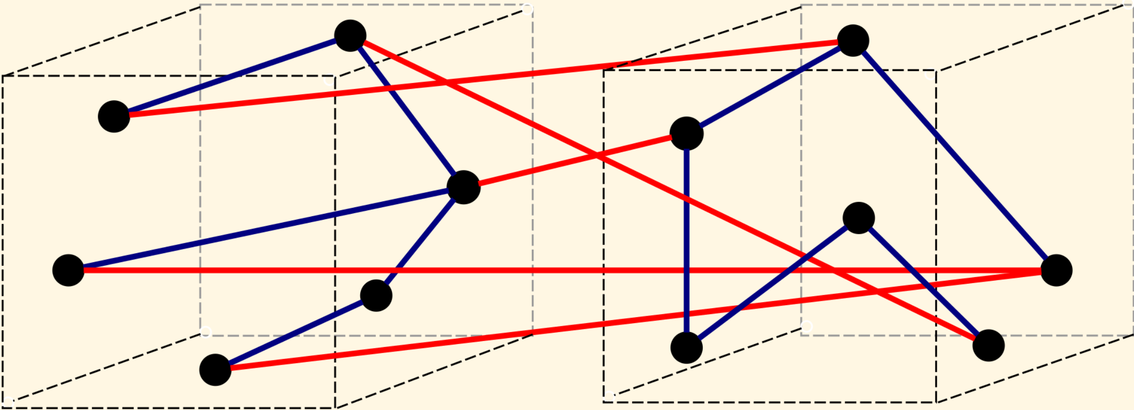
Overall, we'd like to have high cohesion and low coupling.

- Classes should have the methods to address a particular purpose.
- In focusing on maintaining individual purposes, classes will therefore need to depend on others to represent a system.
- Each class should depend on as few others as possible in order to fulfill its purposes.

A Nice Public Domain Diagram



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

A Guide to Code Reviews

A code review is an activity involving a writer and a reader.

- The writer briefly guides the reader through some code they've written.
- The reader follows along, and takes a short amount of time to read over the writer's code.
- The reader asks questions to help understand the writer's code.
- Once the reader understands the shape of the writer's code, the reader identifies good design choices that the reader made and provides suggestions about how to improve other aspects.

Rules of Code Reviews

- Be kind and patient
  - Showing your code to someone else is intimidating and often feels embarrassing at first.
- Be respectful when offering suggestions
  - You should discuss places where you think you might refactor the writer's code, but not criticize the writer themself.
- Make sure to commend good choices that you see, too.

Activity

Pull up your HW4 work, and find a partner. Review each others' code.

This should take at least 10 minutes per side.

- Are the classes cohesive?
- Is there high or low coupling?
- What choices could be changed to raise cohesion and lower coupling?
- Are there repeated pieces of information that could be improved (DRY?