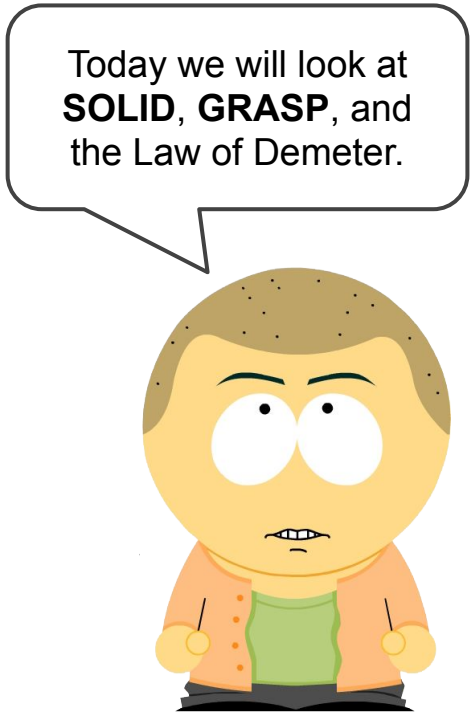
SWEN-262
Engineering of Software Subsystems

Design Principles

Object Oriented Design

- Up to this point, you have studied *Object Oriented Design* (OOD) at the class level.
  - You have considered inheritance, but not much beyond that.
- This semester you will need to expand your skills to larger scale systems comprising multiple subsystems.
  - You will need to consider the interactions between classes and the effect that one class has on other classes in the system.
- The software engineering community has put forward sets of principles that we will remind you of today.



SOLID

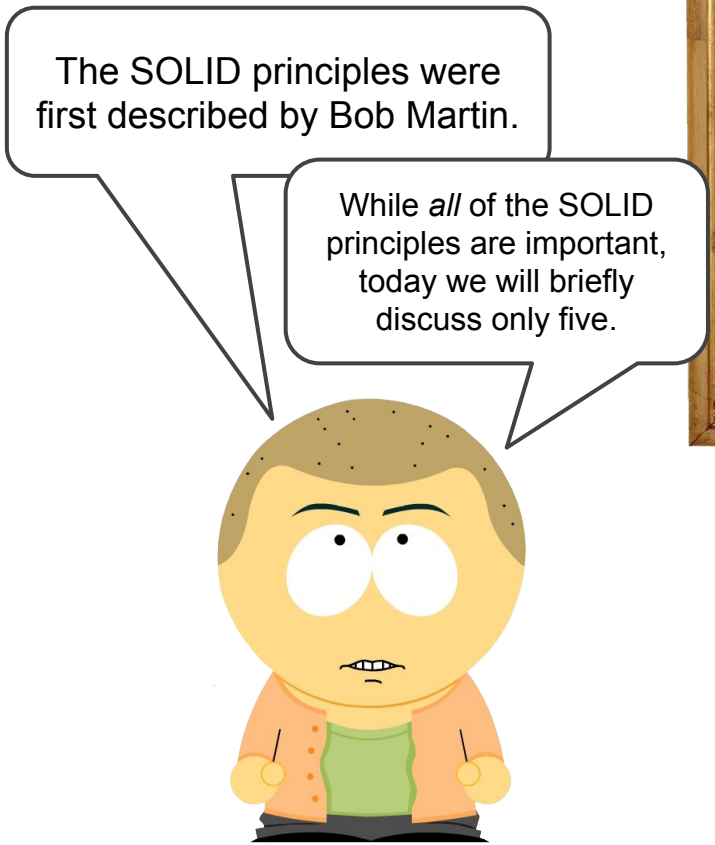
S ingle Responsibility Principle

O pen-closed Principle

L iskov Substitution Principle

I nterface Segregation Principle

D ependency Inversion Principle



Uncle Bob

SOLID

S ingle Responsibility Principle

O pen-closed Principle

L iskov Substitution Principle

I nterface Segregation Principle

D ependency Inversion Principle

Attack of the Blob!

Imagine we're making a drawing program with tools to draw rectangles and circles (among other things).

What if we tried to create one `Shape` class that was used to draw *both* circles *and* squares?

It uses a ***type code*** and a bunch of `if`/`else` statements to decide whether to draw, more, or scale like a circle or a square.

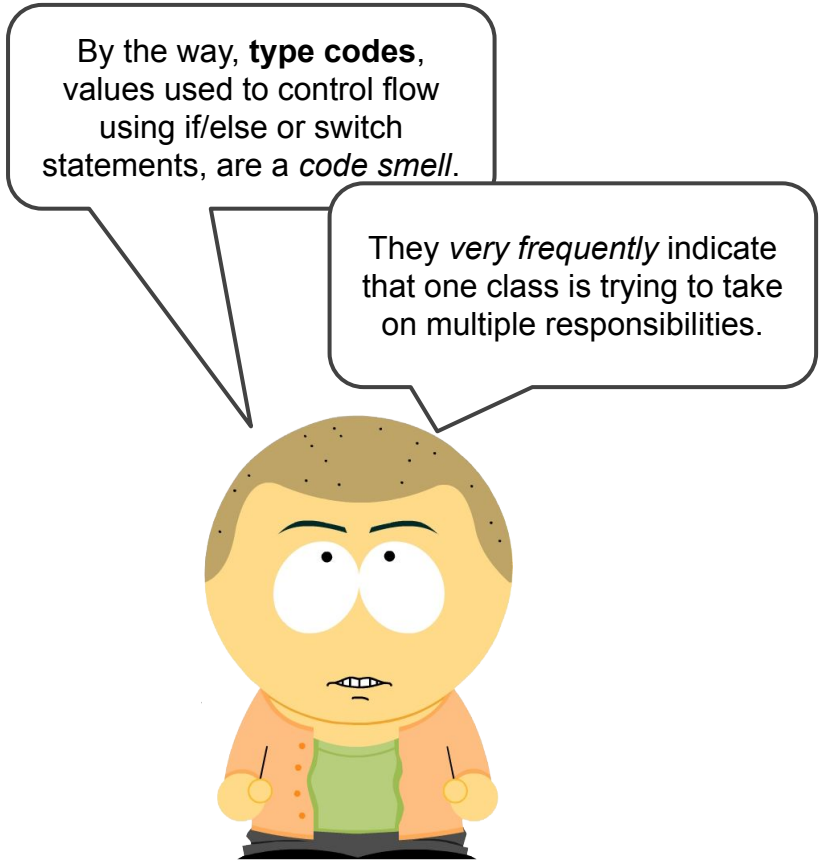
Like this...

Shape
-position: Position -height: double -width: double -radius: double -type: {Circle, Rectangle}
+move(Position p) +scale(Float factor) +draw(graphics g) +contains(Position p)

```
public void draw(Graphics g) {
  if(type == RECTANGLE) {
    // draw a rectangle
  }
  else if(type == CIRCLE) {
    // draw a circle
  }
}
```

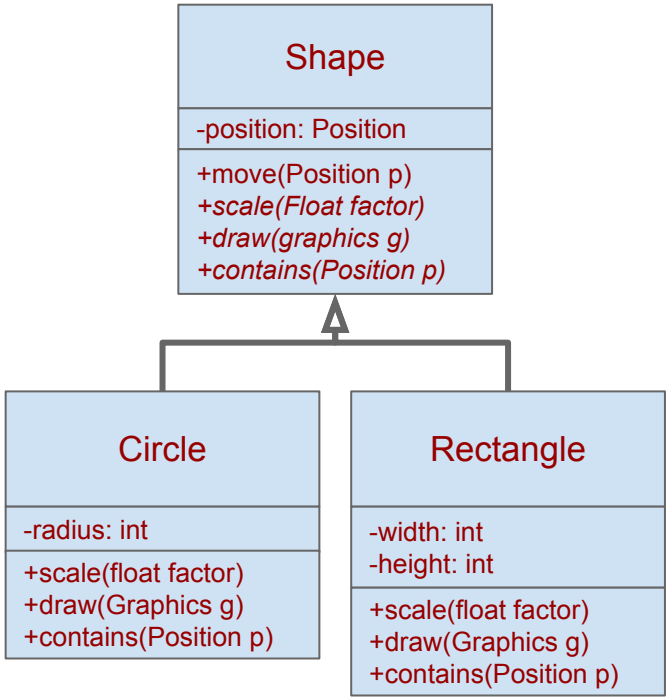
Attack of the Blob!

It uses a ***type code*** and a bunch of `if`/`else` statements to decide whether to draw, more, or scale like a circle or a square.



Single Responsibility Principle

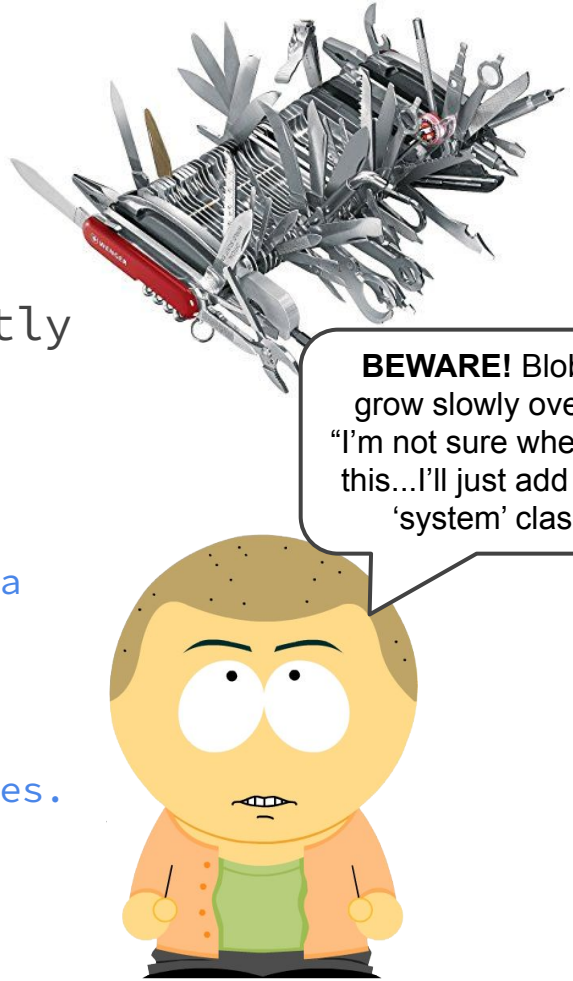
- The ***Single Responsibility Principle*** states that a class should only have a single responsibility.
  - Natch.
  - This is probably *the* most important OO design principle.
- Instead of one class that knows how to draw *any* kind of shape, we should design a different class for each shape.
  - Each class has a *single responsibility*: draw *one* kind of shape.
- This principle allows for a *separation of concerns* among classes.
  - Each class is only concerned with doing one thing and doing it well.



Note that we use leverage abstraction and polymorphism while adhering to single responsibility principle.

Single Responsibility Principle

- A class should have a single, tightly focused responsibility.
- This leads to smaller and simpler classes, but also to more of them.
  - It is easier to understand the scope of a change in a smaller class.
  - It is easier to manage concurrent modifications of smaller classes.
  - Separate concerns go into separate classes.
- This also helps with unit testing!



SOLID

S ingle Responsibility Principle

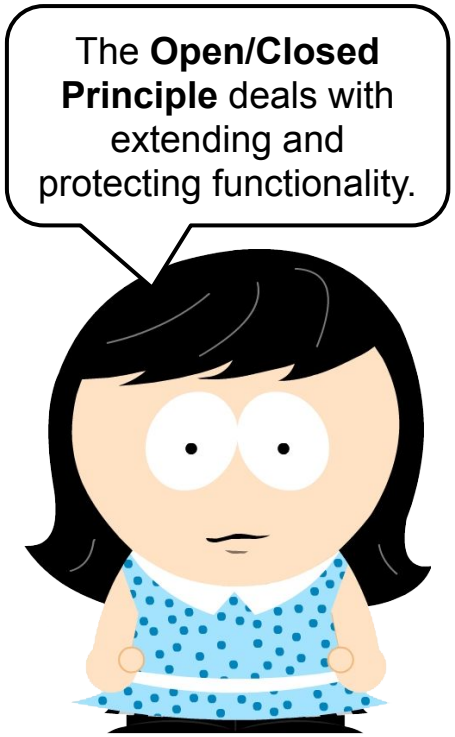
O pen-closed Principle

L iskov Substitution Principle

I nterface Segregation Principle

D ependency Inversion Principle

Open/Closed Principle



"Software entities should be open for extension, but closed to modification."

- Software functionality should be extendable without modifying the base functionality.
  - Mostly provided by features of the implementation language such as interfaces or inheritance.
- Your design should consider appropriate use of
  - Inheritance from abstract classes.
  - Implementation of interfaces.
- Dependency injection provides a mechanism for extending functionality without modification (more on this in a little bit).

SOLID

S ingle Responsibility Principle

O pen-closed Principle

L iskov Substitution Principle

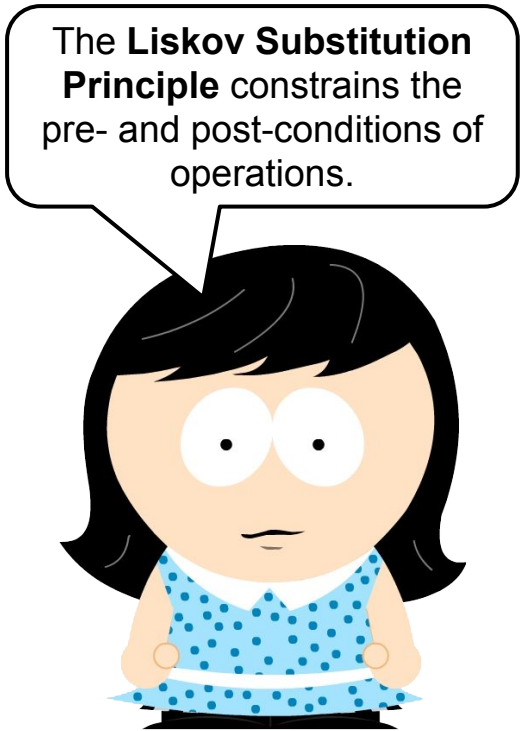
I nterface Segregation Principle

D ependency Inversion Principle

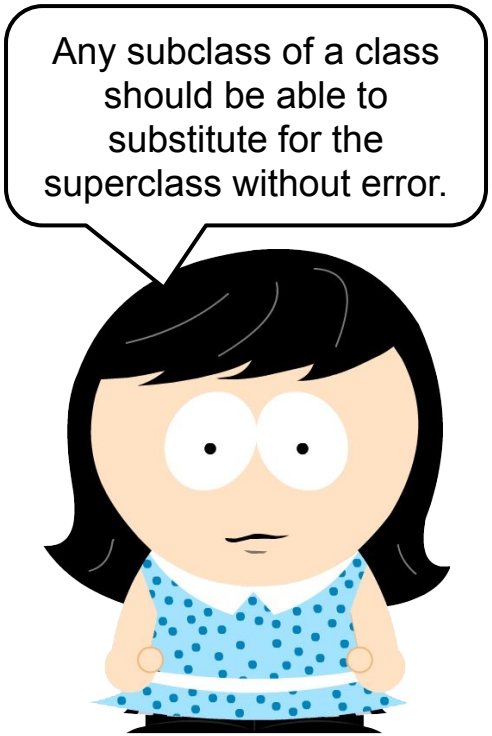
Liskov Substitution Principle

“Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”

- Pre-conditions specify what must be true before a method call.
- Post-conditions specify what will be true after a method call.
- Design by Contract is a programming technique that requires the formal definition of the pre- and post-conditions and has language support for it.



Liskov Substitution Principle



- A subclass must not violate any of the pre- and post-conditions guaranteed by the superclass.
- Superclass clients count on the pre- and post-conditions being true even when polymorphism has the client interacting with a subclass.
- To maintain a pre-condition, a subclass must not narrow the pre-condition, i.e. be a subset.
- To maintain a post-condition, a subclass must not broaden the post-condition, i.e. be a superset.

SOLID

S ingle Responsibility Principle

O pen-closed Principle

L iskov Substitution Principle

I nterface Segregation Principle

D ependency Inversion Principle

Dependency Inversion Principle

- ***Dependency Inversion*** says that a high level module should not depend on a low level model and vice-versa.
  - Both should depend on abstractions.
- This promotes looser coupling between otherwise dependent entities.
- One common manifestation of this is ***dependency injection***.
  - The high level module does not instantiate a low level module on which it depends.
  - An instance of the low level module is created outside and "injected" into the high level module through a constructor or a mutator ("setter").
  - This makes unit testing ***much*** easier.

Designing software systems that are easy to test is a core aspect of software engineering.

One way to do this is to make it possible to test *part* of the system in isolation. By introducing a layer of abstraction (i.e. an interface) between two subsystems we break the direct coupling between them.

This layer of abstraction enables the use of a ***mock***; an artificial implementation of an interface that behaves in a predictable, configurable way.

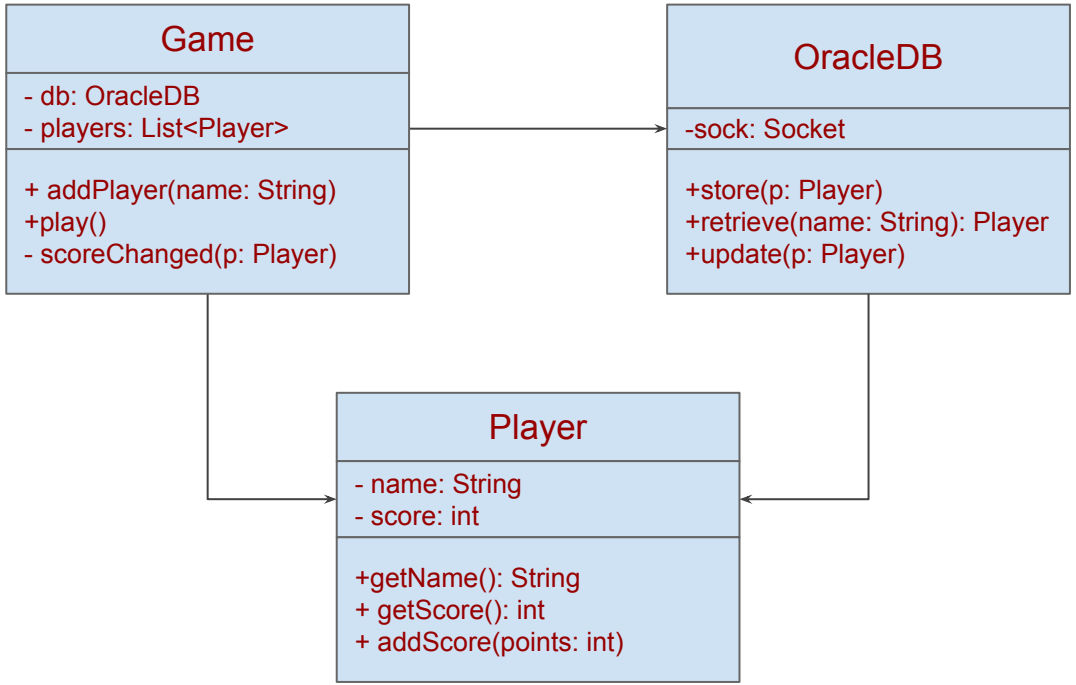
A mock of one subsystem can be used to test another subsystem that depends on the interface being mocked.

A Partial Design

Consider this partial design for some generic **game** that stores **players** in a database.

In this case, the game specifically uses an **Oracle database** to store and update players as their score changes.

Let's take a look at one possible implementation...



Direct Dependency

```
package supergame;

public class Game {

  private OracleDB db;
  private List<Player> players;

  public Game(String host, int port) {
    db = new OracleDB(host, port);
    players = new ArrayList<>();
  }

  public void addPlayer(String name) {
    Player p = db.retrieve(name);
    players.add(p);
  }

  private void scoreChanged(Player p) {
    db.update(p);
  }
}
```

Direct Dependency

```
package supergame;

public class Game {

  private OracleDB db;
  private List<Player> players;

  public Game(String host, int port) {
    db = new OracleDB(host, port);
    players = new ArrayList<>();
  }

  public void addPlayer(String name) {
    Player p = db.retrieve(name);
    players.add(p);
  }

  private void scoreChanged(Player p) {
    db.update(p);
  }
}
```

The `Game` implementation to the left is directly dependent on the concrete `OracleDB`. In fact, it creates an instance of the class in its constructor.

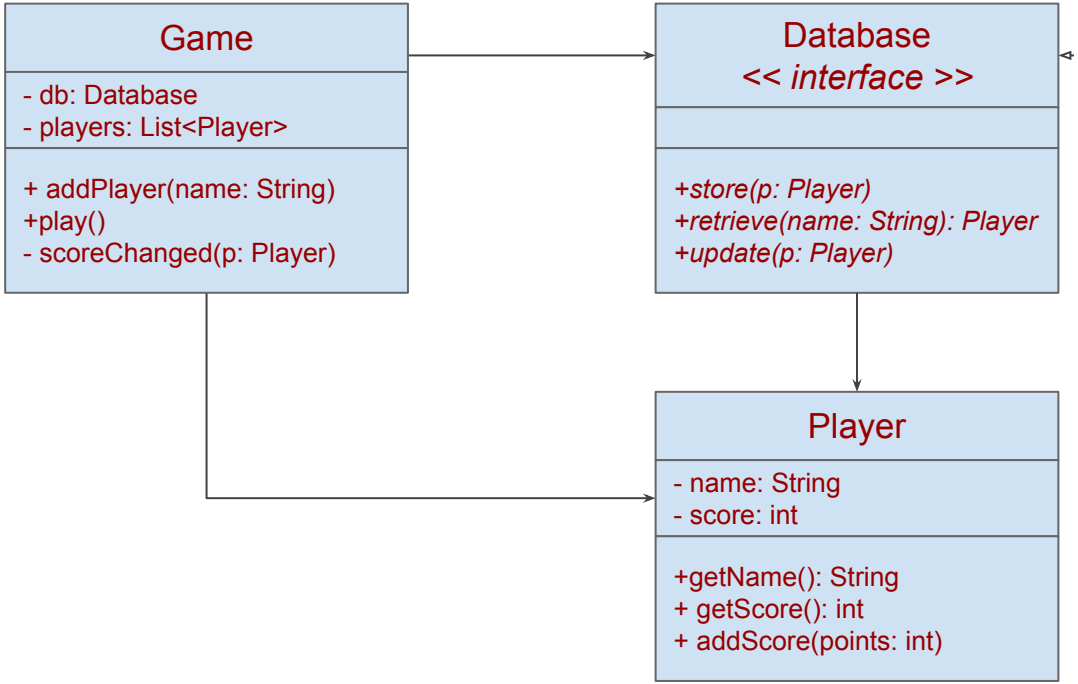
Q: What implications does this have on the testability of the `Game` class?

A: In order to test the class, it must be able to create an instance of the `OracleDB` class. We can infer that this class will attempt to connect to an Oracle database...

This means that we will need to stand up a real Oracle database server in order to create a `Game` and test it (otherwise, we assume that the connection will fail).

Furthermore, the only way to validate that the game is storing, retrieving, and updating players correctly is to look at the data in the database. This all makes testing the `Game` class much harder.

A Layer of Abstraction



Let's modify the design by adding a **layer of abstraction** between the `Game` and `OracleDB` classes, i.e. an **interface** to represent a **generic database** that can store players.

Now the **high level module** (`Game`) does not directly depend on the **low level module** (`OracleDB`). Instead, it depends on an **abstraction**.

Now, let's update the implementation to match the new design...

Direct Dependency II

```
package supergame;

public class Game {

  private Database db;
  private List<Player> players;

  public Game(String host, int port) {
    db = new OracleDB(host, port);
    players = new ArrayList<>();
  }

  public void addPlayer(String name) {
    Player p = db.retrieve(name);
    players.add(p);
  }

  private void scoreChanged(Player p) {
    db.update(p);
  }
}
```

Direct Dependency II

```
package supergame;

public class Game {

  private Database db;
  private List<Player> players;

  public Game(String host, int port) {
    db = new OracleDB(host, port);
    players = new ArrayList<>();
  }

  public void addPlayer(String name) {
    Player p = db.retrieve(name);
    players.add(p);
  }

  private void scoreChanged(Player p) {
    db.update(p);
  }
}
```

First, we update the `Game` class so that the field is of the generic type `Database` rather than the concrete type `OracleDB`.

Q: But the field has to be assigned some value so that we can use it to store, retrieve, and update players. How do we initialize the field?

A: We ***could*** still instantiate an OracleDB in the class, but that would cause a direct dependency between `Game` (**the high level module**) and `OracleDB` (**the low level module**), which is exactly what we are trying to avoid.

(BTW, the UML doesn't show this dependency, but it should - by invoking the constructor on `OracleDB`, we have created a static (compile time) dependency on that class)

How about we use **dependency injection**? Instead of creating the Database directly, we will pass it as an argument to the constructor...

Program to the Interface

```
package supergame;

public class Game {

  private Database db;
  private List<Player> players;

  public Game(Database db) {
    this.db = db;
    players = new ArrayList<>();
  }

  public void addPlayer(String name) {
    Player p = db.retrieve(name);
    players.add(p);
  }

  private void scoreChanged(Player p) {
    db.update(p);
  }
}
```

Program to the Interface

```
package supergame;

public class Game {

  private Database db;
  private List<Player> players;

  public Game(Database db) {
    this.db = db;
    players = new ArrayList<>();
  }

  public void addPlayer(String name) {
    Player p = db.retrieve(name);
    players.add(p);
  }

  private void scoreChanged(Player p) {
    db.update(p);
  }
}
```

Let's modify the constructor to add a `Database` parameter. This way, an instance of an implementing class (e.g. `OracleDB`) can be created outside the class and **injected** in through the constructor.

Creating the `Game` class is not really much more difficult than it was before. Instead creating one like this...

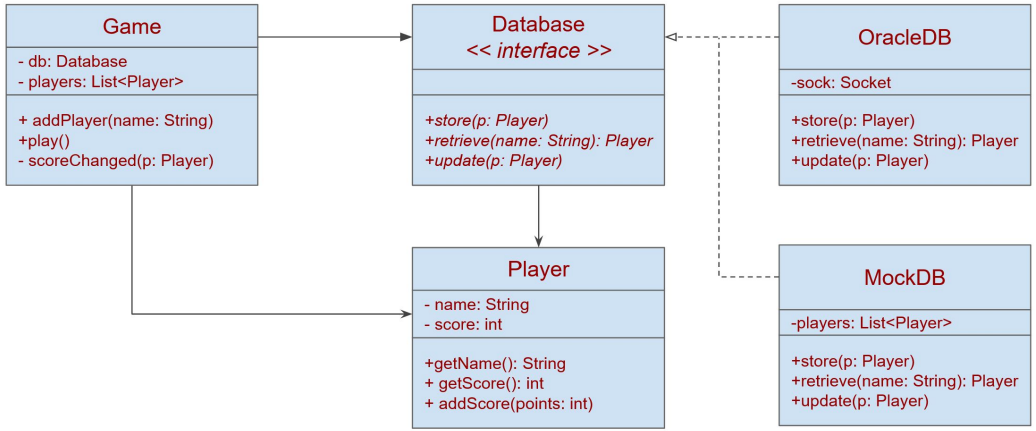
```
Game game = new Game("dbhost", 12357);
```

...we would create one like this...

```
Database db = new OracleDB("dbhost", 12357);
Game game = new Game(db);
```

Q: So what do we gain by doing this? How is the `Game` class more testable than it was before?!

Mock Objects



We have defined the behavior of a generic database as an interface ***and*** we have broken the direct dependency between the high level module (Game) and the low level module (OracleDB).

This means that we are now free to create **alternative implementations** of the `Database` and pass them into the `Game` class instead of the `OracleDB`.

Including, for example, an **artificial** implementation of our Database interface that doesn't use a real database, but just stores players in memory.

We can easily create one such `MockDB`, **inject** it into a `Game` object through the constructor, and verify that the stored players are correct after running tests.

Such artificial objects used only for testing are called **mocks**, and we will talk about them more in the unit testing module.

GRASP

General

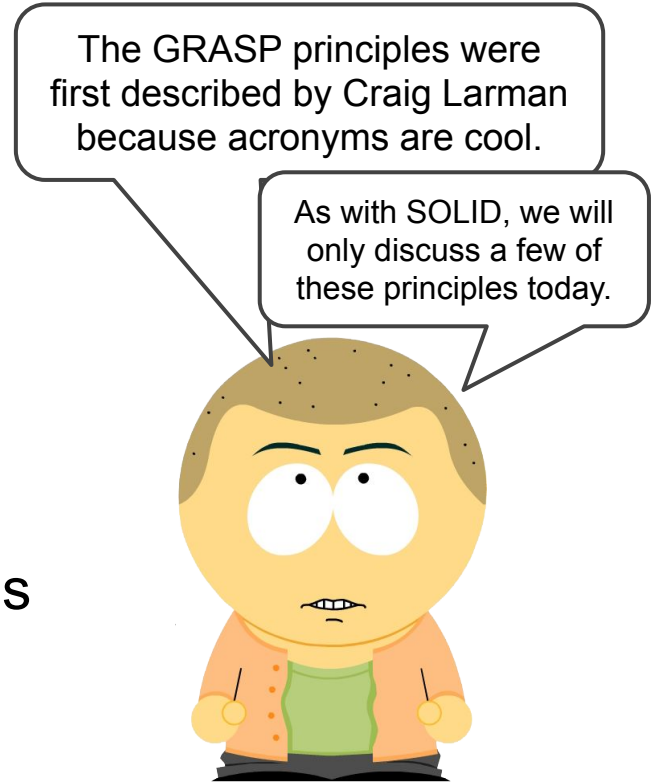
Responsibility

Assignment

Software

Patterns and/or principles

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication



GRASP

General

Responsibility

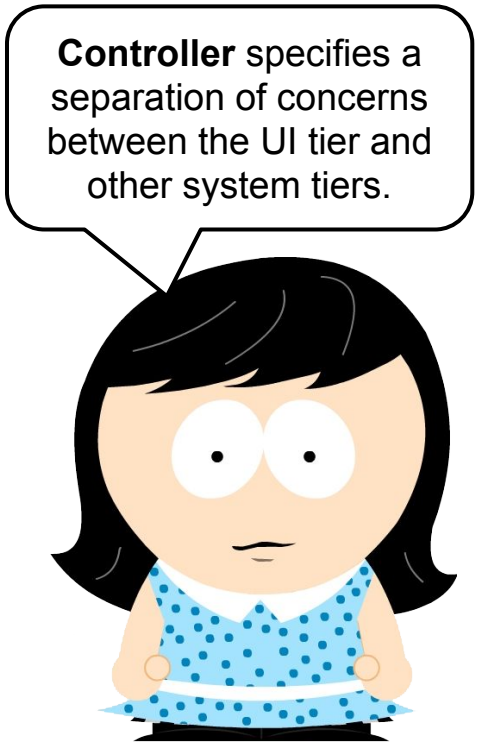
Assignment

Software

Patterns and/or principles

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

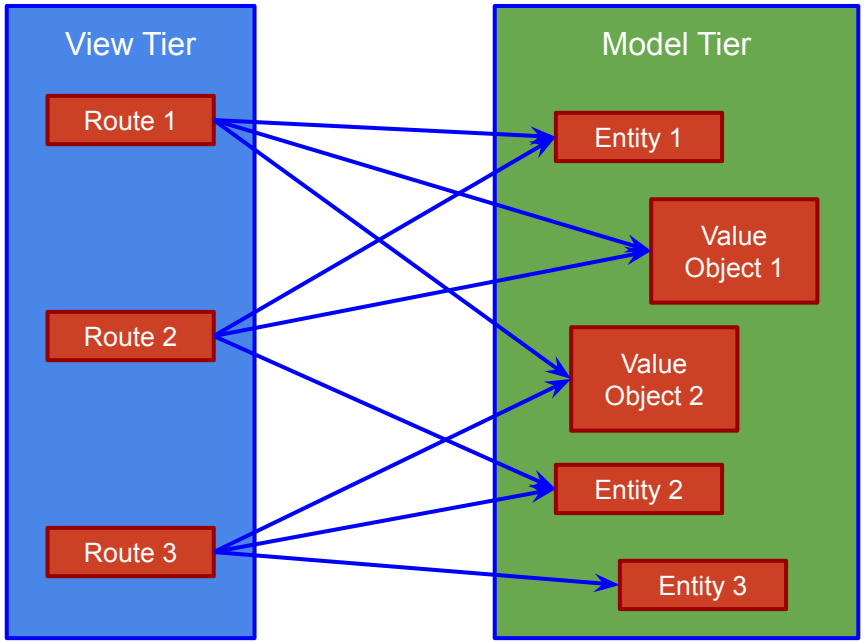
GRASP Controller



“Assign responsibility to receive and coordinate a system operation to a class outside of the UI tier.”

- “Controller” is an overused term in software design.
  - In GRASP, this is not the view “controller” which is firmly in the View tier.
- In simple systems, it may be a single object that coordinates all system operations.
- In more complex systems, it is often multiple objects from different classes, each of which handles a small set of closely related operations.

Without a Controller



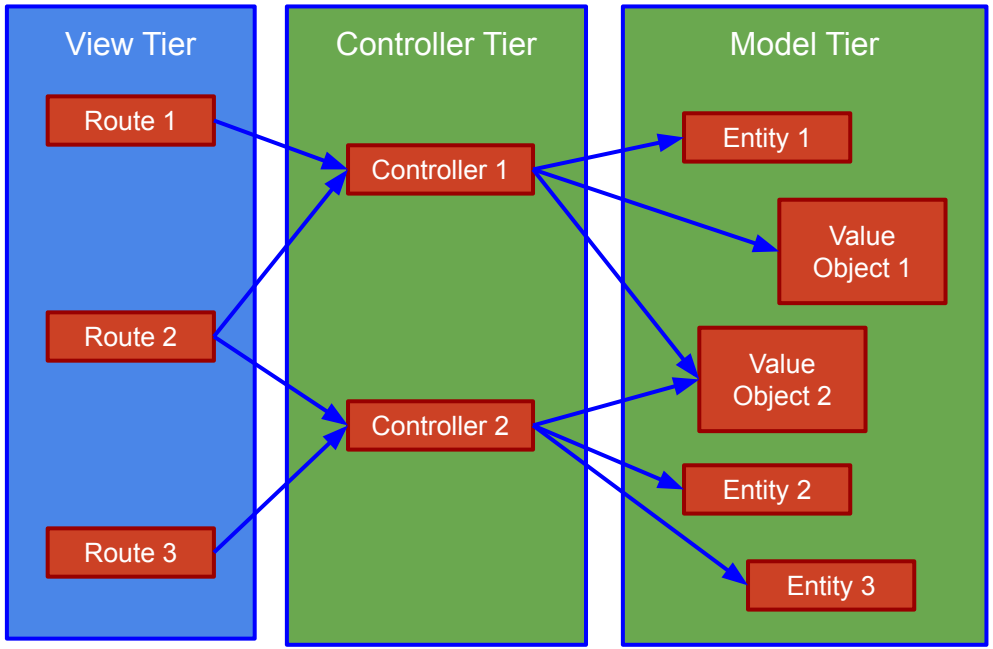
In a system without a controller, the classes in the View tier directly interact with classes in the Model.

This usually manifests as lots of little interactions such as constructor and method calls.

The result is that the View classes become more complex, more tightly coupled with the model, and less focus on UI tasks like receiving input and rendering output.

Consequently, the UI is harder to replace - it requires that complex application logic be rewritten in each new UI (e.g. web, mobile, desktop, etc.).

GRASP Controller



A GRASP Controller is placed in the Controller tier and provides services that encapsulate application logic.

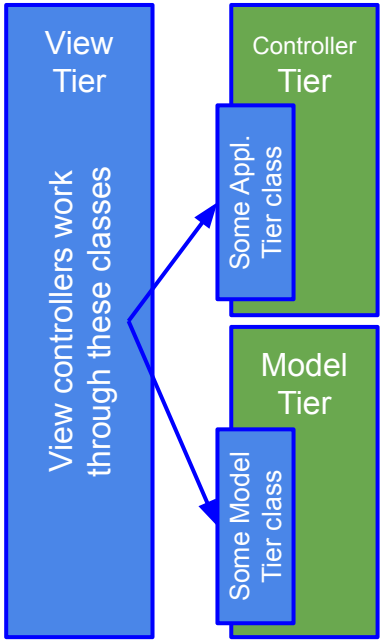
This often has the effect of replacing several method calls with a single call to the service being provided by the controller.

The View tier will probably still need to interact with model classes directly, e.g. to get the name from a player or to check the status of a game.

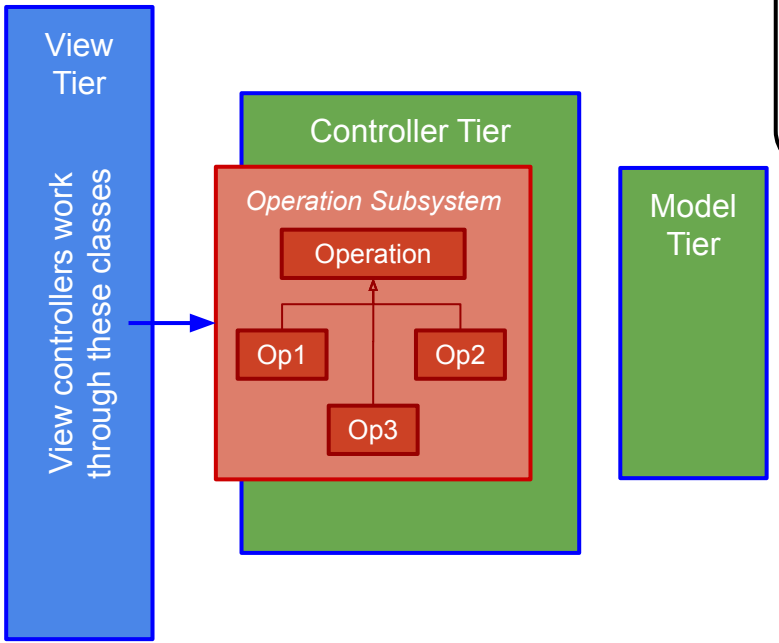
These interactions should be limited to what is necessary for the UI to perform its core function: receiving input from the client and rendering output.

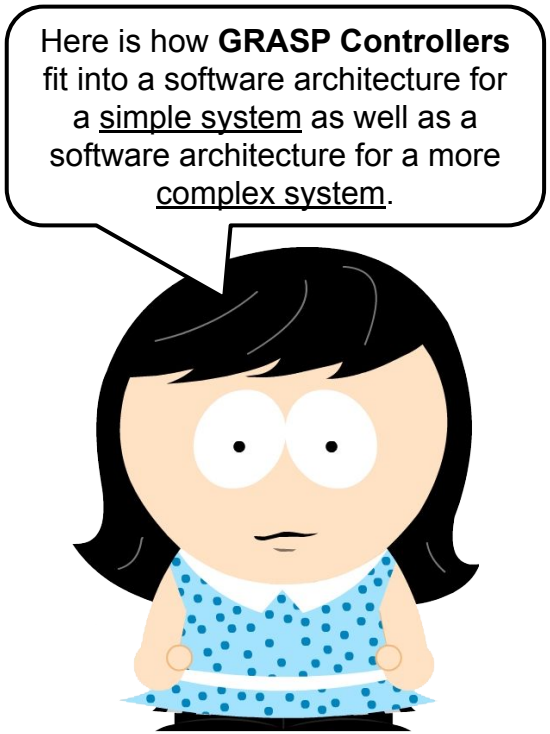
GRASP Controller

Simple System



Complex System





GRASP

General

Responsibility

Assignment

Software

Patterns and/or principles

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

Holdin' Data

Consider the following classes that partially implement a banking application...

```
public class Account {
  private double balance;

  public void deposit(double amt) {
    balance += amt;
  }

  public void withdraw(double amt) {
    balance -= amt;
  }

  public double getBalance() {
    return balance;
  }
}
```

```
public class Customer {
  private String name;
  private int accountNumber;
  private Set<Account> accounts;

  public Set<Account> getAccounts() {
    return accounts;
  }
}
```

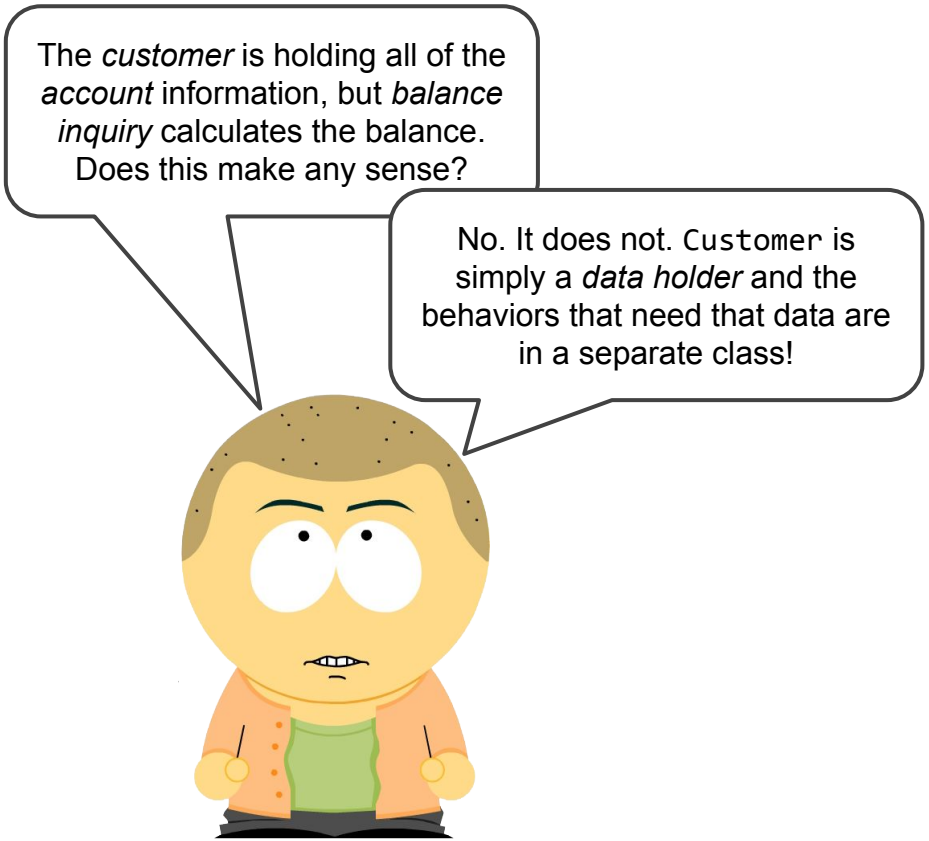
The product owner has requested a new feature in the system that can calculate the total account balance for a customer.

Should we create a new class to handle the new responsibility?

Holdin' Data

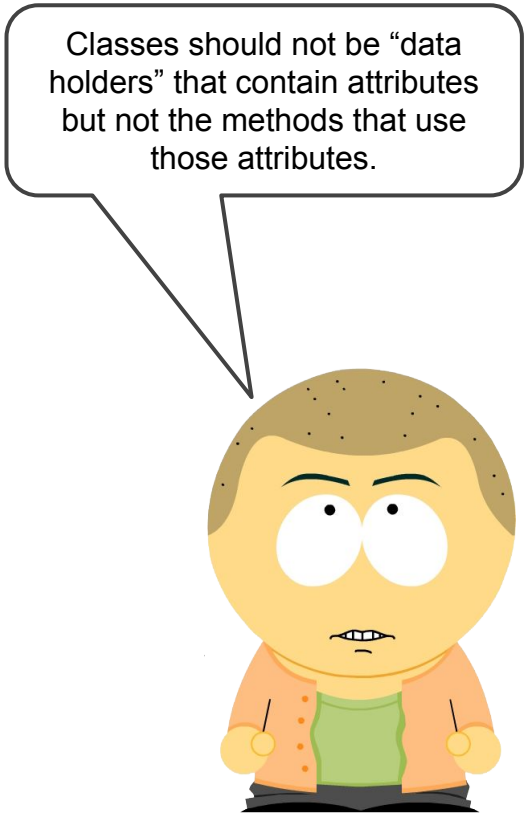
```
public class Customer {
  private String name;
  private int accountNumber;
  private Set<Account> accounts;

  public Set<Account> getAccounts() {
    return accounts;
  }
}
```

```
public class BalanceInquiry {
  public double getTotalBalance(Customer c) {
    double total = 0;

    for(Account account : c.getAccounts()) {
      total += account.getBalance();
    }

    return total;
  }
}
```



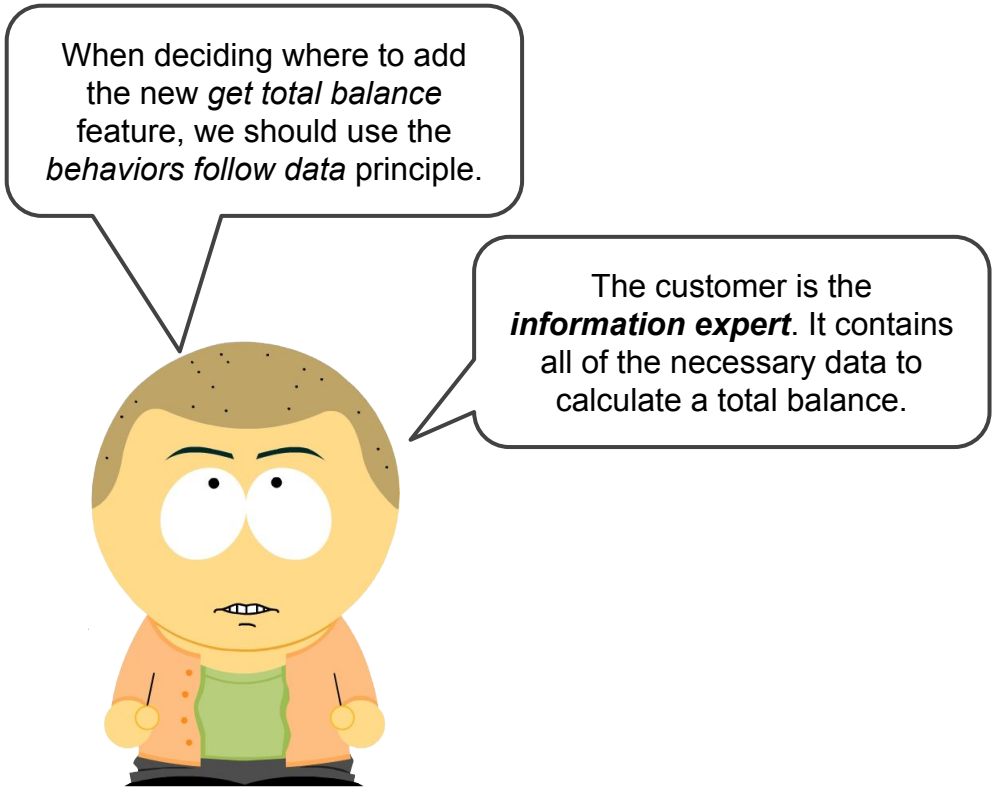
Information Expert

- If a new method has to be added to some class in the system, how do we decide where to add it?
- **Information Expert** states that *“behaviors follow data.”*
  - Assign the responsibility to the class that has the information needed to fulfill the responsibility.
- The first place to consider placing code that uses, processes, or modifies attribute data is the class that holds the attributes.



Information Expert

```
public class Customer {
  private String name;
  private int accountNumber;
  private Set<Account> accounts;

  public Set<Account> getAccounts() {
    return accounts;
  }

  public double getTotalBalance() {
    double total = 0;

    for(Account acct : accounts) {
      total += acct.getBalance();
    }

    return total;
  }
}
```



GRASP

General

Responsibility

Assignment

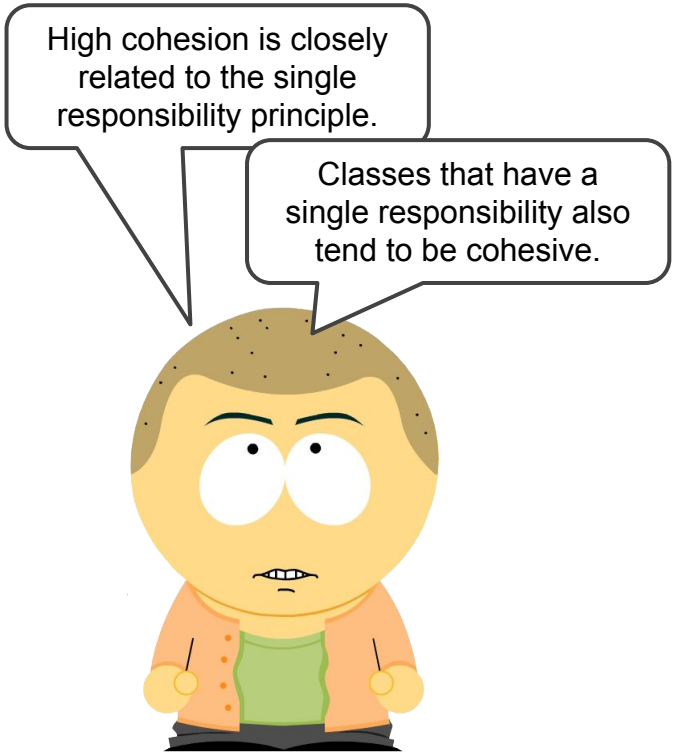
Software

Patterns and/or principles

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

High Cohesion

- **High Cohesion** aims for focused, understandable, and manageable classes.
  - *Responsibilities should be assigned so that the cohesion of individual classes remains high.*
- High cohesion leads to smaller classes with more narrowly defined responsibilities.
- This design goal should have a higher priority than most other design goals.
  - Yes, sometimes design principles are in *competition* with each other, in which cases, we choose the higher priority principle.



GRASP

General

Responsibility

Assignment

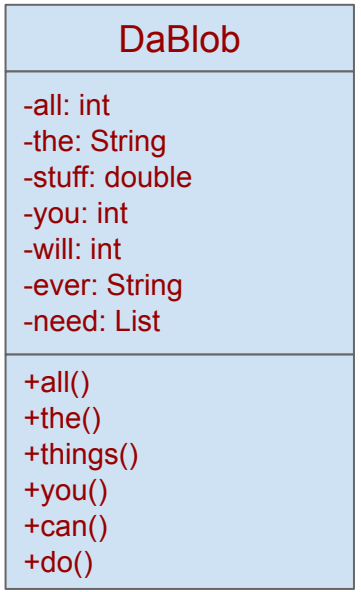
Software

Patterns and/or principles

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

Low Coupling

- **Low coupling** attempts to minimize the impact of changes to a system.
  - Assign responsibilities to a class so that *unnecessary* coupling remains low.
  - Note the keyword “unnecessary;” coupling *is* needed in the system.
- Resist lowering coupling simply to reduce the number of relationships.
  - A design with more relationships is often better than a design with fewer.
  - You need to sometimes balance competing design principles against each other.
  - Beginning designers often want to prioritize low coupling. This usually results in fewer, larger, less cohesive classes with multiple responsibilities.

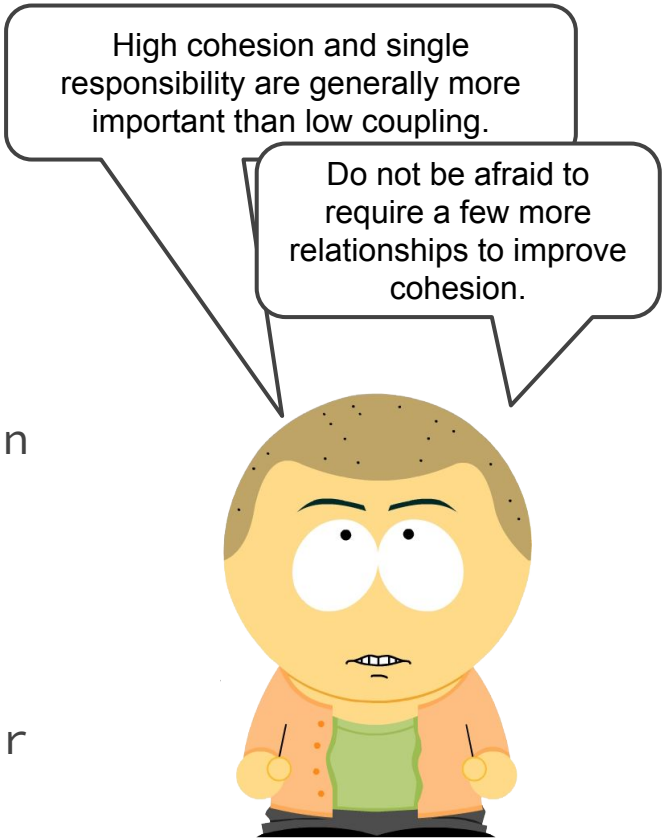


The ultimate form of low coupling is ***no*** coupling: one big class that does everything.

It is hopefully obvious at this point that this is *not* a good design choice.

High Cohesion vs. Low Coupling

- Low Coupling states that we should try to minimize the relationships, and thus *dependencies* between classes.
  - This sometimes results in a system with fewer, larger classes.
- High cohesion and low coupling are often in competition with each other.
  - High cohesion means more and smaller classes.
  - More classes means more relationships and/or dependencies.
- In general, we prefer *high cohesion* over *low coupling*.



GRASP

General

Responsibility

Assignment

Software

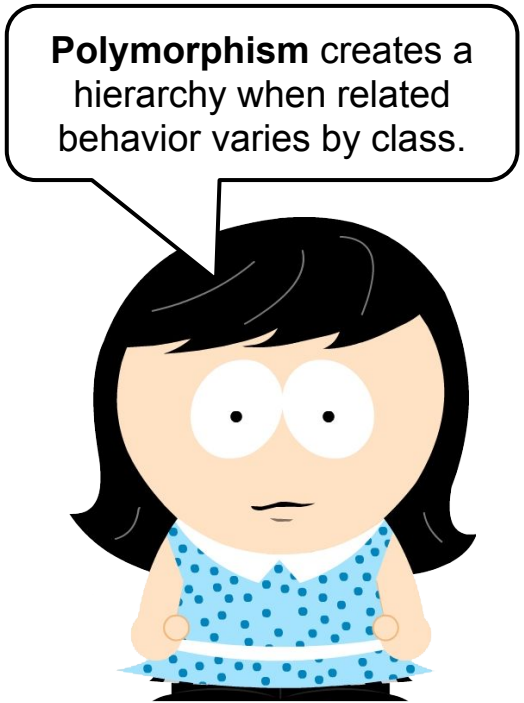
Patterns and/or principles

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

Polymorphism

“Assign responsibility for related behavior that varies by class by using polymorphic behavior.”

- Polymorphism is a primary object-oriented concept and should be used whenever possible.
- There are bad code smells that indicate that polymorphism is not being used effectively.
  - A conditional that selects behavior based on a “type” attribute (numeric code, enum).
  - Use of instanceof or similar language constructs to select operations to perform.



GRASP

General

Responsibility

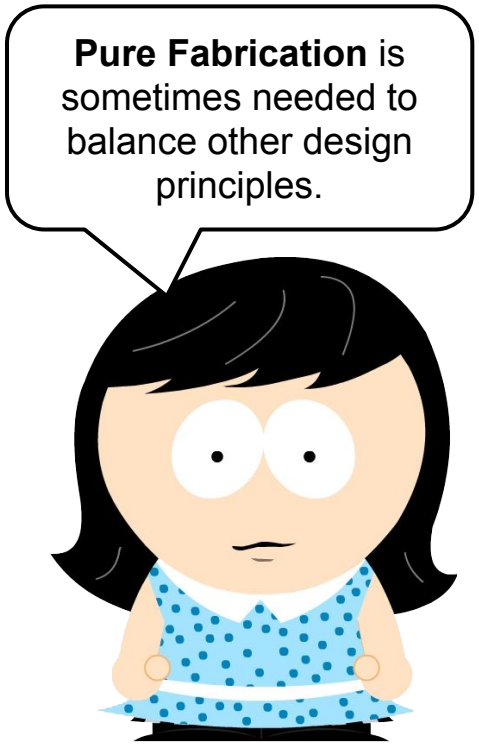
Aassignment

Software

Ppatterns and/or principles

- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

Pure Fabrication



"Assign a cohesive set of responsibilities to a non-domain entity in order to support high cohesion and low coupling."

- Your design should be primarily driven by the problem domain.
  - Classes should be assigned names, attributes, and methods that draw from the domain language.
- However, to maintain a cohesive design, you may need to create classes that are not domain entities.
- In the GRASP controller example, the Operation Subsystem was a pure fabrication.

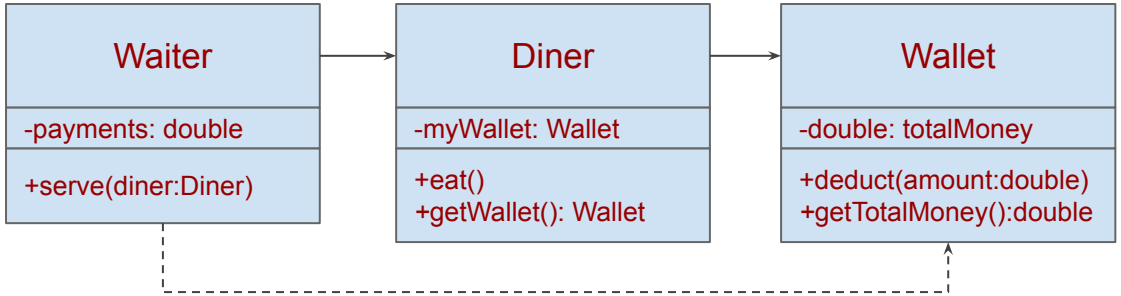
Talking to Strangers

We are designing a simple system to simulate diners at a restaurant. The `Wallet` class is used to pay for dinner at the end of a meal.

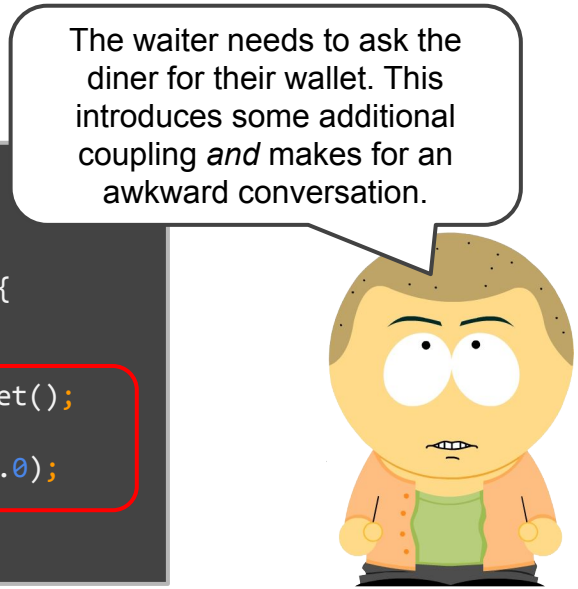
The `Diner` class represents customers at the restaurant. `Diners` have a `Wallet` that they use to pay for their meals, and thus the `Diner` class is coupled with the `Wallet` class.

The `Waiter` class represents an employee serving a meal to a `Diner`, and thus is necessarily coupled with the `Diner`.

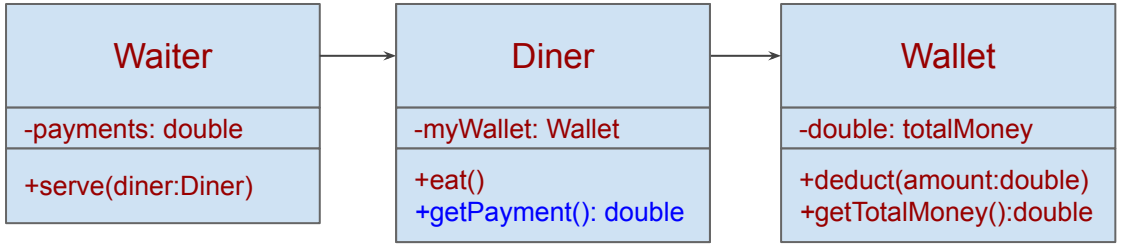
Q: At the end of the meal, the `Waiter` must obtain payment for the meal. Given the current system design, how would this work?




```
public class Waiter {
  private double payments = 0;

  public void serve(Diner diner) {
    diner.eat();

    Wallet wallet = diner.getWallet();

    payments += wallet.deduct(100.0);
  }
}
```



Only Talk to Friends



Q: Does it make sense for `Waiter` to ask the `Diner` to turn over his (private) `Wallet`? Or to depend on `Wallet` at all? How does this usually work in the real world?

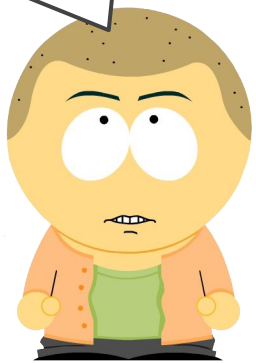
A: The `Waiter` asks the `Diner` for money to pay for the meal. The `Diner` removes the money from the `Wallet` and gives it to the `Waiter`. The `Waiter` never touches the `Wallet`.

Q: Can we fix the system to work that way and eliminate coupling while also protecting access to the `Wallet`?

A: Yes! By adding a method to `Diner` that allows the `Waiter` to ask for payment without requesting the `Wallet`.

```
public class Waiter {
  private double payments = 0;

  public void serve(Diner diner) {
    diner.eat();

    payments += diner.getPayment();
  }
}
```

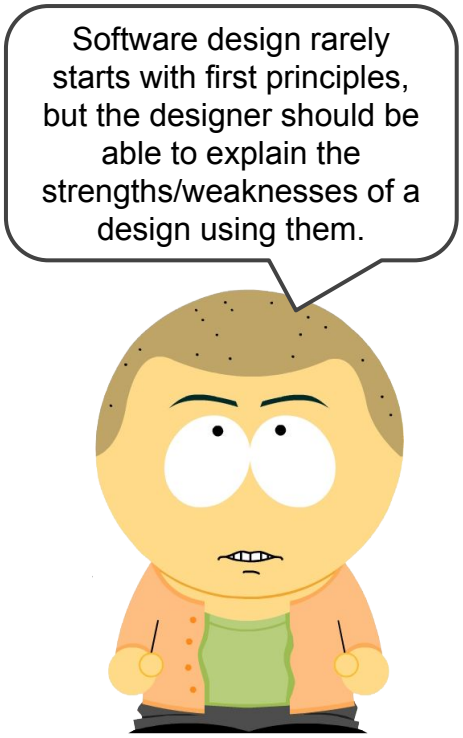
This change results in lower coupling, simpler code, *and* better encapsulation of the diner's wallet. A win-win-win.



The Law of Demeter

- **The Law of Demeter** addresses unintended coupling within a software system.
- Limit the range of classes that a class talks to.
  - Each class only talks to its friends; don't talk to strangers.
  - Each class only talks to its immediate friends; don't talk to friends of friends.
  - Chained access exposes each interface (i.e. the Wallet is exposed to the Waiter)!
- If a class needs to talk to something "far away", do not chain method calls together.
  - Get support from your friends, e.g. getPayment()
  - Get a new friend; establish a direct relationship.

Design Principles



There are some key object-oriented *“first principles”* that will be stressed in SWEN 262:

- Increase cohesion where possible
- Decrease coupling where possible
- Behaviors follow data (Information Expert)
- Prefer type (interface) inheritance over class (implementation) inheritance.
  - Program to the interface, not the implementation
- Prefer composition to inheritance
  - “has-a” relationships rather than “is-a” relationships
- Use delegation to “simulate” runtime inheritance
- Law of Demeter: “Only talk to your friends.”

Design Principles

There are many more object-oriented design concepts:

- **Abstraction**
  - Provide well-defined, conceptual boundaries that focus on the outside view of an object and so serves to separate an object's essential behavior from its implementation.
- **Principle of Least Commitment**
  - The interface of an object provides its essential behavior, and nothing more.
- **Principle of Least Astonishment**
  - An abstraction captures the entire behavior of an object and offers no surprises or side effects that go beyond the scope of the abstraction.
- **Open-Closed Principle (OCP)**
  - Software entities (classes, modules, etc.) should be ***open*** for extension, but ***closed*** for modification.
  - We should design modules that never need to change.
  - To extend the behavior of a system, we add new code. We do not modify old code.