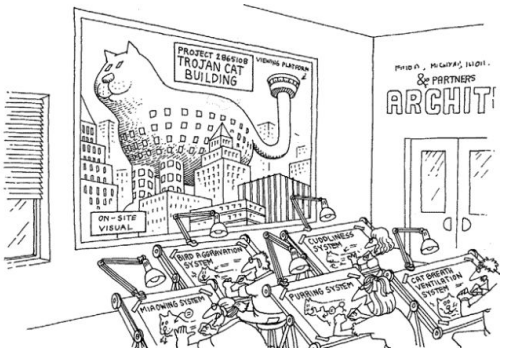
Software Design::OOP Basics



Review of Object Orientation

What is Object Orientation?

Procedural paradigm:

- Software is organized around the notion of *procedures*
- *Procedural abstraction*
  - — Works as long as the data is simple
- *Adding data abstractions*
  - — Groups together the pieces of data that describe some entity
  - — Helps reduce the system's complexity.
    - Such as *Records* and *structures*

Object oriented paradigm:

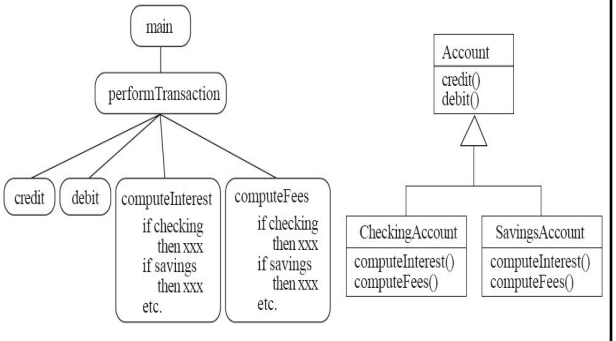
- Organizing procedural abstractions in the context of data abstractions

Object Oriented paradigm

An approach to the solution of problems in which all computations are performed in the context of objects.

- The objects are instances of classes, which:
  - — are data abstractions
  - — contain procedural abstractions that operate on the objects
- A running program can be seen as a collection of objects collaborating to perform a given task

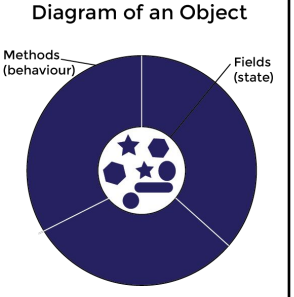
A View of the Two paradigms



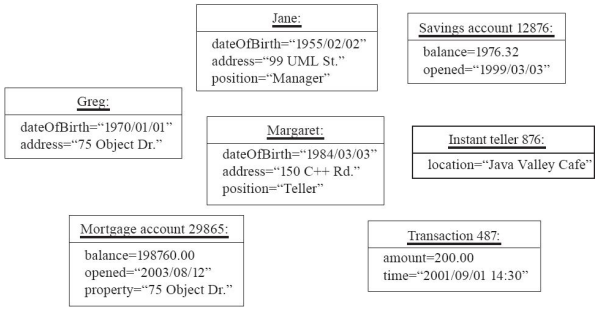
Classes and Objects

Object

- A chunk of structured data in a running software system
- Has *properties*
  - — Represent its state
- Has *behaviour*
  - — How it acts and reacts
  - — May simulate the behaviour of an object in the real world



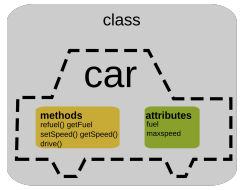
Objects



Classes

A class:

- A unit of abstraction in an object oriented (OO) program
- Represents similar objects
  - — Its *instances*
- A kind of software module
  - — Describes its instances' structure (properties)
  - — Contains *methods* to implement their behaviour



Is Something a Class or an Instance?

- Something should be a *class* if it could have instances
- Something should be an *instance* if it is clearly a *single* member of the set defined by a class

Film

- Class; instances are individual films.

***Reel of Film*:**

- Class; instances are physical reels

Film reel with serial number SW19876

- Instance of **`ReelOfFilm`**

Science Fiction

- Instance of the class **`Genre`**.

Science Fiction Film

- Class; instances include 'Star Wars'

***Showing of 'Star Wars' in the Phoenix Cinema at 7 p.m.*:**

- Instance of **`ShowingOfFilm`**

Naming classes

- Use *capital* letters
  - — E.g. `BankAccount` not `bankAccount`
- Use *singular* nouns
- Use the right level of generality
  - — E.g. `Municipality`, not `City`
- Make sure the name has only *one* meaning
  - — E.g. 'bus' has several meanings

Instance Variables

Variables defined inside a class corresponding to data present in each instance

- Also called *fields* or *member variables*
- Attributes
  - — Simple data
  - — E.g. `name`, `dateOfBirth`
- Associations
  - — Relationships to other important classes
  - — E.g. `supervisor`, `coursesTaken`
  - — More on these in Chapter 5

Variables vs. Objects

A variable

- *Refers* to an object
- May refer to different objects at different points in time

An object can be referred to by several different variables at the same time

***Type* of a variable**

- Determines what classes of objects it may contain

Class variables

A *class variable's* value is *shared* by all instances of a class.

- Also called a *static* variable
- If one instance sets the value of a class variable, then all the other instances see the same changed value.
- Class variables are useful for:
  - — Default or 'constant' values (e.g. PI)
  - — Lookup tables and similar structures

Caution: *do not over-use class variables*

Methods, Operations and Polymorphism

Operation

- A higher-level procedural abstraction that specifies a type of behaviour
- Independent of any code which implements that behaviour
  - — E.g. calculating area (in general)

Methods, Operations and Polymorphism

Method

- A procedural abstraction used to implement the behaviour of a class
- Several different classes can have methods with the same name
  - — They implement the same abstract operation in ways suitable to each class
  - — E.g. calculating area in a rectangle is done differently from in a circle

Polymorphism

A property of object oriented software by which an *abstract operation may be performed in different ways* in different classes.

- Requires that there be *multiple methods of the same name*
- The choice of which one to execute depends on the object that is in a variable
- Reduces the need for programmers to code many `if-else` or `switch` statements

Organizing Classes into Inheritance Hierarchies

Superclasses

- Contain features common to a set of subclasses

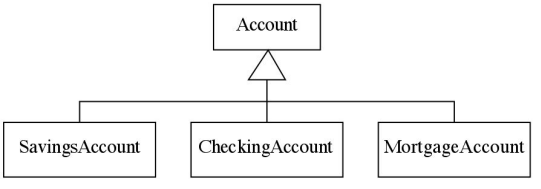
Inheritance hierarchies

- Show the relationships among superclasses and subclasses
- A triangle shows a *generalization* △

Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

An Example Inheritance Hierarchy



Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

The Isa Rule

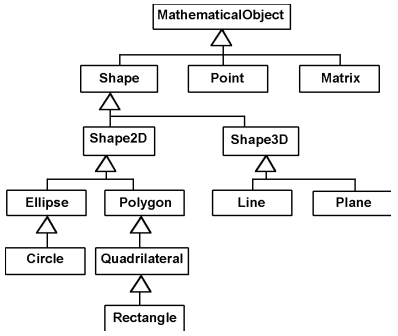
Always check generalizations to ensure they obey the isa rule

- "A checking account ***is an*** account"
- "A village ***is a*** municipality"

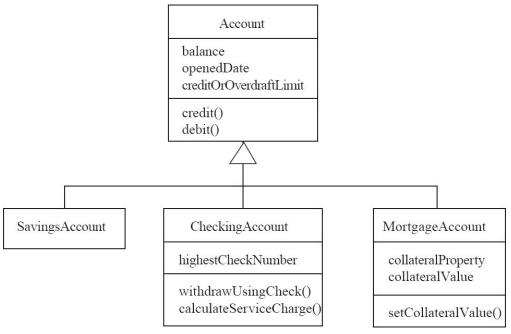
Should 'Province' be a subclass of 'Country'?

- No, it violates the isa rule
  - — "A province ***is a*** country" is invalid!

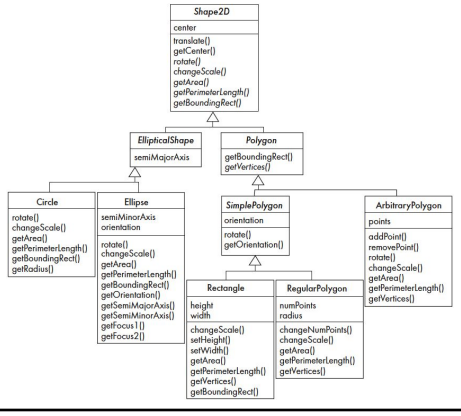
A possible inheritance hierarchy of mathematical objects



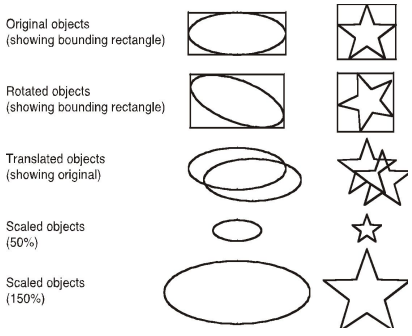
Inherited Features Should Make Sense in Subclasses



Inheritance, Polymorphism and Variables



Some Operations in the Shape Example



Abstract Classes and Methods

An operation should be declared to exist at the highest class in the hierarchy where it makes sense

- The *operation* may be *abstract* (lacking implementation) at that level
- If so, the *class* also must be *abstract*
  - — No instances can be created
  - — The opposite of an abstract class is a *concrete* class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - — Leaf classes must have or inherit concrete methods for all operations
  - — Leaf classes must be concrete

Overriding

A method would be inherited, but a subclass contains a new version instead

- For restriction
  - E.g. `scale(x,y)` would not work in `Circle`
- For extension
  - E.g. `SavingsAccount` might charge an extra fee following every debit
- For optimization
  - E.g. The `getPerimeter Length` method in `Circle` is much simpler than the one in `Ellipse`

How a decision is made about which method to run

1. **If there is a concrete method for the operation in the current class, run that method.**
2. **Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.**
3. **Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.**
4. **If no method is found, then there is an error**
   - In Java and C++ the program would not have compiled

Dynamic binding

Occurs when decision about which method to run can only be made at *run time*

- Needed when:
  - A variable is declared to have a superclass as its type, and
  - There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses
- Searches for method using the *class* of the current *object*

Concepts that Define Object Orientation

The following are necessary for a system or language to be OO

- Identity
  - Each object is *distinct* from each other object, and *can be referred to*
  - Two objects are distinct *even if they have the same data*
- Classes
  - The code is organized using classes, each of which describes a set of objects
- Inheritance
  - The mechanism where features in a hierarchy inherit from superclasses to subclasses
- Polymorphism
  - The mechanism by which several methods can have the same name and implement the same abstract operation.

Other Key Concepts

Abstraction

- Object -> something in the world
- Class -> objects
- Superclass -> subclasses
- Operation -> methods
- Attributes and associations -> instance variables

Modularity

- Code can be constructed entirely of classes

Encapsulation

- Details can be hidden in classes
- This gives rise to *information hiding*:
  - Programmers do not need to know all the details of a class

The Basics of Java

History

- The first object oriented programming language was Simula-67
  - designed to allow programmers to write simulation programs
- In the early 1980's, Smalltalk was developed at Xerox PARC
  - New syntax, large open-source library of reusable code, bytecode, platform independence, garbage collection.
- late 1980's, C++ was developed by B. Stroustrup,
  - Recognized the advantages of OO but also recognized that there were tremendous numbers of C programmers
- In 1991, engineers at Sun Microsystems started a project to design a language that could be used in consumer 'smart devices': Oak
  - When the Internet gained popularity, Sun saw an opportunity to exploit the technology.
  - The new language, renamed Java, was formally presented in 1995 at the SunWorld '95 conference.

Java documentation

Looking up classes and methods is an essential skill

- Looking up unknown classes and methods will get you a long way towards understanding code

Java documentation can be automatically generated by a program called Javadoc

- Documentation is generated from the code and its comments
- You should format your comments as shown in some of the book's examples
  - — These may include embedded html

Overview of Java

The next few slides will remind you of several key Java features

- Not in the book
- See the book's web site for
  - — A more detailed overview of Java
  - — Pointers to tutorials, books etc.

Characters and Strings

`Character` **is a class representing Unicode characters**

- More than a byte each
- Represent any world language

`char` **is a primitive data type containing a Unicode character**

`String` **is a class containing collections of characters**

- + is the operator used to concatenate strings

Arrays and Collections

Arrays are of fixed size and lack methods to manipulate them

`ArrayList` **is the most widely used class to hold a *collection* of other objects**

- More powerful than arrays, but less efficient

`Iterators` **are used to access members of** `Vectors`

- Enumerations were formally used, but were more complex

```
a = new ArrayList();
Iterator i = a.iterator();
while(i.hasNext())
{
  aMethod(i.next());
}
```

Modern Java has other mechanisms for accessing elements of a collection

```
books  = new ArrayList<String>();
for(String book:books){
      ...
}
```

Casting

Java is very strict about types

- If variable v is declared to have type X, you can only invoke operations on v that are defined in X or its superclasses
  - — Even though an instance of a *subclass* of X may be actually stored in the variable
- If you *know* an instance of a subclass is stored, then you can *cast* the variable to the subclass
  - — E.g. if I know a `Vector` contains instances of `String`, I can get the next element of its `Iterator` using:

```
(String)i.next();
```

  - — To avoid casting you could also have used templates::

```
a = ArrayList<String>; i=a.iterator();
i.next()
```

using

Exceptions

Anything that can go wrong should result in the raising of an Exception

- **Exception** is a class with many subclasses for specific things that can go wrong

Use a try - catch block to trap an exception

```
try
{
  // some code
}
catch (ArithmeticException e)
{
  // code to handle division by zero
}
```

Interfaces

Like abstract classes, but cannot have executable statements
- Define a set of operations that make sense in several classes
- Abstract Data Types

A class can implement any number of interfaces
- It must have concrete methods for the operations

You can declare the type of a variable to be an interface
- This is just like declaring the type to be an abstract class

Important interfaces in Java's library include
- Runnable, Collection, Iterator, Comparable, Cloneable

Packages and importing

A package combines related classes into subsystems
- All the classes in a particular directory

Classes in different packages can have the same name
- Although not recommended

***Importing* a package is done as follows:**
import finance.banking.accounts.*;

Access control

Applies to methods and variables
- `public`
  — Any class can access
- `protected`
  — Only code in the package, or subclasses can access
- (blank) (default is called *package protected*)
  — Only code in the package can access
- `private`
  — Only code written in the class can access
  — Inheritance still occurs!

Threads and concurrency

Thread:
- Sequence of executing statements that can be running concurrently with other threads

To create a thread in Java:
- 1. Create a class implementing `Runnable` or extending `Thread`
- 2. Implement the `run` method as a loop that does something for a period of time
- 3. Create an instance of this class
- 4. Invoke the `start` operation, which calls `run`

Programming Style Guidelines

Remember that programs are for people to read
- Always choose the simpler alternative
- Reject clever code that is hard to understand
- Shorter code is not necessarily better

Choose good names
- Make them highly descriptive
- Do not worry about using long names

Programming style …

Comment extensively
- Comment whatever is non-obvious
- Do not comment the obvious
- Comments should be 25-50% of the code

Organize class elements consistently
- Variables, constructors, public methods then private methods

Be consistent regarding layout of code

Programming style …

Avoid duplication of code

- Do not 'clone' if possible
  - — Create a new method and call it
  - — Cloning results in two copies that may both have bugs
    - When one copy of the bug is fixed, the other may be forgotten

Programming style ...

Adhere to good object oriented principles

- E.g. the 'isa rule'

Prefer `private` as opposed to `public`

Do not mix user interface code with non-user interface code

- Interact with the user in separate classes
  - — This makes non-UI classes more reusable

Difficulties and Risks in Programming

Language evolution and deprecated features:

- Java is evolving, so some features are 'deprecated' at every release
- But the same thing is true of most other languages

Efficiency can be a concern in some object oriented systems

- Java can be less efficient than other languages
  - — VM-based
  - — Dynamic binding

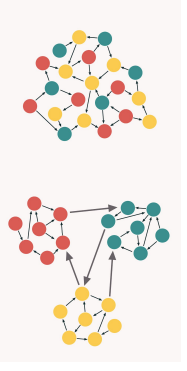
Core Design Principles and Heuristics

Modularity

- The goal of design is to partition the system into modules and assign responsibility among the components in a way that:
  - High *cohesion* within modules, and
  - Loose *coupling* between modules
- Modularity reduces the total complexity a programmer has to deal with at any one time assuming:
  1. Functions are assigned to modules in away that groups similar functions together (Separation of Concerns), and
  2. There are small, simple, well-defined interfaces between modules (information hiding)
- The principles of cohesion and coupling are probably the most important design principles for evaluating the effectiveness of a design.

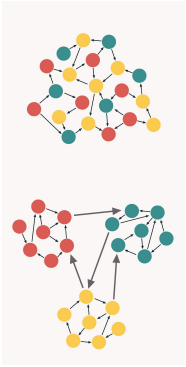
Coupling

- Coupling
  - measure of dependency between modules.
  - A dependency exists between two modules if a change in one could require a change in the other.
- Degree of coupling determined by:
  - The number of interfaces between modules
    - (quantity)
  - Complexity of each interface
    - (determined by the type of communication) (quality)



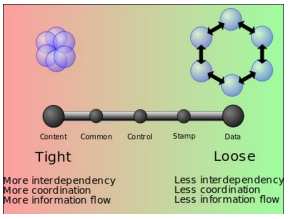
Cohesion

- A measure of how strongly related the functions or responsibilities of a module are.
- A module has high cohesion if all of its elements are working towards the same goal.



Types of Coupling

- Content coupling
  - *also known as Pathological coupling*
  - One module directly references the contents of another
- Common coupling
  - Two or more modules connected via global data.
- Control coupling
  - One module determines the control flow path of another.

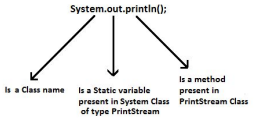


`e.g. public void print(int miles, bool displayMetric)`

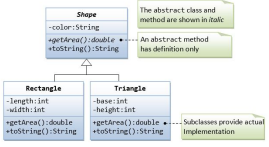
- Stamp coupling
  - Passing a composite data structure to a module that uses only part of it.
- Data coupling
  - Modules that share data through parameters.

Coupling in OOP

- Interaction Coupling
  - Functions using other functions across classes
- Component Coupling
  - Classes have variables of other classes (association)
- Inheritance Coupling
  - Classes inherit from other classes (inheritance)

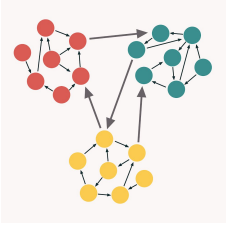



```
class SubHunter {
    Grid myGrid;
    …
}
```



Types of Cohesion

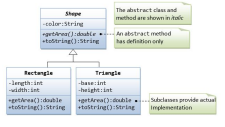
- Coincidental
  - Just got lucky- e.g. Utility classes
- Logical
  - e.g. all inputs or all outputs
- Temporal
  - Init, cleanup
- Procedural
  - i.e., from flowcharts
- Communicational
  - We use the same data



OOP Cohesion

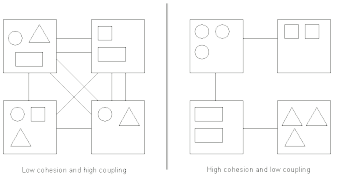
- Method Cohesion
  - Same as cohesion in non OOP programming. Are all elements of this method focused on the task that the method is solving?
- Class Cohesion
  - Why are the methods and attributes in this class, are they cohesive?
- Inheritance Cohesion
  - Why are classes together in a hierarchy?

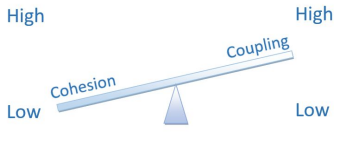
```
public class SubHunter extends Activity
void draw(){
    gameView.setImageBitmap(blankBitmap)
    canvas.drawColor(Color.argb(255,255,

    for(int i=0; i< gridWidth; i++)
            canvas.drawLine(   blockSize
                               numberVer

    for(int i=0; i< gridHeight; i++)
        canvas.drawLine(0,  blockSize * i
                            numberHorizon

    // draw player shot
    canvas.drawRect(horizontalTouched *
                    (horizontalTouched +
                    (verticalTouched - b

    paint.setTextSize(blockSize);
    paint.setColor(Color.argb(255,0,0,25
    canvas.drawText("Shots Taken: " + sh
                    "   Distance: " + d
                    blockSize, blockSize

    Log.d("Debugging", "In draw");
    if(debugging)
        printDebuggingText();
```



Coupling and Cohesion

- Good designs have high cohesion (also called strong cohesion) within a module and low coupling (also called weak coupling) between modules.
- Coupling and Cohesion Tend to be Inversely Correlated
- Results in...
  - Modules are easier to read and understand.
  - Modules are easier to modify.
  - There is an increased potential for reuse
  - Modules are easier to develop and test.





Abstraction

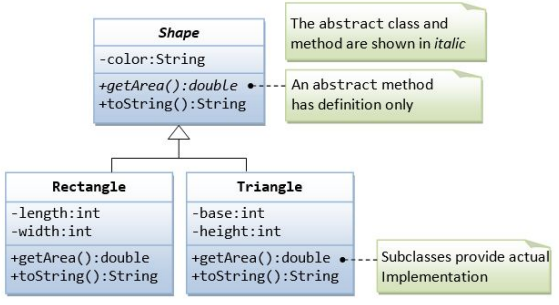
- A generalization of something too complex to be dealt with in its entirety
- For humans not computers
  - it is a technique we use to compensate for the relatively puny capacity of our brains
  - There aren't enough neurons (or connections) in our brain to process the rich detail around us during a single moment in time
- Successful designers develop abstractions and hierarchies of abstractions for complex entities and move up and down this hierarchy with splendid ease

Form Consistent Abstractions

- "Abstraction is the ability to engage with a concept while safely ignoring some of its details."
- Base classes and interfaces are abstractions.
  - i.e UIComponent (any GUI toolkit),
  - Mammal (classic superclass when discussing OO design)
  - The interface defined by a class is an abstraction of what the class represents
- A procedure defines an abstraction of some operation.
  - `someObject.toString()`

Form Consistent Abstractions

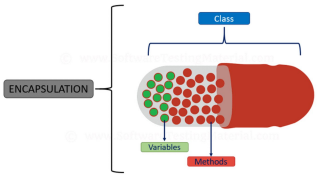
Base classes and interfaces are abstractions



Encapsulation

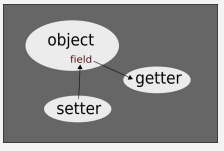
- An implementation mechanism for enforcing information hiding and abstractions.
- There is no clear widely accepted definition of encapsulation. It can mean:
  - A grouping together of related things (records, arrays)
  - A protected enclosure, (object with private data and/or methods),

  e.g. a java class



Example 1: Encapsulation

```
class Point{
   int x;
   int y;
}

Point p = new Point();

p.x = 10;
p.y = 10;
```



Do we need Setters? Getters?

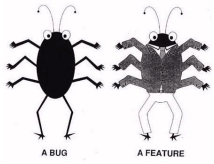
Information Hiding

- Hide:
  - data,
  - data formats,
  - behavior, and
  - design decisions
- Parnas encourages programmers to hide
  - "difficult design decisions
  - design decisions which are likely to change"
- **The clients of a module only need to be aware of its interface.**
  - Implementation details should be hidden
- Information hiding implies encapsulation and abstraction.
  - You are hiding details which creates an *abstraction*.
  - The details are *encapsulated*
- When information is hidden there is an implied separation between interface and implementation.
  - The information is hidden behind the interface

Why Practice Information Hiding?

- Hiding complexity
  - limiting the amount of information you have to deal with at any one time **(7 things +- 2)**
- Reducing dependencies on
  - design and implementation decisions
    - minimize the impact of changes.
    - Avoid the ripple effect of changes.
- Reduce Defects

"Large programs that use information hiding were found …to be easier to modify—by a factor of 4—than programs that don't.[Korson and Vaishnavi 1986]



Example 1:

Evaluate the following design in terms of information hiding

```
class PersistentData {
    public ResultSet read(string sql){...};
    // write returns the number of rows affected
    public int write(string sql){...};
}
```

Sample Client Code

```
PersistentData db = new PersistentData();
db.write("UPDATE Employees SET Dependents = 2
          WHERE EmployeeID = 47");
```

What information is hidden?

- Logic to connect to DB, sessions, session pooling, etc.
- Does it hide the flavor/brand of the database?

Example 1:

Evaluate the following design in terms of information hiding

```
class EmployeeGateway {
    public static EmployeeGateway find(int ID);
    public void setName(string name);
    public string getName();
    public void setDependents(int dependents);
    public int getDependents();
    // insert() returns ID of employee
    public int insert();
    public void update();
    public void delete();
}
```

Sample Client Code

```
EmployeeGateway e = EmployeeGateway.find(47);
e.setDependents(2);
e.update();
```

Example 2:

Evaluate the following class design in terms of information hiding

```
class Course {
    private Set students;

    public Set getStudents() {
        return students;
    }
    public void setStudents(Set s) {
        students = s;
    }
}
```

- Could it do a better job of information hiding?
- Could it have been worse?

Example 2:

Evaluate the following class design in terms of information hiding

```
class Course {
    private Set students;

    public Set getStudents() {
        return students;
    }
    public void setStudents(Set s) {
        students = s;
    }
}
```

- Could it do a better job of information hiding?
  - This is a classic example of how not to do setters and getters.
- Could it have been worse?
  - It could have made students a public field.

Example 2:
Improved information hiding

```
class Course {
    private Set students;

    public Set getStudents() {
        return Collections.
                unmodifiableSet(students);
    }
    public void addStudent(Student student) {
        students.add(student);
    }
    public void removeStudent(Student student) {
        students.remove(student);
    }
}
```

- This class design offers better encapsulation and information hiding. The class Course has complete control over the content of its internal data structure.

Supporting Design Principles and Heuristics

The Basics

Principle of Least Astonishment/Surprise (POLA)

- Don't surprise the user (UI design) or programmer (software design) with unexpected behavior. Users and developers should be able to rely on their intuition, e.g., what does `PrintStringToLog("foobar")` do?

- Separation of Concerns
  - The functions, or more generally concerns, of a program should be separate and distinct such that they may be dealt with on an individual basis.
  - Separation of concerns helps guide module formation. Functions should be distributed among modules in a way that minimizes interdependencies with other modules.

SOLID Principles of Object-Oriented Design

- S – Single responsibility principle
  - Classes should have a single reason to change
- O – Open/closed principle
  - Open for extension, closed for modification
- L – Liskov substitution principle
  - Code with a reference to a base class should be able to use objects of a derived class without knowing it
- I – Interface segregation principle
  - Say no to fat interfaces
- D – Dependency inversion principle
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.
  - Hollywood Principle, *don't call us, we'll call you.*

SRP::Example

```
public class SubHunter extends Activity {
void draw(){
   gameView.setImageBitmap(blankBitmap);
   canvas.drawColor(Color.argb(255,255,255,255));

   for(int i=0; i< gridWidth; i++)  // draw the vertical lines of the grid
           canvas.drawLine(   blockSize * i, 0,blockSize * i,
                              numberVerticalPixels-1,paint);

   for(int i=0; i< gridHeight; i++)    // draw the horizontal lines of the grid
        canvas.drawLine(0, blockSize * i,
                        numberHorizontalPixels -1,blockSize * i,paint);

   // draw player shot
   canvas.drawRect(horizontalTouched * blockSize,verticalTouched * blockSize,
                   (horizontalTouched * blockSize) + blockSize,
                   (verticalTouched * blockSize)+ blockSize,paint);

   paint.setTextSize(blockSize);                          // resize and draw
   paint.setColor(Color.argb(255,0,0,255));               // the text for the
   canvas.drawText("Shots Taken: " + shotsTaken +         // score and
                   "   Distance: " + distanceFromSub,     // distance
                   blockSize, blockSize * 1.0f, paint);

   Log.d("Debugging", "In draw");
   if(debugging)
       printDebuggingText();
}
```

SRP::Example
Note: Content/Common coupling -> Data coupling

```
class Grid {
   private int numberHorizontalPixels, numberVerticalPixels, blockSize;
   private final int gridWidth = 40;
   private int gridHeight;
   private Canvas myCanvas;

   public Grid(Point size, Canvas myCanvas){
       numberHorizontalPixels = size.x;
       numberVerticalPixels = size.y;
       blockSize = numberHorizontalPixels/gridWidth;
       gridHeight = numberVerticalPixels/blockSize;
       this.myCanvas=myCanvas;
   }

   void draw(Paint paint){

       myCanvas.drawColor(Color.argb(255,255,255,255));

       for(int i=0; i< gridWidth; i++) // vertical
          myCanvas.drawLine(blockSize * i, 0,blockSize * i, numberVerticalPixels -1, paint);

       for(int i=0; i< gridHeight; i++)    // horizontal
           myCanvas.drawLine(0, blockSize * i,numberHorizontalPixels -1,blockSize * i,paint);
   }

   int getHeight()          {return gridHeight;}
   int getWidth()           {return gridWidth;}
   int getBlockSize()       {return blockSize;}
}
```

OCP::Example

- Consider this class, and suppose the client would also like to measure the area of a circle, what could we do?

```
public class AreaCalculator{
    public double Area(Rectangle[] shapes){
        double area = 0;
        foreach (var shape in shapes){
            area += shape.Width*shape.Height;
        }
        return area;
    }
}
```

OCP::Example

This is a solution, but, it violates the open closed principle

```
public double Area(object[] shapes){
    double area = 0;
    foreach (var shape in shapes){
        if (shape is Rectangle){
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else{
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }
    return area;
}
```

OCP::Example

This solution does not violate the open closed principle

```
public abstract class Shape{public abstract double Area();}

public class Rectangle : Shape{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area(){return Width*Height;}
}

public class Circle : Shape{
    public double Radius { get; set; }
    public override double Area(){return Radius*Radius*Math.PI;}
}

public double Area(Shape[] shapes){
    double area = 0;
    foreach (var shape in shapes){
        area += shape.Area();}
        return area;
}
```

LSP::Example

```
public class Rectangle {
  protected int _width;
  protected int _height;

  public int getWidth() {
    return _width;
  }

  public int getHeight() {
    return _height;
  }

  public void setWidth(int width) {
    _width = width;
  }

  public void setHeight(int height) {
    _height = height;
  }
}
```

LSP::Example

Let Square be a subtype of Rectangle

```
class Square extends Rectangle {
  @Override
  public void setWidth(int width) {
    _width = width;
    _height = _width;
  }

  @Override
  public void setHeight(int height) {
    _height = height;
    _width = _height;
  }
}
```

LSP::Example

Compiles, but does it pass?

```
import static org.junit.Assert.*;
import org.junit.Test;
public class RectangleTest {

  @Test public void testArea() {
    Rectangle r = new Square();
    r.setWidth(5);
    r.setHeight(2);
    // Does this assertion pass?
    assertEquals(10, r.getWidth() * r.getHeight());
  }
}
```

- **Solution:** If the behavior by Square is unacceptable and unexpected, Square should not be derived from Rectangle.
- What about the approach in the previous example?

ISP::Example

- Java has two interfaces for mouse events, one for common mouse events (MouseListener) and one for motion events (MouseMotionListener).

  Grouping all events into one interface would have violated the ISP.

```
interface MouseListener {
  void mouseClicked(MouseEvent e);
  void mousePressed(MouseEvent e);
  void mouseReleased(MouseEvent e);
  void mouseEntered(MouseEvent e);
  void mouseExited(MouseEvent e);
}
```

```
interface MouseMotionListener {
  void mouseDragged(MouseEvent e);
  void mouseMoved(MouseEvent e);
}
```

DIP::Example

- The Manager *calls* for his worker by declaring a new Worker.
- The higher level class (Manager) depends on the lower level class (Worker)
- In order to add **SuperWorker** we have to modify class Manager

```
class Manager {
    Worker worker;
    public Manager(){
        worker = new Worker(...);
    }
    public void manage() {
        worker.work();
    }
}
```

```
class Worker {
    public void work() {
        // ....working
    }
}

class SuperWorker {
    public void work() {
        //.... working much more
    }
}
```

DIP::Example

- Now both depend on the interface **IWorker.**
- The dependency is *inverted* now the Manager doesn't depend on the lower level class but both depend on the interface. This reduces coupling between Manager and Worker
- Also, client must *inject* the worker dependency through the **setWorker** method call. Rather than the Manager asking what its workers are, it is told, *don't call us, we'll call you*

```
interface IWorker {
    public void work();
}

class Manager {
  IWorker worker;

  public void setWorker(IWorker w){
    worker = w;
  }

  public void manage() {
    worker.work();
  }
}
```

```
class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}
```

The Basics

- Don't Repeat Yourself (DRY)
  - In general, every piece of knowledge, code, test cases, plans, etc, should have a single, unambiguous, authoritative representation within a system.
  - Repeating yourself invites maintenance problems. You have two or more locations that have to be kept synchronized/consistent.
- Principle of Least Astonishment/Surprise (POLA)
  - Don't surprise the user (UI design) or programmer (software design) with unexpected behavior. Users and developers should be able to rely on their intuition, e.g., what does `PrintStringToLog("foobar")` do?
- Separation of Concerns
  - The functions, or more generally concerns, of a program should be separate and distinct such that they may be dealt with on an individual basis.
  - Separation of concerns helps guide module formation. Functions should be distributed among modules in a way that minimizes interdependencies with other modules.