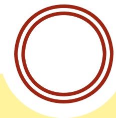
In the name of God

Object-Oriented Programming Concepts

Advanced topics in Software Engineering



Leila Samimi-Dehkordi
PhD. Student
Department of Software Engineering
Faculty of Computer Engineering
University of Isfahan
Fall 2014
Email: *l.samimi@gmail.com*

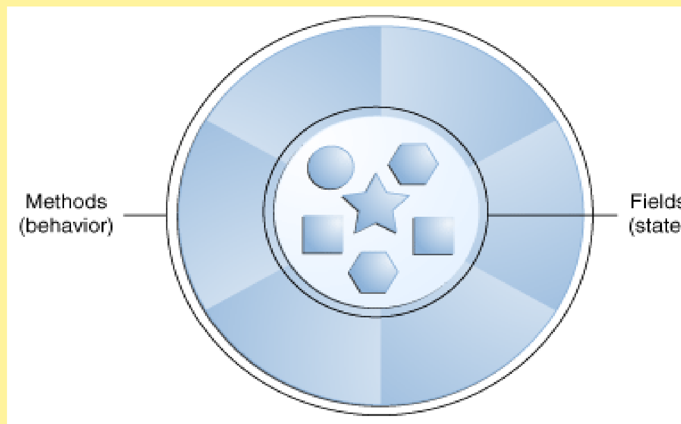
These slides are prepared for Master student of Islamic Azad University of Shahrekord

What Is an Object?

- Objects are key to understanding *object-oriented* technology.
- Real-world objects share two characteristics: *state* and *behavior*.
  - Dogs have state (name, color, breed) and behavior (barking, wagging tail).
  - Bicycles have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes).
- For each object that you see, ask yourself two questions:
  - "What possible states can this object be in?"
  - "What possible behavior can this object perform?”

A software object

- An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages).
- Methods:
  - Operate on an object's internal state
  - Serve as the primary mechanism for object-to-object communication.



A software object (2)

- Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation.*

Benefits

- Modularity
  - The source code for an object can be written and maintained independently of the source code for other objects.
  - Once created, an object can be easily passed around inside the system.
- Information-hiding
  - By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- Code re-use
  - If an object already exists (perhaps written by another software developer), you can use that object in your program.
  - This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- Pluggability and debugging ease
  - If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement.

What Is a Class?

- In the real world, you'll often find many individual objects all of the same kind.
- In object-oriented terms, we say that an special object is an *instance* of the *class of objects*.
- A *class* is the blueprint from which individual objects are created.

What Is an Interface?

- As you've already learned, objects define their interaction with the outside world through the methods that they expose.
- Methods form the object's *interface* with the outside world;
  - the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing.
  - You press the "power" button to turn the television on and off.
- Implementing an interface allows a class to become more formal about the behavior it promises to provide.
- Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.

What Is a Package?

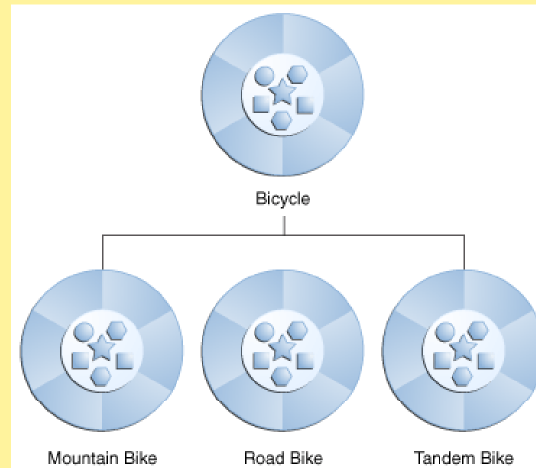
- A package is a namespace that organizes a set of related classes and interfaces.
- Conceptually you can think of packages as being similar to different folders on your computer.
- Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.
- The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications.
- This library is known as the "Application Programming Interface", or "API" for short.

Fundamental principles of OO

- Inheritance
- Abstraction
- Encapsulation
- Polymorphism

What Is Inheritance?

- Different kinds of objects often have a certain amount in common with each other.
- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.
- In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:

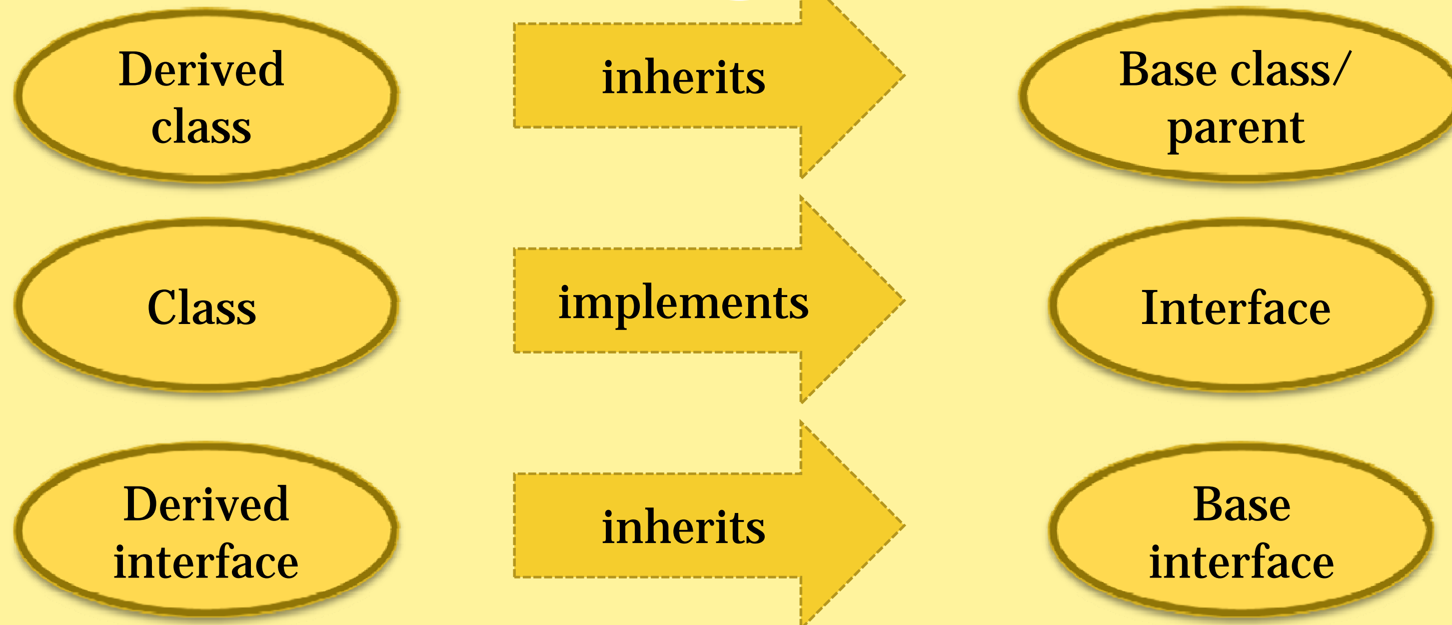


Benefits of Inheritance

- Extensibility
- Reusability
- Provides Abstraction
- Eliminates redundant code

- Use inheritance for is-a relationship not has-a relationship.

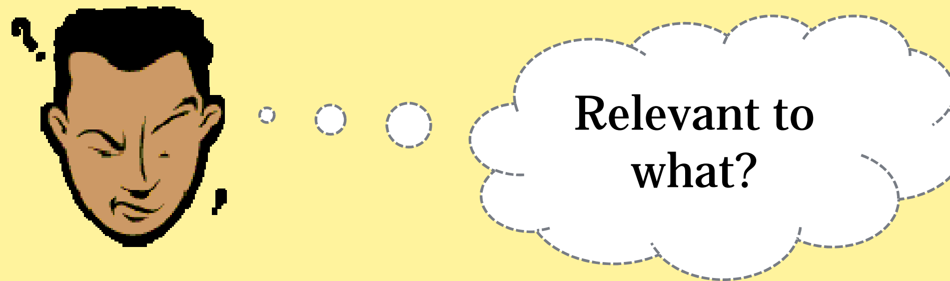
Inheritance Terminology



- ✓A class can inherit only one class.
- ✓A class can implement several interfaces.
- ✓An interface can inherits several interfaces.

Abstraction

- It means ignoring irrelevant features, properties, functions, and emphasizing relevant ones.



- … relevant to the given project (with an eye to the future reuse in similar projects)
- Abstraction = managing complexity

Encapsulation

- Encapsulation hides the implementation details .
- It is also called "**information hiding**".
- An object has to provide its users only with the essential information for manipulation, without the internal details.
  - A **Secretary** using a **Laptop** only knows about its screen, keyboard and mouse.
- This is essentially a contract between the two objects.
  - The invoker is agreeing to send the message in the specific format, including passing any of the required parameter information.
  - The invoked object is agreeing to process the message and if necessary, return a value in the pre-specified format.

Encapsulation (2)

- Class announces some operations (methods) available for its client.
- All data members (fields) of a class should be hidden.
- No interface members should be hidden.
- It is implemented with:
  - Public interfaces
  - controls of visibility of operations & state
  - data is private / not accessible

Encapsulation benefits

- Encapsulation promotes maintenance
  - because code changes can be made independently without effecting other classes.
- Increases usability
  - By keeping data private and providing public well-defined service methods the role of the object becomes clear to other objects.

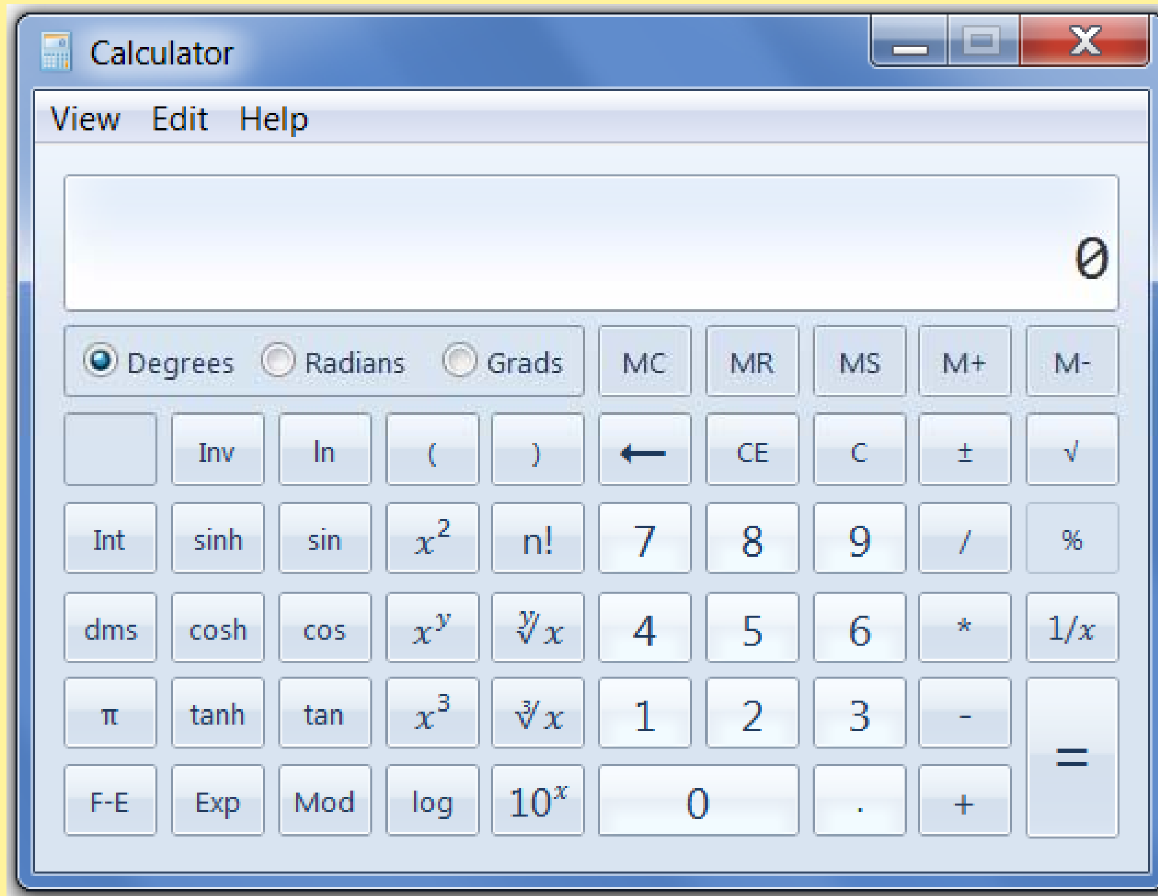
Polymorphism

- Polymorphism = the ability to take more than one form.
- It allows abstract operations to be defined and used
  - Abstract operations are defined in the base class/ interface and implemented in the child classes.
- The purpose of polymorphism is to implement *message-passing* in which objects of various types define a common interface of operations for users.
- It is implemented using a technique called *late method binding (*binding in runtime*)*.

Polymorphism examples


```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

void letsHear(Animal a) {
    println(a.talk());
}

void main() {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

Real world example



Cohesion

- Cohesion describes the focus of an individual software component.
- Cohesion = The degree to which a class has a single, well-focused purpose
- When a component has only one responsibility, and therefore only one reason to change, then it has *high cohesion*.
- When a component has many responsibilities, and therefore many reasons to change, then it has *low cohesion*.
- Low cohesion is also noticeable when common responsibilities are spread throughout unrelated components.

Cohesion benefits

- Highly cohesive classes are much easier to maintain and less frequently changed.
- Such classes are more usable than others as they are designed with a well-focused purpose.
- Keep your components small and focused.

Coupling

- Whereas cohesion is used to describe a single software component, coupling is used to describe the relationship between components.
- Coupling is the extent to which a component of your software depends on other components.
- Coupling = The degree to which one class knows about another class.

Loosely vs. tightly coupled

- If class **A** knows class **B** through its interface only i.e it interacts with class **B** through its API then class **A** and class **B** are said to be *loosely coupled.*
- If on the other hand class **A** apart from interacting class **B** by means of its interface also interacts through the non-interface stuff of class **B** then they are said to be *tightly coupled.*
- Loosely coupled components have fewer dependencies than tightly coupled components.
- Modifying a tightly coupled component is difficult.
  - If a car object, boat object, and plane object are all tightly coupled to an engine object, then a modification to the engine object will necessitate changes to the car, boat, and plane objects.

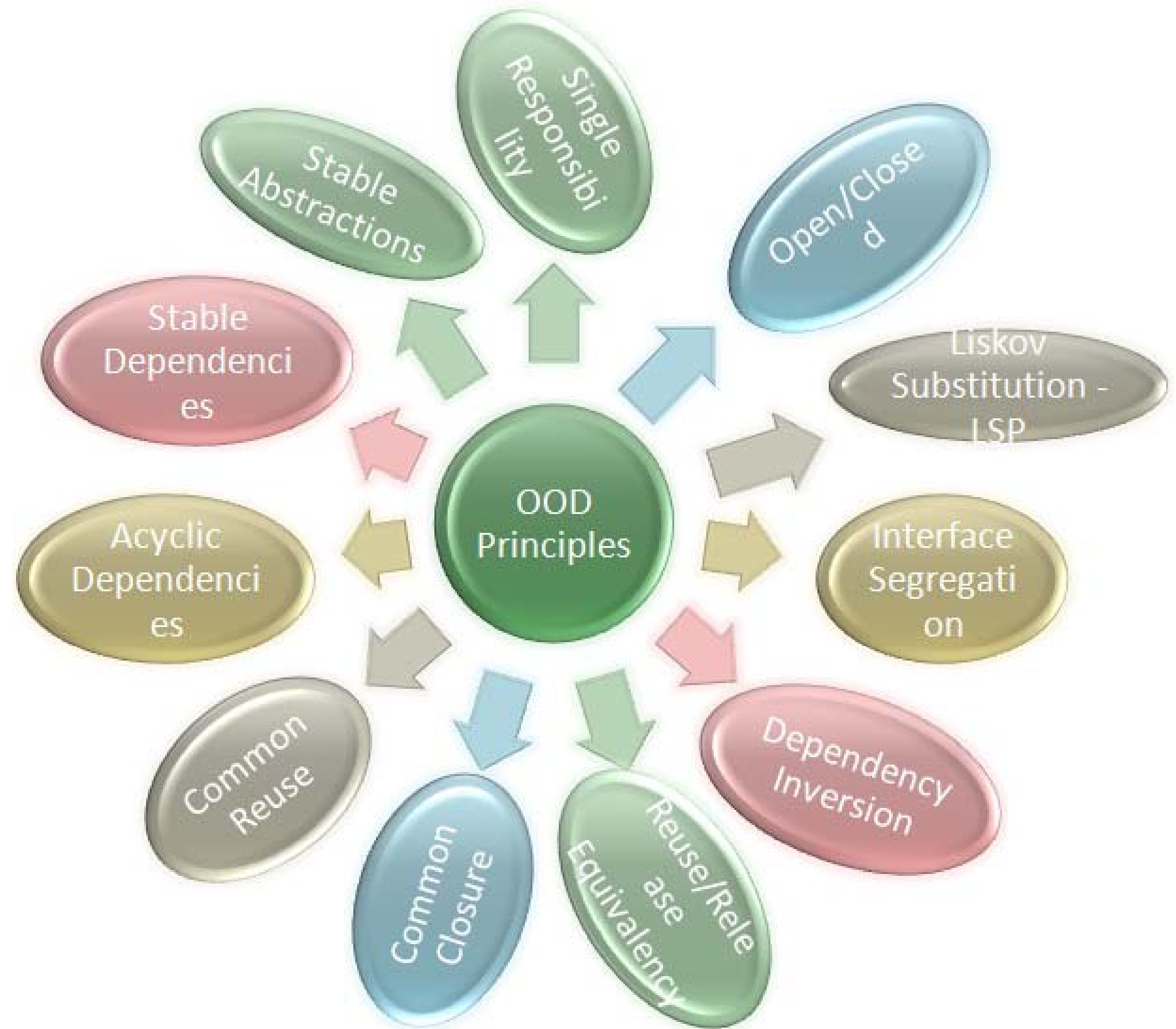
Exercise (1)

- We are given a **school**. The school has classes of students. Each class has a set of **teachers**. Each teacher teaches a set of **courses**. The students have a name and unique number in the class. **Classes** have a unique text identifier. Teachers have names. Courses have a name, count of classes and count of exercises. The teachers as well as the students are people. Your task is to model the classes (in terms of OOP) along with their attributes and operations define the class hierarchy and create a class diagram.

Exercise (2)

- A **bank** holds different **types of accounts** for its customers: **deposit** accounts, **loan** accounts and **mortgage** accounts. Customers can be **individuals** or **companies**. All accounts have a customer, balance and interest rate (monthly based). **Deposit accounts** allow depositing and withdrawing of money. **Loan and mortgage accounts** allow only depositing. Your task is to write an object-oriented model of the bank system. You must identify the classes, interfaces, base classes and abstract actions.

Principles of the OO Design



OO design principles

- **Mr. Robert Martin** (commonly known as Uncle Bob) categorized OO design principles as
  - Class Design principles – Also called SOLID
  - Package Cohesion Principles
  - Package Coupling principle

SOLID principles

- SOLID =
  - Single responsibility
  - Open-closed
  - Liskov substitution
  - Interface segregation
  - Dependency inversion

SRP - Single responsibility Principle

- SRP says "Every software module should have only one reason to change".
  - Software Module – Class, Function etc.
  - Reason to change - Responsibility
- **Note:** This principle also applies to methods. Every method should have a single responsibility.

Can a single class can have multiple methods?

- The answer is YES.
- Now you might ask how it's possible that
  - A class will have single responsibility.
  - A method will have single responsibility.
  - A class may have more than one method.
- Here, responsibility is related to the context in which we are speaking.
- When we say class responsibility it will be somewhat at higher level.

OCP – Open Close Principle

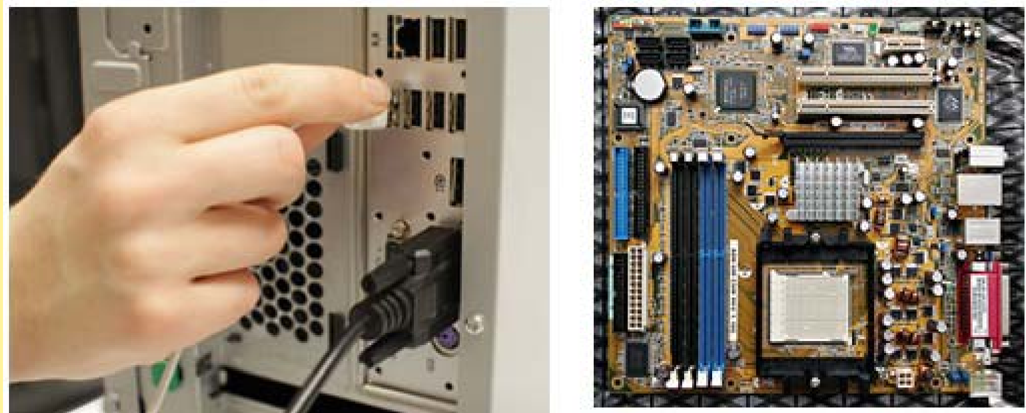
- It says “Software modules should be closed for modifications but open for extensions”.
- In other words, you should never need to change existing code or classes:
  - All new functionality can be added by adding new subclasses and overriding methods,
  - Or by reusing existing code through delegation.
- This prevents you from introducing new bugs in existing code.
  - If you never change it, you can't break it.

LSP – Liskov substitution principle

- It says, "Subclasses should be substitutable for base classes.“
- Derived classes must be usable through the base class interface without the need for the user to know the difference.
- LSP ="An instance of a derived should be able to replace any instance of its superclass"
- Hierarchy allows the use of type families, in which higher level supertypes capture the behavior that all of their subtypes have in common.

ISP– Interface Segregation principle

- It states that "Clients should not be forced to implement interfaces they don't use."
- It can also be stated as "Many client specific interfaces are better than one general purpose interface."
- In simple words, if your interface is fat, break it into multiple interfaces.



DIP– Dependency Inversion principle

- It says, "High level modules should not depend upon low level modules. Rather, both should depend upon abstractions."
  - Different parts such as RAM, a hard disk, and CD-ROM (etc.) are loosely connected to the motherboard. That means that, if, in future in any part stops working it can easily be replaced with a new one.
  - Just imagine a situation where all parts were tightly coupled to each other, which means it would not be possible to remove any part from the motherboard. Then in that case if the RAM stops working we have to buy new motherboard which is going to be very expensive.

Conclusion of SOLID

- SRP should be kept in mind while creating any class, method or any other module.
  - It makes code more readable, robust, and testable.
- We can't follow DIP each and every time, sometimes we have to depend on concrete classes.
  - The only thing we have to do is understand the system, requirements and environment properly and find areas where DIP should be followed.
- Following DIP and SRP will opens a door to implement OCP as well.
- Make sure to create specific interfaces so that complexities and confusions will be kept away from end developers, and thus, the ISP will not get violated.
- While using inheritance take care of LSP.

Package cohesion principles

- REP= The Release Reuse Equivalency Principle
  - *The granule of reuse is the granule of release.*
- CCP= The Common Closure Principle
  - *Classes that change together are packaged together.*
- CRP = The Common Reuse Principle
  - *Classes that are used together are packaged together.*

Release Reuse Equivalency Principle

- REP essentially means that the package must be created with reusable classes
- "Either all of the classes inside the package are reusable, or none of them are".
- The classes must also be of the same family.
- We don't want to include classes that are unrelated to the purpose of the package.
- If we construct a package as a family of reusable classes, we can more closely guarantee a focused, reusable package.

Common Closure Principle

- CCP states that the package should not have more than one reason to change.
- If change were to happen in an application dependent on a number of packages, ideally we only want changes to occur in one package, rather than in a number of them.
- This helps us determine classes that are likely to change and package them together for the same reasons.
- If the classes are tightly coupled, put them in the same package.

Common Reuse Principle

- The CRP states that classes that tend to be reused together belong in the same package together.
- It is a way of helping us decide which classes belong in which package.
- We also want to keep in mind that when we depend on a package, we want to make sure that the classes are inseparable, and interdependent, which is also handy when culling out classes that do or don't belong.

Couplings between packages principles

- ADP = The Acyclic Dependencies Principle
  - *The dependency graph of packages must have no cycles.*
- SDP = The Stable Dependencies Principle
  - *Depend in the direction of stability.*
- SAP = The Stable Abstractions Principle
  - *Abstractness increases with stability.*

Acyclic Dependencies Principle

- In a development cycle with multiple developers, cooperation and integration needs to happen in small incremental releases.
- The ADP states that there can be no cycles in the dependency structure, and that when an incremental release is made, the other developers can adopt and build upon it.

Stable Dependencies Principle

- Designs are changing.
- So we need to design our packages to be able to change as well.
- The SDP states that any packages we want to be volatile should not be depended on by a package that is difficult to change.

Stable Abstractions Principle

- The SAP says that a stable package should also be abstract so that its stability does not prevent it from being extended.
- It also states that an instable package should be concrete since its instability allows the concrete code within it to be easily changed.

Thanks for your attention