Principles

Single Responsibility Principle

: ★★★☆ Can explain single responsibility principle

> **Single responsibility principle (SRP)**: A class should have one, and only one, reason to change. -- Robert C. Martin

If a class has only one responsibility, it needs to change only when there is a change to that responsibility.

Consider a `TextUi` class that does parsing of the user commands as well as interacting with the user. That class needs to change when the formatting of the UI changes as well as when the syntax of the user command changes. Hence, such a class does not follow the SRP.

> Gather together the things that change for the same reasons. Separate those things that change for different reasons. —- *Agile Software Development, Principles, Patterns, and Practices* by Robert C. Martin

Resources

- An explanation of the SRP from www.oodesign.com
- Another explanation (more detailed) by Patkos Csaba
- A book chapter on SRP written by the father of the principle itself, Robert C Martin

Open-Closed Principle

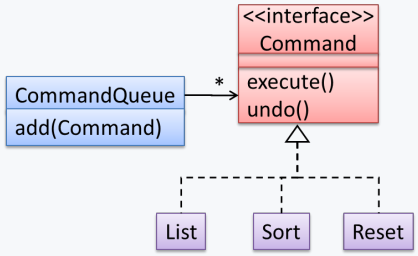
: ★★★☆ Can explain open-closed principle (OCP)

The Open-Closed Principle aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself.

> **Open-closed principle (OCP)**: A module should be *open* for extension but *closed* for modification. That is, modules should be written so that they can be extended, without requiring them to be modified. -- proposed by Bertrand Meyer

In object-oriented programming, OCP can be achieved in various ways. This often requires separating the *specification (i.e. interface)* of a module from its *implementation*.

In the design given below, the behavior of the `CommandQueue` class can be altered by adding more concrete `Command` subclasses. For example, by including a `Delete` class alongside `List`, `Sort`, and `Reset`, the `CommandQueue` can now perform delete commands without modifying its code at all. That is, its behavior was extended without having to modify its code. Hence, it is open to extensions, but closed to modification.



The behavior of a Java generic class can be altered by passing it a different class as a parameter. In the code below, the `ArrayList` class behaves as a container of `Students` in one instance and as a container of `Admin` objects in the other instance, without having to change its code. That is, the behavior of the `ArrayList` class is extended without modifying its code.

```
ArrayList students = new ArrayList<Student>();
ArrayList admins = new ArrayList<Admin>();
```

Exercises

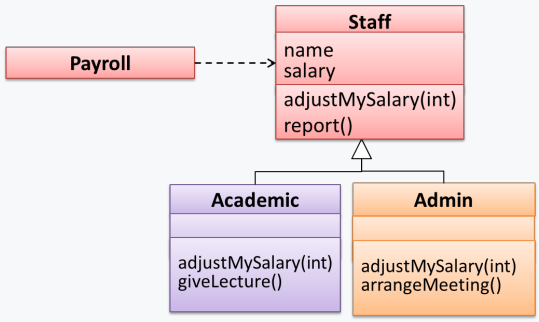
Liskov Substitution Principle

: ★★★☆ Can explain Liskov Substitution Principle

> **Liskov substitution principle (LSP)**: Derived classes must be substitutable for their base classes. -- proposed by Barbara Liskov

LSP sounds the same as substitutability but it goes beyond substitutability; **LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass.** As you know, Java has language support for substitutability. However, if LSP is not followed, substituting a subclass object for a superclass object can break the functionality of the code.

Suppose the `Payroll` class depends on the `adjustMySalary(int percent)` method of the `Staff` class. Furthermore, the `Staff` class states that the `adjustMySalary` method will work for all positive percent values. Both the `Admin` and `Academic` classes override the `adjustMySalary` method.



Now consider the following:

- The `Admin#adjustMySalary` method works for both negative and positive percent values.
- The `Academic#adjustMySalary` method works for percent values `1..100` only.

In the above scenario,

- The `Admin` class follows LSP because it fulfills `Payroll`'s expectation of `Staff` objects (i.e. it works for all positive values). Substituting `Admin` objects for `Staff` objects will not break the `Payroll` class functionality.
- The `Academic` class violates LSP because it will not work for percent values over `100` as expected by the `Payroll` class. Substituting `Academic` objects for `Staff` objects can potentially break the `Payroll` class functionality.

Another example

Exercises

Interface Segregation Principle

★★★★ Can explain interface segregation principle

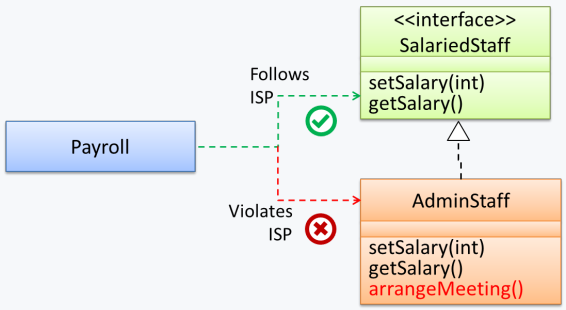
Interface segregation principle (ISP): No client should be forced to depend on methods it does not use.

The `Payroll` class should not depend on the `AdminStaff` class because it does not use the `arrangeMeeting()` method. Instead, it should depend on the `SalariedStaff` interface.

```
public class Payroll {
    // violates ISP
    private void adjustSalaries(AdminStaff adminStaff) {
        // ...
    }

}
```

```
public class Payroll {
    // does not violate ISP
    private void adjustSalaries(SalariedStaff staff) {
        // ...
    }
}
```



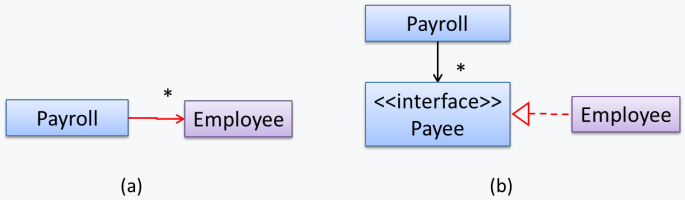
Dependency Inversion Principle

★★★★ Can explain dependency inversion principle (DIP)

Dependency inversion principle (DIP):

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Example:



In design (a), the higher level class `Payroll` depends on the lower level class `Employee`, which is a violation of DIP. In design (b), both `Payroll` and `Employee` depend on the `Payee` interface (note that inheritance is a dependency).

Design (b) is more flexible (and less coupled) because now the `Payroll` class need not change when the `Employee` class changes.

Exercises

SOLID Principles

★★★☆ Can explain SOLID Principles

The five OOP principles given below are known as *SOLID Principles* (an acronym made up of the first letter of each principle):

Single Responsibility Principle (SRP)

Principles →

Single responsibility principle

> **Single responsibility principle (SRP)**: A class should have one, and only one, reason to change. -- Robert C. Martin

If a class has only one responsibility, it needs to change only when there is a change to that responsibility.

Consider a `TextUi` class that does parsing of the user commands as well as interacting with the user. That class needs to change when the formatting of the UI changes as well as when the syntax of the user command changes. Hence, such a class does not follow the SRP.

> Gather together the things that change for the same reasons. Separate those things that change for different reasons. —- *Agile Software Development, Principles, Patterns, and Practices* by Robert C. Martin

Resources

- An explanation of the SRP from www.oodesign.com
- Another explanation (more detailed) by Patkos Csaba
- A book chapter on SRP written by the father of the principle itself, Robert C Martin

Open-Closed Principle (OCP)

Principles →

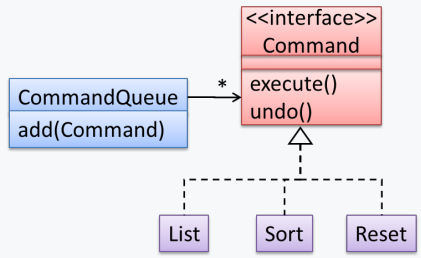
Open-closed principle

The Open-Closed Principle aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself.

> **Open-closed principle (OCP)**: A module should be *open* for extension but *closed* for modification. That is, modules should be written so that they can be extended, without requiring them to be modified. -- proposed by Bertrand Meyer

In object-oriented programming, OCP can be achieved in various ways. This often requires separating the *specification (i.e. interface)* of a module from its *implementation*.

In the design given below, the behavior of the `CommandQueue` class can be altered by adding more concrete `Command` subclasses. For example, by including a `Delete` class alongside `List`, `Sort`, and `Reset`, the `CommandQueue` can now perform delete commands without modifying its code at all. That is, its behavior was extended without having to modify its code. Hence, it is open to extensions, but closed to modification.



The behavior of a Java generic class can be altered by passing it a different class as a parameter. In the code below, the `ArrayList` class behaves as a container of `Students` in one instance and as a container of `Admin` objects in the other instance, without having to change its code. That is, the behavior of the `ArrayList` class is extended without modifying its code.

```
1 ArrayList students = new ArrayList<Student>();
2 ArrayList admins = new ArrayList<Admin>();
```

Exercises

Liskov Substitution Principle (LSP)

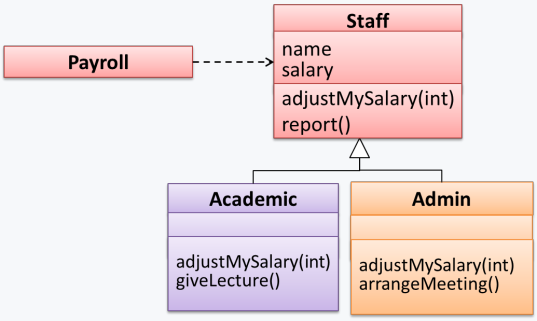
Principles →

Liskov substitution principle

> **Liskov substitution principle (LSP)**: Derived classes must be substitutable for their base classes. -- proposed by Barbara Liskov

LSP sounds the same as substitutability but it goes beyond substitutability; **LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass.** As you know, Java has language support for substitutability. However, if LSP is not followed, substituting a subclass object for a superclass object can break the functionality of the code.

Suppose the `Payroll` class depends on the `adjustMySalary(int percent)` method of the `Staff` class. Furthermore, the `Staff` class states that the `adjustMySalary` method will work for all positive percent values. Both the `Admin` and `Academic` classes override the `adjustMySalary` method.



Now consider the following:

- The `Admin#adjustMySalary` method works for both negative and positive percent values.
- The `Academic#adjustMySalary` method works for percent values `1..100` only.

In the above scenario,

- The `Admin` class follows LSP because it fulfills `Payroll`'s expectation of `Staff` objects (i.e. it works for all positive values). Substituting `Admin` objects for `Staff` objects will not break the `Payroll` class functionality.
- The `Academic` class violates LSP because it will not work for percent values over `100` as expected by the `Payroll` class. Substituting `Academic` objects for `Staff` objects can potentially break the `Payroll` class functionality.

Another example

Exercises

Interface Segregation Principle (ISP)

Principles →

Interface segregation principle

Interface segregation principle (ISP): No client should be forced to depend on methods it does not use.

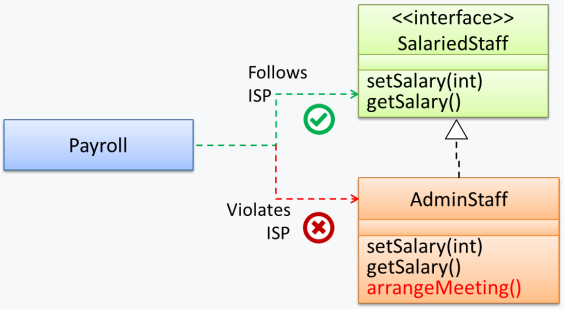
The `Payroll` class should not depend on the `AdminStaff` class because it does not use the `arrangeMeeting()` method. Instead, it should depend on the `SalariedStaff` interface.

```
1 public class Payroll {
```

```
    // violates ISP
    private void adjustSalaries(AdminStaff adminStaff) {
        // ...
    }

}
```

```
public class Payroll {
    // does not violate ISP
    private void adjustSalaries(SalariedStaff staff) {
        // ...
    }
}
```



Dependency Inversion Principle (DIP)

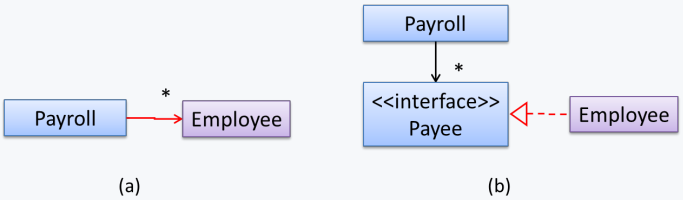
Principles →

Dependency inversion principle

Dependency inversion principle (DIP):

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Example:



In design (a), the higher level class `Payroll` depends on the lower level class `Employee`, which is a violation of DIP. In design (b), both `Payroll` and `Employee` depend on the `Payee` interface (note that inheritance is a dependency).

Design (b) is more flexible (and less coupled) because now the `Payroll` class need not change when the `Employee` class changes.

Exercises

Separation of Concerns Principle

★★☆☆ Can explain separation of concerns principle

> **Separation of concerns principle (SoC)**: To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate *concern*. -- Proposed by Edsger W. Dijkstra

A *concern* in this context is a set of information that affects the code of a computer program.

Examples for *concerns*:

- A specific feature, such as the code related to the `add employee` feature
- A specific aspect, such as the code related to `persistence` or `security`
- A specific entity, such as the code related to the `Employee` entity

Applying SoC reduces functional overlaps among code sections and also limits the ripple effect when changes are introduced to a specific part of the system.

If the code related to *persistence* is separated from the code related to *security*, a change to how the data are persisted will not need changes to how the security is implemented.

This principle can be applied at the class level, as well as at higher levels.

The n-tier architecture utilizes this principle. Each layer in the architecture has a well-defined functionality that has no functional overlap with each other.

This principle should lead to higher cohesion and lower coupling.

Exercises

Law of Demeter

★★★☆ Can explain the Law of Demeter

> **Law of Demeter (LoD)**:
>
> - An object should have limited knowledge of another object.
> - An object should only interact with objects that are closely related to it.
>
> Also known as
>
> - Don't talk to strangers.
> - Principle of least knowledge

More concretely, a method `m` of an object `O` should invoke only the methods of the following kinds of objects:

- The object `O` itself
- Objects passed as parameters of `m`
- Objects created/instantiated in `m` (directly or indirectly)
- Objects from the direct association of `O`

The following code fragment violates LoD due to the following reason: while `b` is a 'friend' of `foo` (because it receives it as a parameter), `g` is a 'friend of a friend' (which should be considered a 'stranger'), and `g.doSomething()` is analogous to 'talking to a stranger'.

```
void foo(Bar b) {
    Goo g = b.getGoo();
    g.doSomething();
}
```

LoD aims to prevent objects from navigating the internal structures of other objects.

An analogy for LoD can be drawn from Facebook. If Facebook followed LoD, you would not be allowed to see posts of friends of friends, unless they are your friends as well. If Jake is your friend and Adam is Jake's friend, you should not be allowed to see Adam's posts unless Adam is a friend of yours as well.

Exercises

YAGNI Principle

★★★★ Can explain YAGNI principle

> **YAGNI (You Aren't Gonna Need It!) Principle**: Do not add code simply because 'you might need it in the future'.

The principle says that some capability you presume your software needs in the future should not be built now because chances are "you aren't gonna need it". The rationale is that you do not have perfect information about the future and therefore some of the extra work you do to fulfill a potential future need might go to waste when some of your predictions fail to materialize.

Resources

- **Yagni** -- A detailed article explaining YAGNI, written by Martin Fowler.

DRY Principle

★★★★ Can explain DRY principle

> **DRY (Don't Repeat Yourself) principle**: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. -- *The Pragmatic Programmer*, by Andy Hunt and Dave Thomas

This principle guards against the duplication of information.

- A functionality being implemented twice is a violation of the DRY principle even if the two implementations are different.
- The value of a system-wide timeout being defined in multiple places is a violation of DRY.

Brooks' Law

★★★★ Can explain Brooks' law

> **Brooks' law**: Adding people to a late project will make it later. -- Fred Brooks (author of *The Mythical Man-Month*)

Explanation: The additional communication overhead will outweigh the benefit of adding extra manpower, especially if done near a deadline.

Exercises