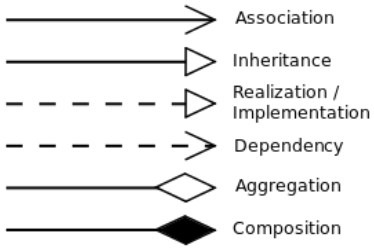
Object-oriented Modelling & Design [EDAF60]

Version 1.0 - by Alfred Hirschfeld

Autumn 2020 - 4.5 credits

Contents

1 Arrows in class diagrams



Association - has a reference of another class, e.g. (*attribute*).

```
class A {
  private B b;  //attribute

  public A(B b) {
    this.b = b;
  }
}
```

Dependency - when you receive a reference as part of a method (*parameter*).

```
class A {
  public void methodFromA(B b) {  //parameter
    b.methodFromB();
  }
}
```

Aggregation - an association where something is a part of something else, but both parts "have their own lives". One example is when a class has a list containing objects. Similar to a football team and football players. The team consists of players but players can also exist without the team.

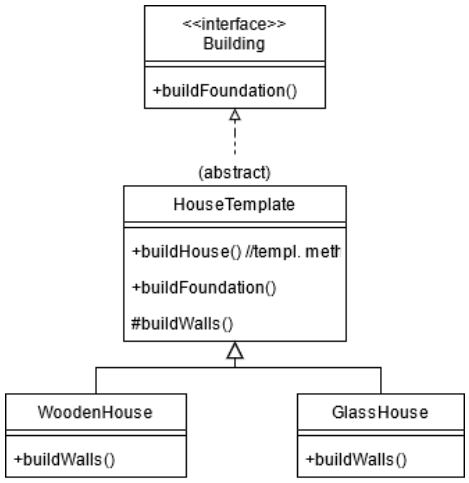
Composition - a stronger form of aggregation where class B is owned by class A and when you remove class A, class B also disappears. Similar to a human and her head, if you remove the human the head cannot exist.

2 Patterns

2.1 Template Pattern - build a house

- *What it does*: gathers common code in an abstract superclass (inheritance).
- *Contains*: one Template-class (which is the superclass) containing a Template-method which defines the order of executing the code.
- *Implementation*: declare the methods in superclass - implement them in subclasses where they differ.
- *Runtime*: superclass will call on methods from subclasses, not the opposite way which is known as **Hollywood Principle** – "don't call us, we'll call you".
- *Benefits*: helps us avoid **DRY** - repetition of code.

`Example`: build a house. To build a house you need foundation, pillars, walls and windows. We cannot change order of execution since the walls must be built before building windows.

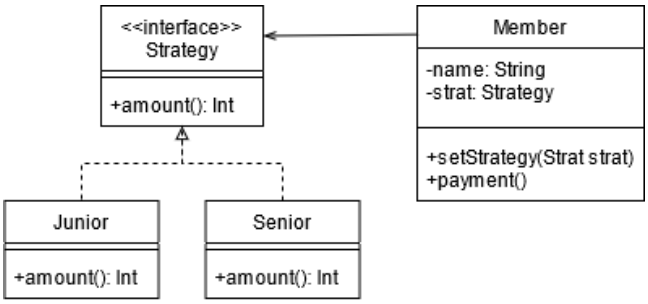



```
1 HouseTemplate glassHouse = new GlassHouse();
2 glassHouse.buildHouse();
```

2.2 Strategy Pattern - change memberships

- *What it does*: lets us choose algorithm when executing the code. We want to be able to change "strategy" during runtime.
- *Contains*: one Strategy-interface which describes how we call our algorithm.
- *Implementation*: several classes can implements the interface and the algorithm in different ways.
- *Benefits*: helps us reach **Open-Closed Principle** because we can add new types without changing existing code.

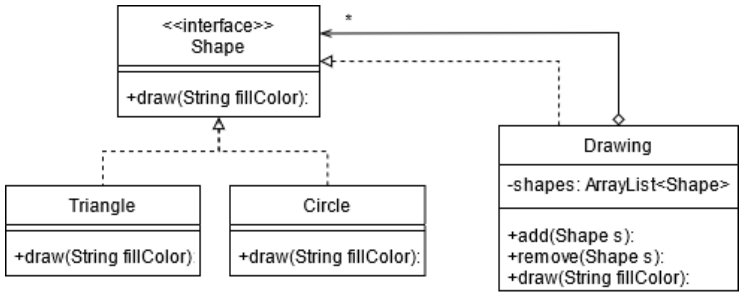
`Example`: implement different membership types - Junior and Senior. Can easily add more memberships (Rookie, Teacher) and change memberships by calling setStrategy() during runtime.



2.3 Composite Pattern - color shapes

- *What it does*: should be applied only when a group of objects should behave as a single object.
- *Contains*: aggregation (*) in class diagram, see below.
- *Implementation*: Composite-class has an ArrayList containing objects, hence the aggregation.
- *Benefits*: helps us avoid **DRY** - repetition of code. Also be able to process several objects uniformly.

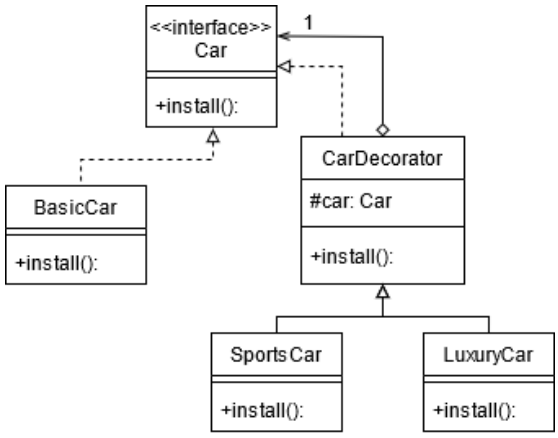
`Example`: different figures shall be colored with the same color. Drawing is our Composite-class. When we call draw(), all figures in our ArrayList will be colored uniformly. We can easily add more figures, such as Square.



2.4 Decorator Pattern - cars

- *What it does*: gives us opportunity to extend a program's functionality during runtime. It allows us to add behaviour to an object without affecting behaviour of the other objects from the same class.
- *Contains*: aggregation (1) in the class diagram, see below.
- *Implementation*: Decorator-class is an abstract superclass (inheritance).
- *Benefits*: helps us reach **Single Responsibility Principle**.

`Example`: implement cars and decorate them with different functionality. We want to get a car at runtime that has both features of a sports car and a luxury car. We can easily add more features to cars (e.g. FastCar, FlyingCar).

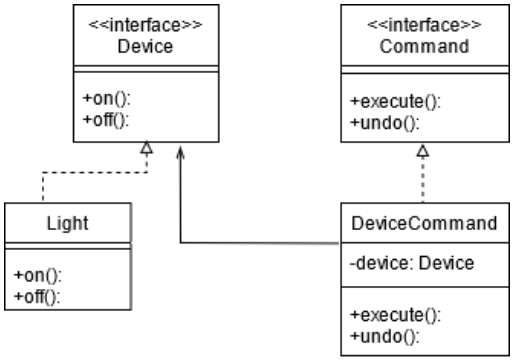



```
1 Car bCar = new BasicCar();
2 Car sportsLuxuryCar = new SportsCar(new LuxuryCar(bCar()));
```

2.5 Command Pattern - smart home

- *What it does*: creates an object, packages it with different functionalities and waits for an execute() call.
- *Contains*: interface with execute()-method. Can eventually have a method for regretting commands (undo).
- *Implementation*: a Command-interface with execute() method.
- *Benefits*: helps us get a history of what has happened by adding commands in stacks and lists. We get **high cohesion**, **loose coupling** and **Single Responsibility Principle** because we pack an object with everything we need to perform a given operation. In addition, we are able to add new commands without having to change existing code - (**Open-Closed Principle**).

`Example`: we have a smart home with a light that turns on and off. We can easily extend to more units: TV, oven, micro, etc.



2.6 Observer-Observable Pattern

- *What it does*: when you are interested in the state of an object and want to get notified whenever there is any change. Use when an object want to notify that it has changed without knowing who listens to it.
- *Contains:* Observer-class which is an interface with method update(). Observable is a superclass.
- *Implementation*: Observable always has a list with Observers. Must have an add(), setChanged(), notifyObservers() method.

```
interface Observer {
    void update(Observable obs, Object obj);
}

public class Observable {
    private List<Observer> observers = new LinkedList<>();
    private boolean changed = false;

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void setChanged() {
        changed = true;
    }

    public void hasChanged() {
        return changed;
    }

    public void clearChanged() {
        changed = false;
    }

    public void notifyObservers(Object obj) {
        if (!changed) {
            return;
        }
        for (obs observer : observers) {
            observer.update(this, obj);
        }
        clearChanged();
    }

    public void notifyObservers() {
        notifyObservers(null);
    }
}
```

2.7 Builder Pattern - person

- *What it does*: when objects contain a lot of attributes. Too many arguments to pass, hard to maintain order of argument.
- *Contains*: set-methods in Builder-class, get-methods outside of Builder-class.
- *Implementation*: put Builder-class as inner nested class and static because we only want to have one Builder-class.
- *Benefits*: to build complex objects. Is also a part of **fluent interfaces** - a design where we can call code in *method chains*:

```
1         var person =
2         Person
3         .builder()
4         .lastName("Einstein")
5         .firstName("Albert")
6         .build();
```

2.7.1 Template of Builder Pattern

```
1  public class X {
2
3    /*attributes*/
4
5    private X(Builder builder) {...} //private constructor
6
7    public static Builder builder() {
8      return new Builder();
9    }
10
11   public static class Builder { //static class
12
13     /*attributes*/
14
15     private Builder() {} //Empty private constructor
16
17     /*methods*/
18
19     public X build() {
20         return new X(this);
21     }
22   }
23 }
```

2.8 Factory Method Pattern - pc or server?

- *What it does*: when we don't know what type of object we want to create. Similar to Builder Pattern - we want to send the creationg of objects to another class, the Factory-class.
- *Contains:* superclass with multiple subclasses and based on input, we need to return one of the subclasses. Superclass can be an interface, abstract or normal class.
- *Implementation*: Factory-class has no constructor, only a Factory-method which returns an object.
- *Benefits*: removes instancing of client code. Makes our code more robust, **loose coupling** and easy to extend.

`Example`: Create a PC or server-object, depending on what client code will send. ComputerFactory has a method where either PC or server-object will be created.

2.9 Singleton Pattern

- *What it does*: defines a type of an object which we can create once, and once only.
- *Benefits*: doesn't pollute global namespace and enables lazy initialization.

```
class Singleton {

    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
            return instance;
        }
        throw new RuntimeException("Object already accessed"
            );
    }

    private Singleton  () {} //Empty constructor

}
```

We can only call Singleton.getInstance() once, so if we want to use our singleton object we need to pass it as a parameter to our methods (or store a reference to it as an attribute).

2.10 Facade Pattern - receptionist

- *What it does*: Client communicates with ONE class and that class has access to other classes. Imagine a receptionist at a hotel who communicates to others (booking system, roomservice, etc).
- *Benefits*: Simplifies the usage of a more complicated interface.

2.11 Optional Pattern - method chaining

- *What it does*: handles the absence of values (*null*).
- *Benefits*: makes the code more "safe" so it doesn't crash when null is present.
- **map**: applies a lambda expression on the value that is encapsulated in an Optional.
- **flatMap**: same function as map except that it returns a new Optional of some kind - we want to avoid having an Optional<Optional<...>>, flatMap 'elevates' the inner value by one step.
- **filter**:
- Use **Optional<...>** in method when they ask for:
  IF there exist...

`Example`:

```
interface Book {
    String title();
    String isbn();
    Optional<Author> author();      //empty if
                                    //author is missing
}

interface Author {
    String name();
    Optional<Address> address();     //empty if author
                                     //doesn't have address
}

interface Address {
    String streetAddress();
    String zipCode();
}

interface Library {
    Optional<Book> bookByIsbn(String isbn);  //empty if
                                             //no book exist
}
```

```
String zipCodeOfAuthor(Library library, String isbn) {
    return
        library //Method-chaining, return first parameter in
            method above
        .bookByIsbn(isbn)
        .flatMap(book -> book.author())
        .flatMap(author -> author.address())
        .map(address -> address.zipCode()) //End with .map
        .orElse("Unknown zipcode");
}
```