Design patterns

Using class associations

Classes

Collection of

- ***Data members*** – to store class or object state
- ***Methods*** – to change and read object state or perform state independent operations

Organization rules

- **Encapsulation** – logically related data and operations
- **Abstraction** – hide implementation, interface of a certain abstraction level
- **Single responsibility** – class is responsible for one task (abstraction dependent)

Class hierarchy

- Reuse of code
- **Inheritance**: Extension of successor class
- **Polymorphism**: change inherited behaviours
- Reflects abstraction hierarchy (general – special connection)

Abstract classes

- Abstract members do not represent business objects, because they are „too generic”. **Can not be instantiated**.
- Body of abstract methods can not be defined.

- To define classes with abstract method
- To specify classes as abstract

Interfaces

- Associations of classes couple (bind) them to each other

- Create abstraction independent connection in class hierarchy
- Make classes independent (decouple)

Component association

- Using association, logically an object can contain other objects
- Other words: object can be composition of other (component) objects
- Unlike inheritance, ask with question: HAS-A?

- Consultation: calling of other object
- Composition: create object from others
- There is no reference to the component out of the container

Objects

Represent items of the real world (business or system)

Objects are **instance**s of classes

All objects of the same type have:

- Custom values of common properties – instance data, instance state
- Shared methods, working on instance data
- Controlled state transitions – consistent internal state descriptors

Link restrictions

- Exactly one: 1
- Zero or one: 0..1
- Zero or more: 0..*
- One or more: 1..*
- Interval: 3..8
- Discrete values: 2, 4, 6, 8

Link **data member**	Ctor, get, set
Link **data member** with NULL value	No restriction
ArrayList, can be empty	No restriction
ArrayList, can NOT be empty	Ctor, add, remove
ArrayList with controlled number of items	Ctor, add, remove
ArrayList with controlled number of items	Ctor, add, remove

Design patterns

Usage of associations

Usage of associations

- Design patterns are predefined and accepted structures
- To achieve a certain optimization aim
- Multiple patterns can be used parallel (combine them)
- There is no ground truth for combination (depending on optimization)

Delegation

- In object-oriented programming, **delegation** refers to evaluating a member (calling a method) of one object (the receiver) in the context of another original object (the sender).
- more precisely referred to as request *forwarding, when receiver has no information about the (context of the) sender*

Delegation example

```
class Worker {
      public int work() { … }
}
class Lazy {
      private Worker worker;
      public Lazy(Worker worker) {this.worker = worker; }
      public int work() { return this.worker.work(); }
}
```

Design patterns

Creational patterns

Singleton

Singleton pattern should be used when we must ensure that only one instance of a class is created and this instance must be available through all the code.

A special care should be taken in multi-threading environments when multiple threads must access the same resources through the same singleton object.

Common usage of singleton

There are many common situations when singleton pattern is used:

- Logger Classes
- Configuration Classes
- Accesing resources in shared mode
- Other design patterns implemented as Singletons:
  Factories and Abstract Factories, Builder, Prototype

Singleton declaration

```
class AppState {
      private static AppState sharedInstance = new
AppState();
      public static AppState getSharedInstance()
      { return sharedInstance; }

      public int numberOfUsers = 0;
      private AppState() { … }
}
```

Singleton usage

```
class AppStateUser {
    public void modifyAppState()
    {

    AppState.getSharedInstance().numberOfUsers++
;
    }
}
```

Factory

- Factory method – abstract method for object creation
- Object Factory – creation of object implementing an interface
- Abstract factory – creation of descendant of an abstract class

Factory method

The factory method is a part of the class using the created object.

This class defines an abstract method for creating object, but let subclasses decide which class to instantiate and refers the newly created object through a common interface.

Created object has to implement the referred interface.

Factory

Along with singleton pattern the factory is one of the most used patterns. Almost any application has some factories.

- Creates objects without exposing the instantiation logic to the client
- Refers to the newly created object through a common interface.

Common usage of Factory

Used when a framework delegate the creation of objects derived from a common superclass to the factory.

With interfaces this is a tool for decoupling the code.

Our method was called factory, but was not indeed a factory method!

Factory example

```
class ConsoleOutputConnectorFactory
        extends OutputConnectorFactory {
    public static OutputConnector createOutputConnector() {
        return new ConsoleOutputConnector();
    }
}
```

```
ConsoleOutputConnector implements OutputConnector
```

Facotry usage

```
class Framework {
        private OutputConnectorFactory factory;
        public Framework(OutputConnectorFactory factory)
        { this.factory = factory; }
        public void connect() {

        factory.createOutputConnector().println(”output”);
        }
}
```

Abstract factory

A factory for creating a family of related objects, without explicitly specifying their classes.

Family means a subbranch of the abstraction hierarchy. A family of related objects is defined by their parent. Because the abstraction, the parent can not be exactly specified, therefore it is abstract.

The facory works with children of an abstract base, not interface implementations.

Builder

In classes that have a lot of fields you oftentimes end up with many constructors as you might need objects using different field combinations. The Builder pattern enables a more readable object creation and let’s you specify the fields that are actually needed.

A *composite* or an *aggregate* object is what a builder generally builds.

Builder vs. Factory

Differences of **`builder`** and **`factory`** patterns

- the builder pattern creates an object step by step
- the factory pattern returns the object in one go.

Prototype

The prototype pattern is used when objects are expensive to create and new objects will be similar to existing objects. It uses the clone method to duplicate existing instances to be used as a prototype for new instances.

Object pool

Basically, object pool is used whenever there are several clients who needs the same stateless resource which is expensive to create.

For example: database connections

Create a connection if needed, put into a pool if released, reuse from pool the needed again – impersonation is necessary

Prototype vs. Object pool

Prototype creates new instances on every cases, based on templates

Object pool reuses objects stored in a pool, creates new, only when pool is empty

Design patterns

Behaviorual patterns

Responsibility chain

When using this method, instead of attaching the sender of a request to its receiver, and couple them tightly.

Giving other objects the possibility of handling the request.

The objects become parts of a chain and the request is sent from one object to another across the chain until one of the objects will handle it.

Exception handling

Command

An object is used to encapsulate all information needed to perform an action or trigger an event at a later time.

More information is not needed and another object, can execute the action.

Allows the parameterization of clients with different requests and Allows saving the requests in a queue.

Iterator

The Iterator is used to traverse a container of data to access the container’s elements without the need to know the underlying structure. Also, new traversal variants can be added without changing the interface of the objects or the data structure itself.

Strategy

The strategy pattern allows grouping related algorithms under an abstraction, which allows switching out one algorithm or policy for another without modifying the client.

Instead of directly implementing a single algorithm, the code receives **runtime** instructions specifying which of the group of algorithms to run.

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Classes differ only in their behaviours

know the same but act differently

State

- The State pattern lets an object alter its behaviour when its internal state changes.
- This pattern is similar to the strategy pattern, but in this case it is decided internally how the objects behaves.
- This is especially helpful if complex conditions define how the object should behave.
- New states can be added independently from existing states.

Observer

Define a one-to-many link between objects so that when one object changes state, all linked objects (dependents) are notified and updated automatically.

This is typically done by calling one of their methods.

The pattern consists of two actors, *the observer* who is interested in the updates and *the subject* who generates the update events.

Observer

A subject can have many observers and is a one to many relationship. However, an observer is free to subscribe to updates from other subjects too.

In case of many subjects and few observers, if each subject stores its observers separately, it’ll increase the storage costs as some subjects will be storing the same observer multiple times.

Observer example

```
class Source implements Observable {
      public void attach(Observer observer) { … }
      public void dettach(Observer observer) { … }
      public void notifyObservers() { … }
}
class Target implements Observer {
      public void update(Observable source) { … }
}
```

Observer usage

```
class Framework {
	public void startObserve(Observable source, Observer target) {
		source.attach(tartget);
	}
	public void update(Observable source) {
		// < update source state >
		source.notifyObservers();
	}
}
```

Design patterns

Structural patterns

Adapter

Adapter allows independent and incompatible classes to work together by converting the interface of one class into another.

In other words, converts the interface of a class into another interface clients expect

This is helpful when third-party code has to be used, and cannot be changed.

Adapter example

If there are two independent classes

- One output as XML
- The other requiring JSON input
- an adapter is needed between the two to make them work

Compsite

The composite pattern allows to treat a group of objects the same way as a single object.

This is for example used in tree-like object structures where a parent node’s operation influences or is dependent on child nodes.

As a client the operation only needs to be called on the parent.

Represent part-whole associations. (as explained earlier)

Decorator

The decorator pattern allows to add functionality to an object at run-time without altering its structure.

This is more flexible than deriving classes.

Decorator

```
interface Output {
    void print();
}
```

Decorator

```
class TextOutput implements Output {
    private int size;
    public TextOutput(int size) {
        this.size = size;
    }
    public void print() {
        System.out.println(”Print with size: ” + size);
    }
}
```

Decorator

```
abstract class Decorator implements Output {
    // Composition
    private Output widget;

    public Decorator(Output widget) { this.widget = widget; }

    // Delegation
    public void print() { widget.print(); }
}
```

Decorator

```
class BorderDecorator extends Decorator {
    public BorderDecorator(Output widget) {
        super(widget);
    }
    public void print() {
        super.print();
        System.out.println("  Decorate with border");
    }
}
```

Decorator

```
class ScrollDecorator extends Decorator {
    public ScrollDecorator(Output widget) {
        super(widget);
    }
    public void print() {
        super.print();
        System.out.println("  Decorate with scroll");
    }
}
```

Decorator

```
public class Client {
    public static void main(String[] args) {
        Output widget =
            new BorderDecorator(
                new ScrollDecorator(
                    new TextOutput(24)
                )
            );
        widget.print();
    }
}
```

Proxy

Using this pattern the object is a proxy to something else and can control the creation and access of it. The proxy could interface to anything, a large object in memory, file, or other resources, but the Proxy is more lightweight.

Proxy

```
public class RealImage implements Image {
  public RealImage(URL url) {
    loadImage(url);
  }

  public void display() { … }

  private void loadImage(URL url)  {
    //do resource intensive operation to load image
  }
}
```

Proxy

```
public class ProxyImage implements Image {
  private URL url;
  public ProxyImage(URL url) {
    this.url = url;
  }

  public void displayImage() {
    RealImage real = new RealImage(url);
    real.displayImage();
  }
}
```