Object Oriented Software Engineering Tutorial on Design Patterns (Part 2)

Fani Deligianni,
Lecturer
School of Computing Science,
University of Glasgow

Exercise 1

Assume you have a class which encrypts data. There are various choices of encryption algorithms:

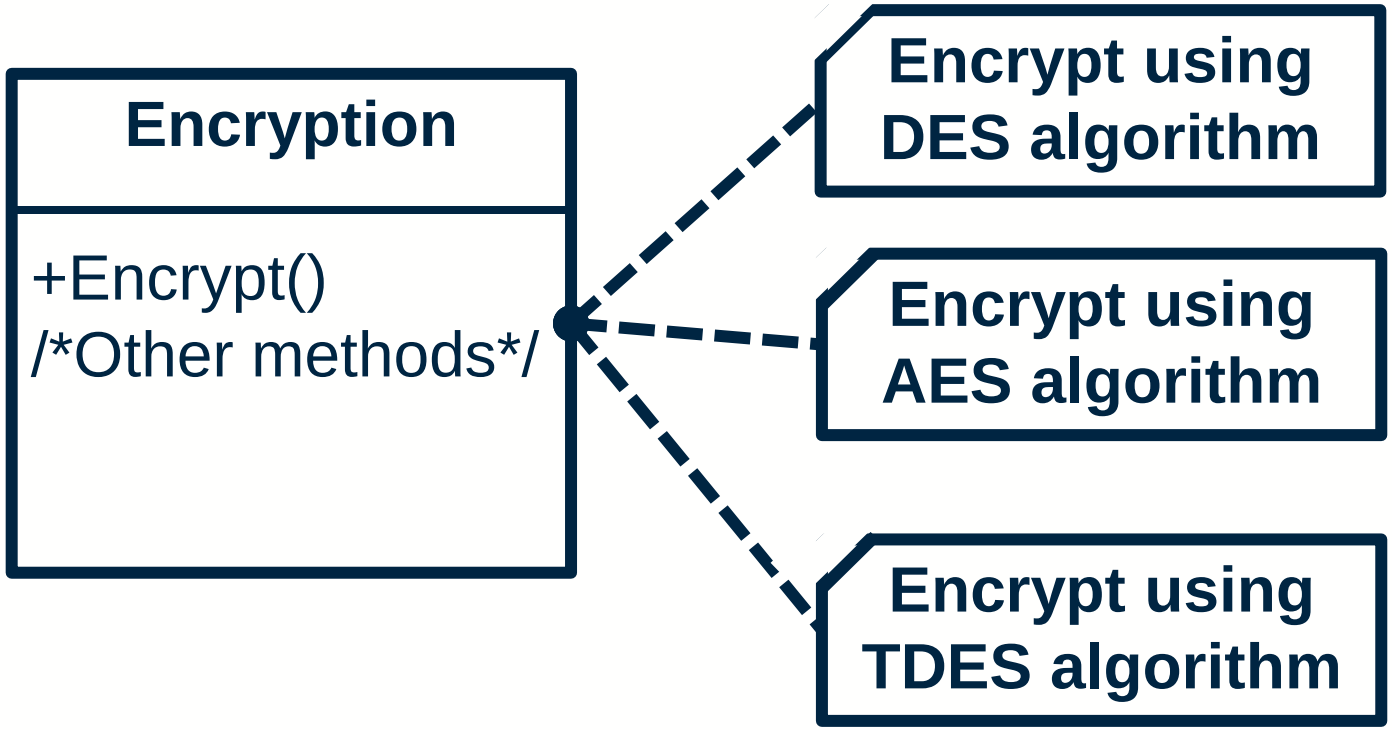
- Data Encryption Standard (DES)
- Advanced Encryption Standard (AES)
- Triple Data Encryption Standard (TDES)

These algorithm also vary according to the data they encrypt: Images or Text or Unstructured data

Explain why the types of Solution A and Solution B are undesirable

Suggest what design pattern could be suitable to refactor these paradigms.

Exercise 1 – Solution A

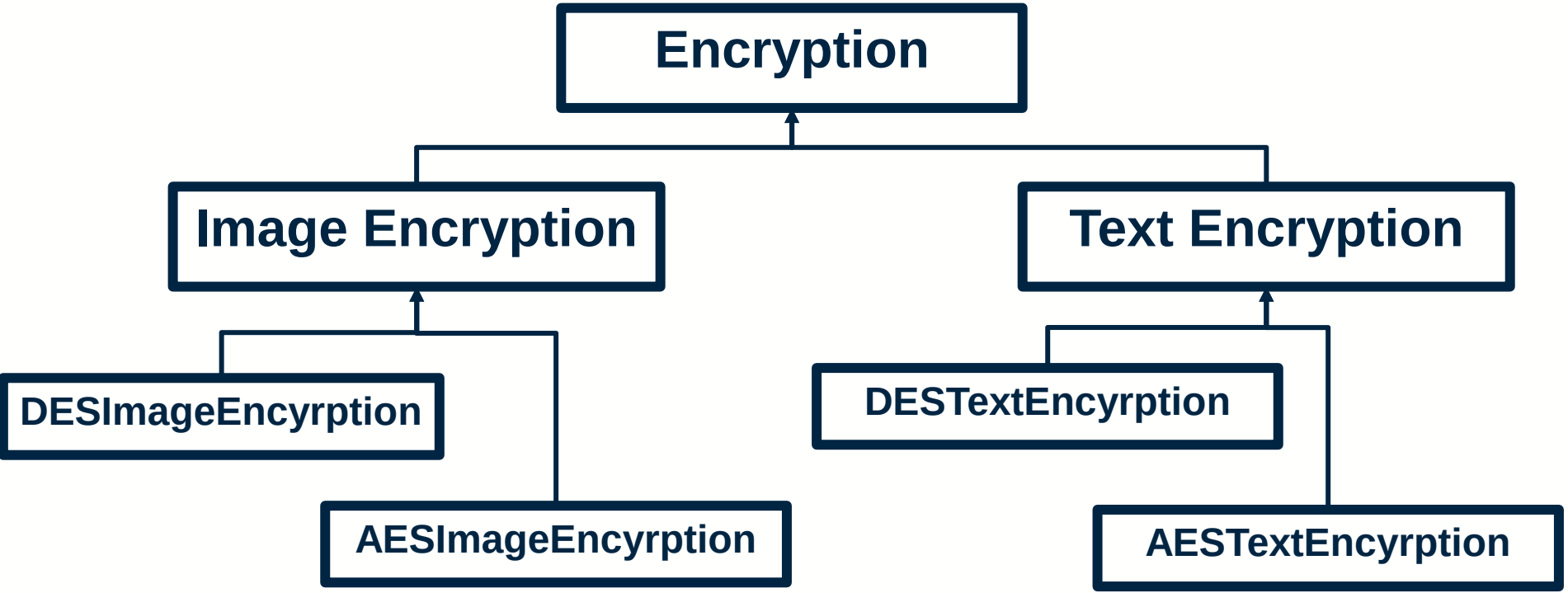


Exercise 1 - Solution

Problems with current solutions:

- The Encryption class becomes bigger and harder to maintain, since it implements multiple encryption algorithms even though only one algorithm is used at a time.
- It is difficult to add new algorithms and to vary the existing ones when a particular encryption algorithm is an integral part of the Encryption class.
- The algorithm implementations are tied to the Encryption class, so they cannot be reused in other contexts.

Exercise 1 – Solution B



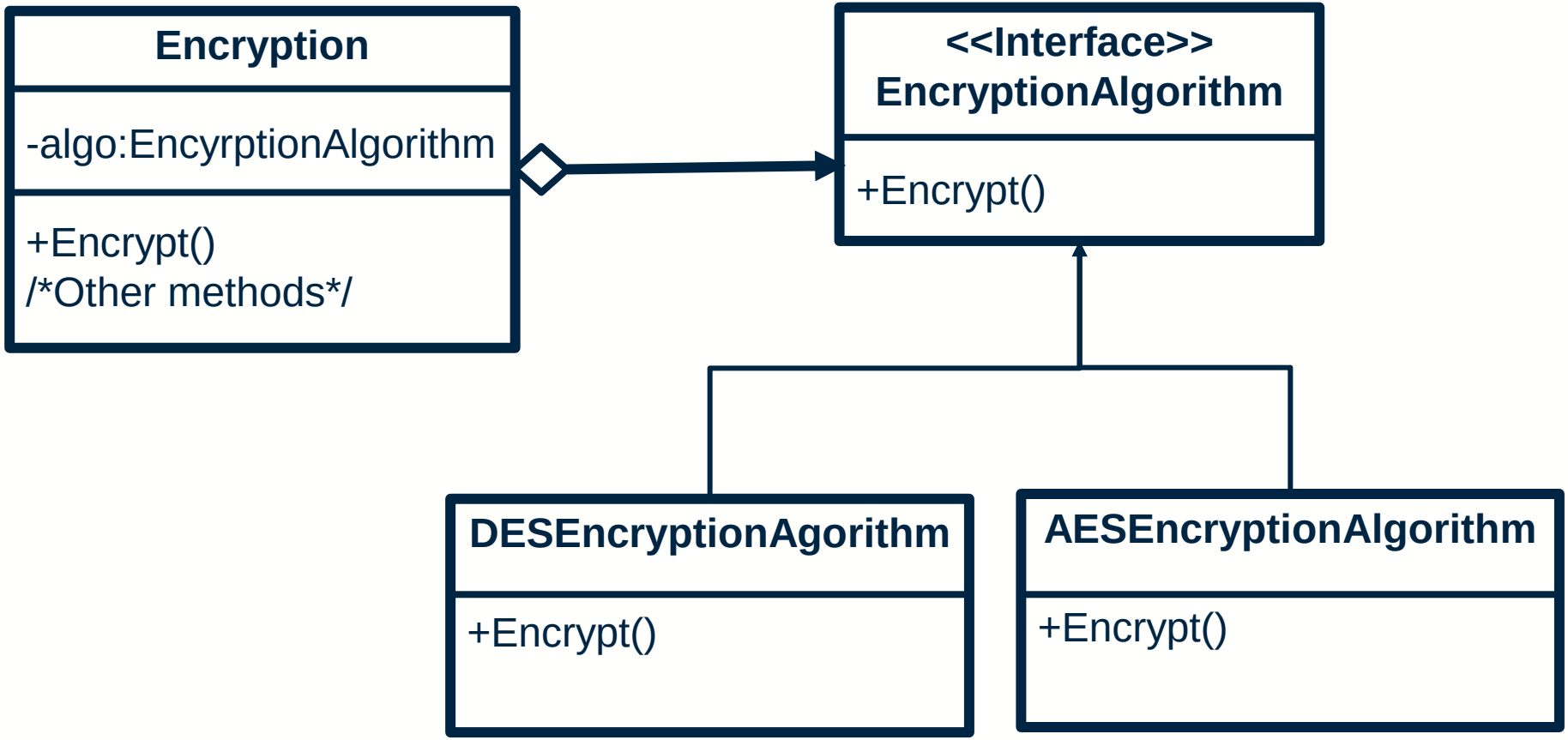
Exercise 1 - Solution

What varies:

- The kind of content
- The kind of encryption algorithm that is supported

Problems:

- How does this design support a new type of content, a new encryption algorithm, or both? (ie. New type of data, new type of content)
- Violates the principle of Open to Extension and Close to Modification


Encryption
-algo:EncyrptionAlgorithm
+Encrypt()
/*Other methods*/
<<Interface>>
EncryptionAlgorithm
+Encrypt()
DESEncryptionAgorithm
+Encrypt()
AESEncryptionAlgorithm
+Encrypt()

Exercise 2

What is the difference between composition and inheritance. Give an example for each.

Exercise 2 - Inheritance

```java
public class Person {
    private final String name;

    // other fields, standard constructors, getters
}

public class Waitress extends Person {

    public String serveStarter(String starter) {
        return "Serving a " + starter;
    }

    // additional methods/constructors
}

public class Actress extends Person {

    public String readScript(String movie) {
        return "Reading the script of " + movie;
    }

    // additional methods/constructors
}
```

- **Inheritance** reflects an is-a solution
- Tight coupling
- Extends functionality

Exercise 2 - Composition

```java
//composition (has-a relationship)
public class Computer {

    private Processor processor;
    private Memory memory;
    private SoundCard soundCard;

    // standard getters/setters/constructors

    public Optional<SoundCard> getSoundCard() {
        return Optional.ofNullable(soundCard);
    }
}

public class StandardProcessor implements Processor {

    private String model;

    // standard getters/setters
}
```

```java
public class StandardMemory implements Memory {

    private String brand;
    private String size;

    // standard constructors, getters, toString
}

public class StandardSoundCard implements SoundCard {

    private String brand;

    // standard constructors, getters, toString
}
```

- **Composition** reflects a has-a solution
- Implements an interface

Exercise 2 - Composition

```
//Composition without abstraction
//Tightly coupled design
public class Computer {

    private StandardProcessor processor
      = new StandardProcessor("Intel I7");
    private StandardMemory memory
      = new StandardMemory("SSD", "256TB");

    //more methods
}
```

Exercise 3

a) Use the strategy design pattern for the following problem:

- Read a file of words
- Print all words that begin with ‘t’
- Print out all words longer than 5 characters
- Print all words that spell the same thing if written backwards (palindromes)

b) With the additional constrain that we don’t want to modify existing code:

- Print how many words it has processed

Exercise 3 - Solution

a) Use the strategy design pattern for the following problem:

- Read a file of words
- Print all words that begin with ‘t’
- Print out all words longer than 5 characters
- Print all words that spell the same thing if written backwards (palindromes)

Strategy Pattern: What is variable in the program?

Exercise 3 - Solution

a) Use the strategy design pattern for the following problem:

- Read a file of words
- Print all words that begin with ‘t’
- Print out all words longer than 5 characters
- Print all words that spell the same thing if written backwards (palindromes)

Strategy Pattern: What is variable in the program?
Answer: Examining a word from the file and determining if it has certain characteristics

```
public interface CheckStrategy
{   public boolean check(String s);
}
```

Interface

```java
public interface CheckStrategy
{   public boolean check(String s);
}

public class StartWithT implements CheckStrategy
{   public boolean check(String s)
    {   if( s == null || s.length() == 0) return false;
        return s.charAt(0) == 't';
    }
}
Public class LongerThanN implements CheckStrategy
{   public LongerThanN(size n) {this.n = n;}
    private int n;

    public boolean check(String s)
    {   if(s == null) return false;
        return s.length() > n;
    }
}
public class Palindrome implements CheckStrategy
{   public boolean check(String s)
    {   if(s == null) return false;
        int length = s.length();
        if(length < 2) return true;
        int half = length/2;
        for(int i = 0; i < half; ++i)
            if(s.charAt(i) != s.charAt(length - 1 - i)) return false;
        return true;
    }
}
```

Interface

Implements different checking Strategies

```java
public interface CheckStrategy
{   public boolean check(String s);
}
```

Interface

```java
public class StartWithT implements CheckStrategy
{   public boolean check(String s)
    {   if( s == null || s.length() == 0) return false;
        return s.charAt(0) == 't';
    }
}
Public class LongerThanN implements CheckStrategy
{   public LongerThanN(size n) {this.n = n;}
    private int n;

    public boolean check(String s)
    {   if(s == null) return false;
        return s.length() > n;
    }
}
public class Palindrome implements CheckStrategy
{   public boolean check(String s)
    {   if(s == null) return false;
        int length = s.length();
        if(length < 2) return true;
        int half = length/2;
        for(int i = 0; i < half; ++i)
            if(s.charAt(i) != s.charAt(length - 1 - i)) return false;
        return true;
    }
}
```

Implements different checking Strategies

```java
public void printWhen(String filename, CheckStrategy which) throws IOException
{
    BufferedReader infile = new BufferedReader(new FileReader(filename));
    String buffer = null;
    while((buffer = infile.readLine()) != null)
    {   StringTokenizer words = new StringTokenizer(buffer);
        while( words.hasMoreTokens() )
        {   String word = words.nextToken();
            if (which.check(word)) System.out.println(word);
        }
    }
}
```

Part that doesn't change

Exercise 3 - Solution

a) Use the strategy design pattern for the following problem:

- Read a file of words
- Print all words that begin with ‘t’
- Print out all words longer than 5 characters
- Print all words that spell the same thing if written backwards (palindromes)

b) With the additional constrain that we don’t want to modify existing code:

- Print how many words it has processed

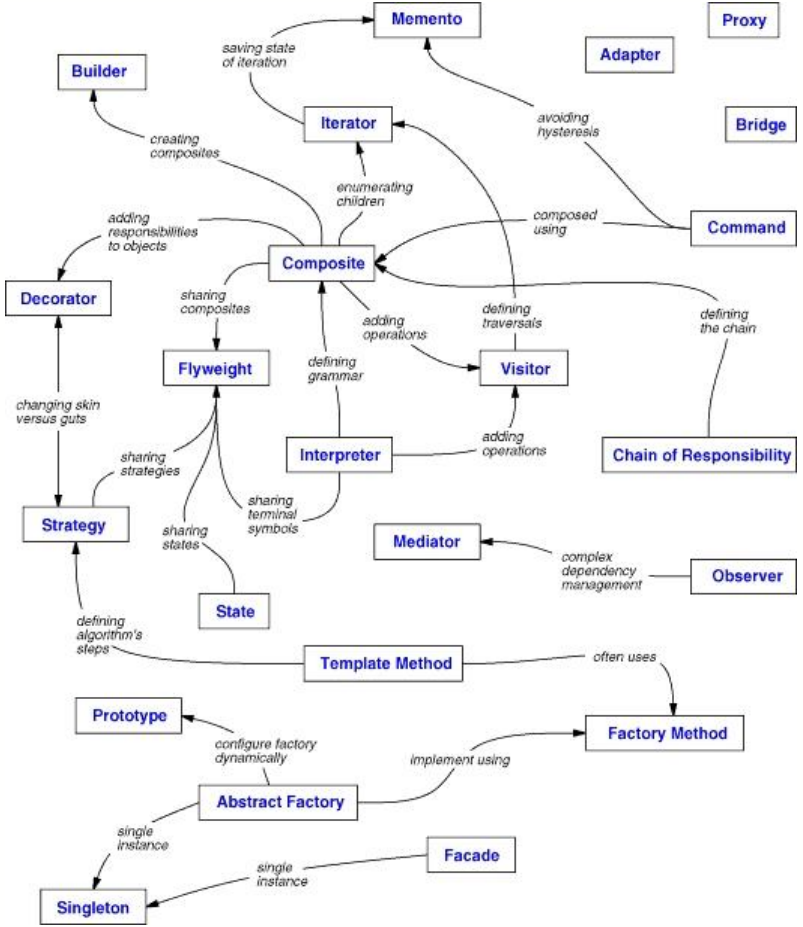
Decorator Design Pattern

```
class CounterDecorator implements CheckStrategy
{   public CounterDecorator(CheckStrategy check)
    {   checker = check;
    }

    public boolean check(String s)
    {   boolean result =  checker.check(s);
        if(result)count++;
        return result;
    }

    public int count(){return count;}

    public void reset(){count = 0;}

    private int count = 0;
    private CheckStrategy checker;
}
```

Strategy and Decorator Pattern

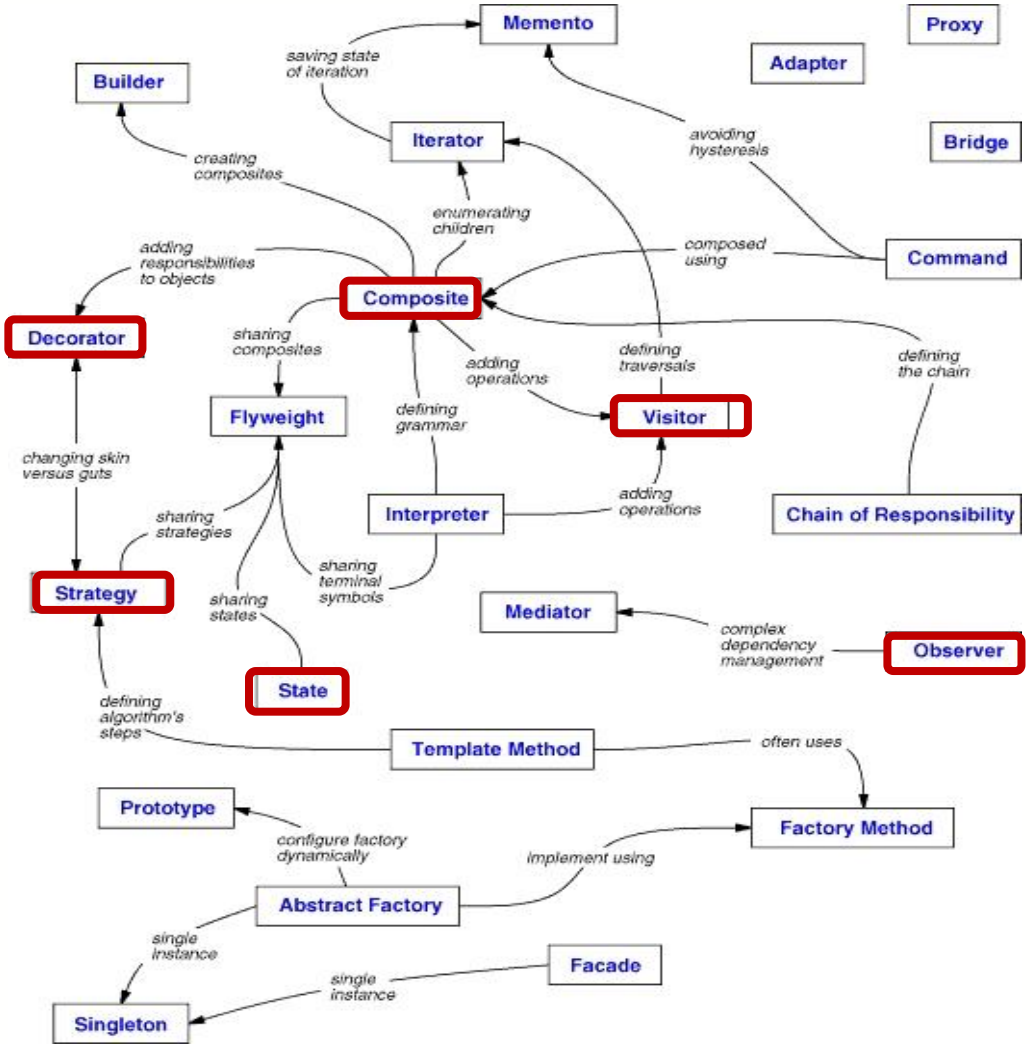
- The key to a **Strategy** is to factor out what might be variable in a set of problems and build an interface for that.
- **Decorators** can provide additional methods beyond what is required/designed by the interface that they decorate.
- The key to **Decorator** is to have a set of objects defined by an interface.
- The **decorator** implements that interface and takes an object of that interface type as a parameter of its constructor,
- When an interface message is sent to the **decorator** it passes sends the same message, or a related one, to the decorated object and gets the result.

Overview



https://www.startertutorials.com/patterns/select-design-pattern.html

Overview


Memento
Proxy
saving state of iteration
Builder
Adapter
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
Strategy
sharing terminal symbols
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

Structural Design Patterns

Composes classes or objects into larger structures

Composite

A tree structure of simple and composite objects

Decorator

Attach additional responsibilities to an object dynamically

Behavioural Design Patterns

How classes and objects interact and distribute responsibility

Visitor

Defines a new operation to a class without change

Strategy

Encapsulate an algorithm inside a class

State

Let the object show other methods after a change of internal state

Observer

Notify observers/ ‘subscribers’ of changes via events

OO-Principles:

- Loose coupling (modularity)
- Prefer composition over inheritance
- Program against interfaces not against implementations
- Open/Close Principle