CS3342 Notes

- W1
    - Software
        - product that SW engineers design and build
        - a set of items or objects that form a **configuration** that include:
            - **programs** (source code)
            - **documents** (design document, user guide) that 'describe the program'
            - **data structure** that 'enable program' to work
        - two role
            - a product: produces, manages, acquires, modifies, displays, or transmits information
            - a vehicle to deliver product: e.g. banking system, os, software tool
        - What is a quality software?
            1. it works
                - SW requirement
                    - meet customer requirement using SW requirement specification:
                        - format of input and output
                        - processing details
                        - performance
                        - error handling procedures
                        - standard
            2. easy to read and understand → help to maintain the code
            3. can be modified → accommodate for new requirements and changes
            4. complete in time and within budget
        - not manufactured (大批生產)
        - doesn't wear out (耗盡的)

        - mostly custom build
        - software cost > hardware cost
        - software maintenance cost > software development cost

    - Software Engineering
        - a processing of solving customers' problems with systematic development and evolution of large, high-quality software systems with time, cost, and other constraints
        - large and complex
        - built by team
        - undergo many change → exist in many version
        - last many years
        - SE models the system VS SP programs the system
        - different behaviour between professional SE and amateurs

- W3
- software development process is a series of predictable steps, road maps, that help us to create a timely, and high-quality result
    - Waterfall model
        - requirement → design → implementation → integration & testing → maintenance
        - Adv.
            - easy & structured & linear
        - Disadv.
            - little feedback from user
            - problem in the specification can be found late
            - take a long time for the first version
            - only used in simple projects when the requirements and technologies are well understood

    - Incremental Process Model
        - provide quick basic functionality to users

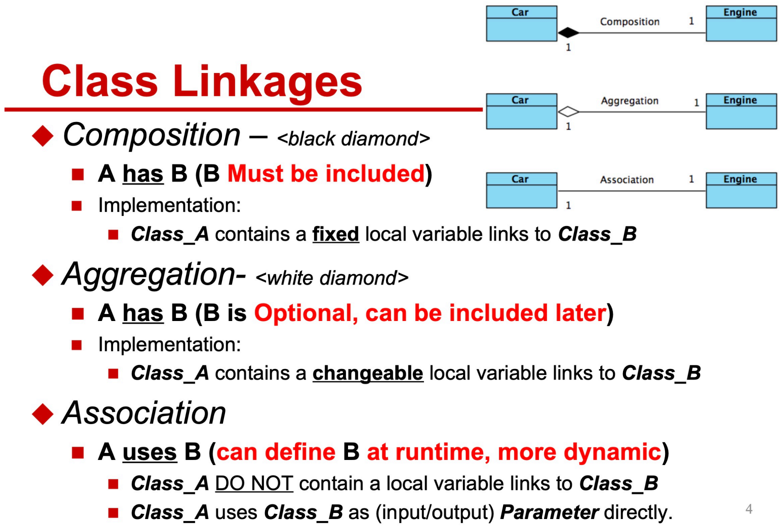
- not linear
- requirements are well defined
- Incremental Model (e.g. Facebook)
  - 1st build CORE functionalities
  - each increment represents a solution
- Rapid Application Development (RAD) Model
  - short development cycle
  - involve multiple teams
  - will fail if the skill of teams is not strong enough

- Evolutionary Process Models
  - Core requirements are well understood
  - additional requirements are evolving and changing fast
  - design most prominent parts first (visual)
  - allow rapid feedback
  - use to develop more complex system
  - X full knowledge of requirement
  - Time estimation is difficult → project completion date may be unknown
  - Prototyping Model
    - Use the prototype to show the user and help refining requirements → better communication
    - quick development
    - used for identify requirement
    - customer may too on9 and think it is a final product
  - Spiral Model
    - Prototype + Waterfall
    - Complexity increase with each release
    - no fixed phrase → more flexible
    - can combine with other model
    - become more complex and hard to manage
  - Concurrent Engineering Model

            - parallel developer
            - using spiral or evolutionary approach

    - Component based software engineering (CBSE)
        - composing solutions from prepackaged software components or classes
        - Frist rule of CBSE:
            - never build what you can buy → become solution oriented
        - Design principles:
            - components are independent → do not interfere with each other
            - hidden implmentations of component
            - communication is through well-defined interfaces (method calls)
        - build up a component library → develop a new system based on components

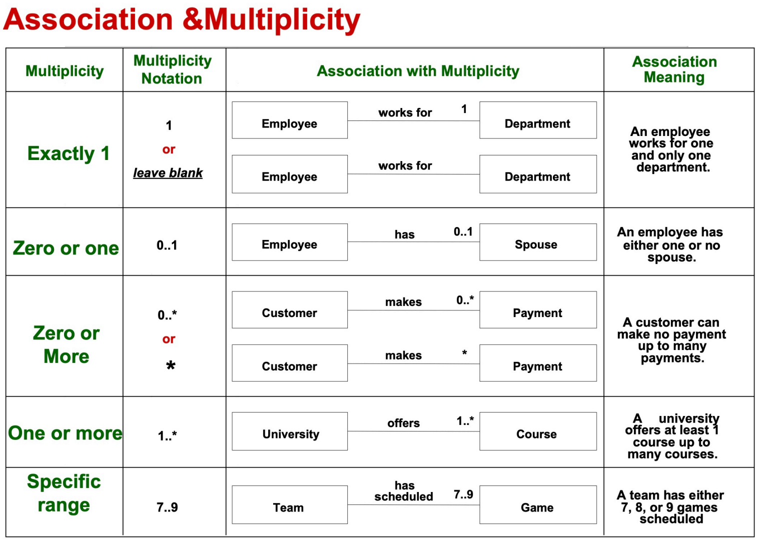
-



Class Linkages

- ◆ *Composition* – *<black diamond>*
    - ■ **A <u>has</u> B (B Must be included)**
    - ■ Implementation:
        - ■ ***Class_A*** contains a **<u>fixed</u>** local variable links to ***Class_B***
- ◆ *Aggregation-* *<white diamond>*
    - ■ **A <u>has</u> B (B is Optional, can be included later)**
    - ■ Implementation:
        - ■ ***Class_A*** contains a **<u>changeable</u>** local variable links to ***Class_B***
- ◆ *Association*
    - ■ **A <u>uses</u> B (can define B at runtime, more dynamic)**
        - ■ ***Class_A*** <u>DO NOT</u> contain a local variable links to ***Class_B***
        - ■ ***Class_A*** uses ***Class_B*** as (input/output) ***Parameter*** directly.

4



Association &Multiplicity

Multiplicity	Multiplicity Notation	Association with Multiplicity	Association Meaning
Exactly 1	1 or *leave blank*	Employee works for 1 Department Employee works for Department	An employee works for one and only one department.
Zero or one	0..1	Employee has 0..1 Spouse	An employee has either one or no spouse.
Zero or More	0..* or *	Customer makes 0..* Payment Customer makes * Payment	A customer can make no payment up to many payments.
One or more	1..*	University offers 1..* Course	A university offers at least 1 course up to many courses.
Specific range	7..9	Team has scheduled 7..9 Game	A team has either 7, 8, or 9 games scheduled

- W4
    - Role of variable
        - **Constant**: initialise without change and calculation
        - **Stepper**: stepping through value that can be predicted (e.g. count)
        - **Most-recent holder**: hold the latest value
        - **Gatherer**: accumulate the individual value (e.g. sum)
        - **Transformation**: get new value from some calculation from value of other variable
        - **One-way flag**: two-valued variable that can be switch to the initial value after change (e.g. isHappen)
        - **Temporary**: hold some value for a short time (e.g. temp)
        - **Organizer**: array
    - Use Case Specification
        1. **Use case-name**
        2. **Actor(s)**
        3. **Description**: state clearly the purpose of the use-case and what trigger the use-case
        4. **Typical course of events**: interactions between actors and the system
        5. **Alternative course of events**

6. **Pre-condition**
7. **Post-condition**

- W6
  - Design Principles (SOLID+L)
    - Open-Closed Principle (OCP)
      - open for extension
      - closed for modification:
        - core functions should not be changed the extension
        - important attributes should not be directly accessible
      - remove if/then/else
      - use subclassing or polymorphism

    - Liskov Substitution Principle (LSP)
      - all derived subclasses must be completely substitutable for their parent class
      - subclass should not inherit features don't exist in the actual context

    - Dependency Inversion Principle (DIP)
      - a way to achieve OCP
      - high-level modules should not depend on low-level modules but on abstractions
      - should be available to reuse for other appliances
      - invert the dependencies by using **interfaces/abstract class** declared in the upper layer
      - high-level modules owns the interface
      - low-level classes should implement toward their interface

    - Single Responsibility Principle (SRP)
      - a object class should have single purpose
      - for future maintenance and upgrades

            - Interface Segregation Principle (ISP)
                - client class should not be forced to depend upon useless interfaces

                  → divide a large interface into a set of smaller interfaces

                  → isolate interface with different purposes

            - Law of Demeter (LoD)
                - each unit should only have limited knowledge about 'closely' units

        - General Design Principles
            - Cohesion (high)
                - ability for a module to work independently
                - should have a single entry and a single exit
                - using:
                    - ISP
                    - LSP
                    - SRP
            - Coupling (low)
                - accomplished by using interface between modules
                - enables data to be passed from one module to another
                    - OCP
                    - DIP
                    - LoD
            - Golden Rules
                - Low coupling between every two package or classes → lower linkage
                - High cohesion within a package or class

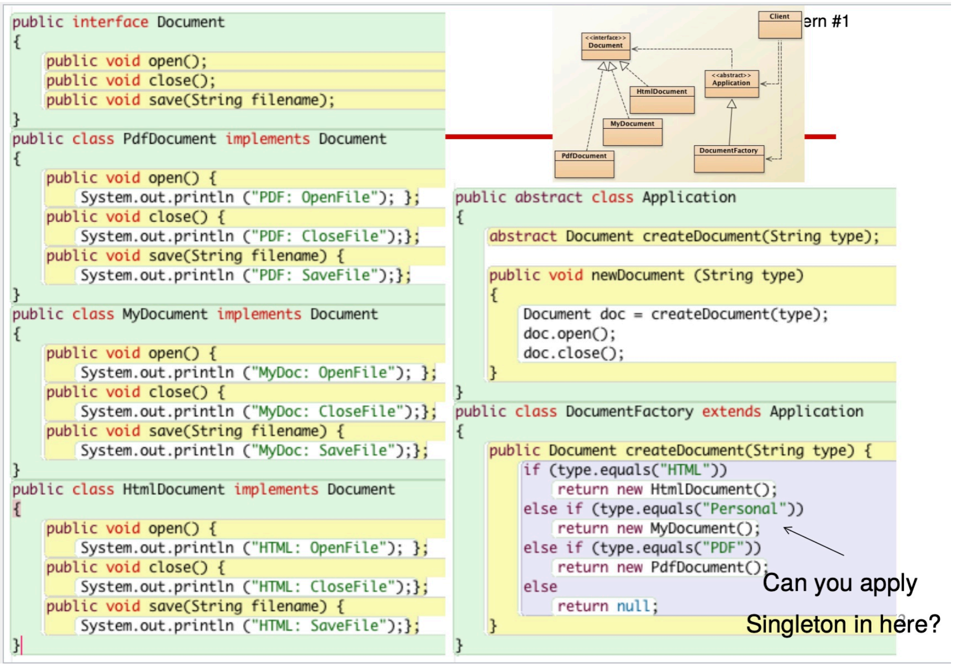
- W8
    - **State Pattern**
        - use interface, e.g. membership

- Pros:
    - Consolidate
    - Consistent: reduce usage of different logic
    - Allow state change: Avoids inconsistent states to reduce complexity
- Cons:
    - Number of object increase

- **Strategy Pattern**
    - similar to state pattern
    - but only one function in State

- **Singleton Pattern**
    - Ensure that only 1 object instance is ever created

- **Factory-Method Pattern**



```
public interface Document
{
    public void open();
    public void close();
    public void save(String filename);
}
public class PdfDocument implements Document
{
    public void open() {
        System.out.println ("PDF: OpenFile"); };
    public void close() {
        System.out.println ("PDF: CloseFile");};
    public void save(String filename) {
        System.out.println ("PDF: SaveFile");};
}
public class MyDocument implements Document
{
    public void open() {
        System.out.println ("MyDoc: OpenFile"); };
    public void close() {
        System.out.println ("MyDoc: CloseFile");};
    public void save(String filename) {
        System.out.println ("MyDoc: SaveFile");};
}
public class HtmlDocument implements Document
{
    public void open() {
        System.out.println ("HTML: OpenFile"); };
    public void close() {
        System.out.println ("HTML: CloseFile");};
    public void save(String filename) {
        System.out.println ("HTML: SaveFile");};
}
```

```
public abstract class Application
{
    abstract Document createDocument(String type);

    public void newDocument (String type)
    {
        Document doc = createDocument(type);
        doc.open();
        doc.close();
    }
}
public class DocumentFactory extends Application
{
    public Document createDocument(String type) {
        if (type.equals("HTML"))
            return new HtmlDocument();
        else if (type.equals("Personal"))
            return new MyDocument();
        else if (type.equals("PDF"))
            return new PdfDocument();
        else
            return null;
    }
}
```

Can you apply Singleton in here?

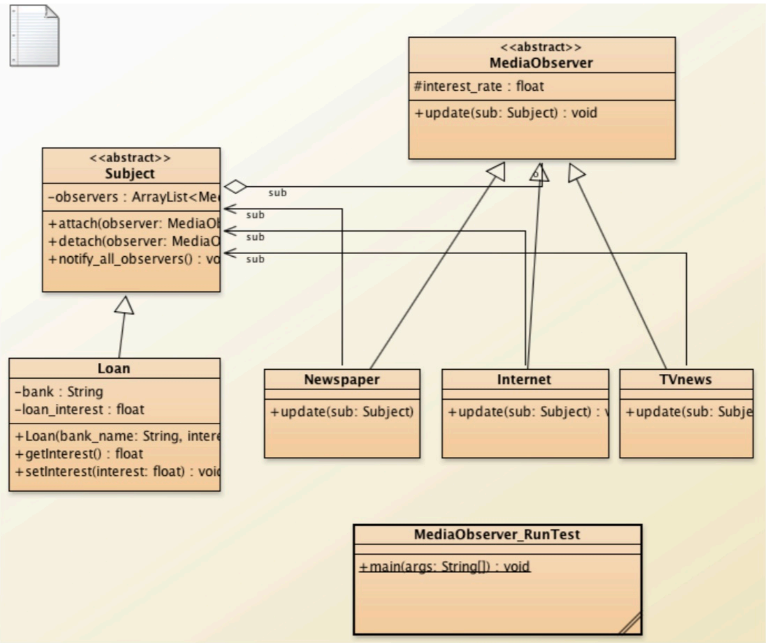
- Client:

```
Application app = new DocumentFactory();
/.. get user_input ../
```

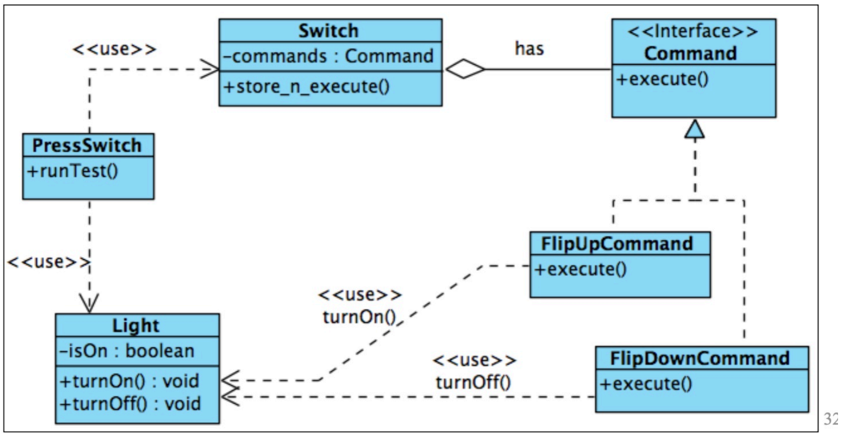
```
app.newDocument(user_input);
```

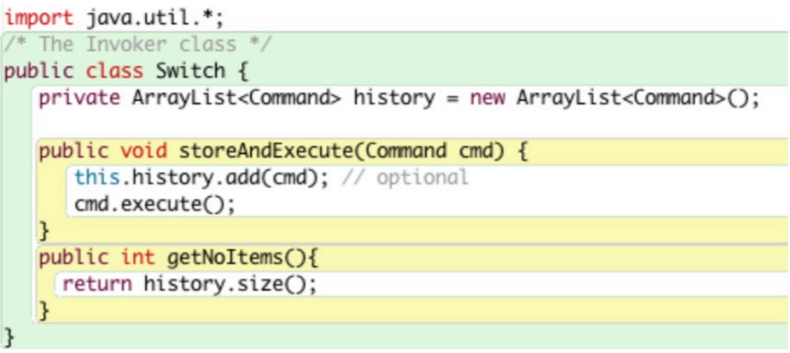
- **Façade Pattern**
    - A simplified interface (front-end) to other code
    - no back-end (no detail code)

- W9
    - **Observer Pattern**
        - separate presentation layer with the data layer
        - three step cycle:
            1. Attach/Register Observers (many)
            2. Notify all attached Observers (many)
            3. Observers calls back the Server for new updates
        - Pros:
            - Consistent
            - avoid tight coupling between object without knowing each other
            - support broadcast communication
            - unexpected updates

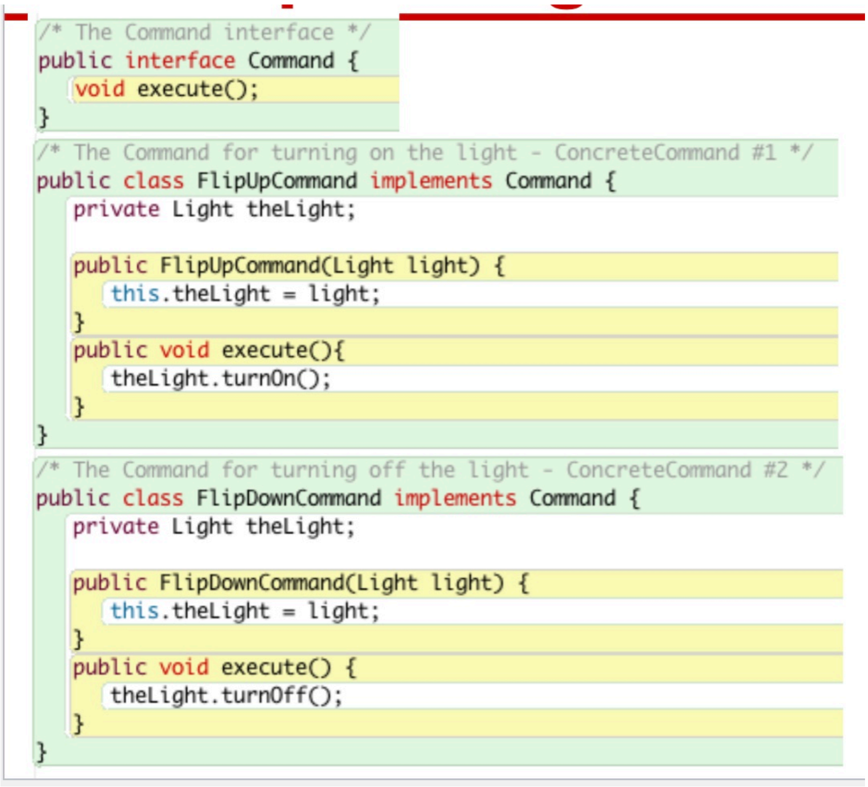


- **Command Pattern**
    - has:
        - Command (interface), ConcreteCommand
        - Invoker (ask the command to execute the request)
        - Received (performs actual actions)
        - Client (e.g. main() function)

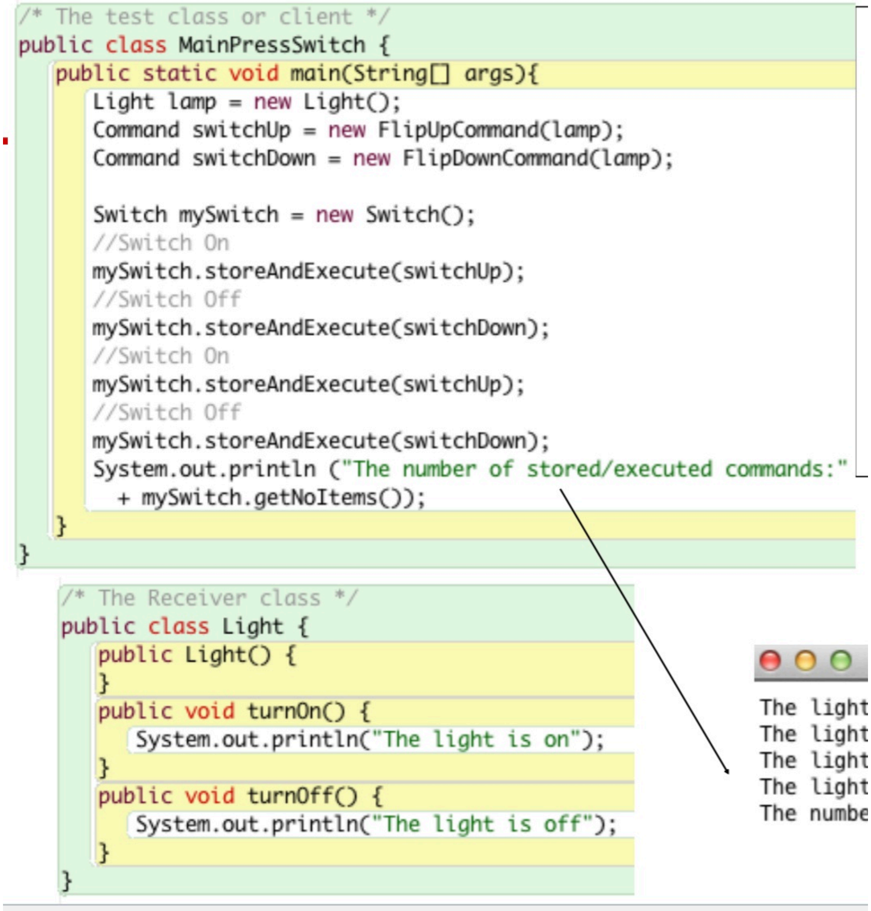




```
import java.util.*;
/* The Invoker class */
public class Switch {
   private ArrayList<Command> history = new ArrayList<Command>();

   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.execute();
   }
   public int getNoItems(){
     return history.size();
   }
}
```



```
/* The Command interface */
public interface Command {
   void execute();
}
/* The Command for turning on the light - ConcreteCommand #1 */
public class FlipUpCommand implements Command {
   private Light theLight;

   public FlipUpCommand(Light light) {
      this.theLight = light;
   }
   public void execute(){
      theLight.turnOn();
   }
}
/* The Command for turning off the light - ConcreteCommand #2 */
public class FlipDownCommand implements Command {
   private Light theLight;

   public FlipDownCommand(Light light) {
      this.theLight = light;
   }
   public void execute() {
      theLight.turnOff();
   }
}
```



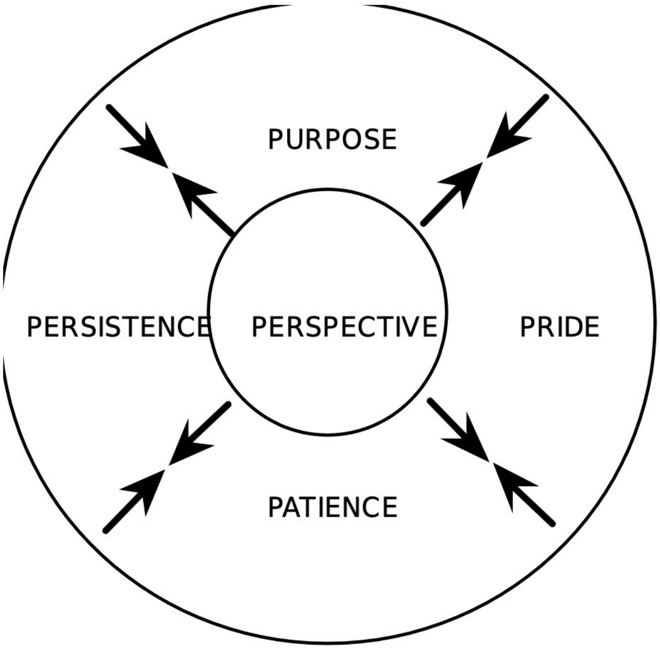
```
/* The test class or client */
public class MainPressSwitch {
   public static void main(String[] args){
      Light lamp = new Light();
      Command switchUp = new FlipUpCommand(lamp);
      Command switchDown = new FlipDownCommand(lamp);

      Switch mySwitch = new Switch();
      //Switch On
      mySwitch.storeAndExecute(switchUp);
      //Switch Off
      mySwitch.storeAndExecute(switchDown);
      //Switch On
      mySwitch.storeAndExecute(switchUp);
      //Switch Off
      mySwitch.storeAndExecute(switchDown);
      System.out.println ("The number of stored/executed commands:"
        + mySwitch.getNoItems());
   }
}
```

```
/* The Receiver class */
public class Light {
   public Light() {
   }
   public void turnOn() {
      System.out.println("The light is on");
   }
   public void turnOff() {
      System.out.println("The light is off");
   }
}
```

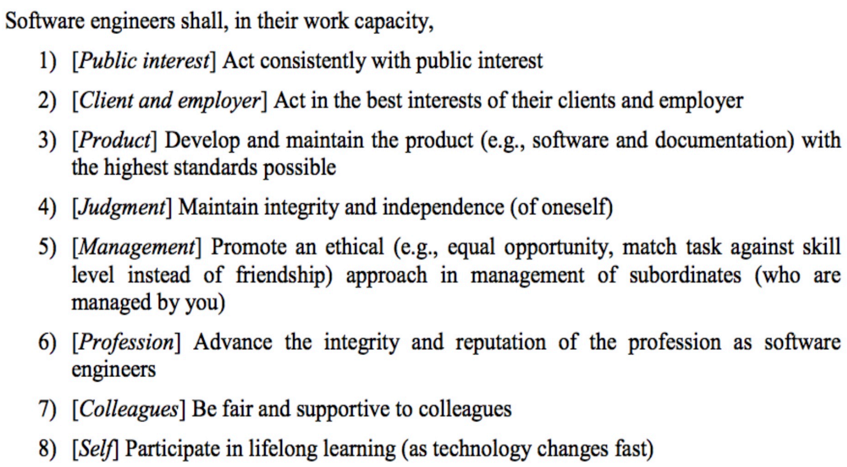
```
The light
The light
The light
The light
The numbe
```

- W10

- Ethics
    - Four Virtues:
        - Prudence (慎重): thicking about moral problem
        - Temperance (節制): suppressing emotions
        - Fortitude (剛毅): not moving blindly away from something we do not like
        - Justice (正義): act in truth and with fairness
    - Evaluation Process: Five P's:
        - Purpose: what's the objective?
        - Pride (自豪感)
        - Patience
        - Persistence (把持/堅持)
        - Perspective (宏觀透視)



    - Code of Ethics in Software Engineering



Software engineers shall, in their work capacity,

1) [*Public interest*] Act consistently with public interest
2) [*Client and employer*] Act in the best interests of their clients and employer
3) [*Product*] Develop and maintain the product (e.g., software and documentation) with the highest standards possible
4) [*Judgment*] Maintain integrity and independence (of oneself)
5) [*Management*] Promote an ethical (e.g., equal opportunity, match task against skill level instead of friendship) approach in management of subordinates (who are managed by you)
6) [*Profession*] Advance the integrity and reputation of the profession as software engineers
7) [*Colleagues*] Be fair and supportive to colleagues
8) [*Self*] Participate in lifelong learning (as technology changes fast)

- Inheritance
    - a mechanism that allows a class to inherit properties and behaviors from another class.

- Functional Requirement
    - specific what the system should perform

- Non-Function Requirement
    - specific how the system perform a certain function