Design Pattern

Lecture 1

Content

- Introduction
- What is Design Pattern? Why use it?
- What is abstraction?
- Principles
  - SRP
  - DIP
- Patterns
  - Factory
    - Simple Factory
    - Factory Method
    - Abstract Factory
  - ~~Observer & event/delegate~~
  - Singleton
- Lazy Loading

Introduction

- Targets
  1. Get ideas of what a pattern is.
  2. Patterns is a flexble **‘programming grammar’**.
  3. When view the source code, you can be associated with patterns.

Introduction

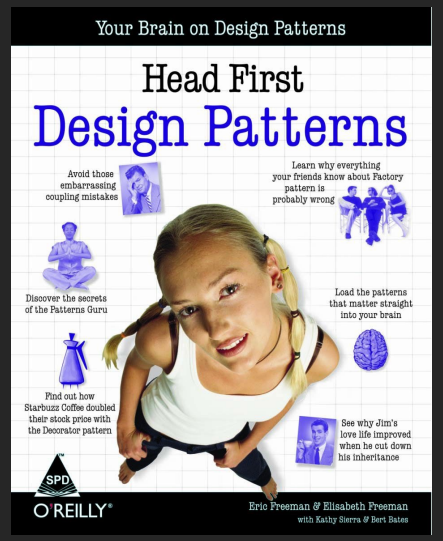
- Lecture

  English slide. Chinses speaking.

  - Main Content: Use a simple example (game) to introduce general usage. And I will try talk about what principles or pattern are used where and why use it.
    - Principle
    - Pattern
  - Conclusion
  - Q&A

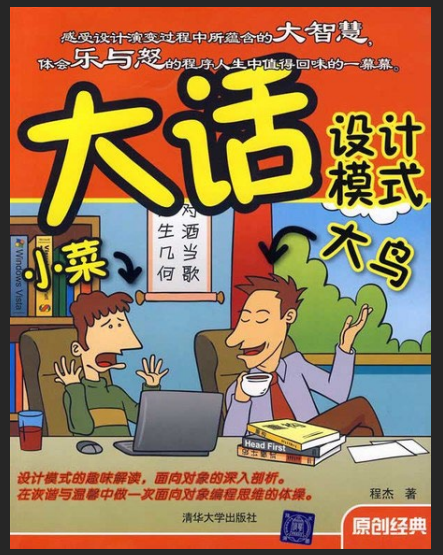
- Feedbacks
  - Anything!!!

References



Head First Design Patterns
Chinese、English

Read together



大話設計模式
Chinese

References

- https://www.notion.so/reikun/d3197e156b754b14890e36a2aa2a2ef7?v=f26241a8237846628c158e5d4f89ff2d
- https://github.com/Talesofwing/DPTutorial
- https://drive.google.com/file/d/1CqL6Sq6a-bgfTyaOKfM-8fHHSPiz7__c/view?usp=sharing

What is Design Pattern? Why use it?

- ~~It is a programming rule?~~ absolutely NO!

- It is a programming idea. In other words, it is an **object-oriented solution**. (Flexible)
- Patterns follow certain principles.
- My Thinking: It is like a programming grammar.

- Example:
  - There is a specific problem needs to be solved.
  - Patterns provide a solution that makes OO.
    - Encapsulation
    - Inheritance
    - Polymorphism
  - Achieve goals of OO
    - Reusability
    - Flexibility
    - Scalability

Principle

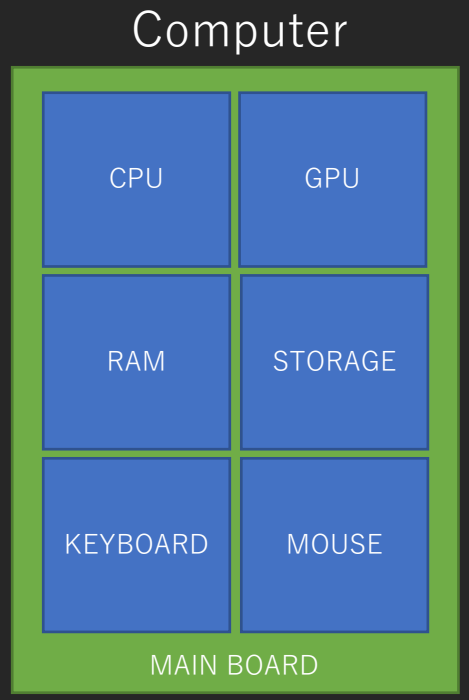
- Principles are like human morality, you should abide by them, but would not be forced into compliance.

- SRP : Single responsibility principle
  - A class should have only one reason to change.

- DIP : Dependency Inversion Principle
  - High level modules should not depend on low level modules; both should depend on abstractions.

- **Repeat! It is not a rule! Just a flexible solution!**

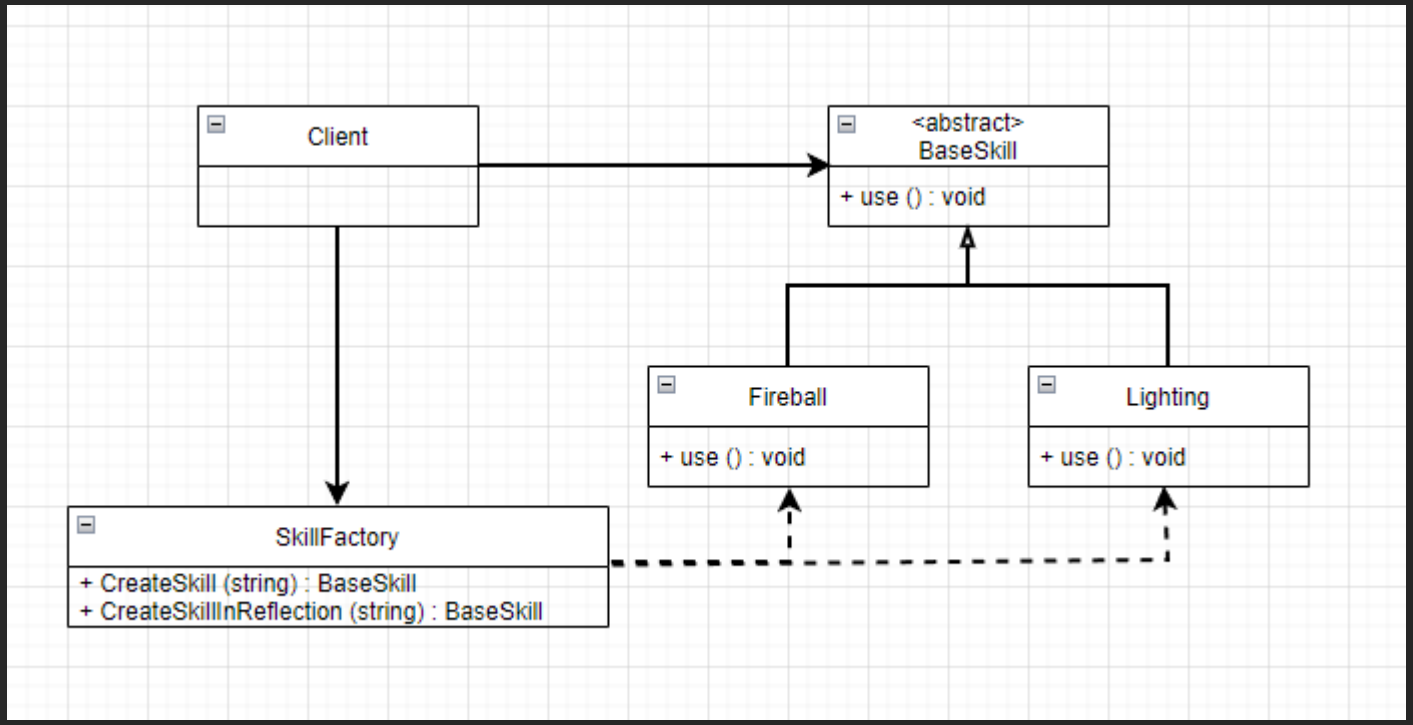
What is abstraction?



- Each socket on the mainboard is an interface.
- The internal process are encapsulated, leaving only interfaces.
- Modular.

Simple Factory

- Some people don’t recognize it as a pattern.
- Encapsulates create the Instance based on the conditionals and returns it.



Simple Factory

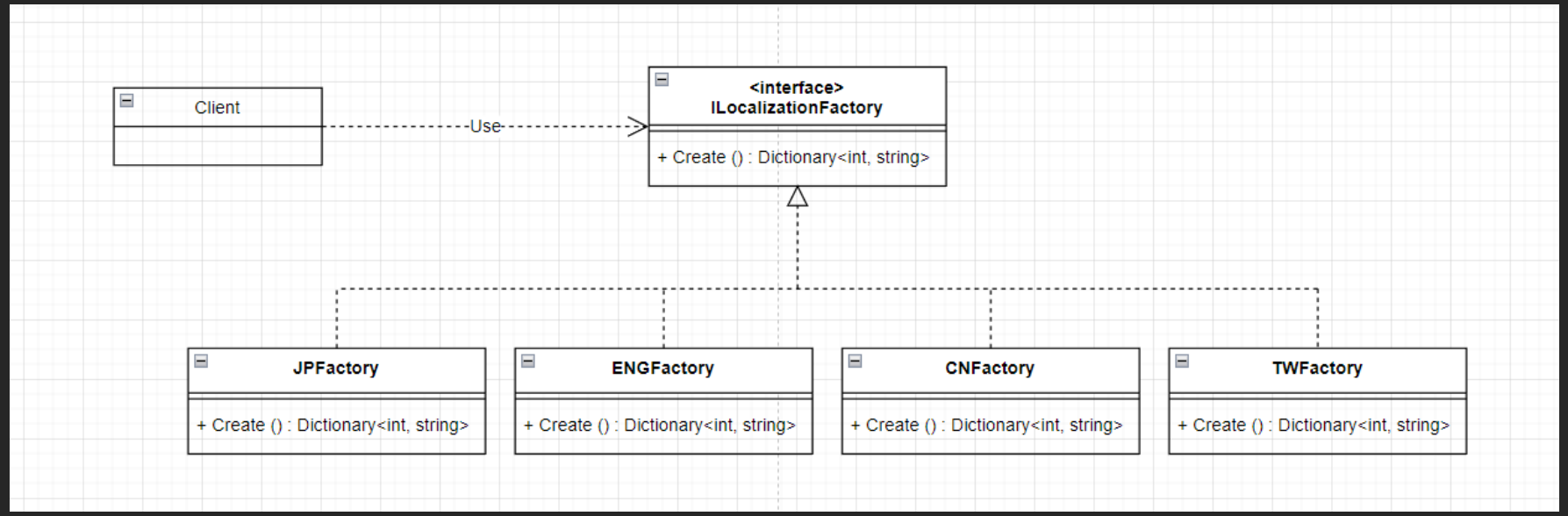
- Advantages
  - Abstract concrete class.
  - Encapsulates the specific creation process.

- Disadvantages
  - When the ‘Create’ method is modified, there is still a lot to change.

- Usage
  - Does not need to pass parameters and only relies on type to create Instances.
  - ‘Create’ method is never changed.

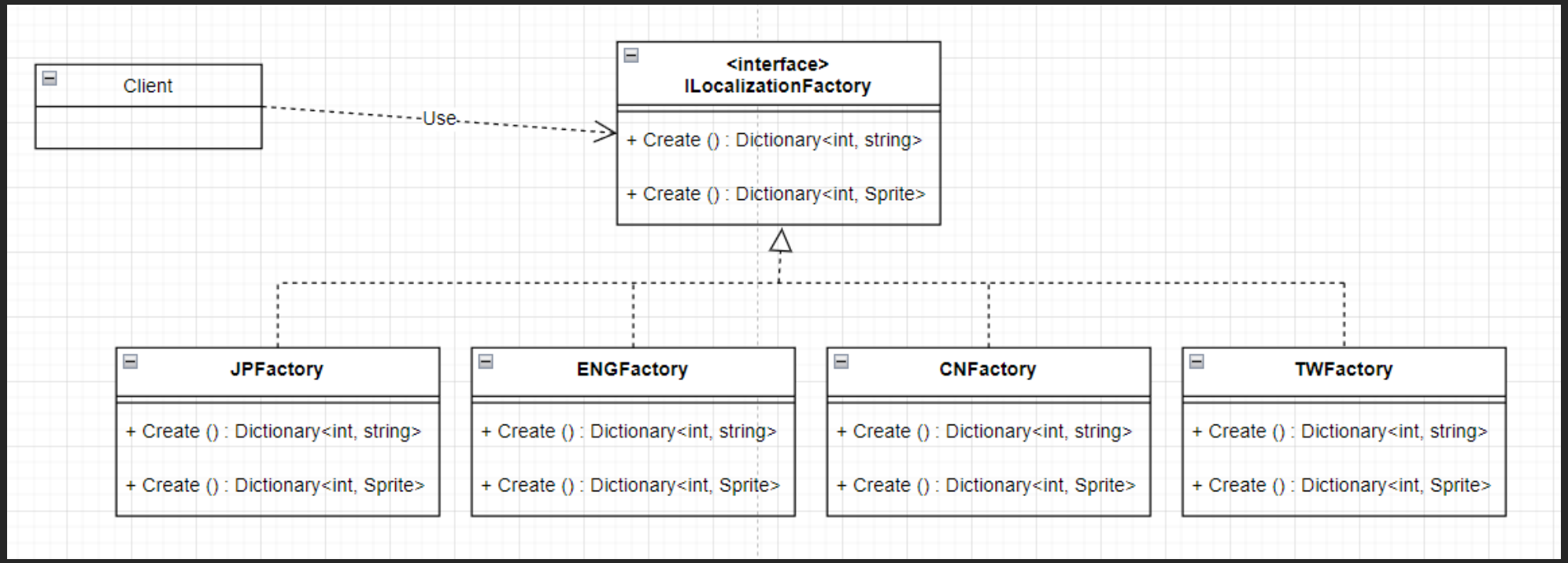
Factory Method

- Defines an interface for creating instances, but the creation process is determined by subclasses.



Abstract Factory

- A collection of factory methods.



Abstract Factory

- Advantages
  - Abstract concrete class.
  - Encapsulates the specific creation process.

- Disadvantages
  - A lot of subclass.

- Usage
  - When different instances need to be created based on the type.
  - Encapsulate the creation of a series of instances in a class.

Q&A

Lazy Loading

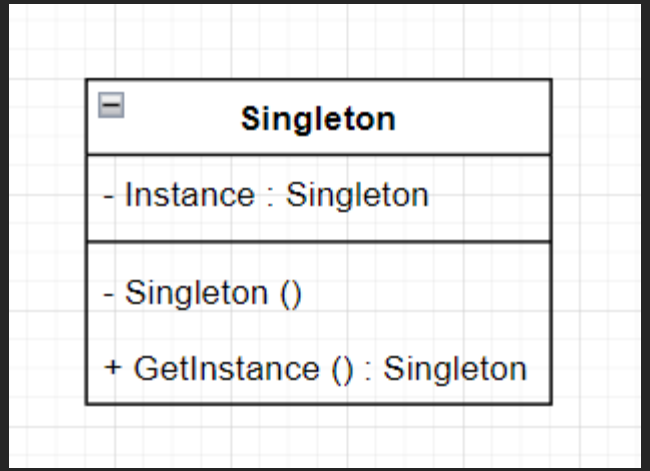
- Load only when needed.
- Often used in singleton loading.

```
private static LocalizationSystem m_Instance;
5 usages
public static LocalizationSystem Instance {
    get {
        if (null == m_Instance) {
            m_Instance = new LocalizationSystem ();
        }

        return m_Instance;
    }
}
```

Singleton

- Ensure that a class has only one instance in its lifetime and provide a global easily access the sole instance of a class.
- Block its instantiation.



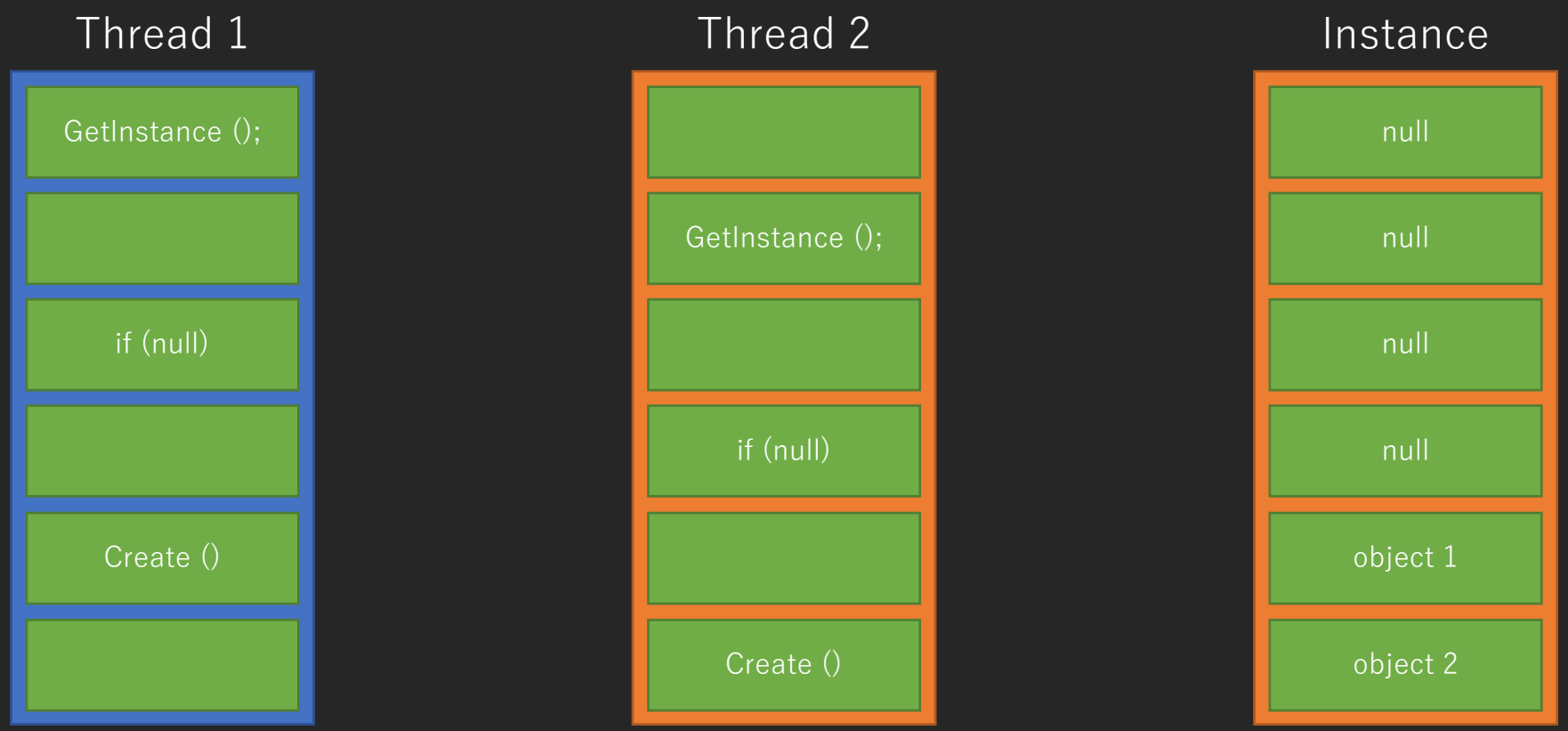
Singleton

- Multithreading considerations

```
private static LocalizationSystem m_Instance;
5 usages
public static LocalizationSystem Instance {
    get {
        if (null == m_Instance) {
            m_Instance = new LocalizationSystem ();
        }

        return m_Instance;
    }
}
```

```
public static Singleton GetInstance () {
    if (m_Instance == null) {
        lock (m_SyncRoot) {
            if (m_Instance == null) {
                m_Instance = new Singleton ();
            }
        }
    }
    return m_Instance;
}
```



Singleton

- Solutions
  - Use ‘lock’
    - Locking is required every time, resulting in a performance penalty
  - Double checked locking
  - Don’t use ‘lazy loading’

```
public static Singleton GetInstance () {
    lock (m_SyncRoot) {
        if (m_Instance == null) {

            if (m_Instance == null) {
                m_Instance = new Singleton ();
            }

        }
        return m_Instance;
    }
}
```

```
public static Singleton GetInstance () {
    if (m_Instance == null) {
        lock (m_SyncRoot) {
            if (m_Instance == null) {
                m_Instance = new Singleton ();
            }
        }
    }
    return m_Instance;
}
```

Singleton vs. Static class

- We can implement an interface with a Singleton class.
- Singleton classes follow the OOP.
- Singleton objects are stored in Heap, but static objects are stored in stack.

- A singleton is an instance that can be passed as a parameter and used as an instance.

Conclusion

- Patterns are OO programming principle.
- Patterns should not be abused to overcomplicate the system.
- It’s normal for beginners to overuse.
- When a system needs to be designed, think about which patterns should be used.

溫故而知新

Q&A