Summary

Definition

leading OO notations

- Grady Booth(BOOCH)
- Jim Rumbaugh(OML: Object Modeling Language)
- Ivar Jacobeson(OOSE: Object Oriented Software Engineering)

UML: Unify Model Language

- Advantage
  - Provide a common, simple, graphical representation of software design and implementation
  - Allows developers, architects and users to discuss the workings of the software
- Type
  - Static Modeling
  - Behavioral Modeling
  - Interaction Modeling

Waterfall Model

- Requirements Engineering
- Design
- Implementation
- Testing(Verification)
- Maintenance

Software Design Principle

- Abstraction
  - Procedural abstraction
  - Data abstraction
- Modularity: structural criteria that tell something about individual modules and their interconnections
  - Cohesion: the glue that keeps a module together
  - Coupling: the strength of the connections between modules
- Information Hiding: A principle for breaking a program into modules
  - Highly related to
    - Abstraction
    - Coupling
    - Cohesion

Decomposition Criteria

- Functional decomposition
- Information hiding decomposition

Design Pattern

- Definition
  - A solution to a problem in a context
  - A language for communication solutions with others
- Best known: "Gang of Four" (Gamma, Helm, Johnson, Vlissides)
- Advantage
  - Provide an abstraction of the design experience
  - Provide a common vocabulary for discussing complete system designs
  - Reduce system complexity by naming abstractions
  - Provide a target for the reorganization or refactoring of class hierarchies

---

Design Principles

- Encapsulate what varies
- Program to an interface, not to an implementation
- Favor composition over inhertance
  - allows you to encapsulate a family of algorithm into a set of class
  - allows you to easily change the behavior at runtime
- Open to extension, close to modification
- Least Knowledge
  - Only invoke methods that belongs to
    - The object itself
    - Objects passed as parameter to the method
    - Any object the method creates or instantiates
    - Any component o the object
  - Do not invoke methods on objects that were returned from calling other methods

---

Refactoring

- A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior
- to restructure software by applying a series of refactoring's without changing its observable behavior

---

Code smells: Indicative of bad software design

- A code smell is a hint that something has gone wrong somewhere in your code.
- A smell is sniffable, or something that is quick to spot.
- A smell doesn't always indicate a problem

Class Diagram

association: 两者之间存在某种关联即可，很弱的关系，如student and course, 每个学生可以选不同的课，每门课上有不同学生；

aggregation: "consist of"：整体与部分之间的关系，但这里部分可以脱离整体单独存在，如MP3上所插的耳机，MP3包含耳机，但这个耳机也可以单独存在，或者插在其他电脑上。

composition: 更强的aggregation，这里部分不能脱离整体而存在，这个部分是整体的私有财产。比如Apple Itouch上的电池，原则不能拆下来单独使用。

Design Pattern

- Strategy Pattern: The Strategy Pattern defines a family of **algorithms**, **encapsulates** each one, and makes them **interchangeable**. Strategy lets the **algorithm vary independently** from the clients that use it.
- Observer Pattern: The Observer Pattern defines a **one-to-many dependency** between objects so that when one object changes state, all its dependences are notified and updated automatically.
- Factory Method Pattern: The Factory Method Pattern defines an **interface** for creating an object but **lets subclasses decide which class to instantiate**. Factory Method lets a class defer instantiation to subclasses.
- Abstract Factory Pattern: The Abstract Factory Pattern provides an interface for **creating families of related or dependent objects without specifying their concrete classes**.
- Singleton Pattern: The Singleton Pattern ensures a class has **only one instance** and provides a **global point** of access to that instance.
- Command Pattern: The Command Pattern encapsulates a **request** as an **object**, thereby letting you parameterize other objects with different requests, queue or log requests, and support **undoable** operations
- Adapter Pattern: The Adapter Pattern **converts the interface of a class into another interface the client expects**. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Façade Pattern: The Façade Pattern provides a **unified interface to a set of interfaces in a subsystem**. Façade defines a **higher-level interface** that makes the subsystem easier to use.
- Bridge Pattern: Bridge Pattern are used to **decouple abstractions from implementations** so that they can vary independently. This type of design pattern is structural in that it decouples abstraction and implementation by providing a bridge between them.
- Flyweight Pattern: The Flyweight Pattern is mainly used to **reduce the number of objects created** to reduce memory footprint and improve performance. This type of design pattern is a structural pattern that provides a way to reduce the number of objects to improve the object structure required by the application.

Design Pattern	Pros	Cons
Strategy Pattern	Favor in Composition instead of Inheritance Program to an interface, not to an implementation Can dynamically change the Algorithms Easy to extend and switch	Need to create many new Strategy classes All the Strategy classes are exposed
Observer Pattern	A loose coupling, Subject doesn't know who is the exact Observer and notify what message Support broadcast communications. The subject sends a notification to all registered observers	If an observed object has many direct and indirect observers, it can take a long time to notify all of them Loop calls among Subject and Observers will cause crashes
Abstract Factory Pattern	Easy to exchange product line Separate concrete class, Client use product through interface instead of concrete class Defer instantiation to subclass Allow creation of families of related objects independent of implementation	One change need many modification to FactoryImpl and create new classes
Singleton Pattern	Ensure every client access only one specific singleton object Save system resource	Concrete class, hard to extend
Command Pattern	A loose coupling, there is no direct coupling between invoker and receiver Easy to extend different kinds of commands Can support undo operation	Need to create many new Concrete command classes
Adapter Pattern	Good flexibility and scalability Provide reusability	Too much use of adapters can make the system very complex
Façade Pattern	Simplify the calling Hierarchy control, information hiding	Has some risk when the subsystems are revised

Code Smell

Code Smell	特点	解决方案
Long Method	过长的方法	Extract method
Long Class	过长的类	Extract class Extract subclass Observer Pattern
Long Parameter List	过长的参数列表	Replace parameter with method **Introduce parameter object** Preserve whole object
Duplicate Code	在同一个类、兄弟类、不相干的关的类中存在相同的代码片段	Extract Method Extract Class **Template Method Pattern** **Strategy Pattern**
Divergent Change	一个类会因为各种原因，以各种不同的方式被更改 当我们每次换一个新的数据库时，`mA()`，`mB()`，`mC()` 就会改变 当我们每次添加一个新的金融工具时，`mD()`，`mE()`，`mF()` 就会改变	Extract Class
Shotgun Surgery	一个小的改变可能会影响到很多的类	Move Method Move field Inline class
Feature Envy	类中的方法似乎对其他类的内部结构比对自己的更感兴趣 最常见的嫉妒对象是数据 一个类反复调用其他类的 getter 和 setter 方法	Extract method Move method Move field
Data Clumps	一团团聚集在一起的数据，在好几个不同的类的字段，参数总是被连在一起	Extract class Preserve whole object Introduce parameter object
Primitive Obsession	拒绝使用一些小的对象，过分强调基本数据类型	Replace data value(s) with object Replace type code with class Replace type code with state/strategy
Long Switch	Switch 语句经常在系统中重复出现	Extract method Move method Replace type code with classes Replace type code with state/strategy **Replace conditional with polymorphism**

Code Smell	特点	解决方案
Lazy Class	有些类随着时间的推移，不怎么调用 / 使用某个类	Collapse hierarchy Inline class
Speculative Generality	有时候我们会去创建一些特殊样例，来处理一些可能甚至都不会发生的事情	Collapse hierarchy Rename method Inline class
Temporary Field	只在某些实例中设置的实例变量 其余的时候，字段是空的，或者（更糟）包含不相关的数据	Extract class **Introduce null object**
Message Chains	当看到一长串的方法调用或临时变量来获取一些数据时发生 一个长串的 string，比如 `x.getA().getB().getC()`	**Hide delegate** Extract method Move method
Middle Man	一个类所做的一切就是将调用传递给另一个对象（不必要的中间人）	Remove middle man Inline method Replace delegation with inheritance
Inappropriate Intimacy	两个类共享了过多的私有变量（这其实是不需要的）	Change bidirectional association to unidirectional association Encapsulate collection
Alternative Classes with Different Interfaces	类在外部样子完全不同，但最终在内部是相同的	Extract superclass **Unify interfaces with adapter**
Refused Bequest	子类继承了它不喜欢也不必要的代码	Push down field Push down method **Replace inheritance with delegation**