

# 软件构造复习笔记

BY: Raymond

声明：未经同意，请勿转载！只供同学们复习使用！

Software Construction Notes		No.	Date
Chapter 1: Views and Quality Objectives of SC			
1.1. Multi-Dimensional Views of SC 软件构造过程中的多维视图			
必背的图，考试必考一道选择			
Moment 时刻	Period 时段		
code-level Component-level <small>构件层次</small> 源代码, AST 抽象语法树 类图 Class Diagram 构建脚本 Build Script	Code-level Component-level Package File, Static Linking, Library, Test case Build Script/Component Diagram	Code-level Component-level 代码演化 Code Churn, Evolution Graph 配置版本 Configuration Item, SCI 配置项	Version 版本 Evolution Graph
Run-time 运行时	Package, Library, Dynamic linking, Configuration, Database, Middleware Network, Hardware/Deployment Diagram	执行栈 Execution stack trace, 多线程 Concurrent multi-threads, 分布式进程 Distributed processes Procedure Call Graph..	事件日志 Event log, 多进程 Multi-processes
Memory dump 内存转储	Message Graph (Sequence Diagram)		
External quality factors 外部质量因素	effects, 用户		
Internal quality factors 内部质量因素	effects → 软件本身/程序员		
关注点 关注 正确性 健壮性 可扩展性 可复用性 兼容性 耐用性 性能	1. 正确性 Correctness - 至高无上 Spec 2. 健壮性 Robustness - 对异常情况的处理 Spec 3. 可扩展性 Extendibility / Maintainability 可维护性 4. 可复用性 Reusability 一次开发，多次使用 5. 兼容性 Compatibility 6. 耐用性 Durability 7. 性能 Efficiency - 总是第一位	7. 可移植性 Portability 在不同硬件上运行 8. 易用性 Easy of use 9. 实用性 Functionality 10. 及时性 Timeliness 11. 可验证性 Verifiability 12. 完整性 Integrity 13. 可修复性 Repairability 14. 经济性 Economy	

No.

Date

Lines of Codes

Internal, LoC, Complexity, Readability, Understandability.  
Clearness, Size.

Tradeoff 折冲.

开发过程中,开发者应该将不同质量因素之间如何做好折中的设计决策和标准明确  
正确性决不折冲”

Easy to understand → Chapter 4. Understandability

Ready for changes → Chapter 5 Reusability:

Cheap for develop → Chapter 6 Extensibility:

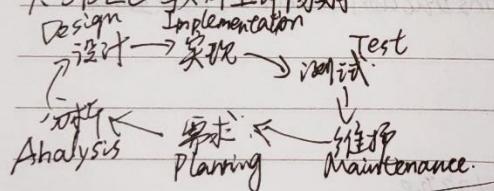
safe from bug → Chapter 7 Robustness / Correctness:

efficient to run → Chapter 8 Efficiency / performance

## Chapter 2. Process and Tools of SC.

### 2.1. Software Lifecycle and Configuration Management. 软件生命周期与配置管理

#### 1. SDLC 软件生命周期



#### 2. 传统的软件过程模型

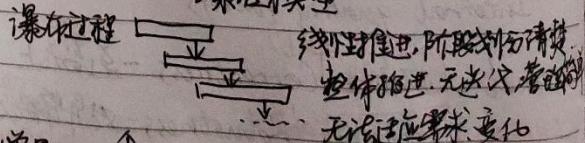
两种基本类型 - 线性过程

迭代过程

现有模型 - 演进过程 增量过程.

V字模型 原型过程.

螺旋模型



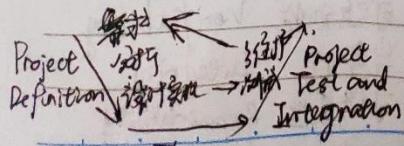
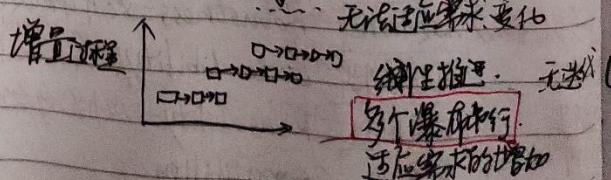
→ 选择合适过程模型的依据.

- 用户参与程度? - 适应变化的能力

- 开发效率 / 管理复杂度.

- 开发出的软件质量.

V字模型 (瀑布模型的应用).



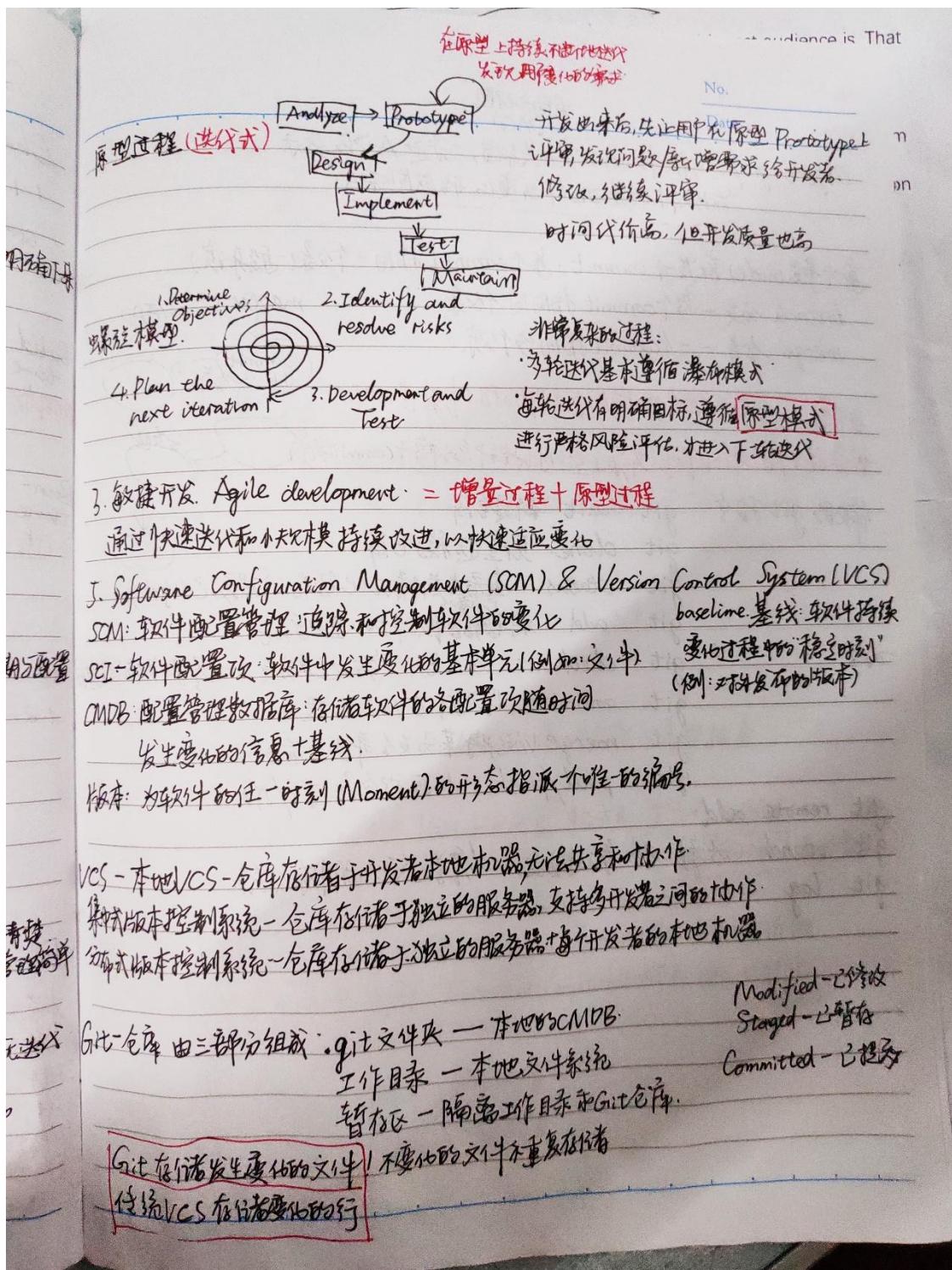
Project Definition

设计

实现

Test and Integration

时间



No.

Date

有向无环图  
(DAG)

Object graph: 版本之间演化关系图，一条边  $A \rightarrow B$  表示  
“在版本 B 的基础上作出演化形成版本 A”

每个节点(node)就是个 commit，每个 commit 指向一个父亲(一般来说)

branch 分支 -> 多个 commit 指向同一个父亲。

merge 合并 -> 1 commit 指向多个父亲。

每个节点 node 存储的是指向具体文件的指针，在增加时。

视图指针的变化即可，节省了空间(指针允许多个 commit 共享)

常用的 git 命令 git init 新建仓库

git clone 克隆(远程)仓库

git status 查看当前情况(已暂存, 已提交, 已修改)

git add 将修改的文件加入至暂存区

git checkout (分支名) 移动到分支 (b 参数能创建新分支)

git commit 将暂存区加入已提交

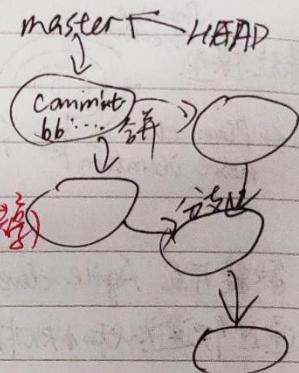
git merge 将某分支合并至当前分支

git pull/push 从远程仓库拉/推

git remote add

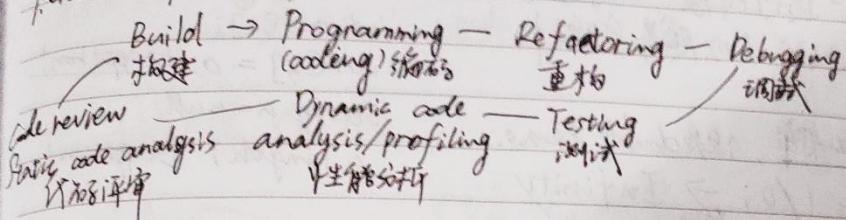
git branch (分支名) 可以新建分支, -d 参数可以删除分支。

git log



## 2.2 Process, Systems, and Tools of SC 软件的过程、系统和工具

### General process of SC



Build - 粗略理解，强调从 build-time 到 run-time 的转化

借助工具，将软件推进各阶段的活动“自动化”

(编译、打包、静态分析、测试、生成文档、部署... 尽可能脱离“手工业”  
提高工作效率)

Build 工具 - Ant, Maven, Gradle

## Chapter 3 ADT & OOP

### 3.1 Data Type & Type Checking 数据类型与类型检查

#### ① Data Type

type 指的是数据类型，而 Variable 变量 格式是 TYPE NAME

用特定数据类型如 String a; 是一个变量

对 Java 有 int, long, short, char, float, double, boolean, byte

基本数据类型 一维值 无 ID (其值无法修改)、Immutable、栈 代价低

对象数据类型 一维值 有 ID、Immutable/mutable 都有 堆 代价高

↳ class / Interface / arrays / enums ...

静态类型语言 - 在编译阶段进行类型检查

动态类型语言 - 在运行阶段进行类型检查

No.                          语义错误  
 Date .                      类/函数名错误.  
                              参数数目/类型错误  
                              返回值类型错误

静态类型检查 - 编译阶段 - 观察类型是否匹配. 以“左进”确定“右出”  
 动态类型检查 - 运行阶段 - 观察运行时的值 以“右进”推“左出”  
 ↳ 非法参数传入(除以0). 越界. 空指针.  
 注意 int 除以0 抛异常, 但是 double 除以0.  
 $\text{double } a = 1/0; \rightarrow \text{Infinity}$   
 输出

改变一个变量 - 将该变量指向另一个值的存储空间.  
 改变一个变量的值 - 将该变量当前指向值的存储空间中写入一个新的值

Immutability. 不变性  
 不变数据类型: 一旦被创建, 其值不能改变; 如果是引用, 则不能再设置指向的对象  
 $\text{final} \rightarrow$  变量不可改变值, 引用/类不可有子类/方法不能被重写/Override.

可变对象: 拥有方法可以修改自己的值/引用  $\rightarrow \text{StringBuilder}$   
 不可变对象: 一旦被创建, 始终指同一个值/引用  $\rightarrow \text{String}$

$\text{String } s = "a"; s += "b";$       snapshot  
 之后新如何  
 $s \rightarrow \text{String } "ab"$   
 $\text{StringBuilder } sb = \dots; sb.append("b"); sb \rightarrow \text{StringBuilder } "ab"$

有多个引用时, mutable 的数据类型  
 就会出现, 使用其中一个引用修改时, 用其它引用得到的结果也是修改后的结果.

- 使用不可变类型, 对其频繁修改会产生大量的临时拷贝(需要GC)
- 可变类型 改变数据以提高效率, 获得更好的性能, 适合在多个模块中共享数据

避免内存泄漏! - 强制式垃圾

用于描述程序运行时的内部状态

便于恢复，监测数据随时间变化，解释设计思路

No.  
Date

叫回溯 Snapshot!

基本类型的值  $\downarrow \downarrow \downarrow \downarrow$   
3 5.0 'c' null.

对象类型的值

Point  
int x  $\rightarrow$  5  
int y  $\rightarrow$  2

Immutable 双线圈

S  
String  
"a"

不可变引用：双箭头

引用不可变，但是其指向的  
值可以改变

Person  
id:  $\rightarrow$   
age  $\rightarrow$  19

例：String s1 = new String("abc");

List<String> list = new ArrayList<String>();  
list.add(s1);

s1 = s1 + "d"; System.out.println(list.get(0))  $\rightarrow$  abc

String s2 = s1 + "e";

list.set(0, s2); System.out.println(list.get(0)).  $\rightarrow$  abcde

list  $\rightarrow$

ArrayList<String>

String  
"abcde"

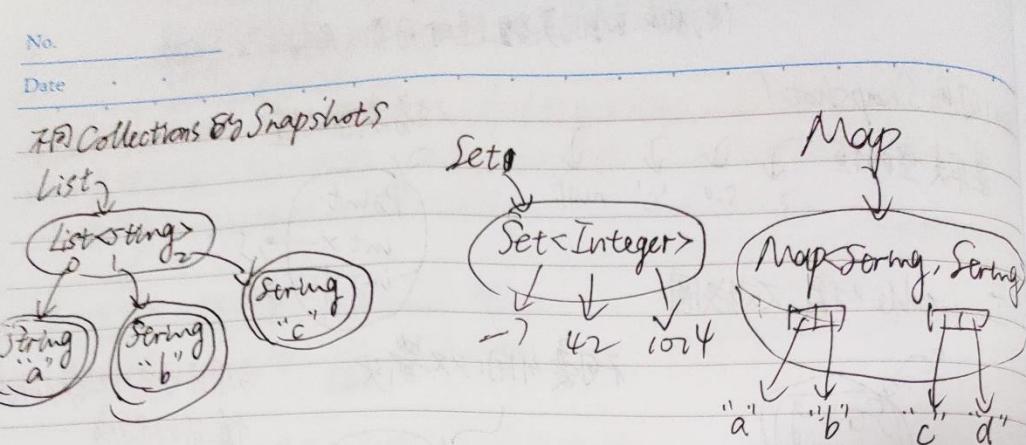
s2

String  
"abcd"

s1

String  
"abc"

\*



## 3.2. Designing Specification. 设计规格

方法的 spec 一脉离散化，明确责任，解耦，不需要知道内部实现。

{ 输入 / 输出 数据类型。

{ 功能和正确性

{ 性能

@param

只讲“能做什么”，不讲“如何实现”

[ pre-condition ] 对客户端的约束，使用时必须满足的条件

[ post-condition ] 对程序员的约束，方法结束时必须满足的条件

↳ @return @throws

行为等价性 — 在客户端的视角看，是否可以相互替换，满足同一个 spec

正确性要求：

→ 前置条件满足时，后置条件必须满足。

但前置条件不满足，方法可以做任何事 → 但健壯性要求应对所有可能的输入情况做出合理的反应

除非在后置条件声明过，否则方法内部不应该改变输入参数。

判断 spec 的好坏：spec 的确定性 / 陈述性 / 强度 重要

→ 强烈的 spec = 更松的前置条件 + 更严格的后置条件 → 就可用 S 代换 S'

$S_2 \geq S_1$ ,  $-S_2$  的前置  $\leq S_1$  的前置

① 在满足  $S_1$  前置的条件下，在  $S_2$  的后置  $\rightarrow S_1$  的后置

No. \_\_\_\_\_  
 Date . . .

That  
 is on

(3) find,  $\text{int}[\text{a}, \text{int val}]$ .  
 requires: val occurs exactly once in a  
 effects: returns index i such that  $a[i] = \text{val}$ .

find<sub>2</sub>,  $\text{int}[\text{a}, \text{int val}]$   
 requires: nothing  
 effects: returns index i such that  $a[i] = \text{val}$ , or -1 if no such i.

在  $S_1$  的前置满足条件下,  $S_2$  的后置弱于  $S_1$  弱. 则  $S_2 < S_1$ ,  
 也在不可比较的情况下

越强的 spec 意味着开发者的责任越重, 客户端的责任越轻.  
 确定的确定性 - 给定一个唯一的 pre 的输入, 输出是唯一的、明确的  
 不确定的 spec - 同一个输入有多件输出 (under-deterministic).  
 非确定的 spec - 同一个输入多次执行得到的输出不同 (Non-deterministic)

Spec 的图

Spec  
 pre  
 post

一个 Spec 固定了一个范围, 在范围内的实现是一个高  
 强 spec. 意味着更小的区域

↳ 更强的 post 意味着实现的自由度更低了 → 在图中面积变小  
 ↳ 更弱的 pre 意味着实现时要处理更多的输入, 自由度低了

设计好的 spec

内聚的一致性的功能应单一、简单、易理解  
 信息丰富的 使用抽象类型/接口, 为后代的实现体提供更大的自由  
 一强般合适的

正确性 健康性

不写 pre-cond, 就要在内部 check 参数合法性 (写了还要 check)  
 对外的 public 方法, 因此写明 pre-cond

No. \_\_\_\_\_ Date \_\_\_\_\_

表示 - private fields

3.3. ADT 抽象数据类型

ADT 特性: 表示 抽象类, 抽象函数 AF, 表示不变量 RI  
 representation exposure: Abstraction functions.

数据抽象由一组操作所刻画的数据类型  
 → 内部的具体实现无关.

ADT 可变的 mutable: 提供了可改变其内部数据值的操作.  
 immutable: 不提供, 或者操作不改变内部值, 直接与外部对象交互.

Creator 创造器	Producer 生产器	Observer 观察器	Mutator 变换器
构造方法/静态方法 (工厂等...)	泛型方法 记住这四个, 并能区分		通常返回 void

设计 ADT:

- ① 设计简洁, 一致的操作.
- ② 支持 Client 对数据所做的操作编写.
- ③ 针对抽象设计, 要针对具体应用设计, 不要混杂.

表示独立性. Representation Independence (不是 RI!!).

→ Client 使用 ADT 时不需要考虑其实现如何实现的, ADT 内部的变化不应该影响外部的 spec 和客户端.

调用 ADT:

- 测试 creators, producers, mutators. → 调用 observers.
- 测试 observes. → 调用 creators, producers, mutators. 方法实现  
 改变观察, 观察其结果是否正确.

ADT

保持不变量, 在任何时候必须满足, 由 ADT 内部保持.  
 一旦表示泄露, 就影响了表示独立性, 客户端可以修改 ADT 内部数据.  
 使用 immutable 类型, 从源头上避免表示泄露.

defensive: 防御式编程

AF: 抽象函数 - R 和 A 之间映射关系的函数，即如何解释 R 中的每一个值为 A 中的  
 每个值

RI: 表示不变量。  
 新版本的表示“是否合法”  
 所有表示值的一个子集包含了所有合法的表示值。

满射 不单  
 R 中的非法值在 A 中  
 无象

- S. Tha  
 ns."  
 agram  
 eos or

```

  例: public class CharSet {
    private String s;
    // RI:
    // S contains no repeated characters.
    // AF:
    // AF(s) = { s[i] | 0 <= i < s.length() }
  }
  
```

选择某种特定的表示方式 R, 进而指定某个子集是“合法”的(RI), 并为该子集中的  
 每个值做出“解释”(AF) - 即如何映射到抽象空间的值

在所有可能改变 Rep 的方法内, 都要 (checkRep), 就连 Observer 里也要  
 (checkRep) 因为 null 检查是必要的 (比如 Java - 唯一 null)

Safety from rep exposure.

```

  例: public class RatNum {
    private final int number;
    private final int denom;
    // Rep Invariant:
    // denom > 0; number/denom is in reduced form, gcd(number, denom) == 1
    // Abstraction Function
    // preserves the rational number/denom
    // Safety from rep exposure
    // All fields are private, and all types in the rep are immutable.
  }
  
```

AF 是给出 client 看到的值是什么  
 不用对所有的 fields 做 AF!  
 (e.g. Tweet author, text, time)  
 $AF(\text{author}, \text{text}, \text{time}) = \text{a tweet posted by author with content text, at time timestamp.}$

No. 完成ADT的  
Date 技术 没有对操作次能突了.

### 3.4 OOP 面向对象编程

1. Criteria of OO 面向对象标准

泛型、继承、多态、动态方法/封装

2. Basic 基本概念：对象、类、方法

3. Interface & Enumerations.

接口之间可以继承与扩展，一个类可以实现多个接口（从而具备多个接口的方法）。

一个接口可有多种实现

Set<String> set = new HashSet<String>();

倾向于用接口类型变量，而不是一个具体的ADT类

如果让 Client 不知道具体实现的子类名，就用工厂方法

对客户信息隐藏 | ① 使用接口类型声明变量

| ② 客户端仅使用接口中定义的方法

| ③ 客户端代码无法直接访问属性

异常不能抛出，只能抛出或不抛

重写 Override → 参数列表、返回值相同，子类型中方法的要求和父类有语义差异不能比父类差。

至少包含一个抽象方法，可以有 fields

Interface → Abstract Class → Concrete class

如果某些操作是所有子类型共有，但彼此有差别，可在父类中设计抽象方法在子类中重写；有些有些没有的操作，不重写父类中之义，实现，在特子类中实

多态 | 特殊多态 → Overload 一规则：① 参数列表必须不同

参数化多态 → 泛型

子类型多态，包含多态

② 返回类型不同

③ 可 public / private / protected

④ 可抛出更多异常 / 更少异常

⑤ 可在同一类中重载，也可在子类中

调用时 Overload 由类型匹配

Override 由运行类型

static check.

run-time

public interface Set<E> { ... }  
public class CharSet implements Set<Char> { ... }

泛型及参数化多态 通配符 "?", 只在使用泛型时出现，不能在定义中出现。  
List<?> list = new ArrayList<String>();  
List<? extends Animal>  
List<? super Animal>

没有泛型参数  
操作检查不通过

类型多态 - LSP 5 → Reusability

Java Object 中的重要方法 `toString()`, `hashCode()`, `equals()`.

仅需同时 Override.

3.5. Equality in ADT & OOP. ADT 和 OOP 中的等价性

等价关系：自反、对称、传递  $s=t, t=r \Rightarrow s=r$

$t=t$        $S=t$

$t=S$

三种方式判定等价  $\left\{ \begin{array}{l} \text{① AF} \\ \text{② 等价关系} \\ \text{③ 通过外部观察者角度} \end{array} \right.$

(immutable)      两个相等的对象  
 $\text{hashCode() 相同}$

"==" 判断引用等价性 / 或者判定基本数据类型相等

"equals()" 判断对象等价性 (父类 Overrule, 子类重写 Overrule hashCode.)

注意参数是 Object 类型！

public boolean equals (Object obj) { ... }

重写的时候参数类型必须也是 Object 类型。

默写实现 `a.equals(null)` returns `false`

判定 mutable 等价 { 行为等价性 → 想要达成但是容易破坏 API.

行为等价性 → 调用一切方法返回相同的结果。

对 mutable, 实现行为等价性即可, 直接继承 `equals()`, `hashCode()`, 提向原有空间

相对方相等。如果一定要判断, 则最好定义一个新方法 `isSimilar()` ...

No.

Date

AutoBoxing

Integer x = new Integer(3)

Integer y = new Integer(3)

x.equals(y) → true. → 等价性  
x == y → false. - 引用等价性

(int)x == (int)y → true.

### 方法的重写要求

- (1) 重写方法名参数列表相同
- (2) 该访问修饰符不能比父类窄
- (3) 所加注释不能比父类更宽泛
- (4) 返回类型要比父类的小或相同

## Chapter 5: Reusability-Oriented SC Approaches.

### 5.1 Metrics, Morphology and External Observations of Reusability

可复用性的度量、形态与外部表现

面向复用编程：开发出可复用的软件 (for reuse)

基于复用编程：利用已有的可复用软件搭建应用系统 (with reuse)

为什么 reuse 一降低成本和开发时间 / 经过充分测试可靠、稳定 / 标准化

如何衡量可复用性？— 复用的机会有多少次？复用的场合有多少？

复用的代价有多大？— 搜索/获取/匹配/扩展/实例化 / 与软件其他部分互连的紧密程度

特点（复用性高）

简单 / 与标准兼容 / 灵活可度 / 可扩展 / 泛型 / 参数化 / 模块化 / 增加局部性 / 平衡 / 对称 /

什么都可能被复用 — Code / 需求 / Spec / 数据 / Test / Doc

降低耦合：methods, statements / class and interface / API / framework.

白盒复用 — 源代码可见，可修改和扩展。

黑盒复用 — 源代码不可见，不能修改。

### 5.2 Construction for Reuse 面向复用的SC技术

子类型多态：客户端可用统一的方式处理不同类型的数据。

Liskov 替换原则 (LSP)、行为子类型化 behavioral subtyping.

Java 编译器强制 static check.

子类型可以增加方法，但不能删除原有方法。

子类型必须实现抽象类型中所有未实现方法。

子类型中重写的方法必须有相同或子类型的返回值。

类型安全

子类型中重写的方法必须使用同样类型的参数或者符合 covariance 的参数。

子类型中重写的方法不能抛出意外的异常。

No.

Date

- 对Method:
- 更强的不变量(或相等)
  - 更弱的 precond ↴
  - 更强的 postcond ↴

LSP - 更强(strong) 行为子类型化

- 前置条件不能强化 - 后置条件不能弱化 - 不变量要保持.

- 子类型方法参数: 逆变 ~~反向变~~ 子类型方法返回值 + 扩展.

- 异常类型: 扩展.

扩度: 父类型 → 子类型: 越来越具体的 spec

异常/返回值类型: 不要或变得更具体.

抽象

从观上看, 异常/返回值

扩度就是 pre-cond 强化

反扩度/逆变: 父类型 → 子类型: 越来越具体的 spec

参数类型: 相反地变化, 不要或越来越抽象.

参数类型的反扩度就叫 post-cond "要弱"

开始且扩度的父类是 T 的子类 ⇒ 子类是 T 的子类.

泛型不是扩度的 ⇒ ArrayList<String> 是 List<String> 的子类

但 List<String> 不是 List<Object> 的子类.

编译器会进行类型擦除

普通配对? 的泛型 ① List<Number> 是 List<?> 的子类

② List<Number> 是 List<? extends Object> 的子类

③ List<Object> 是 List<? super String> 的子类.

④ List<Integer> 是 List<? super Integer> 的子类

第 4 可能.

Comparator<T> 提供 int compare(T t<sub>1</sub>, T t<sub>2</sub>). → Delegation 到其族  
Comparable<T> 提供 int compareTo(T t).

No.

Date

Delegation 委派/委托：一个对象请求另一个对象的功能。  
呈或模式 简之，利用别人的功能达到自己的目的。  
用 Delegation 代替继承

如果子类只需要复用父类一小部分代码的方法，可以不继承，用 Delegation。  
Delegation 在对象层面，“继承”在类层面

Composite Reuse Principle / CRP

Delegation 的类型

{ Use/Dependency  
Association }

使用接口定义系统必须对外展示的不同侧面的  
行为

接口之间通过 extends 实现行为的扩展

类 implements 组合接口，从而避免了复杂的  
耦合

Composition/aggregation 继承关系

Dependency: 临时性的，作为参数或局部变量

Association: 永久性的一有这个变量，但不是“我”创建的

Composition: 有这个变量，由“我”创建，“我”消失时消失

Aggregation: 有这个变量，由“外部”创建，“我”消失时不消失

系统的级别复用。API 设计 - 易用易理解，可增不可删 fail fast

Framework 设计

常用设计模式

白盒 - 通过继承和 Override 实现 - Triple Method.

黑盒 - 实现插件接口 - Delegation → Strategy, Observer.

Java Collection Framework.

宝贵

No.

Date

112

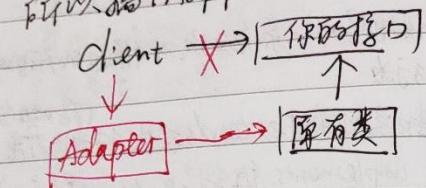
### 5.3 Design Patterns for Reuse.

#### Structural patterns 结构型模式

##### (1) Adapter 适配器模式

Client 希望调用一个他的接口编程，而不是你事先设计好的那个。你又不能杀他

所以搞个 Adapter 给他吃



例如：你写了一个矩形，用它的中心点表示高  
和长宽表示。

但是你的用户希望用左上角点坐标和长宽表示，你必须  
重新搞个接口。

##### (2) Decorator 装饰器模式

Client 希望你写好的类再多个另一个写好的类的特性。但是 Java 又没有多重继承，所以你现在不知道该杀谁的 (client/Java) 的鸟。

让 Decorator 来拯救你

为对象增加不同侧面的特性，对每一个特性都写好子类，增加特性时通过 delegation 实现。

Stack = new SecureStack()

new SynchronizedStack() 希望你的安全性高。

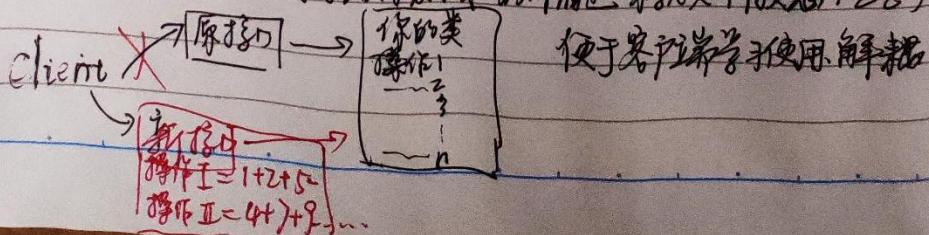
new ArrayStack(S)) 你只好把普通Stack，线程安全Stack。

你把 Stack 都写好，然后直接 delegate

之后也会接触到了工厂方法实现，比如 Collections.synchronizedList(new ArrayList())

##### (3) Facade 外观模式

Client 觉得你的接口太零碎，表示自己只想调一个方法就够了（高是一个复杂操作）  
(所以你现在觉得这个人除了给你发钱，否则看起采就是个懒癌患者)

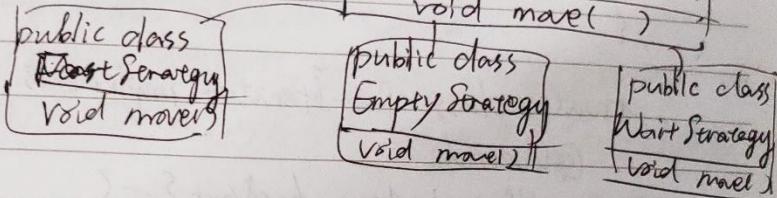


## Behavioral patterns 行为型模式

### ① Strategy 策略模式

Client 希望你的算法有多种，因为他一开始还不知道场景里用什么合适  
(所以你觉得这人真笨多还喜欢挑剔除了反复横跳他学不会还会什么)  
有多种不同的算法来实现任务，但客户 Client 根据可以动态切换。

例：猴子过河三种方法，先弄个接口给他  
再实现3个类。

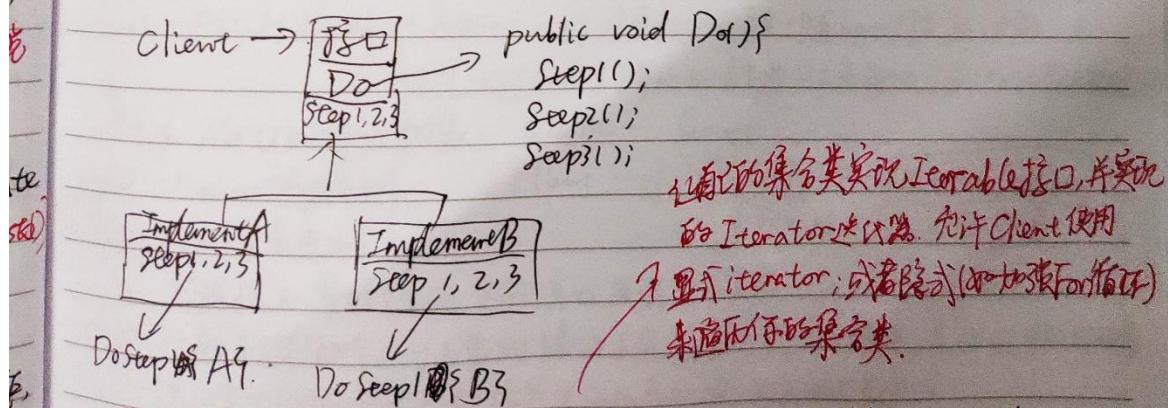


### ② Template Method 模板模式

Client 说我也做件事已经有了确定顺序，但是每一步中他想搞点不同的  
(所以你觉得这个人就是吧?)。

做事情的步骤一样，但具体方法不同。

共性的步骤在抽象类内实现，差异化的步骤在子类中实现



### ③ Iterator Pattern 迭代器模式

Iterable

Iterable<T>

Iterator<T>

Iterator<T>

要求实现的类能够返回一个  
迭代器 iterator;

Iterable<T> iterator();

要求实现的类能够返回一个  
迭代器 iterator;

boolean hasNext();

要求实现的类能够返回一个  
迭代器 iterator;

E next();

要求实现的类能够返回一个  
迭代器 iterator;

void remove();

No.

Date

常见设计模式方法

public class Pair<E> implements Iterable<E>

@Override

public Iterator<E> iterator() {

return new PairIterator();

}

private class PairIterator implements Iterator<E>

@Override

public boolean hasNext() { ... }

@Override

public E next() { ... }

};

## Chapter 6: Maintainability-Oriented SC Approaches

### 6.1 Metrics and Construction Principles for Maintainability

可维护性的度量与构造原则

可维护性的度量指标

圈复杂度、代码行数、可维护性指数、继承的层次数、类之间的耦合度、单元测试的覆盖度

模块化编程：高内聚、低耦合、分离关注点、信息隐藏

大模块-可分解性 / 可组织性 / 可理解性 / 可持续性 / 出现异常之后的保护

小规则 - 直接映射 / 尽可能少的接口 / 尽可能小的接口 / 显式接口 / 信息隐藏

耦合度和聚合度 → 模块内部函数的相关程度

高内聚  
低耦合

衡量模块间的依赖性：模块间的接口数量  
    |  
    | 算法接口的复杂性

### OO 设计原则: SOLID

· SRP 单一责任原则 - 不应该有多个原因让你的 ADT 发生变化，否则就该拆开

· OCP 开放/封闭原则 - OPEN - 对扩展性的开放 - 随时可扩展新功能

CLOSED - 对修改的封闭 - 不允许改已有的代码 (死锁)

方法 Abstraction 抽象 Client - Server

· LSP. Liskov 替换原则 Client - | AbServer | ← ConcreteServer  
    | 不该修改      | 扩展代码

· ISP 接口隔离原则

只提供必要的接口，不同接口提供不同服务

· DIP 依赖反转原则 - 抽象的模块不应依赖于具体的模块，具体依赖于抽象

说人话 - Delegation 的时候通过 interface 建立联系，而非操作类

No.

Date

## 6.2 Design Patterns for Maintainability 面向可维护性的设计模式 (3)

### 1. Creational patterns 创建类模式

#### (1) Factory Method pattern 工厂方法模式

当 Client 不知道要创建哪个类的实例，或不想让 Client 知道子类名，用工厂方法。

定义一个用于创建对象的接口，让子类来决定实例化哪个类，从而做一个类的实例化延迟到其子类

#### (2) Abstract Factory Pattern 抽象工厂模式

提供接口以创建一组相关/相互依赖的对象，但不需对指明具体类。

创建的不是一个完整“产品”，是一组“产品”，重点在“搭配”

#### (3) Builder Pattern 构造器模式 一创建复杂对象，包含多个组成部分 创建的是一个完整的产品

### 2. Structural patterns 结构型模式

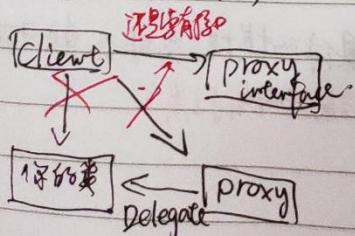
#### (1) Bridge 桥接模式

其实意思让你 Delegate 的时候 Delegate 到接口 代表的时候直接接口连接。

又有 Delete 又有 Extend，显得高级。

#### (2) Proxy 代理模式

(你发现你的 Client 太个凶，为了避免他 删库跑路，给他弄个防火墙 让他在里面玩干嘛)



与 Adapter 的不同之处：

Adapter 是用户是凶，想换一种方式访问接口

Proxy 是用户是凶，为了保护你的复杂对象，搞个防火墙

No.

Date

### (3) Composite 组合模式

加到属性时形成对形结构来加属性

和 Decorator 的不同：Decorator 是一次加一个特性  
Composite = 可加多个特性

### 3. Behavioral patterns 行为型模式

#### ① Observer

· “粉丝”对“偶像CXK”感兴趣，希望知道CXK整天在干嘛

· 粉丝到CXK的Weibo下关注，如果CXK打篮球了，就会发律师函到每个粉丝的私信里，然后鸡汤你两句...

Observable 接口 —— 派生子类，知道CXK

Observer 接口 —— 实现该接口，知道粉丝

#### ② Visitor

对特定类型的object执行操作(Visit)，在运行时将二者分离  
该操作可以灵活更改，不需要改被visit的类。

— 为ADT预留一个扩展“切入点”

#### ③ Mediator / Command / Chain of responsibility 参试大图中元

No.

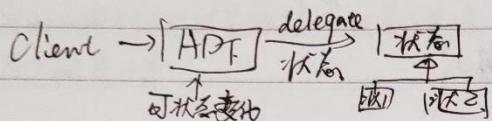
Date

### 6.3 Maintainability-Oriented Construction Techniques 面向可维护性的构造

#### 1. Behavioral pattern 状态/行为型模式

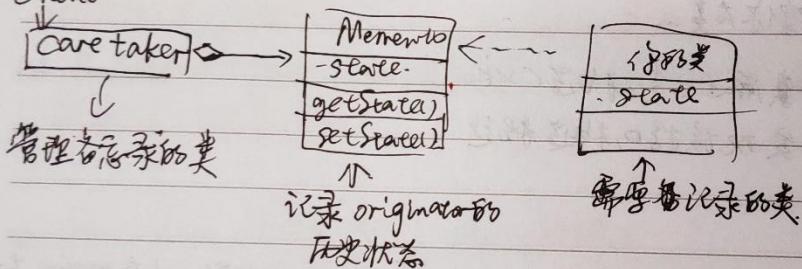
##### (1) State Pattern

不用 if/else 在 APT 中实现状态转换。



##### (2) Memento Pattern 备忘录模式 = 记住对象的内部状态，以便“回溯”

Client



#### 2. Regex 正则表达式

## Chapter 7: SC for Robustness

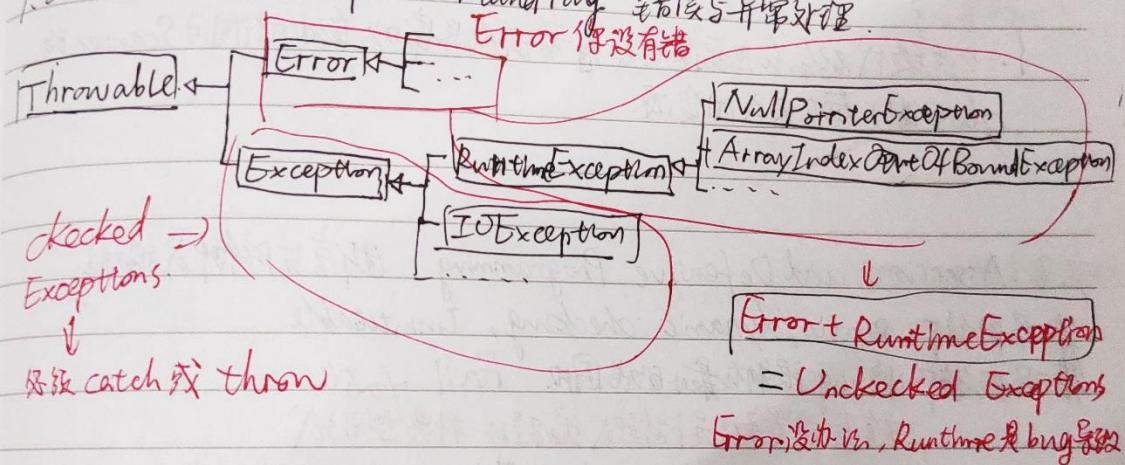
No.

Date

### 7.1 Robustness and Correctness

健壮性：系统在不正常输入或不正常外部环境下仍能够表现正确  
处理未期望的行为和错误终止；即使终止执行，也要准确无歧义的向用户展示错误  
错误信息有助于 debug

### 7.2. Error and Exception Handling 错误与异常处理



try - catch - finally finally 中做一些释放资源的事, like scanner.close()

除非在 catch 中 system.exit(0) 一系统调用

否则 finally 一定会被执行。→ 就算 return 了，也会执行

try/catch

使用 checked 异常来确认可能出现异常情况，之所以是 checked 是因为你知道“可能会发生！”

异常是包括在 Post-cond 中的。要遵守 OOP 原则 - LSP 等等...

final 在 spec 中明确规定所抛出的全部 checked exception.

摘要 → 参见前面的笔记

创建自己的异常类，extends Exception

rethrow 可以转换 Exception 的形式，以便上层更好地处理

No.

Date

也有新关闭资源模式

6600

```
try (Scanner sc = new Scanner(File file)) {
```

```
    while (sc.hasNextLine()) {
```

```
        System.out.println(sc.nextLine());
```

```
}
```

1. 这段代码会在正常退出或者出现异常时，自动地调用 Scanner 的  
.close() 方法来关闭资源

### 7.3 Assertions and Defensive Programming 断言与防御式编程

第一层防御：Static/dynamic checking, Immutable

第二层防御：将 bug 限制在最小的范围里 Fail fast

检查前置条件是防御式编程的一种典型形式

断言：在开发阶段的代码中嵌入，检验某些“假设”是否成立，若成立，表明程序运行正常，否则表明存在错误。实际使用时，断言会被关闭  
出现 AssertionError，违反断言，则证明运行时有 bug

```
assert false : message.
```

使用断言的位置：

判断内部不变量 / 断定表示不变量 / 方法的 pre-cond / 方法的 post-cond

是为了在开发阶段尽快避免错误，调试程序。

只检查是否内部出问题，外部的问题交给 Exception

-ea 开启断言（默认为关闭的 -da）