Apply Orthogonal Design in Java

刘 光聪

2016.10

内容

软件设计

为什么要做软件设计?

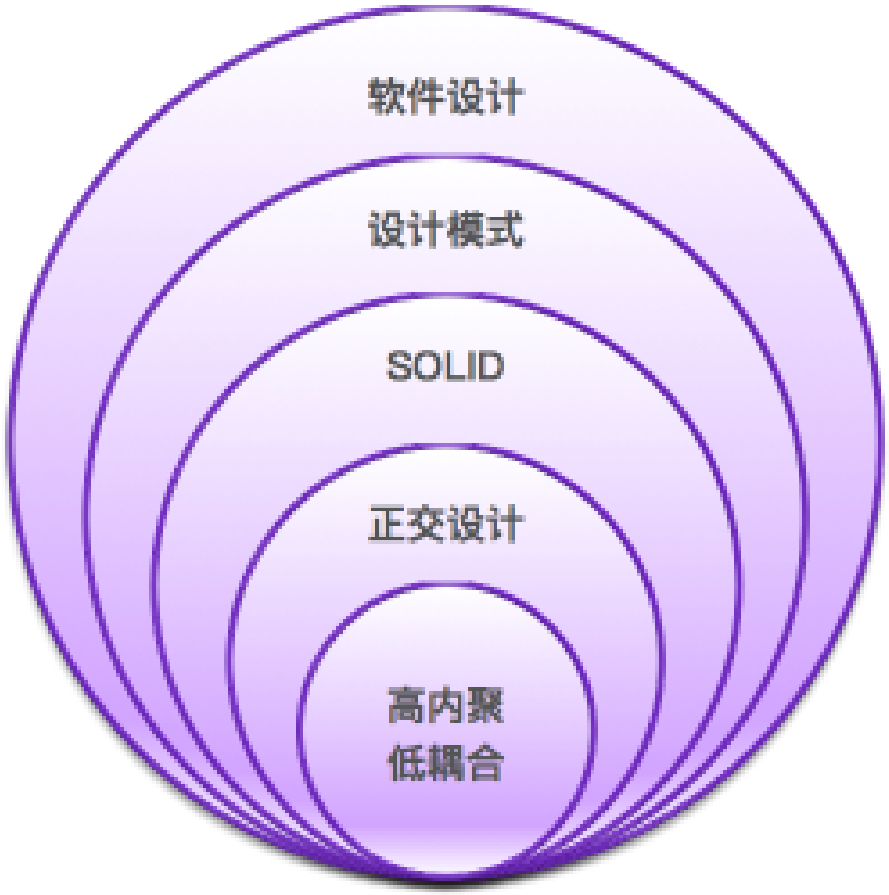
拥抱变化

> Design is there to enable you to keep **changing** the software **easily** in the **long** term.
>
> Kent Beck

什么是好的软件设计?

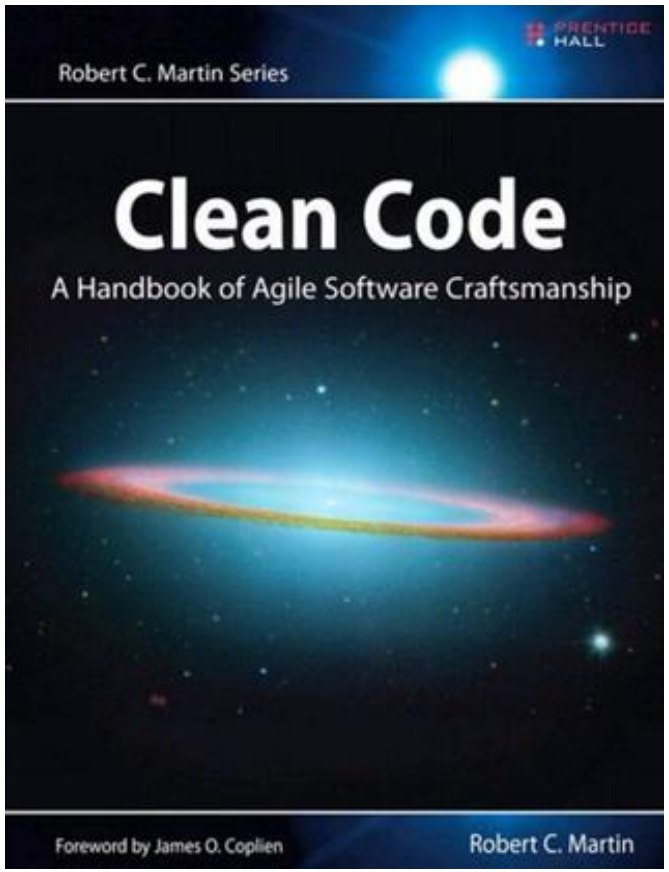
设计初衷

易于重用



易于理解

1. Clean Code
2. Idioms
3. Patterns



没有冗余

1. YAGNI: You Ain't Gonna Need It
2. KISS: Keep it Simple, Stupid

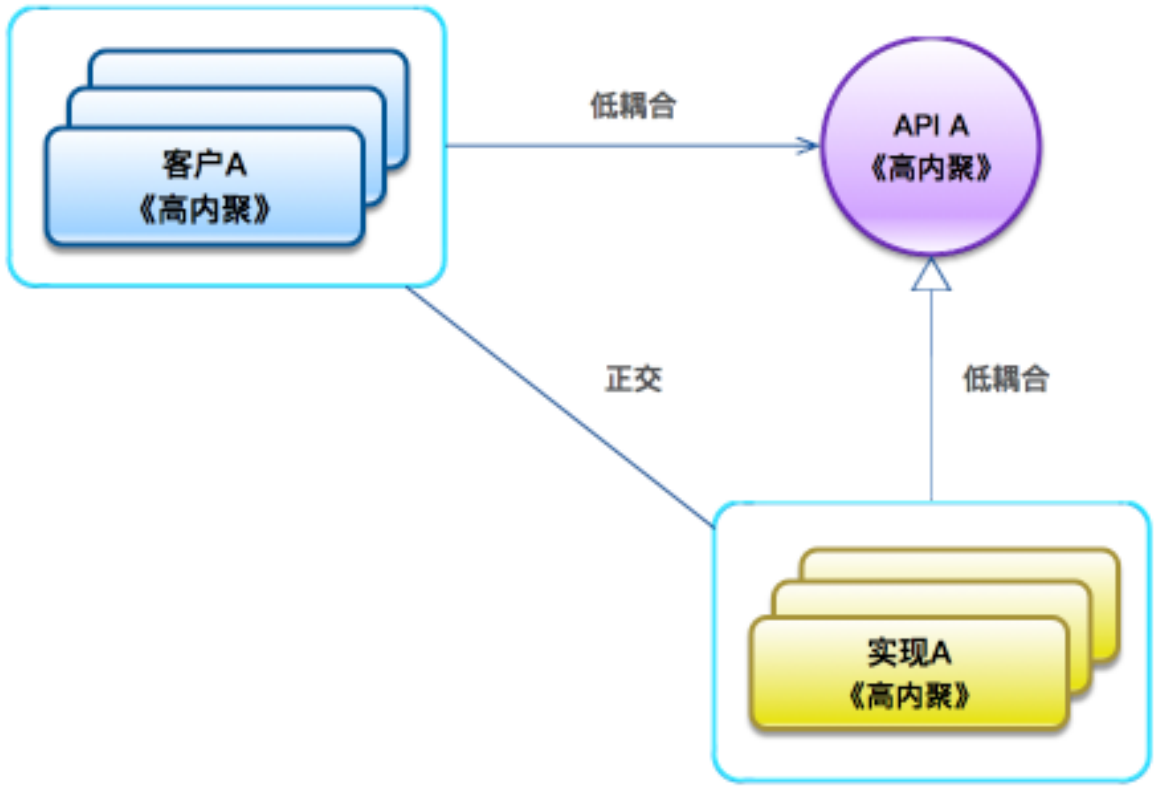
简单设计

以下 4 个原则的重要程度依次降低

1. 通过测试：完成功能
2. 没有重复：易于重用
3. 意图明确：易于理解
4. 没有冗余：恰如其分

如何重用软件?

一个出发点：模块化设计



两个基本问题：分解与组合

1. 分解：当我们划分模块时，要让每个模块都尽可能高内聚
2. 组合：当定义模块之间的 API 时，需要让双方尽可能低耦合

人类认知

心智的活动，除了尽力产生各种简单的认识之外，主要表现在如下三个方面：1）将若干简单认识组合为一个复合认识，由此产生出各种复杂的认识。2）将两个认识放在一起对照，不管它们如何简单或者复杂，在这样做时并不将它们合而为一。由此得到有关它们的相互关系的认识。3）将有关认识与那些在实际中和它们同在的所有其他认识隔离开，这就是抽象，所有具有普遍性的认识都是这样得到的。

John Locke, *An Essay Concerning Human Understanding*

（有关人类理解的随笔，1690）

高内聚：Do One Thing, Do It Well

1. 紧密关联的事物应该放在一起。
2. 只有关联紧密的事物才应该被放到一起。

职责的定义：变化的原因

1. SRP： 一个模块有且仅有一个发生变化的原因
2. 变化原因： 一个变化会导致整个模块内包含的各个元素都要发生变化，那么就不能分离它们；否则，将引入不必要的复杂度。
3. 变化驱动： 当且仅当变化发生时，分离变化的轴线才有意义。

习惯思维：以概念的方式识别职责

```
public interface Modem {
  void dial(String pno);
  void hangup();
  void send(char c);
  char recv();
}
```

分离职责

```
public interface Connection {
  void dial(String pno);
  void hangup();
}

public interface DataChannel {
  void send(char c);
  char recv();
}
```

面向对象：小类，大对象

1. 类： 模块化设计的手段；
2. 对象： 映射真正的领域模型。

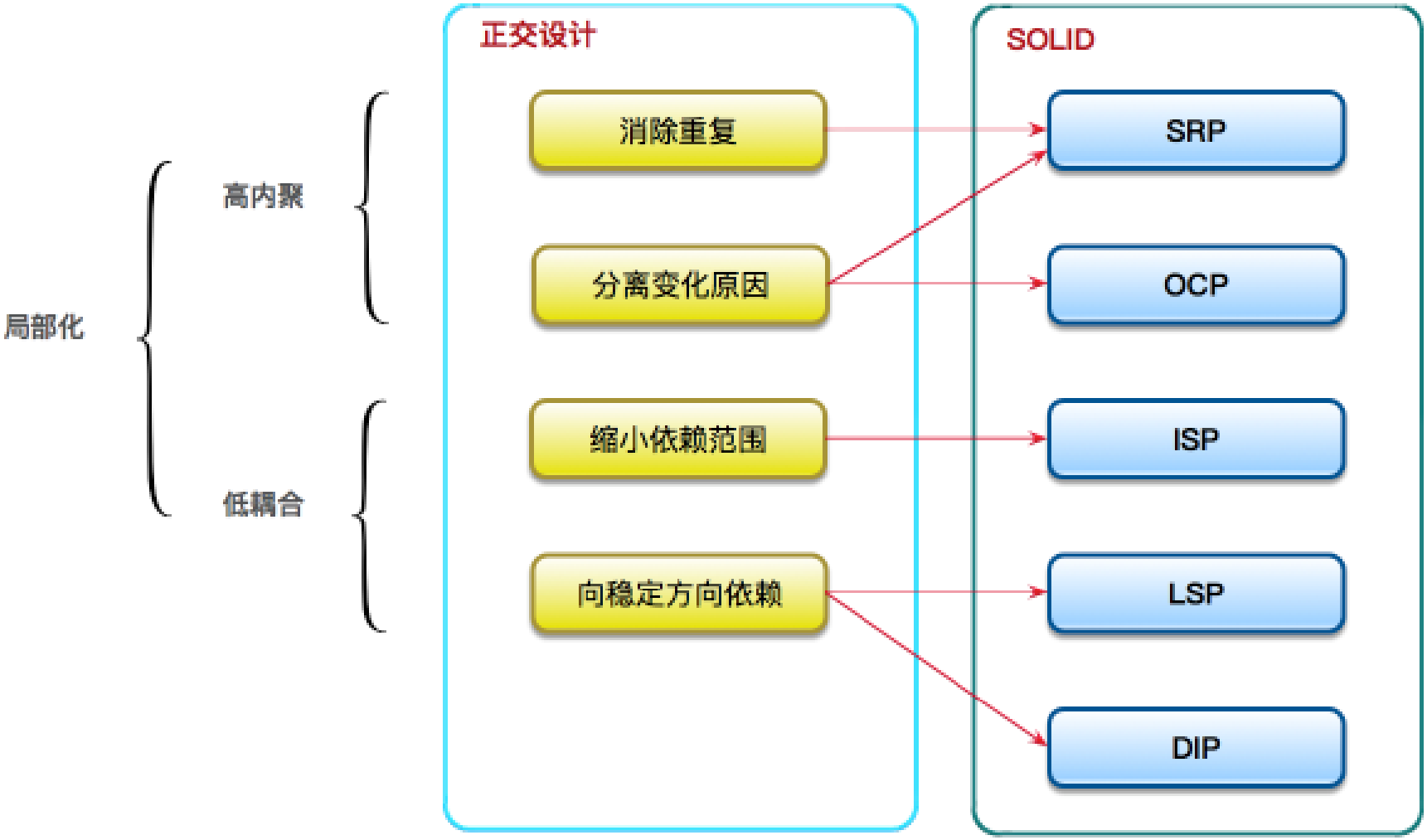
低耦合

1. 耦合性： 强调模块之间的关联紧密程度
2. 低耦合： 模块之间尽可能地互不影响

API 设计

1. 最少知识：应该让客户尽可能地知道最少的知识
2. 最小依赖：不应该强迫客户依赖它不需要的东西
3. 依赖于抽象，而不是实现：站在需求的角度，而不是实现方式的角度定义 API

四个策略：正交设计


局部化
高内聚
低耦合
正交设计
消除重复
分离变化原因
缩小依赖范围
向稳定方向依赖
SOLID
SRP
OCP
ISP
LSP
DIP

拥抱变化

应对变化

1. 一个变化导致多处散弹修改：消除重复
2. 多个变化导致一处频繁修改：分离变化

变化发生时，消除不必要的修改

1. 不依赖不必要的依赖：缩小依赖范围
2. 不依赖不稳定的依赖：向着稳定的方向依赖

重复：万恶之源

1. 完全重复
2. 参数型重复
3. 功能型重复
4. 结构型重复
5. 调用型重复
6. 回调型重复

完全重复

```
////////////////////////////////////////////////////////////////
const unsigned int max_num_of_allowed_connections = 1000;

////////////////////////////////////////////////////////////////
const unsigned int max_num_of_allowed_connections = 1000;
```

参数型重复

```
////////////////////////////////////////////////////////////////
strcpy(buf, packet->src_address);
buf += strlen(packet->src_address) + 1;

////////////////////////////////////////////////////////////////
strcpy(buf, packet->dest_address);
buf += strlen(packet->dest_address) + 1;
```

功能型重复

```
////////////////////////////////////////////////////////////////////////
void say_hello_world()
{
    std::cout << "Hello, World" << std::endl;
}

////////////////////////////////////////////////////////////////////////
void say_hello_world()
{
    for(int i=0; i<::strlen("Hello, World\n"); i++)
    {
        ::putc("Hello, World\n"[i]);
    }
}
```

功能型

结构型重复

```
class Foo
{
public:
    void action1();
    void action2();
    void action3();
private:
    int data1;
    int data2;
};
```

```
class Bar
{
public:
    void action1();
    void action2();
    void action4();
private:
    int data1;
    int data3;
};
```

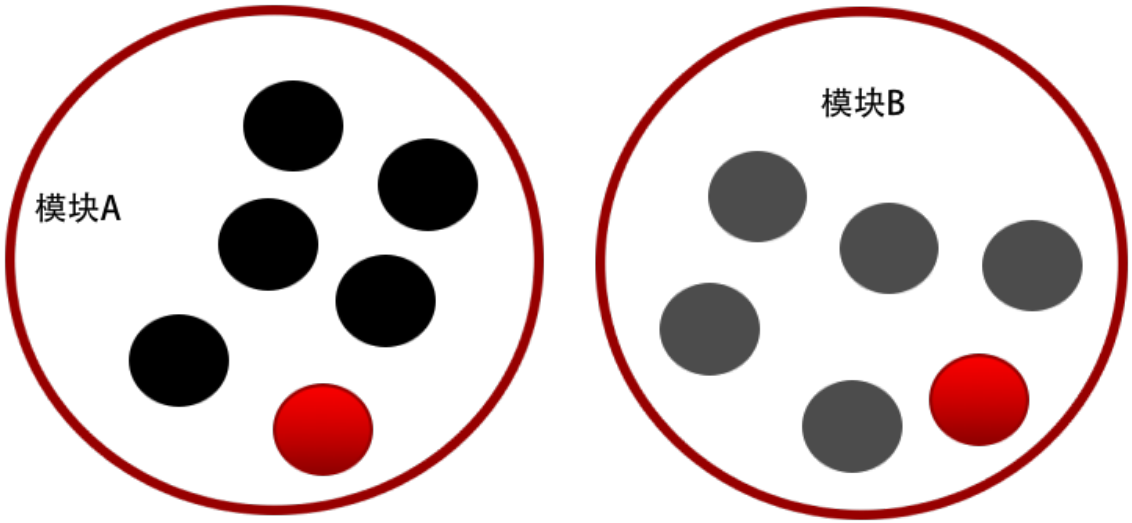
调用型重复

```
////////////////////////////////////////////////////////////////////
void foo()
{
    while(num-- > 0) if(num == packet->pin_num) break;

    strcpy(buf, packet->dest_address);
    buf += strlen(packet->dest_address) + 1;

    if(get_sys_cfg() == SEND) send(buf);
}

////////////////////////////////////////////////////////////////////
void bar()
{
    if(isAllowed()) return;

    strcpy(buf, packet->dest_address);
    buf += strlen(packet->dest_address) + 1;
}
```

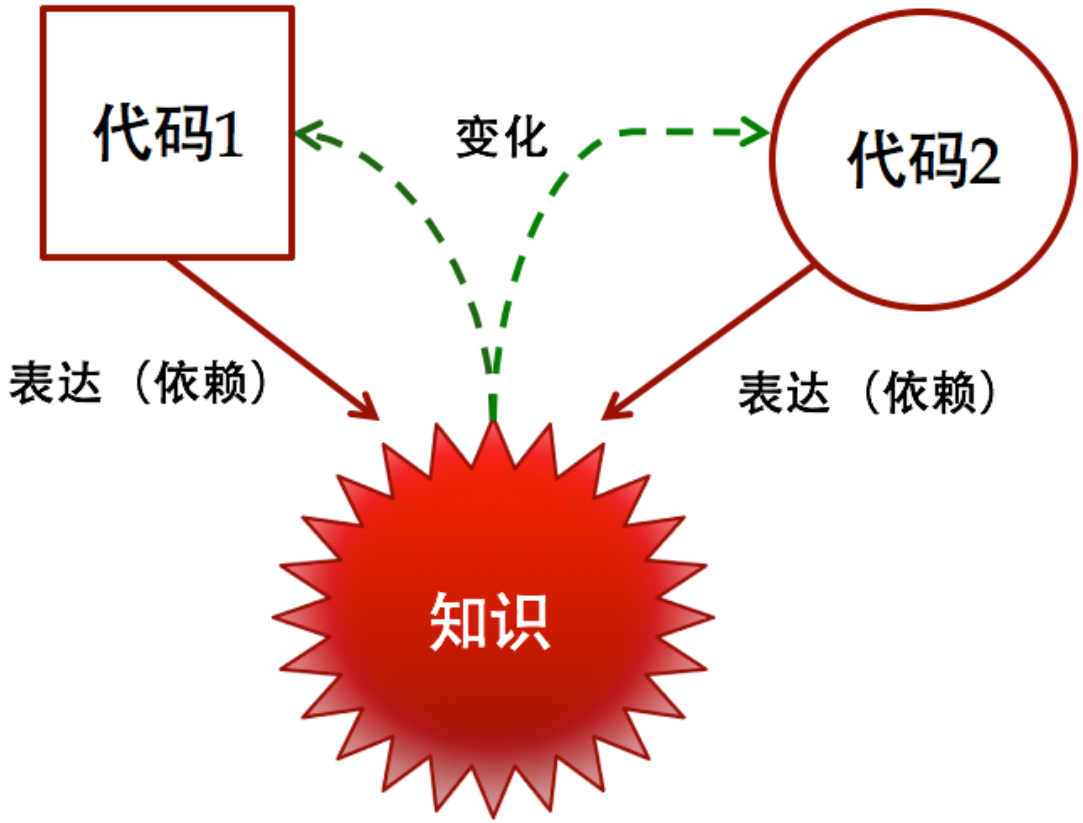
回调型重复

```
////////////////////////////////////////////////////////////
void foo()
{
    while(num-- > 0) if(num == packet->pin_num) break;

    save_to_database();

    if(get_sys_cfg() == SEND) send(buf);
}

////////////////////////////////////////////////////////////
void bar()
{
    while(num-- > 0) if(num == packet->pin_num) break;

    strcpy(buf, packet->dest_address);
    buf += strlen(packet->dest_address) + 1;

    if(get_sys_cfg() == SEND) send(buf);
}
////////////////////////////////////////////////////////////
```

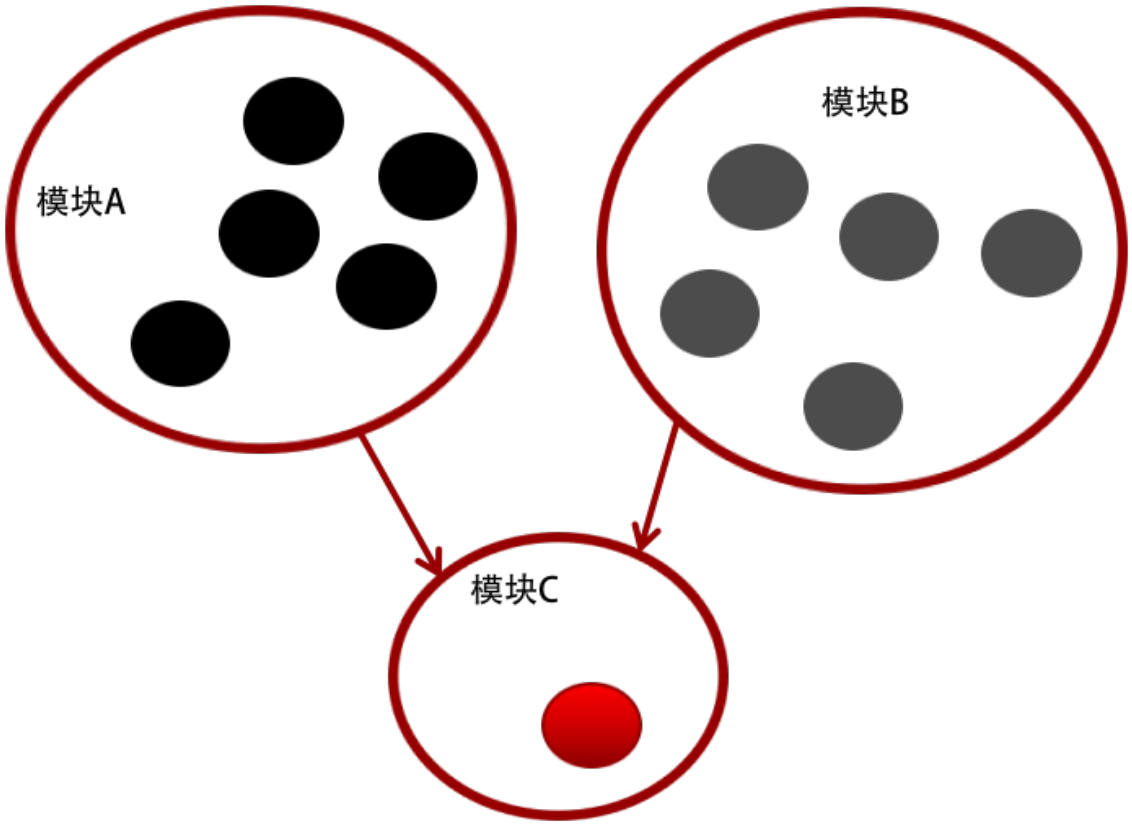
重复: 低内聚



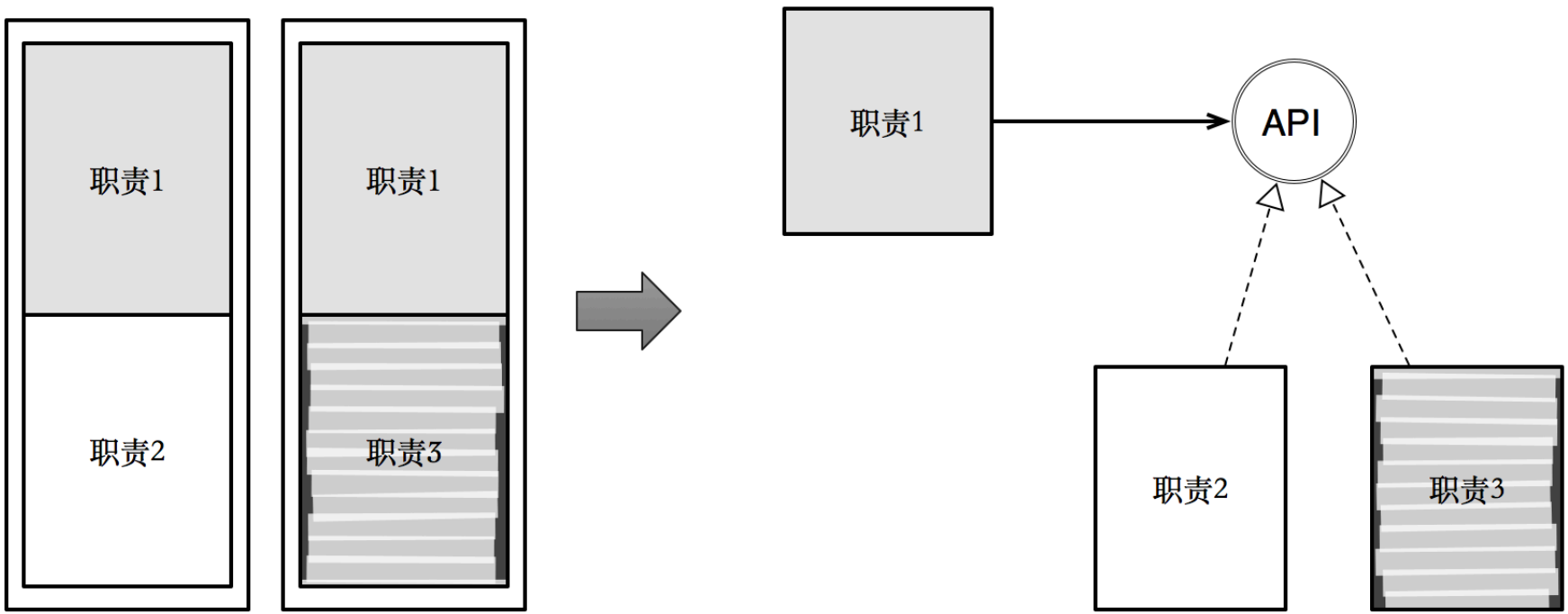
重复: 高耦合



消除重复: 高内聚, 低耦合



消除重复: SRP



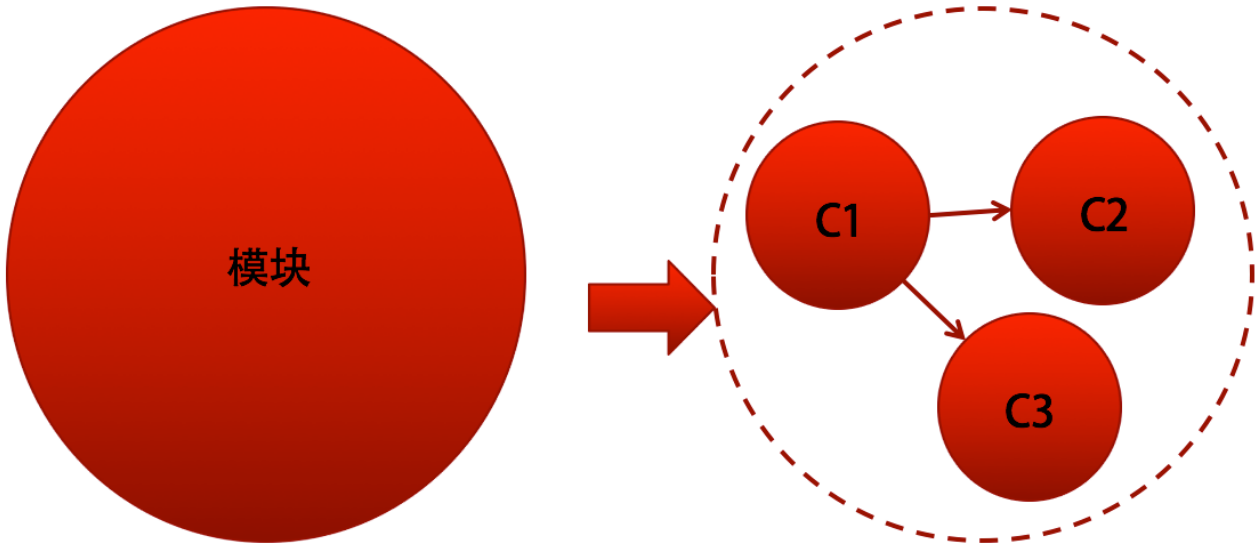
排序：从高到低

```
struct Student
{
    char          name[MAX_NAME_LEN];
    unsigned int height;
};

void sort_students_by_height( Student students[]
                            , size_t  num_of_students)
{
    for(size_t y=0; y < num_of_students-1; y++)
    {
        for(size_t x=1; x < num_of_students - y; x++)
        {
            if(students[x].height > students[x-1].height)
            {
                SWAP(students[x], students[x-1]);
            }
        }
    }
}
```

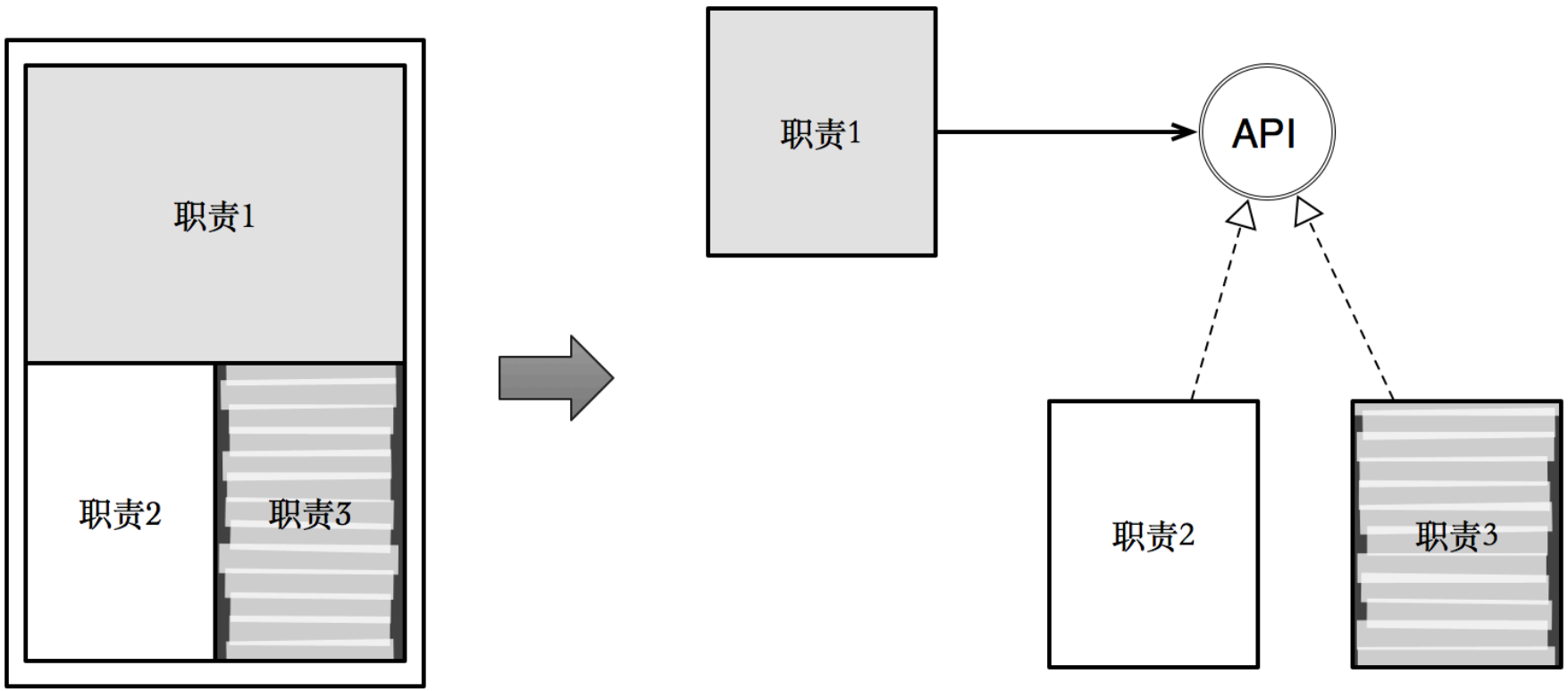
变化原因

1. 排序算法
2. 排序对象
3. 比较准则

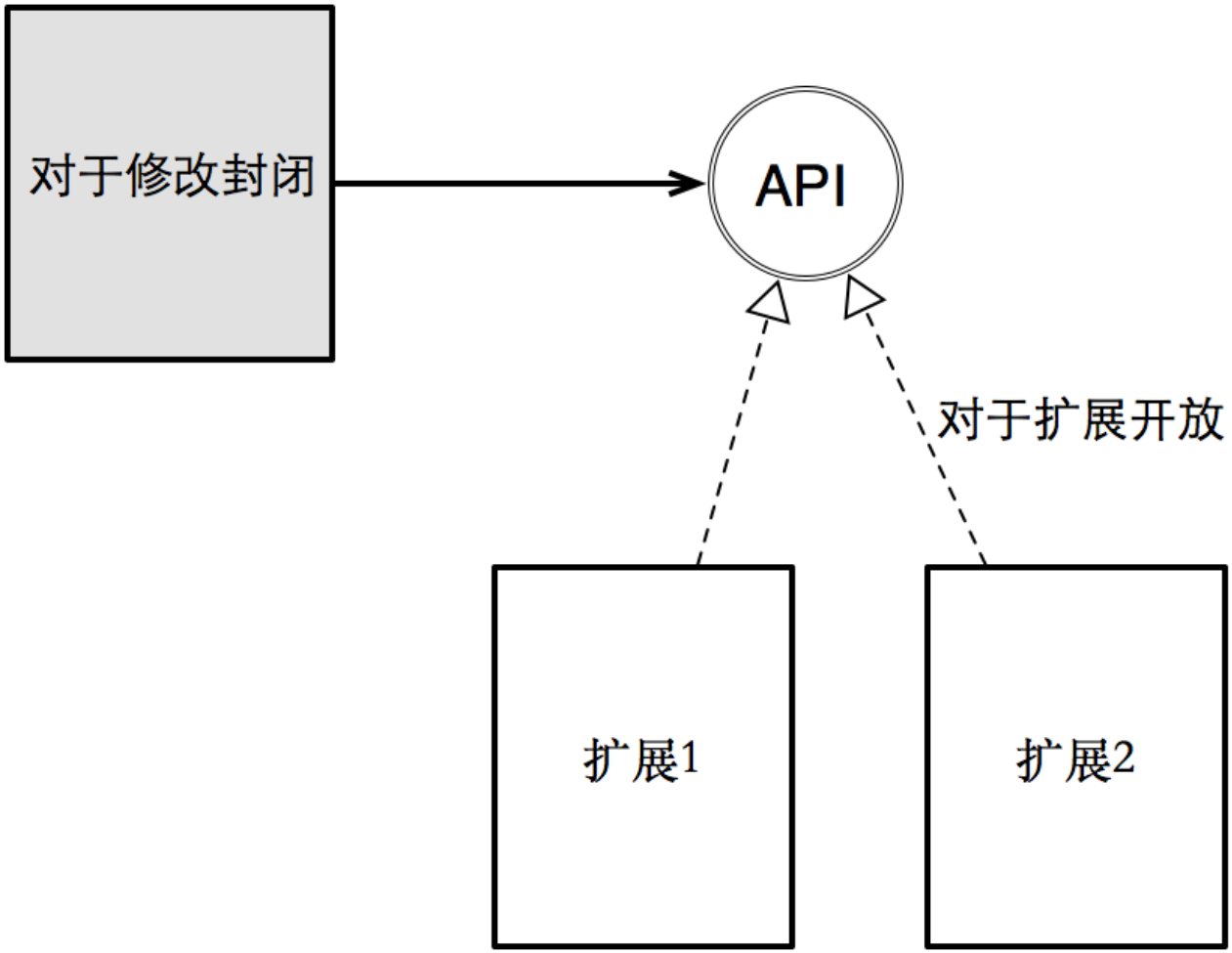
分离关注点



分离关注点: SRP



分离关注点: OCP



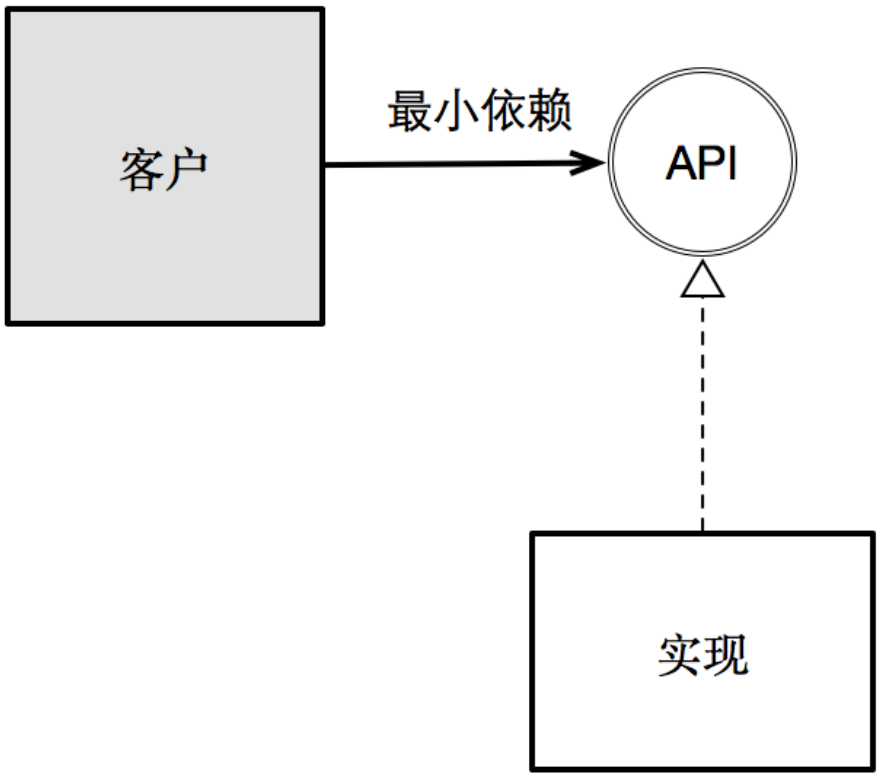
缩小依赖范围

1. 减少对同一关注点的依赖点
2. 减少所依赖关注点的数量

缩小依赖范围

1. 最少知识原则
2. 最小依赖原则

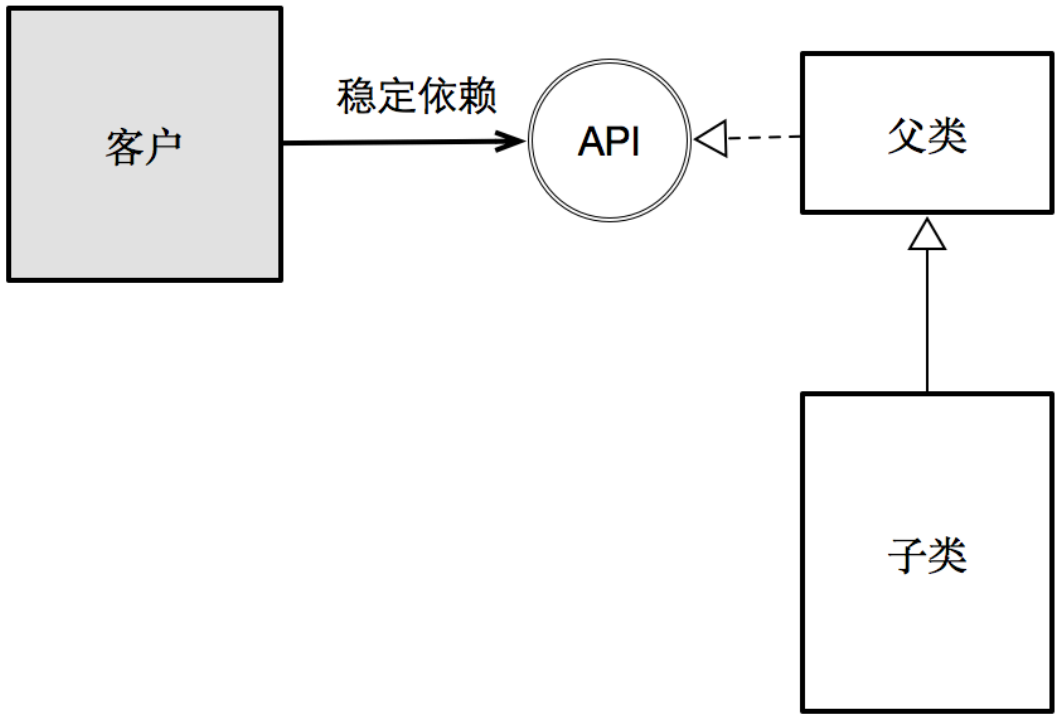
缩小依赖范围: ISP



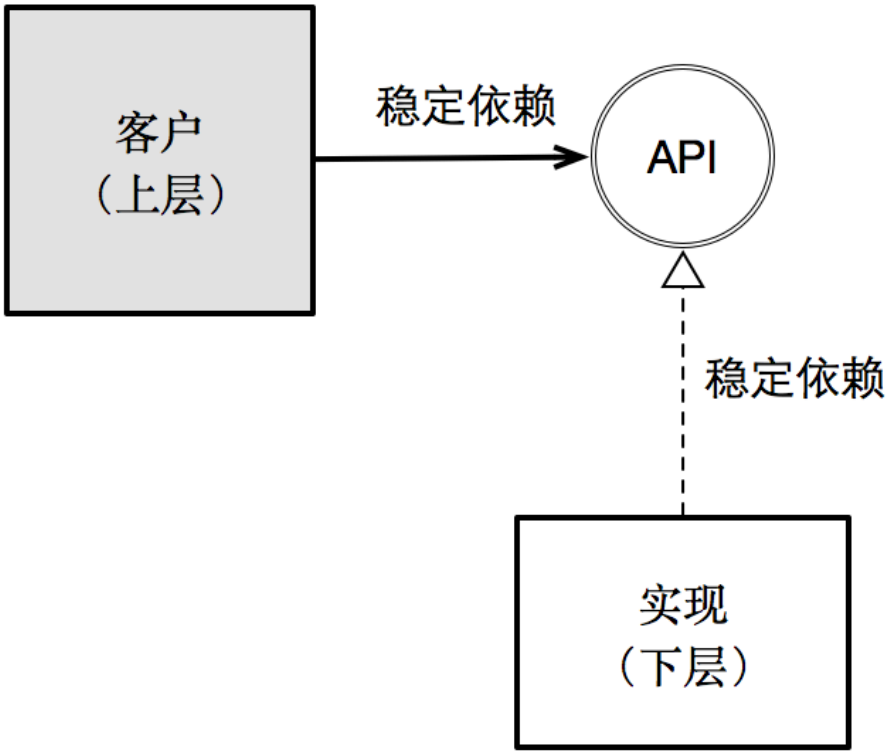
向稳定的方向依赖

1. 依赖于抽象，不要依赖于实现
2. 倒置依赖：高层不依赖与底层，两者都依赖于抽象
3. 按照接口编程

向稳定的方向依赖: LSP



向稳定的方向依赖: DIP



应用正交设计

需求 1：在仓库中查找所有颜色为红色的产品

```
public static ArrayList findAllRedProducts(ArrayList repo) {
  ArrayList result = new ArrayList();
  for (int i=0; i<repo.size(); i++) {
    Product product = (Product)repo[i];
    if (product.getColor() == Color.RED) {
       result.add(product);
    }
  }
  return result;
}
```

识别坏味道

1. 缺乏编译时类型安全性检查
2. 依赖于具体实现类型
3. 硬编码
4. 传递容器

简单重构

```
public static List<Product> findAllRedProducts(
  List<Product> repo) {
  List<Product> result = new ArrayList<>();
  for (Product p : repo) {
    if (p.getColor() == Color.RED) {
      result.add(p);
    }
  }
  return result;
}
```

1. 引入泛型，增强编译时类型安全
2. 依赖于更加抽象的 List，而非具体的 ArrayList
3. 使用 foreach 迭代，避免错误发生的风险

需求 2：在仓库中查找所有颜色为绿色的产品

```
public static List<Product> findAllGreenProducts(
  List<Product> repo) {
  List<Product> result = new ArrayList<>();
  for (Product p : repo) {
    if (p.getColor() == Color.GREEN) {
      result.add(p);
    }
  }
  return result;
}
```

1. 复制/粘贴：导致重复设计
2. 策略：消除重复

消除重复：参数化

```
public static List<Product> findProductsByColor(
  List<Product> repo, Color color) {
  List<Product> result = new ArrayList<>();
  for (Product p : repo) {
    if (p.getColor() == color) {
      result.add(p);
    }
  }
  return result;
}
```

1. 参数化设计，是最常用，最简单的消除重复的手段

需求 3：查找所有重量小于 10 的所有产品

```
public static List<Product> findProductsBelowWeight(
  List<Product> repo, int weight) {
  List<Product> result = new ArrayList<>();
  for (Product p : repo) {
    if (p.getWeight() < weight) {
      result.add(p);
    }
  }
  return result;
}
```

1. 两个参数化实现：重复再现

消除重复：糟糕的设计

```
public List<Product> findProducts(
  List<Product> repo, Color color, int weight, boolean flag) {
  List<Product> result = new ArrayList<>();
  for (Product p : repo) {
    if ((flag && p.getColor() == color) ||
        (!flag && p.getWeight() < weight)) {
      result.add(p);
    }
  }
  return result;
}
```

1. 如果强制再次使用参数化消除两者之间的重复，必然增加设计的复杂度，得不偿失
2. 参数化设计抽象能力较弱，意味着缺失更加稳定的抽象

提取抽象

```
public interface ProductSpec {
  boolean satisfy(Product product);
}
```

1. 愚弄我一次，应感到羞愧的是你；再次愚弄我，应该羞愧的是我。
2. 愿意被第一颗子弹击中，确保不再被同一支枪发射的同个方向发射的子弹。
3. 0, 1, N

分离关注点：职责单一，开放封闭

```
public static List<Product> findProducts(
  List<Product> repo, ProductSpec spec) {
  List<Product> result = new ArrayList<>();
  for (Product p : repo) {
    if (spec.satisfy(p)) {
      result.add(p);
    }
  }
  return result;
}
```

1. 集合类型：List<Product>
2. 迭代算法：线性算法
3. 匹配规则：ProductSpec

算子：封装变化 1

```
public class ColorSpec implements ProductSpec {
  private Color color;

  public ColorSpec(Color color) {
    this.color = color;
  }

  @Override
  public boolean satisfy(Product product) {
    return product.getColor() == color;
  }
}
```

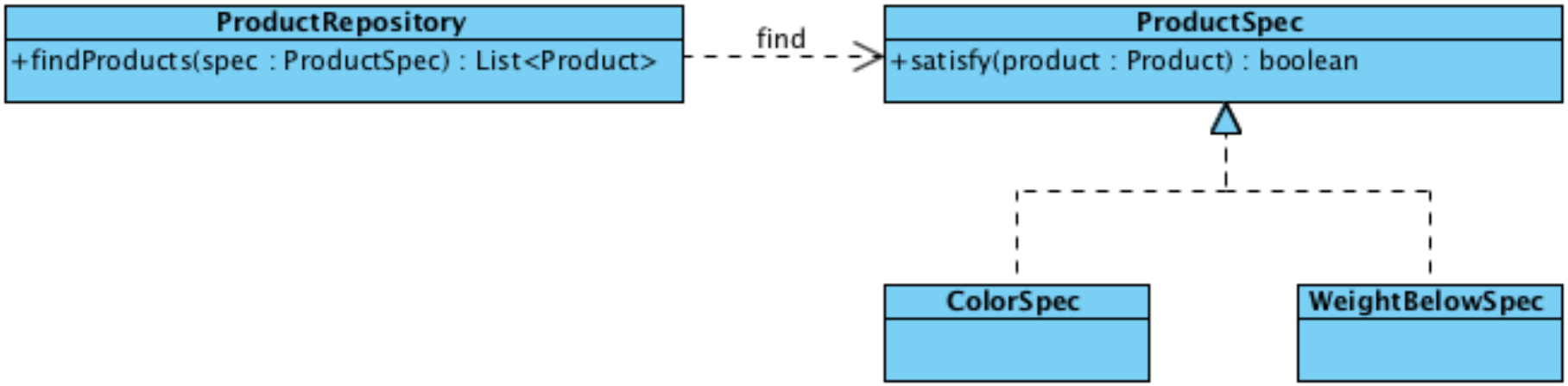
1. 算法实现模块与该具体匹配规则，通过更抽象的 ProductSpec 实现解耦

算子：封装变化 2

```
public class BelowWeightSpec implements ProductSpec {
  private int limit;

  public BelowWeightSpec(int limit) {
    this.limit = limit;
  }

  @Override
  public boolean satisfy(Product product) {
    return product.getWeight() < limit;
  }
}
```

1. 坏味道：两个算子显现结构性重复

重构到模式



1. 设计模式引入，是遵循良好的设计原则的自然结果
2. 不恰当的滥用或误用，必然会增加设计的复杂度

封装

```java
public class ProductRepository {
  private List<Product> products = new ArrayList<>();

  public void add(Product p) {
    products.add(p);
  }

  public List<Product> findProducts(ProductSpec spec) {
    List<Product> result = new ArrayList<>();
    for (Product p : products) {
      if (spec.satisfy(p)) {
        result.add(p);
      }
    }
    return result;
  }
}
```

1. 封装: 将具体实现的数据结构封装起来，并将其算法实现搬迁至领域对象

存储：客户相关

```
public class ProductDb {
  private Database db = new Database();
  private ProductRepository repo;

  public ProductDb(ProductRepository repo) {
    this.repo = repo;
  }

  public void save() {
    for (Product p : repo.getAllProducts()) {
      db.save(p);
    }
  }
}
```

1. 该业务算法实现，导致 repo.getAllProducts 接口的公开，再次开启了传递容器的可能
2. 搬迁该业务的算法实现至仓库可能不合理，甚至不可行，因为算法实现与该业务实现更加紧密

更稳定的抽象

```
@FunctionalInterface
public interface ProductConsumer {
  void accept(Product p);
}
```

1. 实现仓库与其业务 (客户) 之间的解耦

迭代器

```
public class ProductRepository {
  private List<Product> products = new ArrayList<>();

  public void add(Product p) {
    products.add(p);
  }

  public void foreach(ProductConsumer consumer) {
    for (Product p : products) {
      consumer.accept(p);
    }
  }
```

1. 而将仓库内部具体实现的数据结构，及其迭代算法封装起来，实现信息隐藏

常用算法

```
public class ProductRepository {
  ...

  public List<Product> findProducts(ProductSpec spec) {
    List<Product> result = new ArrayList<>();
    foreach(p -> {
      if (spec.satisfy(p))
        result.add(p);
    });
    return result;
  }
}
```

1. 习惯上，可以将更通用，与业务无关的算法搬迁至仓库；而将其他业务算法实现留在客户本地实现解耦

存储：客户相关

```
public class ProductDb {
  private Database db = new Database();
  private ProductRepository repo;

  public ProductDb(ProductRepository repo) {
    this.repo = repo;
  }

  public void save() {
    repo.foreach(Database::save);
  }
}
```

1. 例如，数据库相关业务逻辑，留在本地实现将更为恰当；如果搬迁至仓库，会加剧其实现的规模，甚至造成巨类的坏味道，并增加两者之间的耦合度

需求 4：查找所有颜色为红色或者绿色，并且重量小于 10 的产品

```
public class ColorAndBelowWeightSpec implements ProductSpec {
  private Color color1; private Color color2;
  private int limit;

  public ColorAndBelowWeightSpec(Color c1, Color c2,  int limit) {
    this.color1 = c1; this.color2 = c2;
    this.limit = limit;
  }

  @Override
  public boolean satisfy(Product p) {
    return (p.getColor() == color1 || p.getColor() == color2)
        && (p.getWeight() < limit);
  }
}
```

1. 已存在 ColorSpec, BelowWeightSpec 实现；实现未能更好地复用既有组件，从而导致重复设计

分离关注点：And

```
public class AndSpec implements ProductSpec {
  private ProductSpec[] specs;

  public AndSpec(ProductSpec... specs) {
    this.specs = specs;
  }

  @Override
  public boolean satisfy(Product p) {
    for (ProductSpec spec : specs) {
      if (!spec.satisfy(p))
        return false;
    }
    return true;
  }
}
```

分离关注点：Or

```
public class OrSpec implements ProductSpec {
  private ProductSpec[] specs;

  public OrSpec(ProductSpec... specs) {
    this.specs = specs;
  }

  @Override
  public boolean satisfy(Product p) {
    for (ProductSpec spec : specs) {
      if (spec.satisfy(p))
        return true;
    }
    return false;
  }
}
```

组合

```
repo.findProducts(
  new AndSpec(
    new OrSpec(new ColorSpec(RED), new ColorSpec(Greeen)),
    new BelowWeightSpec(10))
);
```

1. AndSpec, OrSpec 存在重复设计
2. 客户使用时，承担无聊的 new 表达式

消除重复：提取基类

```
class CombinableSpec implements ProductSpec {
  private ProductSpec[] specs;
  private boolean shortcut;

  protected CombinableSpec(
    ProductSpec[] specs, boolean shortcut) {
    this.specs = specs;
    this.shortcut = shortcut;
  }

  @Override
  public boolean satisfy(Product p) {
    for (ProductSpec spec : specs) {
      if (spec.satisfy(p) == shortcut)
        return shortcut;
    }
    return !shortcut;
  }
}
```

子类化：配置差异

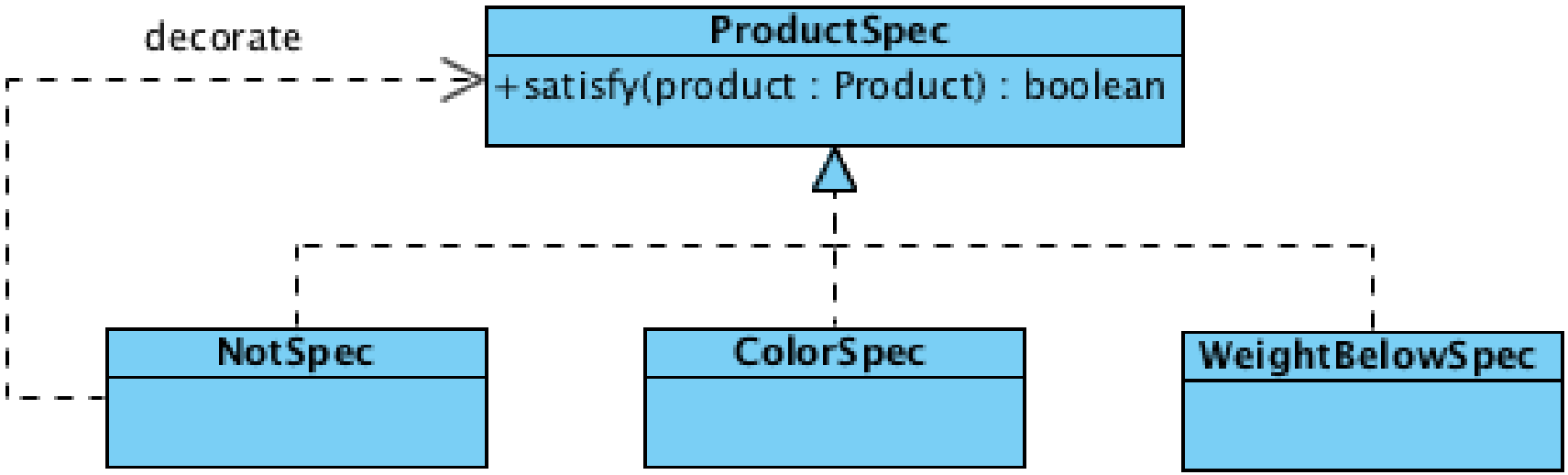
```
public class AndSpec extends CombinableSpec {
  public AndSpec(ProductSpec... specs) {
    super(Arrays.asList(specs), false);
  }
}

public class OrSpec extends CombinableSpec {
  public OrSpec(ProductSpec... specs) {
    super(Arrays.asList(specs), true);
  }
}
```

1. 提取基类后，消除了 AndSpec, OrSpec 的重复实现
2. 但是，两者依然存在结构性重复，有待进一步消除

需求 5：查找所有颜色为不是红色的产品

```
public class NotSpec implements ProductSpec {
  private ProductSpec spec;

  public NotSpec(ProductSpec spec) {
    this.spec = spec;
  }

  @Override
  public boolean satisfy(Product p) {
    return !spec.satisfy(p);
  }
}

repo.findProducts(
  new NotSpec(new ColorSpec(RED))
);
```

修饰器


decorate
ProductSpec
+satisfy(product : Product) : boolean
NotSpec
ColorSpec
WeightBelowSpec

引入工厂

```
public final class ProductSpecs {
  public static ProductSpec color(Color color) {
    return new ProductSpec() {
      @Override
      public boolean satisfy(Product p) {
        return p.getColor() == color;
      }
    };
  }

  public static ProductSpec not(ProductSpec spec) {
    return new ProductSpec() {
      @Override
      public boolean satisfy(Product p) {
        return !spec.satisfy(p);
      }
    };
  }
}
```

改善表达力

```
repo.findProducts(
  new AndSpec(
    new OrSpec(new ColorSpec(RED), new ColorSpec(Greeen)),
    new BelowWeightSpec(10)
  )
);

repo.findProducts(
  and(
    or(color(RED), color(Greeen)),
    belowWeight(10)
  )
);
```

删除 ProductSpecs: Java8

```
public interface ProductSpec {
  boolean satisfy(Product p);

  static ProductSpec color(Color color) {
    return new ProductSpec() {
      @Override
      public boolean satisfy(Product p) {
        return p.getColor() == color;
      }
    };
  }

  static ProductSpec not(ProductSpec spec) {
    return new ProductSpec() {
      @Override
      public boolean satisfy(Product p) {
        return !spec.satisfy(p);
      }
    };
  }
}
```

使用 Lambda

```
findProducts(repo, (Product p) -> p.getColor() == RED);
```

类型推演

```
findProducts(repo, p -> p.getColor() == RED);
```

重构到函数式

```
@FunctionalInterface
public interface ProductSpec {
  boolean satisfy(Product p);

  static ProductSpec color(Color color) {
    return p -> p.getColor() == color;
  }

  static ProductSpec not(ProductSpec spec) {
    return p -> !spec.satisfy(p);
  }

  ...
}
```

链式法则：实现 and/or 的中缀表达式

```
@FunctionalInterface
public interface ProductSpec {
  boolean satisfy(Product p);

  static ProductSpec color(Color color) {
    return p -> p.getColor() == color;
  }

  default ProductSpec and(ProductSpec other) {
    return (p) -> satisfy(p) && other.satisfy(p);
  }

  default ProductSpec or(ProductSpec other) {
    return (p) -> satisfy(p) || other.satisfy(p);
  }
  ...
}

repo.findProducts(color(RED).or(color(GREEN)));
```

分层设计：基础设施

```java
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);

  static Predicate<T> not(Predicate<? super T> pred) {
    return t -> !pred(t);
  }

  default Predicate<T> and(Predicate<? super T> other) {
    return t -> satisfy(t) && other.satisfy(t);
  }

  default Predicate<T> or(Predicate<? super T> other) {
    return t -> satisfy(t) || other.satisfy(t);
  }
}
```

1. 习惯上，not 使用前缀表达式，并具有最高优先级；and/or 使用中缀表达式

分层设计：领域内

```
public final class ProductSpecs {
  public static Predicate<Product> color(Color color) {
    return p -> p.getColor() == color;
  }

  public static Predicate<Product> belowWeight(int limit) {
    return p -> p.getWeight() < limit;
  }

  private ProductSpecs() {
    throw new AssertionError("no instances");
  }
}
```

1. 领域与基础设施分离，实现领域间的扩展，沉淀可复用的基础设施

参考文献

推荐书籍

- Extreme Programming Explained: Embrace Change, 2th, Kent Beck.
- Agile Software Development: Principles, Patterns and Practices, Robert C. Martin.

联系我

- Email: horance@aliyun.com
- Github: https://github.com/horance-liu
- Blog: http://www.jianshu.com/users/49d1f3b7049e

Thanks for Attending