

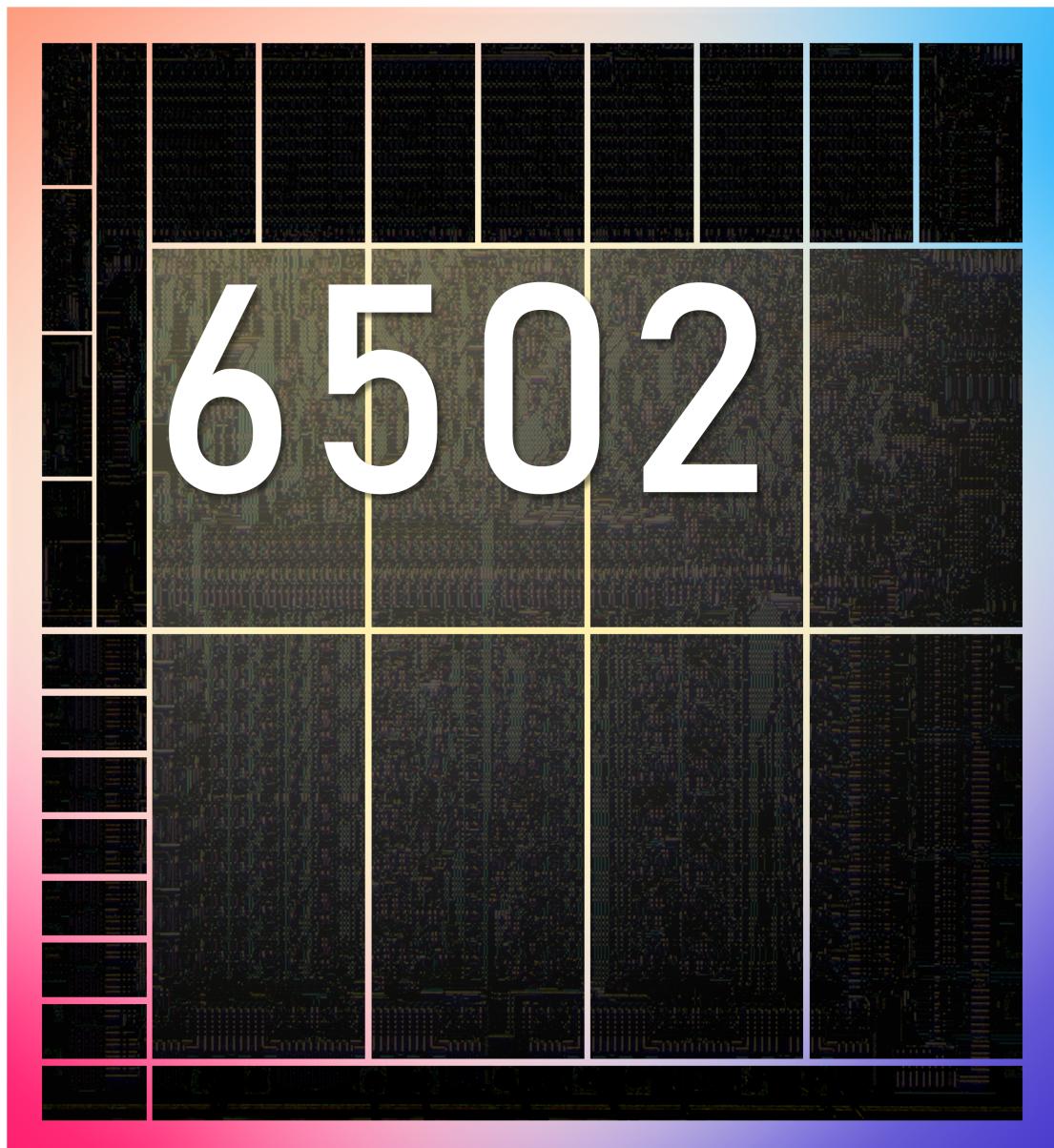
Slow6502：设计模式报告（初稿）

设计模式课程 6502 小组

软件学院

版本: 0.10

更新: December 9, 2022



Contributions

陈 晨 学号：2052717；贡献占比：12.5%。

在本项目中，我主要负责对前端的 MainWindow 模块（位于 Simulator.java）进行设计，其中使用了模板方法模式，前端控制器模式和业务代表模式。

戴仁杰 学号：19516504；贡献占比：12.5%。

在本项目中，我主要负责对 traceLog 模块前端进行设计，期间使用了数据访问对象模式，传输对象模式和 MVP 模式。

姜文渊 学号：1951510；贡献占比：12.5%。

本项目中，我主要负责项目选题和文档排版的工作，兼后端的设计模式总体设计和进度的协调工作。

李乐天 学号：1950849；贡献占比：12.5%。

在本项目中，我主要负责 MVC 模式的实现，以及空对象模式的实现，和项目整体的 UI 样式设计与实现，包括确定应用程序的整体外观、布局和样式。

孟 宇 学号：1951477；贡献占比：12.5%。

在本项目中，我主要负责了 CpuState 中通用寄存器、栈指针、程序计数器的逻辑设计与实现。还包括 ansi 控制序列以及 CPU 运行日志打印部分的代码实现。其中主要涉及到的设计模式为享元模式、适配器模式、装饰器模式、原型模式以及策略模式。

杨淳屹 学号：1953824；贡献占比：12.5%。

在本项目中，我主要负责工厂（合作完成），过滤器和迭代器模式的实现，负责部分后端代码的设计和实现，包括参与整体系统设计的讨论和分析

杨孟臻 学号：1953243；贡献占比：12.5%。

本项目中，我主要负责 CPU 和 CpuState 对象及其代理的设计实现和封装、IR 寄存器及其子类、状态寄存器 FlagRegister 的设计实现和封装，以及项目后端设计模式的综合设计。实现的设计模式为：代理模式、桥接模式、外观模式、组合模式、访问者模式。

杨 鑫 学号：1950787；贡献占比：12.5%。

在本项目中，我主要负责单例模式，工厂模式，抽象工厂模式，建造者模式，备忘录模式的实现，以及后端逻辑功能的实现。

目录

1 项目简介	6
1.1 工作内容简介	6
1.2 选题背景	7
2 设计模式详述 (GoF 模式)	9
2.1 单例模式 (Singleton)	9
2.1.1 单例模式简介	9
2.1.2 单例模式在项目中的应用	9
2.2 原型模式 (Prototype)	10
2.2.1 原型模式简介	10
2.2.2 原型模式在项目中的应用	10
2.3 工厂方法模式 (Factory Method)	10
2.3.1 工厂方法模式简介	10
2.3.2 工厂方法模式在项目中的应用	12
2.4 抽象工厂模式 (Abstract Factory)	12
2.4.1 抽象工厂模式简介	12
2.4.2 抽象工厂模式在项目中的应用	13
2.5 建造者模式 (Builder)	13
2.5.1 建造者模式简介	13
2.5.2 建造者模式在项目中的应用	14
2.6 代理模式 (Proxy)	14
2.6.1 代理模式简介	14
2.6.2 代理模式在项目中的应用	15
2.7 适配器模式 (Adapter)	16
2.7.1 适配器模式简介	16
2.7.2 适配器模式在项目中的应用	16
2.8 桥接模式 (Bridge)	16
2.8.1 桥接模式简介	16
2.8.2 桥接模式在项目中的应用	17
2.9 装饰器模式 (Decorator)	17
2.9.1 装饰器模式简介	17
2.9.2 装饰器模式在项目中的应用	18
2.10 外观模式 (Facade)	18
2.10.1 外观模式简介	18

2.10.2 外观模式在项目中的应用	19
2.11 享元模式 (Flyweight)	19
2.11.1 享元模式简介	19
2.11.2 享元模式在项目中的应用	19
2.12 组合模式 (Composite)	20
2.12.1 组合模式简介	20
2.12.2 组合模式在项目中的应用	21
2.13 模板方法模式 (TemplateMethod)	21
2.13.1 模板方法模式简介	21
2.13.2 模版方法模式在项目中的应用	23
2.14 策略模式 (Strategy)	23
2.14.1 策略模式简介	23
2.14.2 策略模式在项目中的应用	24
2.15 命令模式 (Command)	24
2.15.1 命令模式简介	24
2.15.2 命令模式在项目中的应用	25
2.16 责任链模式 (Chain of Responsibility)	25
2.16.1 责任链模式简介	25
2.16.2 责任链模式在项目中的应用	26
2.17 状态机模式 (State)	26
2.17.1 状态机模式简介	26
2.17.2 状态机模式在项目中的应用	27
2.18 观察者模式 (Observer)	28
2.18.1 观察者模式简介	28
2.18.2 观察者模式在项目中的应用	29
2.19 中介者模式 (Mediator)	29
2.19.1 中介者模式简介	29
2.19.2 中介者模式在项目中的应用	30
2.20 迭代器模式 (Iterator)	30
2.20.1 迭代器模式简介	30
2.20.2 迭代器模式在项目中的应用	31
2.21 访问者模式 (Visitor)	31
2.21.1 访问者模式简介	31
2.21.2 访问者模式在项目中的应用	32
2.22 备忘录模式 (Memento)	33
2.22.1 备忘录模式简介	33

2.22.2 备忘录模式在项目中的应用	33
2.23 解释器模式 (Interpreter)	34
2.23.1 解释器模式在项目中的应用	34
3 设计模式详述 (非 GoF 模式)	35
3.1 过滤器模式 (Filter)	35
3.2 MVC 模式	36
3.3 MVP 模式	37
3.4 数据访问对象模式	38
3.5 传输对象模式	39
3.6 空对象模式	39
3.7 业务代表模式	40
3.8 前端控制器模式	41
4 项目总结	42
4.1 硬件模拟器开发的难点	42
4.2 设计模式的意义	43

1 项目简介

1.1 工作内容简介

Slow6502 是我们小组编写的一个 MOS Technologies 6502 微处理器和兼容系统的通用模拟器。Slow6502 可以模拟 1 MHz NMOS 6502 或 CMOS 65C02、32KB RAM、16KB ROM，并且可以模拟 MOS 6551 Motorola 6850 ACIA、MOS 6522 VIA 和 6545 CRTC 等外设，从而模拟一个完整的系统。此外，Slow6502 还支持 MOS 65C02 等 6502 的变体，以及不同内存和外设地址布局的设备。

Slow6502 提供加载程序和加载 ROM 等功能，使得众多 6502 的程序可以在其上执行。此外，如果用户只有针对特定平台的 ROM 或者二进制文件，则其可以轻松地通过修改 Slow6502 的 machine 部分的源码来添加不同外设和内存布局的设备，从而运行对应的二进制文件。

该项目由 Java 8 写成，实现了共 30 个设计模式，包括 23 个 GoF 设计模式和 7 个非 GoF 设计模式。

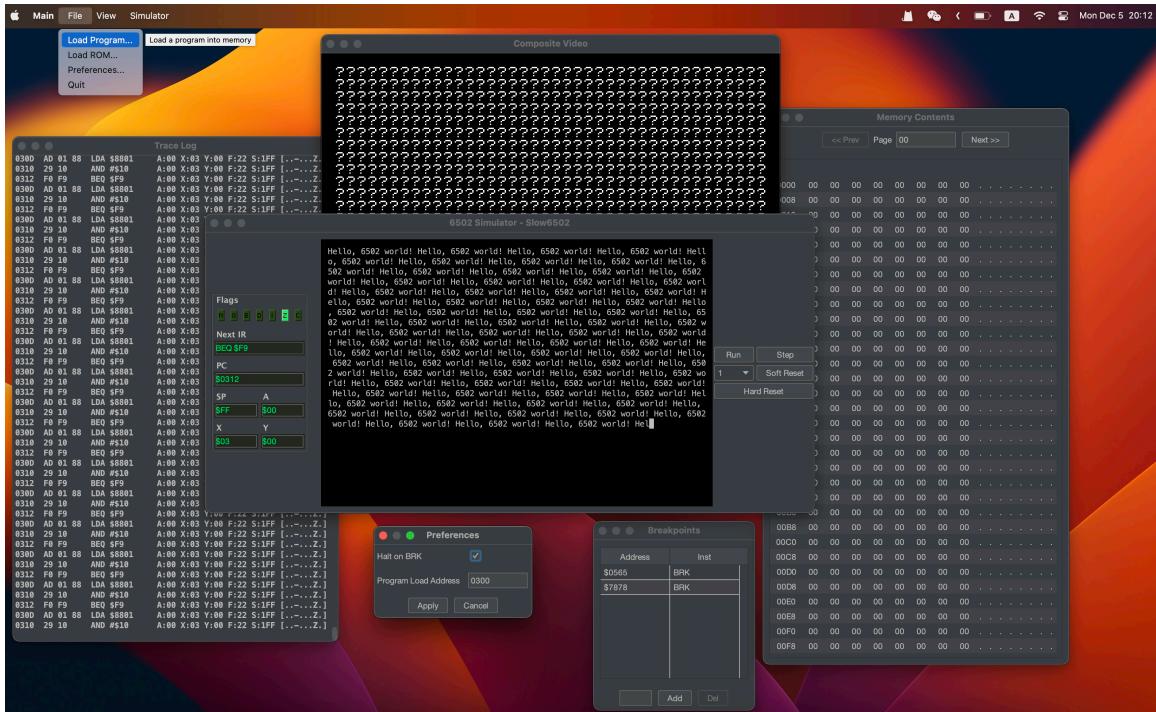


图 1: Slow6502 界面概览

1.2 选题背景

关于 **6502** MOS Technology 6502 是一款 8 位微处理器，最多可以寻址 64KB 内存，拥有两个通用寄存器，一个累加器和一个栈寄存器。



图 2: DIP-40 塑料封装的 MOS 6502 处理器（85 年 45 周产）

MOS 6502 是由 Chuck Peddle 领导的 MOS Technology 团队设计的。设计团队曾在摩托罗拉从事摩托罗拉 6800 项目；6502 本质上是该设计的简化、更便宜和更快的版本。

在 1975 年推出时，6502 是市场上价格最低的微处理器，遥遥领先。它最初的售价不到 6800 或 Intel 8080 等大公司竞争设计成本的六分之一。它的推出导致整个处理器市场的价格迅速下降。它与 Zilog Z80 一起引发了一系列项目，导致了 80 年代初期的家用电脑革命。

1980 年代和 90 年代初期流行的视频游戏机和家用电脑，例如 Atari 2600、Atari 8 位系列、Apple II、Nintendo Entertainment System、Commodore 64、Atari Lynx、BBC Micro 等，使用 6502 或 6502 的变体基本设计。6502 推出后不久，MOS Technology 被 Commodore International 彻底收购，Commodore International 继续向其他制造商销售微处理器和许可。在 6502 的早期，它由 Rockwell 和 Synertek 二次采购，后来授权给其他公司。

时至今日，依然有诸多爱好者仿制 6502 和用它搭建怀旧电脑。即使和同时代的英特尔 8080，摩托罗拉 6800 相比，它也是平平无奇的：3510 个晶体管，56 个指令，最高 3 MHz 的主频。看似平庸的 6502 依靠着仅有对手六分之一的价格、亲民的姿态、和车库文化交融的理念构成了美国 60 到 70 一代年轻计算机爱好者的集体记忆。

MOS 6502 和我国的计算机产业 得益于 MOS 6502 廉价的设计思路和良好的生态，其可能是第一个能广泛被中国学生们用到的 CPU。

著名的中华学习机 CEC-I 就“使用”了 MOS 6502 作为其 CPU。这款由电子工业部计算机与信息局组织，清华大学主持联合设计，电子部六所、国营 734 厂、陕西省计算机厂以及华明计算机有限公司参与研制的机型是 Apple II 的仿制品。

这款机器是芯片级仿制的 Apple II 计算机，如果你拆开机器，会发现除了某些早期试验型号，里面并没有使用 MOS 6502 CPU。而是一些标着 ZH-6、ZH-7、ZH3065 的芯片，而这些芯片就是对 Apple II 内部芯片的完全仿制，包括 MOS 6502。在那个靠 decapping 技术就可以了解 CPU 内部构造的年代，这款机器的诞生虽然不能说是容易的，但也是很神奇了。而且这款机器内置了汉卡和 80 列卡。这两款卡在一般 Apple II 上是需要单独购买的，有了这两块卡，不但可以显示和输入汉字，还可以显示更多的列数，让一行显示的字符数变多一倍，由



图 3: 一块中华学习机的主板，可见中间 DPI-40 封装的 CPU

于中文通常要占用两个字符宽度，这对于中文使用是非常重要的。

这款机器的价格大约是当时中产阶级家庭一年的收入，虽然并不能做到家用，但是很多学校都有一两台这个机器。至少让很多人从小接触到了计算机。

除了中华学习机外，上世纪末红遍大江南北的 FC 兼容机/小霸王学习机们和后来几乎人手一台的文曲星都采用了 6502 CPU。从诞生至今的 40 多年来，6502 对个人电脑和家用游戏主机行业产生了极其深刻的影响，无数人的人生因此而改变。虽然小霸王和文曲星早已经不再流行，各类 NES 上的游戏也逐渐被人遗忘，但直到现在，6502 仍被运用于数以亿计的工业监测和控制计算机当中，为我们服务。

2 设计模式详述（GoF 模式）

2.1 单例模式（Singleton）

2.1.1 单例模式简介

单例模式是一种常用的软件设计模式，它保证一个类只有一个实例，并提供一个全局访问点来访问该实例。

单例模式通常用于某个类需要在整个应用程序中只有一个实例的场景，例如系统的配置类或者线程池。

要实现单例模式，一般需要将类的构造函数设为私有的，并且提供一个静态方法用于获取该类的唯一实例。这样，在整个应用程序中就只能通过这个静态方法来访问该类的实例。

单例模式有一些优点：

可以保证一个类只有一个实例，避免了对象的多次创建，降低了系统的内存开销。

可以提供一个全局访问点，方便访问该实例，避免了创建多个实例带来的麻烦。

可以提供一个统一的访问接口，方便维护和扩展该实例。

但是，单例模式也有一些缺点：

单例类的扩展有一定的困难，因为单例类的实例数量是固定的。

如果单例类中存在大量长生命周期的资源，那么单例模式还有一个缺点就是难以测试，因为它的实例是全局的，如果要测试单例类的方法，那么需要改变全局的实例，这样就会影响到其他的测试用例。

在实际开发中，需要根据实际情况来判断是否使用单例模式。如果需要保证一个类只有一个实例，并提供一个全局访问点，那么可以考虑使用单例模式。

2.1.2 单例模式在项目中的应用

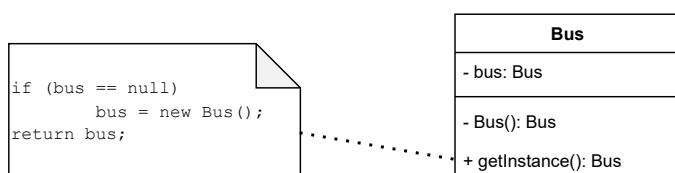


图 4: 单例模式在 Slow6502 中的类图

本项目中 CPU 的 Bus 类被设计为单例模式，因为我们的 CPU 的 Bus 始终只能有一个。这样设计可以确保一个项目中只存在一个 Bus 对象，这样可以避免在项目中出现多个不同的

Bus 对象造成的冲突。另外，使用单例模式还可以提高系统的效率，因为在整个程序运行期间，它只会创建一个 **Bus** 对象，而不会每次都创建一个新的对象。

此外，通过将 **Bus** 类设计为单例模式，可以更容易地访问和控制 **Bus** 类的实例。例如，在项目中可以通过调用单例类的静态方法来访问 **Bus** 对象，这样就可以确保所有对 **Bus** 类的访问都是通过同一个对象进行的。这有助于提高系统的一致性和可靠性。

2.2 原型模式（Prototype）

2.2.1 原型模式简介

原型模式是一种创建型设计模式，它的目的是通过复制一个已经存在的对象来创建新的对象，从而减少创建新对象的开销。原型模式通常适用于创建重复次数较多的对象，例如创建大量相同或相似的对象。

要实现原型模式，一般需要实现一个接口或抽象类，该接口或抽象类包含一个 `clone` 方法，用于复制当前对象。然后，需要创建一个具体的原型类来实现这个接口或抽象类，并在其 `clone` 方法中实现对象的复制。

原型模式有一些优点：

通过复制已经存在的对象，可以避免重新创建对象的开销，提高了效率。可以通过改变复制出来的对象的属性值来实现对象的定制化，满足了开发者的个性化需求。但是，原型模式也有一些缺点：

在实现原型模式时，需要实现一个接口或抽象类，这增加了系统的复杂度。

在 Java 中实现原型模式还需要实现 `Cloneable` 接口，并重写 `clone` 方法。由于 Java 语言中的 `clone` 方法是一个比较复杂的方法，它涉及到对象的浅复制和深复制，所以在使用时需要注意这些细节。

通常，原型模式适用于创建重复次数较多的对象，例如创建大量相同或相似的对象。例如，在游戏开发中，如果需要创建大量的游戏元素，比如怪物、道具等，那么可以使用原型模式来实现这些对象的快速创建。

此外，在项目开发中，如果需要从一个已经存在的对象中创建新对象，并且希望新对象具有与已经存在的对象相似的结构和功能，那么也可以考虑使用原型模式来实现这个功能。

总的来说，原型模式是一种非常有用的设计模式，它可以通过复制已经存在的对象来实现快速创建新对象的功能。在实际开发中，应根据实际情况来判断是否使用原型模式。

2.2.2 原型模式在项目中的应用

2.3 工厂方法模式（Factory Method）

2.3.1 工厂方法模式简介

工厂方法模式是一种创建型设计模式，它提供了一种创建对象的最佳方式。在工厂方法模式中，通过定义工厂类来负责创建具体的对象，通过传递不同的参数来创建不同的对象。工

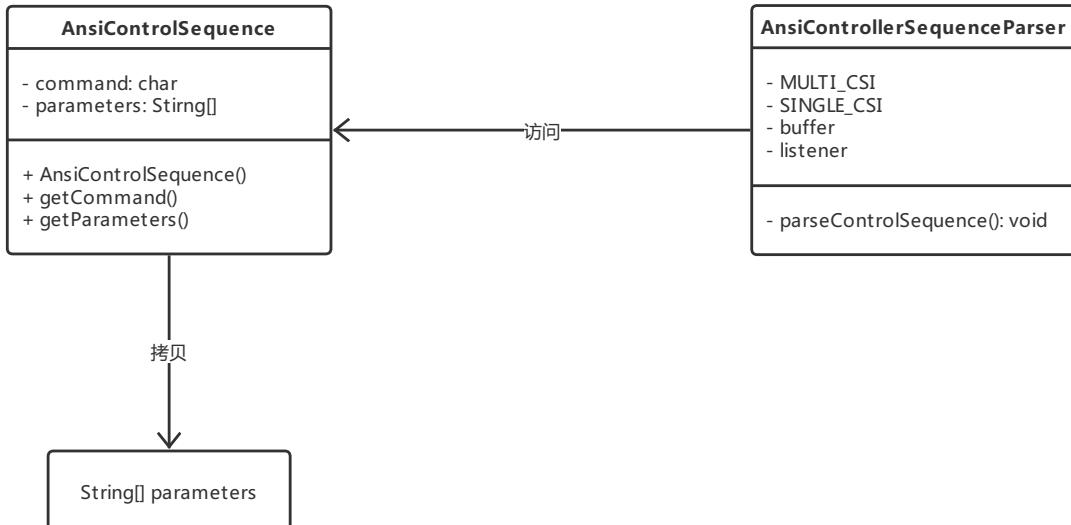


图 5: 原型模式在 Slow6502 中的类图

工厂方法模式的优点是可以将对象的创建和使用分离，通过使用不同的工厂类来创建不同的对象，可以提高系统的灵活性和可扩展性。

要实现工厂方法模式，需要定义一个抽象工厂类来声明创建对象的接口，并定义一个具体工厂类来实现抽象工厂类中声明的创建对象的方法。通常，抽象工厂类和具体工厂类都需要实现一个接口或抽象类，该接口或抽象类定义了具体的对象所具有的功能。

工厂方法模式有一些优点：

可以将对象的创建和使用分离，使系统更加灵活和可扩展。

在工厂方法模式中，客户端可以通过传递不同的参数来创建不同的对象，并且不需要关心对象的创建细节。

工厂方法模式提供了一种更好的扩展方式。

好的，那么工厂方法模式还有一些缺点：

在工厂方法模式中，如果需要增加新的产品，则需要修改抽象工厂类和具体工厂类，这违背了开闭原则。

在工厂方法模式中，增加新的产品时需要对抽象工厂类进行修改，这会导致抽象工厂类变得臃肿，不利于维护。

通常，工厂方法模式适用于以下场景：

创建对象的任务由多个具体子类中的某一个来完成，由客户端决定实例化哪一个具体子类。

需要将对象的创建和对象的使用分离。

在实际开发中，应根据实际情况来判断是否使用工厂方法模式。如果需要将对象的创建和使用分离，并且希望通过传递不同的参数来创建不同的对象，那么可以考虑使用工厂方法模式来实现这个功能。

2.3.2 工厂方法模式在项目中的应用

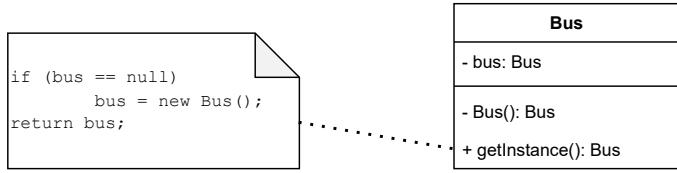


图 6: 工厂方法模式在 Slow6502 中的类图

我们的 Memory 类使用了工厂模式，两个属于 Memory 的类（RAM, ROM）可以通过 MemoryFactory 创建。使用工厂模式可以使得在创建 Memory 类的实例时，将具体的内存创建过程隔离出来。这样，如果你希望更改内存创建的方式，比如使用不同的类型的内存或者使用不同的方法来创建内存，只需要修改工厂类的代码，而不需要修改 Memory 类的代码。这样可以让你更方便地扩展和修改项目，并且降低了代码的耦合度。

2.4 抽象工厂模式 (Abstract Factory)

2.4.1 抽象工厂模式简介

抽象工厂模式是一种创建型设计模式，它提供了一种创建相关或依赖对象的最佳方式。在抽象工厂模式中，抽象工厂类声明了创建一组相关或依赖对象的方法，而具体工厂类则实现了抽象工厂类中声明的创建对象的方法。

要实现抽象工厂模式，需要定义一个抽象工厂类来声明创建一组相关或依赖对象的接口，并定义一个具体工厂类来实现抽象工厂类中声明的创建对象的方法。通常，抽象工厂模式中的抽象工厂类和具体工厂类都需要实现一个接口或抽象类，该接口或抽象类定义了一组创建相关或依赖对象的操作。这些操作包括创建对象的方法和一些必要的操作。通过这个接口或抽象类，抽象工厂类和具体工厂类可以定义相同的创建对象的方法，并且可以通过实现该接口或抽象类来实现相同的操作。

抽象工厂模式有一些优点：

抽象工厂模式提供了一种创建一组相关或依赖对象的最佳方式，可以在系统中动态替换产品系列，增强系统的灵活性和可扩展性。在抽象工厂模式中，客户端不需要关心所创建的具体对象是由哪个具体工厂创建的，只需要知道它所对应的抽象工厂即可。抽象工厂模式提供了一种更好的扩展方式。

抽象工厂模式也有一些缺点：

在抽象工厂模式中，如果需要增加新的产品，则需要修改抽象工厂类和具体工厂类，这违背了开闭原则。在抽象工厂模式中，增加新的产品时需要对抽象工厂类进行修改，这会导

致抽象工厂类变得臃肿，不利于维护。

通常，抽象工厂模式适用于以下场景：

需要创建一组相关或依赖的对象，且对象的类型由应用程序决定。

需要将对象的创建和对象的使用分离，并且希望在系统中动态替换产品系列。

需要提供一个产品族的产品的接口，而又不指定实际的产品。

抽象工厂模式提供了一种创建一组相关或依赖对象的最佳方式，可以在系统中动态替换产品系列，增强系统的灵活性和可扩展性。因此，如果需要创建一组相关或依赖的对象，并且希望在系统中动态替换产品系列，可以考虑使用抽象工厂模式来实现这个功能。

2.4.2 抽象工厂模式在项目中的应用

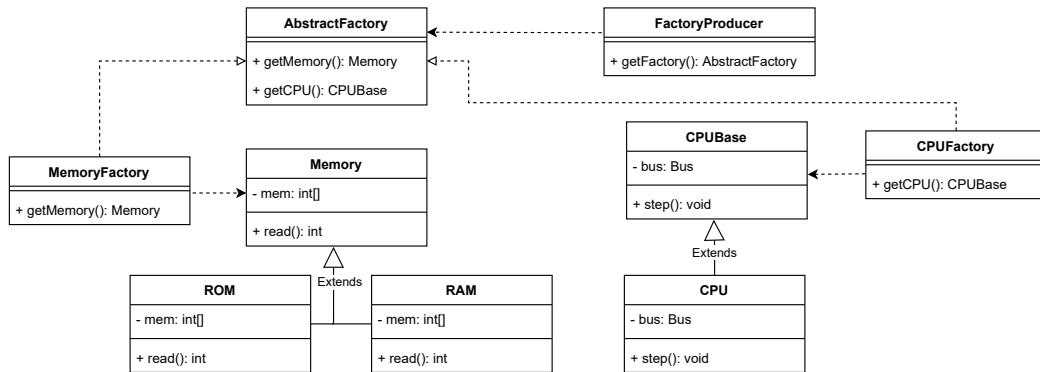


图 7: 抽象工厂模式在 Slow6502 中的类图

2.5 建造者模式 (Builder)

2.5.1 建造者模式简介

建造者模式是一种创建型设计模式，它提供了一种将一个复杂对象的构建与它的表示分离的方式，通过定义一个建造者来构建复杂对象的不同部分，最终构建出完整的复杂对象。

建造者模式包括如下角色：

建造者 (Builder)：定义构建复杂对象的接口，包括构建复杂对象的各个部分的操作。
具体建造者 (Concrete Builder)：实现建造者接口，完成构建复杂对象的各个部分的操作。
指挥者 (Director)：负责构建复杂对象，通过调用建造者接口中的方法来构建复杂对象的各个部分。
产品 (Product)：表示复杂对象，包含多个组成部分。

通常，建造者模式适用于以下场景：

在创建一个复杂对象的同时，需要指定其各个部分的构建顺序。创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式。在创建复杂对象时，提供一个高层接口来指定各个部分的装配方式，并不暴露该对象的内部组成部分。建造者模式提供了一种将复杂

对象的构建与它的表示分离的方式，可以使得客户端不必知道复杂对象的具体构建细节，且可以更加精细地控制复杂对象的构建过程。

2.5.2 建造者模式在项目中的应用

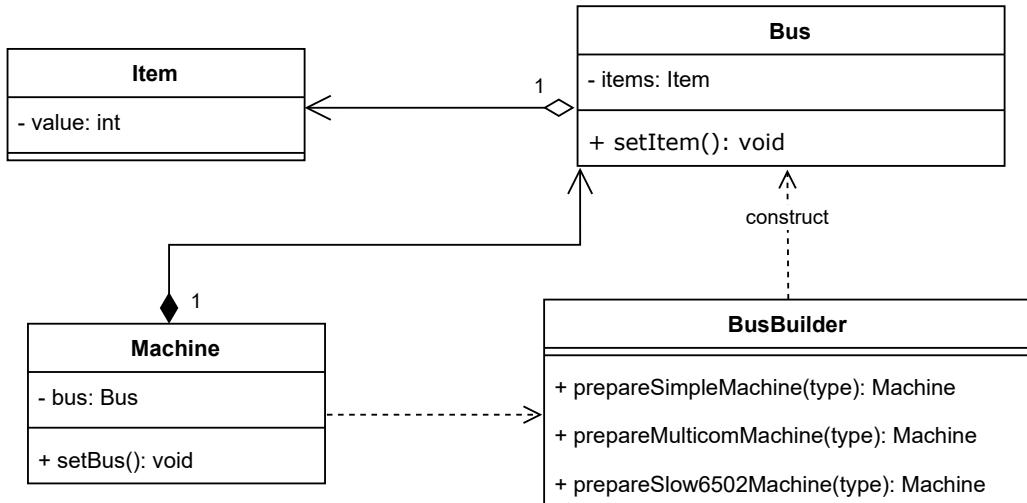


图 8: 建造者模式在 Slow6502 中的类图

项目中 Bus 的构建使用了建造者模式，在不同的 Machine 里面可以使用不同的组装方法来构造出不同的 Bus，并且可以在不同的 Machine 中使用。这样，在不同的 Machine 中，就可以使用适当的组装方法来构造出最合适的 Bus，从而提高系统的灵活性和可扩展性。

另外，使用建造者模式还可以使得在构建 Bus 时分离具体的构造过程和最终的产品。这样，可以在不影响最终的产品的情况下，修改或替换构造过程中的某些步骤，这有助于提高项目的可维护性和可扩展性。

2.6 代理模式（Proxy）

2.6.1 代理模式简介

代理模式是一种设计模式，它提供了一种方法，通过使用一个中介对象来间接访问另一个对象，从而可以在不暴露目标对象的内部实现的情况下，对目标对象进行操作。

在代理模式中，代理对象充当目标对象的接口，从而可以控制对目标对象的访问。这种模式通常用于对目标对象进行限制、增强或保护性包装。

代理模式可以用于很多不同的应用场景，例如远程代理、虚拟代理、保护代理和缓存代理等。

在远程代理模式中，代理对象在客户端所在的机器上，而目标对象在远程服务器上，代理对象通过网络与目标对象进行交互。这种模式可以帮助客户端程序在本地访问远程对象，从而减少网络延迟。

在虚拟代理模式中，代理对象控制对目标对象的访问，并在客户端需要时才会创建目标对象。这种模式可以帮助减少系统开销，因为目标对象只在实际需要时才会创建。

在保护代理模式中，代理对象对目标对象进行保护性包装，只有当客户端满足特定条件时才能访问目标对象。这种模式可以用于实现访问控制，例如在需要身份验证的情况下。

在缓存代理模式中，代理对象缓存目标对象的结果，从而减少对目标对象的重复访问。这种模式可以帮助提高系统性能，因为通过减少对目标对象的访问，可以消除不必要的开销。

2.6.2 代理模式在项目中的应用

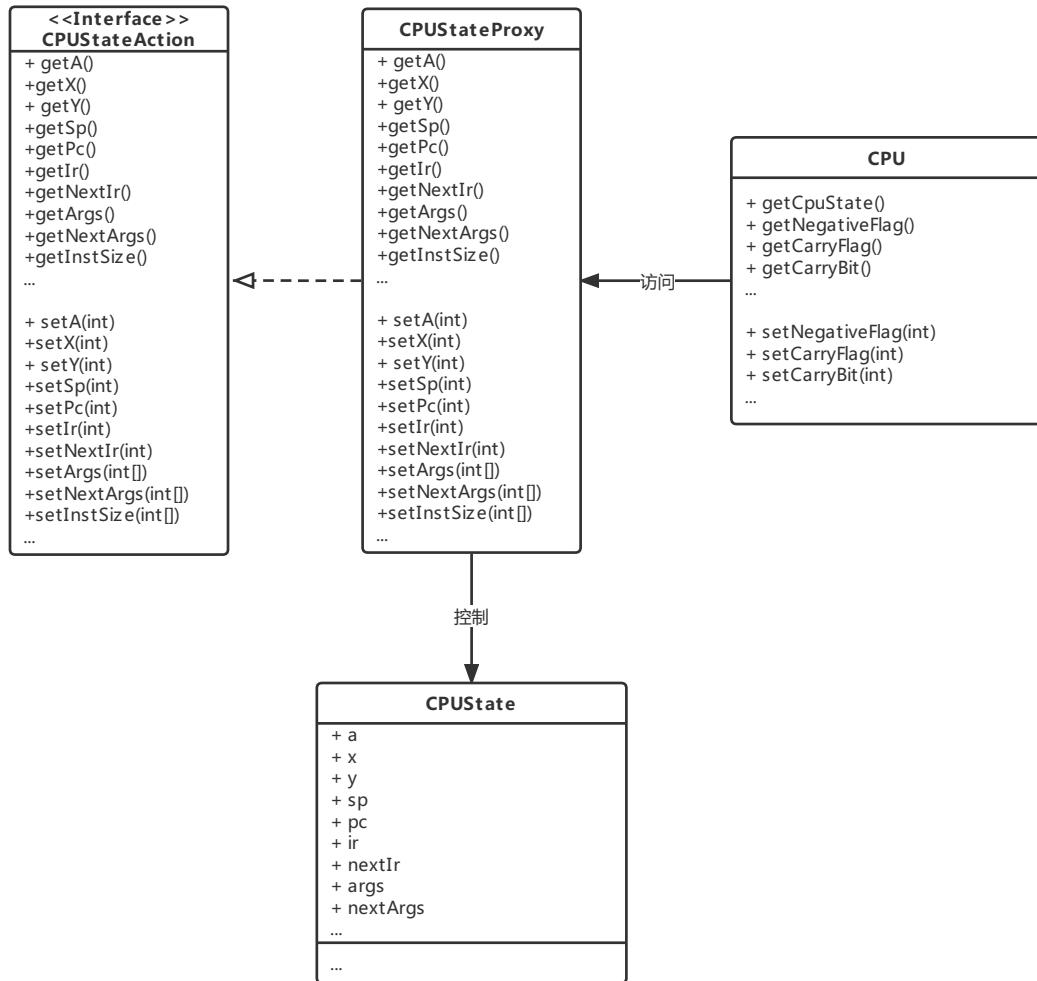


图 9: 代理模式在 Slow6502 中的类图

我们的项目中，使用代理模式包装对 CPU 内部状态的访问和修改。使用代理模式包装对 CPU 内部状态的访问和修改可以在不暴露 CPU 内部实现细节的情况下对其进行操作。这样设计可以帮助保护 CPU 的实现，并且使得模拟器的其他部分只能通过代理对象来访问和修改 CPU 的状态，从而可以更好地控制对 CPU 的访问。

另外，使用代理模式还可以帮助模拟器更好地进行性能优化。例如，代理对象可以在内部缓存 CPU 的状态，从而减少对 CPU 的重复访问，从而提高模拟器的性能。

2.7 适配器模式 (Adapter)

2.7.1 适配器模式简介

适配器模式是一种常用的设计模式。它可以将一个类的接口转换成客户希望的另外一个接口。适配器模式通过创建一个包装类，将原有类作为包装类的一个属性，来实现对原有类接口的转换。

适配器模式还可以用于解决接口不兼容的问题。例如，当我们需要使用一个第三方库中的类时，但该类的接口与我们自己的类不兼容，就可以使用适配器模式来解决这个问题。我们可以创建一个适配器类，将第三方库中的类作为适配器类的属性，并实现与我们自己的类兼容的接口。

2.7.2 适配器模式在项目中的应用

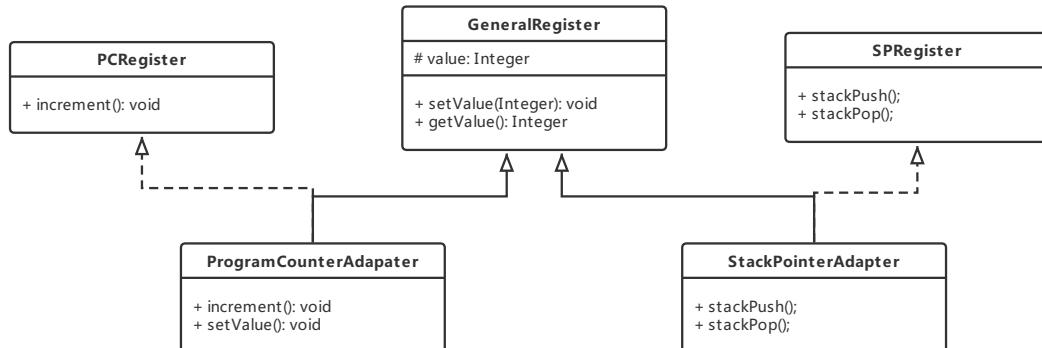


图 10: 适配器模式在 Slow6502 中的类图

我们使用适配器模式将通用寄存器的接口改为栈寄存器和 PC 寄存器的接口。在这种情况下，可能存在一些代码依赖于通用寄存器的接口，但是你希望使用栈寄存器和 PC 寄存器。为了解决这个问题，你可以使用适配器模式来创建一个新的类，该类实现通用寄存器的接口，并将调用转发给栈寄存器和 PC 寄存器。这样，你就可以在不改变现有代码的情况下使用栈寄存器和 PC 寄存器。

总之，使用适配器模式来解决这个问题是合适的。它可以在不改变现有代码的情况下使用栈寄存器和 PC 寄存器，从而提高代码的灵活性和可维护性。

2.8 桥接模式 (Bridge)

2.8.1 桥接模式简介

桥接模式 (Bridge Pattern) 是一种结构型设计模式，它将抽象部分与它的实现部分分离，使它们都可以独立地变化。这种模式提供了一种将抽象部分与实现部分分离的方法，从而使

得它们可以独立地变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

2.8.2 桥接模式在项目中的应用

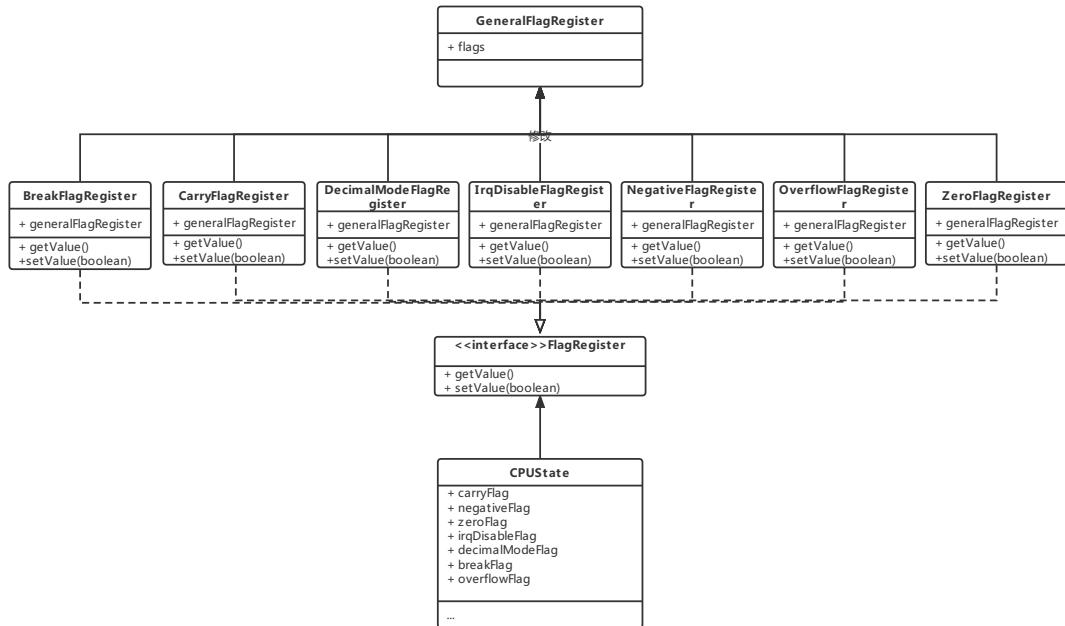


图 11: 桥接模式在 Slow6502 中的类图

在我们的项目中，我们使用桥接模式包装对状态寄存器各个位的访问。使用桥接模式包装对状态寄存器各个位的访问可以有效地分离抽象部分（对状态寄存器各个位的访问）与实现部分（状态寄存器的存储）。

这样设计的好处在于，可以使得抽象部分和实现部分可以独立地变化。例如，如果你需要更改 MOS 6502 模拟器中状态寄存器的存储方式，可以只修改实现部分的代码，而不需要更改对状态寄存器各个位的访问的代码。这样，你就可以更加高效地开发和维护 MOS 6502 模拟器。

另外，使用桥接模式还可以使得 MOS 6502 模拟器的设计更加模块化，更易于复用。例如，如果你需要在其他项目中使用 MOS 6502 模拟器中实现的对状态寄存器各个位的访问功能，你可以将这部分功能抽象为接口，然后在其他项目中直接调用这个接口，而无需拷贝代码。这样，你就可以在不同项目中复用 MOS 6502 模拟器中的代码，提高开发效率。

2.9 装饰器模式 (Decorator)

2.9.1 装饰器模式简介

装饰器模式是一种常用的设计模式。它可以在不更改原有对象的基础上，通过组合的方式为对象添加新的功能。装饰器模式通过创建一个装饰类，并将原有对象作为装饰类的一个属性，来实现对原有对象的功能扩展。

例如，在游戏中，可能会有一个基础的人物类，该类可以描述人物的基本信息和属性。如果我们想为人物添加新的能力，例如飞行能力，就可以使用装饰器模式。我们可以创建一个装饰类，将人物类作为该装饰类的一个属性，并在装饰类中定义飞行能力的相关方法。这样，我们就可以在不更改原有人物类的基础上，为人物添加飞行能力。

2.9.2 装饰器模式在项目中的应用

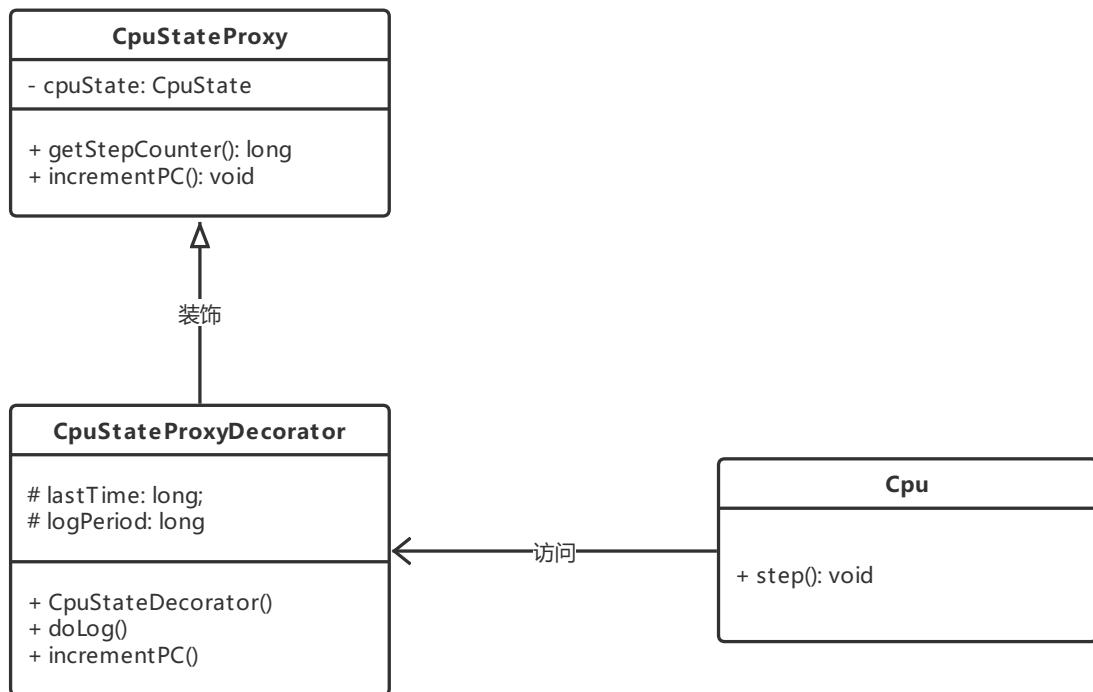


图 12: 装饰器模式在 Slow6502 中的类图

在我们的项目中，我们使用装饰器模式使一些类能够记录相关的日志。使用装饰器模式来实现日志记录是十分自然的。装饰器模式可以在不更改原有对象的基础上，为对象添加新的功能。因此，我们可以通过创建一个装饰类，并将需要记录日志的类作为该装饰类的一个属性，来实现日志记录的功能。这样，我们就可以在不更改原有类的基础上，为这些类添加日志记录的能力。

另外，使用装饰器模式实现日志记录还可以提高代码的可维护性。当我们需要更改日志记录的方式时，只需要更改装饰类的实现，而不需要更改原有的类。这样可以避免对原有类的修改，提高代码的可维护性。

2.10 外观模式 (Facade)

2.10.1 外观模式简介

外观模式是一种设计模式，它允许一个类提供简单的接口来访问一个子系统中的一组类。这个模式为子系统中的一组复杂的类提供一个简单的接口，使得子系统更容易使用。

外观模式的主要用途是简化接口，减少客户端与子系统之间的依赖。这样，客户端可以更简单地使用子系统，而不必深入了解其内部工作原理。通过使用外观模式，可以让子系统的实现更容易扩展和更新，因为客户端的代码不会受到影响。

外观模式的优点包括：

1. 简化了接口：通过提供一个统一的接口，外观模式可以简化访问子系统的接口，使得子系统更容易使用。
2. 减少了客户端与子系统之间的依赖：外观模式可以隐藏子系统的内部工作原理，减少了客户端对子系统内部实现的依赖。
3. 提高了系统的灵活性：通过使用外观模式，可以让子系统的实现更容易扩展和更新，因为客户端的代码不会受到影响。

外观模式的缺点包括：

1. 可能会过度抽象：如果外观模式的接口过于抽象，可能会导致客户端无法理解和使用它。
2. 可能会增加系统的复杂度：如果外观模式过于复杂，可能会增加系统的复杂度，导致系统难以维护和扩展。
3. 可能会掩盖子系统的问题：如果子系统存在问题，外观模式可能会掩盖这些问题，导致客户端无法及时发现并解决问题。

2.10.2 外观模式在项目中的应用

本项目中，我们的 `CPU State` 类就是一个 `Facade`。如果不使用这个 `Facade`，那么各个寄存器的操作逻辑会被分散在 `CPU` 类中的各个部分。而使用了 `Facade` 模式之后，`CPU` 类可以使用 `CPU State` 提供简单的接口来访问一个子系统中的一组类，从而简化了逻辑，使其他类可以更简单地使用子系统，而不必深入了解其内部工作原理。

2.11 享元模式（Flyweight）

2.11.1 享元模式简介

享元模式是一种常用的设计模式。它通过共享来有效地支持大量细粒度的对象。享元模式通过共享内部状态来减少内存的使用，并用不同的外部状态来区分不同的对象。这样可以在不需要额外内存的情况下创建大量的对象。

例如，在游戏中，可能会有大量的敌人。如果每个敌人都是一个单独的对象，将会消耗大量的内存。如果使用享元模式，可以将所有敌人都共享一个基础对象，并使用不同的外部状态来区分不同的敌人。这样就可以在不需要额外内存的情况下创建大量敌人。

2.11.2 享元模式在项目中的应用

在我们的项目中，享元模式被应用在管理寄存器的值这个任务上。享元模式在这种情况下可以帮助减少内存的使用。寄存器通常都是固定大小的，因此使用享元模式可以让我们创建大量的寄存器，而不需要为每个寄存器单独分配内存。相反，我们可以共享一个基础对象来管理寄存器的值，并用不同的外部状态来区分不同的寄存器。

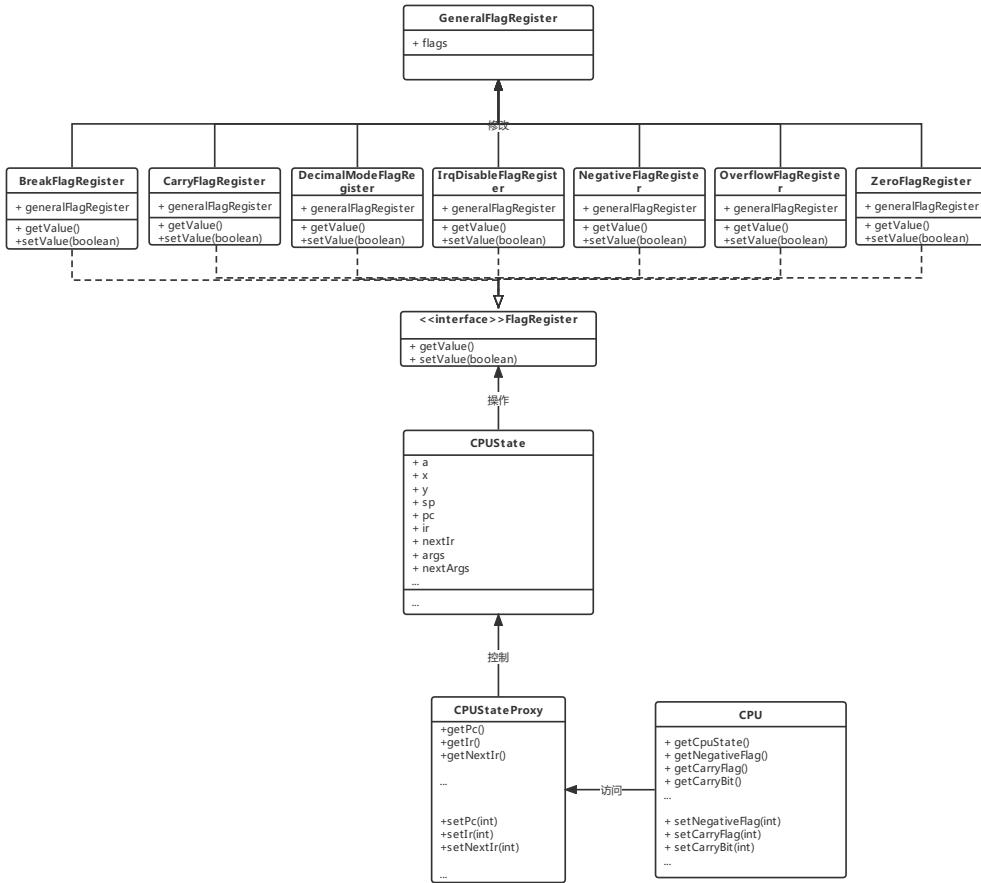


图 13: 外观模式在 Slow6502 中的类图

另外，使用享元模式还可以提高程序的性能。对于大量细粒度的对象，每个对象都需要单独分配内存和维护其状态。这样会消耗大量的 CPU 时间和内存空间。通过使用享元模式，可以减少内存分配和状态维护的开销，从而提高程序的性能。

2.12 组合模式 (Composite)

2.12.1 组合模式简介

组合模式是一种设计模式，它允许一组对象被当作单个对象对待。这样，用户可以对单个对象和组合对象进行相同的操作，无需关心它们是单个对象还是组合对象。

组合模式的主要用途是让用户对单个对象和组合对象进行相同的操作，而无需关心它们是否为组合对象。这样，用户可以更方便地管理一个对象层次结构，并对其进行操作。

例如，在文件系统中，一个文件夹可以包含多个文件和子文件夹。使用组合模式，用户可以对文件夹和文件进行相同的操作，比如查看、删除和复制。无需关心它们是文件夹还是文件，就可以对它们进行统一的处理。

组合模式的优点包括：

- 统一了对象层次结构：组合模式可以将单个对象和组合对象进行统一，让用户可以对它们进行统一的操作。

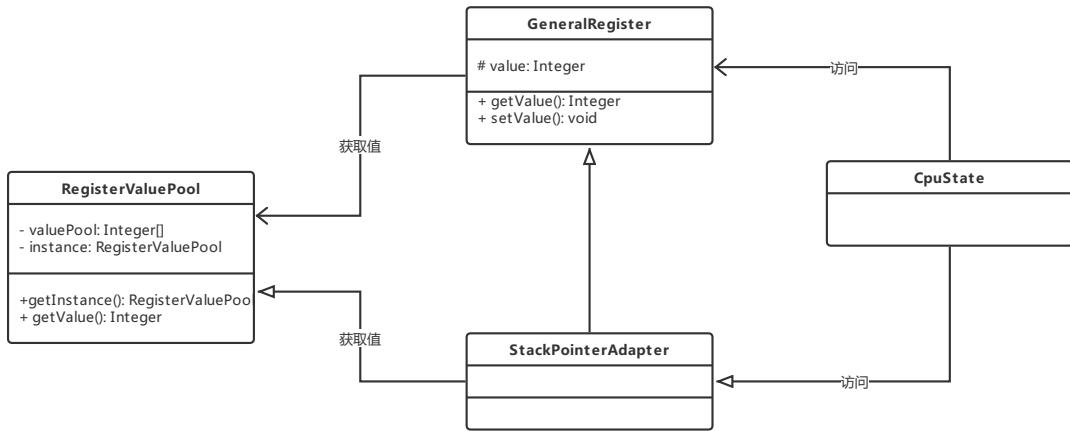


图 14: 享元模式在 Slow6502 中的类图

们进行相同的操作。

2. 方便对对象层次结构的管理: 使用组合模式, 用户可以方便地管理一个对象层次结构, 并对其进行操作。
3. 增加系统的灵活性: 组合模式可以让用户在运行时动态地组合对象, 增加系统的灵活性。组合模式的缺点包括:
 1. 增加系统的复杂度: 组合模式可能会增加系统的复杂度, 因为它需要设计一个组合类来管理单个对象和组合对象。
 2. 可能会过度抽象: 如果组合类的接口过于抽象, 可能会导致用户难以理解和使用它。
 3. 可能会掩盖组合对象的问题: 如果组合对象存在问题, 组合模式可能会掩盖这些问题, 导致用户无法及时发现并解决问题。

2.12.2 组合模式在项目中的应用

在我们的项目中, 我们使用组合模式构建使用的 CPU State。由于 CPU State 中包含大量不同的更小的组成部分 (寄存器、状态位) 等, 故而使用组合模式可方便地管理 CPU State 的层次结构, 并对其进行操作。

2.13 模板方法模式 (TemplateMethod)

2.13.1 模板方法模式简介

模板方法模式是一种设计模式, 它定义了一个算法的步骤, 并允许子类为一个或多个步骤提供实现。这样, 子类可以在不改变算法架构的情况下, 重新定义算法的某些步骤。

模板方法模式的主要用途是让子类可以重新定义算法的某些步骤, 而不改变算法的架构。这样, 子类可以根据自己的需要, 对算法进行定制, 从而获得更高的灵活性和可扩展性。

例如, 在银行业务处理中, 可能需要对不同类型的账户进行处理, 比如活期账户、定期账

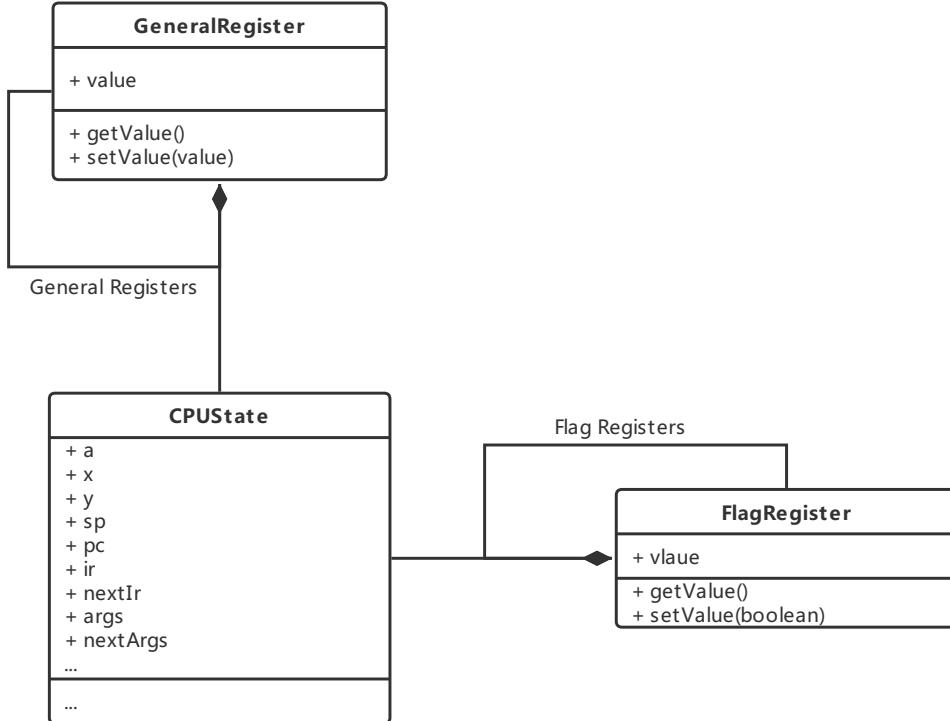


图 15: 组合模式在 Slow6502 中的类图

户和信用卡账户。使用模板方法模式，可以定义一个模板方法来处理这些账户，并允许子类为某些步骤提供实现，比如计算利息和扣除手续费。这样，子类就可以根据自己的需要，对算法进行定制，从而支持不同类型的账户。

模板方法模式的优点包括：

1. 封装算法架构：模板方法模式可以封装算法的架构，让子类可以重新定义算法的某些步骤。
2. 提高代码的复用性：通过抽象父类和模板方法，模板方法模式可以提高代码的复用性，减少重复的代码。
3. 提高系统的灵活性：模板方法模式可以让子类对算法进行定制，从而提高系统的灵活性和可扩展性。

模板方法模式的缺点包括：

1. 可能会过度抽象：如果模板方法的接口过于抽象，可能会导致子类难以理解和使用它。
2. 可能会增加系统的复杂度：如果模板方法过于复杂，可能会增加系统的复杂度，导致系统难以维护和扩展。
3. 可能会掩盖子类的问题：如果子类存在问题，模板方法可能会掩盖这些问题，导致用户无法及时发现并解决问题。

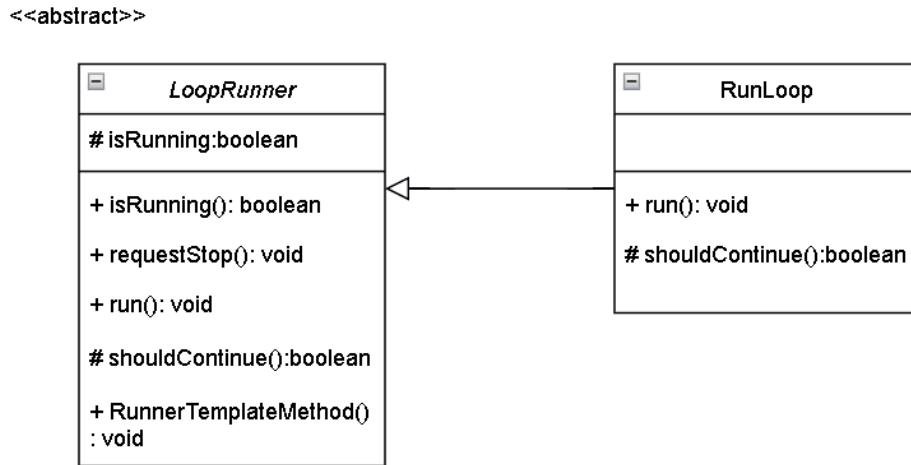


图 16: 模板方法模式在 Slow6502 中的类图

2.13.2 模版方法模式在项目中的应用

在本项目中，界面的事件循环就是基于模版方法模式构建的。在事件循环中，循环的大步骤被`LoopRunner`定义，而`RunLoop`通过继承`LoopRunner`时实现其中的`run()`方法实现在不改变算法架构的情况下，重新定义算法的某些步骤。

2.14 策略模式 (Strategy)

2.14.1 策略模式简介

策略模式是一种设计模式，它定义了一系列的算法，并将每一个算法封装起来，使它们可以相互替换。这样的设计模式使得算法可以在不影响到客户端的情况下发生变化。

策略模式通常被用在以下几种情况：

1. 一个系统需要在多种算法中选择一种来执行。例如，一个图形处理系统可能需要不同的算法来处理不同类型的图形。
2. 一个系统需要在多种算法中动态地选择一种来执行。例如，一个排序系统可能需要根据数据的类型和大小来选择不同的排序算法。
3. 一个系统的算法需要随着时间的推移而发生变化。例如，一个汇率转换系统可能需要根据市场情况来选择不同的汇率计算算法。

总之，策略模式提供了一种方便的方法来封装算法，并使得它们可以相互替换。这样，算法可以在不影响到客户端的情况下发生变化，从而提高了系统的灵活性和可扩展性。

优点：

1. 策略模式提供了一种简单易懂的方法来封装算法。通过定义不同的策略类，可以轻松地在不同的算法之间进行切换。
2. 策略模式提供了可替换算法的功能，使得客户端不需要关心具体的算法是如何实现的。这样，客户端就可以使用不同的算法来解决同一个问题，从而提高了系统的灵活性和可

扩展性。

3. 策略模式可以避免使用多重条件语句，这样可以提高代码的可读性和可维护性。

缺点：

1. 策略模式会增加系统的复杂度，因为它增加了新的策略类和上下文类。
2. 策略模式要求客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这样会增加客户端的复杂度，并且难以维护。
3. 如果一个系统中存在大量的策略类，那么这些类可能会彼此之间重复，这样会降低代码的可重用性。

2.14.2 策略模式在项目中的应用

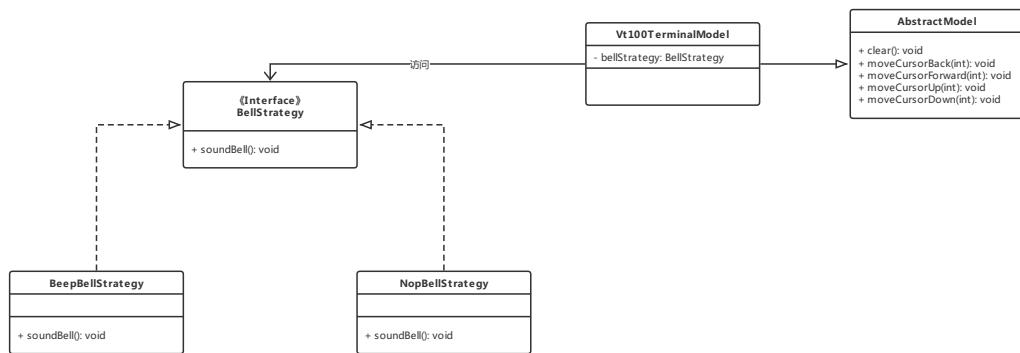


图 17：策略模式在 Slow6502 中的类图

我们的项目中，使用策略模式实现 VT100 终端不同的蜂鸣器的行为。我们将不同的蜂鸣器行为封装起来，使它们可以相互替换，从而使得不同的蜂鸣器行为可以在不影响到调用相关功能的部分的情况下发生变化。

2.15 命令模式 (Command)

2.15.1 命令模式简介

命令模式是一种设计模式，它把请求封装成对象，允许你把请求参数化，并且提供了一种为不同的请求设置不同的撤销和恢复操作的能力。

例如，假设你正在开发一个文本编辑器。在文本编辑器中，用户可以执行操作，例如撤销上一次操作和恢复上一次撤销的操作。为了实现这个功能，你可以使用命令模式，将每个操作封装成一个命令对象，然后把这些对象存储在一个历史记录中。这样，用户就可以撤销和恢复之前的操作了。

另一个例子是电视遥控器。你可以把遥控器的每个按钮看作一个命令，当用户按下按钮时，遥控器就会执行相应的操作。这样，用户可以通过按钮来控制电视，而不用关心具体的操作细节。

总之，命令模式提供了一种将请求封装成对象的方式，使得你可以将请求参数化，并且提供了撤销和恢复操作的能力。

命令模式的优点是：

1. 命令模式将请求封装成对象，使得你可以对请求进行参数化。
2. 命令模式提供了一种撤销和恢复操作的能力。
3. 命令模式使得你可以轻松地实现日志和撤销功能。
4. 命令模式可以把复杂的操作拆分成多个步骤，每个步骤都是一个命令，这样可以方便地实现复杂操作的撤销和恢复。

命令模式的缺点是：

1. 命令模式会增加系统的复杂度，因为它把请求封装成对象，增加了系统中对象的数量。
2. 命令模式需要额外的存储空间来保存命令历史记录，以便支持撤销和恢复操作。
3. 如果命令执行失败，撤销和恢复操作可能会带来一定的困难。

上面这些缺点并不会导致命令模式无法使用，但是在使用命令模式时需要注意这些缺点，并采取适当的措施来应对这些缺点。例如，你可以通过合理地设计系统来减少对象的数量，或者采用更高效的数据结构来存储命令历史记录，以减少存储空间的消耗。

2.15.2 命令模式在项目中的应用

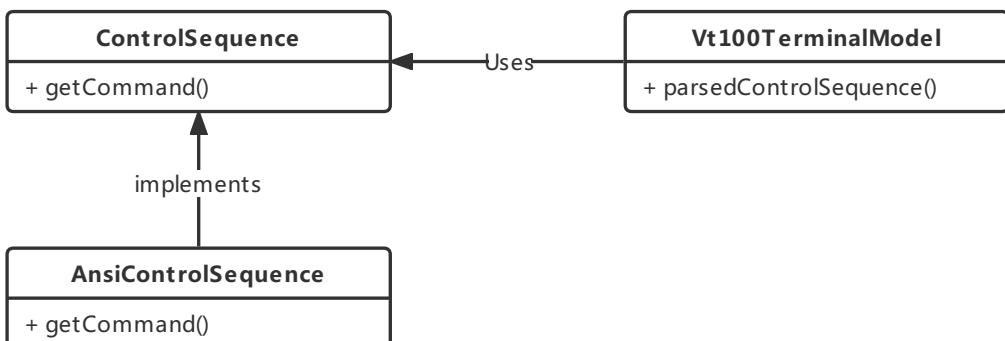


图 18：命令模式在 Slow6502 中的类图

在我们的项目中，我们使用命令模式来抽象终端外设的 Ansi Control Sequence。这样就可以方便地对 ANSI 控制序列进行参数化，从而避免使用硬编码的 ANSI 控制序列。

2.16 责任链模式（Chain of Responsibility）

2.16.1 责任链模式简介

责任链模式是一种软件设计模式，它用于将一个请求的处理分派给多个对象，这些对象形成一条链，并依次处理该请求。

责任链模式的一个常见应用场景是实现一个处理请求的系统。例如，假设我们有一个类，

它可用于处理用户发送的消息。在这种情况下，我们可以使用责任链模式来处理消息，从而让消息能够在多个处理器之间传递，直到被处理为止。

责任链模式具有以下优点：

1. 责任链模式可以实现一种松耦合的设计，使得请求的发送者和处理者之间不需要直接引用来进行通信。
2. 责任链模式可以提高系统的灵活性，因为它允许在不更改对象之间的关系的情况下添加新的处理器。
3. 责任链模式可以使得系统更加模块化，从而更容易实现和维护。

责任链模式也有一些缺点，包括：

1. 责任链模式可能会使得系统变得过于复杂，因为它需要维护请求发送者和处理者之间的关系。
2. 责任链模式可能会导致性能问题，因为它需要在多个对象之间进行额外的通信。
3. 责任链模式可能会导致循环依赖，特别是在请求发送者和处理者之间存在多对多的关系时。

2.16.2 责任链模式在项目中的应用

在我们的项目中，责任链模式被用于处理异常，例如：

```
public Memory(int startAddress, int endAddress) throws  
    ↪MemoryRangeException {  
    this(startAddress, endAddress, false);  
}
```

通过定义 `catch (MemoryRangeException ex)`，完成处理异常的责任的传递。

2.17 状态机模式 (State)

2.17.1 状态机模式简介

状态机模式是一种软件设计模式，它描述了对象在不同状态下的行为，并且能够根据输入来改变对象的状态。这种模式常用于实现系统中复杂的业务逻辑，其中系统的状态可能会因用户的输入或其他因素而改变。

一个简单的例子是一个售货机，它可能有以下几种状态：等待输入、接受输入、处理输入、售出商品、检查库存等。对于每一种状态，售货机都会有不同的行为，例如等待输入时会显示商品列表，接受输入时会接收用户的选择，处理输入时会核对用户输入的金额是否足够等。

在使用状态机模式时，需要定义不同的状态类，每个状态类都包含了一些与该状态相关的行为，并且定义了在进入该状态时执行的操作和在从该状态转移到其他状态时执行的操作。这些状态类可以通过一个状态机的上下文类来组合成一个完整的状态机，上下文类负责维护状态机的状态并通过调用状态类中的方法来实现状态转换。通过使用状态机模式，我们可以将复杂的业务逻辑分解成若干个独立的状态类，使得系统的状态变化和相关的行为变得更加清晰易懂。

在实际应用中，状态机模式可以用于很多不同的场景，例如实现状态转移的动画效果、模拟物理过程中物体的运动状态等。它还可以用于实现复杂的业务逻辑，例如在订单处理系统中根据订单的当前状态和用户的操作来进行订单状态的转换。

状态机模式的优点如下：

1. 可以将复杂的业务逻辑分解成若干个独立的状态类，使得系统的状态变化和相关的行为变得更加清晰易懂。
2. 实现状态转移时，不需要写大量的条件语句，只需要定义好不同状态类和相关的转换规则即可。
3. 可以在运行时动态地改变对象的状态，使得系统更加灵活。
4. 可以使用状态机来模拟复杂的物理过程中物体的运动状态，例如流体动力学模拟中的水流状态转换。

状态机模式的缺点如下：

1. 当系统的状态数量较多时，需要定义较多的状态类和相关的转换规则，系统的复杂度可能会增加。
2. 对于一些需要频繁改变状态的场景，使用状态机模式可能会使系统的性能降低。
3. 状态机模式的实现需要考虑很多细节，如果没有设计好可能会导致系统不稳定。
4. 对于一些没有明确的状态和转换规则的业务逻辑，使用状态机模式可能不是很合适。

2.17.2 状态机模式在项目中的应用

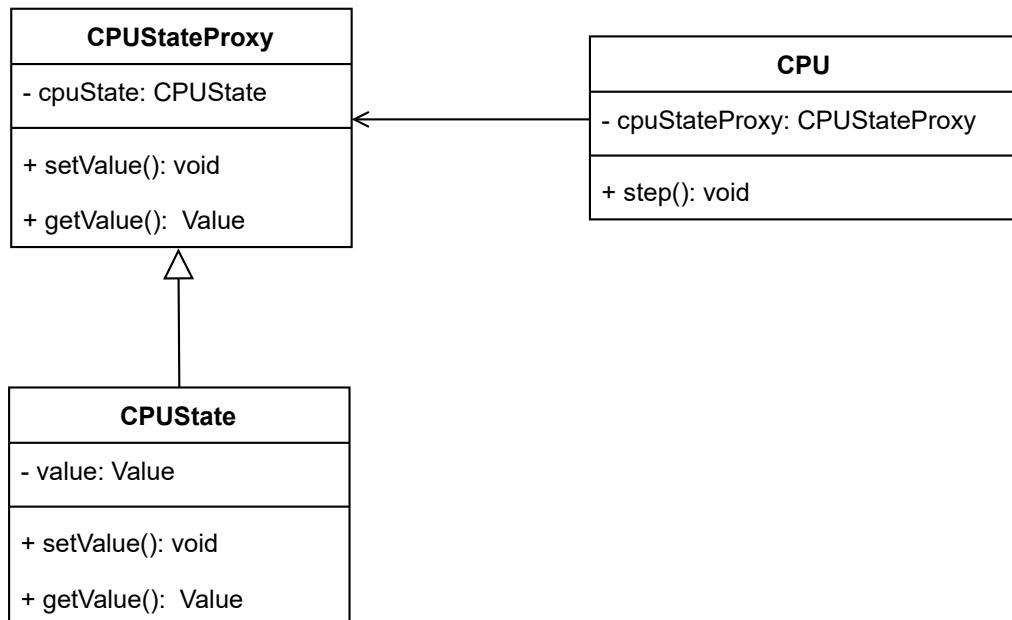


图 19: 状态机模式在 Slow6502 中的类图

我们的 CPU 使用了状态机模式，stateProxy 包含 CPUState 成员，用来表示 CPU 的一些状态（如寄存器的值等）。CPU 在调用 step 函数时，会进入下一个状态，即改变 stateProxy。

使用状态机模式的好处是可以让对象在内部状态改变时改变它的行为，而不需要修改代码。这使得状态机模式非常灵活，可以应对各种复杂的业务场景。

在我们这个例子中，使用状态机模式可以让 CPU 根据不同的状态（例如寄存器的值）来改变它的行为，而不需要在代码中做过多的条件判断。这可以使得 CPU 的实现更加简洁，更容易维护。

2.18 观察者模式 (Observer)

2.18.1 观察者模式简介

观察者模式是一种软件设计模式，它用于实现一种一对多的依赖关系，使得每当一个对象的状态发生改变时，其他依赖于它的对象都能收到通知并自动更新。

观察者模式的一个常见应用场景是实现事件驱动的程序。例如，假设我们有一个用于表示按钮的类，它允许用户点击它，这样它就会触发一个事件。在这种情况下，我们可以使用观察者模式来处理这个事件，从而让其他对象能够监听这个事件并执行相应的操作。

例如，假设我们有一个按钮类，如下所示：

```
class Button {
  constructor() {
    this.observers = [];
  }

  addObserver(observer) {
    this.observers.push(observer);
  }

  onClick() {
    // 触发事件
    for (const observer of this.observers) {
      observer.onClick();
    }
  }
}
```

这个类表示一个按钮，它包含一个用于保存观察者的数组和一个 `addObserver` 方法，用于向数组中添加新的观察者。当按钮被点击时，它会触发一个事件，并通知所有已注册的观察者。这样，观察者就能够监听按钮的点击事件，并执行相应的动作。

观察者模式具有以下优点：

1. 观察者模式可以实现一种松耦合的设计，使得观察者和被观察者之间不需要直接引用来进行通信。
 2. 观察者模式可以提高系统的灵活性，因为它允许在不更改对象之间的关系的情况下添加新的观察者。
 3. 观察者模式可以使得系统更加模块化，从而更容易实现和维护。
1. 观察者模式可能会使得系统变得过于复杂，因为它需要维护被观察者和观察者之间的关系。
 2. 观察者模式可能会导致性能问题，因为它需要在多个对象之间进行额外的通信。

3. 观察者模式可能会导致循环依赖，特别是在观察者和被观察者之间存在多对多的关系时。

2.18.2 观察者模式在项目中的应用

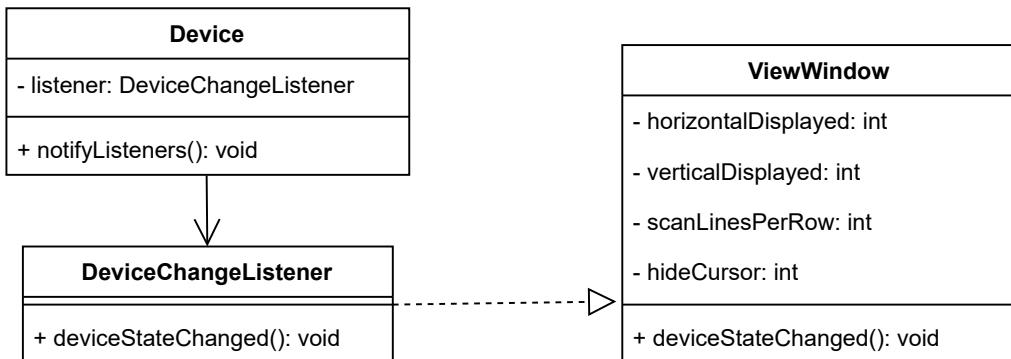


图 20: 观察者模式在 Slow6502 中的类图

我们在 `Device` 类里面使用了观察者模式，它有一个`deviceChangeListeners`数组成员，可以通过`notifyListener`函数以此调用它们的`deviceStateChanged`函数，`ViewWindow`实现了这个函数，会依次修改相应的属性。这样，当`Device`类的状态发生变化时，它就可以通过调用观察者列表中的回调函数来通知观察者。

使用观察者模式的好处在于它可以让主体和观察者之间松散耦合。因为观察者只需要实现特定的回调函数，而主体只需要维护一个观察者列表，所以它们之间的依赖关系很少。这样，主体和观察者都可以独立地变化和扩展，而不会影响对方。

2.19 中介者模式 (Mediator)

模式定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。

2.19.1 中介者模式简介

中介者模式是一种软件设计模式，它用于将对象间的通信进行封装，使各个对象不需要直接互相引用通信。它通过引入一个中介者对象来实现这一目的，该对象负责处理各个对象之间的通信。

中介者模式的一个常见应用场景是在多个对象之间进行通信时，避免对象之间的耦合性。例如，假设我们有两个类，一个用于表示人类，另一个用于表示车辆，它们之间需要进行通信。如果没有中介者模式，我们可能会在两个类中添加大量的代码来处理这种通信，这会导致代码难以理解和维护。

使用中介者模式，我们可以创建一个中介者类，用于处理人类和车辆之间的通信。这样，人类和车辆类就不需要直接引用对方来进行通信了，从而降低了它们之间的耦合度。

中介者模式具有以下优点：

1. 中介者模式可以减少对象之间的耦合度，使各个对象之间不需要直接引用来进行通信。
2. 中介者模式可以提高系统的灵活性，因为它允许在不更改对象之间的关系的情况下添加新的对象。
3. 中介者模式可以使得系统更加模块化，从而更容易实现和维护。

中介者模式也有一些缺点，包括：

1. 中介者模式可能会使得系统变得过于复杂，因为它需要引入新的类来处理对象之间的通信。
2. 中介者模式可能会导致代码难以理解和维护，特别是当有大量的对象时。
3. 中介者模式可能会导致性能问题，因为它需要在多个对象之间进行额外的通信。

2.19.2 中介者模式在项目中的应用

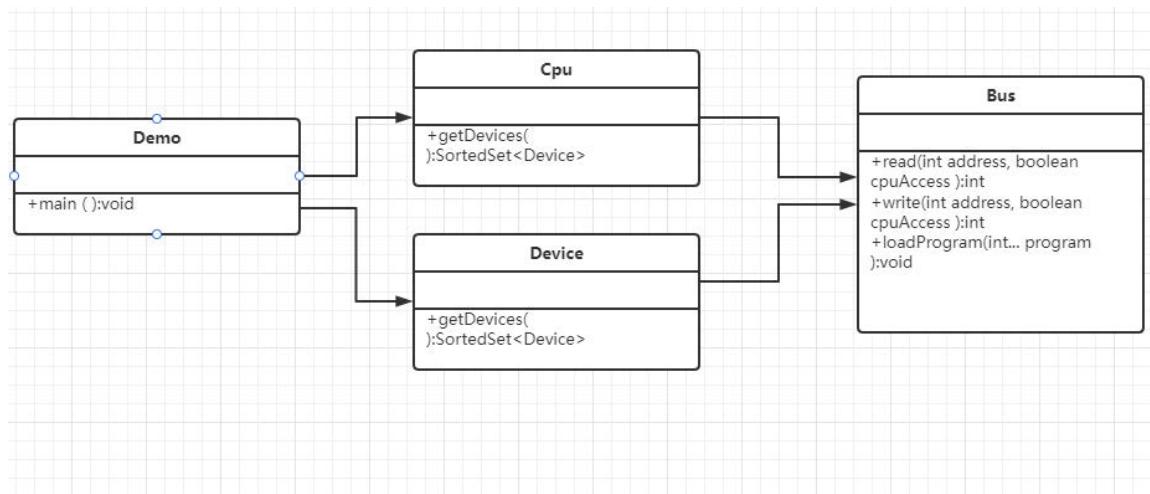


图 21：中介者模式在 Slow6502 中的类图

我们的项目中，Bus 就是作为 CPU、内存和各个外设之间的中介者使用的。如果没有 Bus 作为中介者，模拟器中各个设备对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。通过这种模式，我们使得模拟器能够方便地复用各种外设。

2.20 迭代器模式 (Iterator)

2.20.1 迭代器模式简介

迭代器模式提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

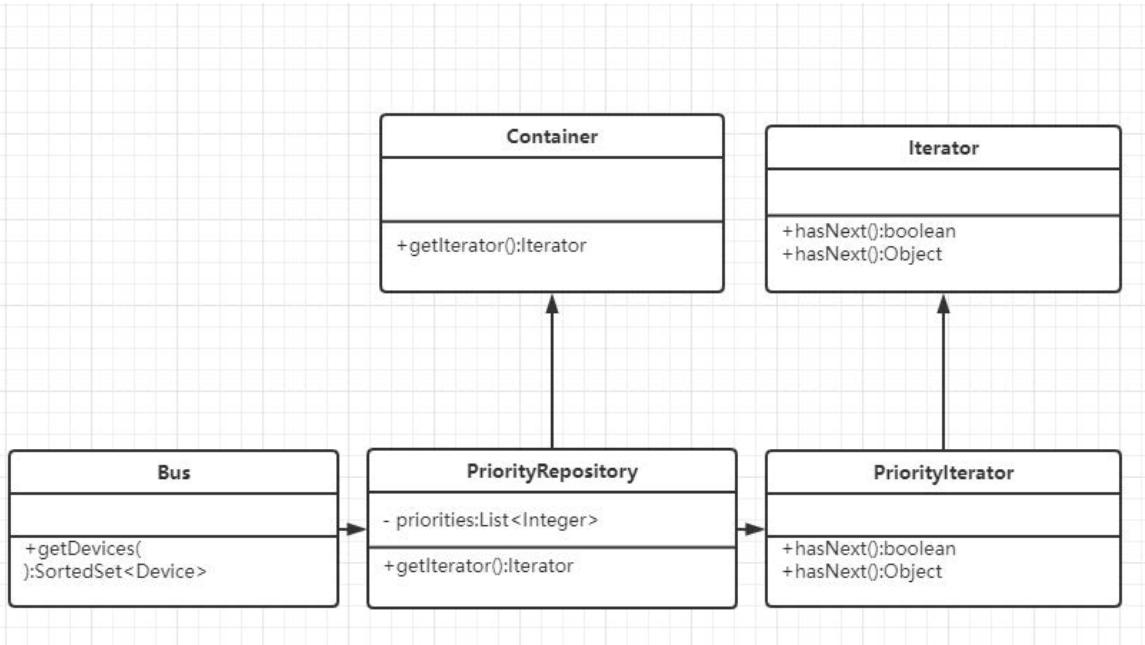


图 22: 迭代器模式在 Slow6502 中的类图

2.20.2 迭代器模式在项目中的应用

使用迭代器可以更方便地遍历容器中的数据。在模拟器中，容器通常包含大量的数据，并且这些数据可能非常复杂。如果想要对这些数据进行遍历，就必须手动编写代码来完成这一过程。这样做不仅耗时，而且容易出错。使用迭代器可以简化遍历容器中的数据的过程。迭代器可以把复杂的遍历过程封装在一个对象中，只需要调用该对象的相关方法就可以完成遍历。这样做可以让您更快速地完成遍历，避免出现错误。此外，使用迭代器还可以让代码更容易理解和维护。迭代器可以把复杂的遍历过程分解成一系列简单的步骤，使代码更加清晰易懂。这样，程序员可以更容易地查看和修改代码，避免出现错误。

总的来说，使用迭代器可以更方便地遍历容器中的数据，并让代码更容易理解和维护。

2.21 访问者模式 (Visitor)

2.21.1 访问者模式简介

访问者模式是一种软件设计模式，它允许在不更改对象结构的情况下添加新的操作。它通过创建一个访问者类来实现这一目的，该访问者类包含用于处理对象结构中的各个元素的方法。

访问者模式的一个常见应用场景是处理对象结构，例如处理树形结构或链表。在这种情况下，访问者模式允许在不更改树形结构或链表的情况下添加新的操作，从而使得程序更加灵活。

例如，假设我们有一个用于表示二叉树的类，如下所示：

```

class BinaryTree {
    constructor(value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

```

```

        this.right = null;
    }
}

```

我们可以创建一个访问者类来处理这个二叉树，如下所示：

```

class TreeVisitor {
    visit(node) {
        // 处理当前节点

        if (node.left) {
            this.visit(node.left);
        }

        if (node.right) {
            this.visit(node.right);
        }
    }
}

```

在这个例子中，访问者类的 `visit` 方法可以用于遍历二叉树的所有节点，并对它们进行处理。我们可以通过更改 `visit` 方法的实现来添加新的操作，而不需要更改二叉树的定义。

访问者模式具有以下优点：

1. 访问者模式允许在不更改对象结构的情况下添加新的操作，从而使得程序更加灵活。
2. 访问者模式将有关的操作和数据结构分离开来，从而更容易实现和维护。
3. 访问者模式可以让程序更加模块化，使得操作和数据结构之间的耦合度降低。
1. 访问者模式可能会导致代码难以理解和维护，特别是当有大量的操作和数据结构时。
2. 访问者模式可能会导致类的数量增加，从而增加系统的复杂度。
3. 访问者模式可能会导致性能问题，因为它需要多次遍历对象结构来实现操作。

2.21.2 访问者模式在项目中的应用

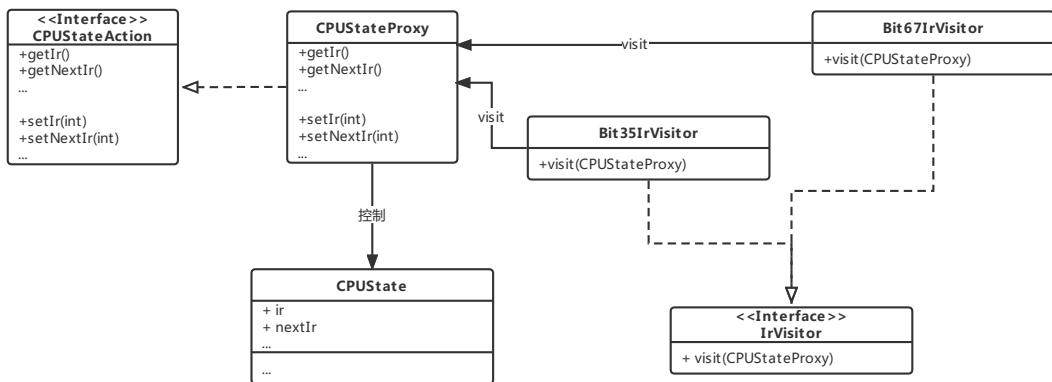


图 23: 访问者模式在 Slow6502 中的类图

在我们的项目中，CPU 对于 IR 寄存器的访问就是基于访问者模式实现的。CPU State 为一个 IR 中的每个元素提供多种访问方式，即 `Bit67IrVisitor` 和 `Bit35IrVisitor`。这样的设

计可以允许在不更改 IR 寄存器结构的情况下添加新的操作，从而使得程序更加灵活。

2.22 备忘录模式 (Memento)

2.22.1 备忘录模式简介

备忘录模式是一种行为型设计模式，它提供了一种方法来保存一个对象的内部状态，并在需要时可以将对象恢复到之前保存的状态。

备忘录模式的实现包括三个部分：

发起人类：发起人类是一个需要保存内部状态的类，它可以创建一个备忘录来保存当前的内部状态，并在需要时通过备忘录恢复先前的内部状态。

备忘录类：备忘录类是一个负责保存发起人类内部状态的类，它可以通过一个对象来保存发起人类的内部状态，并在需要时返回这个对象，以便发起人类恢复先前的状态。

管理者类：管理者类是一个负责管理备忘录的类，它可以保存多个备忘录，并可以提供撤销和恢复的功能，以便发起人类可以方便地撤销和恢复先前的操作。

备忘录模式具有以下优点：

可以保存对象的内部状态，并在需要时将对象恢复到先前保存的状态，从而实现对撤销和恢复操作的支持。

可以将备忘录作为一个独立的对象进行管理，从而避免发起人类过于臃肿，并提高了系统的可维护性和可扩展性。

可以通过管理者类来管理备忘录，从而提高了系统的可控性。

同时，备忘录模式也有一些缺点：

备忘录对象需要占用一定的内存空间，因此在某些情况下可能会导致内存开销增大。

备忘录模式需要为每个需要支持撤销和恢复操作的类定义一个与其对应的备忘录类。

2.22.2 备忘录模式在项目中的应用

项目中使用备忘录模式保存 CPU 的状态，这允许在某个时刻将 CPU 的状态保存下来，然后在需要时可以恢复到保存的状态。这对于撤销操作或者在 CPU 发生故障时进行故障恢复都非常有用。这样的设计有以下优点：

1. 保存内部状态的方式简单明了，可以直接保存内部状态并在需要时恢复，无需定义复杂的撤销和恢复逻辑。
2. 不会影响 CPU 的其他功能，可以在不改变 CPU 的其他功能的情况下实现对内部状态的重置。
3. 可以实现多次重置，备忘录对象可以被保存下来，从而可以实现多次重置。
4. 可以方便地实现撤销和恢复功能，通过管理者类维护备忘录，可以方便地实现撤销和恢复功能。

备忘录模式通过保存内部状态并在需要时恢复来实现对内部状态的重置，并具有良好的灵活性、扩展性和可维护性。

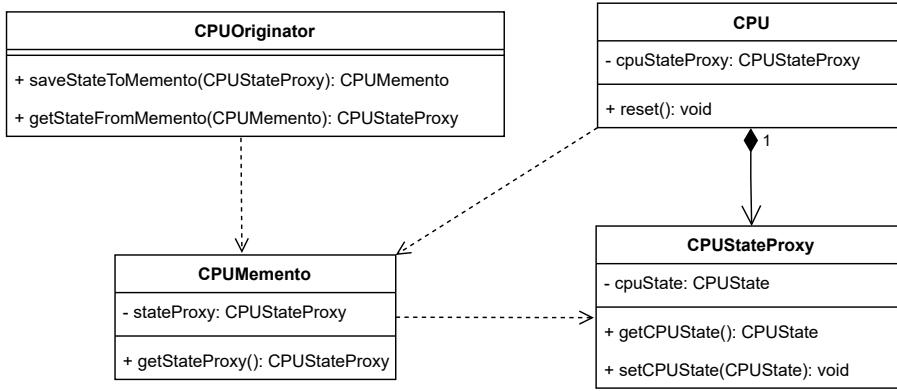


图 24: 备忘录模式在 Slow6502 中的类图

2.23 解释器模式 (Interpreter)

解释器模式是一种行为型设计模式，它提供了一种方法，通过定义一组语言文法规则并使用这些规则来解释语言中的句子，从而可以实现对语言的解释执行。

解释器模式的实现通常包括以下几个部分：

1. 定义语言文法规则：首先，我们需要根据需求定义语言的文法规则，并将其表示为一组符号、终结符和非终结符的集合。
2. 建立抽象语法树：接下来，我们需要使用定义的文法规则来解析语言中的句子，并将其表示为一棵抽象语法树。
3. 定义解释器：最后，我们需要定义一个解释器，它可以接收一棵抽象语法树作为输入，并根据语言的文法规则来执行对应的操作。

解释器模式的实现需要结合文法规则和语言知识，需要比较细致的设计和实现。使用解释器模式可以有以下几个优点：

1. 可以将语言文法规则与语言解释器的实现分离，从而更加灵活地支持对语言的扩展和修改。
2. 可以通过构建抽象语法树来实现对语言的解释执行，从而使得语言的解释更加高效。
3. 可以在不改变原有类的情况下为语言的解释器添加额外的功能，例如语法检查、性能监控等。
4. 可以通过定义统一的抽象语法树来统一语言解释器的实现，从而更加方便地实现多语言支持。

2.23.1 解释器模式在项目中的应用

在我们的项目中，解释器模式主要用于将总线上流动的数据解析为终端外设的 Ansi Control Sequence，解析后的 Ansi Control Sequence 为一串命令，供 VT100 终端使用。这里解释器的语言文法规则较为简洁，即 Ansi Control Sequence 的文法。

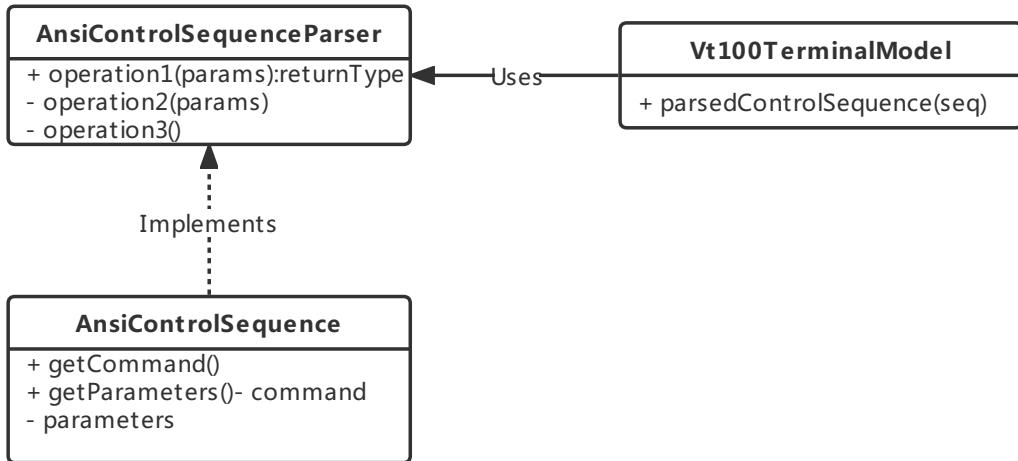


图 25: 解释器模式在 Slow6502 中的类图

3 设计模式详述（非 GoF 模式）

3.1 过滤器模式 (Filter)

过滤器模式 (Filter Pattern) 或标准模式 (Criteria Pattern) 是一种设计模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准。

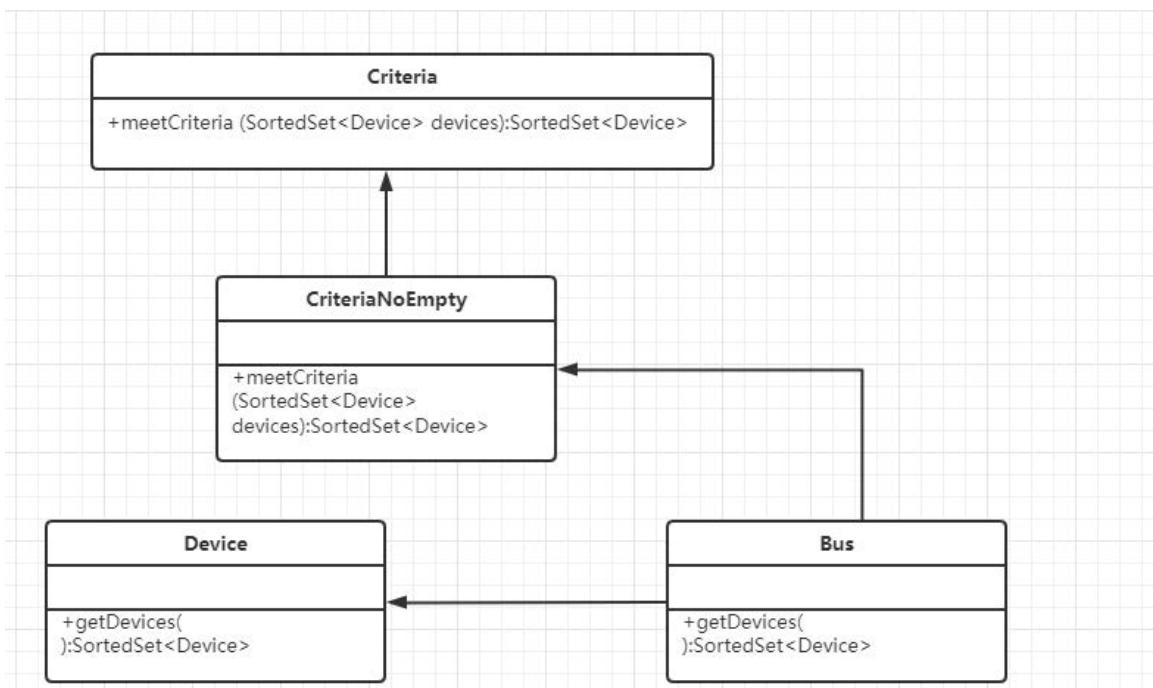


图 26: 过滤器模式在 Slow6502 中的类图

使用过滤器可以有效地提高模拟器的性能。在模拟器中，通常需要处理大量的数据，并且这些数据可能非常复杂。如果模拟器的代码不进行优化，就可能会导致性能下降，影响模

拟器的效率。使用过滤器可以提高模拟器的性能，因为它可以帮助模拟器更快地处理数据。过滤器可以将大量的数据缩小到更小的范围，使模拟器只需处理有用的数据，而不是所有的数据。这样做可以让模拟器更快地运行，提高效率。此外，使用过滤器还可以让模拟器的代码更容易理解和维护。过滤器可以把复杂的数据处理过程分解成一系列简单的步骤，使代码更加清晰易懂。这样，程序员可以更容易地查看和修改代码，避免出现错误。

总的来说，使用过滤器可以提高模拟器的性能，让代码更容易理解维护。

3.2 MVC 模式

MVC 模式 (Model-View-Controller pattern) 是一种用于分离应用程序的数据模型 (Model)、用户界面 (View) 和业务逻辑 (Controller) 的设计模式。

在 MVC 模式中，Model 代表应用程序的数据模型，它负责处理与数据相关的业务逻辑，例如对数据进行验证、查询和修改等。View 代表用户界面，它负责显示数据和与用户交互，例如显示表单和按钮等。Controller 负责接收用户的请求，并调用 Model 和 View 来完成相应的业务逻辑和显示数据。

使用 MVC 模式的主要目的是将应用程序的业务逻辑和用户界面进行分离，使得两者之间的耦合度降到最低。这样，如果需要更改用户界面或业务逻辑，只需要修改相应的 View 或 Controller 即可，而不需要修改整个应用程序。

MVC 模式通常用于构建大型的、复杂的应用程序，例如企业应用程序或 Web 应用程序。它可以帮助开发人员更轻松地处理应用程序的业务逻辑和用户界面，并使应用程序更容易维护和扩展。

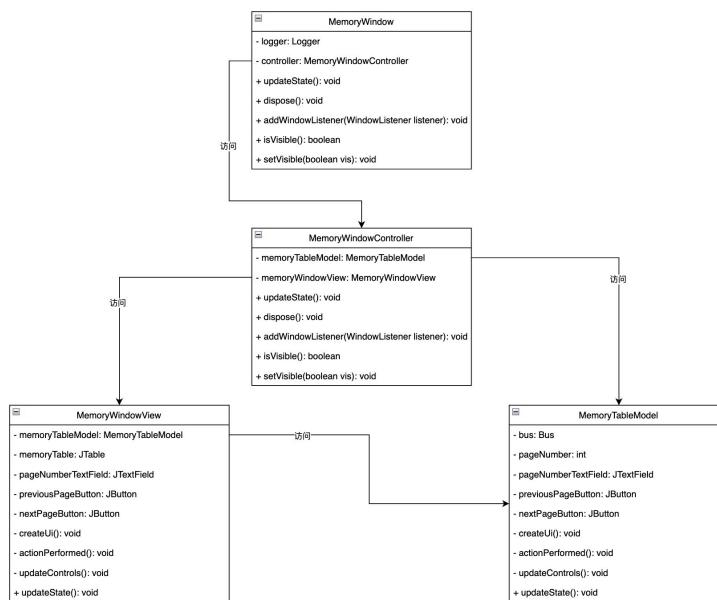


图 27: MVC 模式在 Slow6502 中的类图

在 MemoryWindow 页面使用 MVC 模式，使得逻辑控制、组件 UI 实现、数据模型处理逻辑分离，能够使得前端整体架构更加解耦。控制器的职责是接收用户的输入并将其转发给模型，然后根据模型的响应更新视图。

3.3 MVP 模式

MVP 模式 (Model-View-Presenter pattern) 是一种用于分离应用程序的数据模型 (Model)、用户界面 (View) 和业务逻辑 (Presenter) 的设计模式。

在 MVP 模式中, Model 代表应用程序的数据模型, 它负责处理与数据相关的业务逻辑, 例如对数据进行验证、查询和修改等。View 代表用户界面, 它负责显示数据和与用户交互, 例如显示表单和按钮等。Presenter 负责接收用户的请求, 并调用 Model 和 View 来完成相应的业务逻辑和显示数据。

与 MVC 模式相比, MVP 模式的一个主要区别在于 View 与 Model 之间的交互方式。在 MVC 模式中, View 通常会直接操作 Model, 而在 MVP 模式中, View 通常不会直接操作 Model, 而是通过 Presenter 来操作 Model。这样, Presenter 就成为了 View 与 Model 之间的桥梁, 使得 View 与 Model 之间的耦合度降到最低。

MVP 模式通常用于构建大型的, 复杂的应用程序, 例如企业应用程序或 Web 应用程序。它可以帮助开发人员更轻松地处理应用程序的业务逻辑和用户界面, 并使应用程序更容易维护和扩展。

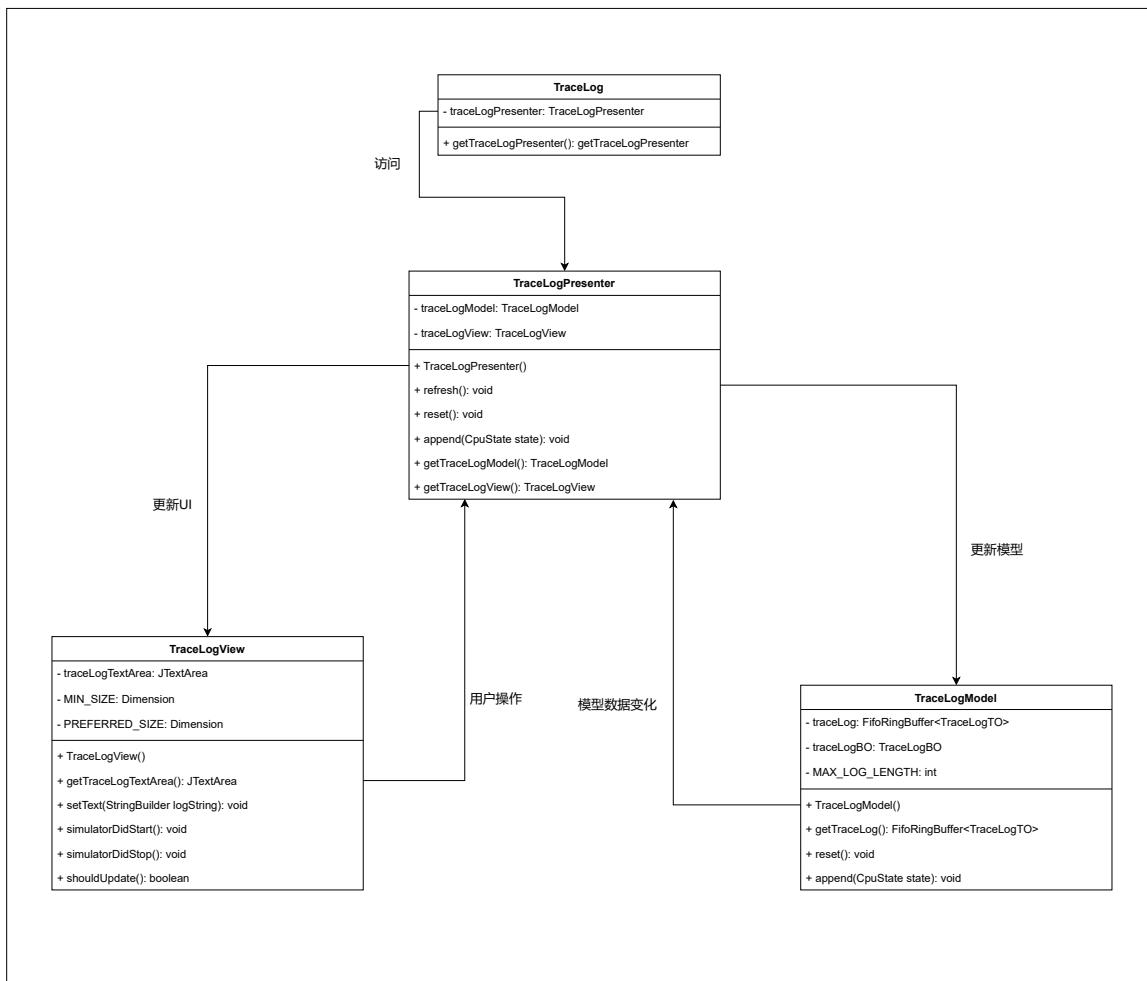


图 28: MVP 模式在 Slow6502 中的类图

对于 traceLog 的前端功能, 通过使用 MVP 模式, 分离了 traceLog 视图展示的逻辑和 traceLog 记录与获取的业务逻辑。TraceLogModel 类和 TraceLogView 类是不直接交流的, 他们

通过 TraceLogPresenter 沟通, TraceLogPresenter 类负责整合 TraceLogModel 类和 TraceLogView 类, 为系统提供 traceLog 展示的功能。

3.4 数据访问对象模式

数据访问对象模式 (Data Access Object pattern) 是一种用于抽象数据访问接口的设计模式。它提供了一种方法来访问数据存储, 例如关系数据库, 而无需关心底层的数据存储技术。

使用数据访问对象模式的主要目的是隔离数据存储和使用层, 使两者之间的耦合度降到最低。这样, 如果需要更换数据存储技术, 只需要修改数据访问对象接口的实现即可, 而不需要修改使用数据的代码。

数据访问对象模式通常用于构建大型的, 分层的应用程序, 例如企业应用程序。在这种应用程序中, 数据访问对象模式常常用于把业务逻辑和数据存储分离开来, 以便二者可以独立地进行更改和扩展。

总的来说, 数据访问对象模式是一种用于提高应用程序的灵活性和可扩展性的设计模式。它可以帮助开发人员更轻松地处理数据存储技术的变化, 并使应用程序更容易维护和扩展。

在本项目中, 使用数据访问对象模式将单条的 traceLog 信息进行包装形成 TraceLogMO 类。设计 TraceLogDao 接口并配备 TraceLogDaoImpl 类实现该接口, 该类作为数据访问实体类来进行数据访问工作, 并在之后供数据传输模块调用。

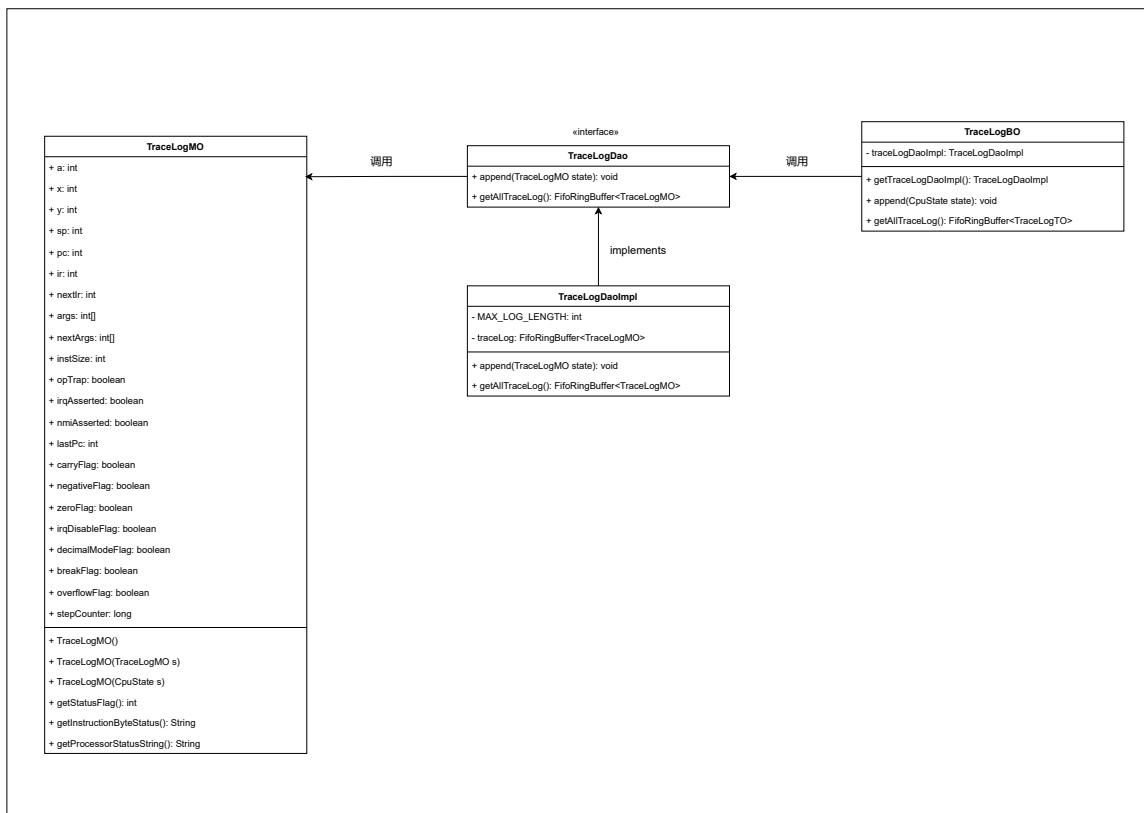


图 29: 数据访问对象模式在 Slow6502 中的类图

3.5 传输对象模式

传输对象模式（Transfer Object Pattern）是一种用于从客户端向服务器一次性传递大量数据的轻量级模式。它通过将数据封装在一个对象中，并将这个对象用作客户端和服务器之间的数据传输方式，来简化客户端和服务器之间的通信。

传输对象模式常用于分布式系统中，它可以帮助减少系统中各个部分之间的通信复杂度，提高系统的性能。例如，一个分布式电商系统中，客户端可能需要检索大量的商品信息，如果每次检索都需要将所有信息一次性传送给客户端，这将会极大地降低系统的性能。使用传输对象模式，可以将所有商品信息封装在一个传输对象中，并由服务器端一次性传送给客户端，从而避免了频繁的网络通信，提高了系统的性能。

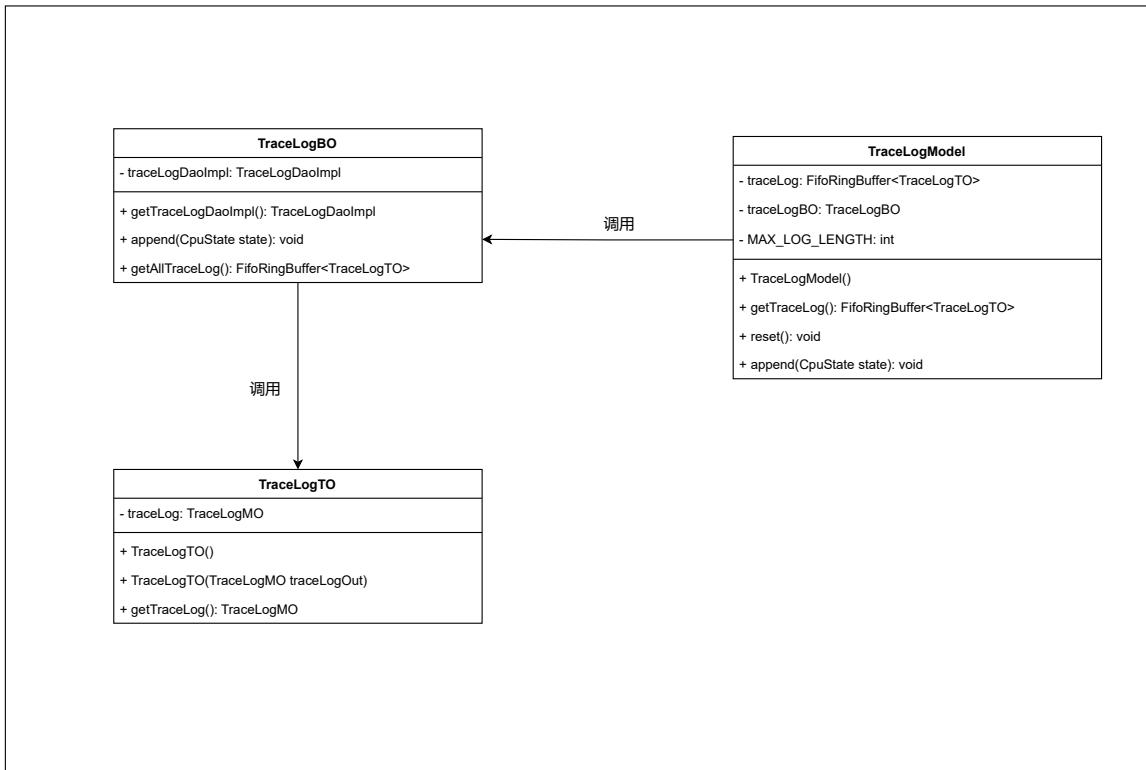


图 30: 传输对象模式在 Slow6502 中的类图

在这里我们设计了 TraceLogTO 类作为要传输的业务对象，其内部包含了 TraceLogMO 类的数据。设计 TraceLogBO 类作为业务对象类，用于向客户端（即 MVP 模式中的 TraceLogModel 类）传输数据，以及从客户端获取新添加的 traceLog 信息。

3.6 空对象模式

空对象模式（Null Object pattern）是一种设计模式，它的主要目的是在处理空值的情况下，避免出现空指针异常。

在空对象模式中，当遇到空值时，会返回一个特殊的空对象，这个空对象实现了与原对象相同的接口，但是它不会执行任何实际的操作。这样，程序在调用空对象的方法时，不会抛出空指针异常，而是会返回默认的结果。

使用空对象模式的主要目的是提高程序的健壮性和可维护性。它可以帮助开发人员避免在处理空值时出现空指针异常，从而使程序更容易维护和扩展。

空对象模式通常用于处理复杂的数据结构，例如树形结构或链表结构。在这些数据结构中，空值是很常见的，如果不使用空对象模式处理这些空值，程序很容易出现空指针异常。

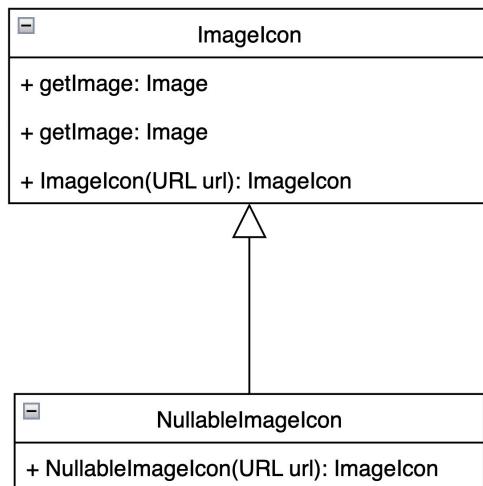


图 31：空对象模式在 Slow6502 中的类图

在 StatusPanel 组件中使用空对象模式，使得即使在没有资源文件图片存在的情况下，也可以正常显示图片组件，以默认图片作为空对象。

3.7 业务代表模式

业务代表模式是一种软件设计模式，它的目的是通过抽象化和封装来减少系统中对象之间的依赖。业务代表模式通过引入一个中介对象（称为业务代表）来处理复杂的业务逻辑，从而减少对象之间的直接依赖关系。

业务代表模式通常用于分离复杂的业务逻辑，使系统中的对象可以更简单地进行单元测试。例如，如果您有一个复杂的业务对象，它与数据库进行交互，并且需要进行大量的计算来完成其工作，那么您可以使用业务代表模式来抽象出这些复杂的操作，从而使您的业务对象变得更简单，更容易测试。

业务代表模式还可以用于分离系统中不同模块之间的依赖关系。例如，如果您有一个业务对象，它依赖于另一个模块来执行某些操作，那么您可以使用业务代表模式来分离这些操作，使得您的业务对象不再依赖于其他模块。这样一来，您就可以更容易地对每个模块进行单独的测试和维护。

业务代表模式的优点如下：

1. 业务代表模式可以将复杂的业务逻辑封装起来，从而使得业务逻辑更加清晰、易于维护。
2. 业务代表模式可以有效地隔离应用程序的用户界面和业务逻辑，从而使得应用程序更加松耦合，更容易实现模块化。

3. 业务代表模式提供了一种统一的访问入口，使得用户可以更方便地访问业务逻辑。
4. 业务代表模式可以实现业务逻辑的灵活扩展，从而使得应用程序能够适应不断变化的需求。

业务代表模式的缺点如下：

1. 业务代表模式可能导致系统变得过于复杂，从而增加系统维护的难度。
2. 业务代表模式的实现可能会增加系统的开发和维护成本。
3. 业务代表模式可能会导致系统的性能下降，因为需要进行大量的数据传递和处理。
4. 业务代表模式的实现可能会带来安全隐患，因为业务逻辑的封装可能会导致安全漏洞难以发现和修复。

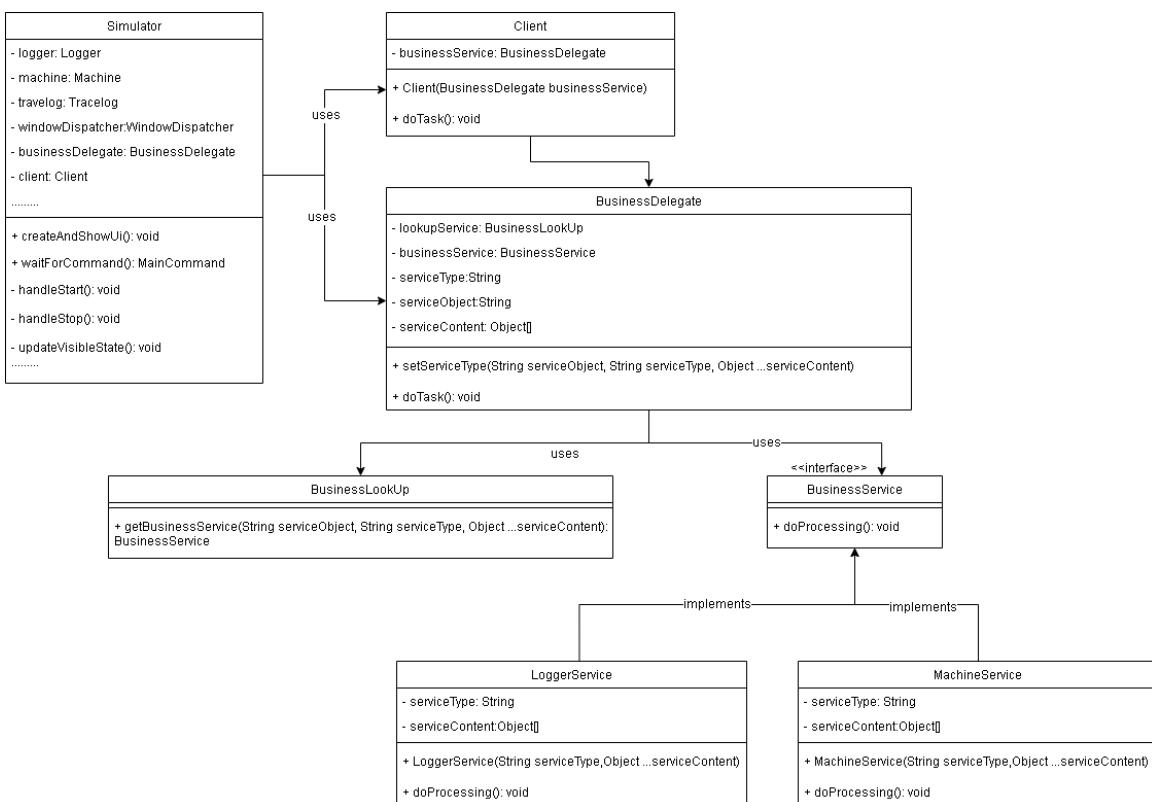


图 32: 业务代表模式在 Slow6502 中的类图

3.8 前端控制器模式

前端控制器模式是一种设计模式，它用于处理应用程序中的用户请求。它是一种中央控制器，它能够处理所有请求并将它们分发给相应的处理器。

前端控制器模式的应用场景主要是在 Web 应用程序中，它能够将用户的请求转发到相应的控制器处理，并返回处理结果给用户。

前端控制器模式的优点是能够提高应用程序的灵活性和可扩展性，并且能够有效地解决请求分发问题。缺点是实现起来比较复杂，需要维护多个控制器和处理器，并且需要花费更多的时间和精力进行设计和实现。

总的来说，前端控制器模式是一种非常有用的设计模式，它能够有效地处理用户请求并将其分发到相应的处理器，提高应用程序的灵活性和可扩展性。

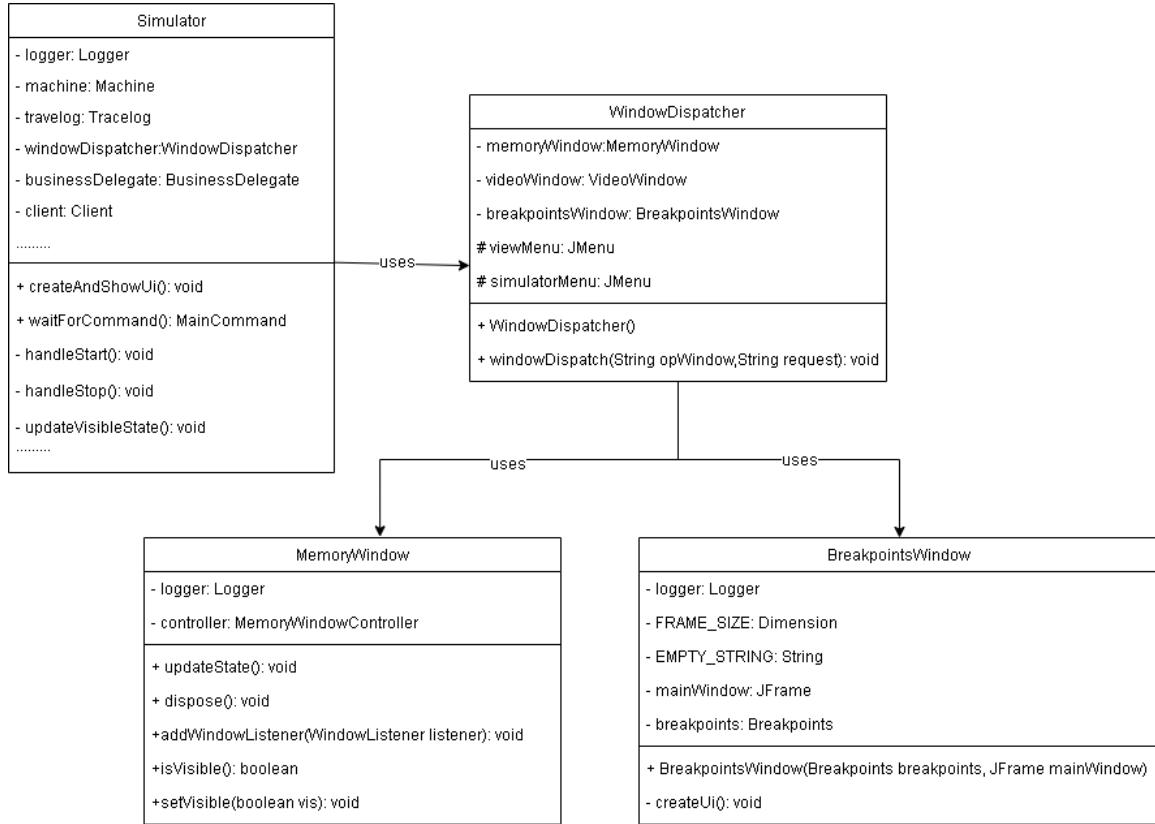


图 33: 前端控制器模式在 Slow6502 中的类图

4 项目总结

4.1 硬件模拟器开发的难点

在本小组的 Slow6502 项目中，我们实现了理论课上讲到的所有 23 个 GoF 设计模式，以及 8 个不属于 GoF 的设计模式。这些设计模式使用并非生搬硬套，而是符合项目的实际情况的。相比较常规的业务软件，构建一个硬件平台的指令级模拟软件，主要面临以下几个难点：

1. 硬件模拟器的行为十分复杂，且必须进行精准的复刻
2. 硬件平台结构和组成十分复杂，且在各类文档中的抽象层次不一致
3. 用户对于硬件模拟器的灵活性要求较高，需要能够模拟不同型号的处理机变种、内存、外设等
4. 由于面向的用户大多为专业用户，故而硬件模拟器的源码需要有良好的可扩展潜力，以方便用户进行自行修改
5. 硬件模拟器需要一定的执行效率

目前最广泛使用的 QEMU 为了执行效率和模拟的精准性，而牺牲了源码的可维护性。考虑到目前的常见设备的计算能力通常是 MOS 6502 的数万倍，故而模拟器的性能并不构成问题（这也就是我们项目中 Slow 一词）的由来。为了保证模拟硬件行为的正确性，并保证代码的可维护性，我买需要对硬件设备以及其运行过程的各个层次进行抽象，同时使用设计模式使得代码能够被更加灵活地复用和修改。

4.2 设计模式的意义

这里以我们项目中的寄存器为例，总结一下设计模式在这个项目中的关键作用：

1. 保障行为的准确实现
2. 方便测试和验证
3. 减少变更和重构的工作量

保障行为的准确实现 虽然 6502 只有 6 个寄存器（FLAG, SP, PC, A, X, Y），但这 6 个寄存器在行为上各有特点，但又有着千丝万缕的联系。传统的硬件模拟器中，通常使用结构体直接保存寄存器的内容，然后通过一系列冗长复杂的控制流（例如超过 3 层的 if-else 语句，大块嵌套的 switch-case）等，来实现这些相似但又不完全相同的功能。当然，如果有足够的时间和精力，这些控制流是可以被正确实现的。但在当今的开发节奏下（例如在一周之内完成一轮敏捷开发），这样编写逻辑的方式是行不通的。

因而，我们需要使用设计模式引入合适的抽象，从而将“揉成一团”的逻辑拆分成小块，然后逐一进行实现。我们的代码中，除了极少部分的特殊情况，几乎没有出现超过 100 行的代码块，或者超过 3 层的 if 语句。合理的顶层设计配合语言的特性，加上设计模式的指导，使得准确实现预定的逻辑变得较为容易。

方便测试和验证 小块封装好的代码除了可以减轻开发人员的心智负担外，也可以方便对于程序的质量控制。

一个显而易见的优点是单元测试变得十分简单。测试一个如 `getNegativeFlag` 的简单方法，要远比测试一堆不知所云的 if-else 并试图找出哪里出现了问题要简单的多。

除了单元测试外，集成测试也能从设计模式中受益。在这些小块的方法中，我们可以对逻辑中的不变量（invariant）进行断言，这样在集成测试时，出现的问题可以迅速通过失败的断言被找到。

减少变更和重构的工作量 一方面，通过一系列合理设计的设计模式构建的源码，本身就为变更和重构提供了丰富的工具。诸如 `getNegativeFlag()` 等方法，使得我们重构时可以不必修改大量的实现细节。

另一方面，即便是对于 `getNegativeFlag()` 等方法本身的修改，也不必担心对于其他源码的影响。一个我们亲身经历上的例子是在一次大规模的重构后，各个小队依次直接进行了 merge，没有出现一个冲突，并且成功通过了所有的回归测试。