4 GoF 设计模式应用-结构性模型

结构化模式设计关注如何将类和对象进行组合以形成更大的结构。结构化类模式使用继承的方式组成接口或实现。多重继承将两个或多个类混合到一个类中的机制是这类实现的一个典型例子。所得到的是一个结合了父类属性的新类。此模式能使独立开发的类库间能够更加高效地协同工作。

结构化对象模式描述的不是组合接口或实现，而是组合对象以实现新功能的方法。对象组合所增加的灵活性来自于在运行时更改组合关系的能力。然而，静态类组合不可能实现这个能力，从而体现了结构化对象模式的强大之处。

4.1 适配器（Adaptor）

4.1.1 简介

适配器设计模式可以将一个类的接口转换为客户期望的另一个接口。适配器设计模式能够使一些本来由于不兼容的接口而无法协调工作的类正常地共同运作。

适配器设计模式的一般使用场景如下：

* 开发者希望使用现有的类，但它的接口与他/她需要的接口不匹配。
* 开发者希望创建一个可重用的类，该类有能力与不相关或不可预见的类(即不一定具有兼容接口的类)协作。
* （仅针对对象适配器）开发者需要使用几个现有的子类，但是通过子类化每个类来调整它们的接口是不现实的。对象适配器可以适应其父类的接口，降低开发工程量。

类适配器使用多重继承使一个接口与另一个接口相匹配。

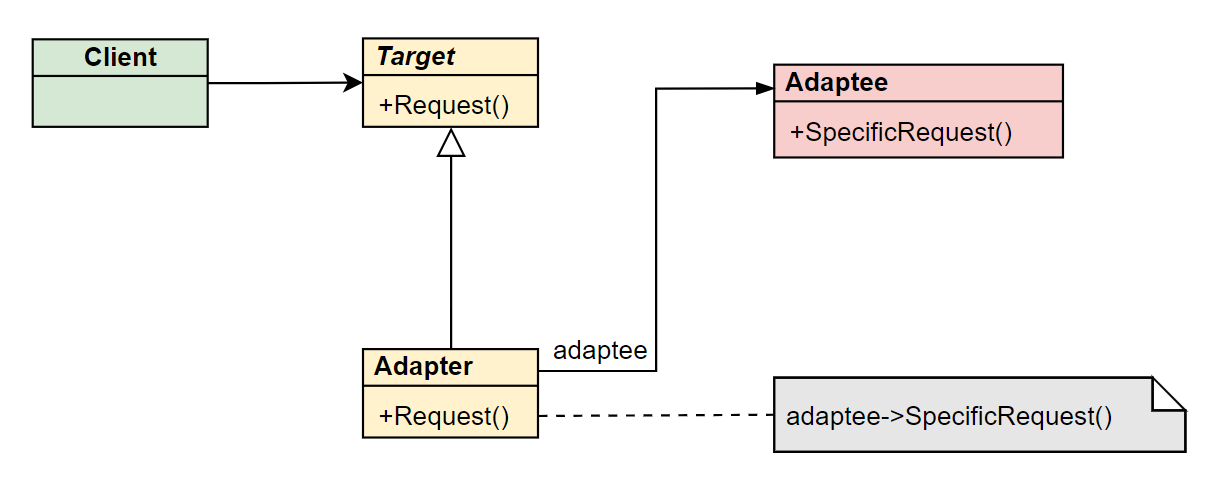


图 1：适配器设计模式结构图

从图1可以看到一共有4个参与者，它们的具体功能如下：

* Target —— 定义了Client类所使用的特定领域接口；
* Client —— 与符合Target类接口的对象进行协作；
* Adaptee —— 定义需要修改的现有接口；
* Adapter —— 将Adaptee类的接口适配到Target类接口。

4.1.2 项目示例

当构造项目所需的类和方法时，开发者最初定义的接口可能会随着项目规模的增长而发生变动，导致新的接口与旧的接口不兼容。此时，本节提出的适配器设计模式适用于解决这一情况。

再次使用图1做说明。假如当前项目已经实现了一个类Adapatee，现在开发者希望通过Target类调用Adaptee类，由于Target类和Adaptee类的接口不兼容，因此开发者可以构建出一个新的Adapter类，以多重继承的方式实现两个不同接口间的相互协作。

4.1.3 优缺点分析

由于类以及对象适配器各有取舍,下面将对它们做分别说明：

1. **类适配器**

优点：由于类适配器通过多重继承实现，因此类适配器能够重定义父类的一些行为；只引入一个对象，不需要通过额外的指针进行间接访问就可以到达adaptee。

缺点：正如图1所示，由于Adaptee接口与Target接口的兼容是通过实现一个具体的Adapter类来完成的。因此，当开发者想要修改一个类及其所有子类时，类适配器将无法工作。

**（b） 对象适配器**

**优点：**可以使一个适配器与多个Adaptee一起工作——即Adaptee本身和它的所有子类(如果有的话)。适配器还可以一次为所有的Adaptee添加功能。

**缺点：**难以覆盖Adaptee的行为。它需要将Adaptee子类化，并让Adapter引用这个子类而不是Adaptee本身。

4.2 桥接（Bridge）

4.2.1 简介

桥接最大的作用在于将抽象与其实现解耦，以便两者能够独立变化。当一个抽象可以有几种不同的实现时，通常适配它们的方法是使用继承。抽象类定义了其抽象的接口，具体的子类能通过不同的方式实现它。但这种方法并不总是能保持灵活性。由于继承会将接口实现永久地绑定到抽象上，这样的机制使得独立地修改、扩展和重用抽象和相应实现变得更加困难。

桥接设计模式的使用场景如下：

* 开发者希望避免抽象与其实现之间做永久绑定；
* 开发者希望抽象及其实现都应该通过子类化进行扩展；
* 抽象实现中的更改不应对客户端产生影响，即客户端的代码无需重新编译；
* 当使用C++开发项目时，开发者希望对客户端完全隐藏抽象的实现；
* 开发者希望在多个对象之间共享同一个实现（可能使用引用计数），而对于用户来说，他们无需了解这一点。

图2显示了桥接设计模式的结构图。

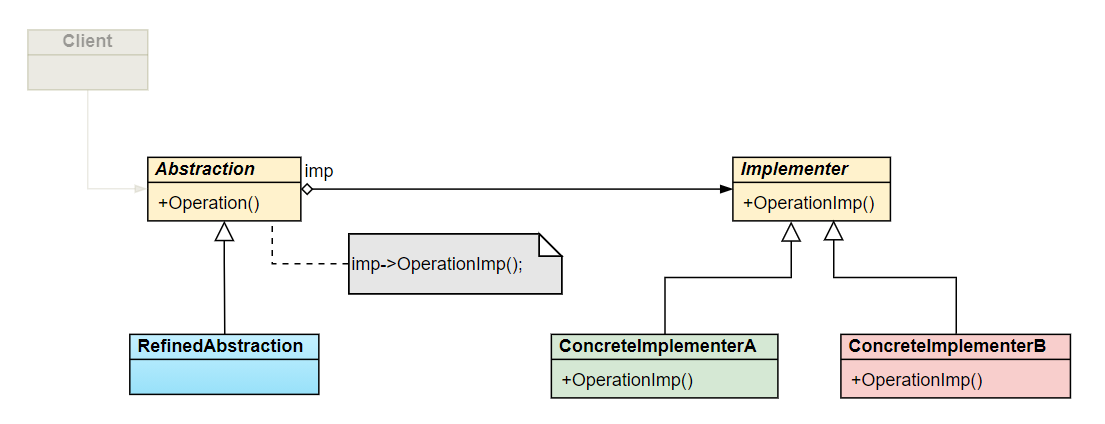


图 2：桥接设计模式结构图

从此图可以看到总共有4个参与者，具体如下：

* Abstraction —— 定义了抽象类的接口，并维护Implementer类对象实例的引用；
* RefinedAbstraction —— 扩展Abstraction类的接口；
* Implementer —— 定义了实现类的接口。这个接口不必与抽象的接口完全对应。一般来说，这两个接口之间可以有较大的差异。通常，实现者接口只提供基本操作，而抽象则基于这些基本操作定义一些更高级别的操作；
* ConcreteImplementor —— 实现Implementer类的接口并定义其自身接口的实现。

4.2.2 项目示例

本项目中客户端app将收集传感器信息，并对收集到的信息进行分类。如图3所示，数据一共有心率、温度、加速度以及报告四种类别，而每种类别各有一个操纵对应类别数据的方法Manipulate。显然，这样的设计耦合度较高，同时继承体系亦较为复杂。因此，对此设计采用桥接设计模式，是一个较为理想的解决方案。

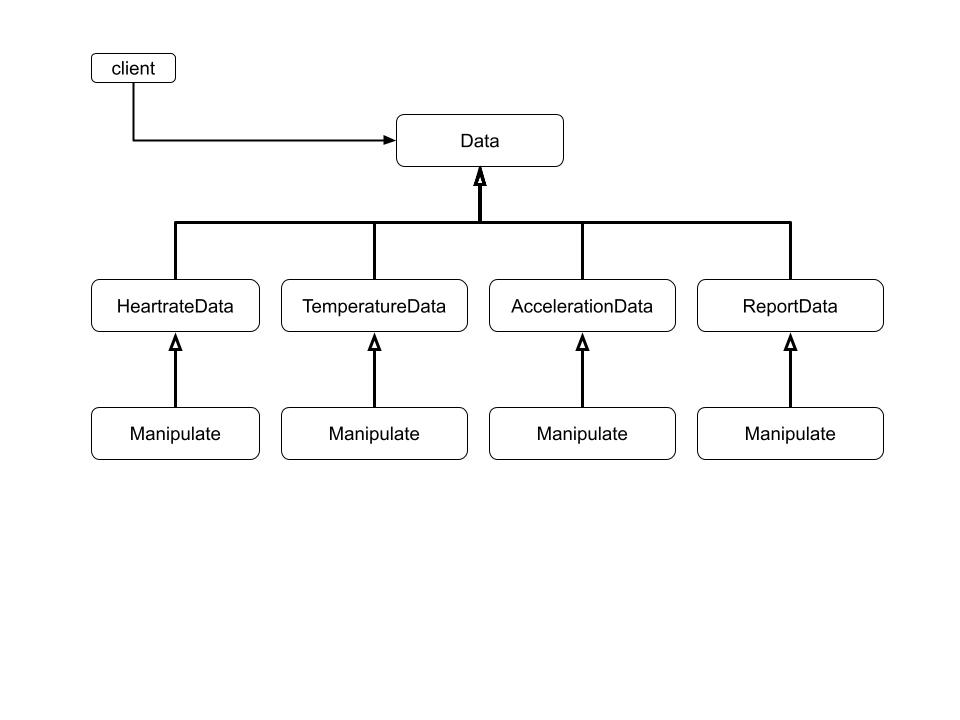


图 3：未使用桥接设计模式的项目模块

如图4所示，当采用桥接设计模式后，操纵数据的方法Manipulate成为了Manipulator类的成员方法，而Manipulator类又为Data类定义了实现的接口。这样，项目的耦合程度便有所降低。同时，亦增加了项目的可读性，提高开发者的编码效率。

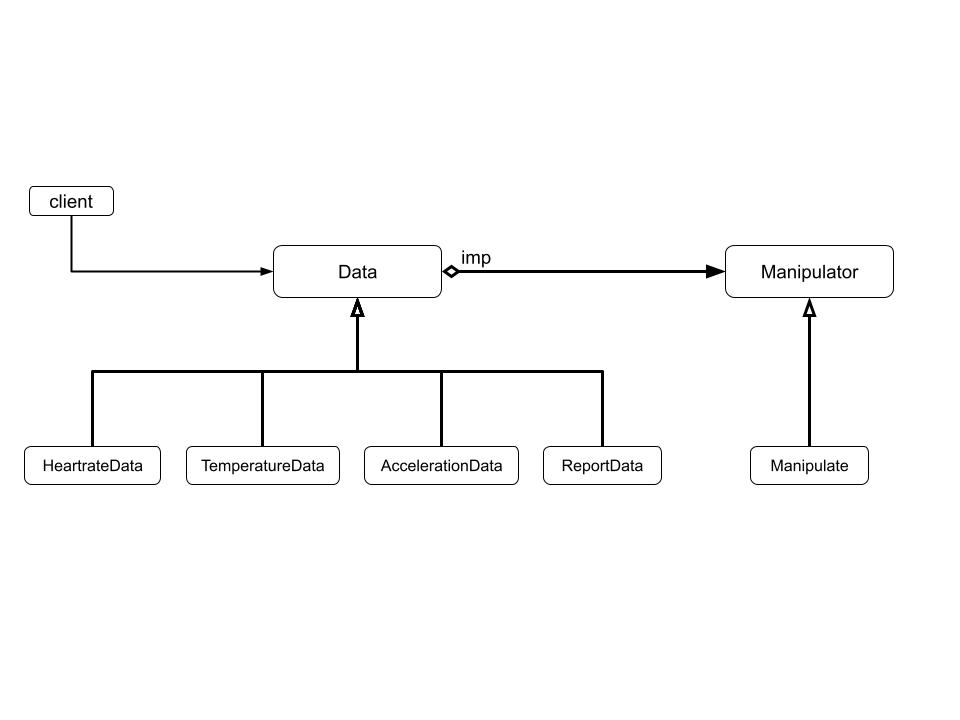


图 4：使用桥接设计模式的项目模块

4.2.3 优缺点分析

现对桥接设计模式的优缺点进行分析。

**优点：**

* 桥接设计模式能够解耦接口及其实现。这样，系统可以在运行时配置抽象的实现，对象甚至可以在运行时更改其实现；
* 可扩展性有所提升，开发者可以独立地扩展抽象及其实现的层次结构；
* 对客户端隐藏实现细节，使得客户端免受实现细节的影响，比如共享对象及其附带的引用计数机制。

**缺点：**

* 通常来说，采用桥接设计模式往往会增加系统的理解与设计难度。桥接设计模式使关联关系建立在抽象层上，这使得开发者必须在项目设计初期时对抽象层进行设计与测试；
* 桥接设计模式要求开发者有能力正确地识别出系统中两个模块独立变化的维度，并对其使用范围做出约束。此外，要正确地识别出模块间独立变化的维度也需要一定程度的开发经验积累。

4.3 复合（Composite）

4.3.1 简介

将对象组合成树结构来表示部分-整体层次结构。组合让客户端统一对待单个对象和对象的组合。

复合设计模式的使用场景如下：

* 开发者希望表示对象的部分-整体层次结构；
* 开发者希望客户端能够忽略对象组合和单个对象之间的差异。客户端将统一处理组合结构中的所有对象。

图5为复合设计模式的结构图。

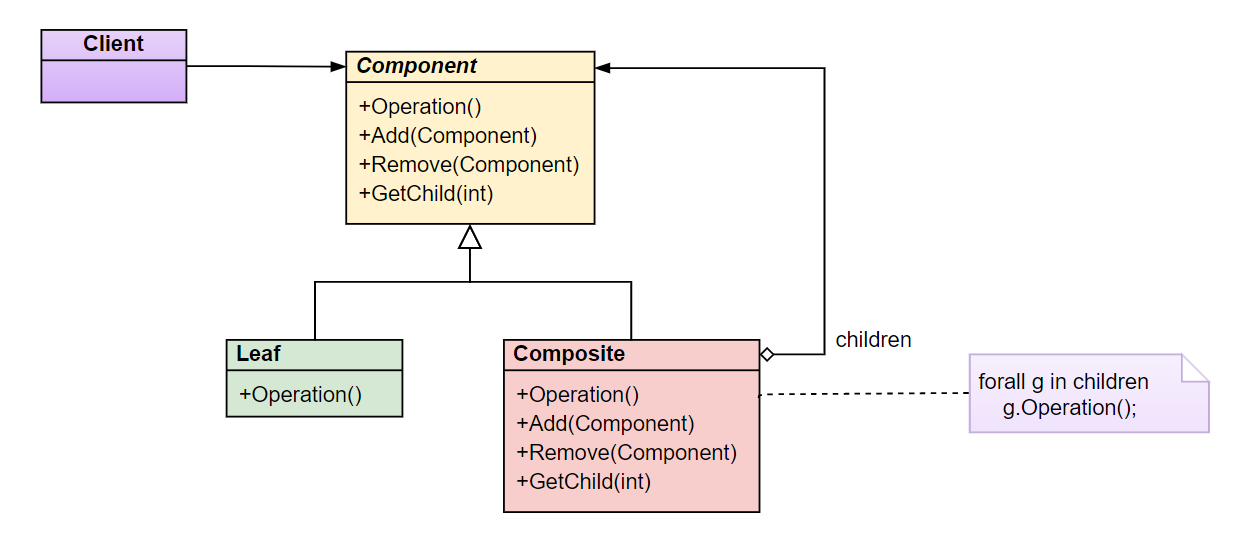


图 5：复合设计模式结构图

从此图可以看到总共有4个参与者，具体如下：

* Component
  + 声明复合类中的接口；
  + 采用恰当的方式，实现所有类共有接口的默认行为；
  + 声明一个访问及管理子组件的接口；
  + （可选）在合适的情况下，定义一个可以在递归结构内访问组件父类的接口。
* Leaf
  + 代表复合组件的叶部件，叶部件处于整体复合组件的最底层；
  + 在复合组件中，定义元部件的行为。
* Composite
  + 为有子类的部件定义它们的行为；
  + 储存子类部件；
  + 在Component类中实现与子类部件相关的操作。
* Client
  + 通过Component类，实现对复合部件的操作。

4.3.2 项目示例

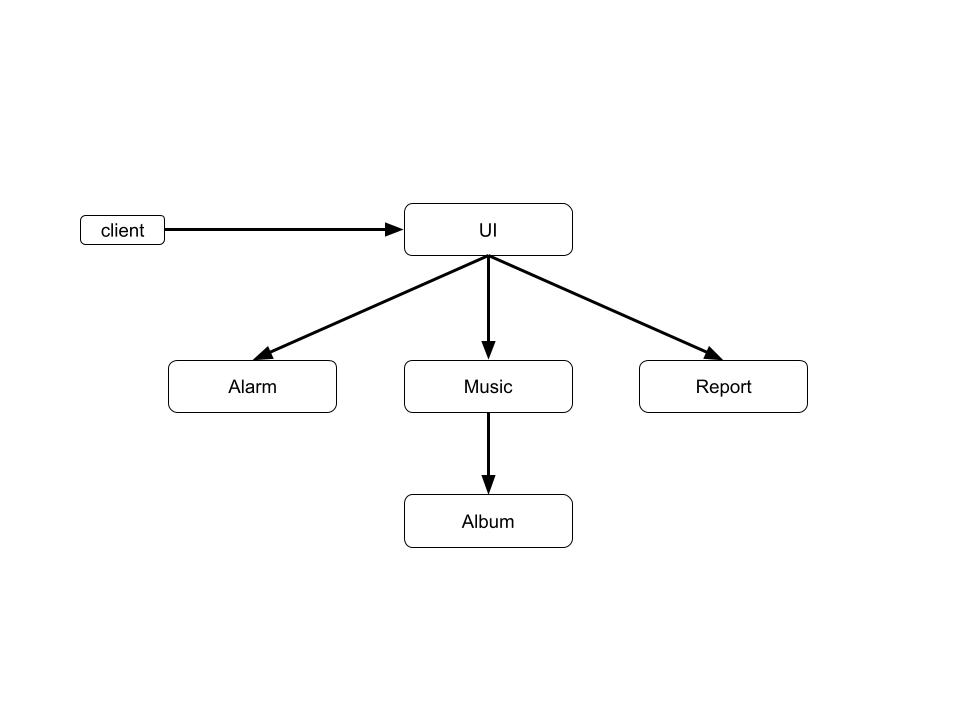


图 6：使用复合设计模式的项目模块

如图6所示，UI是一个封装了多个模块的复合类。其中，Alarm模块为项目的智能闹钟模块，Music为音乐播放模块，Report为睡眠报告模块，Album为Music模块下的助眠专辑模块。通过复合的方式，用户便能通过UI控制系统的每个重要组件，从而为用户的日常使用提供便捷性。

由于Alarm以及Report为叶部件，因此它们将直接处理用户的请求，即时地返回处理结果。对于Music部件而言，它通常会将用户请求进一步地转发至它的子部件，即Album部件中，从而实现复合部件的协调工作。

4.3.3 优缺点分析

现对复合设计模式的优缺点进行分析。

**优点：**

* 定义了由基本对象和复合对象组成的类层次结构。原始对象可以组合成更复杂的对象，然后再组合成更复杂的对象；
* 简化了客户端接口，客户端现在可以统一地处理复合结构和单个对象；
* 使添加新类型的组件更加容易。

**缺点：**

* 使用复合设计模式时，不能对复合类型施加约束，必须在运行时刻对其类型做检查；
* 会使项目的设计过于一般化。由于复合设计模式易于添加新组件，这使得组件的功能可以随意更改，难以对复合组件所能实现的功能做出限制。

4.4 外观（Facade）

4.4.1 简介

外观设计模式为子系统中的一系列接口提供统一的外部接口。它通过定义更高级别的接口（如抽象），使子系统更易于使用。

由于将系统构造成一系列的子系统有利于降低复杂性。通常项目的设计目标是最小化子系统之间的通信和依赖关系。实现该目标的一种方法是引入外观对象，该对象能够为所有的子系统组件提供通用、单一并简化的接口。

外观设计模式的一般使用场景如下：

* 通常随着项目不断发展，子系统也会变得更加复杂。因此，开发者希望为复杂的子系统提供一个简单的接口；
* 在客户端和抽象类的实现之间存在许多依赖关系。引入外观将子系统与客户端和其他子系统解耦，从而提高子系统的独立性和可移植性；
* 开发者想要分层他/她的子系统。使用外观来定义每一级子系统的入口点。如果子系统存在依赖关系，那么开发者可通过外观设计模式的方式，让它们彼此通信，从而简化它们之间的依赖关系。

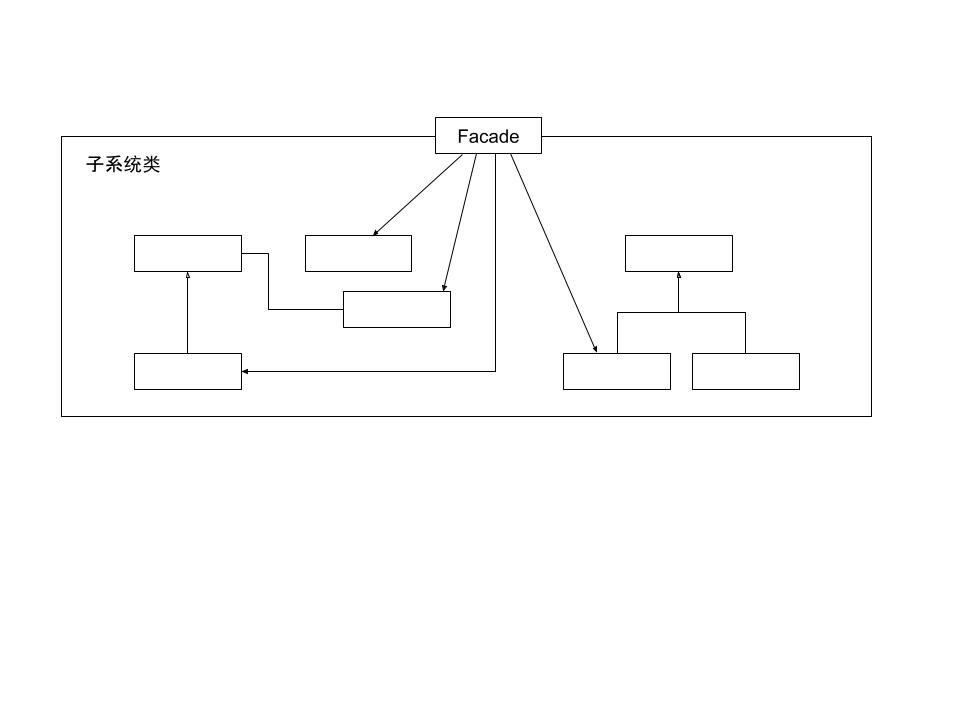
图7为外观设计模式的结构图。

图 7：外观设计模式结构图

上图中共显示了2个参与者，具体如下：

* 外观（Facade）
  + 知晓那些子系统模块能够处理用户请求；
  + 当用户发送一个请求时，外观模块能够根据该请求类型将请求转发至相应的子系统处。
* 子系统模块
  + 实现相应子系统的功能；
  + 处理由外观模块委派下来的工作；
  + 没有任何外观模块的相关信息，即这些模块不了解外观模块的存在。

4.4.2 项目示例

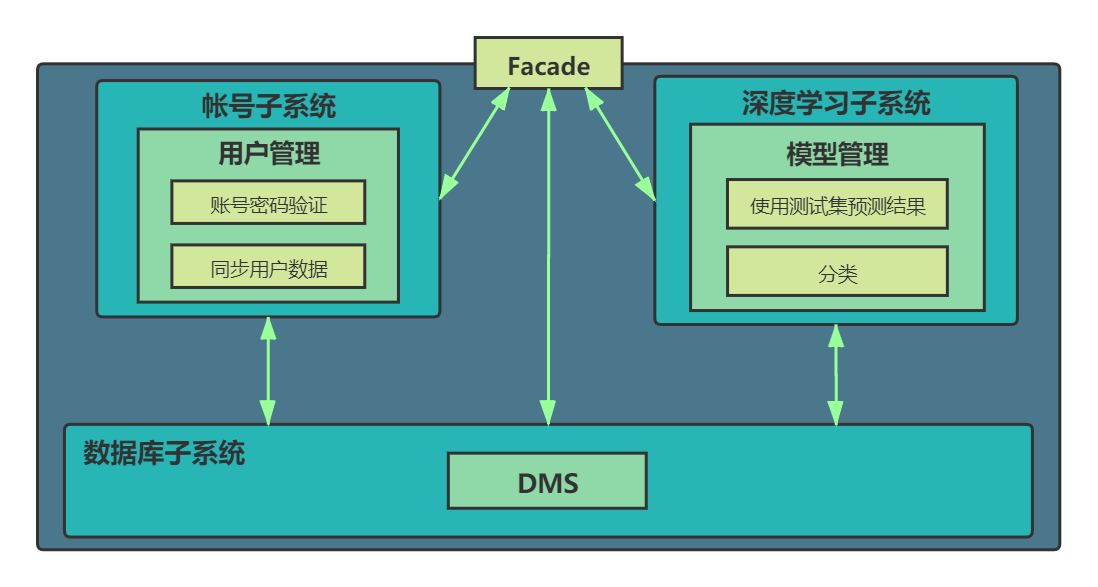


图 8：使用外观设计模式的项目模块

如图8所示，项目中一共由3个子系统，分别为帐号、深度学习以及数据库子系统。其中，DMS（Database Management System）为数据库管理模块；深度学习模块通过收集用户的睡眠数据来训练深度学习模型。完成模型训练后，使用模型预测用户的睡眠情况，并将不同睡眠特征的用户区分为其相对应的类别，如“百灵鸟”（早起的人）与“猫头鹰”（晚期的人）等。

回到图中，可以看到Facade实现了一个客户端与项目子系统进行通讯的接口，并根据客户端请求的类型转发到对应的子系统中。这样的设计使得客户端能够以更高效的方式与子系统通讯，并保证了各子系统可以各司其职，不会相互干扰。

4.4.3 优缺点分析

现对外观设计模式的优缺点做详细分析，具体如下。

**优点：**

* 外观设计模式能使客户端不受子系统组件的影响，从而减少了客户端需进行通讯的对象数，使子系统更易于使用。
* 此模式可以促进子系统与客户端间的弱耦合。子系统中的组件通常是强耦合的。弱耦合允许开发者在不影响客户端的情况下改变子系统的组件；
* 如果应用程序需要使用子系统类，系统将允许这样的行为。因此开发者可以在易用性和通用性之间进行选择。

**缺点：**

* 当不引入抽象外观类时，如果开发者增加新的子系统，便有可能需要修改外观类或客户端的代码，从而违背了“开闭原则”；
* 外观设计模式无法有效地限制用户使用子系统类，降低了系统的整体安全性。相反，如果对用户访问子系统类做出了过多的限制，则会系统整体的可变性与灵活性。