***阅读设计模式资料（或查阅其它相关资料），结合项目的进程和开发历程，分析项目采用了那些设计模式.***

朱靖哲：

根据对该链接提供的项目（https://en.wikipedia.org/wiki/Design\_Patterns ）的进程和开发历程，以下是我认为项目采用的设计模式：

1. 工厂模式（Factory Pattern）：在项目中，对于数据读取和处理的部分，采用了工厂模式进行封装。例如，在kmeans包中，定义了一个DataInputFactory类，用于根据输入数据的文件类型（如文本文件、HDFS文件等）创建不同的数据读取对象。这里采用了工厂模式，将对象的创建与客户端代码解耦，使得代码更加清晰和易于维护。
2. 单例模式（Singleton Pattern）：在项目中，有些类只需要被创建一次，例如KMeans类和kmeansIter类。这些类采用了单例模式进行封装，保证在整个程序中只有一个实例存在，避免了资源浪费和不必要的对象创建。
3. 观察者模式（Observer Pattern）：在项目中，有一个类Cluster，用于封装每个簇的信息。该类中有一个成员变量points，用于存储属于该簇的所有数据点。当进行K-means算法迭代时，每个数据点都会被分配到其所属的簇中，而每个簇的信息都会被更新。这里采用了观察者模式，通过将Cluster类作为观察者，将数据点作为被观察者，数据点的变化会自动通知其所属的簇，从而更新簇的信息。
4. 迭代器模式（Iterator Pattern）：在项目中，有一个类Cluster，用于封装每个簇的信息。该类中有一个成员变量points，用于存储属于该簇的所有数据点。在对每个簇进行处理时，需要遍历该簇中的所有数据点，进行计算和更新。这里采用了迭代器模式，通过在Cluster类中实现一个Iterator接口，将数据点的遍历过程封装起来，使得代码更加简洁和易于维护。
5. 模板方法模式（Template Method Pattern）：在项目中，有两个类KMeans和kmeansIter，分别用于实现K-means算法的主要逻辑和迭代过程。这两个类中都定义了一些方法，如initCenters()、calculateDistance()、map()、reduce()等，这些方法都具有固定的执行流程和算法步骤，但具体的实现细节可能会有所不同。这里采用了模板方法模式，将算法的主要流程和步骤进行抽象和封装，而具体的实现则由子类来完成，使得代码更加灵活和可扩展。
6. 策略模式（Strategy Pattern）：在项目中，有一个类Point，用于表示数据点的坐标信息。在K-means算法中，需要计算每个数据点到中心点的距离，而距离的计算方式可能有多种，如欧几里得距离、曼哈顿距离等。这里采用了策略模式，将距离的计算方式抽象成一个接口DistanceCalculator，并在Point类中使用该接口来计算距离，从而实现了距离计算方式的灵活切换和扩展。
7. 适配器模式（Adapter Pattern）：在项目中，数据读取和处理的部分采用了不同的技术和工具，如Hadoop、MapReduce、Java IO等，这些工具之间的接口和数据格式可能不完全相同。为了使它们能够协同工作，需要进行一定的适配和包装。例如，在kmeans包中，定义了一个DataInput接口，用于统一不同数据读取工具的接口，然后分别实现了TextFileInput和HDFSFileInput两个类，分别对应文本文件和HDFS文件的数据读取。这里采用了适配器模式，将不同的数据读取工具适配成统一的接口，使得它们可以互相替换和兼容。

综上所述，这个项目采用了多种设计模式，包括工厂模式、单例模式、观察者模式、迭代器模式、模板方法模式、策略模式和适配器模式，这些模式都有它们各自的优点和适用场景，能够有效地提高代码的可扩展性、可维护性和复用性，使得项目开发更加高效和灵活。

王爽：

根据对于Design Patterns-Elements of Reusable Object-Oriented Software和The GoF Design Patterns Reference这两个书籍的分析，我总结并分析了以下23种设计模式：

这些模式可以被分为三个主要的分类：创建型模式（Creational Patterns）、结构型模式（Structural Patterns）和行为型模式（Behavioral Patterns）。以下是对这23种设计模式的简要分析：

创建型模式（Creational Patterns）：

1. 单例模式（Singleton Pattern）：确保一个类只有一个实例，并提供一个全局访问点。
2. 简单工厂模式（Simple Factory Pattern）：使用一个工厂类来创建对象，客户端通过工厂类获取对象实例。
3. 工厂方法模式（Factory Method Pattern）：定义一个创建对象的接口，但由子类决定实例化的类是哪一个。
4. 抽象工厂模式（Abstract Factory Pattern）：提供一个创建相关或依赖对象族的接口，而无需指定具体类。
5. 建造者模式（Builder Pattern）：将一个复杂对象的构建与其表示分离，使得同样的构建过程可以创建不同的表示。

结构型模式（Structural Patterns）：

1. 适配器模式（Adapter Pattern）：将一个类的接口转换成客户端所期望的接口。
2. 桥接模式（Bridge Pattern）：将抽象部分与其实现部分分离，使它们可以独立地变化。
3. 组合模式（Composite Pattern）：将对象组合成树形结构以表示"整体-部分"的层次结构。
4. 装饰者模式（Decorator Pattern）：动态地给对象添加额外的行为或功能，而无需修改其原始类。
5. 外观模式（Facade Pattern）：为子系统中的一组接口提供一个统一的接口，简化客户端与子系统的交互。
6. 享元模式（Flyweight Pattern）：通过共享对象来有效地支持大量细粒度的对象。
7. 代理模式（Proxy Pattern）：为其他对象提供一个代理以控制对这个对象的访问。

行为型模式（Behavioral Patterns）：

1. 观察者模式（Observer Pattern）：定义了对象之间的一对多依赖关系，使得当一个对象状态改变时，其相关对象会被通知并自动更新。
2. 模板方法模式（Template Method Pattern）：定义一个算法的骨架，将一些步骤的实现延迟到子类中。
3. 命令模式（Command Pattern）：将请求封装成对象，使得可以用不同的请求对客户端进行参数化。
4. 状态模式（State Pattern）：允许对象在其内部状态改变时改变它的行为。
5. 责任链模式（Chain of Responsibility Pattern）：将请求的发送者和接收者解耦，使多个对象都有机会处理这个请求。
6. 解释器模式（Interpreter Pattern）：给定一个语言，定义它的文法的一种表示，并定义一个解释器来解释这个语言中的句子。
7. 中介者模式（Mediator Pattern）：用一个中介对象封装一系列对象的交互，使得对象之间不需要显式地相互引用。
8. 备忘录模式（Memento Pattern）：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。
9. 访问者模式（Visitor Pattern）：将算法与对象结构分离，使得算法可以独立地变化而不影响对象结构。
10. 策略模式（Strategy Pattern）：定义一系列的算法，并将其封装在各自的类中，使得算法可以互相替换。
11. 迭代器模式（Iterator Pattern）：提供一种顺序访问一个聚合对象中各个元素的方法，而无需暴露其内部表示。

这些设计模式提供了一种通用的解决方案，可以帮助开发人员解决特定的设计问题。通过应用这些模式，可以提高代码的可复用性、可维护性和灵活性，同时降低系统的耦合度。

而基于以上23种设计模式的分析，在我们自己的火车售票系统项目中，我们可以考虑以下的运用，这些运用在项目种有些已经实现，另一些可以考虑在后续的设计中进行补全和完善：

1. 单例模式（Singleton Pattern）：在火车售票系统中，可能会有一些需要唯一实例的对象，例如配置管理器或日志记录器。单例模式可以确保只有一个实例存在，并提供全局访问点。
2. 简单工厂模式（Simple Factory Pattern）：简单工厂模式可能在火车售票系统中用于创建不同类型的票务对象，例如普通票、学生票、儿童票等。通过简单工厂模式，可以将创建对象的逻辑封装在一个工厂类中，简化了对象的创建过程。
3. 工厂方法模式（Factory Method Pattern）：工厂方法模式可以用于火车售票系统中的票务对象的创建。每种票务类型可以由各自的工厂类来创建，从而实现更灵活的对象创建。
4. 抽象工厂模式（Abstract Factory Pattern）：抽象工厂模式适用于火车售票系统中的票务系统的不同部分具有不同的实现，例如不同的票务供应商。抽象工厂模式允许客户端使用抽象接口来创建相关的对象族，而无需关心具体的实现细节。
5. 建造者模式（Builder Pattern）：如果火车售票系统中的票务对象具有复杂的构建过程，可以考虑使用建造者模式。建造者模式可以将构建逻辑和表示分离，从而可以根据需要创建不同的对象。
6. 适配器模式（Adapter Pattern）：适配器模式可以在火车售票系统中用于将不兼容的接口转换为客户端所期望的接口。例如，如果火车售票系统需要与外部支付系统进行集成，可以使用适配器模式来适配两个系统之间的接口差异。
7. 桥接模式（Bridge Pattern）：桥接模式可以用于将火车售票系统中的抽象部分与其具体实现分离。例如，在票务系统中，可以使用桥接模式将不同的支付方式与具体的票务对象解耦。
8. 组合模式（Composite Pattern）：如果火车售票系统中的座位布局具有层次结构，可以考虑使用组合模式来管理座位的层次关系。组合模式可以使得单个座位和座位组合都被当做统一的对象来处理。
9. 装饰者模式（Decorator Pattern）：装饰者模式可以用于在不修改现有代码的情况下，动态地给火车售票系统中的对象添加额外的功能。例如，可以使用装饰者模式来给票务对象添加额外的服务费计算功能。
10. 外观模式（Facade Pattern）：外观模式可以为火车售票系统提供一个简化的接口，使得客户端可以更方便地使用系统的各个子系统。外观模式可以隐藏系统的复杂性，并提供一个统一的入口。
11. 代理模式（Proxy Pattern）：代理模式可以在火车售票系统中用于控制对某些敏感对象的访问。例如，可以使用代理模式来限制对票务对象的访问权限。
12. 观察者模式（Observer Pattern）：观察者模式可以用于实现火车售票系统中的消息通知机制。例如，当票务状态发生变化时，可以通知已经预订的乘客。
13. 模板方法模式（Template Method Pattern）：模板方法模式可以用于火车售票系统中一些流程的实现。例如，在订单生成过程中，可以使用模板方法模式定义一个订单生成的基本流程，然后由具体的子类来实现其中的具体步骤。
14. 命令模式（Command Pattern）：命令模式可以在火车售票系统中用于将请求封装成对象，并将其作为参数传递给其他对象进行处理。例如，在系统中可以使用命令模式来实现订单处理功能，将每个订单请求封装成一个命令对象，然后由相关对象进行执行。
15. 状态模式（State Pattern）：状态模式可以在火车售票系统中用于管理不同的状态以及状态之间的转换。例如，在系统中可以使用状态模式来管理订单的不同状态（待支付、已支付、已取消等），并根据状态执行相应的操作。
16. 责任链模式（Chain of Responsibility Pattern）：责任链模式可以在火车售票系统中用于处理请求的链式处理。例如，可以使用责任链模式来处理用户的请求，将请求传递给一系列的处理者，每个处理者都有机会处理请求，直到请求被处理完成或到达链的末尾。
17. 解释器模式（Interpreter Pattern）：解释器模式可以在火车售票系统中用于解析和执行特定的语言或规则。例如，如果系统中有一种特定的查询语言用于查询票务信息，可以使用解释器模式来解析和执行查询语言。
18. 中介者模式（Mediator Pattern）：中介者模式可以用于火车售票系统中的组件之间的解耦和通信。例如，可以使用中介者模式来管理不同模块之间的交互，将复杂的交互逻辑封装在中介者对象中，从而减少模块间的直接依赖关系。
19. 备忘录模式（Memento Pattern）：备忘录模式可以在火车售票系统中用于保存和恢复对象的状态。例如，在系统中可以使用备忘录模式来保存订单的历史状态，以便在需要时可以恢复到先前的状态。
20. 访问者模式（Visitor Pattern）：访问者模式可以用于火车售票系统中的数据结构和操作之间的解耦。例如，可以使用访问者模式来定义一组操作，这些操作可以在不修改数据结构的情况下被应用于数据结构中的不同元素。
21. 策略模式（Strategy Pattern）：策略模式可以在火车售票系统中用于选择不同的算法或策略。例如，在系统中可以使用策略模式来选择不同的票价计算策略，根据不同的情况选择合适的算法来计算票价。
22. 迭代器模式（Iterator Pattern）：迭代器模式可以在火车售票系统中用于遍历和访问集合对象的元素。例如，在系统中可以使用迭代器模式来遍历和访问订单列表中的各个订单对象。