Que：参考教材6.2，结合项目的进程和开发历程，从设计原则的几个方面，对负责设计的模块进行评估，写出存在的问题和解决方案思考。

设计原则是指把系统功能和行为分解成模块的指导方针。它从两种角度明确了我们应该使用的标准:系统分解，以及确定在模块中将要提供哪些信息（和隐蔽哪些信息)。设计原则在进行创新性设计时十分有用，此外它们还有其他用武之地，特别是在设计公约、设计模式和体系结构风格的设计建议基础的形成过程中。因此，为了合理有效地使用风格和模式，我们必须理解和欣赏它们所隐含的原则，否则，当定义、修改和拓展模式与风格时，我们极有可能违背了该公约或模式使用和提倡某个原则的初衷。

设计原则的几个方面

模块化

模块化，也称作关注点分离，是一种把系统中各不相关的部分进行分离的原则,以便于各部分能够独立研究。关注点可以是功能、数据、特征、任务、性质或者我们想要定义或详细理解的需求以及设计的任何部分。为了实现模块化设计，我们通过辨析系统不相关的关注点来分解系统，并且把它们放置于各自的模块中。如果该原则运用得当，每个模块都有自己的唯一目的，并且相对独立于其他模块。使用这种方法，每个模块理解和开发将会更加简单。同时，模块独立也将使得故障的定位和系统的修改更加简单（因为对于每一个故障，可疑的模块会减少，且一个模块的变动所影响的其他模块会减少)。

为了确定一个设计是否很好地分离了关注点，我们使用两个概念来度量模块的独立程度:耦合度和内聚度。
1.耦合度
当两个模块之间有大量依赖关系时，我们就说这两个模块是紧密耦合的(tighly coupled)。松散耦合的（loosely coupled）模块之间具有某种程度的依赖性，但是它们之间的相互连接比较弱。非耦合的(uncoupled）模块之间没有任何相互连接，它们之间是完全独立的。

模块之间的依赖方式有很多种。
·一个模块引用另一个模块。模块A可能会调用模块B的操作，因此，模块A为了完成其功能或处理，依赖于模块B。
·一个模块传递给另一个模块的数据量。模块A可能会传递参数、数组的内容或者数据块给模块B。
·某个模块控制其他模块的数量。模块A可能会将一个控制标记传给B。标记的值会告诉模块B某些资源或子系统的状态，调用哪个进程,或者是否需要调用某个进程。

实际上，一个系统不可能建立在完全非耦合的模块上。就像一张桌子和几把椅子一样，尽管是互相独立的，但也可以组成一套餐厅用具。因此，上下文环境可能会间接地耦合那些似乎是非耦合的模块。例如，如果一个功能中止另一个功能的执行，那么这两个不相关的功能就会进行交车(比如，授权认证功能会禁止非授权用户取得受保护的服务)，因此，没有必要使模块完全独立。只要尽可能地减少模块之间的

锅合度即可。
某些类型的耦合与其他类型相比，是不尽如人意的。最不希望发生的情况是，一个模块实际上修改了另外一个模块。如果出现这样的情况，被修改的模块就完全依赖于修改它的那个模块。我们称之为内容耦合(content coupling)。当一个模块修改了另外一个模块的内部数据项，一个模块修改了另一个模块的代码，或一个模块内的分支转移到另外一个模块中的时候，就可能出现内容耦合。

通过对设计进行组织，使其从公共数据存储区来访问数据。我们可以在某种程度上减少耦合的数量。但是，依赖关系仍然存在，因为对公共数据的改变意味着需要通过反向跟踪所有访问过该数据的模块来评估该改变的影响。这种依赖关系称为公共耦合（common coupling)。就公共耦合而言，很难确定是哪个模块把某个变量设置成一个特定值的。

当某个模块通过传递参数或返回代码来控制另外一个模块的活动时，我们就说这两个模块之间是控制耦合(control coupling)的。受控制的模块如果没接收到来自控制模块的指示，是不可能完成其功能的。控制耦合的设计有一个优点:可以使每个模块只完成一种功能或只执行一个进程。这种限制把从某个模块传送到另外一个模块所必需的控制信息量减到最少，并且把模块的接口简化成固定的、可识别的参数和返回值的集合。
如果使用一个复杂的数据结构来从一个模块到另一个模块传送信息，并且传递的是该数据结构本身，那么两个模块之间的耦合就是标记耦合(stamp coupling)。如果传送的只是数据值，不是结构数据，那么模块之间就是通过数据耦合（data coupling)连接的。标记耦合体现了模块之间更加复杂的接口，因为在标记耦合中，两个交互的模块之间的数据的格式和组织方式必须是匹配的。因此，数据耦合更简单，而且因数据表示的改变而出错的可能性很小。如果模块之间必须有耦合，那么数据耦合是最受欢迎的。它是跟踪数据并进行改变的简便方法。
面向对象设计中的模块通常有较低的耦合度，因为每个对象模块的定义都包含了它自己的数据和作用于这些数据上的操作。实际上，面向对象设计方法目标之一就是为了实现松散耦合。然而，基于对象进行的设计并不能保证在最终的设计结果中所有模块之间都有低耦合。例如，如果我们生成一个对象，这个对象存储了公共数据，通过访问它的方法，其他对象可以控制这些公共数据，这样,这些控制对象之间就形成了公共耦合。

2. 内聚度

与度量模块之间的相互依赖性相比，内聚度是指模块的内部元素（比如，数据、功能、内部模块）的“粘合”程度。一个模块的内聚度越高，模块内部的各部分之间的相互联系就越紧密，与总体目标就越相关。一个模块如果有多个总体目标，它的元素就会有多种变化方式或变化值。例如，一个模块同时包含了数据和例程，并用以来显示那些数据，这个模块可能会频繁更改且以不同的方式变更，因为每次使用这些数据时都需要使用改变这些值的新功能和显示这些值的新方法。我们的目的是尽可能地使模块高内聚，这样各个模块才能易于理解，减少更。

内聚度最低的是巧合内聚(coincidental cohesion)，这时，模块的各个部分互不相关。在这种情况下，只是由于为了方便或是偶然的原因，不相关的功能、进程或数据处于同一个模块中。例如，一个设计含有若干个内聚的模块，但是其他的系统功能都统统放在一个或多个杂项模块中，这种设计不是我们所期望的。
当一个模块中的各个部分只通过代码的逻辑结构相关联时，我们称这个模块具有逻辑内聚(logical cohesion)。

有时，设计被划分为几个用来表示不同执行状态的模块:初始化、读进输入、计算、打印输和清除，这样的内聚是时态内聚(（temporal cohesion)，在这种模块中数据和功能仅仅因在一个任中同时被使用而形成联系。这样的设计会造成代码的重复，因为会有多个模块对关键数据结构都有类似的操作。在这种情况下，对数据结构的改动会引起所有与之相关的模块的变动。在面向对象程序中，对象的构造函数和析构函数有助于避免初始化模块和清除模块中的时态内聚。
通常，必须按照某个确定的顺序执行一系列功能。例如，必须先输入数据，然后进行检查，然后操纵数据。如果模块中的功能组合在一起只是为了确保这个顺序，那么该模块就是过程内聚的(procedurally cohesive)。过程内聚和时态内聚类似，但过程内聚有另外一个优点:其功能总是涉及相关的活动和针对相关的目标。然而，这样一种内聚只会出现在模块本身运用的上下文环境中。倘若不知道该模块的上下文环境,我们很难理解模块如何以及为什么会这样工作，也很难去修改此模块。
或者，我们还可以将某些功能关联起来，因为它们是操作或生成同一个数据集的。例如，有时可以将不

相关的数据一起取出，因为它们由同一个输入传感器搜集，或者通过一次磁盘访问就可以取到它们。这样围绕着数据集构造的模块是通信内聚的(communicationally cohesive)。解决通信内聚的对策是将各数据元素放到它本身的模块中去。
我们理想的情况是功能内聚(functional cohesion)，它满足以下两个条件:在一个模块中包含所有必需的元素，并且每一个处理元素对于执行单个功能来说都是必需的。某个功能内聚的模块不仅执行设计的功能，而且只执行该功能，不执行其他任何功能。
在功能内聚的基础上，将其调整为数据抽象化和基于对象的设计，这就是信息内聚(informationcohesion)。它们的设计目的是相同的:只有当对象和动作有着一个共同且明确的目标时，才将它们放到一起。例如，如果每一个属性、方法或动作对于一个对象来讲都是必需的，那就说这个面问对象的设计模块是内聚的。面向对象系统通常有较高内聚的设计，因为每个模块中含有单一的，且可能是复杂的数据类型和所有对该数据类型的操作。

存在的问题及解决方案

当前项目离低耦合高内聚仍有较大距离，有些模块比较混乱，代码耦合度很高，需要修改

接口

在第4章中我们看到，软件系统有一个外部边界和一个对应的接口，通过这个接口软件系统可以感知和控制它的环境。类似地，每个软件单元也有一个边界将它和系统的其余部分分开，以及一个接口来和其他软件单元进行交互。接口(interface)为系统其余部分定义了该软件单元提供的服务，以及如何获取这些服务。一个对象的接口是该对象所有公共操作以及这些操作的签名(signature)的集合，指定了操作名称、参数和可能的返回值。更全面地讲，依据服务或假设，接口还必须定义该单元所必需的信息，以能确保该单元能够正确工作。例如，在上述对象的接口中，对象的操作可能要使用程序库、调用外部服务，或者只有上下文环境符合一定条件时才能被调用，而一旦不满足或违背这些条件中的任何一个，操作将不能如预期那样提供应有的功能。因此，软件单元的接口描述了它向环境提供哪些服务，以及对环境的要求。

接口是这样一种设计结构，它对其他开发人员封装和隐藏了软件单元的设计和实现细节。比如，我们定义堆栈对象stack和操作堆栈的方法push利pop，而不是直接控制堆栈的变量，我们使用对象和方法而不是堆栈本身来改变堆栈的内容。我们还可以定义探头(probe)来向我们提供堆栈的信息(是空是满，栈顶元素是什么)，而不需要对堆栈状态做任何改变。
软件单元接口的规格说明(specification）描述了软件单元外部可见的性质。正如需求规格说明从系统边界的角度描述系统行为一样，接口的规格说明的描述以单元的边界为依据对软件单元做出描述:该单元的访问函数、参数、返回值和异常。一个接口的规格说明需向其他系统开发人员传达正确应用该软件单元的所有信息，这些信息并不仅仅局限于单元的访问功能和它们的签名，还有如下几点。
目标:我们为每个访问函数的功能性建立充分详细的文档，以帮助其他开发人员找出最符合他们需要的访问函数。
前置条件:我们列出所有假设，又称为前置条件（precondition)（如，输入参数的值、全局资源的状态，或者存在哪些程序库及软件单元)，以帮助其他开发人员了解在何种情况下该软件单元才能正确工作。
协议:协议的信息包括访问函数的调用顺序、两个构件交换信息的模式。比如，一个模块进行调用共享资源之前需要被授权允许。
·后置条件:我们将可见的影响称为后置条件（ postcondition)。我们为每个访问函数的后置条件编写文档，包括返回值、引发的异常以及公共变量(如输出文件）的变化，这样，调用它的代码才能对函数的输出做出适当的反应。
质量属性:这里描述的质量属性（如，性能、可靠性）是对开发人员和用户可见的。例如，我们软件的一名客户可能想知道是否已经为数据插入或检索优化了相关的内部数据结构。(一种操作的优化往往会降低其他操作的性能。)
理想的情况是，单元的接口规格说明精确定义了所有可接受的实现的集合。但实际中，该规格说明必须足够精确，才能保证任何满足规格说明的实现都是可接受的。例如，操作Find返回列表中一个元素的索引，这个操作的规格说明必须陈述相同元素多次在列表出现时会发生的情况（如，可以返回第一次出现的该元素的索引，或任一次出现该元素的索引)，没有找到该元素或者列表为空时会发生什么，等等。另外，规格说明也不能过于严格以避免将一些可接受的实现排除在外，例如,操作Find的规格说明不应该指定操作必须返回第一次出现该元素的索引，因为任何一次都是可以的。

接口规格说明也使得其他开发人员不能深入了解和探究我们的设计决策。初看上去，允许其他开发人员在我们的软件设计的基础上优化他们的代码似乎是合理的，但是这种优化是软件单元之间的一种形式的耦合，它会降低软件的可维护性。如果--名开发人员要依赖于我们软件的实现方式来编写他的代码，那么他的代码和我们的代码之间的接口就已经改变了:他需要了解更多已有规格说明以外的信息。当我们想对软件做出改动来满足新的实现时，要么我们保持当前接口不变，要么其他开发人员改变他的代码。
软件单元的接口还暗示着耦合的本质含义。如果一个接口将访问限制在一系列可被调用的访问函数之内，那么它们之间就没有内容耦合。但如果其中一些访问函数有复杂的数据参数，那么可能会存在标记耦合。为了实现低耦合，我们想将单元的接口设计得尽可能的简单，同时我们也想将软件单元对于环境的假设和要求降至最低，来降低系统其他部分的改变会违背这些假设条件的可能性。

存在的问题及解决方案

目前程序接口应用的较好 , 暂无问题

信息隐藏

信息隐藏（information hiding)(Parnas 1972)的目标是使得软件系统更加易于维护。它以系统分解为特征:每个软件单元都封装了一个将来可以改变的独立的设计决策，然后我们根据外部可见的性质，在接口和接口规格说明的帮助下描述各个软件单元。因此，这个原则的名称本身也反映了它的结果:单元的设计决策被隐藏了。
“设计决策”这种说法其实是很笼统的，它可以有很多指代，包括数据形式或数据操作、硬件设备或者其他需要和软件交互的构件、构件之间消息传递的协议，或者算法的选择。因为设计过程牵涉到软件很多方面的决策，所以最终的软件单元封装了各种类型的信息。和第5章所述的分解方法学(例如，功能性分解、面向数据的分解）相比，根据信息隐藏来分解系统是有很大的不同的，因为前者只封装了同种类型的信息（换言之，它们只封装了函数、数据类型或过程)。补充材料6-2阐述了面向对象的设计方法如何很好地实现了信息隐藏。

因为我们想封装可变的设计决策，所以我们必须保证接口本身不涉及设计中可变的部分。以排序算法的封装为例,排序模块可以将输入串排序成有序的输出串，然而该方法却引起了标记耦合(即单元间传递的数据被限制成了字符串)。如果数据格式的可变性是一个设计决策，那么数据格式不应该暴露在模块的接口中。一个更好的设计应该是把数据封装在单个的、独立的软件单元中，而排序模块可以输入和输出任何对象类型，并使用在数据单元接口中声明的访问函数，实现对对象的数据值检索和排序的功能。
通过遵循信息隐藏原则，--个设计将会被分解成很多小的模块，再者，这些模块可能具有了所有类型的耦合，比如:隐藏了数据表达形式的模块可能是信息内聚的;隐藏了算法的模块可能是功能内聚的;·隐藏了任务执行顺序的模块可能是过程内聚的。
因为每个软件单元只隐藏了一个特定的设计决策，所以它们都具有高内聚度。即使是过程内聚，软件单元都隐藏了单个任务的设计。随之而来的结果是大量的模块，这使我们的操作看起来变得难以控制，但是我们会有方法在模块数量和信息隐藏之间做出权衡。在本章的后面，我们将会看到如何使用依赖图和抽象来管理大量的模块集合。
信息隐藏的一个很大的好处是使得软件单元具有低耦合度。每个单元的接口列出了该单元提供的访问函数和需要使用的其他访问函数的集合。这个特征使得软件单元易于理解和维护，因为每个单元相对来说都是自包含的。如果我们能够正确地预测出随着时间的增长设计的哪些部分会有变化，那么随后的维护便会更容易，因为我们能够把变化的位置定位到特定的软件单元。

存在的问题及解决方案

目前信息隐藏做的较差 , 有些模块很偏向于面向过程的设计.

增量式开发

假定一个软件设计是由软件单元和它们的接口所组成的，我们可以使用单元之间的依赖关系来设计出一个增量式设计开发进度表。首先,我们指定单元间的使用关系(uses relation)(Parnas 1978b),它为各个软件单元和它依赖的单元之间建立关联。回顾我们关于耦合的讨论，两个软件单元A和B,它们不彼此调用也可能会互相依赖，例如，单元A依赖单元B构造一个数据结构，并存储在一个独立的单元C中，随后A再访问C。总的说来,如果软件单元A如它接口中描述的那样“需要一个正确的B",才能完成A的任务，那么我们

说软件单元A“使用”软件单元B (Parnas 1978b)。因此，倘若只有单元B正确工作才能保证A也能正确工作，则单元A使用单元B。以上的讨论是假设我们能从单元的接口规格说明中得知系统的使用关系，而当接口规格说明不能完整地描述单元间的依赖关系时，我们需要对每个单元的实现计划有充分认识，才能知道它将使用哪些其他单元。
从满足增量式开发的角度考虑，使用图也可以帮助我们确定可改善的设计范围。
设计良好的使用图应具有树型结构或者是树型结构的森林。在这样的结构中，每棵子树都是系统的一部分，所以我们可以一次一个软件单元地增量开发我们的系统，每个完成的单元都是我们系统的部分实现。在开发过程中，每一次的增量都会越来越易于测试和修改，因为错误只可能出现在新代码中，而不是在已经过测试和验证的被调用单元中。此外，我们总有一个可运行的系统版本用来展示给客户。更多地，系统频繁且可见的进展也鼓舞了开发人员的士气(Brooks 1995)。和其他方法相比，增量式开发有着不可多得的优势，因为前者只有当每个单元都能工作时系统才能工作。

存在的问题及解决方案

当前项目暂未采用增量式开发 , 由于前面解耦做的较差 , 完成一个阶段后需要重整增量式开发

抽象

抽象是一种忽略一些细节来关注其他细节的模型或表示。而在定义中，关于模型中的哪部分细节被忽略是很模糊的，因为不同的目标会对应不同的抽象，会忽略不同的细节。因此，通过回顾我们已经建立起的抽象，理解抽象这个概念将会更加容易。
系统被分解为各个子系统，每个子系统再被分解成更小的子系统，一直分解下去。其中分解的顶层给我们提供了问题系统层次上的纵览，同时对我们隐藏了那些可能会影响我们注意力的细节，有助于我们集中关注我们想要研究和理解的设计功能和特性。当我们观察低一层次的抽象时，我们会发现更多关于各软件单元的细节，它们牵涉到它的主要元素以及这些元素间的关系。各个抽象层次以这种方式隐藏了它的元素如何进一步分解的方法，而每个元素在接口规格说明中将被一一描述，这是另一种关注元素外部行为和避免元素内部设计细节被引用的抽象类型，这些细节将会在分解的下一个层次中显现出来。
正如我们在第5章中所见，一个系统可能不仅仅只有一个分解方法，我们会创建若干种不同的分解来展示不同的结构，譬如，--种视图可能展示了不同运行进程以及它们内部的联系，另一种视图则展示了分解成代码单元的系统。每个视图都是:-种抽象，它强调了系统结构设计的某个方面(如,运行进程)而忽略了其他结构信息（如，代码单元)和非结构细节。
第四种抽象类型是虚拟机(virtual machine)，例如在层次体系结构中的情况。任一层次i都使用了它下一层次i-1层所提供的服务，这样第i层便拥有了强大且可靠的服务，然后向它的上一层i+1层提供该服务。回想实际中的层次体系结构，每一层只能访问紧邻它的下一层所提供的服务，而不能访问更低层的服务（当然也不可能访问更高层的服务)。根据以上所述，层次i是将底层细节抽象化仅向下一层展现它的服务的虚拟机。设计的指导原则是:层次i的服务是它下面各层次所提供的服务的加强，因此也取代了它们。
对于一个特定的模型，一个好的抽象的关键是决定哪些细节是不相关的，进而可以被忽略的。抽象的性质取决于开始时我们建立这个模型的初衷:我们想交互哪些信息，或者我们想展示哪个析过程。补充材料6-3阐述了我们可以如何为有着不同目标的算法建立不同的抽象模型。

存在的问题及解决方案

暂无问题

通用性

通用性(generality)是这样一种设计原则:在开发软件单元时，使它尽可能地能够成为通用的软件，来加强它在将来某个系统中能够被使用的可能性。我们通过增加软件单元使用的上下文环境的数量来开发更加通用的软件单元，下面是几条实现规则。

- 将特定的上下文环境信息参数化:通过把软件单元所操作的数据参数化，我们可以开发出更加通用的软件。
- 去除前置条件:去除前置条件，使软件在那些我们之前假设不可能发生的条件下工作。
- 简化后置条件:把一个复杂的软件单元分解成若干个具有不同后置条件的单元，再将它们集中起来解决原来需要解决的问题，或者当只需其中一部分后置条件时单独使用。

存在的问题及解决方案

在开发中由于敏捷开发，大量功能模块之间的共同点没有理清楚，在不断堆叠中造成了通用性低的问题，例如在多个算法模块中都用到了同样的子算法，完全可以参考简化后置条件合并起来.

同时软件的运行条件做了大量假设，例如对用户输入进行的假设，可以参考去除前置条件，提高通用性.