Object design with Patterns

1. Introduction

After identifying your requirements and creating a domain model, the method of object design is carried out by adding methods to the software classes, and defining the messaging between the objects to fulfill the requirements. There are deep principles and issues involved in these steps. Deciding what methods belong where, and how the objects should interact, is important. And, this is a critical step - this is at the heart of what it means to develop an object-oriented system, not drawing domain model diagrams, package diagrams, and so forth.

It is possible to communicate the detailed principles and reasoning required to grasp basic object design, and to learn to apply these in a methodical approach that removes the magic and vagueness. The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way. This approach to understanding and using design principles is based on patterns of assigning responsibilities,

2. Responsibilities and Methods

A responsibility is defined as "a contract or obligation of a classifier." Responsibilities are related to the obligations of an object in terms of its behavior. Basically, these responsibilities are of the following two types:

1. Knowing
2. Doing

Doing responsibilities of an object include:

- doing something itself, such as creating an object or
- doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data knowing about related objects
- knowing about things it can derive or calculate

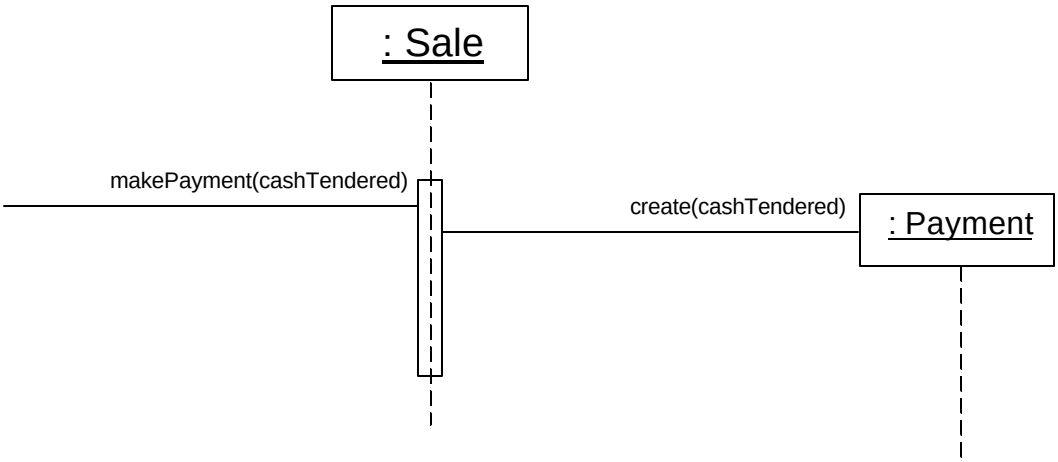
Responsibilities are assigned to classes of objects during object design. Relevant responsibilities related to "knowing" are often inferable from the domain model, because of the attributes and associations it illustrates. The translation of responsibilities into classes and methods is influenced by the granularity of the responsibility. The responsibility to 'provide access to relational databases" may involve dozens of classes and hundreds of methods, packaged in a subsystem.

A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities. Responsibilities are implemented using methods that either act alone or collaborate with other methods and objects.

3. Responsibilities and Interaction Diagrams

Fundamental principles for assigning responsibilities to objects are applied methodically during the programming. A common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams (which are part of the UP Design Model).

The figure indicates that Sale objects have been given a responsibility to create Payments, which is invoked with a makePayment message and handled with a corresponding makePayment method. Furthermore, the fulfillment of this responsibility requires collaboration to create the SalesLineItem object and invoke its constructor.



Responsibility: to create Payments

In summary, interaction diagrams show choices in assigning responsibilities to objects. When created, decisions in responsibility assignment are made, which are reflected in what messages are sent to different classes of objects. These choices are reflected in interaction diagrams.

4. Patterns

Experienced object-oriented developers (and other software developers) buildup a repertoire of both general principles and idiomatic solutions that guide them in the creation of software. These principles and idioms, if codified in a structured format describing the problem and solution, and given a name, may be called patterns.

An example of a Pattern

Pattern Name : Information Expert

Solution: Assign a responsibility to the class that has the information needed to fulfill it.

Problem It Solves: What is a basic principle by which to assign responsibilities to objects?

In object technology, a pattern is a named description of a problem and solution that can be applied to new contexts; ideally, it provides advice in how to apply it in varying circumstances, and considers the forces and trade-offs. Many patterns provide guidance for how responsibilities should be assigned to objects, given a specific category of problem.

“One person’s pattern is another person’s primitive building block” is an object technology adage illustrating the vagueness of what can be called a pattern. This treatment of patterns will bypass the issue of what is appropriate to label a pattern, and focus on the pragmatic value of using the pattern style as a vehicle for naming, presenting, learning, and remembering useful software engineering principles.

All patterns ideally have suggestive names. Naming a pattern, technique, or principle has the following advantages;

- It supports chunking and incorporating that concept into our understanding and memory.
- It facilitates communication.

Naming a complex idea such as a pattern is an example of the power of abstraction - reducing a complex form to a simple one by eliminating detail. When a pattern is named, we can discuss with others a complex principle or design idea with a simple name. Also, chunking design idioms and principles with commonly understood names facilitate communication and raises the level of inquiry to a higher degree of abstraction

5. GRASP

They describe fundamental principles of object design and responsibility assignment, expressed as patterns. Understanding and being able to apply these principles during the creation of interaction diagrams is important because a software developer new to object technology needs to master these basic principles as quickly as possible; they form the foundation of how a system will be designed.

GRASP is an acronym that stands for General Responsibility Assignment Software Patterns. The name suggests the importance of grasping these principles to successfully design object-oriented software.

5.1. The first five GRAS Patterns

1. Information Expert
2. Creator
3. High Cohesion
4. Low Coupling
5. Controller

5.1.1. Information Expert (or Expert)

Information Expert is frequently used in the assignment of responsibilities; it is a basic guiding principle used continuously in object design. Expert is not meant to be an obscure or fancy idea; it expresses the common intuition that objects do things related to the information they have.

The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many "partial" information experts who will collaborate in the task. Whenever information is spread across different objects, they will need to interact via messages to share the work.

Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents. For example, in the real world, without the use of electro-mechanical aids, a sale does not tell you it total; it is an inanimate thing. Someone calculates the total of the sale. But in object-oriented software land, all software objects are "alive" or "animated," and they can take on responsibilities and do things. Fundamentally, they do things related to the information they know.

The Information Expert pattern - like many things in object technology - has a real analogy. We commonly give responsibility to individuals who have the information necessary to fulfill a task.

By Information Expert, we should look for that class of objects that has the information needed to determine the total. We look both in the Domain Model and the Design Model to analyze the classes that have the information needed. The Domain Model illustrates conceptual classes of the real-world domain; the Design Model illustrates software classes.

5.1.1.1. Problem/Solution pair

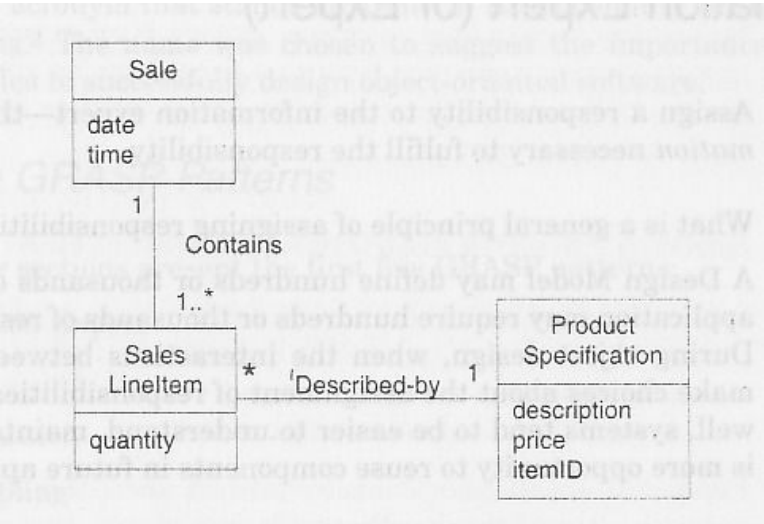
Solution: - Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.

Problem - What is a general principle of assigning responsibilities to objects?

A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. Done well, systems tend to be easier to understand, maintain, and extend, and there is more opportunity to reuse components in future applications.

5.1.1.2. Example: -

Consider the following partial domain model: -



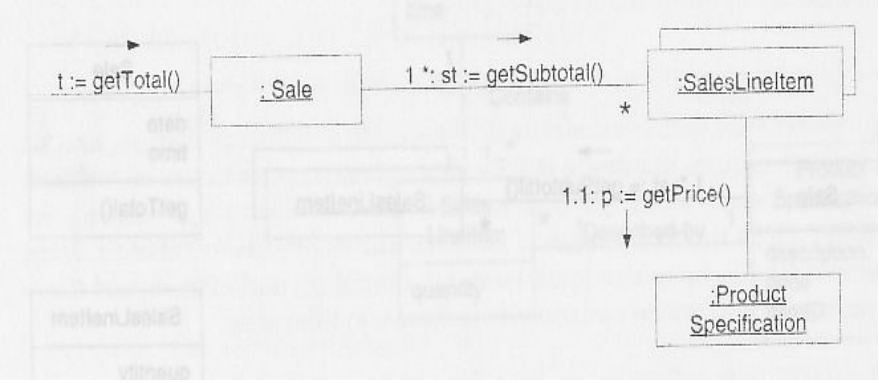
The problem here is knowing the grand total of a sale. According to the principles of Information Expert

1. If there are relevant classes in the Design Model, look there first.
2. Else, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes.

It is necessary to know about all the *SalesLineItem* instances of a sale and the sum of their subtotals. A Sale instance contains these; therefore, by the guideline of Information Expert, Sale is a suitable class of object for this responsibility; it is an information expert for the work.

SalesLineItem.quantity and *ProductSpecification*.price are also needed to determine the line item subtotal. The former know its quantity and its associated *ProductSspecification*; therefore, by Expert, *SalesLineItem* should determine the subtotal; it is the information expert.

Now, the solution can be represented by mean of the interaction design as shown below.



5.1.1.3. Benefits

Information encapsulation is maintained, since objects use their own information to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems. Behavior is distributed across the classes that have the required information, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain. High cohesion is usually supported.

5.1.1.4. Contraindications

There are situations where a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion.

5.1.2. Creator

Creator guides assigning responsibilities related to the creation of objects. The basic intent of the Creator pattern is to find a creator that needs to be connected to the created object in any event.

The concept of aggregation is used in considering the Creator pattern. Aggregation involves things that are in a strong Whole-Part or Assembly-Part relationship, such as Body aggregates Leg or Paragraph aggregates Sentence. *Aggregate aggregates Part*, *Container contains Contents*, and *Recorder records Recorded* are all very common relationships between classes in a class diagram. The only guideline of the Creator is that the enclosing container or recorder class is a good candidate for the responsibility of creating the thing contained or recorded.

Sometimes a creator is found by looking for the class that has the initializing data that will be passed in during creation. This is actually an example of the Expert pattern. Initializing data is passed in during creation via some kind of initialization method, such as a Java constructor that has parameters. For example, assume that a Payment instance needs to be initialized, when created, with the Sale total. Since Sale knows the total, Sale is a candidate creator of the Payment.

5.1.2.1. Problem/Solution pair

Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B aggregates A objects.
- B contains A objects.
- B records instances of A objects.
- B closely uses A objects,
- B has the initializing data that will be passed to A when it is created (thus B is an Expert with respect to creating A).

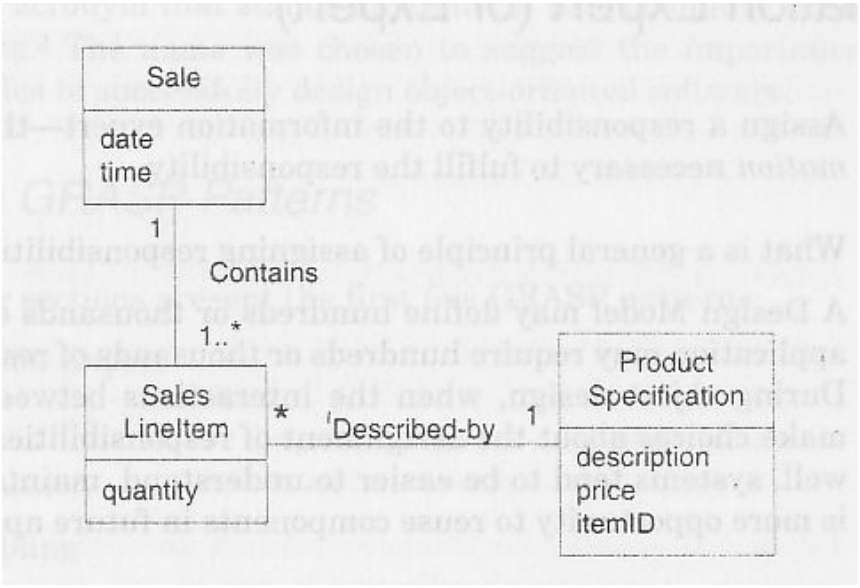
B is a creator of A objects. If more than one option applies, prefer a class B that aggregates or contains class A.

Problem: Who should be responsible for creating a new instance of some class?

The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation and reusability.

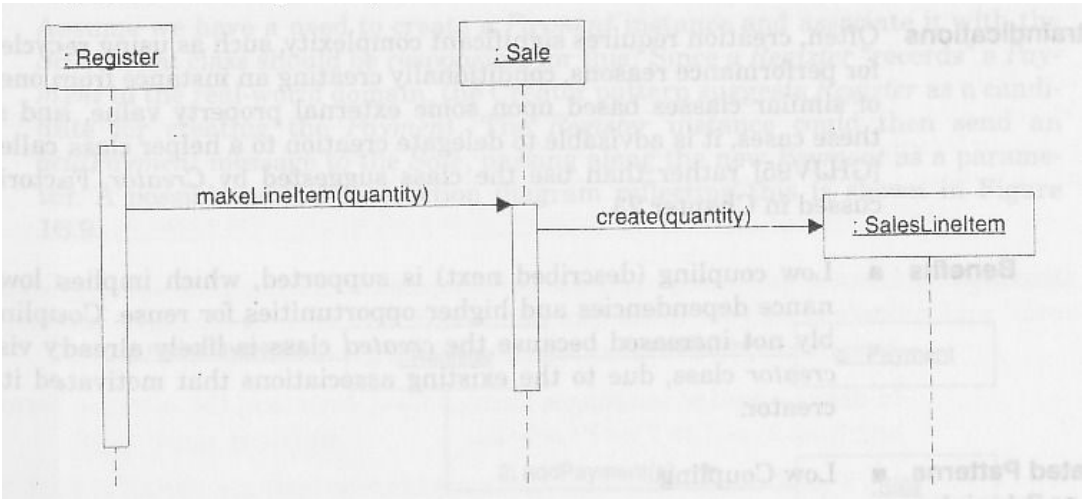
5.1.2.2. Example:

Consider the partial domain model in the figure given below.



Who should be responsible for creating a *SalesLineItem* instance? By Creator, we should look for a class that aggregates, contains, and so on, *SalesLineItem* instances. Since a Sale contains (in fact, aggregates) many *SalesLineItem* objects, the Creator pattern suggests that Sale is a good candidate to have the responsibility of creating *SalesLineItem* instances.

This leads to a design of object interactions as shown in the figure below.



This assignment of responsibilities requires that a *makeLineItem* method be defined in Sale. The method section of a class diagram can then summarize the responsibility assignment results, concretely realized as methods.

5.1.2.3. Benefits

Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse. Coupling is probably not increased because the created class is likely already visible to the creator class, due to the existing associations that motivated its choice as creator.

5.1.2.4. Contraindications

Often, creation requires significant complexity, such as using recycled instances for performance reasons, conditionally creating an instance from one of a family of similar classes based upon some external property value, and so forth. In these cases, it is advisable to delegate creation to a helper class called a Factory rather than use the class suggested by Creator.

5.1.3. Low Coupling

Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an evaluative principle that a designer applies while evaluating all design decisions.

In object-oriented languages such as C++. Java, and C common forms of coupling from TypeX to TypeY include:

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or Y itself
- A TypeX object calls on services of a Type object.
- TypeX has a method that references an instance of TypeY or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY
- TypeY is an interface, and TypeX implements that interface.

Low Coupling encourages assigning a responsibility so that its placement does not increase the coupling to such a level that it leads to the negative results that high coupling can produce. Low Coupling supports the design of classes that are more independent, which reduces the impact of change. A subclass is strongly coupled to its superclass. The decision to derive from a superclass needs to be carefully considered since it is such a strong form of coupling.

For example, suppose that objects need to be stored persistently in a relational or object database. In this case it is a relatively common design to create an abstract superclass called PersistentObjeet from which other classes derive. The disadvantage of this subclassing is that it highly couples domain objects to a particular technical service and mixes different architectural concerns, whereas the advantage is automatic inheritance of persistence behavior.

The extreme case of Low Coupling is when there is no coupling between classes. This is not desirable because a central metaphor of object technology is a system of connected objects that communicate via messages. If Low Coupling is taken to excess, it yields a poor design because it leads to a few incohesive, bloated, and complex active objects that do all the work, with many very passive zero-coupled objects that act as simple data repositories. Some moderate degree of coupling between classes is normal and necessary to create an object-oriented system in which tasks are fulfilled by a collaboration between connected objects.

5.1.3.1. Problem/Solution pair

Solution: Assign a responsibility so that coupling remains low.

Problem: How to support low dependency, low change impact, and increased reuse?

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements.. These elements include classes, subsystems, systems, and so on.

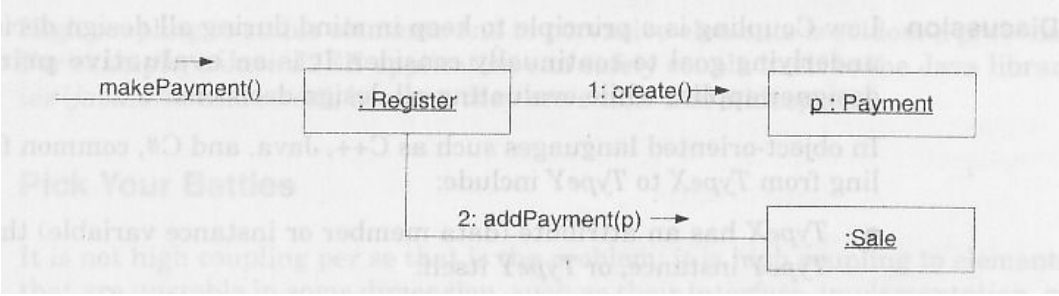
A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:

- Changes in related classes force local changes.
- Harder to understand in isolation.
- Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

5.1.3.2. Example

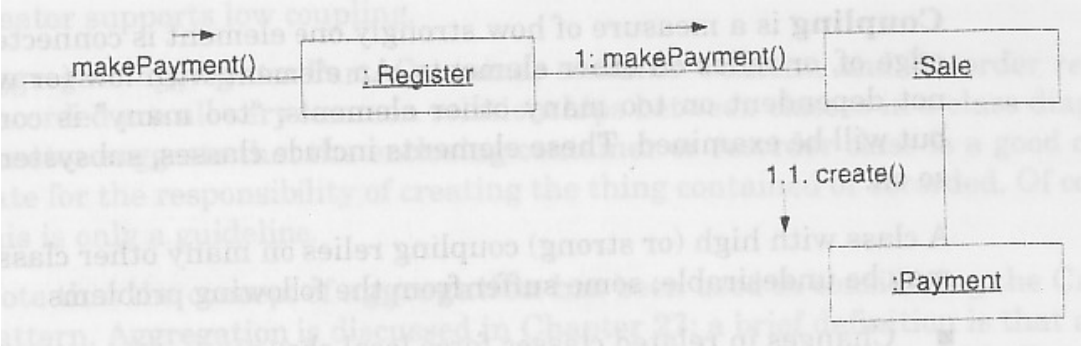
Consider the partial domain containing classes **Payment**, **Register** and **Sale** from the previous example. Assume we have a need to create a Payment instance and associate it with the Sale. What class should be responsible for this? Since a Register "records" a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment. The Register instance could then send an *addPayment* message to the Sale, passing along the new Payment as a parameter.

A possible partial interaction diagram reflecting this is shown in the figure below.



This assignment of responsibilities couples the Register class to knowledge of the Payment class.

An alternative solution to creating the Payment and associating it with the Sale is shown below.



In both cases we will assume the Sale must eventually be coupled to knowledge of a Payment, Design 1, in which the Register creates the Payment, adds coupling of Register to Payment, while Design 2, in which the Sale does the creation of a Payment, does not increase the coupling. Purely from the point of view of coupling, Design 2 is preferable because overall lower coupling is maintained. This an example where two patterns - Low Coupling and Creator - may suggest different solutions.

5.1.3.3. Contraindication

High coupling to stable elements and to pervasive elements is seldom a problem, For example, a Java J2EE application can safely couple itself to the Java libraries (java.util, and so on), because they are stable and widespread.

5.1.3.4. Benefits

- not affected by changes in other components
- simple to understand in isolation
- convenient to reuse

5.1.4. High Cohesion

Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an evaluative principle that a designer applies while evaluating all design decisions.

Grady Booch describes high functional cohesion as existing when the elements of a component (such as a class) ***"all work together to provide some well-bounded behavior."***

Here are some scenarios that illustrate varying degrees of functional cohesion:

1. **Very low cohesion:** A class is solely responsible for many things in very different functional areas. Assume there exists a class called *RDB-RPC-Interface*, which is completely responsible for interacting with relational databases and for handling remote procedure calls. These are two vastly different functional areas, and each requires lots of supporting code. The responsibilities should be split into a family of classes related to RDB access and a family related to RPC support.
2. **Low cohesion**: A class has sole responsibility for complex task in one functional area. Assume that there exists a class called *RDBInterface*, which is completely responsible for interacting with relational databases. The methods of the class are all related, but there are lots of them, and a tremendous amount of supporting code; there may be hundreds of thousands of methods. The class should split into a family of lightweight classes sharing the work to provide RDB access.
3. **High cohesion:** A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks. Assume that there exists a class called *RDBInterface*, which is only partially responsible for interacting with relational databases. It interacts with a dozen other classes related to RDB access in order to retrieve and save objects.
4. **Moderate cohesion**: A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept, but not to each other. Assume that there exists a class called Company, which is completely responsible for (a) knowing its employees and (b) knowing its financial information. These two areas are not strongly related to each other, although both are logically related to the concept of a company: In addition, the total number of public methods is small, as is the amount of supporting code.

As a rule of thumb, a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.

5.1.4.1. Problem/Solution pair

Solution: Assign a responsibility so that cohesion remains high.

Problem: How to keep complexity manageable?

In terms of object design, cohesion (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and soon.

A class with low cohesion does many unrelated things, or does too much work. Such classes are undesirable; they suffer from the following problems:

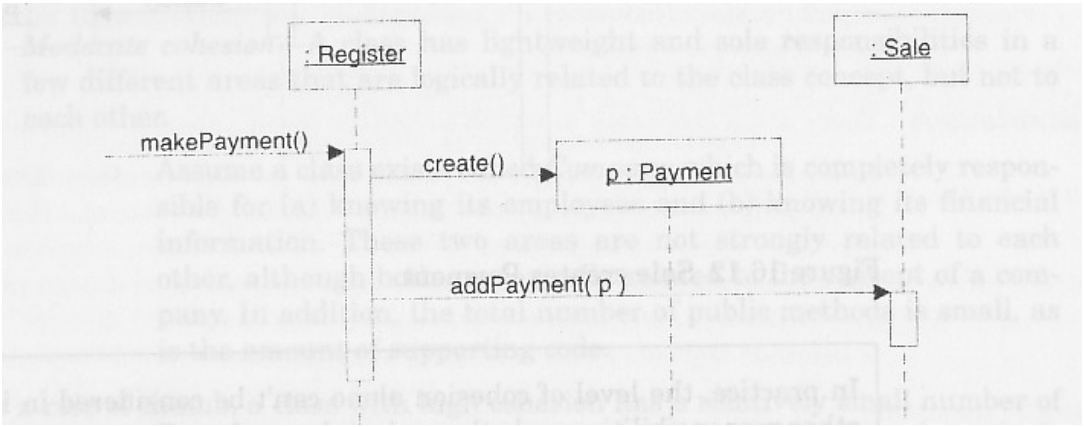
- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly effected by change

Low cohesion classes often represent a very 'large grain" of abstraction, or have taken on responsibilities that should have been delegated to other objects.

5.1.4.2. Example

The same example problem used for the Low Coupling pattern can be analyzed for High Cohesion,

Assume we have a need to create a (cash) Payment instance and associate it with the Sale. What class should be responsible for this? Since Register records a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment. The Register instance could then send an *addPayment* message to the Sale, passing along the new Payment as a parameter, as shown in the figure below:



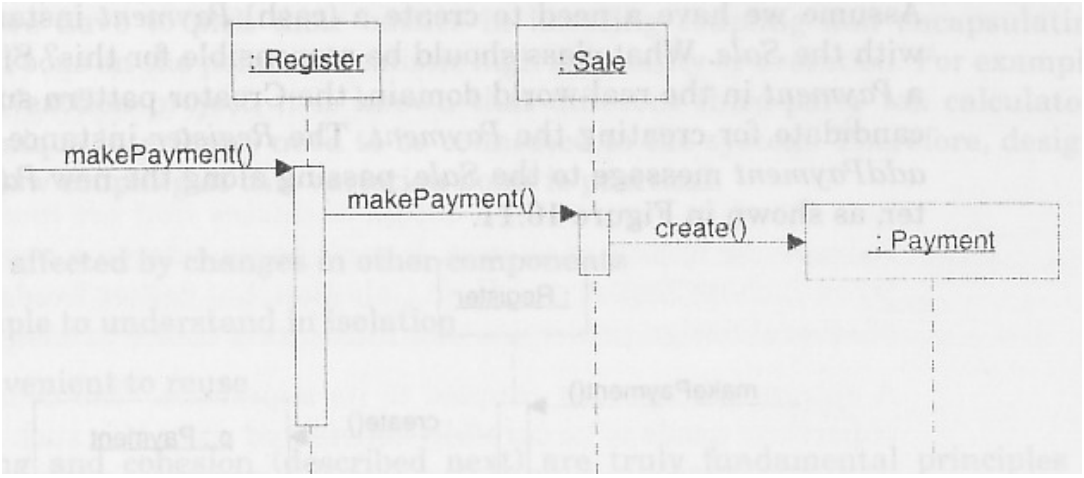
This assignment of responsibilities places the responsibility for making a payment in the Register. The Register is taking on part of the responsibility for fulfilling the *makePayment* system operation.

In this isolated example, this is acceptable; but if we continue to make the Register class responsible for doing some or most of the work related to more and more system operations, it will become increasingly burdened with tasks and become incohesive.

Imagine that there were fifty system operations, all received by Register. If it did the work related to each, it would become a "bloated" incohesive object. The point is not that this single Payment creation task in itself makes the Register incohesive, but as part of a larger picture of overall responsibility assignment, it may suggest a trend toward low cohesion.

And most important in terms of developing skills as an object designer, regardless of the final design choice, the valuable thing is that at least a developer knows to consider the impact on cohesion.

Since the second design (shown graphically below) supports both high cohesion and low coupling, it is desirable.



5.1.4.3. Contraindications

There are a few cases in which accepting lower cohesion is justified.

One case is the grouping of responsibilities or code into one class or component to simplify maintenance by one person (although be warned that such grouping may also make maintenance worse). But for example, suppose an application contains embedded SQL statements that by other good design principles should be distributed across ten classes, such as ten "database mapper" classes. Now, it is common that only one or two SQL experts know how to best define and maintain this SQL even if there are dozens of object-oriented programmers on the project. The software architect may decide to group all the SQL statements into one class, RDBOperations, so that it is easy for the SQL expert to work on the SQL in one location.

Another case for components with lower cohesion is with distributed server objects. Because of overhead and performance implications associated with remote objects and remote communication, it is sometimes desirable to create fewer and larger, less cohesive server objects that provide an interface for many operations. This is also related to the pattern called **Coarse-Grained Remote Interface**, in which the remote operations are made more coarse-grained in order to do or request more work in remote operation call, because of the performance penalty of remote calls over a network. As a simple example, instead of a remote object with three fine-grained operations *setName*, *setSalary* and *setHireDate*, there is one remote operation *setData*, which receives a set of data. This results in less remote calls, and better performance.

5.1.4.4. Benefits

- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- The fine grain of highly related functionality supports increased reuse because a cohesive class can be used for a very specific purpose.

5.1.5. Controller

Systems receive external input events, typically involving a GUI operated by a person. Other mediums of input include external messages such as in a call processing telecommunications switch, or signals from sensors such as in process control systems.

In all cases, if an object design is used, some handler for these events must be chosen. The Controller pattern provides guidance for generally accepted, suitable choices. The controller is a kind of facade into the domain layer from the interface layer. Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work

The first category of controller is a facade controller representing the overall system, device, or a subsystem. The idea is to choose some class name that suggests a cover, or facade, over the other layers of the application, and that provides the main point of service calls from the UI layer down to other layers. It could be an abstraction of the overall physical unit, a class representing the entire software system, a sub system or any other concept that the designer chooses to represent the overall system.

If a use-case controller is chosen, then there is a different controller for each use case. Note that this is not a domain object; it is an artificial construct to support the system (a Pure Fabrication in terms of the GRASP patterns). For example, if an application contains use cases such as Process Sale and Handle Returns, then there maybe a *ProcessSaleHandler* class and so forth. The use case controller is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling, typically when the facade controller is becoming 'bloated' with

excessive responsibilities. A use-case controller is a good choice when there are many system events across different processes; it factors their handling into manageable separate classes, and also provides a basis for knowing and reasoning about the state of the current scenario in progress.

5.1.5.1. Problem/Solution pair

Solution: Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- Represents the overall system, device, or subsystem (**facade controller**).
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <UseCaseName>Session (**use-case or session controller**).
  - Use the same controller class for all system events in the same use case scenario.
  - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length, but are often organized in terms of use cases (use case sessions).

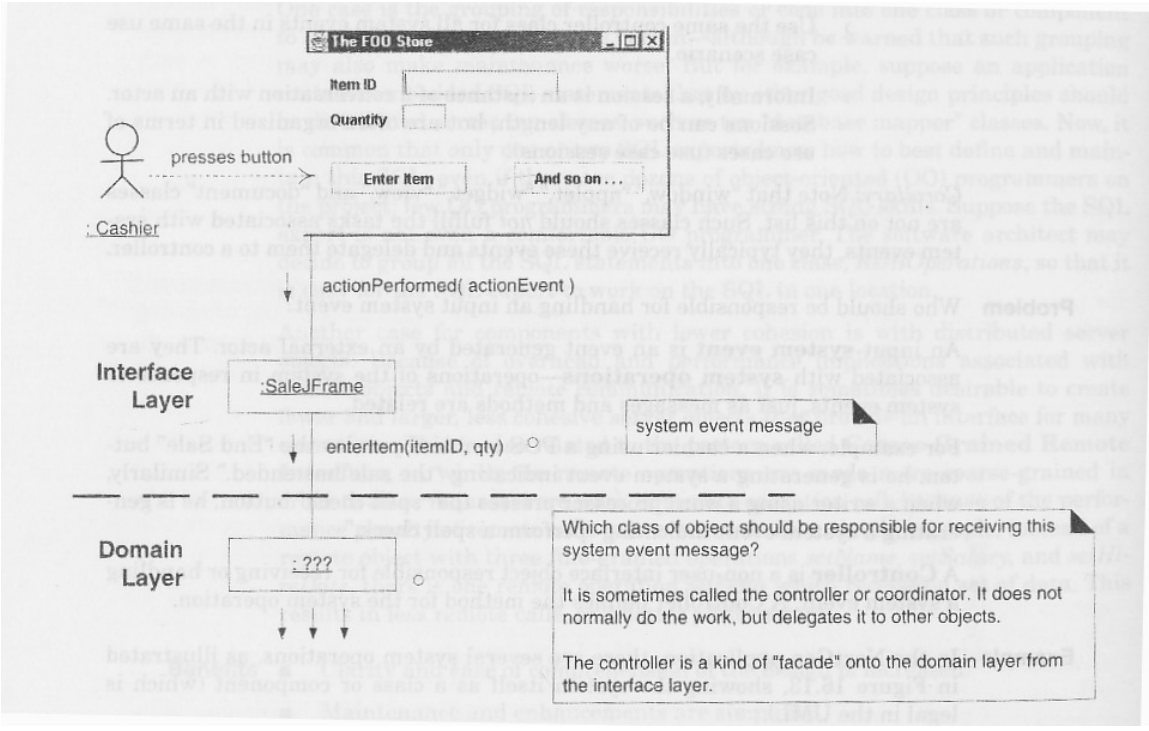
Problem: Who should be responsible for handling an input system event?

An input **system event** is an event generated by an external actor. They are associated with **system operations** - operations of the system in response to system events, just as messages and methods are related.

For example, when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended." Similarly, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.

5.1.5.2. Example

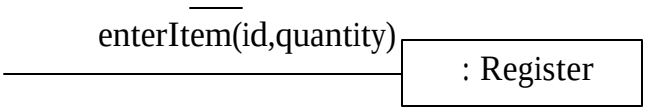


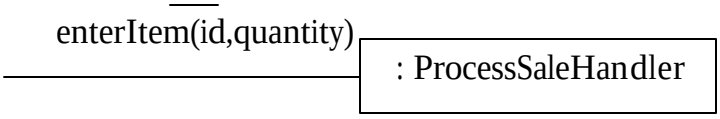
Consider an application, where several system operations are there e.g. enterItem(), maleNewSale, makePayment(), endSale() etc. During analysis, system operations may he assigned to the class System, to indicate they are system operations. However, this does not mean that a soft ware class named System fulfills them during design. Rather, during design, a Controller class is assigned the responsibility for system operations

Who should be the controller for system events such as *enterItem* and *endSale*? By the Controller pattern, here are some choices:

Register, POSSystem	represent the overall system," device, or subsystem
ProcessSaleHandler, ProcessSaleSession	represent a receiver or handler of all system events of a use ease scenario

In terms of interaction diagrams, it means that one of the examples in the figure given below may be useful.





The choice of which of these classes is the most appropriate controller is influenced by other factors. During design, the system operations identified during system behavior analysis are assigned to one or more controller classes, such as Register, as shown in the figure.

5.1.5.3. Benefits

Increased potential for reuse, and pluggable interfaces. It ensures that application logic is not handled in the interface layer. The responsibilities of a controller could technically be handled in an interface object, but the implication of such a design is that program code and logic related the fulfillment of application logic would be embedded in interface or window objects. An interface-as-controller design reduces the opportunity to reuse logic in future applications, since it is bound to a particular interface (for example, window-like objects) that is seldom applicable in other applications. By contrast, delegating a system operation responsibility to a controller supports the reuse of the logic in future applications. And since the application logic is not bound to the interface layer, it can be replaced with a different interface.

Reason about the state of the use case. It is sometimes necessary to ensure that system operations occur in a legal sequence. or to be able to reason about the current state of activity and operations within the use case that is underway. For example, it maybe necessary to guarantee that the *makePayment* operation can not occur until the *endSale* operation has occurred. If so, this state information needs to be captured somewhere; the controller is one reasonable choice, especially if the same controller is used throughout the use case (which is recommended).

Source:

- Craig Larman, *Applying UML and Patterns*