Design: Assigning Responsibilities to Objects
Use-Case Realization

The Micro Development Process (by Grady Booch) (*)

The following four steps start with analysis and continue with the design.

1. **Identifying Elements (Classes and Objects)**
   - Abstractions that form the vocabulary of the problem domain are discovered: What is and what is not of interest?
   - Product: Dictionary (list of things) consisting of all significant classes and objects, using meaningful names that imply their semantics.

   As development proceeds, the dictionary grows.
2. **Defining Element Collaborations and Responsibilities**
   - The purpose is to describe how the identified elements work together to provide the system's behavioral requirements.
   - We refine the identified elements through the distribution of responsibilities.
   - Assignment of responsibilities, separation of concerns

(*) Grady Booch, Robert A. Maksimchuk, Michael W. Engle, "**Object-oriented analysis and design with applications**", (3rd Edition), Addison-Wesley, 2007.

3. **Defining Element Relationships**
   - The associations among classes and objects (including specific inheritance (is-a) and aggregation (has-a) relationships are specified.
   - Defining the element relationships establishes the shape of the solution.
4. **Detailing Element Semantics**
   - The detailed internal structure of the elements
   - Attributes and algorithms that provide the semantics (responsibilities) of the elements (classes and objects) we identified earlier.

The Macro Development Process (by Grady Booch)

The overall software development lifecycle, the controlling framework for the micro process.

Activities of the entire development team on the scale of weeks to months.

- Requirements
- Analysis and design
- Implementation
- Test
- Deployment

In this course we focus on the Micro Development Process.

Steps of Design (See the Figure in 4.4)

1. Identify responsibilities from use cases (and operation contracts).
2. Search for proper classes to assign the responsibilities.
   First search in the set of previously designed software classes.
   If there is no proper software class, search in the domain model.
   Take a conceptual class from the domain model (real-world), then create a software class with the same name and assign responsibility to this class.
3. Use design principles and patterns to make your decisions.
4. Express your design using UML class diagrams and interaction (sequence, communication) diagrams.

Responsibilities of objects: knowing and doing

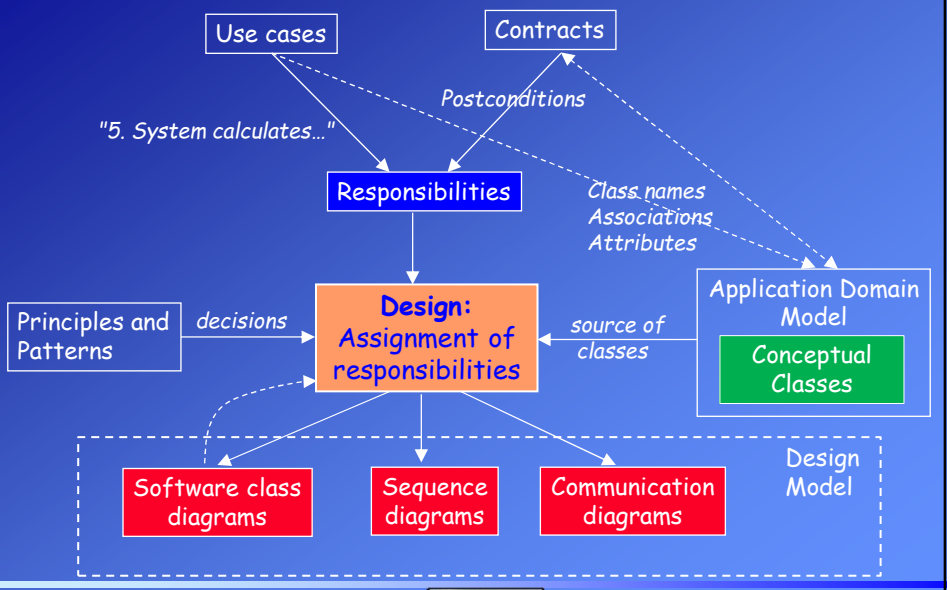
Doing responsibilities:

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

Knowing the responsibilities of an object include:

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

Components of the design:



Design Principles and Design Patterns

Design principles and software design patterns are used as guidelines for making decisions at the design level.

Design principles are basic advice about object-oriented design.

For example;

"Model-view separation",
"Favor composition over inheritance",
"Assign responsibilities so that coupling remains low".

A **software design pattern** is a named and well-known problem/solution pair that can be applied in new contexts.

Patterns describe solutions discovered by experienced software developers for common problems in software design.

In this course, first, we will see GRASP patterns, which are proposed by Larman.

After GRASP, we will discuss popular GoF (Gang of Four) design patterns, which are widely used.

Design with GRASP

GRASP (*General Responsibility Assignment Software Patterns*) is a collection of some principles and basic patterns.

It is composed by Craig Larman * as a learning aid.

However, they also form a good starting point for industrial software projects.

There are 9 GRASP patterns:

1. Controller
2. Creator
3. Information Expert
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

* Craig Larman, Applying UML and Patterns , An Introduction to OOA/D and Iterative Development, 3/e, 2005.

Controller (GRASP)

The Controller pattern depends on the **Model-View Separation Principle.**

Model-View Separation Principle:

- Do not connect or couple non-UI objects (business layer objects) directly to UI (user interface) objects.
- Do not put application logic (such as a tax calculation) in the UI object methods.

  UI objects should only initialize UI elements, receive UI events (such as a mouse click on a button), and delegate requests for application logic to non-UI objects (such as domain objects).

The motivation for Model-View Separation includes:

- To allow separate development of the model and user interface layers.
- To minimize the impact of requirements changes in the interface upon the domain layer.
- To allow multiple simultaneous views on the same model object.
- To allow execution of the model layer independent of the user interface layer, such as in a message-processing or batch-mode system.
- To allow easy porting of the model layer to another user interface framework.

Controller Pattern:

Problem: What first object beyond the UI layer receives and coordinates ("controls") a system operation? (See 4.9)

Solution: (advice)

Place a controller object between two layers.

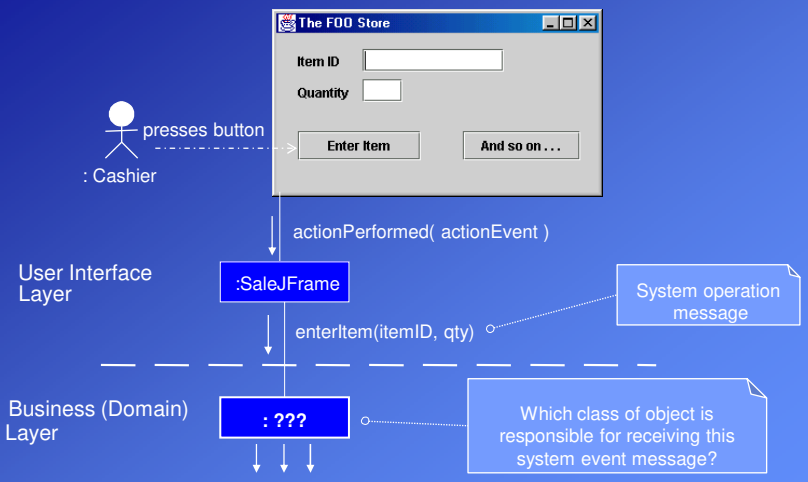
This object will receive messages from one layer and delegate them to a proper object in the other layer.

Assign the responsibility to an object representing one of these choices:

a. **Facade Controller:** Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a facade controller).

b. **Session Controller:** Represents a use case scenario within which the system operation occurs (a use case or session controller).

Problem: What first object beyond the UI layer receives and coordinates ("controls") a system operation?

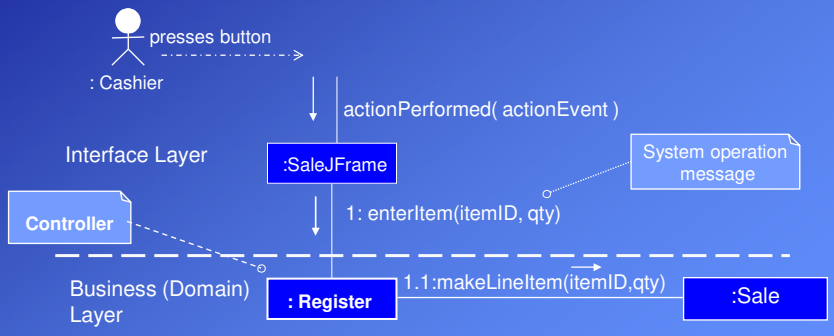


Solution: Controller

The controller can be a real-world object (low representational gap) or an artificial object.

Note that in the domain of POS, a Register (called a POS Terminal) receives all system inputs from the cashier.

An object of a **Register** can be the controller in our system.



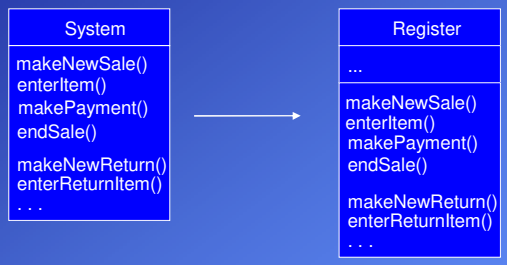
The controller is a kind of "facade" on the domain layer from the UI layer.

The controller does not perform the operation; it only delegates it.

Two types of controllers:

Facade Controller:

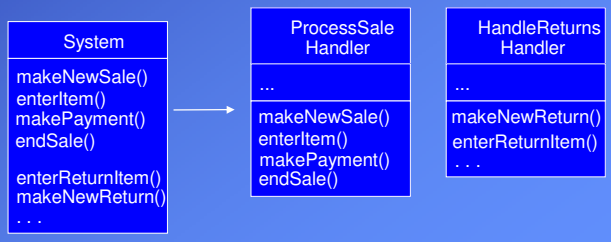
All system operations are assigned to one controller.



Session Controller:

There is a separate controller for each use case.

Choosing a use case controller is suitable when we have many system operations and wish to distribute responsibilities to keep each controller class lightweight and focused.



Creator (GRASP)

One of the first problems you will face in OO design is: Who creates object X?

The creation of objects is one of the most common activities in an object-oriented system.

If the responsibility is assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.

Creator pattern:

Problem:

Who should be responsible for creating a new instance (object) of some class?

Solution:

Assign class B the responsibility to create an instance of class A if one of these is true:

- B "contains" or compositely aggregates A.
- B records A.
- B closely uses A.
- B has the initializing data for A that will be passed to A when it is created.

Later, we will see the *Factory (GoF)* pattern that provides a detailed solution to the problem of creating objects.

Design Example: Starting a new sale, makeNewSale

Assume that we have written an operation contract for the makeNewSale operation.

Actually, "make a new sale" (or "start a new sale") is a simple operation and responsibilities regarding this operation can also be defined without contracts.

However, to be familiar with responsibilities, we make our first designs using operation contracts.

Contract CO1: makeNewSale

Operation:	makeNewSale()
Cross References:	Use Cases: Process Sale
Preconditions:	none
Postconditions:	- A Sale instance s was created (instance creation). - s was associated with the Register (association formed). - Attributes of s were initialized (attribute modification).

1. Finding responsibilities:

Postconditions give us the responsibilities.

- Who will create the object s of class Sale?
- Who will associate s with Register?
- Who will initialize s?
- If we haven't chosen the controller yet, we must decide "who will get the makenewSale operation and delegate it".

Example: makeNewSale (cont'd)

2. Assigning responsibilities:

To assign these responsibilities, we will first search in the set of design (software) classes.

Assume that we are at the beginning of the design; therefore, there is no software class.

In this case, we will look at the domain model.

- Controller:

When we analyze our POS system, we see that all system operations are entered via the POS terminal (register).

Therefore, choosing a real-world, device-object facade controller like Register is satisfactory if there are only a few system operations and the facade controller is not taking on too many responsibilities.

- Creating the Sale and associating it with the Register:

The Domain Model shows that a Register records a Sale;

Thus, Register is a reasonable candidate for creating a Sale.

By having the Register create the Sale, we can easily associate the Register because the Register will have a reference to the current Sale instance.

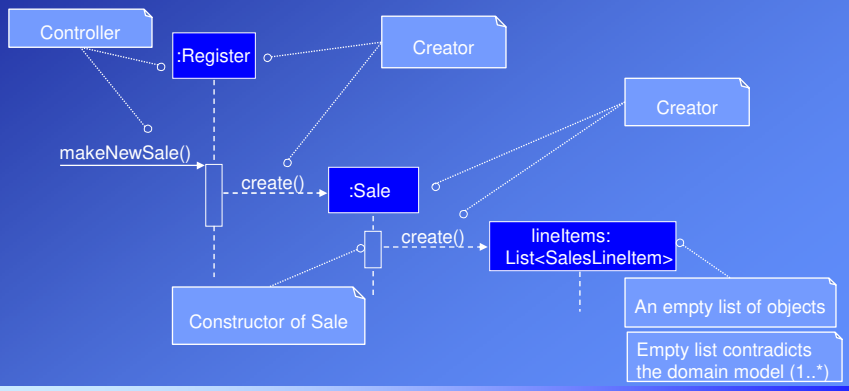
Example: makeNewSale (cont'd)

- Initializing the Sale:

When the Sale is created, it must create an empty collection (such as a List) to record all the future SalesLineItem instances that will be added.

3. Visualizing the design as UML class and interaction diagrams:

Design: makeNewSale



Information Expert (or Expert) (GRASP)

Problem:
What is the general principle of assigning responsibilities to objects?

Solution:
Assign responsibility to the information expert, the class, with the information necessary to fulfill the responsibility.

It is a fundamental guiding principle of object design.
It expresses the common "intuition" that objects do things related to the information they have.

Design Example: Calculating the grand total of a sale

From the use case "UC1 Process Sale":

5. System presents the total with taxes calculated.

Because of the Model-View Separation principle, we are not concerned with how the sale total will be displayed (UI), but we must ensure that the total is known.

Besides, we do not consider the calculation of taxes in this iteration.

The responsibility:

Who should be responsible for knowing the grand total of a sale?

Solution: Who should be responsible for knowing the grand total of a sale?

Where to find software classes?

Remember: Firstly, look at the Design Model; assign the responsibility if there is a relevant software class.

Otherwise, look at the Domain Model and use it to inspire the creation of corresponding design (software) classes.

In this example, we assume that we are just starting design work, and there is no, or a minimal, Design Model.

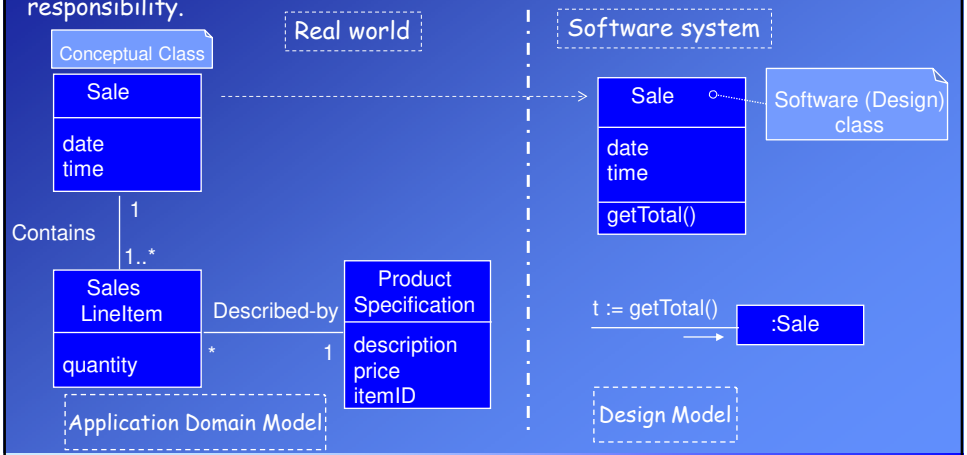
Therefore, we look directly to the Domain Model (3.17) for information experts.

Solution: Calculating the grand total of a sale (contd)

We need to know about all the SalesLineItem instances of a sale and the sum of their subtotals.

As per the Domain Model (3.17), A Sale instance contains them.

Following the Information Expert guideline, Sale is an appropriate class for this responsibility.



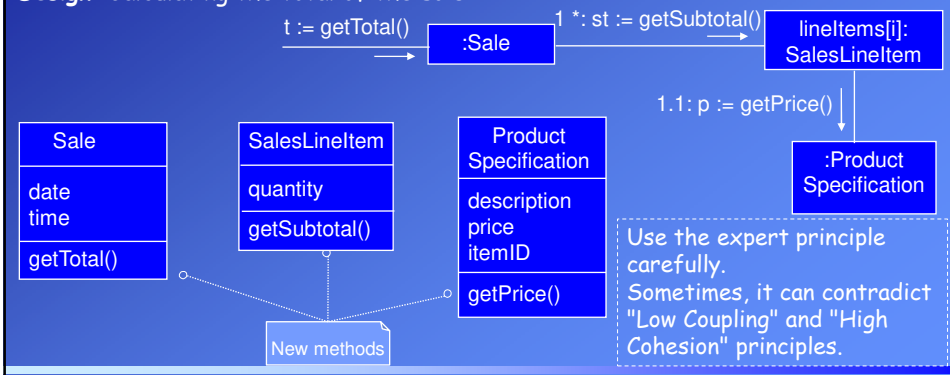
Delegation of responsibilities to other classes:

Sale cannot calculate the total by itself.

We need to determine the subtotal of SalesLineItems.

- The SalesLineItem knows its quantity and its associated ProductDescription; therefore, by Expert, SalesLineItem should determine the subtotal;
- The ProductDescription is an information expert on answering its price; therefore, SalesLineItem sends it a message asking for the product price.

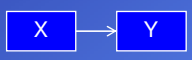
Design: Calculating the total of the Sale



Low Coupling (GRASP)

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 many other elements.

Forms of coupling from class X to class Y : 

- X has an attribute (reference or instance variable) of type Y.
- An X object calls on services (methods) of a Y object.
- X has a method that references an instance of Y. It means a method of X includes a parameter, local variable, or return value of type Y.
- X is a direct or indirect subclass of Y.
- Y is an interface, and X implements that interface (Java).

A class with high (or strong) coupling is not desirable because

- Changes in other classes (Y) affect the class (X),
- Harder to understand (X) in isolation,
- Harder to reuse (X) because its use requires the additional presence of the classes on which it depends (Y).

Low Coupling pattern:

Problem:

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

Solution:

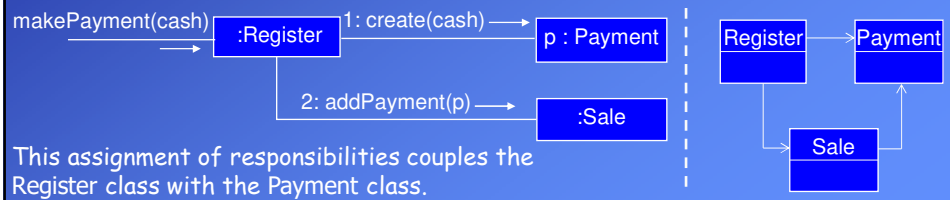
Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

Design Example: Making the Payment, makePayement operation

What class should be responsible for creating a Payment instance and associating it with the Sale?

Solution 1 *(coupling is high):*

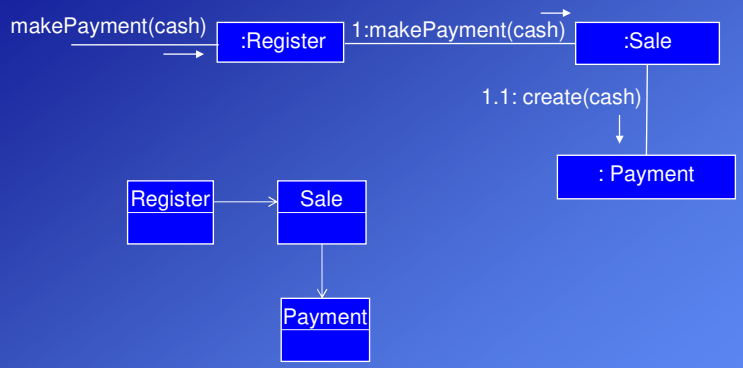
Since a Register "records" a Payment in the real-world domain, the Creator (and also Expert) pattern suggests Register as a candidate for creating the Payment.



This assignment of responsibilities couples the Register class with the Payment class.

***Solution 2** (coupling is lower)* Making the Payment :

- Sale creates the Payment.



With this assignment, the coupling is lower than the first solution.

There is no coupling between Register and Payment.

According to the low coupling pattern, **Solution 2 is better than Solution 1.**

High Cohesion (GRASP)

Cohesion (more specifically, **functional cohesion**) is a measure of how strongly related and focused the responsibilities of a class are.

A class with high cohesion
- has a relatively small number of methods,
- with highly related functionality,
- and does not do too much work.

If the task is large, it collaborates with other objects to share the effort.

A class with low cohesion does many unrelated things or too much work. Such classes (with low cohesion) are not desirable because
- hard to understand,
- hard to reuse,
- hard to maintain,
- affected by many changes.

High Cohesion pattern:

Problem:

How to keep objects focused, understandable, and manageable?

Solution:

Assign a responsibility so that cohesion remains high. Use this principle to evaluate alternatives.

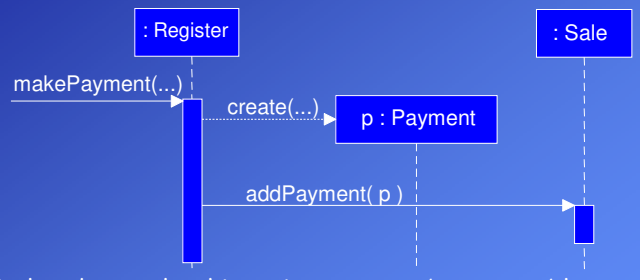
Design Example 1: Making the Payment, makePayement operation

We look at the example used in the "Low Coupling" pattern.

What class should be responsible for creating and associating a Payment instance with the Sale?

Solution 1:

Since a Register "records" a Payment in the real-world domain, the Creator (and also Expert) pattern suggests Register as a candidate for creating the Payment.



In this isolated example, this assignment may be reasonable.
However, since the Register class is a controller, it will become increasingly incohesive if we continue to make it responsible for doing most of the work related to system operations.

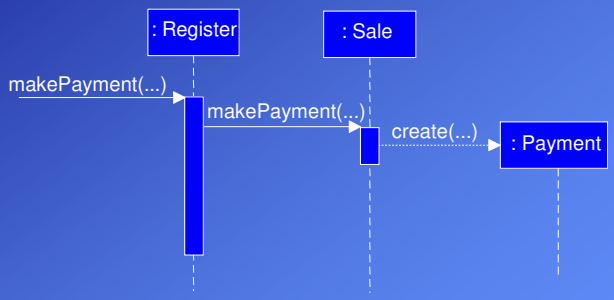
Solution 2:

The single "Payment creation" task does not make the Register incohesive.

However, if there are, for example, fifty system operations, all received by Register, and if Register did the work related to each, it would become a "bloated" incohesive object.

So, it must delegate some of the work.

As a result, the Sale creates the Payment.



Design Example 2: Storing a sale into a database

Who is responsible for writing data of a Sale into the database?

Since Sale is the information expert, we may put methods in this class to handle database operations.

This decision violates "high cohesion" and "separation of concerns" principles.

The Sale class is responsible for the financial operations of a sale.

Database operations should be delegated to another class.

Conclusion:

A real-world analogy: It is a common observation that if a person takes on too many unrelated responsibilities, especially ones that should properly be delegated to others, then this person is ineffective.

Like Low Coupling, High Cohesion is a principle to remember during all design decisions; it is an underlying goal to consider continually.

A highly cohesive class is advantageous because it is relatively easy to maintain, understand, and reuse.

The reusability of fine-grained, highly related functionality increases because a cohesive class can fit into various systems.

Design Principles so far:

- **Low Representational Gap (between real-world and software)**
  This is the main idea in object orientation.
  We take inspiration from the application (real-world) domain in creating software classes.
  Software classes have the same (similar) names as domain classes.
  Software classes have domain-familiar information and responsibilities.
  The aim is to improve the understandability of software.
- **Separation of concerns:** Concerns are related to features of the software.
  For example, UI, data, and business models are different concerns.
  Calculating the total of a sale, credit card operations, and inventory operations are different concerns.
  Do not insert responsibilities about different concerns into the same class.
  The class Sale should not contain methods about UI, database, or inventory.
- **Model-View separation:**
  This principle is a particular case of the "separation of concerns" principle.
  Do not directly connect non-UI objects (business layer objects) to UI objects.
  Do not put application logic (such as a tax calculation) in the UI object methods.

Design Principles so far: (cont'd)

- **Controller** (GRASP): Put a controller object between two layers.
- **Creator** (GRASP): The answer to "Who creates the object X ?".
- **Information expert** (GRASP): Assign responsibility to the class with the information necessary to fulfill the responsibility.
- **Low Coupling** (GRASP): Assign a responsibility so that coupling remains low.
- **High cohesion** (GRASP): A class with high cohesion has a relatively small number of methods with highly related functionality and does not do too much work.
- **Modular Design:** Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.