Applying UML and Patterns

**An Introduction to
Object-oriented Analysis
and Design
and Iterative Development**

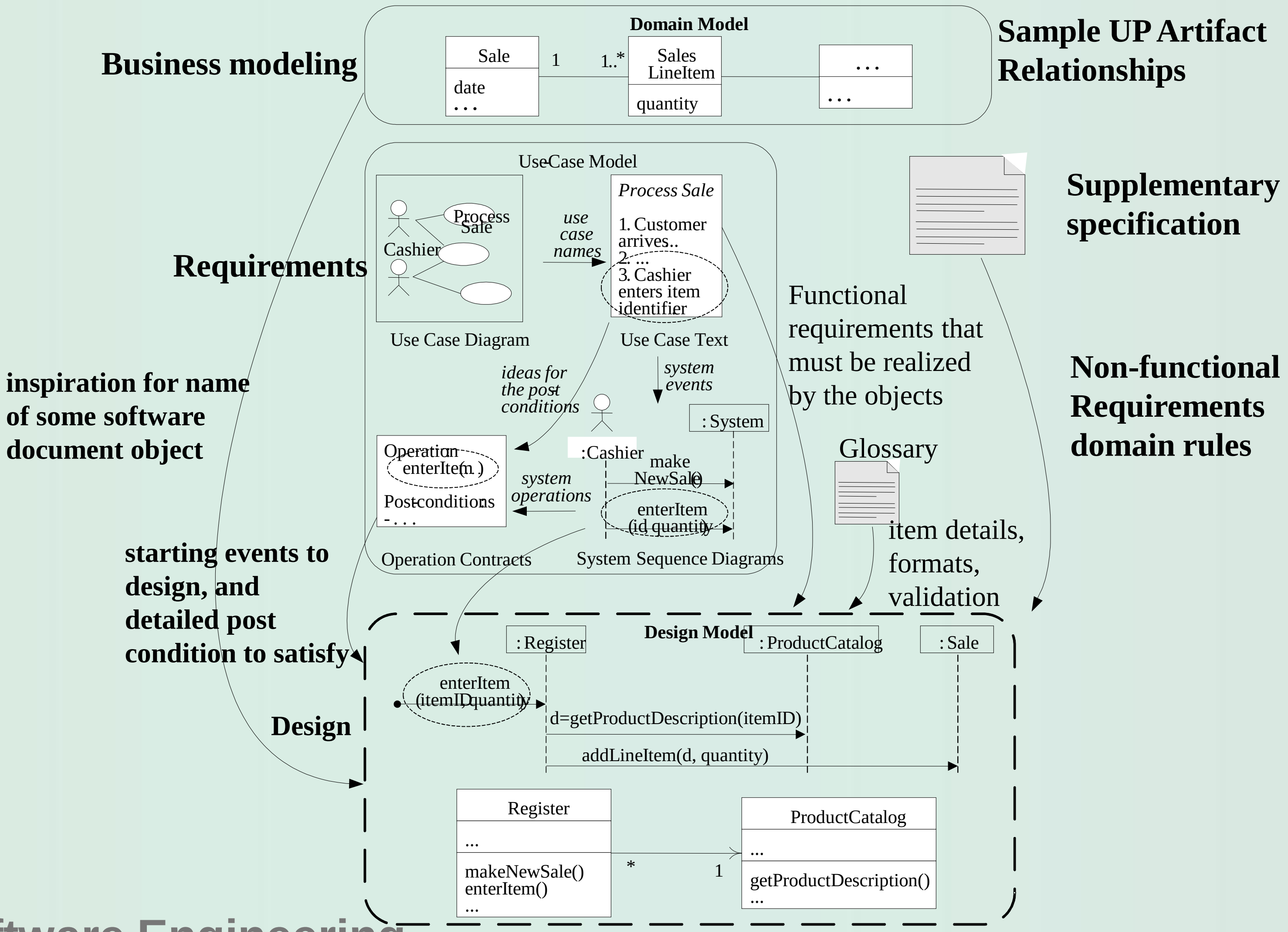
Part III Elaboration Iteration I – Basic[3]

Chap 17
GRASP:
Designing Objects with Responsibilities

Requirements Analysis Scenario

The first two-day requirements workshop is finished.	The chief architect and business agree to implement and test some scenarios of Process Sale in the first three-week timeboxed iteration.
Three of the twenty use cases that are the most architecturally significant and of high business value have been analyzed in detail, including, of course, the Process Sale use case. UP iterative methods recommends, analyzing only 10%~20% of the requirements in detail before starting to program.	Other artifacts have been started: Supplementary Specification, Glossary, and Domain Model.
Programming experiments have resolved the show-stopper technical questions, such as whether a Java Swing UI will work on a touch screen.	The chief architect has drawn some ideas for the large-scale logical architecture, using UML package diagrams. This is part of the UP Design Model.

Inputs of Object Design 1



Inputs of Object Design 2

use case text defines the visible behavior that the software objects must ultimately support objects are designed to implement the use cases. In the UP, this OO design is called the use case realization.	The Supplementary Specification defines the non-functional goals, such as internalization, our objects must satisfy.
The system sequence diagrams identify the system operation messages, which are the starting messages on our interaction diagrams of collaborating objects.	The Glossary clarifies details of parameters or data coming in from the UI layer, data being passed to the database, and detailed item-specific logic or validation requirements, such as the legal formats and validation for product universal product codes.
The operation contracts may complement the use case text to clarify what the software objects must achieve in a system operation. The post-conditions define detailed achievements.	The Domain Model suggests some names and attributes of software domain objects in the domain layer of the software architecture.

Outputs of Object Design

- UML interaction, class, and package diagrams for the difficult parts of the design that we wished to explore before coding
- UI sketches and prototypes
- database models with UML data modeling profile notation.
- report sketches and prototypes

Responsibility-Driven Design 1

- **Responsibility** as contract or obligation of class [UML].
  - Doing responsibilities of an object
    - 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.
  - a Sale is responsible for creating SalesLineItems" (doing)
  - a Sale is responsible for knowing its total (a knowing).

Responsibility-Driven Design 2

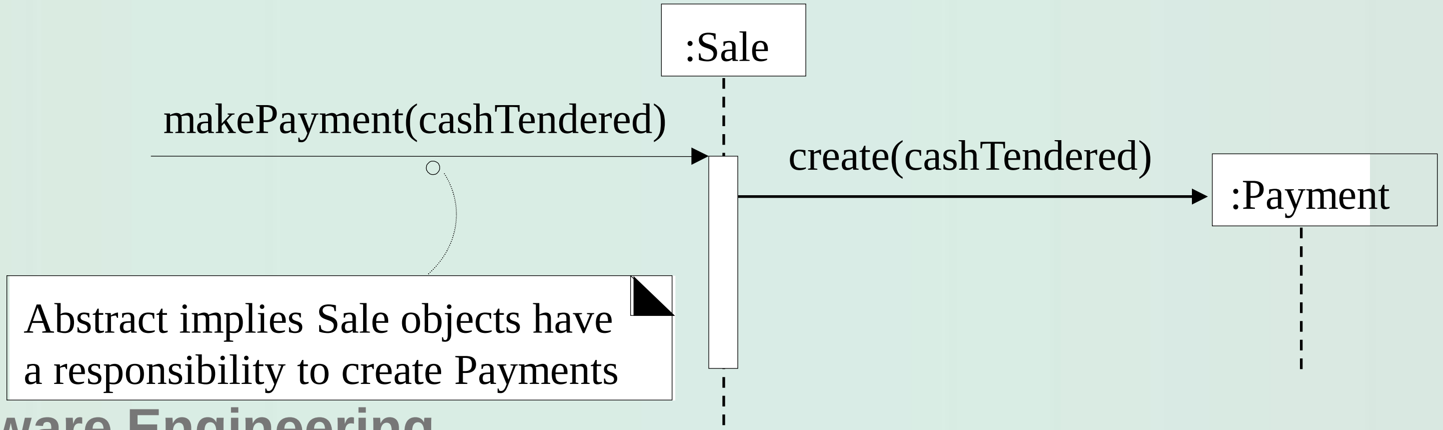
- Guideline: Domain objects illustrate the attributes and associations, relevant responsibilities related to "knowing."
  - if the domain model Sale class has a time attribute, that a software Sale class knows its time.
- Big responsibilities take hundreds of classes and methods. Little responsibilities might take one method.
  - the responsibility to "provide access to relational databases" may involve two hundred classes and thousands of methods.
  - the responsibility to "create a Sale" may involve only one method in one class.

Responsibility-Driven Design 3

- RDD includes the idea of **collaboration**.
    - Responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects.
    - the Sale class might define one or more methods to know its total; getTotal() method fulfill that responsibility, the Sale may collaborate with other objects, such as sending a getSubtotal message to each SalesLineItem object asking for its subtotal.
- RDD is a general metaphor for thinking about OO software design.
    - Think of software objects as similar to people with responsibilities who collaborate with other people to get work done.

Responsibilities/GRASP/UML

- GRASP: General Responsibility Assignment Software Patterns
  - A Learning Aid for OO Design with Responsibilities
  - Assigning responsibilities to objects.
- Drawing UML interaction diagrams realized these responsibilities to methods.
  - Sale objects are given a responsibility to create Payments, which is invoked with a makePayment message and handled with a corresponding makePayment method.


:Sale
makePayment(cashTendered)
create(cashTendered)
:Payment
Abstract implies Sale objects have a responsibility to create Payments

Patterns

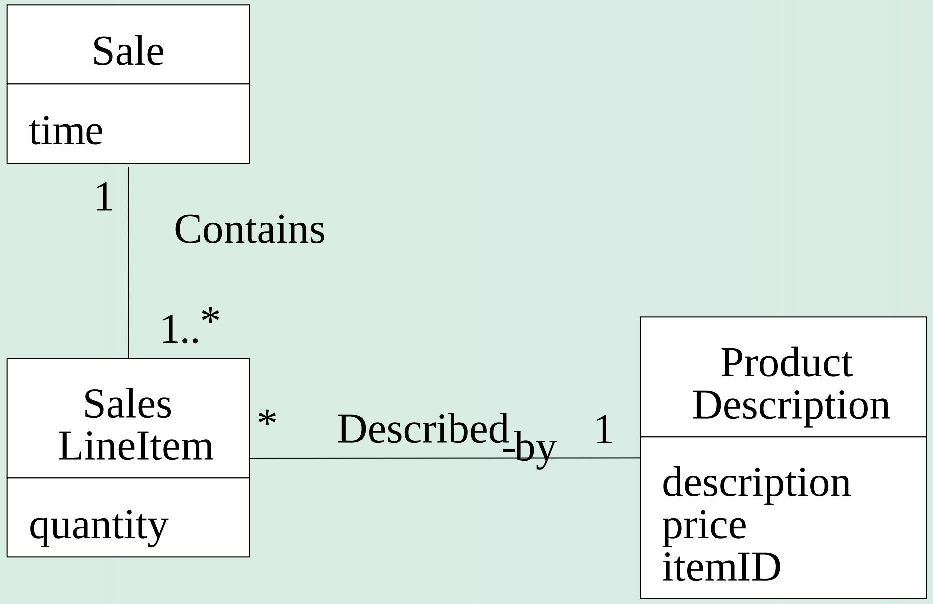
- **Pattern** is a well-known problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussion of its trade-offs, implementations, variations.
  - Pattern Name: Information Expert
  - Problem: What is a basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfill it.
- The Gang-of-Four Design Patterns.
- GRASP defines nine basic OO design principles or basic building blocks in design.
  - One person's pattern is another person's primitive building block.

Creator 1

- Problem
  - Who should be responsible for creating a new instance of some class?
  - Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.
- 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. Thus B is an Expert with respect to creating A.
  - B is a creator of A objects

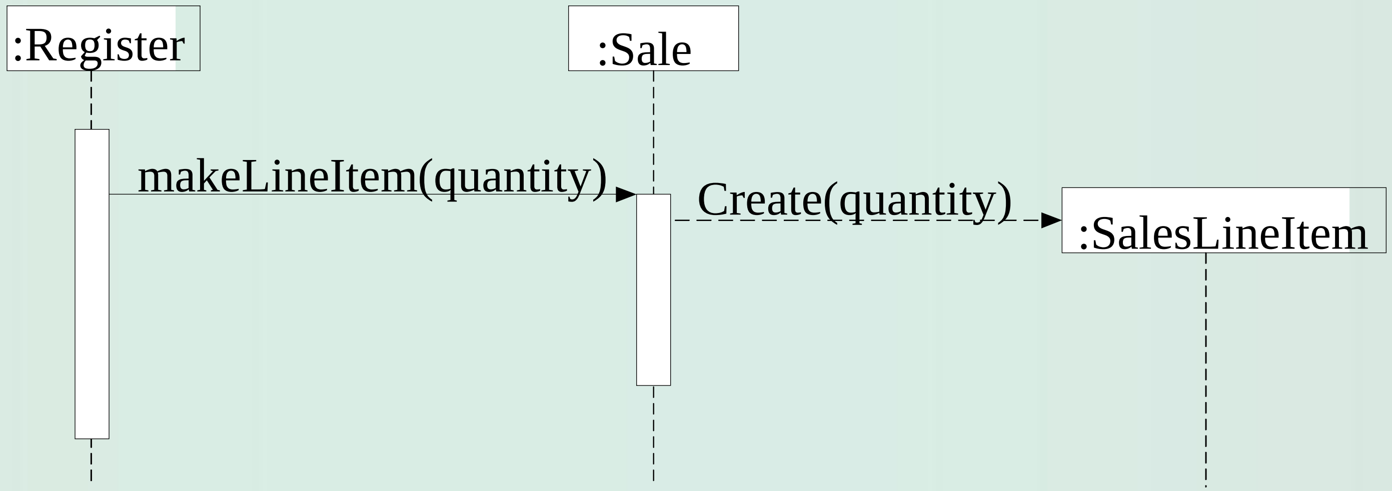
Creator 2

- NextGen POS Example: who should be responsible for creating a SalesLineItem instance?
  - By Creator, look for a class that aggregates, contains SalesLineItem instances.



Creator 3

- A Sale contains (aggregates) many SalesLineItem objects
  - Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.
  - This leads to the design of object interactions shown
  - This assignment of responsibilities requires that a makeLineItem method be defined in Sale



Creator 4

- Discussion
  - Expert pattern: Initializing data is passed in during creation via some kind of initialization method, such as a Java constructor that has parameters.
  - assume that a Payment instance, when created, needs to be initialized with the Sale total. Since Sale knows the total, Sale is a candidate creator of the Payment.
- Contraindications
  - creation requires significant complexity, conditionally creating an instance from one of a family of similar classes based upon some external property value.
  - Concrete Factory/Abstract Factory rather than use Creator.

Creator 5

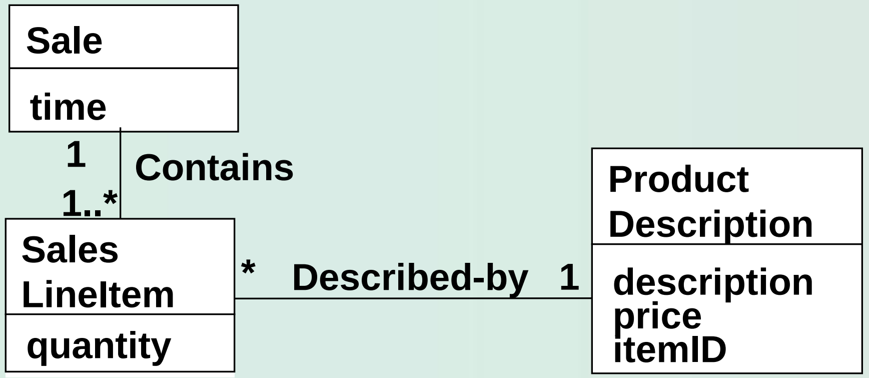
- Benefits
  - Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse.
    - Coupling is not increased because the created class is visible to the creator class, due to the existing associations.
- Related Patterns or Principles
  - Low Coupling
  - Concrete Factory and Abstract Factory
  - Whole-Part describes a pattern to define aggregate objects that support encapsulation of components.

Information Expert 1

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

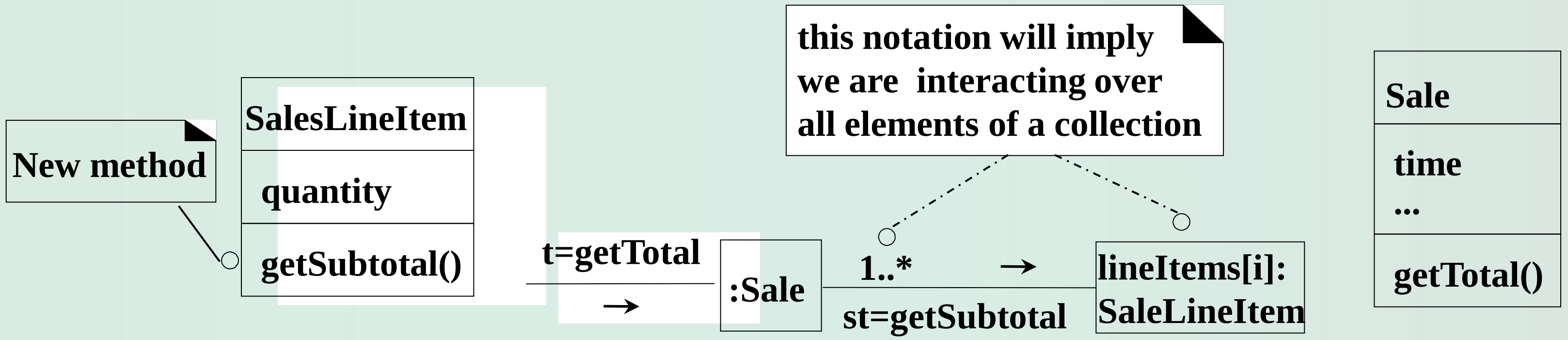
Information Expert 2

- NextGEN POS Example: some class needs to know the grand total of all the SalesLineItem instances of a sale.
  - Who should be responsible for knowing that?
  - Information Expert of the Domain Model : look for class that has the information needed to determine the total - Sale.
    - give the responsibility of knowing its total - method getTotal.
  - This approach supports **low representational gap** between software design of objects and the concepts of real domain.



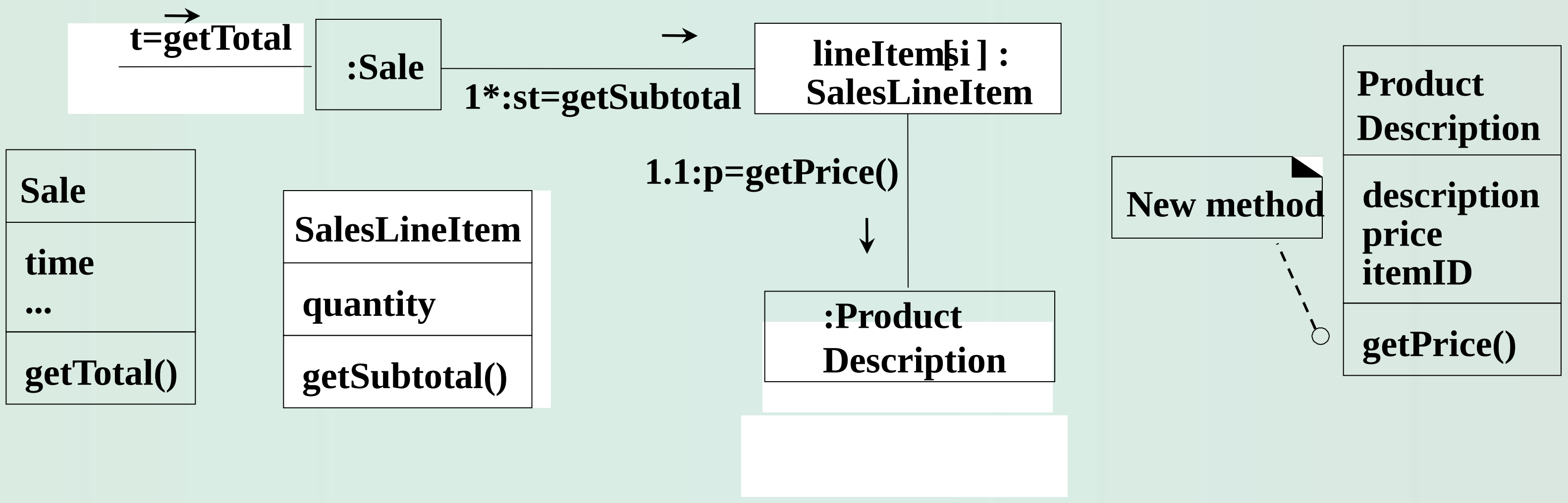
Information Expert 3

- To create the interaction diagrams in order to assign responsibilities to objects.
- Who should know the line item subtotal?
  - The SalesLineItem knows its quantity and its associated ProductDescription; therefore, by Expert, SalesLineItem should determine the subtotal.
  - For interaction diagram, the Sale should send getSubtotal messages to each of the SalesLineItems and sum the results



Information Expert 4

- To fulfill the responsibility of knowing its subtotal, a SalesLineItem has to know the product price.
- The ProductDescription is an information expert on answering its price;
- SalesLineItem sends a message asking for the product price



Information Expert 5

- To fulfill the responsibility of knowing and answering the sale's total.
  - assign three responsibilities to three design classes of objects in the interaction diagram.
  - summarize the methods in the method section of a class diagram.

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price

Information Expert 6

- Discussion
  - the fulfillment of a responsibility often requires information crossing different classes of objects.
    - many "partial" information experts will collaborate in the task.
    - the sales total problem required the collaboration of three classes of objects. they need to interact via messages to share the work.
  - "Do It Myself" strategy.
    - In the real world, a sale does not tell you its total; it is an inanimate thing.
    - In object-oriented software, all software objects are alive, and they can take on responsibilities and do things related to the information they know.
  - In a business, the chief financial officer should be responsible for creating a profit-and-loss statement.
    - software objects collaborate because the information is spread around, so it is with people. The company's chief financial officer may ask accountants to generate reports on credits and debits.

Information Expert 7

- Contraindications
  - Coupling, cohesion, and duplication problems.
    - who should be responsible for saving a Sale in a database? Sale class.
    - The Sale class must contain database handling, SQL and JDBC.
    - Lower cohesion: The class no longer focuses on just the pure application logic of "being a sale.“.
    - Increase coupling: The class must be coupled to the database/JDBC services of another subsystem, rather than being coupled to other objects in the domain layer.
    - Similar database logic would be duplicated in many persistent classes.
  - The above violate basic architectural principle
    - design for a separation of major system concerns.
    - Keep application logic in one place (domain software objects).
    - keep database logic in another place

Information Expert 8

- Benefits
  - Information encapsulation is maintained since objects use their own information to fulfill tasks. This usually supports low coupling.
  - Behavior is distributed across the classes that have the required information, encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain.
- Related Patterns or Principles
  - Low Coupling
  - High Cohesion
- Also Known As
  - Place responsibilities with data, That which knows, Do It Myself, Put Services with the Attributes They Work On.

Low Coupling 1

- 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.
- **Coupling**
  - a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
  - An element with low coupling is not dependent on too many other classes, subsystems, systems.
  - High coupling problems:
    - Forced local changes because of changes in related classes.
    - Harder to understand in isolation.
    - Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

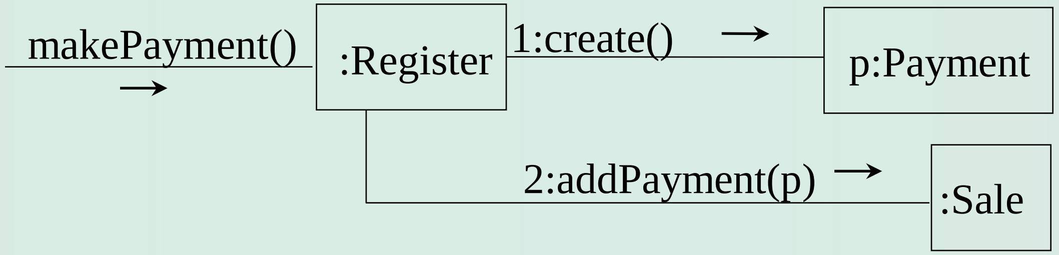
Low Coupling 2

- NextGen POS example
  - To create a Payment instance and associate it with the Sale. What class should be responsible for this?
  - Which design, based on assignment of responsibilities, supports Low Coupling?
  - In both cases we assume the Sale must eventually be coupled to knowledge of a Payment.

Register
Payment
Sale

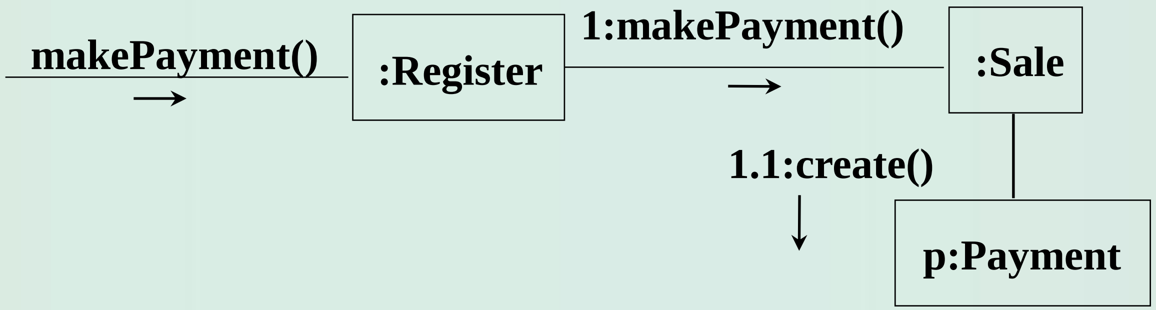
Low Coupling 3

- Design 1
  - 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.
  - **Adds coupling of** Register to Payment



Low Coupling 4

- Design 2
  - the Sale does the creation of a Payment, **does not increase the coupling**.
  - Purely from the point of view of coupling, prefer Design 2 because it maintains overall lower coupling.
  - In practice, the level of coupling alone can't be considered in isolation from other principles such as Expert and High Cohesion.



Low Coupling 5

- Discussion
  - 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 TypeY itself.
    - A TypeX object calls on services of a TypeY 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 6

- Discussion
  - A subclass is strongly coupled to its superclass.
    - suppose objects must be stored persistently in a relational or object database.
    - creating an abstract superclass called PersistentObject.
    - Disadvantage: it highly couples domain objects to a particular technical service and mixes different architectural concerns.
    - Advantage: automatic inheritance of persistence behavior.
  - Classes that are inherently generic and high reuse should have low coupling.
  - Contraindications: High coupling to stable elements and to pervasive elements is seldom a problem.
    - J2EE application can safely couple itself to the Java libraries (java.util..), because they are stable and widespread.

Low Coupling 7

- Focus on the points of realistic high instability or evolution.
  - the NextGen project, the different third-party tax calculators (with unique interfaces) need to be connected to the system. Designing for low coupling at this variation point is practical.
- Benefits
  - not affected by changes in other components
  - simple to understand in isolation
  - convenient to reuse
- Related Patterns: Protected Variation

Controller 1

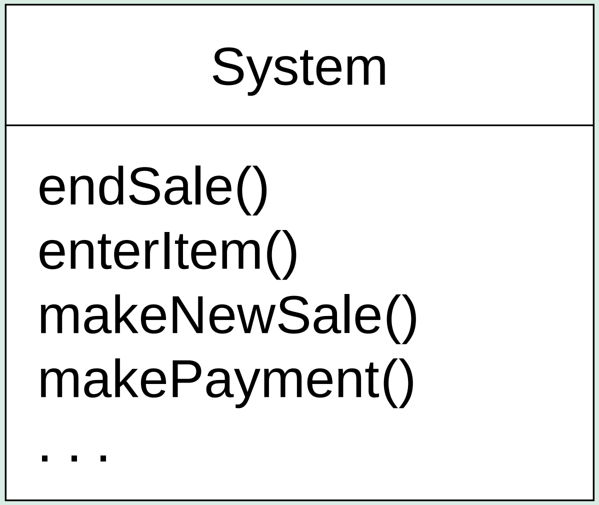
- Problem
  - What first object beyond the UI layer receives and coordinates ("controls") a system operation?
    - A controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.
    - System operations were first explored during the analysis of SSD. These are the major input events upon our system. e.g.,
      - when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended."
      - when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

Controller 2

- Solution
  - Assign the responsibility to a class
    - Represents the overall "system," a device that the software is running within, or a major subsystem. These are all variations of a facade controller.
    - Represents a use case scenario within which the system event occurs, 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.
    - "window," "view," and "document" classes are not on this list. They typically receive these events and delegate them to a controller.

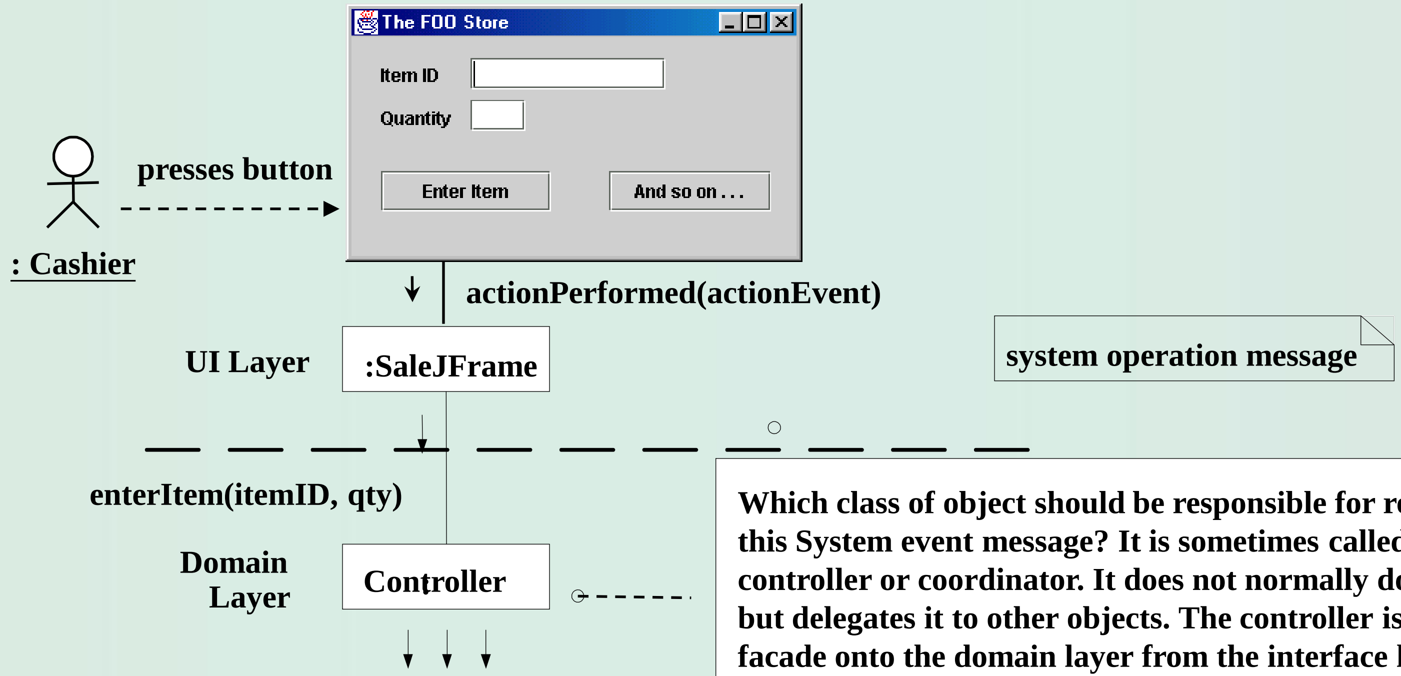
Controller 3

- NextGen POS example
  - Some system operations shown.
  - This model shows the system itself as a class (which is legal and sometimes useful when modeling).



Controller 4

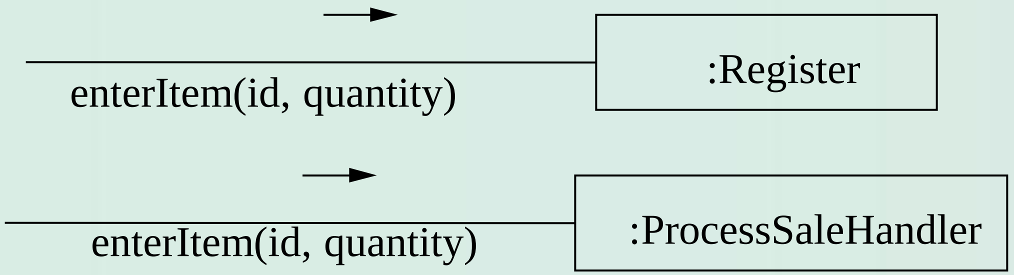
- During analysis, system operations may be assigned to the class System.
- During design, a controller class is assigned the responsibility for system operations



Which class of object should be responsible for receiving this System event message? It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects. The controller is a kind of facade onto the domain layer from the interface layer

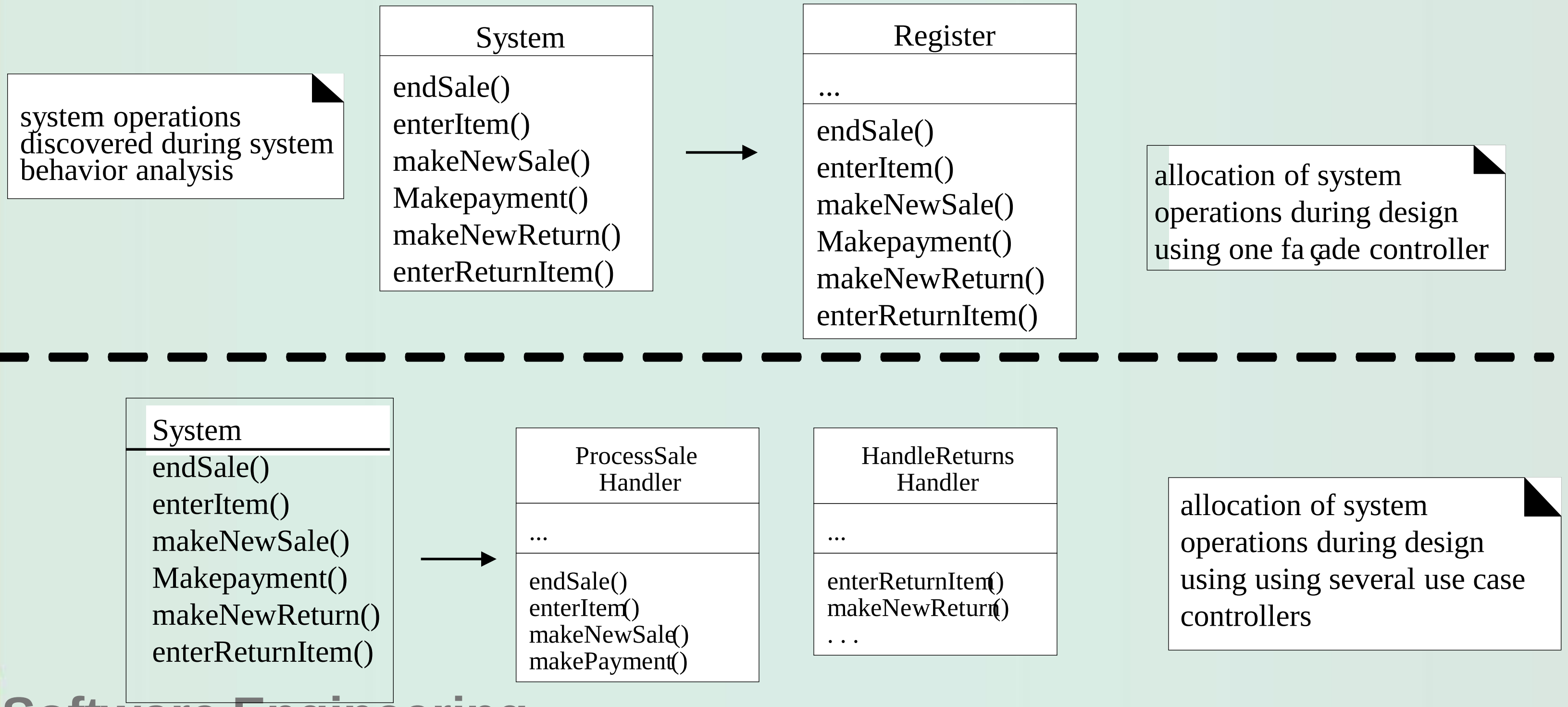
Controller 5

- Who should be the controller for system events such as enterItem and endSale?
- By the Controller pattern
  - Represents the overall "system," "root object," device, or subsystem: Register, POSSystem
  - Represents a receiver or handler of all system events of a use case scenario: ProcessSaleHandler, ProcessSaleSession
  - the domain of POS, a Register (POS Terminal) is a specialized device with software running in it.
  - The interaction diagrams:



Controller 6

- During design, the system operations identified during system behavior analysis are assigned to one or more controller classes, such as Register



Controller 7

- Discussion
  - This is a **delegation** pattern.
    - The UI layer shouldn't contain application logic.
    - UI layer objects must delegate work requests to domain layer.
    - In the domain layer, the Controller pattern summarizes choices, make for the domain object delegate that receives the work requests.
    - the controller is a kind of **facade** into the domain layer from the UI layer.
  - To use the same controller class for all the system events of one use case so that the controller can maintain information about the state of the use case.
    - to identify out-of-sequence system events (a makePayment operation before an endSale operation).
    - Different controllers may be used for different use cases.

Controller 8

- Discussion
  - A common defect in the design of controllers from over-assignment of responsibility.
    - A controller suffers from bad/low cohesion, violating the principle of High Cohesion.
  - **Guideline:** a controller should delegate to other objects the work that eeds to be done; it coordinates or controls the activity. It does not do much work itself.
  - Boundary objects are abstractions of the interfaces, entity objects are the application-independent/persistent domain objects, and control objects are use case handlers as described in this Controller pattern.

Controller 9

- Discussion
  - facade controller representing the overall system, or a subsystem.
    - to choose some class name over the other layers of the application and that provides the main point of service calls from the UI layer.
    - The facade could be an **abstraction** of the overall physical unit, such as a Register; a class representing the entire software system, such as POSSystem.
  - Facade controllers are suitable when there are not "too many" system events, or when the UI cannot redirect system event messages to alternating controllers, such as in a message-processing system.

Controller 10

- Web UIs and Server-Side Application of Controller
  - A delegation approach can be used in ASP.NET WebForms: The "code behind" file that contains event handlers for Web browser button clicks will obtain a reference to a domain controller object (e.g., a Register object in the POS), and then delegate the request for work.
  - Web-MVC, the "controller" is part of the UI layer and controls the UI interaction and page flow. The GRASP controller is part of the domain layer and controls the handling of the work request, essentially unaware of what UI technology is being used (e.g., a Web UI, a Swing UI).

Controller 11

- Web UIs and Server-Side Application of Controller
  - Java server-side design is delegation from the Web UI layer to an EJB Session object. The EJB Session object may itself delegate farther on to the domain layer of objects, and to apply the Controller pattern to choose a suitable receiver in the pure domain layer.
  - A rich-client Swing/UI forwards the request to the local client-side controller, and the controller forwards all or part of the request handling to remote services. This design lowers the coupling of the UI to remote services and makes it easier, for example, to provide the services either locally or remotely, through the indirection of the client-side controller.

Controller 12

- Benefits
  - Increased potential for reuse and pluggable interfaces
    - Application logic is not handled/bound in the interface layer, it can be replaced with a different interface.
    - Delegating a system operation responsibility to a controller supports the reuse of the logic in future applications.
  - Opportunity to reason about the state of the use case.
    - To ensure that system operations occur in a legal sequence, or we want to be able to reason about the current state of activity and operations within the use case.
    - E.g., to guarantee that the makePayment operation cannot occur until the endSale operation has occurred. To capture this state information somewhere; the controller is one reasonable choice.

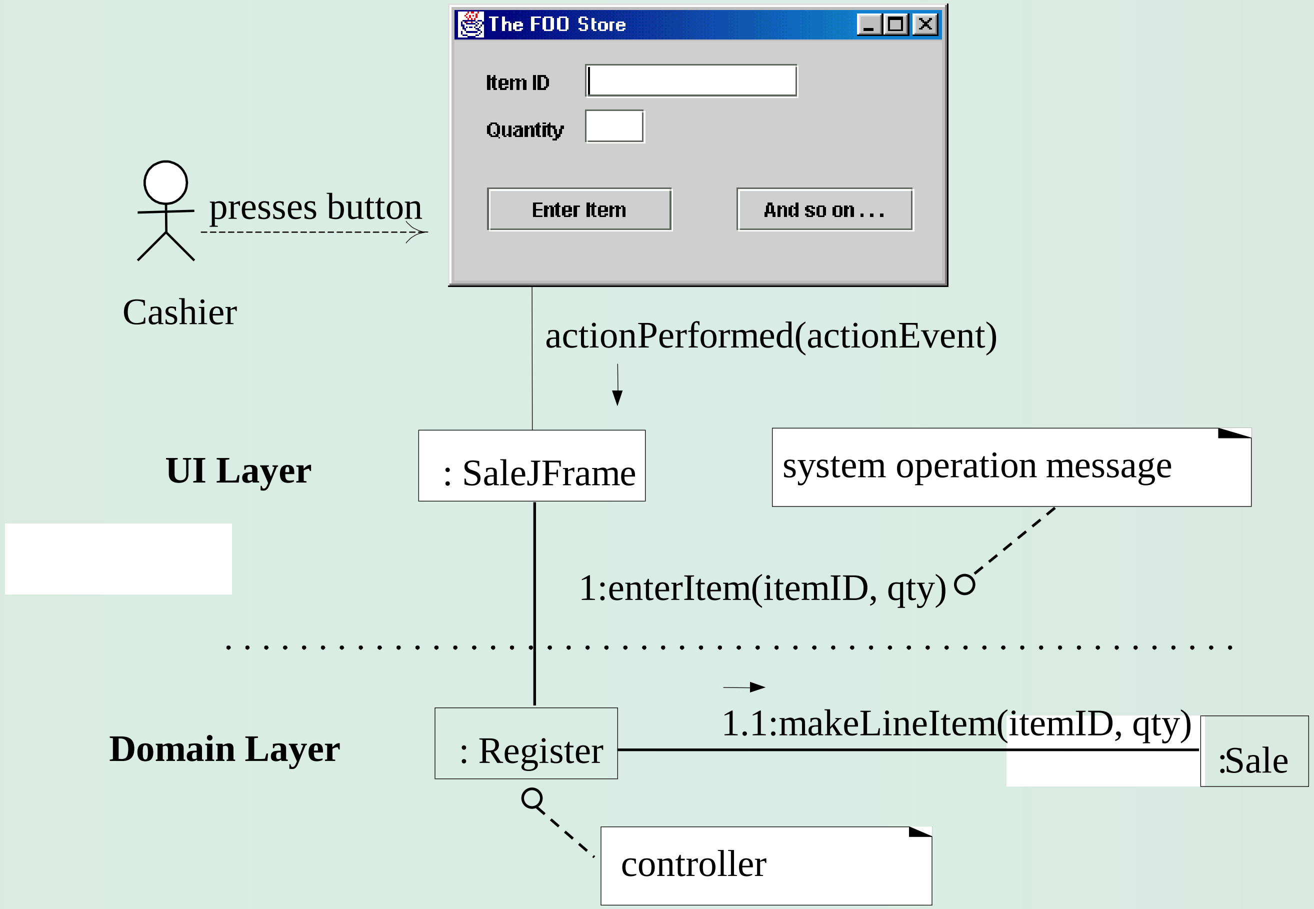
Controller 13

- Bloated Controllers: has low cohesion unfocused and handling too many areas of responsibility
  - There is only a single controller class receiving all system events in the system, and there are many of them.
  - The controller performs many of the tasks to fulfill the system event, without delegating the work.
  - A controller has many attributes, and maintains significant information about the system, which should have been distributed to other objects, or it duplicates information found elsewhere.
- The cures for a bloated controller
  - Add more controllersa system does not have to need only one. Instead of facade controllers, employ use case controllers.
    - e.g., an application with many system events - an airline reservation system - contain controllers: Use case controllers: MakeReservationHandler, ManageSchedulesHandler, ManageFaresHandler
  - Design the controller so that it primarily delegates the fulfillment of each system operation responsibility to other objects.

Controller 14

- UI Layer Does Not Handle System Events
  - Assume the NextGen application has a window that displays sale information and captures cashier operations.
  - Using the Controller pattern illustrates an acceptable relationship between the JFrame and the controller and other objects in a portion of the POS system.
  - SaleJFrame classpart of the UI layer delegates the enterItem request to the Register object.

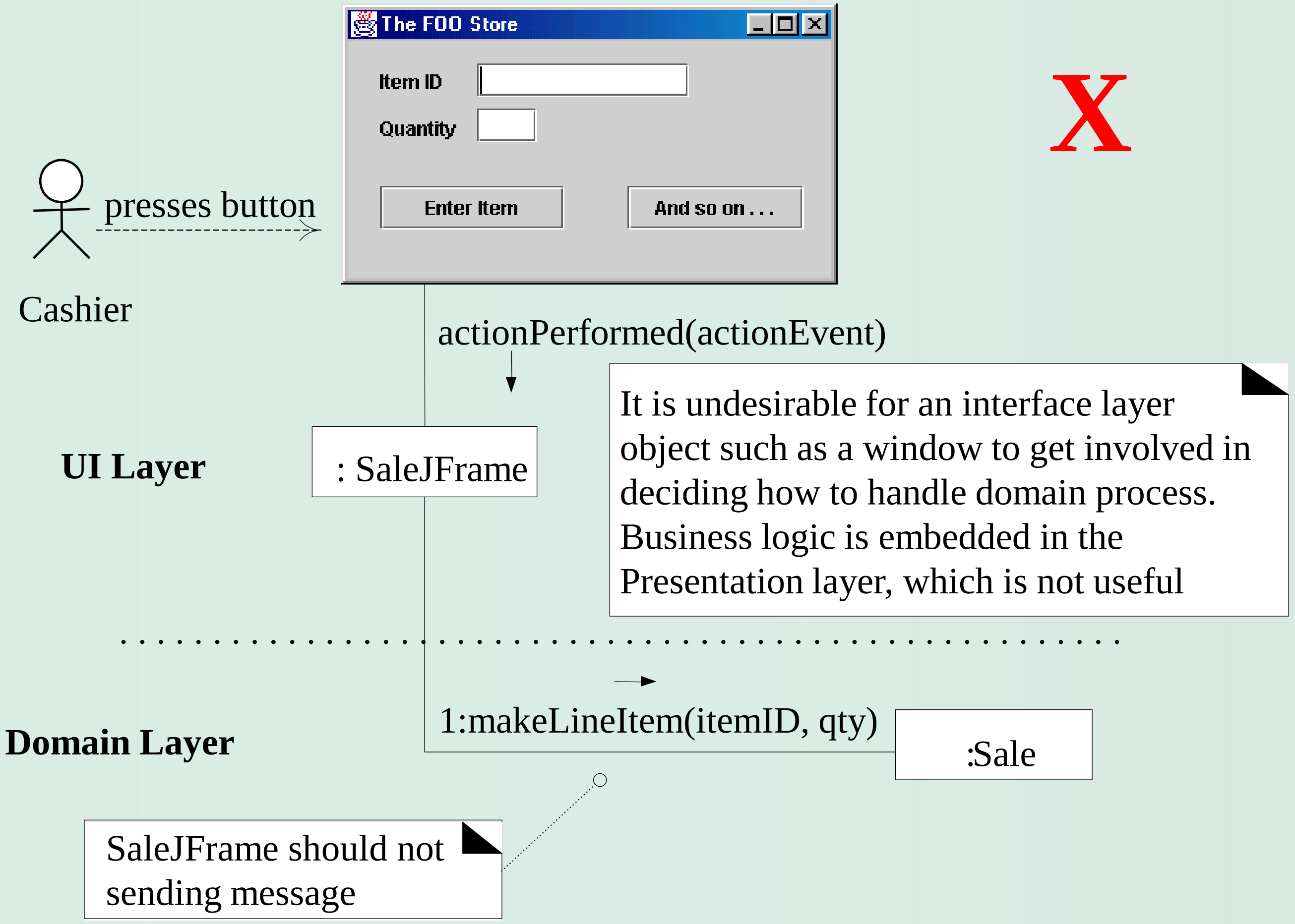
Controller 15



Controller 16

- If a UI layer object (SaleJFrame) handles a system operation that represents part of a business process, then business process logic would be contained in an interface object;
  - the opportunity for reuse of the business logic then diminishes because of its coupling to a particular interface and application.

Controller 17



It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain process. Business logic is embedded in the Presentation layer, which is not useful

SaleJFrame should not sending message

Controller 18

- Related Patterns
  - Command: In a message-handling system, each message may be represented and handled by a separate Command object.
  - Fa çade: A facade controller is a kind of Facade [GHJV95].
  - Layers: This is a POSA pattern. Placing domain logic in the domain layer rather than the presentation layer is part of the Layers pattern.
  - Pure Fabrication: This GRASP pattern is an arbitrary creation of the designer, not a software class whose name is inspired by the Domain Model. A use case controller is a kind of Pure Fabrication.

High Cohesion 1

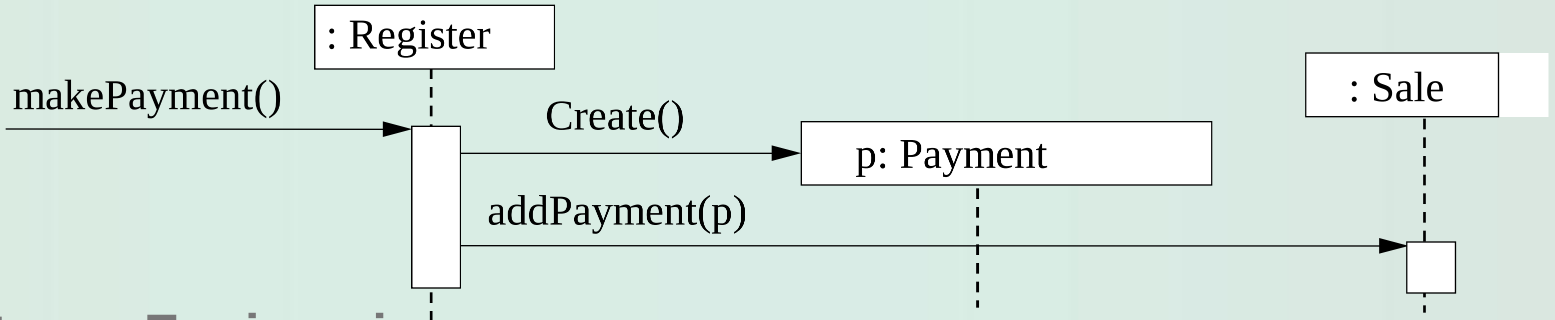
- Problem
  - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - Cohesion: a measure of how strongly related and focused the responsibilities of a class or subsystem are.
  - Low cohesion class
    - does many unrelated things or too much work.
    - represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects
    - hard to comprehend, reuse, maintain, delicate; constantly affected by change
- Solution
  - Assign a responsibility so that cohesion remains high.
  - Use this to evaluate alternatives.

High Cohesion 2

- POS Example: Low Coupling pattern for High Cohesion.
  - to create a (cash) Payment instance and associate it with the Sale. What class should be responsible for this?

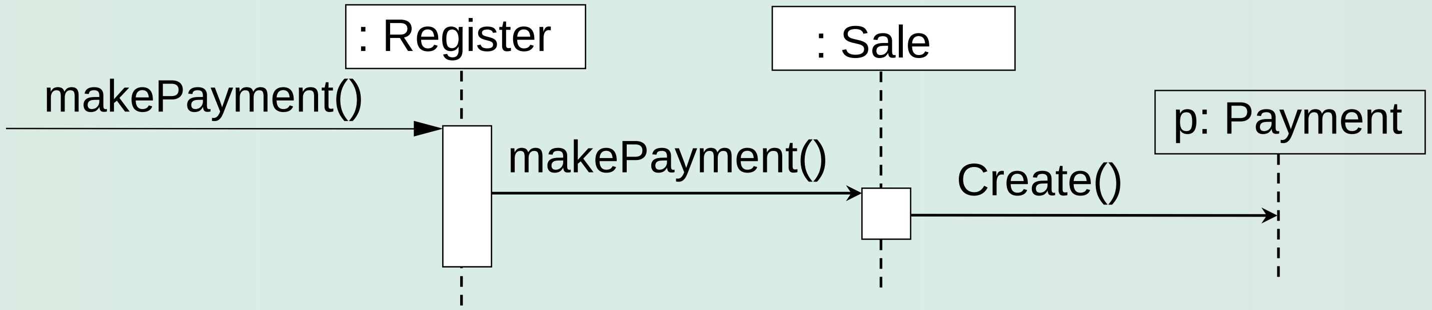
High Cohesion 3

- Design 1
  - Since Register records a Payment in the real-world domain, the Creator pattern suggests <u>Register</u> for <u>creating the Payment</u>.
  - The Register instance could send an addPayment message to the Sale, passing along the new Payment as a parameter
  - To places the responsibility for making a payment in the Register.
- To continuous make the Register class responsible for doing most of the work related to more system operations
  - it will become increasingly with tasks and become incohesive.



High Cohesion 4

- The second design delegates the payment creation responsibility to the Sale supports higher cohesion in the Register.
  - Since the second design supports both high cohesion and low coupling.
- In practice, the level of cohesion alone can't be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.



High Cohesion 5

- Discussion
  - Very low cohesion: A class is responsible for many things in very different functional areas.
    - RDB-RPC-Interface class is completely responsible for interacting with relational databases and for handling remote procedure calls.
    - The responsibilities should be split into a family of classes related to RDB access and a family related to RPC support.
  - Low cohesion: A class has responsibility for a complex task in one functional area.
    - RDBInterface class is completely responsible for interacting with relational databases.
    - The methods of the class are all related, but hundreds or thousands of methods.
    - The class should split into a family of lightweight classes sharing the work to provide RDB access.

High Cohesion 6

- Discussion
  - High cohesion: A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
    - **RDBInterface class** 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.
  - 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.
    - **Company class** is completely responsible for (1) knowing its employees and (2) 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.
    - The total number of public methods is small.

High Cohesion 7

- Discussion
  - High cohesion class 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.
  - This principle is an evaluative principle evaluating all design decisions.
  - Real world
    - if a person takes on too many unrelated responsibilities, then the person is not effective.
    - Managers have not learned how to delegate, they are ready to become "unglued."

High Cohesion 8

- **Modular Design Principle:** a system has been decomposed into a set of cohesive and loosely coupled modules.
- Contraindications
  - Case 1
    - Only one or two SQL experts know how to best define and maintain SQL.
    - Few OO programmers may have strong SQL skills.
    - Suppose the SQL expert is not even a comfortable OO programmer.
    - 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.

High Cohesion 9

- Contraindications
  - Case 2: 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.
    - 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 fewer remote calls and better performance.

High Cohesion 9

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