Unit 4

System Sequence Diagrams

Objectives

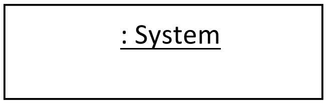
- Identify System Events.
- Create System Sequence diagrams for use case scenarios.

System Sequence Diagram

- A System Sequence Diagram is an artifact that illustrates input and output events related to the system under discussion.
- System Sequence Diagrams are typically associated with use-case realization in the logical view of system development.
- Sequence Diagrams (not system Sequence Diagrams) display object interactions arranged in time sequence.
- Sequence Diagrams depict the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the system.
- Sequence diagrams can be used to drive out testable user interface requirements.

SSD—System Behavior

- System behaves as "Black Box".
- Interior objects are not shown, as they would be on a Sequence Diagram.

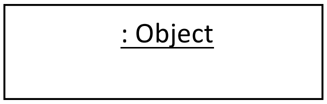


System Sequence Diagrams

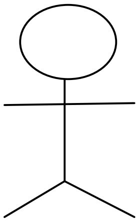
- SSD shows a particular course of events in the Use cases .
- How actors interact with system.
- Typical course of events that external actors generate and
- The order of the events.
- The operations of the system in response to the events generated.
- System Sequence Diagrams depict the temporal order of the events.
- System Sequence Diagrams should be done for the main success scenario of the use-case, and frequent and alternative scenarios.

Notation

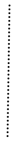
- **Object:** Objects are instances of classes. Object is represented as a rectangle which contains the name of the object underlined.
- Because the system is instantiated, it is shown as an object.



Actor: An Actor is modeled using the ubiquitous symbol, the stick figure.



- **Lifeline:** The Lifeline identifies the existence of the object over time. The notation for a Lifeline is a vertical dotted line extending from an object.



- **Message:** Messages, modeled as horizontal arrows between Activations, indicate the communications between objects.

Example of an SSD

- Following example shows the success scenario of the Process Sale use case.
- Events generated by cashier (actor)-

makeNewSale
enterItem
endSale and
makePayment.

SSD for Process Sale scenario

System Sequence Diagrams and Use Cases

- System Sequence Diagram is generated from inspection of a use case.
- Constructing a systems sequence diagram from a use case-

1. Draw a line representing the system as a black box.
2. Identify each actor that directly operates on the system. Draw a line for each such actor.
3. From the use case, typical course of events text, identify the system (external) events that each actor generates. They will correspond to an entry in the right hand side of the typical use case. Illustrate them on the diagram.
4. Optionally, include the use case text to the left of the diagram.

SSDs are derived from use cases.

System Events and System Boundary

- To identify the system events, system boundary is critical.
- For the purpose of software development, the system boundary is chosen to be the software system itself.
- Identifying the System events-

1. Determine the actors that directly interact with the system.
2. In the process Sale example, the customer does not directly interact with the POS system. Cashier interacts with the system directly. Therefore cashier is the generator of the system events. Defining system boundary.

Naming System Events and Operations

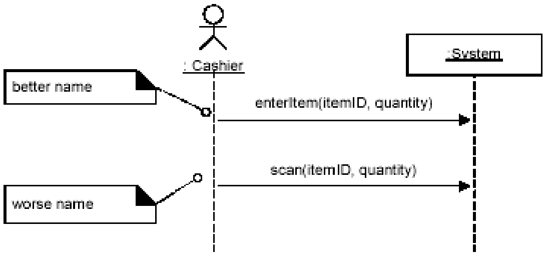
System event

- External input event generated by an actor.
- Initiates a responding operation by system.

System operation

- Operation invoked in response to system event.
- System events and their associated system operations should be expressed at the level of intent rather than in terms of the physical input medium.
- In order to improve the clarity, it is appropriate to start the name of the system event with a verb (for example- add….,enter….,end….,make…. etc.,). It also emphasizes the command origination of these events.
- For example "enterItem" is better than "scan" as it captures the intent of operation rather than what interface is used to capture the system event (design choice).

Choose event and operation names at an abstract level



- I text for the scenario.
- T iagram visually summarizes the interaction.

SSD with use case text

SSD and the Glossary

- Since the terms used in SSDs need proper explanation. If these terms are not explained in use cases, glossary could be used.
- A glossary is less formal, it is easier to maintain and more intuitive to discuss with external parties such as users and customers.
- However, glossary data should be meaningful; otherwise it will be an unnecessary work.

System Sequence Diagrams Within the Unified Process

- System Sequence Diagrams are a visualization of the interactions implied in the use cases.
- System Sequence Diagrams were not explicitly mentioned in the original UP description.
- **Phases**

1.Inception: System Sequence Diagrams are not usually motivated in inception.

Elaboration: It is useful to create System Sequence Diagrams during elaboration in order to -

- Identify the system events and major operations.
- To write system operation contracts (Contracts describe detailed system behavior) and To support estimation.

Conclusion

- System Sequence Diagrams provide a way for us to visually step through invocation of the operations defined by Use-Cases.
- It is not necessary to create SSDs for all scenarios of all use-cases, at least not at the same time.

LOGICAL ARCHITECTURE AND UML PACKAGE DIAGRAMS

Objectives

- **Introduce a logical architecture using layers.**
- **Illustrate the logical architecture using UML package diagrams.**

Roadmap

Sample UP artifact influence

LOGICAL ARCHITECTURE AND LAYERS

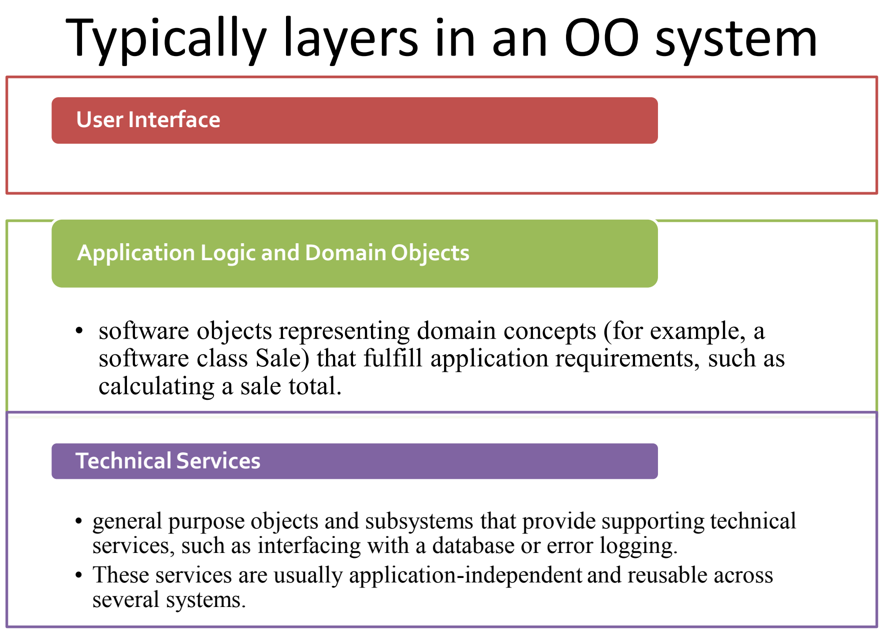
What is the Logical Architecture?

- The logical architecture is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers.
- It's called the logical architecture because
  - there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network
- So architecture required many VIEWS* to fully describe it.

Applying UML: Package Diagrams

- A UML package diagram provides a way to group elements.
- A UML package can group anything: classes, other packages, use cases, and so on.
- UML package diagrams are often used to illustrate the logical architecture of a system the layers, subsystems, packages (in the Java sense), etc.
- A layer can be modeled as a UML package; for example, the UI layer modeled as a package named UI.
- Nesting packages is very common.

Layers shown with UML package diagram notation



Typically layers in an OO system

User Interface

Application Logic and Domain Objects

- software objects representing domain concepts (for example, a software class Sale) that fulfill application requirements, such as calculating a sale total.

Technical Services

- general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging.
- These services are usually application-independent and reusable across several systems.

What is Software Architecture?

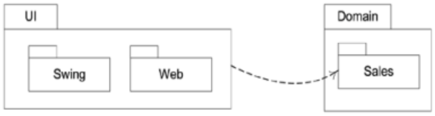
- An architecture is the set of significant decisions about:
  - the organization of a software system,
  - the selection of the structural elements and their interfaces by which the system is composed,
  - together with their behavior as specified in the collaborations among those elements,
  - the composition of these structural and behavioral elements into progressively larger subsystems,
  - and the architectural style that guides this organization these elements and their interfaces, their collaborations, and their composition.

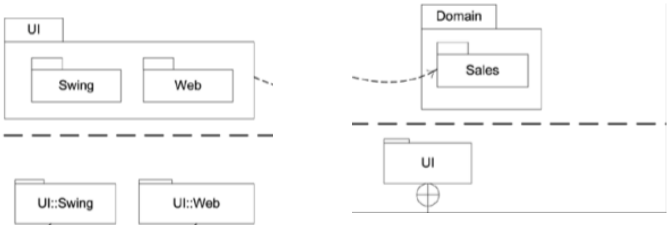
UML Packages and Notation

- UML package is a more general concept than simply a Java package or .NET namespace, though a UML package can represent those and more.
- The package name may be placed on the tab if the package shows inner members, or on the main folder, if not.
- It is common to want to show dependency (a coupling) between packages so that developers can see the large-scale coupling in the system.
- The UML dependency line is used for this, a dashed arrowed line with the arrow pointing towards the depended-on package.
- A UML package represents a namespace so that, for example, a Date class may be defined in two packages.
- If you need to provide fully-qualified names, the UML notation is, for example, java::util::Date in the case that there was an outer package named "java" with a nested package named "util" with a Date class.
- The UML provides alternate notations to illustrate outer and inner nested packages. Sometimes it is awkward to draw an outer package box around inner packages.

Package Nesting & Notation

- Alternate UML approaches to show package nesting,
  - using embedded packages
  - UML fully-qualified names and the circle-cross symbol





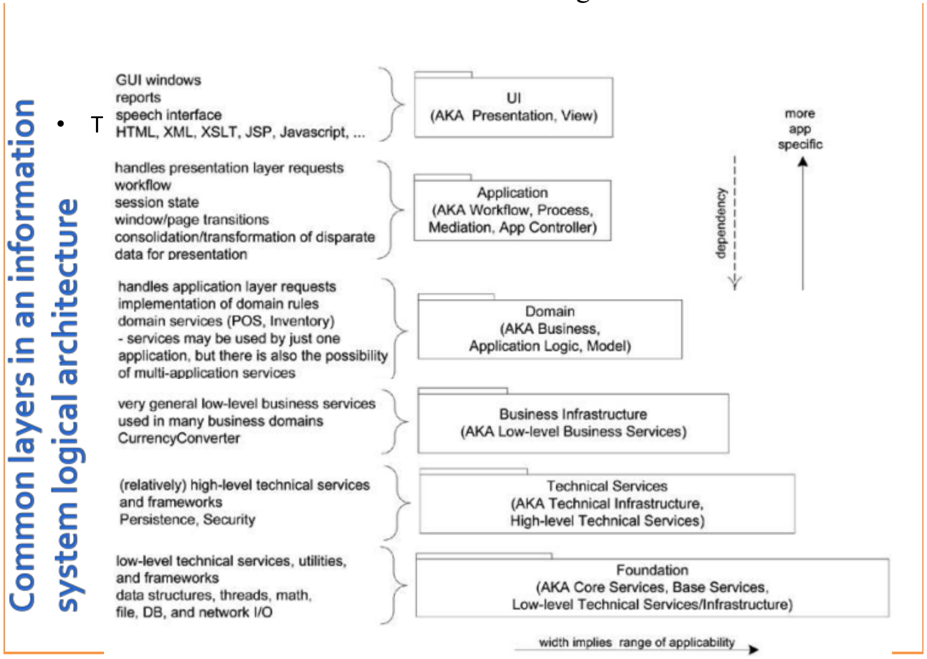
DESIGN WITH LAYERS

- The essential ideas of using layers are simple:
  - Organize the large-scale logical structure of a system into discrete layers
    - distinct, related responsibilities,
    - with a clean, cohesive separation of concerns such that
      - the "lower" layers are low-level and general services,
      - and the higher layers are more application specific.
      - Collaboration and coupling is from higher to lower layers;
- lower-to-higher layer coupling is avoided.

Problems that Layering Addresses

- Source code changes are rippling throughout the system many parts of the systems are highly coupled.
- Application logic is intertwined with the user interface, so it cannot be reused with a different interface or distributed to another processing node.

- Potentially general technical services or business logic is intertwined with more application-specific logic,
  - so it cannot be reused,
  - Cannot be distributed to another node,
  - Cannot be easily replaced with a different implementation.
- There is high coupling across different areas of concern.
  - Difficult to divide the work along clear boundaries for different developers.
- T


Common layers in an information system logical architecture
GUI windows
reports
speech interface
HTML, XML, XSLT, JSP, Javascript, ...
UI
(AKA Presentation, View)
handles presentation layer requests
workflow
session state
window/page transitions
consolidation/transformation of disparate data for presentation
Application
(AKA Workflow, Process, Mediation, App Controller)
handles application layer requests
implementation of domain rules
domain services (POS, Inventory)
- services may be used by just one application, but there is also the possibility of multi-application services
Domain
(AKA Business, Application Logic, Model)
very general low-level business services used in many business domains
CurrencyConverter
Business Infrastructure
(AKA Low-level Business Services)
(relatively) high-level technical services and frameworks
Persistence, Security
Technical Services
(AKA Technical Infrastructure, High-level Technical Services)
low-level technical services, utilities, and frameworks
data structures, threads, math, file, DB, and network I/O
Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)
more app specific
dependency
width implies range of applicability

Benefits of Using Layers

- In general, there is a separation of concerns
  - a separation of high from low-level services and of application-specific from general services
  - This reduces coupling, dependencies and improves cohesion , reuse potential and increases clarity.
- Related complexity is encapsulated and decomposable.
- Some layers can be replaced with new implementations.
  - Generally not possible for lower-level Technical Service or Foundation layers.
  - May be possible for UI, Application, and Domain layers
- Lower layers contain reusable functions
- Some layers (primarily the Domain and Technical Services) can be distributed
- Development by teams is aided because of the logical segmentation.

Cohesive Responsibilities; Maintain a Separation of Concerns

- The responsibilities of the objects in a layer
  - should be strongly related to each other and should not be mixed with responsibilities of other layers.
- For example, objects in the UI layer should focus on UI work,
  - such as creating windows and widgets, capturing mouse and keyboard events, and so forth.
- Objects in the application logic or "domain" layer should focus on application logic,
  - such as calculating a sales total or taxes, or moving a piece on a game board.
- UI objects should not do application logic.

  - For example, a Java Swing JFrame (window) object should not contain logic to calculate taxes or move a game piece.
  - And on the other hand, application logic classes should not trap UI mouse or keyboard events.
  - Violate a clear separation of concerns and maintaining high cohesion basic architectural principles.

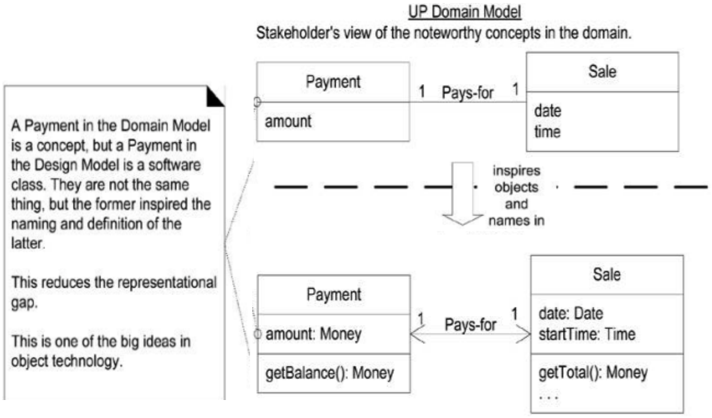
Domain Layer vs. Application Logic Layer; Domain Objects

- A typical software system has UI logic and application logic, such as GUI widget creation and tax calculations.
- How do we design the application logic with objects?
  - We could create one class called XYZ and put all the methods, for all the required logic, in that one class.
  - It could technically work (though be a nightmare to understand and maintain), but it isn't the recommended approach in the spirit of OO thinking.
- what is the recommended approach?
  - To create software objects with names and information similar to the real-world domain, and assign application logic responsibilities to them.
  - For example, in the real world of POS, there are sales and payments. So, in software, we create a Sale and Payment class, and give them application logic responsibilities.
  - This kind of software object is called a domain object. It represents a thing in the problem domain space, and has related application or business logic, for example, a Sale object being able to calculate its total.

What's the Relationship Between the Domain Layer and Domain Model?

- Relationship between the domain model and the domain layer
  - Domain model (which is a visualization of noteworthy domain concepts)
    - Inspiration for the names of classes in the domain layer.
- Domain layer and domain model relationship
  - The domain layer is part of the software and the domain model is part of the conceptual-perspective analysis they aren't the same thing.
  - But by creating a domain layer with inspiration from the domain model, we achieve a lower representational gap, between the real-world domain, and our software design.
  - For example, a Sale in the UP Domain Model helps inspire us to consider creating a software Sale class in the domain layer of the UP Design Model.

Domain layer and domain model relationship

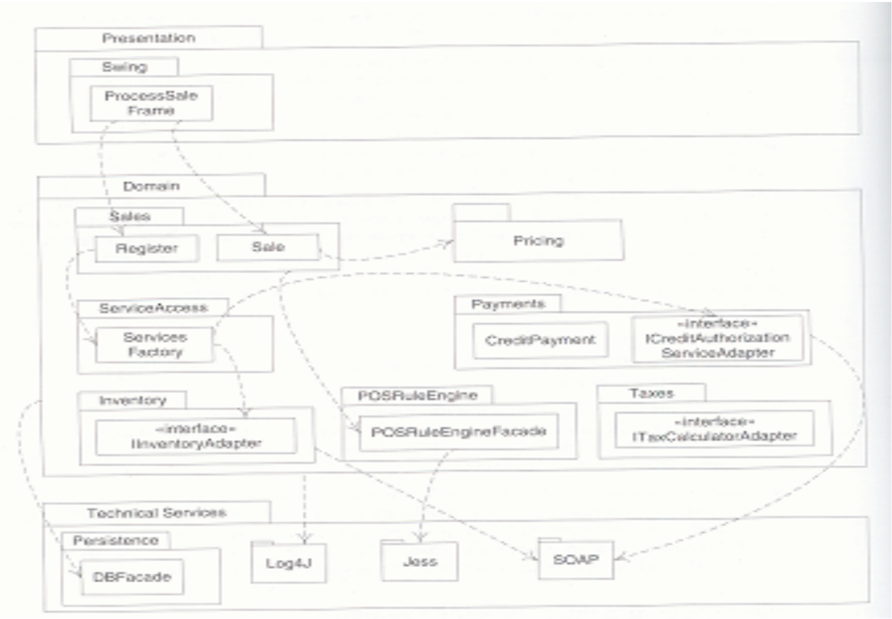


A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

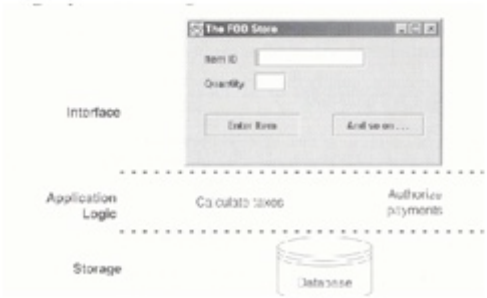
Partial coupling between Packages



Information Systems

- In IS layered architecture was known as three-tier architecture.
- A three-tier architecture has interface, Application logic and a storage.
- The singular quality of 3-tier architecture is:
- Separation of the application logic into distinct logical middle tier of software.
- The interface tier is relatively free of application processing.
- The middle tier communicates with the back-end storage layer.
- The following is an example of 3-tier architecture.

Example:



Two-tier Design

- In this design, the application logic is placed within window definitions, which read and writes directly to database.
- There is no middle tier that separates out the application logic.

The Model-View Separation Principle

- The principle states that model(domain) objects should not have direct knowledge of view(presentation) objects.
- Furthermore, the domain classes should encapsulate the information and behavior related to application logic.

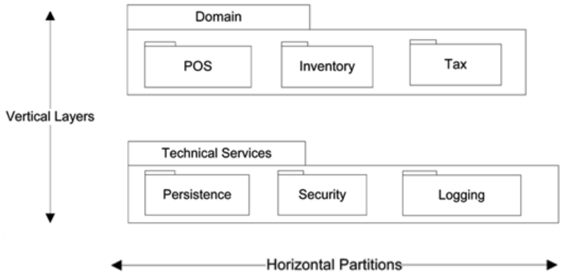
Need for Model-View separation

- To support cohesive model definitions that focus on the domain process, rather than on interfaces.
- 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 new views to be easily connected to an existing domain layer, without affecting the domain layer.

Tiers, Layers, and Partitions

- The original notion of a tier in architecture was a logical layer, not a physical node.
- The layers of an architecture are represent the vertical slices, while partitions represent a horizontal division of relatively parallel subsystems of a layer.
- For example, the Technical Services layer may be divided into partitions such as Security and Reporting

Layers and partitions.



MODEL VIEW PRINCIPLE

The Model-View Separation Principle

- What kind of visibility should other packages have to the UI layer?
- How should non-window classes communicate with windows?
- This principle has two parts:
- **Do not connect or couple non-UI objects directly to UI objects.**
    - don't let a Sale software object (a non-UI "domain" object) have a reference to a Java Swing JFrame window object. Why?
    - Because the windows are related to a particular application, while the non-windowing objects may be reused in new applications or attached to a new interface.
- **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 on to non-UI objects (such as domain objects).

Model vs. Domain and View vs.UI

- In this context, model is a synonym for the domain layer of objects .
- View is a synonym for UI objects, such as windows, Web pages, applets, and reports.

Model-View Separation principle

- The Model-View Separation principle states that
    - model (domain) objects should not have direct knowledge of view (UI) objects, at least as view objects.
    - For example, a Register or Sale object should not directly send a message to a GUI window object ProcessSaleFrame, asking it to display something, change color, close, and so on.

MVC – Model View Controller

- This is a key principle in the pattern Model-View-Controller (MVC).
- MVC was originally a small-scale Smalltalk-80 pattern, and related data objects (models), GUI widgets (views), and mouse and keyboard event handlers (controllers).
- The term "MVC" has been co-opted by the distributed design community to apply on a large-scale architectural level.

The Model is the Domain Layer, the View is the UI Layer, and the Controllers are the workflow objects in the Application layer

Implications of MVC

- The domain classes encapsulate the information and behavior related to application logic.

- The window classes are relatively thin; they are responsible for input and output, and catching GUI events, but do not maintain application data or directly provide application logic.
- For example, a Java JFrame window should not have a method that does a tax calculation.
- A Web JSP page should not contain logic to calculate the tax.
- These UI elements should delegate to non-UI elements for such responsibilities.

The Motivation/Need for Model-View Separation

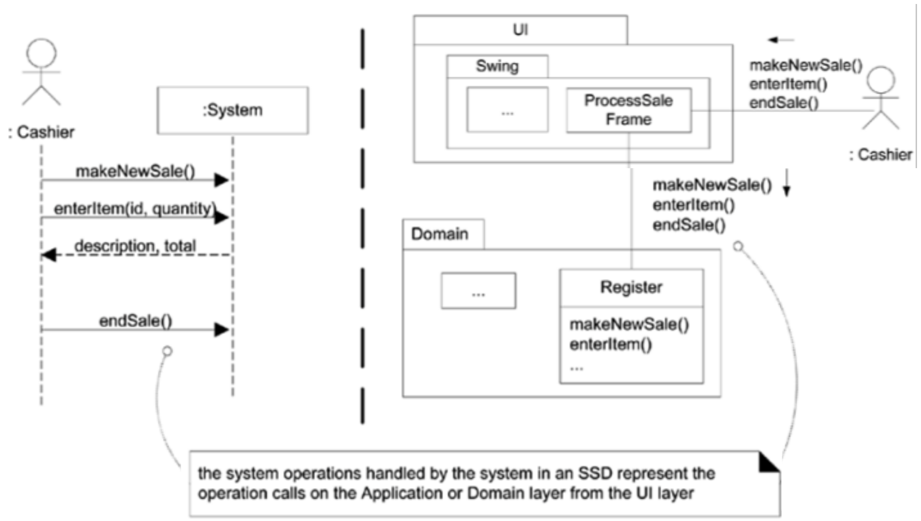
- To support cohesive model definitions that focus on the domain processes, rather than on user interfaces.
- To allow separate development of the model and user interface layers.
- To minimize the impact of requirements changes in the interface upon the domain layer.
- To allow new views to be easily connected to an existing domain layer, without affecting 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.
- To allow easy porting of the model layer to another user interface framework.

MORE ON LAYERED ARCHITECTURE

What's the Connection Between SSDs, System Operations, and Layers?

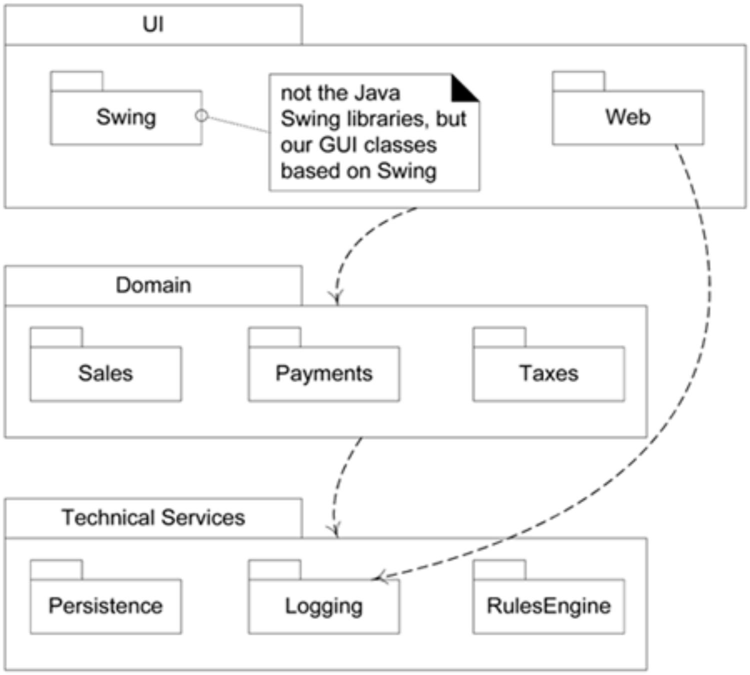
- During analysis work,
    - SSDs for use case scenarios.
    - input events from external actors into the system
        - calling upon system operations such as makeNewSale and enterItem.
        - The SSDs illustrate these system operations, but hide the specific UI objects.
- In a well-designed layered architecture
    - Supports High cohesion and separation of concerns
    - The UI layer objects will then forward or delegate the request from the UI layer onto the domain layer for handling.

System operations in the SSDs and in terms of layers



the system operations handled by the system in an SSD represent the operation calls on the Application or Domain layer from the UI layer

Example: NextGen Logical Architecture and Package Diagram



Learning Outcomes

- Logical Architecture and Layers
  - What is the Logical Architecture?
  - What are Layers
  - Typically layers in an OO system
  - What is Software Architecture?
  - Applying UML: Package Diagrams, Notation and Nesting
- Design with Layers
  - Problems that Layering Addresses
  - Benefits of Using Layers
  - Domain Layer vs. Application Logic Layer; Domain Objects
  - What's the Relationship Between the Domain Layer and Domain Model?
  - Tiers, Layers, and Partitions
- Model View Principle
  - Model-View Separation Principle
- More on Layered Architecture
  - What's the Connection Between SSDs, System Operations, and Layers?

UML Class Diagrams

Objective

- Create design class diagrams (DCDs).
- Identify the classes, methods, interfaces and their associations.
- The class diagram represent the static object modeling.
- In conceptual perspective – class diagram represent domain model.
- But in s/w perspective – they represent design model.

Class Diagrams

- The UML has notation for showing design details in static structure.
  - Class diagrams
- The definition of design class diagrams occurs within the design phase.
  - The UML does not specifically define design class diagram.
  - It is a design view on SW entities, rather than an analytical view on domain concepts.

Design Class Diagrams

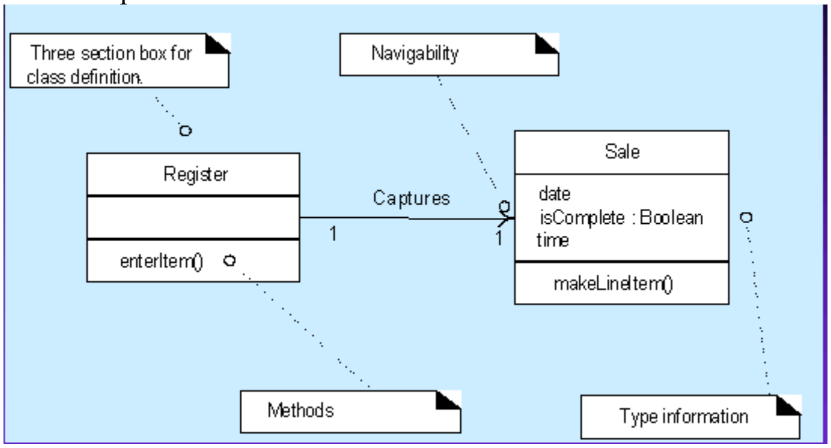
The creation of design class diagrams is dependent upon the prior creation of:

- Interaction diagrams
  - Identifies the SW classes that participate in the solution, plus the methods of classes.
- Conceptual model
  - Adds detail to the class definitions.

When to create DCDs?

- Although this presentation of design class diagrams follows the creation of interaction diagrams, in practice they are usually created **in parallel.**

  Example of a DCD



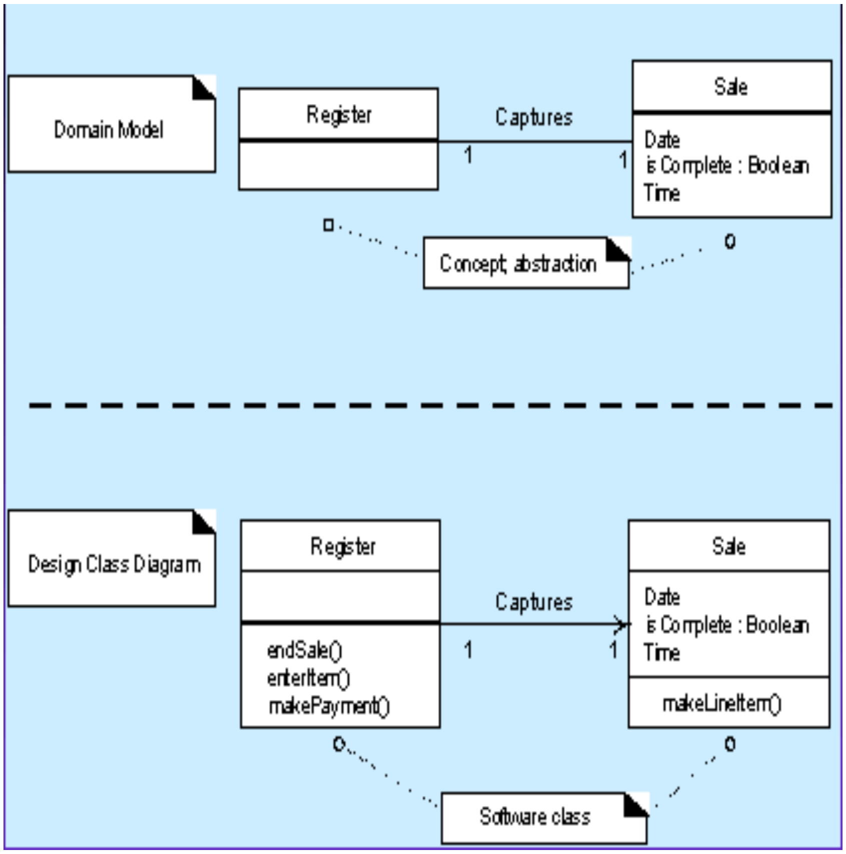
As we can see….

- A design class diagram illustrates the specifications for SW classes and interfaces.
- Typical information included:
  - Classes, associations, and attributes
  - Interfaces, with their operations and constants
  - Methods
  - Attribute type information
  - Navigability
  - Dependencies

How to make a DCD.

- Identify all the **classes** participating in the SW solution by analyzing the interaction diagrams.
- Draw them in a class diagram.
- Duplicate the **attributes** from the associated concepts in the conceptual model.
- Add **method** names by analyzing the interaction diagrams.
- Add **type** information to the attributes and methods.
- Add the **associations** necessary to support the required attribute visibility.
- Add **navigability** arrows to the associations to indicate the direction of attribute visibility.
- Add **dependency** relationship lines to indicate non-attribute visibility.

  Two perspective of class diagram



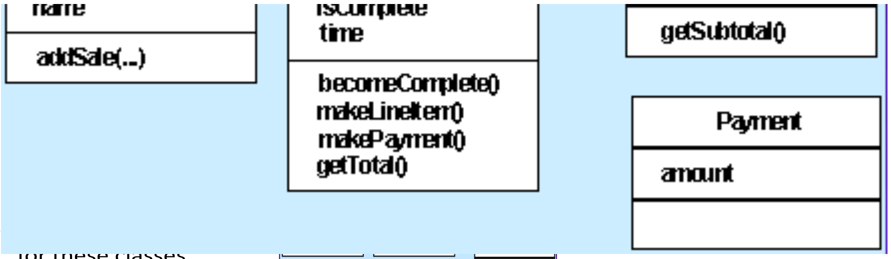
Classifier

- A UML classifier is a “model element that describes behavioral and structure features.
- Classifiers can also be specialized.
- They are a generalization of many elements of the UML, including classes, interfaces, use cases and actors.

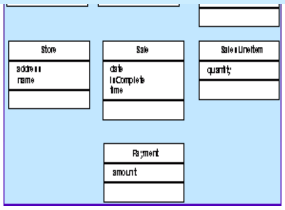
Creating the NextGen POS DCD.

- Identify SW classes and illustrate them.

  Register
  Product Catalog
  Store
  Payment
  Sale
  ProductSpecification
  SalesLineItem

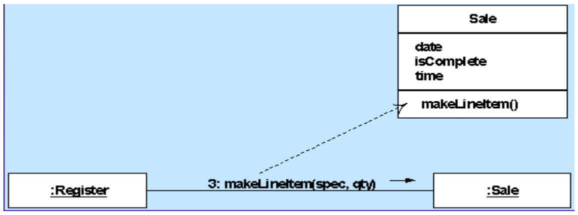


- for these classes.
- Include the **attributes** previously identified in the domain model.

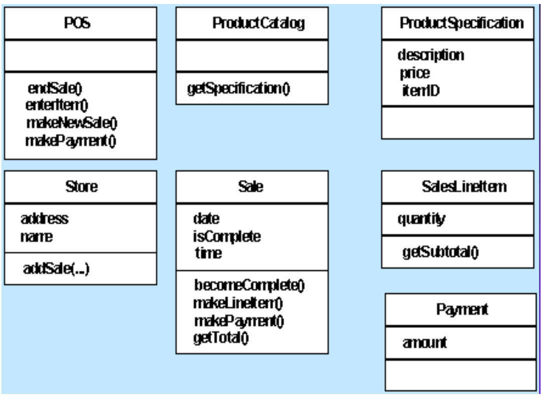


Add method name

- Method names from interaction diagrams



Methods……



Method names: issues

- The following special issues must be considered with respect to method names:
  - Interpretation of the create() message.
  - Depiction of accessing methods.
  - Interpretation of messages to multi-objects.
  - Language-dependent syntax.

Method names: create

- The create message is the UML language independent form to indicate instantiation and initialization.
- When translating the design to an OOPL it must be expressed in terms of its idioms for instantiation and initialization.

Method names: Accessing methods

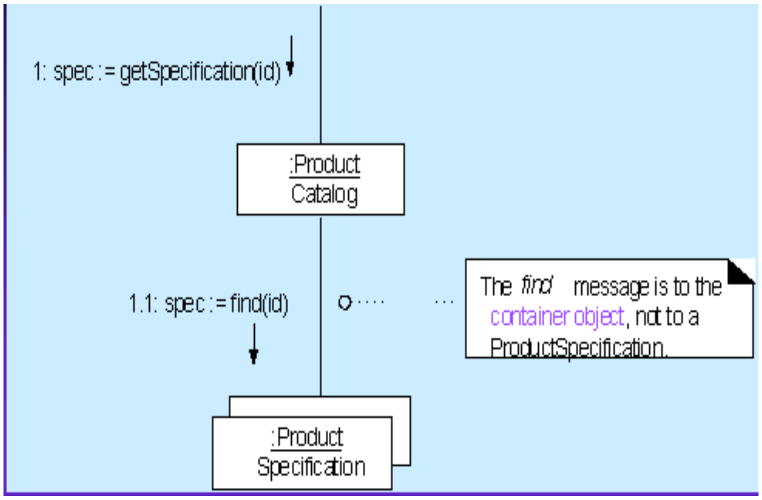
- Accessing methods are those which retrieve (accessor method - get) or set (mutator method) attributes.
- It is common idiom to have an accessor and mutator for each attribute, and to declare all attribute private (to enforce encapsulation).

Method names: Multi-objects

- A message to a multi-object is interpreted as a message to the container/collection object.

Message to a Multi-objects

- Java's map, C++'s map, Smalltalk's dictionary.


1: spec := getSpecification(id)
:Product Catalog
1.1: spec := find(id)
The find message is to the container object, not to a ProductSpecification.
:Product Specification

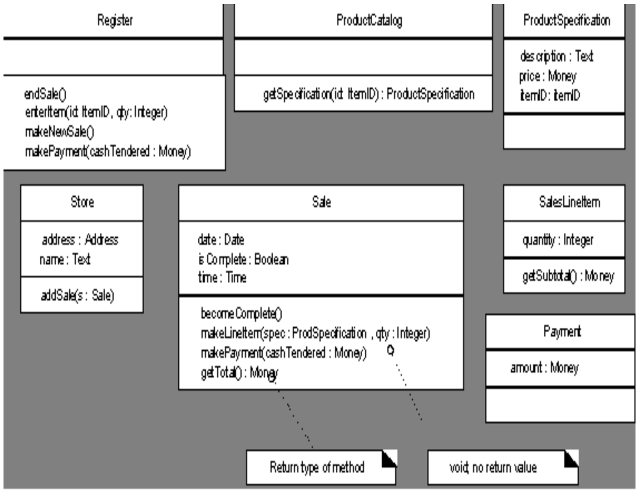
Method names:Language dependent syntax

- The basic UML format for methods:

  methodName(parameterList)

Adding more type information

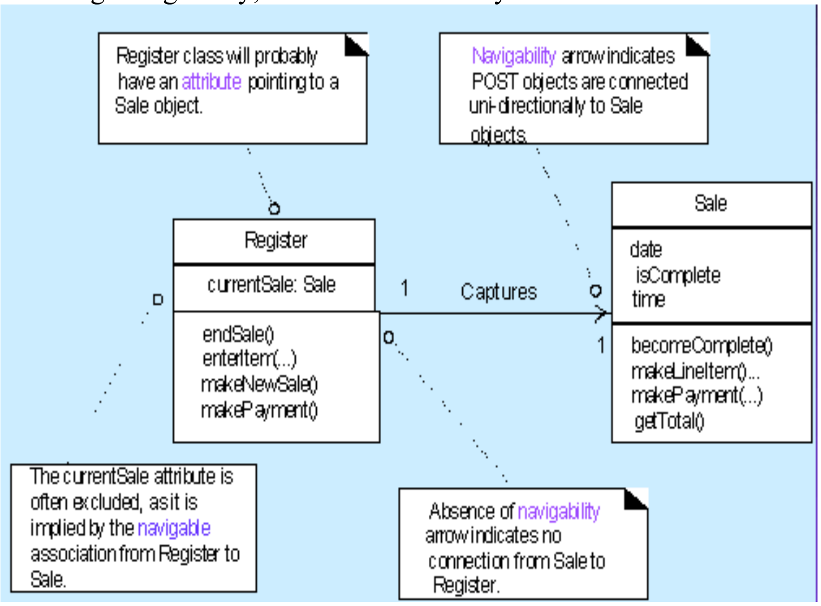
- The type of the attributes, method parameters, and method return values may all optionally be shown.
- The design class diagram should be created by considering the audience.
  - If it is being created in a CASE tool with automatic code generation, full and exhaustive details are necessary.
  - If it is being created for SW developers to read, exhaustive low-level detail may adversely effect the noise-to-value ratio.



Adding associations and navigability

- Navigability is a property of the role which indicates that it is possible to navigate uni-directionally across the association from objects of the source to target class.
- Navigability implies visibility.
- Most, if not all, associations in design-oriented class diagrams should be adorned with the necessary navigability arrows.

Showing navigability, or attribute visibility

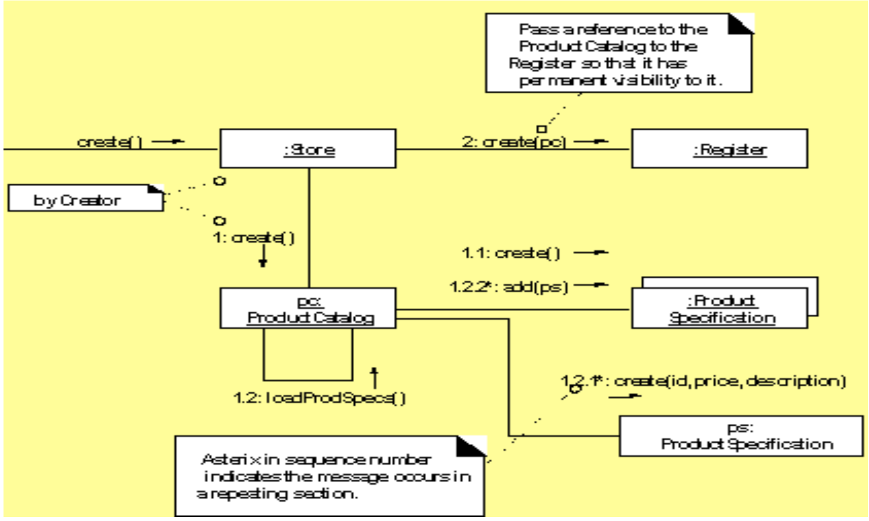

Register class will probably have an attribute pointing to a Sale object.
Navigability arrow indicates POST objects are connected uni-directionally to Sale objects.
Register
currentSale: Sale
endSale()
enterItem(...)
makeNewSale()
makePayment()
1
Captures
1
Sale
date
isComplete
time
becomeComplete()
makeLineItem()...
makePayment(...)
getTotal()
The currentSale attribute is often excluded, as it is implied by the navigable association from Register to Sale.
Absence of navigability arrow indicates no connection from Sale to Register.

Association with navigability

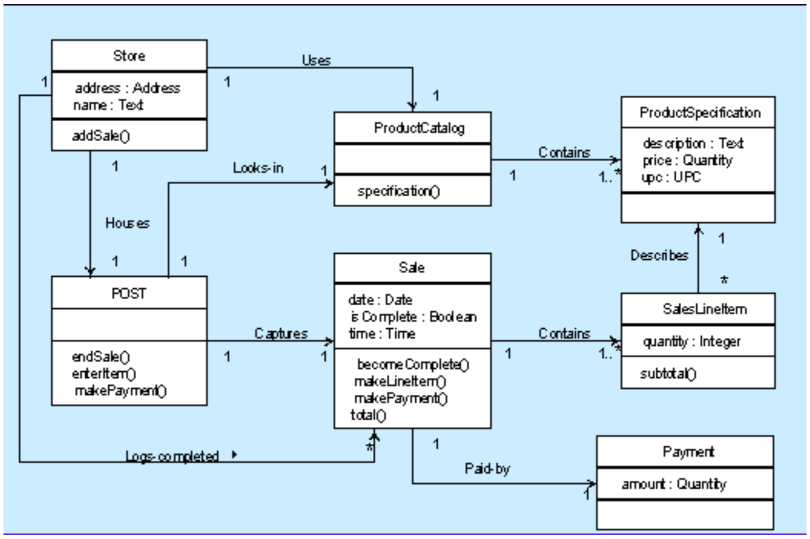
- Define an association with a navigability adornment from A to B:
  - A sends a message to B.
  - A creates an instance B.
  - A needs to maintain a connection to B.

Navigability from CDs.

- Navigability is Identified from Interaction Diagrams.



Associations with navigability adornments

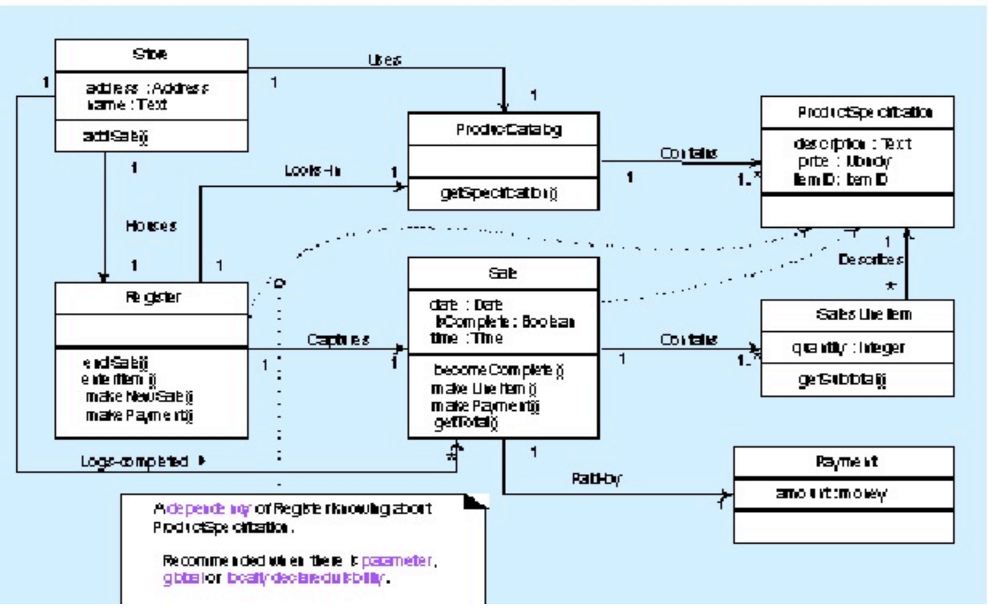


Adding dependency relationships

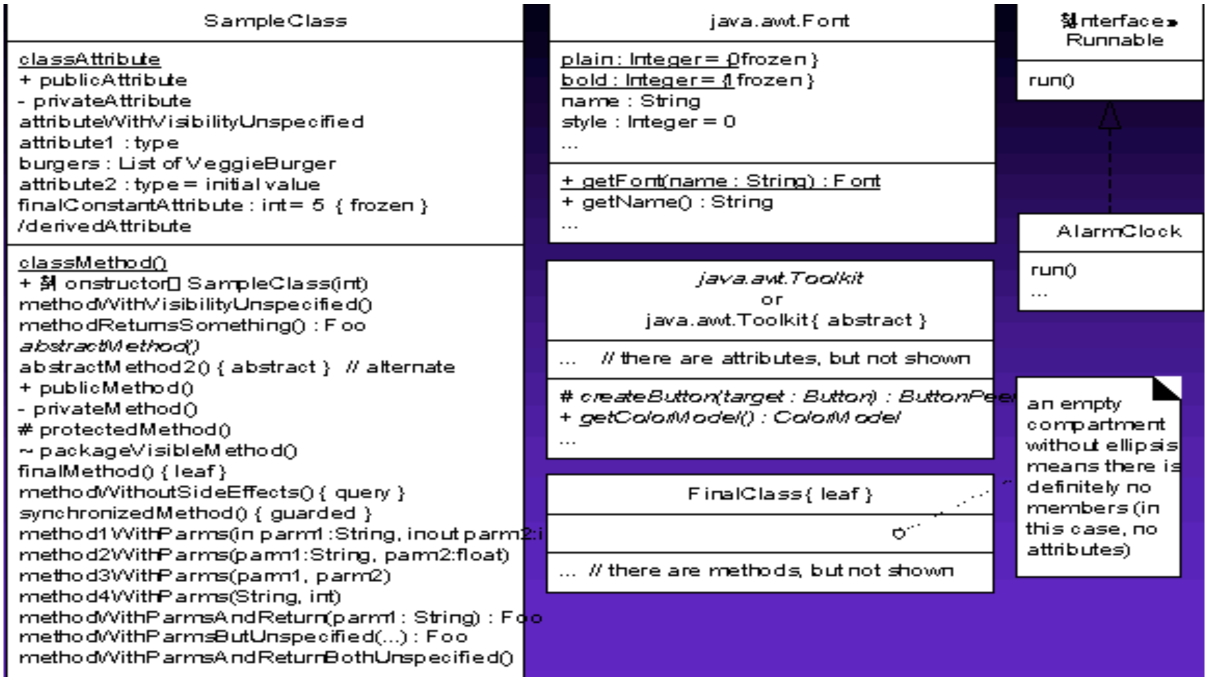
- Dependency relationship indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element.
  - A dependency is a using relationship that states a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse.
- The dependency relationship is useful to depict non-attribute visibility between classes.
  - Parameters
  - Global or local visibility
  - A dashed arrow line
  - A dashed directed line

------------------------▶

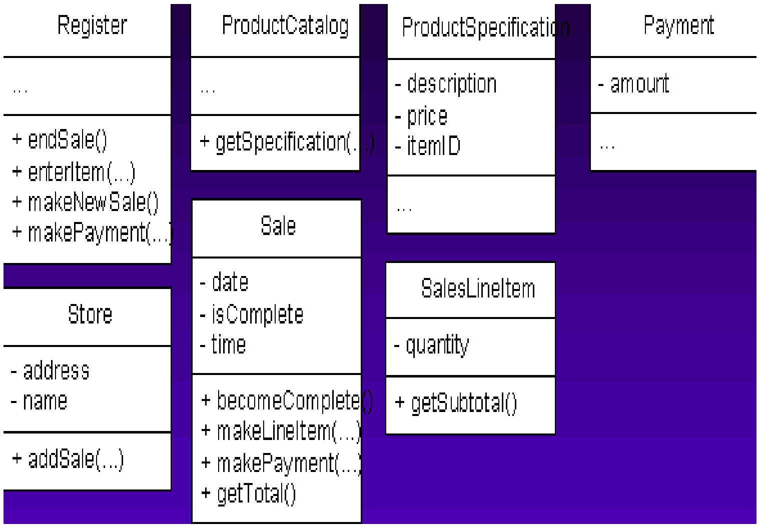
Dependency relationships non-attribute visibility



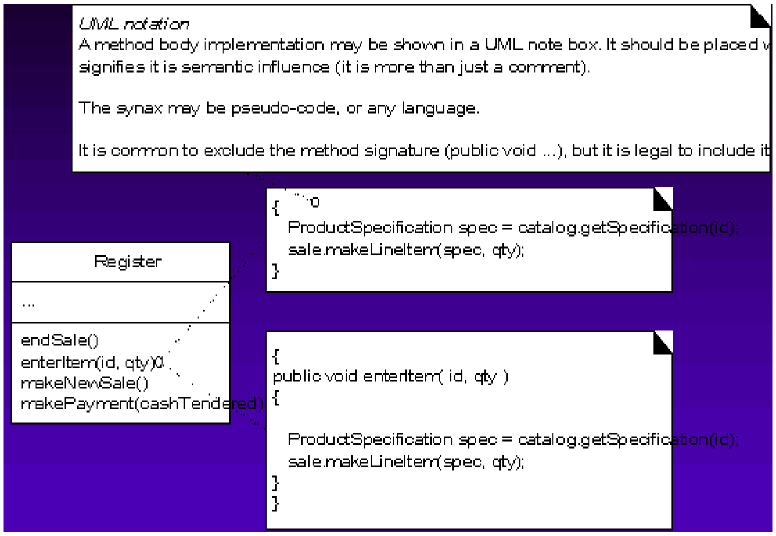
Notation for member details


SampleClass
classAttribute
+ publicAttribute
- privateAttribute
attributeWithVisibilityUnspecified
attribute1 : type
burgers : List of VeggieBurger
attribute2 : type = initial value
finalConstantAttribute : int = 5 { frozen }
/derivedAttribute
classMethod()
+ «constructor» SampleClass(int)
methodWithVisibilityUnspecified()
methodReturnsSomething() : Foo
abstractMethod()
abstractMethod2() { abstract } // alternate
+ publicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
finalMethod() { leaf }
methodWithoutSideEffects() { query }
synchronizedMethod() { guarded }
method1WithParms(in parm1:String, inout parm2:i
method2WithParms(parm1:String, parm2:float)
method3WithParms(parm1, parm2)
method4WithParms(String, int)
methodWithParmsAndReturn(parm1 : String) : Foo
methodWithParmsButUnspecified(...) : Foo
methodWithParmsAndReturnBothUnspecified()
java.awt.Font
plain : Integer = 0 { frozen }
bold : Integer = 1 { frozen }
name : String
style : Integer = 0
...
+ getFont(name : String) : Font
+ getName() : String
...
java.awt.Toolkit
or
java.awt.Toolkit { abstract }
... // there are attributes, but not shown
# createButton(target : Button) : ButtonPeer
+ getColorModel() : ColorModel
...
FinalClass { leaf }
... // there are methods, but not shown
«interface»
Runnable
run()
AlarmClock
run()
...
an empty compartment without ellipsis means there is definitely no members (in this case, no attributes)

Member details in the POS class diagram



Notation for method bodies in DCDs.


UML notation

A method body implementation may be shown in a UML note box. It should be placed v signifies it is semantic influence (it is more than just a comment).

The synax may be pseudo-code, or any language.

It is common to exclude the method signature (public void ...), but it is legal to include it

```
{
   ProductSpecification spec = catalog.getSpecification(id);
   sale.makeLineItem(spec, qty);
}
```

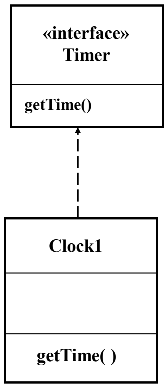
```
{
public void enterItem( id, qty )
{

   ProductSpecification spec = catalog.getSpecification(id);
   sale.makeLineItem(spec, qty);
}
}
```

Interfaces

- The UML has several ways to show interface implementations. Since both the socket line notation and the dependency line notation used curved lines not common in drawing tools, most people use a dependency arrow and the «interface» stereotype.

Interface Example

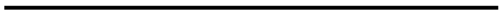


UML stereotypes

- UML uses *stereotypes* encapsulated in guillemots (« and », *not* << and >>) to extend many diagram symbols.
- Guillemots can be found in the **latin-1** symbol set in Windows *Insert Symbol* command)
- Examples: «interface» «extends» «includes» «actor» «creates»

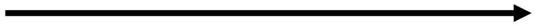
Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another.



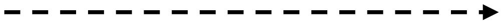
Navigation

- Navigability is a property of the role which indicates that it is possible to navigate uni-directionally across the association from objects of the source to target class.
- Navigability implies visibility.



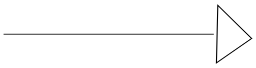
Dependency

- A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it.



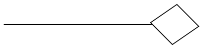
Generalization

- A generalization is a relationship between a general thing (called the super class or parent) and a more specific kind of that thing (called the subclass or child).
- is-a-kind-of relationship.



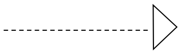
Aggregation

- Whole/part relationship
- Has-a relationship



Realization

- A realization is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
- We use realization in two circumstances:
  - In the context of interfaces.
  - In the context of collaborations.

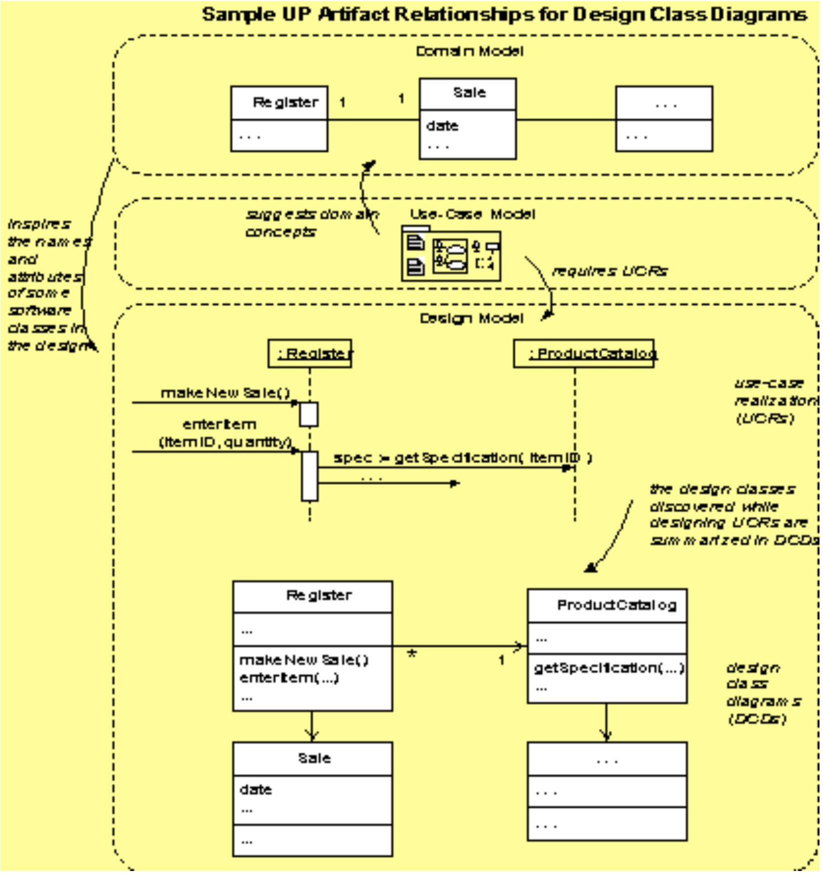


Phases

- Inception

The Design Model and DCDs will not usually be started until elaboration because it involves detailed design decisions, which are premature during inception.

- Elaboration
  - During this phase, DCDs will accompany the UC realization interaction diagrams; they may be created for the most architecturally significant classes of the design.
  - CASE tools can reverse-engineer (generate) DCDs from source code.
- Construction
  - DCDs will continue to be generated from the source code as an aid in visualizing the static structure of the system.


Sample UP Artifact Relationships for Design Class Diagrams
Domain Model
Register
1
1
Sale
date
...
...
...
...
inspires the names and attributes of some software classes in the design
suggests domain concepts
Use-Case Model
requires UCRs
Design Model
: Register
: ProductCatalog
makeNewSale()
enterItem (itemID, quantity)
spec := getSpecification(itemID)
...
use-case realization (UCRs)
the design classes discovered while designing UCRs are summarized in DCDs
Register
...
makeNewSale()
enterItem(...)
...
*
1
ProductCatalog
...
getSpecification(...)
...
design class diagrams (DCDs)
Sale
date
...
...
...
...
...

Interaction Diagram

Objectives

- Interaction diagrams are created to represent interactive behavior of the system.
- UML defines 2 types of interaction diagrams
  - 1. sequence diagram
  - 2. communication diagram

Decomposition Tools

- Different styles of software development are often characterized by a strong reliance on a particular modeling tool during the design phase.
- Relational Database designers tend to depend heavily on Entity Relationship Diagrams.
- Functional/Procedural designers may use a tool like Function Decomposition Diagrams.
- eXtreme Programming designers often use Class Responsibility Collaboration Cards.

Object-Oriented Decomposition

- The modeling choice for most object-oriented designers is the Unified Modeling Language.
- The most important activity in object-oriented design is assigning responsibility to objects.

- The preferred tool to assist object-oriented designers in **assigning responsibility to objects** are the two UML interaction diagrams.

Introduction

- Why do objects exist?
  - To perform an activity to help fulfill a system's purpose.
- Interaction Diagrams are used to model system dynamics
  - How do objects change state?
  - How do objects interact (message passing)?

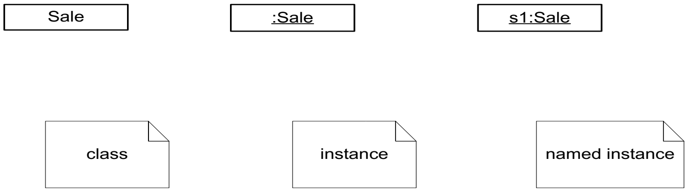
Communication & Sequence Diagrams

- An Interaction Diagram is a generalization of two specialized UML diagram types
  - Communication Diagrams: Illustrate object interactions organized around the objects and their links to each other
  - Sequence Diagrams: Illustrate object interactions arranged in time sequence
- Both diagram types are semantically equivalent, they may not show the same information
  - Communication Diagrams emphasize the structural organization of objects, while Sequence Diagrams emphasize the time ordering of messages
  - Communication Diagrams explicitly show object linkages, while links are implied in Sequence Diagrams

Interaction Diagrams Are Valuable

- Interaction Diagrams provide a thoughtful, cohesive, common starting point for inspiration during programming
- Patterns, principles, and idioms can be applied to improve the quality of the Interaction Diagrams

Common Interaction Diagram Notation



Communication Diagrams

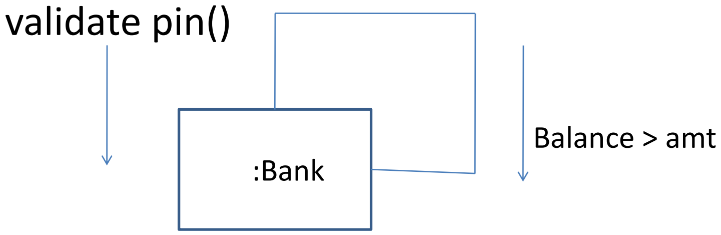
- Objects are connected with numbered (sequenced) arrows along links to depict information flow.
- Arrows are drawn from the interaction source.
- The object pointed to by the arrow is referred to as the target.
- Arrows are numbered to depict their usage order within the scenario.
- Arrows are labeled with the passed message.

Basic Communication Diagram Notation

- Link - connection path between two objects (an instance of an association)
- Message - represented with a message expression on an arrowed line between objects
- Sequence Number - represents the order in which the flows are used
- Conditional Message
  - Seq. Number *[variable = value]* : message()

  - Message is sent only if clause evaluates to *true*
- Iteration (Looping)
  - Seq. Number * *[i := 1..N]*: message()
  - “*” is required; [...] clause is optional

- Self or this : the message can be set for the object itself.



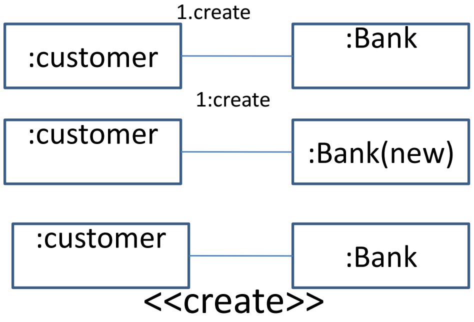
validate pin()

:Bank

Balance > amt

Creation of instances

- Any message can be used to create instance.
- Ex create message which creates an instance.
- Parameters can be passed to create message.
- <<create>> used as a stereotype.



1.create

:customer

:Bank

1:create

:customer

:Bank(new)

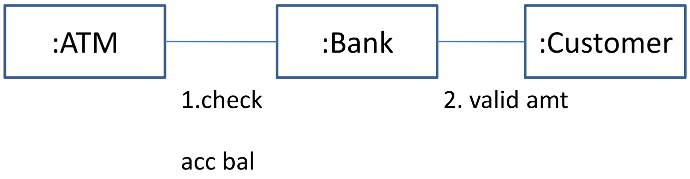
:customer

:Bank

<<create>>

Message numbering scheme

- 1. the first message must not be numbered.
- 2. the order and nesting of message is shown with legal numbering.
- Nesting can be denoted by a perpendicular incoming message number to outgoing message.

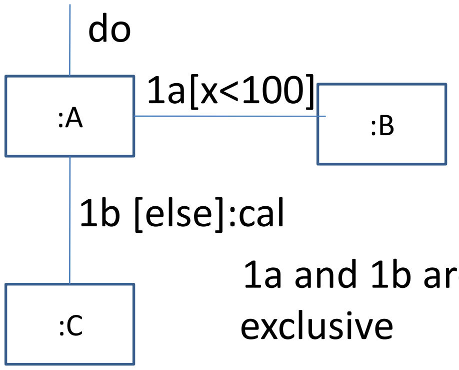


Conditional message

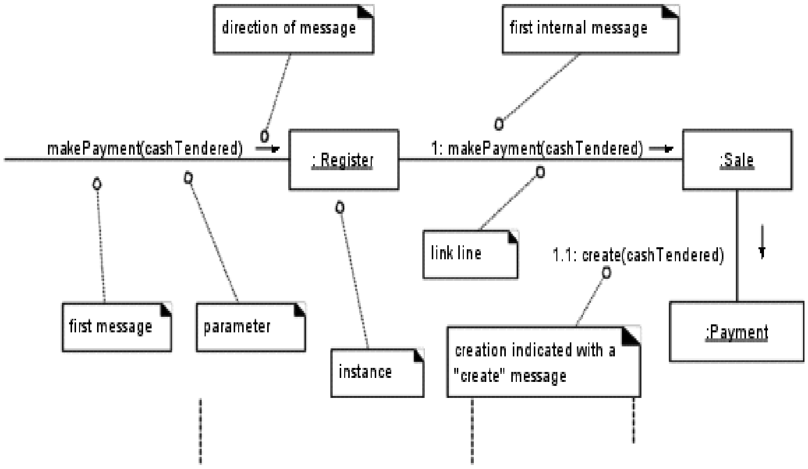
- It is shown in a square bracket.
- The message is sent only when the condition is true.

Mutually exclusive conditional paths

- The conditional expression can be modified using the conditional path.



1a and 1b are mutually exclusive

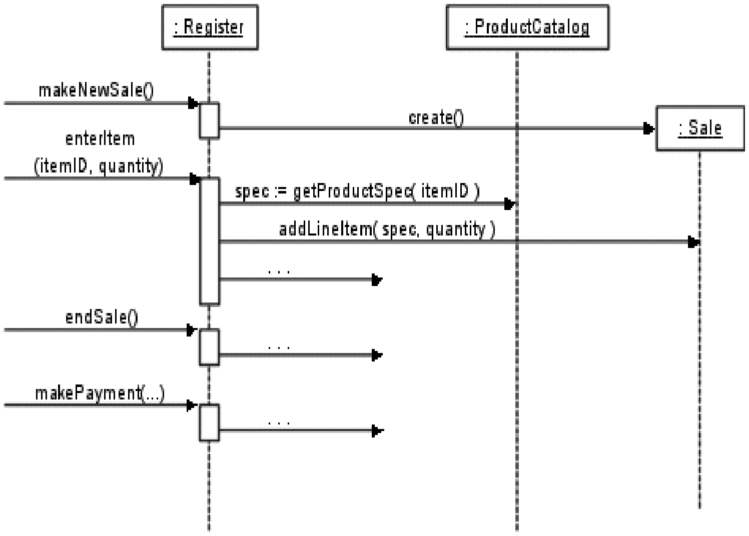

direction of message
first internal message
makePayment(cashTendered)
: Register
1: makePayment(cashTendered)
:Sale
link line
1.1: create(cashTendered)
first message
parameter
instance
creation indicated with a "create" message
:Payment

Sequence Diagrams

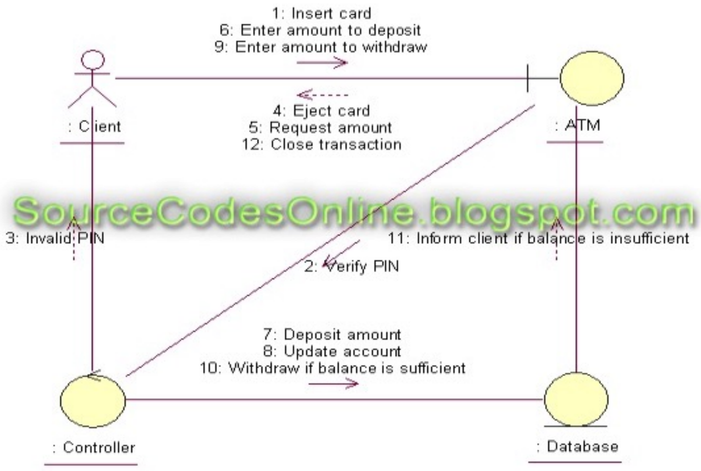
- Correspond to one scenario within a Use Case
- Model a single operation within a System over time
- Identify the objects involved with each scenario
- Identify the passed messages and actions that occur during a scenario
- Identify the required response of each action

Basic Sequence Diagram Notation

- Links - Sequence Diagrams do not show links
- Message - represented with a message expression on an arrowed line between objects
- Object Lifeline - the vertical dashed line underneath an object
    - Objects do not have a lifeline until they are created
    - The end of an object’s life is marked with an “X” at the end of the lifeline
    - Passage of time is from top to bottom of diagram
    - Activation - the period of time an object is handling a message (box along lifeline)
    - Activation boxes can be overlaid to depict an object invoking another method on itself
    - Conditional Message
        - *[variable = value]* message()
        - Message is sent only if clause evaluates to *true*
    - Iteration (Looping)
        - ** [i := 1..N]*: message()
        - “*” is required; [...] clause is optional



Communication diagram ATM



Interaction Diagram Strengths

- Communication Diagram
  - Space Economical - flexibility to add new objects in two dimensions
  - Better to illustrate complex branching, iteration, and concurrent behavior
- Sequence Diagram

    - Clearly shows sequence or time ordering of messages
    - Simple notation

Interaction Diagram Weaknesses

- Communication Diagram
    - Difficult to see sequence of messages
    - More complex notation
- Sequence Diagram
    - Forced to extend to the right when adding new objects; consumes horizontal space

Conclusions

- Beginners in UML often emphasize Class Diagrams. Interaction Diagrams usually deserve more attention.
- There is no rule about which diagram to use. Both are often used to emphasize the flexibility in choice and to reinforce the logic of the operation. Some tools can convert one to the other automatically.