# What will we learn?

Introduction to the Logical Architecture and layers

How to use UML package diagrams

Understanding the static versus dynamic object modeling

# Moving from Analysis to Design

So far we have studied …

- How to create use cases
- How to create a Domain Model
- How to create SSDs and Operations Contracts

These are the main tools in OOA, i.e. they help explore and define *what* the system does

We will now move to design, where we define *how* the system works

Note that this may still be part of the Agile UP process: Analysis and design are occurring at the same time as requirements are refined

# Analysis vs Design

A bad domain model will mean bad communication of the what the system is; a bad design model often leads to a bad implementation

Making the correct choices now becomes more important

- Here is where iterative development can help – constant feedback and requirements enhancement can help the design evolve correctly

Note the initial requirements analysis phase may be very short, and of course coding can start immediately

# Artifacts Interaction

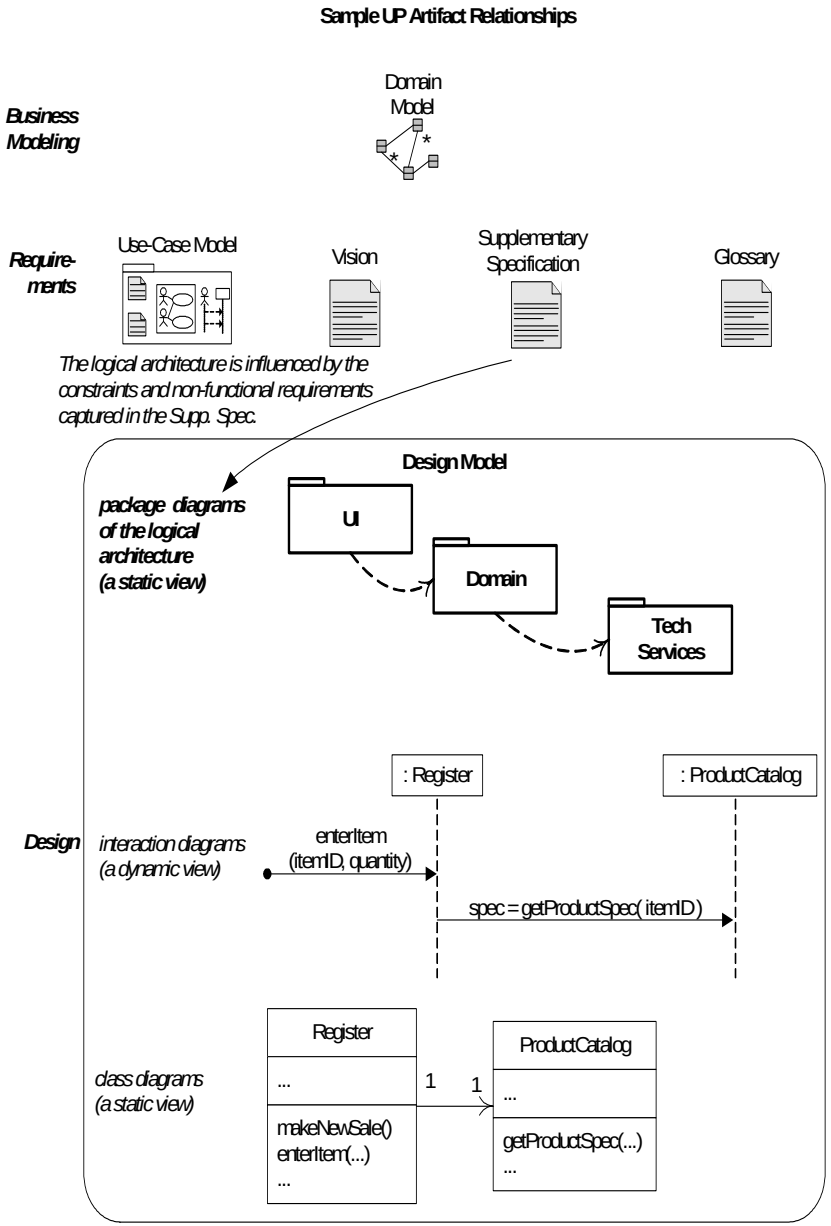
We are now ready to begin to build the Design Model artifact

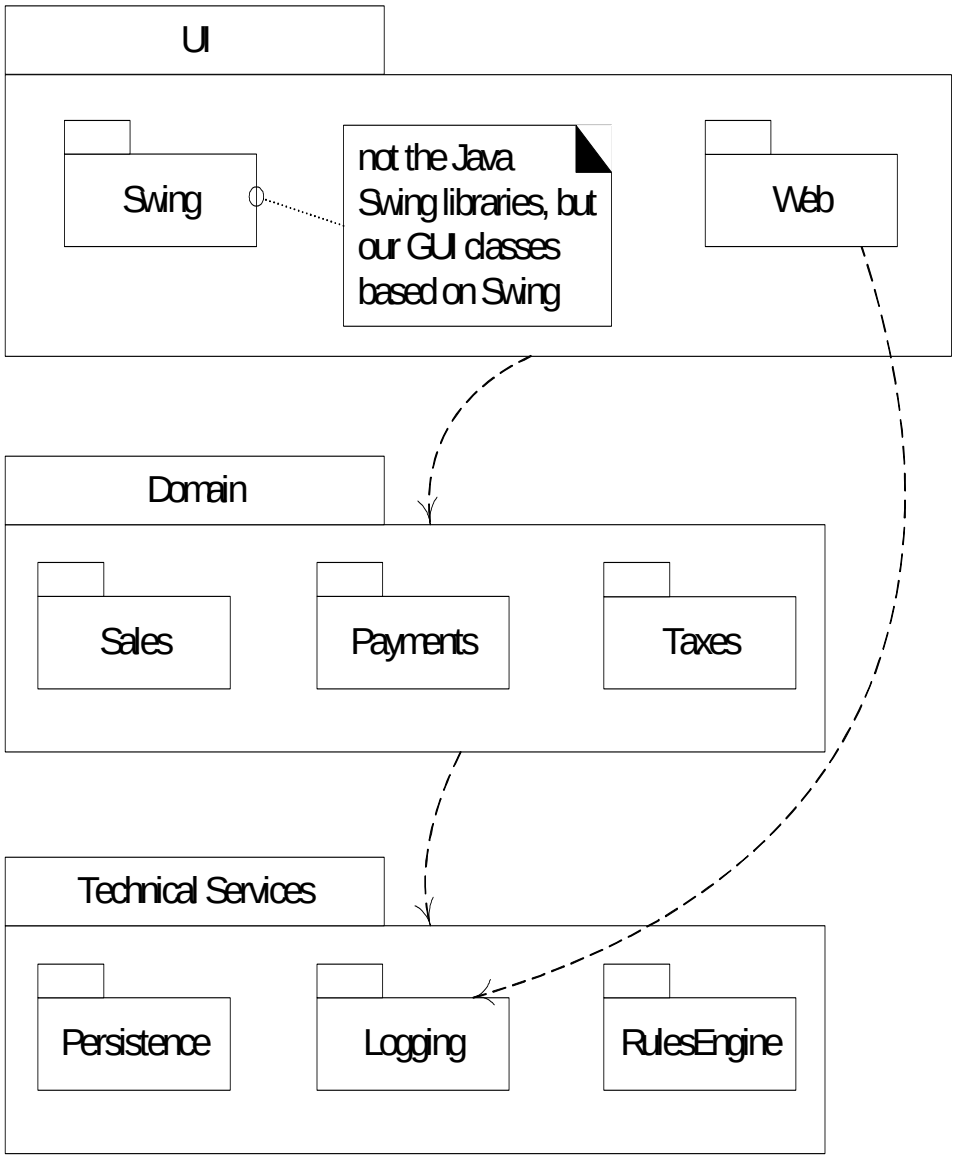
Key influences: Most of the artifacts we have already worked on influence the Design Model:

- Use-Case Model
- Vision
- Supplementary Specification
- Glossary
- Domain Model

The Logical Architecture (LA) is part of the Design Model artifact, and is often shown using package diagrams (we will learn about these shortly)

Note the interaction diagrams (which are dynamic) and class definition diagrams (which are static) are also part of the Design Model artifact


Sample UP Artifact Relationships
Business Modeling
Domain Model
Require-ments
Use-Case Model
Vision
Supplementary Specification
Glossary
The logical architecture is influenced by the constraints and non-functional requirements captured in the Supp. Spec.
Design Model
package diagrams of the logical architecture (a static view)
UI
Domain
Tech Services
Design
interaction diagrams (a dynamic view)
: Register
: ProductCatalog
enterItem (itemID, quantity)
spec = getProductSpec( itemID )
class diagrams (a static view)
Register
...
makeNewSale()
enterItem(...)
...
1
1
ProductCatalog
...
getProductSpec(...)
...


UI
Swing
not the Java Swing libraries, but our GUI classes based on Swing
Web
Domain
Sales
Payments
Taxes
Technical Services
Persistence
Logging
RulesEngine

# Logical Architecture

The Logical Architecture is a large-scale organization of the software classes into packages, subsystems, and layers.

- *Logical*: Implies that there is no decision about how these elements are actually deployed on physical systems, i.e. how many computers will host the system, how they are connected, etc.

The LA organizes the software classes into *layers*, which are just collections of classes

- Each layer is made up of similar classes that have similar responsibilities for a major aspect of the system
- Normally, the higher or upper layers call upon the services of the lower layers, but not viceversa

# Logical Architecture: The Layer Model

| Name | Purpose |
| --- | --- |
| User Interface (UI) | Software objects that directly interact with the user, e.g. GUI interface forms |
| Application Logic and Domain Objects | Software objects representing domain concepts that fulfill application requirements, such as calculating a total |
| Technical Services | General purpose objects and subsystems that provide supporting technical services, e.g. error logging or interfacing to a database. Usually reusable objects that are used across multiple applications |

These are types of layers – the actual model may have more

# Logical Architecture

In a strictly layered model, each layer only calls the services of the layer below it

For information services, the layered model is usually considered relaxed

- For example, the GUI may utilize logging, or a form may directly access a database for a query

We will primarily concentrate on the middle layer, the Domain or Application Logic layer

For UI, we will primarily be concerned with how it interacts with the middle layer

# UML Package Diagrams

We saw this concept when we were developing domain models – it is a way to organize the model into sub-models, with a “folder” notation for easier reading

Often used to illustrate the logical architecture

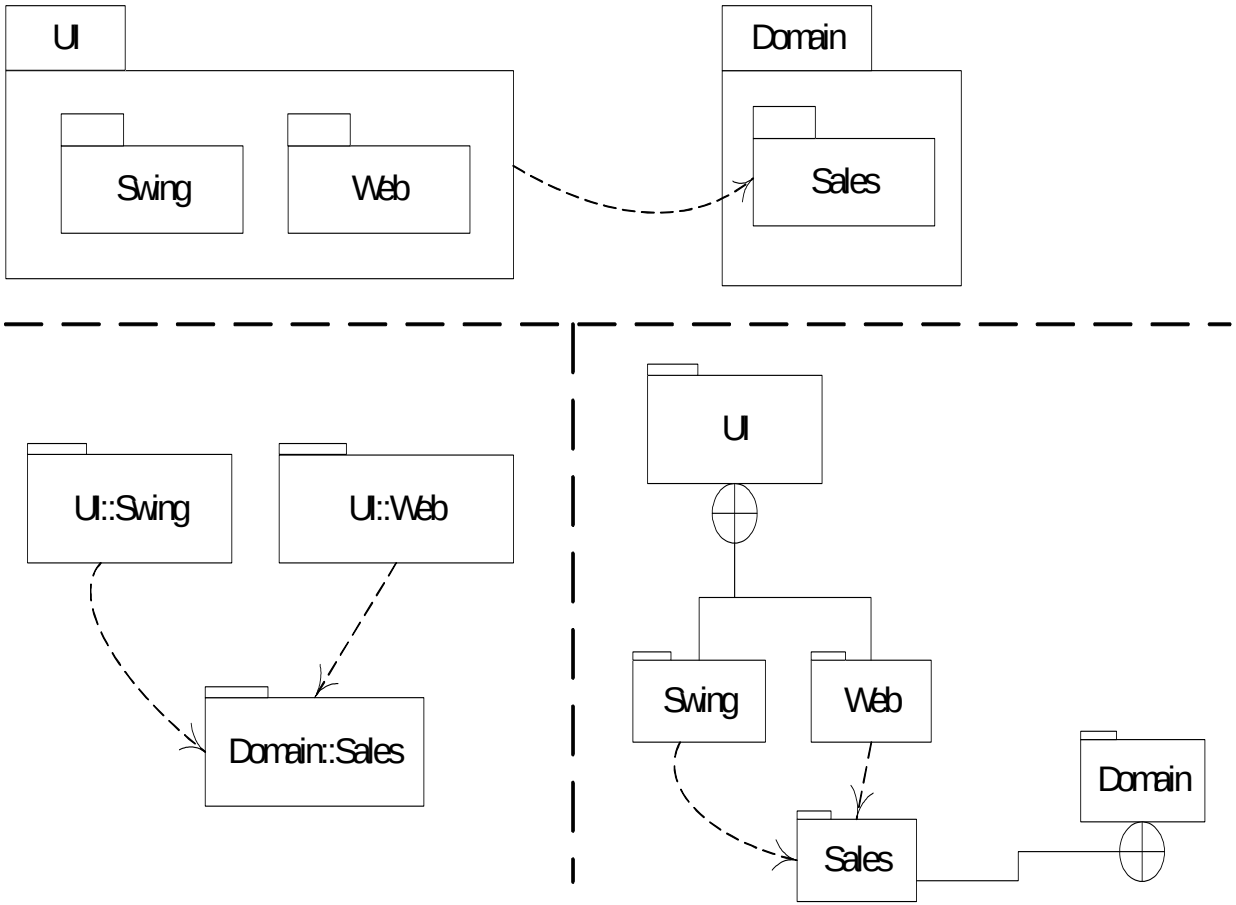
- Model a layer as a package

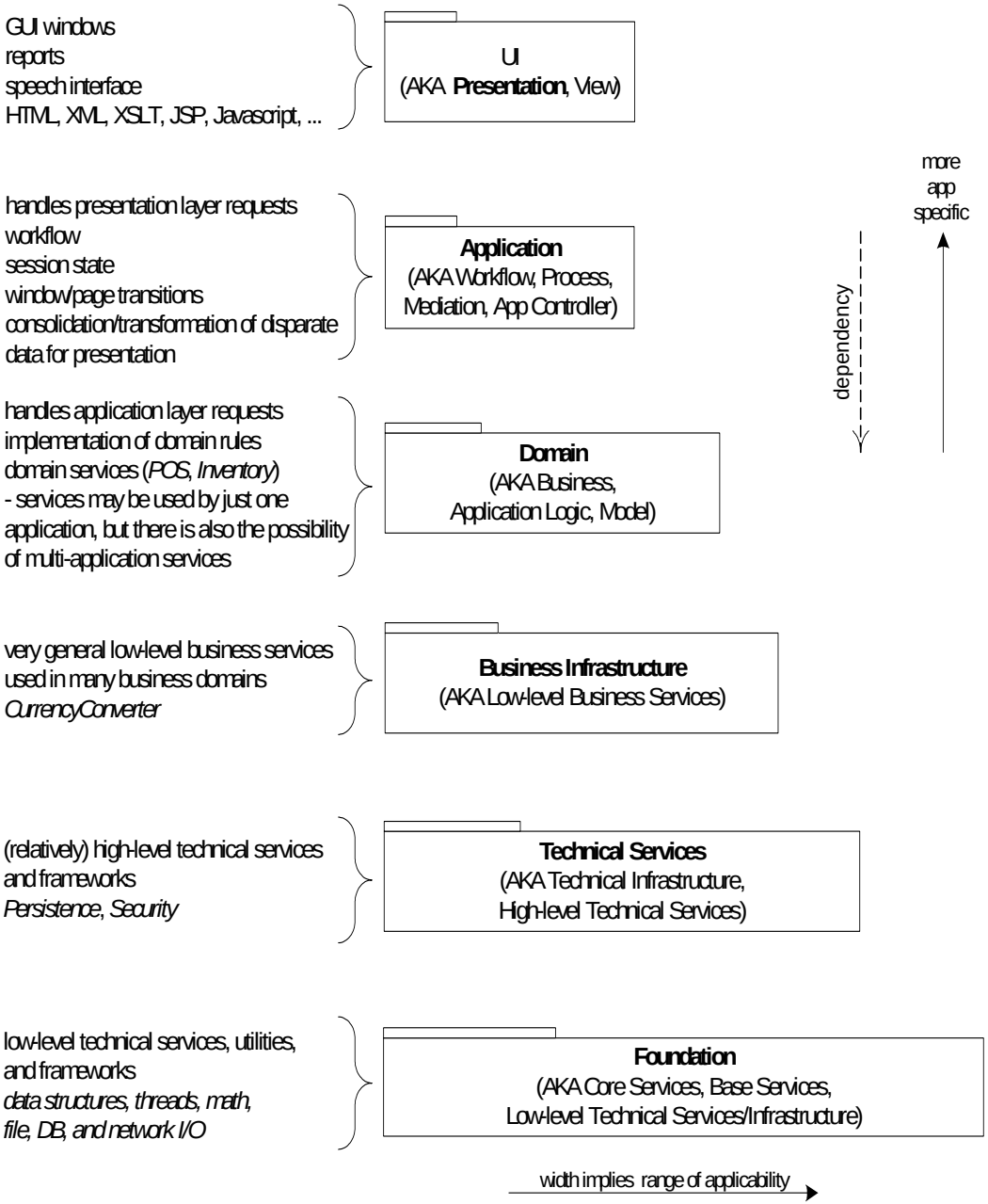
The UML package diagram provides a way to group elements – can be used to group classes, other packages, etc.

Common to show dependency between packages – use a UML dependency line (dashed arrow pointing towards depended-on package)

UML package provides a *namespace*, which allows classes to be defined differently in different packages

Some UML CASE tools will reverse engineer code into packages


UI
Swing
Web
Domain
Sales
UI::Swing
UI::Web
Domain::Sales
UI
Swing
Web
Domain
Sales


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)
more app specific
dependency
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)
width implies range of applicability

# Design with Layers

The key benefit is that objects and classes are separated into cohesive units

This reduces complexity – less coupling of objects

- We will see later that this is a good thing

For us, the key question will be how to design the objects that make up the Application Layer in an OOD manner

**The basic approach is to define software objects with names and information similar to the real-world domain (Domain Model), and assign application logic responsibilities to them**

These types of software objects are called *Domain Objects*

- Represents something in the problem domain space, and has related application of business logic

Often the Application Logic layer is called the Domain Layer for this reason

# Domain Layer and Domain Model

We look to the Domain Model for inspiration for the names and classes that will make up the Domain layer

Note: Not all conceptual classes in the Domain Model need to have classes in the Domain Layer

This is why we do the Domain Model – it helps close the gap between the software system and the real world model it is representing

Also very useful to name objects and associations after their real world counterparts – easier to comprehend the software

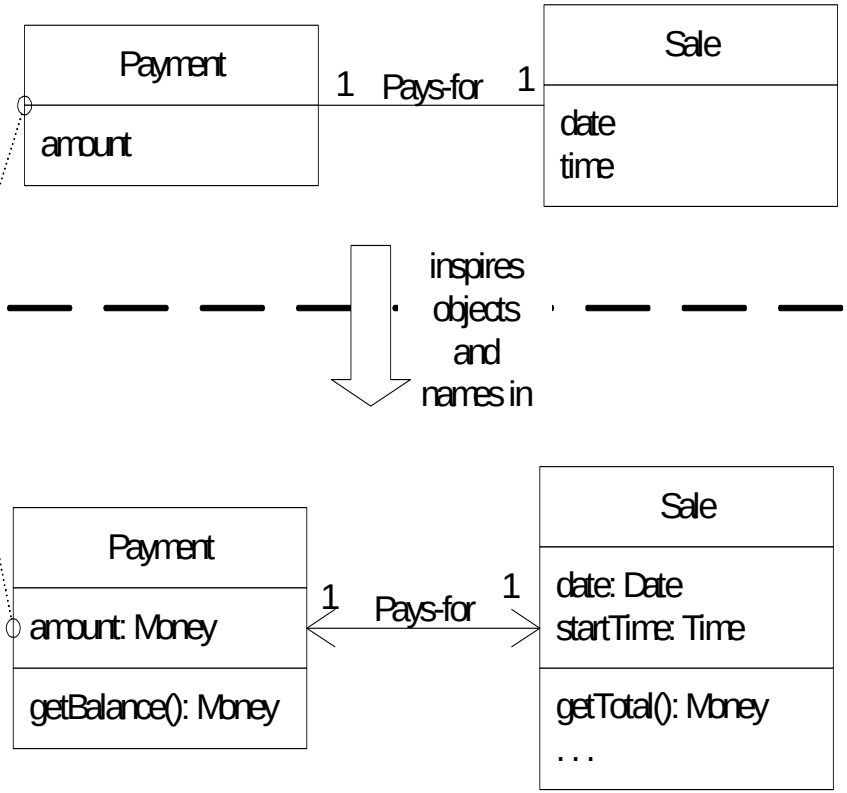
A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain.




Domain layer of the architecture in the UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

# Layer Do's and Don'ts

Don't show external resources (like a database) in a layer at the bottom of the layer diagram – these resources should be included in the layer diagram as separate entities within an existing layer (e.g. the Technical Services)

Model – View Separation: Don't put application logic in UI objects, and don't let non-UI objects reference UI objects

- Traditionally, the term *model* refers to the Domain Layer while the term *view* refers to the UI layer
- Generally, model objects do not have direct knowledge of view objects, but this can be relaxed.
- For example, in the POS, the Sale object should not directly send a message to a GUI window (but of course the other way is okay)
- This helps promote re-use by not connecting domain objects to applications (UI objects)

# SSDs, System Operations, Layers

Recall that SSDs were created from use cases to help explain how the system handles input events from the user

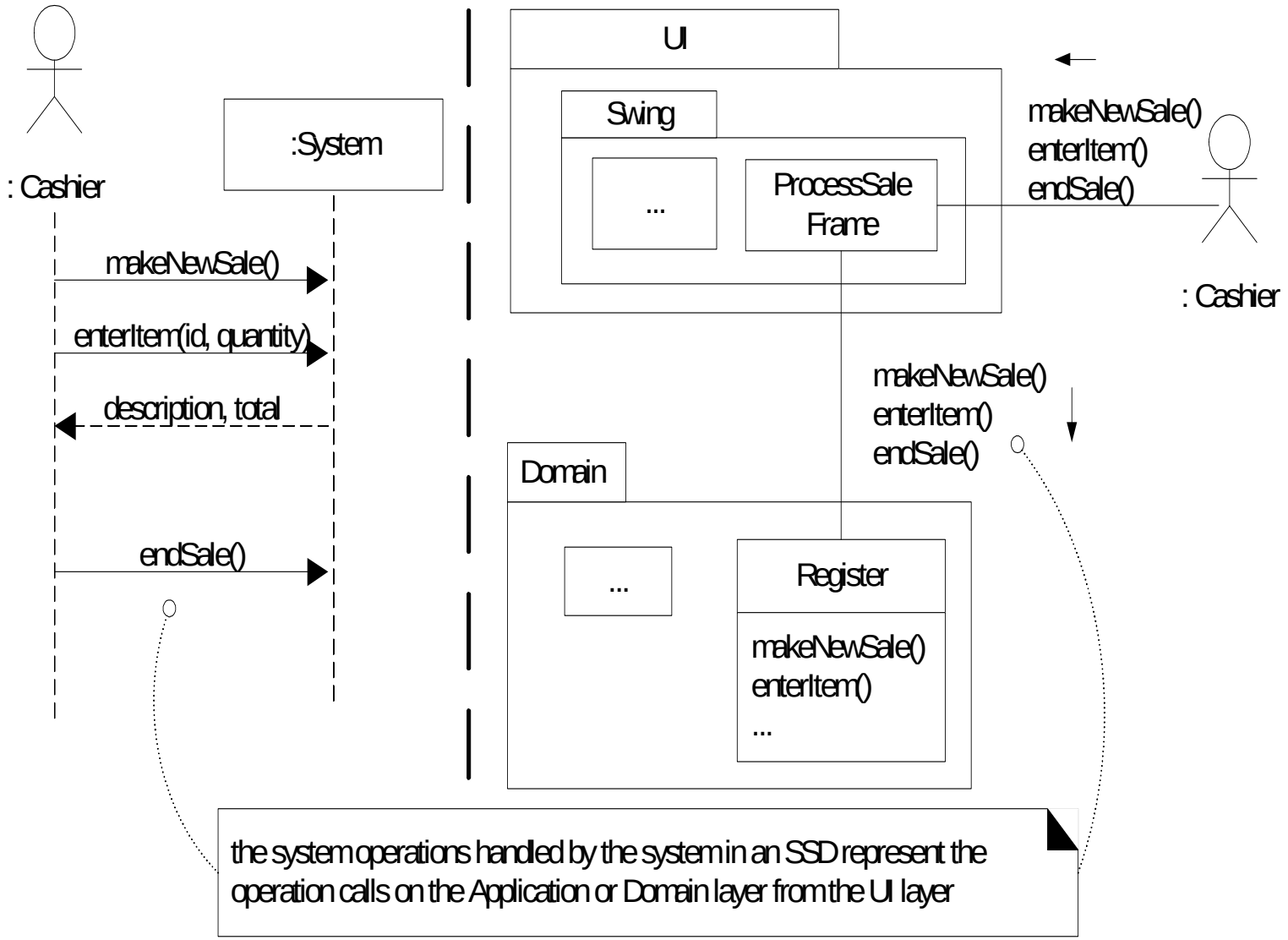
In these diagrams, the internal objects of the UI layer are usually hidden, although in the actual design there will probably be some kind of GUI or web page to gather the input

Normally, these UI objects will then forward the input information to objects in the Domain (or Application) Layer for processing

**These are the system event messages we see in the SSDs**

Example: In our POS system, if the Cashier is entering the items on a terminal (touchscreen or keyboard), there may be a Java Swing object ProcessSaleFrame that captures the information from the Cashier and forwards it to the Register

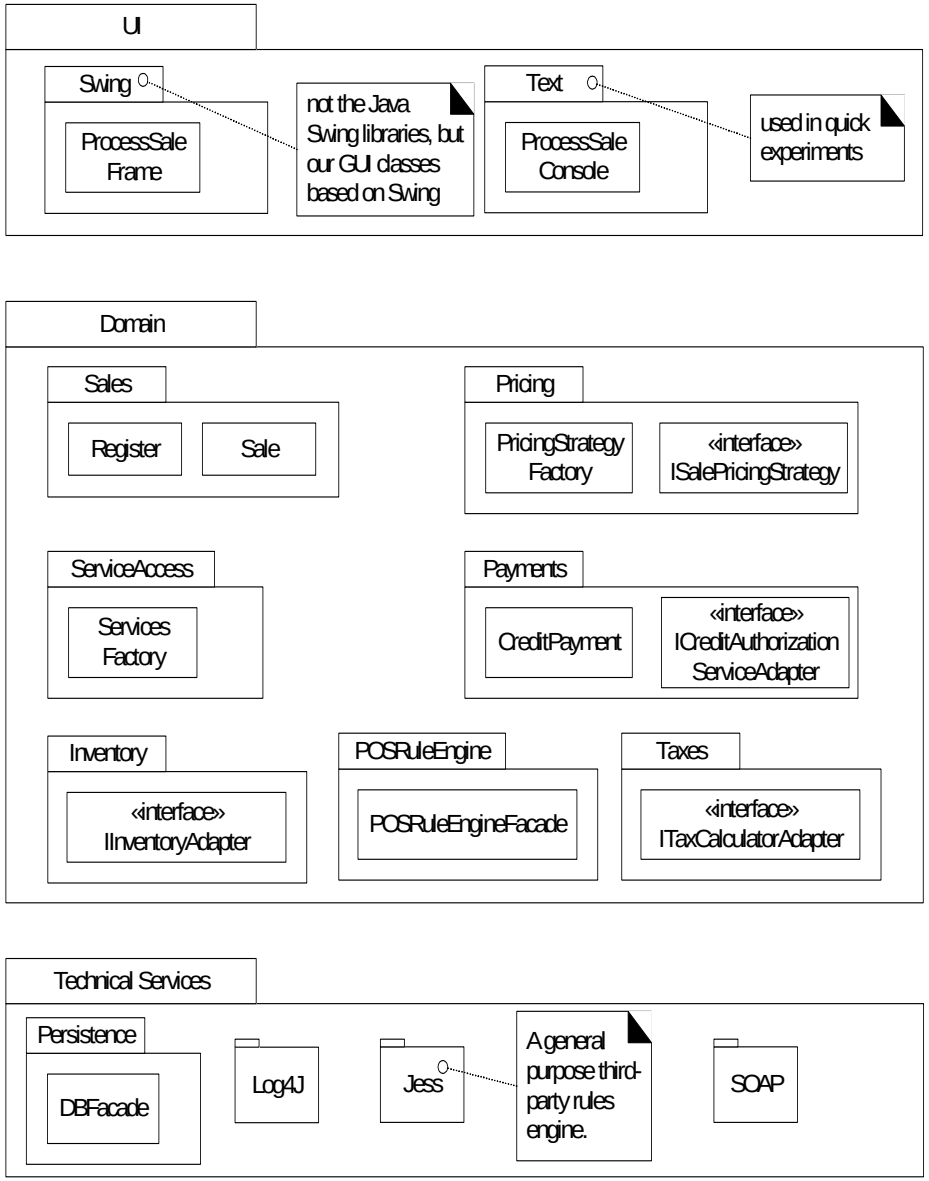
See next slide for a diagram


: Cashier
:System
makeNewSale()
enterItem(id, quantity)
description, total
endSale()
UI
Swing
...
ProcessSale Frame
makeNewSale()
enterItem()
endSale()
: Cashier
makeNewSale()
enterItem()
endSale()
Domain
...
Register
makeNewSale()
enterItem()
...
the system operations handled by the system in an SSD represent the operation calls on the Application or Domain layer from the UI layer

# Logical Architecture for POS and Monopoly

- The preceding diagram suggests a logical architecture for the POS system; we will study this in more detail in Chapter 34 after we expand our knowledge of OOD
- But the LA shown on the previous slide would actually work for the first “cash payment” only iteration of the POS
- For Monopoly, the LA would not really show anything useful –

  It is a very simple UI, Domain, Technical Services architecture
- See next slide for a more detailed view of what the POS LA will look like in our next iterations


UI
Swing
ProcessSale Frame
not the Java Swing libraries, but our GUI classes based on Swing
Text
ProcessSale Console
used in quick experiments
Domain
Sales
Register
Sale
Pricing
PricingStrategy Factory
«interface» ISalePricingStrategy
ServiceAccess
Services Factory
Payments
CreditPayment
«interface» ICreditAuthorization ServiceAdapter
Inventory
«interface» IInventoryAdapter
POSRuleEngine
POSRuleEngineFacade
Taxes
«interface» ITaxCalculatorAdapter
Technical Services
Persistence
DBFacade
Log4J
Jess
A general purpose third-party rules engine.
SOAP

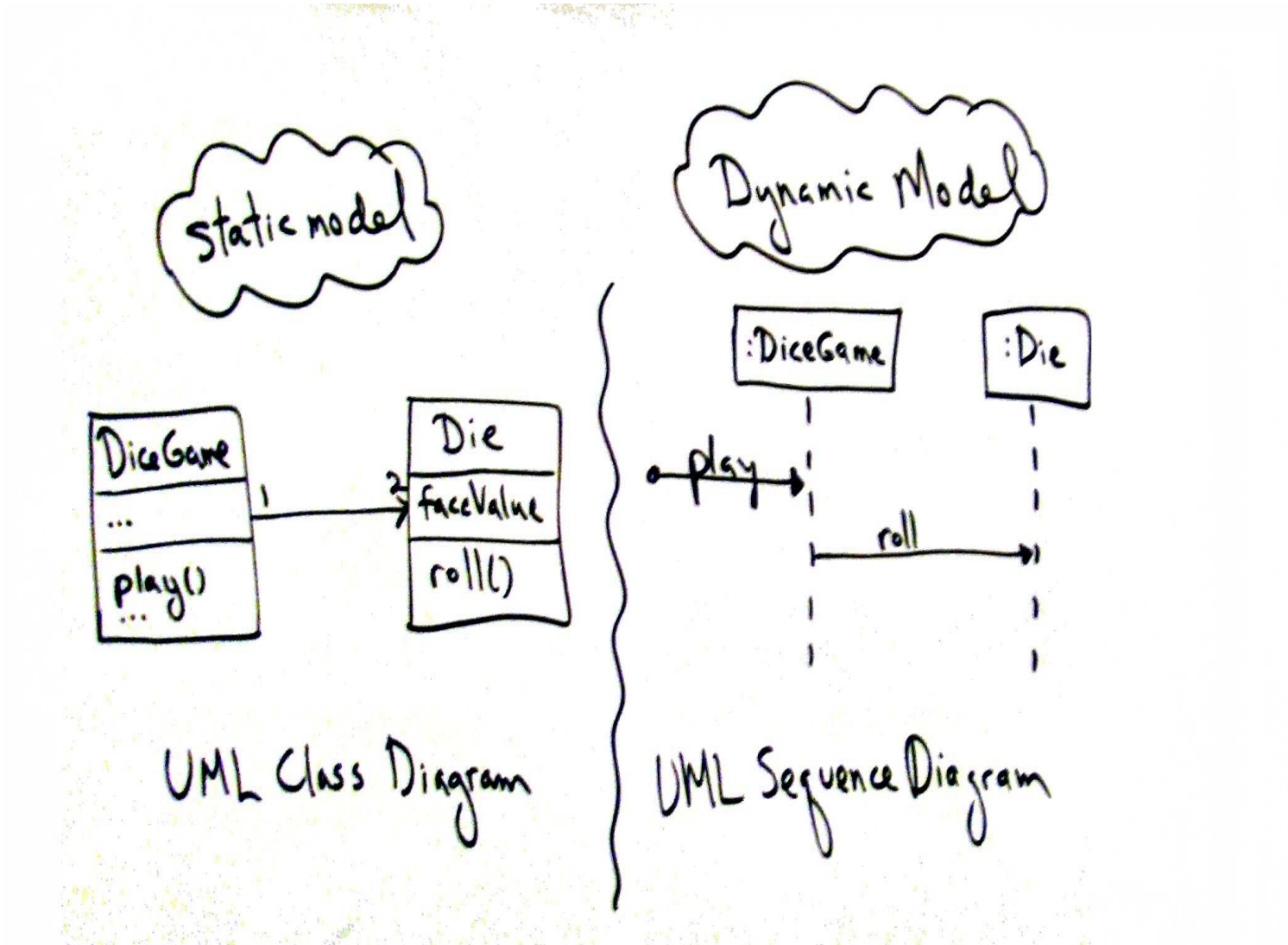
# Designing Objects

- There are two kinds of object models: *dynamic* and *static*
- Dynamic models help design the logic (i.e. the behavior) of the code or methods

  These are the UML interaction diagrams (sequence or communication) we have seen
- Static models help design the definition of packages, class names, attributes, and method signatures  or operations (names and input parameters, return values – not method bodies)

  These are the UML Class diagrams
- Both types of models are usually created together in Agile UP

  Can start with dynamic model and then move to static to fill in details
- Dynamic models tend to be more challenging and useful – they explain the behavior, and will reveal the software objects that need to exist to make the behavior happen, as well as the messages/methods that allow these objects to communicate


static model
Dynamic Model
DiceGame
...
play()
...
1
2
Die
faceValue
roll()
:DiceGame
:Die
play
roll
UML Class Diagram
UML Sequence Diagram

# Designing Objects (cont.)

- It is during the dynamic model creation stage that *responsibility-driven design* principles (like GRASP) are used
- Recall that one the critical aspects of OOD was how to assign responsibilities to the objects in our design – this is essentially what the GRASP principles help with
- **Basic process: For each system event, create software classes (using the Domain Model for inspiration) that will be responsible for handling the event**
- For static models, the most common tool is the UML class diagram

  Often created in parallel with the dynamic models

  Organized into packages, as we have already seen
- Remember, UML is a tool, not the actually design skill. Anyone can draw a UML diagram, but creating a good OO design is more challenging

# Object-Oriented Design: Overview

- Start with the use cases, Domain Model (possibly), SSDs, operations contracts (possibly)
- Select a system operation (from the SSD, possibly detailed in an operations contract)
- Using design patterns (like GRASP or GoF), or general experience, create software classes that will allow the system to handle the responsibilities of the system operation

Usually start with a dynamic model, like an sequence diagram or communication diagram

The operations contract or use case will describe the “responsibilities” – i.e. what the system is supposed to do, or what changes in the system

Use the Domain Model to inspire software objects that will create the desired behavior

# The Model-View-Controller (MVC) Architecture

- Very common logical architecture, similar to what we just saw
- The View is the UI layer – it contains all the GUI objects, and has very little application logic

Passes information down

The Model – this is the layer that contains all of the logical functionality of the system – what it actually does

The Controller – this is the “glue” layer between the View and the Model.

Takes requests from the View, passes to the Model

Takes information from the Model, passes to the View for display

This architecture is popular because it promotes *reuse* of components