So Far …

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code

System Sequence Diagrams

Abdulkareem Alali

ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

In theory, there is no difference between theory and practice. But, in practice, there is.

—Jan L.A. van de Snepscheut

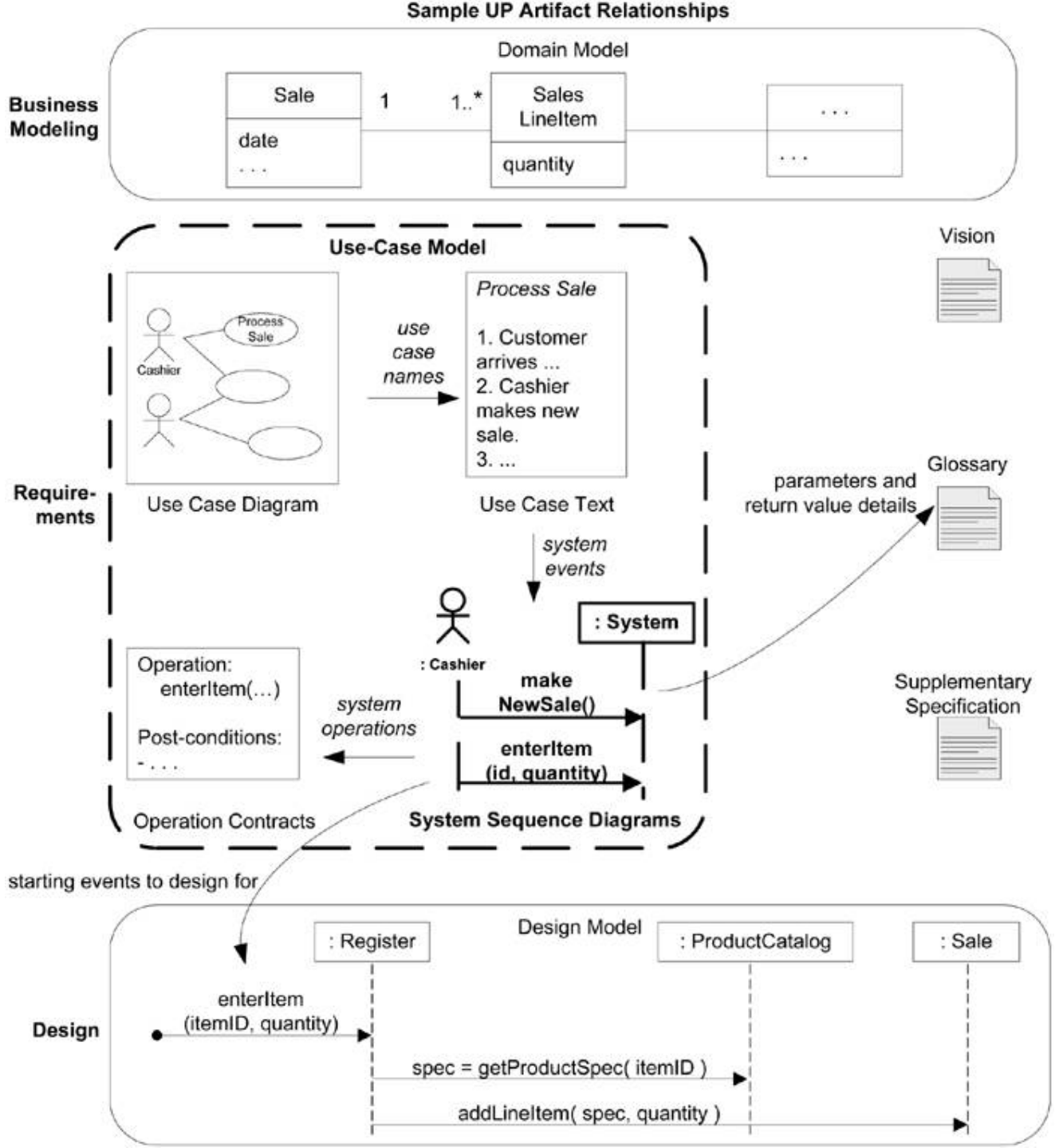
System Sequence Diagrams

A system sequence diagram (**SSD**) is an

artifact that visualizes the input and output events to/from the system under discussion

An SSD can be quickly and easily created

SSDs will be used to help form **Operation Contracts** (Next) for the system


Sample UP Artifact Relationships
Business Modeling
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
Vision
Use-Case Model
Process Sale
Cashier
use case names
Process Sale
1. Customer arrives ...
2. Cashier makes new sale.
3. ...
Use Case Diagram
Use Case Text
Glossary
parameters and return value details
Require-ments
system events
: System
: Cashier
Operation:
enterItem(…)
Post-conditions:
- . . .
system operations
make NewSale()
enterItem (id, quantity)
Supplementary Specification
Operation Contracts
System Sequence Diagrams
starting events to design for
Design
Design Model
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
addLineItem(spec, quantity)

System Sequence Diagrams

For a particular scenario, external actors interact directly with the system, and the system **events** that the actors generate

System is a **Black Box**

- Inner workings cannot be seen (and are not of interest at this point)?

Use cases depict actors interacting with the software system,

Actor generates events to the system

- Usually **requesting some system operation** to **handle the event**

System Sequence Diagrams

A **Cashier** enters an item's ID, cashier is requesting POS system to record that item's sale, an enter item sale event

Enter item sale ***event*** initiates an ***operation*** upon the system

Event or request -driven architectures

Draw an SSD for a main success scenario of each use case

Draw an SSD for complex alternative scenarios

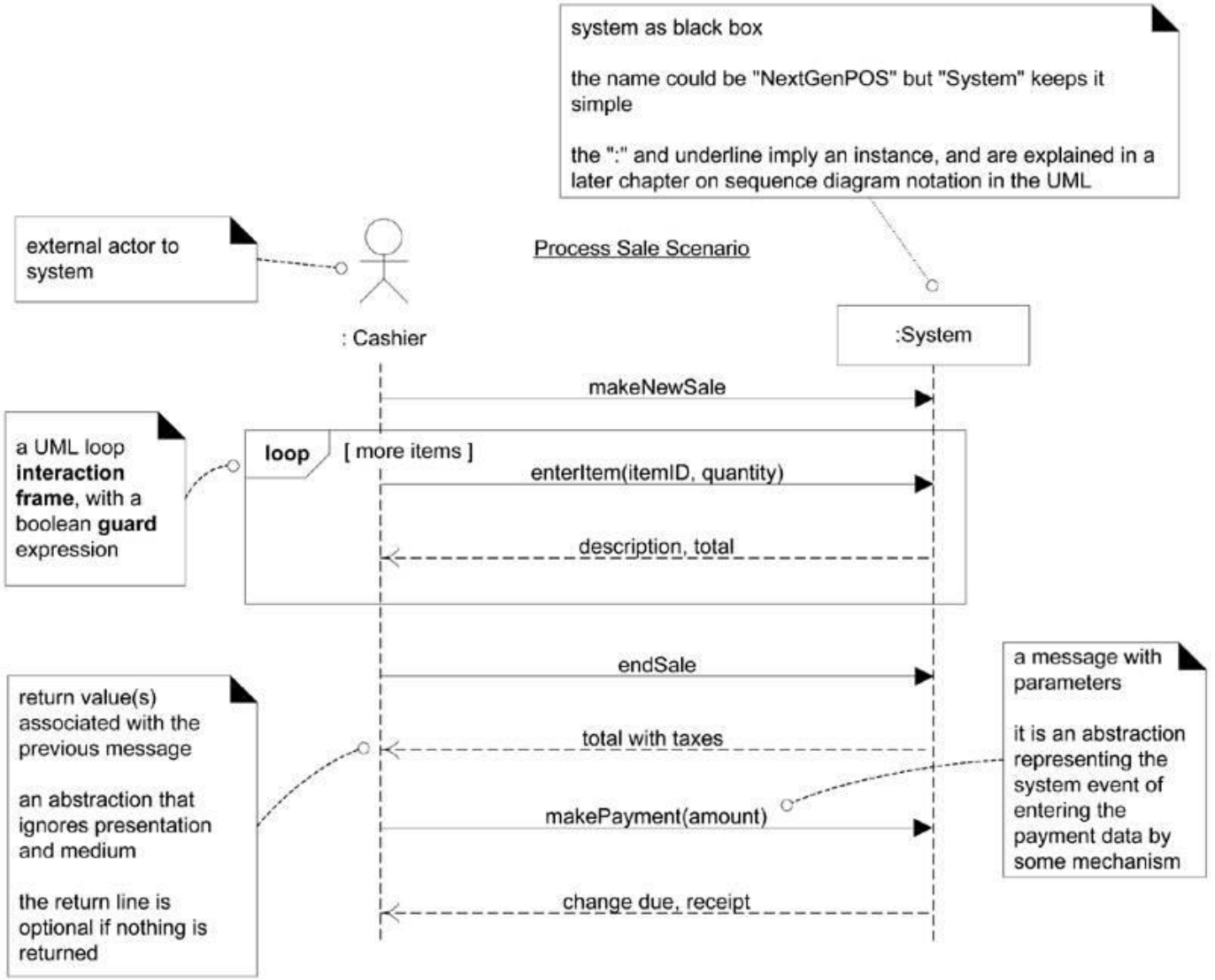
SSD and the POS

The Process Sale SSD depicts the **Cashier** generating:

1. ***makeNewSale***,
2. ***enterItem***,
3. ***endSale***, and
4. ***makePayment*** **system events**

A reading of the use case suggests or implies these events

SSD For A Process Sale Scenario



Why Draw an SSD?

A software system reacts to three things:

- External events from actors, humans or computers
- Timer events
- Faults or exceptions, often from external sources

The system must be able to handle and respond to events it receives

So it is good to know what events the system will receive

At this point we want a description of what a system does, we are not concerned about explaining how it does it, **this is requirements** not design!

UML and Sequence Diagrams

The UML specifies sequence diagrams

System sequence diagrams are not specifically specified

The word "**system**" in the term is to emphasize the application of sequence diagrams to systems that are viewed as **black boxes**

System Sequence Diagrams and Use Cases

Use cases describe users interacting with the system, sending events to the system

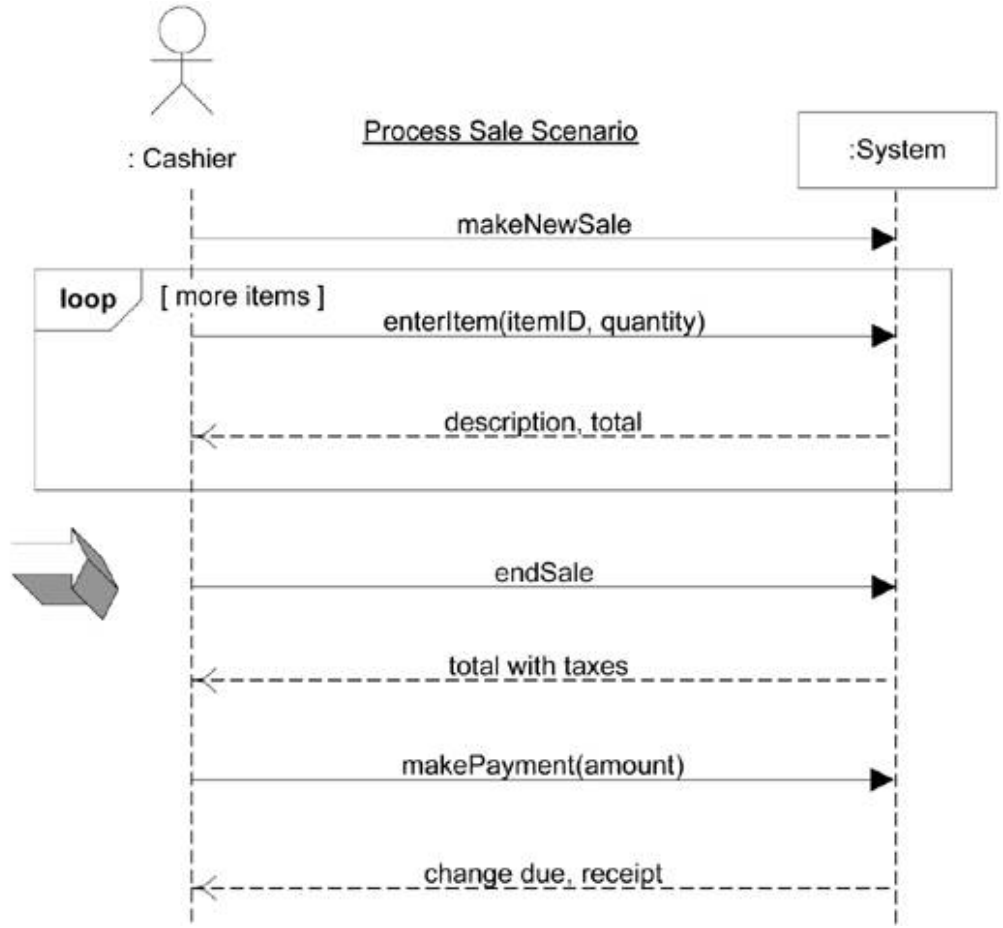
From inspection of a use case the actor and the events can be identified

This interaction can be represented graphically in an SSD

SSDs are derived from use cases; A scenario

Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
...

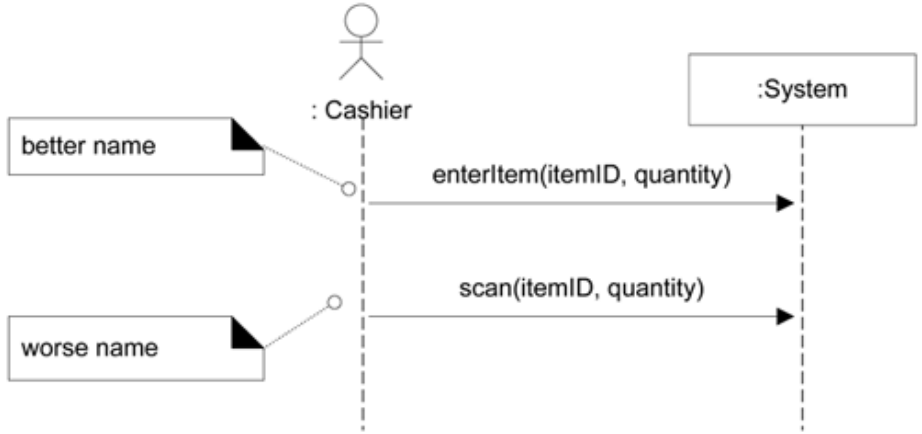


How to Name System Events and Operations

Consider the two event names, Which is better?

scan(itemID) or enterItem(itemID)

enterItem captures intent of the operation and remains abstract and noncommittal, essential style, with respect to design choices about what the UI



Abstract level of intention rather than physical input device

Start the name of a system event with a **verb** (add..., enter..., end..., make...), to emphasize that these are **commands or requests**

How to Model SSDs Involving Other External Systems

SSDs can also be used to illustrate collaborations between systems

At this point, for the POS, this isn't part of the iteration—1 task.

What SSD Information to Place in the Glossary

SSDs (operation name, parameters, return data) is terse

Additional explanation of the terms (parameter types, etc.) may be necessary placed in glossary

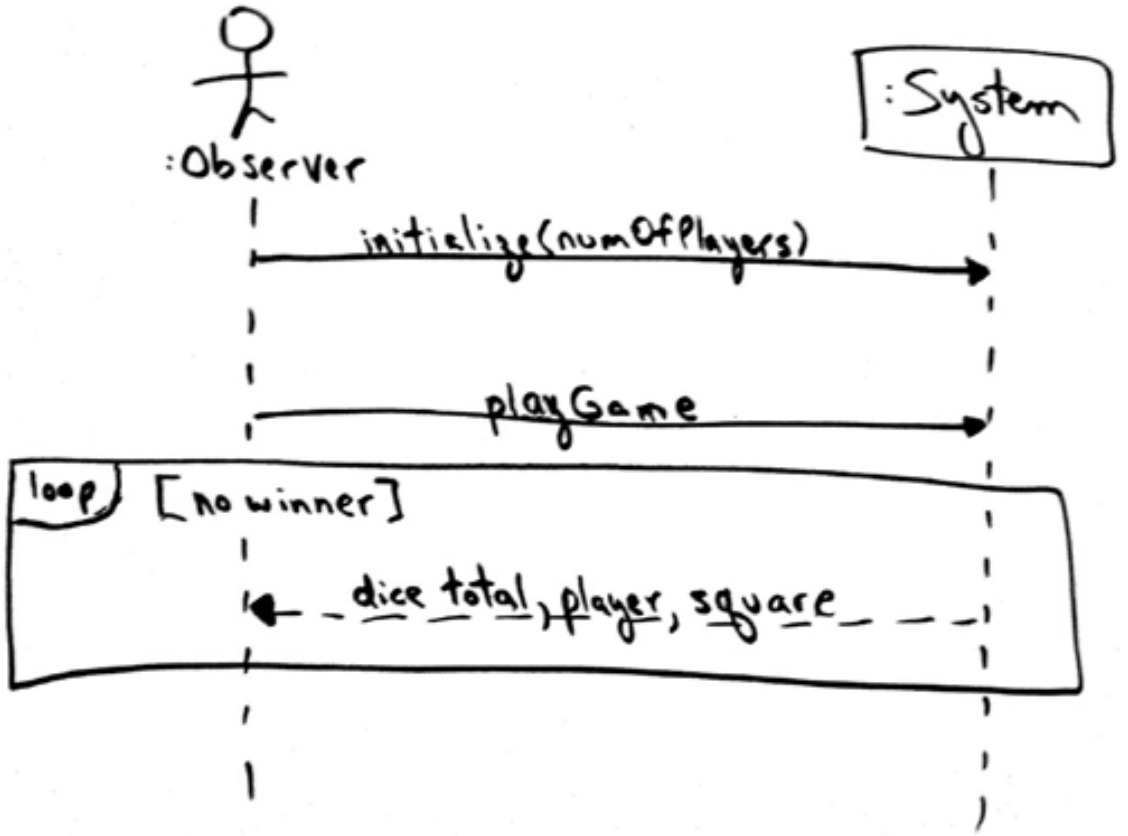
UP Glossary could have a receipt entry that shows sample receipts and detailed contents and layout

For many artifacts the glossary is a good place to show details

Example: Monopoly SSD

main scenario:

Person observes who initializes the play with the number of players, and then requests the simulation of play, watching a trace of the output until there is a winner.



Process: Iterative and Evolutionary SSDs

It is not necessary to create SSDs for all scenarios

Draw SSDs only for the scenarios chosen for the next iteration

SSDs shouldn't take long to sketch, perhaps a few minutes or a half hour

SSDs and the UP

SSDs are part of the **Use-Case Model**

SSDs are a visualization of the **interactions** implied in the scenarios of use cases

SSDs are not usually done during inception; most SSDs are created during elaboration:

- Details of the system events
- Major operations the system must be designed to handle, to write system **operation contracts** (next)
- Possibly to support estimation

Operation Contracts

Abdulkareem Alali

ack Dale Haverstock

Based on Larman’s Applying UML and Patterns Book, 3d

When ideas fail, words come in very handy.

—Johann Wolfgang von Goethe

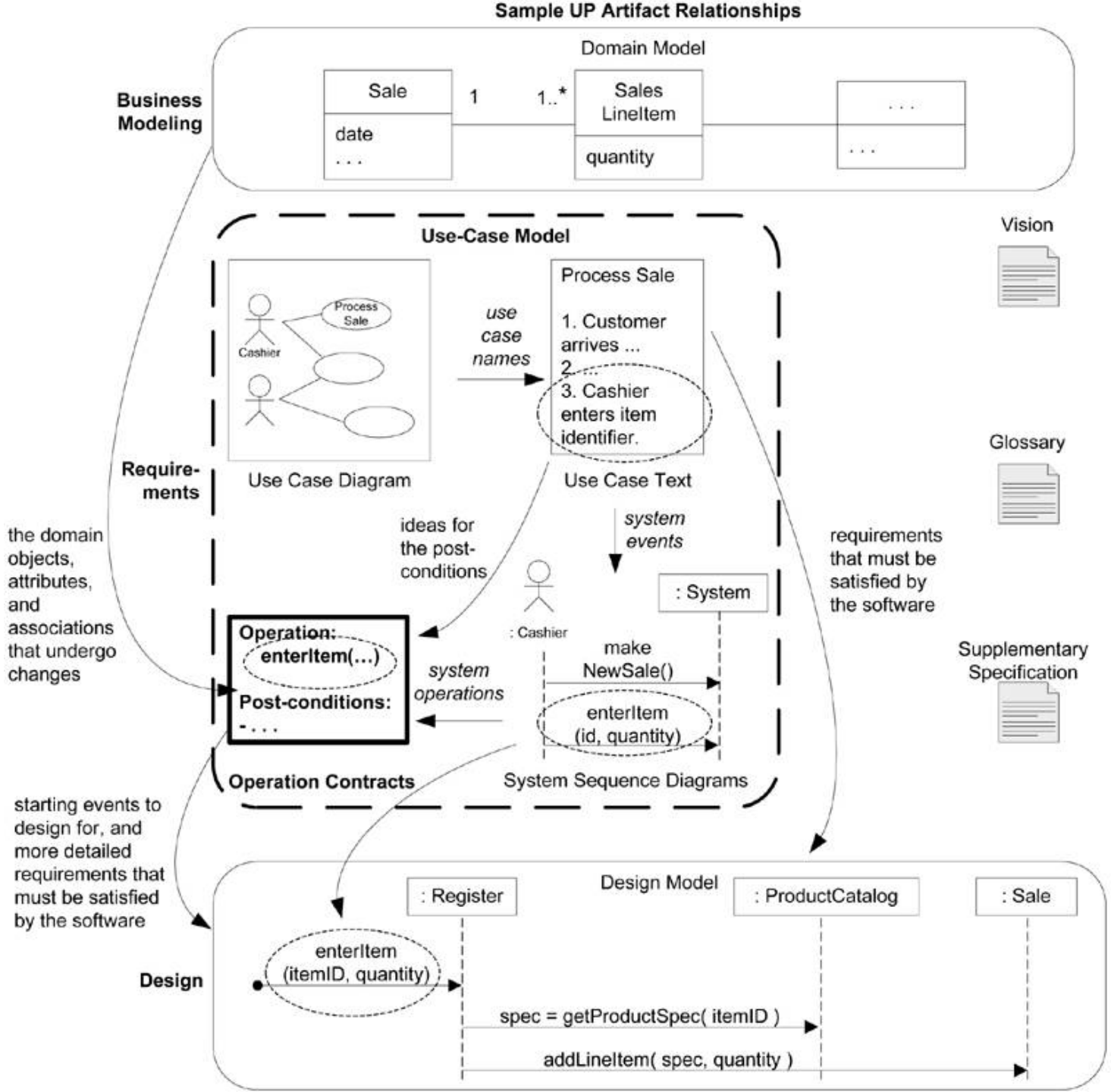
Operation Contracts

In the UP **use cases** or system features are the main ways to describe system behavior. Usually these are sufficient

Sometimes a more detailed or precise description of **system behavior** is useful

Operation Contracts specify a **pre-** and **post-conditions** to describe detailed changes to **objects** (**DM**) that occur as the result of a system operation

Operation contracts may be considered part of the UP Use-Case Model because they provide more analysis detail on the effect of the system operations implied in the use cases, again, what is happening and not how.


Sample UP Artifact Relationships
Domain Model
Business Modeling
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
Use-Case Model
Vision
Process Sale
Cashier
use case names
Process Sale
1. Customer arrives ...
2. ...
3. Cashier enters item identifier.
Glossary
Require-ments
Use Case Diagram
Use Case Text
the domain objects, attributes, and associations that undergo changes
ideas for the post-conditions
system events
requirements that must be satisfied by the software
: System
: Cashier
Operation:
enterItem(...)
Post-conditions:
- . . .
system operations
make NewSale()
enterItem (id, quantity)
Supplementary Specification
Operation Contracts
System Sequence Diagrams
starting events to design for, and more detailed requirements that must be satisfied by the software
Design Model
: Register
: ProductCatalog
: Sale
Design
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
addLineItem(spec, quantity)

The Sections of a Contract

Operation	The name of the operation, and the parameters
Cross References	The use cases this operation can occur within
Preconditions	The noteworthy assumptions about the state of the system or objects in the Domain Model BEFORE execution of the operation
Postconditions	The state of objects in the Domain Model AFTER completion of the operation

What is a System Operation?

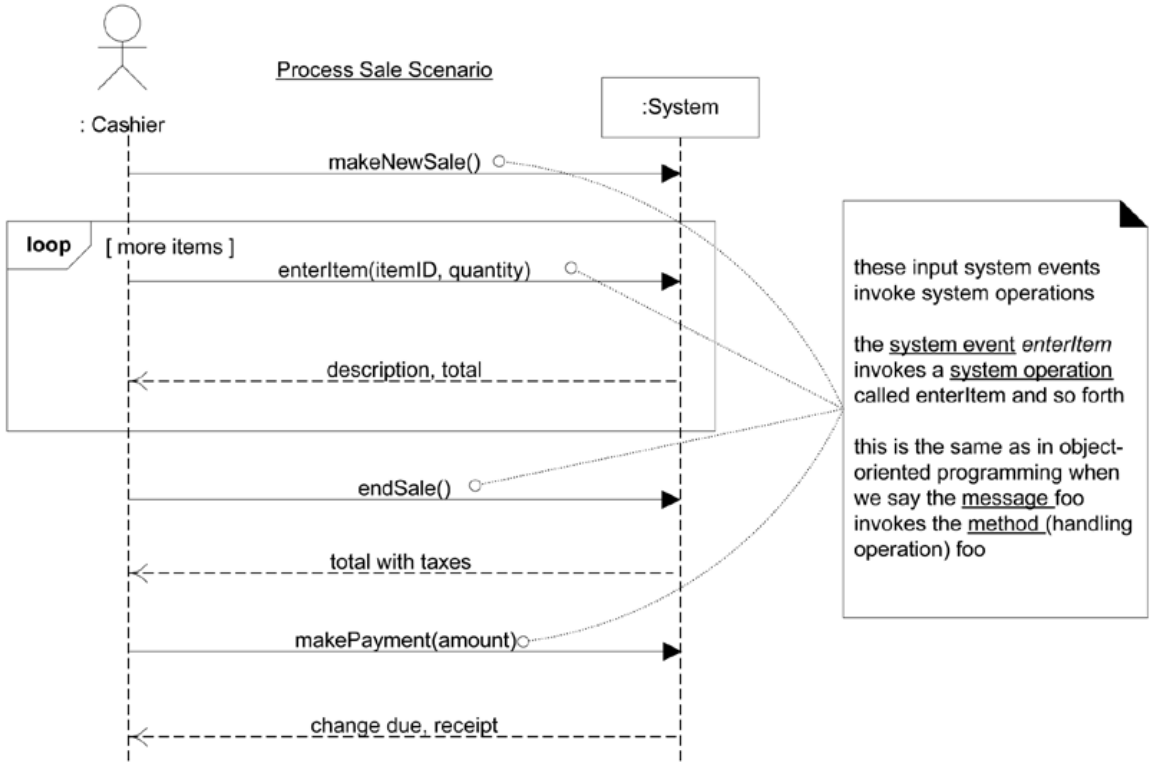
Operation contracts may be defined for **System Operations**, system operations can come from SSDs or use cases

The system is viewed as a single component or class, **Black Box**

The system receives system events or I/O messages, and the system carries out System Operations in response

System Operations, across all use cases, defines the system's public interface, the API!

SSD, System Operations Handle Input System Events



Postconditions

Describes changes in the state of objects in the **domain model** that are guaranteed to be true when an operation has completed

Domain model state changes include (Types of Postconditions):

- **Instances created**,
- **Associations formed or broken**, and
- **Attributes changed**

Breaking associations is not common

- However an example would be the removal of a **SalesLineItem**

Why Postconditions?

Postconditions aren't always necessary

System operation is relatively clear to the developers, with detail and precision

Operation contracts are a way to provide the additional fine-grained detail and precision detail

With contracts the design can be deferred, and focus can remain on the analysis of what must happen, rather than how it is to be accomplished

A Receipt!

Berghotel
Grosse Scheidegg
3818 Grindelwald
Familie R.Müller

Rech.Nr. 4572 30.07.2007/13:29:17
Bar Tisch 7/01

2xLatte Macchiato	à	4.50	CHF	9.00
1xGloki	à	5.00	CHF	5.00
1xSchweinschnitzel	à	22.00	CHF	22.00
1xChässpätzli	à	18.50	CHF	18.50

Total :			CHF	54.50

Incl. 7.6% MwSt 54.50 CHF: 3.85

Entspricht in Euro 36.33 EUR
Es bediente Sie: Ursula

MwSt Nr.: 430 234
Tel.: 033 853 67 16
Fax.: 033 853 67 19
E-mail: grossescheidegg@bluewin.ch

Example: *enterItem* Postconditions

A **SalesLineItem** instance *sli* was created (instance creation)

sli was associated with the current **Sale** (association formed)

sli.quantity became quantity (attribute modification)

sli was associated with a **ProductDescription**, based on *itemID* match (association formed)

Note, it is not stated how a **SalesLineItem** instance is created, or associated with a **Sale**

Writing Postconditions

Guideline: Express postconditions in the past tense to emphasize they are observations about state changes that arose from an operation, not an action to happen

Example:

A **SalesLineItem** was created (better)

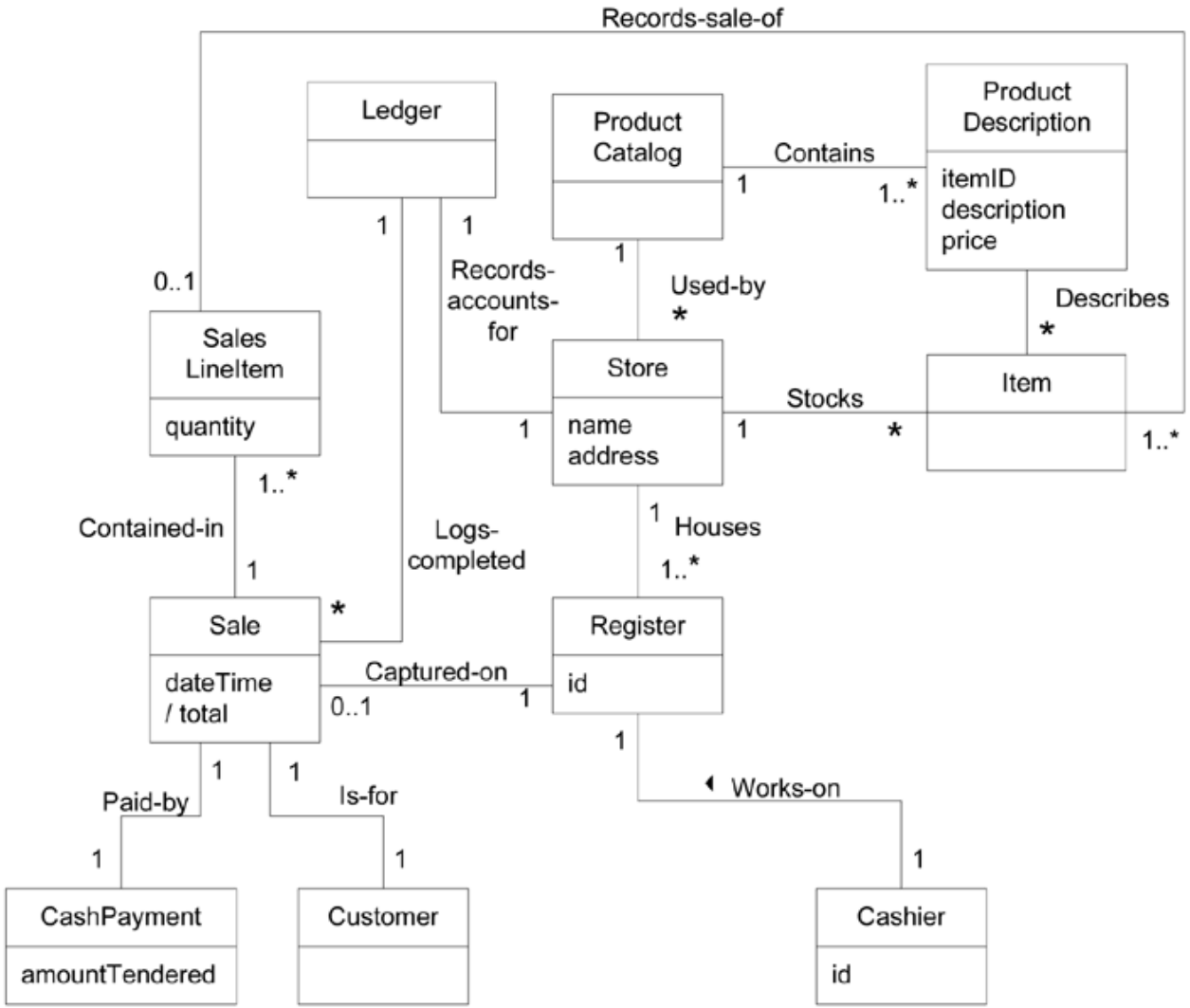
Create a **SalesLineItem** (not as good)

SalesLineItem is created (not as good)

How to Create and Write Contracts

1. From the SSDs identify system operations
2. If system operations are complex or subtle in their results, or which are not clear in the use case, construct a contract
3. Use the following categories to describe the postconditions:
   - Instance creation and deletion
   - Attribute modification
   - Associations formed and broken

POS Partial Domain Model



NextGen POS Contracts —*makeNewSale*

Contract CO1: makeNewSale	
Operation	makeNewSale()
Cross References	Use Cases: Process Sale
Preconditions	none
Postconditions	- A **Sale** instance *s* was created (instance creation) - *s* was associated with a **Register** (association formed) - Attributes of *s* were initialized

NextGen POS Contracts —*enterItem*

Contract CO2: enterItem	
Operation	enterItem(itemID: ItemID, quantity: integer)
Cross References	Use Cases: Process Sale
Preconditions	There is a sale underway.
Postconditions	- A **SalesLineItem** instance *sli* was created (instance creation) - *sli* was associated with the current **Sale** (association formed) - *sli*.quantity became quantity (attribute modification) - *sli* was associated with a **ProductDescription**, based on itemID match (association formed)

NextGen POS Contracts —*endSale*

Contract CO3: endSale	
Operation	endSale()
Cross References	Use Cases: Process Sale
Preconditions	There is a sale underway.
Postconditions	- **Sale**.isComplete became true (attribute modification)

NextGen POS Contracts —*makePayment*

Contract CO4: makePayment	
Operation	makePayment(amount: Money)
Cross References	Use Cases: Process Sale
Preconditions	There is a sale underway.
Postconditions	- A **Payment** instance *p* was created (instance creation). - *p*.amountTendered became amount (attribute modification). - *p* was associated with the current **Sale** (association formed). - The current **Sale** was associated with the **Store** (association formed); (to add it to the historical log of completed sales)

Changes to the POS Domain Model

Object Constraint Language

The UML provides a language for precise specification of contracts

The language is OCL, Object Constraint Language.

Precise specification is useful for automating UML processing

OCL is not normally needed

OCL

The specification can be seen at http://www.omg.org/spec/OCL/2.3.1/

Example: **class invariant**

```
context Student
inv: self.age >= 0
```

Example: **operation contract**

```
context Typename::operationName(parameter1 : Type1, ... ): ReturnType
  pre : parameter1 > ...
  post: result = ...
```

Notes

Complete contracts and postconditions are not necessary

Contracts indicate non-trivial assumptions the reader should be told

If used at all, most contracts will be written during elaboration, when most use cases are written

The **domain model** will likely be modified, when contracts are created it is common to discover new conceptual classes, attributes, or associations in the domain model

Requirements to Design

Abdulkareem Alali

ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

Requirements to Design

Thus far the focus has been on analysis of the requirements

According to UP guidelines, roughly 10% of the requirements have been investigated in inception

Slightly deeper investigation was started in the first iteration of elaboration

Focus shifts on emphasis toward designing a solution for this iteration in terms of collaborating software objects

Requirements to Analysis to Design

Requirements Analysis + **OOA** focus has been on do the right thing. What?

Design work is do the thing right. How?

Transition within an Iteration

Each iteration, a transition from a primarily requirements or analysis focus to a primarily design and implementation focus

Early iterations, relatively, more time analysis activities, later iterations **vision and specifications** will begin to stabilize

Stabilization occurs due to early programming, testing, and feedback

Later iterations, analysis lessens and more focus on building solution

Provoking Early Change

Agile, iterative development change is expected, should not be avoided, especially in the early iterations

Early change, easier to handle, **difficult** issues **resolved**, greater **stability** later iterations, **this idea lies at the heart of why iterative development works**

Elaboration ends, ~80-90% requirements are reliably defined as a result of early programming, testing, and feedback

How Much Time?

Not a lot of time is spent on the modeling

Modeling necessary to achieve the current iteration

Avoid waterfall mode

Logical Architecture

Abdulkareem Alali

ack Dale Haverstock

Based on Larman’s Applying UML and Patterns Book, 3d

Logical Architecture

Think architecture of the system in the early stages

Transition, Analysis to Design, adds software objects

How will the objects be **organized**? Here we focus on **layers** and **logical grouping**, physical grouping is deployment

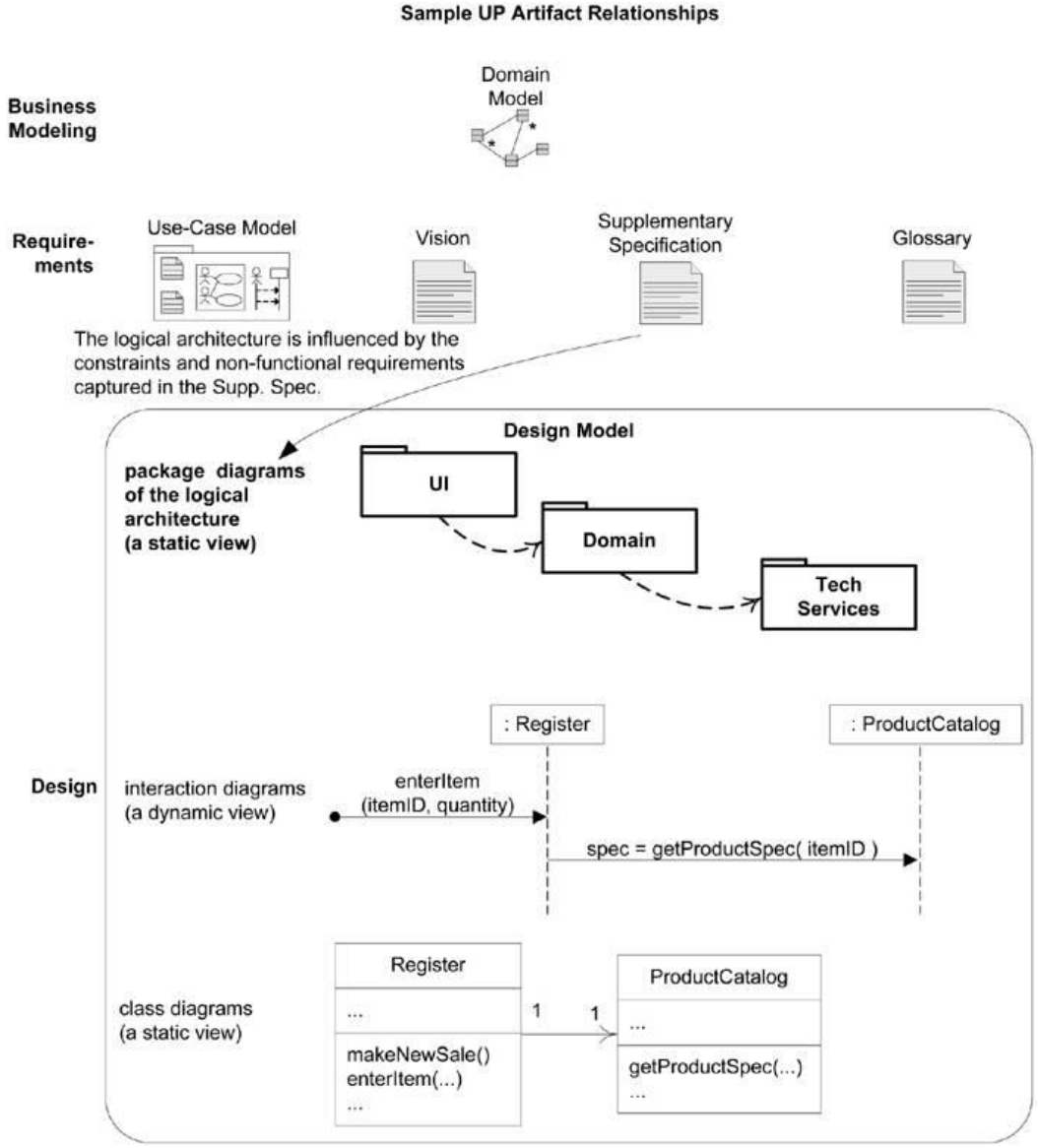
Logical Architecture (LA)

UML package diagrams may be used to illustrate the logical architecture as part of the **Design Model**

Prime input is Supplementary Specification

LA defines packages that holds software classes or sub-packages

UP Artifact Influences LA



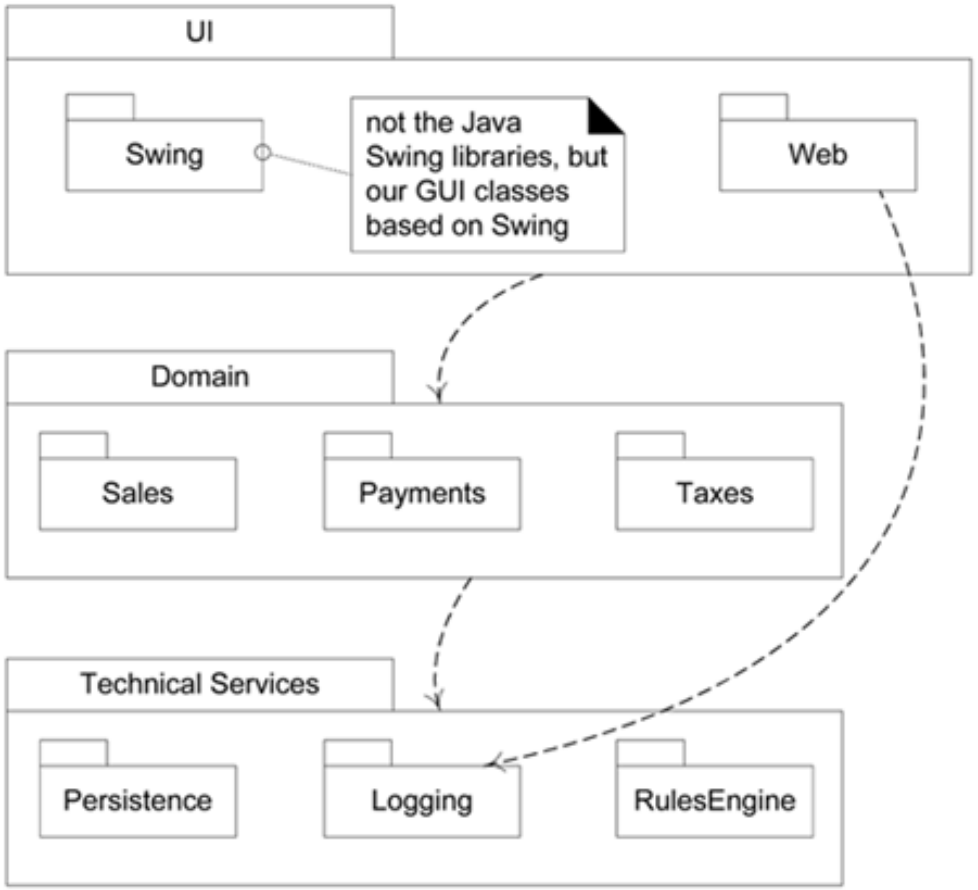
Example

UML package diagrams are drawn as a rectangle with a "tab" along the top, to the left

The tab may contain the package name, or the package name may be placed inside the rectangle

The rectangle may contain other packages, in which case the package name is put in the tab

Partial POS Layers Shown With UML Package Diagram Notation



The Logical Architecture

Is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers

No decision about how the various elements are deployed across different operating system processes or across physical computers in a network, that is deployment architecture

Layers

A **coarse-grained grouping** of classes, packages, or subsystems that has **cohesive responsibility** for a major aspect of the system

Layers rest on other layers, thus there are "higher" and "lower" layers

- **Strict Layered Architecture:** **Typically,** a layer **uses** only layers below itself, Layer only calls upon the **services** of the layer directly below it

Typical Layers

User Interface (UI)

Application Logic and Domain Objects: Software objects representing domain concepts that fulfill application requirements

Technical Services: General purpose objects and subsystems that provide supporting technical services,

for example, database interfaces or logging

LA doesn't have to be organized in layers, but layering is common

Layers in the Case Studies

With the POS example the focus is on the core application logic and so other layers, such as the UI layer, are not focused on

Similarly with the Monopoly example

What is Software Architecture?

Significant decisions about organization of a software system

- Selection and Composition of the structural elements
- Their interfaces, and
- Their behavior/collaborations

  → into progressively larger subsystems

Architectural style that guides this organization

What is Software Architecture?

Main idea:

Big picture,
high-level structure,
organization,
responsibilities,
patterns,
connections, and
constraints, of a system are considered

UML Package Diagrams

UML package diagrams can be used to illustrate the logical architecture of a system

A layer can be modeled as a UML package,

for example, the UI layer modeled as a package named UI

A UML package can group a variety of things including classes, other packages, and use cases

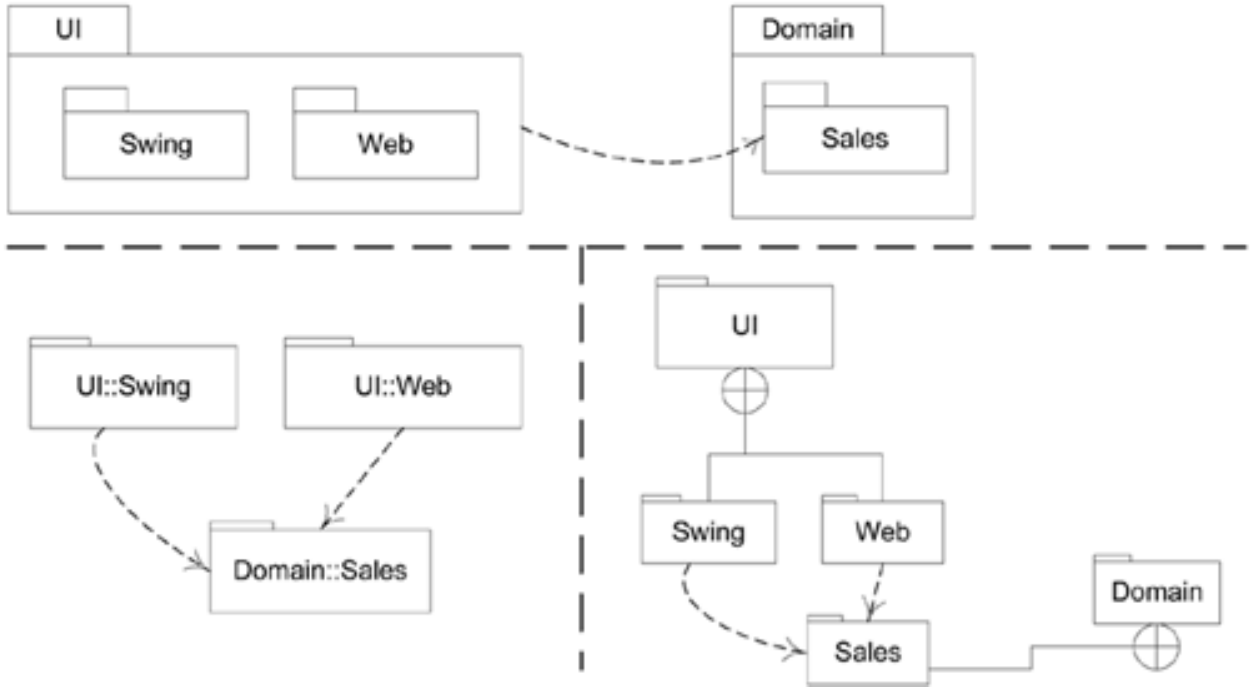
UML Package Diagrams

Dependencies may be shown, as a dashed line with a stick arrowhead pointing towards the dependent item

A line ending with a circle enclosing a cross may indicate a package with the included items at the other end of the line

UML Package Diagrams

- Package nesting using embedded packages
- UML fully-qualified names
- The circle-cross symbol



Using UML Package Diagrams

UML package diagrams may be sketched in **early development**

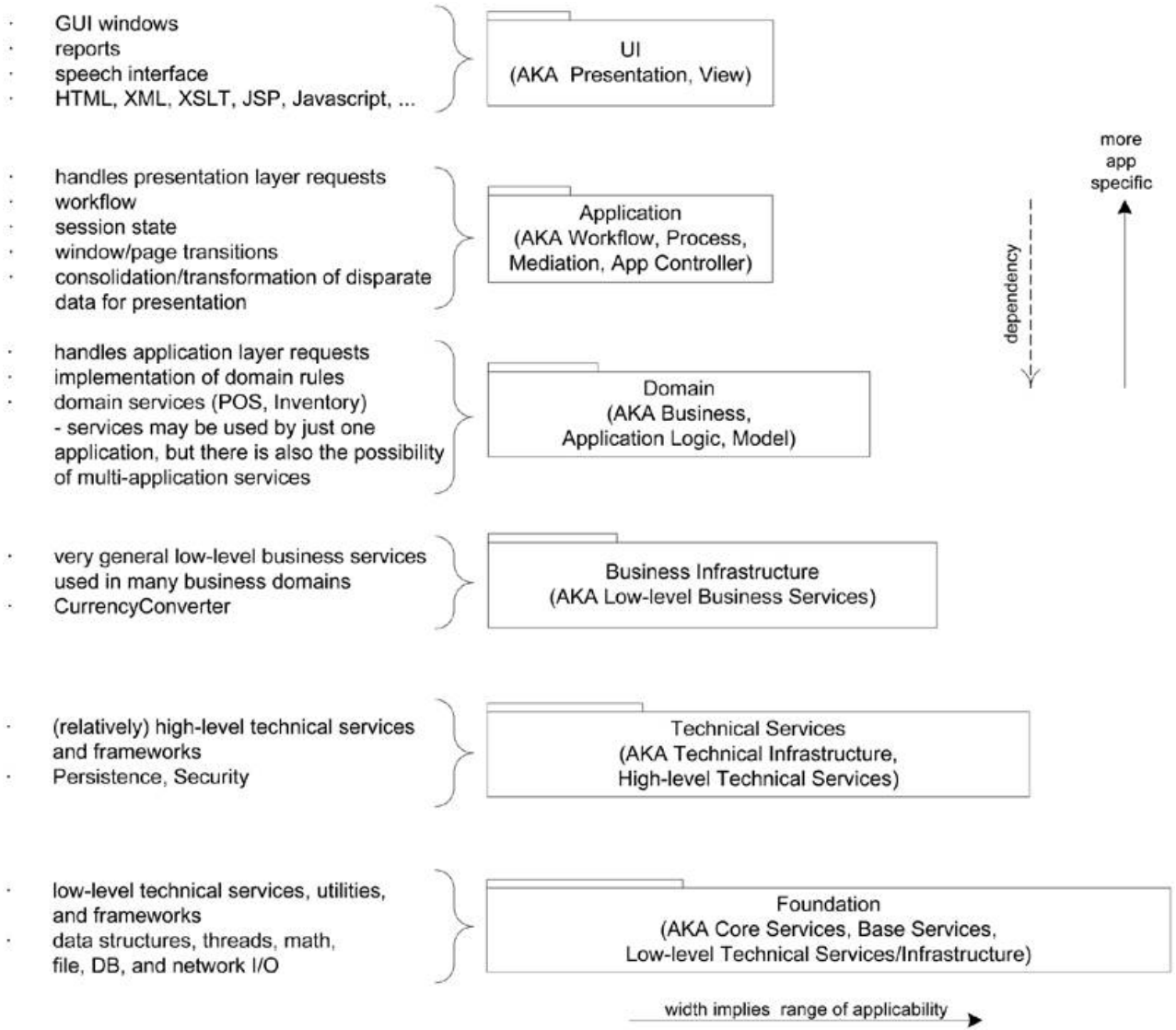
Over time the organization may change

UML CASE tools may be used to **reverse-engineer** the source code and generate a package diagram automatically

Example of Layers

The number of layers varies according to what is needed

Example of Layers



Layers?
—Separation of Concerns

Design with Layers —Separation of Concerns

Organize large-scale logical structure of a system into:

- Discrete and distinct layers
- Each layer items share related responsibilities
- Clean and cohesive
  - Lower layers are general services
  - Higher layers are more application specific

Design with Layers —Separation of Concerns

Cohesion

Higher layers will use lower layers, not the other way around!

Avoid coupling from lower-to-higher layers

Benefits of Using Layers

- Coupling is reduced and managed
- Separation of Concerns
- Cohesion is improved
  - a layer's well-defined responsibility
- Re-usability possibilities are enhanced, a layer can be replaced
- Complexity is encapsulated and decomposable
- Development by different teams is accomplished more easily

Cohesive Responsibilities

Strive for cohesive responsibilities, maintain a separation of concerns

The responsibilities of the elements in a layer should be strongly **related** to each other and

should not be mixed with responsibilities of other layers

Programming languages provide various packaging facilities

Cohesive Responsibilities, Separation of Concerns, Examples

Objects in the UI layer should be concerned with UI-related work

- Creating windows, capturing mouse and keyboard events, etc.

Domain layer objects should focus on **Application Logic**

- Calculating a sales total or taxes, or moving a piece on a game board

Not to violate a clear separation of concerns and the need to maintaining high cohesion—basic architectural principle

- UI objects should not do application logic. Java Swing JFrame (window) object should not contain logic to calculate taxes
- Application logic classes should not trap UI mouse or keyboard events

Mapping Code Organization to Layers and UML Packages

Most popular OO languages (Java, C#, C++, Python, …) provide

support for packages (namespaces in C# and C++)

```
// --- UI Layer
com.mycompany.nextgen.ui.swing
com.mycompany.nextgen.ui.web

// --- DOMAIN Layer
// packages specific to the NextGen project
com.mycompany.nextgen.domain.sales
com.mycompany.nextgen.domain.payments

// --- TECHNICAL SERVICES Layer
// our home-grown persistence
// (database) access layer
com.mycompany.service.persistence

// third party
org.apache.log4j
org.apache.soap.rpc

// --- FOUNDATION Layer
// foundation packages that our team creates
com.mycompany.util
```

Domain Layer vs. Application Logic Layer (DM) Objects

A typical software system has UI logic and application logic, such as GUI widget creation and tax calculations

Create software objects with names and information like the real-world domain (LRG), and assign application logic responsibilities to them

In **POS**, there are sales and payments.

In software, we create a **Sale** and **Payment** classes, and give them application logic responsibilities

This kind of software object is called a domain object

Domain Layer vs. Domain Model

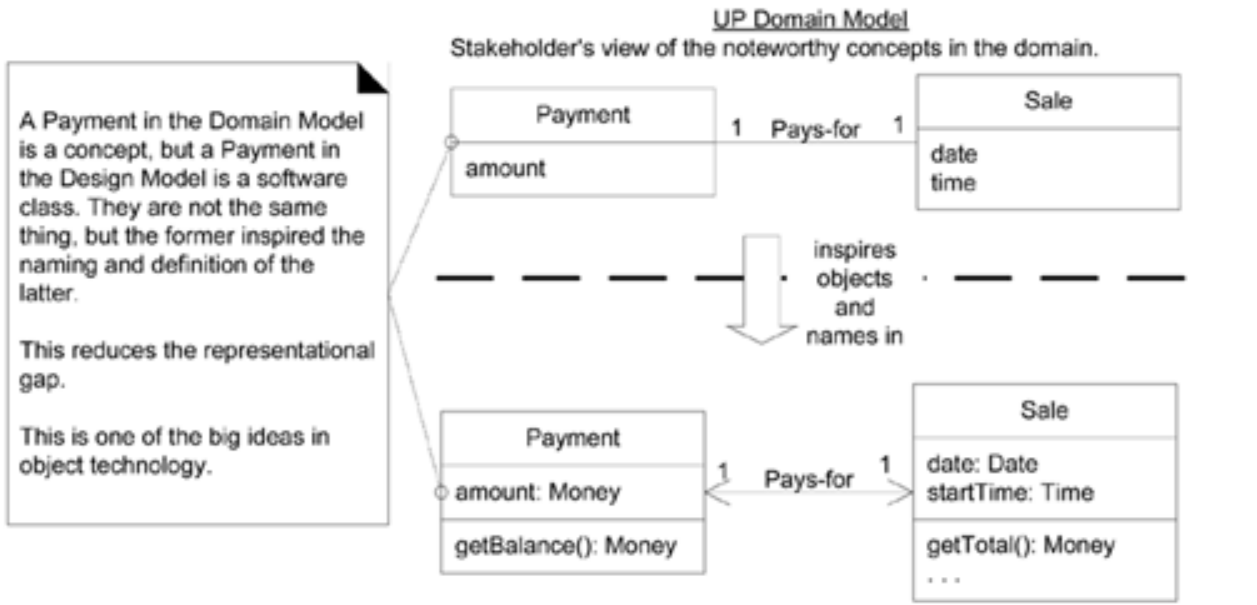
A domain object represents a thing in the problem domain space and has related application or business logic

For example, a **Sale** object can calculate its total

With this approach the application logic layer is more accurately called the domain layer of the architecture

Thus, we have a **domain layer** representing the **domain model** (**LRG**)

Domain Layer And Domain Model Relationship, LRG



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.

Domain Layer, Domain Model

Although they may seem the same, they are not, the domain layer is part of the software and

the domain model is part of the conceptual-perspective analysis

This however gives a lower representational gap (**LRG**) between the real-world objects and the software design

This helps in the **understanding**, **comprehension** of the responsibilities these software objects should have

Tiers, Layers, and Partitions

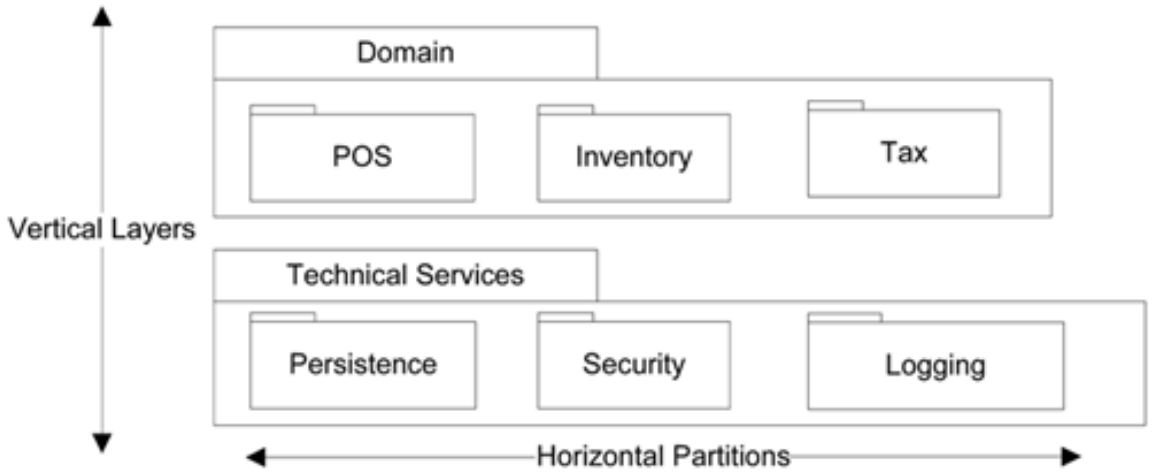
In architecture the term tier originally meant a logical layer

Layers of an architecture are said to represent the vertical slices

Partitions represent a horizontal division of relatively parallel subsystems of a layer

e.g., the Technical Services layer may be divided into partitions such as Security and Reporting

Layers And Partitions



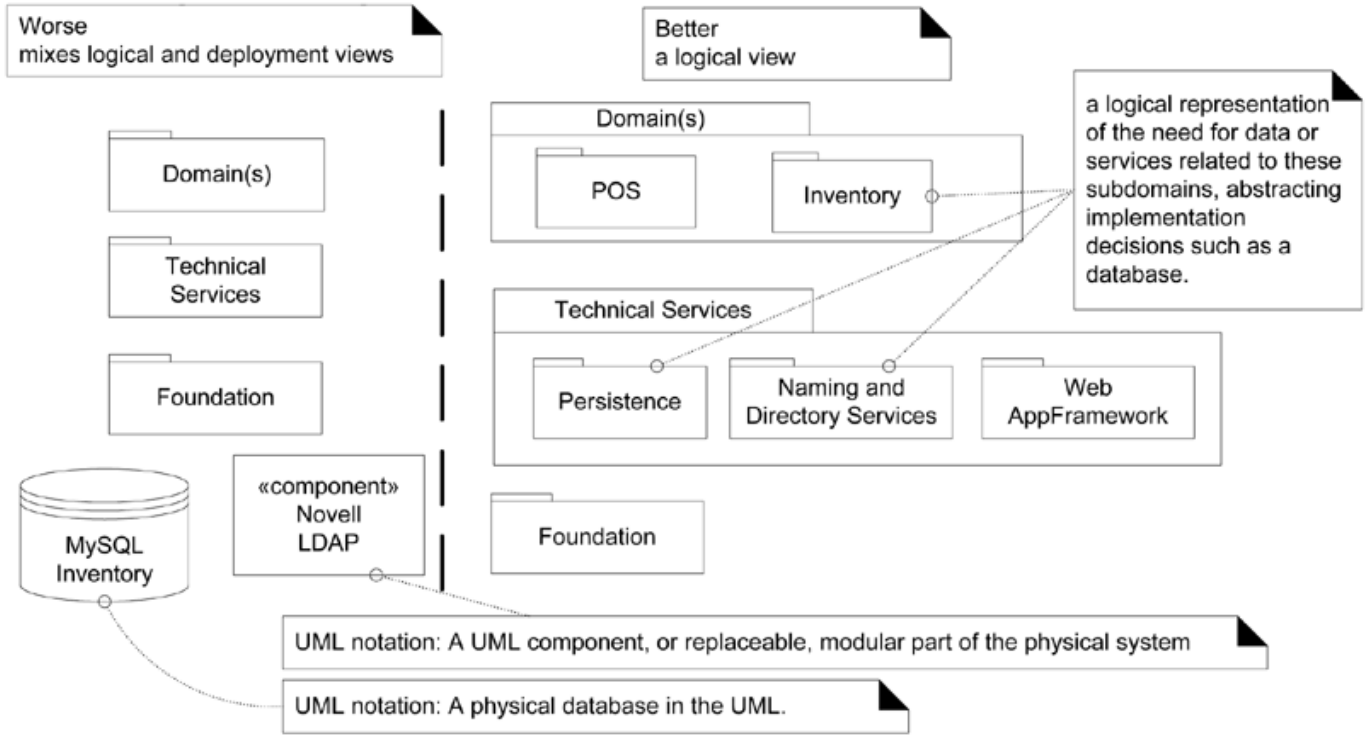
Don't Show External Resources As The Bottom Layer

For example, the application may rely on a databases, however the database is not a bottom layer, it is an external resource

Viewing the database as a bottom layer mixes up the logical view and the deployment view of the architecture

Database is a physical implementation components, not a layer in the logical architecture

Mixing Views of The Architecture



Model-View Separation Principle

The Model-View Separation Principle

The term **model** denotes the domain model elements, objects and responsibilities

The term **view** denotes how the model is "**seen**“, UI

The model and the view should be kept separate, application logic should not be put in a view object

For example, a UI object should not be involved in totaling a sale

The Model-View Separation Principle

The Model-View Separation principle means that model (domain) objects should not have direct knowledge of view (UI) objects

Domain classes encapsulate application logic information and behavior

Window classes are only responsible for **input** and **output**, and catching **GUI events**, they do not maintain **application data** or directly provide **application logic**

The **Model-View-Controller** (**MVC**) architectural pattern is based on the Model-View separation principle and is applicable at many levels (application, component)

The Model-View Separation Principle - Why?

1. To support cohesive model definitions that focus on the domain processes, rather than on user interfaces

2. To allow **separate development** of the model and user interface layers

3. To **minimize the impact of requirement changes** in the interface upon the domain layer

4. To allow **new views** to be easily connected to an existing domain layer, without affecting the domain layer

The Model-View Separation Principle - Why?

5. To allow **multiple simultaneous** views on the same model object, such as both a tabular and business chart view of sales information

6. To allow **execution** of the model layer independent of the user interface layer

7. To allow easy **porting** of the model layer to another user interface framework

SSDs, System Operations, Layers —Connection

SSDs illustrate system operations or events, no specific UI objects

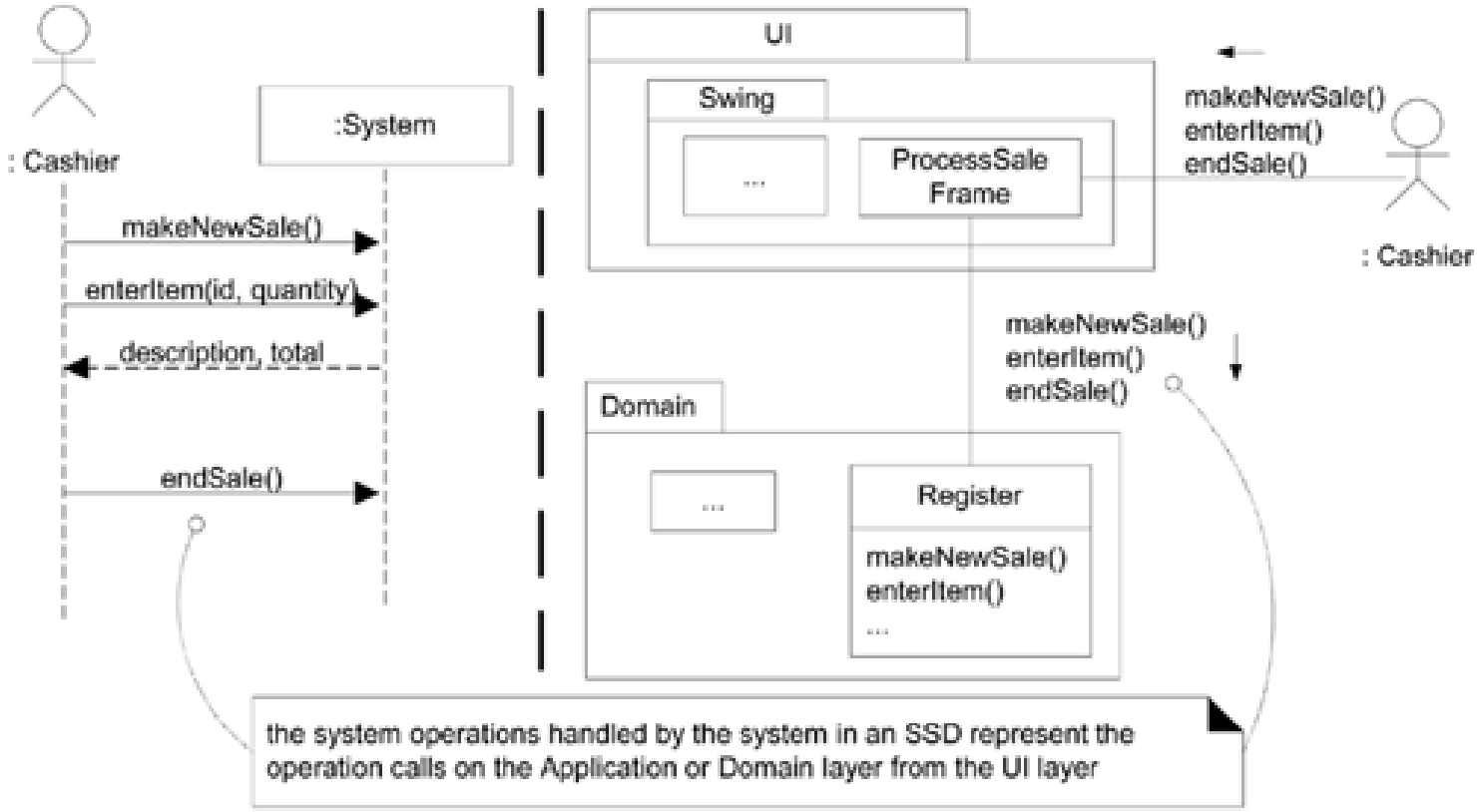
UI layer of the system captures system operation requests

In a well-designed layered architecture:
UI layer objects will forward, or delegate,
the request from the UI layer onto the domain layer for handling

This supports **high cohesion** and a **separation of concerns**

The messages from the UI layer to the domain layer will be the messages given in the SSDs, for example ***enterItem***

SSDs, System Operations, Layers —Connection



Logical Architecture for POS and Monopoly

We have seen the logical architecture for the POS is the figures

The Monopoly architecture is a simple layered design with UI, domain, and services layers

On to Object Design

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

Agile Modeling and Lightweight UML Drawing

Modeling to **understand** and **communicate**, rather than to document

Modeling with others

Creating **several models in parallel**, interaction diagrams, class diagrams, …

Use lots of whiteboards, a record of the modeling may be kept via photos

UML CASE Tools

Use UML CASE (Computer-Aided SE) tools as appropriate

Some tools can integrate with IDEs

Some tools can reverse-engineer (generate diagrams from code)
Class diagrams (common) and
Interaction diagrams (less common)

How Much UML Drawing Time Before Coding?

For a three-week time-boxed iteration,

Spend a few hours or at most one day (with partners) near the start of the iteration "at the walls" (or with a UML CASE tool)

Drawing UML for the hard, creative parts of the detailed object design

How Much UML Drawing Time Before Coding?

Then stop, and if sketching-perhaps take digital photos,

Print the pictures,

And transition to coding for the remainder of the iteration,

Using the UML drawings for inspiration as a starting point,

But recognizing that the final design in code will converge and improve

Object models —Static and Dynamic Modeling

Static models

- Including UML **class diagrams**,
- help design the definition of packages,
- class names,
- attributes,
- and method signatures, but not method bodies/behavior

Object Models —Static, Dynamic Modeling

Dynamic Models

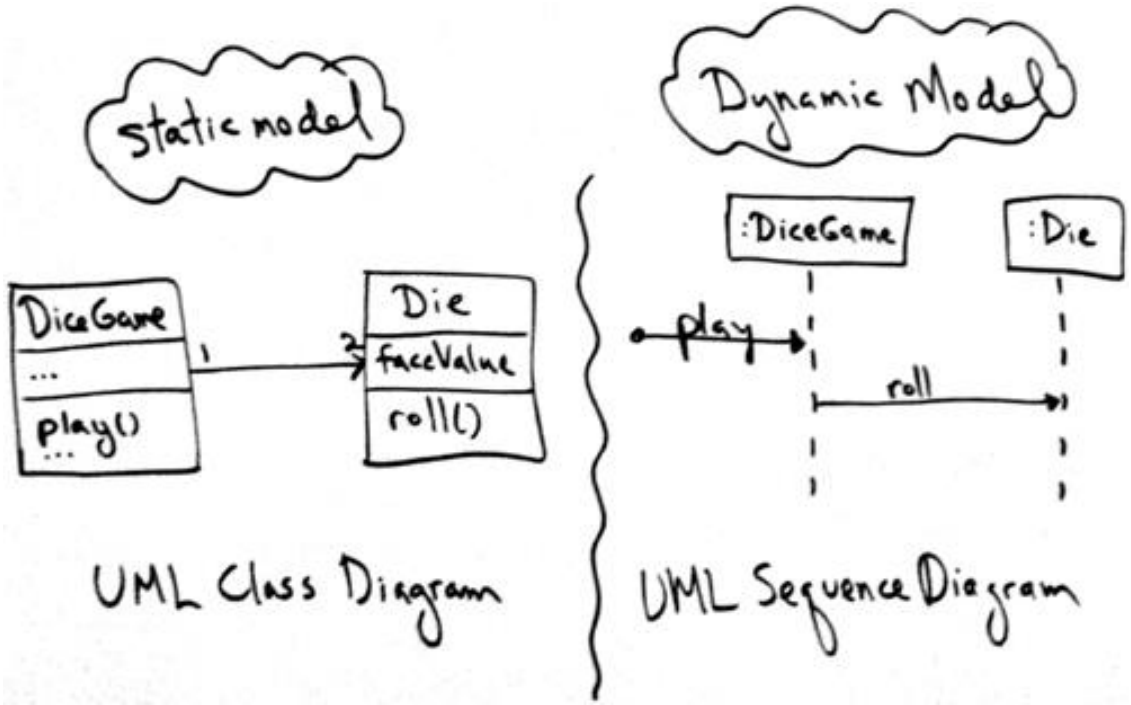
UML interaction diagrams

Help design the logic,

The behavior of the system

Dynamic models tend to be the more interesting, difficult, and important diagrams to create

Static And Dynamic UML Diagrams For Object Modeling



Static and Dynamic Modeling in Parallel

It is often useful to **create dynamic models and static models together**

They feed from each others

From the dynamic model we can see what the static model (classes) need

People often think that the important diagram is the static view, the class diagram

Static and Dynamic Modeling in Parallel

Most of the challenging, interesting, useful design work happens during interaction modeling

With dynamic object modeling the exact

Details of what objects need to exist and

how they collaborate via messages and methods is seen

Dynamic Modeling is Important

Spend significant time doing interaction diagrams (sequence or communication diagrams)

During dynamic modeling Responsibility-Driven Design (RDD) may be applied, including the GRASP principles

GRASP: General Responsibility Assignment Software Patterns, consist of guidelines for assigning responsibility to classes

Patterns to answer some software problems, and these problems are common to almost every software development project .

Object Design Skill is More than UML Notation Skill

How to think and design in terms of **objects**,

And apply to object design **best-practice patterns**,

This is a different, and a much more valuable skill, than knowing UML notation

Object Design Skill is More than UML Notation Skill

While object modeling, we need to answer key questions:

- What are the **responsibilities** of the object?
- Who does the object **collaborate** with?
- What **design patterns** should be applied?

Other Object Design Techniques —CRC Cards

An additional popular text-oriented modeling technique is **Class/Responsibility/Collaboration** (CRC) cards

A CRC card is a paper index card on which the responsibilities and collaborators of a class is written

Other Object Design Techniques: CRC Cards

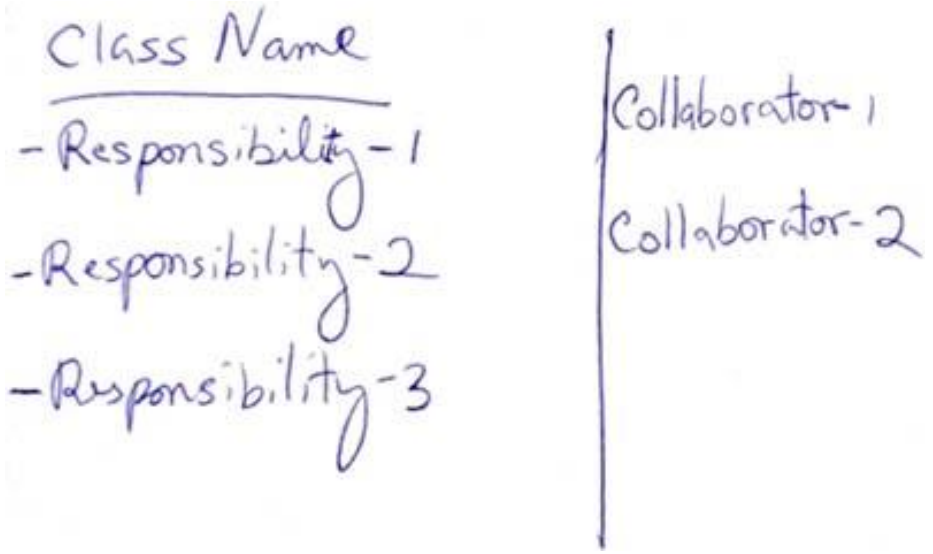
In a CRC modeling session a group sits around a table, discussing and writing on the cards as they play "what if" scenarios with the objects

The group considers what

each object must do and

what other objects it must collaborate with

Template For A CRC Card



Four Sample CRC Cards