Object-Oriented Analysis and Design using UML and Patterns

Logical Architecture &

Package Diagram

Logical Architecture & Package Diagram

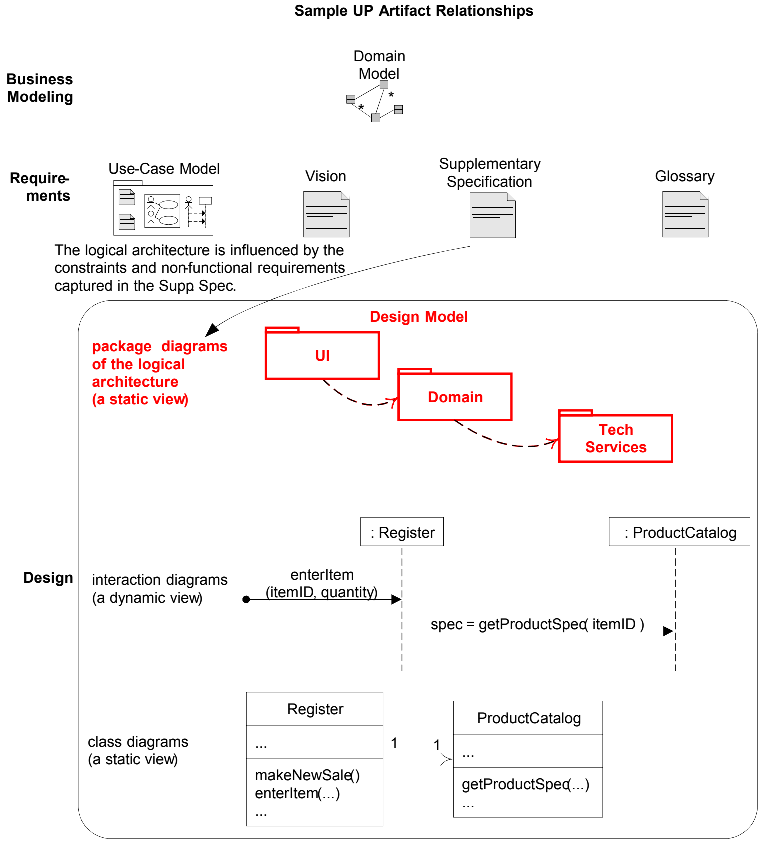
Objectives

Introduce a logical architecture using layers

Illustrate the logical architecture using UML package diagrams

Compare packages and subsystems

Where are we?


Sample UP Artifact Relationships
Business Modeling
Domain Model
Requirements
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(...)
...

What is Software Architecture?

"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both"

A structure is a set of elements held together by a relation, and is architectural if it supports reasoning about the system and the it's properties

The reasoning should be about an attribute of the system that is important to some stakeholder, including:

functionality achieved by the system

the availability of the system in the face of faults

the difficulty of making specific changes to the system

the responsiveness of the system to user requests, etc.

Categories of Structures

Module structures

show how a system is to be structured as a set of module units (classes, or layers, or packages, or merely divisions of functionality, all of which are units of implementation) that have to be constructed or procured

Component-and-connector (C & C) structures

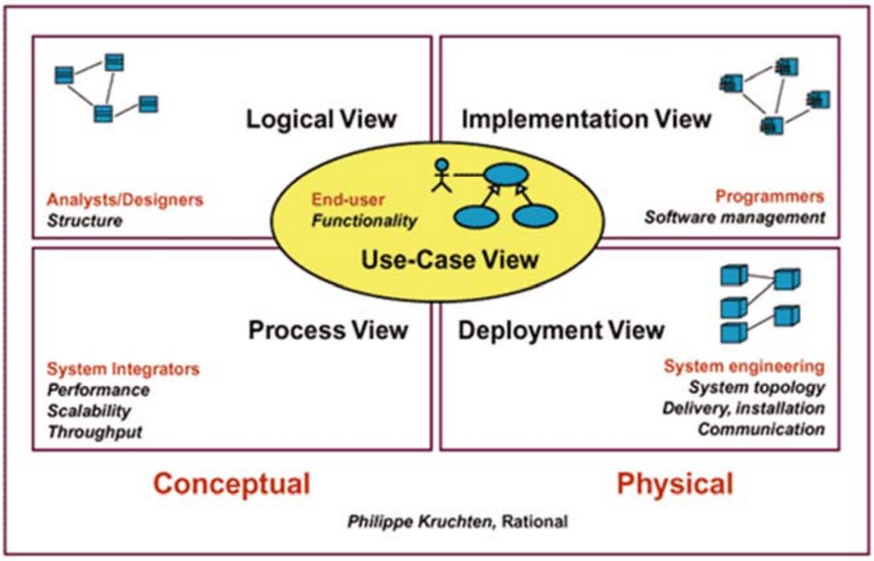
show how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors)

Allocation structures

show how the system will relate to nonsoftware structures in its environment (such as CPUs, networks, development teams, etc.).

Category reflects specific viewpoints!

RUP 4+1 View Model



Why is Software Architecture Important?

- Thirteen Reasons -

1. An architecture will inhibit or enable a system’s driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system’s qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that form the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

Logical Architecture and Layers

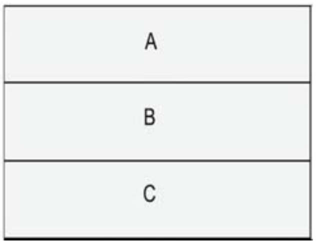
The **logical architecture** is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers

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 (these latter decisions are part of the **deployment architecture**)

One of the most common styles of the logical architecture is the **layered architecture**

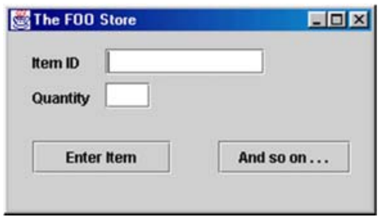
A **layer** is a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system

Layers are organized such that "higher“ layers (such as the UI layer) call services in "lower" layers, but not normally vice versa



Typical Layers

User Interface



Application logic and domain object

Sale

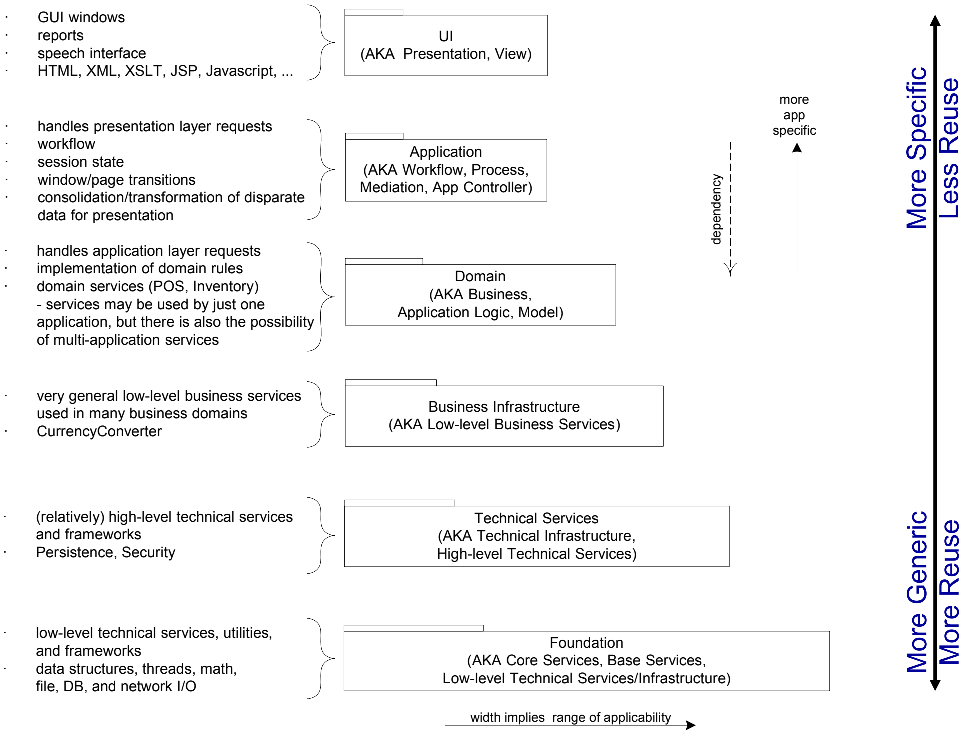
Payment

Technical services

Log

PersistenceFacade

Layered 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)

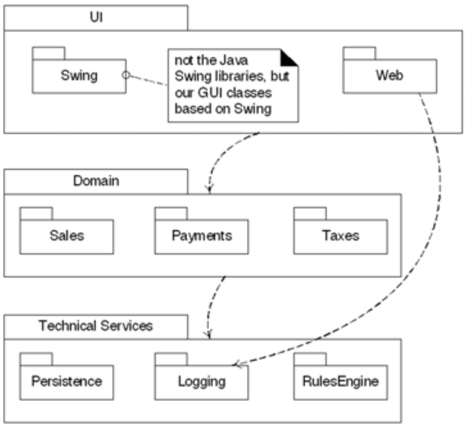
- low-level technical services, utilities, and frameworks
- data structures, threads, math, file, DB, and network I/O

Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)

Applying UML: Package Diagrams

Package diagram shows the organization of the model elements into packages and their dependencies

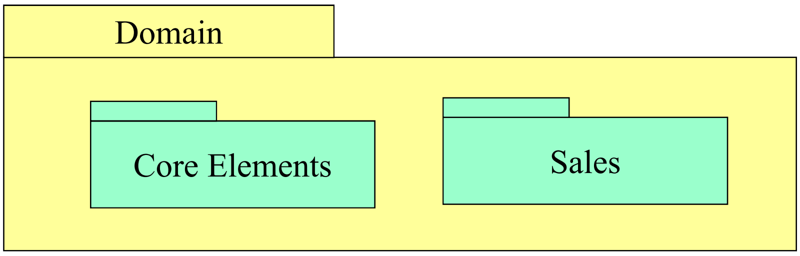
The benefit of organizing elements into packages is that it chunks detailed elements into larger abstractions -- supporting a higher-level view and viewing a model in simple grouping



Packages

A UML package is a general-purpose modeling element used to group closely related modeling elements such as classes, interfaces, use cases, other packages, etc.

- Group semantically related elements
- Provide a *namespace* within which all names must be unique
- Define a "semantic boundary" in the model
- Provide units for parallel working and configuration management



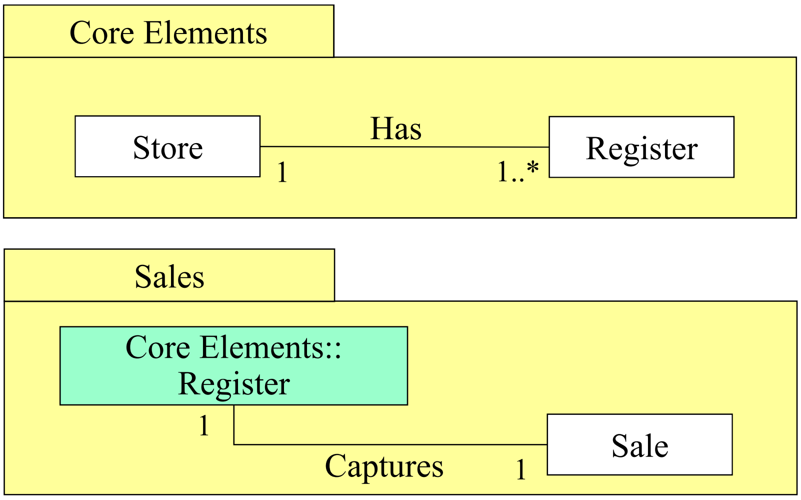
Domain

Core Elements

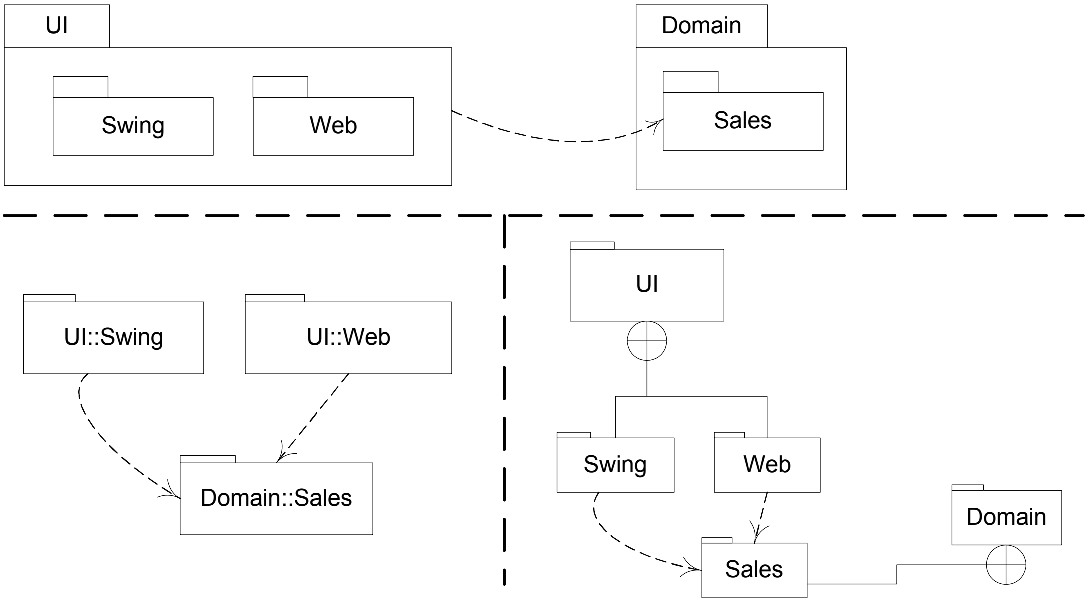
Sales

Ownership and References

An element (e.g., type or class) is *owned* by the package within which it is defined, but may be *referenced* in other packages



Package Nesting and Dependencies



Package Organization Guidelines

Package Functionally Cohesive Vertical and Horizontal Slices

Package a Family of Interfaces

Package by Work and by Clusters of Unstable Classes

Factor out Independent Types

Use Factories to Reduce Dependency on Concrete Packages

No Cycles in Packages

Guideline: Package a Family of Interfaces

Place a family of functionally related *interfaces* in a separate package -- separate from implementation classes

The Java technologies EJB package *javax.ejb* is an example: It is a package of at least twelve interfaces; implementations are in separate packages

Guideline:
Package by Work and by Clusters of Unstable Classes

A package needs to be refactored into more stable and less stable subsets, or more generally, into groups related to work. That is, if most types in a package are worked on together, then it is a useful grouping

If an element is related to an optional service, group it with its collaborators in a separate subsystem

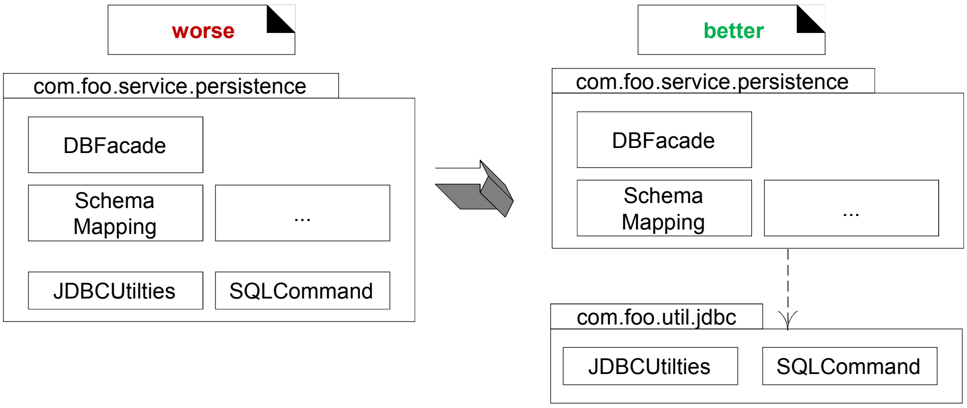
Consider how stable your design element is:

Try to move stable elements down the layer hierarchy, unstable elements up

Guideline: Factor out Independent Types

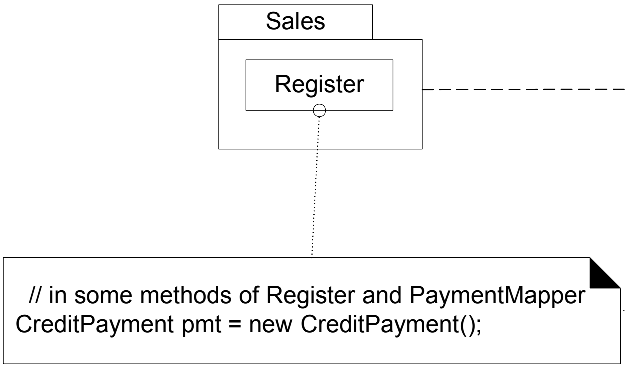
Organize types that can be used independently or in different contexts into separate packages.

Without careful consideration, grouping by common functionality may not provide the right level of granularity in the factoring of packages.



Guideline:
Use Factories to Reduce Dependency on Concrete Packages

One way to increase package stability is to reduce its dependency on concrete classes in other packages

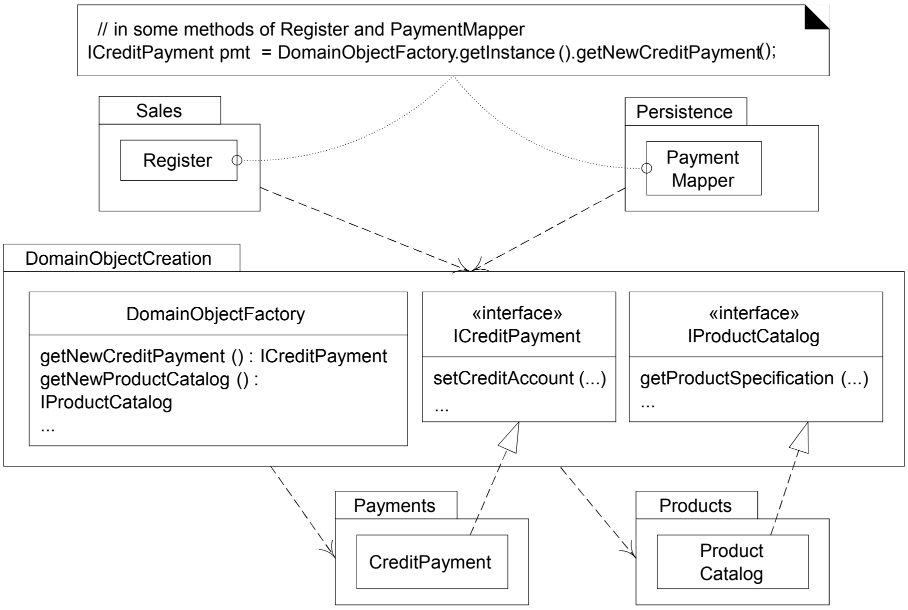


// in some methods of Register and PaymentMapper
CreditPayment pmt = new CreditPayment();

Before (Worse) *: Direct coupling to concrete package due to creation*

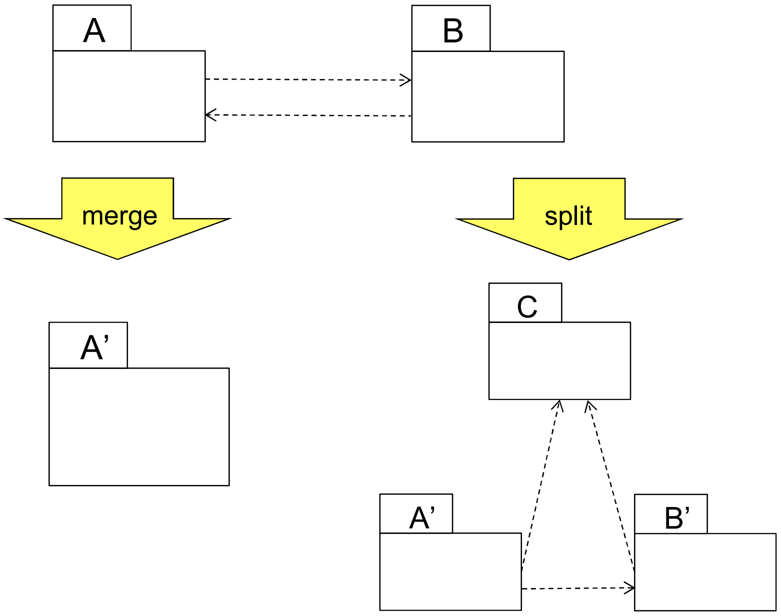
Guideline: (Cont'd)
Use Factories to Reduce Dependency on Concrete Packages

// in some methods of Register and PaymentMapper
ICreditPayment pmt = DomainObjectFactory.getInstance().getNewCreditPayment();


Sales
Register
Persistence
Payment Mapper
DomainObjectCreation
DomainObjectFactory
getNewCreditPayment () : ICreditPayment
getNewProductCatalog () : IProductCatalog
...
«interface» ICreditPayment
setCreditAccount (...)
...
«interface» IProductCatalog
getProductSpecification (...)
...
Payments
CreditPayment
Products
Product Catalog

After (Better) *: No direct coupling to concrete package*

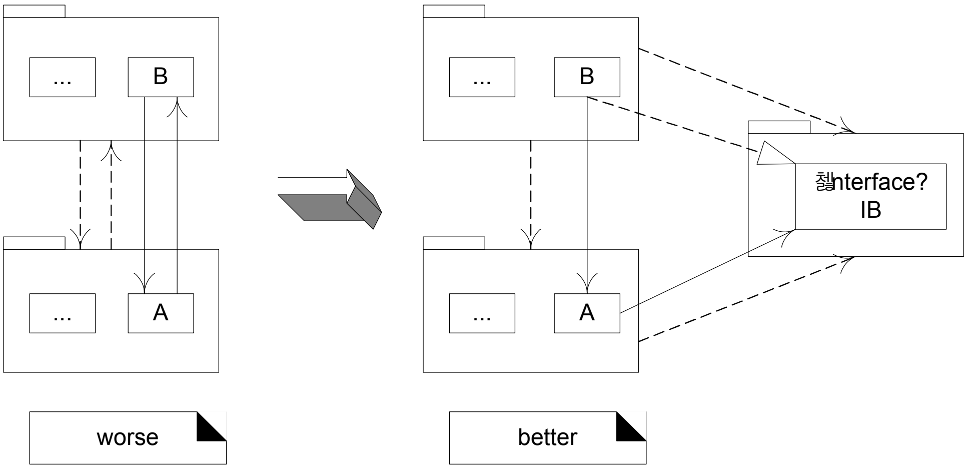
Guideline: No Cycles in Packages



Guideline: No Cycles in Packages (Cont'd)

The steps to break the cycle with an interface are:

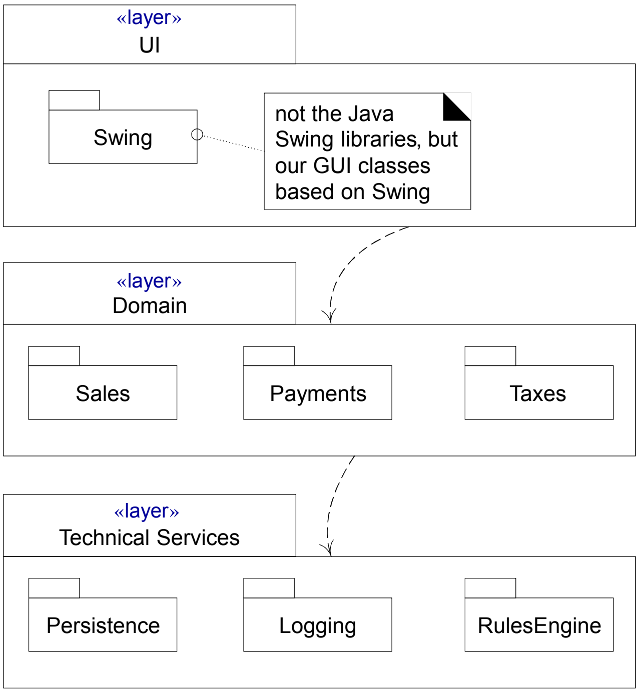
1. Redefine the depended-on classes in one of the packages to implement new interfaces
2. Define the new interfaces in a new package
3. Redefine the dependent types to depend on the interfaces in the new package, rather than the original classes



Modeling Architectural Layers

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 with stereotype «layer»



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 and dependencies, improves cohesion, increases reuse potential, and increases clarity.

Related complexity is encapsulated and decomposable.

Some layers can be replaced with new implementations. This is generally not possible for lower-level Technical Service or Foundation layers (e.g., java.util), but 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

Domain Layer vs. Domain Objects

How do we design the application logic with objects?

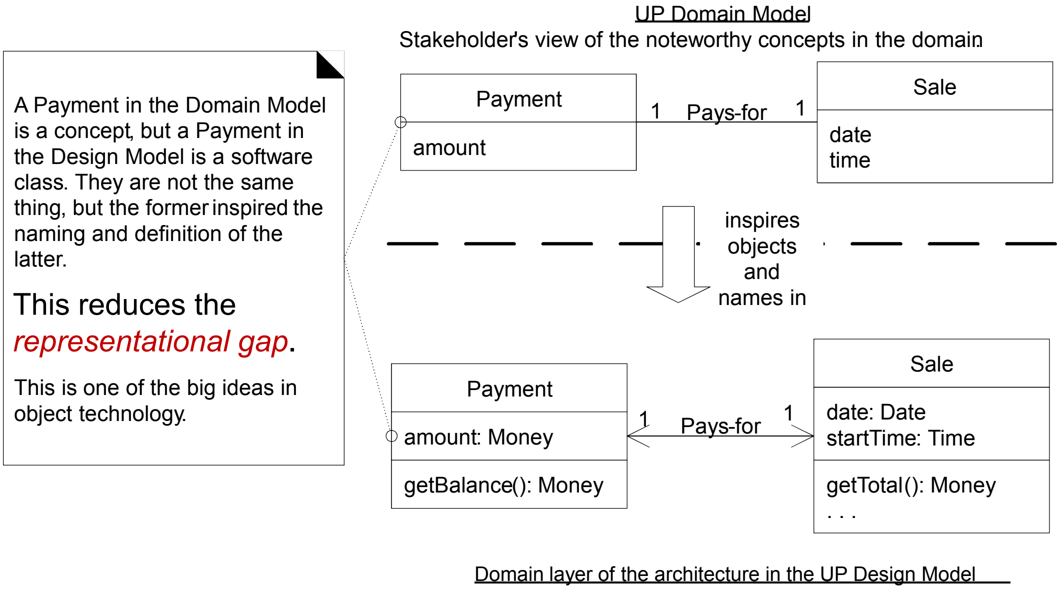
To create software objects with names and information similar to the real-world domain, and assign application logic responsibilities to them

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.

Designing objects this way leads the **domain layer** of the architecture -- the layer that contains domain objects to handle application logic work

Domain layer and Domain model Relationship



UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain

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.

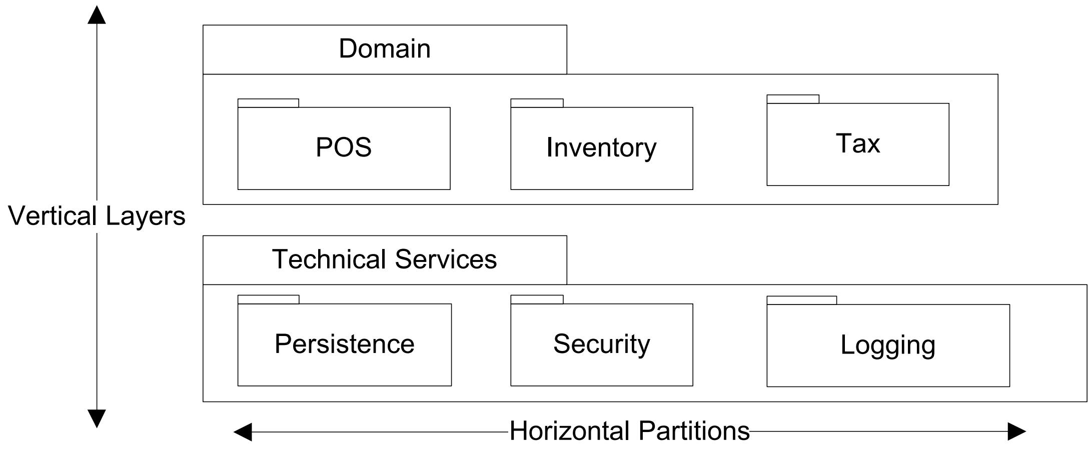
Domain layer of the architecture in the UP Design Model

The OO 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

Layers and Partitions

The **layers** of an architecture are said to represent the vertical slices, while **partitions** represent a horizontal division of relatively parallel subsystems of a layer


Vertical Layers

Domain

POS

Inventory

Tax

Technical Services

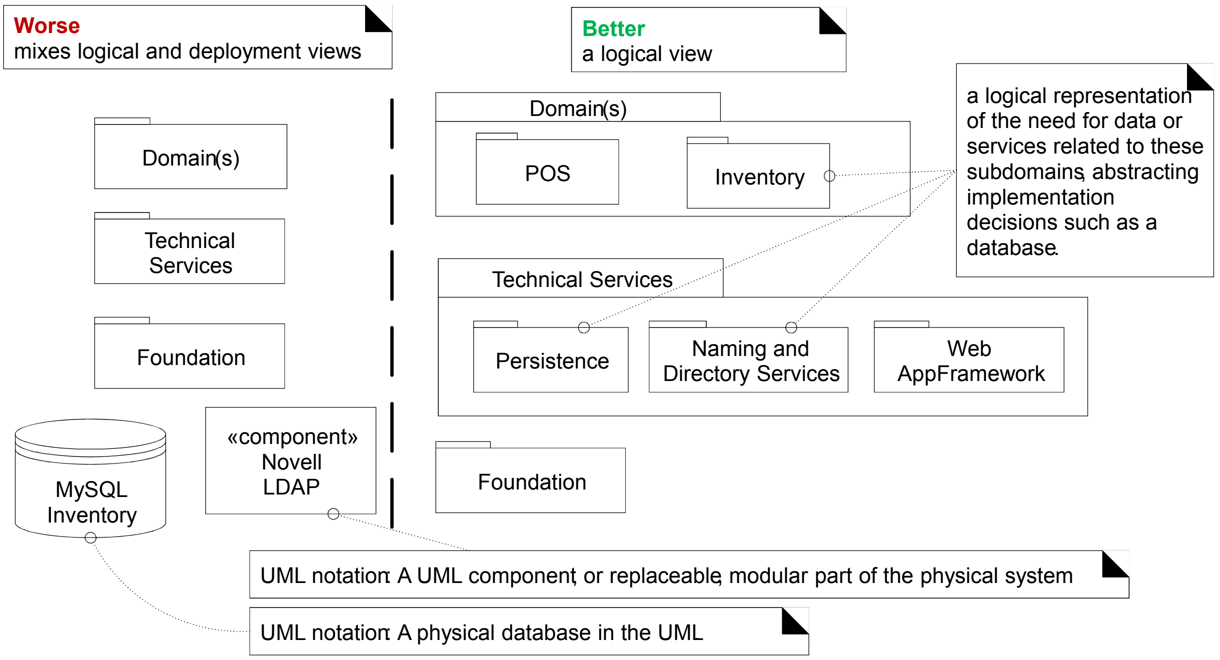
Persistence

Security

Logging

Horizontal Partitions

Don't Show External Resources as the Bottom Layer


Worse
mixes logical and deployment views

Domain(s)

Technical Services

Foundation

MySQL Inventory

«component»
Novell
LDAP

Better
a logical view

Domain(s)

POS

Inventory

Technical Services

Persistence

Naming and Directory Services

Web AppFramework

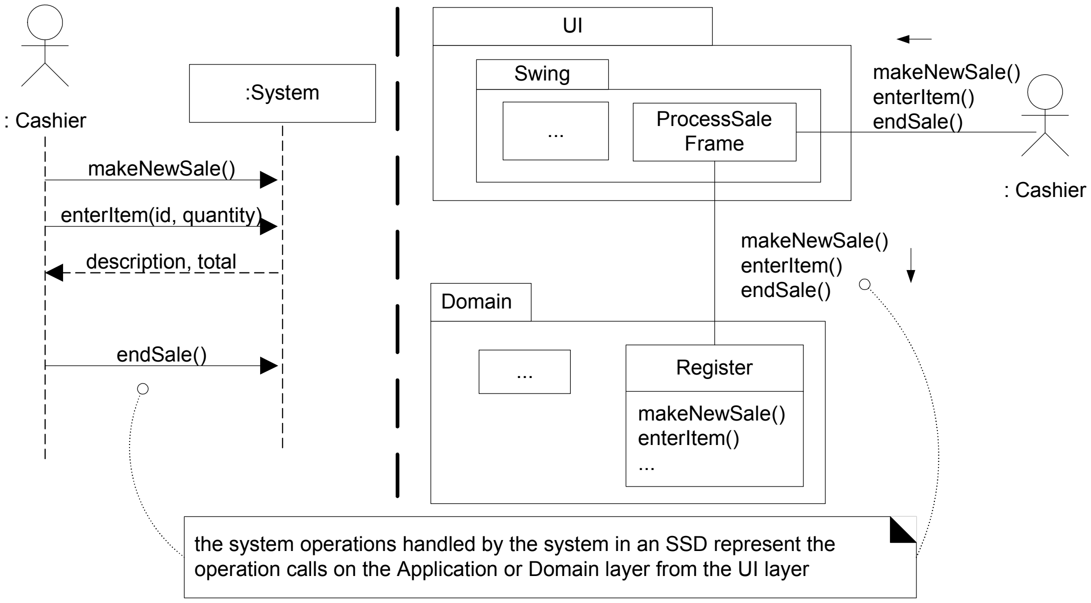
Foundation

a logical representation of the need for data or services related to these subdomains, abstracting implementation decisions such as a database.

UML notation A UML component, or replaceable, modular part of the physical system

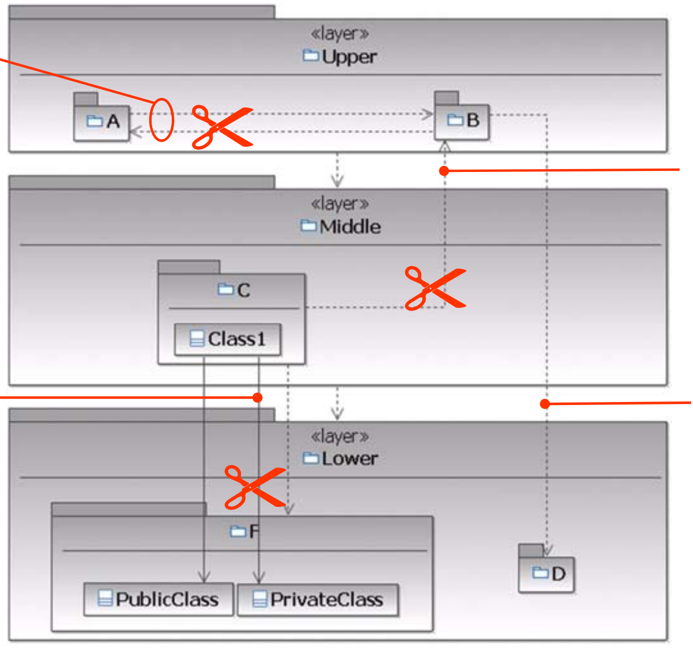
UML notation A physical database in the UML

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

Evaluating Package Coupling in/between Layers



Avoid circular dependencies

Packages in lower layers should not be dependent upon packages in upper layers

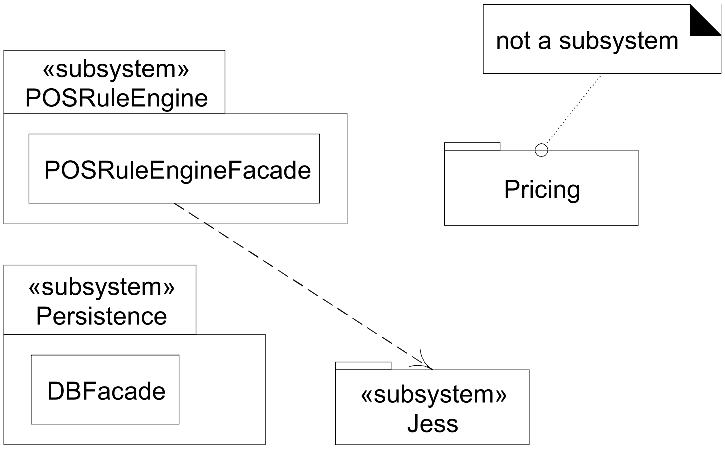
Only public classes can be referenced outside of the owning package

Avoid skipping layers

Packages versus Subsystems

Some packages or layers are not just conceptual groups of things, but are true **subsystems** with **behavior** and **interfaces**. To contrast:

- The *Pricing* package is not a subsystem; it simply groups the factory and strategies used in pricing. Also Foundation packages such as *java.util*
- The *Persistence, POSRuleEngine,* and *Jess* packages are subsystems. They are discrete engines with cohesive responsibilities that do work

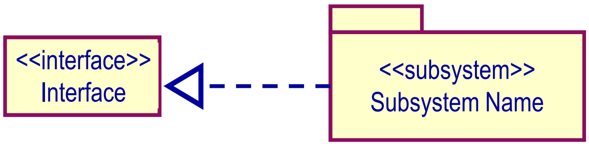


Packages versus Subsystems (Cont'd)

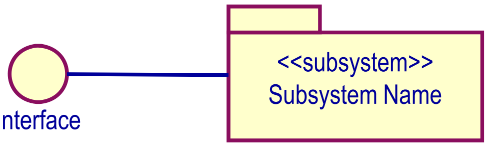
A subsystem is a "cross between" a package and a class

A subsystem realizes one or more interfaces which define its behavior

As replaceable resign elements, subsystems are ideal for modeling components - the replaceable units of assembly in CBD



Realization (Canonical form)



Realization (Elided form)

Packages	Subsystems
Do not provide behavior	Provide behavior
Do not completely encapsulate their contents	Completely encapsulate their contents
May not be easily replaced	Are easily replaced

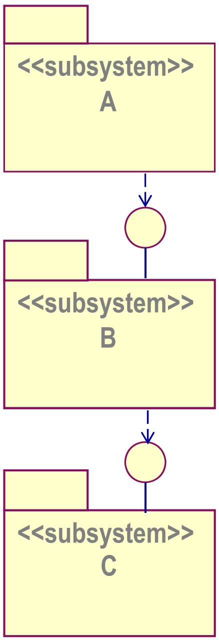
Subsystem Guidelines

Goals

- Loose coupling
- Portability, plug-and-play compatibility
- Insulation from change
- Independent evolution

Strong Suggestions

- Don’t expose details, only interfaces
- Only depend on other interfaces



<<subsystem>>
A

<<subsystem>>
B

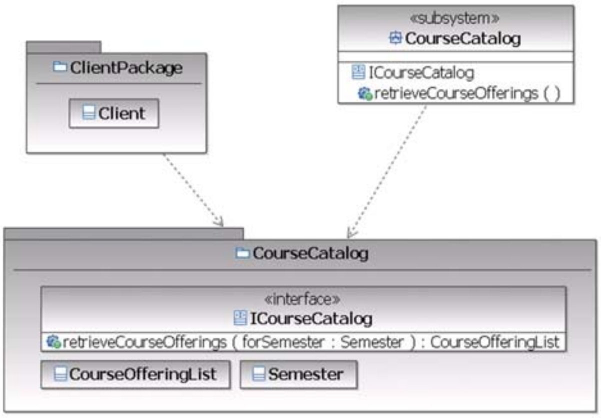
<<subsystem>>
C

Subsystem Dependencies

Keep in mind: The interfaces provided (and/or required) by a subsystem are **outside** the subsystem

Often the services described by an interface will involve non-standard types, e.g. *Semester* and *CourseOfferingList*

- You can group the interfaces and types in a single package
- Both the client packages and the realizing subsystem have dependencies on this package



«subsystem»
CourseCatalog
ICourseCatalog
retrieveCourseOfferings ( )

ClientPackage
Client

CourseCatalog
«interface»
ICourseCatalog
retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
CourseOfferingList
Semester

Candidate Subsystems

Analysis Classes providing complex services and/or utilities

For example, security authorization services

Boundary classes

User interfaces

Access to external systems and/or devices

Classes providing optional behavior or different levels of the same services

Highly coupled elements

Existing products that export interfaces (communication software, database access support, etc.)

Architectural Analysis

Objectives

Introduce a logical architecture using layers

Illustrate the logical architecture using UML package diagrams

Architectural Analysis

Architectural analysis is concerned with the identification and resolution of the system's nonfunctional requirements (for example, security), in the context of the functional requirements (for example, processing sales)

It includes identifying *variation points* and the most probable *evolution points*

The essence of architectural analysis is

- to identify factors that should influence the architecture,
- to understand their variability and priority, and
- to resolve them.

In the UP, the term encompasses both *architectural investigation* (identification) and *architectural design* (resolution)

Definition: Variation and Evolution Points

Variation point

Variations in the existing current system or requirements, such as the multiple tax calculator interfaces that must be supported.

Evolution point

Speculative points of variation that may arise in the future, but which are not present in the existing requirements.

Why is Architectural Analysis Important?

To reduce the risk of missing something centrally important in the design

To avoid applying excessive effort to low priority issues

To help align the product with business goals

Purpose of Architectural Analysis

To define a candidate architecture for the system, based on experience gained from similar systems or in similar problem domains

To define the architectural patterns, key mechanisms and modeling conventions for the system

To define the reuse strategy

To provide input to the planning process

Common Steps in Architectural Analysis

1. Identify and analyze the non-functional requirements. Functional requirements are also relevant (especially in terms of variability or change), but the non-functional are given thorough attention. In general, all these may be called **architectural factors** (also known as the **architectural drivers**)

   It is considered a part of architectural analysis rather than requirements analysis in the UP

2. Analyze alternatives and create solutions that resolve the impact. These are **architectural decisions**

   Decisions range from “remove the requirement,” to a custom solution, to “stop the project,” to “hire an expert.”

Architectural Factors

Quality Attributes

Business Constraints

Technical Constraints

Quality Scenarios

When defining quality requirements during architectural factor analysis, *quality scenarios* are recommended

- As they define measurable (or at least observable) responses, and thus can be verified
- It is not much use to vaguely state "the system will be easy to modify" without some measure of what that means
- Quantifying some things, such as performance goals and mean time between failure, are well known practices

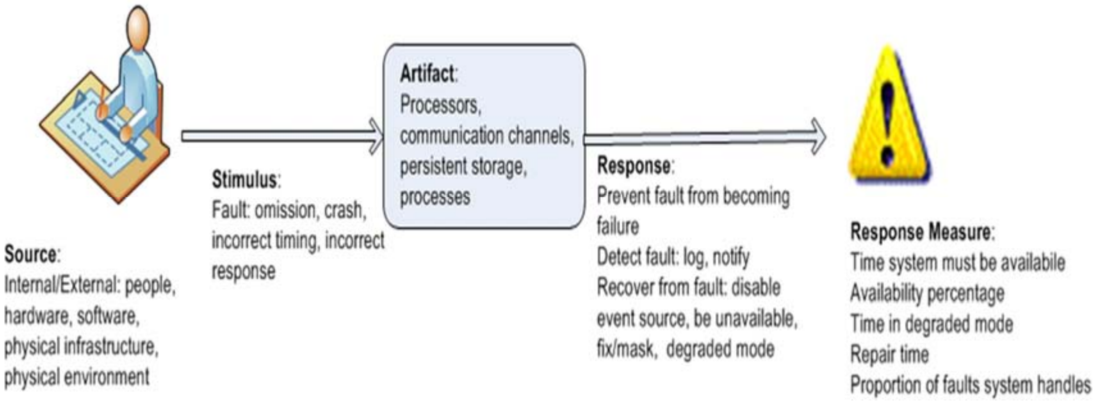
Quality scenarios are short statements of the form <stimulus> <measurable response>; for example:

- When the completed sale is sent to the remote tax calculator to add the taxes, the result is returned within 2 seconds "most" of the time, measured in a production environment under "average" load conditions

Quality Scenarios: Six-part Rule

1. **Source of stimulus.** This is some entity (a human, a computer system, or any other actuator) that generated the stimulus
2. **Stimulus.** The stimulus is a condition that requires a response when it arrives at a system
3. **Environment.** The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, “normal” operation can refer to one of a number of modes
4. **Artifact.** Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it
5. **Response.** The response is the activity undertaken as the result of the arrival of the stimulus
6. **Response measure.** When the response occurs, it should be measurable in some fashion so that the requirement can be tested

Example general scenario for availability



Source: Internal/External: people, hardware, software, physical infrastructure, physical environment

Stimulus: Fault: omission, crash, incorrect timing, incorrect response

Artifact: Processors, communication channels, persistent storage, processes

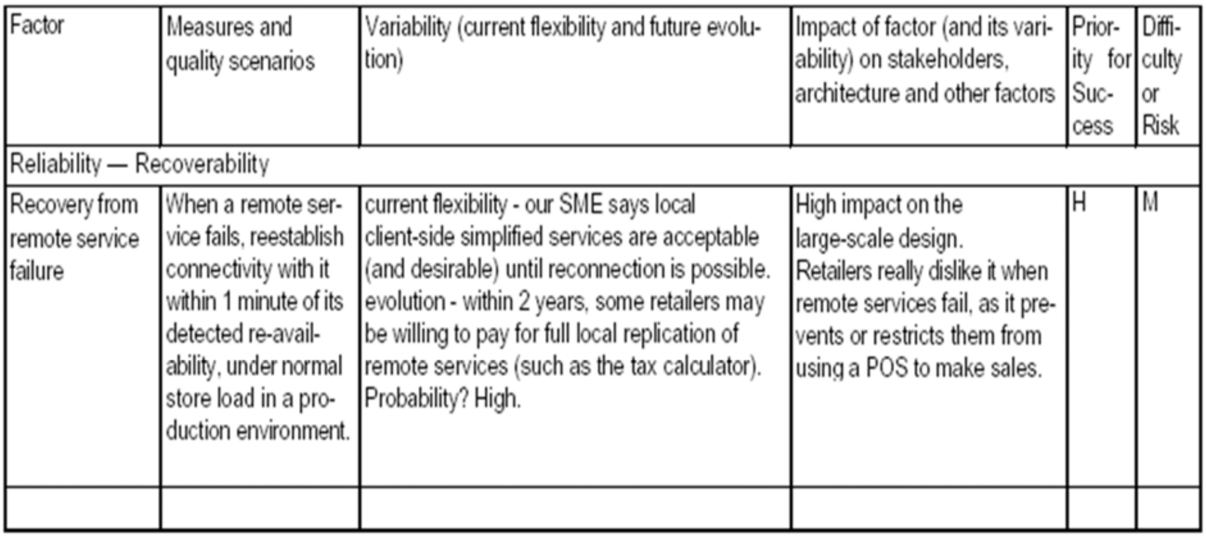
Response: Prevent fault from becoming failure
Detect fault: log, notify
Recover from fault: disable event source, be unavailable, fix/mask, degraded mode

Response Measure: Time system must be availabile
Availability percentage
Time in degraded mode
Repair time
Proportion of faults system handles

Factor Table

Most architectural methods advocate creating a table or tree

The following is called a factor table (part of Supplementary Specification)

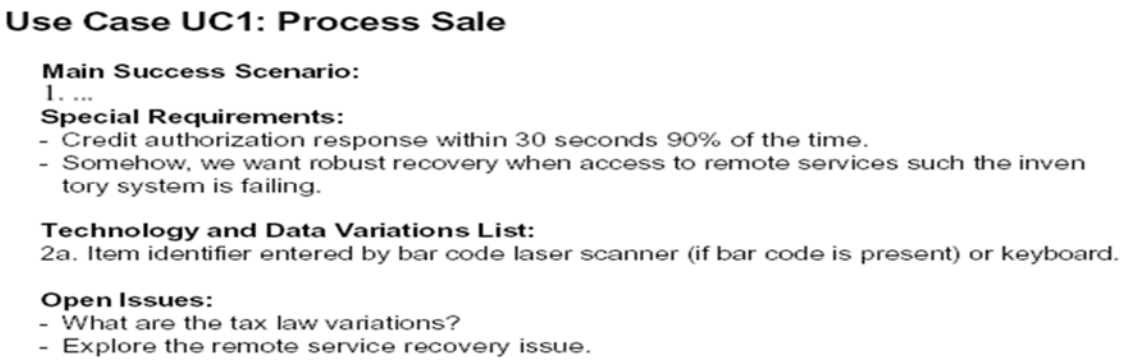


Factor	Measures and quality scenarios	Variability (current flexibility and future evolution)	Impact of factor (and its variability) on stakeholders, architecture and other factors	Priority for Success	Difficulty or Risk
Reliability — Recoverability					
Recovery from remote service failure	When a remote service fails, reestablish connectivity with it within 1 minute of its detected re-availability, under normal store load in a production environment.	current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible. evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High.	High impact on the large-scale design. Retailers really dislike it when remote services fail, as it prevents or restricts them from using a POS to make sales.	H	M

Factors and UP Artifacts

The central functional requirements repository in the UP are the **use cases**, and they, along with the **Vision** and **Supplementary Specification**, are an important source of inspiration when creating a factor table.

In the use cases, the *Special Requirements, Technology Variations*, and *Open Issues* should be reviewed, and their implied or explicit architectural factors consolidated in the Supplementary Specification



Use Case UC1: Process Sale

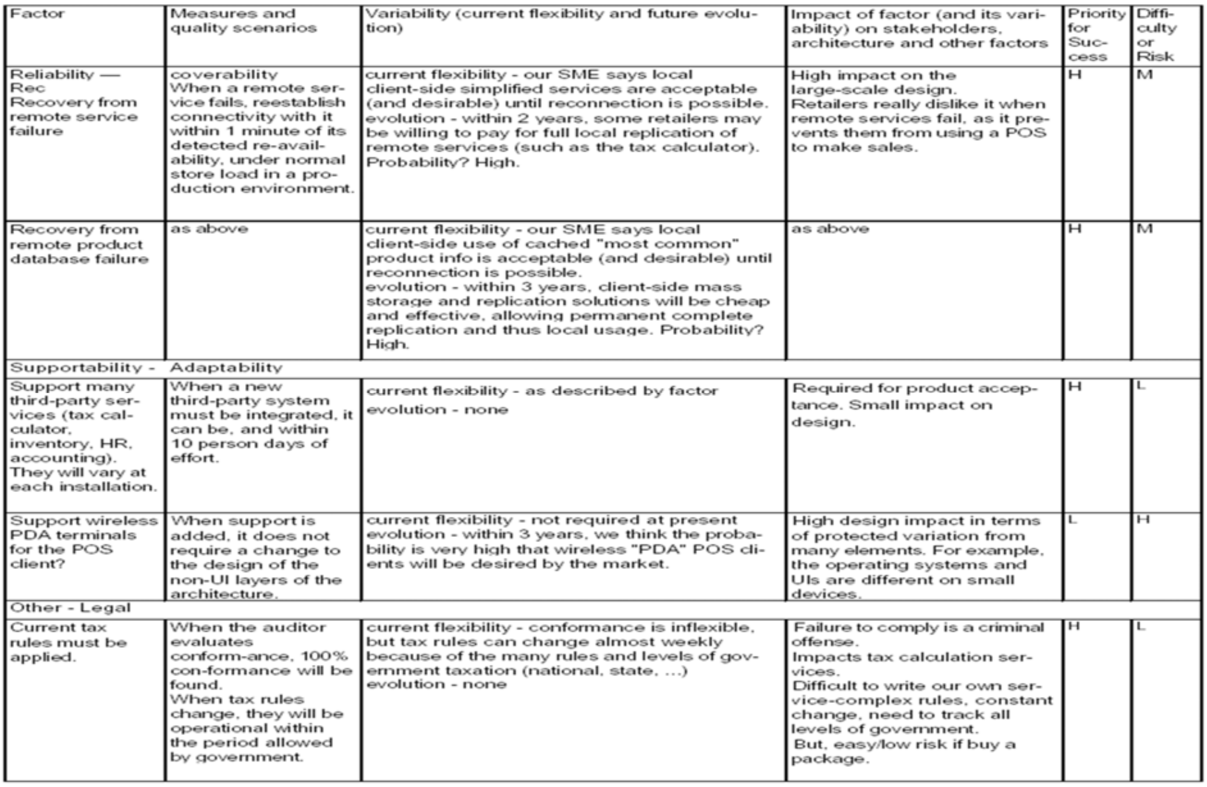
Main Success Scenario:
1. ...

Special Requirements:
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inven tory system is failing.

Technology and Data Variations List:
2a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.

Open Issues:
- What are the tax law variations?
- Explore the remote service recovery issue.

Example: Partial NextGen POS Architectural Factor Table



Factor	Measures and quality scenarios	Variability (current flexibility and future evolution)	Impact of factor (and its variability) on stakeholders, architecture and other factors	Priority for Success	Difficulty or Risk
Reliability — Rec Recovery from remote service failure	coverability When a remote service fails, reestablish connectivity with it within 1 minute of its detected re-availability, under normal store load in a production environment.	current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible. evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High.	High impact on the large-scale design. Retailers really dislike it when remote services fail, as it prevents them from using a POS to make sales.	H	M
Recovery from remote product database failure	as above	current flexibility - our SME says local client-side use of cached "most common" product info is acceptable (and desirable) until reconnection is possible. evolution - within 3 years, client-side mass storage and replication solutions will be cheap and effective, allowing permanent complete replication and thus local usage. Probability? High.	as above	H	M
Supportability -	Adaptability				
Support many third-party services (tax calculator, inventory, HR, accounting). They will vary at each installation.	When a new third-party system must be integrated, it can be, and within 10 person days of effort.	current flexibility - as described by factor evolution - none	Required for product acceptance. Small impact on design.	H	L
Support wireless PDA terminals for the POS client?	When support is added, it does not require a change to the design of the non-UI layers of the architecture.	current flexibility - not required at present evolution - within 3 years, we think the probability is very high that wireless "PDA" POS clients will be desired by the market.	High design impact in terms of protected variation from many elements. For example, the operating systems and UIs are different on small devices.	L	H
Other - Legal					
Current tax rules must be applied.	When the auditor evaluates conform-ance, 100% con-formance will be found. When tax rules change, they will be operational within the period allowed by government.	current flexibility - conformance is inflexible, but tax rules can change almost weekly because of the many rules and levels of government taxation (national, state, ...) evolution - none	Failure to comply is a criminal offense. Impacts tax calculation services. Difficult to write our own service-complex rules, constant change, need to track all levels of government. But, easy/low risk if buy a package.	H	L

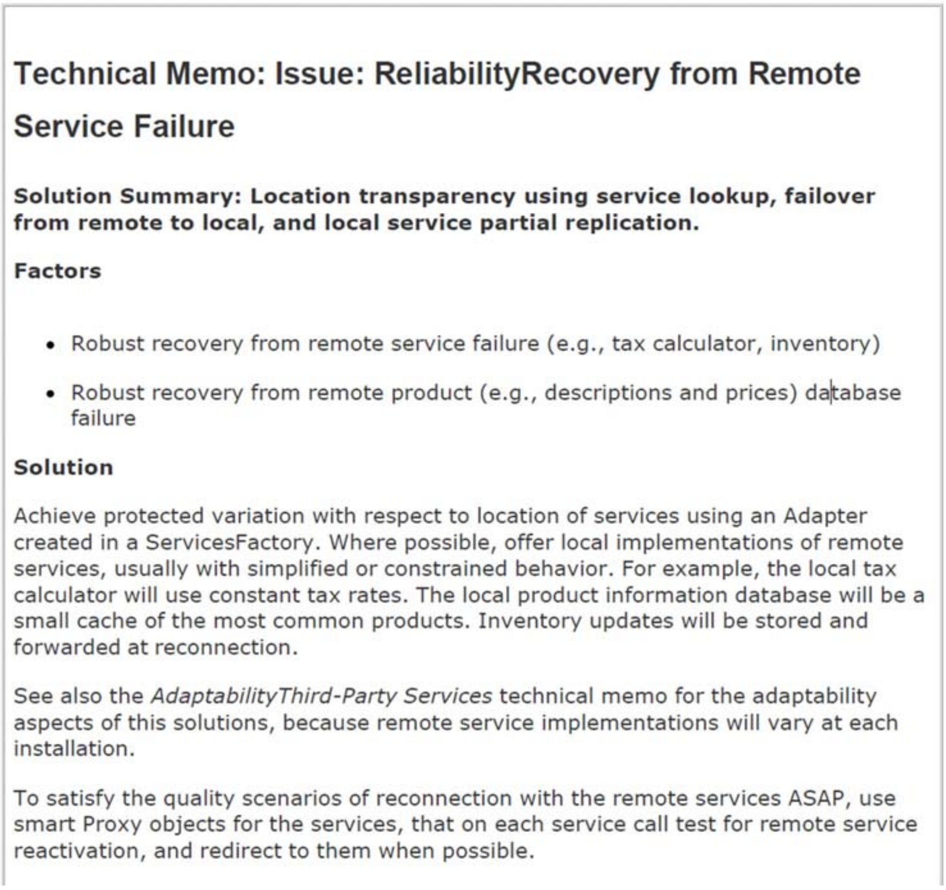
Recording Architectural Alternatives, Decisions, and Motivation

Such records have been called **technical memos** or **architectural approach documents**

In the UP, the memos should be recorded in the **Software Architecture Document (SAD)**

An important aspect of the technical memo is the *motivation* or *rationale*

Explaining the rationale of rejecting the alternatives is important, as during future product evolution, an architect may reconsider these alternatives, or at least want to know what alternatives were considered, and why one was chosen



Technical Memo: Issue: ReliabilityRecovery from Remote Service Failure

Solution Summary: Location transparency using service lookup, failover from remote to local, and local service partial replication.

Factors

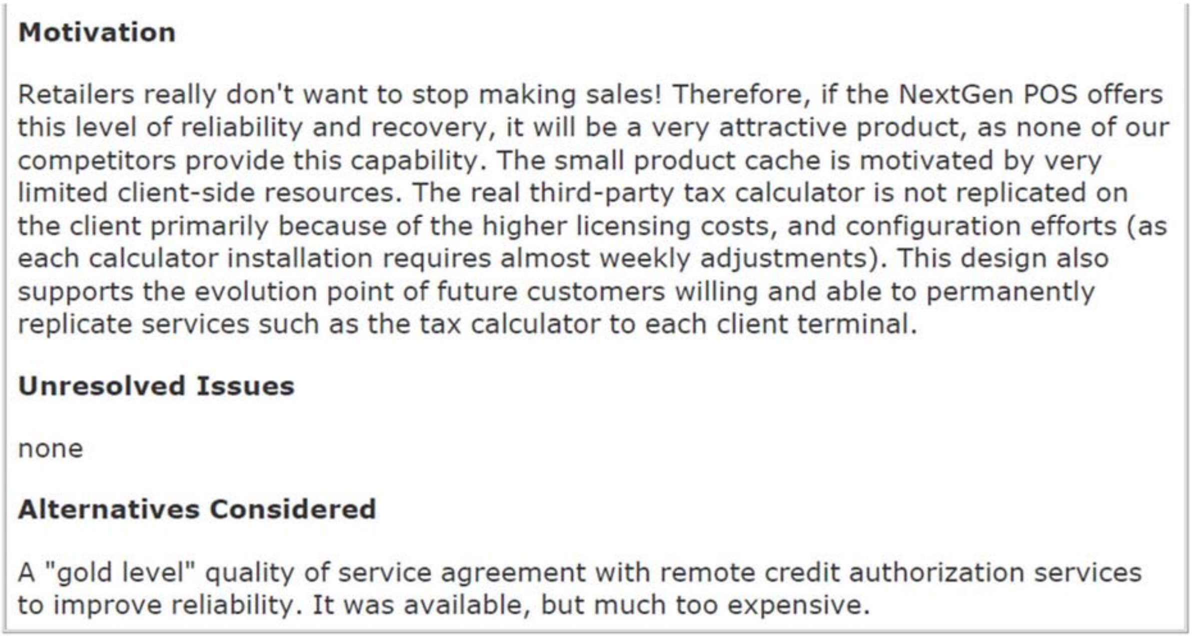
- Robust recovery from remote service failure (e.g., tax calculator, inventory)
- Robust recovery from remote product (e.g., descriptions and prices) database failure

Solution

Achieve protected variation with respect to location of services using an Adapter created in a ServicesFactory. Where possible, offer local implementations of remote services, usually with simplified or constrained behavior. For example, the local tax calculator will use constant tax rates. The local product information database will be a small cache of the most common products. Inventory updates will be stored and forwarded at reconnection.

See also the *AdaptabilityThird-Party Services* technical memo for the adaptability aspects of this solutions, because remote service implementations will vary at each installation.

To satisfy the quality scenarios of reconnection with the remote services ASAP, use smart Proxy objects for the services, that on each service call test for remote service reactivation, and redirect to them when possible.



Motivation

Retailers really don't want to stop making sales! Therefore, if the NextGen POS offers this level of reliability and recovery, it will be a very attractive product, as none of our competitors provide this capability. The small product cache is motivated by very limited client-side resources. The real third-party tax calculator is not replicated on the client primarily because of the higher licensing costs, and configuration efforts (as each calculator installation requires almost weekly adjustments). This design also supports the evolution point of future customers willing and able to permanently replicate services such as the tax calculator to each client terminal.

Unresolved Issues

none

Alternatives Considered

A "gold level" quality of service agreement with remote credit authorization services to improve reliability. It was available, but much too expensive.

Basic Architectural Design Principles

Low coupling

High cohesion

Protected variation (interfaces, indirection, service lookup, and so forth)

separation of concerns

- especially important during architectural analysis
- e.g., *cross-cutting concerns* like persistent service

Architectural Patterns

An architectural pattern expresses a fundamental structural organization schema for software systems

It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them → *Buschman et al, "Pattern-Oriented Software Architecture: A System of Patterns"*

Examples:

- Layers
- Model-view-controller (MVC)
- Pipes and filters
- Blackboard

Architectural Information in the UP Artifacts

The architectural factors (for example, in a factor table) are recorded in the **Supplementary Specification**.

The architectural decisions are recorded in the **Software Architecture Documentation (SAD)**. This includes the technical memos and descriptions of the architectural views.

Summary of Themes in Architectural Analysis

The ***first*** theme to note is that architectural analysis is concerned with the identification and resolution of the system's non-functional requirements in the context of the functional requirements. Further, identification of their variability is architecturally significant

A ***second*** theme is that architectural concerns involve system-level, large-scale, and broad problems whose resolution usually involves large-scale or fundamental design decisions

A ***third*** theme in architectural analysis is interdependencies and trade-offs.

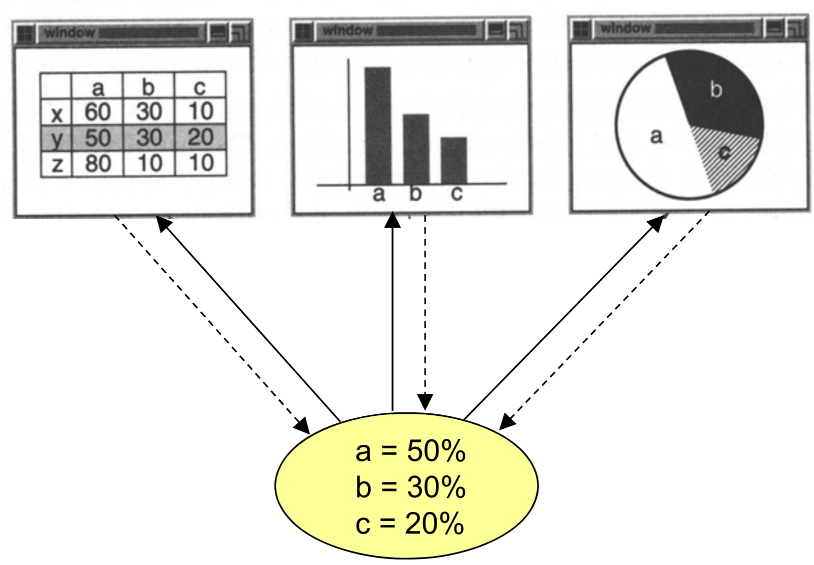
- For example, improved security may affect performance or usability, and most choices affect cost

A ***fourth*** theme in architecture analysis is the generation and evaluation of alternative solutions

Object-Oriented Analysis and Design using UML and Patterns

GRAP Patterns (I)

Observers (views)



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

a = 50%
b = 30%
c = 20%

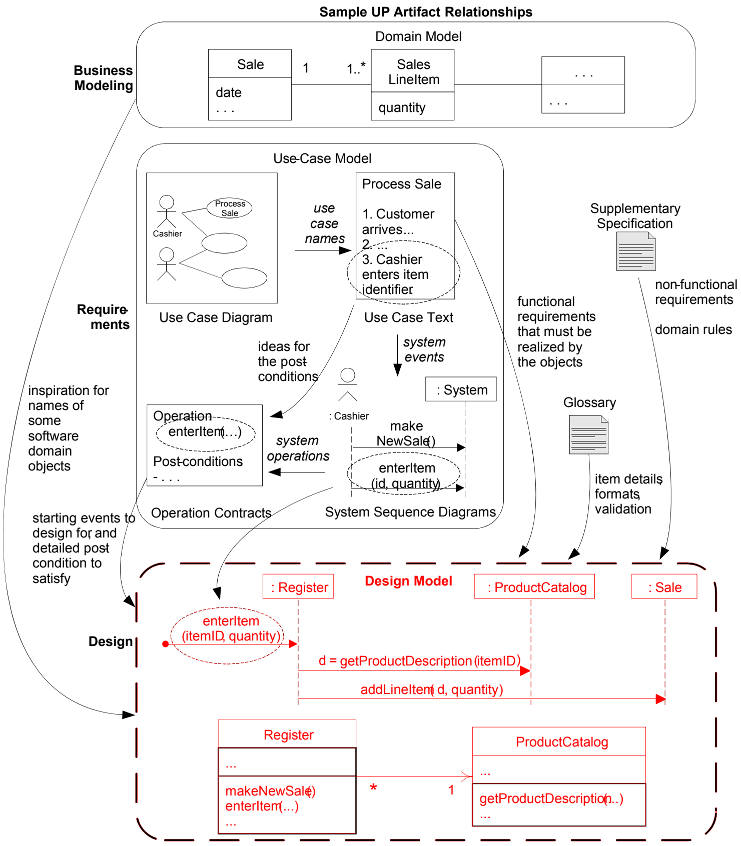
Subject (model)

From Analysis to Design

Objectives

- Understand dynamic and static object design modeling

Where are we?



Analysis Phase Conclusion

- The analysis phase emphasizes an understanding of the requirements, concepts, and system operations.
- Investigation and analysis focuses on questions of *what* -- what are the processes, concepts and so on.

Analysis Phase Artifacts

ANALYSIS ARTIFACTS	QUESTIONS ANSWERED
Use cases	What are the domain processes? What are functional requirements?
Domain Model	What are the concepts, terms?
System sequence diagrams	What are the system events and operations?
Contracts	What do the system operations do?

Design Phase

- The design phase focuses on questions of *how* -- how the system fulfills the requirements.
- The design phase emphasizes a *logical solution* from the perspective of objects (and interfaces).
- The heart of the solution is the creation of the following two diagrams, which are part of the Design Model in UP:
  - ➢ **Interaction diagrams**
    - – how objects communicate, dynamic aspects
  - ➢ **Class diagrams**
    - – static object relationships, static aspect

Responsibilities

Obligations of an object in terms of its behavior.

- "Doing" responsibilities:
  - doing something itself
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- "Knowing" responsibilities:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

Responsibilities and Methods

- Responsibilities are assigned during OOD.
- Responsibilities are implemented using methods which either act alone or collaborate with other methods and objects.
  - Draw itself on the canvas.
  - Print a sale.
  - Provide access to relational database.

Increasing levels of granularity

Responsibility Assignment & Interaction Diagrams

- Interaction diagram shows the design decisions in assigning responsibilities to object(s).
  - The design decisions are reflected in what messages are sent to which classes of objects.
- The amount of time and effort spent on their generation, and the careful consideration of responsibility assignment, should absorb a significant percentage of the design phase of a project.
- **Creation of interaction diagram requires knowledge of responsibility assignment principles and codified patterns to improve the quality of their design.**

GRASP Patterns (1)

Objectives

- Learn to apply five of the GRASP principles or patterns.

GRASP Patterns

General Responsibility Assignment Software Patterns

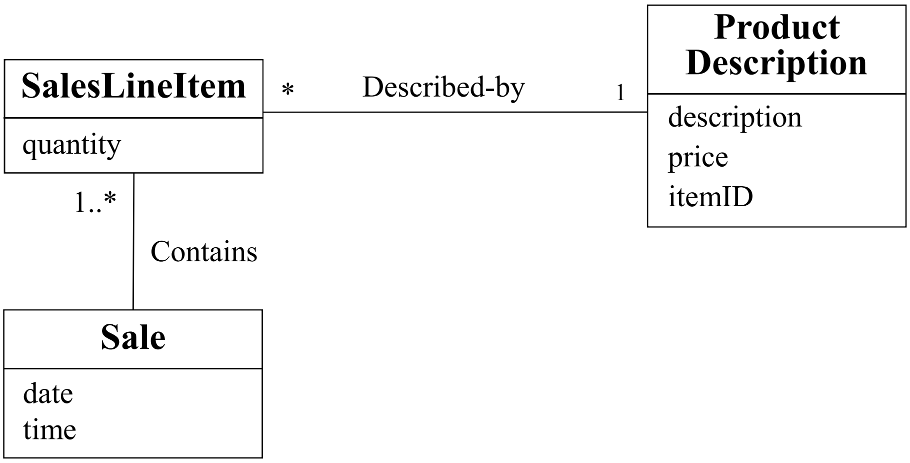
- Information Expert (or Expert)
- Creator
- High Cohesion
- Low Coupling
- Controller

Information Expert Pattern

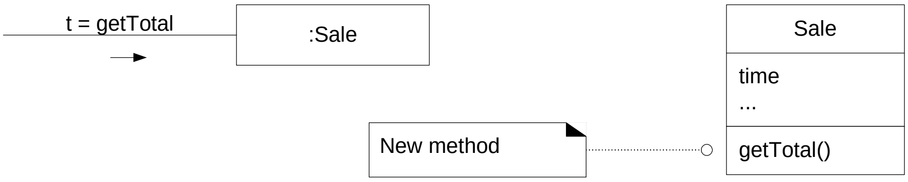
- Problem
  - What is the most basic principle by which responsibilities are assigned in OOD?
- Solution
  - **Assign a responsibility to the information expert -- the class that has the *information* necessary to fulfill the responsibility.**

Example: Expert Pattern

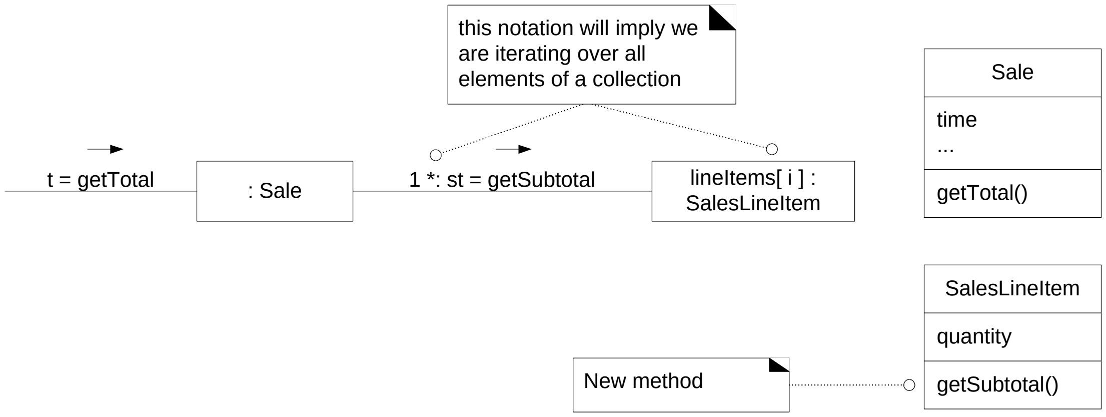
- We need to know the grand total of the sale. Then *Who should be responsible for knowing the grand total of the sale?*



Example: Expert Pattern (Cont'd)



Example: Expert Pattern (Cont’d)



this notation will imply we are iterating over all elements of a collection

t = getTotal

: Sale

1 *: st = getSubtotal

lineItems[i] : SalesLineItem

Sale

time
...

getTotal()

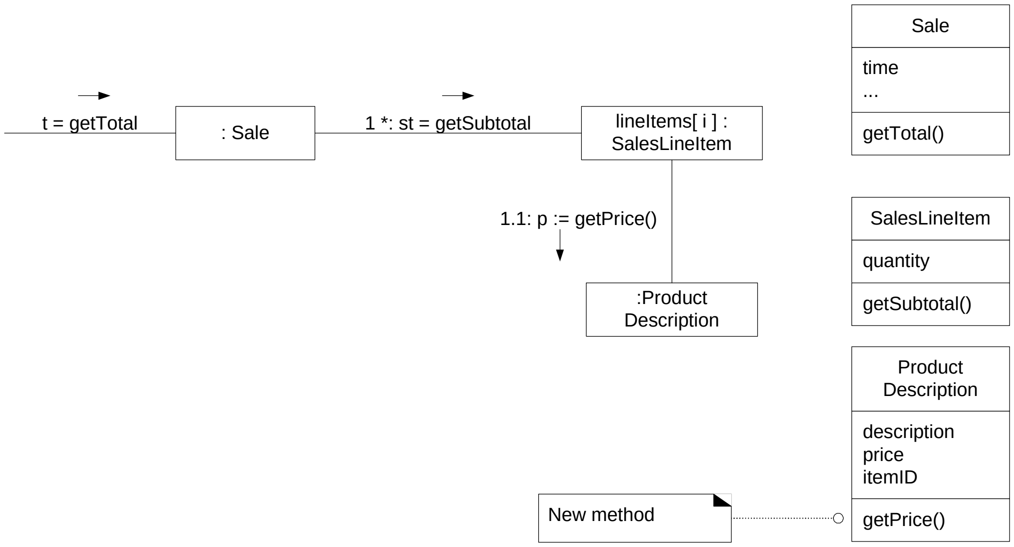
SalesLineItem

quantity

getSubtotal()

New method

Example: Expert Pattern (Cont’d)



t = getTotal

: Sale

1 *: st = getSubtotal

lineItems[i] : SalesLineItem

1.1: p := getPrice()

:Product Description

Sale

time
...

getTotal()

SalesLineItem

quantity

getSubtotal()

Product Description

description
price
itemID

getPrice()

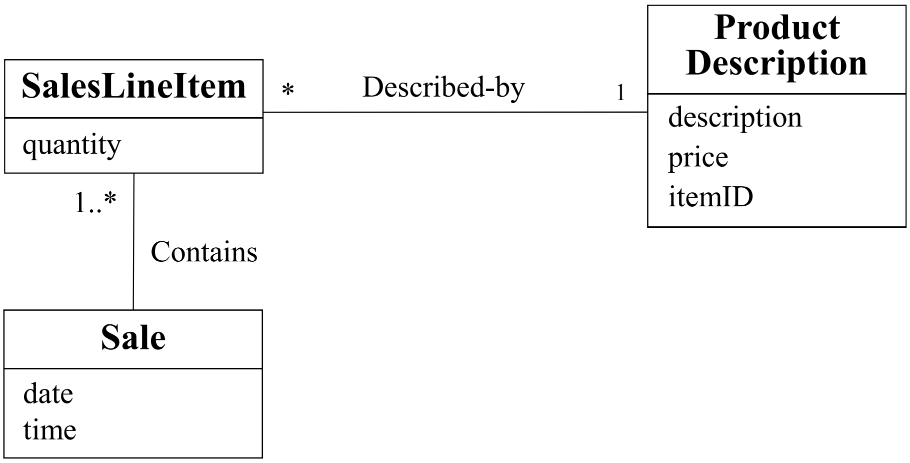
New method

Creator Pattern

- Problem
  - Who should be responsible for creating a new instance of some class?
- Solution
  - **Assign class B the responsibility to create an instance of class A if one of the following is true:**
    - **B *aggregates* A objects.**
    - **B *contains* A objects.**
    - **B *records* instances of A objects.**
    - **B *closely uses* A objects.**
    - **B *has the initializing data* that will be passed to A when it is created.**

Example: Creator Pattern

- *Who should be responsible for creating the SalesLineItem instance?*



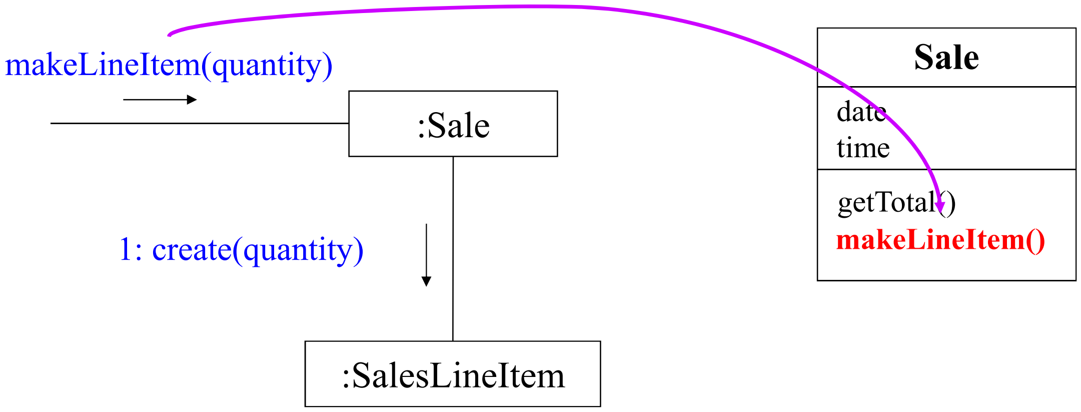
SalesLineItem
quantity

* Described-by 1

Product Description
description
price
itemID

1..*
Contains

Sale
date
time



makeLineItem(quantity)

:Sale

1: create(quantity)

:SalesLineItem

Sale

date
time

getTotal()
makeLineItem()

Low Coupling Pattern

- Problem
  - How to support low dependency and increased reuse?
- Solution
  - **Assign a responsibility so that coupling remains low.**

Low vs. High Coupling

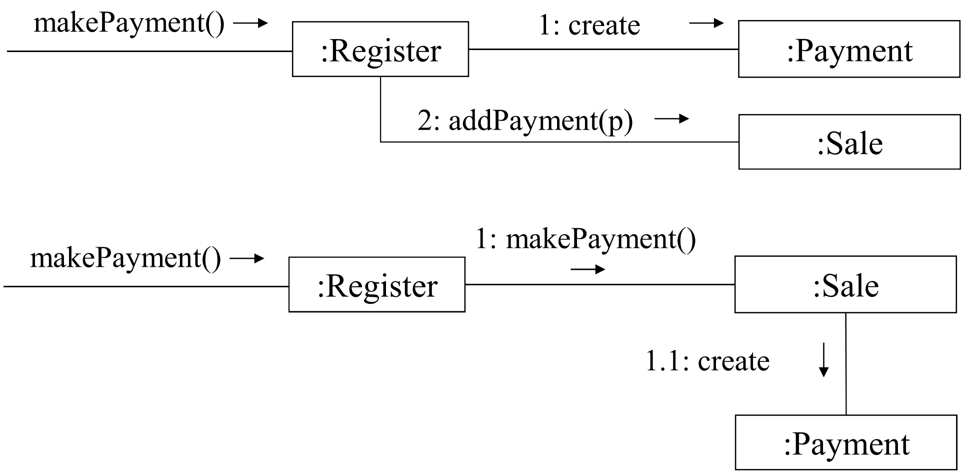
- Coupling represents the degree of dependencies of a class on other classes.
- Low coupling supports the design of classes that are
  - more independent, more reusable, and easier to understand in isolation ==> higher productivity
  - less effected by changes in other components.

Common Forms of Coupling

- *TypeX* has *TypeY* as its instance variable.
- *TypeX* has a method which has a parameter of type *TypeY*, or a return value of type *TypeY*, or a local variable of type *TypeY.*
- *TypeX* is a direct or indirect subclass of *TypeY*.
- *TypeY* is an interface, and *TypeX* implements that interface.

Example: Low Coupling Pattern

- We need to create a *Payment* and associate it with the *Sale*. Who should be responsible for this?



High Cohesion Pattern

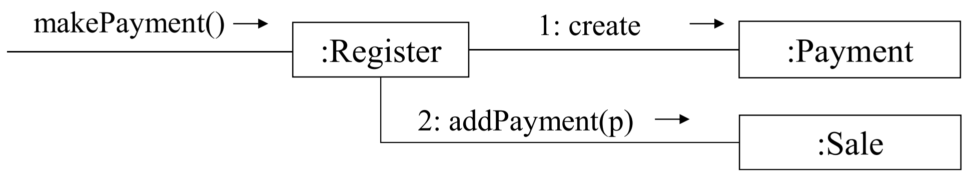
- Problem
  - How to keep complexity manageable?
- Solution
  - **Assign a responsibility so that cohesion remains high.**

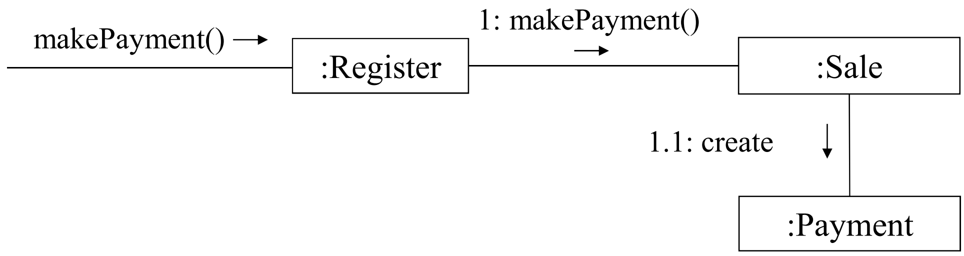
Low vs. High Cohesion

- (Functional) cohesion is a measure of how strongly related and focused the responsibilities of a class are.
- A class with highly related responsibilities, and which does not do a lot of work, has high cohesion.
- A class with low cohesion has the following problems:
  - hard to comprehend
  - hard to reuse
  - hard to maintain
  - delicate; constantly effected by change

Example: High Cohesion Pattern

- We need to create a *Payment* and associate it with the *Sale*. Who should be responsible for this?





High Cohesion & Low Coupling

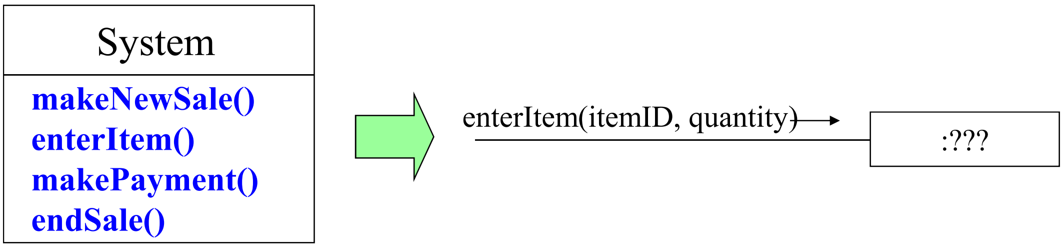
- The *high cohesion and low coupling* principles must be kept in mind during all design decisions.
- They are *evaluative patterns* which a designer applies while evaluating all design decisions.

The Controller Pattern

- Problem
  - Who should be responsible for handling a system event?
- Solution
  - **Assign the responsibility for handling a system event to a class representing one of the following choices:**
    - **Represents the overall system, device, or subsystem (façade controller)**
    - **Represents an artificial handler of all system events of a use case, usually named "<UseCaseName>Handler", or "<UseCaseName>Coordinator", "<UseCaseName>Session", or (use-case controller or session controller)**

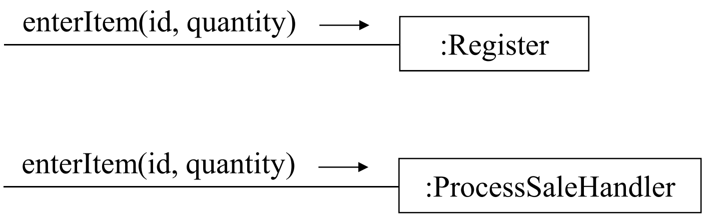
From System to Controllers

- During system behavior analysis, system operations are assigned to the type System, to indicate they are system operations. However, this does not mean that a class named System fulfills them during design.
- Rather, during design a controller class is assigned the responsibility for system operations.



Controllers

- A controller is a non-GUI object responsible for handling a system event. A controller defines the method for the system operations.
- In case of use-case handler, use the same controller for all the system events in the same use case. ***Why?***



Which one to choose?

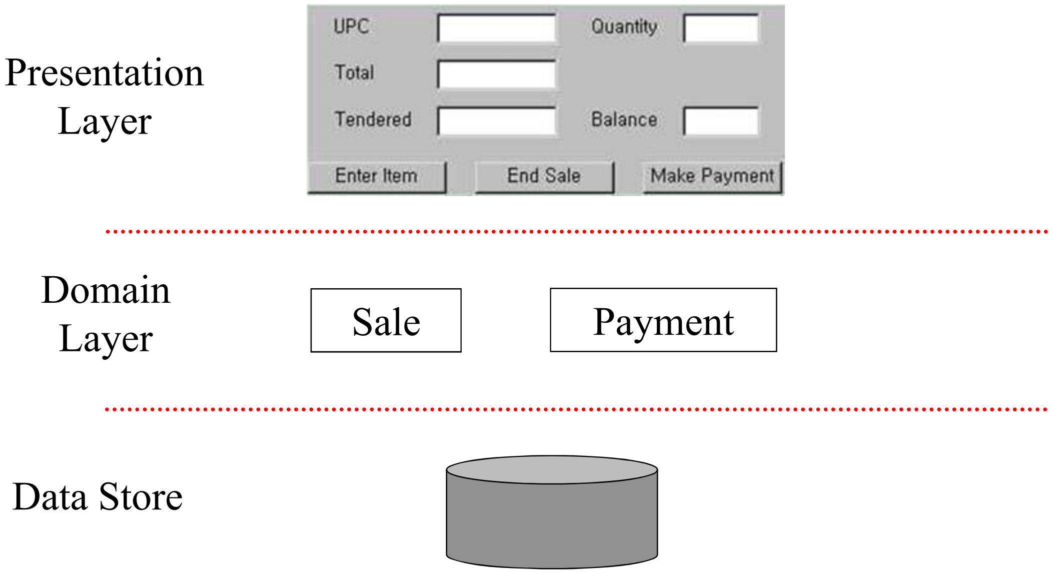
Bloated Controllers

- A bloated controller has a low cohesion -- unfocused and handling too many responsibilities.
- Signs of bloating include:
  - There is only a single controller receiving all system events in the system, and there are many of them: role or façade controller.
  - The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work.
  - A controller has many attributes, and maintains significant information about system or domain, or duplicates information found elsewhere.

Cures to Bloated Controller

- Add more controllers.
  - In addition to façade controller, use use-case controllers.
- Design the controller so that it primarily *delegates* the fulfillment of each system operation responsibility to other objects; it coordinates or controls the activity. It does not do much work itself.

Typical Layered Architecture



Presentation Layer

Domain Layer

Sale

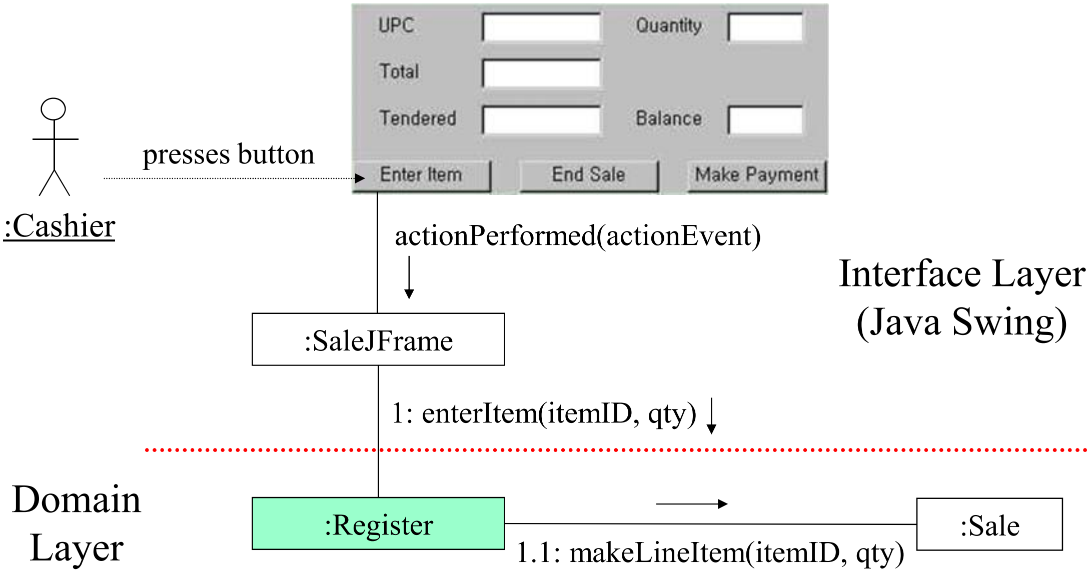
Payment

Data Store

Corollary of Controller Pattern

- The GUI objects (e.g., window objects, applets, etc) in the presentation layer should *not* have responsibility for handling system events.
- It is not a good design practice to make the presentation layer have parts of domain logic which reflects business or domain process. ***Why?***

The Good



:Cashier

presses button

actionPerformed(actionEvent)

:SaleJFrame

1: enterItem(itemID, qty)

Interface Layer
(Java Swing)

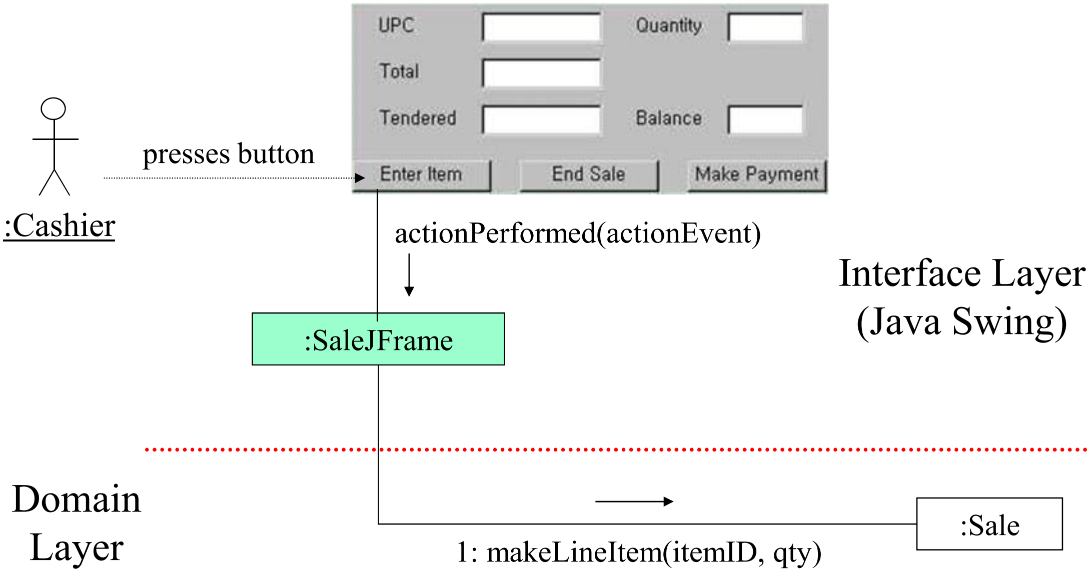
Domain
Layer

:Register

1.1: makeLineItem(itemID, qty)

:Sale

The Bad and/or The Ugly



:Cashier

presses button

actionPerformed(actionEvent)

:SaleJFrame

Interface Layer
(Java Swing)

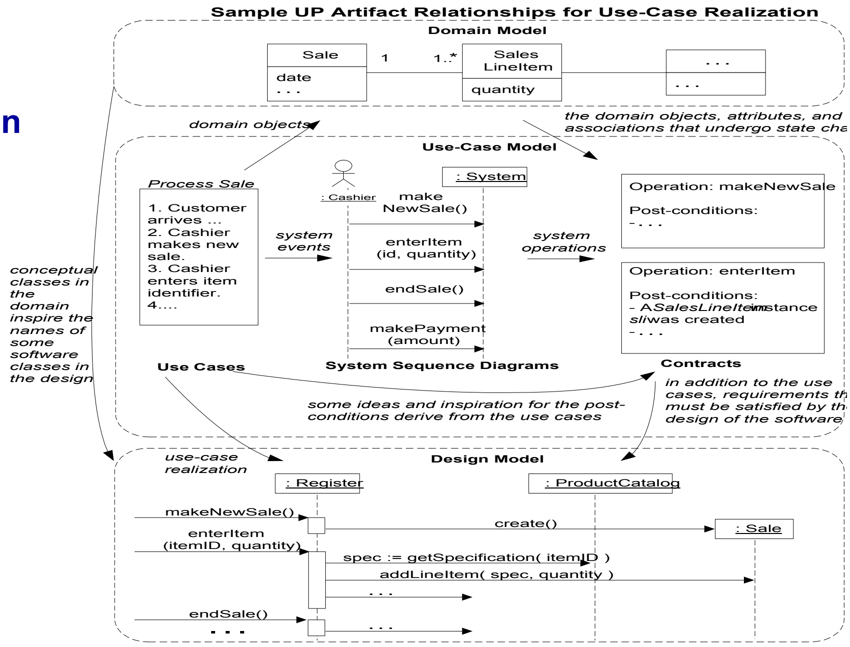
Domain
Layer

1: makeLineItem(itemID, qty)

:Sale

Object-Oriented Analysis and Design using UML and Patterns

Use Case Realization

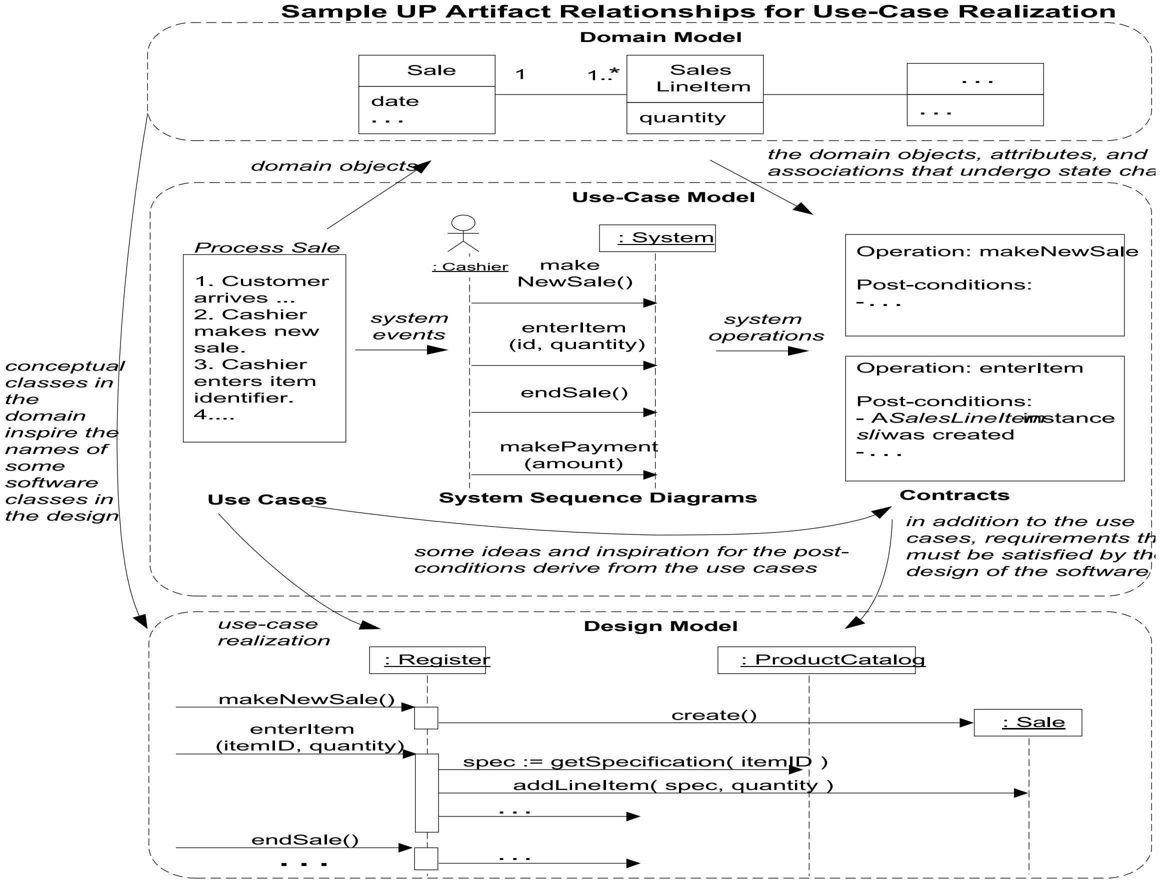

Sample UP Artifact Relationships for Use-Case Realization
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
domain objects
the domain objects, attributes, and associations that undergo state cha
Use-Case Model
Process Sale
1. Customer arrives ...
2. Cashier makes new sale.
3. Cashier enters item identifier.
4....
: Cashier
: System
make NewSale()
enterItem (id, quantity)
endSale()
makePayment (amount)
system events
system operations
Operation: makeNewSale
Post-conditions:
- . . .
Operation: enterItem
Post-conditions:
- A SalesLineItem instance sli was created
- . . .
conceptual classes in the domain inspire the names of some software classes in the design
Use Cases
System Sequence Diagrams
Contracts
in addition to the use cases, requirements th must be satisfied by th design of the software
some ideas and inspiration for the post-conditions derive from the use cases
use-case realization
Design Model
: Register
: ProductCatalog
: Sale
makeNewSale()
create()
enterItem (itemID, quantity)
spec := getSpecification(itemID)
addLineItem(spec, quantity)
. . .
endSale()
. . .
. . .

Use-Case Realization With GRASP Patterns

Objectives

- Design use case realizations.
- Apply GRASP to assign responsibilities to classes.
- Apply UML to illustrate and think through the design of objects.

Sample UP Artifact Relationships for Use-Case Realization



Domain Model

Sale | date | . . .

1 | 1..*

Sales LineItem | quantity

. . . | . . .

domain objects

the domain objects, attributes, and associations that undergo state cha

Use-Case Model

Process Sale

1. Customer arrives ...
2. Cashier makes new sale.
3. Cashier enters item identifier.
4....

system events

: Cashier

: System

make NewSale()

enterItem (id, quantity)

endSale()

makePayment (amount)

system operations

Operation: makeNewSale

Post-conditions:
- . . .

Operation: enterItem

Post-conditions:
- A *SalesLineItem* instance *sli* was created
- . . .

Use Cases

System Sequence Diagrams

Contracts

conceptual classes in the domain inspire the names of some software classes in the design

some ideas and inspiration for the post-conditions derive from the use cases

in addition to the use cases, requirements th must be satisfied by th design of the software

use-case realization

Design Model

: Register

: ProductCatalog

: Sale

makeNewSale()

create()

enterItem (itemID, quantity)

spec := getSpecification(itemID)

addLineItem(spec, quantity)

. . .

endSale()

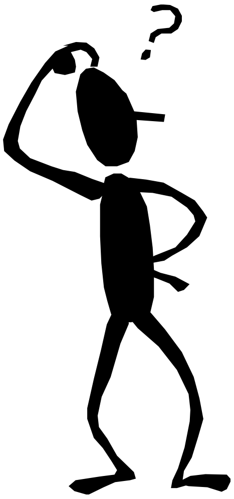
. . .

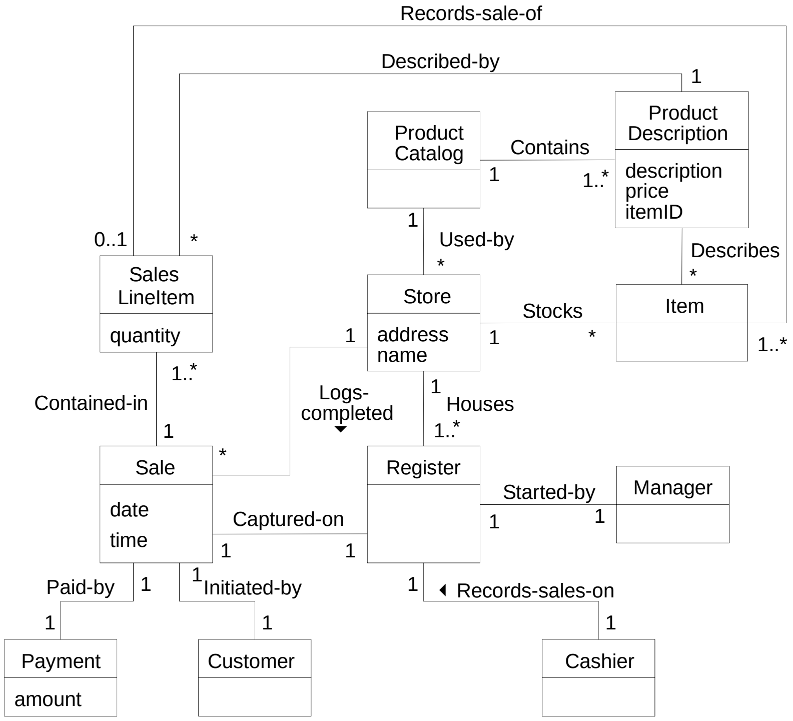
. . .

Case Study

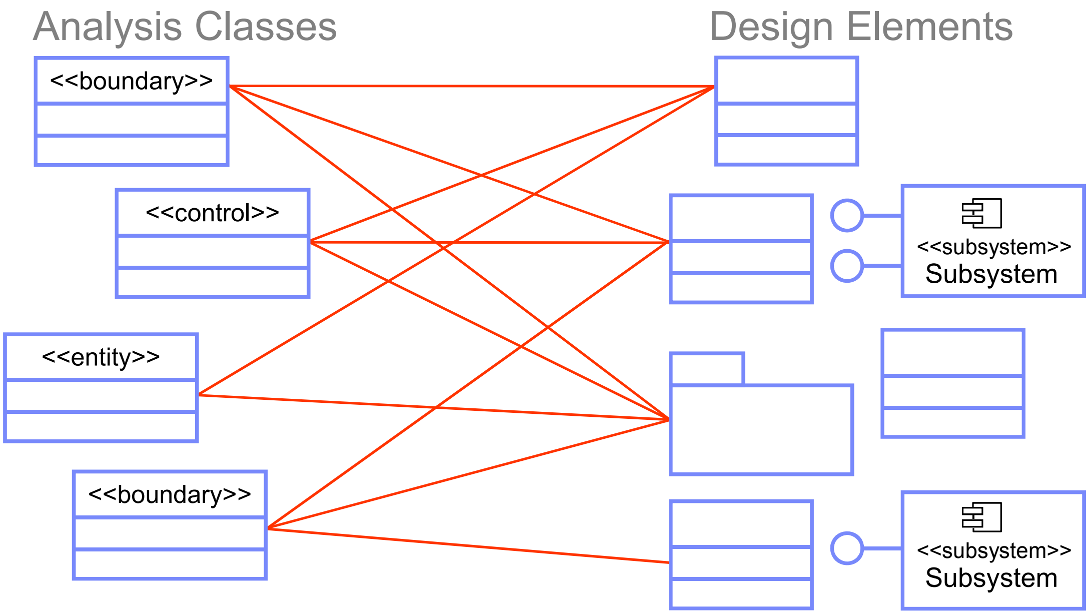
- We will consider the “*Process Sale*” and “*StartUp*”use cases and their associated system events.
  - Process Sales: makeNewSale, enterItem, endSale, makePayment
  - StartUp: startUp
- Using Controller pattern, the Register class will be chosen as the controller for handling the events.

Must the set of interacting objects be limited to this model?





From Analysis Classes to Design Elements



Many-to-Many Mapping

Analysis Classes vs. Design Elements

- Analysis classes:
  - Handle primarily functional requirements
  - Model objects from the “problem” domain
- Design elements:
  - Must also handle nonfunctional requirements
  - Model objects from the “solution” domain

Drivers When Identifying Design Elements

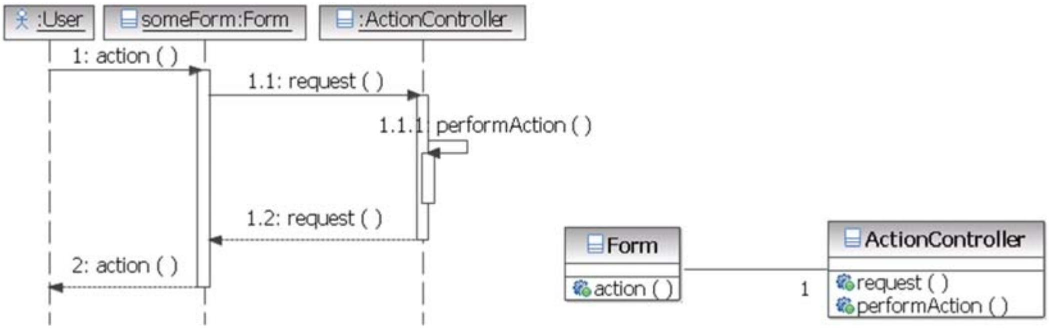
- Non-functional requirements, for instance consider:
  - Application to be distributed across multiple servers
  - Real-time system vs. e-Commerce application
  - Application must support different persistent storage implementations
- Architectural choices
  - For instance, .NET vs. Java Platform
- Technological choices
  - For instance, Enterprise Java Beans can handle persistence
- Design principles (identified early in the project’s life cycle)
  - Use of patterns (discussed in detail in the Identify Design Mechanisms module)
  - Best practices (industry, corporate, project)
  - Reuse strategy

Identifying Design Classes

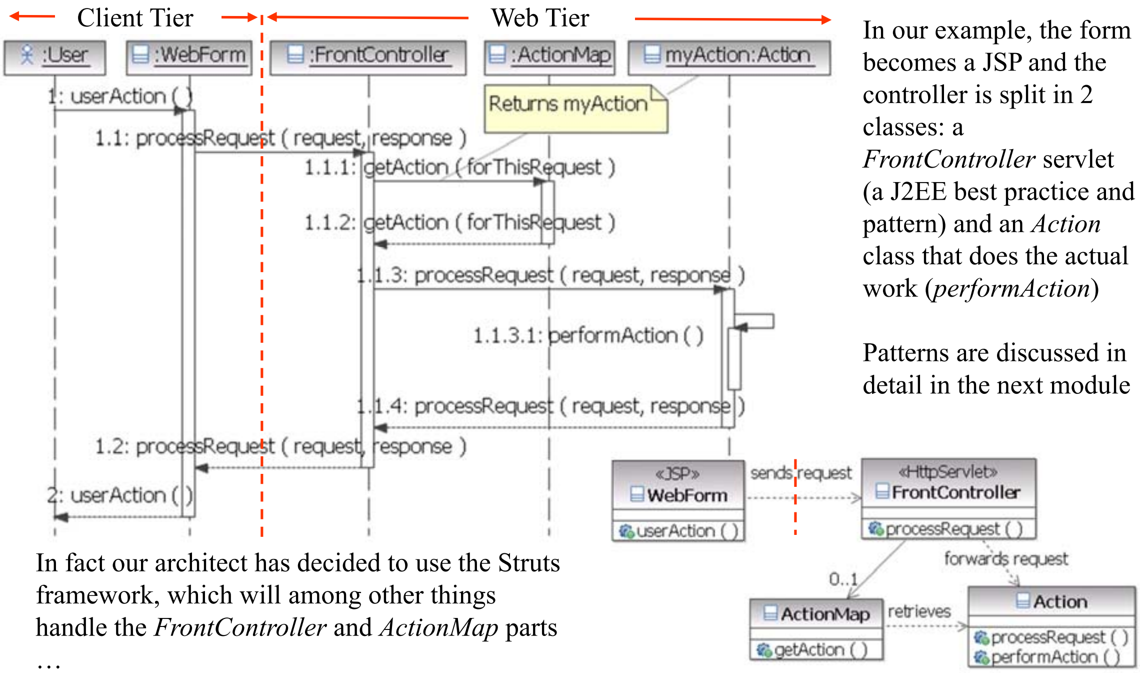
- An analysis class maps directly to a design class if:
  - It is a simple class
  - It represents a single logical abstraction
    - Typically, entity classes survive relatively intact into Design
- A more complex analysis class may:
  - Be split into multiple classes
  - Become a part of another class
  - Become a package
  - Become a subsystem (discussed later)
  - Become a relationship
  - Be partially realized by hardware
  - Not be modeled at all
  - Any combination …

Example: Analysis

- At the end of Analysis, let's assume we ended up with the (very simple and yet generic) model below
  - Our requirements stipulate that this is a typical J2EE Web application, with a thin client and a Web server…



Example: Design


Client Tier
Web Tier
:User
:WebForm
:FrontController
:ActionMap
myAction:Action
Returns myAction
1: userAction ( )
1.1: processRequest (request, response)
1.1.1: getAction (forThisRequest)
1.1.2: getAction (forThisRequest)
1.1.3: processRequest (request, response)
1.1.3.1: performAction ( )
1.1.4: processRequest (request, response)
1.2: processRequest (request, response)
2: userAction ( )
«JSP» WebForm
userAction ( )
sends request
«HttpServlet» FrontController
processRequest ( )
forwards request
0..1
ActionMap
getAction ( )
retrieves
Action
processRequest ( )
performAction ( )

In our example, the form becomes a JSP and the controller is split in 2 classes: a *FrontController* servlet (a J2EE best practice and pattern) and an *Action* class that does the actual work (*performAction*)

Patterns are discussed in detail in the next module

In fact our architect has decided to use the Struts framework, which will among other things handle the *FrontController* and *ActionMap* parts …

Contracts CO1: makeNewSale()

Operation: makeNewSale()

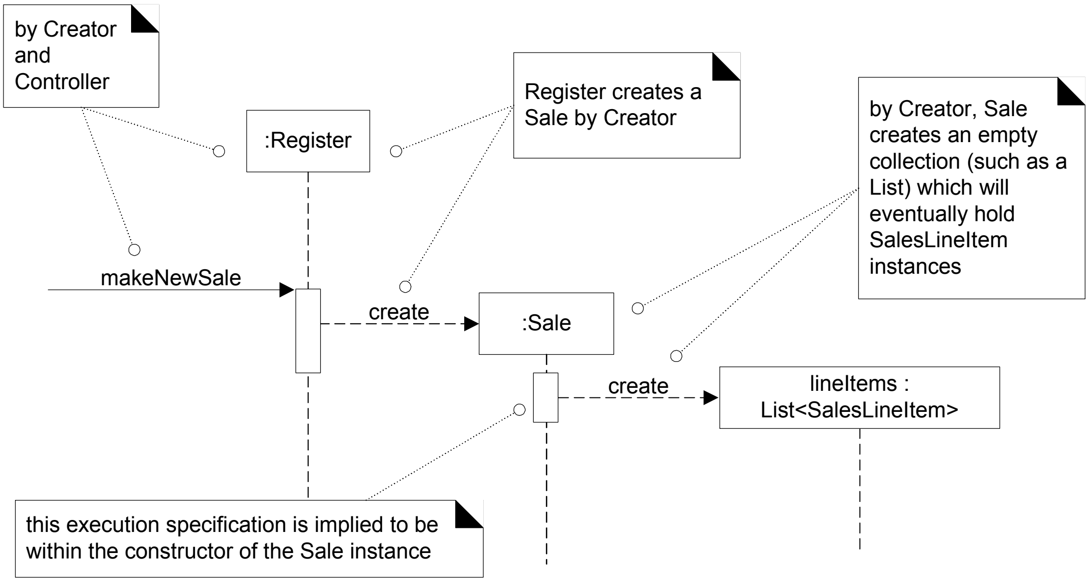
Cross References: Use Cases: Process Sale

Pre-conditions: none

Post-conditions:

- A ***Sale*** instance ***s*** was created (instance creation).
- ***s*** was associated with the ***Register*** (association formed).
- Attributes of *s* were initialized.

Creating a New Sale



by Creator and Controller
:Register
Register creates a Sale by Creator
by Creator, Sale creates an empty collection (such as a List) which will eventually hold SalesLineItem instances
makeNewSale
create
:Sale
create
lineItems : List<SalesLineItem>
this execution specification is implied to be within the constructor of the Sale instance

Contract CO2: enterItem

Operation: enterItem(itemID : ItemID, quantity : integer)

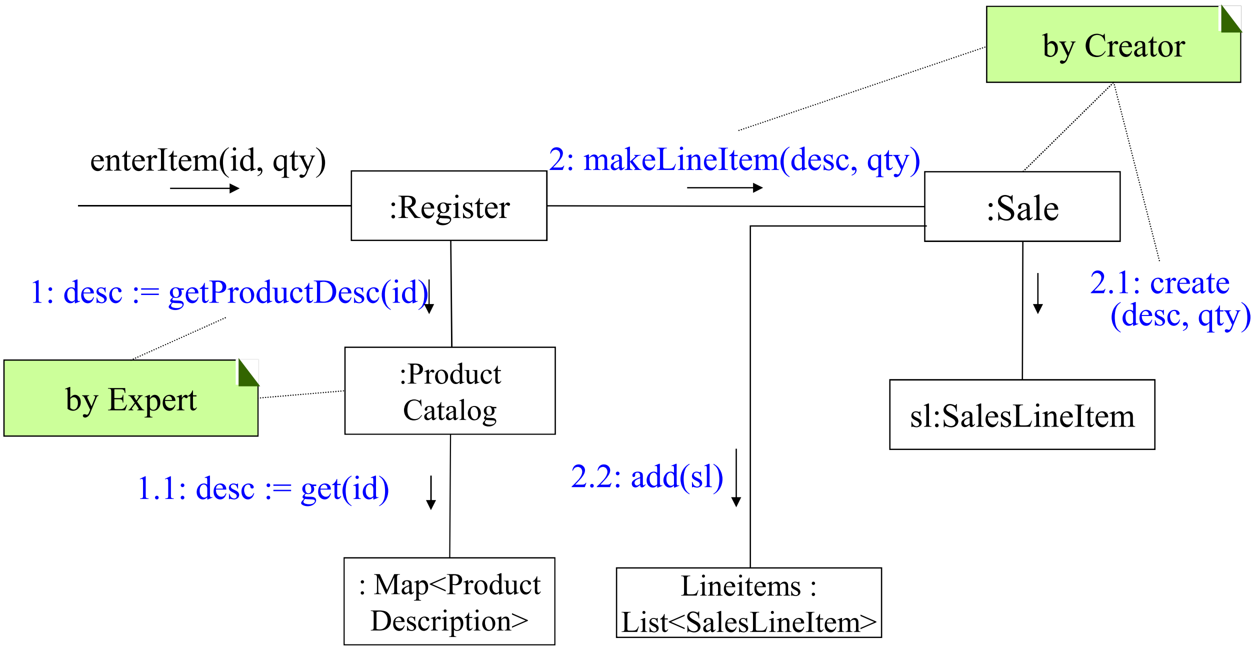
Cross References: Use Cases: Process Sale

Pre-conditions: There is a sale underway.

Post-conditions:

- A ***SalesLineItem*** instance ***sli*** was created (instance creation)
- ***sli*** was associated with the current ***Sale*** (association formed).
- ***sli.quantity*** become quantity (attribute modification).
- ***sli*** was associated with a ***ProductDescription***, based on *itemID* match (association formed).

Communication Diagram: enterItem



enterItem(id, qty)

:Register

2: makeLineItem(desc, qty)

:Sale

by Creator

1: desc := getProductDesc(id)

by Expert

:Product Catalog

2.1: create (desc, qty)

sl:SalesLineItem

1.1: desc := get(id)

2.2: add(sl)

: Map<Product Description>

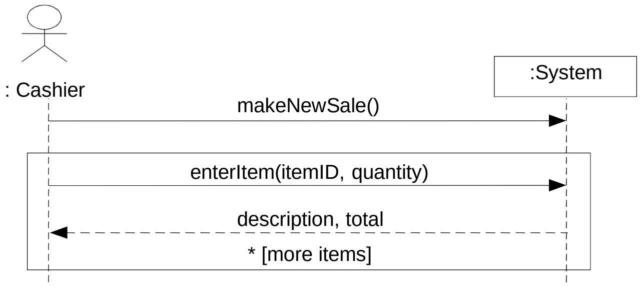
Lineitems : List<SalesLineItem>

Display item description & price

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.



- It is not the responsibility of domain objects (such as *Register* or *Sale*) to communicate with the user interface layer ==> *Model-View Separation Pattern*.
- However, the data to be displayed (item *description* and its *price*) should be known at this time.

Model-View Separation

Context/Problem

- It is desirable to de-couple domain objects (i.e., models) from GUIs (i.e., views), to *support increased reuse of domain objects*, and *minimize the impact of changes in the interface* upon the domain objects. What to do?

Solution

- Define the domain (model) classes so that they do not have direct coupling or visibility to the GUI (view) classes, and so that application data and functionality is maintained in domain classes, not GUI classes.

Motivation for MV Separation

- To support cohesive model definitions that focus on the domain processes, rather than on computer 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.
- To allow execution of the model layer independent of the GUI layer, such as in a message-processing or batch-mode system.
- To allow easy porting of the model layer to another GUI framework.

Displaying the Sale Total

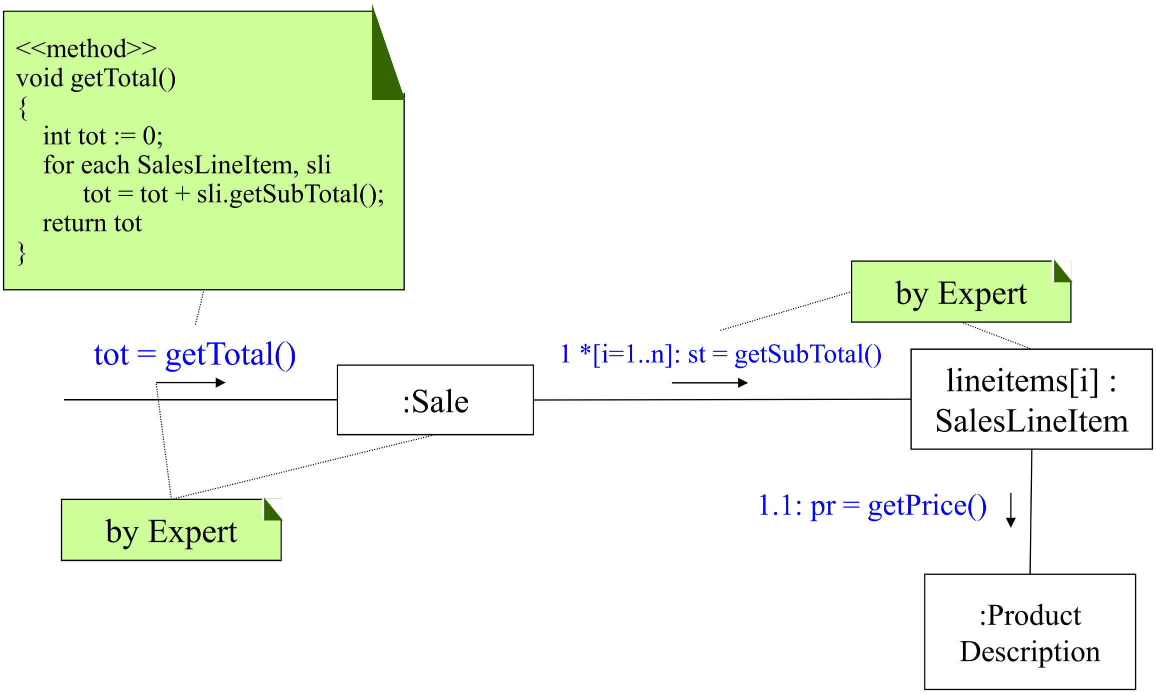
- Remember the Model View Separation Pattern.
- During the creation of collaboration diagrams, do not be concerned with the display of information, *except* insofar as the required information is known.
- Ensure that all information that must be displayed is known and available from the domain objects.

Calculating the Sale Total

1. State the responsibility:
   - Who should be responsible for knowing the sale total?
2. Summarize the information required:
   - The sale total is the sum of the subtotals of all the sales line-items.
   - Sales line-item subtotal := line-item quantity * product description price
3. List the information required to fulfill this responsibility and the classes (i.e., Expert) that know this information.

ProductDescription.price	*ProductDescription*
SalesLineItem.quantity	*SalesLineItem*
all the *SalesLineItems* in the current sale	*Sale*

Communication Diagram: getTotal()



```
<<method>>
void getTotal()
{
   int tot := 0;
   for each SalesLineItem, sli
        tot = tot + sli.getSubTotal();
   return tot
}
```

Contract CO3: endSale()

Operation: endSale()

Cross References: Use Cases: Process Sale

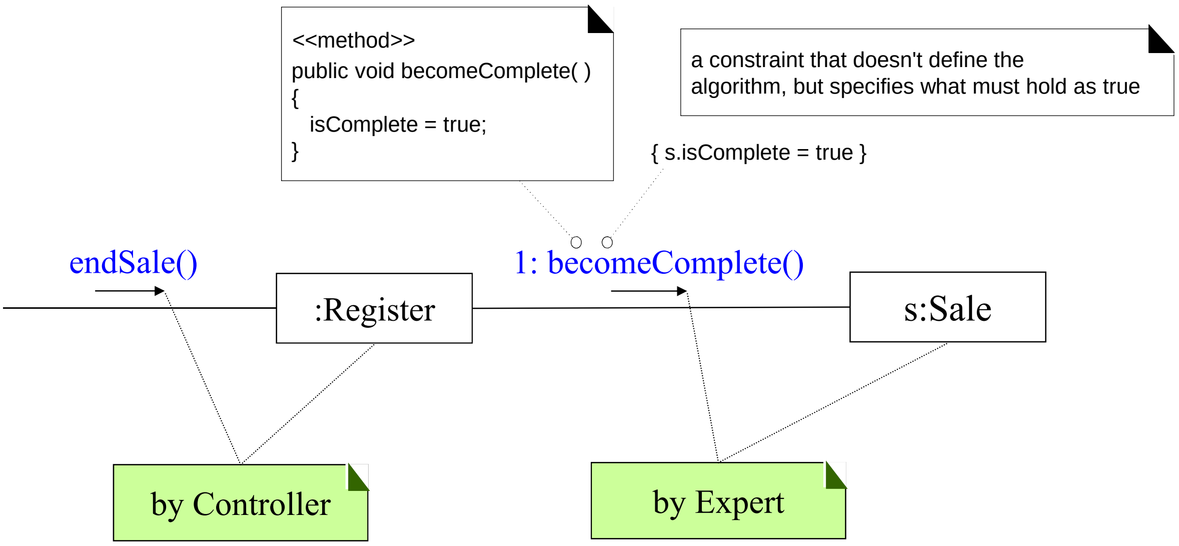
Pre-conditions: There is a sale underway.

Post-conditions:

- ***Sale.isComplete*** become true (attribute modification).

Communication Diagram: endSale()

Constraints, Notes, and Algorithms



```
<<method>>
public void becomeComplete( )
{
   isComplete = true;
}
```

a constraint that doesn't define the algorithm, but specifies what must hold as true

{ s.isComplete = true }

endSale()

:Register

1: becomeComplete()

s:Sale

by Controller

by Expert

Collaboration Diagram: makePayment()

- *Study for yourself!*

Comments on *StartUp* System Operation

- *startUp* operation abstractly represents the initialization phase of execution when an application is launched. It depends on the programming language and OS.
- A common design idiom is to create an ***initial domain object***, which is responsible for creating other domain objects.

```
public class RegisterApplet extends Applet {
   public void  init() {  register = store.getRegister();   }
   …
   private Store store  = new Store();    // initial domain object
   private Register register;
   private Sale sale;
}
```

Interpretation of *startUp*

1. In one collaboration diagram, send a “*<<create>>*” message to create the initial domain object.
2. (optional) If the initial object is taking control of the process, in a second collaboration diagram send a “*run*” message (or something equivalent) to the initial object.

- Create the start up collaboration diagram last. ***Why?***

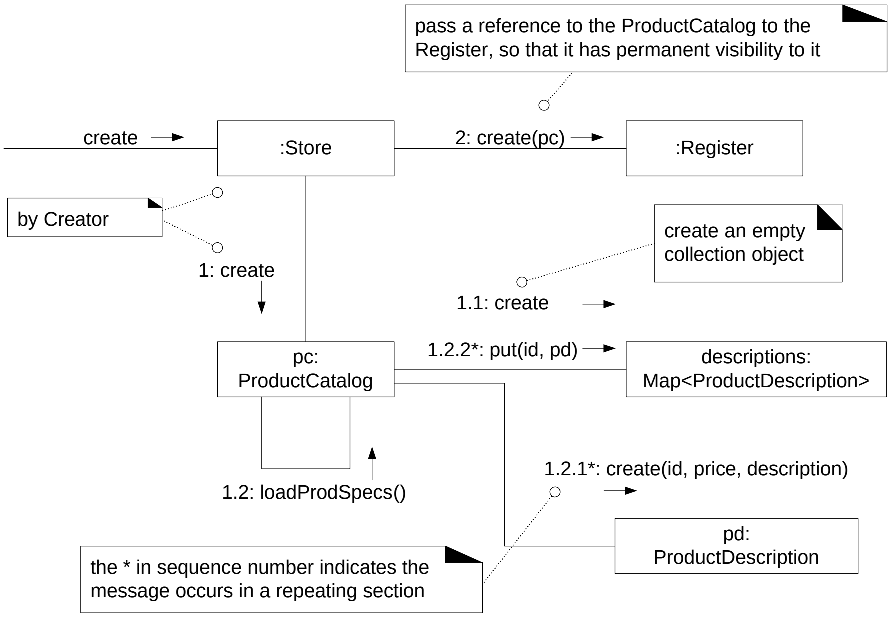
Choosing The Initial Domain Object

- What should be the class of the initial domain object?
- Choose as an initial domain object:
  - A class representing the entire logical information system
    - *Register*
  - A class representing the overall business or organization
    - *Store*
- The choice may be influenced by High cohesion and Low coupling principles.

Derived Initialization Tasks From Prior Design Work

- A ***Store***, ***Register***, ***ProductCatalog*** and ***ProductDescription***s needs to be created.
- The ***ProductCatalog*** needs to be associated with ***ProductDescription***s.
- ***Store*** needs to be associated with ***ProductCatalog***.
- ***Store*** needs to be associated with ***Register***.
- ***Register*** needs to be associated with ***ProductCatalog***.

Collaboration Diagram: startUp()



pass a reference to the ProductCatalog to the Register, so that it has permanent visibility to it

create

:Store

2: create(pc)

:Register

by Creator

create an empty collection object

1: create

1.1: create

pc: ProductCatalog

1.2.2*: put(id, pd)

descriptions: Map<ProductDescription>

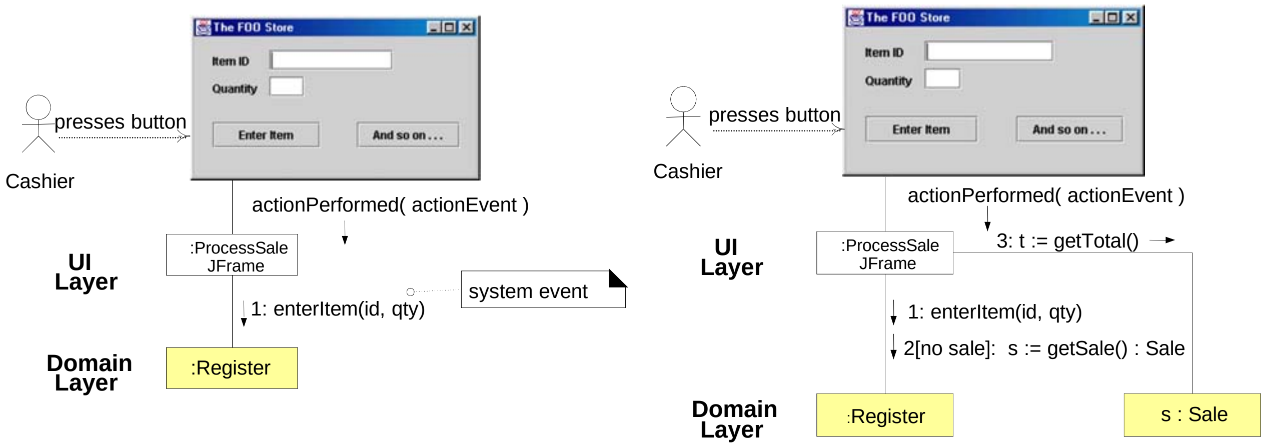
1.2: loadProdSpecs()

1.2.1*: create(id, price, description)

pd: ProductDescription

the * in sequence number indicates the message occurs in a repeating section

Connecting UI Layer to Domain Layer



The FOO Store

Item ID

Quantity

Enter Item

And so on ...

presses button

Cashier

actionPerformed(actionEvent)

UI Layer

:ProcessSale JFrame

system event

1: enterItem(id, qty)

Domain Layer

:Register

The FOO Store

Item ID

Quantity

Enter Item

And so on ...

presses button

Cashier

actionPerformed(actionEvent)

UI Layer

:ProcessSale JFrame

3: t := getTotal()

1: enterItem(id, qty)

2[no sale]: s := getSale() : Sale

Domain Layer

:Register

s : Sale

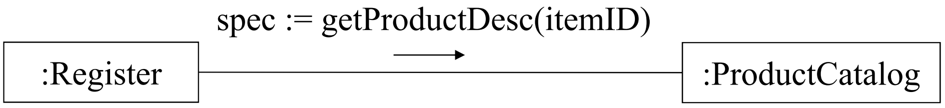
Determining Visibility & Creating Class Diagrams

Objectives

- Identify four kinds of visibility.
- Design to establish visibility.
- Understand the interplay between interaction & class diagrams.

Visibility

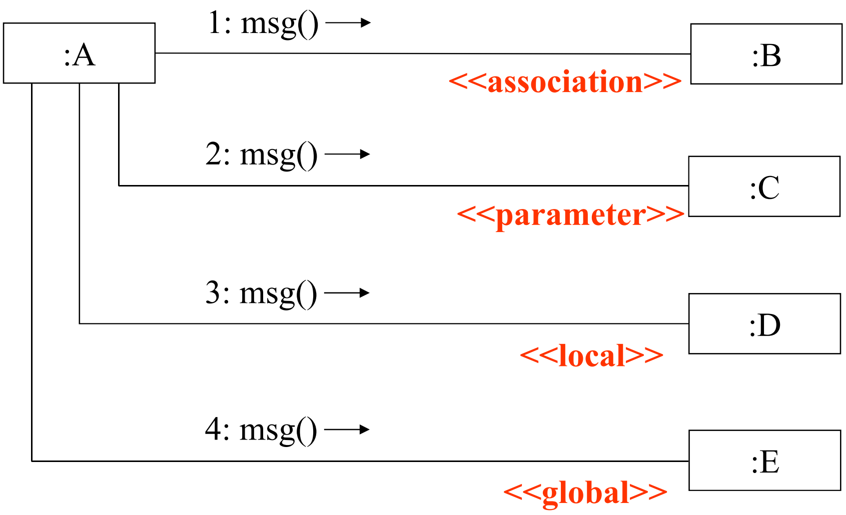
- For an object *A* (sender) to send a message to an object *B* (receiver), *B* must be *visible* to *A*.
- Visibility is the ability of one object to “see” or have reference to another.



Visibility From A to B

- Attribute visibility
  - *B* is an attribute of *A*.
- Parameter visibility
  - *B* is a parameter of a method of *A*.
- Local visibility
  - *B* is declared as a (non-parameter) local object in a method of *A*.
- Global visibility
  - *B* is in some way globally visible.

UML Stereotypes for Visibility



Activities and Dependencies

The creation of design class diagram (DCD) relies upon

- *Interaction diagram* -- from this, the designer identifies the software classes that participate in the solution, plus the methods of the classes.
- *Domain model* -- from this, the designer adds detail (such as attributes) to the class definitions.

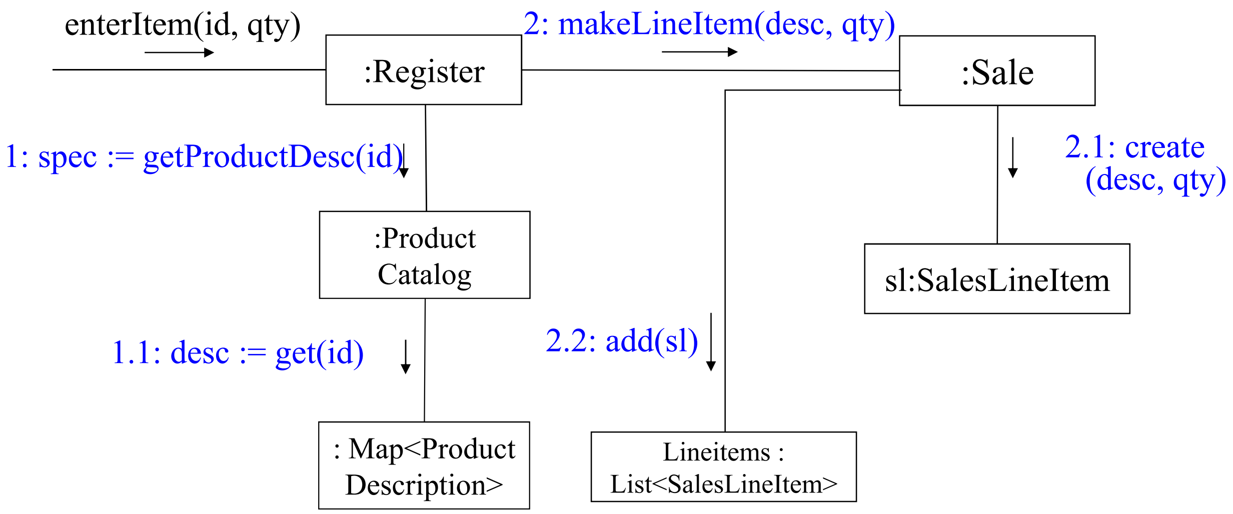
Interplay Between Interaction and Class Diagrams

- In practice, interaction diagrams (dynamic model) and class diagrams (static model) are usually created in parallel.
- Draft class diagrams can be sketched early in the design phase prior to the creation of interaction diagrams ==> can also be used as an alternative to CRC cards.
- The designer must iterate between the two kinds of models, driving them to converge on acceptable solution.

How to Create Design Class Diagram

1. Identify and draw all the classes participating in the software solution by analyzing the interaction diagram.
2. Duplicate the attributes from the associated concepts in the domain model.
3. Add method names by analyzing the interaction diagram.
4. Add type information to the attributes and methods.
5. Add the associations necessary to support the required attribute visibility.
6. Add navigability arrows to the associations to indicate the direction of attribute visibility.
7. Add dependency relationship lines to indicate non-attribute visibility.

Communication Diagram: enterItem



enterItem(id, qty)

:Register

2: makeLineItem(desc, qty)

:Sale

1: spec := getProductDesc(id)

:Product Catalog

2.1: create (desc, qty)

sl:SalesLineItem

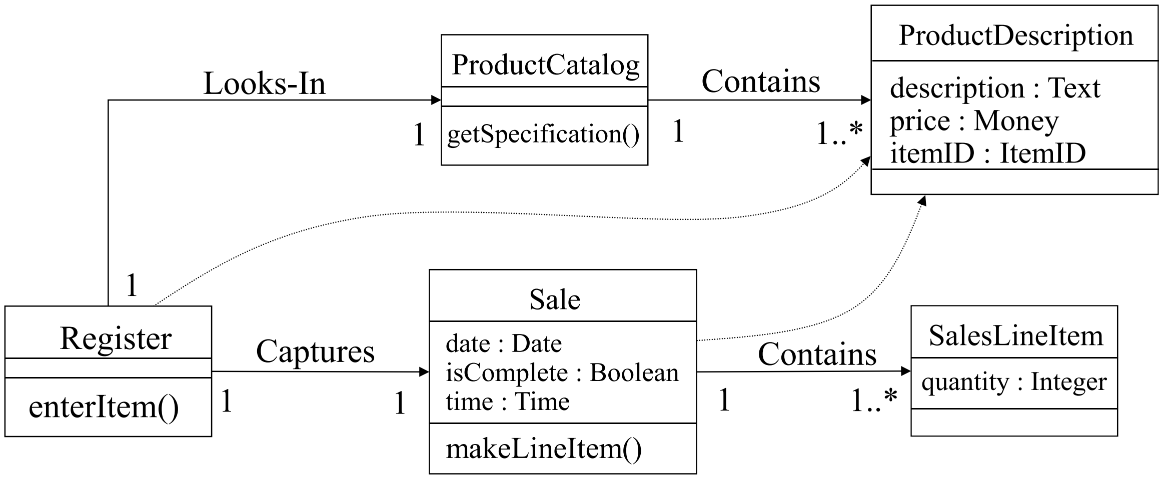
1.1: desc := get(id)

2.2: add(sl)

: Map<Product Description>

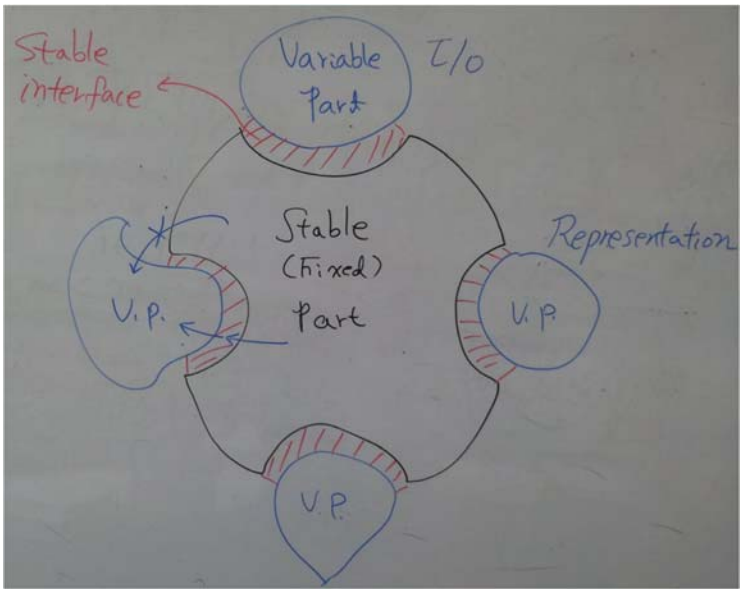
Lineitems : List<SalesLineItem>

Derived Design Class Diagram



Object-Oriented Analysis and Design using UML and Patterns

GRASP Pattern (II)



GRASP: More Patterns For Assigning Responsibilities

Objectives

- Learn to apply the remaining GRASP patterns.

GRASP Patterns

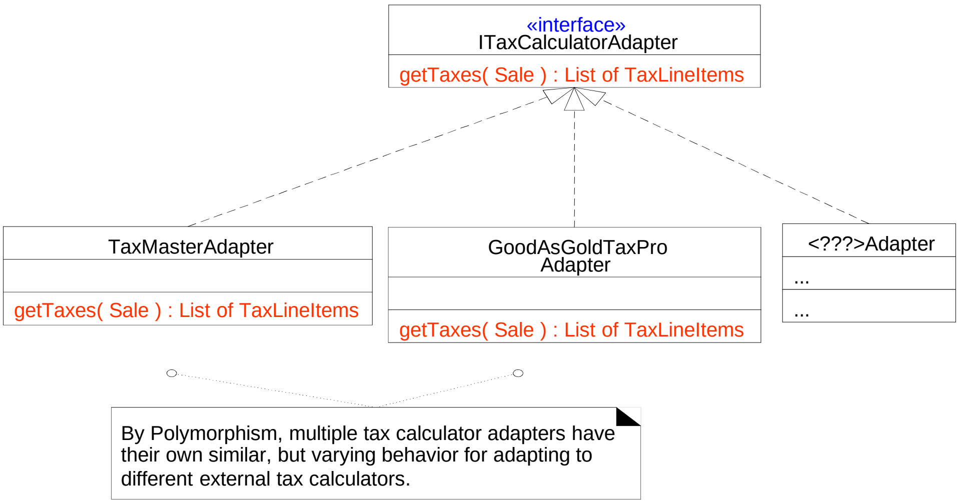
- Information Expert (or Expert)
- Creator
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Polymorphism

- Problem
  - How to handle alternatives based on type? How do you create pluggable software components?
- Solution
  - **When related alternatives or behaviors vary by type (class), assign responsibility for the behavior -- using polymorphic operations -- to the types for which the behavior varies.**
  - **Corollary: Do not test for the type of an object and use conditional logic.**

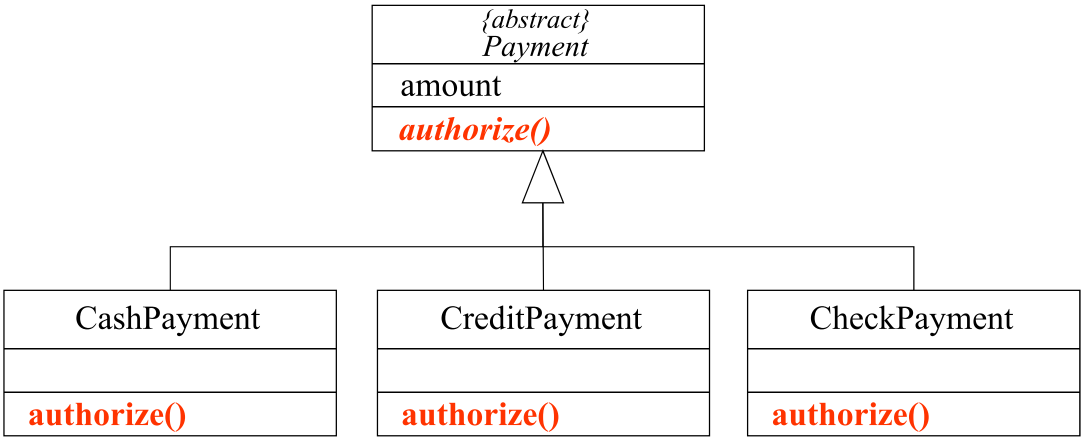
Example: Polymorphism

- In the NextGen POS application, who should be responsible for handling the varying external tax calculator interfaces (like TCP sockets, SOAP, or RMI)?


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
<???>Adapter
...
...
By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

Example: Polymorphism

- In the NextGen POS application, who should be responsible for authorizing different kinds of payments?

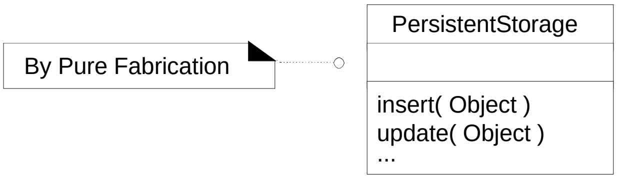


Pure Fabrication

- Problem
  - What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling or other goals but solutions offered by Expert (for example) are note appropriate?
    - There are situations in which assigning responsibilities only to domain classes leads to problems in terms of poor cohesion and coupling, or low reuse potential.
- Solution
  - **Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent anything in the problem domain -- something made up (i.e., *fabrication*).**

Example: Pure Fabrication

- Suppose that support is needed to save *Sale* instances in a relational database. Who should do that?
- By Information Expert, there is some justification to assign this responsibility to the *Sale* class itself, but …



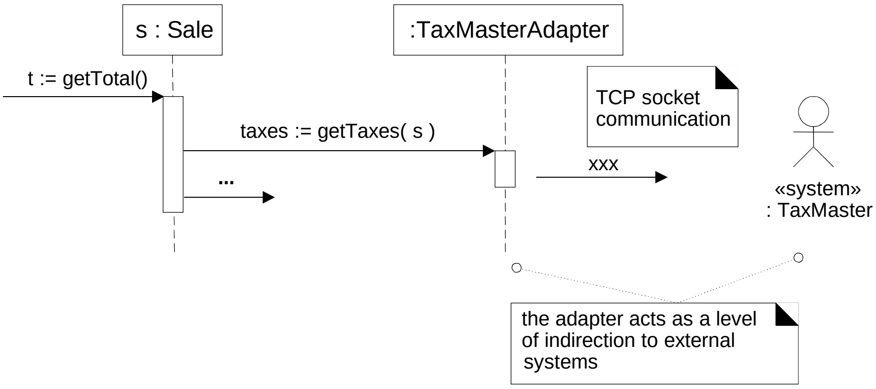
- Note: Pure Fabrications are usually a consequence of *behavior decomposition* rather than *representational decomposition*. That is, they are a kind of function-centric or behavioral objects. ***Caution***: *Do not overuse!*

Indirection

- Problem
  - Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that Low Coupling is supported and reuse potential remains higher?
- Solution
  - **Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an *indirection* between the other components or services.**

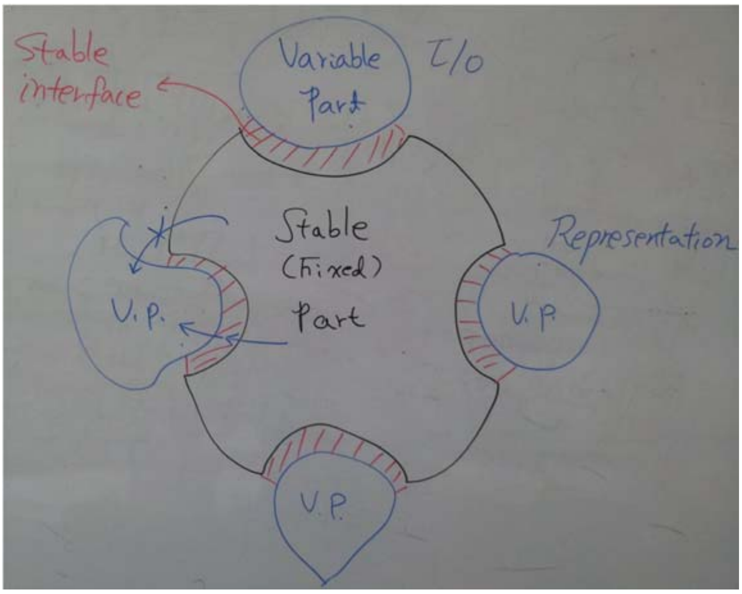
Example: Indirection

- *TaxCalculatorAdapter* objects acts as intermediaries to the external tax calculators. By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces.



- Decoupling *Sale* and the relational database through *PersistentStorage* is also an example of Indirection.

Protected Variations



- Problem
  - How to assign objects, subsystems, and systems so that the ***variations or instability*** in these elements does ***not have an undesirable impact on other elements***?
- Solution
  - **Identify points of predicted variations or instability; assign responsibilities to create a stable interface around them.**

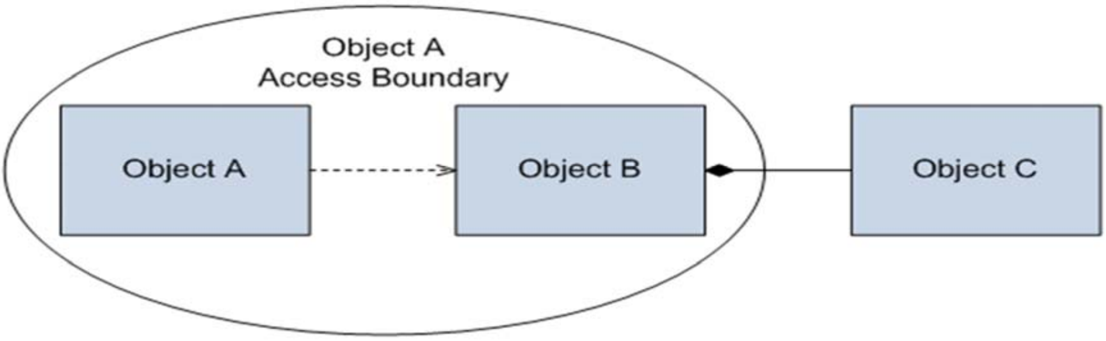
Mechanism Motivated by PV

- Core Protected Variations Mechanisms
  - Data encapsulation, interfaces, polymorphism, brokers, virtual machines, etc.
- Data-Driven Designs
- Service Lookup
- Interpreter-Driven Design
- Reflective or Meta-level Designs
- Uniform access
- The Liskov Substitution Principle (LSP)
- Structure-Hiding Designs (Don’t Talk to Strangers)

Don’t Talk to Strangers

- Problem
  - Who, to avoid knowing about the structure of indirect objects?
  - If an object has knowledge of the internal structure of other objects, it suffers from high coupling.
  - If a client has to use a service or obtain information from an indirect object, how can it do so without being coupled to knowledge of the internal structure of its direct server or indirect objects?

Don’t Talk to Strangers



- Solution
  - **Assign the responsibility to a client’s direct object (i.e., *familiar*) to collaborate with an indirect object (i.e., *stranger*), so that the client does not know about the indirect object.**

Law of Demeter
(Principle of Least Knowledge)

Constrains on what objects you should send a message to within a method.

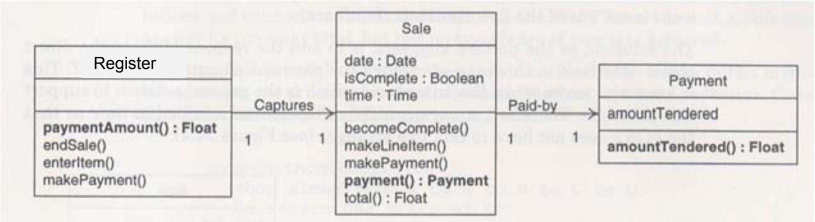
- The *this* object (or *self*).
- The parameter of the method.
- An attribute of *self*.
- An element of a collection which is an attribute of *self*.
- An object created within the method.

Example: Don’t Talk ...

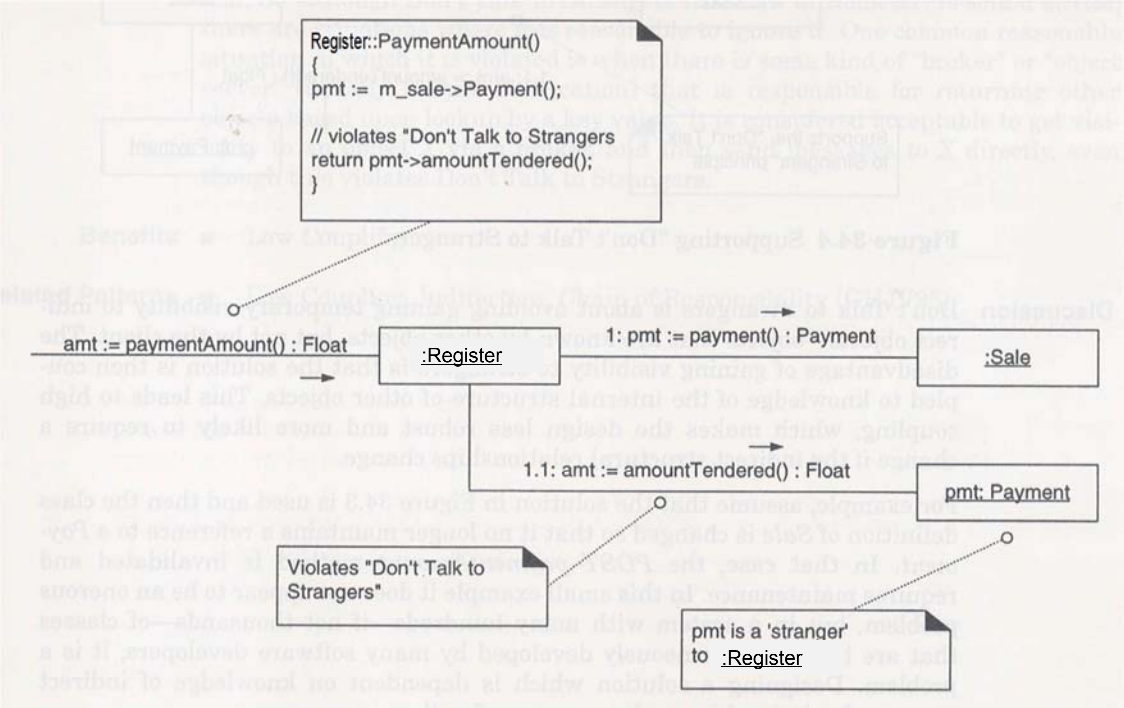
Assume that:

- *Register* instance has an attribute referring to a *Sale*, which has an attribute referring to a *Payment*.
- *Register* supports the *paymentAmount()* operation, which returns the current amount tendered.
- *Sale* supports *payment()* operation, which returns the *Payment* instance associated with the *Sale*.

  Then, how should the *Register* instance return the payment amount?



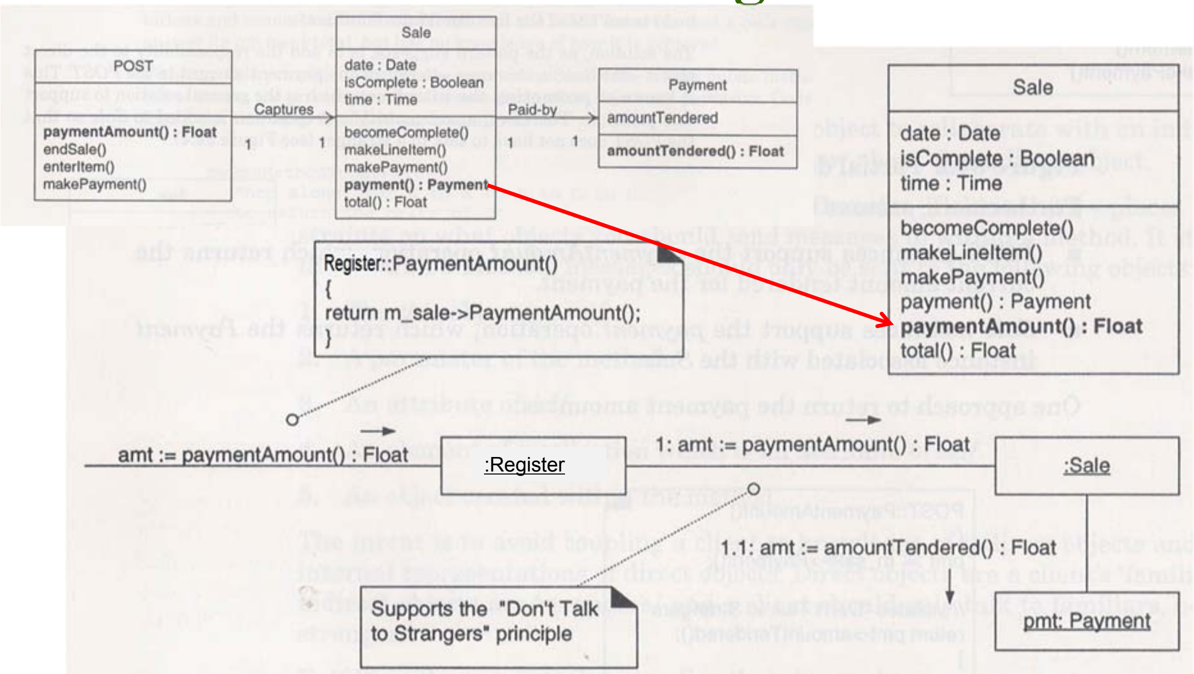
A Solution Violating Don’t Talk …



```
Register::PaymentAmount()
{
pmt := m_sale->Payment();

// violates "Don't Talk to Strangers
return pmt->amountTendered();
}
```

amt := paymentAmount() : Float
:Register
1: pmt := payment() : Payment
:Sale
1.1: amt := amountTendered() : Float
pmt: Payment
Violates "Don't Talk to Strangers"
pmt is a 'stranger' to :Register

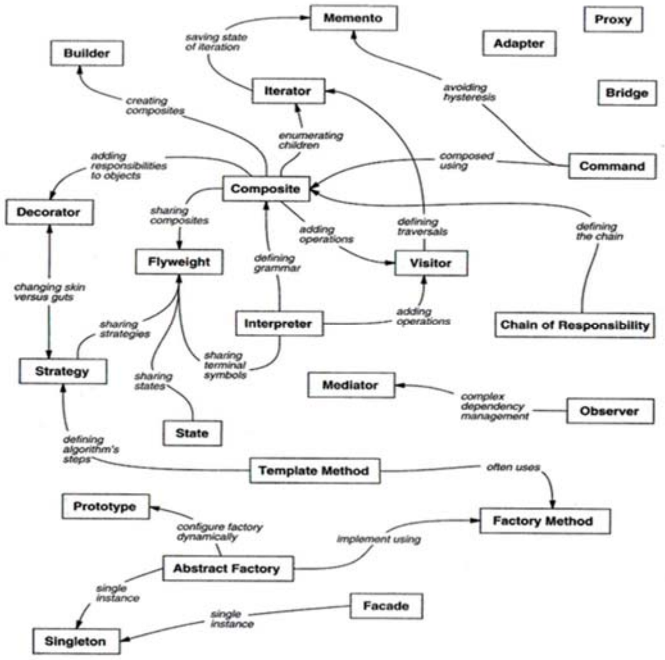
A Solution Advocating Don’t Talk …



```
Register::PaymentAmount()
{
return m_sale->PaymentAmount();
}
```

Object-Oriented Analysis and Design using UML and Patterns

More Object Design with GoF



More Object Design with GoF

Objectives

Apply some GoF design patterns

Show GRASP principles as a generalization of other design patterns

Adapter Pattern

Problem

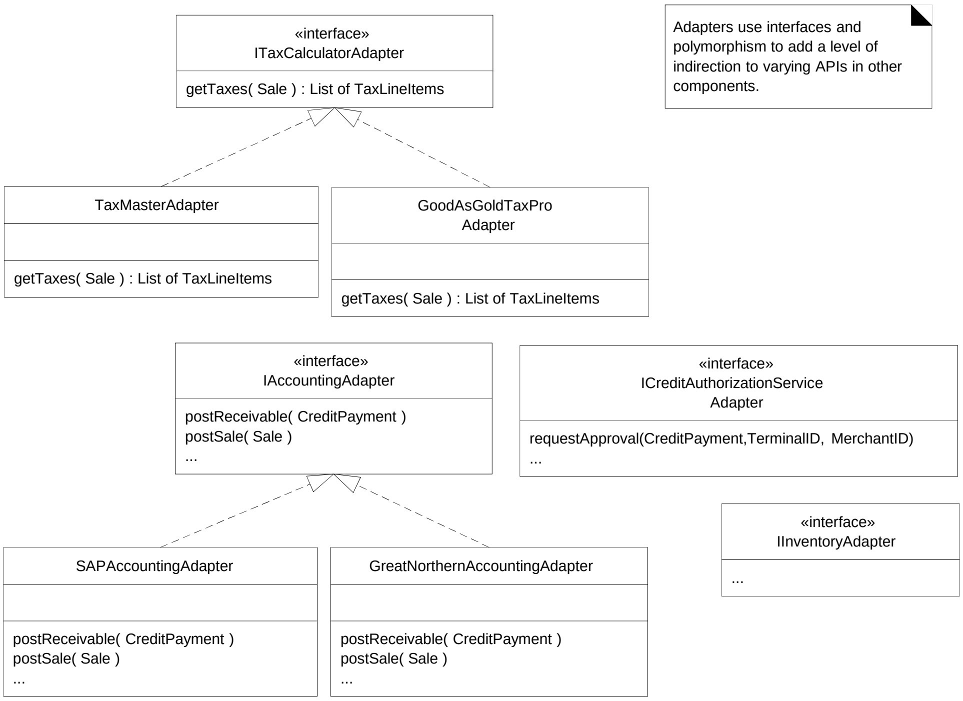
How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

Solution

Convert the original interface of a component into another interface, through an intermediate adapter object

The NextGen POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed.

Adapter Pattern (Cont'd)



«interface»
ITaxCalculatorAdapter

getTaxes(Sale) : List of TaxLineItems

Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.

TaxMasterAdapter

getTaxes(Sale) : List of TaxLineItems

GoodAsGoldTaxPro Adapter

getTaxes(Sale) : List of TaxLineItems

«interface»
IAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

«interface»
ICreditAuthorizationService Adapter

requestApproval(CreditPayment,TerminalID, MerchantID)
...

«interface»
IInventoryAdapter

...

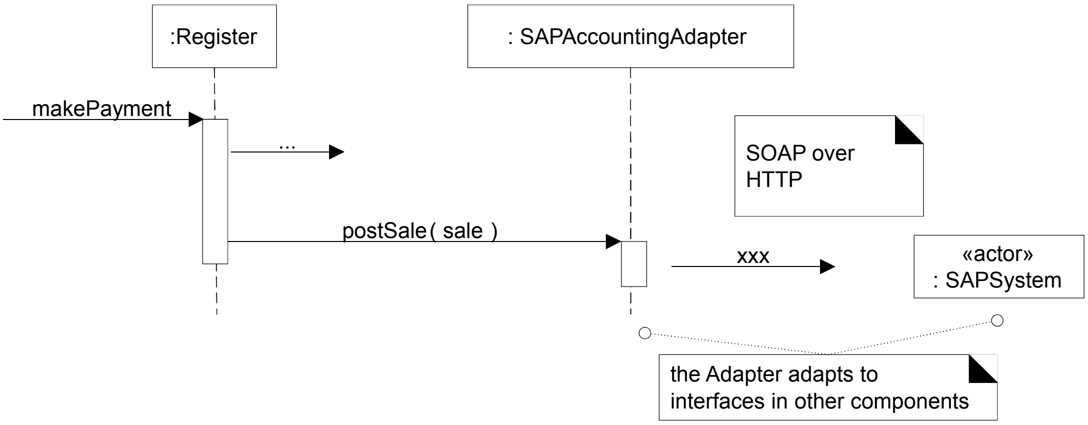
SAPAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

GreatNorthernAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

Adapter Pattern (Cont'd)



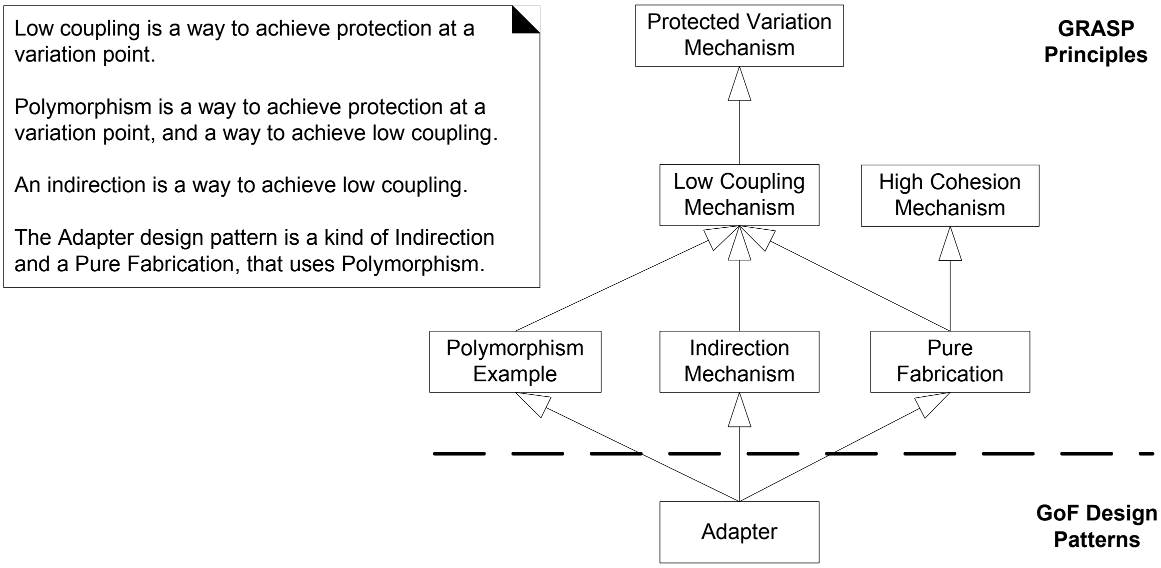
Adapter Pattern vs. GRASP Pattern

Adapter supports *Protected Variations* with respect to changing external interfaces or third-party packages through the use of an *Indirection* that applies interfaces and *Polymorphism*

The published patterns (e.g., Pattern Almanac) lists around +1000 patterns
➔ Pattern Overload!

A Solution: See the underlying principles

Adapter Pattern vs. GRASP Pattern (Cont'd)



Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.

Protected Variation Mechanism

GRASP Principles

Low Coupling Mechanism

High Cohesion Mechanism

Polymorphism Example

Indirection Mechanism

Pure Fabrication

Adapter

GoF Design Patterns

Note: Factory is not GoF pattern

Factory Pattern

Problem

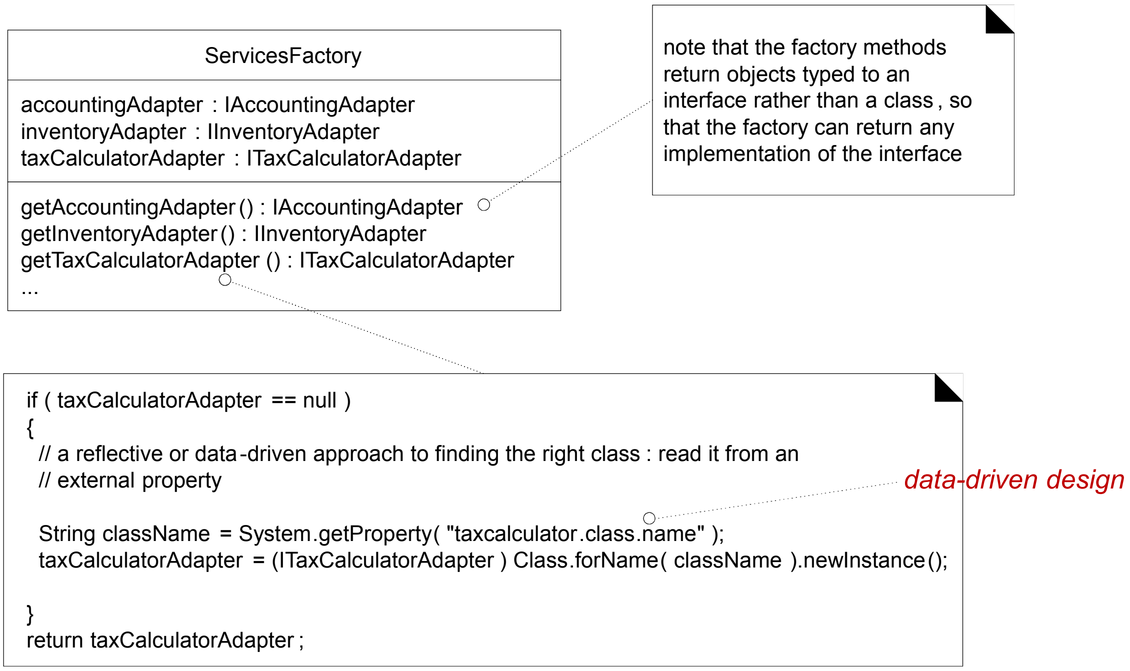
Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?

Solution

Create a **Pure Fabrication** object called a **Factory** that handles the creation

The adapter raises a new problem in the design: In the prior Adapter pattern solution for external services with varying interfaces, who creates the adapters? And how to determine which class of adapter to create, such as *TaxMaster-Adapter or GoodAsGoldTaxProAdapter*?

Factory Pattern (Cont'd)



ServicesFactory

accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter

getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
if ( taxCalculatorAdapter == null )
{
  // a reflective or data-driven approach to finding the right class : read it from an
  // external property

  String className = System.getProperty( "taxcalculator.class.name" );
  taxCalculatorAdapter = (ITaxCalculatorAdapter ) Class.forName( className ).newInstance();

}
return taxCalculatorAdapter;
```

data-driven design

Advantages of Factory Pattern

Separate the responsibility of complex creation into cohesive helper objects

Hide potentially complex creation logic

Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling

Singleton Pattern

Problem

Exactly one instance of a class is allowed. it is a "singleton." Objects need a global and single point of access

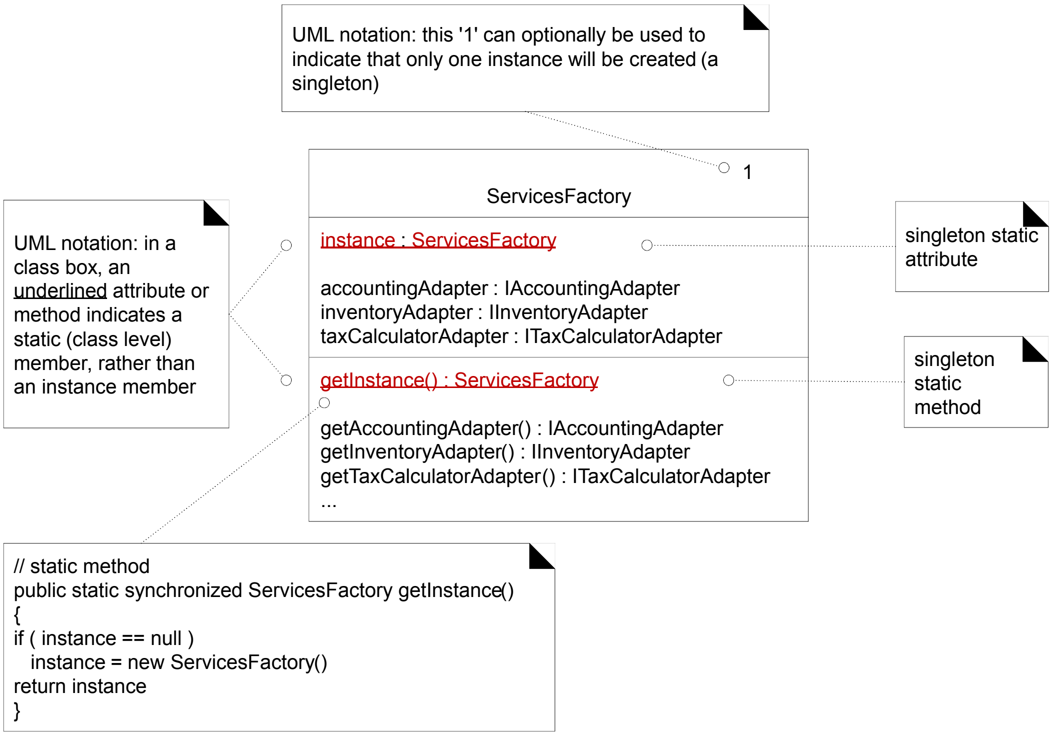
Solution

Define a static method of the class that returns the singleton

The *ServicesFactory* raises another new problem in the design: Who creates the factory itself, and how is it accessed?

First, observe that only one instance of the factory is needed within the process. Second, quick reflection suggests that the methods of this factory may need to be called from various places in the code, as different places need access to the adapters for calling on the external services. Thus, there is a visibility problem: How to get visibility to this single *ServicesFactory* instance?

Singleton Pattern (Cont'd)



UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member

singleton static attribute

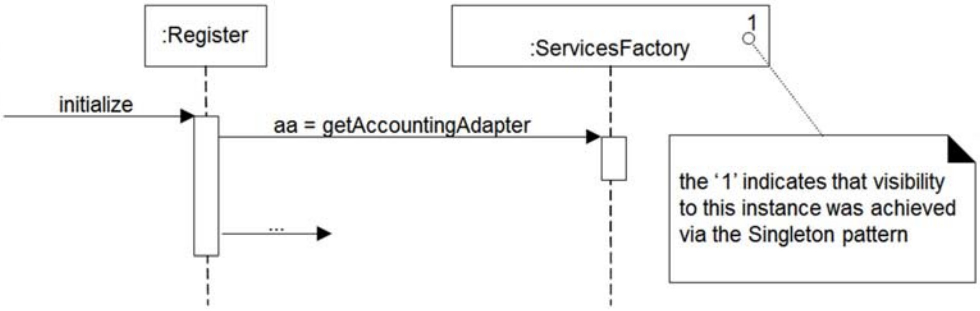
singleton static method

```
// static method
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
   instance = new ServicesFactory()
return instance
}
```

1
ServicesFactory
instance : ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getInstance() : ServicesFactory
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

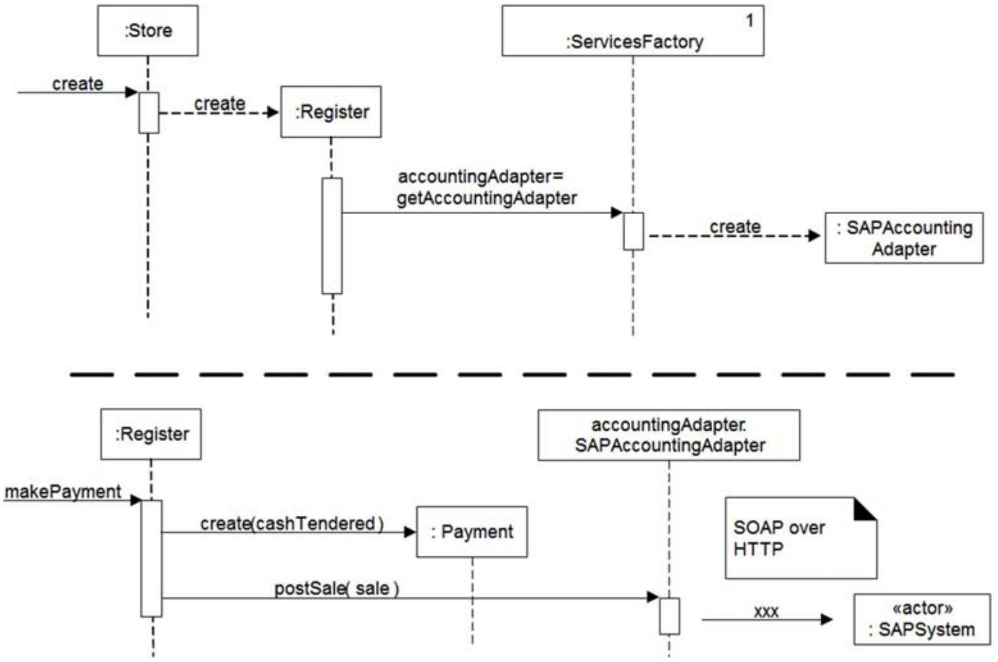
Singleton Pattern (Cont'd)

```
public class Register
{

public void initialize()
{
   … do some work …

   // accessing the singleton Factory via the getInstance call
   accountingAdapter =
      ServicesFactory.getInstance().getAccountingAdapter();

   … do some work …
}

// other methods…

} // end of class
```



the '1' indicates that visibility to this instance was achieved via the Singleton pattern

A combination of Adapter, Factory, and Singleton patterns have been used to provide Protected Variations from the varying interfaces of external tax calculators, accounting systems, and so forth



SOAP over HTTP

Strategy Pattern

Problem

How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?

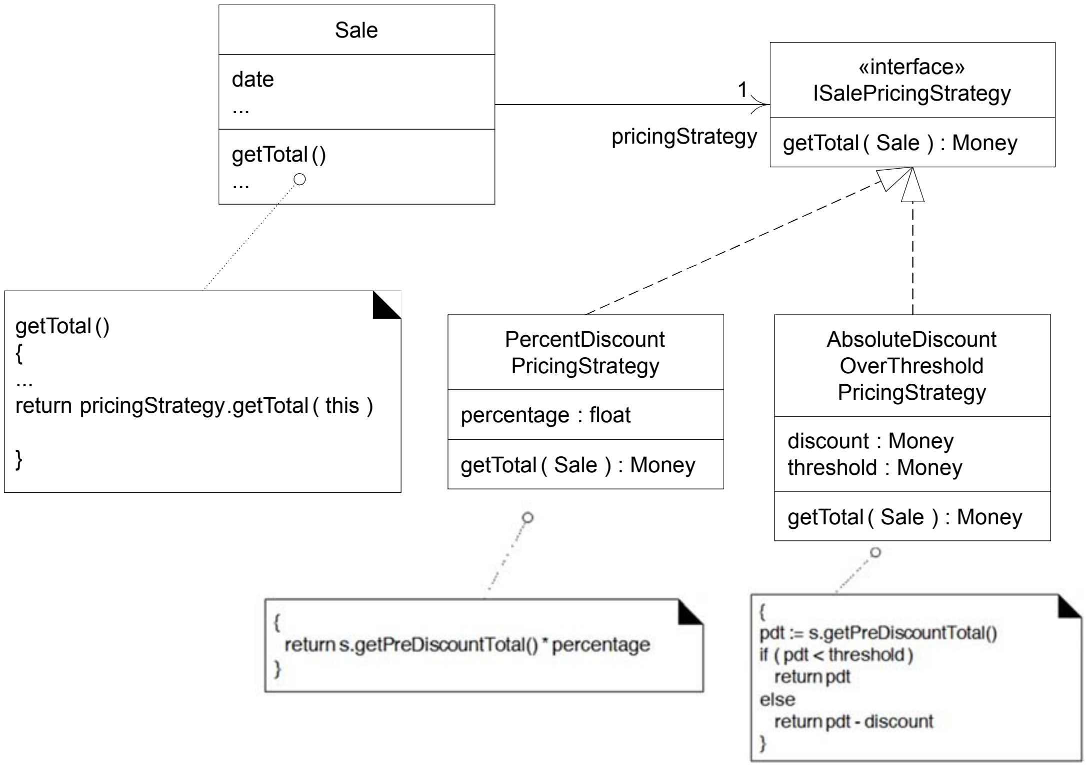
Solution

Define each algorithm/policy/strategy in a separate class, with a common interface

The next design problem to be resolved is to provide more complex pricing logic, such as a storewide discount for the day, senior citizen discounts, and so forth

The pricing strategy (which may also be called a rule, policy, or algorithm) for a sale can vary. During one period it may be 10% off all sales, later it may be $10 off if the sale total is greater than $200, and myriad other variations. How do we design for these varying pricing algorithms?

Strategy Pattern (Cont'd)

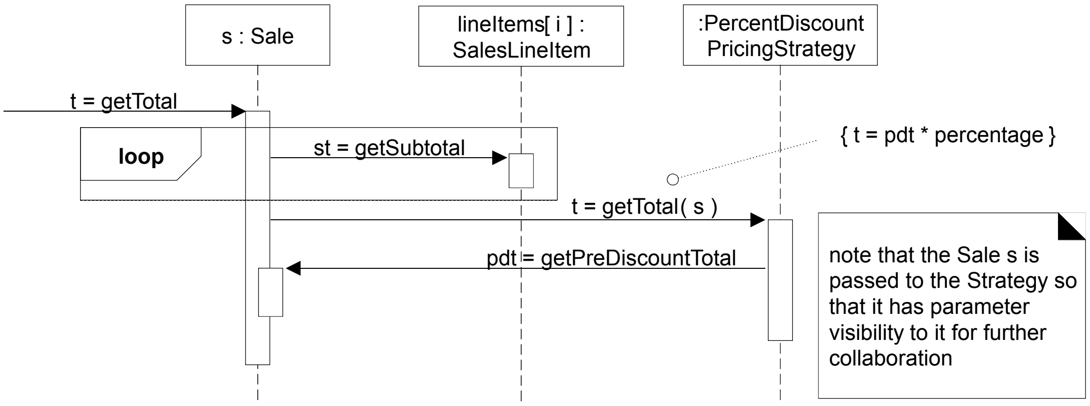

Sale
date
...
getTotal ()
...
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal (Sale) : Money
PercentDiscount PricingStrategy
percentage : float
getTotal (Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal (Sale) : Money

```
getTotal ()
{
...
return pricingStrategy.getTotal ( this )

}
```

```
{
  return s.getPreDiscountTotal() * percentage
}
```

```
{
pdt := s.getPreDiscountTotal()
if ( pdt < threshold )
   return pdt
else
   return pdt - discount
}
```

Strategy Pattern (Cont'd)


s : Sale
lineItems[i] : SalesLineItem
:PercentDiscount PricingStrategy
t = getTotal
loop
st = getSubtotal
t = getTotal(s)
pdt = getPreDiscountTotal
{ t = pdt * percentage }
note that the Sale s is passed to the Strategy so that it has parameter visibility to it for further collaboration

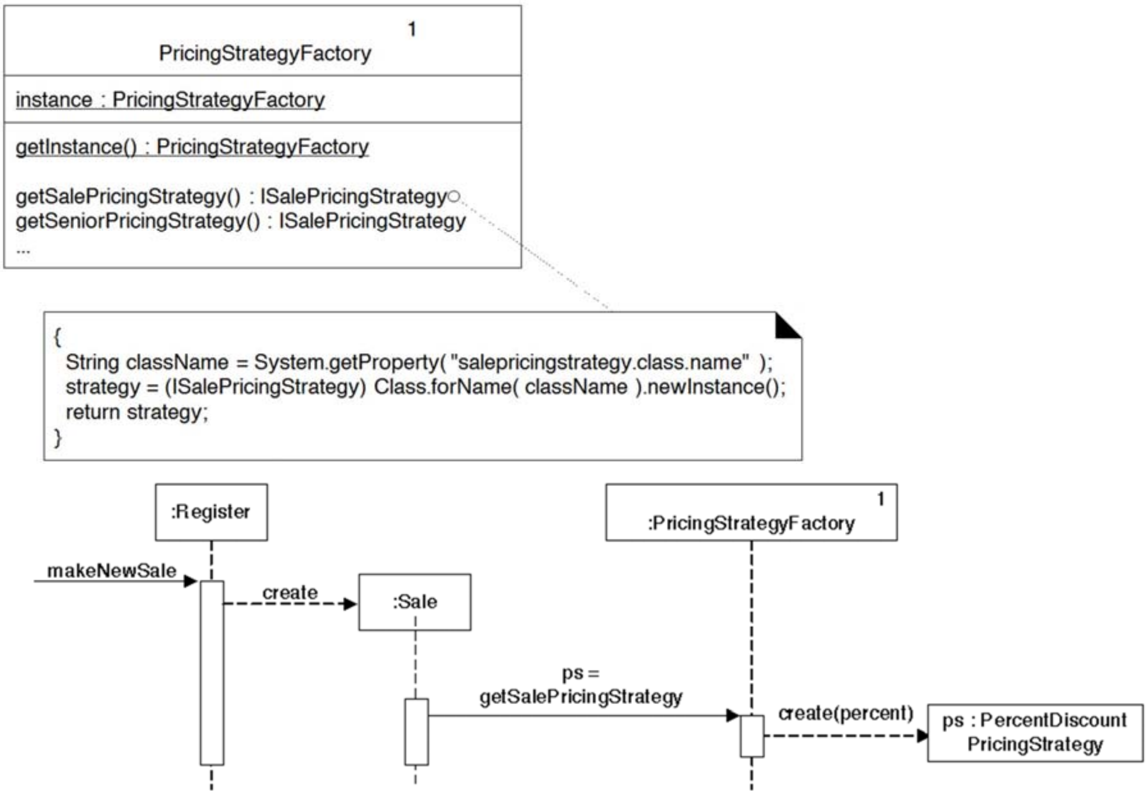
Creating a Strategy with a Factory

There are different pricing algorithms or strategies, and they change over time. Who should create the strategy?

Why separate Factory other than *ServicesFactory*?

Note that because of the frequently changing pricing policy (it could be every hour), it is *not* desirable to cache the created strategy instance in a field of the *PricingStrategyFactory*, but rather to re-create one each time, by reading the external property for its class name, and then instantiating the strategy.

Creating a Strategy with a Factory (Cont'd)


1
PricingStrategyFactory
instance : PricingStrategyFactory
getInstance() : PricingStrategyFactory
getSalePricingStrategy() : ISalePricingStrategy
getSeniorPricingStrategy() : ISalePricingStrategy
...
:Register
1
:PricingStrategyFactory
makeNewSale
create
:Sale
ps =
getSalePricingStrategy
create(percent)
ps : PercentDiscount
PricingStrategy

```
{
  String className = System.getProperty( "salepricingstrategy.class.name" );
  strategy = (ISalePricingStrategy) Class.forName( className ).newInstance();
  return strategy;
}
```

More on Pricing Strategies

How do we handle the case of multiple, conflicting pricing policies? For example, suppose a store has the following policies in effect today (Monday):

- 20% senior discount policy
- preferred customer discount of 15% off sales over $400
- on Monday, there is $50 off purchases over $500
- buy 1 case of Darjeeling tea, get 15% discount off of everything

Suppose a senior who is also a preferred customer buys 1 case of Darjeeling tea, and $600 of veggie burgers (clearly an enthusiastic vegetarian who loves chai). What pricing policy should be applied?

To clarify: There are now pricing strategies that attach to the sale by virtue of three factors:

1. time period (Monday)
2. customer type (senior)
3. a particular line item product (Darjeeling tea)

More on Pricing Strategies (Cont'd)

There can exist multiple co-existing strategies

Pricing strategy can be related to the type of customer

Design implications: The customer type must be known by the ***StrategyFactory*** at the time of creation of a pricing strategy for the customer.

Similarly, a pricing strategy can be related to the type of product being bought

Design implications: The ***ProductDescription*** must be known by the ***StrategyFactory*** at the time of creation of a pricing strategy influenced by the product.

Is there a way to change the design so that the ***Sale*** object does not know if it is dealing with one or many pricing strategies, and also offer a design for the conflict resolution?

Composite Pattern

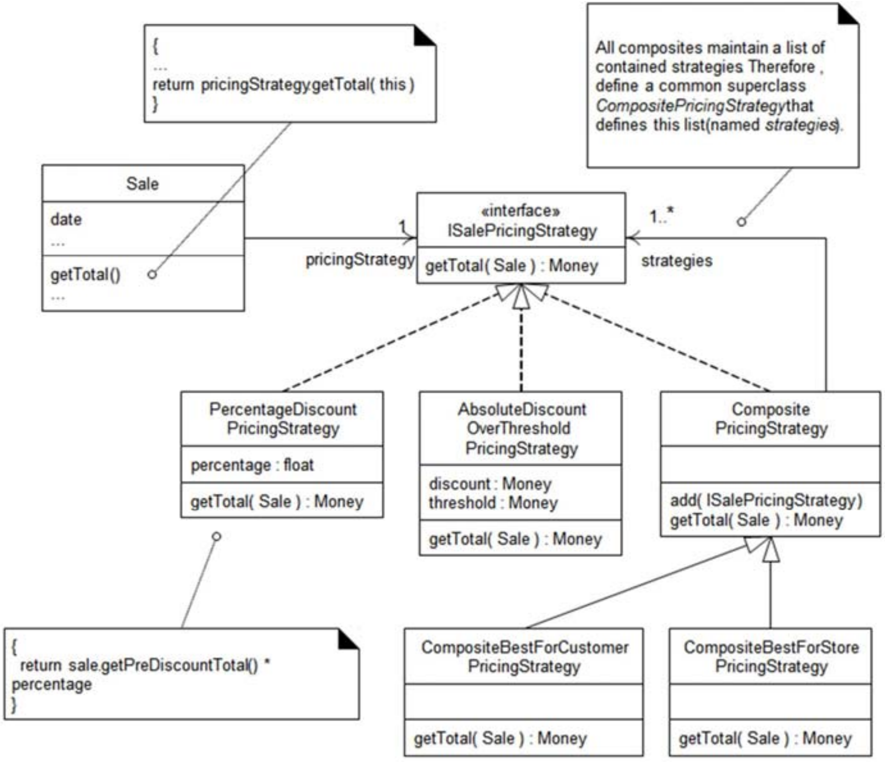
Problem

How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?

Solution

Define classes for composite and atomic objects so that they implement the same interface

Composite Pattern (Cont'd)

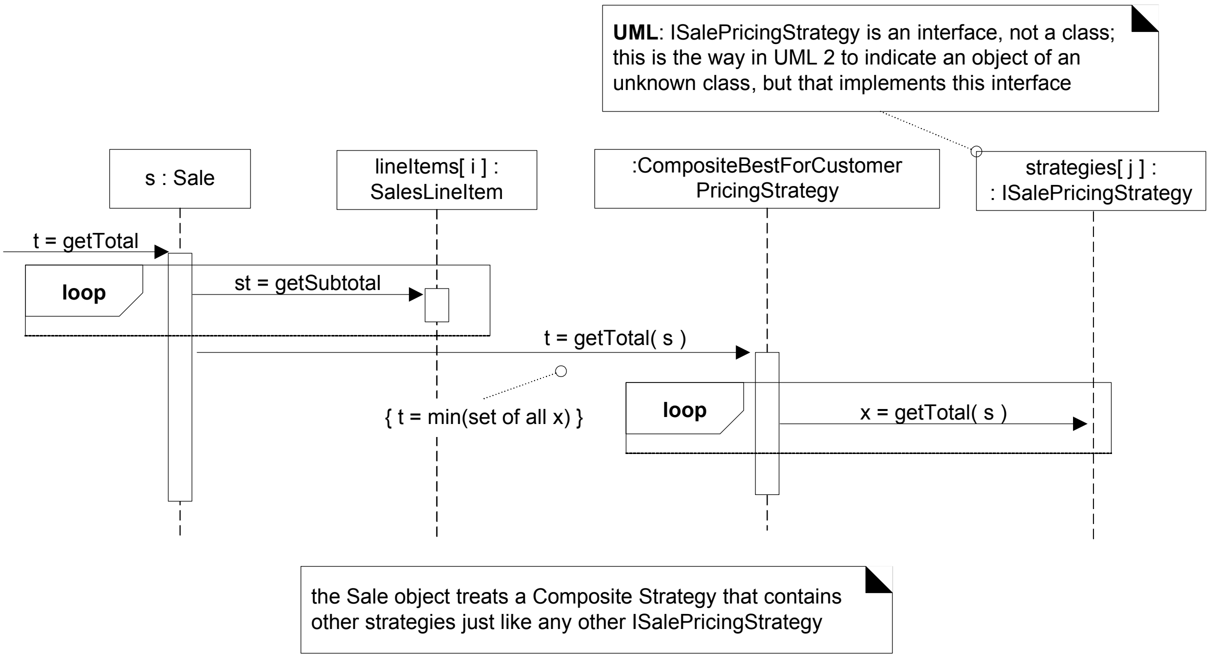


{
...
return pricingStrategy.getTotal(this)
}

All composites maintain a list of contained strategies Therefore, define a common superclass *CompositePricingStrategy*that defines this list(named *strategies*).

{
return sale.getPreDiscountTotal() *
percentage
}

Composite Pattern (Cont'd)



UML: ISalePricingStrategy is an interface, not a class; this is the way in UML 2 to indicate an object of an unknown class, but that implements this interface

s : Sale
lineItems[i] : SalesLineItem
:CompositeBestForCustomer PricingStrategy
strategies[j] : : ISalePricingStrategy
t = getTotal
loop
st = getSubtotal
t = getTotal(s)
{ t = min(set of all x) }
loop
x = getTotal(s)

the Sale object treats a Composite Strategy that contains other strategies just like any other ISalePricingStrategy

When do we create SalePricingStratigies?

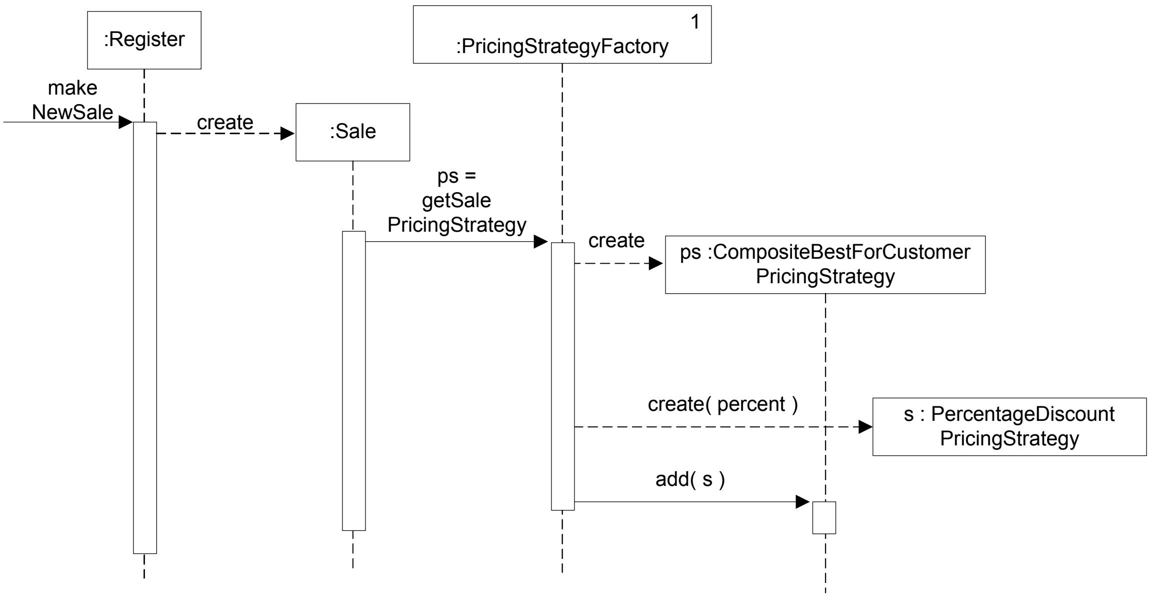
There are three points in the scenario where pricing strategies may be added to the composite:

1. Current store-defined discount, added when the sale is created

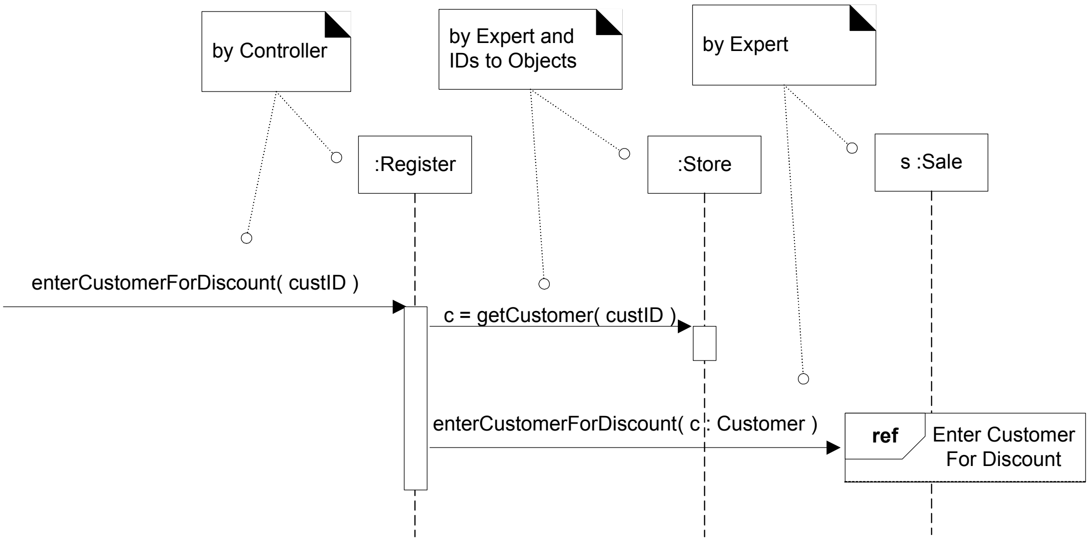
2. Customer type discount, added when the customer type is communicated to the POS

3. Product type discount (if bought Darjeeling tea, 15% off the overall sale), added when the product is entered to the sale

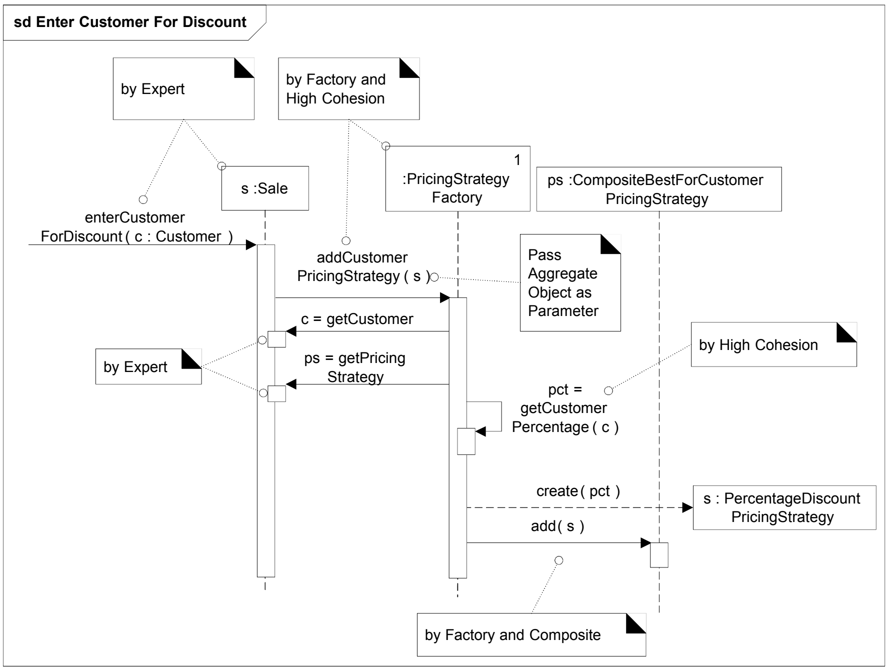
1. Current store-defined discount, added when the sale is created


:Register
1
:PricingStrategyFactory
make NewSale
create
:Sale
ps = getSale PricingStrategy
create
ps :CompositeBestForCustomer PricingStrategy
create(percent)
s : PercentageDiscount PricingStrategy
add(s)

2. Customer type discount, added when the customer type is communicated to the POS


by Controller
by Expert and IDs to Objects
by Expert
:Register
:Store
s :Sale
enterCustomerForDiscount(custID)
c = getCustomer(custID)
enterCustomerForDiscount(c : Customer)
ref
Enter Customer For Discount

2. Customer type discount, added when the customer type is communicated to the POS (Cont'd)


sd Enter Customer For Discount
by Expert
by Factory and High Cohesion
s :Sale
1
:PricingStrategy Factory
ps :CompositeBestForCustomer PricingStrategy
enterCustomer ForDiscount (c : Customer)
addCustomer PricingStrategy (s)
Pass Aggregate Object as Parameter
c = getCustomer
by Expert
ps = getPricing Strategy
by High Cohesion
pct = getCustomer Percentage (c)
create (pct)
s : PercentageDiscount PricingStrategy
add (s)
by Factory and Composite

Façade Pattern

Problem

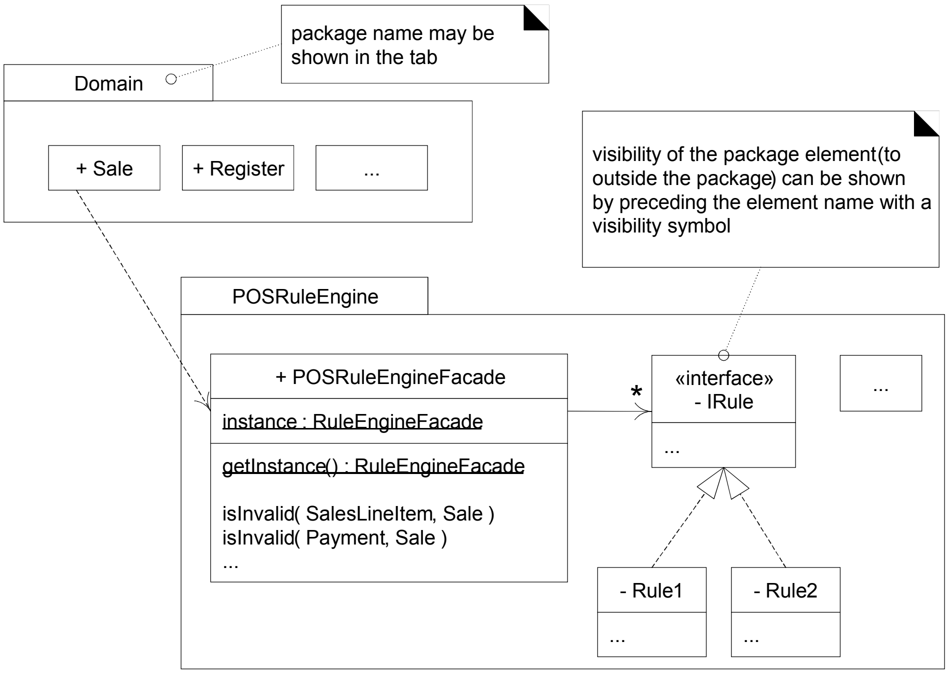
A common, unified interface to a disparate set of implementations or interfaces such as within a subsystem is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do?

Solution

Define a single point of contact to the subsystem, a facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components

We will define a “pluggable rule engine" subsystem, whose specific implementation is not yet known. It will be responsible for evaluating a set of rules against an operation. The architect also wants to design for a separation of concerns, and factor out this rule handling into a separate subsystem

Façade Pattern (Cont’d)


package name may be shown in the tab
Domain
+ Sale
+ Register
...
visibility of the package element(to outside the package) can be shown by preceding the element name with a visibility symbol
POSRuleEngine
+ POSRuleEngineFacade
instance : RuleEngineFacade
getInstance() : RuleEngineFacade
isInvalid(SalesLineItem, Sale)
isInvalid(Payment, Sale)
...
*
«interface»
- IRule
...
...
- Rule1
...
- Rule2
...

Observer/Publish-Subscribe Pattern

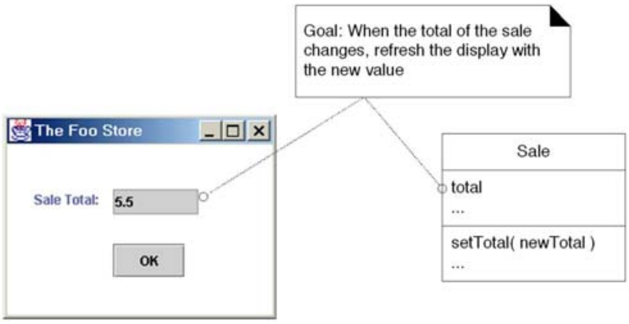
Problem

Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?

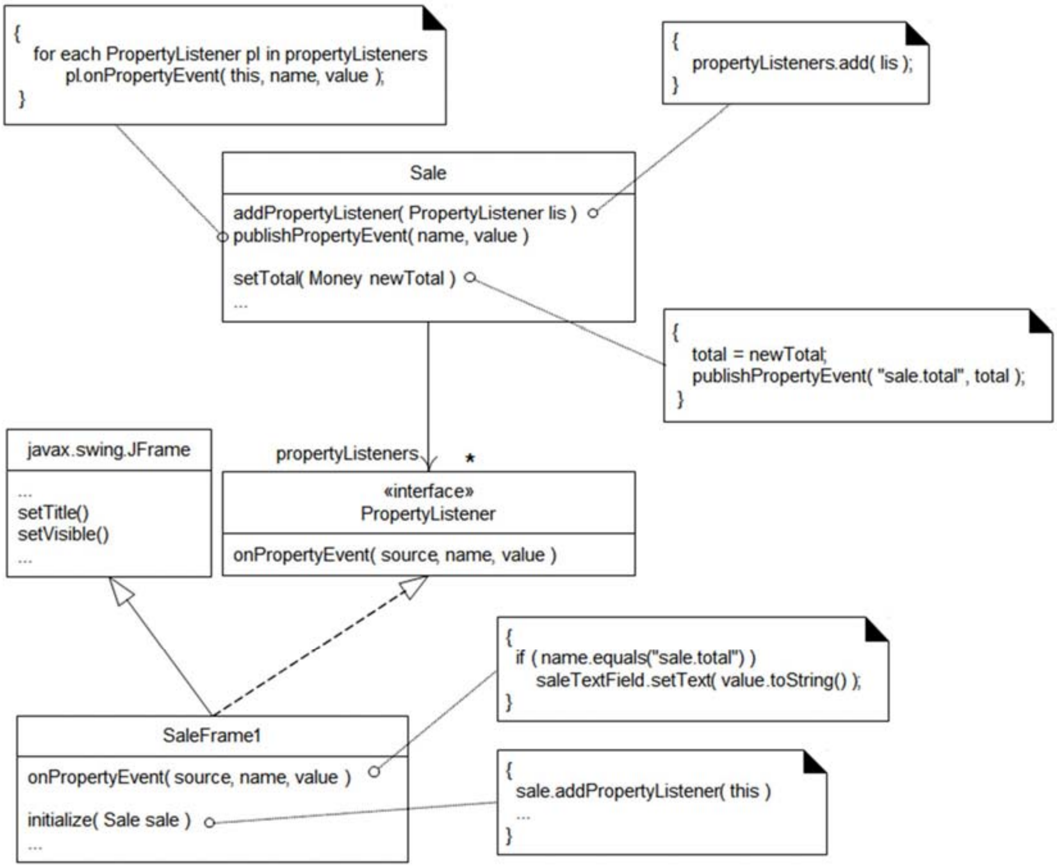
Solution

Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

Another requirement for the iteration is adding the ability for a GUI window to refresh its display of the sale total when the total changes



Goal: When the total of the sale changes, refresh the display with the new value



```
{
  for each PropertyListener pl in propertyListeners
      pl.onPropertyEvent( this, name, value );
}
```

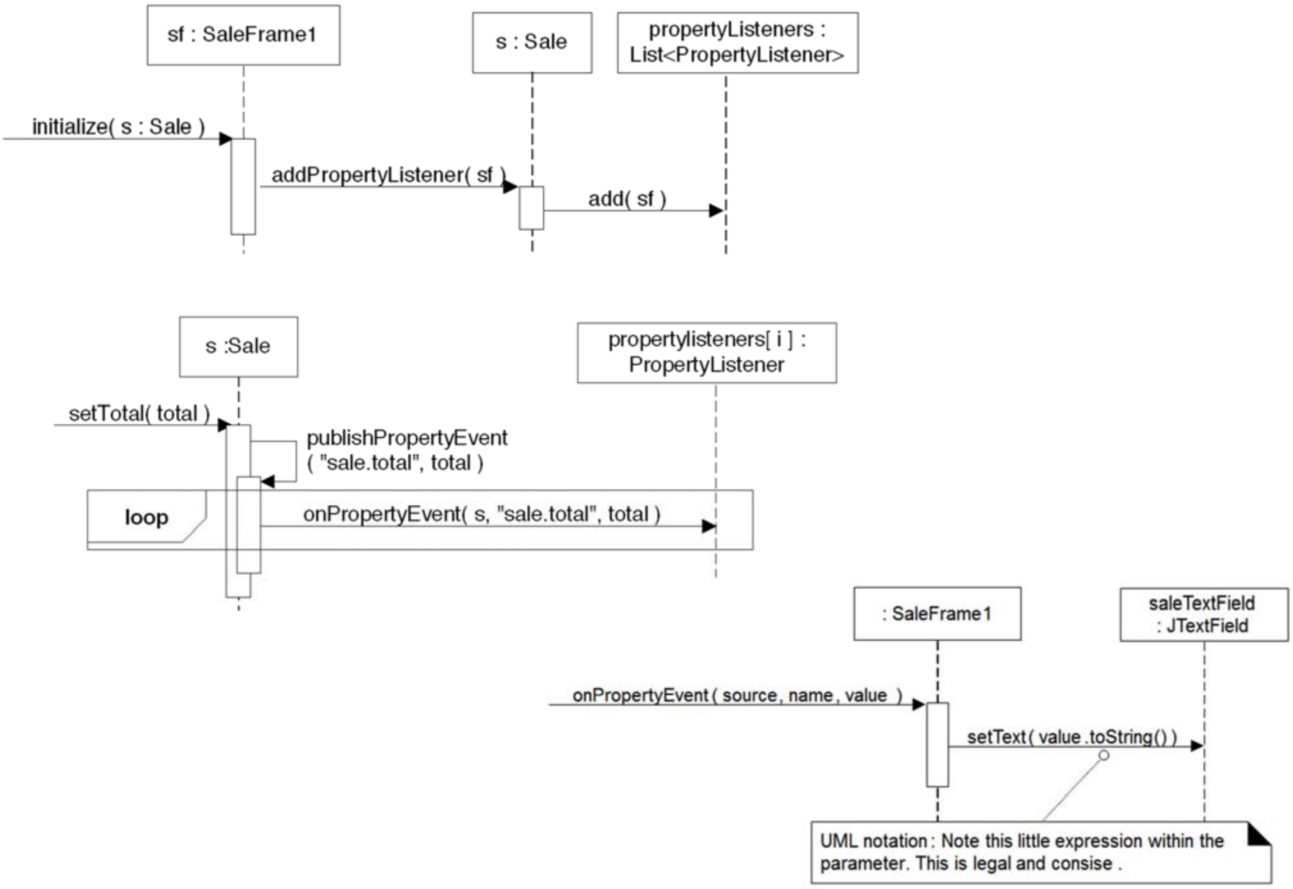
```
{
  propertyListeners.add( lis );
}
```

```
{
  total = newTotal;
  publishPropertyEvent( "sale.total", total );
}
```

```
{
 if ( name.equals("sale.total") )
    saleTextField.setText( value.toString() );
}
```

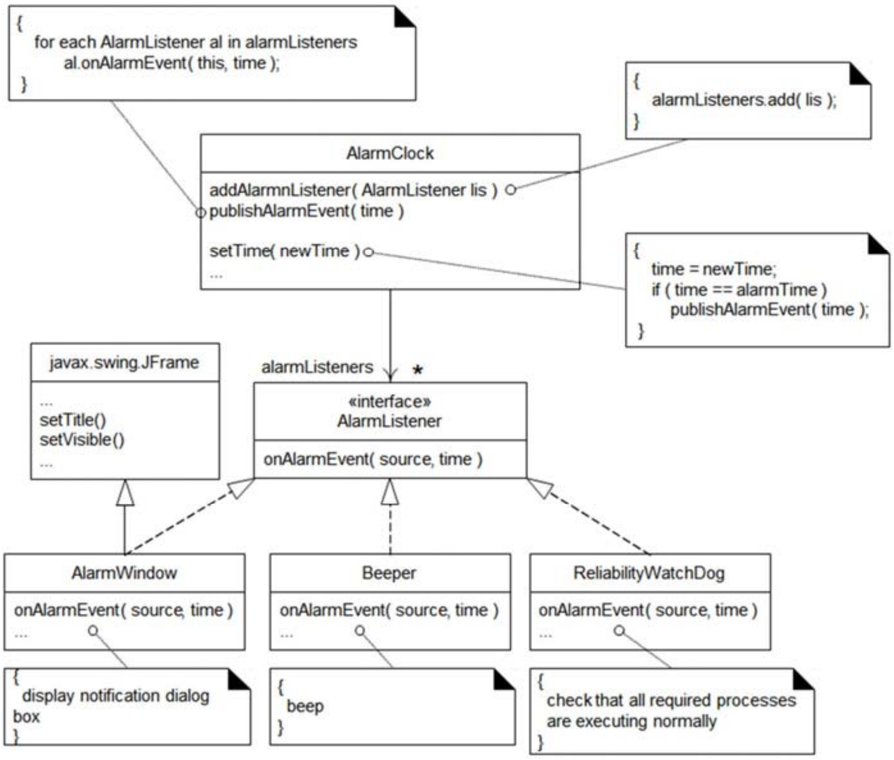
```
{
 sale.addPropertyListener( this )
 ...
}
```

Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
propertyListeners
*
javax.swing.JFrame
...
setTitle()
setVisible()
...
«interface»
PropertyListener
onPropertyEvent(source, name, value)
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...


sf : SaleFrame1
s : Sale
propertyListeners : List<PropertyListener>
initialize(s : Sale)
addPropertyListener(sf)
add(sf)
s :Sale
propertylisteners[i] : PropertyListener
setTotal(total)
publishPropertyEvent ("sale.total", total)
loop
onPropertyEvent(s, "sale.total", total)
: SaleFrame1
saleTextField : JTextField
onPropertyEvent (source, name, value)
setText(value .toString())

UML notation : Note this little expression within the parameter. This is legal and consise .

Other Usage of Observer Pattern


AlarmClock
addAlarmnListener(AlarmListener lis)
publishAlarmEvent(time)
setTime(newTime)
...
javax.swing.JFrame
...
setTitle()
setVisible()
...
alarmListeners
*
«interface»
AlarmListener
onAlarmEvent(source, time)
AlarmWindow
onAlarmEvent(source, time)
...
Beeper
onAlarmEvent(source, time)
...
ReliabilityWatchDog
onAlarmEvent(source, time)
...

```
{
  for each AlarmListener al in alarmListeners
        al.onAlarmEvent( this, time );
}
```

```
{
  alarmListeners.add( lis );
}
```

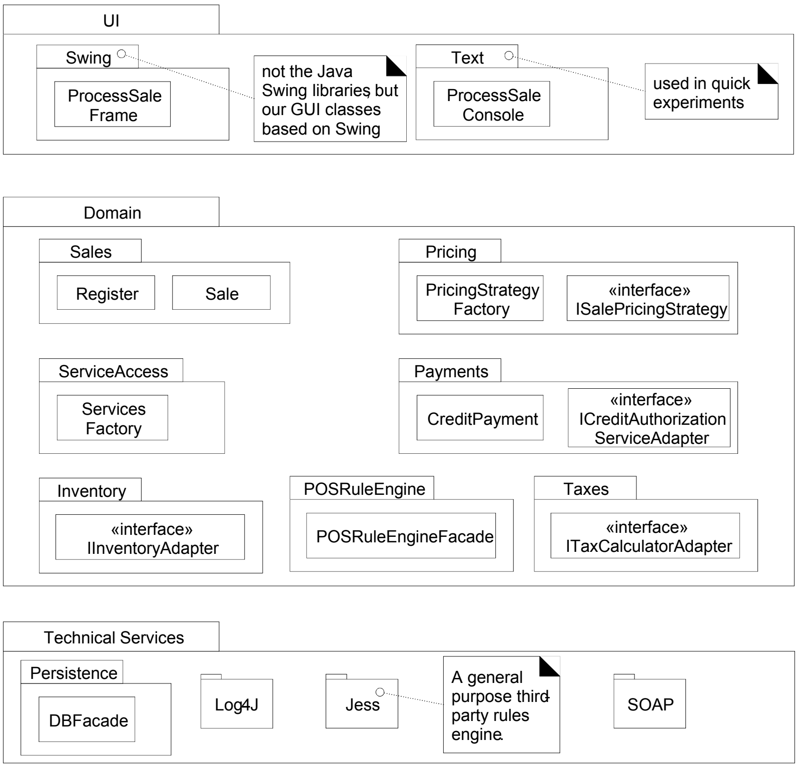
```
{
  time = newTime;
  if ( time == alarmTime )
       publishAlarmEvent( time );
}
```

```
{
  display notification dialog
box
}
```

```
{
  beep
}
```

```
{
  check that all required processes
  are executing normally
}
```

NextGen POS System Architecture



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

Advanced Principles of Object-Oriented Design

SRP: The Single Responsibility Principle

OCP: The Open Closed Principle

LSP: The Liskov Substitution Principle

ISP: The Interface Segregation Principle

DIP: The Dependency Inversion Principle