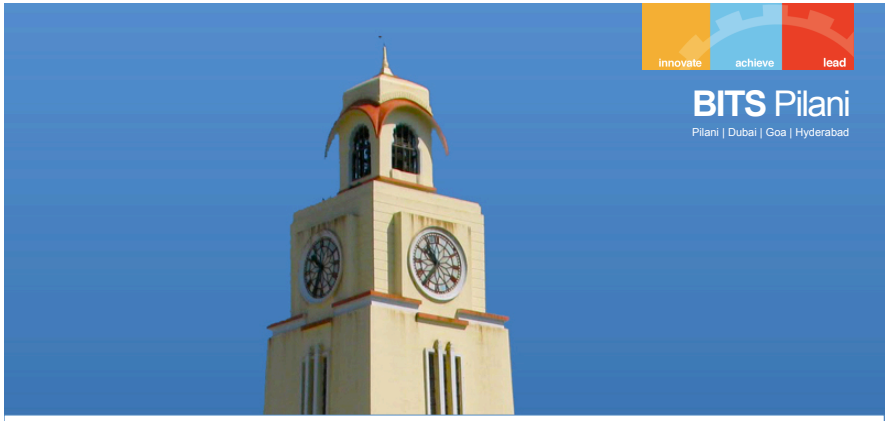
SS ZG514 Object Oriented Analysis and Design

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Harvinder S Jabbal
Session 01: Introduction



innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Modules

Modules

innovate achieve lead

No	Title of the Module
M1	Introduction SDLC Models - Waterfall, Unified Process, Agile Introduction to Object Oriented Analysis & Design
M2	Starting with Object Oriented Analysis : Building Use Case Model
M3	Creating System Level Artefacts : Domain Model, SSD & Operation Contracts
M4	Getting into Object Oriented Design : Refinements in Use Cases & Domain Model, Interaction Diagrams, State Transition Diagram, Activity Diagram
M5	Visibility between Objects, Class Diagram, Package Diagram
M6	Design Patterns : GRASP, Additional Patterns, SOLID Design Principles
M7	Design Patterns : Some Gang Of Four (GoF) Patterns
M8	Design Patterns : Further Gang Of Four (GoF) Patterns

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Software Development Life Cycle

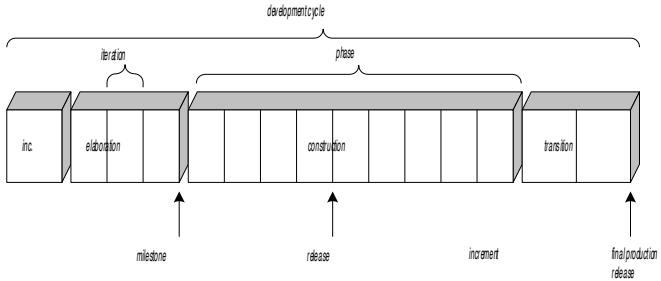
SDLC Models

- Waterfall,
- Unified Process,
- Agile

This course emphasizes the Unified Process.

The Agile Manifesto and its implementation is stressed.

UNIFIED PROCESS phases



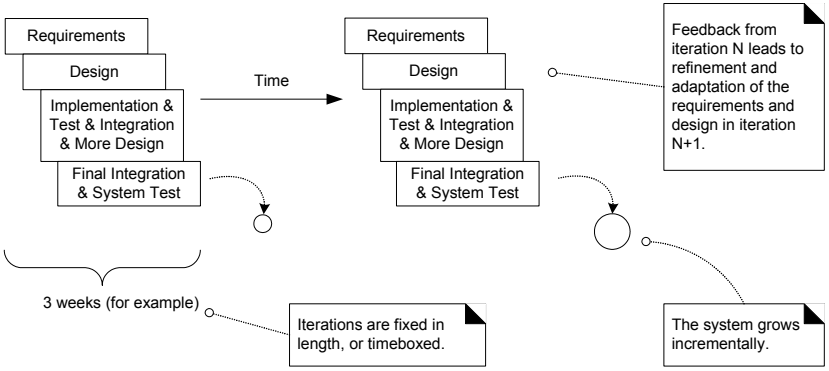
An iteration end-point when some significant decision or evaluation occurs

A stable executable subset of the final product. The end of each iteration is a minor release.

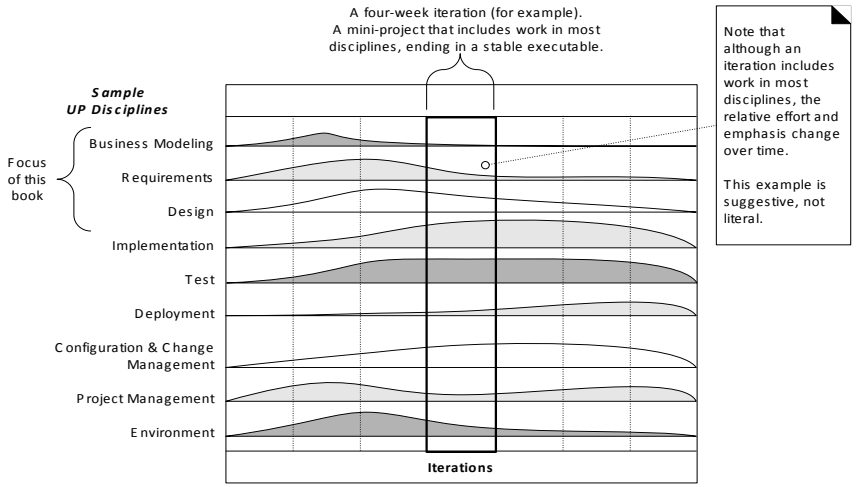
The difference (delta) between the releases of 2 subsequent iterations.

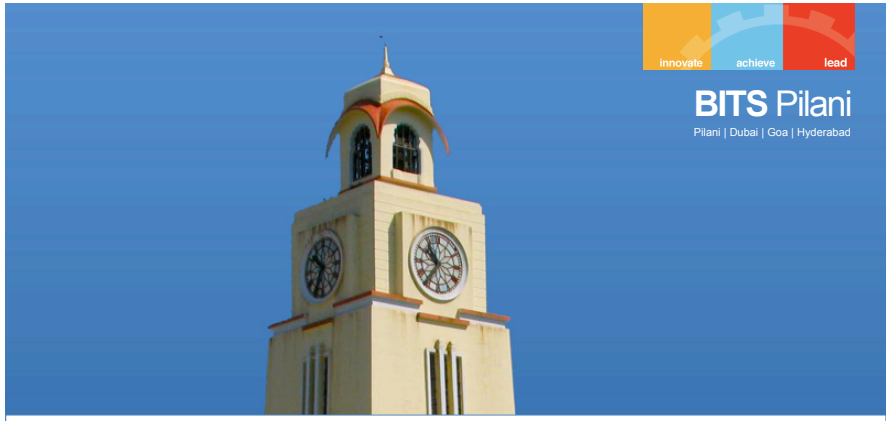
At this point, the system is release for production use

Iterative Process



UNIFIED PROCESS disciplines





Unified Modeling Language

What is UML

"The Unified Modeling Language is a family of graphical notations, backed by a single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented style."

UML

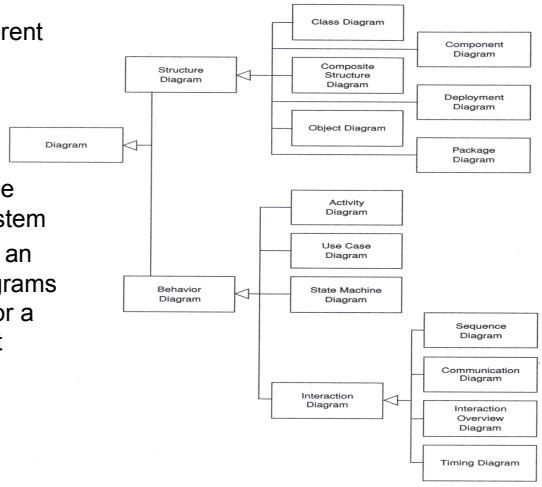
UML is standardized. Its content is controlled by the Object Management Group (OMG), a consortium of companies.

Unified

- UML combined the best from object-oriented software modeling methodologies that were in existence during the early 1990's.
- Grady Booch, James Rumbaugh, and Ivor Jacobson are the primary contributors to UML.

UML Diagrams

- UML 2 supports 13 different types of diagrams
- Each diagram may be expressed with varying degrees of detail
- Not all diagrams need be used to model a SW system
- The UML does not offer an opinion as to which diagrams would be most helpful for a particular type of project



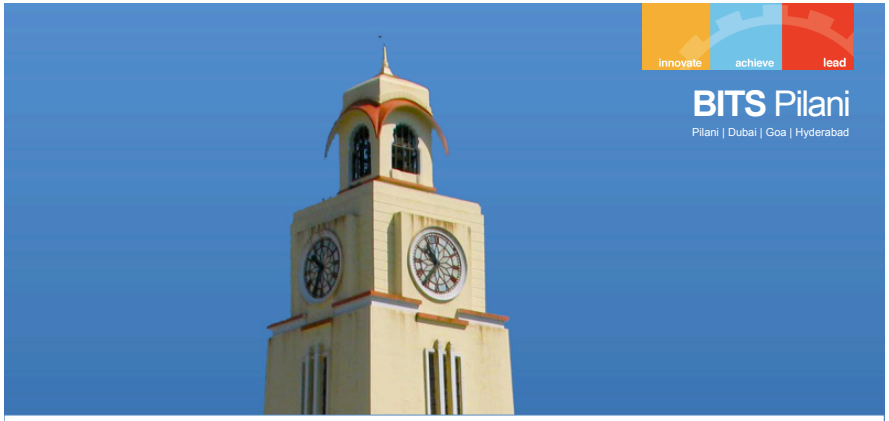
A Conceptual Model of the UML

A conceptual model needs to be formed by an individual to understand UML.

UML contains three types of building blocks:

- things,
- relationships,
- diagrams.

- Things
  - Structural things
    - Classes, interfaces, collaborations, use cases, components, and nodes.
  - Behavioral things
    - Messages and states
  - Grouping things
    - Packages
  - Annotational things
    - Notes
- Relationships: dependency, association, generalization and realization.
- Diagrams: class, object, use case, sequence, collaboration, state-chart, activity, component and deployment.



Design Patterns or Principles

OOAD Patterns or Principles

GRASP

1. Information Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion

Polymorphism

Pure Fabrication

Indirection

Protected Variation

- Single responsibility
- Open/Close
- Liskov Substitution
- Interface Segregation
- Dependency Intersion

Patterns

Creational	Structural	Behavorial
Class • Factory Method Object • Abstract Factory Builder Prototype Singleton	Class • Adapter Object • Adapter • Bridge • Composite • Decorator • Facade • Proxy • Flyweight	Class • Interpreter • Template Method Object • Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

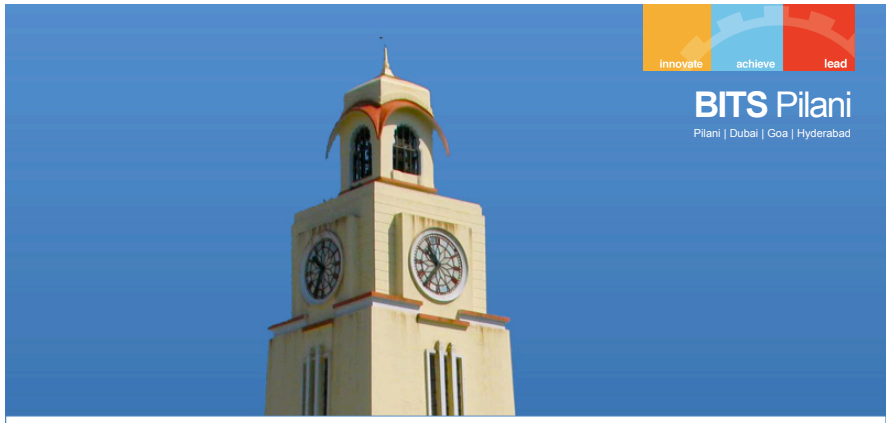
SS ZG514 Object Oriented Analysis and Design

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Harvinder S Jabbal
Session 02: Object Oriented Programming

Outline

- Programming Concepts
- Programming Paradigms
- OOP Paradigm
- Time Line
- Thinking Objects
- Classes and Objects
- Encapsulation
- Extensibility and Reusability



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Programming Concepts

Programming Concepts

- High-level languages
  - FORTRAN, COBOL, PASCAL
  - English like languages where-in the written code is converted to machine code by a compiler so that the language is easy for the programmer to code while it hides the run time details like implementation detail and memory addressing from the user.
- Low-level languages
  - Assembly Language
  - These languages leave memory addressing and register handling details to the programmer. While these languages give complete control to the programmer, they require expert knowledge of computer hardware to code.
- Middle-level languages
  - C Language.
  - These languages are english like and easy to code.
  - Command have been provided for direct handling of computer hardware units.
  - Can be used both by Application Programmer as well as Systems Programmer.

Programming Paradigms…

1. Structured Programming
   1. Primacy to Code. Data is an outside entity used by various procedures.
   2. Modular Structure has been emphasized, with each module handling a distinct sub-task.
   3. Each module should be able to stand-alone.
   4. A Module should be such that it can be used repetitively, thus providing reusability of code.
   5. A Procedural, function oriented language such as C is thus a “Structured Language”.

…Programming Paradigms…

2. Features of C
   1. All code is placed in functions receiving values as arguments and returning them through return values or by changing the value of locations pointed by the arguments
   2. All values are passed by value or by reference as arguments to a function.
   3. Data Structures are available which allow heterogeneous data elements to be compiled into a single data unit.
   4. Memory can be accessed directly through a mechanism called pointers. Values of memory address are held in variables which are called pointers. Pointers are typed by the type of memory storage block to which they point. Pointers values, representing memory address values, can thus increment based on the number of units of memory used by the data being addressed.
   5. An extensive library of routines of Hardware Managements of IO has been provided.
   6. A rich library of general purpose routines to handle activity such as string manipulation and mathematical computations have been provided.

…Programming Paradigms…

3. What then is the problem.
   1. Need for reusability over time.
   2. Need for complete independent units of code for very large projects.
   3. Code that can survive change in user requirements.
   4. The need to bundle data with code and a shift to data primacy from code primacy.
   5. Need for component based production. A component should be usable as a black box, fully tested and fully functional against a specification.
   6. These building blocks should be extensible to enhancements in data structures and functionality.

…Programming Paradigms

4. Object-oriented Programming Paradigm
   1. Data hiding.
   2. Data abstraction
   3. Encapsulation
   4. Inheritance
   5. Polymorphism

OOP Paradigm

1. Data primacy over procedure.
2. Division of task between objects.
3. Each object is an independent memory unit of data and functionality (methods).
4. Method and data are bound together and instantiated in tandem with one another.
5. Access to data by external code is controlled and not normally accessible directly.
6. Objects communicate with one another and collaborate by passing messages.

Time Line….

- C
  - 1969-1973 : Bell Laboratories. The C programming language was devised as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today and the base syntax provider for popular Object Oriented Languages.
  - 1978, Brian Kernighan and Ritchie wrote the book () often referred to as “the write book” that became the language definition for several years
- C++
  - C++ was written by Bjarne Stroustrup at Bell Labs during 1983-1985. C++ is an extension of C. Prior to 1983, Bjarne Stroustrup added features to C and formed what he called "C with Classes". He had combined the Simula's use of classes and object-oriented features with the power and efficiency of C. The term C++ was first used in 1983.

…. Time Line….

- Java
  - **Java** is a programming language originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform.
  - The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities.
  - Java applications are typically compiledto bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.
  - Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible.
  - It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another.

…. Time Line

- C#
  - During the development of the .NET Framework, the class libraries were originally written using a managed code compiler system called Simple Managed C (SMC).
  - In January 1999, Anders Hejlsberg formed a team to build a new language at the time called Cool, which stood for "C-like Object Oriented Language".
  - Microsoft had considered keeping the name "Cool" as the final name of the language, but chose not to do so for trademark reasons.
  - By the time the .NET project was publicly announced at the July 2000 Professional Developers Conference, the language had been renamed C#, and the class libraries and ASP.NET runtime had been ported to C#.

Thinking Objects

- Object Based Programming
  - Data encapsulation
  - Data hiding
  - Operator Overloading
  - Initialisation and automatic clearing of Objects after use
- Object Oriented Programming
  - Inheritance
  - Dynamic binding
  - Extensive and well-defined standard template library

Classes and Objects

Class

- An array or Collection of Objects
- Class defines the hidden characteristics of an Object.
- Class defines Member Functions (Methods) and Member Data (Attributes) provided by the Object when it has been instantiated.

Object

- There are several instances of an Object belonging to the same Class.
- Each Object has its own data.
- The value of data in each Object is independent of other objects of the same class.
- The data elements of an object are referred to as attributes or state variables and a set of values at any point of time for a given object are referred to as the state of the object.

Encapsulation….

Data is stored in Stack and free space

Functions are stored in Code area.

- The is a need to keep a control on the access of data by functions.
- This is handled by Encapsulation, by binding data and calling functions into a single unit with security classification for each, so that no unauthorised access to data cannot take place.

1. Security Privileges
   - Security Access Specifies of Public, Private and Protected are provided to data and functions.

….Encapsulation

2. Data Hiding/Data Abstraction
   - Member Data is generally declared as private so that it can not be accessed directly but it may only be accessed through Public Functions (Methods).
   - Thus direct access to data is hidden but an abstraction of data is provided through a Public Method that supplies the data.
3. Function Overloading
   - The Object can mean different things to different users based on the nature of client who makes the call. A function to calculate area may use a different mathematical construct depending on the object that is to be measured.
   - A single function by name can make any of the computations. We say that the function has been overloaded so that it may perform various functions based on the nature of parameter.
4. Operator Overloading
   - Normal operators like +, -, / and * may be used with various object type differently. The '+' may concatenate strings, while it adds integers, and do a more complex operation for polar coordinate addition.

Extensibility and Reusability

These are achieved through:

- Containment
- Inheritance
- Virtual functions
- Run Time Polymorphism
- Abstract data type
- Standard Template Libraries (STL)

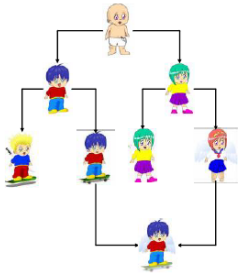
Extending normally implies deriving new classes based on existing classes.

Containment

- Containment
  - Composite or Aggregation of a Class
  - A “has a” type of relation can be built.
  - A object can be a member data for another object.
  - Example: A Student object may contain Date: birthDate, Date: joiningDate, String name

Inheritance

- Inheritance and Class Hierarchy
  - Create sub-class from a class. A derived class (sub-class) inherits from a base class (class).
  - Derived class gets the use of all member functions and data of base class but may add its own specialisation
- Single and Multiple Inheritance
  - Single Inheritance
  - Multi Level Inheritance
  - Hierarchical Inheritance
  - Multiple/Hybrid Inheritance



Multiple Inheritance

Virtual functions

- In inheritance relationship, the derived class gets access to protected member data through public access functions, thereby achieving reusability.
- The objects in the base class may also use member functions belonging to the sub-class based on the users choice at run-time.
- This is done by defining a virtual function in the base class and over-riding it in the derived class.
- In this way the base class function(method) runs a different function at run time depending on the type of object instantiated at run time.

Run Time Polymorphism

- A derived class can access all the data members and functions declared in the base class with protected access specifier.
- A derived class can override the functions in the base class.
- By using Virtual Functions a pointer to the base class can use a derived class function.
- The above features combine to achieve Run Time Polymorphism
- Polymorphism is also called Dynamic Data Binding.

Abstract data type

- A Class in OOP is called ADT.
- These classes are called ADT because they hide the implementation
- An abstract data type called Shape may have a virtual draw routine without any implementation.
- The implementation may be provided by derived classes for Circle, Square etc.
- The actual drawing will depend on the derived class.

Standard Template Libraries (STL)

- A programmer can write code using data structures and algorithm implementation provided by STL.
- C++ has a well developed standard library in the form of stream IO library that provides extensive functionality for input and output.
- Templates and STL are considered the most significant feature of C++ that provides OOP capability.
- If you are using pre-developed templates containing code provided by third party, you achieve reusability in a very significant way.

Summing it up

- We have understood a class helps us to encapsulate member functions (behaviours) and member (attributes) data into a single entity called an object.
- These classes have relations with other classes (association).
- We have seen there is also an inheritance relationship whereby new classes can be created from existing classes.
- Classes can communicate with one another by passing messages.

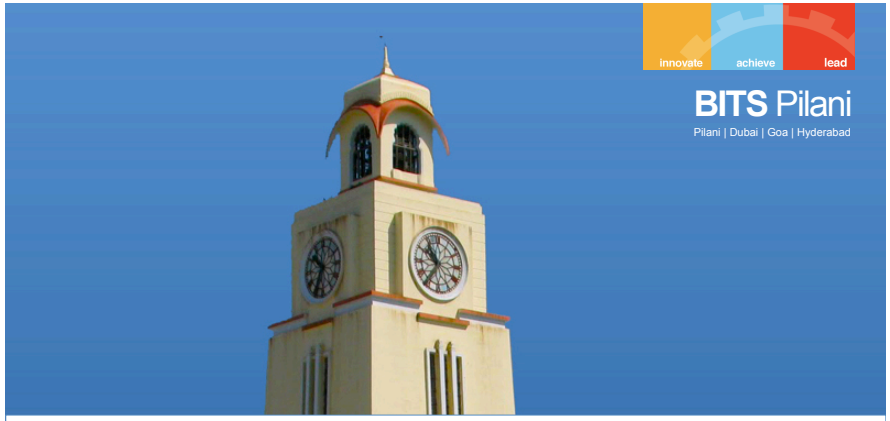
Thank you……..

Credits

- Vishal Gupta – Implementation Support
- Ishan Sukhla – Video Production
- Java, Object-Oriented C++ Programming by Hirday Narayan Yadav Laxmi Publications © 2008
- C++ and Java by Avinash C. Kak John Wiley & Sons © 2003
- Java: The Complete Reference, Eighth Edition by Herbert Schildt Oracle Press © 2011
- Michael Berman, Data Structures Via C++ Objects by Evolution, Oxford University Press,2007
- Ramesh Vasappanavara et al, Object-oriented Programminh Using C++ and First Impression, Pearson, 2011

Outline

No	Title
CS1.1.1	Explain with example difference between Procedural and Object Oriented Programming. Highlight the fact that, maintainability of software is easy in Object Oriented software.
CS1.1.2	SDLC Phases, Role of Analyst, Designer, Programmer and Tester
CS1.1.3	Waterfall Model : How it is good or bad?
CS1.1.4	Briefing of all other models like Spiral Model, Incremental Model, V-Model etc.
CS1.1.5	UP Model, Phases & Disciplines, How it is different than Waterfall and other models?



Procedural and Object Oriented Programming

Procedural and Object Oriented Programming

innovate achieve lead

"Think in Objects"

Analyze requirements with use cases

Create domain models

Apply an iterative & agile Unified Process (UP)

Relate analysis and design artifacts

Read & write high-frequency UML

Think Objects

innovate achieve lead

Practice

Apply agile modeling

Design object solutions
- Assign responsibilities to objects
- Design collaborations
- Design with patterns
- Design with architectural layers
- Understand OOP (e.g., Java) mapping issues

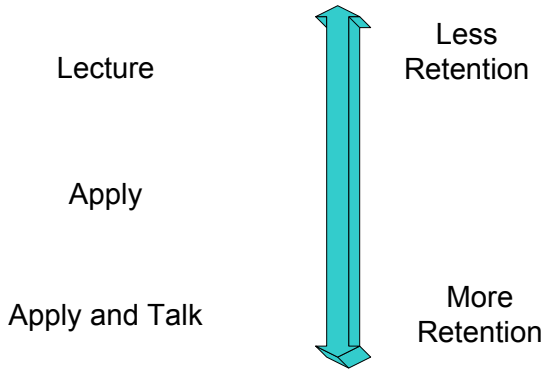
GRASP

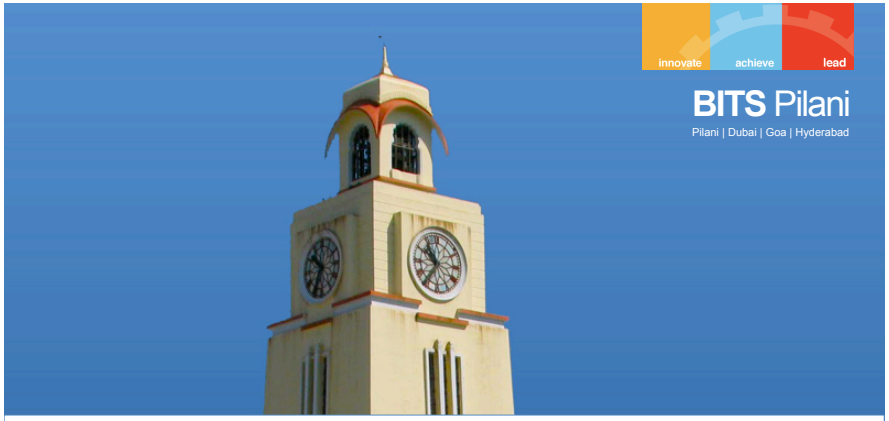
innovate achieve lead

1. Assign responsibilities to objects
   - The GRASP patterns are the key learning aid

After that. . . ?

Learning Groups





Software Development Life Cycle

SDLC

"The Unified Modeling Language is a family of graphical notations, backed by a single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented style."

Fitting a Process to a Project

Adapt a process to fit your environment

Remember that it's usually easier to start with too little and add things than it is to Start with too much and take things away .

Iterative development supports frequent process improvement

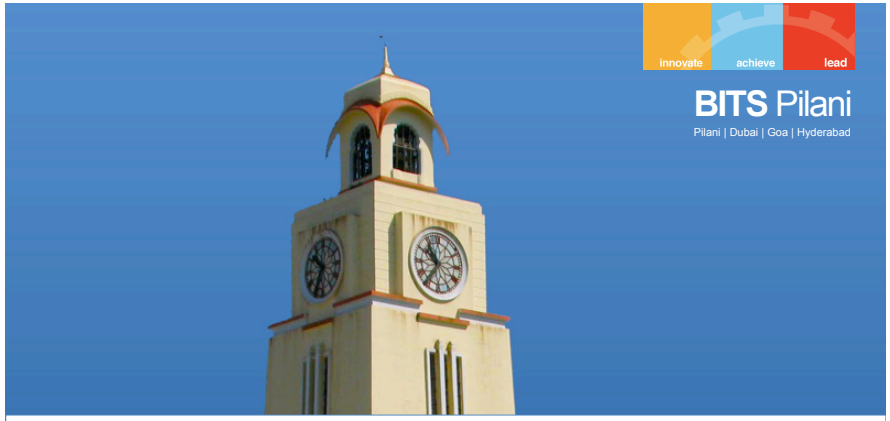
iteration retrospective: List with 3 categories

1 . Keep: things that worked well that you want to ensure you continue to do

2 . Problems : areas that aren't working well

3 . Try: changes to your process to improve it

http://www.retrospectives .com/



Legacy Model

Waterfall Model Vs Iterative

Chunks
- Waterfall by activity
- Iterative by subsets of functionalities

Production Quality

Multiple Releases

Incremental, spiral, evolutionary, jacuzzi.

Compromise: Staged Delivery

Time Boxing

Slipping a date or slipping functionality.

Predictive and Adaptive Planning

Waterfall endures for predictive nature
- 2 stages of predictive planning

Adaptive Planning
- Requirement churn is unavoidable.

Change is a constant

1. Don't make a predictive plan until you have precise and accurate requirements and are confident that they won't significantly change .
2. If you can't get precise, accurate, and stable requirements, use an adaptive planning style .

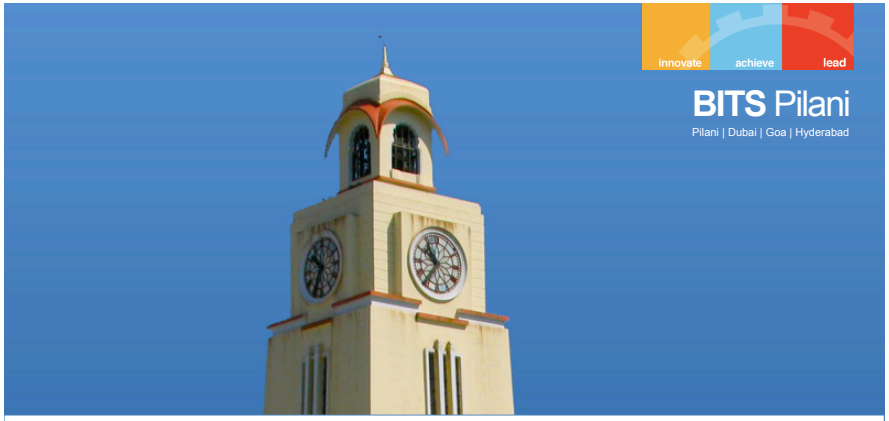
Fitting the UML into the process

- UML for documentation
- UML for communication
- UML for feeding a case tool
- UML used in Waterfall.
- One week waterfalls in iterative

Patterns

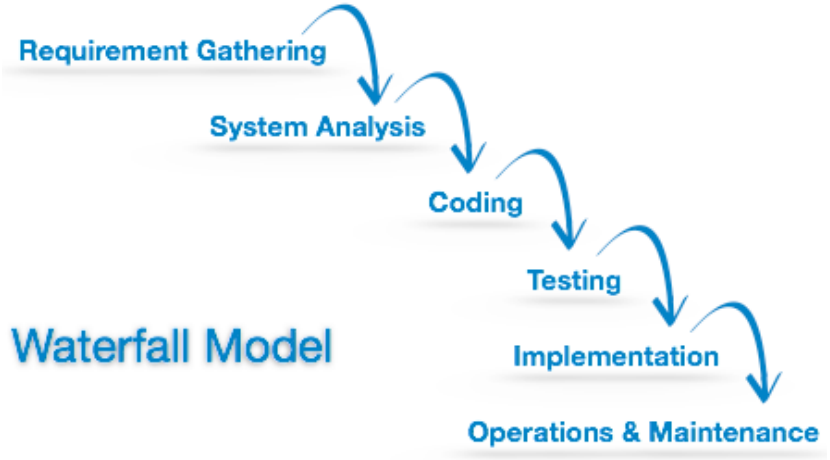
- Many people have commented that projects have problems because the people involved were not aware of designs that are well known to those with more experience . Patterns describe common ways of doing things änd are collected by people who spot repeating themes in designs . These people take each theme and describe lt so that other people can read the pattern and sec how to apply it .

  http://www.hillside.net/patterns .

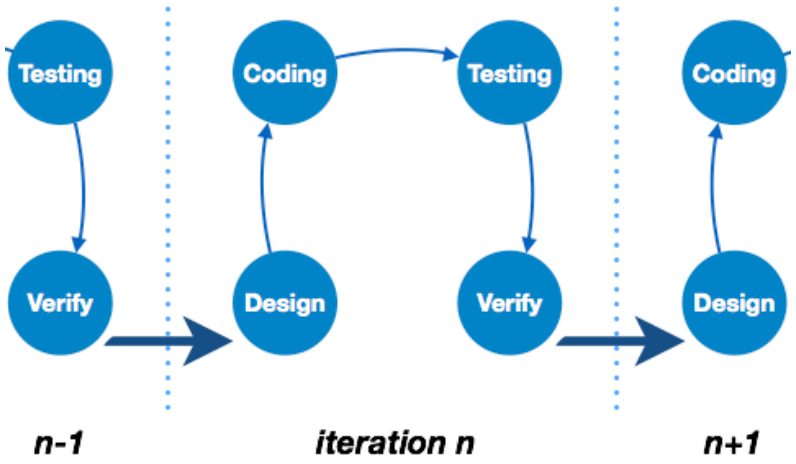


Spiral Model, Incremental Model, V-Model

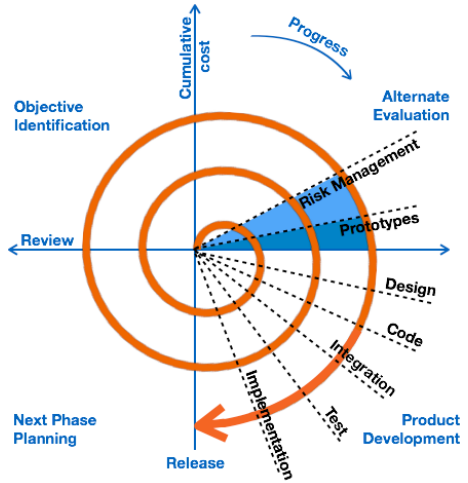
Waterfall Model



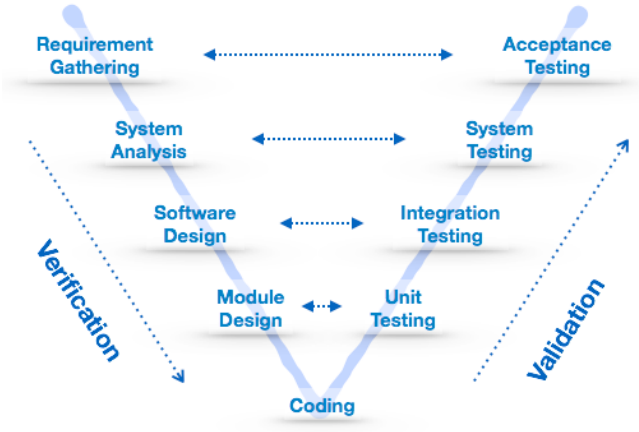
Iterative



Spiral

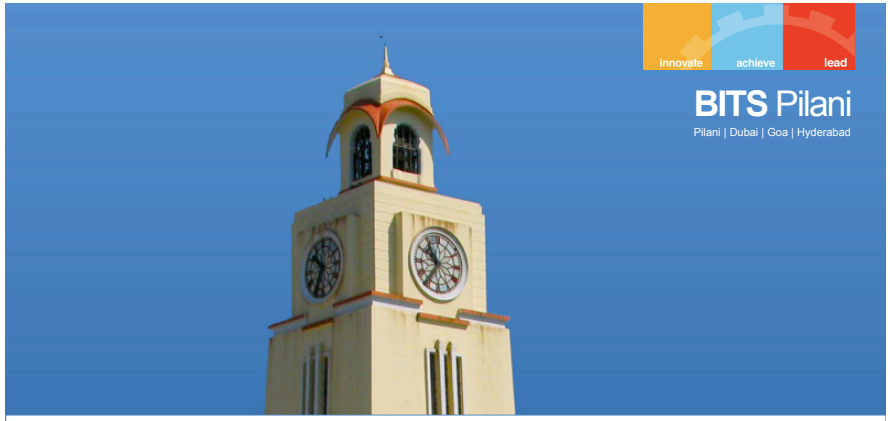


V Model



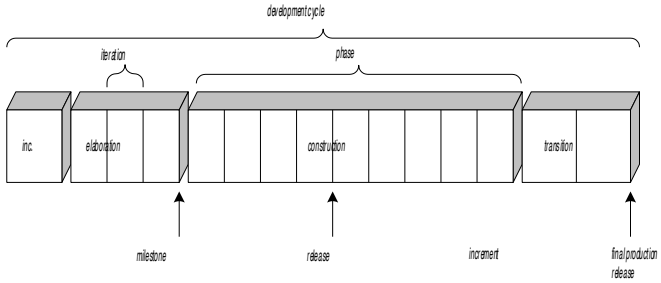
Big Bang

Various Models



UP Model, Phases & Disciplines, How it is different than Waterfall and other models?

UNIFIED PROCESS phases



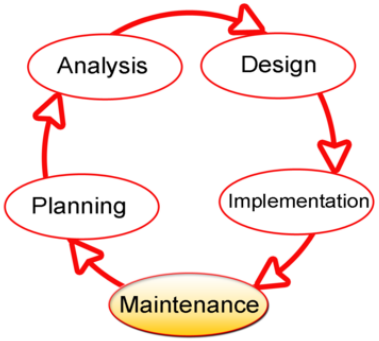
An iteration end-point when some significant decision or evaluation occurs

A stable executable subset of the final product. The end of each iteration is a minor release.

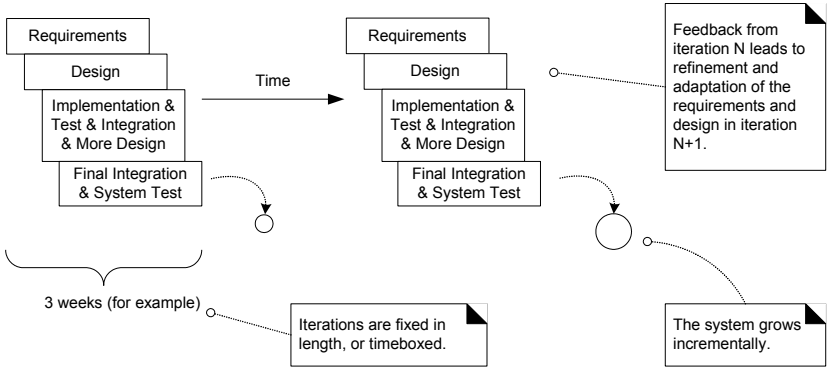
The difference (delta) between the releases of 2 subsequent iterations.

At this point, the system is release for production use

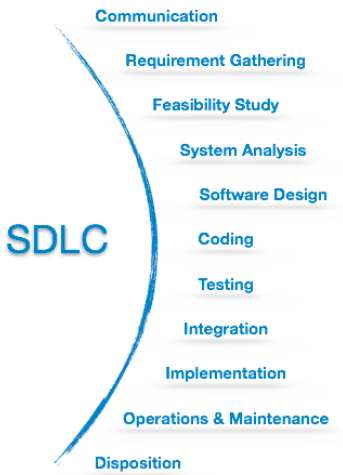
System Development Life Cycle



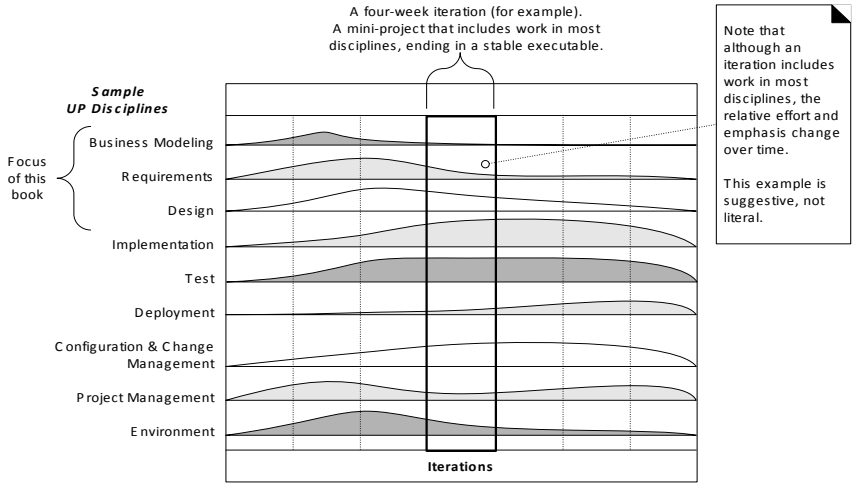
Iterative Process



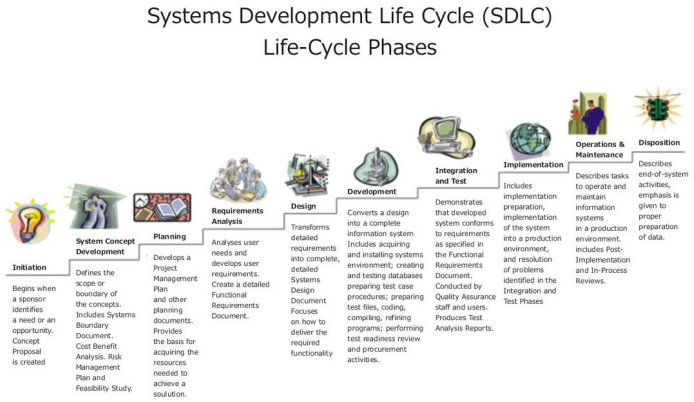
Software Development Life Cycle Activity



UNIFIED PROCESS disciplines

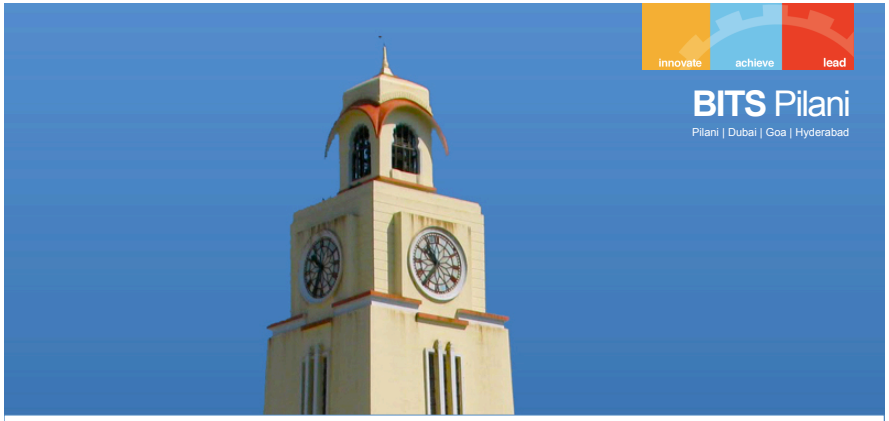


The complete life cycle of any System



Outline

No	Title
CS2.1.1	Give mall example as PoS case study, First step in OOA is requirement gathering and requirement categorization (Functional & Non Functional)
CS2.1.2	Explain difference between Use Case Diagram and Use Cases. Use Case Diagram is pictorial and Use Case is textual artefact.
CS2.1.3	Demonstrate how Use Case Diagram can be drawn for PoS Case Study.



Agile

Agile Manifesto

http://agileManifesto.org

Extreme Programming(XP), Scrum, Feature Driven Development (FDD), Crystal, Dynamic Sytem Development Method (DSDM)

Adaptive

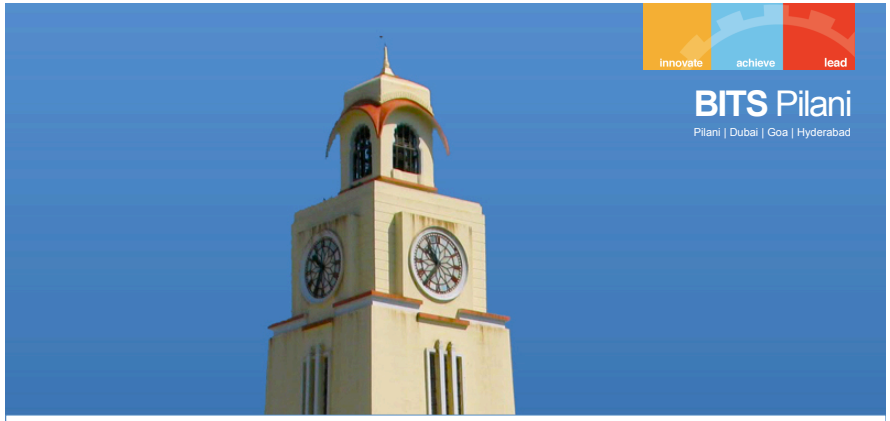
- Quality of people
- Working together

Low in ceremony,

Documentation

- Many people have commented that projects have problems because the people involved were not aware of designs that are well known to those with more experience . Patterns describe common ways of doing things änd are collected by people who spot repeating themes in designs . These people take each theme and describe It so that other people can read the pattern and sec how to apply it .

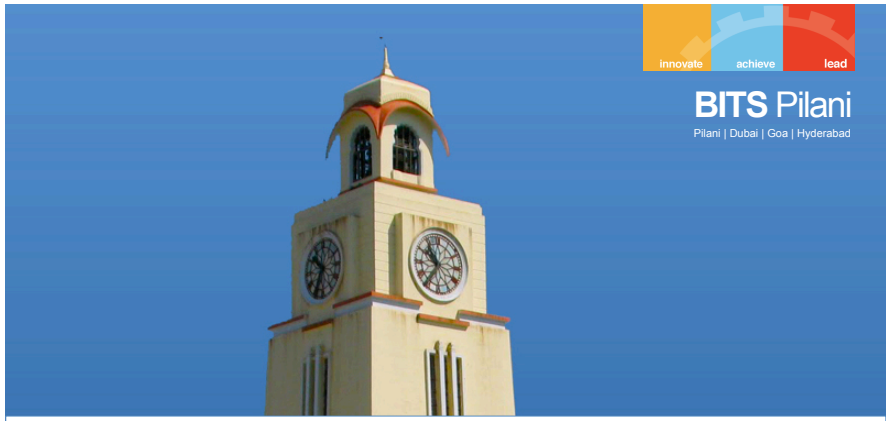
http://www.hillside.net/patterns .



Various Agile approaches

Various Agile Approaches

- Scrum
- XP
- TDD
- Refactoring



Class, Object and their representation in UML

Classes

- Learn to recognise Classes and interaction between classes.

What is Important

- Harmful-
- knowing how to read and draw UML diagrams, but not being an expert in design and patterns.

Important:
object and architectural design skills, not UML diagrams, drawing, or CASE tools.

http://staruml.io/

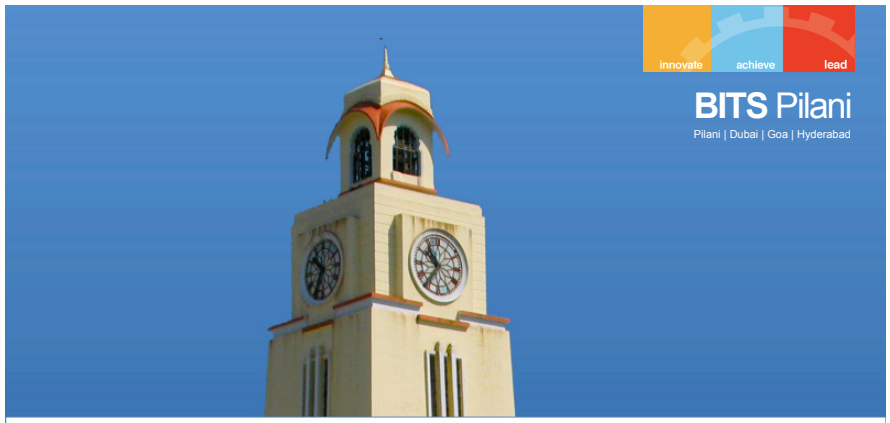
http://staruml.io/

Please go to this page.

- StarUML is one of the most popular UML tools in the world. It has been downloaded over than 4 millions and used in more than 150 countries.

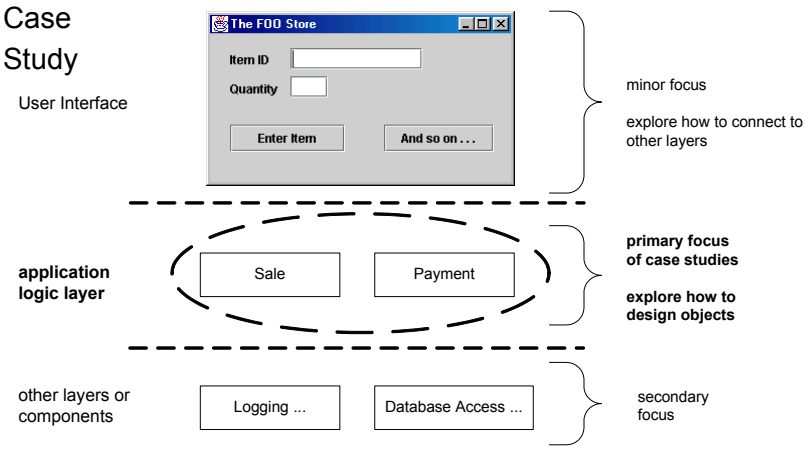
Outline

No	Title
CS2.1.1	Give mall example as PoS case study, First step in OOA is requirement gathering and requirement categorization (Functional & Non Functional)
CS2.1.2	Explain difference between Use Case Diagram and Use Cases. Use Case Diagram is pictorial and Use Case is textual artefact.
CS2.1.3	Demonstrate how Use Case Diagram can be drawn for PoS Case Study.



Case Study

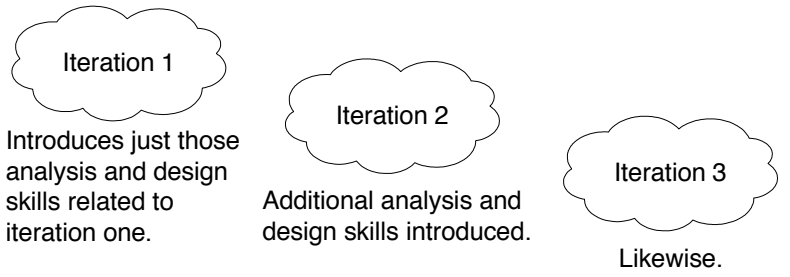
Fig. 3.1: Sample Layers and Objects

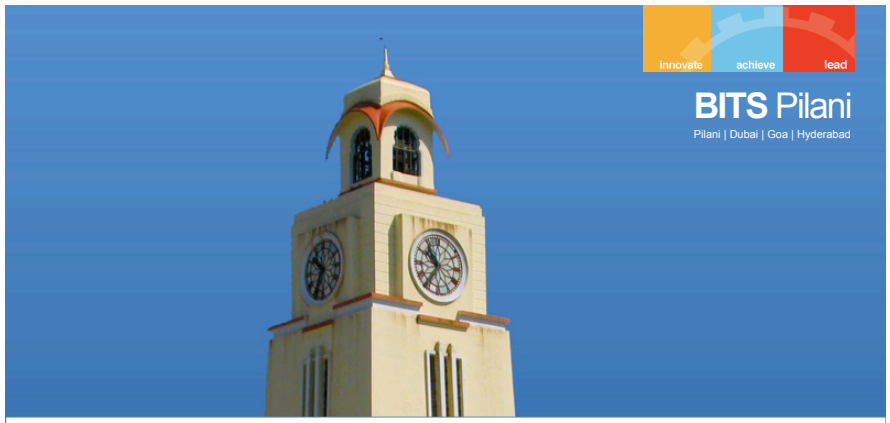


OOA/D Focus

- Core Application Logic Layer
  - Other layers Technology dependent
  - Application Logic similar across technologies
  - OO skills learned in this layer apply to all layers
  - Design Approach/patterns in other layers change very fast

Fig. 3.2 Learning Path follows iterations



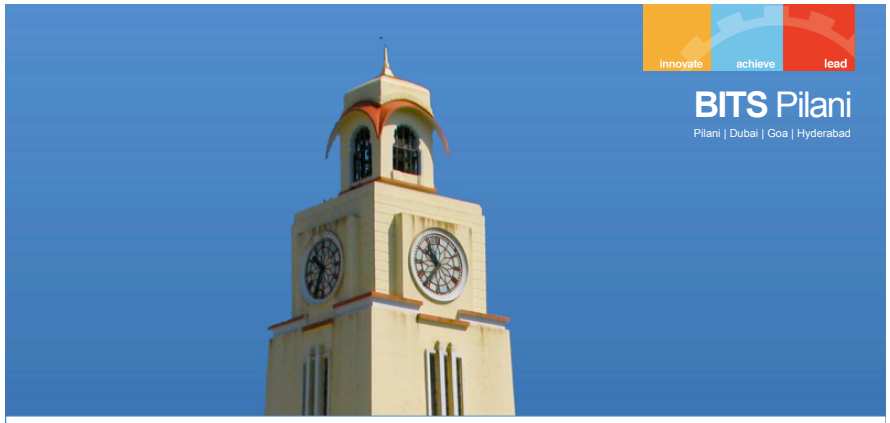


Inception Phase

innovate achieve lead

Inception

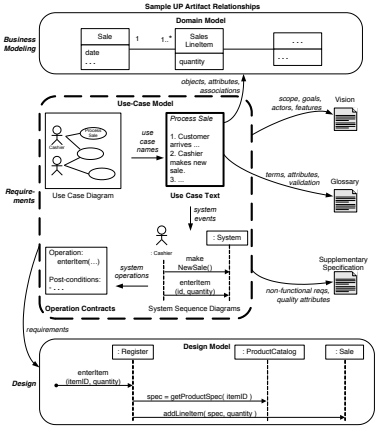
- Vision and Business Case
- Use-Case Model
- Supplementary
- Glossary
- Risk List & Risk Management Plan
- Prototype and Proof of Concept
- Iteration Plan
- Phase Plan and Software Development Plan
- Development Case



Use Cases

innovate achieve lead

UP Artifacts



Types of Actors

- Primary Actor
  - Whose user goals are fulfilled through using services of the System under Discussion. Eg Cashier
- Supporting Actor
  - Provides a service to the SuD. Eg Payment Authorisation
- Offstage Actor
  - Has an interest in the behaviour of the Use case. Eg Govn Tax Agency.

Common Use Case Formats

Brief
- Terse one paragraph summary
- Main Success Scenario.
- eg Process Sale

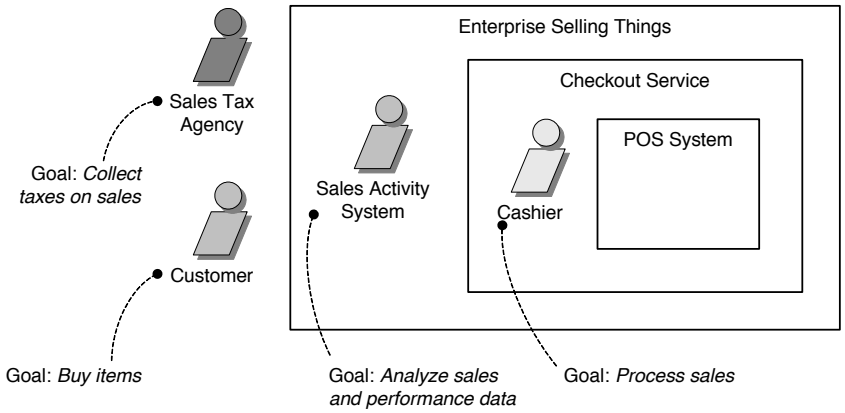
Casual
- Informal Paragraph format.
- Multiple paragraphs that cover various scenatios

Fully Dressed
- All steps and variations in detail
- Preconditions and success guarantees.

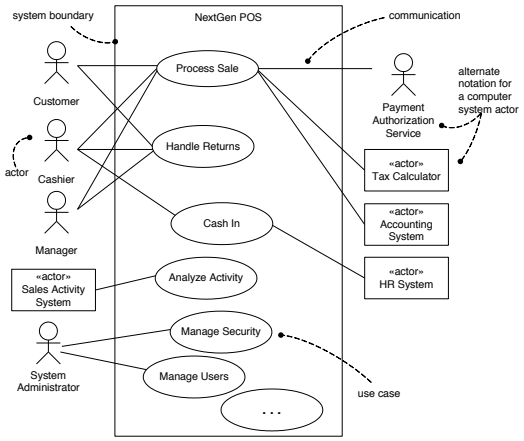
Fully Dresses Style

1. Use Case Name
2. Scope
3. Level
4. Primary Actor
5. Stakeholders and Interests
6. Preconditions
7. Success Guarantee
8. Main Success Scenario
9. Extensions
10. Special requirements

Actors and Goals



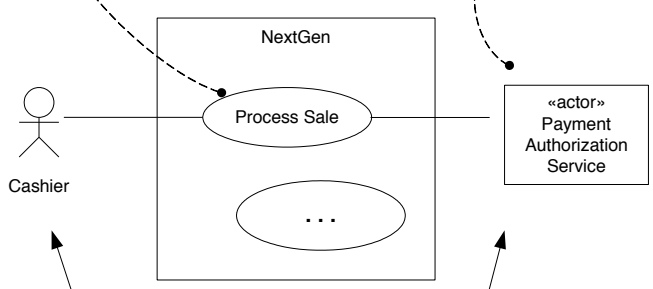
Partial Use Case Context



Notation for Diagrams

For a use case context diagram, limit the use cases to user-goal level use cases.

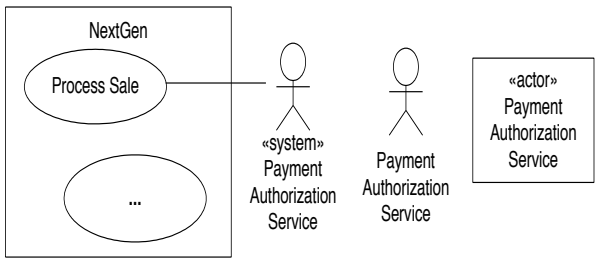
Show computer system actors with an alternate notation to human actors.



primary actors on the left

supporting actors on the right

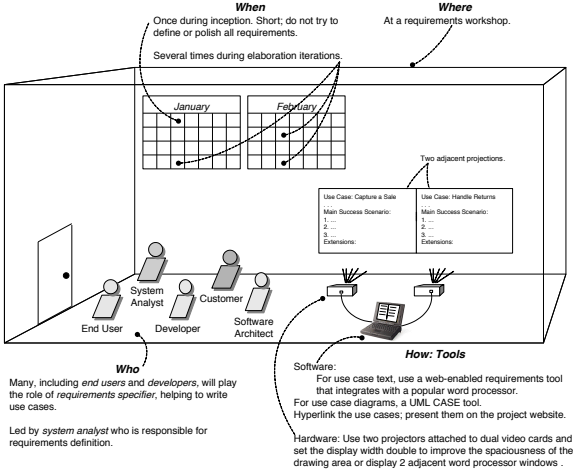
Alternate Actor Notation



Some UML alternatives to illustrate external actors that are other computer systems.

The class box style can be used for any actor, computer or human. Using it for computer actors provides visual distinction.

Process and setting context - Writing use cases



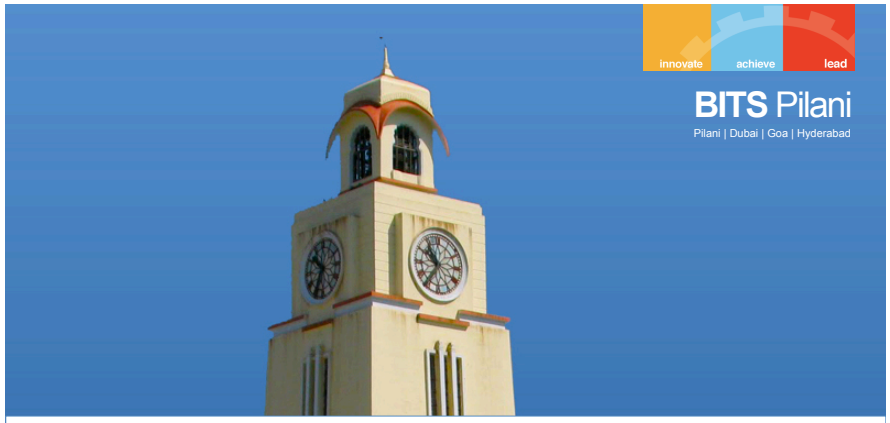
SS ZG514 Object Oriented Analysis and Design

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Harvinder S Jabbal
Session 06:Fully Dressed Use case

Outline

No	Title
CS2.2.1	Explain Fully dressed use case syntax
CS2.2.2	Demonstrate writing Fully Dressed Use Case for Process Sale scenario in PoS case study.



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Fully dressed use case syntax

What is Use Case

Use cases are one way to capture (especially) functional requirements.

- They are stories of using a system.

You should be able to:

- Identify different use case levels
- Write use cases in the popular Cockburn format
- Contrast essential and concrete use cases
- Apply guidelines
- Read and write use case diagrams

Common Use Case Formats

Brief
- Terse one paragraph summary
- Main Success Scenario.
- eg Process Sale

Casual
- Informal Paragraph format.
- Multiple paragraphs that cover various scenatios

Fully Dressed
- All steps and variations in detail
- Preconditions and success guarantees.

Use case

Informally, a *use case* is a story of using a system to fulfill a goal.
- *Rent Videos*

Used by *primary actors*
- *E.g., Clerk*
- *External agents*
- *something with behavior*

Use *supporting actors*.
- *CreditAuthorizationSystem*

Brief Use Case

Rent Videos.

A Customer arrives with videos to rent. The Clerk enters their ID, and each video ID. The System outputs information on each. The Clerk requests the rental report. The System outputs it, which is given to the Customer with their videos.

Informal and Formal Use Case

- Informally, a *scenario* is a specific sequence of actions and interactions in a use case.
  - One path through the use case.
  - E.g., The scenario of renting videos and first having to pay overdue fines.
- Formally, a *use case* is a collection of success and failure scenarios describing a primary actor using a system to support a goal.

Nothing OO about it. Why Use Case?

- We need some kind of requirements input for the design steps.
- They are common/popular.
- There is a UML-related topic.

Elementary Business Process (EBP) guideline

- Focus on Use Cases at the level of EPBs.
  - We can end up with too many fine-grain use cases
    - management and complexity problems.
  - Or "fat" use cases which span an entire organization.
- *"A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state." - Cockburn's Guideline.*

Test for Level

- Would the Boss be satisfied if the concerned person was doing this all day long?
- An EBP-level use case *usually* is composed of several steps, not just one or two. It isn't a single step.

UML and Use Case

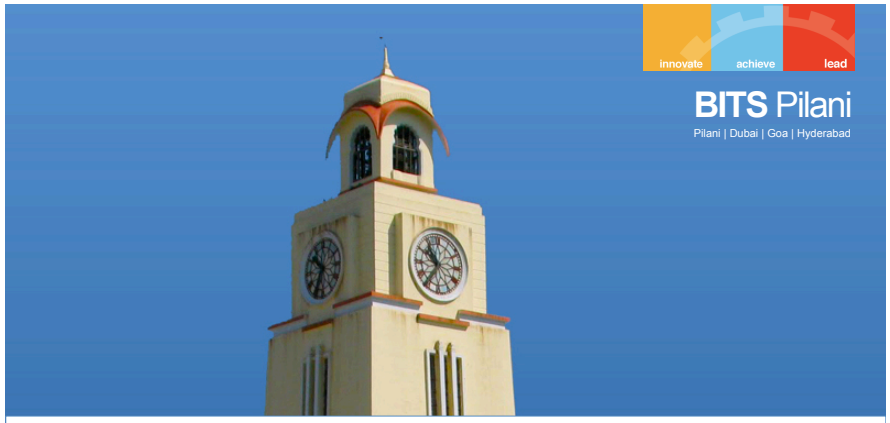
- The UML has use case diagrams.
- Use cases are *text*, not diagrams.
  - Use case analysis is a *writing* effort, not drawing.
- But a *short* time drawing a use case diagram provides a context for:
  - identifying use cases by name
  - creating a "context diagram"

Fully Dressed Use Case

- Rich notation for detailed analysis.
- Shows branching scenarios.
- Can include non-functional requirements related to the functional.

Fully Dresses Style

1. Use Case Name
2. Scope
3. Level
4. Primary Actor
5. Stakeholders and Interests
6. Preconditions
7. Success Guarantee
8. Main Success Scenario
9. Extensions
10. Special requirements



Example: Fully Dressed Use Case

Example of Fully Dressed Use Case

Use Case UC1: Rent Video
Level: User-level goal (EBP level)

Primary Actor: Clerk
Preconditions:
Clerk is identified and authenticated.

Stakeholders and their Interests:
Clerk: Wants accurate, fast entry.
Customer: Wants videos, and fast service with minimal effort.
Accountant: Wants to accurately record transactions.
Marketing: Wants to track customer habits.
. . .

Main Success Scenario

Main Success Scenario (or Basic Flow or "Happy Path"):

1. Customer arrives at a checkout with videos or games to rent.
2. Clerk enters Customer ID.
3. Clerk enters rental identifier.
4. System records rental line item and presents item description.
   (Clerk repeats steps 3-4 until indicates done.)
5. System displays total.
6. Customer pays. System handles payment.
7. Clerk requests rental report.
8. System outputs it. Clerk gives it to Customer.
9. Customer leaves with rentals and report.

Extensions (or Alternatives)

a*. At any time, System fails:
   1. Clerk restarts System
   2. logs in
   3. requests recovery from prior state

1a. New Customer.
   1. Perform use case Manage Membership.

2a. Customer ID not found.
   1. Cashier verifies ID. If entry error, reenter, else Manage Membership.

2b. Customer has unpaid fines (usually for late returns).
   1. Pay Fines.

Other Items

Special Requirements:
- Language internationalization on the display messages and rental report.
- Large text on display. Visible from 1 m.

Technology and Data Variations:
- ID entries by bar code scanner or keyboard.

Frequency:
- Near continuous

Open Issues:
- Should we support a magnetic stripe cards for customer ID, and allow customer to directly use card reader?

Essential and Concrete Use Cases

"Keep the UI out"

Essential use cases defer the details of the UI, and focus on the *intentions* of the actors.

Essential: Clerk enters Customer ID.

Concrete/worse: Clerk takes Customer ID card and reads the bar code with laser scanner.

Guideline

It is common to group CRUD operations into one use case.

- Manage Users

CRUD (Create Read Update Delete)

Name starts with a verb.

- *Manage* Users

All systems have a *Start Up* and *Shut Down* use case (perhaps trivial and low level).

- But sometimes, important.
  - an avionics system

Guideline for writing

Start sentence 1 with "<Actor> does <event>"

- Customer arrives with videos to rent.

First write in the essential form, and "Keep the UI out."

Capitalize "actor" names.

1. …
2. *Clerk* enters…
3. *System* outputs…

Guidelines

- Terse is good. People don't like reading requirements ;). Avoid noisy words.

More verbose

1. …
2. The Clerk enters…
3. The System outputs…

Less

1. …
2. *Clerk* enters…
3. *System* outputs…

Requirements in Context

- Use Case: Requirements in Context
  - Use cases bring together related requirements.
  - More cohesion and context for related requirements.

Concrete Use Case

- Sometime after the essential form of the use case has been written, one may *optionally* write it in a concrete form.

1. Customer arrives at a checkout with videos or games to rent.
2. Clerk *scans* Customer ID…

Extensions

2a. Scanner failed.

1. Clerk enters ID on keyboard (see GUI window example, fig 5)…

Use Cases are Stories

Comprehensible

&

Familiar

Thank You

The slides are based on:

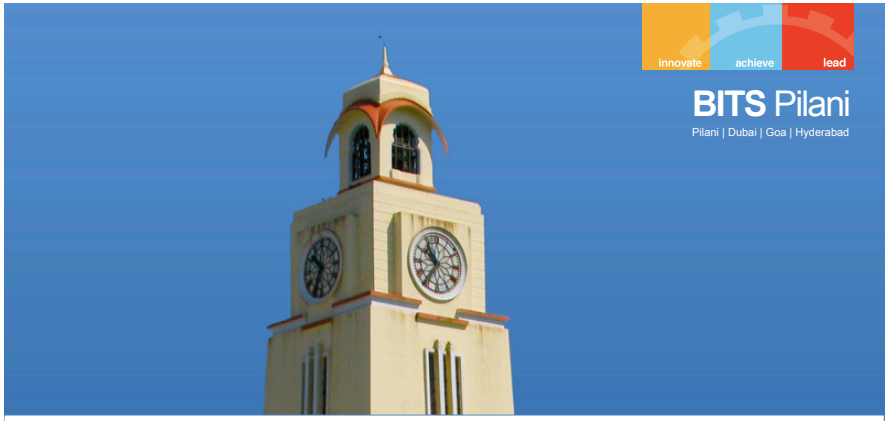
Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

No	Title
CS3.1.1	Explain how Domain Concepts are different than software classes, how domain concepts to be identified?
CS3.1.2	How to identify Associations and Multiplicity among domain concepts
CS3.1.3	Explain how to add attributes to Domain Model?
CS3.1.4	Demonstrate drawing complete Domain Model for PoS System



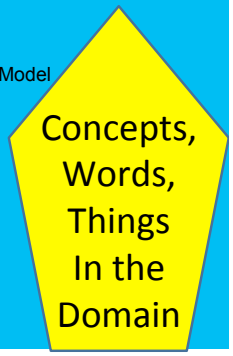
Domain Concepts

Overview

A Domain Model visualizes noteworthy concepts or objects in the domain.

You should be able to:

- Read and write the UML class diagram notation for a Domain Model
- Create a Domain Model
- Apply guidelines
- Relate it to other artifacts



DEFINITION & MOTIVATION: Domain Model

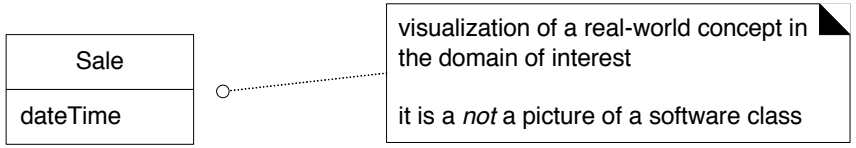
A *Domain Model* visualizes, using UML class diagram notation, noteworthy concepts or objects.

- It is a kind of "visual dictionary."
- *Not* a picture of software classes.

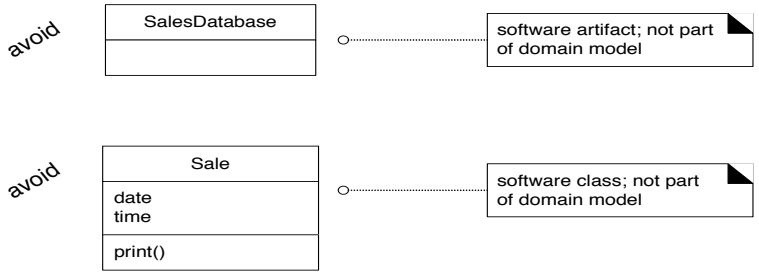
It helps us identify, relate and visualize important information.

It provides inspiration for later creation of software design classes, to reduce "representational gap."

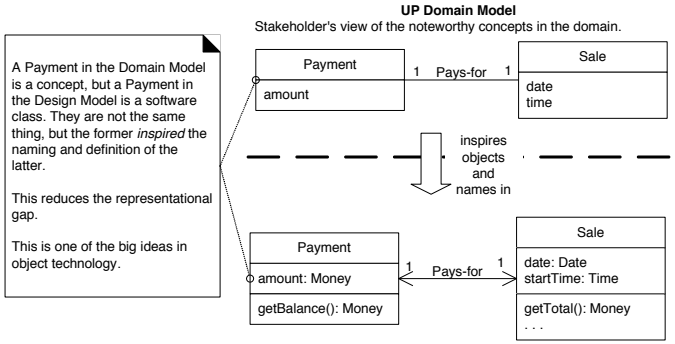
Guideline: show real-situation conceptual classes, not software classes



A domain model does not show software artifacts or classes



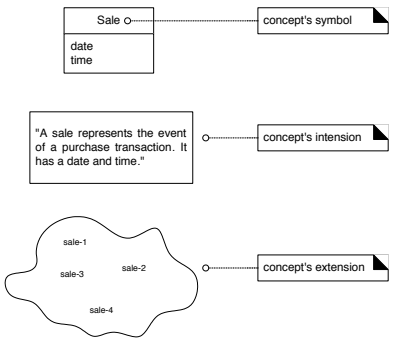
Lower Representational Gap



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

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

Conceptual Class – symbol, intension, extension



Guideline: Creting Domain Model

- Find the Conceptual Classes
  1. Reuse or modify existing models
  2. Use category list (see next slide)
  3. Identify noun phrases
- Draw them in UML
- Add Associations and attributes

Conceptual Class Category List

Conceptual Class Category	Guideline	Examples
Business transaction	They are critical (involve money), so start with transactions	Sale, Purchase, reservation
Transaction line item	Transactions often come with related items, so consider these next	SelesLineItem
Product or service related to a Transaction or Transaction line item	Transactions are for something (a product or service). Consider next.	Item, Flight, Seat, Meal
Where is the transaction recorded	Important	Register, Ledger, FlightManifest
Role of people or organisation related to the transaction; actors in a use case		Cashier, Customer, Store, MonopolyPlayer, Passenger, Airline
Place of transaction; place or service		Store, Airport, Plane, Seat

Conceptual Class Category List

Conceptual Class Category	Guideline	Examples
Noteworthy events, often with a time or place we need to remember		Sale, Payment, MonoployGame, Flight
Physical object	Especially relevant when creating a device-control software or simulation	Item, register, Board, Piece, Die, Airplane
Description of things		ProductDescription, FlightDescription
Catalogs	Descriptions are often in a Catalog.	ProductCatalog, FlightCatalog
Containers of things(physical or informtion)		Store, Bin, Board, Airplane

Conceptual Class Category List

Conceptual Class Category	Guideline	Examples
Things in a container		Item, Square (in a board), Passenger
Other collaboration systems		CeditAuthorisationSystem, AirTrafficCOntrol
Records of finance, work, contracts, legal matters		Receipt, Ledger, MaintenanceLog
Financial instruments		Cash,Cheque, LineofCredit, TicketCredit
Schedules, manuals, documents that are regularly referred to in order to perform work		DailyPriceChangeList, RepairSchedule

GUIDELINES: Finding Domain Concepts

"Abbott" Analysis; Linguistic Analysis

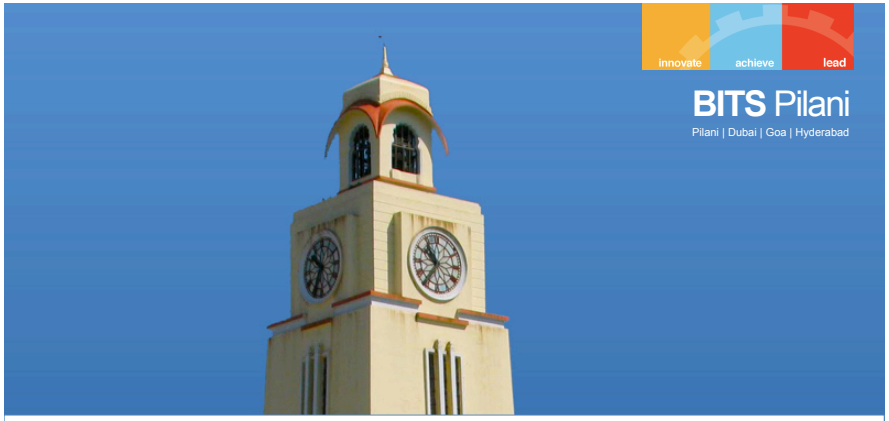
- Most simply, "pick out the nouns" – a mechanical noun-to-class mapping isn;'t possible, and words in natuaral language are ambiguous.

Avoid a waterfall-mindset big-modeling effort to make a thorough or "correct" domain model –

- It won't ever be either, and such over-modeling efforts lead to analysis paralysis, with little or no return on investment.

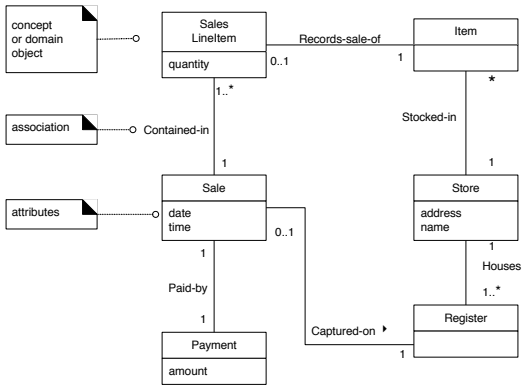
Initial PoS Domain Model

Register	Item	Store	Sale
Sales LineItem	Cashier	Customer	Ledger
Cash Payment	Product Catalog	Product Description	

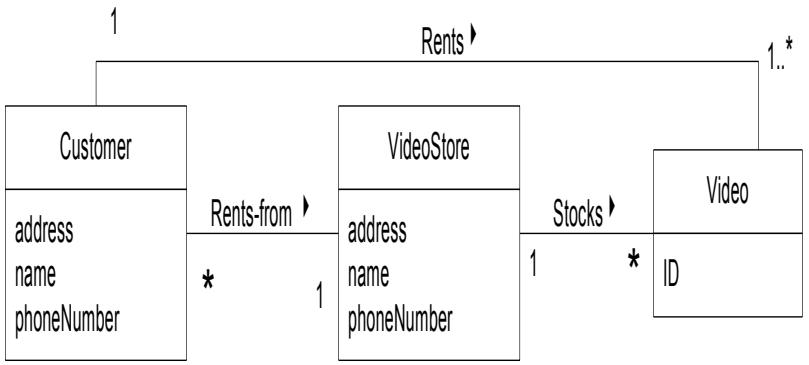


Adding Attributes to Domain Models

Partial Domain Model - A visual dictionary



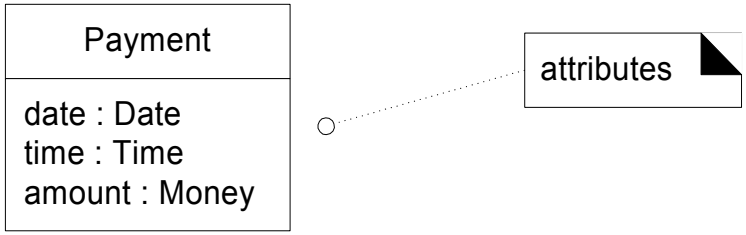
Partial Domain Model



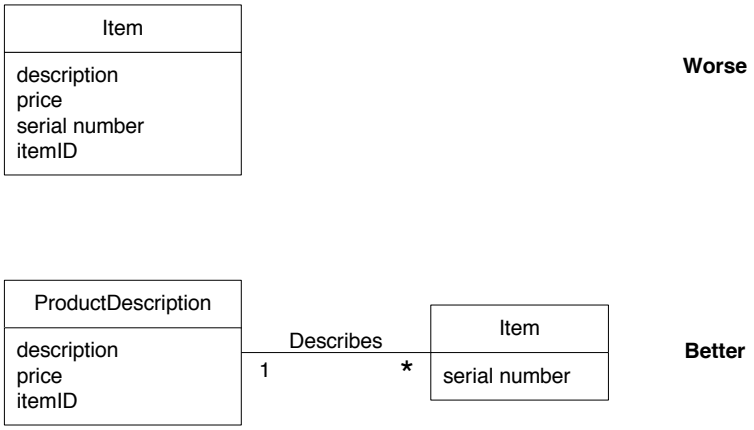
UML and GUIDELINES: Attributes

Show only “simple” relatively primitive types as attributes.

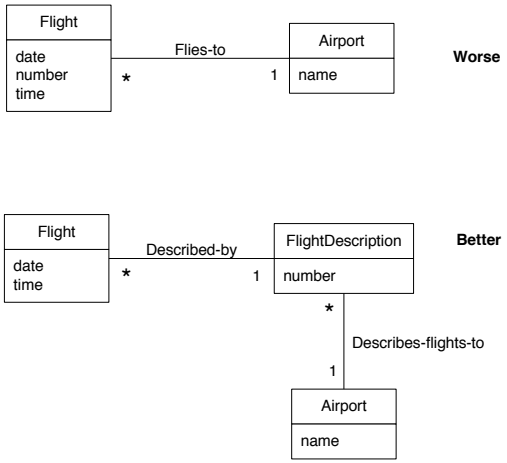
Connections to other concepts are to be represented as associations, not attributes.



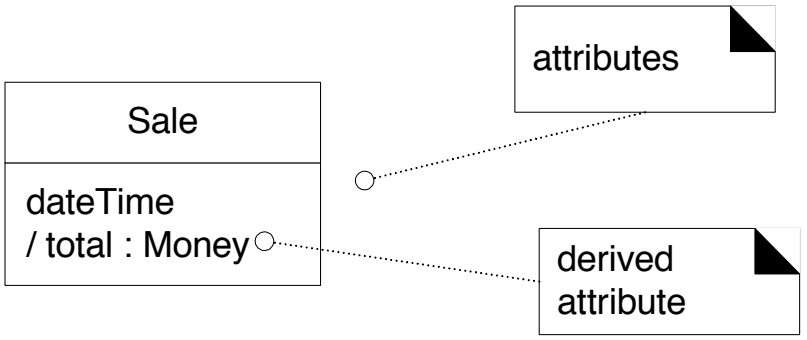
When a ProductDescriotion has many items



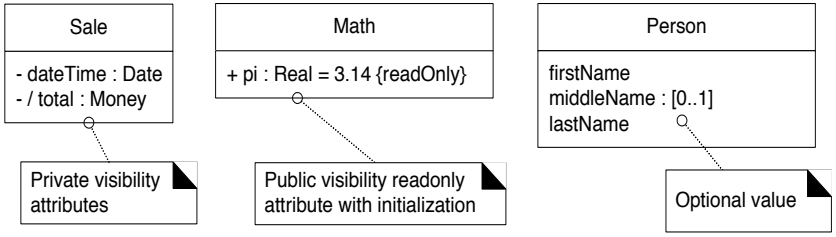
Description about other things

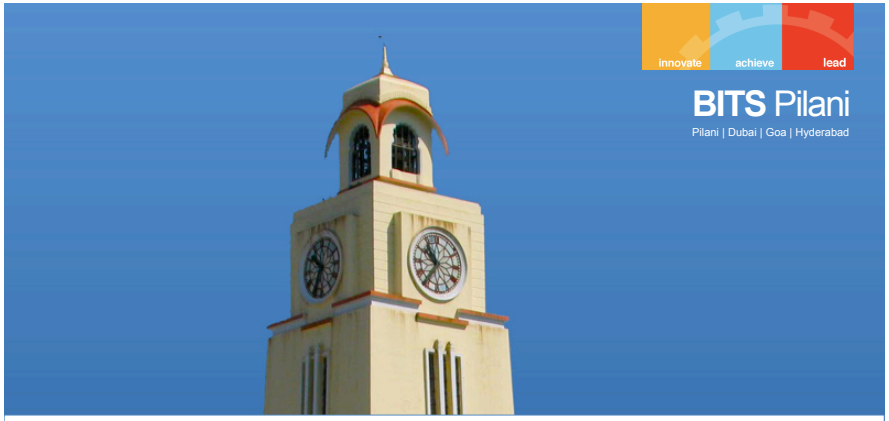


UML: Class and attributes



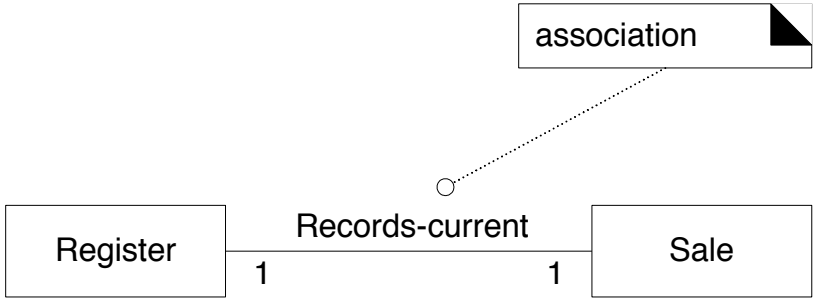
UML: Attribute Notations



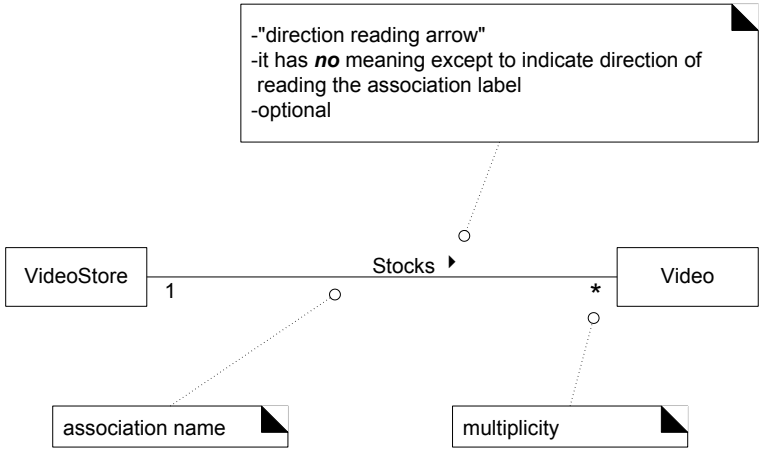


Associations and Multiplicity

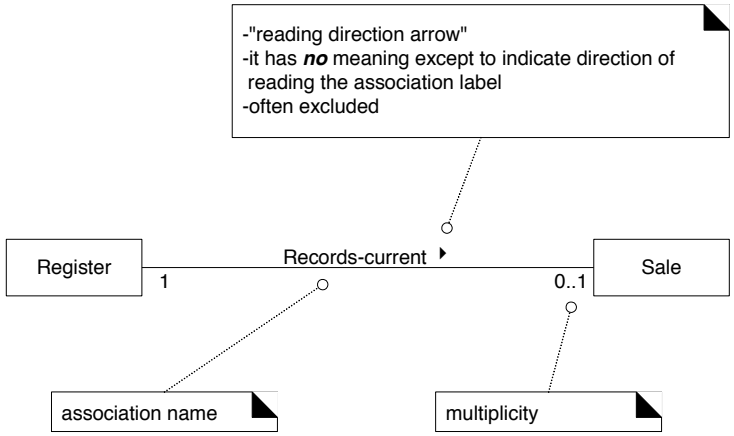
Association



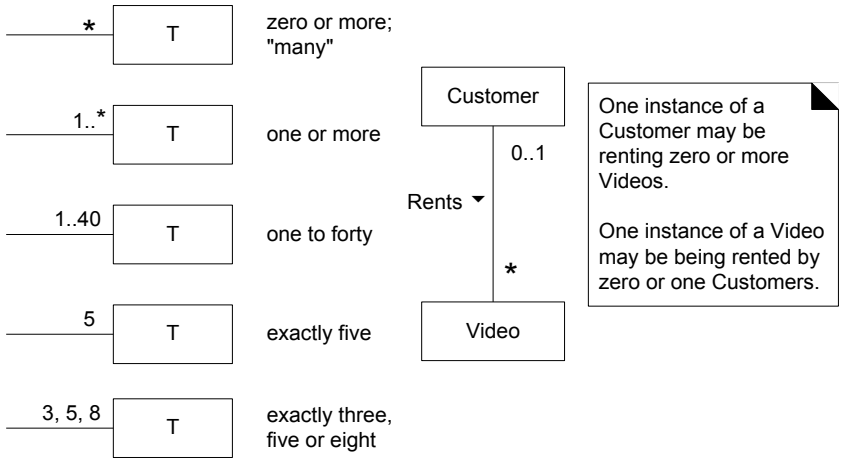
UML notation for Associations



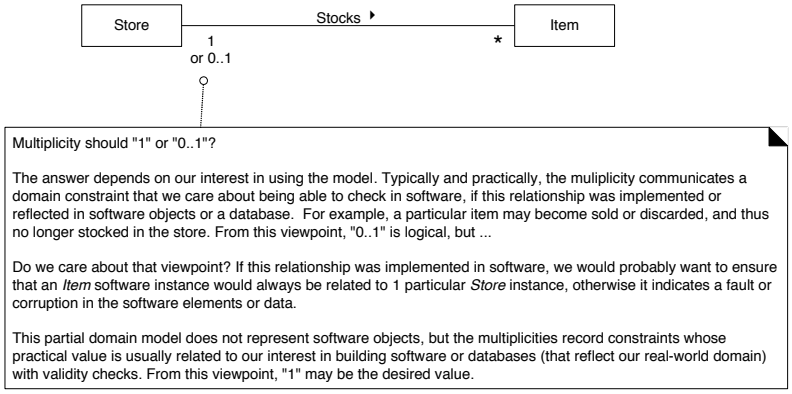
Application UML: Multiplicity



UML: Multiplicity

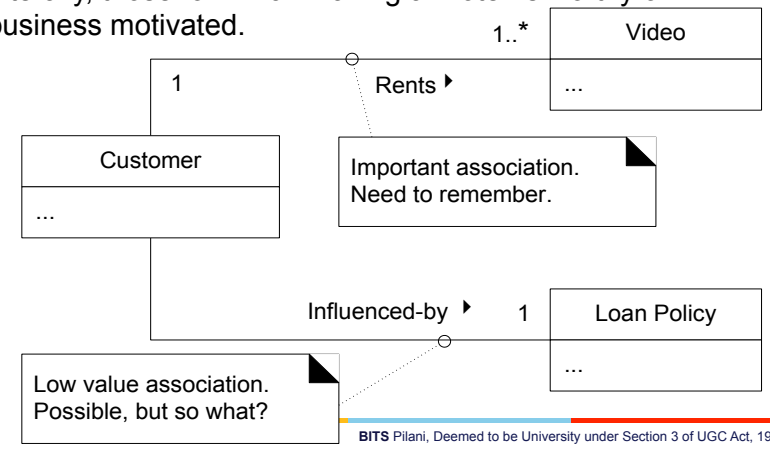


Multiplicity is context dependent



GUIDELINES: Associations

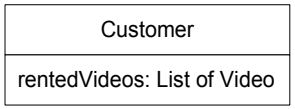
- Only add associations for *noteworthy* relationships. Literally, those for which making a “note” is worthy or business motivated.

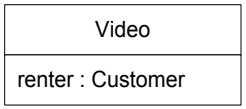


GUIDELINES: Attributes

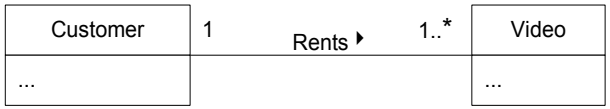
Why??

Worse

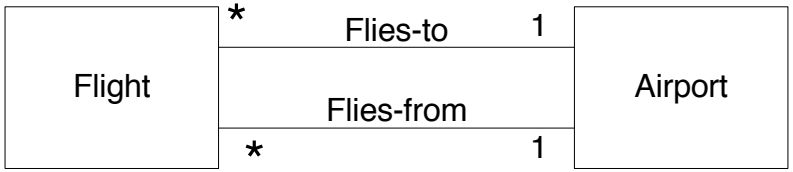


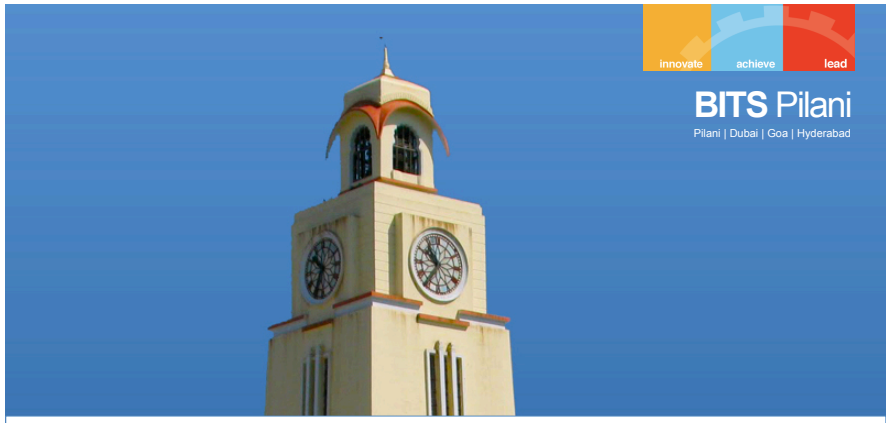


Better



Multiple Association

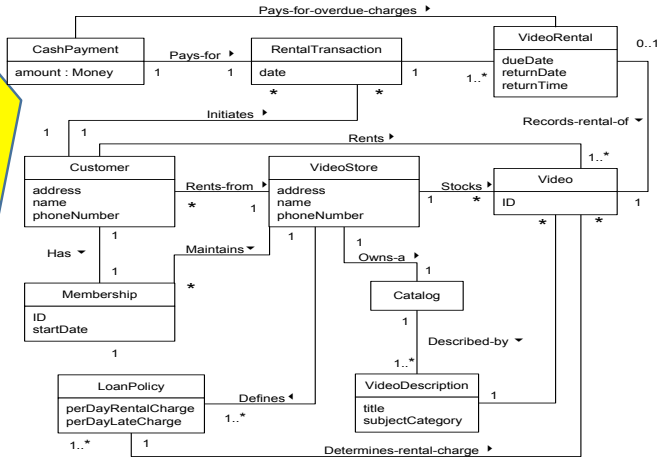




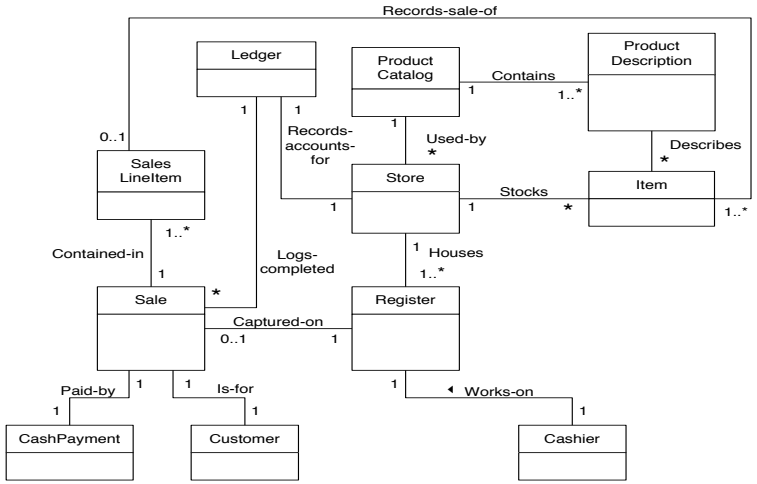
Demonstration

Domain Model- Visual Dictionary

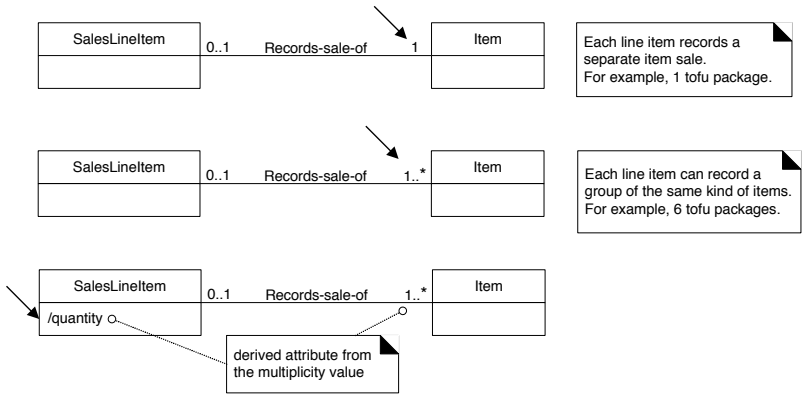
Concepts, Words, Things In the Domain



PoS Partial Domain Model



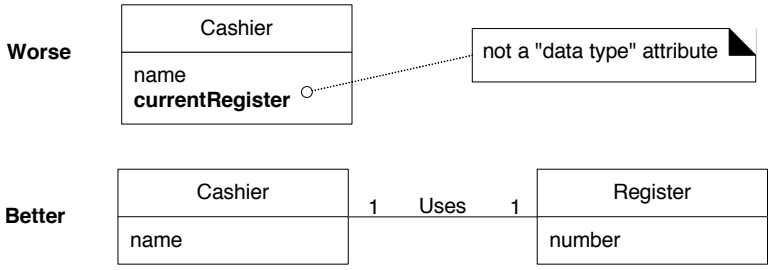
Derived Attributes



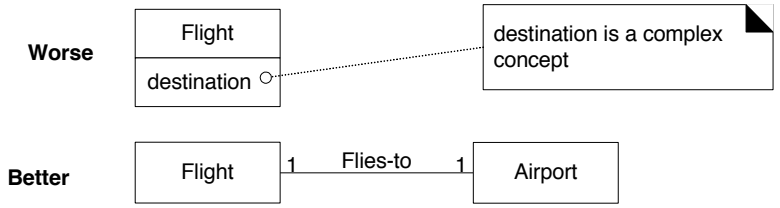
Focus on data type attributes

- Data types
  - Boolean
  - Date (or DateTime)
  - Number
  - Character
  - String (Text)
  - Time
- Other common types
  - Address
  - Colour
  - Geometrics (Point, Rectangle)
  - PhoenNumber
  - SocialSecurityNumber
  - UniversalProductCOde (UPC)
  - SKU,
  - ZIP or postal code
  - Enumerated types

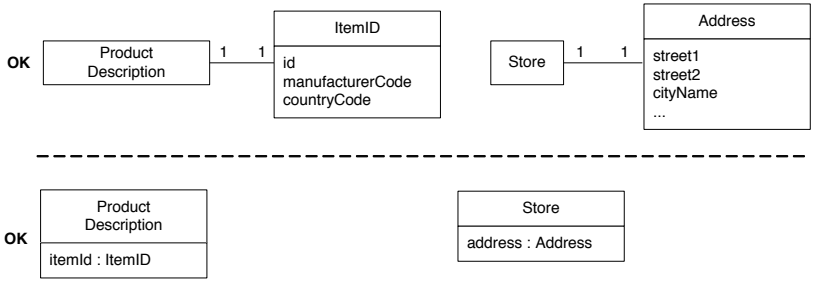
Relate Conceptual Class with an association, not an attribute.



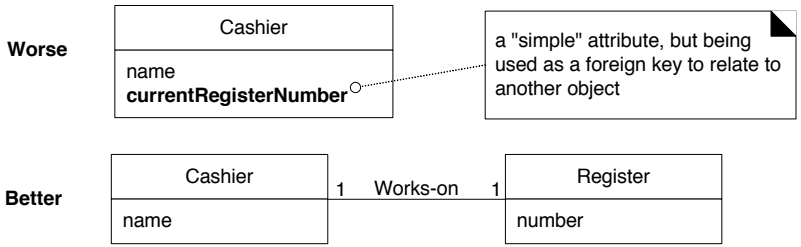
Don’t show complex concepts as attributes, use associations



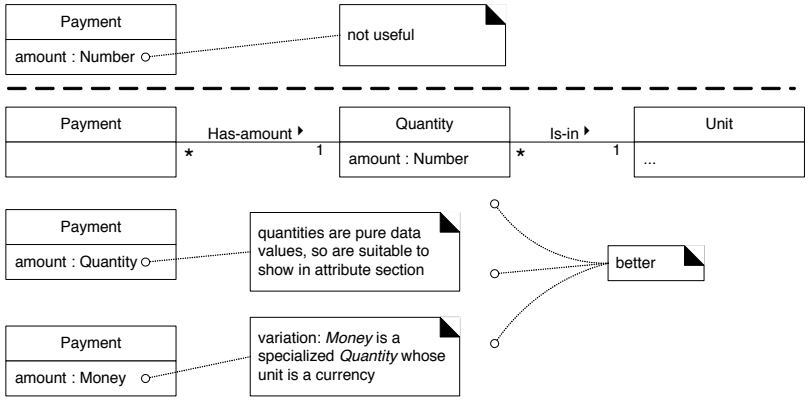
Data Type Property of an Object – 2 representations



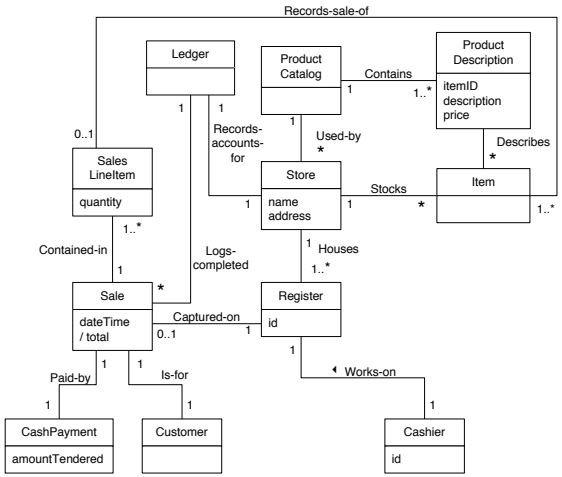
Do not use attributes as foreign keys



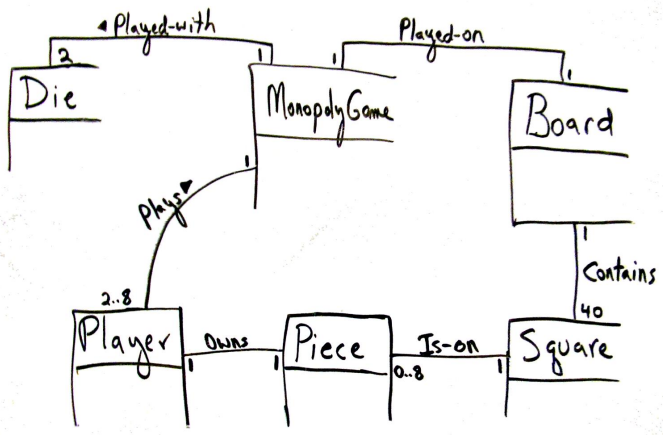
Modeling Quantities



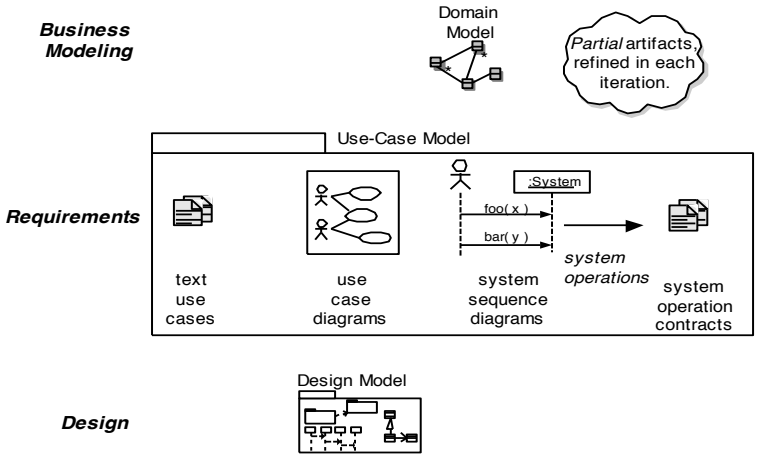
NextGen POS partial domain model



Monopoly Game- Partial Domain Model



UP use Case Model- Artifacts



Thank You

The slides are based on:

Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

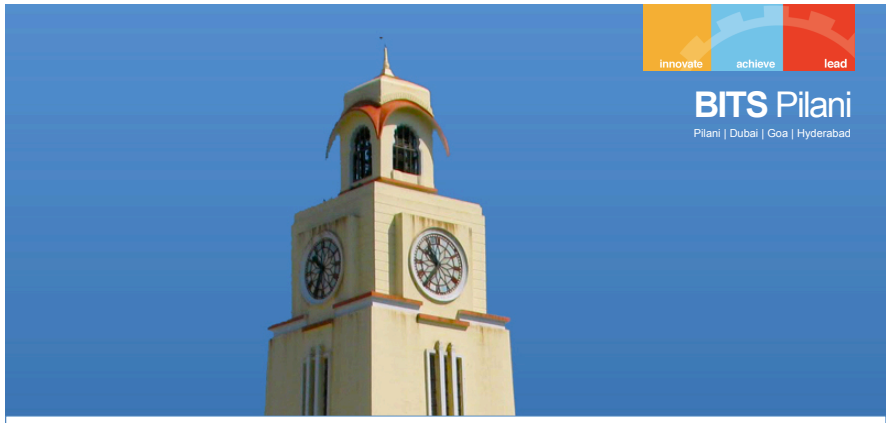
SS ZG514 Object Oriented Analysis and Design

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Harvinder S Jabbal
Session 08: SSD and Contracts

Outline

No	Title
CS3.2.1	Explain significance of SSD, Operation Contracts
CS3.2.2	Demonstrate drawing SSD and writing operation contract for PoS System

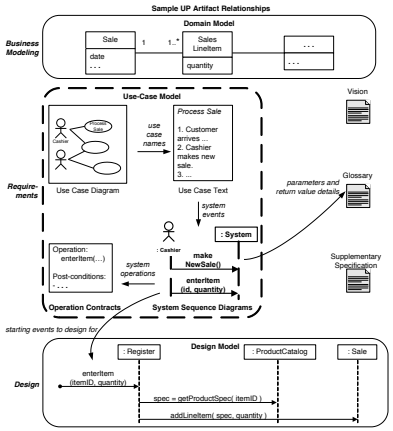


BITS Pilani
Pilani | Dubai | Goa | Hyderabad

SSD

Context of SSD

- Use Case Text and its implied system events are inputs to SSD
- SSD operations are in turn analysed in the operation contract.
- SSD are the starting point for designing collaborating objects.



Overview

An SSD shows (for a particular course of events within a use case),

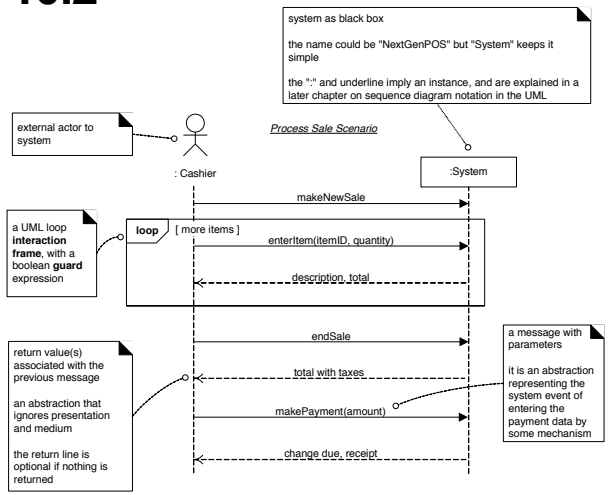
- the external actors that interact directly with the system,
- the system (as a black box), and
- The system events that the actors generate.

Starting point for designing collaborating objects

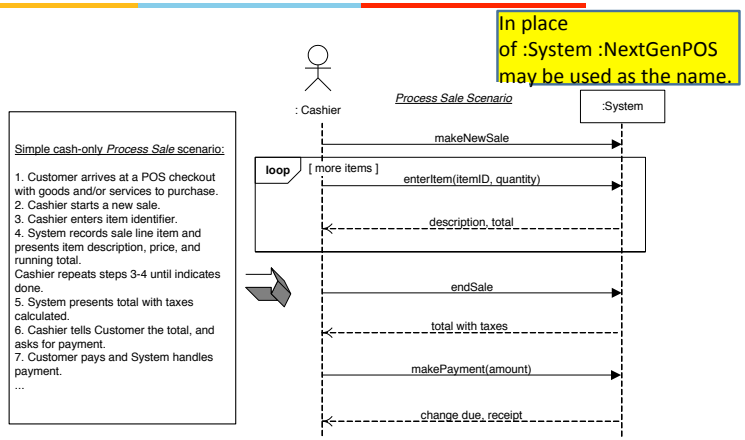
TIME PROCEEDS DOWNWARDS –

- ordering of events should follow the order they follow in the scenario.

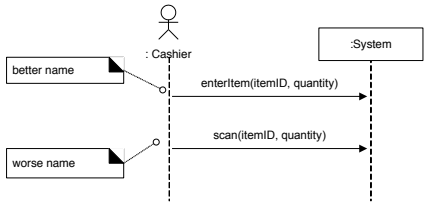
Fig. 10.2



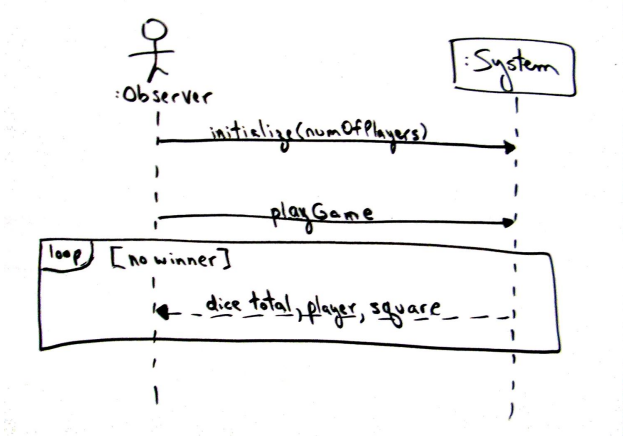
SSD for Process Sale scenario (main success scenario)

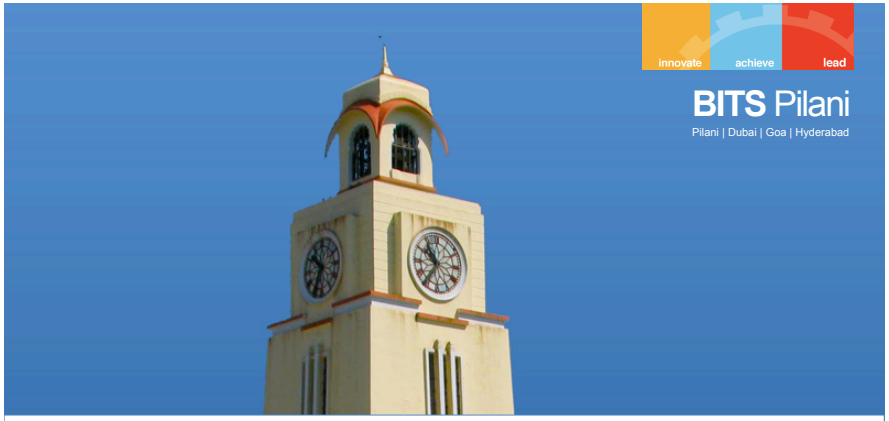


Event and Operation names at Abstract Level



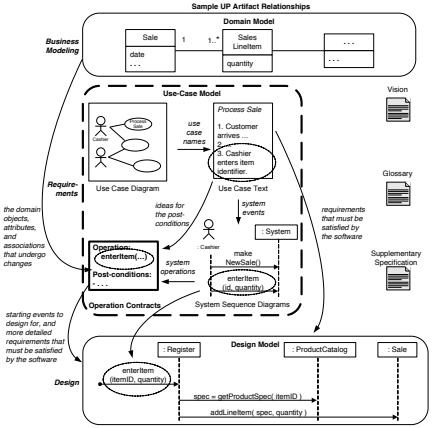
SSD for Play Monopoly



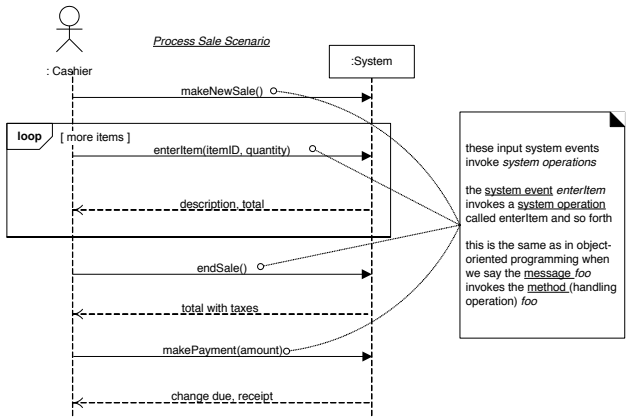


Operating Contract

UP Artifact influencing Operation Contracts



Systems operations handle input system event



Thank You

The slides are based on:

Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

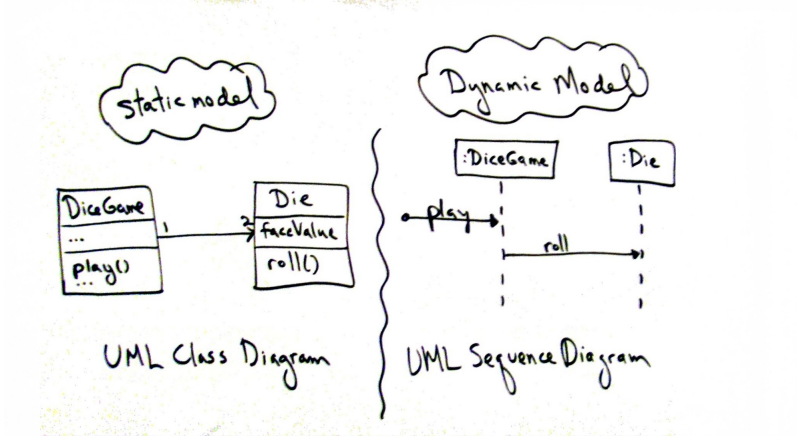
By Craig Larman

Outline

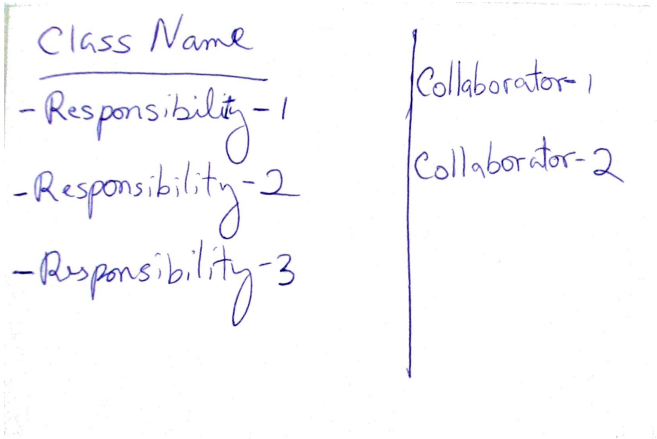
No	Title
CS4.1.1	Explain how transition happens from Object Oriented Analysis to Object Oriented Design? How OOA artefacts gets utilized in OOD?
CS4.1.2	Refinements done by Designer in Use Case Model & Domain Model
CS4.1.3	Demonstrate refinements in already created Use Case Model and Domain Model for PoS System.

Object Design

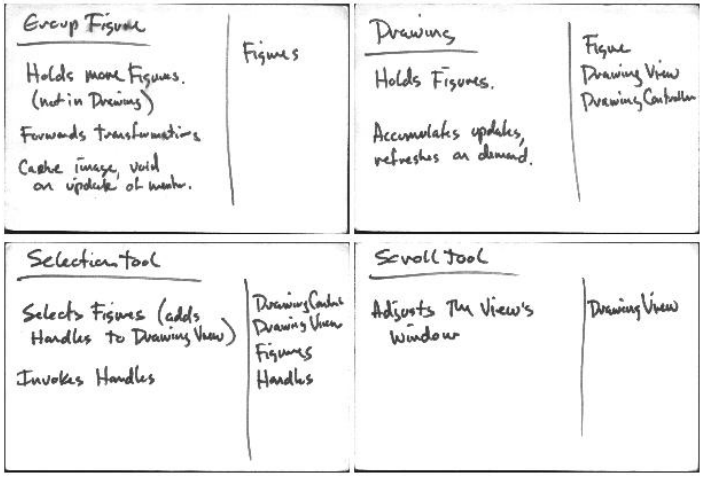
Static and Dynamic UML diagrams for object modeling



Template for a CRC Card - Class Responsibility Collaboration

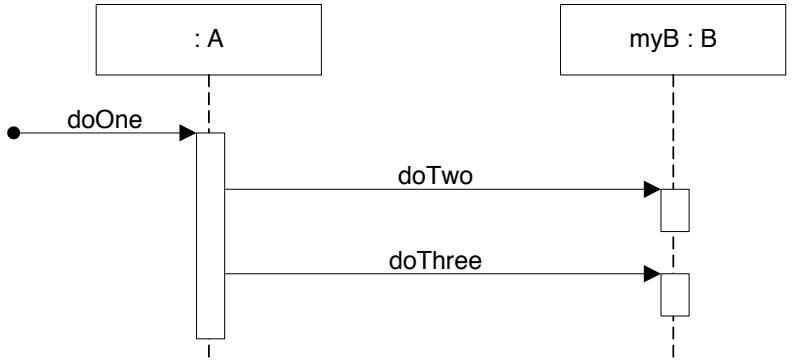


Sample CRC Cards

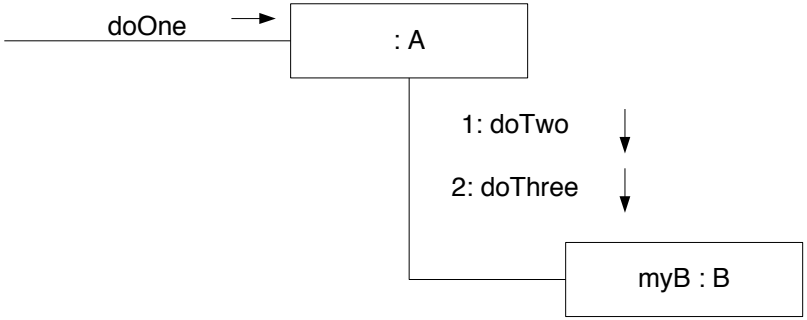


Interaction Diagrams

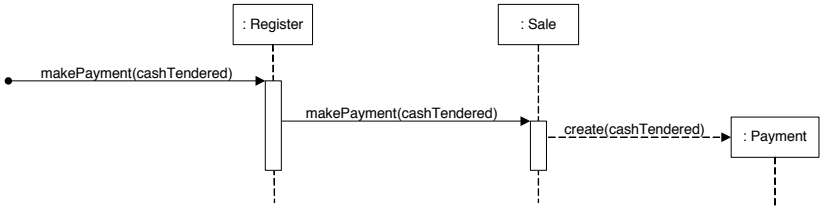
Sequence Diagram



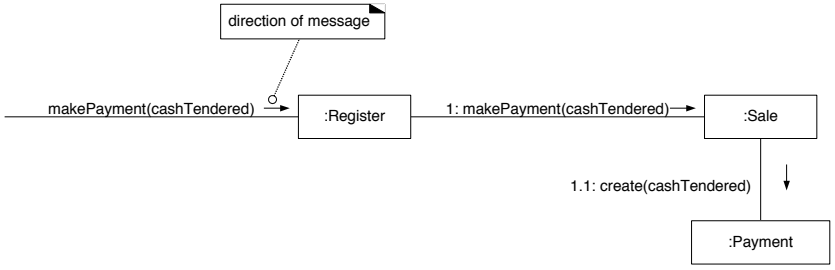
Communication Diagram



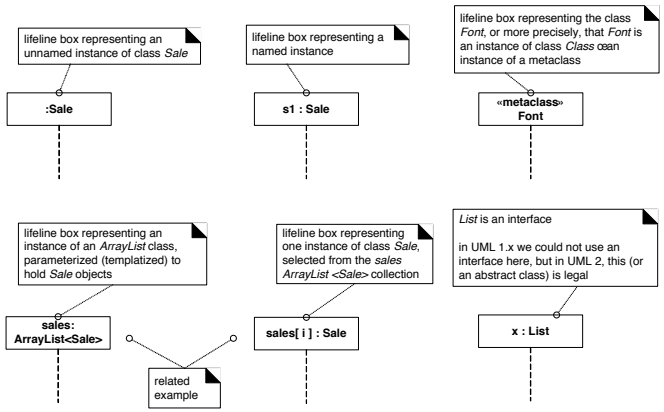
Example Sequence Diagram: makePayment



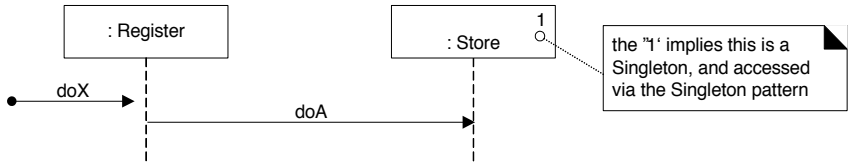
Communication diagram



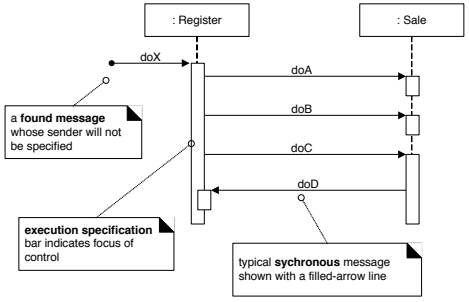
Life boxes to show participants in interactions.



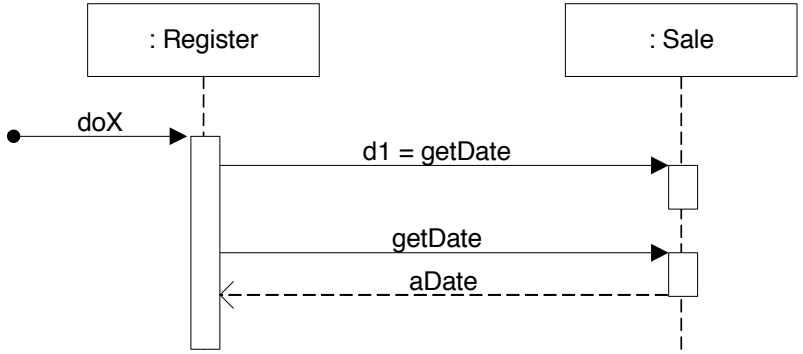
Singleton in interaction diagrams



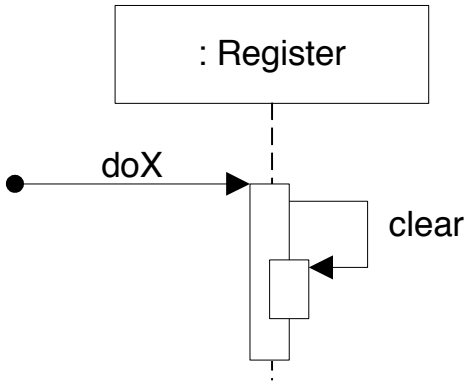
Messages and focus of control with execution specific bar



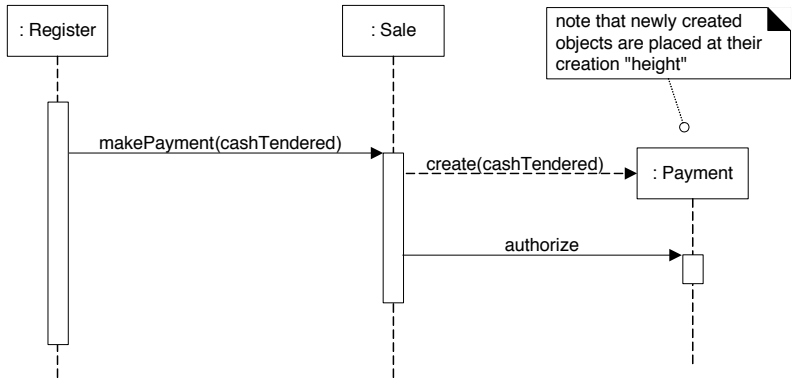
Two ways of showing return message



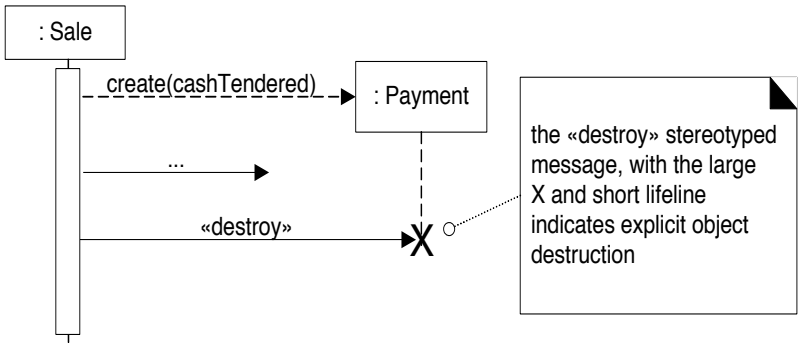
Message to “this”



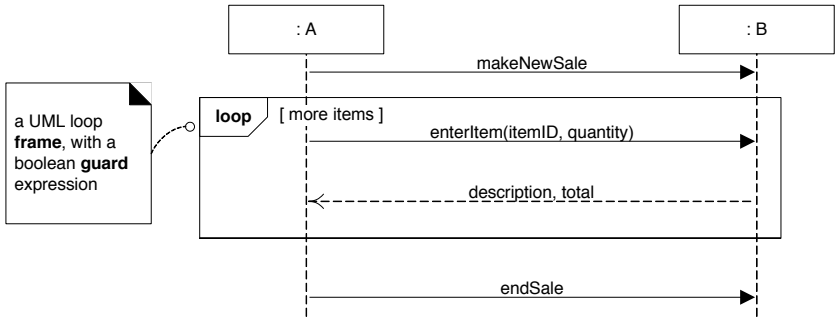
Interction creation and object lifelines



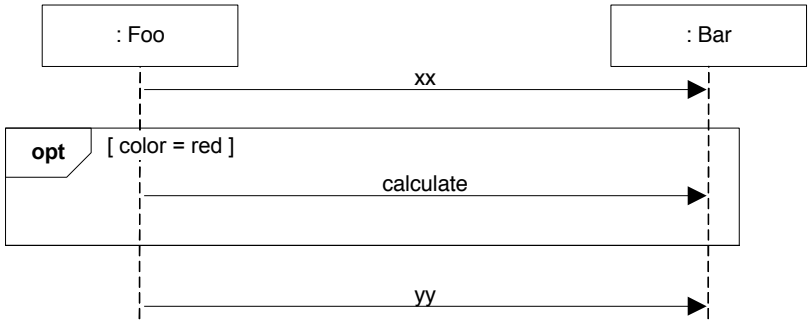
Object destruction



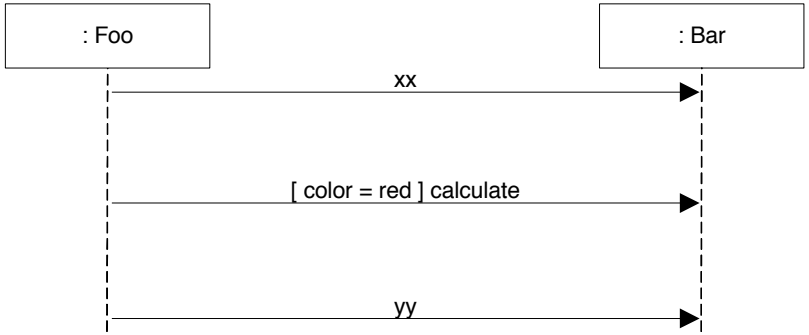
UML frame example



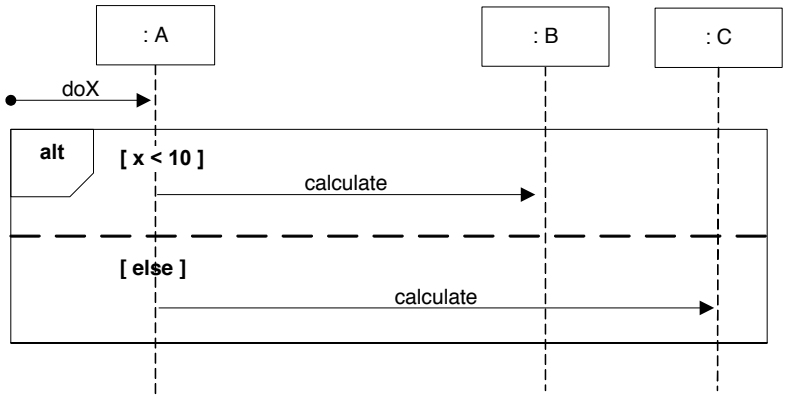
Condition message



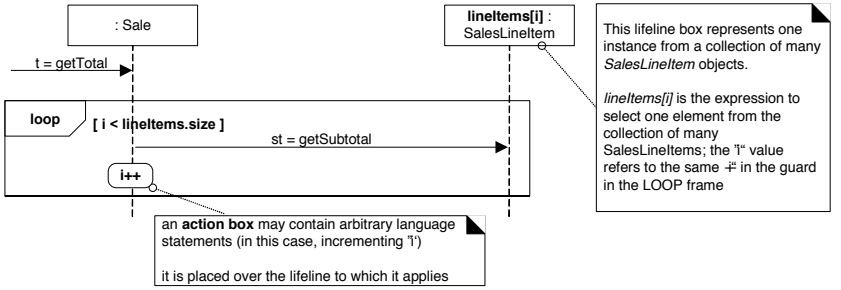
Alternate condition message notation (UML 1.x)



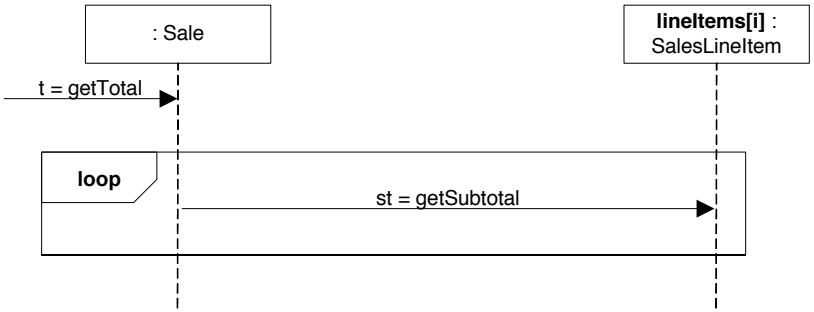
Mutually exclusive condition message



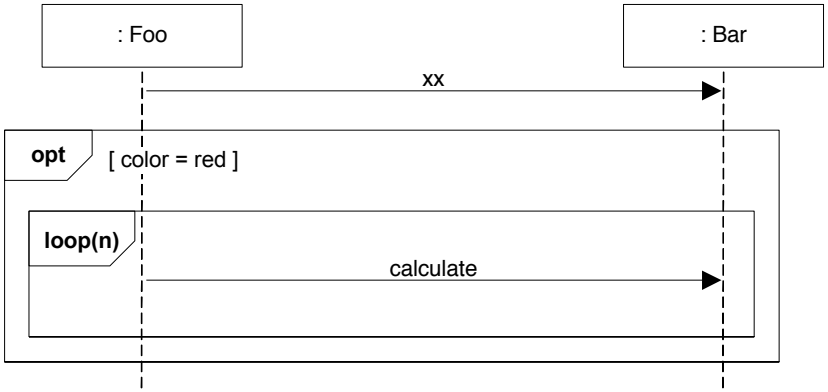
Iteration over a collection – relatively explicit notation



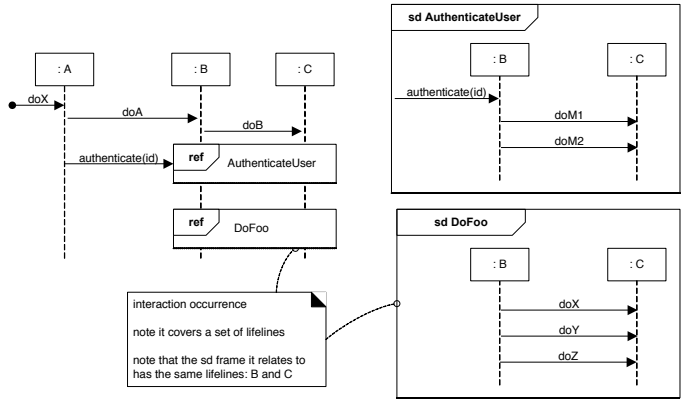
Iteration over a collection - implicit



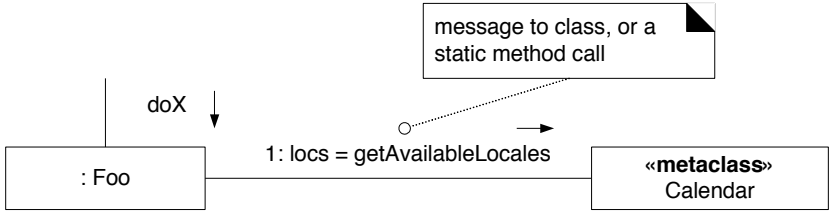
Nesting of frames



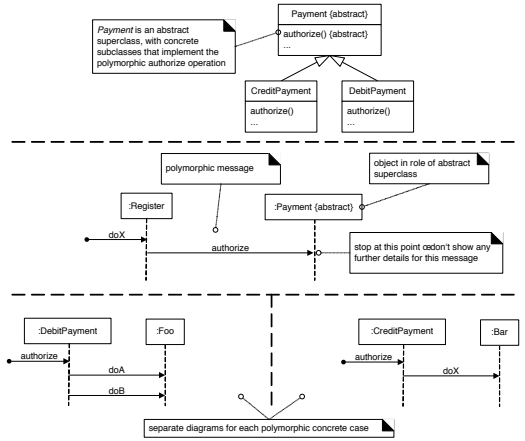
Sd and ref frames: interaction



Invoking class or static method; showing a class object as an instance of a metaclass



Modeling polymorphic cases in sequence diagrams



Asynchronous calls and active objects

a stick arrow in UML implies an asynchronous call

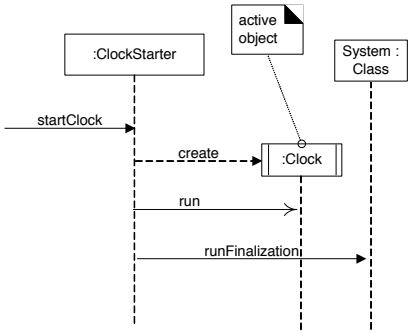
a filled arrow is the more common synchronous call

In Java, for example, an asynchronous call may occur as follows:

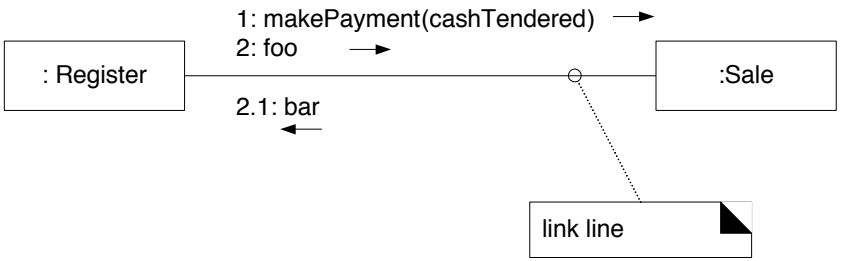
```
// Clock implements the Runnable interface
Thread t = new Thread( new Clock() );
t.start();
```

the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

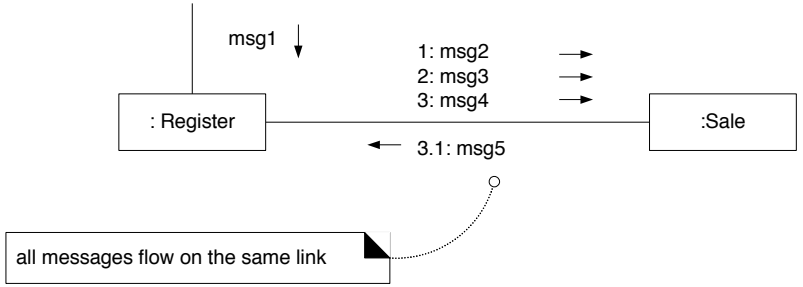
to simplify the UML diagram, the *Thread* object and the *start* message may be avoided (they are standard -overhead"); instead, the essential detail of the *Clock* creation and the *run* message imply the asynchronous call



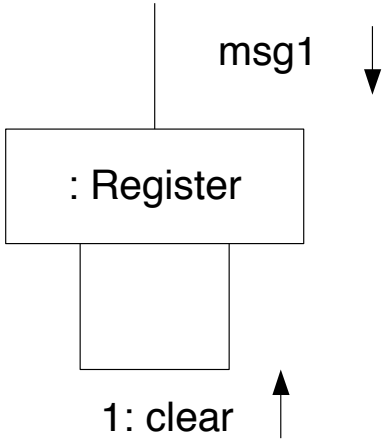
Link Lines



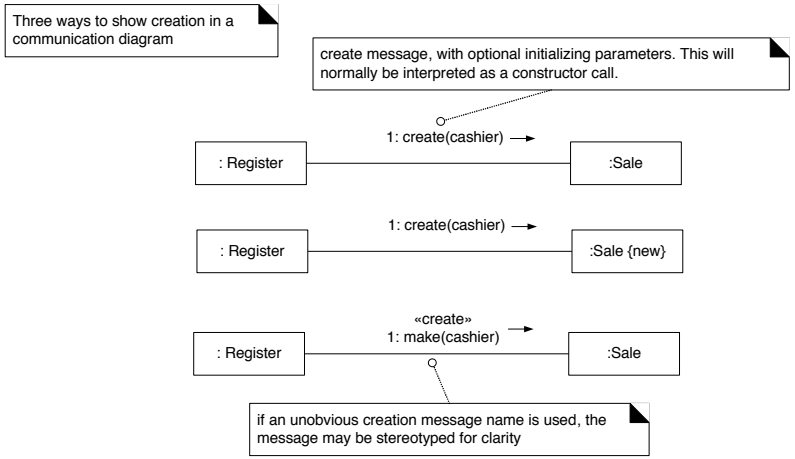
Messages



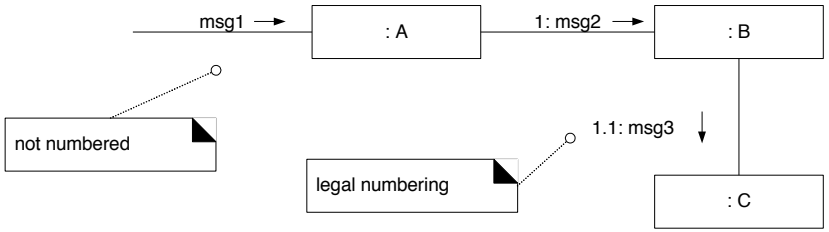
Messages to “this”



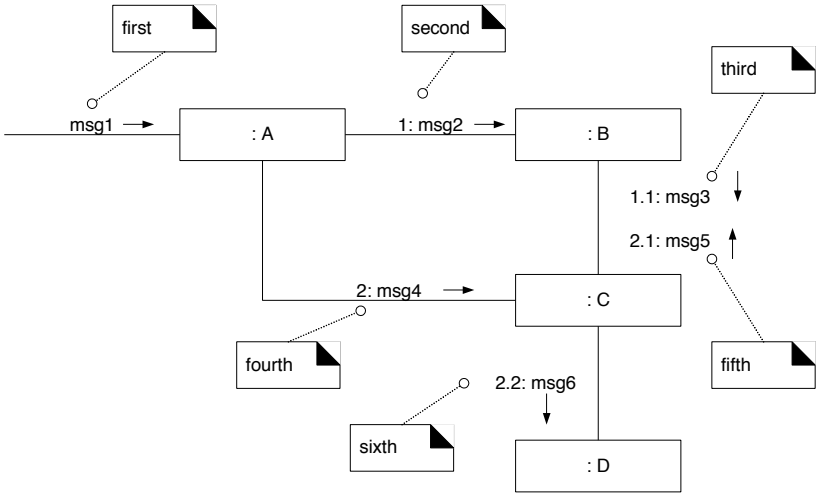
Instance Creation



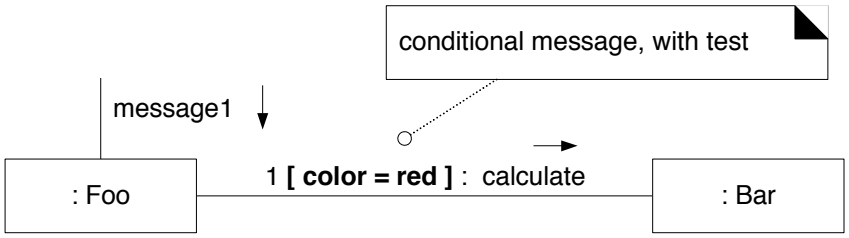
Sequence Numbering



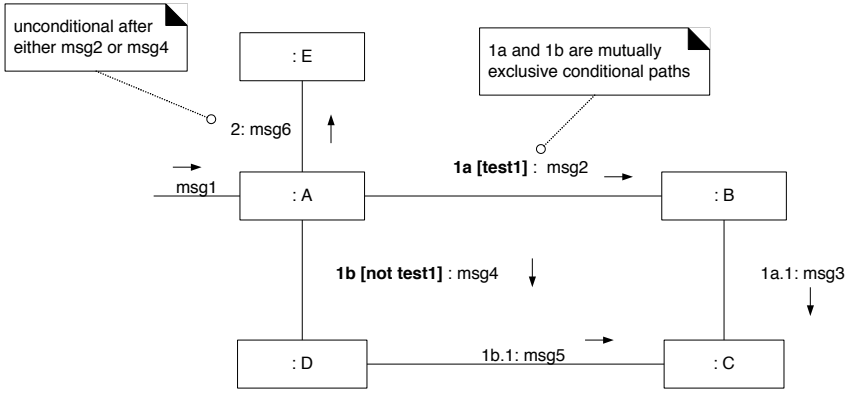
Complex Sequence Numbering



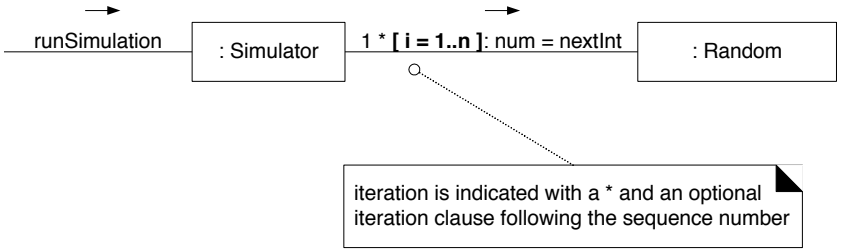
Conditional Message



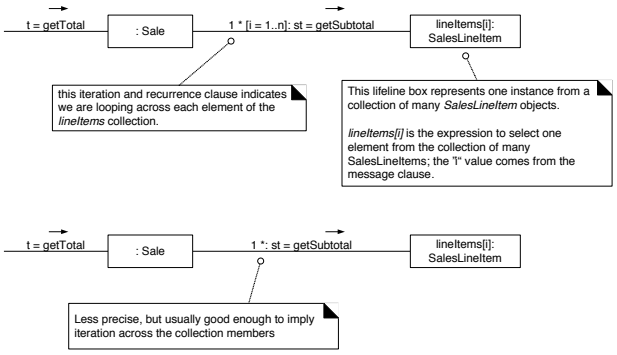
Utually Exclusive Messages



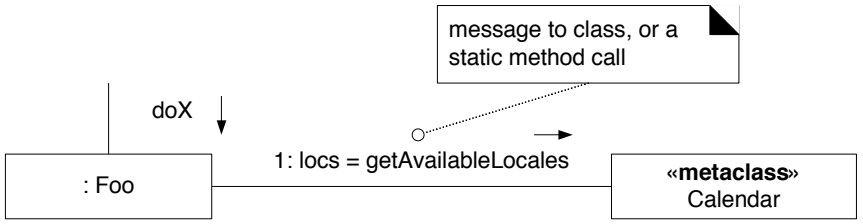
Iteration



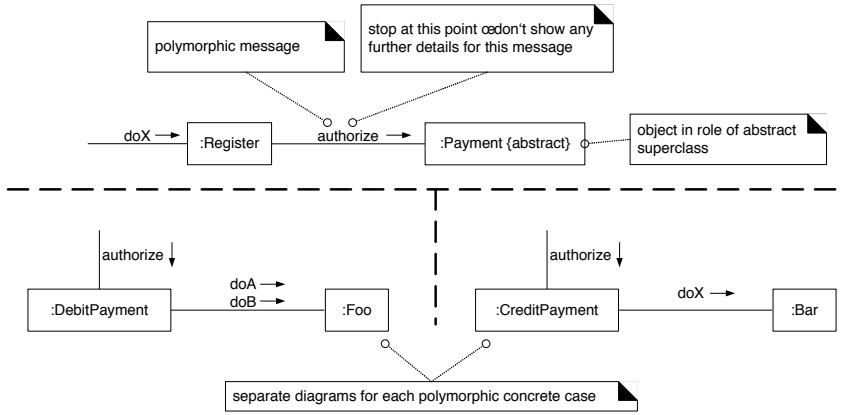
Iteration over a collection



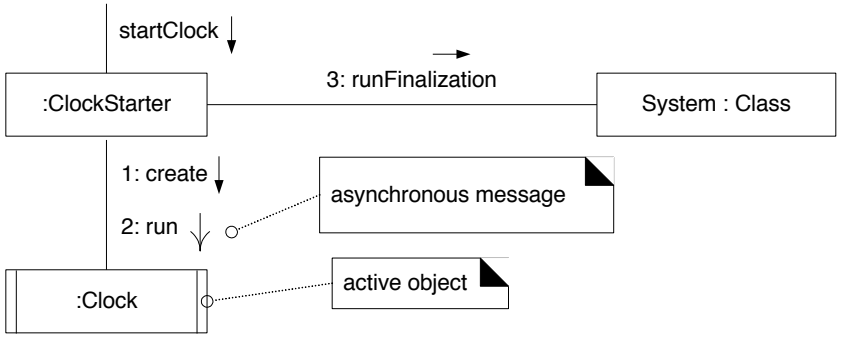
Messages to a class object (static method invocation)



Modeling Polymorphic cases in communication diagrams

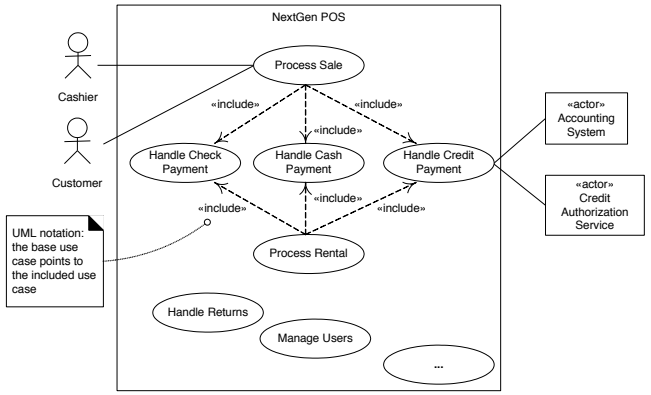


Asynchronous call in a communication diagram

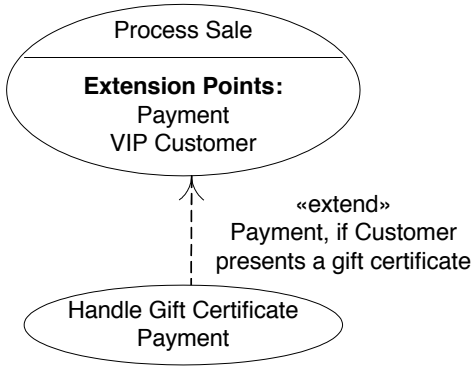


Relating Use Cases

Use case include relationship in the Use-Case Model

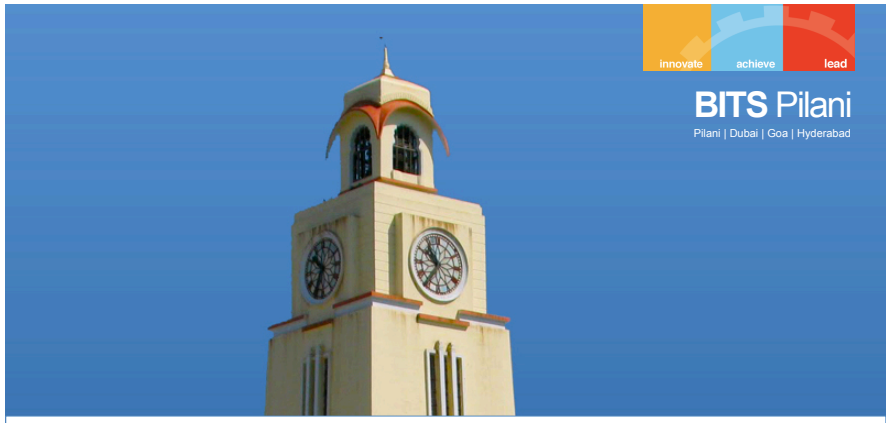


The extend relationship



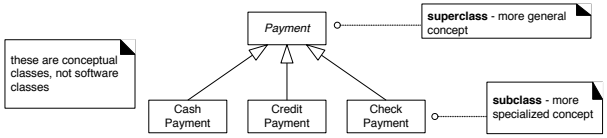
UML notation:
1. The extending use case points to the base use case.

2. The condition and extension point can be shown on the line.

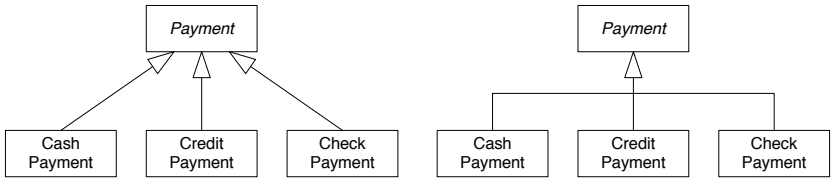


Domain Model Refinement

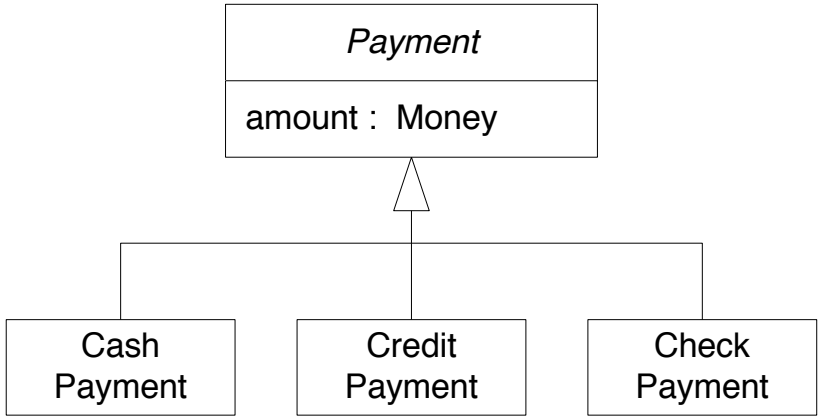
Generalisation Specialisation Hierarchy



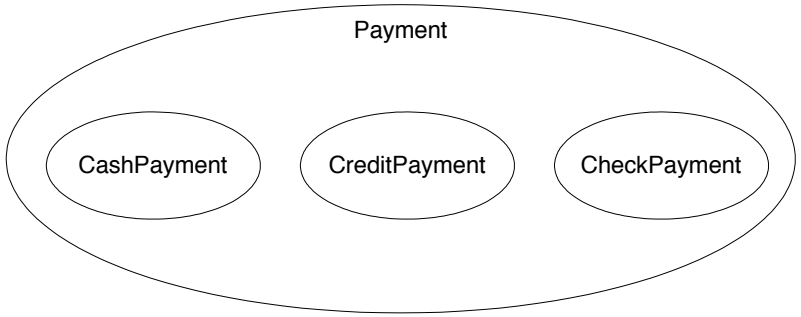
Class Hierarchy – separate and shared arrow notation.



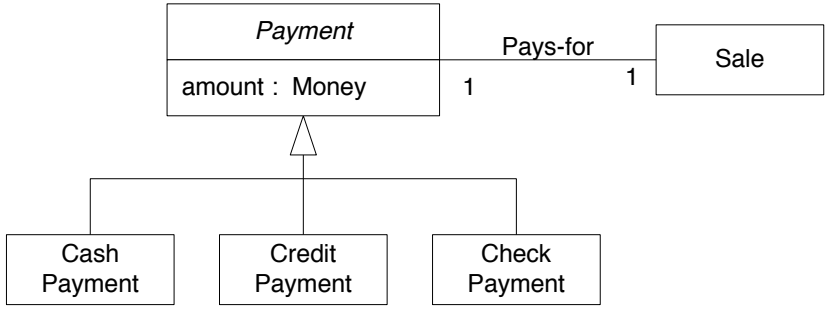
Payment Class Hierarchy



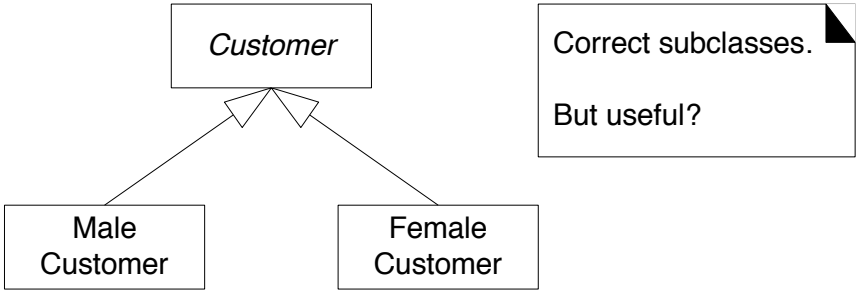
Set Relationship: venn diagram



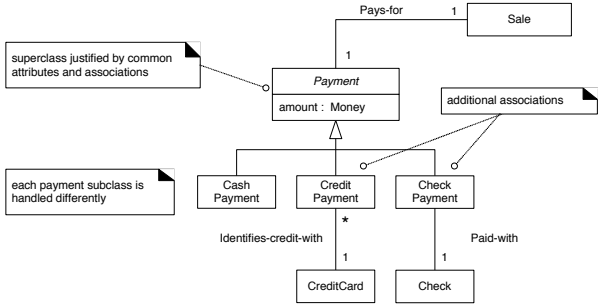
Subclass conformance



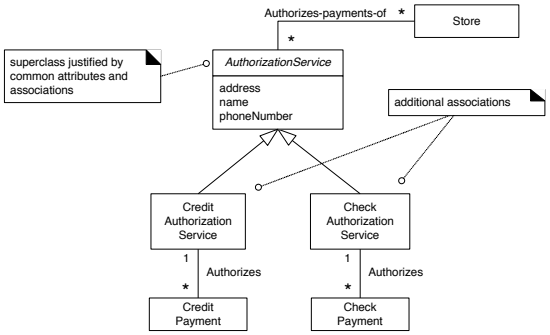
Legal conceptual class partition



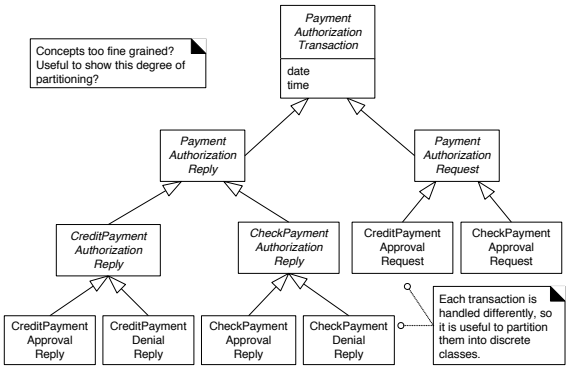
Justifying Payment subclasses



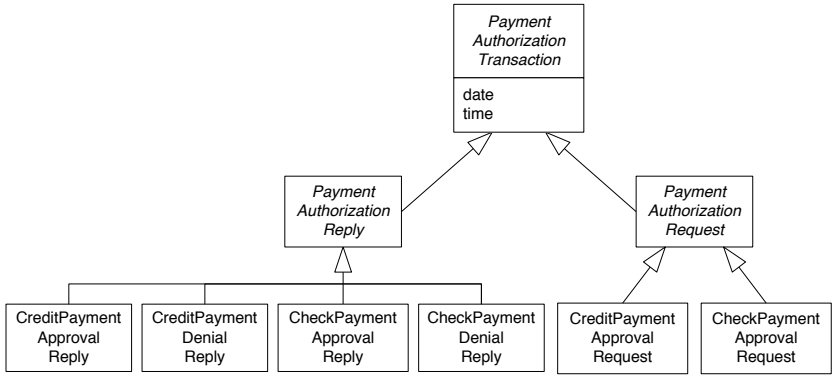
Justifying the AuthorizationService hierarchy



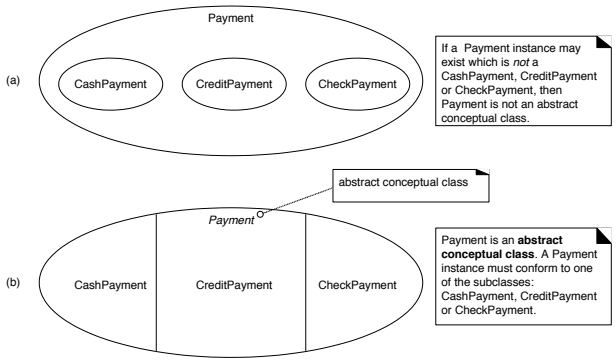
Possible hierarchy for external service transaction



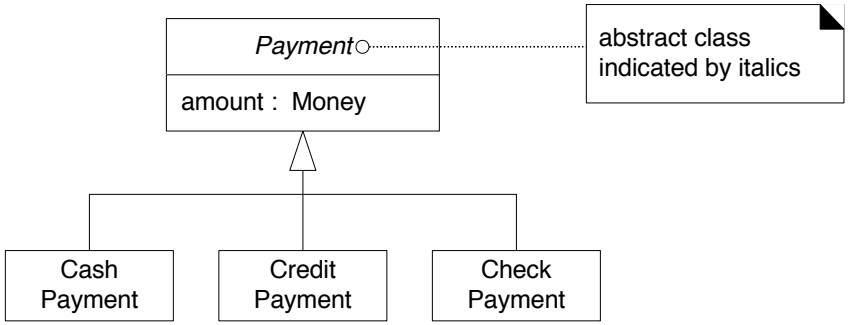
Alternate transaction class hierarchy



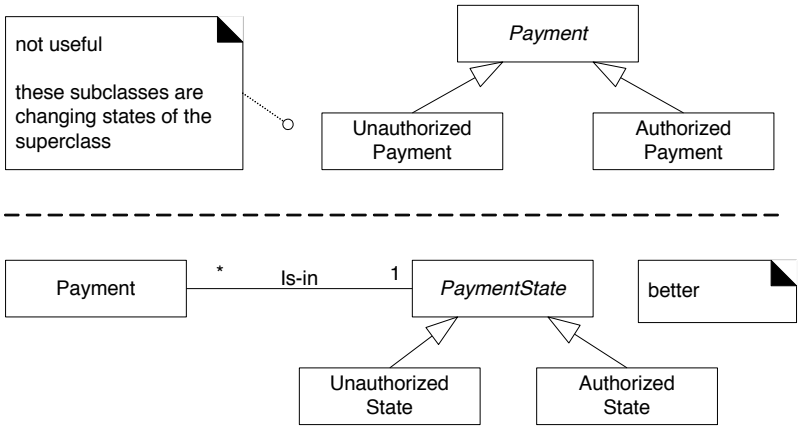
Abstract conceptual class



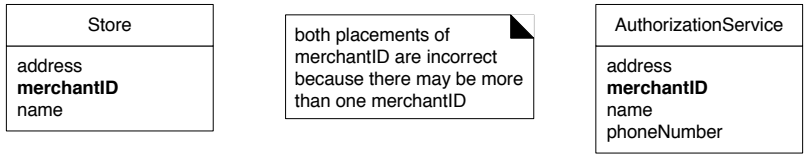
Abstract class notation



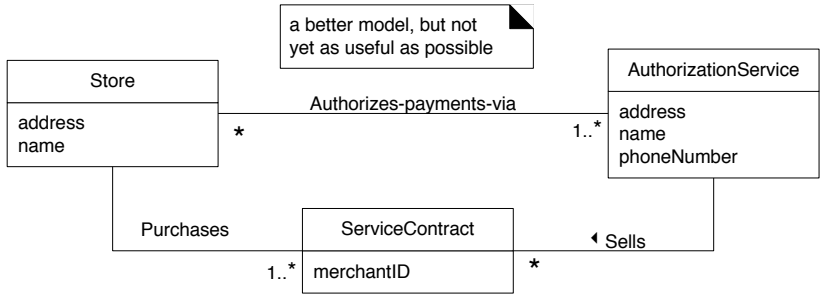
Modeling state changes



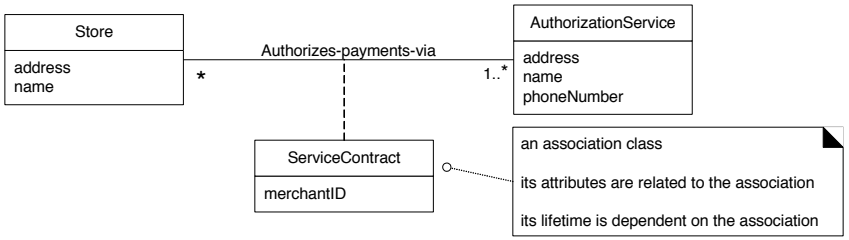
Inappropriate use of an attribute



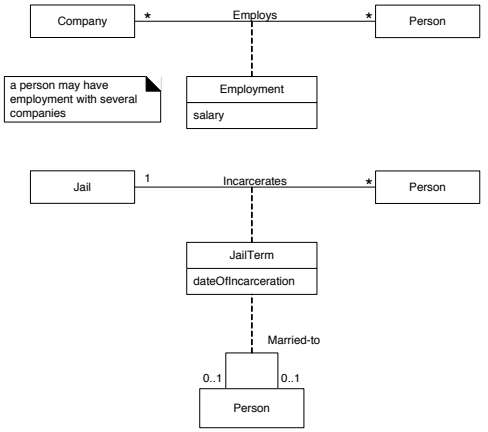
1st Attempt at modeling merchantID problem



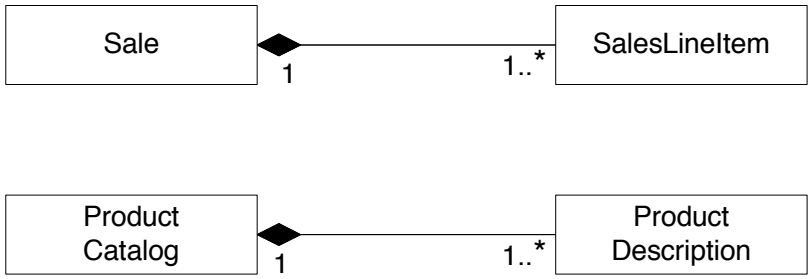
An association class



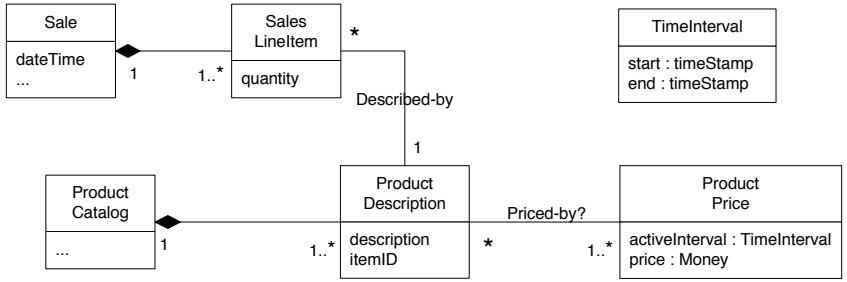
Association classes



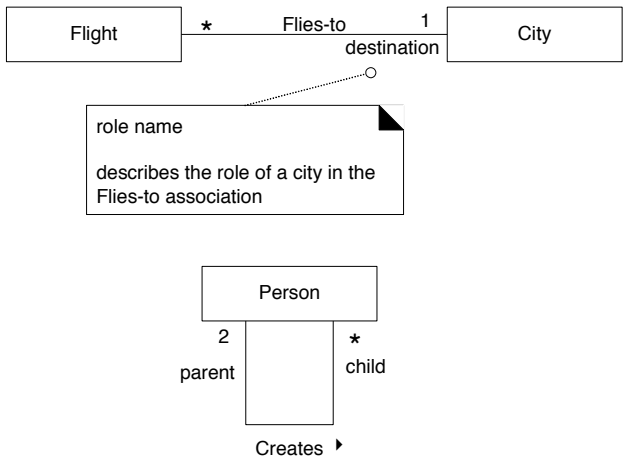
Aggregation in POS application



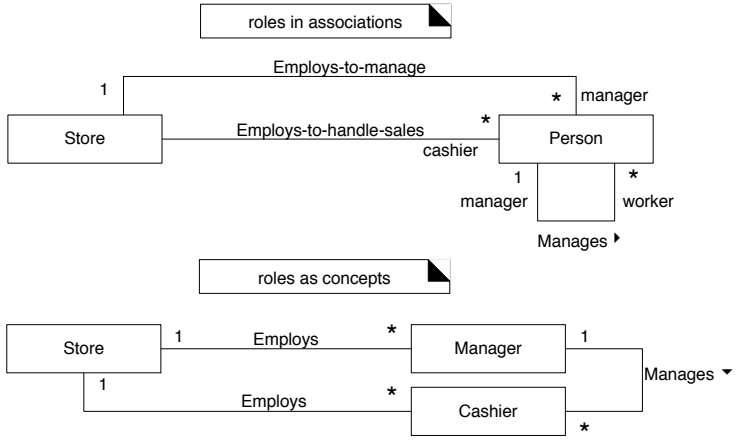
ProductPrice and time intervals



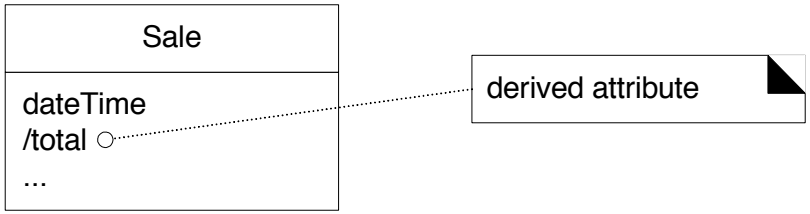
Role names



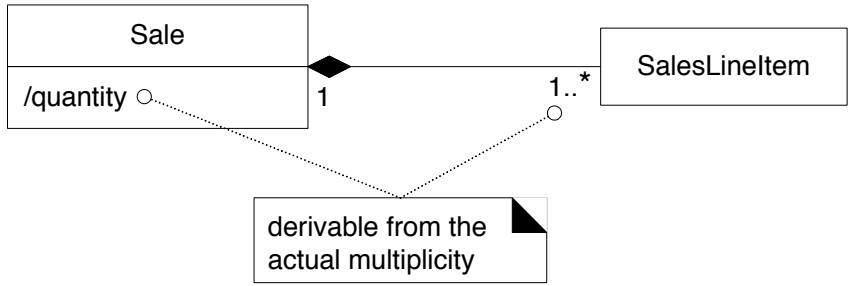
2 ways to model human roles



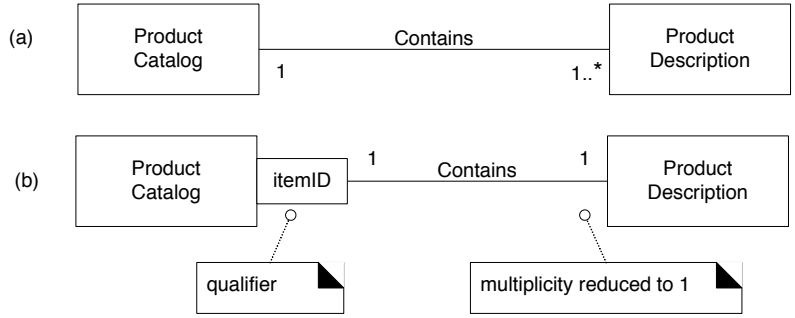
Derived attributes



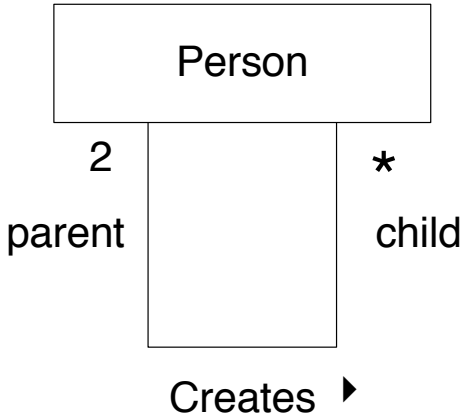
Derived attribute related to multiplicity



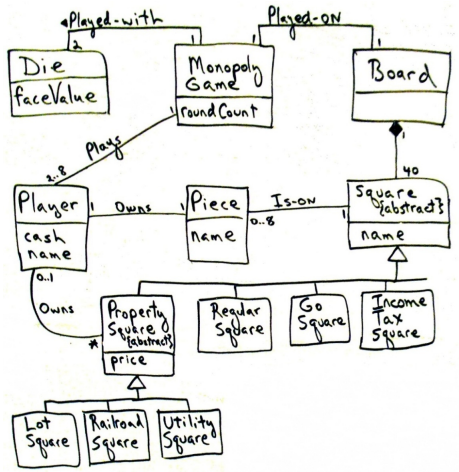
Qualified association

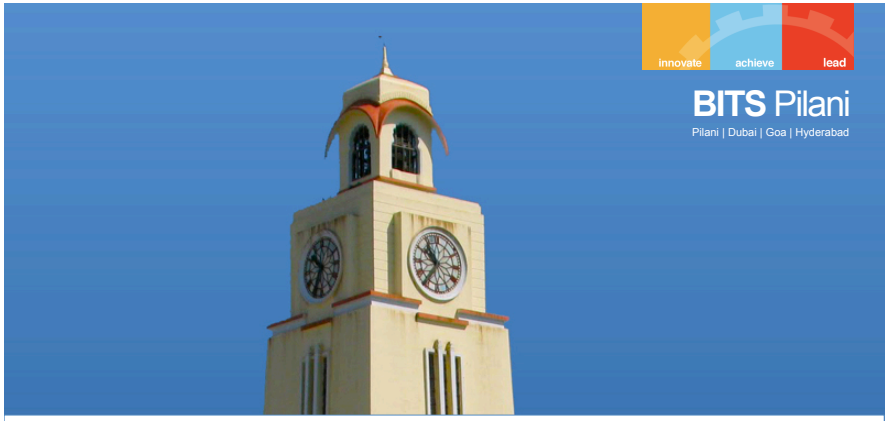


Reflexive association



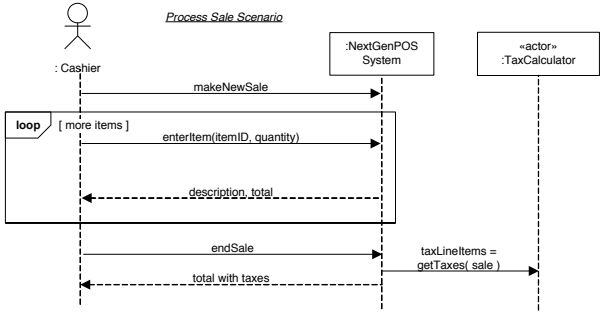
Iteration 3 Monopoly domain model



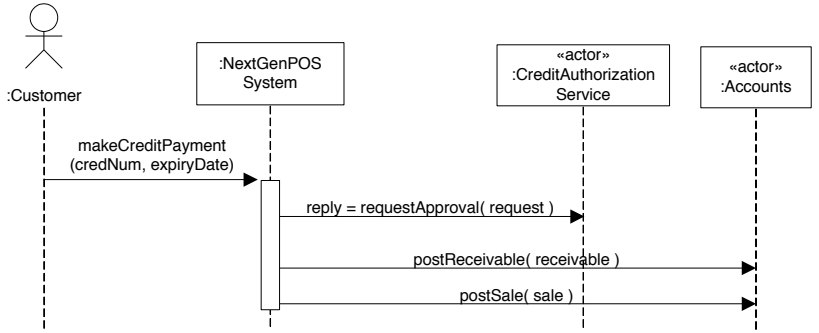


More SSDs and Contracts

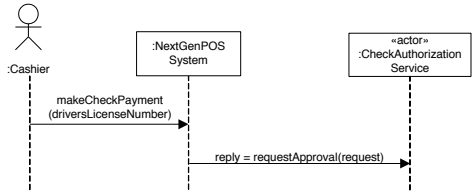
SSD common beginning



Credit Payment SSD



Cheque payment SSD



New System Operation Contract

Operation:

Cross Reference:

Preconditions:

Postconditions:

See page 538/539 of text

Thank You

The slides are based on:

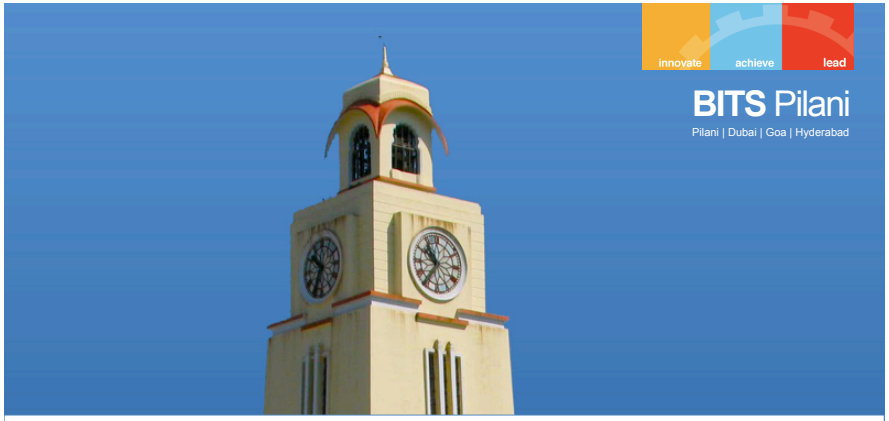
Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

No	Title
CS4.2.1	Explain how an Interaction Diagram plays an important role in Blueprint of the software? Types and syntax of Interaction Diagrams?
CS4.2.2	Demonstrate drawing Sequence Diagram, Collaboration Diagram for any one scenario of PoS System
CS4.2.3	State Transition Diagram Syntax & Activity Diagram Syntax. What is different between them?
CS4.2.4	Demonstrate drawing State Chart Diagram, Activity Diagram for PoS System.

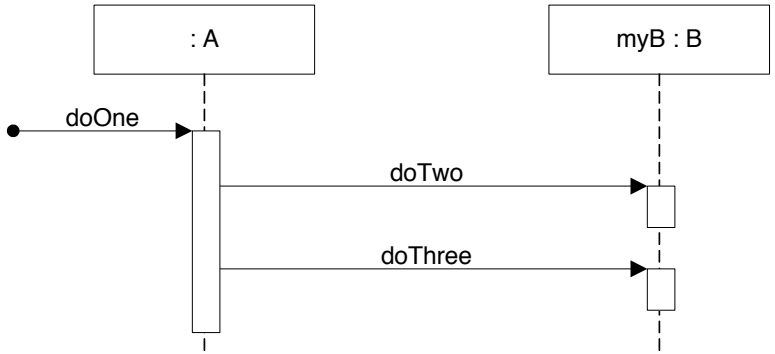


Interaction Diagrams

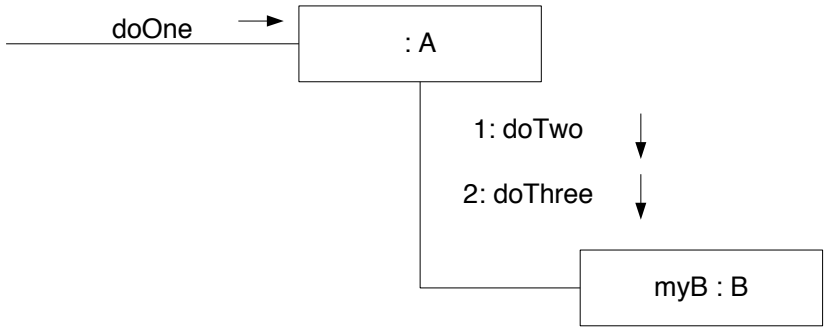
Interaction Diagrams

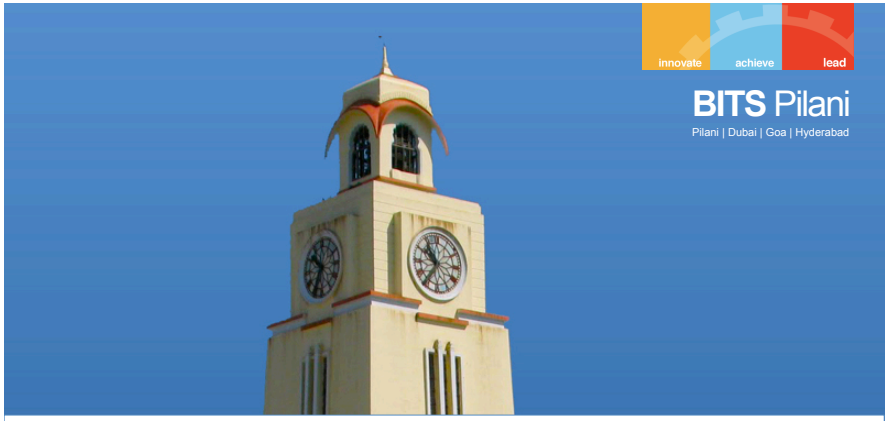
- Explain how an Interaction Diagram plays an important role in Blueprint of the software? Types and syntax of Interaction Diagrams?
- Sequence Diagrams
- Collaboration/Communication Diagrams

Sequence Diagrams



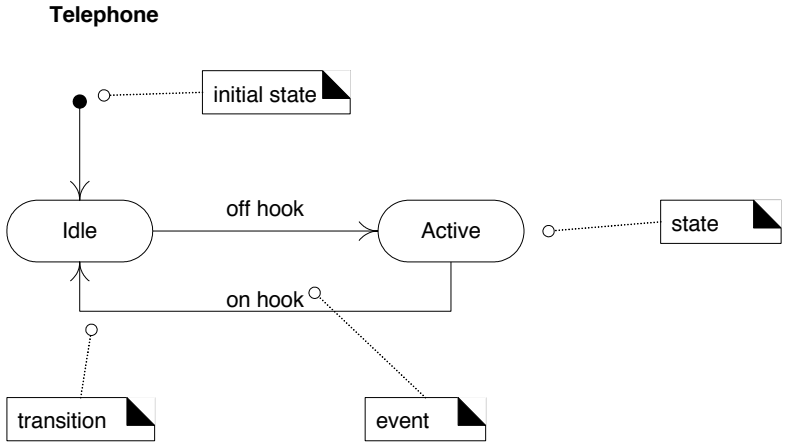
Collaboration Diagrams





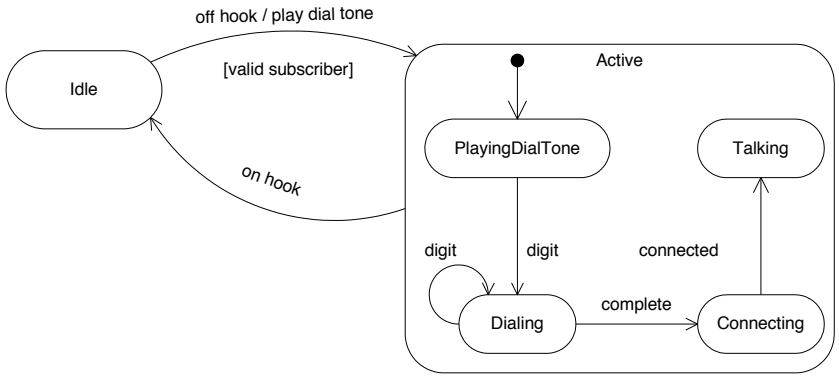
State Transition Diagrams

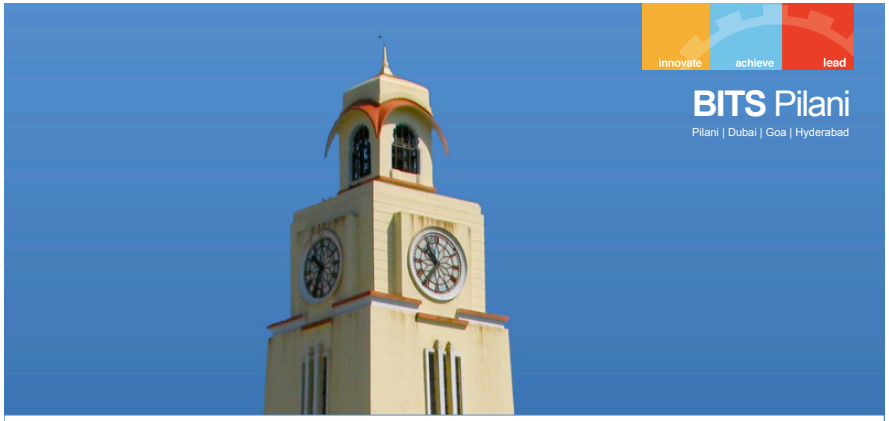
State Transition Diagrams



State Chart Diagrams

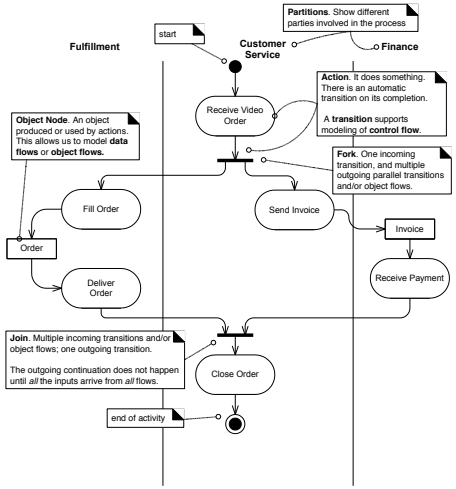
State Chart Diagrams



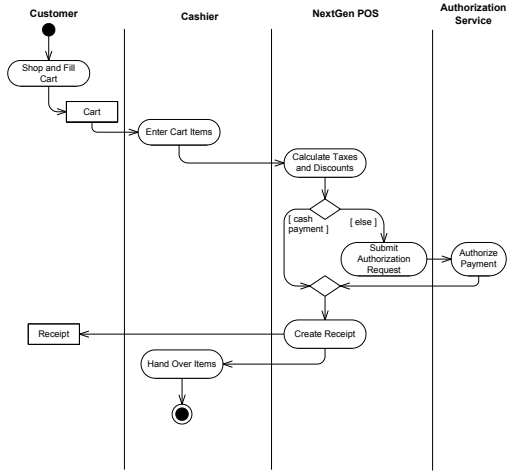


Activity Diagrams

Activity Diagrams



PoS



Thank You

The slides are based on:

Applying UML and Patterns

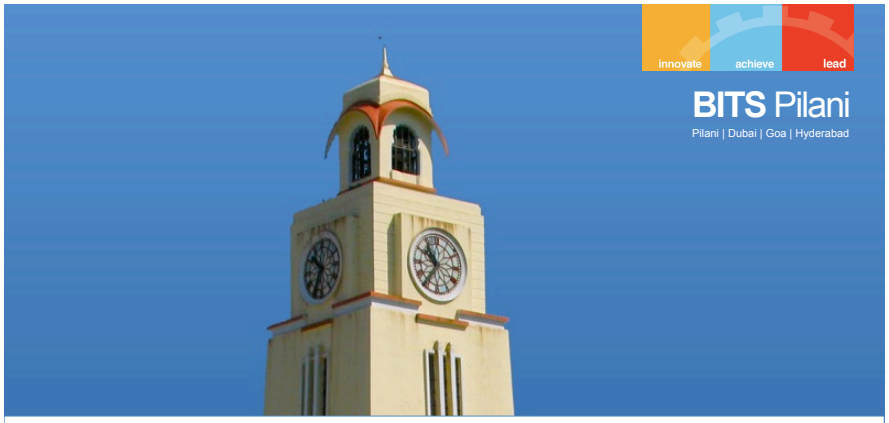
An Introduction to Object-Oriented Analysis and Design and Iterative Development

Explain how an Interaction Diagram plays an important role in Blueprint of the software?
Types and syntax of Interaction Diagrams?

By Craig Larman

Outline

No	Title
CS5.1.1	Concept & Significance of Visibility among Objects
CS5.1.2	Types of Visibility with source code example
CS1.1.3	Transition from Domain Model to Class Diagram



Concept & Significance of Visibility among Objects

Concept & Significance of Visibility among Objects

- Refer RL 5.1.1 to 5.1.5
- There are 4 common ways of achieving visibility
  - Attribute Visibility
  - Parameter Visibility
  - Local Visibility
  - Global Visibility

Visibility Between Objects

For Systems Operations to take place messages need to pass between objects.

- Sender Object Sends Message
- To a Received Object.

The UML provides four abbreviations for visibility:
- + (public),
- - (private),
- ~ (package), and
- # (protected)

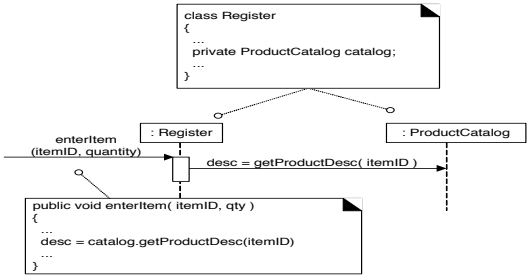
Sender must be visible to receiver.

Sender must have some sort of pointer or reference to Receiver.

Significance

Sender:
Register
Receiver:
ProductCatalogue

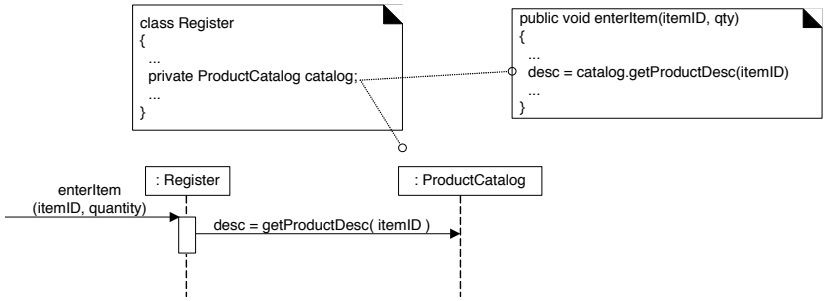
Message:
getProductDesc



Visibility from the Register to ProductCatalogue is required

ProductCatalogue must be visible to Register for interaction to take place.

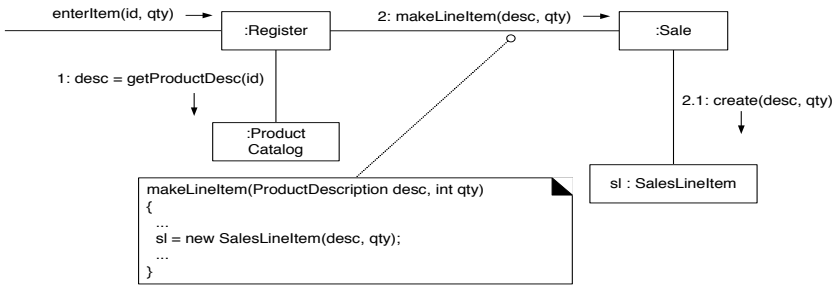
Attribute: Permanent Visibility Persists as long as A and B exist



A ProductCatalog object (catalog)
Is visible to a Register Object
Because catalog is an attribute of Register.
This is necessary as the message
getProductDesc is to be sent

Parameter: Temporary Visibility

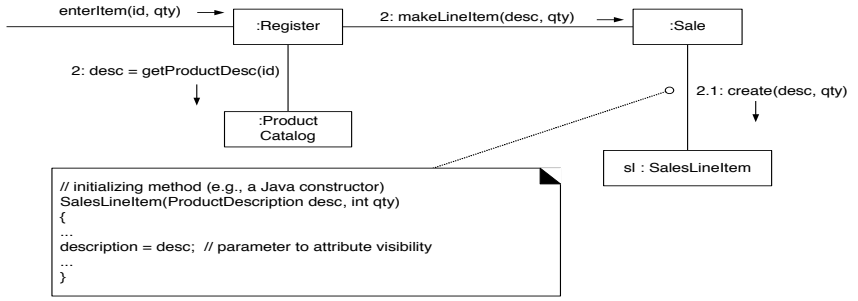
Persists only within the scope of the method



desc is an Object of Type ProductDescription and is passed as a parameter by a Register Object to method in an Object of Sale.

desc is visible to the Sale Object while the method is being executed.

Parameter to Attribute Visibility

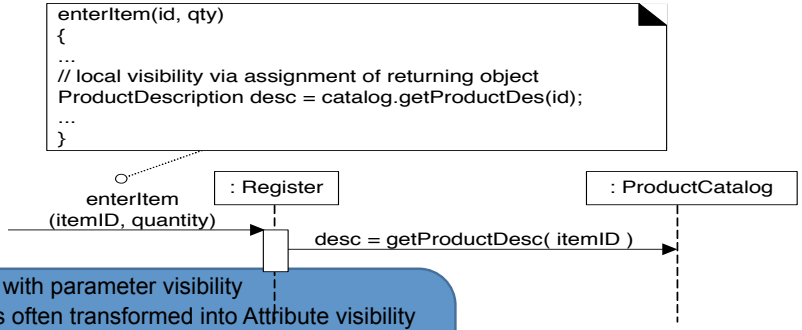


The Object received as a parameter may be used in a Constructor to assign the Object to an Attribute of the Class.

This establishes Attribute Visibility.

Local: Temporary Visibility

Persists only within the scope of the method



As with parameter visibility
It is often transformed into Attribute visibility

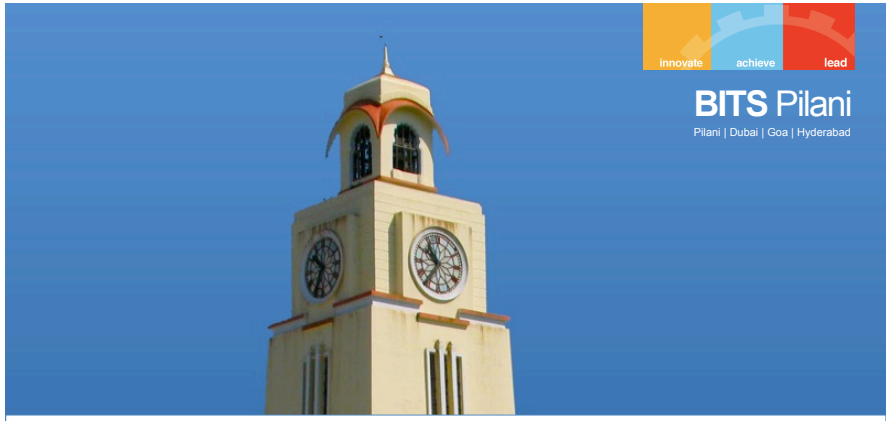
An interesting case is when an Object is Created as a result of a returning object from a method invocation:

anObject.getFoo().doBar();

Global Visibility

Object B will be visible to Object A
If
Object B has been instantiated as a globally visible object.

All globally visible Objects are visible to each Class in the environment to which they apply.



Types of Visibility with source code example

Types of Visibility with source code example - Public

```
public class MathUtil {

    public double cubedRoot(int num) {
        return Math.pow(num, 1.0/3);
    }

    public double circumference(double radius) {
        return 2*Math.PI*radius;
    }

    public double area(double radius) {
        return Math.PI *radius*radius;
    }

}
```

A variable or method that is public means that any class can access it. This is useful for when the variable should be accessible by your entire application. Usually common routines and variables that need to be shared everywhere are declared public.

NO PROTECTION.

Types of Visibility with source code example - Private

```
public class Dog extends Animal {

    private int numberOfLegs;
    private boolean hasOwner;

    public Dog() {
        numberOfLegs = 4;
        hasOwner = false;
    }

    private void bark() {
        System.out.println("Woof!");
    }

    public void move() {
        System.out.println("Running");
    }

}
```

Achieve Encapsulation:

- There is actually only one way a private method or variable can be accessed: within the class that defined them in the first place.

Types of Visibility with source code example – NOT ALLOWED

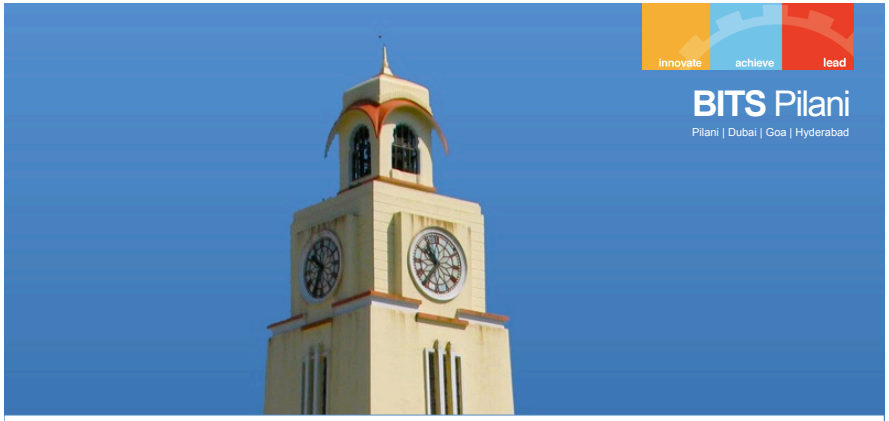
```
public class Cat extends Animal {

    public void makeDogBark() {
        Dog d = new Dog();
        d.bark();
    }

    public void meow() {
        System.out.println("Meow!");
    }

    public void move() {
        System.out.println("Prancing");
    }

}
```

- In that example, bark() and the variables numberOfLegs and hasOwner are private, which means only the Dog class has access to them.

Private access from within

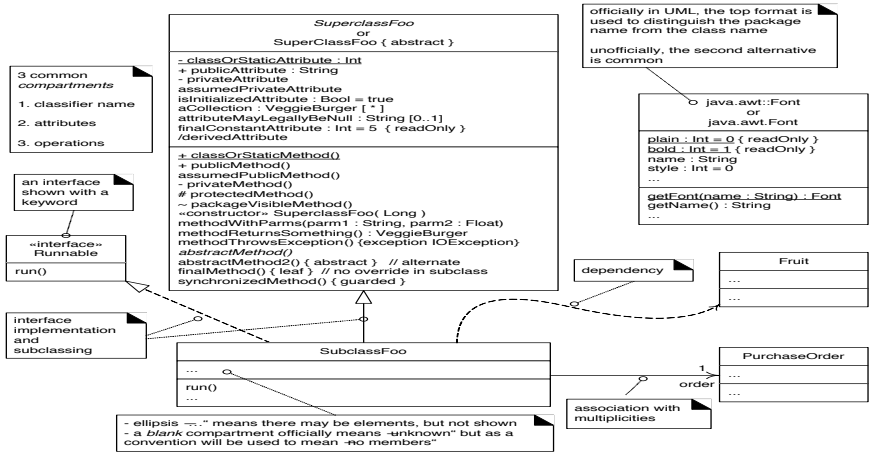
```
public class Dog extends Animal {

    private int numberOfLegs;
    private boolean hasOwner;

    public Dog() {
        numberOfLegs = 4;
        hasOwner = false;
    }

    public void makeDogBark() {
        Dog d = new Dog();
        d.bark();
    }

    private void bark() {
        System.out.println("Woof!");
    }

    public void move() {
        System.out.println("Running");
    }

}
```

- if you are inside of the dog class, you can call any dog's private methods and variables.
- That means you're allowed to access another dog's method from within the dog class.



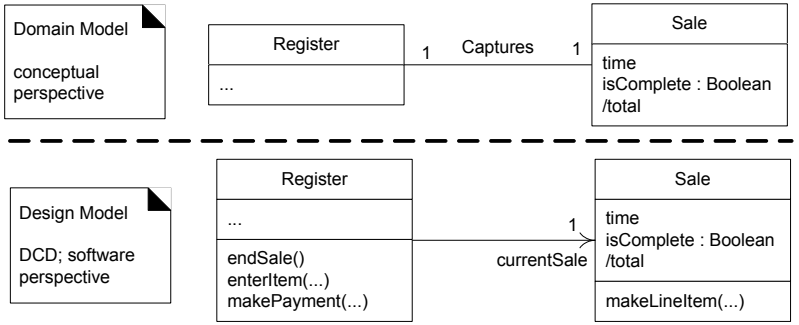
Transition from Domain Model to Class Diagram

UML Class Diagram Notations

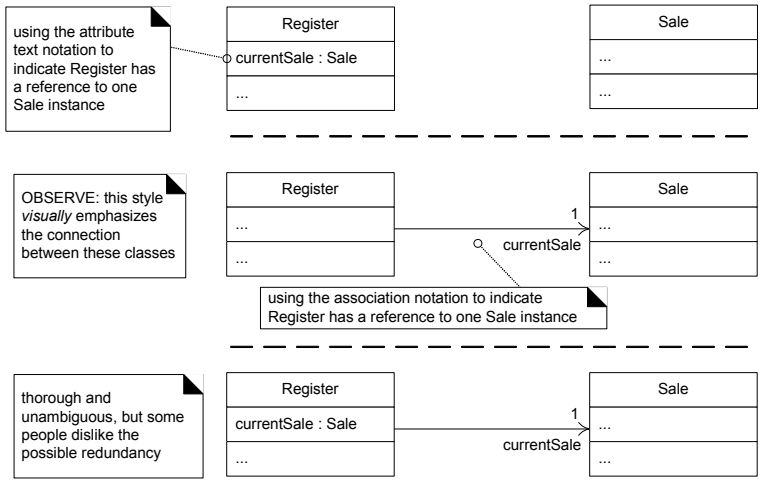


Domain and Design Class Diagram

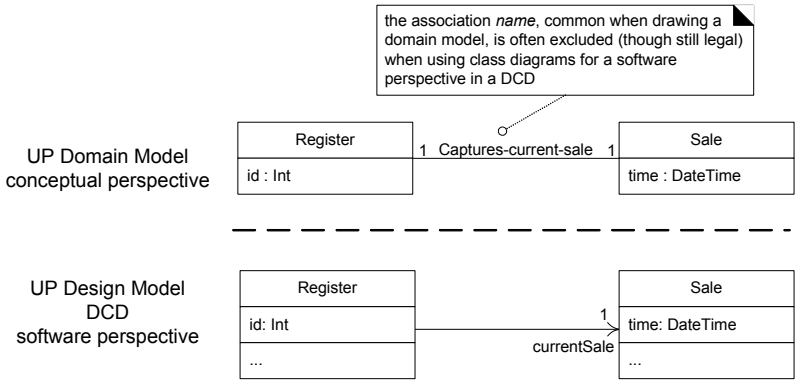
Different perspectives



Attribute Text vs line notation for UML attribute



Idioms in association notation usage



Thank You

The slides are based on:

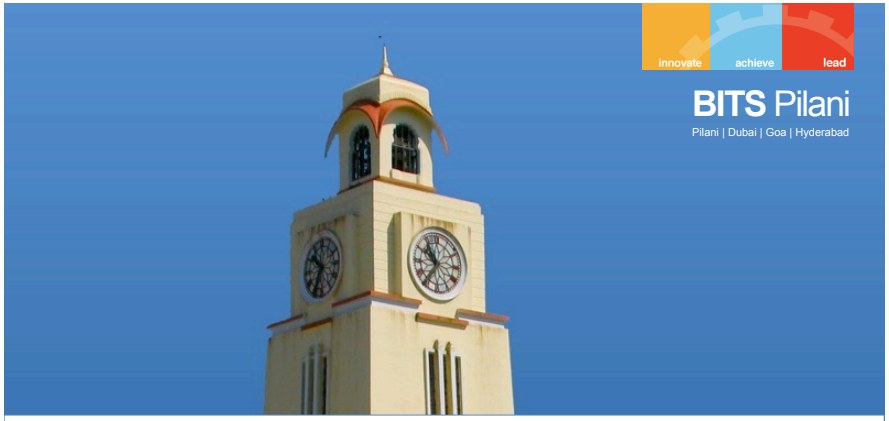
Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

No	Title
CS5.2.1	Drawing Class Diagram leveraging Domain Model drawn by an Analyst
CS5.2.2	Demonstrate drawing Class Diagram for PoS System, show all types of visibility in Class Diagram
CS1.2.3	Drawing package class diagram for PoS System

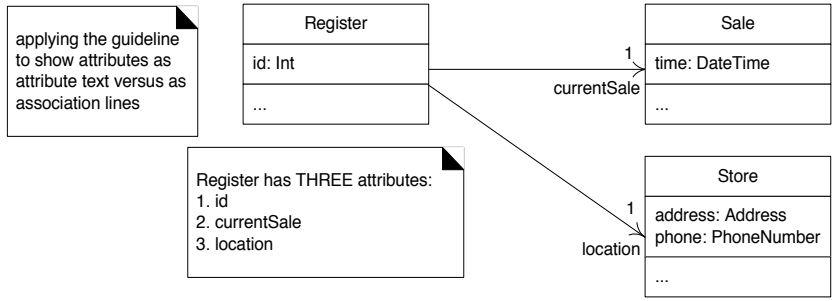


Drawing Class Diagram leveraging Domain Model drawn by an Analyst

Drawing Class Diagram leveraging Domain Model drawn by an Analyst

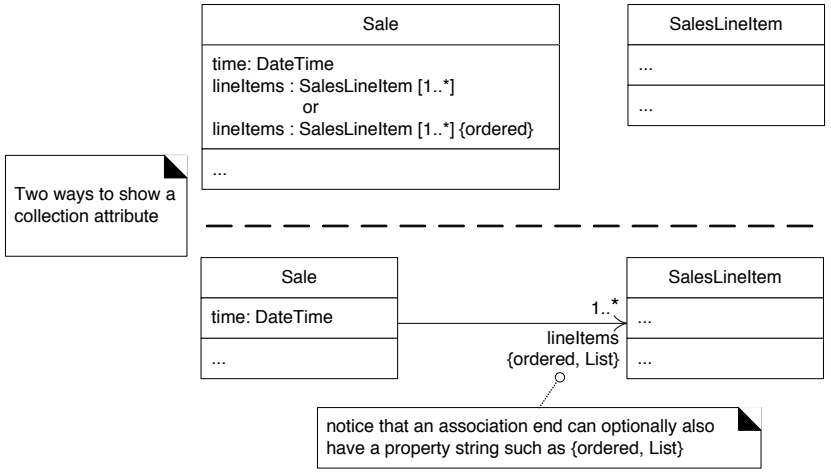
BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Showing attribute in two notations



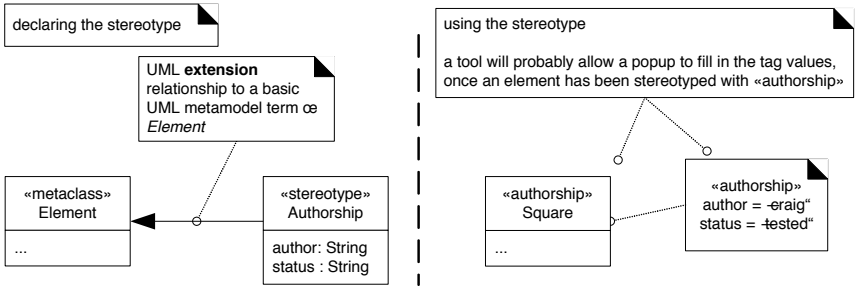
BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

2 ways to show a collection attribute in the UML

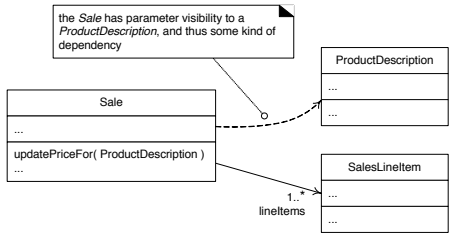


Showing a method body in class diagram

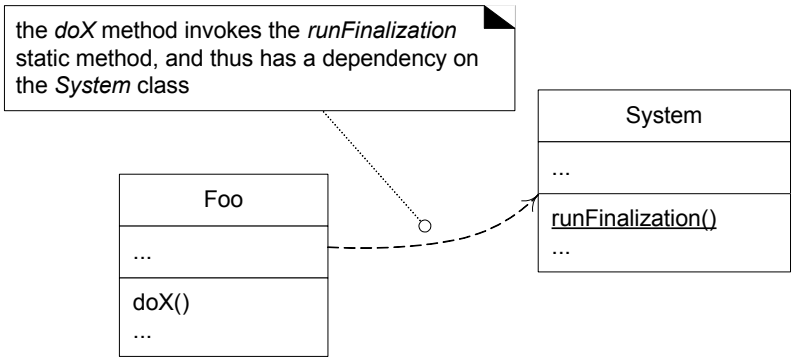
Stereotype declaration and use



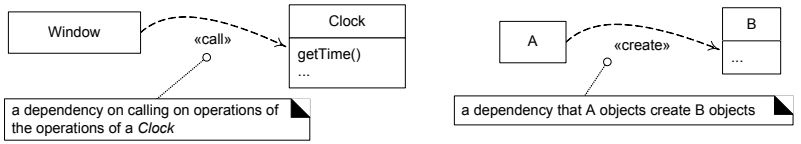
Showing Dependency



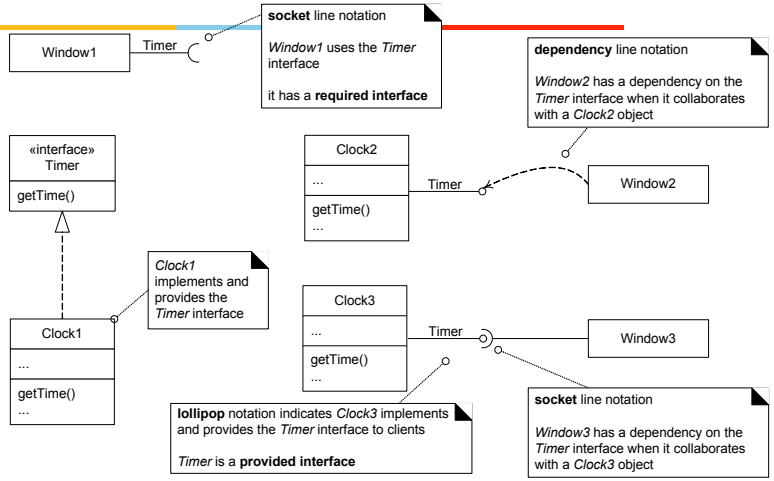
Showing Dependency



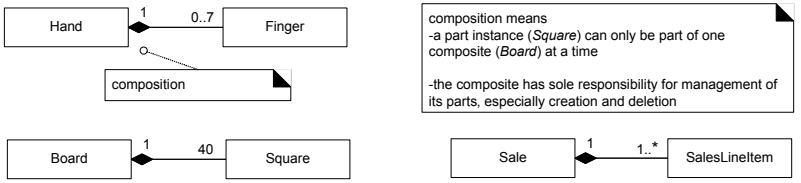
Optional dependency labels in UML



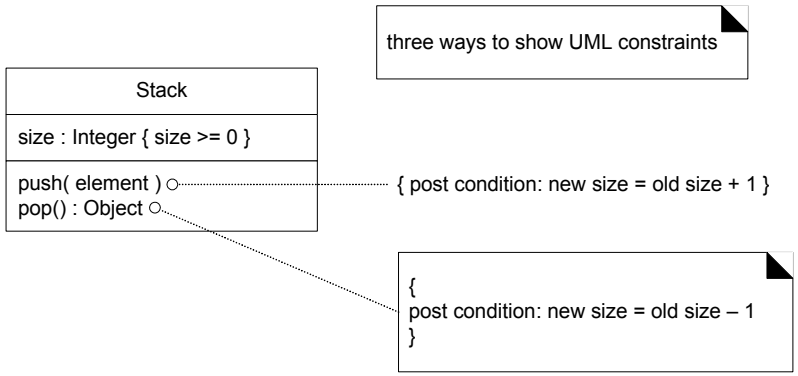
Interface notations



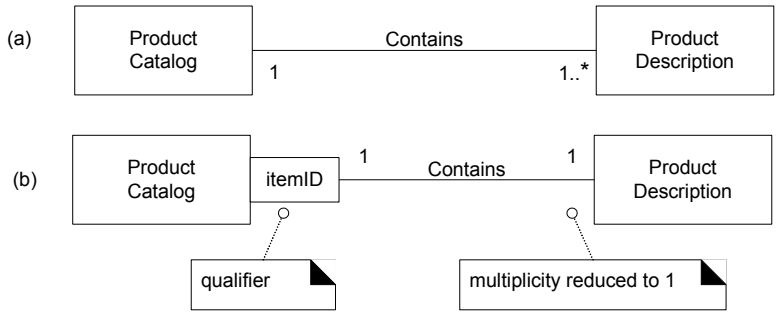
Composition in UML



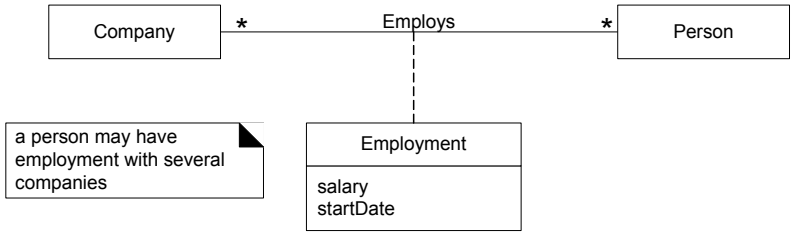
Constraints



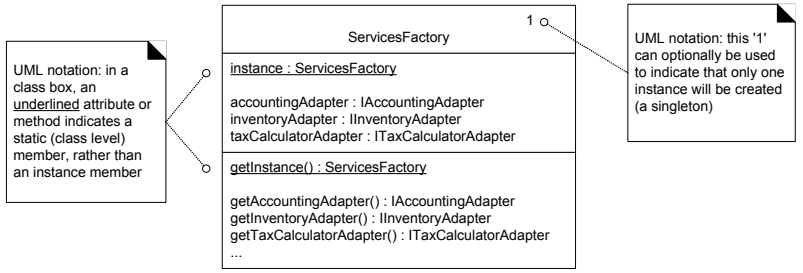
Quaified Associations in UML



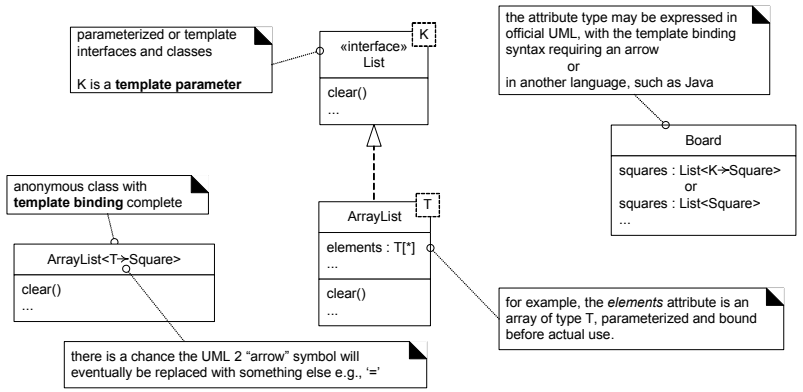
Association Classes



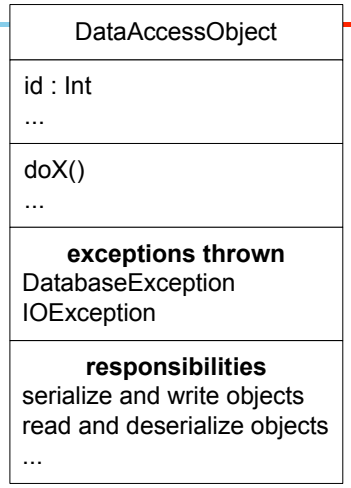
Showing a singleton



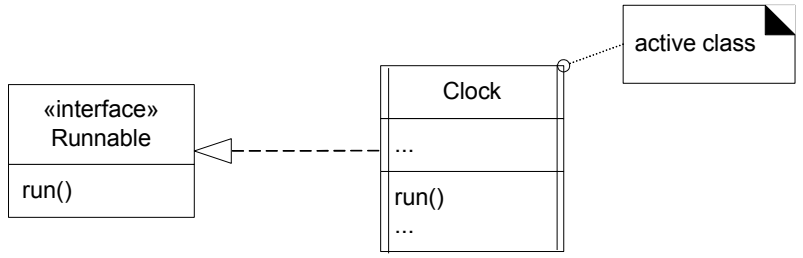
Template in the UML



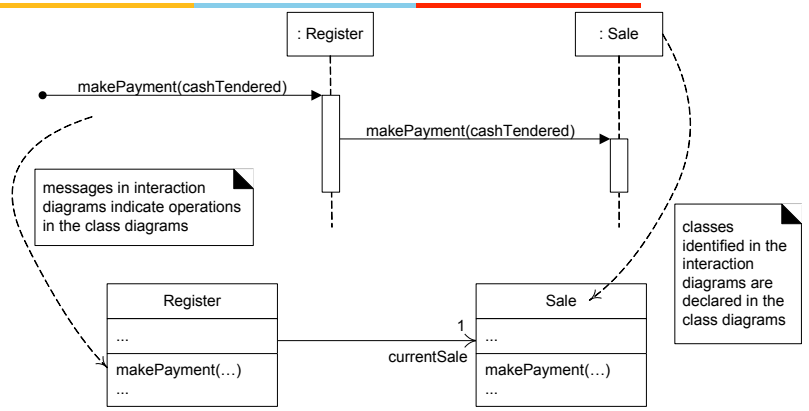
Compartments

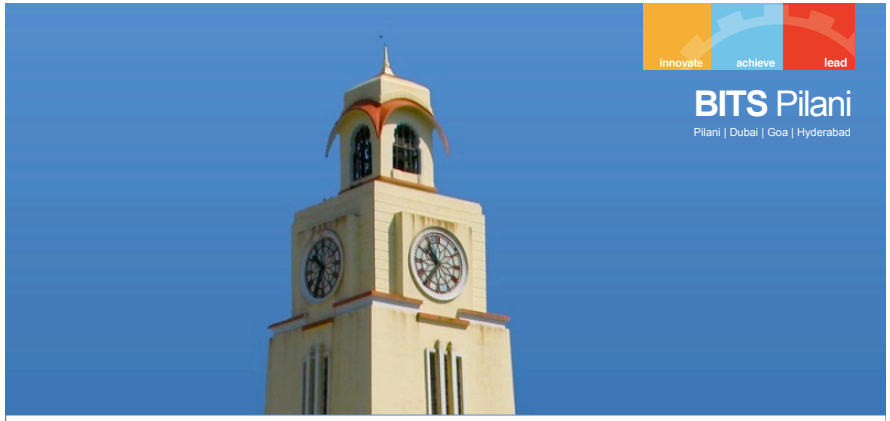


Active Classes in the UML



Influence of Interaction Diagrams on class Diagrams

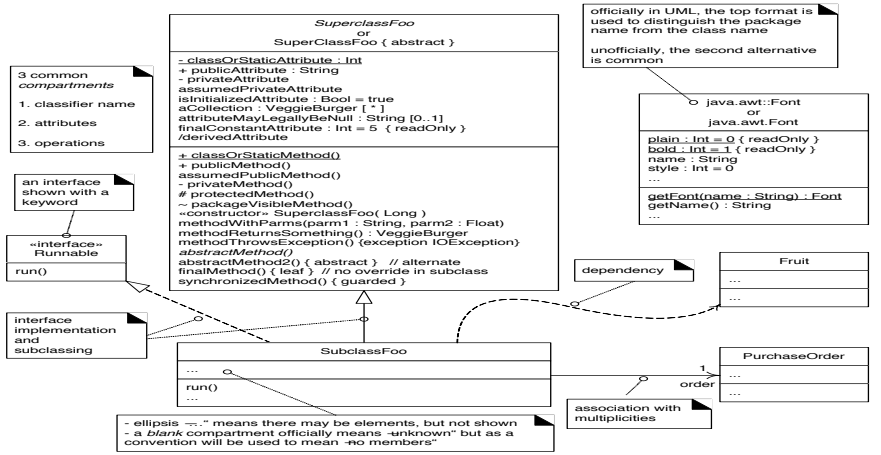


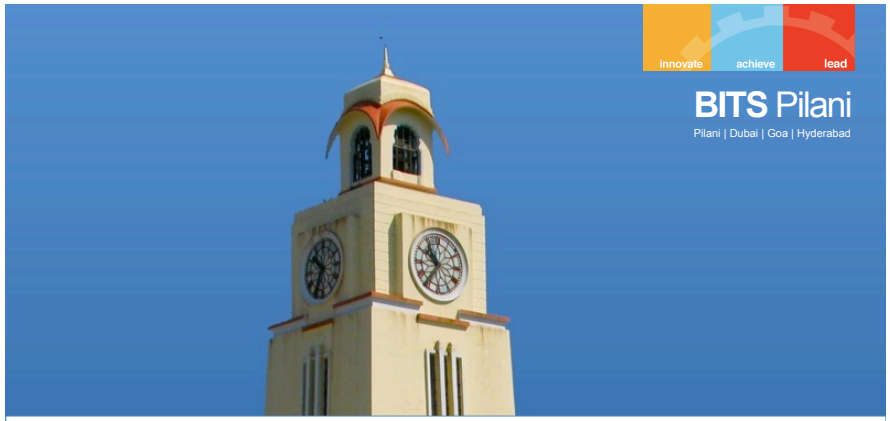


Demonstrate drawing Class Diagram for PoS System, show all types of visibility in Class Diagram

innovate achieve lead

Show all types of visibility in Class Diagram

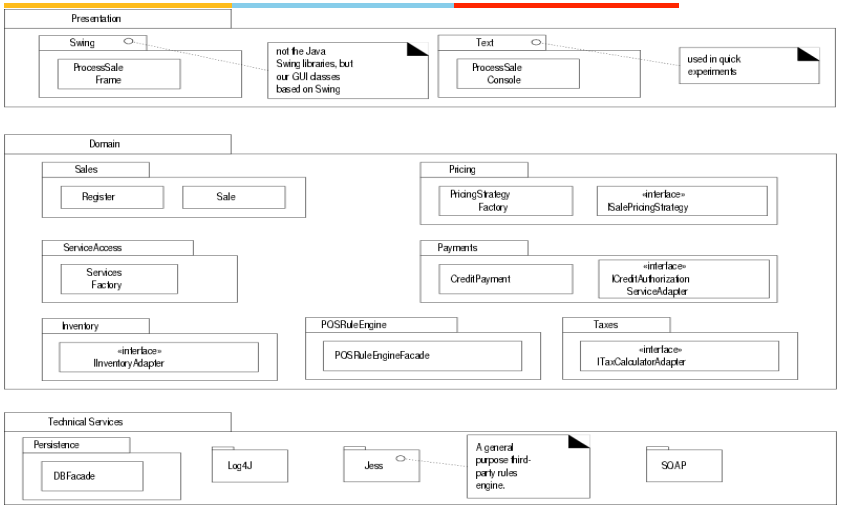




Drawing package class diagram for PoS System

innovate achieve lead

Drawing package class diagram for PoS System



Thank You

The slides are based on:

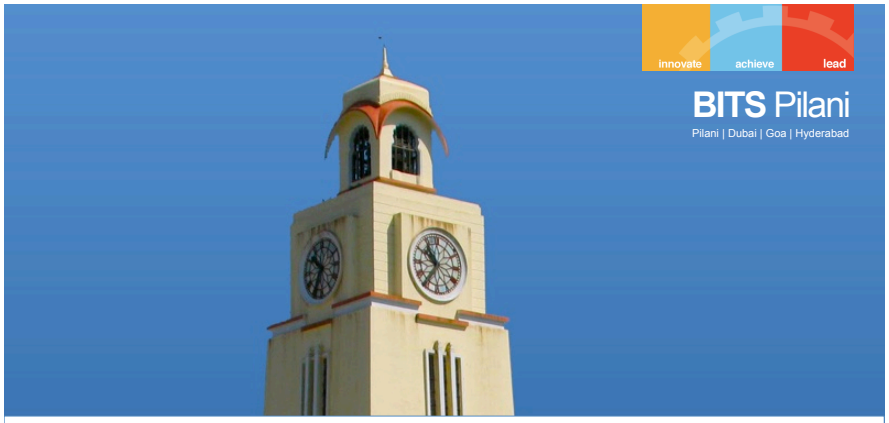
Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

No	Title
CS6.1.1	Explain meaning of Patterns, Design Patterns and how they matter for Programmers and Designers?
CS6.1.2	Introduce 5 GRASP Patterns, problem and application of each pattern.
CS6.1.3	Demonstrate the use of each of the GRASP pattern for PoS System



Explain meaning of Patterns, Design Patterns and how they matter for Programmers and Designers?

Patterns

- When experts work on a particular problem, it Is unusual for them to tackle it by inventing a new solution that Is completely distinct from existing ones.
- They often recall a similar problem they have already solved. and reuse the essence of Its solution to solve the new problem.
- This kind of 'expert behaviour' the thinking in problem-solution pairs is common to many different domains, such as architecture, economics and software engineering.
- It Is a natural way of coping with any kind of problem or social Interaction.

What Makes a Pattern?

A three-part schema that underlies every pattern:

Context: a situation giving rise to a problem.

Problem: the recurring problem arising in that context.

Solution: a proven resolution of the problem.

- The schema as a whole denotes a type of rule that establishes a relationship between
  - a given context,
  - a certain problem arising in that context, and
  - an appropriate solution to the problem.

Design Patterns

- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- Design Pattern
  - Context - Design situation giving rise to a design problem
  - Problem - Set of forces repeatedly arising in the context
  - Solution - Configuration to balance the forces
    - Structure with components and relationships
    - Run-time behaviour

Other 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.
- An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Example Context

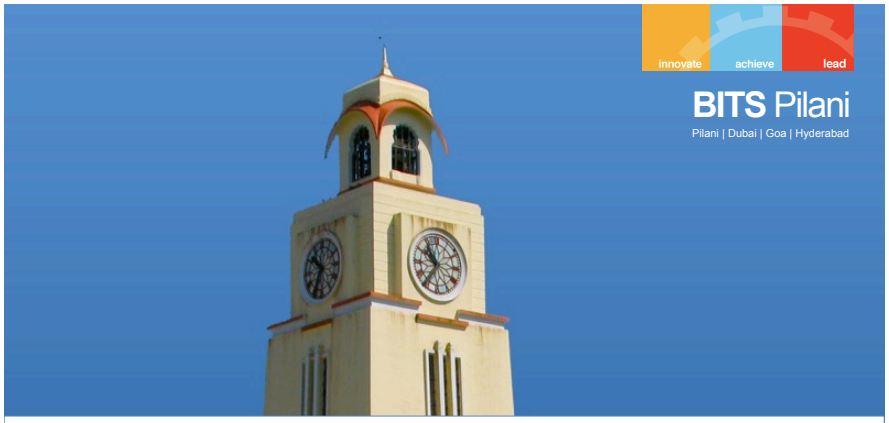
- A component uses data or information provided by another component.

Example Problem

- Changing the internal state of a component may introduce inconsistencies between cooperating components. To restore consistency, we need a mechanism for exchanging data or state information between such components.
- Two forces are associated with this problem:
  - The components should be loosely coupled- the information provider should not depend on details of its collaborators.
  - The components that depend on the information provider are not known a-priori.

Example Solution
Observer or Publisher-Subscriber

- Implement a change-propagation mechanism between the information provider- the subject; and the components dependent on it- the observers. Observers can dynamically register or unregister with this mechanism. Whenever the subject changes its state, it starts the change-propagation mechanism to restore consistency with all registered observers. Changes propagated by invoking a special update function common to all observers. To implement change propaga-tion (the passing of data and state information from the subject to the observers) you can use a pull-model, a push-model, or a combination of both.



5 GRASP Patterns

5 GRASP Patterns

1. Information Expert
2. Creator
3. Controller
4. Low Coupling (evaluative)
5. High Cohesion (evaluative)

Information Expert

- A general principle of object design and responsibility assignment.
- Assign a responsibility to the information expert – the class that has the information necessary to fulfil the responsibility

Creator

- Who creates? (Note that factory is a common alternative solution.)
- Assign class B the responsibility to create an instance of class A if one of these is true:
  1. B contains A
  2. B aggregates A
  3. B has the initialisation data for A
  4. B records A
  5. B closely uses A

Controller

- What first object beyond the UI layer receives and coordinates (“controls”) a system operation?
- Assign the responsibility to an object representing one of these choices:
  1. Represent the overall “system”, a “root object,” a device that the software is running within, or a major subsystem (these are all variations of a facade controller)
  2. Represent a use case scenario within which the system operation occurs (a use case or session controller)

Low Coupling (evaluative)

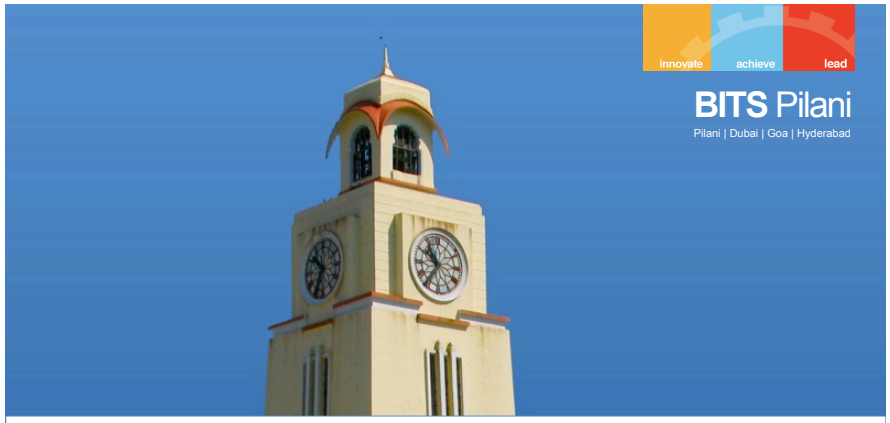
How to reduce the impact of change?

- Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

High Cohesion (evaluative)

How to keep objects focused, understanable, and manageable, and as a side-effect, support Low Coupling?

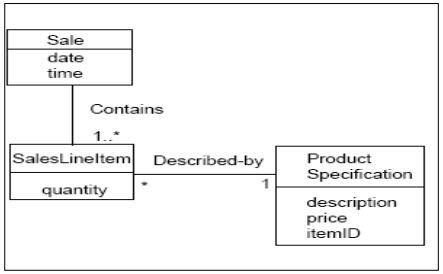
- Assign responsibilities s that cohesion remains high. Use this to evaluate alternatives.



GRASP pattern for PoS System

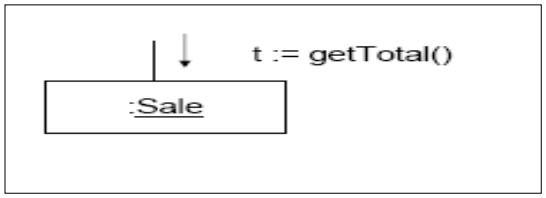
Information Expert (or Expert)

- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, i.e. it is an information expert for this responsibility.



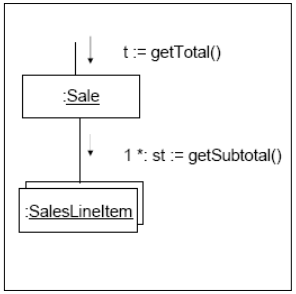
Information Expert (or Expert)

- This is a partial interaction diagram.



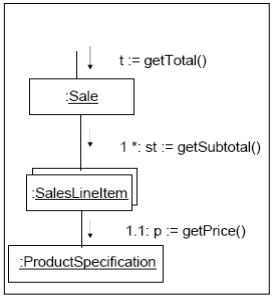
Information Expert (or Expert)

- What information is needed to determine the line item subtotal?
  - quantity and price.
- SalesLineItem should determine the subtotal.
- This means that Sale needs to send getSubtotal() messages to each of the SalesLineItems and sum the results.



Information Expert (or Expert)

- To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductSpecification is the information expert on answering its price.



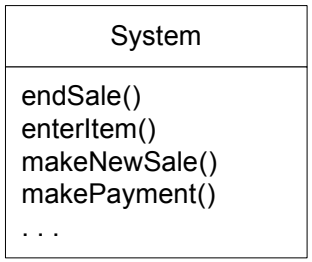
Information Expert (or Expert)

- To fulfil the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes
- The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many "partial experts" who will collaborate in the task.

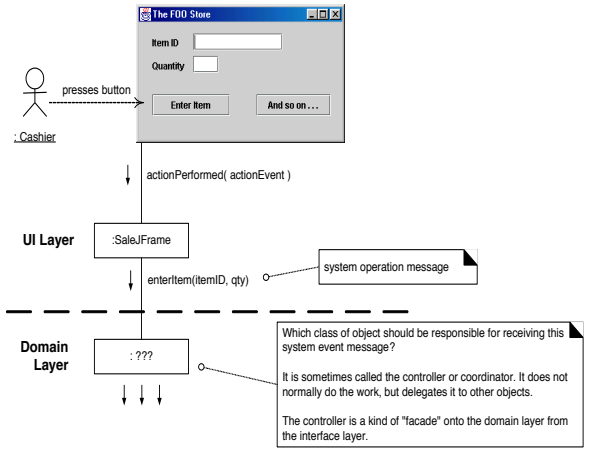
Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductSpecification	Knows product price

Controller

- The NextGen application contains several operations.
- You can model the system itself as a class.
- During analysis , the system operations may be assigned to the class System in some analysis model, to indicate that these are system operations. This does not mean that a software class named System fulfils them during design. During design a controller class is assigned the responsibility for system operations. (Next Slide)



Controller



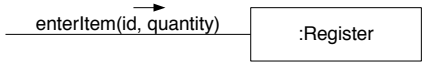
- Who should be the controller for system events such as enterItem and endSale?

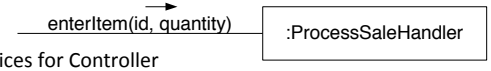
Controller

By the Controller Pattern, here are some choices:

- Represent the overall “system”, “root object”, device or subsystem: Register, POSSystem.
- Represent a receiver or handler of all system events of a use case scenario: ProcessSaleHandler, ProcessSaleSession.

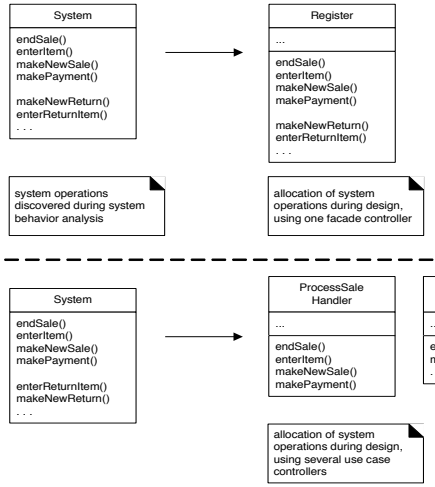
Note: domain of POS, aRegister (POS terminal) is a specialized device with software running on it.





Choices for Controller

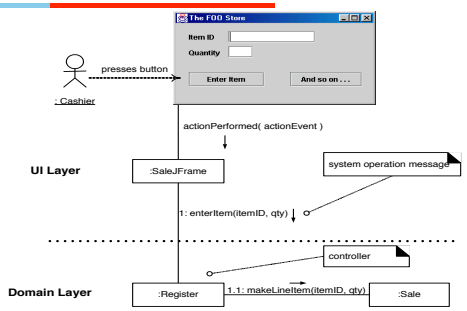
System Behavious Analysis



The assignment of responsibility for systems operation may be assigned to one or more classes. See examples on left.

Assignment by UI Layer

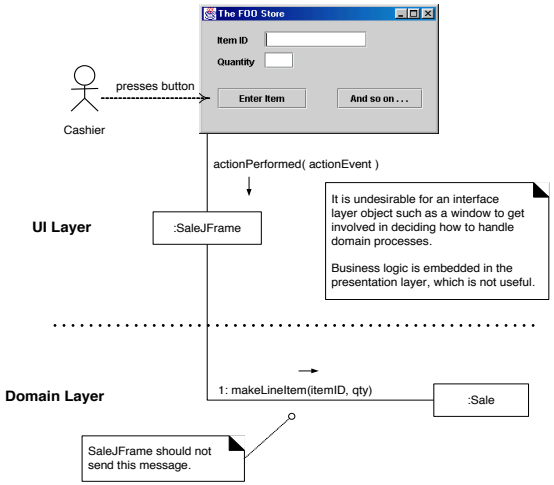
- Controller is basically a delegation pattern.
- It does not itself do work.
- UI Layer should not get involved with business logic.
- Controller pattern common choices of developer with respect to domain object delegate that receive the work request.
- Controller is a façade for the domain layer from UI layer.



Some Concepts

- Boundary objects – abstraction of interfaces
- Entity objects- application-independent (typically persistent) domain software objects.
- Control objects – case handlers
- Façade controllers – suitable when there are not many system events.
- Use case controllers are specialised controllers for each use case. All requests in one use case should go to one controller to maintain sequence.

Undesirable situations



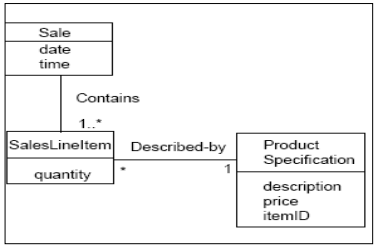
- UI Layer not to handle domain layer (as in the sketch on left)

<u>Bloated Controllers to be avoided</u>

- A single controller receiving all systems requests and there may be many.
- Controller handling systems events
- Controller maintains attributes

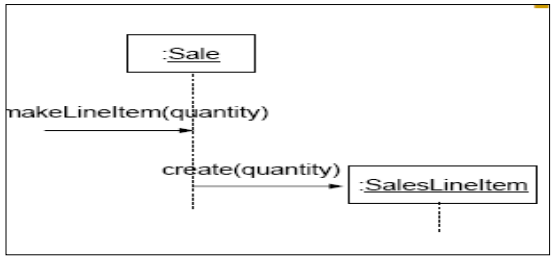
Creator

- In the POS application, who should be responsible for creating a SalesLineItem instance?
- Since a Sale contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate.



Creator

- This assignment of responsibilities requires that a makeLineItem method be defined in Sale.

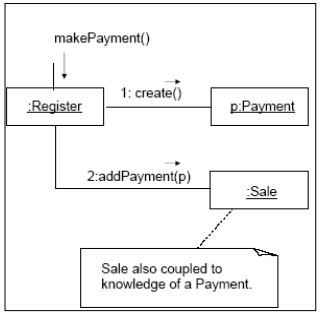


Low Coupling

- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- By Creator, Register is a candidate.

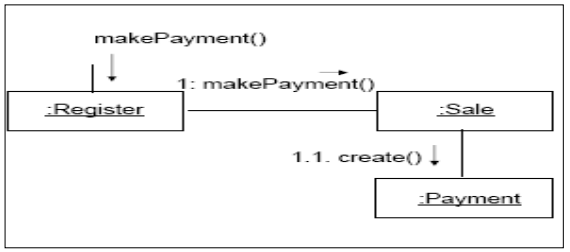
Low Coupling

- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- The assignment of responsibilities couples the Register class to knowledge of the Payment class.



Low Coupling

- An alternative solution is to create Payment and associate it with the Sale.
- No coupling between Register and Payment.

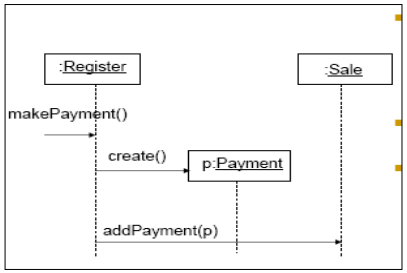


Low Coupling

- Some of the places where coupling occurs:
  - Attributes: X has an attribute that refers to a Y instance.
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Subclasses: X is a subclass of Y.
  - Types: X implements interface Y.
- There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.
- There will always be some coupling among objects, otherwise, there would be no collaboration.

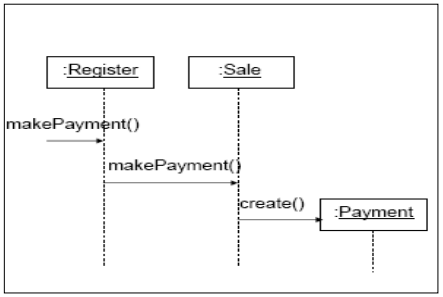
High Cohesion

- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- By Creator, Register is a candidate.
- Register may become bloated if it is assigned more and more system operations.



High Cohesion

- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.



Thank You

The slides are based on:

Applying UML and Patterns

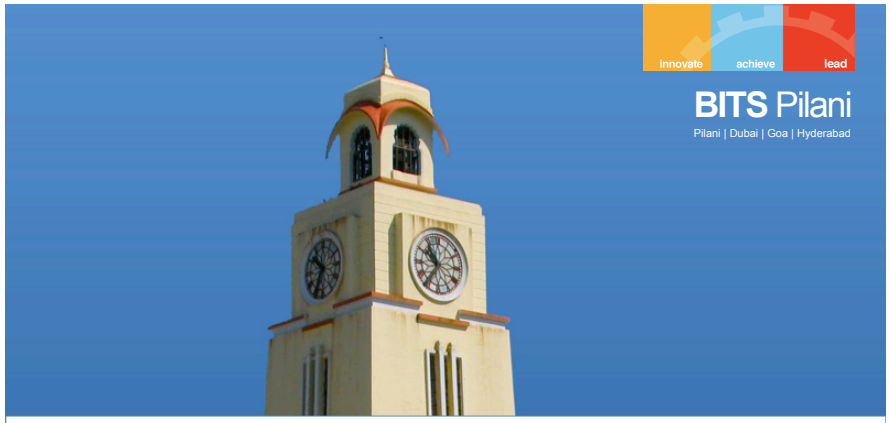
An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

innovate achieve lead

No	Title
CS6.2.1	Some more patterns (3 Ps and 1 I) and their application
CS6.2.2	Demonstrate how these patterns will be used in PoS System
CS6.2.3	Overview of all Design Principles and their usage in real time examples



Some more patterns (3 Ps and 1 I) and their application

Some more patterns (3 Ps and 1 I) and their application

innovate achieve lead

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Polymorphism

Who is responsible when behaviour varies by type?

- When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour- using polymorphic operations- to the type for which the behaviour varies.

Pure Fabrication

- Who is responsible when you are desperate and you do not want to violate high cohesion and low coupling.
- Assign a high cohesive set of responsibilities to an artificial or convenience “behaviour” class that does not represent a problem domain concept- something made up, in order to support high cohesion, low coupling, and reuse.

Indirection

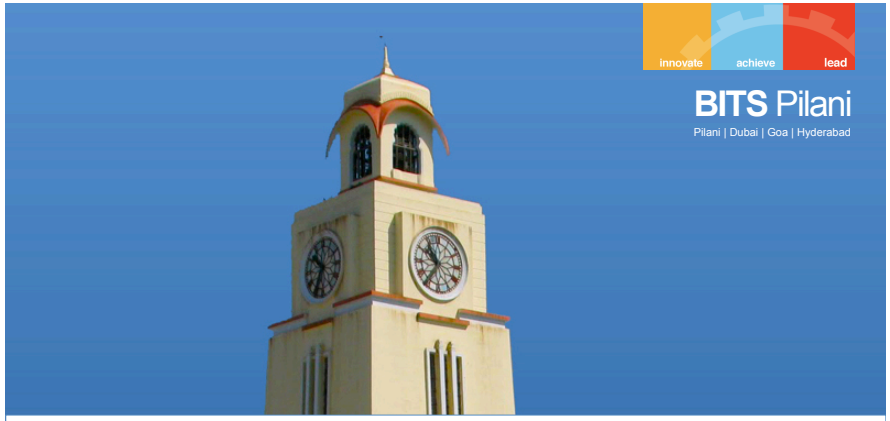
How to assign responsibility to avoid direct coupling?

- Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

Protected Variations

How to assign responsibility to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?

- Identify points of predicted variation or instability; assign responsibilities to create a stable “interface” around them.



Demonstrate how these patterns will be used in PoS System

Polymorphism

- When related alternatives or behaviours vary by type (class) assign responsibility for the behaviour- using polymorphism – to the types for which the behaviour varies.

Who is responsible when behaviour varies with type?

Pure Fabrication

- Assign a highly cohesive set of responsibilities to an artificial or convenient "behaviour" class that does not represent a problem domain concept;
- This may sometimes be made up, in order to support high cohesion, low coupling and reuse.

Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

Indirection

- Assign the responsibility to an intermediate object to mediate between the components or services, so that they are not directly coupled.

How is assigned responsibility to avoid direct coupling?

Protected Variation

- Identify points of predicted variation or instability;
- Assign responsibility to create a stable "interface" around them.

How to assign responsibility to objects, subsystems and systems so that the variations or services, so that they are not directly coupled.

Polymorphism

Solution: When related alternatives or behaviours vary by type (class), assign responsibilities for the behaviour (using polymorphic operations) to the types for which the behaviour varies.

Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

Problem:
How to handle alternatives based on type?
How to create pluggable software components?

Polymorphism

In this pattern an interface that contains the services of the type is created and several adapter classes implement this interface providing their own implementation for the services. When a message is sent to one of these adapters, the adapter applies its own method implementation.

Extensions required for new variations are easy to add, new implementations can be introduced without affecting clients.

Polymorphism has several related meanings.
In this context it means:
"giving the same names to services in different objects".

Polymorphism

The event is sent to the generalized (abstract) type, and sub-classes of the generalized type will exhibit polymorphism in having different implemented methods to respond to the event.

- When related responses (to the same event) vary by object type.
- Do not test object type and use conditional logic, apply "polymorphism."

Polymorphism

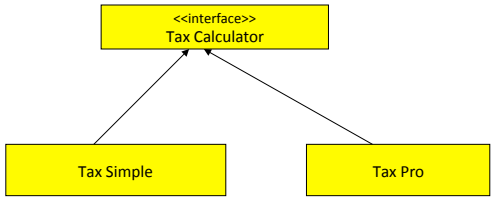
Assign a polymorphic operation to the family of classes for which the cases vary.

- Don't use case logic.

e.g., draw()

- Square, Circle, Triangle

e.g.



Pure Fabrication

Solution:

Assign a highly cohesive set of re
an artificial or convenience clas
represent a problem domain co
made up to support high cohes
and reuse.

Such class is a fabrication of the
Ideally the responsibilities assi
fabrication support high cohesi
coupling.

Problem:
What object should have the responsibility when you do not want to violate High Cohesion and Low Coupling.

Pure Fabrication

High Cohesion is supported because responsibilities are
factored into a fine grained class tha
very specific set of related task.

e.g. Instead of writing database logic i
separate artificial class "Persistance

Pure Fabrication is chosen by behavioural decomposition. Reuse potential may increase.

Pure Fabrication

Consider the following factor against **Expert**

- The tasks requires a relatively large number of database related operations, none of which are related to Sale (Low Cohesion)
- Sales Class will have to be coupled to a relational database which is external to the domain. (High Coupling)
- Saving objects to a database is a General task required by many classes. Placing it is sale will yield Low Reuse.

Case

You need to save Sale instance in a relational database.

By Information Expert: Assign the responsibility to Sale.

Consider Pure Fabrication of Persistent Storage Class. This term is not in the domain model but may be constructed for the convenience of the programmer.

Indirection

Solution: Assign the responsibility
intermediate object to mediate
components or services so tha
directly coupled. The intermedi
indirection between the other c

Lower coupling between compon

Problem:
Where to assign a responsibility to avoid direct coupling between two or more things?
How to decouple objects so that low coupling is supported and reuse potential remains higher?

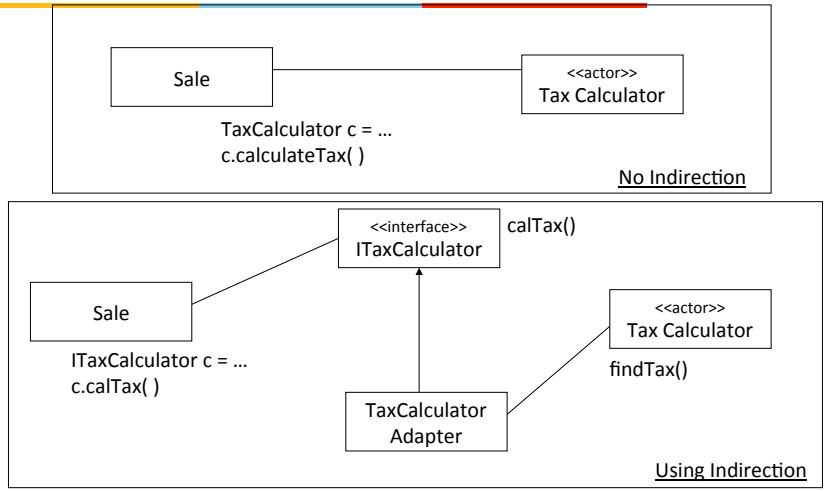
Indirection

When two objects need to interact, but they cannot get to know each other well.

We may abstract the interaction into another object which acts as a mediator, working on behalf of one of the objects.

Example: Listener objects in Java graphical user interface;

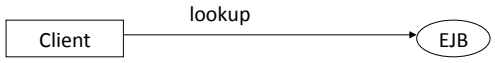
Indirection



Protected Variation

Solution:
Identify points of predicted variation or instability;

assign responsibilities to create a stable interface (or protection mechanism) around them.



Problem:
How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements.

Protected Variation

Technology like Service Lookup is an example of PV because clients are protected from variations in the location of services using the lookup service.

Externalizing properties in a property file is another example of PV.

Extensions required for new variations are easy to add.

New implementations can be introduced without affecting clients.

Protected Variation

PV through various mechanisms.
They are
- Core: The "Core" mechanism for PV is as following
  - Data encapsulation
  - Interfaces
  - Polymorphism
  - Indirection
- Date driven Design
  - Reading codes, class file paths, class name
- Service Look up
  - UDDI :Universal Description, Discovery and Integration is a registry which is platform independent which is modality to register and locate web services i.e. in turn it helps to look up the services
  - LDAP: It is Lightweight Directory Access Protocol which is meant for accessing (connecting), search and modify the internet directories. This is service protocol running a layer above TCP/IP stack.

Protected Variation

- Interpreter Driven Design
  - Virtual machines
  - Neural networks
  - Logic engines
  - Rule interpreters
- Reflective of meta level design
  - Java introspector
  - .NET reflectors
- Uniform access
  - Language supported constructs that do not change with underlying implementation
  - E.g. method and attributes are invoked same way (C#), Standards (e.g. APIs like ADO.NET, ODBC, JDBC etc)
- Wherever these mechanisms are encountered one has an implementation of PV in some or other form.

Protected Variation

This pattern is in line with the pure fabrication and Indirections in a sense that it addresses similar concerns.

The principles are same like low coupling high cohesion, reuse but the focus is on protecting the existing objects from variations. i.e. creating a stable interface so to protect from variations in coupled objects.

Approach: "Find what varies and encapsulate it."
Step I: Closely look for elements with direct coupling and also relatively prone to change.
Step II: Identify for the objects or points of predicted variation
Step III: Assign these responsibilities in such a way to create a stable interface around them.

Protected Variation

According to Larman, PV (Protected Variations) is a very important and root principle motivating most of the mechanism and patterns in programming and design to provide flexibility and protection from variation, almost every design trick in his book is a specialization of the Protected Variations pattern.

POS.

- Consider Classes: Sale, Payment, and SaleLineItem etc.
- Every sale amount has taxes applied to arrive at the total.
- There are different tax rules at different places and tax rates are different also.
- On top of this the rates and rules are bound to changes which makes the design of such systems tricky.

Protected Variation

The best approach is design the tax calculation as separate system.

There could be different tax calculator systems (third party) available or there could be web service providing the tax calculation services.

There might be reason for having multiple systems or move to different system.

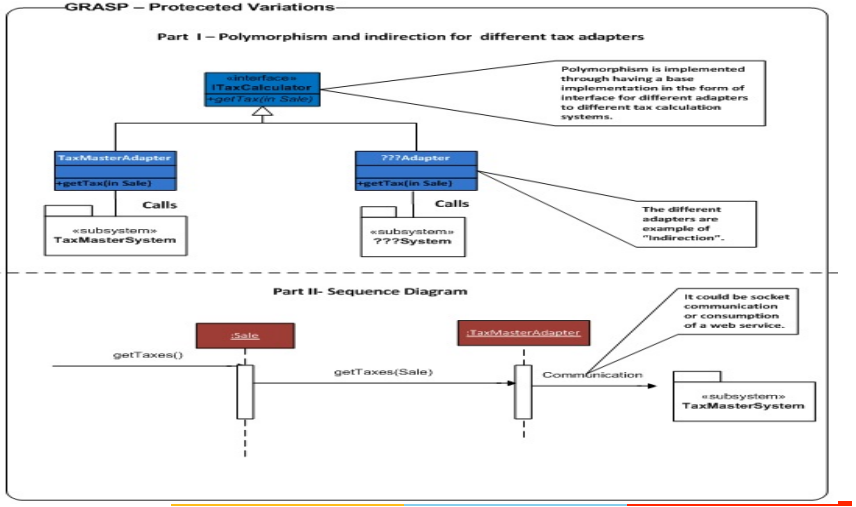
- As mentioned above, the rules, rates are the moving target and as these changes the tax calculation systems have to follow.
- This is the point of variation or stability. In such scenario, the changes would require the consumers or users of such systems, to change also.

Protected Variation

These changes or variations would cause the changes in POS system or it may require revision.

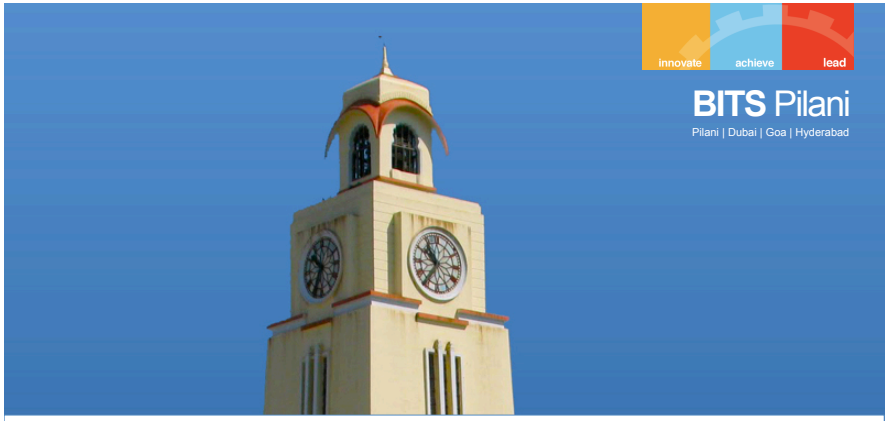
This is where the “protected Variations” pattern helps to design in such a way that the changes in the tax calculation system would not cause major problems. This is accomplished through implementation of polymorphism and indirection as depicted in the following diagram.

Protected Variation



Protected Variation

Class	Responsibility and method	Remarks
ITaxCalculator	Facilitator for different adapters to different systems	Provides polymorphic behaviour and handles the variations based on the type.
….Adapter	Facilitator for particular Tax System	These classes provide indirection i.e. facilitates low coupling and reuse.



Overview of all Design Principles and their usage in real time examples

What is Test Driven Development?

A software development technique where you write automated unit tests ***before*** you write your implementation code

A technique for ensuring good quality and good design

CHANGE - TESTING

Goal: Better Quality

Fewer bugs – it costs money to find bugs, fix bugs, and clean up the mess created by bugs.

More flexibility – since code *will* need to change, it should be easy to change

Object-Oriented Programming

Definition:

"A method of programming based on a hierarchy of classes, and well-defined and cooperating objects."

OOP vs. Procedural Programming

- **Object-oriented programming**

  Collaboration between objects that send and receive messages with each other
  Functionality is grouped by object
  Objects and their behavior can be reused
  NET, Java, Ruby, C++

- **Procedural programming**

  A series of functions, subroutines, or tasks
  Functionality is grouped by task
  Functions/subroutines/tasks can be reused
  SQL, VB6

Why object-oriented programming?

Objects are a natural way of representing behavior and properties in code
Objects are more reusable
Objects are more maintainable
Objects are more extensible

Some tips

- Just because you are using an OO language does not mean that you are doing object-oriented programming
- Reuse is good (avoid NIH syndrome)
- The purely object-oriented solution is not always the best solution
- Don't be a code hoarder

3 Tenants of OOP

Encapsulation
Inheritance
Polymorphism

Encapsulation

```
public class BankAccount
{
    public decimal Balance { get; set; }
}
```

BankAccount has a Balance,

but this class does not contain information about the ***behaviors*** of a BankAccount.

Encapsulation

```
public class BankAccount
{
    public decimal Balance { get; set; }

    public void Deposit(decimal amount)
    {
        Balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        Balance -= amount;
    }
}
```

We could do it this way, but we are violating encapsulation by exposing the inner workings of this class (the Balance property).

Encapsulation

```
public class BankAccount
{
    private decimal _balance;
    public decimal Balance
    {
        get { return _balance; }
    }

    public void Deposit(decimal amount)
    {
        _balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        _balance -= amount;
    }
}
```

Much better – the inner workings of the class are hidden from consumers of the class.

Encapsulation

```
public class BankAccount
{
    private decimal _balance;

    public decimal Balance
    {
        get { return _balance; }
    }

    public void Deposit(decimal amount)
    {
        _balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (_balance - amount < 0)
            throw new InsufficientFundsException();
        _balance -= amount;
    }
}
```

Change! Throw an exception if there are insufficient funds during a withdrawal.

Encapsulation

```
public bool Withdraw(BankAccount bankAccount, decimal amount)
{
    bool couldWithdrawMoney;

    // check to see if there is enough money for the withdrawal
    if (bankAccount.Balance >= amount)
    {
        bankAccount.Withdraw(amount);
        couldWithdrawMoney = true;
    }
    else
        couldWithdrawMoney = false;
    return couldWithdrawMoney;
}
```

Change! You cannot withdraw money if the account is closed.

Encapsulation

```
public bool Withdraw(BankAccount bankAccount, decimal
   amount)
{
    bool couldWithdrawMoney;

    // check to see if there is enough money for the
   withdrawal
    // and if the account is open
    if (bankAccount.Balance >= amount &&
        bankAccount.Status == BankAccountStatus.Open)
    {
        bankAccount.Withdraw(amount);
        couldWithdrawMoney = true;
    }
    else
        couldWithdrawMoney = false;
   return couldWithdrawMoney;
}
```

Change! You cannot withdraw money if the account is closed. The code in the *if* statement is a ***leaky abstraction*** – the details about whether you can withdraw money is spread out among the consumers of the BankAccount instead of being ***encapsulated*** inside the BankAccount object.

Encapsulation

```
public class BankAccount {
    public bool CanWithdraw(decimal amount) {
        return Balance >= amount && Status ==
   BankAccountStatus.Open;
    }
}

public bool Withdraw(BankAccount bankAccount, decimal amount){
    bool couldWithdrawMoney;
    // no need for a comment anymore, this code says what it
   does!
    if (bankAccount.CanWithdraw(amount)) {
        bankAccount.Withdraw(amount);
        couldWithdrawMoney = true;
    }
    else
        couldWithdrawMoney = false;
    return couldWithdrawMoney;
}
```

The details about whether an amount of money can be withdrawn from a bank account is now ***encapsulated*** inside the BankAccount class.

Law of Demeter

```
public class AtmMachine
{
    public void DoSomething(BankAccount bankAccount)
    {
        if (bankAccount.PrimaryAccountHolder.State == "OH")
        {
            DoSomethingElse(bankAccount);
        }
    }
}
```

A method of an object may only call methods of:

1) The object itself.
2) An argument of the method.
3) Any object created within the method.
4) Any direct properties/fields of the object.

Law of Demeter

```
public class AtmMachine
{
    public void DoSomething(BankAccount bankAccount)
    {
        if (bankAccount.StateWhereAccountIsHeld == "OH")
        {
            DoSomethingElse(bankAccount);
        }
    }
}

public class BankAccount
{
    public string StateWhereAccountIsHeld
    {
        get { return PrimaryAccountHolder.State; }
    }
}
```


Inheritance

"Is-a" vs. "Has-a"

What is the relationship between these objects?

Customer	**PreferredCustomer**	**CustomerContact**
• string FirstName	• string FirstName	• string FirstName
• string LastName	• string LastName	• string LastName
• string Address	• string Address	• string Address

Inheritance

"Is-a" vs. "Has-a"

What is the relationship between these objects?

PreferredCustomer
: Customer
• IList<CustomerContact> Contacts

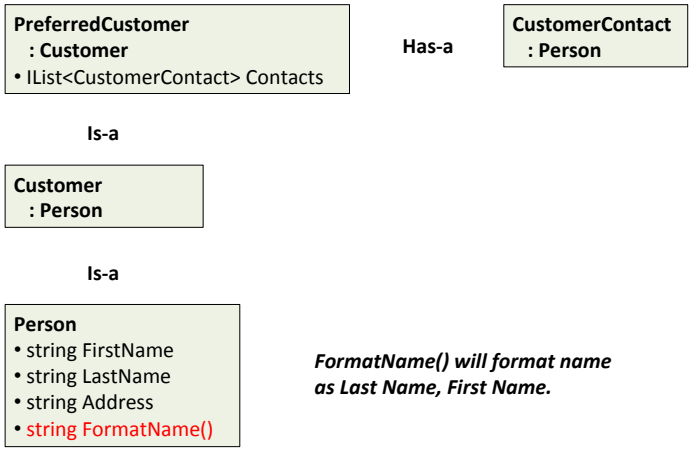
Has-a

CustomerContact
• string FirstName
• string LastName
• string Address

Is-a

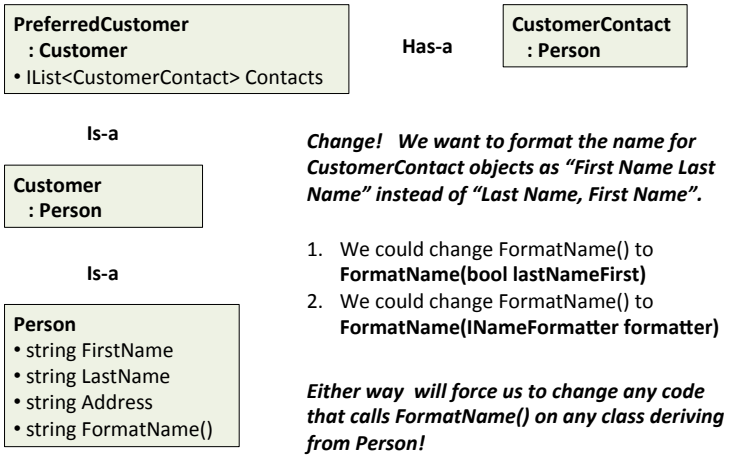
Customer
• string FirstName
• string LastName
• string Address

We could do this…



FormatName() will format name as Last Name, First Name.

We could do this…



Change! We want to format the name for CustomerContact objects as "First Name Last Name" instead of "Last Name, First Name".

1. We could change FormatName() to **FormatName(bool lastNameFirst)**
2. We could change FormatName() to **FormatName(INameFormatter formatter)**

Either way will force us to change any code that calls FormatName() on any class deriving from Person!

Rethinking inheritance

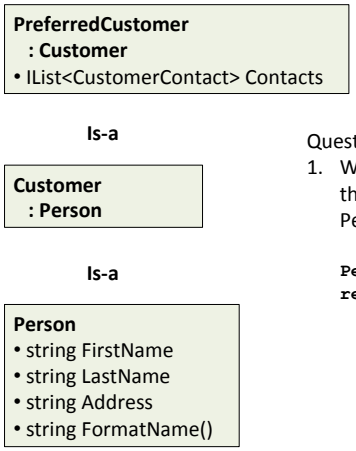
- What's good about inheritance

  Enables code reuse
  Polymorphism ("one with many forms")

- What's bad about inheritance

  Tight coupling – changes to the base class can cause all derived classes to have to changes
  Deep inheritance hierarchies are sometimes hard to figure out

Rethinking inheritance



Question:

1. Will we ever have code that requires us to use the methods and properties that we put in the Person class?

```
Person person = GetImportantPerson();
return person.FormatName();
```

Rethinking inheritance

PreferredCustomer
: Customer
• IList<CustomerContact> Contacts

Has-a

CustomerContact
• string FirstName
• string LastName
• string Address
• string FormatName()

Is-a

Customer
• string FirstName
• string LastName
• string Address
• string FormatName()

Employee
• string FirstName
• string LastName
• string FormatName()

What do we have now:

1. Our classes are no longer tightly coupled by inheritance, so we can change FormatName() on one class without affecting another
2. But we still want to reuse the name formatting code!

Rethinking inheritance

Create classes that format names – an object-oriented solution!

```
public class FirstNameLastNameNameFormatter
{
    public string FormatName(string firstName, string lastName)
    {
        return firstName + " " + lastName;
    }
}

public class LastNameFirstNameNameFormatter
{
    public string FormatName(string firstName, string lastName)
    {
        return lastName + ", " + firstName;
    }
}
```


Rethinking inheritance

Any class that needs to format names can use the name formatter objects.

```
public class Customer
{
    public string FormatName()
    {
        return new FirstNameLastNameNameFormatter()
            .FormatName(FirstName, LastName);
    }
}
```

This technique is called ***composition*** – classes can add functionality by using functionality in other objects.

Ruby mixins


```
module FirstNameLastNameNameFormatter
  def format_name
    return first_name + " " + last_name;
  end
end

class Customer
  include FirstNameLastNameNameFormatter
  attr :first_name, true
  attr :last_name, true
end

customer = Customer.new
customer.first_name = 'Jon'
customer.last_name = 'Kruger'
puts customer.format_name  # this line will output: Jon Kruger
```

Composition vs. inheritance

Composition allows you to reuse code without being tightly coupled to a base class

Many loosely coupled, small, testable classes that provide bits of functionality that can be used by anyone who wants to use them

Example: going shopping at a grocery store vs. growing your own food on a farm – at the grocery store, it's much easier to change what you get!

More Composition

```
public class Fruit
{
    // Return number of pieces of peel that
    // resulted from the peeling activity.
    public int Peel()
    {
        return 1;
    }
}

public class Apple : Fruit
{
}

public class Example1
{
    public static void Main(String[] args)
    {
        Apple apple = new Apple();
```


More Composition

```
public class Fruit
{
    // Return number of pieces of peel that
    // resulted from the peeling activity.
    public PeelResult Peel()
    {
        return new PeelResult { NumberOfPeels = 1, Success = true };
    }
}

public class Apple : Fruit
{
}

public class Example1
{
    public static void Main(String[] args)
    {
        Apple apple = new Apple();
```


More Composition

```
public class Fruit
{
    // Return number of pieces of peel that
    // resulted from the peeling activity.
    public int Peel()
    {
        return 1;
    }
}

public class Apple
{
    private Fruit fruit = new Fruit(); // Composition instead of inheritance
    public int Peel()
    {
        return fruit.Peel();
    }
}

public class Example2
{
    public static void Main(String[] args)
    {
        Apple apple = new Apple();
```

More Composition

```
public class Fruit
{
    // Return number of pieces of peel that
    // resulted from the peeling activity.
    public PeelResult Peel()
    {
        return new PeelResult { NumberOfPeels = 1, Success = true };
    }
}

public class Apple
{
    private Fruit fruit = new Fruit(); // Composition instead of inheritance
    public int Peel()
    {
        return fruit.Peel().NumberOfPeels; // Changes stop here!
    }
}

public class Example2
{
    public static void Main(String[] args)
    {
        Apple apple = new Apple();
```


Ground Rules

These are **guidelines**, not hard and fast rules

Use your brain – do what makes sense

Ask **why**

This should make things easier!

Software should be:

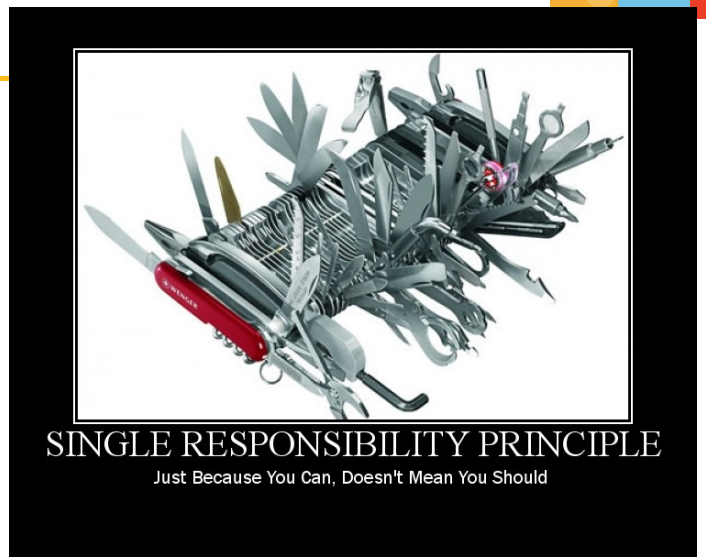
- Easy to test
- Easy to change
- Easy to add features to

Easy != not learning a new way of doing things

Complexity

Have only as much complexity as you need – have a reason for complexity

“I don’t want to learn this new way” != too complex



Single Responsibility Principle

A class should have one, and only one, reason to change.

SRP Violation – Multiple Responsibilities

```
public class Person
{
    private const decimal _minimumRequiredBalance = 10m;

    public string Name { get; set; }
    public decimal Balance { get; set; }

    public decimal AvailableFunds
    {
        get { return Balance - _minimumRequiredBalance; }
    }

    public void DeductFromBalanceBy(decimal amountToDeduct)
    {
        if (amountToDeduct > Balance)
            throw new InvalidOperationException("Insufficient
   funds.");

        Balance -= amountToDeduct;
    }
}
```


SRP Fix – Split big classes

```
public class Account
{
    private const decimal _minimumRequiredBalance = 10m;

    public decimal Balance { get; set; }

    public decimal AvailableFunds
    {
        get { return Balance - _minimumRequiredBalance; }
    }

    public void DeductFromBalanceBy(decimal amountToDeduct)
    {
        if (amountToDeduct > Balance)
            throw new InvalidOperationException("Insufficient
   funds.");

        Balance -= amountToDeduct;
    }
}
```

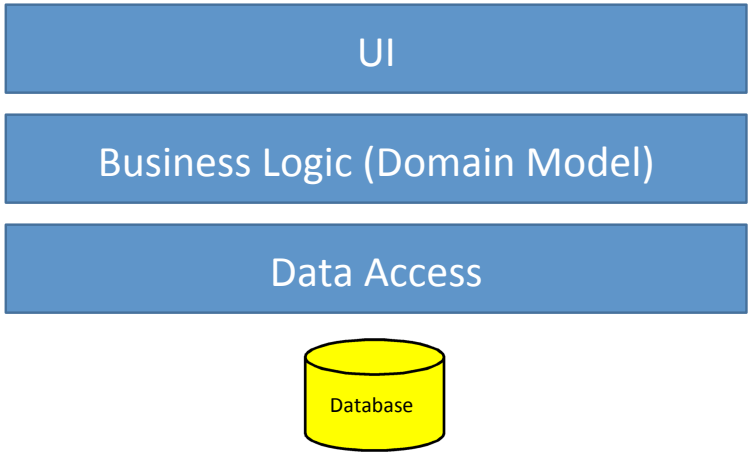
SRP Fix – Split big classes

```
public class Person
{
    public string Name { get; set; }
    public Account Account { get; set; }

    public decimal AvailableFunds
    {
        get { return Account.AvailableFunds; }
    }

    public decimal AccountBalance
    {
        get { return Account.Balance; }
    }

    public void DeductFromBalanceBy(decimal amountToDeduct)
    {
        Account.DeductFromBalanceBy(amountToDeduct);
    }
```


SRP Violation -Spaghetti Code

```
public class OrderProcessingModule {
  public void Process(OrderStatusMessage orderStatusMessage) {
    // Get the connection string from configuration
    string connectionString = ConfigurationManager.ConnectionStrings["Main"].ConnectionString;

    Order order = null;

    using (SqlConnection connection = new SqlConnection(connectionString)) {
      // go get some data from the database
      order = fetchData(orderStatusMessage, connection);
    }

    // Apply the changes to the Order from the OrderStatusMessage
    updateTheOrder(order);

    // International orders have a unique set of business rules
    if (order.IsInternational)
      processInternationalOrder(order);

    // We need to treat larger orders in a special manner
    else if (order.LineItems.Count > 10)
      processLargeDomesticOrder(order);

    // Smaller domestic orders
    else
      processRegularDomesticOrder(order);

    // Ship the order if it's ready
```


Tips for not violating SRP - Layers



SRP Violation – "god" classes

```
public class OrderService
{
    public Order Get(int orderId) { ... }
    public Order Save(Order order) { ... }
    public Order SubmitOrder(Order order) { ... }
    public Order GetOrderByName(string name) { ... }
    public void CancelOrder(int orderId) { ... }
    public void ProcessOrderReturn(int orderId) { ... }
    public IList<Order> GetAllOrders { ... }
    public IList<Order> GetShippedOrders { ... }
    public void ShipOrder { ... }
}
```

Tips for not violating SRP – XML Comments

Fill out the XML doc comments for the class – be wary of words like *if, and, but, except, when,* etc.

```
/// <summary>
/// Gets, saves, and submits orders.
/// </summary>
public class OrderService
{
    public Order Get(int orderId) { ... }
    public Order Save(Order order) { ... }
    public Order SubmitOrder(Order order) { ... }
}
```


Tips for not violating SRP - Verbs

Domain services should have a verb in the class name

```
public class GetOrderService
{
    public Order Get(int orderId) { ... }
}

public class SaveOrderService
{
    public Order Save(Order order) { ... }
}

public class SubmitOrderService
{
    public Order SubmitOrder(Order order) { ... }
}
```


Why SRP matters

We want it to be easy to reuse code

Big classes are more difficult to change

Big classes are harder to read

Smaller classes and smaller methods will give you more flexibility, and you don't have to write much extra code (if any) to do it!

When violating SRP *might* be OK

The violating class is not going to be reused and other classes don't depend on it

The violating class does not have private fields that store values that the class uses

Your common sense says so

Example: MVC controller classes, web services

Don’t overcomplicate!

Don’t code for situations that you won’t ever need

Don’t create unneeded complexity

However, more class files != more complicated

Remember, this is supposed to make your lives **easier**!
(but not easier to be lazy)

You can always refactor later (if you write tests)

Offshoot of SRP - Small Methods

A method should have one purpose (reason to change)

Easier to read and write, which means you are less likely to write bugs

Write out the steps of a method using plain English method names

Small Methods - Before

```
public void SubmitOrder(Order order)
{
    // make sure that the order has products
    if (order.Products.Count == 0)
    {
        throw new InvalidOperationException(
            "Select a product.");
    }

    // calculate tax
    order.Tax = order.Subtotal * 1.0675;

    // calculate shipping
    if (order.Subtotal < 25)
        order.ShippingCharges = 5;
    else
        order.ShippingCharges = 10;

    // submit the order
    _orderSubmissionService.SubmitOrder(order);
}
```


Small Methods - After

```
public void SubmitOrder(Order order)
{
    ValidateOrderHasProducts(order);
    CalculateTax(order);
    CalculateShipping(order);
    SendOrderToOrderSubmissionService(order);
}
```

Small Methods - After

```
public void SubmitOrder(Order order)
{
    ValidateOrderHasProducts(order);
    CalculateTax(order);
    CalculateShipping(order);
    SendOrderToOrderSubmissionService(order);
}

public void ValidateOrderHasProducts(Order order)
{
    if (order.Products.Count == 0)
        throw new InvalidOperationException("Select a product.");
}

public void CalculateTax(Order order)
{
    order.Tax = order.Subtotal * 1.0675;
}

public void CalculateShipping(Order order)
{
    if (order.Subtotal < 25)
        order.ShippingCharges = 5;
    else
        order.ShippingCharges = 10;
}

public void SendOrderToOrderSubmissionService(Order order)
{
    _orderSubmissionService.SubmitOrder(order);
}
```


Open Closed Principle

Software entities (classes, modules, methods) should be open for extension but closed for modification.

OCP Violation

```
public class GetUserService
{
    public IList<UserSummary> FindUsers(UserSearchType type)
    {
        IList<User> users;
        switch (type)
        {
            case UserSearchType.AllUsers:
                // load the "users" variable here
                break;
            case UserSearchType.AllActiveUsers:
                // load the "users" variable here
                break;
            case UserSearchType.ActiveUsersThatCanEditQuotes:
                // load the "users" variable here
                break;
        }

        return ConvertToUserSummaries(users);
    }
}
```

OCP Fix – Strategy Pattern

```
public interface IUserQuery
{
    IList<User> FilterUsers(IList<User> allUsers);
}

public class GetUserService
{
    public IList<UserSummary> FindUsers(IUserQuery query)
    {
        IList<User> users = query.FilterUsers(GetAllUsers());
        return ConvertToUserSummaries(users);
    }
}
```


Why OCP matters

Anytime you change code, you have the potential to break it

Sometimes you can't change libraries (e.g. code that isn't yours)

May have to change code in many different places to add support for a certain type of situation

When violating OCP is OK

When the number of options in the if or switch statement is unlikely to change (e.g. switch on enum)

```
public void UpdateFontStyle (Paragraph paragraph)
{
    switch (IsBoldCheckBox.CheckState)
    {
        case CheckState.Checked:
            paragraph.IsBold = true;
            break;
        case CheckState.Unchecked:
            paragraph.IsBold = false;
            break;
        case CheckState.Indeterminate:
            break;
    }
}
```


OCP Rules of Thumb

Use if/switch if the number of cases is unlikely to change

Use strategy pattern when the number of cases are likely to change

Always use common sense!

Don’t overcomplicate!

Don’t code for situations that you won’t ever need
Don’t create unneeded complexity
However, more class files != more complicated
Remember, this is supposed to make your lives **easier**!
You can always refactor later (if you write tests)

Liskov Substitution Principle

Functions that use references to base classes must be able to use objects of derived classes without knowing it.

LSP Violation – Bad Abstraction

```
public class Product
{
    public string Name { get; set; }
    public string Author { get; set; }
}

public class Book : Product  {}

public class Movie : Product  {}
```

If someone had a Product object (which was actually a Movie) and asked for the Author, what should it do (a Movie doesn’t have an Author)?

LSP in other words…

People using derived classes should not encounter unexpected results when using your derived class.

LSP Violation – Unexpected Results

```
public class MyList<T> : IList<T>
{
    private readonly List<T> _innerList = new List<T>();

    public void Add(T item)
    {
        if (_innerList.Contains(item))
            return;
        _innerList.Add(item);
    }

    public int Count
    {
        get { return _innerList.Count; }
    }
}
```

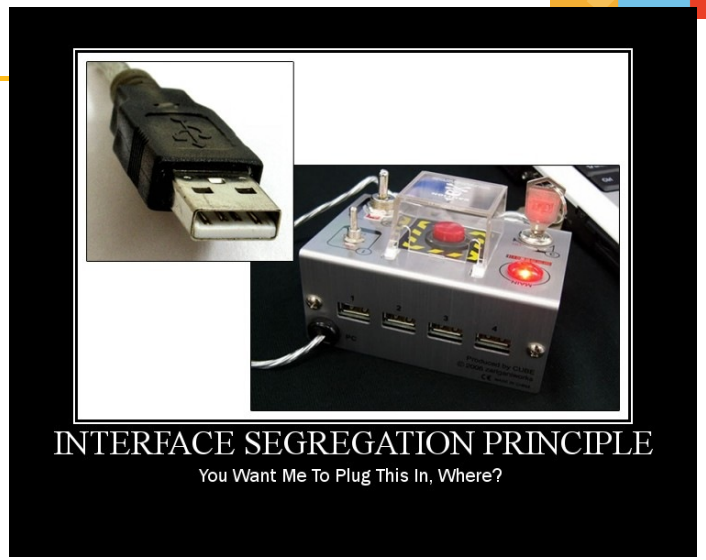

If you violate LSP…

Throw exceptions for cases that you can't support (still not recommended)

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area
    {
        get { return Width * Height; }
    }
}

public class Cube : Shape
{
    public override double Area
    {
        get { throw new NotSupportedException(); }
    }
}
```


Interface Segregation Principle

Clients should not be forced to depend on interfaces that they do not use.




```
public abstract class MembershipProvider : ProviderBase
{
  // Fields
  private MembershipValidatePasswordEventHandler _EventHandler;
  private const int SALT_SIZE_IN_BYTES = 0x10;

  // Events
  public event MembershipValidatePasswordEventHandler ValidatingPassword;

  // Methods
  protected MembershipProvider();
  public abstract bool ChangePassword(string username, string oldPassword, string newPassword);
  public abstract bool ChangePasswordQuestionAndAnswer(string username, string password, string newPasswordQuestion, string newPasswordAnswer);
  public abstract MembershipUser CreateUser(string username, string password, string email, string passwordQuestion, string passwordAnswer, bool isApproved, object providerUserK
  protected virtual byte[] DecryptPassword(byte[] encodedPassword);
  public abstract bool DeleteUser(string username, bool deleteAllRelatedData);
  internal string EncodePassword(string pass, int passwordFormat, string salt);
  protected virtual byte[] EncryptPassword(byte[] password);
  public abstract MembershipUserCollection FindUsersByEmail(string emailToMatch, int pageIndex, int pageSize, out int totalRecords);
  public abstract MembershipUserCollection FindUsersByName(string usernameToMatch, int pageIndex, int pageSize, out int totalRecords);
  internal string GenerateSalt();
  public abstract MembershipUserCollection GetAllUsers(int pageIndex, int pageSize, out int totalRecords);
  public abstract int GetNumberOfUsersOnline();
  public abstract string GetPassword(string username, string answer);
  public abstract MembershipUser GetUser(object providerUserKey, bool userIsOnline);
  public abstract MembershipUser GetUser(string username, bool userIsOnline);
  internal MembershipUser GetUser(string username, bool userIsOnline, bool throwOnError);
  public abstract string GetUserNameByEmail(string email);
  protected virtual void OnValidatingPassword(ValidatePasswordEventArgs e);
  public abstract string ResetPassword(string username, string answer);
  internal string UnEncodePassword(string pass, int passwordFormat);
  public abstract bool UnlockUser(string userName);
  public abstract void UpdateUser(MembershipUser user);
  public abstract bool ValidateUser(string username, string password);

  // Properties
  public abstract string ApplicationName { get; set; }
  public abstract bool EnablePasswordReset { get; }
  public abstract bool EnablePasswordRetrieval { get; }
  public abstract int MaxInvalidPasswordAttempts { get; }
  public abstract int MinRequiredNonAlphanumericCharacters { get; }
  public abstract int MinRequiredPasswordLength { get; }
  public abstract int PasswordAttemptWindow { get; }
  public abstract MembershipPasswordFormat PasswordFormat { get; }
  public abstract string PasswordStrengthRegularExpression { get; }
  public abstract bool RequiresQuestionAndAnswer { get; }
  public abstract bool RequiresUniqueEmail { get; }
}
```

ASP.NET MembershipProvider class – a very "fat" interface!

ISP Smells

If you implement an interface or derive from a base class and you have to throw an exception in a method because you don't support it, the interface is probably too big.

Why ISP matters

Single Responsibility Principle for interfaces/base classes

If your interface has members that are not used by some inheritors, those inheritors may be affected by changes in the interface, even though the methods that they use did not change.

When violating ISP is OK

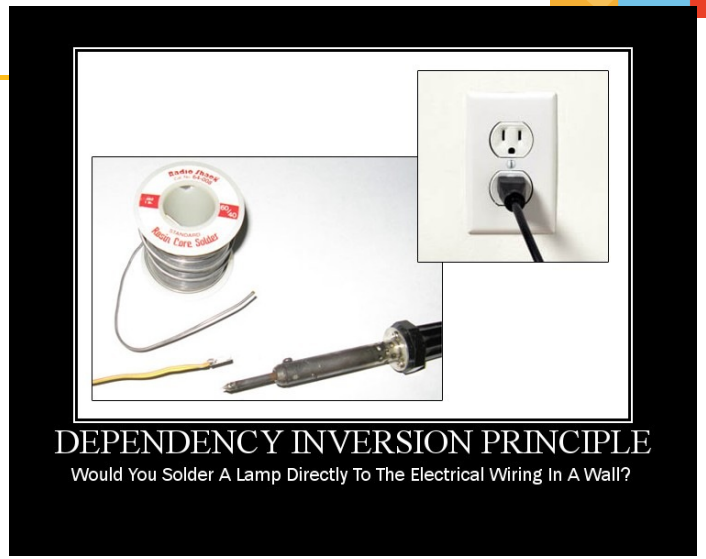
Why would you want to?

Use common sense

Dependency Inversion Principle

High level modules should not depend on low level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.



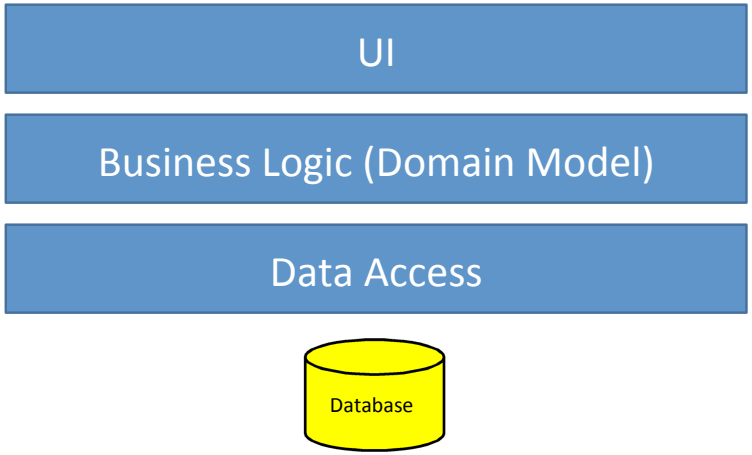
Tight Coupling

Two classes are ***tightly coupled*** if they are linked together and are dependent on each other

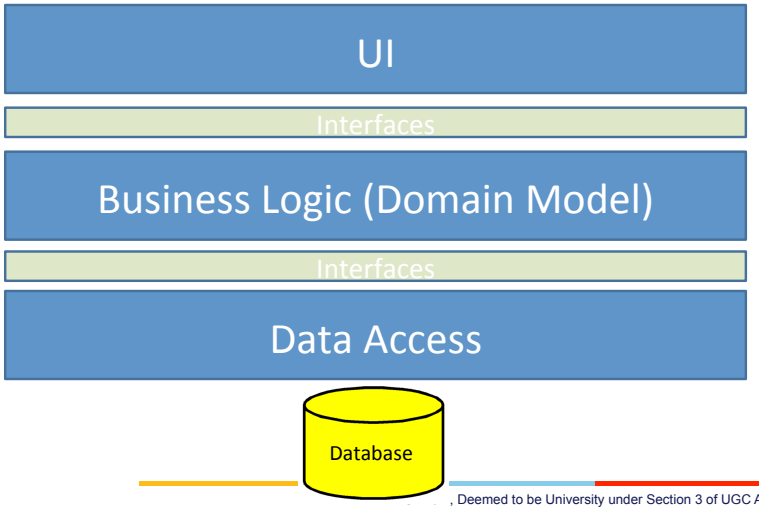
Tightly coupled classes can not work independent of each other

Makes changing one class difficult because it could launch a wave of changes through tightly coupled classes

Tips for not violating DIP - Layers



Tips for not violating DIP - Layers



Tips for not violating DIP - Layers

Each layer should not know anything about the details of how the other layers work.

Example: your domain model should not know how data access is done – it shouldn't know if you're using stored procedures, an ORM, etc.

Layers – What's the big deal?

Tight coupling is bad – if your business layer contains code related to data access, **changes** to how data access is done will affect business logic

Harder to **test** because you have to deal with implementation details of something you're not trying to test

DIP Enables Testability

Stubs, mocks, and fakes in unit tests are only possible when we have an interface to implement

Unit Tests vs. Integration Tests

Unit tests:

- Tests a small unit of functionality
- Mock or “fake out” external dependencies (e.g. databases)
- Run fast

Integration tests:

- Test the whole system working together
- Can run slow
- Can be brittle

DIP Violations

Use of “new”

```
public class GetProductService
{
    public IList<Product> GetProductById(int id)
    {
        var productRepository = new ProductRepository();
        return productRepository.Get(id);
    }
}
```


DIP Violations

Use of “static”

```
public class SecurityService
{
    public static User CurrentUser { get; set; }
}
```

DIP Violations

Use of singletons using “static”

```
public class ProductCache
{
    private static readonly _instance = new ProductCache();
    public static ProductCache Instance
    {
        get { return _instance; }
    }
}
```


Enabling DIP – create interfaces


```
public class ProductRepository : IProductRepository
{
    public Product Get(int id) { ... }
}

public interface IProductRepository
{
    Product Get(int id);
}
```


Enabling DIP – Constructor Injection


```
public class GetProductService : IGetProductService
{
    private IProductRepository _productRepository;

    public GetProductService(
        IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }

    public IList<Product> GetProductById(int id)
    {
        return _productRepository.Get(id);
    }
}
```


What is a DI (IoC) Container?

Creates objects that are ready for you to use

Knows how to create objects and their dependencies

Knows how to initialize objects when they are created (if necessary)

DI Containers

Popular .NET choices:

– StructureMap, Ninject

Other .NET choices:

– Unity, Castle Windsor, Autofac, Spring .NET

Java

– Spring

Ruby

– We don't need no stinkin' DI containers!

Enabling DIP – Constructor Injection

```
public class GetProductService : IGetProductService
{
    private IProductRepository _productRepository;

    public GetProductService(
        IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }

    public IList<Product> GetProductById(int id)
    {
        return _productRepository.Get(id);
    }
}
```


Setting up StructureMap

```
ObjectFactory.Initialize(x =>
{
    x.ForRequestedType<IGetProductService>()
     .TheDefaultIsConcreteType<GetProductService>();
});
```


StructureMap - Conventions

Automatically map "ISomething" interface to "Something" class

```
ObjectFactory.Initialize(x =>
{
    x.Scan(scan =>
    {
        scan.WithDefaultConventions();
        scan.AssemblyContainingType<IProductRepository>();
    });
});
```

StructureMap - Initialization

We just reduced duplication and complexity by setting this up here!

```
ObjectFactory.Initialize(x =>
{
    x.ForRequestedType<IProductRepository>()
     .TheDefaultIsConcreteType<ProductRepository>()
     .OnCreation(repository =>
         repository.ConnectionString =
         ConfigurationManager.AppSettings["MainDB"]);
});
```


StructureMap - Singletons

There will only ever be one IProductCache

We don't violate DIP by having static variables

```
ObjectFactory.Initialize(x =>
{
    x.ForRequestedType<IProductCache>()
     .TheDefaultIsConcreteType<ProductCache>()
     .AsSingletons();
});
```


StructureMap – Object Lifetimes

InstanceScope.Hybrid means that we will only have one of these objects per request (web) or thread (in our tests)

Testing will be easier because we won't reference Thread.CurrentPrincipal

```
ObjectFactory.Initialize(x =>
{
    x.ForRequestedType<ICurrentUser>()
     .CacheBy(InstanceScope.Hybrid)
     .TheDefault.Is.ConstructedBy(
        c => new CurrentUser(Thread.CurrentPrincipal));
});
```


DI Container Rules

Don't "new" up anything that is a dependency

- Don't new up classes that you want to create a fake for in a test
- Do new up entity objects
- Do new up value types (e.g. string, DateTime, etc.)
- Do new up .NET Framework types (e.g. SqlConnection)

Entity objects should not have dependencies

If you have to have static variables, isolate them behind the DI container (e.g. example in previous slide)

Use ObjectFactory.GetInstance() to create objects when you can't take them in as constructor parameters

Don't use the DI container when writing *unit* tests (usually)

Resources

References:

A Page from Robert C Martin's blogs.

– http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

An Audio file with Robert C Martin

– https://s3.amazonaws.com/hanselminutes/hanselminutes_0145.mp3

– http://www.hanselminutes.com/145/solid-principles-with-uncle-bob-robert-c-martin

"Draw Five"

Draw Five is a card game where you draw five cards and get points based on the cards that you draw. You can also view the high scores and save high scores to a database.

When drawing cards
- 5 cards are drawn from a standard 52-card deck (keep in mind that the same card cannot be drawn twice)
- should show the user what cards they drew

When scoring the draw
- face cards = 10
- aces = 15
- numbers = number value

in addition to base score:
- each pair +50
- three of a kind +150
- four of a kind +300
- each spade +1

When showing the high scores
- should show the top 5 scores (name and score)

When saving a high score
- should save the name (entered by the user) and the score

==========

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

"Blackjack"

In this simple version of blackjack, you just draw two cards and calculate the score for the two cards (no drawing of extra cards). High scores are not saved in Blackjack.

When drawing cards
- 5 cards are drawn from a standard 52-card deck (keep in mind that the same card cannot be drawn twice)
- should show the user what cards they drew

When scoring the draw
- face cards = 10
- aces = 11
- numbers = number value

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

change #1

- Draw Five has 2 Jokers in the deck
- jokers = 20 pts each
- two jokers = 200 pt bonus (in addition to the 20 pts for each Joker)

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

change #2

- in Draw Five:
  - a "run" of 3 sequential cards is worth 50 pts
  - a "run" of 4 sequential cards is worth 100 pts
  - a "run" of 5 sequential cards is worth 150 pts
  - the order of cards for "runs" is: A, 2,3,4,5,6,7,8,9,10,J,Q,K,A (notice the Ace can be used at either end)

Exercise

- create an interface that will allow people to write their own scoring system as a plugin

Thank You

The slides are based on:

Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

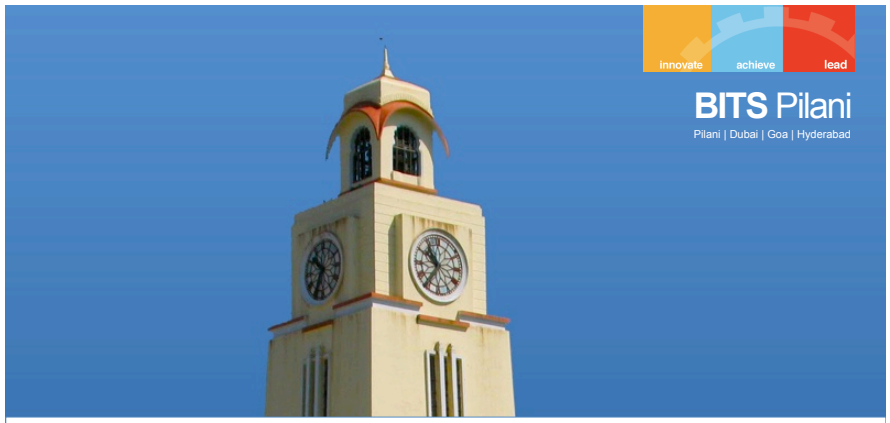
Some content for these slides have been taken from:

JON KRUGER

http://jonkruger.com/blog/oopsolid-talk-links-and-resources/

Outline

No	Title
CS7.1.1	Introducing GoF Patterns
CS7.1.2	Explain Adapter & Factory Patterns with help of coding example
CS7.1.3	Showcase the use of above patterns in PoS System



Introducing GoF Patterns

The "gang of four" (GoF)

Design Patterns
Elements of Reusable Object Oriented Software
Erich Gamma | Richard Helm | Ralph Johnson | John Vlissides

- Solutions to different classes of problems, in C++ & Smalltalk
- Problems and solutions are broadly applicable, used by many people over many years
- Patterns suggest opportunities for reuse in analysis, design and programming
- GOF presents each pattern in a structured format

Elements of Design Patterns

Design patterns have 4 essential elements:

- Pattern name:
  - increases vocabulary of designers
- Problem:
  - intent, context, when to apply
- Solution:
  - UML-like structure, abstract code
- Consequences:
  - results and tradeoffs

Design Patterns are NOT

Data structures that can be encoded in classes and reused *as is* (i.e., linked lists, hash tables)

Complex domain-specific designs
(for an entire application or subsystem)

They are:

- "Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

Three Types of GoF Patterns

Creational patterns:
- Deal with initializing and configuring objects

Structural patterns:
- Composition of classes or objects
- Decouple interface and implementation of classes

Behavioral patterns:
- Deal with dynamic interactions among societies of objects
- How they distribute responsibility

Creational Patterns

Purpose:

Creational patterns concern the process of object creation.

Scope: CLASSES

Simple Factory

Scope: OBJECTS

Abstract Factory

Builder

Prototype

Singleton

Creational Patterns…

Purpose:

Creational patterns concern the process of object creation.

Scope: CLASSES

Factory Method:

- Create specialized, complex objects

Ref. Factory Method (107) Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

…Creational Patterns…

Purpose:

Creational patterns concern the process of object creation.

Scope: OBJECTS

Abstract Factory:

- Create a family of specialized factories

Ref. Abstract Factory (87) Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder:

- Construct a complex object step by step

Ref. Builder (97) Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Prototype: ….

…Creational Patterns

Purpose:

Creational patterns concern the process of object creation.

Scope: OBJECTS

….

Prototype:

- Clone new instances from a prototype

Ref. Prototype (117) Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton:

- Guarantee access to a singular (sole) instance.

Ref. Singleton (127) Ensure a class only has one instance, and provide a global point of access to it.

(Lazy initialization: Delay costly creation until it is needed)

Structural Patterns…

Purpose:

Structural patterns deal with the composition of classes or objects.

Assemble objects to realize new functionality

- Exploit flexibility of object composition at run-time
- Not possible with static class composition

Scope: CLASSES

Adapter

Scope: OBJECTS

Bridge
Composite
Decorator
Façade
Flyweight
Proxy

Structural Patterns…

Purpose:

Structural patterns deal with the composition of classes or objects.

Assemble objects to realize new functionality

- Exploit flexibility of object composition at run-time
- Not possible with static class composition

Scope: CLASSES

Adapter:

– Converts interface of a class into one that clients expect.

Ref. Adapter (139) Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

…Structural Patterns…

Purpose:

Structural patterns deal with the composition of classes or objects.

Assemble objects to realize new functionality

- Exploit flexibility of object composition at run-time
- Not possible with static class composition

Scope: OBJECTS

Bridge:

– Links abstraction with many possible implementations

Ref. Bridge (151) Decouple an abstraction from its implementation so that the two can vary independently.

Composite:

– Represents part-whole hierarchies as tree structures

Ref. Composite (163) Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator:

– Attach additional responsibilities to object dynamically

Ref. Decorator (175) Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade: ……

…Structural Patterns

Purpose:
Structural patterns deal with the composition of classes or objects.
Assemble objects to realize new functionality
- Exploit flexibility of object composition at run-time
- Not possible with static class composition

Scope: OBJECTS

Facade:
– Simplifies the interface for a subsystem
Ref. Facade (185) Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight:
– Shares many fine-grained objects efficiently
Ref. Flyweight (195) Use sharing to support large numbers of fine-grained objects efficiently.

Proxy:
– Provides a surrogate or placeholder for another object to control access to it
Ref. Proxy (207) Provide a surrogate or placeholder for another object to control access to it.

Structural patterns

Example: Proxy
- ***Proxy*** acts as convenient surrogate or placeholder for another object.
- Examples?
  - Remote Proxy: local representative for object in a different address space
  - Virtual Proxy: represent large object that should be loaded on demand
  - Protected Proxy: protect access to the original object

Behavioral Patterns

Purpose:
Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility..

Scope: CLASSES
Interpreter
Template Method

Scope: OBJECTS
Chain of Responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Visitor

Behavioral Patterns…

Purpose:
Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility..

Scope: CLASSES

Interpreter:
– Language interpreter for a small grammar
Ref. Interpreter (243) Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Template Method:
– Algorithm with some steps supplied by derived class
Ref. Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

…Behavioral Patterns…

Scope: OBJECTS

Chain of Responsibility:
– Request delegated to the responsible service provider
Ref. Chain of Responsibility (223) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command:
– Request or Action is first-class object, hence storable
Ref. Command (233) Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Iterator:
– Aggregate and access elements sequentially
Ref. Iterator (257) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator ….

…Behavioral Patterns…

Scope: OBJECTS

Mediator:
– Coordinates interactions between its associates
Ref. Mediator (273) Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento:
– Snapshot captures and restores object states privately
Ref. Memento (283) Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer:
– Observers update automatically when observed object changes
Ref. Observer (293) Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

State …..

…Behavioral Patterns

Scope: OBJECTS

State:
– Object whose behavior depends on its state
Ref. State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy:
– Abstraction for selecting one of many algorithms
Ref. Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Visitor:
– Operations applied to elements of a heterogeneous object structure
Ref. Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Patterns in software libraries

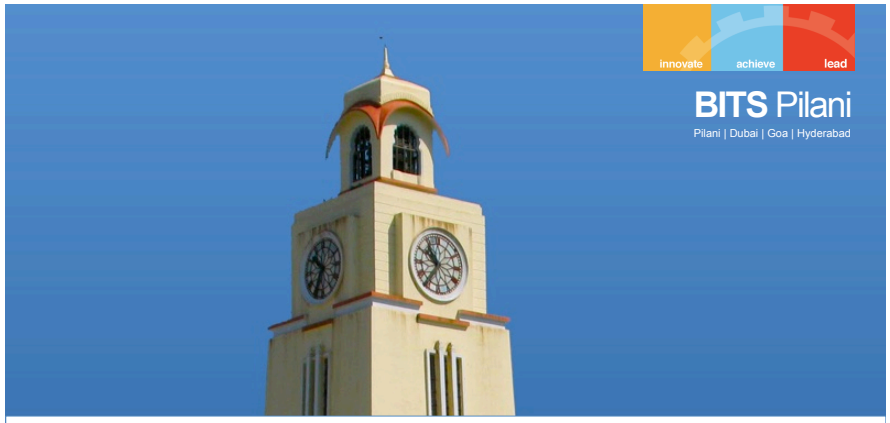
- AWT and Swing use Observer pattern
- Iterator pattern in C++ template library & JDK
- Façade pattern used in many student-oriented libraries to simplify more complicated libraries!
- Bridge and other patterns recurs in middleware for distributed computing frameworks
- …

More software patterns

- Language idioms **(**low level, C++): Jim Coplein, Scott Meyers
  - I.e., when should you define a virtual destructor?
- Architectural (systems design): layers, reflection, broker
  - Reflection makes classes self-aware, their structure and behavior accessible for adaptation and change: Meta-level provides self-representation, base level defines the application logic
- *Java Enterprise Design Patterns* (distributed transactions and databases)
  - E.g., ACID Transaction: *A*tomicity (restoring an object after a failed transaction), *C*onsistency, *I*solation, and *D*urability
- **Analysis patterns** (recurring & reusable analysis models, from various domains, i.e., accounting, financial trading, health)
- **Process patterns** (software process & organization)

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary



Adapter Patterns

Adapter pattern

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

Solution: Convert original interface component into another one through an intermediate adapter.

Use interfaces and polymorphism to add indirection to varying APIs

Adapters

Cable adapter: adapts plug to foreign wall outlet

OO Programming; Want to adapt class to foreign interface type

Example: Add CarIcon to container

Problem: Containers take components, not icons

Solution: Create an adapter that adapts Icon to Component

Adapter - Understanding

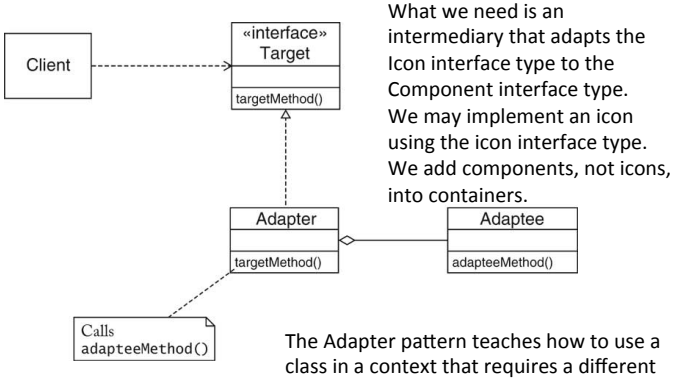
Context

1. You want to use an existing class (adaptee) without modifying it.
2. The context in which you want to use the class requires target interface that is different from that of the adaptee.
3. The target interface and the adaptee interface are conceptually related.

Solution

1. Define an adapter class that implements the target interface.
2. The adapter class holds a reference to the adaptee. It translates target methods to adaptee methods.
3. The client wraps the adaptee into an adapter class object.

Adapter – Class Diagram



What we need is an intermediary that adapts the Icon interface type to the Component interface type.
We may implement an icon using the icon interface type.
We add components, not icons, into containers.

The Adapter pattern teaches how to use a class in a context that requires a different interface.

Icon Adapter – an example

Name in Design Pattern	Actual Name (Icon->Component)
`Adaptee`	`Icon`
`Target`	`JComponent`
`Adapter`	`IconAdapter`
`Client`	The class that wants to add icons into a container
`targetMethod()`	`paintComponent(), getPreferredSize()`
`adapteeMethod()`	`paintIcon(), getIconWidth(), getIconHeight()`

Java Stream Library - Example

In stream library
Input streams read bytes
Readers read characters
Non-ASCII encoding: multiple bytes per char
System.in is a stream
What if you want to read characters?
Adapt stream to reader
InputStreamReader

Reader reader = new InputStreamReader(System.in);
// uses the default character encoding
Or
Reader reader = new InputStreamReader(System.in,"UTF-8");
// uses the specific character encoding

Java Stream Library - Example

Name in Design Pattern	Actual Name (Stream->Reader)
`Adaptee`	`InputStream`
`Target`	`Reader`
`Adapter`	`InputStreamReader`
`Client`	The class that wants to read text from an input stream
`targetMethod()`	`read (reading a character)`
`adapteeMethod()`	`read (reading a byte)`

Using an Adapter: adapt postSale request to SOAP XML interface

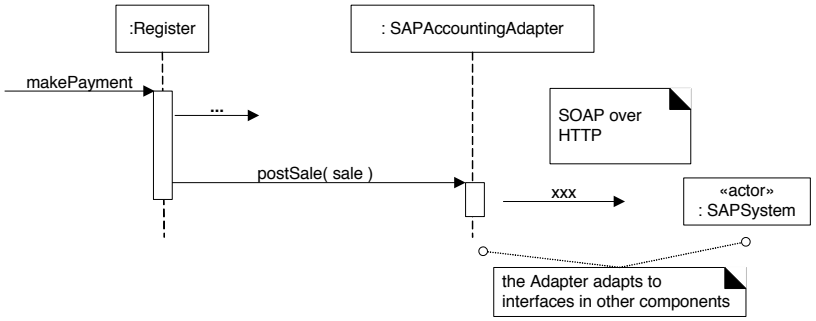


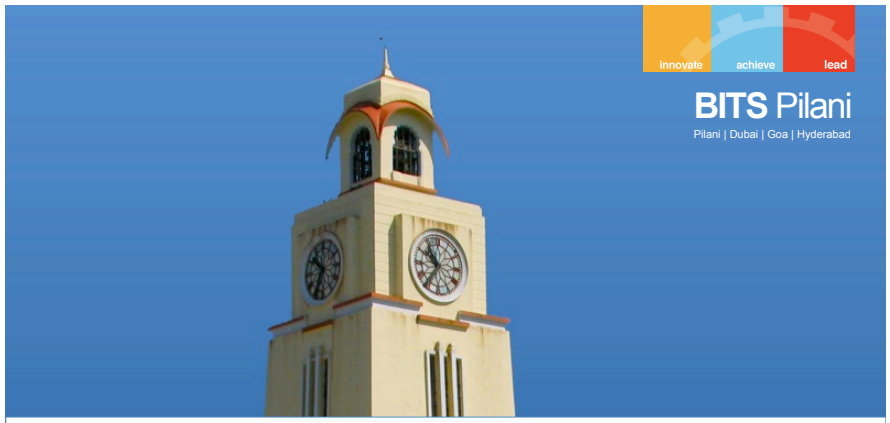
Fig. 26.2

Benefits of Adapter pattern

Reduces coupling to implementation specific details

Polymorphism and Indirection reveals essential behavior provided

Including name of design pattern in new class (e.g., TaxMasterAdapter) in class diagrams and code communicates to other developers in terms of known design patterns



Factory Patterns

Factory

- Every collection can produce an iterator
  Iterator iter = list.iterator()
- Why not use constructors?
  Iterator iter = new LinkedListIterator(list);
- Drawback: not generic
  Collection coll = ...;
  Iterator iter = new ???(coll);
- Factory method works for all collections
  Iterator iter = coll.iterator();
- Polymorphism!

Simple Factory pattern

Context/Problem

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

Solution

- Create a Pure Fabrication to handle the creation

Factory Method

- Teaches how to supply a method that can be overridden to create objects of varying types.

Factory Method

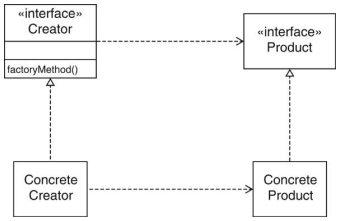
Context

1. A type (the creator) creates objects of another type (the product).
2. Subclasses of the creator type need to create different kinds of product objects.
3. Clients do not need to know the exact type of product objects.

Solution

1. Define a creator type that expresses the commonality of all creators.
2. Define a product type that expresses the commonality of all products.
3. Define a method, called the factory method, in the creator type. The factory method yields a product object.
4. Each concrete creator class implements the factory method so that it returns an object of a concrete product class.

Factory Method Pattern – Class Diagram




```
LinkedList list = ....;
Iterator iter = new
LinkedListIterator(list)
Collection coll = ....;
Uterator iter = coll.iterator();
```

Java collection produces an iterator for traversing ints elements.
The Collection interface type defines a method Iterator iterator()

Each subclass of Collection (such as LinkedList or our own Queue class) omplements that method in a different way.
Each iterator method returns an object of a class that implements the Iterator interface type, but the implementation of these sub-types are completely different.
An iterator through a linked lis keeps a reference to the last visited node. Our queue iterator keeps an index of the last visited array element.

Factory Method - iterators

Name in Design Pattern	Actual Name (iterator)
`Creator`	`Collection`
`ConcreteCreator`	A subclass of `Collection`
`factoryMethod()`	`iterator()`
`Product`	`Iterator`
`ConcreteProduct`	A subclass of `Iterator` (which is often anonymous)

Advantages of Factory Objects?

- Separates responsibility of complex creation into cohesive helper classes
- Hides complex creation logic, such as initialization from a file
- Handles memory management strategies, such or recycling or caching

Adapter pattern for PoS System

Adapter (GoF) PoS

Problem:

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

Solution:

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

We are adding a level of Indirection:

- Add an objects that adapt the varying external interfaces to a consistent interface used by the application.

POS example: Instantiate adapters for external services

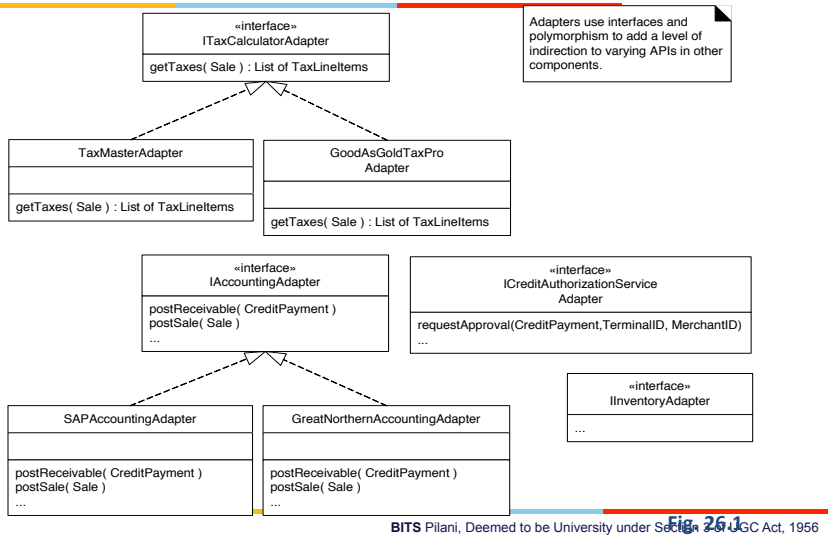
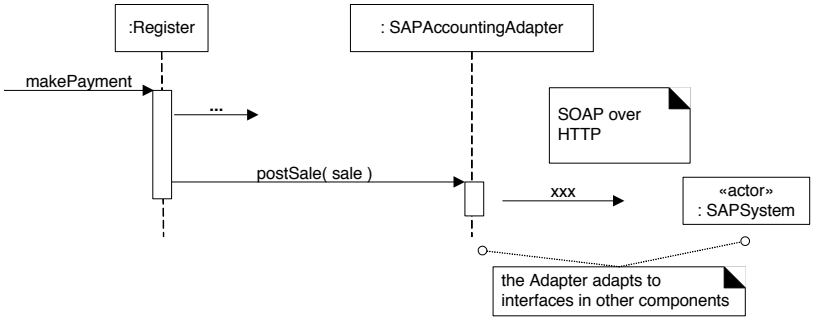


Fig. 26.1

Using a Adapter



Adapter supports Protected Variation with respect to changing external interfaces or third-party packages through use of an Indirection object that applies interfaces and Polymorphism.

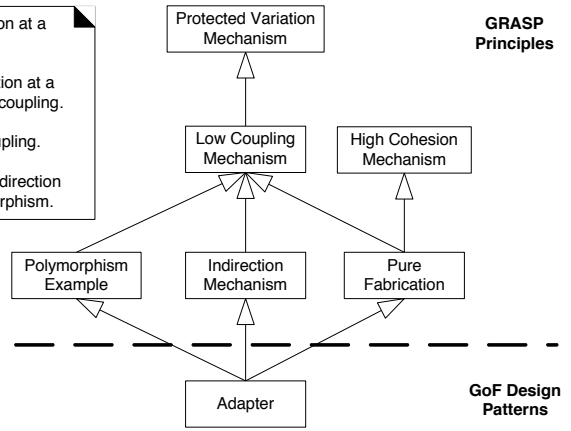
Relating Adapter to some GRASP principles.

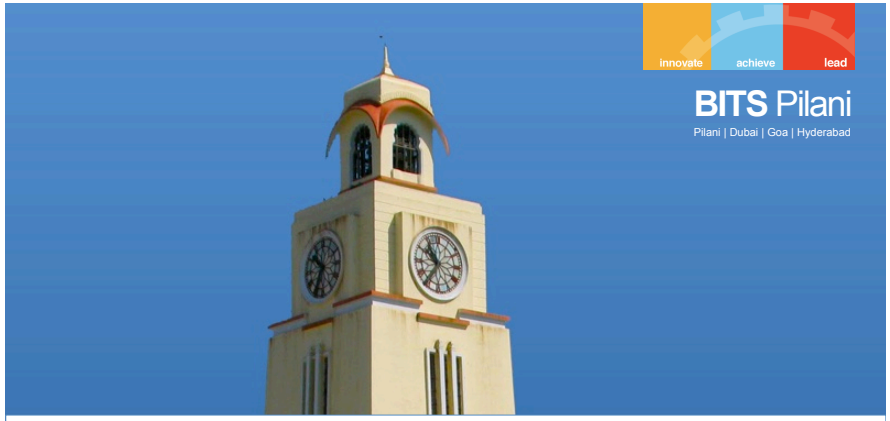
Low coupling is a way to achieve protection at a variation point.

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.





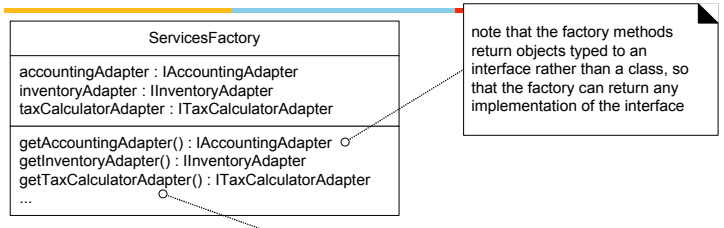
Factory pattern for PoS System

Factory

Simple Factory or Concrete Facory

- Simplification of GoF Abstract Factory Pattern
- In the previous case- Who creates the Adapter?
- The is a Pure Fabrication (Factory) is designed for creating objects.
  - The responsibility of creating cmplex objects is assigned to cohesive helper objects
  - Hide complex creation logic
  - Introduce performance-enhancing strategies such as object caching and recycling.

Factory can create different objects from a file




```
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;
```

The logic to decide which class to create is resolved by reading in the class name from an external storage and then dyanamically loading the class- data driven design.

Thank You

The slides are based on:

Applying UML and Patterns

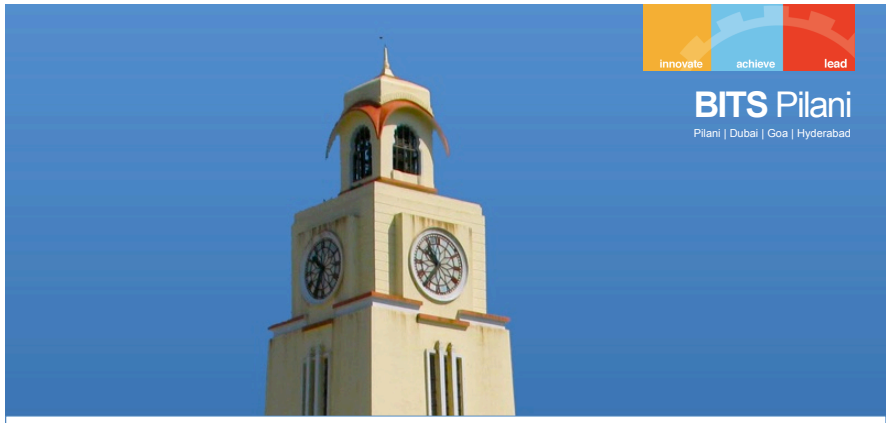
An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

innovate achieve lead

No	Title
CS7.2.1	Explain Singleton & Strategy Patterns with help of coding example
CS7.2.2	Showcase the use of above patterns in PoS System



Singleton Patterns

- A singleton class has exactly one instance

Singleton - Notes

innovate achieve lead

- "Random" number generator generates predictable stream of numbers
- Example: seed = (seed * 25214903917 + 11) % 2^{48}
- Convenient for debugging: can reproduce number sequence
- Only if all clients use *the same random number generator*
- Singleton class = class with one instance

Singleton pattern (creational)

A class with just instance and provide a global point of access
- *Global Variables can be dangerous!*
  (side effects, break information hiding)


```
class Singleton
{ public:
     static Singleton* getInstance();
  protected: //Why are the following protected?
     Singleton();
     Singleton(const Singleton&);
     Singleton& operator= (const Singleton&);
 private: static Singleton* instance;
};
Singleton *p2 = p1->getInstance();
```


Singleton – Code Example

```
public class SingleRandom
  {
    private SingleRandom() { generator = new Random(); }
    public void setSeed(int seed) {
        generator.setSeed(seed); }
    public int nextInt() { return generator.nextInt(); }
    public static SingleRandom getInstance() {
        return instance; }
    private Random generator;
    private static SingleRandom instance =
       new SingleRandom();
  }
```

This class constructs as well as returns an instance of itself over a static method.

Each filed has to be declared somewhere. This is only a convenient choice.

Static field ("global" variable) instance stores a reference to the unique SingleRandom object.

Calling the Static getInstance method

How a client obtains a SingleRandom object:

int randomNumber = SingleRandom.getInstance.nextInt();

- Static field are initialized when the virtual machine loads the class.
- A class must be loaded before any of its methods can be called.
- The static instance field is initialised with the singleton object before the first client call to the getInstance method occurs.

We can delay the construction of the instance until the getInstance method is called for the first time, by setting the initial value to "null":

```
Public static synchronized SingleRandom getinstance()
{
  if (instance==null) instance = new SingleRandom();
  return instance;
}
```

// Synchronized is required to avoid a race condition if two threads call at the same time.

The SINGLETON Pattern

Context

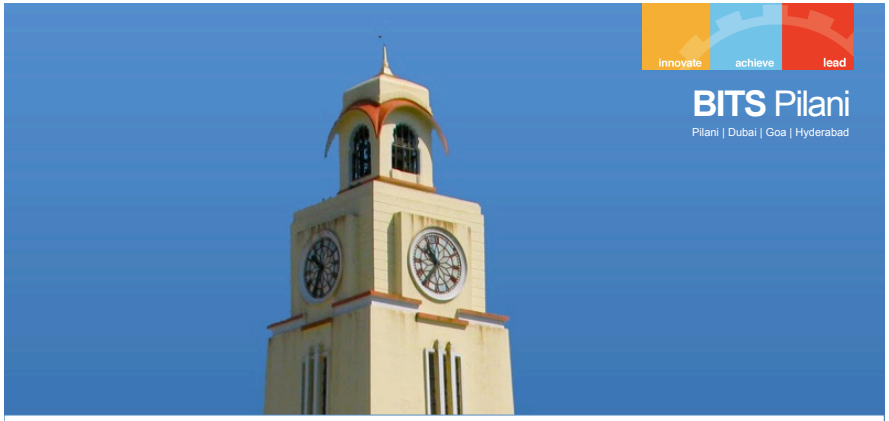
1. All clients need to access a single shared instance of a class.
2. You want to ensure that no additional instances can be created accidentally.

Solution

1. Define a class with a private constructor.
2. The class constructs a single instance of itself.
3. Supply a static method that returns a reference to the single instance.

NOT A Singleton

- Toolkit used for determining screen size, other window system parameters
- Toolkit class returns default toolkit
  Toolkit kit = Toolkit.getDefaultToolkit();
- Not a singleton--can get other instances of Toolkit
- Math class not example of singleton pattern
- *No* objects of class Math are created. It is a utility class; a class with only static methods.



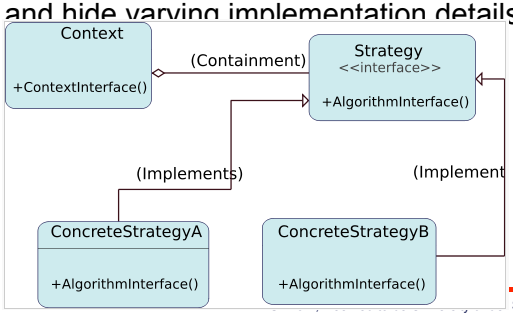
Strategy Patterns

Strategy design pattern

Problem: How to design a family of algorithms or policies that are essentially the same but vary in details?

Solution: "Define a family of algorithms, encapsulate each one, and make them interchangeable." [Gamma, p315]

Use abstraction and polymorphism to show high level algorithm and hide varying implementation details



Singleton pattern for PoS System

Factory can create different objects from a file

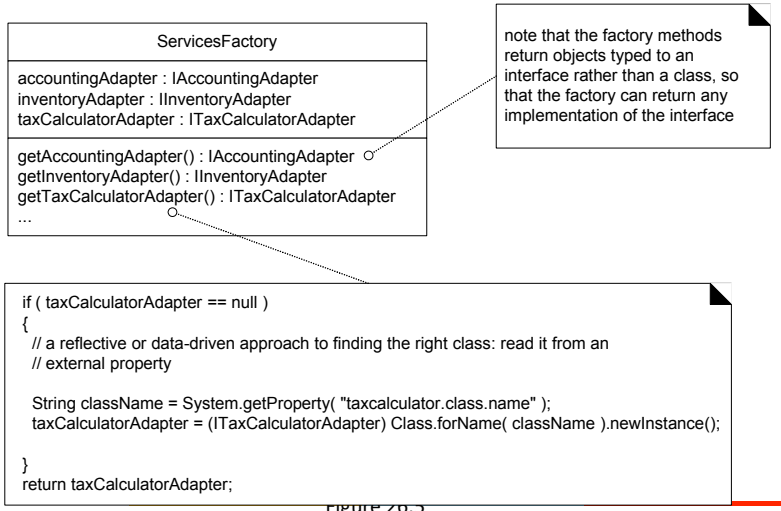
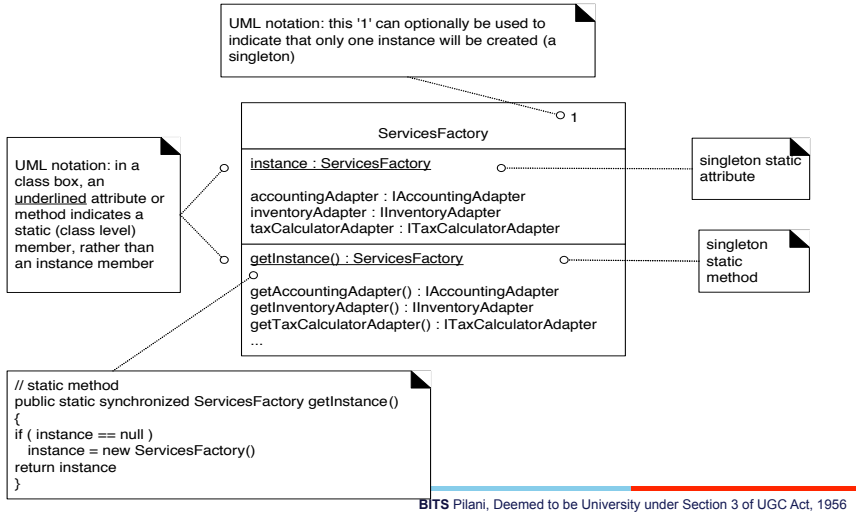


Figure 26.5

Use Singleton to create a Factory



Adapter, Factory and Singleton working together

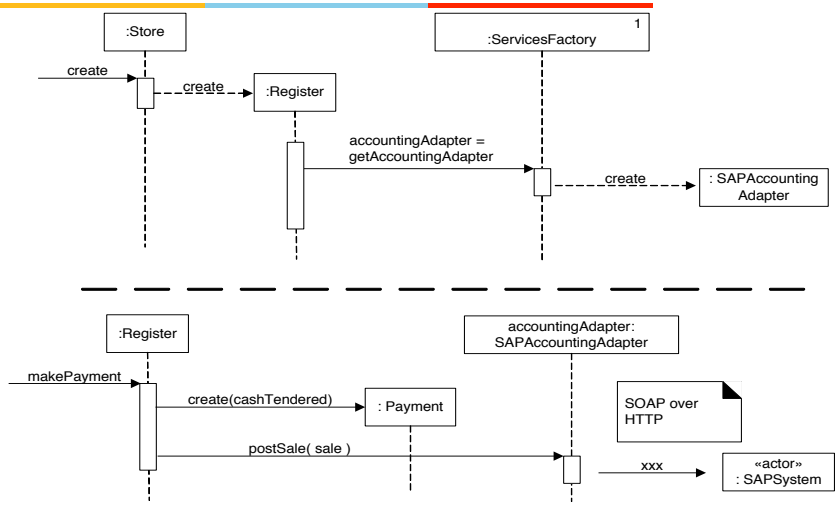


Figure 26.8

Strategy pattern for PoS System

Multiple *SalePricingStrategy* classes with polymorphic *getTotal* method

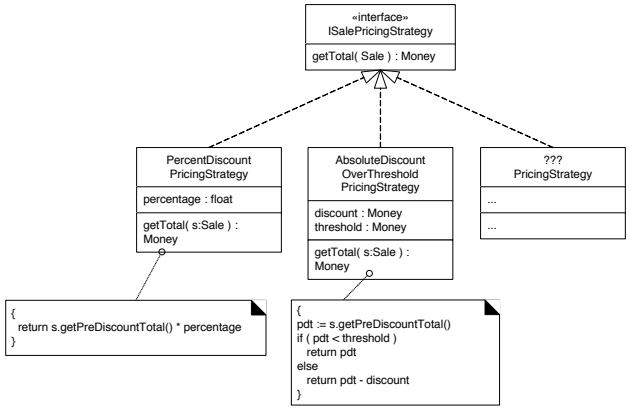
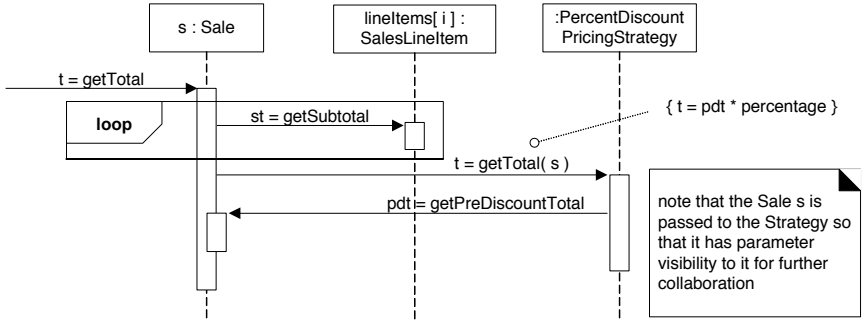
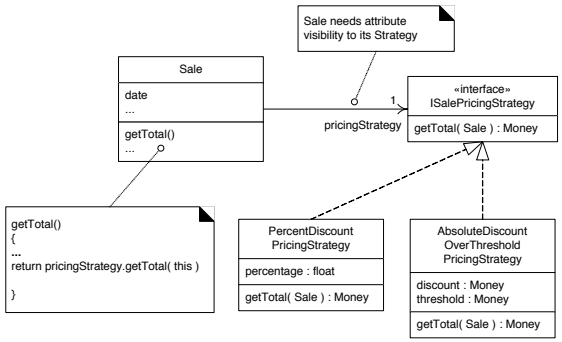


Figure 26.9

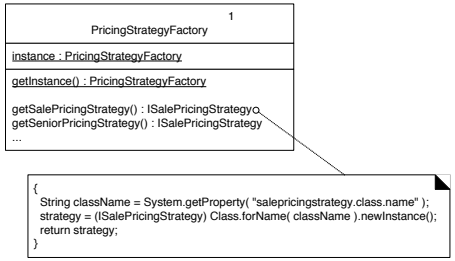
Strategy in collaboration



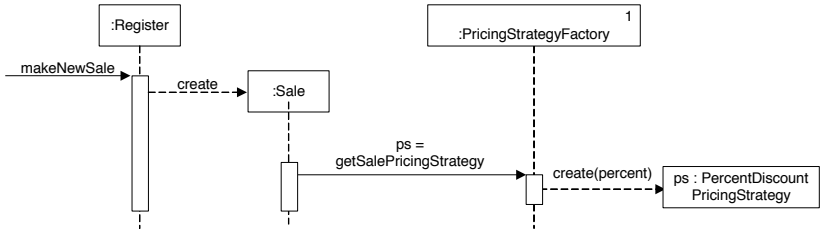
Context Object needs attribute visibility to its strategy



Factory for Strategies



Creating a strategy



Thank You

The slides are based on:

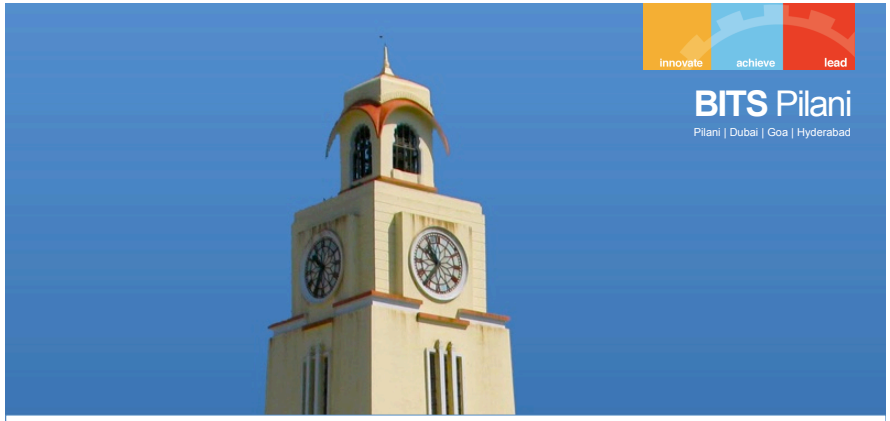
Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

No	Title
CS8.1.1	Explain Composite & Facade Patterns with help of coding example
CS8.1.2	Showcase the use of above patterns in PoS System



Composite Patterns

Composite Pattern

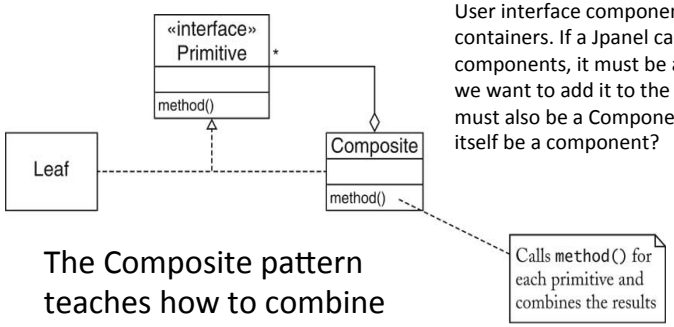
Context

1. Primitive objects can be combined to composite objects
2. Clients treat a composite object as a primitive object

Solution

1. Define an interface type that is an abstraction for the primitive objects
2. Composite object collects primitive objects
3. Composite and primitive classes implement same interface type.
4. When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results

Composite Pattern



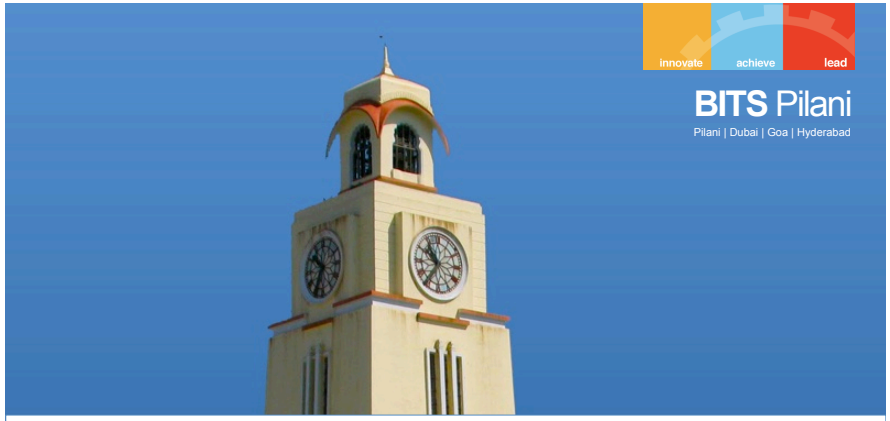
User interface components are contained in containers. If a Jpanel can contain other components, it must be a Container. But if we want to add it to the content pane, it must also be a Component. Can a container itself be a component?

The Composite pattern teaches how to combine several objects into an object that has the same behaviour as its parts.

Composite Pattern

Name in Design Pattern	Actual Name (AWT components)
`Primitive`	`Component`
`Composite`	`Container`
`Leaf`	a component without children (e.g. `JButton`)
`method()`	a method of `Component` (e.g. `getPreferredSize`)

The Container class contains components, and it also extends the Component class.
Note: The method in the composite object must apply the method to all its primitive objects and the combine the results.
You may study how the size of a container is worked out in collaboration with the layout manager by combining the size of the components.



Facade Patterns

Façade Class

1. Bean usually composed of multiple classes
2. One class nominated as *facade class*
3. Clients use only facade class methods

Façade Pattern

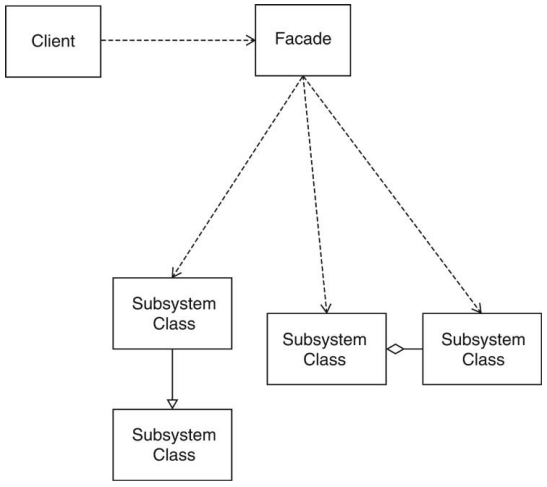
Context

1. A subsystem consists of multiple classes, making it complicated for clients to use
2. Implementer may want to change subsystem classes
3. Want to give a coherent entry point

Solution

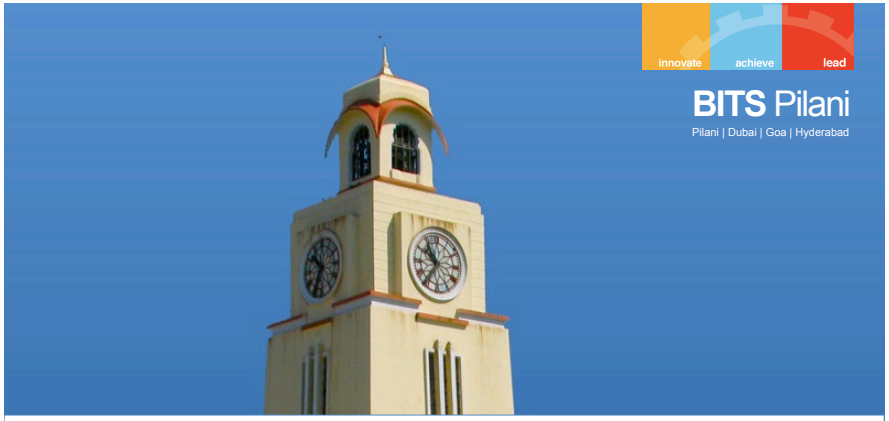
1. Define a facade class that exposes all capabilities of the subsystem as methods
2. The facade methods delegate requests to the subsystem classes
3. The subsystem classes do not know about the facade class

Facade



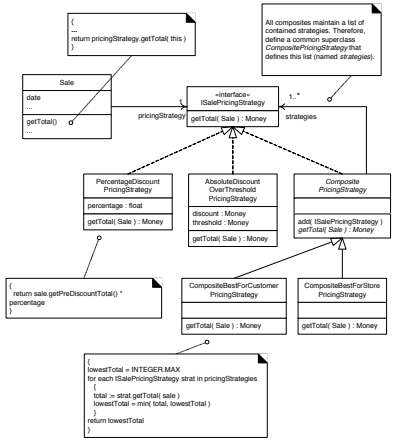
Facade

Name in Design Pattern	Actual Name (Beans)
`Client`	Builder tool
`Facade`	Main bean class with which the tool interacts
`SubsystemClass`	Class used to implement bean functionality

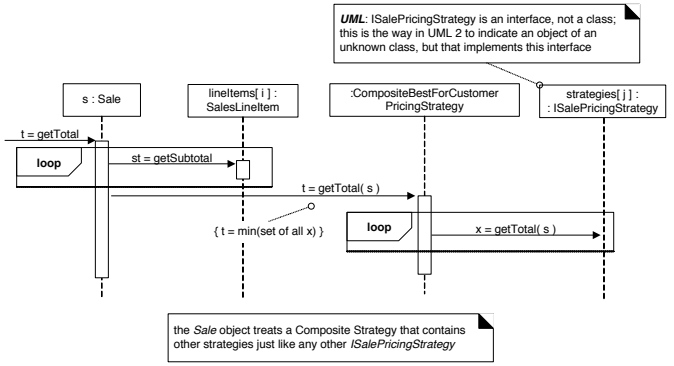


Composite pattern for PoS System

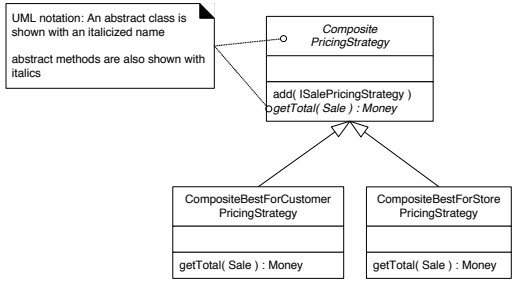
Composite Pattern



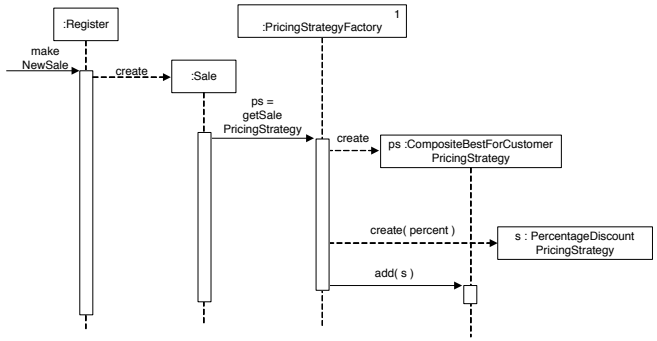
Composite Pattern



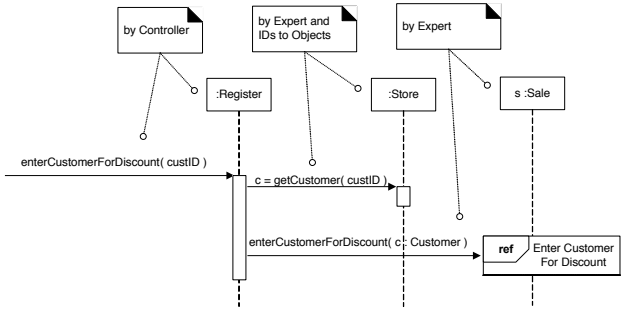
Composite Pattern



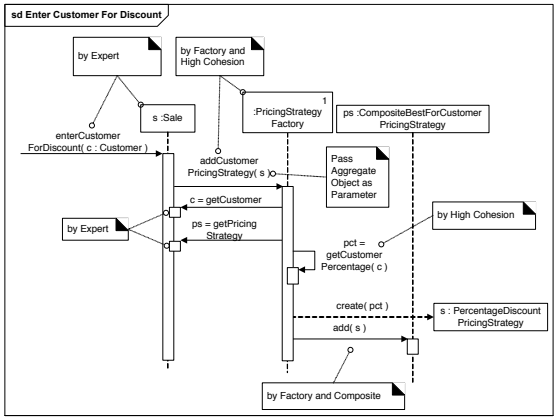
Composite Pattern

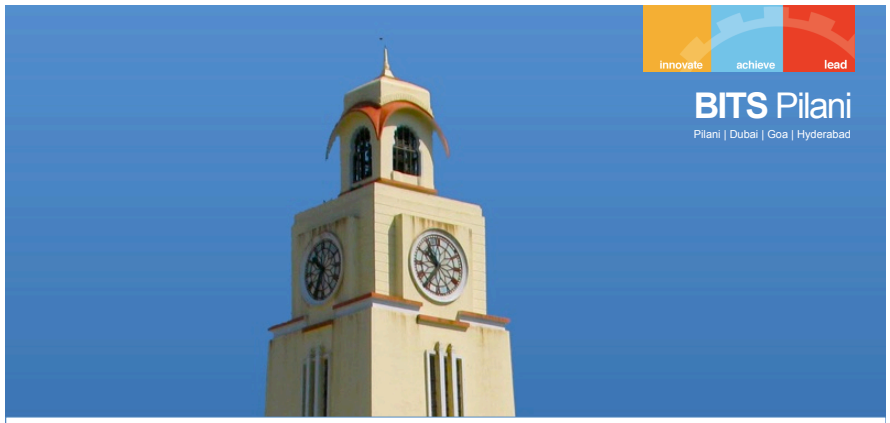


Composite Pattern



Composite Pattern





Facade pattern for PoS System

Façade for PoS

Problem:

1. 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 subsystems, or the implementation of the subsystem may change. What to do?

Solution:

1. 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.

Code Example

```
public class Sale
{
   public void makeLineItem(ProductDescription desc, int quantity)
   {
       /// call to the Facade
       if (POSRuleEngineFacade.getInstance().isInvalid(sli, this))
            return;
        lineItems.add(sli);
    }
} // end of class
```

Facades are often accessed via Singleton.

Facade Notes

1. Facade is a frontend object that is a single point of entry for the services of a subsystem.
2. The implementation and other components of the subsystem are private and cannot be seen by external components.
3. Provides Protected Variation from changes in the implementation of subsystem.
4. Eg Rule-engine subsystem, the specific implementation of which may not even have been decided. Complexity and implementation are hidden.
5. The subsystem hidden by the facade object may contain a large number of objects and even not object oriented solution, yet the client sees only one access point.
6. Separation of Concern – all rule-handling concerns have been delegated to another subsystem,

Thank You

The slides are based on:

Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman

Outline

No	Title
CS8.2.1	Explain concept of event source and event handler. How it is called as Observer as well as Publish-Subscribe Pattern?
CS8.2.2	Showcase the use of above pattern in PoS System

Observer pattern

Intent:

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Used in Model-View-Controller framework

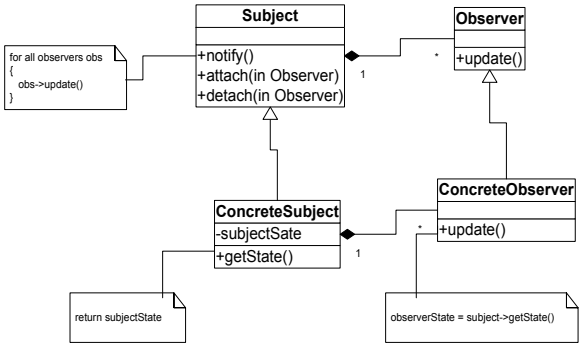
- Model is problem domain
- View is windowing system
- Controller is mouse/keyboard control

How can Observer pattern be used in other applications?

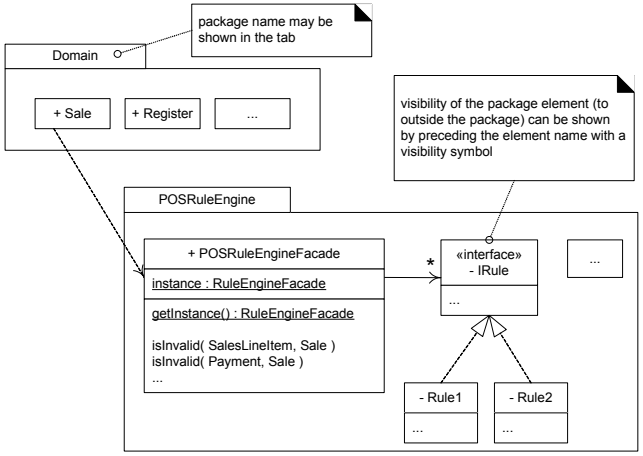
JDK's Abstract Window Toolkit (listeners)

Java's Thread monitors, notify(), etc.

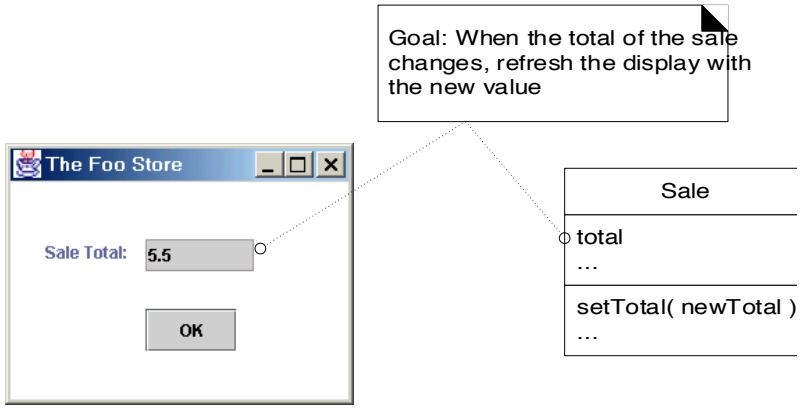
Structure of Observer Pattern



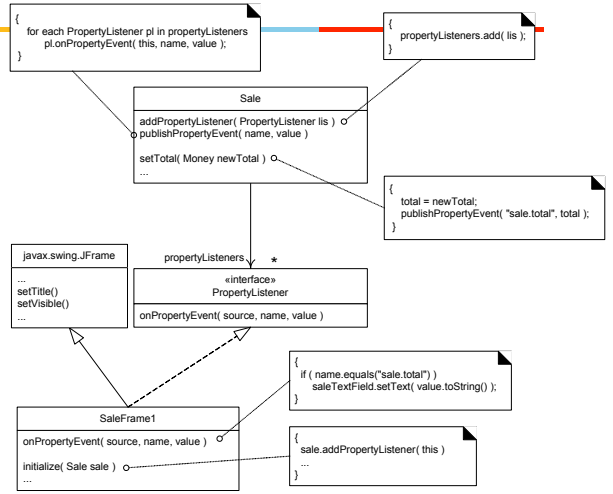
UML Package Diagram for a facade



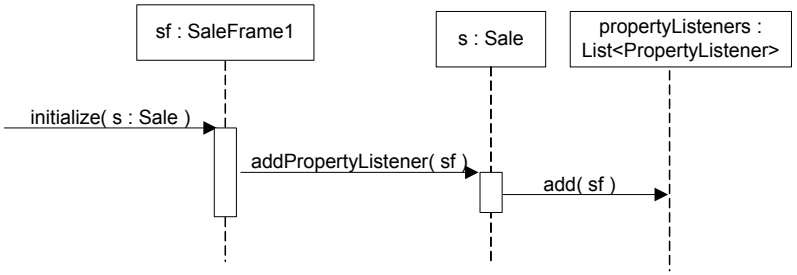
Updating the interface when the sale total changes



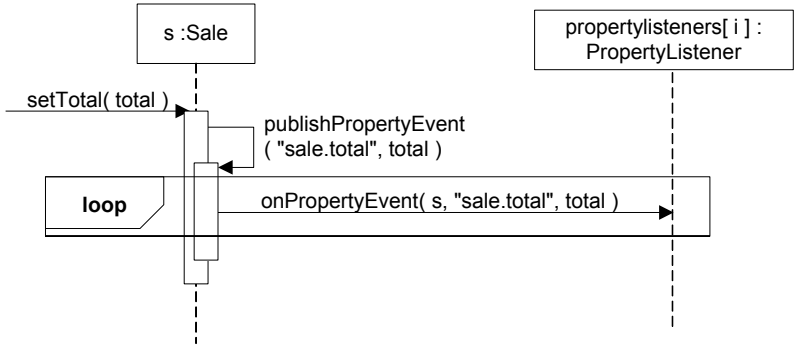
The Observer Pattern



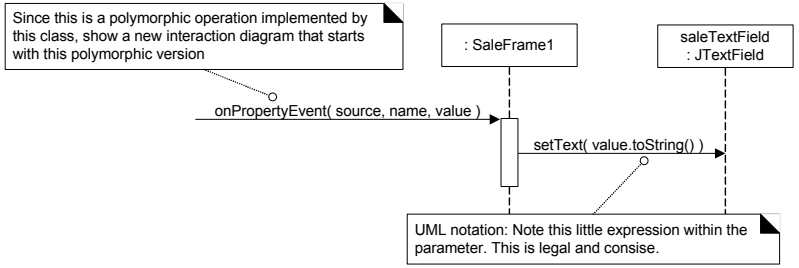
The observer SaleFrame1 subscribes to the publisher Sale



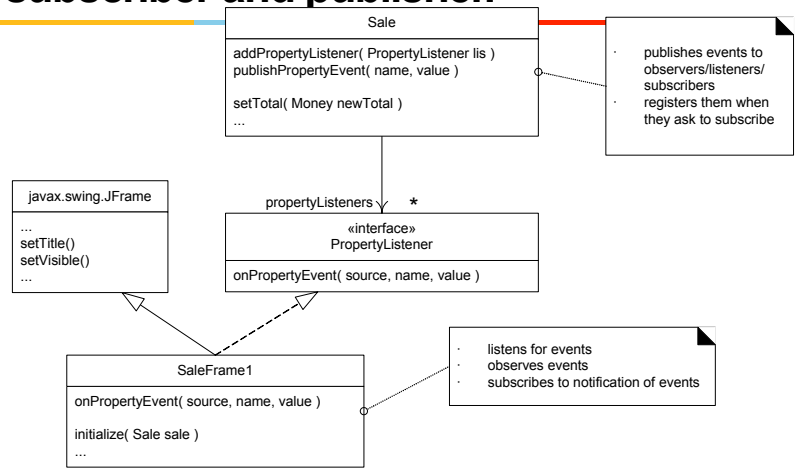
The Sale publishes a property event to all its subscribers



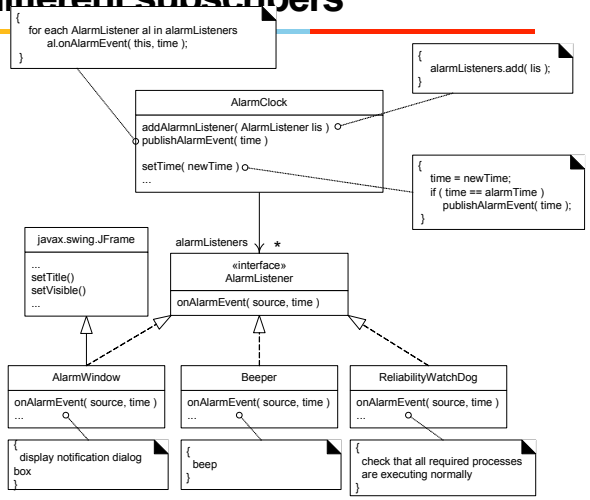
The subscriber SaleFrame1 receives notification of a published event



Who is the observer, listner, subscriber and publisher.



Observer applied to alarm event, with different subscribers



Program

```
Class PropertyEvent extends Event {
    private Object sourceOfEvent;
     private String propertyName;
    private Object oldValue;
    private Object newValue;
     // ...
}
```


Program

```
class Sale {
    private void publishPropertyEvent (
        String name, Object old, Object new ) {
        PropertyEvent evt =
        new PropertyEvent (this, “sale.total”, old, new );
        for each AlarmListener al in alarmListeners
            al.onPropertyEvent( evt );
        }
        //..
}
```


Thank You

The slides are based on:

Applying UML and Patterns

An Introduction to Object-Oriented Analysis and Design and Iterative Development

By Craig Larman