

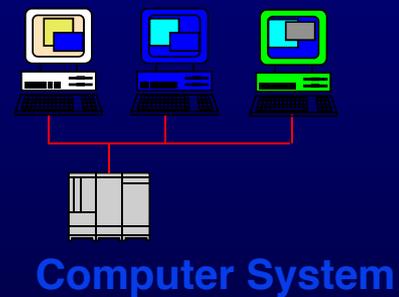
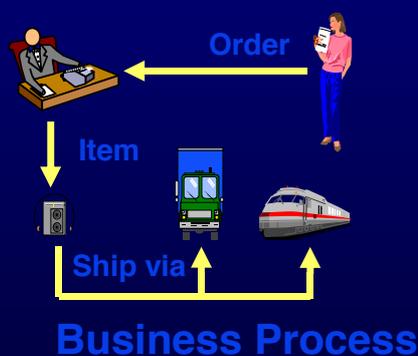
Analysis and Design with UML

*In collaborazione con Paola Turci
(Università di Parma)*

Slide 1

What is Visual Modelling?

*“Modelling captures essential
parts of the system.”
Dr. James Rumbaugh*



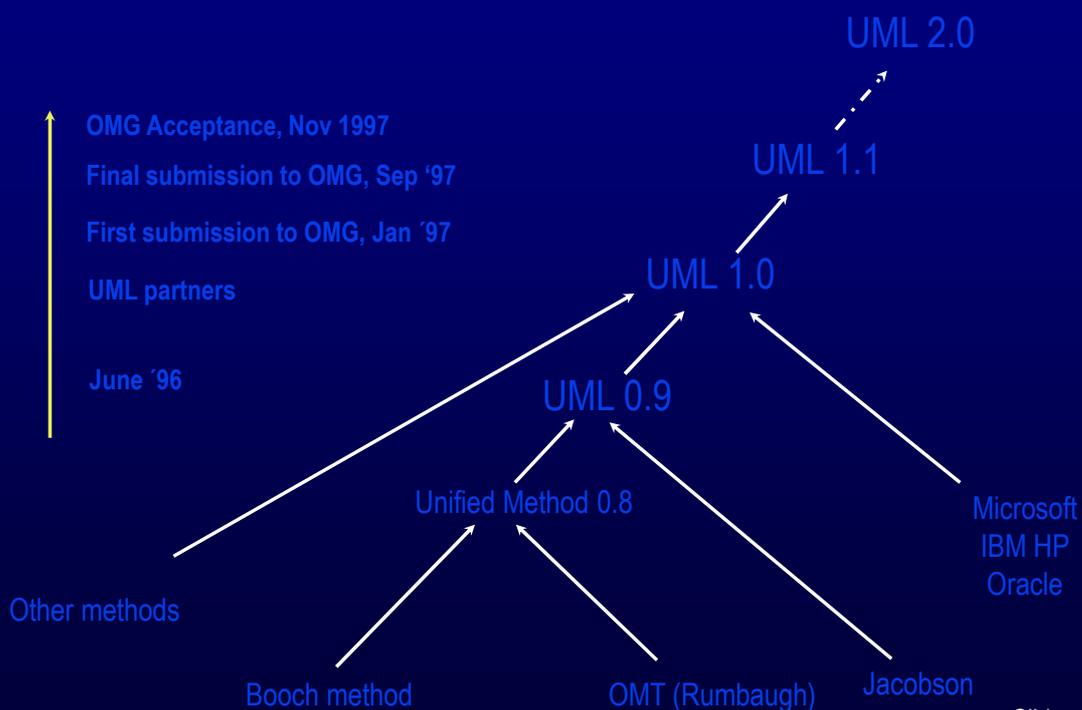
Slide 2

What is UML?

- UML stands for *Unified Modeling Language*
- The UML combines the best from
 - Data Modelling concepts (Entity Relationship Diagrams)
 - Business Modelling (work flow)
 - Object Modelling
 - Component Modelling
- The UML is the standard language for visualizing, specifying, constructing, and documenting the artefacts of a software-intensive system (based on object concepts)
- It can be used with all processes, throughout the development life cycle, and across different implementation technologies
- An OMG (Object Management Group) standard

Slide 3

History of UML



Slide 4

... UML Is Not Enough

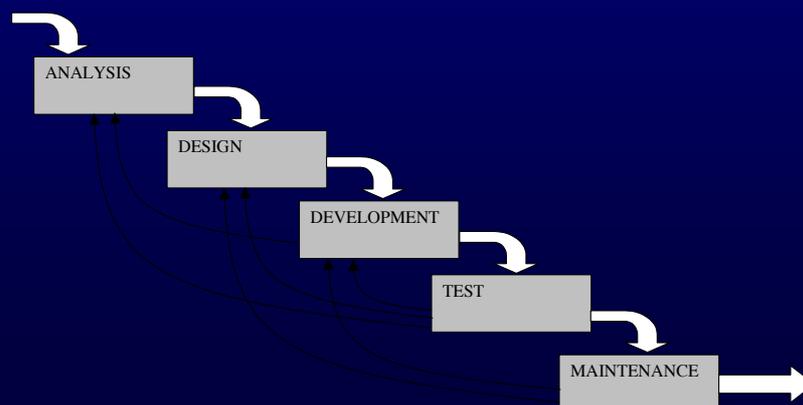


- Most methods consist of both a modeling language and a process
 - The modeling language is the (mainly graphical) notation that methods use to express designs.
 - The key part for communication
 - The process concerns advices on what steps to take in doing a design.

Slide 5

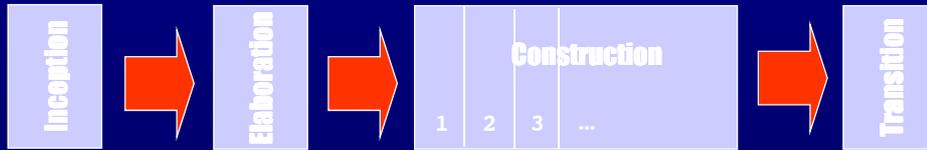
The Classical “Waterfall” Process

- The phases of software development:
 - Independent of programming paradigm;
 - Methodologies are typically organized around this classical process.
 - Inputs, outputs, internal activities of “phases”



Slide 6

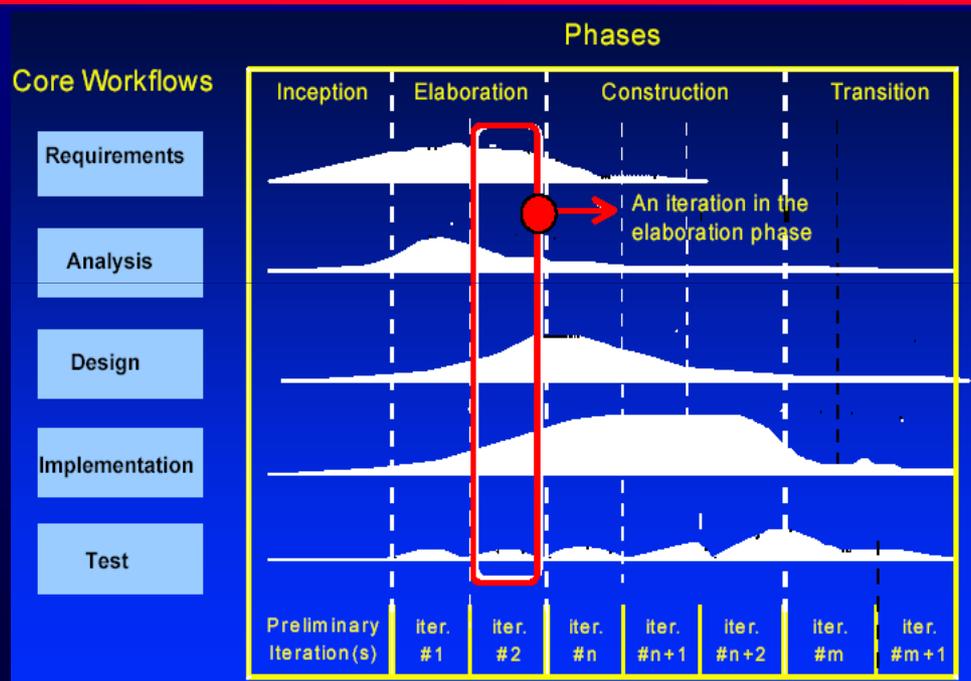
Rational Unified Process (RUP)



- It is an iterative and incremental development process.
 - **Inception:** business rationale for the project and the scope of the project.
 - **Elaboration:** more detailed requirements, high-level analysis and design to establish a baseline architecture, and the plan for construction
 - The **construction** phase consists of many **iterations**. Each iteration is a mini-project: analysis, design, coding, testing, and integration for the use cases assigned to each iteration.
 - **Transition** phase can include beta testing, performance tuning, and user training.

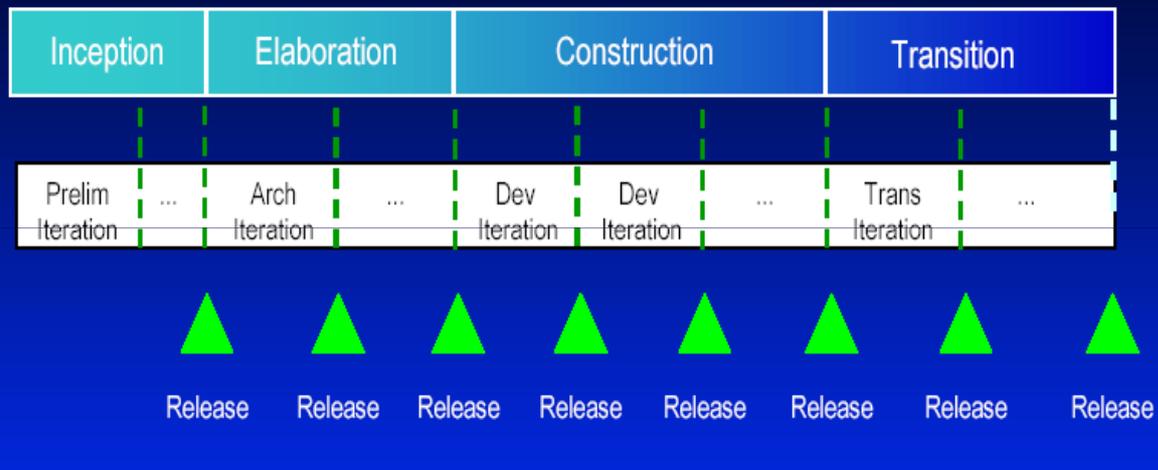
Slide 7

Phases and Workflows



Slide 8

Phases and Iterations



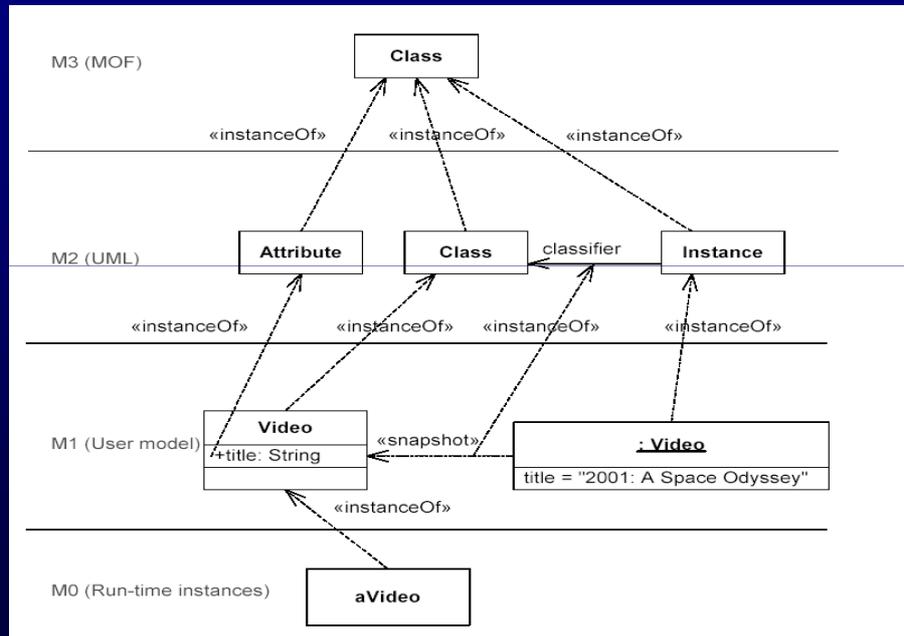
Slide 9

Notation and Metamodels

- UML defines a notation and a meta-model.
- The **notation** is the graphical stuff used in models; it is the syntax of the modelling language.
 - For instance, class diagram notation defines how items and concepts such as class, association, and multiplicity are represented.
 - But ...
this leads to the question of what exactly is meant by an association or multiplicity or even a class. Common usage suggests some informal definitions, but more rigor is needed
- One way to improve the rigor of models without sacrificing their usefulness is to define a **meta-model**: a diagram, usually a class diagram, that defines the notation
 - it defines what is a well-formed model - that is, one that is syntactically correct

Slide 10

UML 2.0 - Four-Layer Metamodel Hierarchy



Source : 3rd revised submission to OMG RFP ad/00-09-01

Slide 11

Why Do Analysis and Design?

“Diagrams are, after all, just pretty pictures. No user is going to thank you for pretty pictures; what a user wants is software that executes”

- ... some motivations:

- **Communication**

in particular communication with **Domain Experts**

- Natural language is too imprecise
- Code is precise but too detailed
- UML allows to achieve a certain amount of precision without get lost in details (to highlight *important* details)

- **Learning OO**

“Object languages *allow* advantages but don't *provide* them. To use these advantages, you have to make the infamous paradigm shift”

Tom Hadfield

- The techniques define in UML were to some degree designed to help people do good OO

Slide 12

Putting the UML to Work

- The case study is a **point-of-sale terminal*** system. POST is a Web-based system used to allow customers to browse through products, record sales and handle payments (used in online-shops)
- We have been requested to create the software to run a POST
- We will use an iterative-incremental development strategy
 - Requirements
 - Object-oriented analysis
 - Object-oriented design
 - Implementation

**Source : Craig Larman, "Applying UML and Patterns"*

Slide 13

ANALYSIS PHASE

Slide 14

Requirements

- Unambiguous description of needs or desires for a product.
 - **Goals**
 - **System functions (functional requirements)** - what system is supposed to do (e.g. system should do credit payment authorization)
 - **System attributes (non-functional requirements)** - characteristics or dimensions of the system (e.g. fault tolerance, response time, ...)
 - **Use cases** (narrative descriptions, stories or cases of using a system)
 - Analysis of risks
 - **Requirements risks**
 - **Technological risks**
 - **Skills risks**
 - ...
- The artefacts produced in this phase are not UML-specific

Slide 15

Use Case

- A use case is a pattern of behaviour the system exhibits
 - It is a relatively large end-to-end process description that typically include many steps and transactions
- A flow of events document is created for each use case
 - Written from an actor point of view
 - Specification of the interactions of an actor with the system
 - None of the inner workings of the system is discussed, nor is the user interface described in any detail
- Typical contents
 - How the use case starts and ends
 - Normal flow of events
 - Alternate flow of events
 - Exceptional flow of events

Slide 18

Talking with the customer

- **Azienda:** *mi spieghi come si usa il servizio?*
- **Cliente:** *Il cliente naviga nel catalogo e raccoglie gli articoli in un carrello della spesa. Quando il cliente desidera pagare, descrive la modalità di spedizione e fornisce le informazioni relative alla carta di credito per confermare l'acquisto.*
- **Azienda:** *l'acquisto va sempre a buon fine?*
- **Cliente:** *No, il sistema controlla se la carta di credito è valida e conferma l'acquisto sia immediatamente che con un successivo messaggio e-mail*

Slide 19

Use Case - Example

- ◆ A simple format for capturing a use case involves describing
 - Its primary scenario as a sequence of numbered steps
 - The alternatives as variations on that sequence
 - The amount of detail you need depends on the risk in the use case: the more risk, the more detail you need

Buy a Product

1. Customer browses through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
- ~~6. System authorizes purchase~~
7. System confirms sale immediately
8. System sends confirming email to customer

Alternative: Authorization Failure

At step 6, system fails to authorize credit purchase
Allow customer to re-enter credit card information and re-try

Alternative: Regular Customer

3a. System displays current shipping information, pricing information, and last four digits of credit card information
3b. Customer may accept or override these defaults
Return to primary scenario at step 6

Slide 20

Actors

- An actor is an entity external to the system who participate in the story of the use case
 - Actors don't need to be human
- An actor typically stimulate the system with input events or receives something from it.
- Actors are represented by the role they play in the use case.



Customer

- A single actor may perform many use cases; conversely, a use case may have several actors performing it
 - There is one **initiator actor** and other participating actors

Slide 21

Identifying Use Cases

- *Actor-based*
 - Identify the actors related to a system or organization
 - For each actor identify the processes they initiate or participate in
- *Event-based*
 - Identify the external events that a system must respond to
 - Relate events to actors and use cases

- e.g.

<i>Cashier</i>	Log In	Cash Out
<i>Customer</i>	Buy Items	Refund Items

- The system functions should all be allocated to use cases

Slide 22

Use Case Diagram

- Captures system functionality as seen by users
- Built in early stages of development
- Purpose
 - Specify the context of a system
 - Capture the requirements of a system
 - Validate a system's architecture
 - Drive implementation and generate test cases
- Developed by analysts and domain experts
- The **use case diagram** is now part of the UML

Slide 23

Example Use Case Diagrams

Slide 24

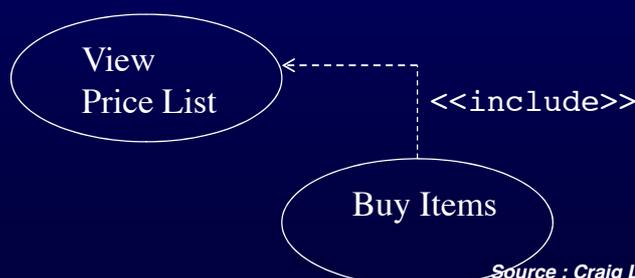
Include and Extend Use Cases

- `<<include>>` relationship is used to make the structure of the use cases more efficient by collapsing repeated operation in smaller use cases that can be shared among many other use cases
- `<<extend>>` relationship is used when there are many alternatives or options within a use case.
 - Separation of the invariant part of the use case from the variable parts
 - The invariant part becomes the use case that is extended
 - The variable parts become the extending use cases
 - A use case may have many extension points, and an extending use case may extend one or more of these extension points

Slide 25

`<<include>>` Relationship - Notation

- The “Buy Items” use case is bound to the “View Price List” use case by an oriented dashed line. The arrowhead points to the included use case and has the stereotype `<<include>>`
 - The description of the “View Price List” use case is inserted in the appropriate location of the “Buy Items” use case

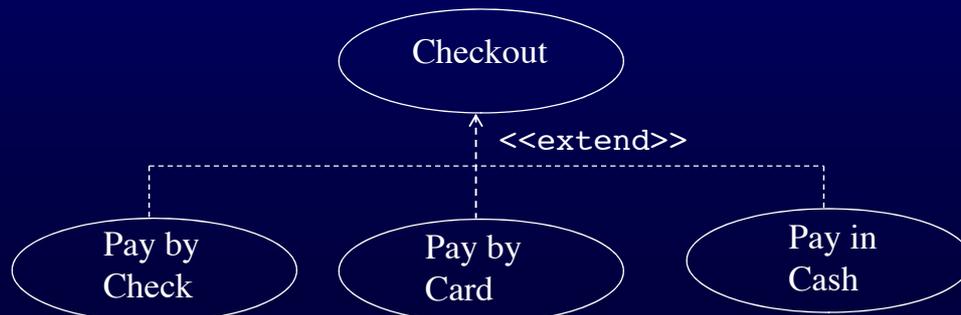


Source : Craig Larman, “Applying UML and Patterns”

Slide 26

<<extend>> Relationship - Notation

- The “Check out” use case has an extension point. The descriptions of the extending use cases are inserted in the “Checkout” use case at the extension point
 - They describe the optional data that need to be entered depending on the selected payment method



Source : Craig Larman, “Applying UML and Patterns”

Slide 27

Terminology

- **Scenario**
 - A specific sequence of actions that illustrates behaviors (a sequence of steps describing an interaction between a user and a system)
 - A **use case** is a set of scenarios tied together by a common user goal
- **Business vs. system use case**
 - The general usage is that a system use case is an interaction with the software, whereas a business use case discusses how a business responds to a customer or an event
 - The focus is first on business use cases, and then to the system use cases that satisfy them. At least one set of system use cases is expected for each business use case identified

Slide 28

When to Use “Use Cases”

“I can't imagine a situation now in which I would not use use cases. They are an essential tool in requirements capture and in planning and controlling an iterative project. Capturing use cases is one of the primary tasks of the elaboration phase”

- Use cases represent an *external* view of the system. There are no correlations between use cases and the classes inside the system
- *How many use cases should a project have?*
 - During an OOPSLA panel discussion, several use case experts said that for a 10-person-year project, they would expect around a dozen use cases (base use cases; each use case would have many scenarios and many variant use cases).

“As ever, use what works for you”

Slide 29

What Do We Use these Diagrams for?

- These diagrams are used for human communication, primarily between the analysts and stakeholders
 - Use cases and use case diagrams are not software-structure diagrams, they do not give any information about partitioning of the software elements of the system to be created
- These diagrams can be useful, but they are often unnecessary
 - ***Produce no document unless its need is immediate and significant***

Source : R. Martin, “Agile Software Development”

Slide 30

Ranking Use Cases

- Development cycles are organized around use case requirements; a development cycle is assigned to implement one or more use cases, or simplified versions of use cases.
- Ranking use cases
 - High ranking need to be tackled in early development cycles.
 - Qualities affecting the ranking
 - Significant impact on the architectural design
 - Significant information
 - Include risk, time-critical or complex function
 - Involve significant research and/or new and risky technology
 - Represent primary line-of-business processes
 - ...

Slide 31

The Domain Model

- Is a set of diagrams that helps to define the terms appearing in the use cases
 - shows the key objects within the problem and their interrelationship
- Is a descriptive tool that is used to help humans record their decisions and communicate with each other
 - does not represent an element of software and it does not map directly to code
 - The objects in the domain model do not necessarily correspond to the object-oriented design of the software

Slide 32

Conceptual Model

- The most important artefact to create during object-oriented analysis. It may show:
 - Concepts (a concept has a symbol, intension and extension)
 - Associations between concepts
 - Attributes of concepts
- Representation of **real-word things**, not of software components
 - It is better to over specify a conceptual model with a lots of fine-grained concepts, than to under specify it
 - Finding concepts
 - The concept category list
 - Identifying nouns and noun phrases in textual descriptions of a problem domain

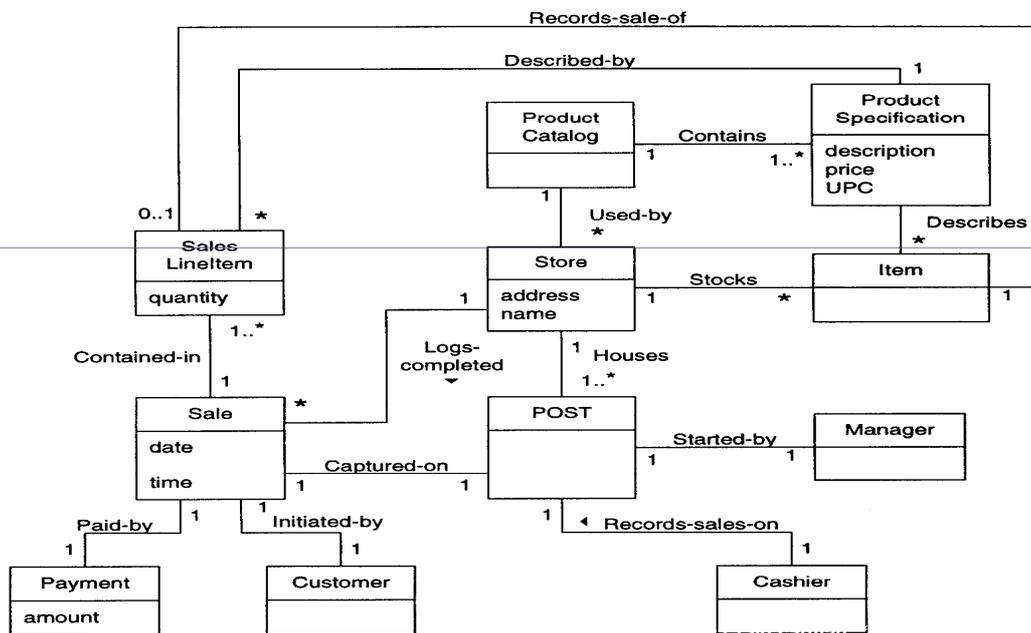
Slide 33

Conceptual Model

- Central distinction between object-oriented and structured analysis
 - Division by concepts (objects) rather than division by functions
- A conceptual model is not absolutely correct or wrong, but **more or less useful**
 - It is a tool of communication (with Stakeholders).
- Developers tend to do use cases and conceptual modeling at the same time

Slide 34

POST – Conceptual Model



DESIGN PHASE

The Architecture

“Architecture” is a term that lots of people try to define, with little agreement. There are two common elements: one is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime.

Slide 37

The Architecture (II)

- Represents the skeleton of the application
- Choosing the software platform
 - A **Web-based application**, would be accessed by a Web browser, the data would reside at the Web server
 - A **database application**, using a relational database. The forms package and the proprietary language to write the application
 - A **stand alone application**, using a visual object-oriented programming language. User interfaces could be created by using visual construction tools, invoking software functions needed to store, retrieve and manipulate data

Slide 38

Layering

- To break apart a complicated software system
 - Benefits:
 - A single layer can be understood as a coherent whole without knowing much about the other layers
 - Layers can be substituted with alternative implementations
 - Minimize dependencies between layer
 - Layers make good places for standardization
 - A single layer can be used for many higher-level services
 - Downsides:
 - Layers encapsulate some, but not all, things well. Sometimes cascading changes
 - Extra layers can harm performance. Transformation from one representation to another

But the hardest issue is deciding what layers to have and what the responsibility of each layer should be

Slide 39

Evolution of Layers in EA

- In the '90s client–server systems
 - Two-layer systems: the client held the user interface and other application code; the server was usually a relational database
 - Generally a screen was built by *dragging* controls onto a design area and then using property sheets to connect the controls to the database
- Problems came with domain logic: business rules, validations, calculations, ...
 - Usually written on the client, by embedding the logic directly into the UI
 - As the domain logic got more complex, this code became very difficult to work with
 - Simple changes resulted in hunting down similar code in many screens
 - Alternative: put the domain logic in the database as stored procedures
 - Stored procedures gave limited structuring mechanisms
 - SQL as a standard would allow changing the database vendor (few people actually did this). Stored procedures removed that option. since they are all proprietary,

Slide 40

Evolution of Layers in EA (II)

- The object oriented community had an answer to the problem of domain logic:
 - Move to a three-layer system
 - A presentation layer for the UI, a domain layer for the domain logic, and a data source layer
 - Issues
 - ⊙ Many systems were simple
 - Although the three-layer approach had many benefits, the tooling for client–server was compelling if the problem was simple
 - ⊙ The client–server tools were difficult, or even impossible, to use in a three-layer configuration

Slide 41

Evolution of Layers in EA (III)

- With the rise of the Web ...
 - Need to deploy client–server applications with a Web browser
 - A business logic embedded in a rich client needed to be redone to have a Web interface
 - A well-designed three-layer system could just add a new presentation layer
 - Java, an object-oriented language, hits the mainstream
 - The tools that appeared to build Web pages were much less tied to SQL and thus more open to a third layer

Slide 42

The Three Principal Layers

- *Presentation logic* - handles the interaction between the user and the software
 - Can be as simple as a command-line or text-based menu system, or a rich-client graphics UI or an HTML-based browser UI
 - Responsibilities: to display information to the user and to interpret commands from the user invoking the corresponding actions
- *Domain logic*, aka business logic
 - Involves calculations based on inputs and stored data, validation of any data, and choosing the right data source logic
- *Data source logic* - communicates with other systems
 - e.g. transaction monitors, other applications, messaging systems
 - Generally a database, responsible for storing persistent data

Slide 43

Choosing Where to Run Layers

- Running everything on servers. An HTML front end that uses a Web browser
 - Everything is easy to upgrade and fix
 - No deployment to many desktops and keeping them all in sync with the server
 - No problems of compatibilities with other desktop software
- Running code on a client improves responsiveness
- The data source generally runs only on servers
 - Exception: duplication of the server functionality onto a powerful client, usually for disconnected operation
- Where to run the presentation depends mostly on the kind of user interface
 - Running a rich client means running the presentation on the client
 - Running a Web interface means running on the server
 - B2C system, all processing is on the server
 - Alternative: use of scripting and downloadable applets, but this reduces the browser compatibility
 - *Web presentation if you can, the rich client if you must*
- Domain logic: all on the server or all on the client, or split it

Slide 44

Class Diagram – *Structural Modelling*

- The class diagram technique has become truly central within object-oriented methods
 - Every method has included some variation on this technique.
- A **class diagram** describes the types of objects in the system and the various kinds of static relationships that exist among them. There are two principal kinds of static relationships:
 - **associations** (e.g., a product catalog contains a product specification)
 - **subtypes** (a paper is a kind of document)
- Class diagrams also show the attributes and operations of a class and the constraints that apply to the way objects are connected

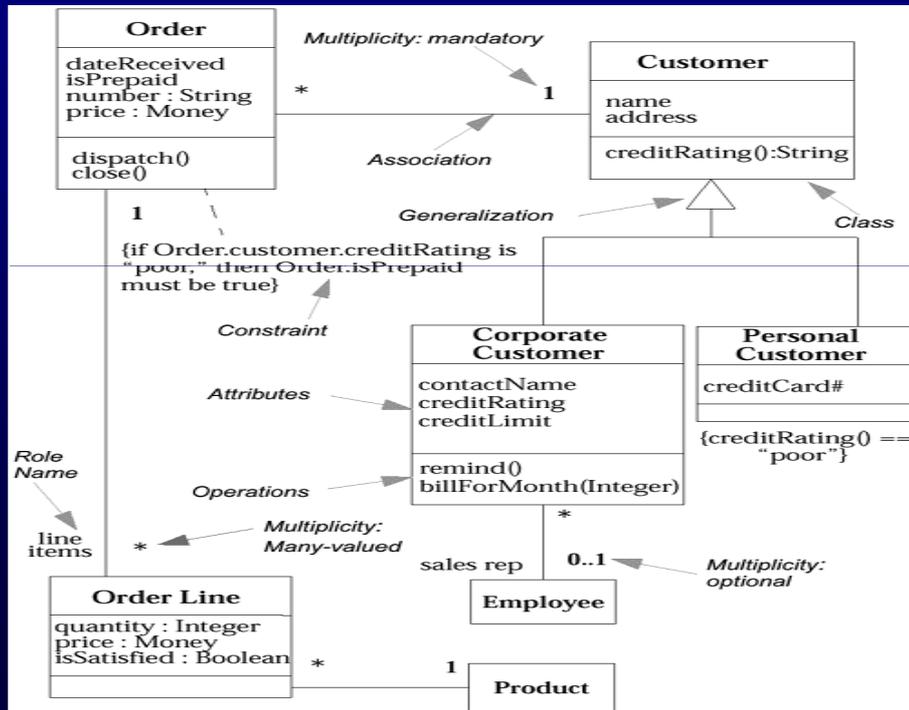
Slide 45

Class Diagram - Perspectives

- There are three perspectives one can use in drawing class diagrams
 - **Conceptual**
 - Conceptual model
 - **Specification**
 - Focus on the interfaces of the software, not the implementation
 - **Implementation**
 - This is probably the perspective used most often, but in many ways the specification perspective is often a better one to take.

Slide 46

Typical Class Diagram

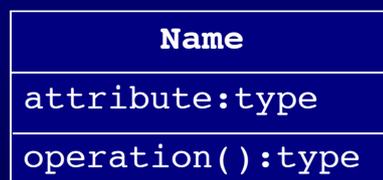


Source :
Fowler, Martin
UML Distilled
Addison Wesley

Slide 47

Class Diagram - Notation and Semantics

- In UML, a class is drawn as a rectangle with three compartments



- Properties are primarily structured comments
- Attributes and operations (the body is called method) can be adorned with encapsulation specifiers (+, -, #) = (public,private,protected)

Slide 48

Associations

- Represent relationships between instances of classes (e.g. *a person works for a company*). Instances pass messages
 - Each association has two **association ends**. An end can be explicitly named with a label, called a **role name**
 - The **multiplicity** indicates lower and upper bounds for the participating objects
- Within the specification perspective, associations represent **responsibilities**
 - There are methods within the class providing the corresponding information
 - E.g. there are methods in Order that will let us know which Customer placed a given Order and what Line Items are on an Order
 - If there are standard conventions for naming these methods, one can probably infer the class interface. For instance in Java one can infer :

```
class Order {  
    public Customer getCustomer();  
    public Set getOrderLines();
```

Slide 49

... more on Associations

- Within the implementation perspective one can deduce information about data structure
 - Associations are most often implemented as instance variables (attribute visibility) in one class that point or refer to the other
- The **navigability** of an association can be restricted by adding arrowheads
 - When an arrowhead is present, the association can only be navigated in the direction of the arrow. This means that the class to which the arrow points does not know about its associate



Slide 50

Aggregation and Composition

- Aggregation is a special form of association
 - Is denoted with a white diamond on the aggregate class
 - Implies a “*whole/part*” relationship
 - The “*whole/part*” relationship is purely connotative; there is no semantic difference from association



- Composition is a special (stronger) form of aggregation
 - Is denoted with a black diamond
 - Implies that the “*whole*” is responsible for the lifetime of its “*part*” (directly or indirectly), when the whole is created/destroyed all components are created/destroyed



Slide 51

Constraints

- The basic constructs of association, attribute, and generalization can be used to specify important constraints
 - But ... they cannot indicate every constraint
- The UML allows to use anything to describe further constraints
 - The only rule is to put them inside braces “{}”.
 - One can use an informal English, emphasizing readability
 - The UML also provides a formal Object Constraint Language (OCL)

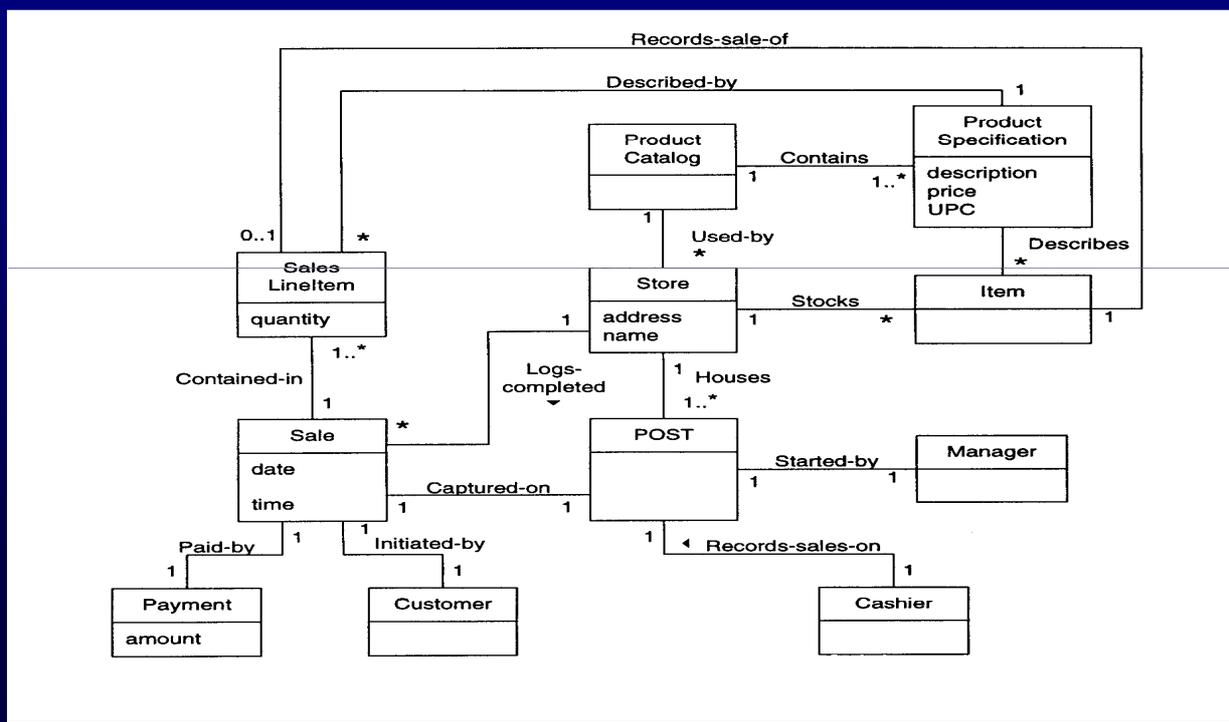
Slide 52

Making a Class Diagram

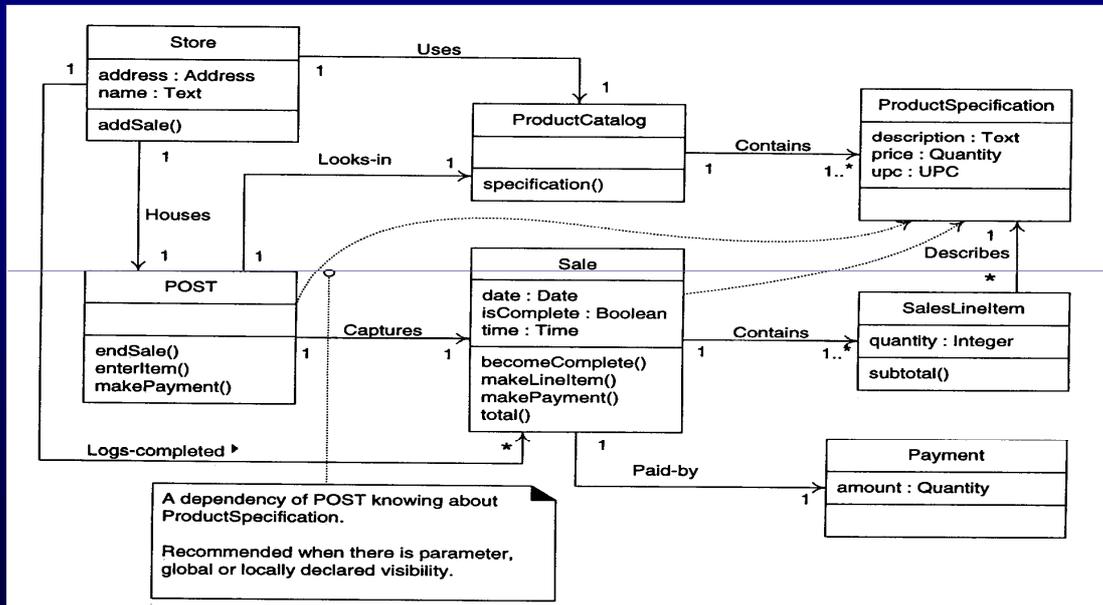
- Identify all the classes participating in the software solution. Do this by analysing the conceptual model
 - It is not usually useful to show container/collection classes explicitly in the class diagram
- Draw them in a class diagram
- Add methods names by analysing the interaction in the conceptual model
 - Do not add *accessing* methods
- Add type information to the attributes and methods
 - Too many details may adversely effect the noise-to-value ratio
- Add the associations necessary to support the required attribute visibility
- Add navigability arrows to the associations to indicate the direction of the attribute visibility
- Add dependency relationship lines to indicate non-attribute visibility

Slide 53

POST – Conceptual Model



POST – Class Diagram



Source : Craig Larman, "Applying UML and Patterns"

Slide 55

Generalization

- The activity of identifying commonality among concepts and defining supertype (general concept) and subtype (specialized concept) relationship
- Create a subtype of a supertype when either:
 - The subtype has additional attributes of interest
 - The subtype has additional associations of interest
 - The subtype concept is operated upon, handled or behaves differently than the supertype or other subtypes, in ways that are of interest

Slide 56

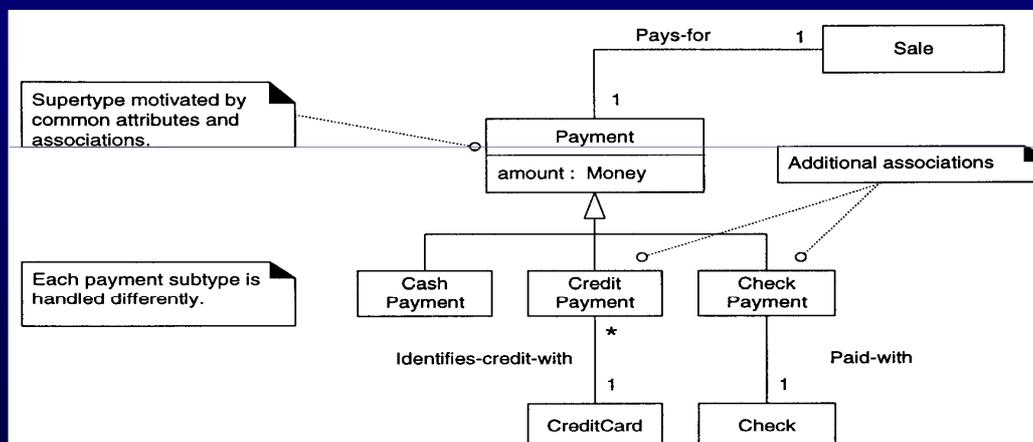
Generalization

- Create a supertype in a generalization relationship to subtypes when:
 - The potential subtypes represent variations on a similar concept
 - The subtypes will conform to the *is-a* rule
 - All subtypes have the same attribute which can be factored out and expressed in the supertype
 - All subtypes have the same association which can be factored out and related to the supertype

Slide 57

POST – Generalization

- Type hierarchy of various kind of payments



Source : Craig Larman, "Applying UML and Patterns"

Slide 58

Interaction Diagram

Slide 59

Interaction Diagrams – *Behavioural Modelling*

- One of the most important artefacts created in object-oriented design
 - Typically, an interaction diagram captures the behavior of a single use case
 - Illustrate the message interactions between instances. UML defines two kinds:
 - **Collaboration diagram**
 - The layout indicates how objects are statically connected
 - The sequence is indicated by numbering the messages
 - **Sequence diagram**
 - Emphasizes the time ordering of messages
 - Two features distinguish it from collaboration diagram:
 - There is the object *lifeline*
 - There is the *focus of control* (period of time during which an object is performing an action)

Slide 60

Interaction Diagrams – *Behavioural Modelling*

- The skilful assignment of *responsibilities* that occurs while creating interaction diagrams is very important
 - The amount of time and effort spent on their generation and the careful considerations of responsibility assignment, should absorb a significant percentage of the design phase of a project
 - Codified patterns and principles can be applied to improve the quality of the design
- Interaction diagrams are good at showing collaborations among the objects; they are not so good at precise definition of the behavior.
 - If one want to look at the behavior of a single object across many use cases, one should use a *state diagram*
 - If one want to look at behaviors across many use cases or many threads, one should consider an *activity diagram*

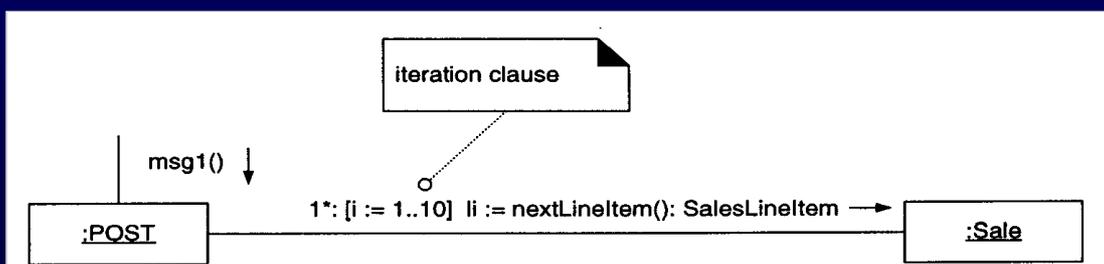
Slide 61

Collaboration Diagram – UML Notation

- Message syntax:

```
return := message(parameter: parameterType): returnType
```

- Illustrating iteration

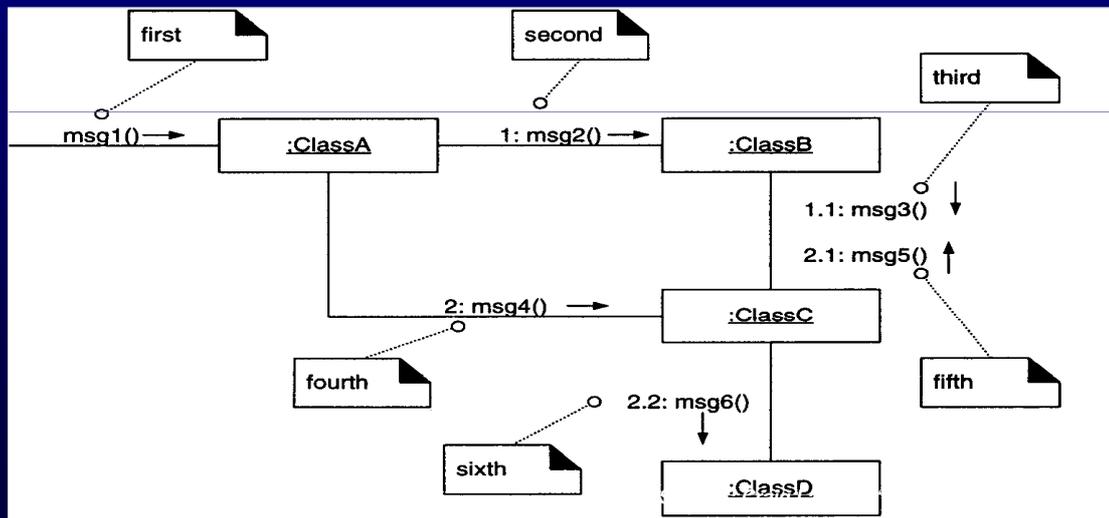


Source : Craig Larman, "Applying UML and Patterns"

Slide 62

Collaboration Diagram – UML Notation

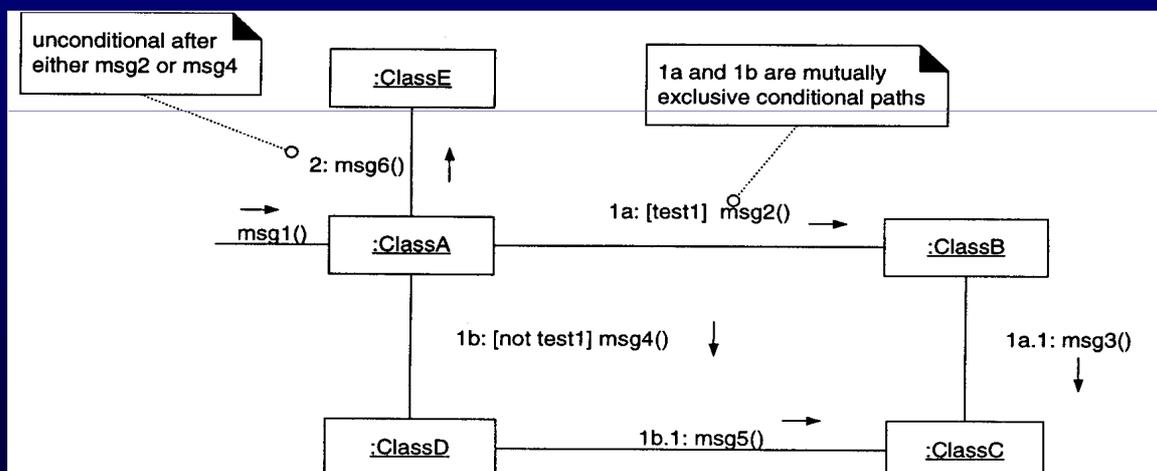
- Illustrating message number sequencing



Slide 63

Collaboration Diagram – UML Notation

- Illustrating mutually exclusive conditional paths

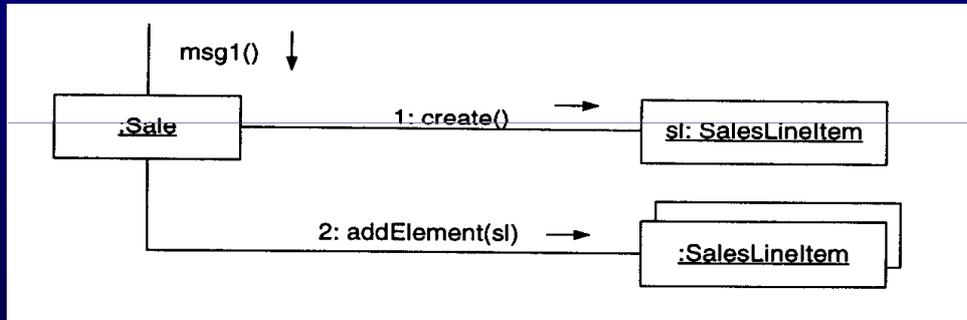


Source : Craig Larman, "Applying UML and Patterns"

Slide 64

Collaboration Diagram – UML Notation

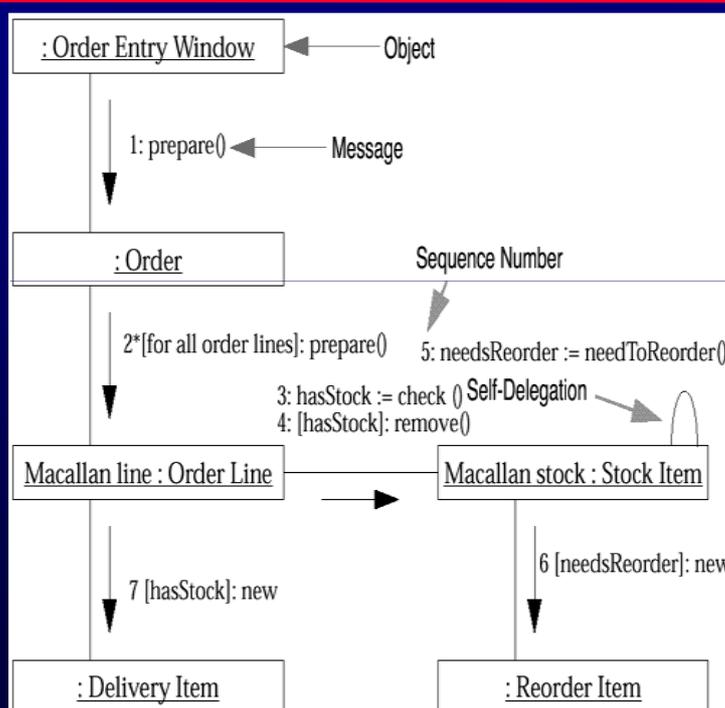
- Illustrating message to multiple objects



Source : Craig Larman, "Applying UML and Patterns"

Slide 65

Typical Collaboration Diagram



Source :
Fowler, Martin
UML Distilled
Addison Wesley

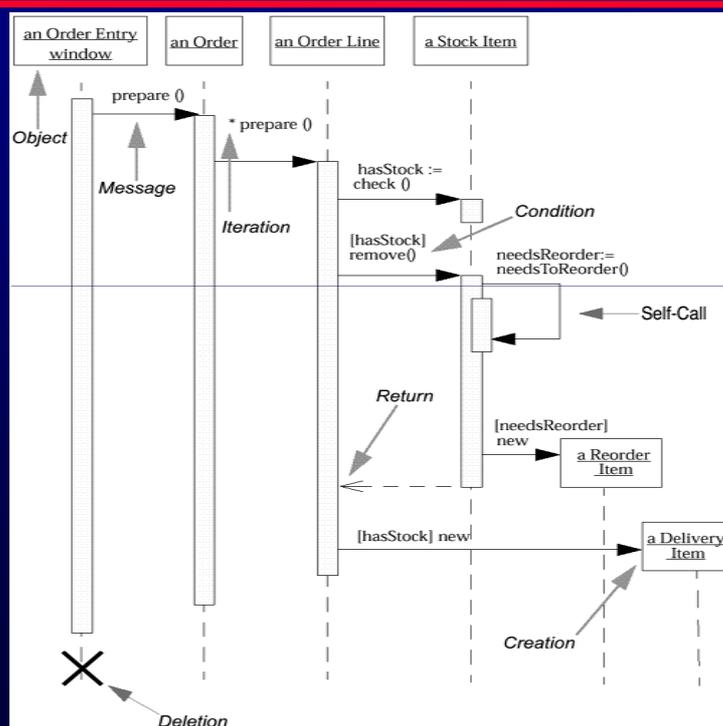
Slide 66

Sequence Diagram

- Within a **sequence diagram**, an object is shown as a box at the top of a dashed vertical line, called the object's **lifeline**
 - The lifeline represents the object's life during the interaction
 - Each message is represented by an arrow between the lifelines of two objects
 - The order in which these messages occur is shown top to bottom on the page
 - To show when an object is active one can include an activation box
 - Control markers
 - **Condition**, which indicates when a message is sent (for example, *[needsReorder]*). The message is sent only if the condition is true
 - **Iteration marker**, which shows that a message is sent many times to multiple receiver objects. One can show the basis of the iteration within brackets, such as **[for all order lines]*
 - **Return** indicates a return from a message, not a new message

Slide 67

Typical Sequence Diagram



Source :
Fowler, Martin
UML Distilled
Addison Wesley

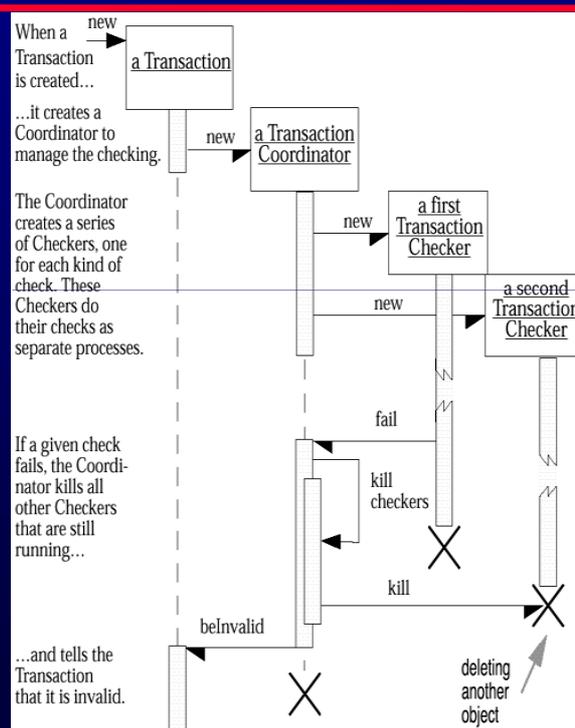
Slide 68

Sequence Diagram

- Sequence diagrams are also valuable for concurrent processes
 - The messages exchanged are **asynchronous** messages represented by means of half-arrowheads
 - An asynchronous message does not block the caller, so it can carry on with its own processing.
 - An asynchronous message can do one of three things:
 - Create a new thread, in which case it links to the top of an activation
 - Create a new object
 - Communicate with a thread that is already running
 - Object **deletion** is shown with a large X. Objects can self-destruct, or they can be destroyed by another message

Slide 69

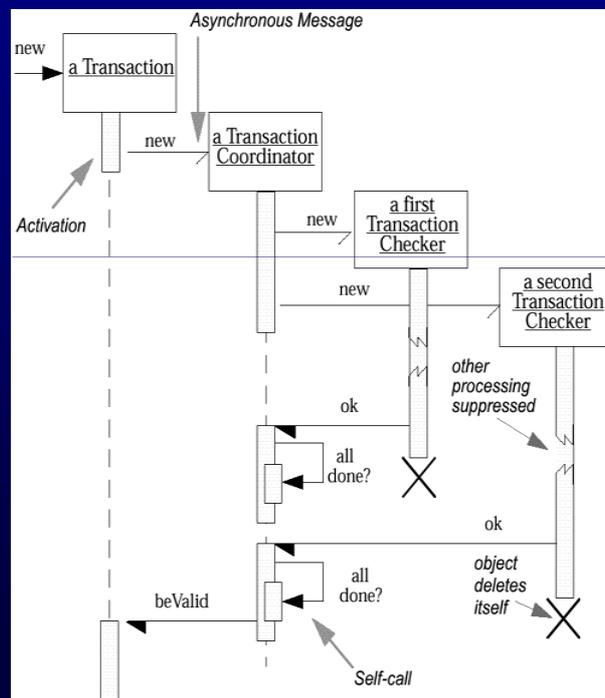
Sequence Diagram - Concurrent Processes



Source :
Fowler, Martin
UML Distilled
Addison Wesley

Slide 70

Sequence Diagram - Concurrent Processes (cont'd)



Source :
Fowler, Martin
UML Distilled
Addison Wesley

Slide 71

State Diagrams

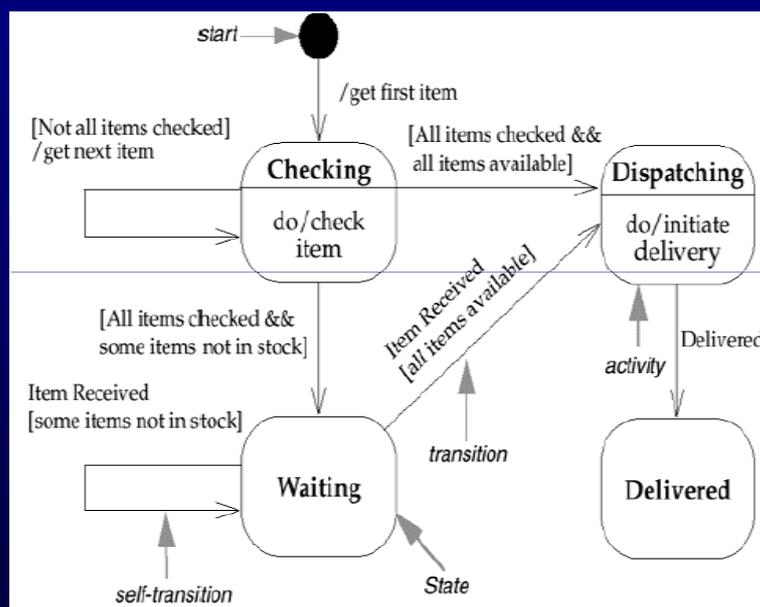
Slide 72

State Diagrams

- A familiar technique to describe the behavior of a system
- They describe all the possible states that a particular object can get into and how the object's state changes as a result of events that reach the object.
- In most OO techniques, state diagrams are drawn for a single class to show the lifetime behavior of a single object across several use cases
- When to use
 - Use state diagrams only for those classes that exhibit interesting behavior
 - Many people find that UI and control objects have the kind of behavior that is useful to depict with a state diagram.

Slide 73

A Typical State Diagram



- A UML state diagram for an order, indicating the various states of the order

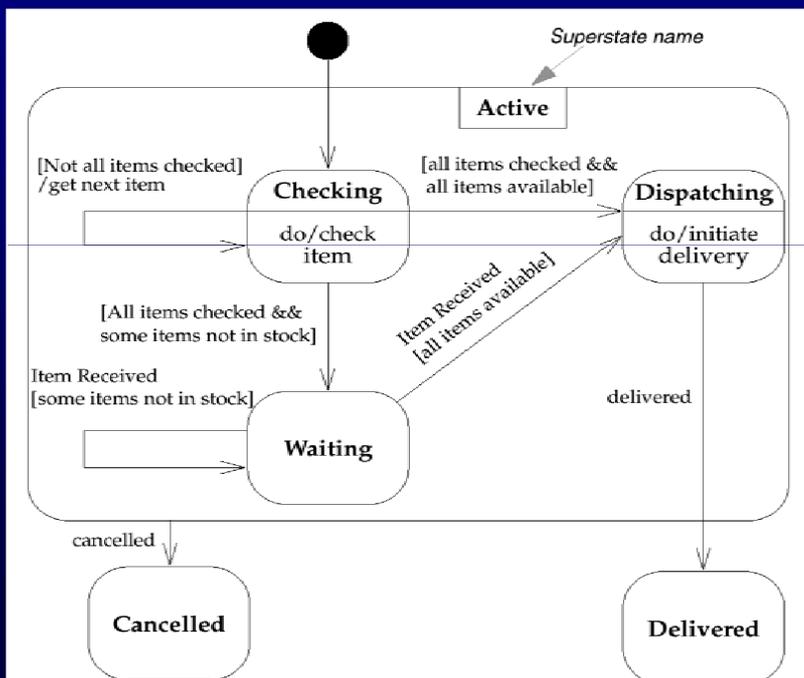
Slide 74

State Diagrams - Semantics

- The syntax for a transition label has three parts, all of which are optional:
Event [Guard] / Action
 - A **guard** is a logical condition that will return only "true" or "false." A guarded transition occurs only if the guard resolves to "true."
- A state can have an activity associated with it, indicated by a label with the syntax
do/activity
- Note: the term "action" is used for the transition whereas the term "activity" is used for the state.
 - They are both processes, typically implemented by some method, but they are treated differently.
 - **Actions** are associated with transitions and are considered to be processes that occur quickly and are not interruptible
 - **Activities** are associated with states and can take longer. An activity may be interrupted by some event
- When a transition has no event within its label, it means that the transition occurs as soon as any activity associated with the given state is completed
 - Only one transition can be taken out of a given state, so the guards must be mutually exclusive for any event

Slide 75

Example



Source :
Fowler, Martin UML Distilled
Addison Wesley

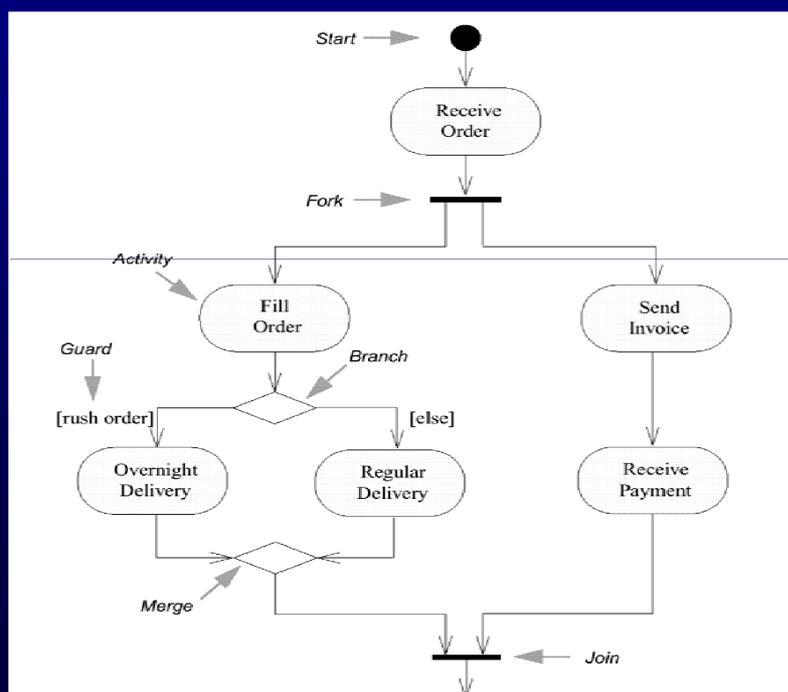
Slide 76

Activity Diagrams

- These diagrams are particularly useful in connection with workflow and in describing behavior that has a lot of parallel processing
- The activity diagram describes the sequencing of activities, with support for both conditional and parallel behavior
 - An activity diagram is a variant of a state diagram in which most, if not all, the states are activity states
- The core symbol is the **activity state**, or simply **activity**.
 - An activity is a state of doing something: either a real-world process, such as typing a letter, or the execution of a software routine, such as a method on a class

Slide 77

A Typical Activity Diagram



Source :
Fowler, Martin UML Distilled
Addison Wesley

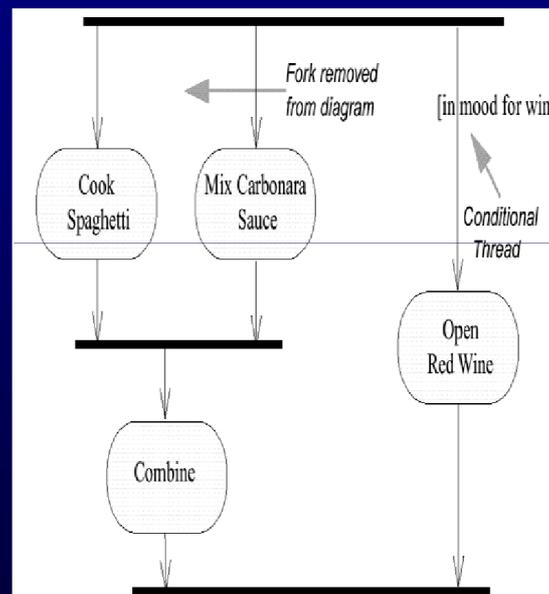
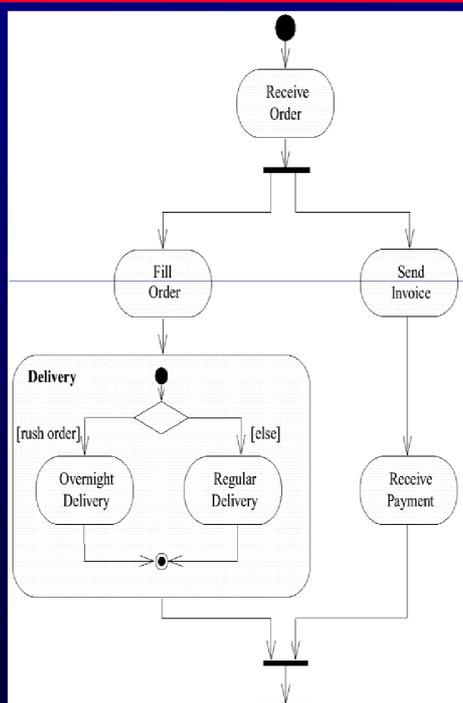
Slide 78

Activity Diagrams - Semantics

- Conditional behavior is delineated by branches and merges.
 - A **branch** has a single incoming transition and several guarded outgoing transitions, mutually exclusive.
 - The [else] guard indicates that the transition should be used if all the other guards on the branch are false.
 - A **merge** has multiple input transitions and a single output. A merge marks the end of conditional behavior
- Parallel behavior is indicated by forks and joins.
 - A **fork** has one incoming transition and several outgoing transitions. When the incoming transition is triggered, all of the outgoing transitions are taken in parallel.
 - **join** is used to synchronize parallel behaviors. With a join, the outgoing transition is taken only when all the states on the incoming transitions have completed their activities.

Slide 79

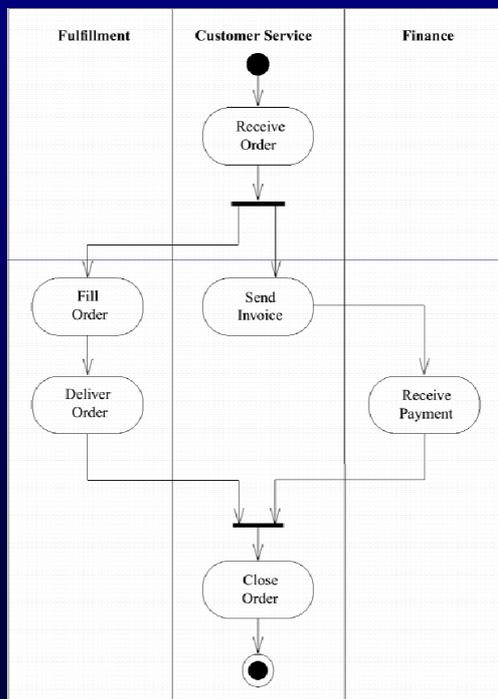
Activity Diagrams



Source : Fowler, Martin UML Distilled Addison Wesley

Slide 80

Swimlanes



- To indicate which class is responsible for each activity
 - Activity diagrams are arranged into vertical zones separated by lines
 - Each zone represents the responsibilities of a particular class or a particular department
 - They combine the activity diagram's depiction of logic with the interaction diagram's depiction of responsibility
 - They can be difficult to draw on a complex diagram
- “Sometimes one have to stop trying to say too much in one diagram”*

Slide 81

Use Activity Diagrams - Pros

- Activity diagrams offer a suitable support for the representation of parallel behaviors
- One can use activity diagrams in the following situations:
 - **Understanding workflow**
 - Useful for understanding a business process.
 - Can easily be drawn together with business experts to understand how a business operates and how it may change.
 - **Analyzing a use case**
 - At this stage (conceptual modeling), one is not interested in allocating actions to objects, instead one just need to understand what actions need to take place and what the behavioral dependencies are.
 - **Describing a complicated sequential algorithm**
 - In this case, an activity diagram is just a UML-compliant flowchart
 - **Dealing with multithreaded applications**

Slide 82

Use Activity Diagrams - Cons

- Their great disadvantage is that they do not make the links among actions and objects very clear
- One should not use activity diagrams in the following situations:
 - **Trying to see how objects collaborate**
 - An interaction diagram is simpler and gives a clearer picture of collaborations
 - **Trying to see how an object behaves over its lifetime**
 - Use a state diagram for that
 - **Representing complex conditional logic**
 - Use a truth table

Slide 83

Packages and Deployment

Slide 84

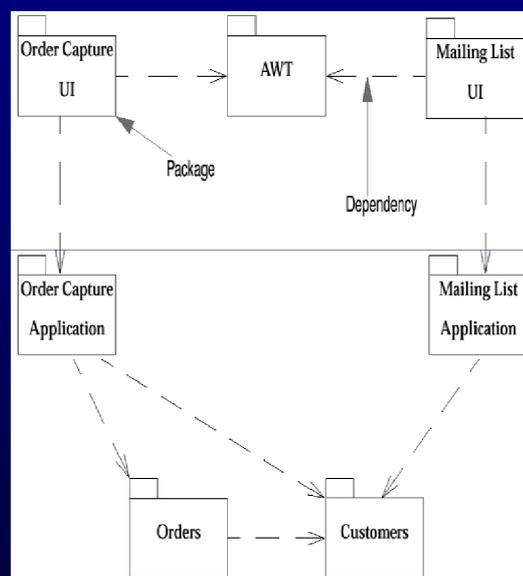
Package

- One of the oldest questions in software methods is:
“How do you break down a large system into smaller systems?”
- Structured methods used **functional decomposition**
 - The overall system was mapped as a function and broken down into sub-functions, ...
- OO methods promote the idea to group the classes together into higher-level units
 - In the UML, this grouping mechanism is called the **package**
 - The idea of a package can be applied to any model element, not just classes
 - A package is a namespace for its members, and may contain other packages

Slide 85

Packages

- Group elements that provide common service (or family of related services), with relatively high coupling and collaboration (dependency)
 - At some level of abstraction the package will be view as highly cohesive – it has strongly related responsibility
- A package is shown as a large rectangle with a small rectangle (a “tab”) attached to the left side of the top
 - The members of the package may be shown within the large rectangle
- A dependency between two packages exists if any dependency exists between any two classes in the packages



- A diagram representing relations among packages

Source :
Fowler, Martin
UML Distilled
Addison Wesley

Slide 86

When to Use Package

- Packages are a vital tool for large projects.
 - The suggestion is to use packages whenever a class diagram that encompasses the whole system is no longer legible on a single A4-size sheet of paper.
- Packages are particularly useful for testing
 - One can do unit testing on a package-by-package basis.

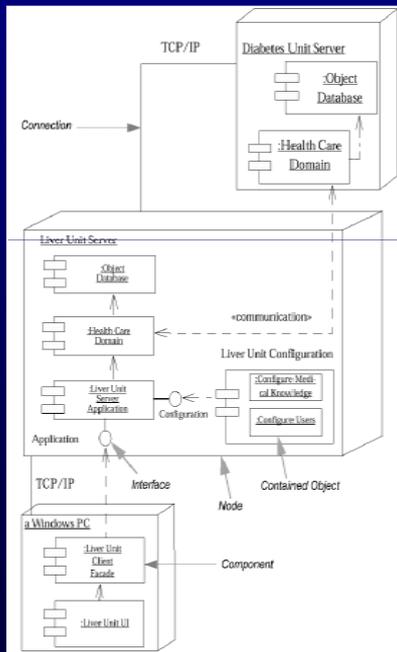
Slide 87

Deployment and Component Diagrams

- A **deployment diagram** shows the physical relationships among software and hardware components in the delivered system.
 - Each **node** on a deployment diagram represents some kind of computational unit - in most cases, a piece of hardware.
 - The hardware may be a simple device, or a mainframe.
- A **component diagram** shows the various components in a system and their dependencies.
 - A **component** represents a physical module of code.
 - A component is often the same as a package, but it may be different, since components represent the physical packaging of code.
- One can place the component diagram on the deployment diagram to show which components run on which nodes.

Slide 88

Deployment Diagram



- ◆ One can draw separate component and deployment diagrams, or can place the component diagram on the deployment diagram
- ◆ A component may have more than one interface, in which case one can see which components communicate with each interface
- ◆ One should draw these diagrams when there is the need to show physical information