- **Topics:**

OO methods:

class diagrams, use cases

OO-based Notations for Req. Analy.

- What's next?
  - Common OO-based techniques or notations
  - What's the UML?
  - Class diagrams as used in requirements modeling
  - Use cases

The Unified Modeling Language —UML

- The standard graphical languages (notation)
- Unfortunately, turned into large bin that contains everything
- UML lets one visualize, construct, document things:
  - Conceptual: business processes, requirements, system functions.
  - Concrete: classes, database schemas, components
- Three leading OO gurus, Booch, Rumbaugh, & Jacobson, joined forces in one company, Rational (now IBM):
  - Rational sold ***Rose***, A CASE tool for UML and OOA&D
  - … and other CASE tools: config. mgmt., requirement, testing tools
- UML tools: Violet (simple), Together (Borland, VS), Omondo (Eclipse plug-in), Visual Paradigm
  - You'll need one. How to choose?

The Unified Modeling Language —UML

- Notations vs. methodologies that use notations
  - Rational defined a well-defined, documented process that uses the UML
  - Now just the Unified Process (UP) not RUP
  - Other methods and processes use the UML too (e.g. HP's Fusion)
- UML is a common language, not a “how to”
- UML is a good idea
  - You should be ***familiar*** with it
- UML is ***not*** a panacea:
  - You should know that

Brief Introduction To UML

- UML has several graphic notations including:
  - *Use cases (and Scenarios)*
  - Class diagrams
  - Interaction diagrams
  - Package diagrams
  - State diagrams
  - Activity diagrams
- A good short book:

  Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*

Brief Introduction To UML

- Often learned as a way of doing "OO A&D"
- Object-oriented ***analysis***:
  - Use-cases and scenarios
  - Conceptual modeling with a class diagram
- Object-oriented ***design*** *(we'll do this later)*:
  - Package diagrams
  - Class diagrams (refinements of OOA diagrams)
  - Interaction diagrams
  - State diagrams

UML Class Diagram Notation

- Documents classes and their static relationships:
  - Subtypes
  - Associations: structural relations with other classes
- Classes represented as box with three sections:
  - Class name (perhaps other class characteristics e.g. «Abstract»)
  - Attributes (with type, visibility)
  - Operations (with parameters, return type, visibility)
  - Public/protected/private indicated by +/#/- signs

UML Class Diagram Notation

- Class Associations:
  - Sometimes called “static structure” diagram
  - Lines drawn between classes show they have a relationship
  - Name describes relationship—optional only if obvious
  - Multiplicity provides “how-many” information
  - Purpose varies depending on current activity
    - OO Analysis: capture external entities, state, relations. Not so much about behavior. Not a code blue-print.
    - OO Design: fully document classes to be written
      - All attributes and operations. All classes to be used in program.
    - For CS340, we’ll require you to do this for OO Design, following the “blueprint” idea
    - For requirements analysis, not required!

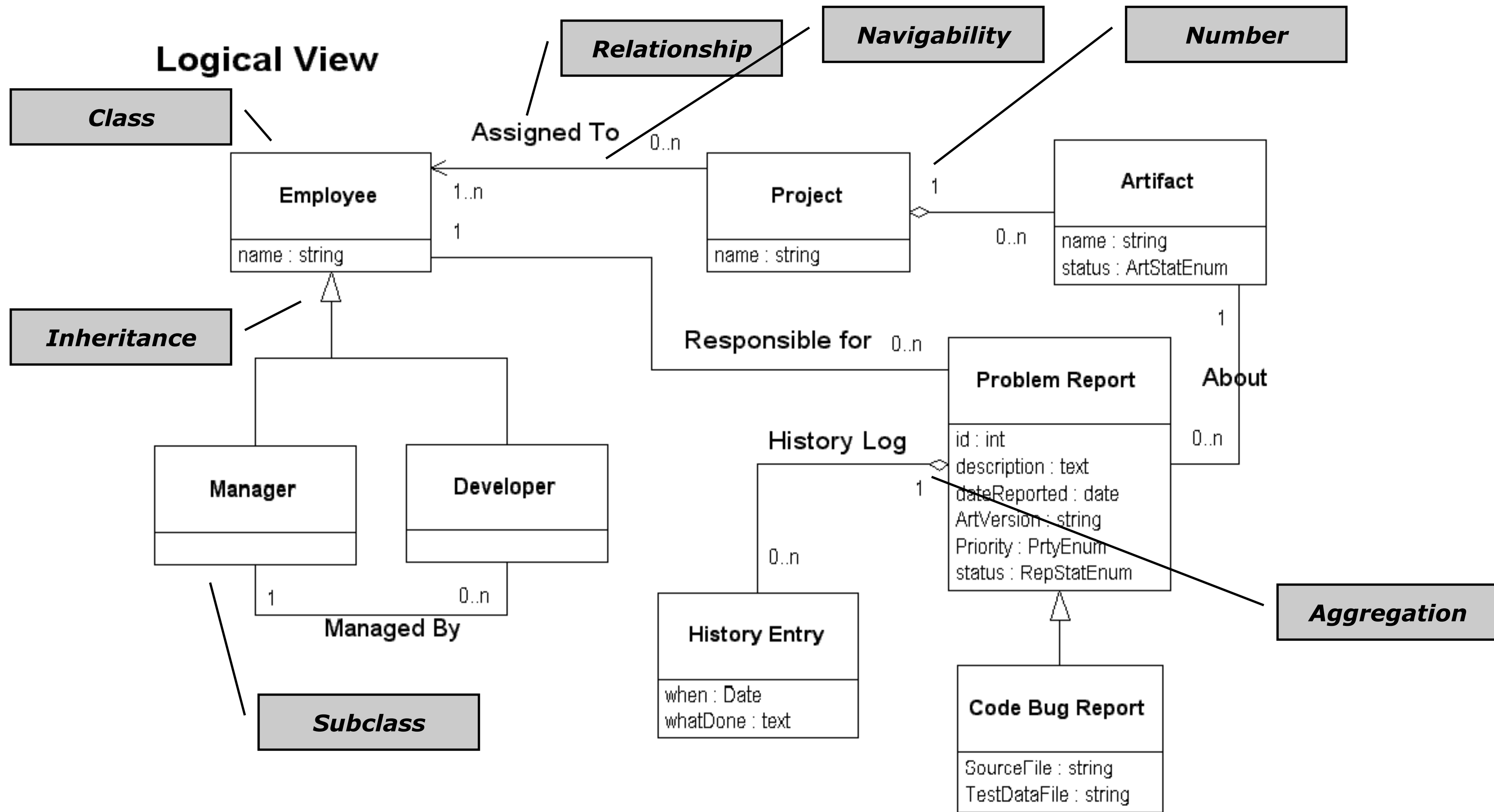
UML Class Diagram Notation

- Certain standard types of relationships:
  - Inheritance:
    - Classes have superclass, subclass relationship
    - Basically the “isa” relationship
    - Also shows “inheritance of interface” (e.g. Java’s interfaces)
  - Aggregation / Composition:
    - Basically the “part-of / has-a” relationship
    - There are some subtleties. See Fowler's *UML Distilled*
  - Possibly confusing: diagram shows both:
    - inheritance, a *definition-relationship*, and
    - aggregation, a *structural-relationship*
- Note: all UML diagram notations allow for *notes* to be attached to objects or to the diagram as a whole

UML Class Diagram Notation

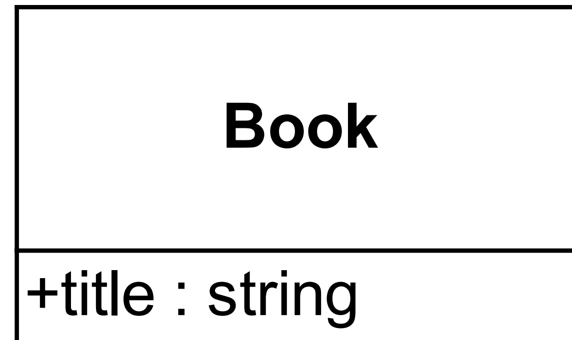
- Box for class:
  - In the box, sections for: attributes, operations
  - Each named
- Types of arrows/lines:
  - Line with *white triangle* pointer shows <u>*inheritance*</u>
  - Line with *white diamond* pointer shows <u>*aggregation*</u>
  - Line with *black diamond* pointer shows <u>*composition*</u>
  - Simple line shows <u>*association*</u> -- give it a name!
- For aggregation, composition, and association:
  - Numbers at line-ends show how many, the *mulitiplicity*
  - Arrows at one or both ends if you want to show *navigability*

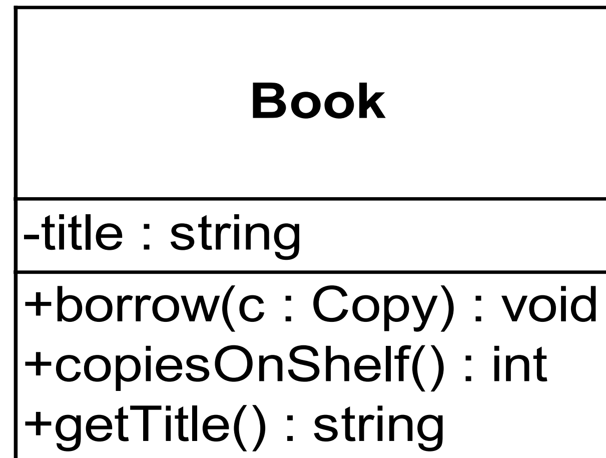
Conceptual Model for Problem Report Tool



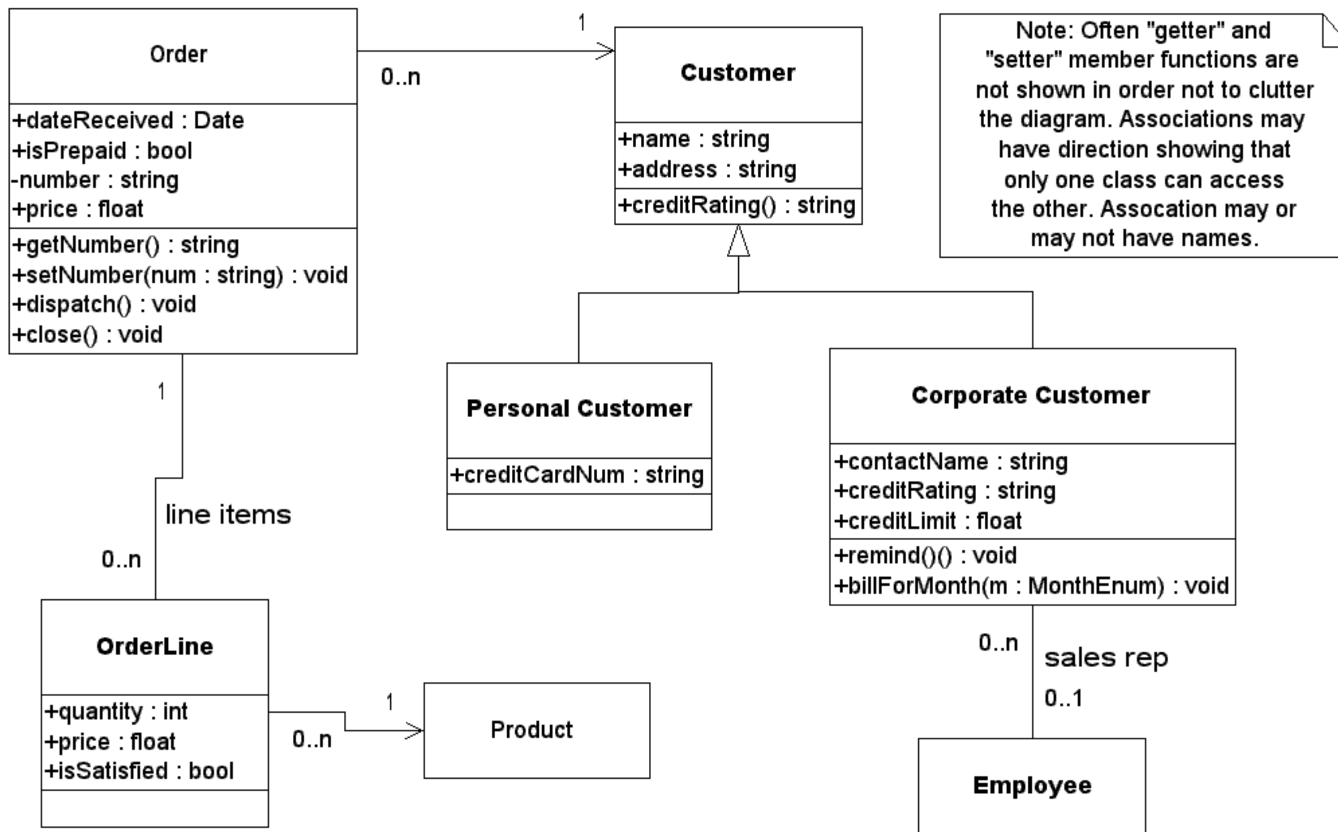
Classes in UML Diagrams

- Attributes in middle
- Operations at bottom
  - Can be suppressed. (What level of abstraction?)
- Attribute syntax:
  name : type = default
- Operation syntax:
  name (params) : return type
- Visibility
  - \+ public
  - \- private
  - # protected etc.
  - nothing? Java's default-package?





Another UML Class Diagram Example



Use Cases - Interactions

- Similar to other scenario-based methods
- Goal: define interactions between system and external "entities"
  - Other systems, users
- Defines *tasks* or *services* from an external view
- *Use case*:
  - Defines a goal-oriented set of interactions between external actors and the required system
- Literally a description of the interaction between the user and the system

Use Cases

- Use-case modeling to describe user needs.
  - **Use case**: “a sequence of actions a system performs to yield an observable result of value to a particular actor.”
  - **Actor**: someone or something outside the system that interacts with the system.
- Remember: we want an “external” view of the system. How it interacts with its environment
- Use cases are described as flows of events, usually in text form

Use Cases

- *Actors*:
  - Parties outside the system that interact with the system.
  - Actor may be a class of users, roles users can play, or other systems.
  - Non-actor Example: Server for a client.
- Use case describes all possible sequence of interactions between actors & system from initiating event (goal) to completion of event (goal satisfied).

Use Cases

- Includes possible variants of main sequence:
  - Sequences that also satisfy the goal
  - Sequences that might lead to failure because of exceptions.
- System is a “black box”, interactions with system, including system responses, are as seen from outside the system.

Use Cases

- Use cases capture *who* (actor) does *what* (interaction) with the system, for what *purpose* (goal).
- Complete set of use cases specifies all the different ways to use the system
- Defines all behavior required of the system, if done completely
- ***Question***: how can one be sure that use case is complete?

Use Cases

- Generally, use case steps are documented in natural language.
- ***Scenarios***:
  - A scenario is an instance of a use case
  - Single path through the use case.
  - Note this is a specialized use of generic term “scenario”.
- Possible approach:
  - Construct scenario for the *usual* path through the use case
  - Construct other scenarios for each possible variation of flow through the use case, e.g.., triggered by optional paths, error conditions, etc.).

Example Template for Use Cases

- *Use case number or id:*
- *Use case title:*
- *Text description (a few sentences)*
- *Preconditions (if applicable):*
- *Flow of Events:*
- *Basic path:*
- *1.First step*
- *2.Second step*
- *3.etc*
- *Alternative Paths:*
  - *Name and short description (in words) of first alternative path/scenario.*
  - *Name and short description (in words) of 2nd alternative path/scenario.*
  - *etc.*
- *Postconditions (if applicable)*
- *Special conditions (if applicable).*

Example: ATM Withdrawal

- Three use cases for ATM: withdraw money; transfer money; check balance
- *Withdraw money* description—user withdraws money from automatic bank-teller machine:
  - Client inserts card and system reads it
  - System prompts for PIN.
  - Client enters and it's validated.
  - System asks “which operation”
  - Client selects “withdrawal”.
  - Client chooses which account (checking, savings).
  - System requests amount and Client enters it.
  - System communicates with ATM Network to validate account ID, PIN, amount.
  - System asks Client if they want a receipt (if there is paper left to print one).
  - System asks Client to withdraw their card. Client does this.
  - System dispenses requested amount of cash.
  - System prints receipt.

Example: ATM Withdrawal

- Actors: client, system, ATM network
- But it's more complex, isn't it?
  - Alternative branches ("quick cash" withdrawal).
  - Exceptions and errors (invalid pin, network down).
- We group all of these into one single use case
- A **use case** is thus a *family* or *class* of scenarios.
- **Scenarios** are instances of a use case.
- Most systems described by only a small number of use cases, each with a lot of variation.
- Relations between use cases in our models:
  - Some use cases are "included" in others (e.g.. Insert card and check PIN)
  - Some use cases extend functionality in a base use case.

Advantages of Use Cases

- Represent an external view of the system. (Helps avoid the pitfall of designing too early.)
- Easily understood by customers and engineers. (Customers and users can validate them.)
- Provide a natural organization for system functionality
- Force us to think how the user interacts with the system—very helpful in user-interface design
- Can form a basis for testing the software using functional testing
- *Can validate a design by stepping through it to see how it "executes" each scenario*
- But:
  - Not the whole story: constraints, non-functional requirements
  - Part of requirements documentation, not all of it

END

Where Are You Re: Requirements

- Groups need to…
  - Understand requirements (not design – that's later)
  - Document requirements (for themselves, stakeholders)
- … and produce:
  - Complete requirements description for part the system
    - both groups do communications protocol requirements
    - System boundary, functional requirements, non-functional req.
    - Representations: use cases, other notations/documentation
  - User interface prototypes (both robot control or debugging).
- Need a process for developing a requirements specifications

Basic Requirements Analysis Sub-Process

- Develop milestones, schedule, WBS—remember risks and slack time
- Get all available documentation about project:
  - Domain/application information
  - Target computer system
  - Operating environment
- Develop ***and document*** use cases
- Develop ***draft*** specification document including:
  - All required/structural material (boilerplate)
  - Overview to characterize system to new reader
  - ***Functional*** requirements
  - ***Non-functional*** requirements

Basic Requirements Analysis Sub-Process

- Develop mockup of critical elements (prototype) —"is this OK?"
- Review inside corporation—develop detailed question list
- Review with customer
- Revise document and ***repeat***

Documentation

- For CS340, software requirements specification (SRS)
  - Note many methodologies don't demand a formal document
- Ask what this document is for: Your ideas?
  - For developers: why?
  - For other stakeholders: why?

Requirements Analysis Techniques

- Carefully identify:
  - Functional requirements
  - Non-functional requirements
    - for each function / feature
    - for the system as a whole
- Ways of structuring the ***analysis***:
  - Top down
  - Bottom up
  - Object oriented
  - Document oriented

Requirements Analysis Techniques

- Information acquisition methods:
  - Intelligent query of customers or users
  - Careful review of specification by both sides
  - Prototyping of ideas for evaluation
  - Build models (e.g. diagrams, prototypes)
- Customers ***cannot*** state what they want but they can ***criticize***
  - A view of the process might look like the "Elicit, Document, Verify" diagram we put on the board

Use Case Example *Sketch*

- Goal:
  - Move robot under ***sequence*** control to new location
- Actors:
  - Driver, robot
- Possible sequence of commands (completely hypothetical):
  - Set interface to motion command mode
  - (Position camera to required direction)
  - Set motion limits (speed, turning radius, direction, etc.)
  - Turn on robot sensors
  - Define initiate and continue movement sequence
- Other scenarios to consider:
  - Robot hits something
  - Robot exceeds performance limits
  - Robot loses communication
  - Etc.

Use-Case Resources

- Web Sites:
  - http://www.usecases.org/
  - http://www.pols.co.uk/use-case-zone/
  - http://www.bredemeyer.com/use_cases.htm
- Textbooks:
  - Object-Oriented Software Engineering: A Use-Case Driven Approach, Jacobson, I. et al.
  - The Unified Modeling Language User Guide, Grady Booch, et al
  - The Unified Modeling Language Reference Manual, James Rumbaugh, et al

Always Remember

**Once implemented,
wrong requirements are *very* expensive to repair**

**Once software is in the field,
wrong requirements are *extremely* expensive to repair**

Customers think that requirements can be changed very easily.

You must make the costs clear.