

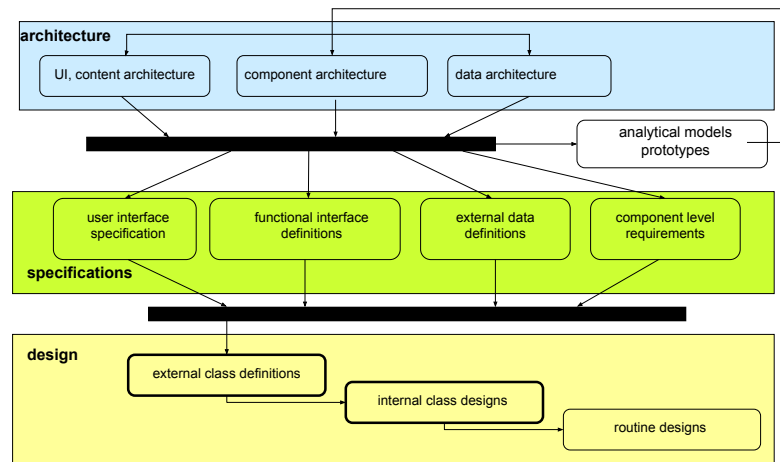
Component Level Class Design

- Specification, Design, Components
- Classes
 - reasons for class creation
 - elements of good class design
 - classes in non object oriented languages
- Packages
 - what are packages
 - elements of good packaging
- Diagramming Classes in UML

Class Design

1

Model Hierarchy/Succession



Models and Prototypes

2

Reqt/Specs/Design

- Requirements
 - characteristics a successful solution must exhibit
 - they tend to be user-facing
- Specifications
 - a complete description of the interfaces and behavior that a component must have in order to correctly perform its role in a system
 - they must be specific and measurable, and any component that meets these specifications will be acceptable
- Design
 - description of the internal structure and operations that will be used to implement a specified component

Class Design

3

What is a “component”?

- A modular, deployable, and replaceable part of a system, that encapsulates implementation, and exposes a set of interfaces.
 - a defined piece of a larger system
 - can be added or removed from that system (not necessarily a **Field Replaceable Unit**)
 - contributes to the working of the system
 - inner mechanisms may be hidden
 - functionality is defined by an interface spec

Class Design

4

Component Specifications

- a step between requirements & design
 - a component specific list of requirements
 - the basis for the component design
- functional specifications
 - written from the user’s point of view
 - enumerate component capabilities, interfaces
- technical specifications
 - written to guide the implementer
 - capture key design decisions or suggestions

Class Design

5

When to create a new class

- provide needed objects
 - obvious objects from the problem domain
- provide better behaved objects
 - kinder, gentler versions of real objects
- compartmentalize complexity
 - bring all related code into a single place
 - simplify interface seen by rest of system
- make applications more stable & portable
 - isolating implementation specifics in a class
 - abstraction protects app from future evolution

Class Design

6

Characteristics of a good class

- it is well abstracted for its users
- it is cohesive
- it exhibits good information hiding
- Other principles are tests of goodness
 - Open/Closed principle
 - open for extension, closed for modification
 - Liskov Substitution principle
 - derived sub-class can substitute for its parent
 - Dependency Inversion Principle
 - depend on abstraction – not implementation

Class Design

7

Design vs Refactoring

- we need adequate design before coding
 - or we will not know what code to write
- how “good” must that design be?
 - generality? extensibility? optimization?
- after we’ve written and tested our code
 - we will better understand how it works
 - we will have observed its performance
 - we will see ways it could be simplified
- plan on time for periodic refactoring
 - more valuable than more up-front design

8

OO Languages and Design

- OO languages provide valuable features
 - mechanisms to support class inheritance
 - mechanisms to encourage information hiding
 - explicit support for interface polymorphism
 - automatic object instantiation
- these help us design better software
 - organizing our designs into modular classes
 - consciously decide what is public/private
 - encourage us to reuse common components

Class Design

9

Classes in non-OO languages

- the basic principles of good design
 - apply to all software: C, FORTRAN, asm, perl
 - any module, in any language, should
 - implement a general and intuitive “class”
 - export a well abstracted interface to that class
 - be cohesive with respect to that class
 - employ good information hiding
 - be usable, w/o change, for many purposes
 - be organized/grouped with related modules
- “Program into your language, not in it.”

Class Design

10

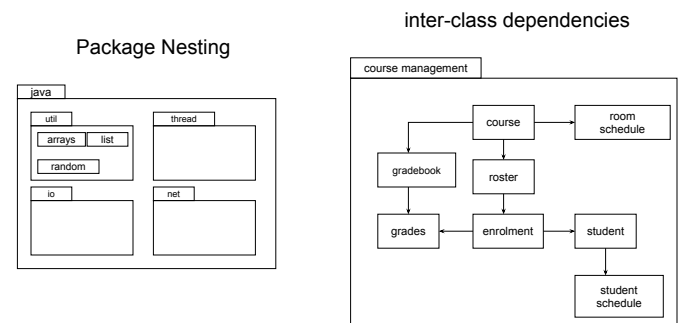
Software Packages

- provide well defined functionality
- come in a wide range of formats
 - perl.pm is little more than a source module
 - Linux deb/rpm, Windows installer
 - architecture, version, dependencies, properties
 - manifest, pre/post install/remove scripts
- may be supported by package mgt s/w
 - find/install/remove/upgrade/list operations
 - file registry, w/version and checksum info
 - remote repository management, auto update

Class Design

11

UML Package Contents



Class Design

12

(packages vs classes)

- some classes make sense in isolation
 - stacks, queues, strings, input files
- some classes naturally come in groups
 - courses, rosters, programs, grades
- a package is a collection of classes
 - that is aggregated together into a group
 - that are added and removed as a group
- some OO languages support packages
 - may not correspond to install-time packages

Class Design

13

Package Interfaces

- clients depend on packages
 - the functionality they provide
 - the interfaces they provide
- packages are intended to be distributed
 - perhaps merely shared w/team members
 - perhaps to be used by world-wide consumers
- distribution raises stability requirements
 - owner can no longer fix all affected code
 - owner cannot control client upgrades
 - some clients may have trouble adapting

Class Design

14

Class Packaging Principles

- Release/Reuse Equivalency Principle
 - “the granule of reuse is the granule of release”
 - (create opaque, cohesive packages)
 - if someone only needs classes A and B, package and support a package containing only those.
- Common Closure Principle
 - “classes that change together travel together”
 - (avoid strong inter-package coupling)
 - if class B depends on the implementation of class A, deliver both of them in a single package.

Class Design

15

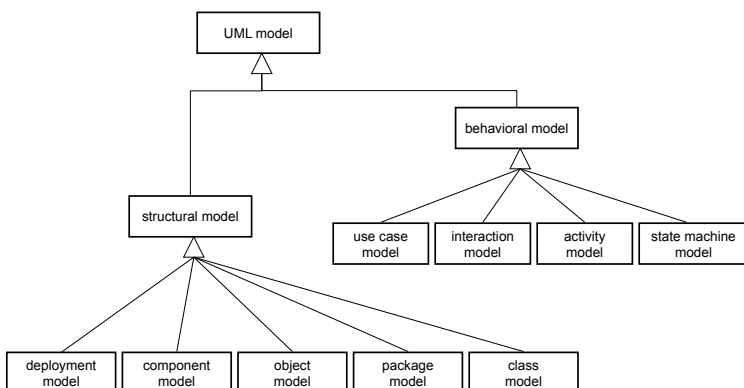
UML Class Models

- describe classes and static relationships
 - these are not models of run-time objects!
- boxes represent classes
 - may have 3 parts: **name**, attributes, methods
- lines represent class relationships
 - inheritance (source is derived from target) \triangle
 - aggregations (multiple instances of target) \diamond
 - compositions (source is sum of the targets) \blacklozenge
 - dependency (source uses target) $----->$
 - associations (source refers to target) \longrightarrow

Class Design

16

UML Class Inheritance



Class Design

17

UML Class Properties

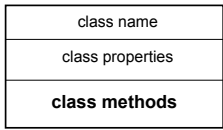
class name	<i>visibility</i> name : type [#] = default { properties }
class properties	EXAMPLES: dueDate
class methods	+ ... public - ... private ~ ... package # ... protected + grade: int {readonly} ~ students: string[1..100]

- specification may be complete
 - all properties listed w/complete declarations
- specification may be partial
 - list only properties/information “of interest”
 - list only properties different from parent class
 - may list no types (or even properties) at all

Class Design

18

UML Class Methods

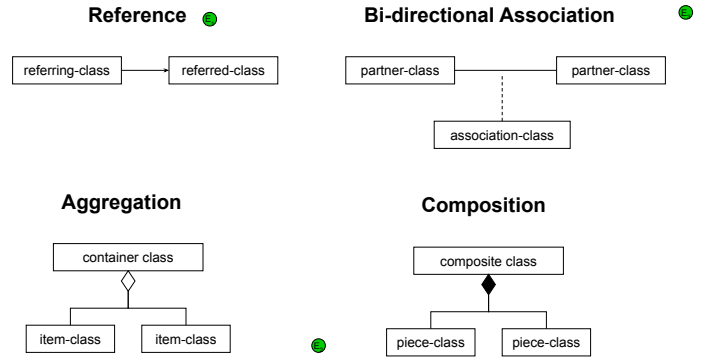


visibility **name** (parameters) : return type { properties }

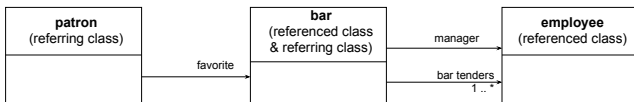
EXAMPLES: addStudent
 deposit(amount : dollars)
 cancelJob(reason) : boolean

- specification may be complete
 - all methods listed w/complete declarations
 - parameters can be defined as **in**, **out**, **inout**
- specification may be partial
 - list only methods “of interest”
 - simple get/set methods are routinely ignored
 - specify only non-obvious return types
 - specify only key parameters, non-obvious types

UML Class Associations

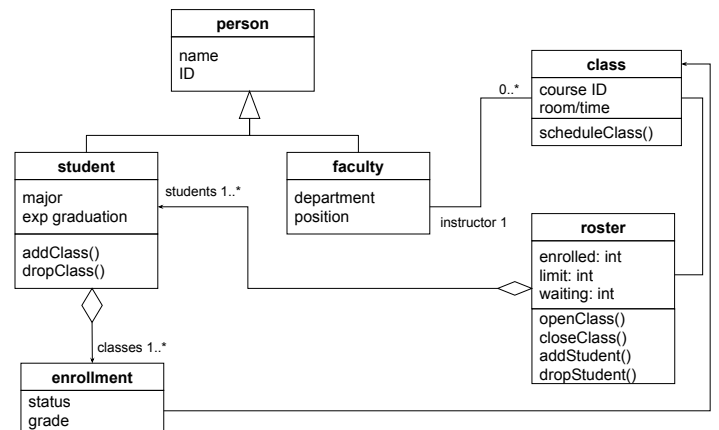


Labeling UML Associations



- names & counts are association-specific
 - these are **class** diagrams, **not objects**
 - classes may share multiple associations
- labels go on target end of association
 - name by which **this** object is known
 - number of **this** object that can be referred to
 - this becomes an issue for bidirectional associations
- you can also label the association line itself
 - to explain, or to distinguish among multiple associations

UML Class Diagrams

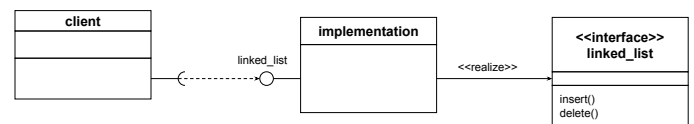


UML Class Dependencies



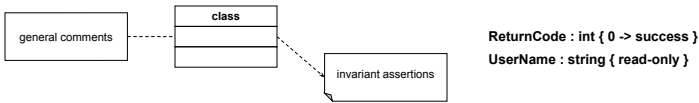
- Partnerships:
 - <<call>>
 - <<create>>
 - <<instantiate>>
 - <<permit>>
 - <<use>>
- Sub-classes, etc
 - <<derive>>
 - <<realize>>
 - <<refine>>
 - <<substitute>>
 - <<trace>>

Consumers, Providers & Interfaces



- Classes that define an interface are labeled as <<interface>> classes in their title blocks.
- Classes that implement interfaces export them in named interface circles
- Classes that require an interface provider can indicate this need with an external socket.

Constraints & Comments



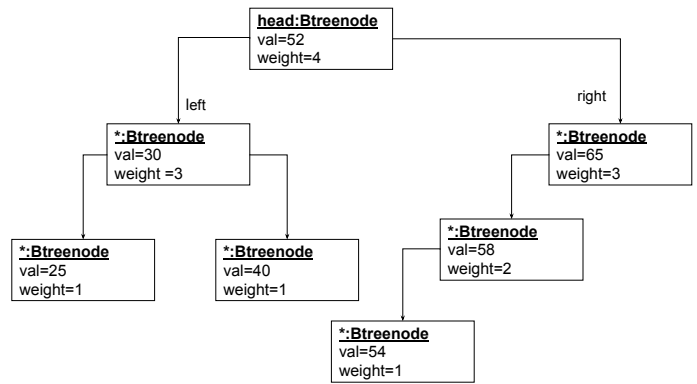
- any property or method can have constraints or comments after it { ... }
- free standing comment boxes can be added anywhere, with dependency lines to indicate where they apply.
- any line can be annotated w/description of the dependency or operation.

For the next lecture

- McConnell ch 7-7.2 (skim 7.3-7.7)
 - routine design and development
- McConnell, ch 9-9.3
 - pseudo-code programming
- McConnell, ch 18
 - table driven methods
- Agile: UML Sequence Diagrams
 - representing multi-component interactions

Supplementary Slides

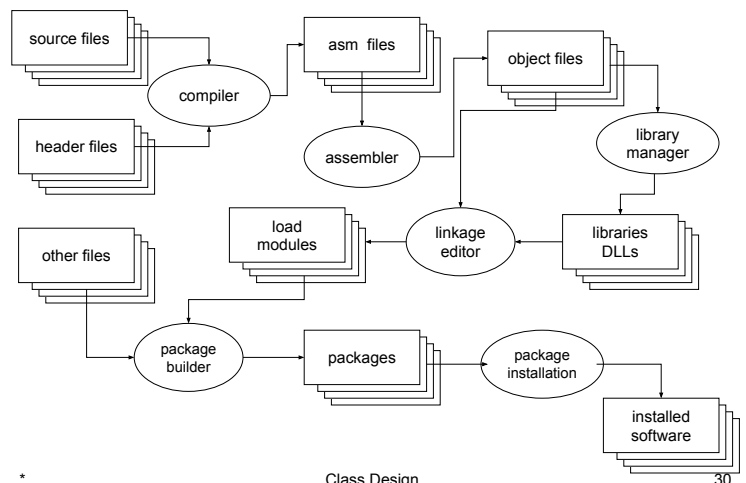
UML Object Diagrams



(UML Object Models)

- describe relationships among instances
 - these are specific instance relationships
 - not general (possible) class relationships
- each box represents an object
 - names are of the form: instance:classname
 - interesting properties are shown w/values
 - ranks of boxes used for factory classes
- lines represent associations
 - association name may appear on the line
 - only interesting associations are shown

Whense all these components?



UML Package Models

- describe package contents/relationships
- package is a collection of related classes
 - each could be described by a class diagram
 - contained within a single, large, package box
- tab-folders represent packages
 - with the package name at the top
- dashed lines represent dependencies
 - source package uses the target package

UML Package Dependencies

