Component Level Class Design

- Midterm: elegance, reqts, specs, design
- Specifications, Design, and Components
- Packages and Classes
  - patterns and reasons for class creation
  - elements of good class design
  - classes in non object oriented languages
  - class packages
  - elements of good packaging
- Diagramming Classes & Objects
  - UML class models
  - UML object models

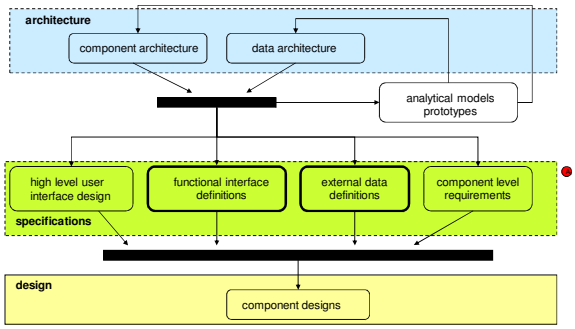
Mathematical Elegance

- Common elegance:
  - exhibiting refined, tasteful beauty in movement, appearance or manners
- Etymology:
  - from Latin *eligere* … to select with care
- Mathematical Elegance (a subjective term)
  - recognized as excellent work by experts in field
  - not obvious from the problem,
  - not following from any known rules
  - unusually succinct
  - based on (surprisingly) minimal assumptions
  - easily generalized to solve similar problems

Reqts/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

Model Hierarchy/Succession



What is a "component"?

- A modular, deployable, and replaceable part of a system, that encapsulates implementation, and exposes a set of interfaces.
  - it is a defined piece of a larger system
  - it can be added or removed from that system (not necessarily a **F**ield **R**eplaceable **U**nit)
  - it contributes to the working of the system
  - its inner mechanisms may be hidden
  - its functionality is defined by an interface

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

Design Patterns

- Much is written about design patterns
  - architectural, class structure, algorithmic
  - good solutions to recurring problems
  - I encourage you to study about them
- view them as Chess gambits or Go joseki
  - using them will improve your game
  - studying them will improve your understanding
  - but in the end, they are merely tactical aids
    - they can't analyze the board for you
    - they aren't a substitute for strategy

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

Characteristics of a good class

- it is well abstracted
- 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

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

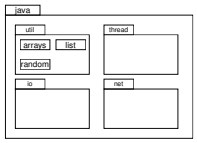
Classes in non-OO languages

- 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

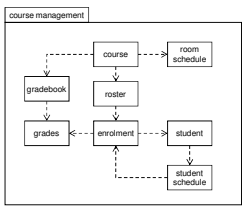
"Program into your language, not in it."

UML Package Contents

Package Nesting



Contained Classes



(Class Packages)

- 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

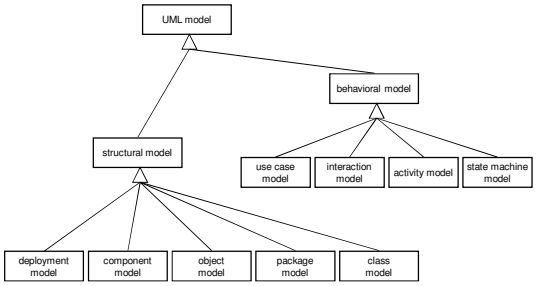
Class Packaging Principles

- Release/Reuse Equivalency Principle
  "the granule of reuse is the granule of release"
  (keep your packages cohesive)
  if someone only needs classes A and B, don't force him to take the unrelated class C as well.
- 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.

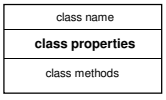
UML Class Models

- 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)
  - associations (source refers to target)
  - aggregations (multiple instances of target)
  - compositions (source is sum of the targets)
  - dependency (source uses target)

UML Class Inheritance



UML Class Properties



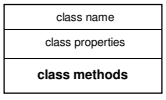
visability **name:** type [#] = default { properties }

\+ … public
\- … private
~ … package
# … protected

EXAMPLES: dueDate
assgtName: string
+ 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

UML Class Methods

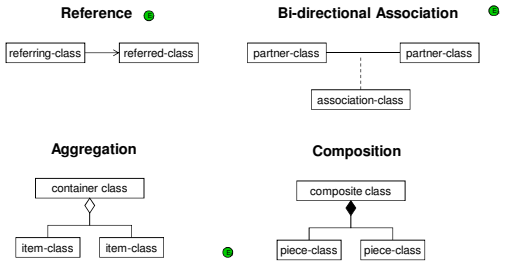


visability **name (***parameters***) :** return type { properties }

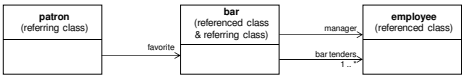
EXAMPLES: addStudent
deposit(amount : dollars)
cancelJob(reason) : boolean

- 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

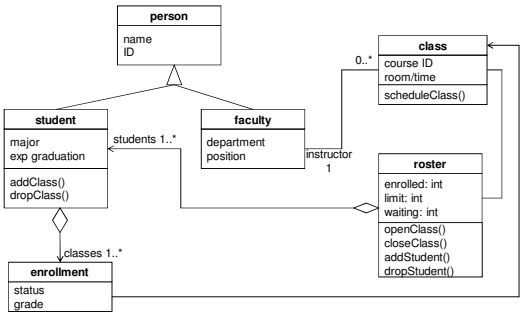


Labeling UML Associations

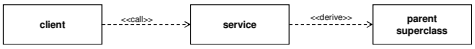


- names & counts are association-specific
  - these are class diagrams, not objects
  - classes may share multiple associations
- labels go on target end of association
  - name by which this object is known
  - number of this object that can be referred to
  - this becomes an issue for bidirectional 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>>

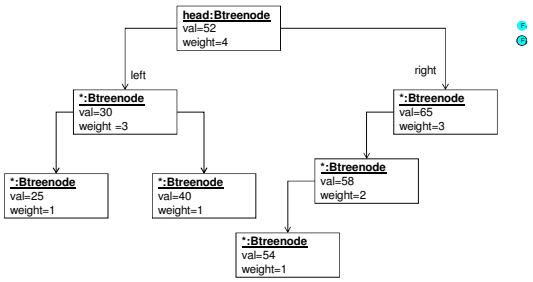
Consumers, Providers & Interfaces

- 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.

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

For the next lecture

- McConnell ch 19.6, 22
  - Code Complexity, Testing theory and practice
- Wikipedia: software testing
  - types, terminology, schools of thought
- Kampe: Introduction to S/W test cases
  - risk & testing, black/white-box testing, plans
- Cornett: Code coverage
  - types and relative advantages
- (review) Kampe: Testability

For the next lecture

- McConnell ch 7, 9
  - routine design and development
- McConnell, ch 18
  - table driven methods
- UML Interaction Diagrams
  - diagramming messages and calls
- Ambler: UML State Diagrams
  - representing finite state machines
- Wikipedia: selected algorithmic patterns

For the next lecture

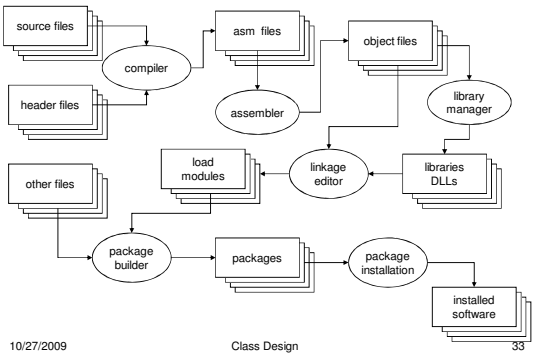
- Wikipedia: Design Patterns
  - brief introduction
- Model View Controller Architecture
  - a general U/I paradigm
- The Bridge design pattern
  - decoupling abstraction from implementation
- The Visitor design pattern
  - walking complex structures
- Nguyen/Wong: Design patterns for games

Supplementary Slides

Further Reading

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design patterns: Elements of reusable object-oriented software, Addison-Wesley, 1995
- Mary Shaw, David Garlan, Software Architectures: Perspectives on an emerging discipline, Prentice-Hall, 1996

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