Component Level Design

- Components and Specifications
- Packages and Classes
  - elements of good class design
  - classes in non object oriented languages
  - elements of good packaging
- Routines
  - elements of good routine design
  - representing routine designs

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 Field Replaceable Unit)
  - it contributes to the working of the system
  - its inner mechanisms may be hidden
  - its functionality is defined by an interface

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
- make applications more stable & portable
  - isolating implementation specifics in a class
  - abstraction protects app from future evolution
- compartmentalize complexity
  - bring all related code into a single place
  - simplify interface seen by rest of system

Component Specifications

“The definition of what a program is expected to do”

- a description of a component's function
  - usually, in terms of its output values
    - for specified input values
    - under specified preconditions.
- this is different from
  - system requirements (focus on component)
  - system architecture (more complete)
  - component design (what, not how)

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**

- **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 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**
  - not the same as 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 changes to class A regularly break class B, deliver both of them in a single package.

---

**Component Specification vs. Design**

- **Component/Class Specification**
  - creates (relatively) fixed interfaces
  - is driven by encapsulation and abstraction
  - views every component as a black box
- **Component/Class Implementation Design**
  - very few things are ever cast in concrete
  - the public/private distinction becomes weaker
  - the user/implementer distinction vanishes
  - what routines do becomes a mere detail
  - how the component works is everything
Where do routines come from?

- many are already defined for us
  - our public methods (external entry points)
- some emerge naturally from our approach
  - just as ADTs emerge from problem domain
  - some methods and functions will be obvious
  - easier to specify because they are private
- many (most?) are artifacts of our solution
  - we create them to simplify the implementation
  - routines can do this in many ways

Why create a new routine?

- creating useful private classes
  - create better abstractions to work with
  - we may even derive private sub-classes
- detail encapsulation
  - move complex sequences out of main code
  - segregate portable from non-portable code
  - hide-wrap global data structures
- centralize a recurring computation
  - one copy of an oft-repeated code sequence
  - enable interception of key operations

elements of good routines

- simplicity and clarity
  - obvious what routine does, how to use it
- good abstraction is still important
  - a well thought out function is easier to use
- information hiding is still important
  - avoid shared data, distributed algorithms
  - interactions mean complexity and bugs
  - encapsulate nasty details within a routine
- cohesion is still valuable
  - shorter routines are easier to understand

Routine Level Designs

- all routines are not simple
  - many embody complex algorithms
  - many cases to handle, many decisions
- such designs must be put in writing
  - help designer flesh out, record the design
  - present design to others for review
  - basis for implementation, white-box testing
- such designs can still be high level
  - they need not spell out simple/obvious steps

Representing Routine Designs

- there are many possible representations
  - prose: e.g. pseudo-code
  - graphical: e.g. UML activity or state diagrams
  - tabular: enumerating cases and handling
  - formal: e.g. Object Constraint Language
- none is intrinsically superior to the others
  - but each has advantages for some problems
  - some may have development tool support
- Choose one that makes sense
  - but, “when in Rome, do as the Romans do”

Mid-Term Exam

- Format
  - closed book, 12 question
- Scope
  - entirely based on key learning objectives (L1-15)
  - primarily (3/4) focused on issues and concepts
  - less (1/4) focus on representations and techniques
- Difficulty
  - 3 simple similar to quiz questions
  - 6 moderate similar to in-class discussion points
  - 3 moderate questions, not discussed in class
  - I prepared the answer sheet in about 30 minutes
Supplementary Slides

Whense all these components?

Pseudo Code

```plaintext
if local request
  find the record
else
  do remote
    if error
      return failure
    create new transaction
    including record
    return transaction id
find record:
  hash the key
  run down the chain
  if end of chain
    allocate new record
    label with this key
    return record pointer
else
  translate error
  return error
```

macros

```plaintext
#define BoundsCheck(s,x,y,z)
    {if (x<y || x>z) log_error("bounds",s)}
```

- compile-time functions
  - expanded at compile time, duplicated code
- advantages
  - can be changed by compile time options
  - faster, no procedure calls, stack maintenance
- disadvantages
  - multiple copies, take up more space
  - can't have local storage (static or dynamic)