# CS 494

# Object-Oriented Analysis & Design

## Packages and Components

## in Java and UML

# Readings

- Any Java text on packages
  - E.g. *Just Java 1*, in Chapter 5

# Packages in Java

- **A collection of related classes that form a library**
- **Also, packages in Java are <u>namespaces</u>**
  - **Avoid name-clashes.**
- **Usually means .java and .class files in a directory tree that mimics package structure**
  - **E.g. for the class called A.B.SomeClass, then the files will be:**
    - **<sourceroot>/A/B/SomeClass.java**
    - **<classroot>/A/B/SomeClass.class**
  - **Not required: could be in a database somehow**
  - **Note some IDEs (e.g. Eclipse) give a package view (better than a physical directory view of the files)**

# Packages in Java (reminders)

- Putting classes into packages. At top of file:
  package edu.virginia.cs494
- No package statement in file? Still in a package: the *default* package
  - Recall if you don't declare something public, private or protected, it has "default visibility"
  - "Real" programmers always use packages ☺

# Compiling and Running

- To compile:  javac <filename>
  - Example:  javac edu\uva\cs494\Foo.java
- To run:  java <classname>
  - Run-time starts looking at one or more "package roots" for a class with the given name
  - Example:  java edu.uva.cs494.Foo
  - The argument is not a file!  It's a class.
  - Where to look?  CLASSPATH variable
    - Also, you can list jar files in this variable

# jar files

- Bundles package directory structure(s) into one file
  - Like a zip file
  - Easier to distribute, manage, etc.
  - Let Java run-time know to look in a jar file, or Make the jar file “clickable” like a .EXE file
- Note: think of jar files as components (like DLLs)
  - If you recompile a .java file, must update the jar file

# UML and Packages

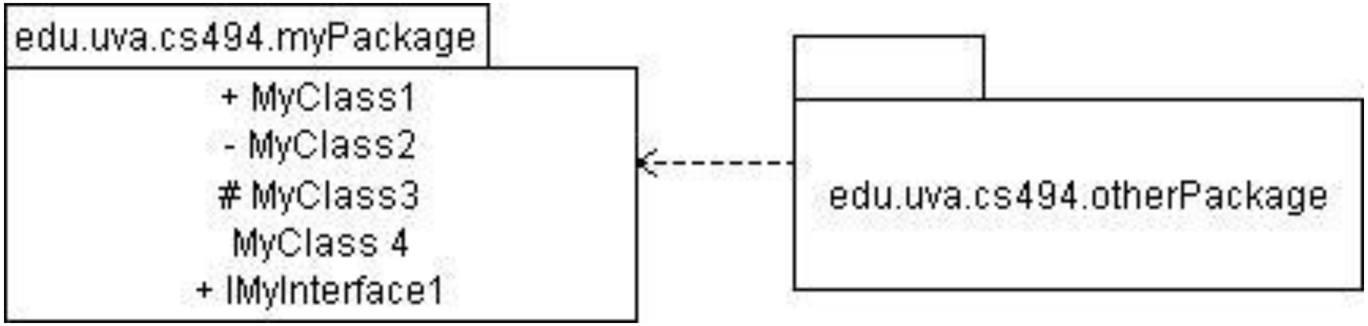
- UML supports a way to group model elements
  - Calls this a package.  Roughly equivalent to Java packages.
  - Can be applied to any UML modeling element, not just classes
- Some UML tools rely on UML packages to organize their models
  - E.g. Visio, Together

# UML Packages and Java

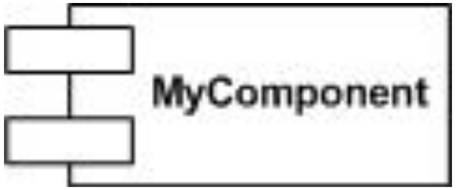
- For Java, want to show:
  - What packages exist
  - What's in them
  - How they depend on each other
- Create a class diagram with just packages
  - Think of it as a "package diagram" (but this is not a standard UML term)
  - List what classes (or classifiers) are in it
  - Show dependencies

# Drawing Packages in UML

- Symbol looks like folder icon
  - Name in tab or in “body”
  - Can put classifiers names in body with visibility (but not with Visio ☹)
- Dashed arrows mean dependencies
  - Code in otherPackage must use a class in myPackage
    - Not just import the package.  Use a class somehow.
- Can next packages; tag them; stereotype them; etc.



# UML Component Diagrams



- UML also has a diagram to show components
  - And also *deployment diagrams*:
    show how they're deployed *physically* (perhaps on different nodes)
  - Both of these are higher-level design views, e.g. architectural
- Component means physical module of code
  - In Java, a jar file
- Do we need this in CS494?
  - Probably not:  packages are probably enough
  - But, one component (e.g. a jar file) can contain more than one package

# Principles of Package Design

- How to group classes? How to analyze a package?
- General principles
  - Gather volatile classes together
    - Isolate classes that change frequently
  - Separate classes that change for different reasons
  - Separate high-level architecture from low-level
    - Keep high-level architecture as independent as possible

- From Robert Martin's work
  - *UML for Java Programmers*
  - *Agile Software Development: Principles, Patterns, and Practices*

# REP: Release/Reuse Equivalency Princple

- We reuse packages not individual classes
- One reason to create a packages is to create a reusable "component"
- "Granule of reuse is the granule of release"
- Author should maintain and release by package
  - Release management: older versions, announce changes, etc.
  - More trouble to do this for individual classes!

# CCP: Common Closure Principle

- **Classes in a package should be closed against the same kind of changes.**

- **Group classes by susceptibility to change**
  - **If classes change for the same reason, put them in one package**
  - **If that change is required, that entire package changes**
    - **But no other packages**

# CRP: Common Reuse Principle

- Classes in a package are reused together.
  If you reuse one class, you will reuse them all.
  - Group related things together for reuse.
- If scattered, then changes will affect multiple packages
  - And more things many depend on multiple packages
- Try not to include classes that don't share dependencies
- This is a form of "package cohesion"

# ADP: Acyclic Dependencies Princple

- Allow no cycles in the package dependency graph.

- When cycles exist
  - in what order do you build?
  - what’s affected when package X is modified?

- Note we’ve moved on to “package coupling”.

# SDP: Stable Dependencies Principle

- Depend in the direction of stability.
  - A package should not depend on other packages that are less stable (i.e. easier to change)
  - Target of a dependency should be harder to change

- A package X may have many incoming dependencies
  - Many other packages depend on it
  - If X depends on something less stable, then
    by transitivity all those other packages are less stable

# SAP: Stable Abstractions Principle

- A package should be as abstract as it is stable
- How to keep a package stable? If it's more "abstract", then other can use it without changing it
  - Like the Open/Closed Principle for classes (OCP)
  - Extend but don't modify

# Package Metrics Tool: JDepend

- Tool that processes Java packages and provides package-level metrics
- Benefits (from the author)
  - Measure Design Quality
  - Invert Dependencies
  - Foster Parallel, Extreme Programming
  - Isolate Third-Party Package Dependencies
  - Package Release Modules
  - Identify Package Dependency Cycles

# JDepend Metrics (1)

- **Number of Classes and Interfaces**
  - number of concrete and abstract classes (and interfaces)
  - an indicator of the extensibility of the package.
- **Afferent Couplings (Ca)**
  - number of other packages that depend upon classes within the package
  - an indicator of the package's responsibility
- **Efferent Couplings (Ce)**
  - number of other packages that the classes in the package depend upon
  - an indicator of the package's independence

# JDepend Metrics (2)

- **Abstractness (A)**
  - ratio of the number of abstract classes (and interfaces) to the total number of classes
  - range for this metric is 0 to 1
    - A=0 indicating a completely concrete package
    - A=1 indicating a completely abstract package

# JDepend Metrics (3)

- **Instability (I)**
  - ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca)
  - an indicator of the package's resilience to change
  - range for this metric is 0 to 1:
    - I=0 indicating a completely stable package
    - I=1 indicating a completely instable package

# JDepend Metrics (4)

- **Distance from the Main Sequence (D)**
  - perpendicular distance of a package from the idealized line A + I = 1
  - an indicator of the package's balance between abstractness and stability
- **Package Dependency Cycles**
  - package dependency cycles are reported

# JDepend Links

- Home for JDepend
  - http://www.clarkware.com/software/JDepend.html
- OnJava article:
  http://www.onjava.com/pub/a/onjava/2004/01/21/jdepend.html
- Eclipse plug-in: JDepend4Eclipse
  - http://andrei.gmxhome.de/jdepend4eclipse/