Packages

Jim Fawcett

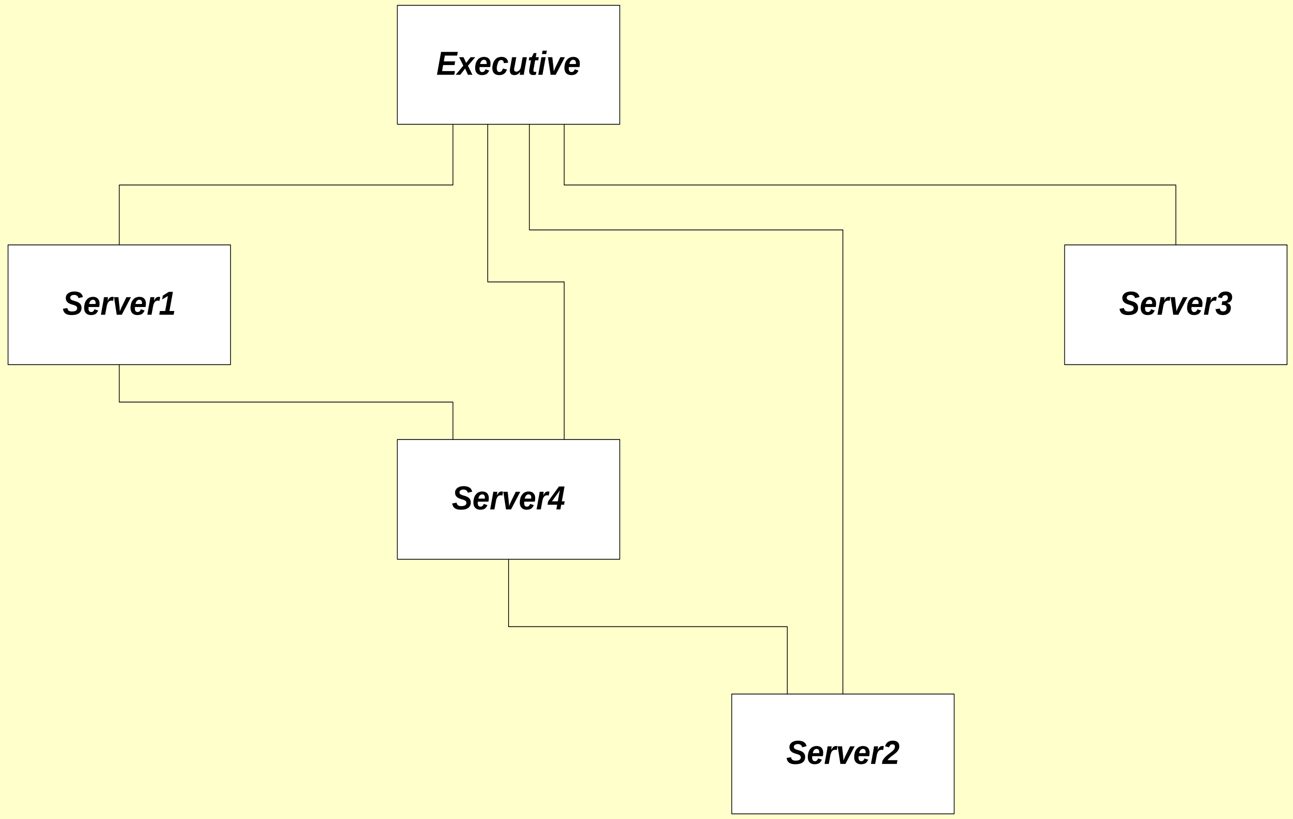
CSE687 – Object Oriented Design

Spring 2010

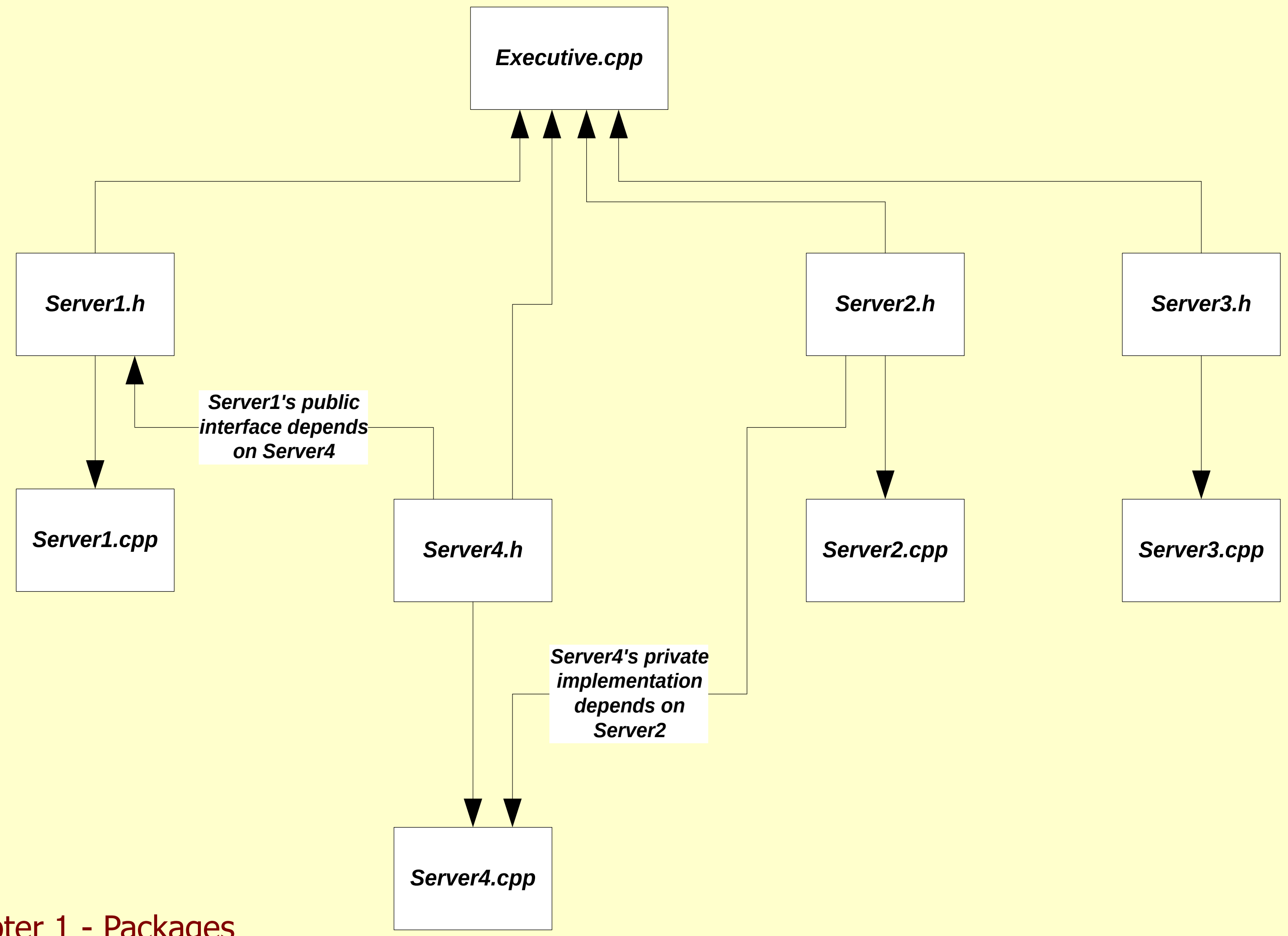
Package Definition

- A *Package* is a physical packaging of source code into files.
- A ***server*** Package provides services to higher level Packages.
- An ***executive*** Package coordinates the activities of the program's server Packages.

A Typical Structure



Include Relationships


Executive.cpp
Server1.h
Server2.h
Server3.h
Server1's public interface depends on Server4
Server1.cpp
Server4.h
Server2.cpp
Server3.cpp
Server4's private implementation depends on Server2
Server4.cpp

Package Definition

- A ***Package*** is a physical package of source code into files.
  - Each reusable Package should focus on a single activity, e.g., it is cohesive. Most server Packages should attempt to be reusable.
  - Not all Packages should try to be reusable. Executive Packages are not - they must focus on all of the program's responsibilities – but at a very high level.
- A ***server*** Package provides services to higher level Packages. A C++ Package consists of:
  - A header file (.h extension)
    - Has a typed interface, provided to ensure type consistency with all Packages using its services.
    - Announces the services of the Package to potential users through a manual page and function and class declarations.
  - An implementation file (.cpp extension) that provides the Package's services through global function and class member function implementations.
    - This file provides a prologue, the Package's definitions, and a test stub.
    - Test stubs are used for construction tests to quickly check the correctness of recently added or modified code.
- An ***executive*** Package coordinates the activities of the program's server Packages.
  - Often an executive Package only has an implementation file since it has no need to announce it's services.

A Note about this Definition

- In all of the design courses covered by this web site, we use the definition for Packages provided on the previous page, and you are expected to implement all of your submitted projects in this modular fashion.
- I often use the term Module as a synonym for Package. However, not everyone uses this definition. For example, throughout Microsoft's .Net literature they use the term Module to mean a binary file that becomes part of an assembly, e.g., a dll or an exe.
- Some others use the preceding definition in a looser way, for example, allowing multiple header files for a single implementation file. This is expressly forbidden by our definition.

Why Two Files?

- Why not allow a Package to consist of just a header file?
  - You often see this done to define macros and inline functions.
- Header files announce services which, occasionally, can be implemented as inline functions or macros.
  - Even if everything a header declared is implemented inline, the Package should provide a test stub.
  - If the services are important enough to place in their own Package, then they are important enough to be tested before including in a larger system.
- The only time a Package with only a header file is permitted in this course is when the header defines an interface, but no more.
  - Interfaces have no implementation.
  - They just declare a contract for services, provided by other Packages.

Limitation of Single File Program

- **What's wrong with a single file program?**
  - Nothing, but as programs grow larger, this format becomes very limiting.
  - Programs grow too large to comprehend as a single entity.
  - Eventually, as file size grows, a compiler can not swallow the whole file.

Limitation of Single File Program

- **Solution:**
  - break programs up into Packages
  - Packages allow a program to be decomposed into smaller units of understanding, test, release, and configuration management.
  - In C++ each server Package consists of two files:
    - A header file contains all public declarations so other Packages know how to use this Package's services
    - an implementation file contains all data and function definitions; that is, it provides the services that the header file announces

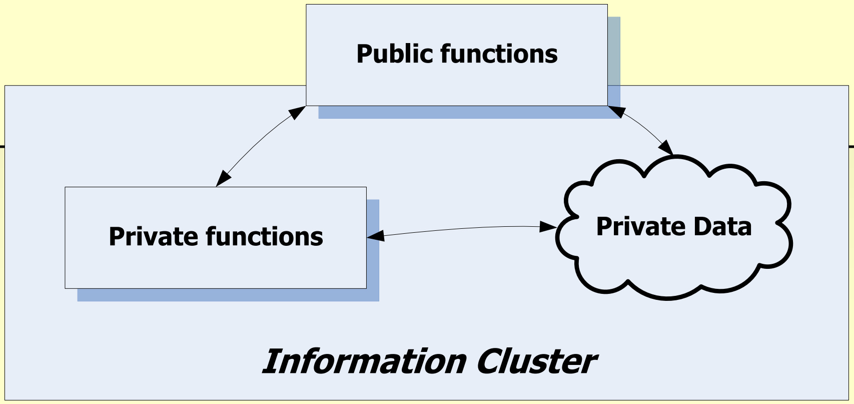
Information Cluster

Logical View:

- An information cluster encapsulates complex, sensitive, global, or device dependent data, along with functions which manage the data, in a container with internals not accessible to client view.
- Public access is provided by a series of accessor and mutator functions. Clients don’t have access to private functions or data.

Implementation:

- C Package using file scope for encapsulation. All private functions and global data are qualified as static.
- C++ class using public, protected, and private keywords to implement public and protected interfaces. Each class (or small, intimately related group of classes) should be given its own Package.
- Each Package implementing an information cluster contains a manual page and maintenance page which describe the logical model for the cluster and its chronological modification history.



Package Structure

Package Header File

```
#ifndef PKGNAME_H
#define PKGNAME_H
         - manual page goes here
         - maintenance page goes here
         - declarations go here
#endif
```

These preprocessor directives are essential. They prevent a header file from being compiled more than once, even though it is included multiple times in headers included in the cpp file being compiled.

Package Implementation File

```
         - prologue goes here

#include "pkgname.h"

         - function definitions go here

#ifdef TEST_PKGNAME

void main() {
         - test code goes here
}
#endif
```

This preprocessor directive allows a designer to compile the Package alone, for test, but to then remove the main test stub for compilation as part of a larger program.

What goes in a Package's Files?

- ***Elements of header file:***
  - Manual page
    - Prologue listing file name, platform, and author.
    - Description of Package operation
    - Illustration of its public interface
  - Maintenance page
    - Build process
    - Maintenance history
  - **Declarations** of the Package's public classes and public global functions
  - Definitions of **inline** functions, e.g., inline function bodies
  - Function bodies of public **template-based** member functions and global functions
  - Definitions for public **constant** data items
  - Includes of other header files with declarations of types used in this header
    – **no other files should be included**.

What Goes in a Package’s Files?

- ***Elements of Implementation File***
  - Prologue, e.g., top part of the manual page, listing file name, platform, and author. This should be a clone of the prologue inserted at the beginning of the header’s manual page, except that the file name is changed.
  - Includes of header files declaring types needed by the code in this implementation – **<u>no other files should be included</u>**.
  - Implementation of the Package’s global and member functions.
  - Test stub – a main function which tests each function implemented in this file. It is also used to illustrate how a client would use the Package and what its outputs look like.
  - The test stub must be enclosed by compilation guards which enable compilation only when TEST_PKGNAME is defined.

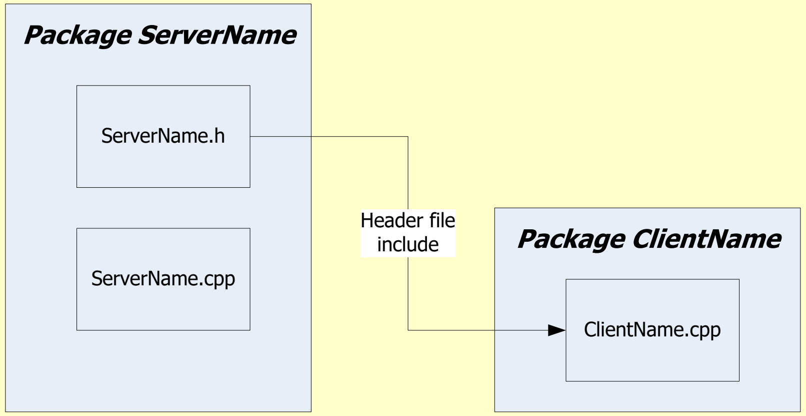
Include Rules

- Syntax:
  - Include system headers with angle brackets, e.g.,
    #include <iostream>.
  - Include local headers with quotes, e.g.,
    #include “server1.h”
  - **Never, ever, include implementation (.cpp) files**. That makes managing build processes very difficult.
- Semantics:
  - Always include header files at the lowest possible level:
    - If a Package’s implementation depends on another Package, include the other’s header file in the implementation file, <u>not the header</u>.
    - Do not include a header simply to pass it on to another file.

Modularity

- "In object oriented languages classes and objects form the logical structure of the system; we place these abstractions in Packages to produce the system's physical architecture."
  Grady Booch, Object Oriented Design with Applications,
  Benjamin/Cimmings, 1991
- Modularization consists of dividing a program into Packages which can be compiled separately. C++ performs type checking across Package boundaries, using included header file declarations.

- Packages in C and C++ are simply separately compiled files.
- We place Package interface declarations in header files.
- Package implementations are placed in separate files which include the header file at compilation time via a preprocessor #include "pkg_name.h" directive.



Incremental Development Model

Begins with requirements analysis, development of architecture and preliminary design. Result should be a design concept and partitioning into Packages.

As soon as Packages are defined, one is selected that has no dependencies on other Packages (at least one almost always exists). This Package's development proceeds by implementing one or two functions, adding tests of the new capabilities to the test stub, and verifying nominal operation.

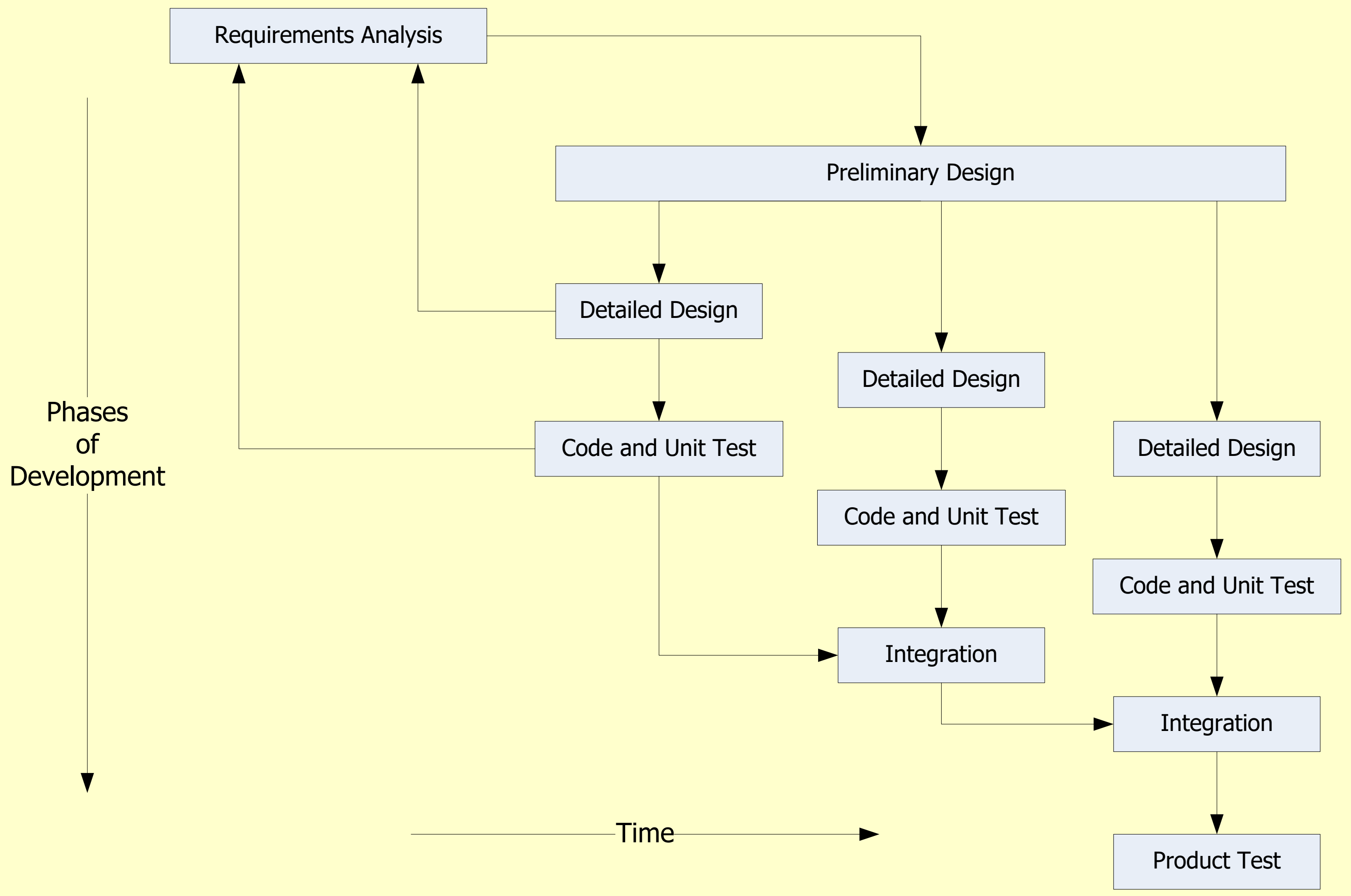
This process continues iteratively until Package is complete. Detailed unit test then begins. Its goals are to exercise all paths and predicates in code to find all latent errors, correct them, and verify.

Next a Package is selected which depends at most on the tested Package. It is subjected to same process to develop a complete and verified Package. It is then integrated with the first Package.

This continues until all the Packages have been integrated. Development completes by carrying out product test to demonstrate that software meets all its contractual obligations.

The outstanding virtue of this approach is that only a small amount of new code is brought into a throughly tested baseline at each stage of development.

Incremental Development



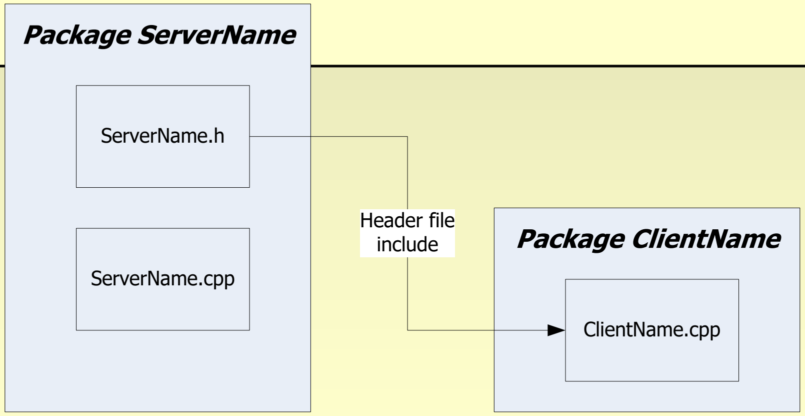
Modular System

Logical Model:

collection of information clusters communicating through public interfaces.

Implementation:

- One executive client Package and a series of server Packages.
- One C or C++ server Package for each major program activity.
- The executive Package is the coordinator.
- Each server Package has:
  - an implementation file containing a test stub, with conditional compilation controls for stand-alone testing.
  - a header file which announces the Package's services to clients, e.g., the other servers and/or the client Package.



Limitations of Packages

- **What's wrong with modular decomposition?**
  - nothing, but this format alone is not very flexible
  - Packages are defined statically (at compile time)
  - new instances can not be created at run time
  - this means that the flexibility afforded by run-time creation of data does not extend to Packages which contain both functions <u>and</u> data

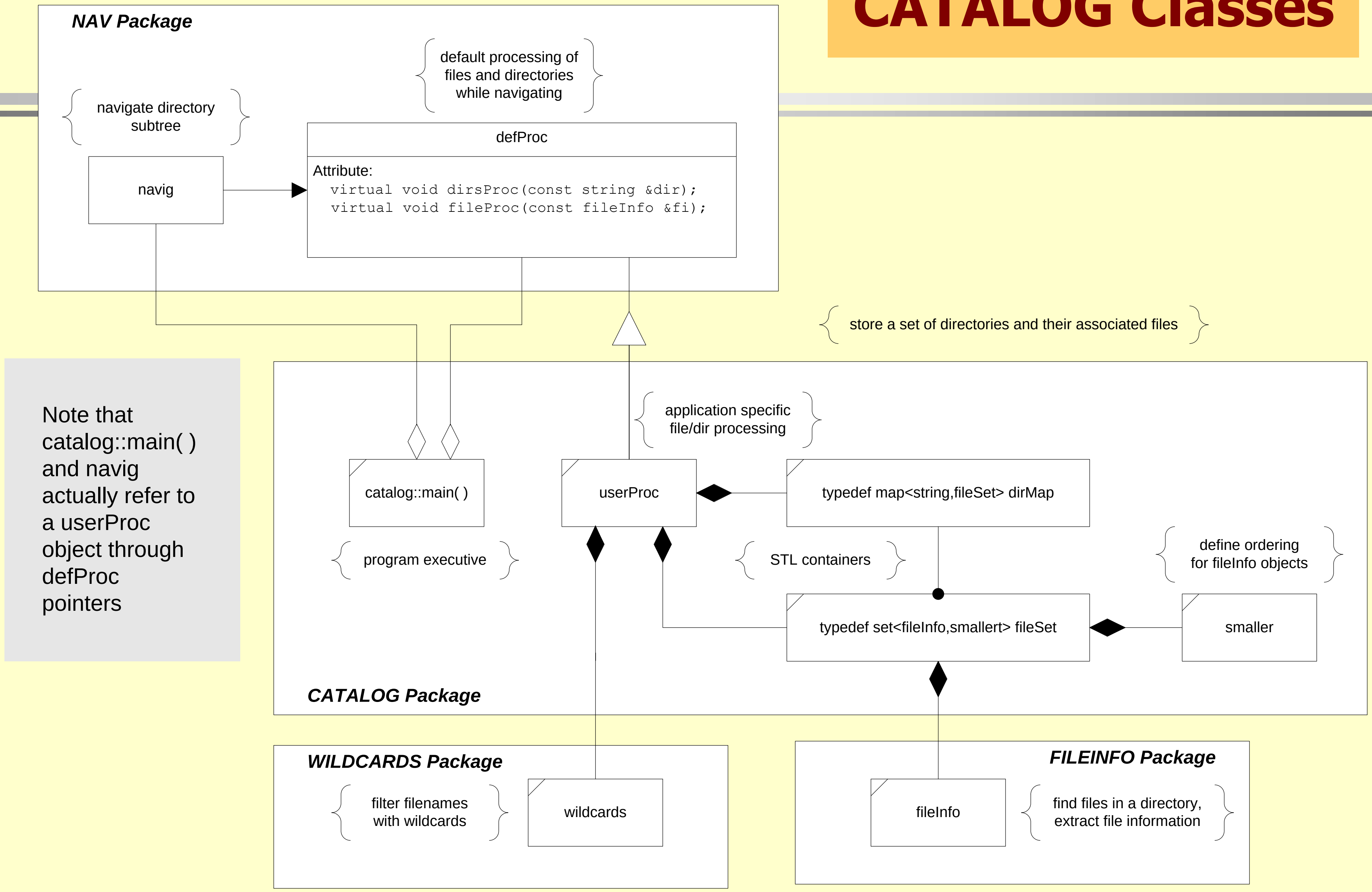
Limitations of Packages

- **Solution:**
  - classes support declaration of objects which can be defined either statically or dynamically
  - classes define both functions and data. An object, which is simply an instance of a class, can be defined statically in static memory, in local memory (on the stack), or dynamically on the heap, just like any intrinsic data type
  - a program can define as many objects as it needs up to the amount that your computer memory can hold
  - The classes you define will, of course, be packaged in Packages.

Package Example

- The following slide shows the decomposition of a Catalogue program.
  - Catalog simply walks a directory sub-tree and collects an ordered list of directories and their contents.
    - When it completes, it holds the catalog information in an STL data structure for any subsequent processing the designer might like to add.
  - It nicely illustrates the use of classes and Packages for the logical and physical packaging of the program.
  - Besides being a nice illustration of a modular program, it also provides a lot of functionality you will need in Project #1.

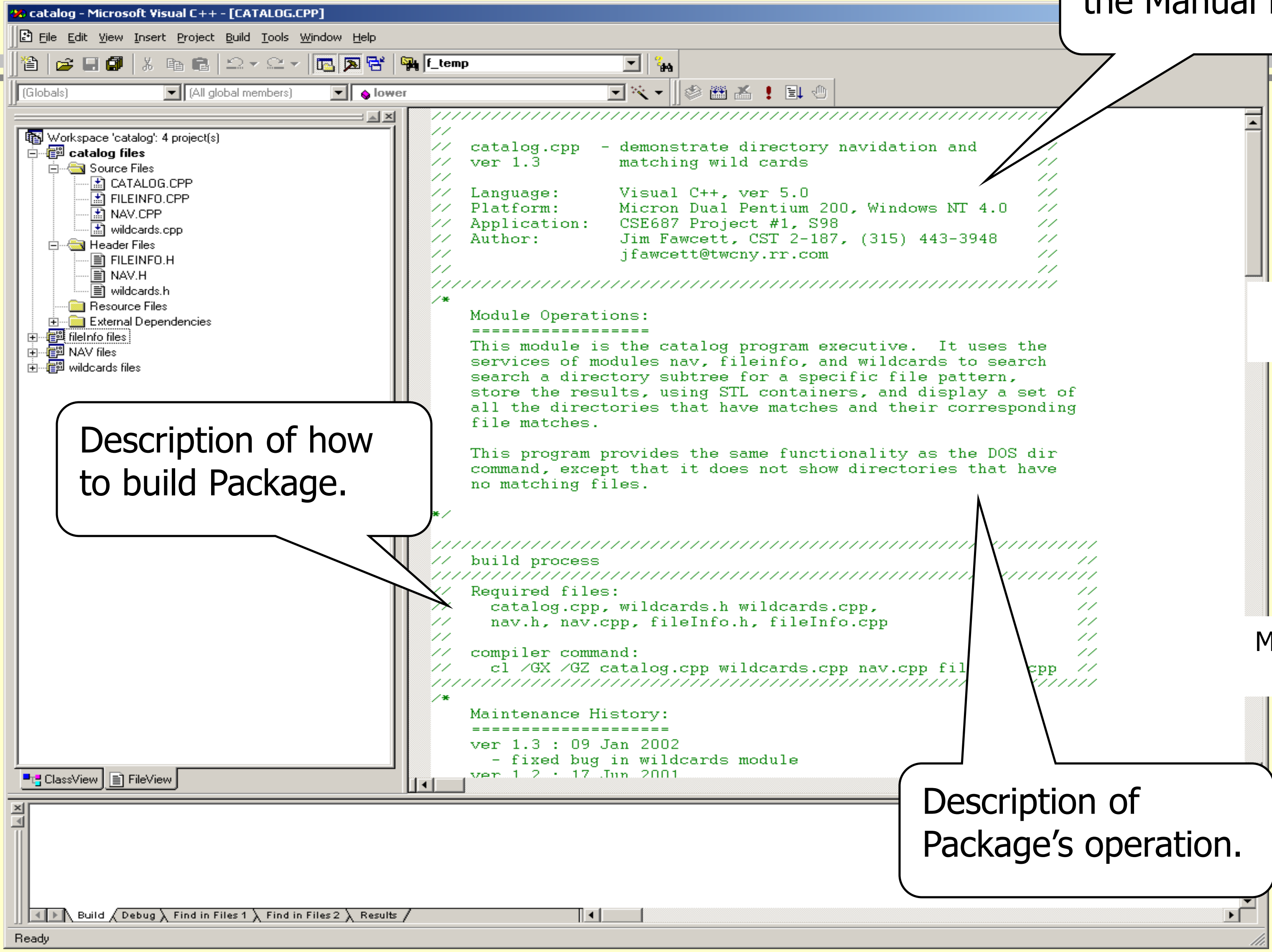
CATALOG Classes



NAV Package
navigate directory subtree
default processing of files and directories while navigating
navig
defProc
Attribute:
virtual void dirsProc(const string &dir);
virtual void fileProc(const fileInfo &fi);
store a set of directories and their associated files
application specific file/dir processing
catalog::main( )
userProc
typedef map<string,fileSet> dirMap
program executive
STL containers
define ordering for fileInfo objects
typedef set<fileInfo,smallert> fileSet
smaller
CATALOG Package
WILDCARDS Package
filter filenames with wildcards
wildcards
FILEINFO Package
fileInfo
find files in a directory, extract file information

Note that catalog::main( ) and navig actually refer to a userProc object through defProc pointers

CATALOG Program



```
///////////////////////////////////////////////////////////////////
//                                                               //
//  catalog.cpp  - demonstrate directory navidation and          //
//  ver 1.3        matching wild cards                           //
//                                                               //
//  Language:      Visual C++, ver 5.0                           //
//  Platform:      Micron Dual Pentium 200, Windows NT 4.0       //
//  Application:   CSE687 Project #1, S98                        //
//  Author:        Jim Fawcett, CST 2-187, (315) 443-3948        //
//                 jfawcett@twcny.rr.com                         //
//                                                               //
///////////////////////////////////////////////////////////////////
/*
    Module Operations:
    ==================
    This module is the catalog program executive.  It uses the
    services of modules nav, fileinfo, and wildcards to search
    search a directory subtree for a specific file pattern,
    store the results, using STL containers, and display a set of
    all the directories that have matches and their corresponding
    file matches.

    This program provides the same functionality as the DOS dir
    command, except that it does not show directories that have
    no matching files.

*/

///////////////////////////////////////////////////////////////////////
//  build process                                                    //
///////////////////////////////////////////////////////////////////////
//  Required files:                                                  //
//    catalog.cpp, wildcards.h wildcards.cpp,                        //
//    nav.h, nav.cpp, fileInfo.h, fileInfo.cpp                       //
//                                                                   //
//  compiler command:                                                //
//    cl /GX /GZ catalog.cpp wildcards.cpp nav.cpp fil        cpp    //
///////////////////////////////////////////////////////////////////////
/*
    Maintenance History:
    ====================
    ver 1.3 : 09 Jan 2002
      - fixed bug in wildcards module
    ver 1.2 : 17 Jun 2001
```

Prologue, part of the Manual Page

Description of how to build Package.

Description of Package's operation.

Manual Page

Maintenance Page

End of Packages Presentation