Appendix 1 - Packages

Jim Fawcett
copyright (c) 1999-2010

Limitation of Single File Program

- **What’s wrong with a single file program?**
  - nothing, but as programs grow larger, this format becomes very limiting
  - all functions are at the same (global) level; that’s what C does, you don’t have a choice
  - programs grow too large to comprehend as a single entity
  - eventually, as file size grows, the compiler can not swallow the whole file
- **Solution:**
  - break programs up into packages
  - each package consists of one or two files:
    - C++ uses header files that contain all public declarations so other packages know how to use the package’s services
    - A C++ implementation file contains all data and function definitions; that is, it provides the services that the header file announces
    - C# has no header files, instead placing all implementations inline in class declarations.

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++ and C# perform type checking across package boundaries.
  - C++ uses header files to accomplish this.
  - C# uses metadata, embedded in every assembly.

Information Cluster

Model Type:

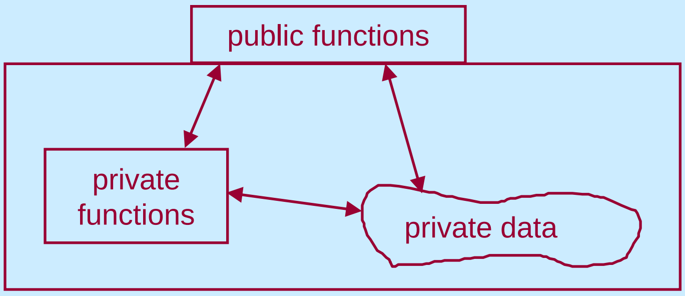
abstract system model

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#, 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. Each package includes a test stub with compilation controls.



Packages

- C# Packages
  - A C# package consists of a single source code file with the extension ".cs".
  - It starts with a Manual Page and Maintenance Page and ends with a Test Stub.
  - The test stub is surrounded with compilation guards so that it is compiled when "TEST_PACKAGENAME" is defined as a compilation constant.
  - Each package is given its own project which includes the package file and the files of any packages it depends upon.
- C++ Packages
  - A C++ server package consists of two files:
    - Header file with extension ".h" which declares the public services of the package with class and global function declarations.
    - Implementation file with extension ".cpp" which implements the public services of the package and may also implement private classes and global functions to simplify its development and maintenance.
  - The header file starts with Manual and Maintenance pages. The entire header is surrounded by compilation controls to avoid duplicate declarations.
  - The implementation file starts with a prolog block (which is also the first part of the header's Manual Page) and ends with a Test Stub, surrounded by compilation controls so it is compiled only when "TEST_PACKAGENAME" is defined as a preprocessor pragma.

Modular System

Model Type:

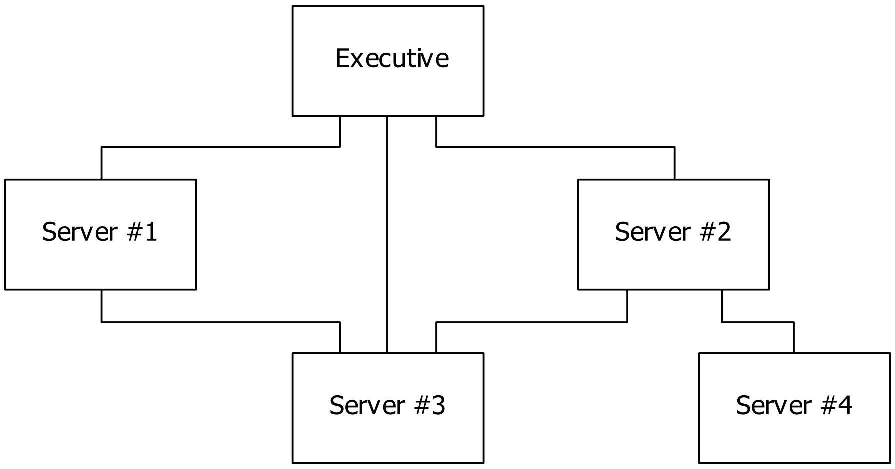
abstract system model

Logical Model:

collection of information clusters communicating through public interfaces. Each information cluster is packaged in a file or files.

Implementation:

- One ***Executive*** package and a series of ***Server*** packages.
- One C, C++, or C# server package for each major program activity.
- The Executive package is responsible for all program activities, using the services of dedicated servers.
- Each server package:
  - Implements one cohesive set of responsibilities.
  - Is self documenting and self testing through the use of Manual Pages, Maintenance Pages, and Test Stubs.



C# Package Implementation

C# has no header files, so each appropriately decorated source file is a package.

Each package begins with a manual page and maintenance page:

Manual page describes a logical view of the package and provides brief descriptions of all public services provided.

Maintenance page provides a chronological record of changes to the package since its creation, cited by version number.

The file may also contain a design notes page which describes organizing principles, major data structures, and key processes which set the course of the package's design.

The test stub is the last part of the file and is guarded to permit compilation without the stub in the context of a larger system:

```
#if(TEST_MODNAME)
  static void Main(string[] args)
  {
     // tests go here
  }
#endif
```

C++ Package Implementation

Each C++ package begins with preprocessor guard statements which prevent multiple declarations, e.g.:

```
#ifndef MODNAME_HPP
#define MODNAME_HPP
         :
        header body
         :
#endif
```

The endif statement is the last in the header file.

The test stub is the last part of implementation file and is also guarded to permit compilation without the stub:

```
#ifdef TEST_MODNAME
  int main(int argc, char* argv[]) {
           :
  return err_code;
}
#endif
```

Each package's header file begins with a manual page and maintenance page:

Manual page describes the logical view of package and provides brief descriptions of all public services provided by the package

Maintenance page provides a chronological record of changes to the package since its creation, cited by version number.

Either header file or implementation file may also contain a design notes page which describe the organizing principles, major data structures, and key processes which set the course of the package's design.

C# Package Structure

File: modname.cs

manual page goes here
maintenance page goes here

declarations with inline implementations go here

```
#if(TEST_MODNAME)
Static void Main(string[] args)
{
        - test code goes here
}
#endif
```

C++ Package Structure

Header File: modname.h

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

Implementation File: modname.cpp

```
        - prologue goes here
#include “modname.hpp”
        - function definitions go here
#ifdef TEST_MODNAME
void main(int argc, char* argv[])
{
        - test code goes here
}
#endif
```

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.

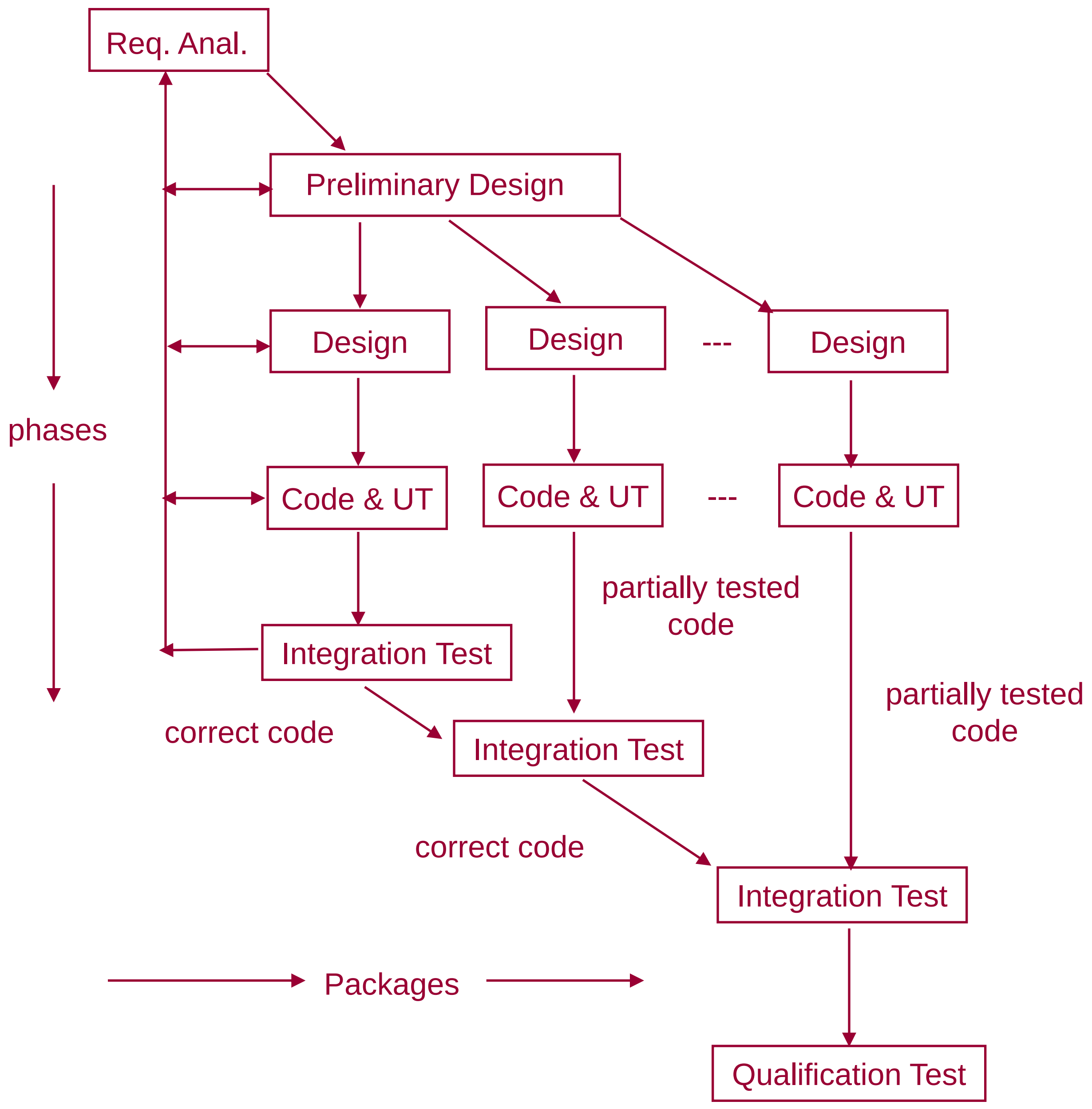
This process continues iteratively until package is complete. Detailed unit test then begins. Its goals are to exercise all paths and predicates in code to find all latent errors, correct them, and verify.

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 qualification 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 devel-opment.

Incremental Development



Limitations of Packages

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

End of Presentation