Software Design:
Goals and Attributes

Jim Fawcett
CSE687 – Object Oriented Design
Spring 2009

Software Development

Architecture Design Implementation

← Strategy → ← Tactics →

- Strategy:
  Concept
  - Rationale, options

  Organizing ideas and structure
  - Uses
  - Partitions and responsibilities
  - Critical issues

- Tactics
  Implementing ideas and structure
  - Activities
  - Classes and relationships
  - Algorithms
  - Data management

Design Goals

Make each software Component:

- **Simple**
  - Small functions
  - Low Cyclomatic Complexity
  - Small number of functions
- **Understandable**
  - Self documented
  - Descriptive names
  - Simple
- **Maintainable**
  - Simple, flexible, and robust
- **Selectable**
  - Capability summary
  - Keywords
- **Reusable**
  - Selectable, understandable, low fan-out (not counting framework lib calls)
- **Reliable**
  - Repeatable behavior
  - Free of latent errors
- **Robust**
  - Will not crash with unexpected inputs or environment
- **Flexible**
  - Changes don't propagate
  - Supports substitution of implementations
- **Extendable**
  - Supports addition of new implementations

Simplicity

- Small functions
  – Lines of code ≤ 50
- Low cyclomatic complexity
  – All functions CC ≤ 10
  – Average much lower
- Small number of functions
  – Functions per module ≤ 20
  – Average much lower
- Measurable by size and complexity

```
 04/05/2007  08:57:59 AM     13640  Sockets.cpp
==================================================
   cyclo lines function name
     3    32   SocketSystem::GetLastMsg
     3    14   SocketSystem::SocketSystem
     2     9   SocketSystem
     1     6   SocketSystem::getHostName
     3    22   SocketSystem::getIpFromName
     2    10   SocketSystem::getNameFromIp

     1     4   Socket::Socket
     1     4   Socket::Socket
     1     5   Socket::Socket
     1     1   Socket::Socket
     1     5   Socket
     1     6   operator=
     3    14   Socket::connect
     1     5   Socket::disconnect
     1     4   operatorSOCKET
     5    25   Socket::send
     5    19   Socket::recv
     1     7   Socket::getLocalIP
     2    12   Socket::getLocalPort
     2    12   Socket::getRemoteIP
     2    12   Socket::getRemotePort

     3    22   SocketListener::SocketListener
     1     6   SocketListener
     1     7   SocketListener::waitForConnect
     1     6   SocketListener::stop
     1     4   SocketListener::getLocalIP
     1     4   SocketListener::getLocalPort
     3    97   main

 04/05/2007  08:57:59 AM      4905  Sockets.h
==================================================
   cyclo lines function name
  - type:   class SocketSystem
  - type:   class Socket
     1     1   error
     1     1   getHandle
  - type:   class SocketListener
```

Socket is Almost Simple

- Small functions
- Low complexity
- Interface is fairly large
  - 15 member functions
- Couples well with SocketListener

Understandable

- Self documented
  - Manual page
    - read about operations and interface
  - Maintenance page
    - see how to build
  - Test stub
    - see how it works
- Descriptive names
  - Name describes operation or result
- Simple
- Measurable by detecting decorations

```
////////////////////////////////////////////////////////////////////
//  Tokenizer.h - Reads words from a file                          //
//  ver 1.4                                                        //
//                                                                 //
//  Language:      Visual C++ 2005                                 //
//  Platform:      Dell Dimension 9150, Windows XP SP2             //
//  Application:   Prototype for CSE687 Pr1, Sp06                  //
//  Author:        Jim Fawcett, CST 2-187, Syracuse University     //
//                 (315) 443-3948, jfawcett@twcny.rr.com           //
////////////////////////////////////////////////////////////////////
/*
  Module Operations:
  ==================
  This module defines a tokenizer class.  Its instances read words from
  an attached file.  Word boundaries occur when a character sequence
  read from the file:
  - changes between any of the character types: alphanumeric, punctuator,
    or white space.
  - certain characters are treated as single character tokens, e.g.,
    "(", ")", "{", "}", "[". "]", ";", ".", and "\n".
  A tokenizer is an important part of a scanner, used to read and interpret
  source code.

  Public Interface:
  =================
  Toker t;                          // create tokenizer instance
  returnComments();                 // request comments return as tokens
  if(t.attach(someFileName))        // select file for tokenizing
    string tok = t.getTok();        // extract first token
  int numLines = t.lines();         // return number of lines encountered
  t.lines() = 0;                    // reset line count
```

Tokenizer is Understandable

- Simple model
- Simple interface
- Cohesive
- Couples only to input stream

Maintainable

- Maintenance consists of
  - Fixing latent errors
  - Modifying existing functionality
  - Adding new functionality
- Is maintainable if:
  - Needs no maintenance
    - So simple it obviously has no defects
    - Additions made by composing with new components
  - Easy to fix, modify, and extend
    - Used through interface so changes don't propagate
    - Interface can be bound to new implementations
    - Simple so easy to test
- Only indirectly measurable

```
class IAction
{
public:
  virtual ~IAction() {}
  virtual void
   doAction(SemiExp& se)=0;
};

class IRule
{
public:
  virtual ~IRule() {}
  void addAction(IAction*
   pAction);
  void doActions(SemiExp& se);
  virtual bool doTest(SemiExp&
   se)=0;
protected:
  std::vector<IAction*>
   actions;
};
```

```
class Parser
{
public:
  Parser(SemiExp& se);
  ~Parser();
  void addRule(IRule* pRule);
  bool parse();
  bool next();
private:
   ITokCollection* pTokColl;
  std::vector<IRule*> rules;
};
```

Parser is Maintainable

- Very simple structure
- Very simple operation
- Partitions activities into Parsing, Rules, and Actions
- Very loose coupling
- Example of Open/Closed Principle

Selectable

- Five million lines of code project
  - Has roughly 10, 000 packages
    - Average of 500 lines of code per package
      - 10 functions with 50 lines of code each
- Need ways to find parts to salvage and reuse
  - Need to make quick decisions
    - Localize candidates by functionality or application
      - has operational summaries in prologue and manual page
  - Need to quickly evaluate candidates
    - Easy to build
      - has maintenance information with build process
    - Easy to run
      - has test stub
- Measurable by detecting decorations

```
////////////////////////////////////////////////////////////////////////
//  blockingQueue.cpp - queue that blocks on deQ of empty queue     //
//  ver 1.0                                                           //
//  Language:        Visual C++, ver 7.1, SP 2                        //
//  Platform:        Dell Dimension 8300, Win XP Pro, SP2             //
//  Application:     CSE687 - Object Oriented Design                  //
//  Author:          Jim Fawcett, CST 2-187, Syracuse Univ            //
//                   (315) 443-3948, jfawcett@twcny.rr.com            //
//                                                                    //
////////////////////////////////////////////////////////////////////////
/*
    Module Operations
    =================
    This module provides a simple thread-safe blocking queue, based
    on the STL queue container adapter.  When a client thread attempts
    to deQ an empty queue, it will block until another thread enQs an
    item.  This prevents very high CPU utilization while a reading
    thread spin locks on an empty queue.

    Public Interface
    ================
    BQueue<std::string> Q       // create blocking queue holding std::strings
    Q.enQ("an item");           // push onto queue
    std::string str = Q.deQ();  // pop from queue
    size_t s = Q.size();        // number of elements in queue
    Q.clear()                   // remove all contents from queue
*/
```

BlockingQueue is Selectable

- Simple functionality
- Simple interface
- Clear Manual Page
- Clear Maintenance Page
- Test Stub
  - Easy to see what BQueue does

Reusable

- Selectable
  - Has prologue and Manual Page
- Understandable
  - Has module operation description
  - Meaningful names
  - Simple structure
- Low fan-out
  - Dependent on very few other components
- Needs no application specific code
  - Uses delegates
  - Provides base class "hook"
- Fan-out and selectable/maintainable are measurable

```
class defProc
{
public:
  virtual ~defProc() { }
  virtual void dirsProc(const std::string &dir);
  virtual void fileProc(const fileInfo &fi);
};

class navig
{
public:

  navig(defProc &dp);                       // accept user defined proc
  ~navig();                                 // restore user's dir
  void start(std::string dir, const std::string& fileMask="*.*");
                                           // start dir navigation
  std::string getPath();

private:
  static const int PathBufferSize = 256;
  void walk(const std::string &dir, const std::string& fileMask);
                                            // directory walker
  std::string userDir_;                     // user's working directory
  defProc &dp_;                             // provides extendable processing
                                            //   of file and directory names
};
```

Navig is Reusable

- Provides a base class “hook” called defproc
- Application code derives from defproc so that Navig calls application code whenever it encounters a file or directory.

Reliable

- Understandable model
- No surprises
  - Operates according to known model
  - Processing is repeatable
  - No race conditions or deadlocks
- Thoroughly tested
- Probably only measurable “after the fact” by keeping statistics on bugs and testing.

Tokenizer Maintenance

Maintenance History:
====================
ver 1.4 : 10 Feb 07
- fixed bug in braceCount to eliminate changing count when brace
  is in a quoted string or comment
ver 1.3 : 24 Feb 06
- fixed bug in eat comment that hung on ending comment with no
  newline, by testing for stream state good.
ver 1.2 : 06 Feb 06
- added stream closing to destructor and attach memeber functions
ver 1.1 : 01 Feb 06
- added if test at end of getTok() to avoid returning space after
  C comment as a token
ver 1.0 : 12 Jan 06
- first release

Tokenizer is Reliable

- Code is not simple
  - Many special cases that you may not think of while designing
- It took awhile to get there
- Kept records of bugs and fixes
- Responded to bug reports

Robust

- Will not crash with unexpected inputs or environment
  - – Use partitions to isolate processing
    - Interfaces, AppDomains, COM components, controls
  - – Use exception handling to trap unexpected events
  - – Validate user input, especially strings and paths
- Indirectly measurable by looking for partitions, exception handling, and validation code.

```
Parser* ConfigParseToConsole::Build()
{
  try
  {
    // configure to detect and act on preprocessor statements
    pToker = new Toker;
    pSemi = new SemiExp(pToker);
    pParser = new Parser(*pSemi);
    pPreprocStatement = new PreprocStatement;
    pPrintPreproc = new PrintPreproc;
    pPreprocStatement->addAction(pPrintPreproc);
    pParser->addRule(pPreprocStatement);
    // configure to detect and act on function definitions
    pFunctionDefinition = new FunctionDefinition;
    pPrintFunction = new PrintFunction;
    pFunctionDefinition->addAction(pPrintFunction);
    pPrettyPrintFunction = new PrettyPrintFunction;
    pFunctionDefinition->addAction(pPrettyPrintFunction);
    pParser->addRule(pFunctionDefinition);
    return pParser;
  }
  catch(std::exception& ex)
  {
    std::cout << "\n\n  " << ex.what() << "\n\n";
    return 0;
  }
}
```

ConfigParse is Robust

- Uses try and catch blocks
- Returns exception message consistent with application
  - Uses cout for console application

Flexible

- Changes don’t propagate
  - Provide an interface so users don’t bind to your implementation
  - Change to some implementation detail won’t cause changes to other components
- Supports changes of implementation
  - Interfaces guarantee substituability of any implementing class
  - Template parametrization supports compile-time substitution.
- Weakly measurable, by looking for interfaces and template parametrization.

```
template <thread_type type>
class Thread
{
public:
  Thread(Thread_Processing& p);
  ~Thread();
  void start();
  void wait();
  static void wait(HANDLE tHandle);
  unsigned long id();
  HANDLE handle();
  void sleep(long Millisecs);
  void suspend();
  void resume();
  thread_priority getPriority();
  void setPriority(thread_priority p);
  void endThread(unsigned int exit_code);
private:
  Thread_Processing* pProc;
  HANDLE hThread;
  static unsigned int __stdcall threadProc(void *pSelf);
  unsigned int _threadID;
  thread_priority _priority;
  // disable copy and assignment
  Thread(const Thread<type>& t);
  Thread<type>& operator=(const Thread<type>& t);
};
```

Thread Class is Flexible

- Template policy supports
  - Stack-based default threads
    - Allows interaction while processing unfolds
  - Heap-based terminating threads
    - Fire-and-forget paradigm

Extendable

- Supports addition of new implementation
  - Use of interface and object factory supports adding new components
    - No changes to users of the interface and factory
    - Parser: easy to add new rules and actions
  - Templates support compile-time substitutability
    - Template policies support customization of behavior
- Weakly measurable, by looking for interfaces and template parametrization

Protocol DLL Demo

```
class protocol {

  public:
    virtual DLL_DECL int getInt()      = 0;
    virtual DLL_DECL void putInt(int) = 0;
    virtual DLL_DECL std::string passVal(std::string s)  = 0;
    virtual DLL_DECL std::string passRef(std::string &s) = 0;
    static  DLL_DECL protocol* makeObj();
              // static member object factory
};

extern "C" { DLL_DECL protocol* gMakeObj(); }
              // global object factory
```

Protocol Derived Classes are Extendable

- Use of
  - Interface
  - Object factory
  - DLL packaging

  Supports modification with no breakage or rebuilding of clients

Design Attributes

- Abstraction
- Modularity
- Encapsulation
- Hierarchy
- Cohesion
- Coupling
- Locality of reference
- Size and complexity
- Use of objects
- Performance

Abstraction

- Logical model or metaphor used to think about, and analyze, component
  - Toker
    - collect words from a stream
  - SemiExpression
    - group tokens for analysis
  - Parser
    - apply set of grammar rules to each semiExp
    - apply set of actions to each rule

Modularity

- Package abstractions in cohesive packages and modules
  - Parser applies rules
  - Rules do grammar detections
  - Actions respond to detections
  - Configure parser builds parts

Encapsulation

- Hide implemenation behind interface
  - Prevents binding to internal implementation details
  - Helps to prevent propagation of errors
- No non-constant public data
- Return pointers or references to private data only to give access to an object:
  - char& str::operator[](int n); O.K.
  - T* T::clone(); O.K.

Hierarchy

- Layering of responsibility. Each layer hides its decendents
  - Anal: Application level
  - Scanner: Processes documents
  - Parser: Mechanizes processing
  - Semi: Generates source for processing
  - Toker: Generates tokens
- Hierarchy is a dependency relationship
  - Inheritance, composition, aggregation, using

Cohesion

- Cohesive component is focused on a single activity
  - Parser: apply grammar rules to each semiExp
  - Rule: detect a specific gramatical structure
  - Semi: gather tokens for analysis
  - Toker: generate tokens from file

Coupling
How is data passed to functions?

- Narrow coupling
  - Only a few arguments
- Normal coupling
  - Requires no knowledge of the design of arguments or their references
  - No pointers, no structures
- Properly scoped
  - Explicitly entered into scope
    - Passed as argument
    - Declared in local scope
- No assumption coupling

Locality of Reference

- References to local data are easier to understand
  - We see the declaration
  - Know all the qualifiers
- Non local references can be powerful
  - Inheritance: base may be defined elsewhere
  - Composition: Composed may be defined elsewhere
  - Delegates: called functions may be defined elsewhere
- Global data is poster child for non-local reference
  - Hard to understand
  - Not powerful in any sense

Size and Complexity

- Large and complex packages and functions are:
  - Hard to understand
  - Hard to test
  - Hard to maintain
  - Hard to document
- Complex functionality + small simple modules
  - implies Lots of modules
  - implies need for fine-grained configuration control

Use of Objects

- Class is a form of information cluster
  - Provides a simple abstraction
  - Hides possibly complex implementation behind simple interface
  - Provides methods guaranteed to maintain integrity of state data while supporting user's data transformations
  - Class is responsible for managing all its needed resources
  - Manual page and test stub provide a lot of self documentation
- Inheritance, composition, aggregation, and using relationships provide effective modeling tools

Performance

- Performance is determined by:
  - Locality of calls
    - Within process, within machine, within network, across internet
  - Caching
    - Avoid unnecessary calls
  - Algorithms
    - Log, linear, log-linear, power law, exponential
  - Memory foot-print
    - Affects rate of page faults
  - Creation of copies
  - Creation and destruction of objects

Object Oriented Design

- Structuring design with classes and class relationships
  - Develop application-side objects:
    - Executive, WorkingSet (inputs), Analysis, Display
    - Supports reasoning about application
      - Requirements
      - Principles of operation
  - Develop solution-side objects:
    - Socket, SocketListener, BlockHandler
    - Supports reasoning about solution
      - Performance
      - Quality
      - Errors and Test

Design Principles

- ***LSP*** supports loose coupling
  - Don't need to bind to concrete names
- ***OCP*** demands flexibility and extendability
  - Don't modify, do extend
- ***DIP*** avoids rigid coupling
  - Depend on abstraction not implementation
- ***ISP*** supports cohesion
  - Factor to avoid bloated interfaces with inadvertant coupling of clients

Class Relationships

- Classes support several types of relationships:
  - **Inheritance** supports substitution
    - Derived classes are subtypes of base class
    - Derived class has access to public and protected, but not private members of base class
  - **Composition** supports ownership
    - Composed classes provide functionality through their public interfaces
    - Composer has no special access to private or protected members of composed
  - **Using** provides access to an object
    - Provides access to public members of an object without ownership

Inheritance Relationship

- Inheritance comes in two flavors
  - Inheritance of interface
    - Provides a public contract for service, but no implementation
    - interface ISomeIF { … } in C#
    - struct ISomeIF { // all pure virtual methods }; in C++
  - Inheritance of implementation
    - Provides a public interface
    - Provides implementation of one or more functions, fields, properties, and/or delegates in C#
    - Provides implementation of one or more functions and/or fields in C++

Inheritance Relationship

- Inheritance
  - "is-a"
  - Supports substitutability (polymorphism)
    - IMsgPass provides contract
    - Allows posting message to any substitute:
      - Executive, Comm, ToolUI, ToolLib
  - Supports inheritance of implementation
    - AWrapper provides:
      - BlockingQueue
      - asynchronous dequeuing on child thread

Inheritance Relationship

- A frequently recurring idiom is to provide three levels:
  - An interface providing a contract for service
  - An abstract class that provides the common part of an implementation for all derived classes
  - Derived concrete classes that complete the functionality provided by the hierarchy
- This is just what the ADAM Prototype does
- Note:
  It is considered to be a serious design flaw to have a deep inheritance hierarchy with concrete classes deriving from other concrete classes.

Composition Relationship

- Composition comes in two flavors:
  - **Strong Composition** supports a strong form of ownership
    - Composed lifetime is same as that of composer
    - Makes an instance of composed a field of composer
      - Supported by C++ but not by C# or Java
  - **Weak Composition** (Aggregation) supports a weaker form of ownership
    - Composer creates and disposes the composed in member functions
    - Composer holds references to composed objects on the heap
      - Supported by C++, C#, Java

Composition Relationship

- Composition
  - "owned-by", "part-of"
  - Provides layering
    - Supports building incrementally
    - Supports decomposition of testing
  - Provides strong encapsulation

Using Relationship

- Using
  - "used by"
  - References to "used" passed as arguments of a member function
  - User not responsible for creation or disposal

Implementation

- A module consists of:
  - Prologue identifying
    - Module
    - Platform
    - Application
    - Author
  - Manual page that discusses
    - Module operations
    - Its public interface
  - Maintanence Page
    - Build process
    - Maintenance History
  - Code structured as interfaces and classes
  - A test stub, e.g., a Main function surrounded by compilation constant guards

Modules in C#

- A GUI module consists of:
  - Three files defining a Form
    - A file containing event handlers
    - A file containing control declarations and designer code
    - A file containing resource information as an XML schema
  - A file providing a Main function that runs the form application
- A Console module consists of:
  - A file containing a class with a Main function
- A Library module consists of:
  - A single file containing one or more classes

Modules in C++

- Managed C++
  - GUI modules, Console Applications, and Library modules have the same structure as C# modules
- Unmanaged (standard native) C++
  - We tend not to build GUI modules in unmanaged C++
  - Console executive modules consist of one file that contains:
    - zero or more classes
    - one global main function
  - Library modules consist of two files
    - Header file with:
      - class declarations
      - inline function definitions
      - template class and function definitions
    - Implementation file with class member definitions

What makes a good implementation?

- Proper Encapsulation
  - No public data
  - Any functions that require design knowledge to call properly are private
- Error Handling
  - Input data is validated, especially strings and paths
  - Use Exception handling
- Make assumptions explicit
  - Use manual page to disclose any assumptions made about callers
- Make low-level modules reusable

What makes a poor implementation?

- Vague or imprecise abstractions:
  - Manual page should be clear, consise, and effective
  - Public interface should be small and consistent with the module's abstraction
- Lack of design modularity, encapsulation, and layering
  - Should have an executive module and server modules
  - Every form should delegate all of its computations to testable libraries
  - Oversize or complex functions
  - Modules and functions with poor cohesion
- Latent defects
- Unhandled exceptions

End of Presentation