CMSC433, Fall 2001
Software Components
and
Programming by Contract

Adam Porter
Nov. 27, 2001

"Reusable code"

- Say I develop a String class in C++
- It is used in 5 different applications
- Each of which is statically linked
  - a self-contained executable

Problems with static linking

- Redundant executable code
  - on disk and
  - in memory
- Can't update defective class without replacing application
- Can't add extensions of String class

Dynamic linking

- Only one copy of DLL on disk
  - and in memory
- Can upgrade/replace library
  - all applications that use class get upgrade

Dynamic linking problems

- All applications must use same link format and object layout
  - member alignment, virtual method table, virtual base classes, RTTI
- Class frozen in C++
  - adding variables or methods breaks everything
- Upgrade "breaks" your software
  - upgrade changed things it depended on

Versioning

- Every time DLL changes, give it a new version ID
- Application only uses DLL version ID it was compiled against
- Problems:
  - multiple version bloat
  - inability to upgrade/enhance

Compatible DLL's

- Binary compatibility
  - old code is able to invoke methods in new library
- Semantic compatibility
  - code that expected the old functionality won't be broken by the new functionality

Binary compatibility

- Need to agree on link and runtime standards
- Can't make changes that would violate compatibility
  - e.g., add/delete variables/methods in C++

Semantic compatibility

- Restrictions on changes in pre and post conditions
- Which can be "upgraded" to the other:
  - int search1(int[] a, int x)
    - pre: exists i s.t. a[i] = x
  - int search2(int[] a, int x)
    - pre: a sorted **and** exists i s.t. a[i] = x

Which upgrades are compatible?

- Which can be "upgraded" to the other:
  - int search1(int[] a, int x)
    - pre: exists i s.t. a[i] = x
    - post: a[retVal] = x
  - int search2(int[] a, int x)
    - pre: a sorted **and** exists i s.t. a[i] = x
    - post: a[retVal] = x
      and (retVal = 0 or a[retVal-1] != x)

Rules for semantic compatibility

- f has pre: P and post: Q
- f' has pre: P' and post: Q'
- f can be upgraded to f' iff
  - P => P'
    - makes precondition looser
  - Q' => Q
    - makes postcondition stricter

Problems with semantic compatibility

- Pre and post conditions are rarely specified
  - when they are, they are rarely complete and correct
- Programmers might depend upon conditions of your implementation rather than the interface
  - e.g., "I know that this function always returns a sorted array"

Programming to an interface

- An interface is a contractual binding of two (or more) parties
- Should not depend on examining implementations
- Documentation is essential
  - e.g., for MiniServlet, the fact that the servlet is responsible for closing the output stream

Indirect interfaces

- Function pointers, virtual method dispatch, callbacks
- A word processing component might invoke a grammar checker it didn't know existed

What belongs in a contract?

- Pre and post conditions
- Resource requirements
  - execution time
  - memory requirements
- Leads-to
  - e.g., "invokes all registered notifiers"

Callbacks

- In purely procedural word, library calls run to completion
- But allowing them to invoke methods in the caller code complicates matters
- A library function is often thought of as transitioning from one consistent state to another consistent state
  - possibly passing through inconsistent states

Directory Service w/Callback

```
Module Directory
    Type Notifier
            = Procedure (name: String);
    Procedure AddNotifier(n : Notifier);
    // pre: n != null
    // post: n is registered, will be called on
    //       AddEntry and RemoveEntry
    Procedure AddEntry(n : String, f: File);
    // pre: n != "" && f != null
    // post: ThisFile(n) == f
    Function ThisFile(n : name) : File
    // pre: n != ""
    // post: result = file named n
    //       or result = null and no such file
```

```
Module DirectoryDisplay
    Procedure myNotifier(n : String) {
      if Directory.ThisFile(n) == null
            // delete n in display
      else
            // n has been added, display it
    }
    ..
    Directory.AddNotifier(myNotifier);
    ...
```


What is missing?

- Precondition for notifier
  - that state of directory reflects change being notified about
  - or that it doesn't
- Postcondition for notifier
  - that it doesn't change the state of the directory

When is the notifier called?

- Before or after change?
- When file is renamed, how many times?
- When file is replaced, how many times?

Why is changing state bad?

- Say we attach a notifier that, whenever an entry with the name X.htm is added, renames it to X.html
  - violates postcondition of AddEntry

Locking doesn't help

- Could make Add/RemoveEntry lock directory
  - wouldn't help
  - notifier, and subsequent potential modification, are all done in same thread
- If locks weren't recursive, would just result in deadlock

Thread spawning?

- One way to handle renaming notifier
- Use locks
- When notified that X.htm has been added, spawn thread to do renaming
  - will have to wait to obtain lock
- What is the pre/post condition of a method that spawns a thread?

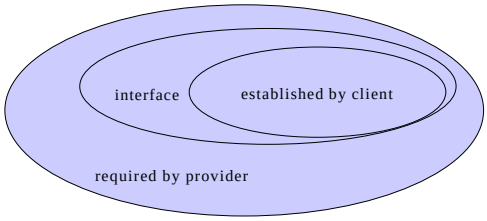
inProgress functions

- Add method that detects whether an operation is in progress
- Add precondition for operation that no other operation is in progress

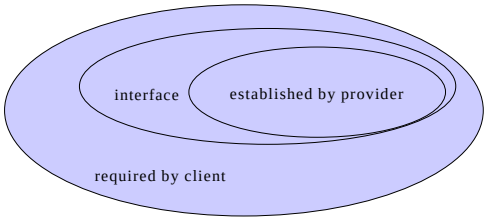
Substitutability

- Clients use an interface to invoke a method in a provider
  - pre: established by client
    => demanded by interface
    => required by provider
  - post: established by provider
    => demanded by interface
    => required by client

subtyping of preconditions



subtyping of postconditions



Co- and Contra- variance

- Co-variance = vary in same way
- Contra-variance = vary in opposite ways
- Co-variant return types are sound and useful
  - Supported in C++ but not in Java
- Contra-variant arguments are sound but rarely useful
  - Not supported in C++ or Java

Co and Contra variance

```
interface View {
      Model getModel();
      }
  interface TextView  extends View {
      TextModel getModel();
      }
  interface GraphicsView extends TextView {
      GraphicsModel getModel();
      }
```


Co and Contra variance

```
interface View {
      Model getModel();
      void setModel(Model);
      }
  interface TextView  extends View {
      TextModel getModel();
      void setModel(TextModel);
      }
```


Structural or Declared Subtyping

- Structural subtyping
  - B is a subtype of A iff B supports the interface of A
- Declared subtyping
  - B is a subtype of A iff B is declared as a subtype of A

Problems

- With declared subtyping
  - Can't write supertype of third-party class
    - e.g., TextView, subtype of TextObserver and View
- With structural subtyping
  - doesn't check (implied) pre and post conditions
  - Same names doesn't imply same meaning

Three Facets of Inheritance

- Implementation inheritance
  - reuse of code
- Interface inheritance
  - reuse of signatures
- Promise of substitutability
  - not in SmallTalk nor Eiffel
    - In Eiffel, can undefine methods

Fragile base class problem

- Syntactic fragile base class problem
  - Changing a class requires that all subclasses be recompiled
    - offsets of fields and methods changed
- Semantic fragile base class problem
  - Recompiling may not fix things

Semantic fragile base class problem

- Lots of calls between superclass code and subclass code
  - check those pre and post conditions
- Back and forth nature creates problems
  - code in superclass invokes code in subclass which invokes code in superclass
- Super/subclass setting/getting variables directly

Solutions

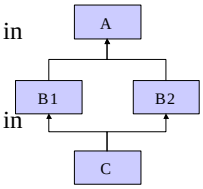
- Use private variables and setter/getter methods
- Use protected members when appropriate
- Figure out groups of mutually dependent methods
  - if A invokes B and B invokes A, then A and B are mutually dependent
  - have to override an entire set of MD methods

Multiple inheritance

- multiple supertyping
  - inherit interface from multiple supertypes
  - promise substitutability for all of them
- inheritance from multiple supertypes is tricky

Multiple inheritance of implementations

- What if a function is defined in both B1 and B2?
  - what version does C invoke?
- What if a function is defined in both A and B1?
  - what version does B2 invoke?
- Do B1 and B2 share an A or each have their own A?



More issues?

- Does it matter which is declared first, B1 or B2?
- In C++, B1 and B2 share an A if both declare A as a virtual base class
  - Don't know that B1 and B2 might need to share an A until class C is written
  - virtual base classes are less efficient

Virtual base classes