Programming with C#

Jim Fawcett

CSE681 – SW Modeling & Analysis

Fall 2014

Overview

- Terminology
- Managed Code
- Taking out the Garbage
- Interfaces

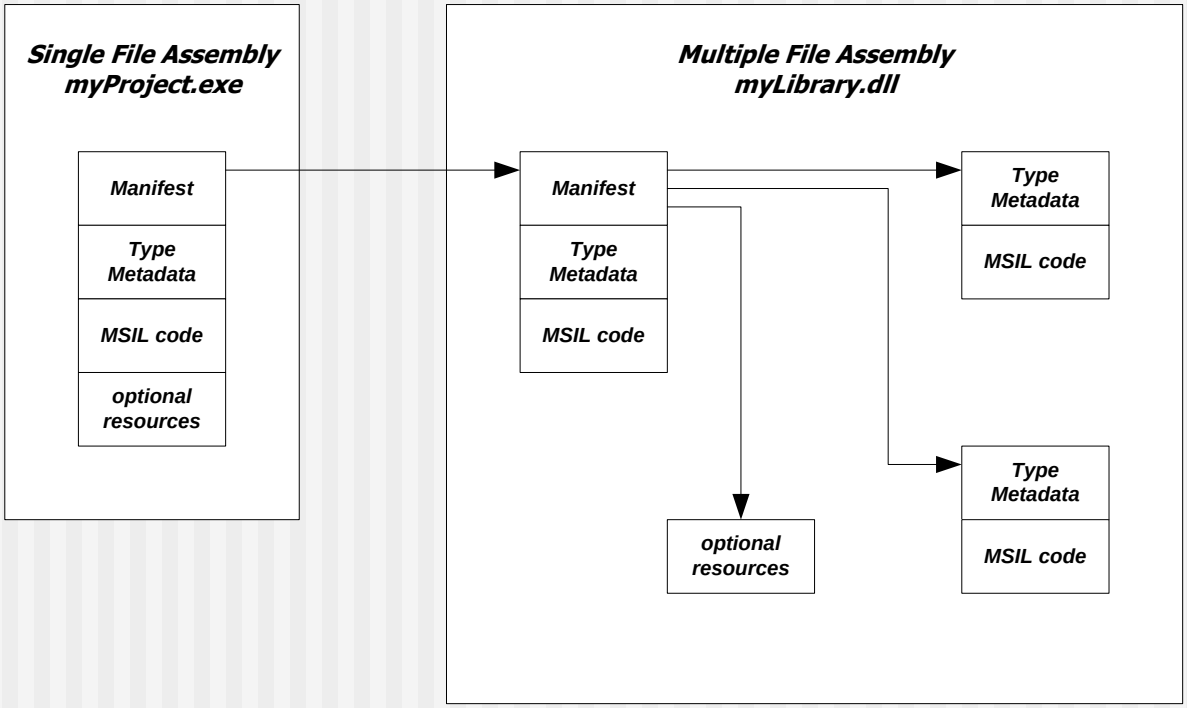
Terminology

- CLI: Common Language Infrastructure
  - CTS: Common Type System, the .Net types
  - Metadata: type information in assembly
  - VES: Virtual Execution System - provided by CLR
  - IL: Intermediate Language
  - CLS: Common Language Specification.
    - Core language constructs supported by all .Net languages.
- CLR is Microsoft's implementation of CLI.

Managed Code

- CLR provides services to managed code:
  - Garbage collection
  - Exception handling
  - Type discovery through metadata
  - Application domains and contexts
    - Fault isolation
    - Interception
      - Security management
      - Attributes

.Net Assembly Structures



Taking out the Garbage

- All .Net languages, including C# use garbage collection
- Garbage collection is a multi-tiered, non-deterministic background process
- You can’t deallocate resources immediately when objects go out of scope.

More about Garbage

- C# provides destructors which implement Finalize() for disposing of unmanaged resources.
  - Destructors allow you to tell the garbage collector how to release unmanaged resources.
- You should Implement IDisposable::Dispose()
  - Users of your class call it's Dispose() to support early release of unmanaged resources
  - Your dispose should call Dispose() on any disposable managed objects aggregated by your class and unregister event handlers.
  - Your member functions should call Dispose() on any local disposable managed objects.

Implementing Dispose()

- Here's the standard way:

```
public void Dispose()
{
  Dispose(true);  // garbage collector calls Dispose(false)
  GC.SuppressFinalize(this);
}
private void Dispose(bool disposing)
{
  if(!this.disposed)
  {
    if(disposing)
    {
      // call Dispose() on managed resources.
    }
    // clean up unmanaged resources here.
  }
  disposed = true;  // only call once
}
```

Minimizing Garbage

- If you have local managed objects in frequently called methods, consider making them members of your class instead.
- Using member variable initializers is convenient:
  class X
  { private: arrayList col = new ArrayList();
  …
  }
  but don't if col may be reinitialized to something else in a constructor. That immediately generates garbage.

Try - Finally

- Managed classes that use unmanaged resources:
  handles, database locks, …
  Implement Dispose() and Finalize() to provide for early, and ensure eventual, release of these resources.
- But Dispose() may not be called if the using code throws an exception. To avoid that, catch the exception and use a finally clause:

```
try { /* code using disposable x */ }
catch { /* do stuff to process exception */}
finally { x.Dispose(); }
```

The using short-cut

- C# provides a short cut for try-finally:

```
using(x) { /* use x object */ }
```

  is equivalent to:

```
try { /* use x object */}
finally { x.Dispose(); }
```

- You can't have multiple objects in the using declaration. You will need to nest the using statements to handle that case. It's probably easier just to use try-finally if you need to dispose multiple objects.

Interfaces

- Abstract class provides the root of a class hierarchy.
- Interface provides a contract:
  it describes some small functionality that can be implemented by a class.
- Interfaces can declare all the usual types:
  - Methods, properties, indexers, events.
- Interfaces can not declare:
  - Constants, fields, operators, instance constructors, destructors, or types.
  - Static members of any kind.
- Any type that implements an interface must supply all its members.

Using Interfaces

- Functions that accept and/or return interfaces can accept or return any instance of a class that implements the interface.
- These functions bind to a contract, not to a specific class hierarchy.

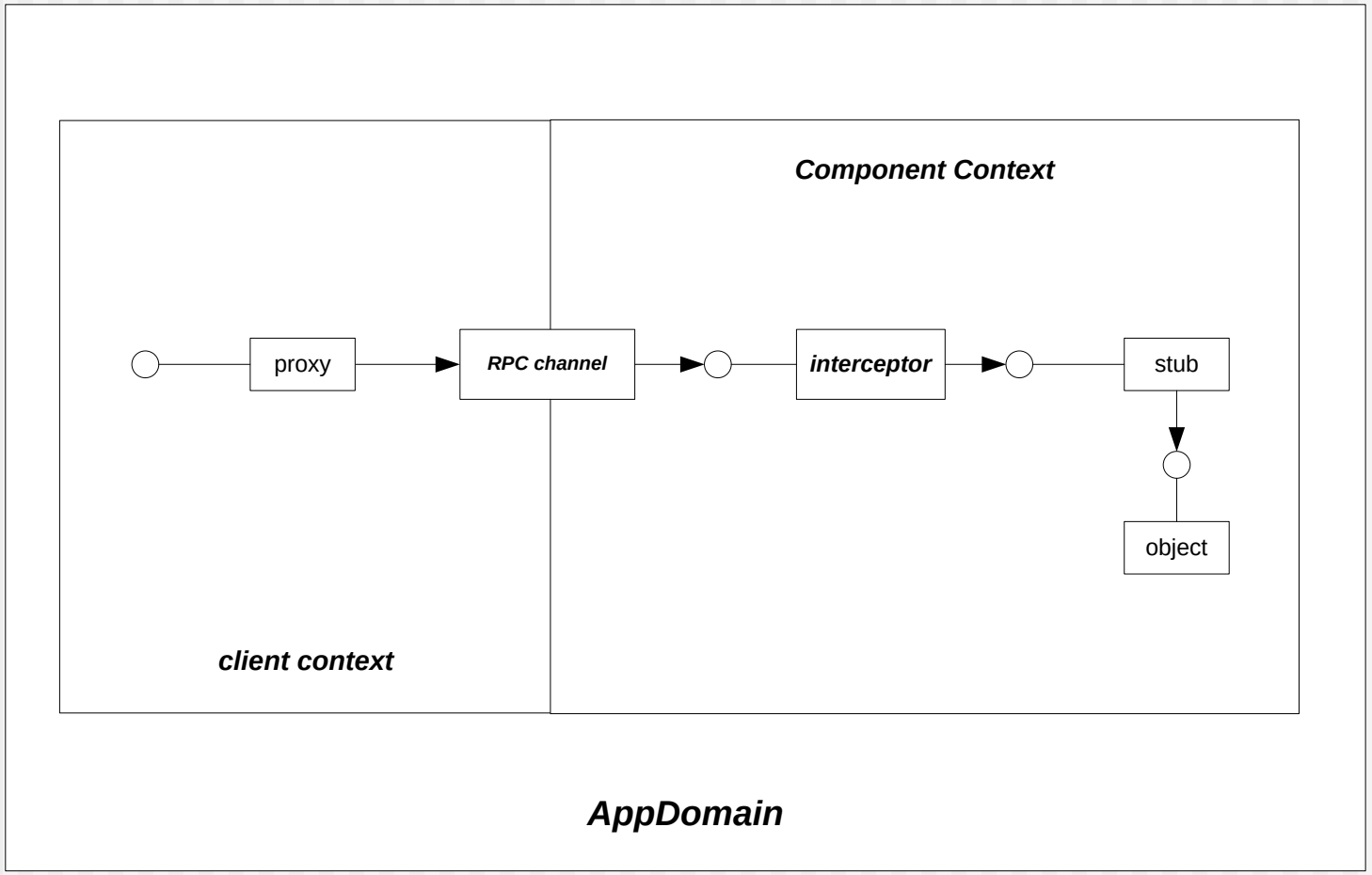
Implementing Interfaces

- .Net languages support only single inheritance of implementation, but multiple inheritance of interfaces.
- Members declared in an interface are not virtual.
  - Derived classes cannot override an interface method implemented in a base class unless the base declares the method virtual.
  - They can reimplement it by qualifying the method signature with new.
  - This hides the base's method, which is still accessible to a client by casting to the interface.
  - Hiding is generally not a good idea.

Overrides vs. Event Handlers

- Prefer overriding an event handler over subscribing to an event delegate.
  - If an exception is thrown in an event handler method the event delegate will not continue processing any other subscribers.
  - Using the override is more efficient.
  - There are fewer pieces of code to maintain.
  - But make sure you call the base handler.
- When do you subscribe to an event?
  - When your base does not supply a handler.

Interception



The End for now