Software Reuse

November 6, 2007

Chuck Allison

Utah Valley University

About This Webinar

- All About Reuse of Software Assets:
- Code
- Design
- Architecture
- Requirements (MKS)

Someone has already solved your code issues…

CODE REUSE

The Best Code is that which you don't have to write.

Software Libraries

- C, C++, Java, .NET, Python, etc.
- You use these every day
- What makes them so useful (reusable)?

Attributes of Reusable Code

- Solves a common problem
- Well-defined Interface
  - High cohesion
  - Right level of “granularity”
    - function, class, component, framework
- Generic in nature
- Configurable
  - To apply in your context

C Example (1980s)

- The **qsort** function
- Performs the Quicksort algorithm on any type of array
- `qsort(a, n, sizeof(int), cmp);`

C++ Examples (1990s)

- `sort(a, a+n);`
- `sort(a, a+n, greater<T>());`
- `sort(v.begin(), v.end());`
- `find_if(a, a+n, pred);`
-
```
transform(a, a+n, b,
          bind2nd(plus<int>(), 1));
```
-
```
accumulate(istream_iterator<int>(f),
           istream_iterator<int>(), 1,
           multiplies<int>());
```

Commonly Used Java Classes

- Collection classes
- Thread-related classes
- GUI Classes (Swing)
- Network-related classes
- Too many to mention in Core API
  - 3,000+ in 200+ packages

Frameworks

- Object-Oriented Reuse
  - Related classes that serve some application area
- Frameworks provide much of the needed functionality for a particular domain
  - And they typically have a *large footprint*
  - You provide the missing detail
- Examples:
  - GUIs
  - Persistence
  - Security

wxPython Example

```
# Application Object
class MyApp(wx.App):
    def OnInit(self):
        MyFrame("Chuck Allison - Project#1").Show()
        return True

# Top-level Window
class MyFrame(wx.Frame):
    def __init__(self, title):
        wx.Frame.__init__(self,None,-1,title)
        # Create/bind needed GUI objects…

    # Event Handlers
    def OnExit(self, event):
        self.Close()

# Launch Application
if __name__ == "__main__":
    MyApp(False).MainLoop()
```

Reusable Software Components

- Usually refer to self-contained software assets available *remotely* in *binary form*
  - Different processes, different platforms
- Examples: COM, J2EE
- Features:
  - *Complete* separation of interface and implementation
    - implementation exists elsewhere!
  - A broker or registry finds components
  - Proxy objects (stubs, skeletons, etc.) bridge platform barriers (data marshalling)

Python COM Example

```
from win32com.client import Dispatch

# Open and save a backup copy of c:\PDA2CFG.doc
wapp = Dispatch("Word.Application")
wapp.Documents.Open("/PDA2CFG.doc")
wapp.ActiveDocument.SaveAs("/PDA2CFG-Bak.DOC")
wapp.ActiveDocument.Close()
```

Configurability

- Using software components as “black boxes” doesn’t always solve user needs
  - They need to be able to tweak things
  - But this requires a deeper understanding of how the software works
- So, effective reuse can come with noticeable *learning curve*
  - You must decide if it’s worth it

Alexandrescu's Singleton

- Has 4 template parameters:
  - The class to "singleton-ize"
  - Storage Policy
  - Lifetime Policy
  - Threading Model
- **Singleton<MyClass,CreateStatic,NoDestroy> x;**
  - This instance defaults to **SingleThreaded**

Reusable Source Code

- Most of our examples so far have reused *runnable* code
  - Pre-compiled libraries
  - Distributed binary objects
- You can also reuse *source code*
  - But beware…

Source Code Search Engines

- Googling for “source code search engine”
  - code.google.com, koders.com, krugle.com,…
- Remember it is *code*:
  - Bugs included!
    - tends to be less reliable than released binaries
  - You need to understand it
  - Who is going to maintain it?
- A good *learning tool*

Designing for Reuse

- **Commonality/Variability Analysis** (CVA)
  - A key to design flexibility
- "Separate things that vary from things that stay the same" in a given context
  - Interfaces "stay the same"
  - Implementation details "vary"
- Promotes high cohesion, low coupling, genericity
- Such well-designed software assets tend to be reusable

Someone has already solved your design issues…

DESIGN REUSE

How Do You Reuse a Design?

- Many problems have similar abstractions
  - Design experience is too valuable a thing to waste
- What drives a design?
  - Forces that follow requirements
  - Forces resulting from architecture decisions
- How can design decisions be shared so they can be employed under similar conditions in the future?

Design Patterns

- Solutions to common recurring software design problems
- Based on *principles* gleaned from decades of experience, such as…
  - Don’t repeat yourself
  - Minimize coupling between things that interact
  - Abstractions should *not* depend on (or “see”) details; rather, details should conform to *abstractions*

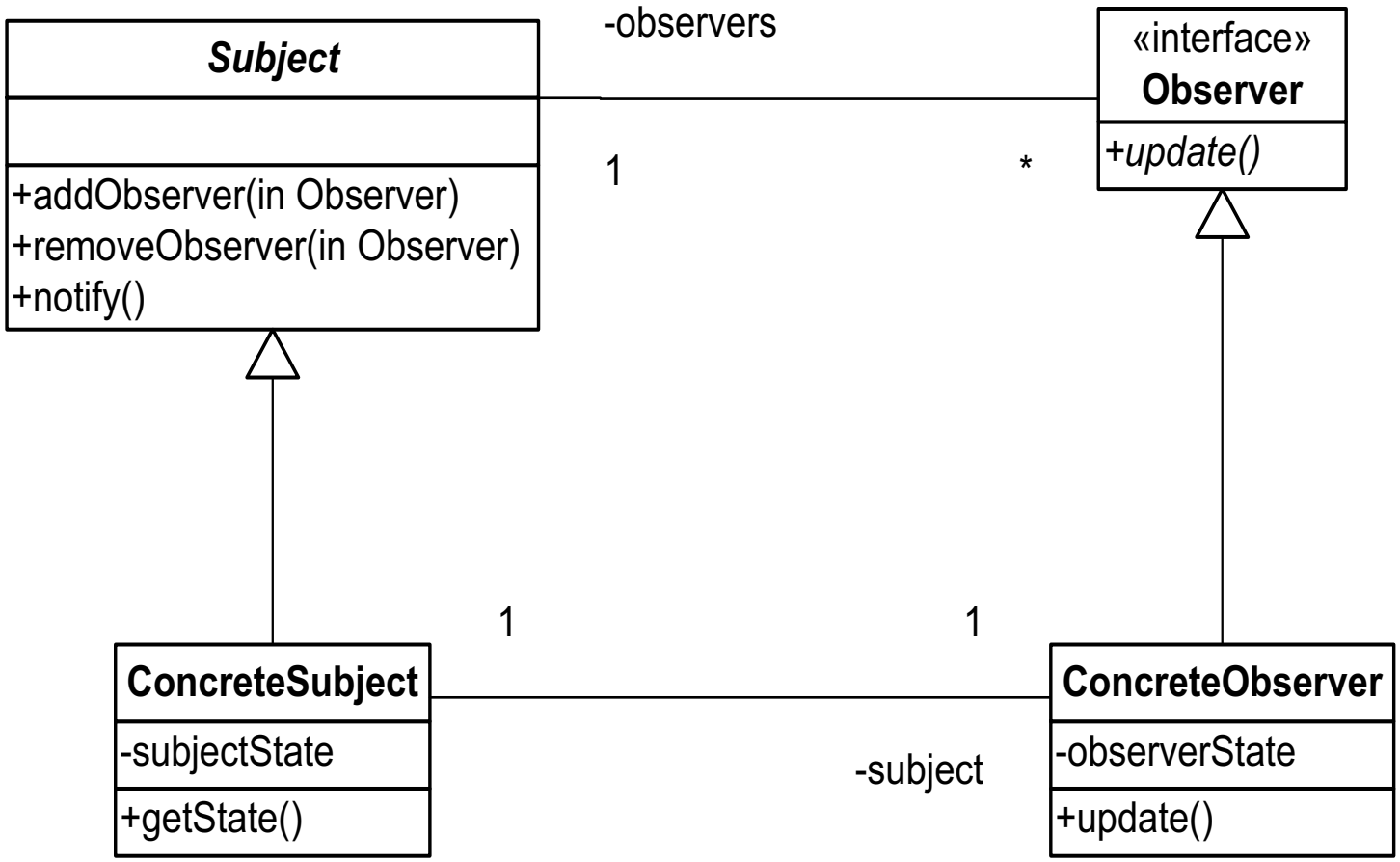
Sample Design Dilemma

- Clients generally obtain information from objects of interest (the “subject”) by calling the subject’s methods
- But how can the client keep current if the subject keeps changing?
  - How do clients know *when* to query the subject?

The Observer Pattern

- Allows an object to be tracked (observed) by an arbitrary number of observers with minimal coupling
- AKA “Publisher-Subscriber”
- Think of RSS feeds:
  - You subscribe to get notifications
  - You can unsubscribe at any time

The Observer Pattern (GoF)



You Don't Really "Reuse" Design

- Not literally, anyway
- You learn from the experience of others
  - Those who created the pattern
- You use that knowledge (of principles, especially) to complete your own designs
  - Your design may vary a little

Patterns In-the-Large

- Patterns aren't confined to design issues
- Many types of patterns exist
  - Because many types of problems exist!
- Patterns describe high-level, generic solutions that can be applied in many contexts
  - They can adapt to many contexts
  - They can lead to various implementations

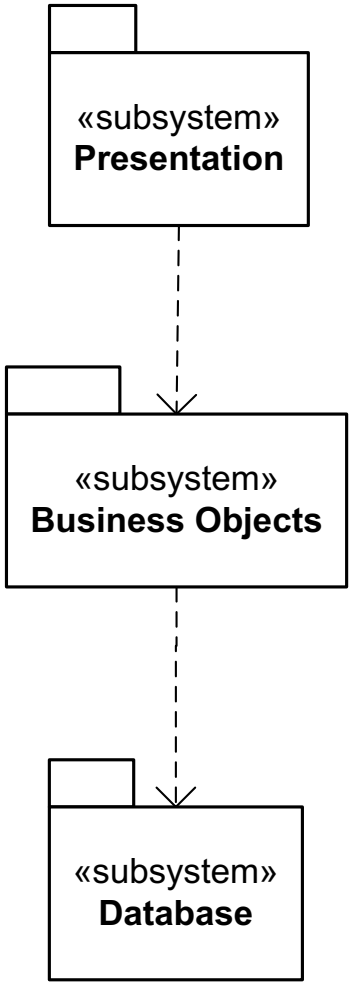
Architecture Patterns

- Resolve very high-level, “in-the-large” issues of software projects
- Describes fundamental structure of a system:
  - How does the data flow?
  - How do components communicate?
  - Where are components deployed?

Common Architecture Patterns

- Layers (aka “n-tier”)
- Pipeline (aka “Pipes and Filters”)
- Blackboard
- Peer-to-peer
- Broker

Layered Architecture Example

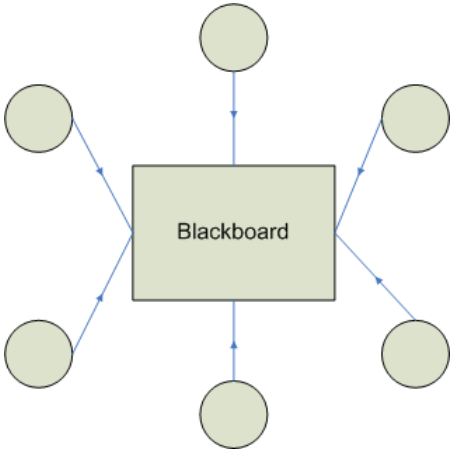


Pipeline Architecture

- Like UNIX “pipes and filters”
- When processes/components work in sequence
- The output of one step becomes input into the next

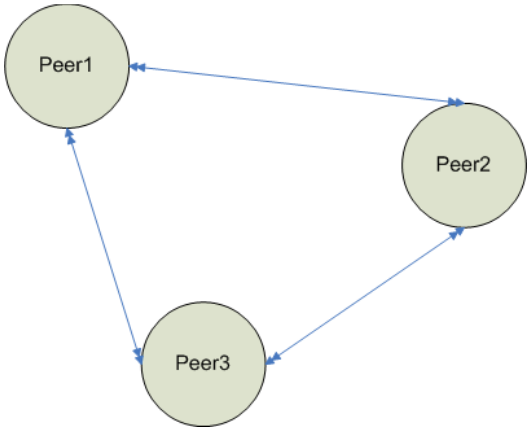
Blackboard Architecture

- For very complex problems that defy simple organization
- The “blackboard” is a shared repository of information
- Components update the repository throughout execution
- Some monitoring facility is needed to coordinate the shaed use of the blackboard



Peer-to-Peer

- A de-centralized network of cooperating components
- “Message passing” protocol
- Example: **e-mail**



Broker

- A way of managing *distributed* applications
- The Broker is a *mediator* for components that want to interact on demand
- Examples: DCOM, J2EE, SOA

Conclusion

- Many of your issues have been resolved before at some level or another:
  - Implementation, design, system architecture
- Reuse requires:
  - Being informed
  - Climbing a learning curve
  - Sharing your solutions