Design Patterns...
...Beyond the Gang of Four

Dennis Mancl

dmancl@acm.org
What are Patterns?

- Reusable design ideas
  - Good software design approaches
  - “Discovered” rather than “invented”
  - Solution to a problem in a context

- How many patterns?
  - General software design (GoF)
  - Communications software
  - Reliability
  - Analysis
  - Agile development process

- Why do we use them? Profit from the experience of others...

“Gang of Four” = popular 1994 book by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
How do patterns work?

Step 1: You have a design problem...
• “I have three applications need to display changing data”

Step 2: Look for a pattern that matches your problem and context
• “I think the **Observer** pattern might help”

Step 3: Create the design based on the pattern example
• “I am going to define two classes: Subject and Observer”
  • Subject holds the data, Observer updates the display
• “I am going to define attach() and notify() functions in the new classes”
  • Subject class has a linked list of pointers to Observers
  • Observer objects can call xSubject.attach(myself) to register for notification
  • Whenever a Subject changes state, it calls xObserver.notify() on each registered Observer

Step 4: Include information about the pattern in the design documentation
What could go wrong?

• Patterns are not a simple cookie cutter...
  • You need to consider the context
  • Each pattern has “Consequences” (for example, Observer pattern could cause a slow and inefficient cascade of updates)

When you use your pattern, it might trigger the need for one or more related patterns:
  • A “pattern language” is a group of connected patterns
  • We will talk about a few pattern languages for specialized contexts

It’s easy to go “pattern happy”
  • (making the application extra complicated just so we can show off how many patterns we can use)
Reliability patterns

• How to make a complex system more reliable
  • Replication
  • Check data inputs
  • Monitor critical processes
  • Overload control policies
  • Recover/restart failed elements

• Several good sources of reliability patterns
  • *Patterns for Fault-Tolerant Software* by Robert Hanmer
  • *Robust Communications Software* by Greg Utas
Reliability patterns

• A complex system needs to use a group of patterns
  • Error detection, error recovery, error mitigation

• A few “error detection” patterns
  • **Routine Audit** – the system is designed to run periodic checks on its internal data
    • If errors are found, the system might use a “correcting audit” to repair the data
  • **Watchdog** – there is special hardware or software that watches a key element of the system
    • Monitor one key task to make sure it is alive and working correctly – trigger a restart if fails
  • **System Monitor** – more elaborate than a Watchdog, monitor the behavior of multiple system elements
    • Trigger repair or recovery when there is a problem
Quantity pattern

- Analysis patterns are a set of patterns that are used in doing the initial problem analysis:
  - They help answer the question “what objects should I define in my system?”
- The **Quantity** pattern is from the book *Analysis Patterns* by Martin Fowler
  - Recording measurements and manipulating results might be error-prone
  - Each value really should be recorded with its units:

A Money object will have both a number and an identifier to say which currency:

Length and weight also need units:
- [100, “miles”]; [15.5, “kg”]
Justification for the Quantity pattern

• A frequent problem – someone tries to perform an invalid operation on two different types of quantities:
  • adding apples to oranges, people to money, dates to time intervals
  • conversion mistakes: adding dollars to euros, inches to feet
  • performing an average of a mixed bag of objects (this should never be legal)

• Using explicit units in the design makes it easier for someone else to understand the software later
  • what does this number mean??
Reengineering patterns

• Some patterns go beyond the initial design
  • The book Object-oriented Reengineering Patterns has some valuable “redesign” ideas
  • Redesign = changing an existing software system to meet new needs
  • And... the original developers might not be available

• The Facade pattern is really useful (and it is a GoF pattern)
  • Build wrappers around existing modules
  • Analyze the data to decide what to wrap
  • Benefit: Reduces coupling
  • Benefit: Helps support evolution – some modules can be updated without affecting others
Reengineering patterns

• An extremely useful reengineering pattern: **Write Tests to Enable Evolution**
  • Analyze key system scenarios – create some automated tests that exercise parts of the scenarios
  • Use automated test frameworks, to make it easy to run the tests frequently

• The tests can support refactoring
  • When you make minor changes to algorithms or data structures, it is easier to test if anything was broken
  • Tests have an impact on overall system quality

• Focus on parts of the system that are changing rapidly
  • Add new tests in each product release
Automated testing – costs and benefits

- It can take a lot of effort to add new tests to software
  - Don’t do it all at once – focus on key scenarios

- Tests are sometimes difficult to maintain
  - Avoid writing automated tests of the user interface details – it is better to test the internal functionality before the UI interactions

- Benefits:
  - Increased confidence in the system as a whole
  - Less risk when modules are turned over to new staff
  - It is easier to make small changes with confidence
  - Tests are a good form of “documentation” – much more precise than text descriptions
Communications patterns

• Most interesting software applications are not “isolated”
  • Applications designed to interact with other applications
  • Or use a database on a central server
  • Or relay events to a central controller

• Applications that are split
  • between a small device (cell phone, smart appliance)
  • and a larger network-based system

• Concurrency – take advantage of multi-core systems
  • Use “threads” for independent operations
  • But some synchronization is still needed
Communications patterns

• Patterns for processing “events” in a complex system
  • **Reactor, Proactor** – two different approaches for reacting to events from multiple processes

• Patterns for communication – distributed, concurrent, multi-threaded
  • **Monitor, Active Object** – two different approaches for setting up communicating services

• A good place to start is the book *Pattern-Oriented Software Architecture, Vol. 2*
Active Object pattern

• Problem: how to build small collaborating modules
• Context: distributed or multi-threaded application; modular structure is needed to support frequent changes to the application
• Solution: make each module an Active Object
  • Each Active Object has a “message queue” – where it receives service requests
  • The implementation of the Active Object is an infinite loop: processing requests from other parts of the system

It is easy to do this in multiple programming languages:
  • In Java or Python, build on the Thread class
  • In C++, use C++11 threads, Boost library, or the ACE framework
  • Commercial and open source frameworks (QP, Theron, Orbit, libagents)
Active Object example

- Word frequency counter in Python (based on an example by Crista Lopes)

$ python ./wfcounter.py inputfile.txt
mostly - 2
live - 2
in - 2
africa - 1
tigers - 1
india - 1
lions - 1
wild - 1
white - 1

We could write a “monolithic program” to do the counting,
But let’s try doing it with a multi-threaded application!
Active Object example

• Create abstract base class for Active Objects in our application – inherits from Python Thread class:

```python
class ActiveWFObject(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.name = str(type(self))
        self.queue = Queue()
        self._stop = False
        self.start()

    def run(self):
        while not self._stop:
            message = self.queue.get()
            self._dispatch(message)
            if message[0] == 'die':
                self._stop = True
```

• Inherits from Python Thread class
• Must implement a “run” function (infinite loop reading commands from a queue)
Active Object – Word Frequency Manager

• **WordFrequencyManager** – keeps a Python dictionary with “words” and “counts”

• Other objects will send it some words:

```python
class WordFrequencyManager(ActiveWFObject):
    """Keeps the word frequency data """
    _word_freqs = {}

    def _dispatch(self, message):
        if message[0] == 'word':
            self._increment_count(message[1:])

    def _increment_count(self, self, message):
        word = message[0]
        if word in self._word_freqs:
            self._word_freqs[word] += 1
        else:
            self._word_freqs[word] = 1
```

A typical message might contain: ['word', 'tigers']
Active Object – Data Storage Manager

• DataStorageManager – read in words from a file, send one word at a time to the WordFrequencyManager

• First step: read in the entire file, eliminate extra white space and punctuation, convert to lower case

```python
class DataStorageManager(ActiveWFObject):
    _data = ''

    def _dispatch(self, message):
        if message[0] == 'init':
            self._init(message[1:])

    def _init(self, message):
        path_to_file = message[0]
        self._word_freqs_manager = message[1]
        with open(path_to_file) as f:
            self._data = f.read()
        pattern = re.compile(r'[\W_]+')
        self._data = pattern.sub(' ', self._data).lower()
```

If the file was:

White tigers live mostly in India.

the new self._data string will be:

white tigers live mostly in india
Active Object – Data Storage Manager

- DataStorageManager – process all of the words in the file

class DataStorageManager(ActiveWFObject):
    _data = ""

    def _dispatch(self, message):
        if message[0] == 'init':
            self._init(message[1:])
        elif message[0] == 'send_word_freqs':
            self._process_words(message[1:])

    def _process_words(self, self, message):
        data_str = "".join(self._data)
        words = data_str.split()
        for w in words:
            send(self._word_freqs_manager, ['word', w])
        send(self._word_freqs_manager, ['top25', message[1]])

The send function will add a request to the queue for the WordFrequencyManager Active Object...
Active Object – Word Frequency Controller

- WordFrequencyController – starts the counting, reports results

class WordFrequencyController(ActiveWFObj):

    def _dispatch(self, message):
        if message[0] == 'run':
            self._run(message[1:]):

    def _run(self, message):
        self._storage_manager = message[0]
        send(self._storage_manager, ['send_word_freqs', self])

Not done yet... still need to report the frequency counts...
Active Object – Word Frequency Controller

- WordFrequencyController – starts the counting, reports results

class DataStorageManager(ActiveWFObj):
    def _process_words(self, message):
        data_str = ".join(self._data)
        words = data_str.split()
        for w in words:
            send(self._word_freqs_manager, ['word', w])
        send(self._word_freqs_manager, ['top25', message[1]])
Active Object – report results

- Add a new “top25” message to WordFrequencyManager – create a sorted list of word counts, send to the controller

```python
class WordFrequencyManager(ActiveWFOBJECT):
    """ Keeps the word frequency data """
    _word_freqs = {}

    def _dispatch(self, message):
        if message[0] == 'word':
            self._increment_count(message[1:]):
        elif message[0] == 'top25':
            self._top25(message[1:]):

    def _top25(self, message):
        recipient = message[0]
        freqs_sorted = sorted(self._word_freqs.iteritems(),
                               key=operator.itemgetter(1), reverse=True)
        send(recipient, ['top25', freqs_sorted])
```

Active Object – report results

- Add a new “top25” message to WordFrequencyController – display the word counts

```python
class WordFrequencyController(ActiveWFObject):
    def _dispatch(self, message):
        if message[0] == 'run':
            self._run(message[1:])
        elif message[0] == 'top25':
            self._display(message[1:]),

    def _display(self, message):
        word_freqs = message[0]
        for (w, f) in word_freqs[0:25]:
            print w, ' - ', f
        send(self._storage_manager, ['die'])
        self._stop = True
```

Report the data that was sent by the WordFrequencyManager...
Is this a good pattern?

• Is this a good way to implement this program?
  • Maybe – it is very modular, and we can add new modules to augment the functionality
  • For example: to filter out “common words”, we can add a new Active Object called StopWordsFilter – between the DataStoreManager and the WordFrequencyManager

```
WordFrequency Controller  send_word_freqs

DataStorage Manager  filter

StopWords Filter  word

WordFrequency Manager
```

Stop words will be a list of simple words that we don’t want to count: “a, the, and, but, if, …”
Is this a good pattern?

- The pattern is even more useful for simple control and communications applications:
  - Active Objects to monitor the state of real-world objects
  - Active Objects to “wrap” some of the services available in a large client-server application

A distributed application that requires information from multiple GPS devices, server objects, and some application objects
Useful links related to Active Object

• The Word Frequency Counter example is based on a section of the book *Exercises in Programming Styles* by Cristina Lopes
  • github.com/crista/exercises-in-programming-style/tree/master/28-actors

• Useful notes on implementing Active Objects:
  • pragprog.com/magazines/2013-05/java-active-objects
  • www.codeproject.com/articles/991641/revisiting-the-active-object-pattern-with-cplusplu
  • www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095

• There are other approaches to building multi-threaded systems:
  • Active Object is a “thread per object” approach
  • In some server-based applications, “thread per request” can be better – especially for services that have a long execution time
  • More complex: several concurrent operations might be changing the state of a single object – the design of the request code might need to use semaphores to control access to critical sections
Books and articles

- Martin Fowler, *Analysis Patterns* (Addison-Wesley, 1996)
- Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz, *Object Oriented Reengineering Patterns* (Morgan-Kaufmann, 2003)
  - http://www.iam.unibe.ch/~scg/OORP
What have we learned?

• Extend your range!
  • The GoF book is great, but...
  • More patterns for other contexts
  • We are writing more concurrent and distributed software
  • Reliability is increasingly important
  • And building on legacy software is always valuable

• Add to your design vocabulary...

This talk: