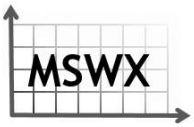
Design Patterns

Dennis Mancl

dmancl@acm.org



MSWX ◊ Mancl ◊
Software ◊ Experts ◊ http://manclswx.com

Models and software design

- In many fields, we use models to understand and think about problems and designs
  - Mechanical engineers, civil engineers, and architects have standard detailed drawings of individual design elements and entire designs
  - Electrical engineers work with circuit-level models – plus they draw diagrams of gates, registers, control lines, buses, and so on
  - In software, we use simple block diagrams to describe the structure and behavior of functions, modules, interfaces, and components
    - UML = Unified Modeling Language
- Can we ever "reuse" some design elements?
  - Common catalog of electronic components
  - Standard subroutine libraries
  - Design patterns – a higher level of design reuse

What is a pattern?

- A **pattern** is a solution to a problem in a context
- The problem and context come from some domain (software development, project management, …)
  - The problem and context contain some "unresolved forces" that need to be addressed.
- The solution is a proposed way to resolve the problem

***Note**: in the example, we are looking for a **solution** to a software design problem. The **context** is an existing partial design.*

Name: Information Expert

Problem: How to decide which class or object to assign responsibilities to?

Context: A responsibility usually requires some information or data for its fulfillment – information about other objects, an object's own state, the world around an object, and so on.

Solution: Assign a responsibility to the class that has the information needed to fulfill it.

(from *Applying UML and Patterns*, third edition, Craig Larman)

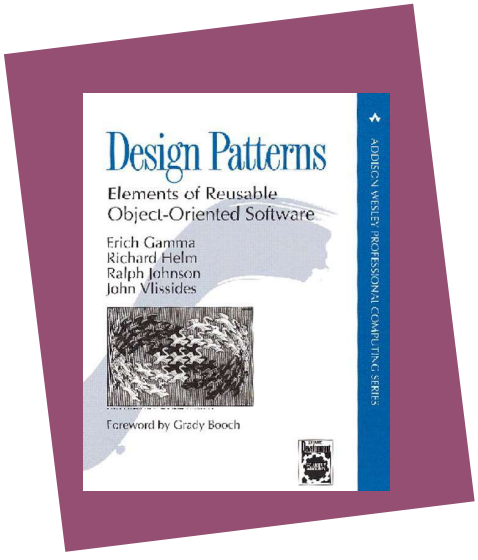
Question: *Where to put the "Print Shipping Label" operation?*

CustomerOrder
Order Id
Customer Id
Billing Address
Shipping Address
Items To Ship
Add New Item

Customer
Customer Id
Customer Name
Default Billing Address
My Shipping Addresses
Send Latest Catalog

What are Patterns?

- The first "Object Oriented Design Patterns" are found in the book
  - *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (known as "Gang of 4" or "GOF")
  - 23 patterns: three categories (Creational, Structural, Behavioral)
  - The patterns are for **software problems**
  - Examples in C++ and Smalltalk
  - The patterns also apply to other languages (Java, Visual Basic, Perl)



"Gang of Four" = popular 1994 book by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

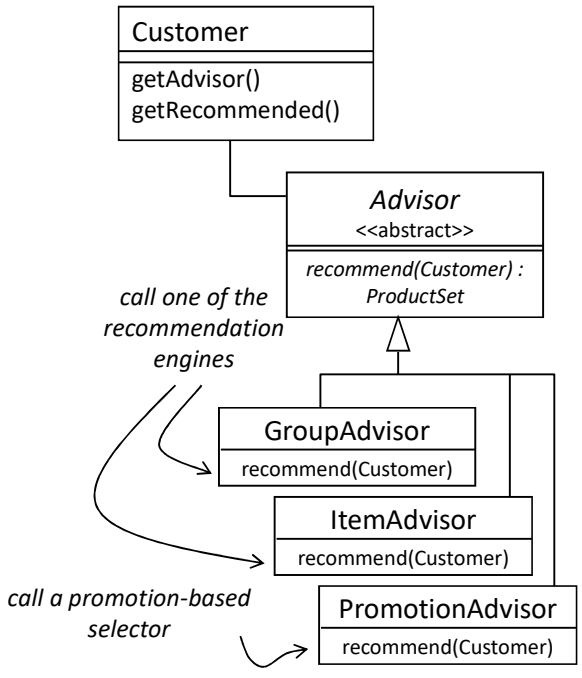
Strategy pattern

- Problem: An application needs to use a family of similar algorithms
  - the selection of which algorithm depends on the client making the request or some characteristics in the data
- Context: Two possible situations:
  - you have a group of classes that contain the same data, but they have different behavior (functions);
  - or you have a class that defines many behaviors, and these behaviors appear as multiple conditional statements in the operations of the class
- Solution: Define a family of algorithms, encapsulate each one, and make them interchangeable
  - there will be a family of classes, one per algorithm variation

Example of Strategy

- Internet-based business
  - When a customer accesses the home page, you want to present a few products that the customer might be interested in
  - Web presentation program uses one or more "recommendation engines"
    - commercial off the shelf software
    - uses customer survey information or previous purchase history to make a recommendation
  - Might use multiple engines
  - Might also want to highlight special sale items

- ***GroupAdvisor:*** *Use data from an "interest survey"*
- ***ItemAdvisor:*** *Use data from customer's previous purchases*
- ***PromotionAdvisor:*** *Seasonal recommendations*



Consequences of Strategy

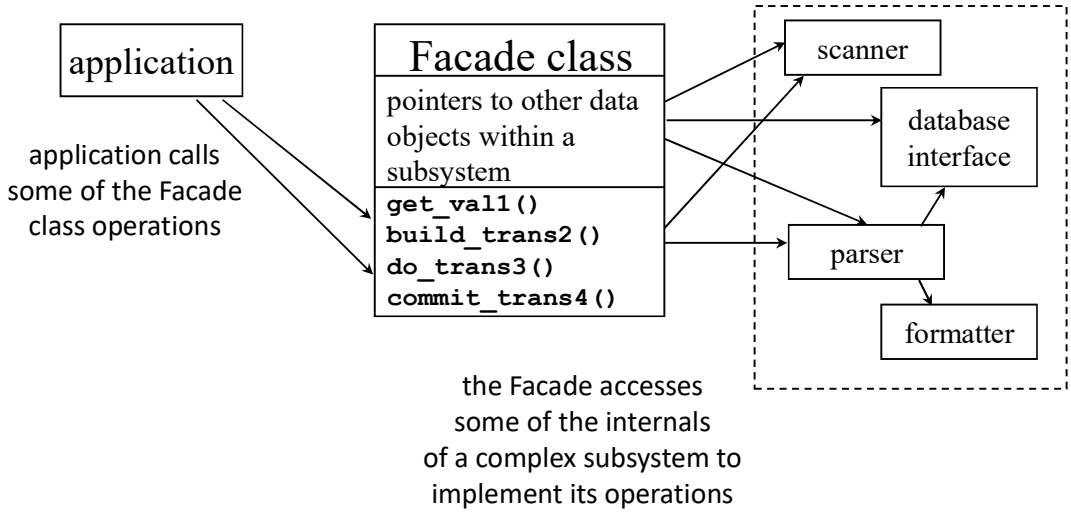
- Benefits
  - It is easy to add new variations of the algorithm
  - The modules that use the Strategy do not need to be aware of changes to the Strategy implementations
- Things to watch out for
  - There will be some communications overhead between the Context objects and the Strategy objects
  - Using the Strategy pattern increases the number of objects in the system

Facade pattern

- Problem: The application needs a simple interface to a complex subsystem
  - the subsystem might have been written by someone else
  - you don't want the entire development team to go back to the subsystem documentation with lots of questions
- Context:
  - it is important to control the dependencies between the application and the complex subsystem – you want to reduce the effort to maintain the system
- Solution: Define a single Facade class
  - the Facade class has knowledge of the internal details of the subsystem, but the Facade class provides a simple to use interface for the application
  - each Facade public function might call many subsystem operations

Facade diagram

- A Facade class wraps a bunch of operations on other classes (or a bunch of legacy code operations) into a convenient package

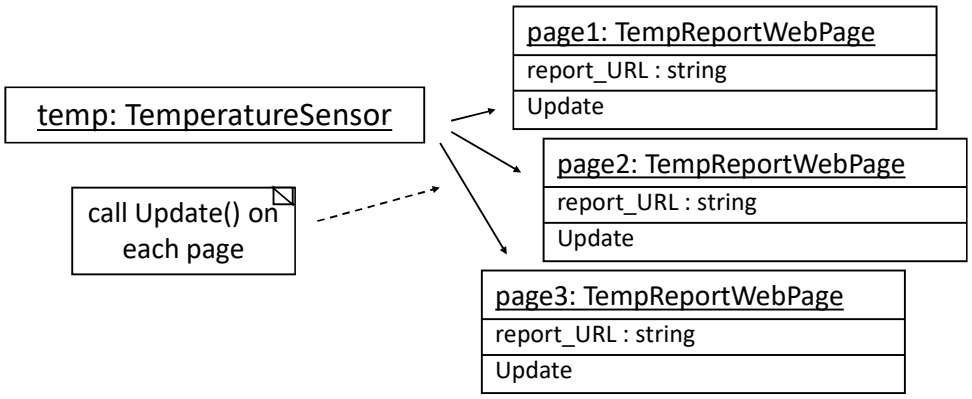


Planning a Facade

- Facade affects the design process – do some "modeling" to plan the Facade interfaces
  - Create a simplified model of the classes in the overall system
    - using CRC cards (informal technique using index cards, one card per class)
    - using UML Class Diagrams
  - Look for a group of classes that collaborate closely together: call them a "subsystem"
  - Try to "put the subsystem behind a wall" – define a single interface class that defines an API for the subsystem

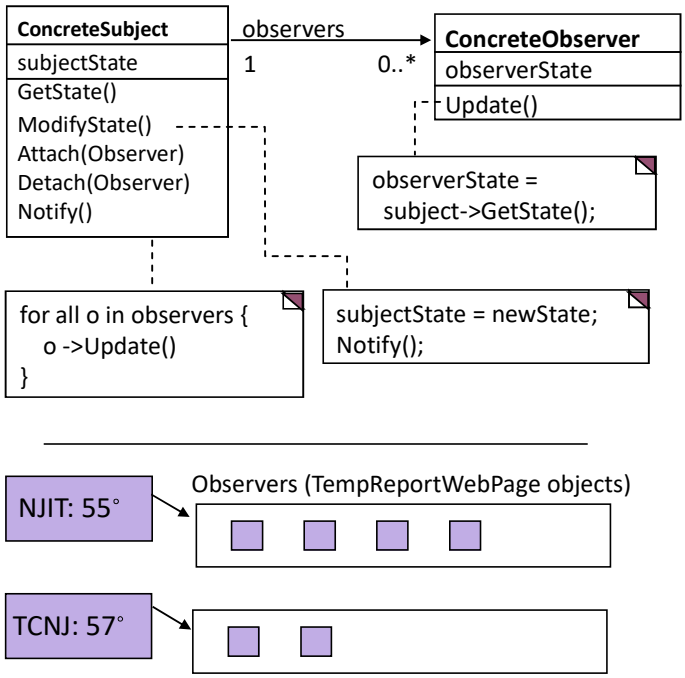
Observer – a useful design technique

- Observer pattern always has at least 2 classes: Subject and Observer
  - TemperatureSensor – maintains information about the current temperature
  - TempReportWebPage – displays temperature value on a web page
- Each temperature value might appear on multiple pages
- Update each page when the TemperatureSensor changes state



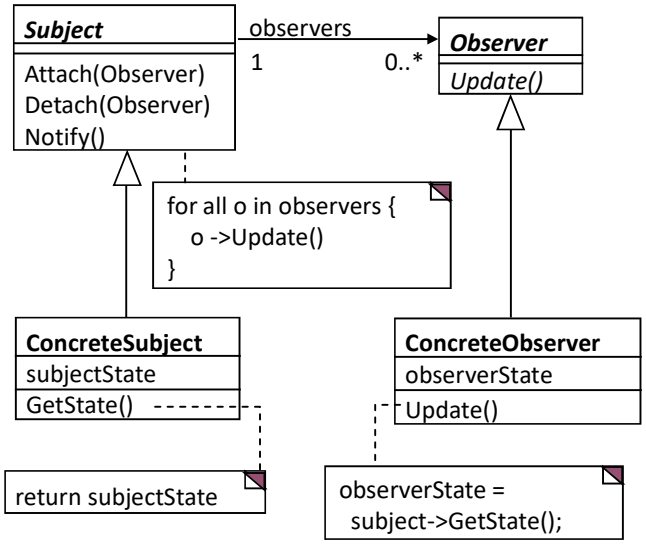
Simple class diagram for Observer

- In the Observer Pattern: there are links between two kinds of concrete classes
  - subjects and observers
- Each observer object (such as **TempReportWebPage)** needs to register by calling the Attach() operation on a subject object (like **TemperatureSensor**)
- Each observer object will have its Update() operation called whenever its subject changes state



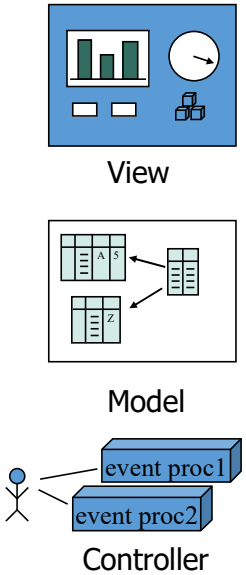
Complete class diagram for Observer

- **Observer** is an abstract interface that each **ConcreteObserver** must implement (must implement an Update() function)
- Observer objects still register by calling the Attach() operation on a **ConcreteSubject** object
- Each ConcreteObserver object will have its Update() operation called whenever its ConcreteSubject changes state



Motivation for Observer

- An early OO user interface architecture: Model-View-Controller
  - A **model**, is an object representing data (often data about an object in the real world)
  - A **view** is some form of visualization of the state of the model.
  - A **controller** offers facilities for the user to change the state of the model (clicking on buttons, sliding a slider, typing in a text box, etc.)
- Some elements within the View observe the Model
  - if the Model changes state, then the View may need to update its objects and pass the changes to the user interface implementation
- The Model observes the Controller
  - after the Controller accepts command input from the user, it needs to notify the Model to update its state

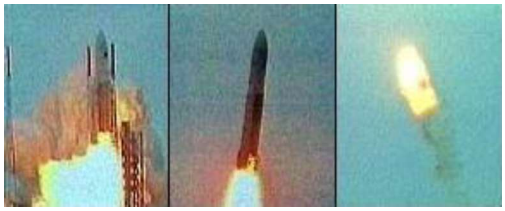


Consequences of Observer

- Benefits
  - The coupling between the Subject classes and the Observer classes is very minimal
  - Within the implementation of the Subject class, the calls to the Update function don't need to specify a "receiver" – the recipients are automatically managed within the pattern
- Things to watch out for
  - Performance: Observer objects are usually unaware of the existence of other Observer objects, so there can sometimes be a very long chain of updates caused by a modification to one Subject

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:

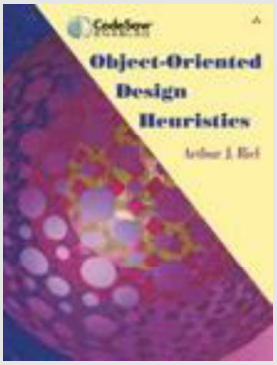
- A "pattern language" is a group of connected patterns
- There are some useful pattern languages for specialized contexts

It's easy to go "pattern happy"

- (making the application extra complicated just so we can show off how many patterns we can use)

Patterns versus Design Heuristics

- Another good way to "improve your designs" is to apply some standard design principles and rules:
  - Modularization – always divide a big design into small independent pieces
  - Encapsulation – each component has a simple interface, hiding complex details
  - Low coupling – minimize dependencies between components
  - Extensibility – components can be extended without changing the base code
- An excellent approach is to use Design Heuristics
  - Some "rules of thumb" – they help you to avoid some serious design problems



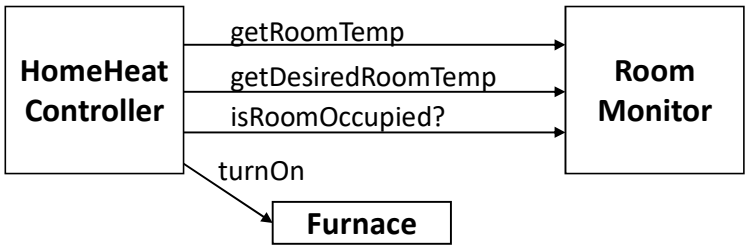
Object-Oriented Design Heuristics by Arthur Riel (Addison-Wesley, 1996)

Some key heuristics

- Information hiding
  - All data should be hidden within its class (no public data)
  - Don't clutter the public interface of a class (keep classes simple)
- Distribution of responsibilities
  - Distribute system intelligence horizontally as uniformly as possible; avoid "god classes"
- Use inheritance correctly
  - Inheritance should be used only to model a specialization hierarchy; don't use inheritance to model a "has-a" or "is-part-of" relationship
  - Do not use inheritance to model the dynamic semantics of a class (objects that change their "class" in different states)
  - Do not create a derived class that overrides a base class method with a NOP method, i.e. a method which does nothing

God class

- One class controls everything
  - Most of the classes in the system are dumb "data containers" – only one class contains all of the application logic
- It is a poor distribution of responsibilities
  - We prefer to have a number of top-level classes that share their work equally
- One common mistake – classes that have many "get" and "set" operations

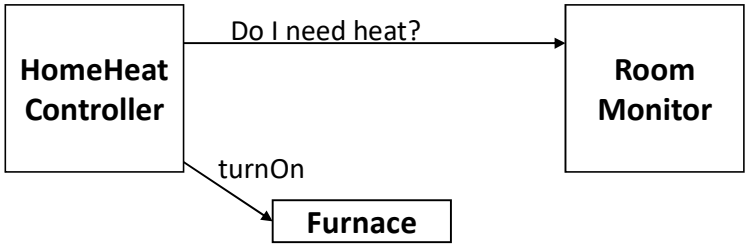


The **RoomMonitor** is just a dumb data-holder.

HomeHeatController is doing all of the work… it is a god class

Avoiding a god class

- What is HomeHeatController doing with the data it gets from RoomMonitor?
  - It is a controller that makes some policy decisions
  - Needs to decide whether to start the Furnace
- But RoomMonitor has most of the data needed to make the decision
- Maybe it's better to move the policy decision closer to the data



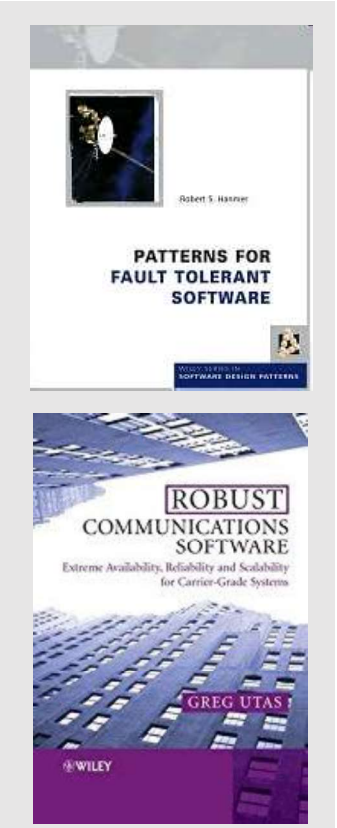
The **RoomMonitor** is now a smarter class.

HomeHeatController delegates the decision…

Design is now simpler: less coupling between classes

Reliability patterns

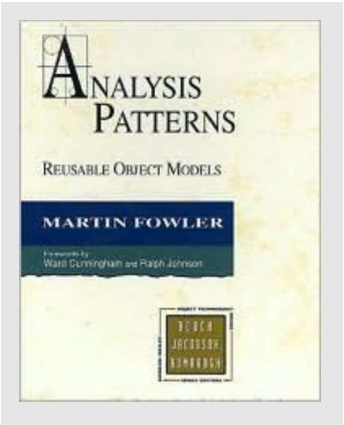
- How to make a complex system more reliable
  - Replication
  - Check data inputs
  - Monitor critical processes
  - Overload control policies
  - Recover/restart failed elements
- Several good sources of reliability patterns
  - *Patterns for Fault-Tolerant Software* by Robert Hanmer
  - *Robust Communications 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: [19.95, "US Dollars"]; [700, "Euros"]; [100, "Yuan"]

- Length and weight also need units: [100, "miles"]; [15.5, "kg"]

Justification for the Quantity pattern

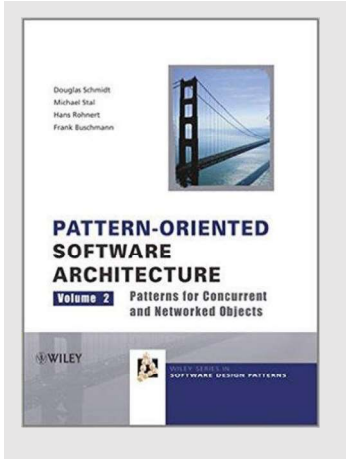
- A frequent problem – someone tries to perform an invalid operation on two different types of quantities:
  - adding apples to oranges, people to money, dates to time intervals
  - conversion mistakes: adding dollars to euros, inches to feet
  - performing an average of a mixed bag of objects (this 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??

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

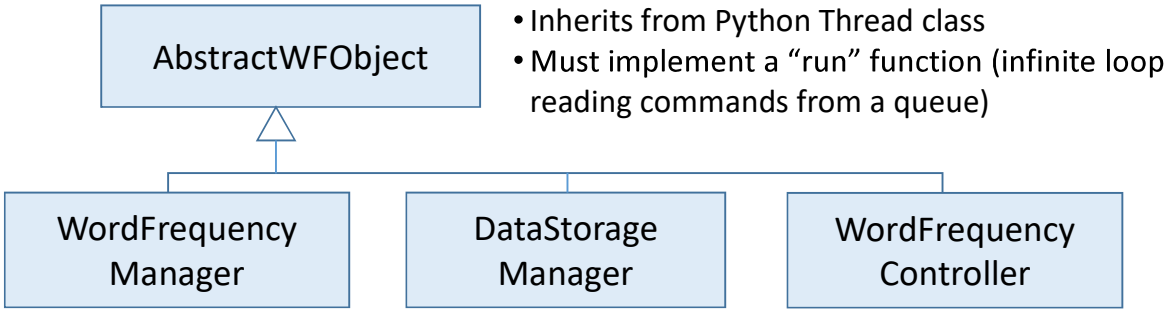
```
mostly - 2
live - 2
in - 2
africa - 1
tigers - 1
india - 1
lions - 1
wild - 1
white - 1
```

inputfile.txt
White tigers live mostly in India
Wild lions live mostly in Africa

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:



- Inherits from Python Thread class
- Must implement a "run" function (infinite loop reading commands from a queue)

```
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
```


Active Object – Word Frequency Manager

- WordFrequencyManager – keeps a Python dictionary with "words" and "counts"
- Other objects will send it some words:

```
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, 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

```
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('[\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

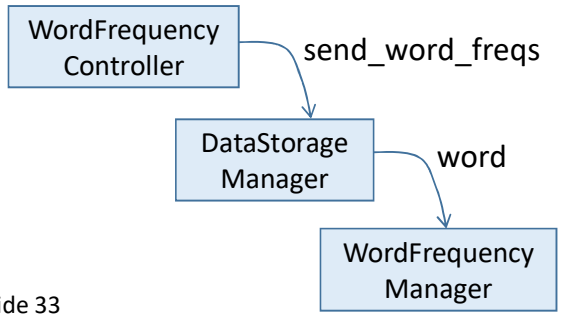
```
class DataStorageManager(ActiveWFObject):
    _data = ''

    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, 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(ActiveWFObject):

    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])
```



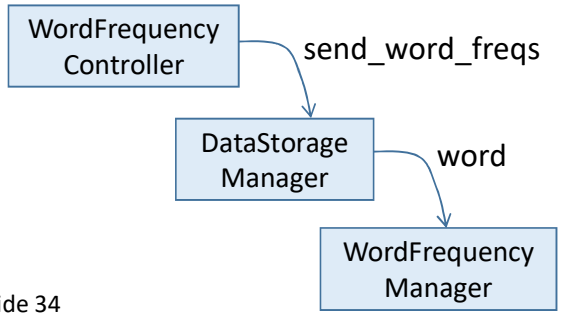
Not done yet… still need to report the frequency counts…

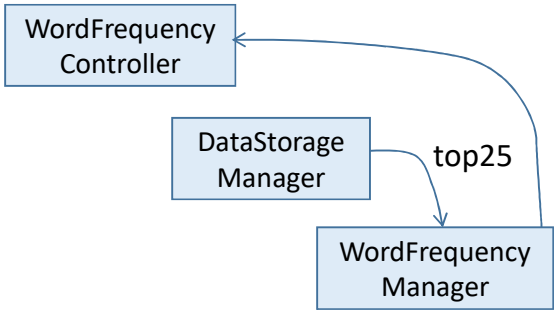
Active Object – Word Frequency Controller

- WordFrequencyController – starts the counting, reports results

```
class DataStorageManager(ActiveWFObject):
    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]])
```

Tell the WordFrequencyManager to sort and report

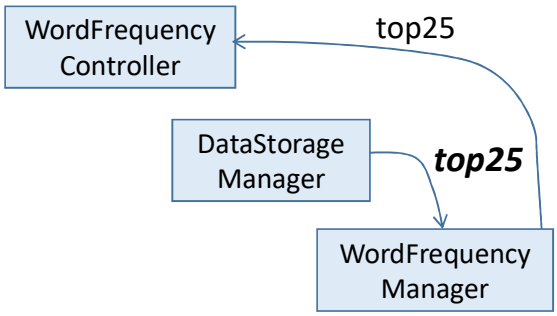




Active Object – report results

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

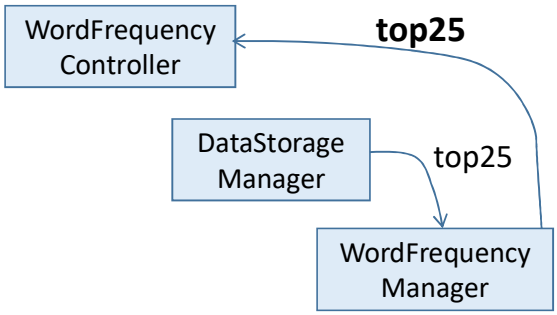
```
class WordFrequencyManager(ActiveWFObject):
    """ Keeps the word frequency data """
    _word_freqs = {}

    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

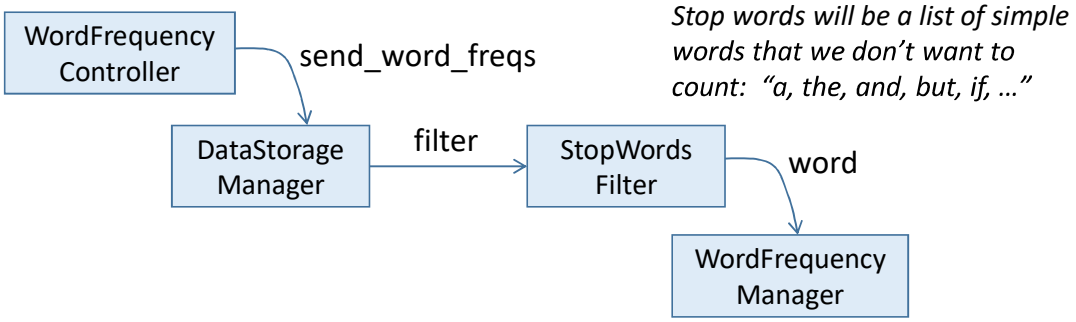
```
class WordFrequencyController(ActiveWFObject):
    def _dispatch(self, message):
        if message[0] == 'run':
            self._run(message[1:])
        elif message[0] == 'top25':
            self._display(message[1:])

    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?

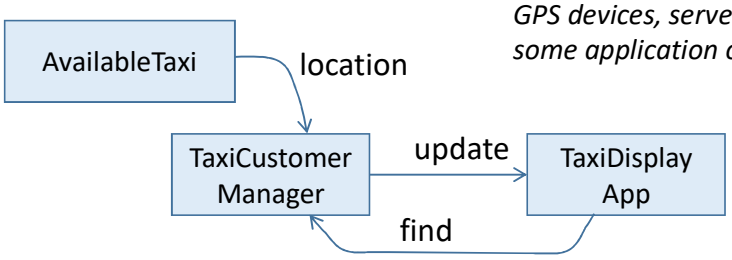
- Is this a good way to implement this program?
  - Maybe – it is very modular, and we can add new modules to augment the functionality
  - For example: to filter out "common words", we can add a new Active Object called StopWordsFilter – between the DataStoreManager and the WordFrequencyManager



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

Consequences of Active Object

- Active Object is a good for simple multi-threaded systems
  - We can use a "thread per object" model – assuming that the number of parallel objects is relatively small
- Avoids complex synchronization… you don't need to "lock" an object for each operation
  - All operations are sent to the object's request queue

✷ Problems: dealing with hundreds of objects, high volume of requests, real-time response
  - For example, this is not a good pattern to use for implementing a high-performance web server
  - Operations on "popular objects" will sequential – you won't get to use parallelism..
  - Consider other techniques instead: Thread-per-request, Thread Pool
  - www.metachris.com/2016/04/python-threadpool/

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

Books

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns* (Addison-Wesley, 1994)
- Eric Freeman and Elisabeth Robson, *Head First Design Patterns* (O'Reilly, 2005)
- Joshua Kerievsky, *Refactoring to Patterns* (Addison-Wesley, 2005)
- Martin Fowler, *Analysis Patterns* (Addison-Wesley, 1996)
- Greg Utas, *Robust Communications Software* (Wiley, 2005)
- Robert S. Hanmer, *Patterns for Fault Tolerant Softwar*e (Wiley, 2007)
- *Pattern Oriented Software Architecture, volume 2* by Doug Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann (Wiley, 2000)

Websites

- wiki.c2.com/?DesignPatternsBook
- hillside.net/patterns/patterns-catalog
- www.martinfowler.com/articles/enterprisePatterns.html
- www.headfirstlabs.com/books/hfdp

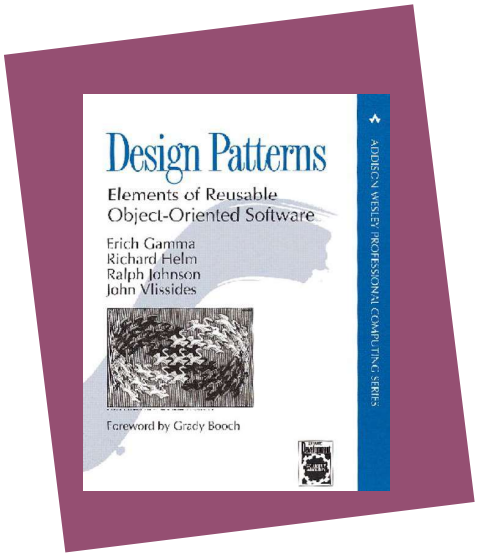
Books and articles

Articles on principles and heuristics

- Robert Martin's OO design principles
  - http://manclswx.com/talks/Principles_and_Patterns.pdf
- Arthur Riel's OO design heuristics
  - http://manclswx.com/talks/top_heuristics.html

What have we learned?

- Design Patterns: an important set of object oriented design concepts
  - these patterns are useful in many applications
  - every pattern has a documented set of “Consequences”
- Add to your design vocabulary…



This talk:
http://manclswx.com/talks/patterns_overview_talk_2017.html