Patterns as solutions to problems

- **A design pattern is the solution to a problem in a context**
  - Has a name that helps in remembering/understanding
  - Has forces that describe the situations in which applicable
    - Should supply pros/cons in using the pattern
  - Has a description that summarizes purpose

- **Adapter**
  - You have a class that’s close to what you want, but the interface isn’t quite right, or some functionality is missing
  - Use an *adapter*, adapt the existing class to a new interface
  - Also known as *wrapper*, similar to proxy but changes interface/adds functionality (proxy doesn't)
Recall Anaword class from CPS100

- **We want to find anagrams, solve jumbles**
  - created, catered, reacted are equal anagramatically
  - We want a class that works like a string except when compared to other strings
  - *Adapt* the string class by creating a *wrapper*: Anaword

- **In C++ we can overload operators to help syntactically**
  - Why can we print, sort, compare, read Anaword objects?
  - See details in anaword.h, Tapestry Howto E

- **Other patterns in Anaword implementation**
  - *Factory*, toward the use of *Singleton*
Consider Card class in FreeCell

- Tradeoffs in creating the following query methods?
  - `sameColor, sameSuit, sameRank, rankOneLess`
  - All are bool methods, envision use in playing games
  - Worry about creating too many methods? Too few?

- What about Construction, Copy, Assignment of cards
  - Should we able to make copies of cards? Why?
  - Should we think about this? Worry about this?
  - Are there idiomatic (language) solutions for this?

- We can make constructor, assignment operator private
  - Who can call private methods?
Classes, compilers, dependencies

```cpp
#include <string>
#include "day.h"

typedef string TimeRange;
class ostream;

class Appointment
{
    public:
        TimeRange duration();
        void print(ostream & output);
    private:
        Day myDay;
}
```

- **why use** class ostream **instead of** #include <stream>
- **what is a typedef and how is it used?**
- **make depend for Appointment/ostream?**
- **Do changes to Day force recompile for Appointment clients?**
.h guidelines, preprocessor in action

- **minimize #includes in every .h file**
  - avoid circular dependencies
  - avoid re-compile by minimizing dependencies

- **class Foo in foo.h, class Bar in bar.h, client foobar.cpp**

```cpp
#ifndef _FOO_H
#define _FOO_H
#define _BAR_H
#include "bar.h"
class Foo
{
  Bar getBar();
  Foo getFoo();
}
```

// from foo.cpp

```cpp
#include "bar.h"
#include "foo.h"
void Foo::doStuff(const Bar & b)...
```

- **use forward references, avoid #include when possible**
#include “foo.h”

- will be needed in .cpp file, e.g., foo.cpp and bar.cpp
- using pointers and references in .h files minimizes dependencies
  - minimize recompiles when .h changes
  - loose coupling: avoid implementation dependencies when possible

- avoid letting implementation leek into public view
  - what about private section?
  - opaque pointer: `FooImpl * myImpl;`
    - implementation of FooImpl is hidden, class can be implemented in foo.cpp (handle-body idiom)
C++ idioms

• **What happens with the statement** `myDay = d;`  
  ➢ assignment is memberwise unless operator `=` overloaded  
  ➢ copy constructor used in passing parameters by value

• **If you need one of:** destructor, assignment operator, copy constructor, you need all of them  
  ➢ heuristic only: managing resources other than memory  
  ➢ preventing objects from being copied  
  ➢ what about non-copyable state, e.g., stream

• In assignment operator, watch for self-assignment
• Study implementation of string/vector
copy constructor

- **Used for “first-time” creation**
  
  ```cpp
  Date d(1,1,2000);
  Date copy(d);
  ```

- **Used for pass-by-value**
  
  ```cpp
  DoStuff(Date d);
  //...
  Date first(1,1,2000);
  DoStuff(first);
  ```

- **what about use of myLength in code as opposed to length()?**

  ```cpp
  Template <class Item>
  tvector(const tvector<Item> & vec)
  // precondition: Item supports assignment
  // postcondition: return copy of vec
  {
    // allocate storage
    myList = new Item[myLength=vec.myLength];
    assert(myList != 0);
    // copy elements
    for(int k = 0; k < vec.myLength; k++)
    {
      myList[k] = vec.myList[k];
    }
  }
  ```