G64OOS (Spring 2014)

Lecture 09

Elements of Reusable Object Oriented Software

Peer-Olaf Siebers

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Motivation

- Understand the concept of Templates and when to use them
- Absorb the principles of the C++ Standard Template Library (the interplay between containers, iterators and algorithms)
- Get acquainted with the ideas of SOLID Design Principles and Design Patterns (what they are and how to use them)

Templates

Templates

- What is a template?
  - A template is a pattern (data type) from which we can create multiple instances (variables or class objects)
  - In C++ templates are entire functions (function templates) or classes (class templates)

- What is the benefit of using templates?
  - Templates make it possible to use one function or class to handle many different data types
    - The compiler creates multiple versions of a function
    - The compiler creates multiple versions of a class

Function Templates

- Example: Standard function (using overloading)

```
int abs(int n){
    return (n<0)?-n:n;
}

double abs(double n){
    return (n<0)?-n:n;
}
```

- These are completely different functions because they handle arguments and return values of different types

- Is overloading the best solution?
  - You can use same name but have to write a separate definition; time and space consuming; errors need to be corrected in each function

Function Templates

- Example: Template function

```
#include <iostream>
using namespace std;

template<class T>
T abs(T n){
    return (n<0)?-n:n;
}

int main(){
    int intX=-5;
    double dblX=-5.0;
    cout<<"abs(int "<<intX<<")="<<abs(intX)<<endl;
    cout<<"abs(double "<<dblX<<")="<<abs(dblX)<<endl;
}
```

- T is called "template argument"
- It will work on all basic data types and even on user-defined data types (once the operators used are overloaded)

Function Templates

- How does it technically work?
  - During compilation code generation does not take place until the function is actually invoked by a statement within the program
  - The compiler then generates a specific version of the abs() function for the data type required by substituting the data type wherever it sees the template argument in the function template

- How to develop a template function?
  - Start with a normal function that works on a fixed type (e.g. int) and debug it
  - Once it works turn it into a template function
  - Check that it works for additional types

Function Templates

- Example: Template function with multiple arguments

```
#include <iostream>
using namespace std;

template<class T>
int find(T* array,T value,int size){
    for(int j=0;j<size;j++)
    if(array[j]==value)return j;
    return -1;
}

int main(){
    int intArr[]={1,3,5,9,11,13};
    int intV=6;
    double dblArr[]={1.0,3.0,5.0,9.0,11.0,13.0};
    double dblV=11.0;

    cout<<"Find "<<intV<<" in intArr: Return value: "<<find(intArr,intV,6)<<endl;
    cout<<"Find "<<dblV<<" in dblArr: Return value: "<<find(dblArr,dblV,6)<<endl;
}
```

Function Templates

- Example: Template function with two template arguments

```
#include <iostream>
using namespace std;

template <class T1,class T2>
T2 find(T1* array,T1 value,T2 size){
    for(int j=0;j<size;j++)
    if(array[j]==value)return j;
    return static_cast<T2>(-1);
}

int main(){
    int intArr[]={1,3,5,9,11,13};
    int intV=6;
    int intSize=6;
    double dblArr[]={1.0,3.0,5.0,9.0,11.0,13.0};
    double dblV=11.0;
    long longSize=6L;
    cout<<"Find "<<intV<<" in intArr: Return value: "<<find(intArr,intV,intSize)<<endl;
    cout<<"Find "<<dblV<<" in dblArr: Return value: "<<find(dblArr,dblV,longSize)<<endl;
}
```

Class Templates

- Example: Standard class

```
class Stack{
    private:
        int st[MAX]; //array of ints
        int top; //index number of top of stack
    public:
        Stack(); //constructor
        void push(int var); //takes int as argument
        int pop(); //returns int value
};

class Stack{
    private:
        double st[MAX]; //array of doubles
        int top; //index number of top of stack
    public:
        Stack(); //constructor
        void push(double var); //takes double as argument
        double pop(); //returns double value
};
```

Class Templates

- Example: Template class

```
#include <iostream>
using namespace std;

const int MAX = 100; //size of array

template<class T>
class Stack{
    private:
        T st[MAX]; //stack: array of any object type
        int top; //index number of top of stack
    public:
        Stack(){top=-1;} //constructor
        void push(T var){st[++top]=var;} //put object on stack
        T pop(){return st[top--];} //take object off stack
};

int main(){
    Stack<int> s1; //s1 is an object of class Stack<int>
    s1.push(2);
    s1.push(3);
    cout<<"Stack<int>:"<<endl;
    cout<<"v1/v2:"<<(s1.pop()/s1.pop())<<endl;
    Stack<double> s2; //s2 is an object of class Stack<double>
    s2.push(2.0);
    s2.push(3.0);
    cout<<"Stack<double>:"<<endl;
    cout<<"v1/v2:"<<(s2.pop()/s2.pop())<<endl;
}
```

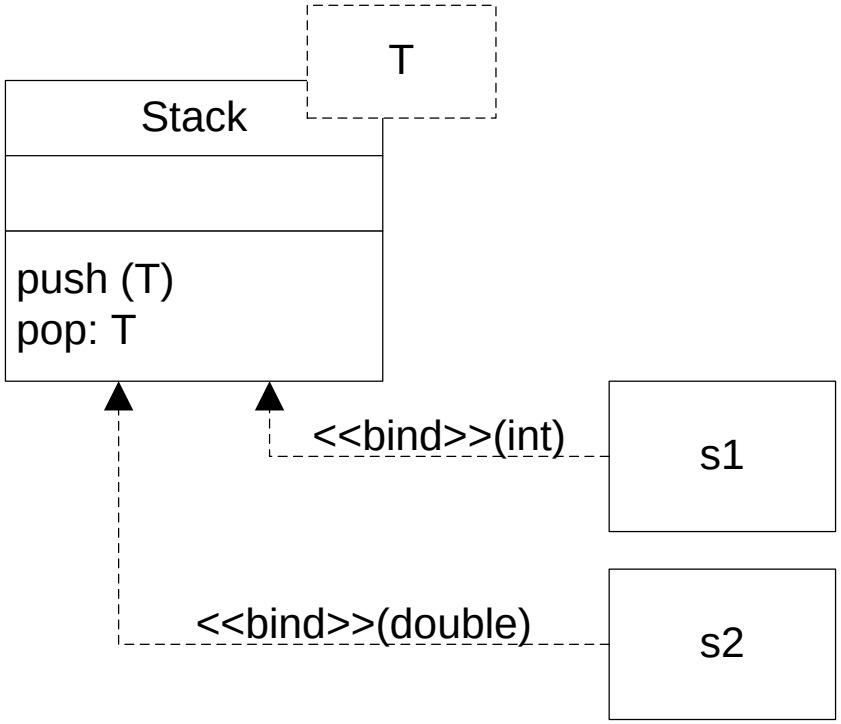
```
Stack<int>:
v1/v2:1
Stack<double>:
v1/v2:1.5
```

Class Templates

- How does it technically work?
  - Classes are instantiated by defining an object using the template argument (e.g. "Stack<double> s2;“)
  - The compiler provides space in memory for this object’s data, using type double wherever the template argument appears in the class specification
  - It also provides space for the member functions (if these have not been placed in memory by another object of type Stack<double>); these member functions also operate exclusively on type double

UML and Templates

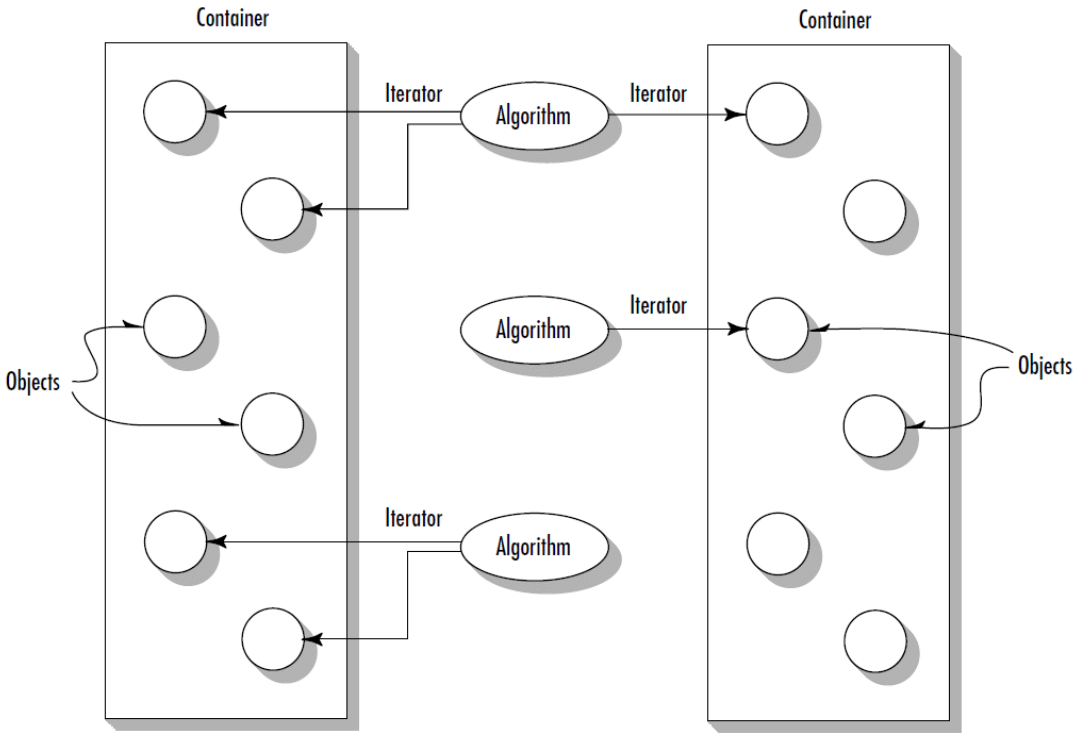
- Example: Template class "Stack"



Standard Template Library

Standard Template Library (STL)

- STL is a collection of classes that provides
  - Template Containers
  - Iterators
  - Algorithms



Algorithms use iterators to act on objects in containers

Containers

- A container is a way to store data - whether the data consists of build-in types or of class objects
- A container usually include functions for
  - Creating an empty container
  - Insert a new object into the container
  - Remove an object from the container
  - Report the current number of objects in the container
  - Empty the container
  - Provide access to the stored objects
  - Sort the elements (optional)

Containers

- Three basic categories
  - Sequence containers (vector; deque; list)
    - Maintain the ordering of elements inside the container; you can chose the position of the element you insert
  - Associative containers (set; multiset; map; multimap)
    - Automatically sort their input when inserted into the container
  - Container adaptors (stack; queue; priority queue)
    - Predefined containers that are adapted for specific use

Containers

- Some useful member functions
  - push_front() push_back(): Inserts a new element at the beginning/end of the container effectively increasing the container size by one .
  - pop_front() pop_back(): Removes first/last element of container, effectively reducing the container size by one and invalidating all iterators and references to it

- The vector and deque containers provide
  - [] Subscripting access without bounds checking (array[…])
  - at Subscripting access with bounds checking (array.at(…))

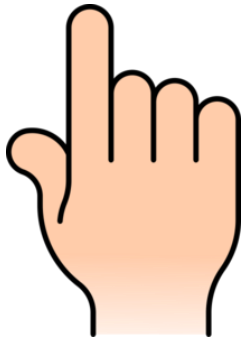
Containers

- Some useful member functions
  - empty Boolean indicating if the container is empty
  - size Returns the number of elements
  - insert Inserts an element at a particular position
  - erase Removes an element at a particular position
  - clear Removes all the elements
  - resize Resizes the container
  - front Returns a reference to the first element
  - back Returns a reference to the last element

Iterators

- What?
  - Objects that can iterate over a container class without the programmer having to know how the container class is implemented
  - Iterators make it easy to step through each element of a container without having to understand how the container class is implemented

- How?
  - An iterator is a pointer to a given element in a container with a set of overloaded operators to provide a set of well-defined functions



Iterators

- Operators
  - "*": Dereferencing the iterator (returns the element that the iterator is currently pointing at)
  - "++": Moves the iterator to the next element in the container (most iterators also provide "--" to move to previous element)
  - "=="; "!=": Basic comparison of operators to determine if two iterators point to the same element (to compare the values that two iterators are pointing at iterators need to be dereferenced first)
  - "=": Assign the iterator to a new position (typically the start or end of the container's elements)

Iterators

- Each container includes four basic functions for use with "="
  - begin() returns iterator representing the beginning of elements in the container; cbegin() returns const iterator
  - end() returns iterator representing the element just past the end of elements; cend() returns const iterator

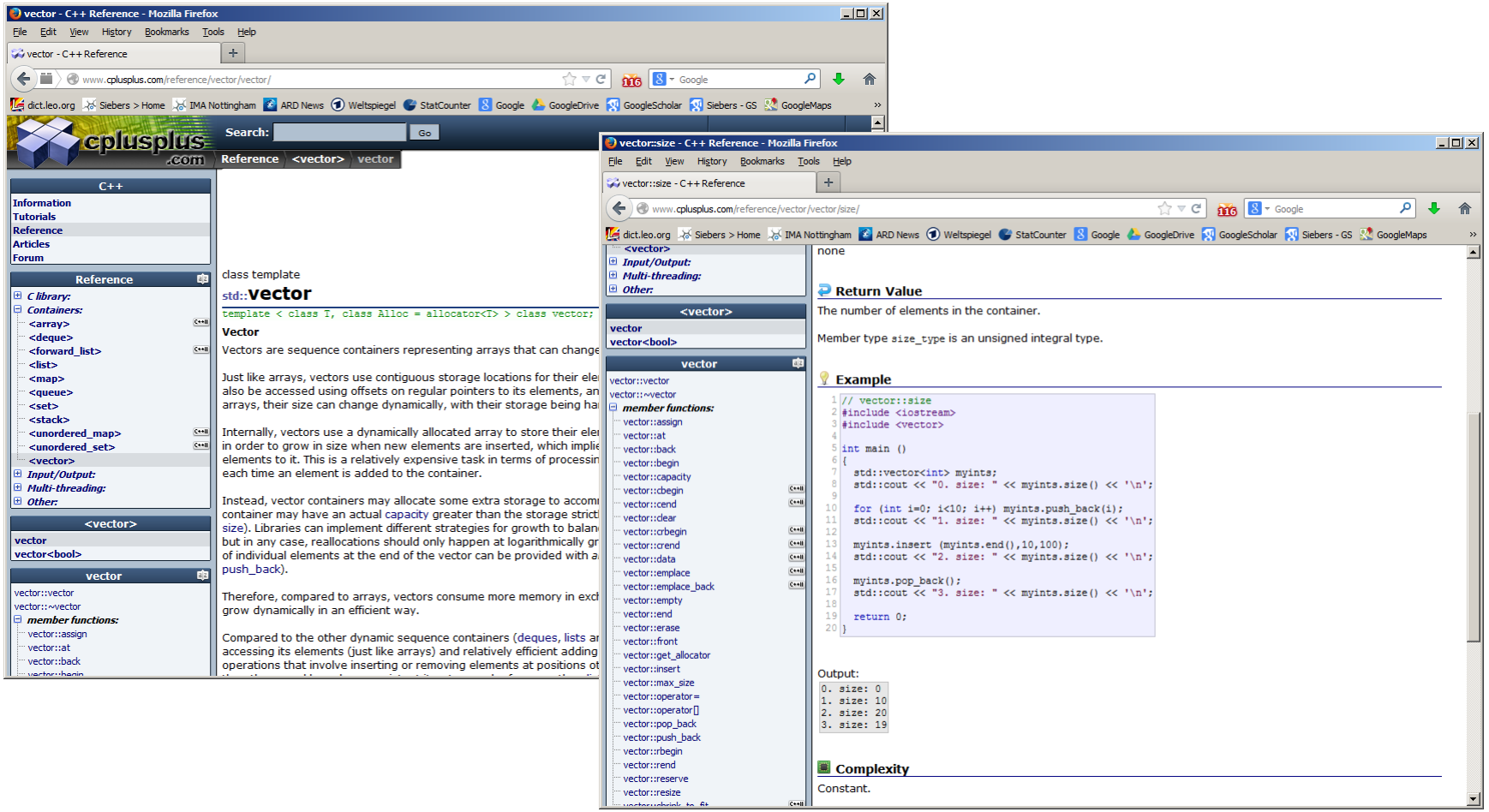
- All containers provide (at least) two types of iterators
  - "container::iterator" provides a read/write iterator
    - for(vector<int>::iterator i=rData.begin();i!=rData.end() ; ++i) cout<<*i;
  - "container::const_iterator" provides a read-only iterator
    - for(vector<int>::const_iterator i=rData.begin(); i!=rData.end();++i) cout<<*i;

Algorithms

- An algorithm is a function that does something to the items in a container (or containers)
  - Examples: find(); count(); equal(); search(); copy(); swap(); fill(); sort()
- Algorithms are stand-alone template functions (global functions that operate using iterators)
- You can use algorithms with built-in C++ arrays or with container classes

Standard Template Library Examples

Source of Information: "cplusplus.com"



Vector::size

```
1  // vector::size
2  #include <iostream>
3  #include <vector>
4
5  int main ()
6  {
7    std::vector<int> myints;
8    std::cout << "0. size: " << myints.size() << '\n';
9
10   for (int i=0; i<10; i++) myints.push_back(i);
11   std::cout << "1. size: " << myints.size() << '\n';
12
13   myints.insert (myints.end(),10,100);
14   std::cout << "2. size: " << myints.size() << '\n';
15
16   myints.pop_back();
17   std::cout << "3. size: " << myints.size() << '\n';
18
19   return 0;
20 }
```

```
0. size: 0
1. size: 10
2. size: 20
3. size: 19
```

Vector::operator[]

```
// vector::operator[]
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (10);   // 10 zero-initialized elements

  std::vector<int>::size_type sz = myvector.size();

  // assign some values:
  for (unsigned i=0; i<sz; i++) myvector[i]=i;

  // reverse vector using operator[]:
  for (unsigned i=0; i<sz/2; i++)
  {
    int temp;
    temp = myvector[sz-1-i];
    myvector[sz-1-i]=myvector[i];
    myvector[i]=temp;
  }

  std::cout << "myvector contains:";
  for (unsigned i=0; i<sz; i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

```
myvector contains: 9 8 7 6 5 4 3 2 1 0
```

Inserting into a list

```
// inserting into a list
#include <iostream>
#include <list>
#include <vector>

int main ()
{
  std::list<int> mylist;
  std::list<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=5; ++i) mylist.push_back(i); // 1 2 3 4 5

  it = mylist.begin();
  ++it;       // it points now to number 2           ^

  mylist.insert (it,10);                        // 1 10 2 3 4 5

  // "it" still points to number 2                     ^
  mylist.insert (it,2,20);                      // 1 10 20 20 2 3 4 5

  --it;       // it points now to the second 20            ^

  std::vector<int> myvector (2,30);
  mylist.insert (it,myvector.begin(),myvector.end());
                                                // 1 10 20 30 30 20 2 3 4 5
                                                //               ^
  std::cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

```
mylist contains: 1 10 20 30 30 20 2 3 4 5
```

Sort algorithm example

```
// sort algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector

bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33

  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)

  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

```
myvector contains: 12 26 32 33 45 53 71 80
```

Find example

```
1  // find example
2  #include <iostream>     // std::cout
3  #include <algorithm>    // std::find
4  #include <vector>       // std::vector
5
6  int main () {
7    int myints[] = { 10, 20, 30 ,40 };
8    int * p;
9
10   // pointer to array element:
11   p = std::find (myints,myints+4,30);
12   ++p;
13   std::cout << "The element following 30 is " << *p << '\n';
14
15   std::vector<int> myvector (myints,myints+4);
16   std::vector<int>::iterator it;
17
18   // iterator to vector element:
19   it = find (myvector.begin(), myvector.end(), 30);
20   ++it;
21   std::cout << "The element following 30 is " << *it << '\n';
22
23   return 0;
24 }
```

```
The element following 30 is 40
The element following 30 is 40
```

S.O.L.I.D.

Principles

SOLID Design Principles

- Software solves real life business problems and real life business processes evolve and change - always.
- A smartly designed software can adjust changes easily; it can be extended, and it is re-usable.

- SOLID Principles (by Uncle Bob) [http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod]
  - S = Single Responsibility Principle
  - O = Open-Closed Principle
  - L = Liscov Substitution Principle
  - I = Interface Segregation Principle
  - D = Dependency Inversion Principle

SOLID Design Principles

- Single Responsibility Principle
  - A class should have one and only one responsibility
- Open-Closed Principle
  - Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification
- Liskov's Substitution Principle
  - Subtypes must be substitutable for their base types

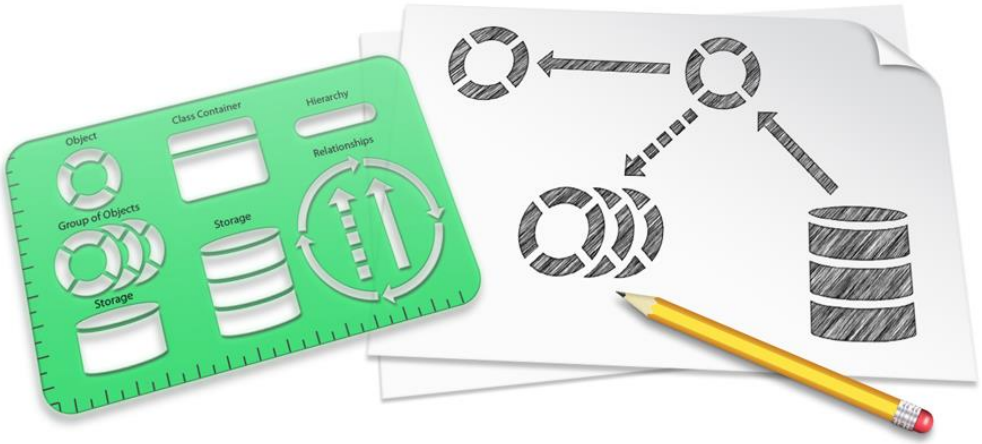
SOLID Design Principles

- Interface Segregation Principle
  - Clients should not be forced to depend upon interfaces that they do not use

- Dependency Inversion Principle
  - High level modules should not depend upon low level modules. Rather, both should depend upon abstractions

http://www.codeproject.com/Articles/93369/How-I-explained-OOD-to-my-wife

Design Patterns



Design Patterns

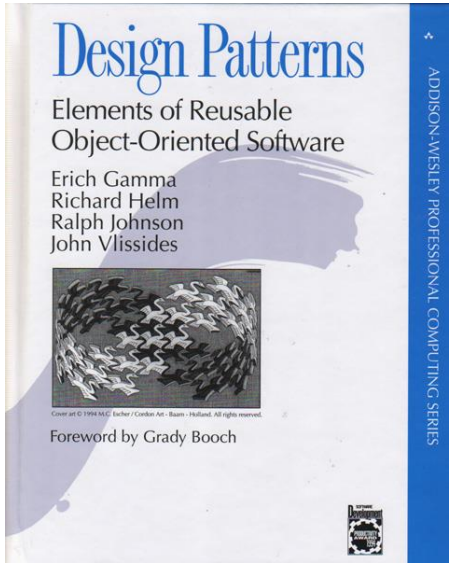
- What is a Design Pattern?
  - A pattern describes a problem which occurs over and over again and then describes the core of the solution to that problem in such a way that it can use the solution over and over again without ever doing it the same way again.
  - A pattern provides an abstract description of a design problem and how a general arrangement of elements solves it
  - A design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities

Design Patterns

- Design patterns are organised in two ways:
  - Purpose: Reflects what the pattern does
    - **Creational:** Concern the process of object creation
    - **Structural:** Deal with the composition of classes and objects
    - **Behavioural:** Characterise the way in which classes and objects interact and distribute responsibility
  - Scope: Specifies whether the pattern applies to classes or objects
    - **Class:** These patterns deal with relationships between classes and sub-classes (which are fixed at compile time)
    - **Object:** Deal with object relationships (which can be changed at runtime)

Design Patterns

- Design pattern organisation



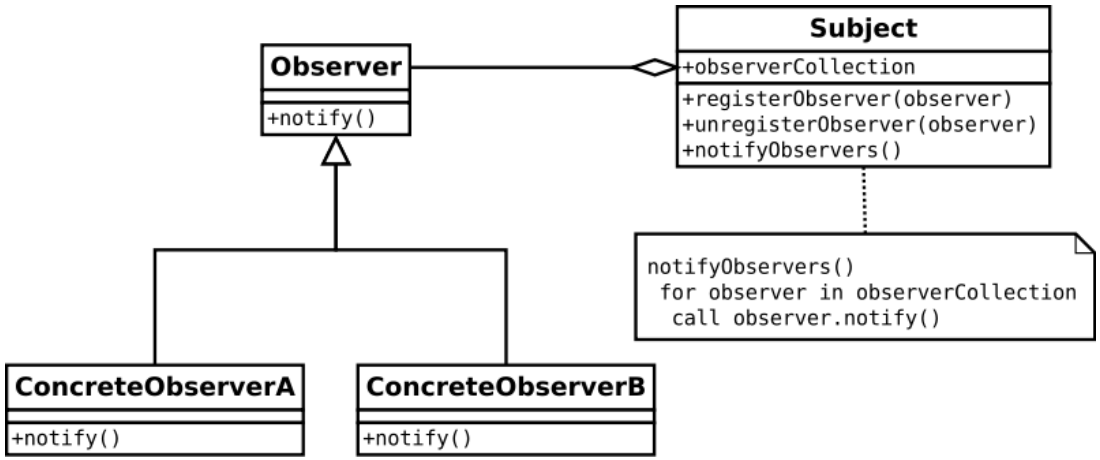
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter
				Template Method
	Object	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Proxy	Flyweight
				Observer
				State
				Strategy
				Visitor

Design Patterns Video Tutorials on YouTube: http://www.youtube.com/playlist?list=PLF206E906175C7E07

Demystifying Design Patterns: How many ways are there to write "hello world"? http://calumgrant.net/patterns/

Example: Observer Pattern

- Defines a one-to-many dependency between objects; when one object changes state others are notified and updated
  - Principle: Strive for loosely coupled designs between objects that interact; these are much more flexible and resilient to change
    - Observers are loosely coupled in that the Observable knows nothing about them , other than that they implement the Observer interface



Example: Observer Pattern

```
50  int main(){
51      // we have several shops that want to be kept up to date about price changes
52      ConcreteSubject product;
53      ConcreteObserver shop1("Shop1");
54      ConcreteObserver shop2("Shop2");
55      product.registerObserver(&shop1);
56      product.registerObserver(&shop2);
57      product.changePrice(25.5);
58      product.unregisterObserver(&shop1);
59      product.changePrice(26.0);
60      return 0;
61  }
```

```
Shop1: new price (25.5) received
Shop2: new price (25.5) received
Shop2: new price (26) received
```

Example: Observer Pattern

```
25  class Subject{ // generic product
26  private:
27      vector<ConcreteObserver*> shopList;
28  public:
29      void registerObserver(ConcreteObserver* shop){
30          shopList.push_back(shop);}
31      void unregisterObserver(ConcreteObserver* shop){
32          shopList.erase(remove(shopList.begin(),shopList.end(),shop),shopList.end());
33          }
34      void notifyObserver(float price){
35          for(vector<ConcreteObserver*>::const_iterator it=shopList.begin();it!=shopList.end();++it){
36              if(*it!=0){
37                  (*it)->notify(price);
38              }
39          }
40      }
41  };
42
43  class ConcreteSubject:public Subject{ // specific product
44  public:
45      void changePrice(float price){
46          notifyObserver(price);
47      }
48  };
```

Example: Observer Pattern

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>
4
5   using namespace std;
6
7   class Observer{ // interface for shop
8       virtual void notify(float price)=0;
9   };
10
11  class ConcreteObserver:public Observer{ // specific shop
12  private:
13      string name;
14      float price;
15  public:
16      ConcreteObserver(string name){
17          this->name=name;
18      }
19      void notify(float price){
20          this->price=price;
21          cout<<name<<": new price ("<<this->price<<") received"<<endl;
22      }
23  };
```

Questions / Comments