Design Patterns

Design Patterns

19-12-2013

Table of Contents

Design Patterns

Programming Patterns

"To understand is to perceive patterns"
—Isaiah Berlin

Software design patterns are abstractions that help structure system designs. While not new, since the concept was already described by Christopher Alexander in its architectural theories, it only gathered some traction in programming due to the publication of Design Patterns: Elements of Reusable Object-Oriented Software book in October 1994 by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, known as the **Gang of Four (GoF)**, that identifies and describes 23 classic software design patterns.

A design pattern is neither a static solution, nor is it an algorithm. A **pattern** is a way to describe and address by name (mostly a simplistic description of its goal), a repeatable solution or approach to a common design problem, that is, a common way to solve a generic problem (how generic or complex, depends on how restricted the target goal is). Patterns can emerge on their own or by design. This is why design patterns are useful as an abstraction over the implementation and a help at design stage. With this concept, an easier way to facilitate communication over a design choice as normalization technique is given so that every person can share the design concept.

Depending on the design problem they address, design patterns can be classified in different categories, of which the main categories are:

- Creational Patterns
- Structural Patterns
- Behavioral Patterns.

Patterns are commonly found in objected-oriented programming languages like C++ or Java. They can be seen as a template for how to solve a problem that occurs in many different situations or applications. It is not code reuse, as it usually does not specify code, but code can be easily created from a design pattern. Object-oriented design patterns typically show relationships and interactions between classes or objects without specifying the final application classes or objects that are involved.

Each design pattern consists of the following parts:

Problem/requirement

To use a design pattern, we need to go through a mini analysis design that may be coded to test out the solution. This section states the requirements of the problem we want to solve. This is usually a common problem that will occur in more than one application.

Forces

This section states the technological boundaries, that helps and guides the creation of the solution.

Solution

This section describes how to write the code to solve the above problem. This is the design part of the design pattern. It may contain class diagrams, sequence diagrams, and or whatever is needed to describe how to code the solution.

Design patterns can be considered as a standardization of commonly agreed best practices to solve specific design problems. One should understand them as a way to implement good design patterns within applications. Doing so will reduce the use of inefficient and obscure solutions. Using design patterns speeds up your design and helps to communicate it to other programmers.

Creational Patterns

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

In this section of the book we assume that the reader has enough familiarity with functions, global variables, stack vs. heap, classes, pointers, and static member functions as introduced before.

As we will see there are several creational design patterns, and all will deal with a specific implementation task, that will create a higher level of abstraction to the code base, we will now cover each one.

Builder

The Builder Creational Pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.

Problem

We want to construct a complex object, however we do not want to have a complex constructor member or one that would need many arguments.

Solution

Define an intermediate object whose member functions define the desired object part by part before the object is available to the client. Build Pattern lets us defer the construction of the object until all the options for creation have been specified.

```
#include <string>
#include <iostream>

using namespace std;

// "Product"
class Pizza
{
        public:
                void setDough(const string& dough)
                {
                        m_dough = dough;
                }
                void setSauce(const string& sauce)
                {
                        m_sauce = sauce;
                }
                void setTopping(const string& topping)
                {
                        m_topping = topping;
                }
                void open() const
                {
                       cout << "Pizza with " << m_dough << " dough, " << m_sauce
 << " sauce and "
                        << m_topping << " topping. Mmm." << endl;
                }
        private:
                string m_dough;
                string m_sauce;
                string m_topping;
};

// "Abstract Builder"
class PizzaBuilder
{
        public:
                Pizza* getPizza()
                {
                        return m_pizza;
                }
                void createNewPizzaProduct()
                {
                        m_pizza = new Pizza;
                }
                virtual void buildDough() = 0;
                virtual void buildSauce() = 0;
                virtual void buildTopping() = 0;
        protected:
                Pizza* m_pizza;
};

//----------------------------------------------------------------

class HawaiianPizzaBuilder : public PizzaBuilder
{
        public:
                virtual void buildDough()
                {
                        m_pizza->setDough("cross");
                }
                virtual void buildSauce()
                {
                        m_pizza->setSauce("mild");
                }
                virtual void buildTopping()
                {
                        m_pizza->setTopping("ham+pineapple");
                }
};

class SpicyPizzaBuilder : public PizzaBuilder
{
        public:
                virtual void buildDough()
                {
                        m_pizza->setDough("pan baked");
                }
                virtual void buildSauce()
                {
                        m_pizza->setSauce("hot");
                }
                virtual void buildTopping()
                {
                        m_pizza->setTopping("pepperoni+salami");
                }
};
```

```
//------------------------------------------------------------------

class Cook
{
        public:
                void setPizzaBuilder(PizzaBuilder* pb)
                {
                        m_pizzaBuilder = pb;
                }
                Pizza* getPizza()
                {
                        return m_pizzaBuilder->getPizza();
                }
                void constructPizza()
                {
                        m_pizzaBuilder->createNewPizzaProduct();
                        m_pizzaBuilder->buildDough();
                        m_pizzaBuilder->buildSauce();
                        m_pizzaBuilder->buildTopping();
                }
        private:
                PizzaBuilder* m_pizzaBuilder;
};

int main()
{
        Cook cook;
        PizzaBuilder* hawaiianPizzaBuilder = new HawaiianPizzaBuilder;
        PizzaBuilder* spicyPizzaBuilder   = new SpicyPizzaBuilder;

        cook.setPizzaBuilder(hawaiianPizzaBuilder);
        cook.constructPizza();

        Pizza* hawaiian = cook.getPizza();
        hawaiian->open();

        cook.setPizzaBuilder(spicyPizzaBuilder);
        cook.constructPizza();

        Pizza* spicy = cook.getPizza();
        spicy->open();

        delete hawaiianPizzaBuilder;
        delete spicyPizzaBuilder;
        delete hawaiian;
        delete spicy;
}
```

Factory

Definition: A utility class that creates an instance of a class from a family of derived classes

Abstract Factory

Definition: A utility class that creates an instance of several families of classes. It can also return a factory for a certain group.

Factory Method

The Factory Design Pattern is useful in a situation that requires the creation of many different types of objects, all derived from a common base type. The Factory Method defines a method for creating the objects, which subclasses can then override to specify the derived type that will be created. Thus, at run time, the Factory Method can be passed a description of a desired object (e.g., a string read from user input) and return a base class pointer to a new instance of that object. The pattern works best when a well-designed interface is used for the base class, so there is no need to cast the returned object.

Problem

We want to decide at run time what object is to be created based on some configuration or application parameter. When we write the code, we do not know what class should be instantiated.

Solution

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

In the following example, a factory method is used to create laptop or desktop computer objects at run time.

Let's start by defining `Computer`, which is an abstract base class (interface) and its derived classes: `Laptop` and `Desktop`.

```
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;
};
```

```
class Laptop: public Computer
{
public:
    virtual void Run(){mHibernating = false;}
    virtual void Stop(){mHibernating = true;}
private:
    bool mHibernating; // Whether or not the machine is hibernating
};
class Desktop: public Computer
{
public:
    virtual void Run(){mOn = true;}
    virtual void Stop(){mOn = false;}
private:
    bool mOn; // Whether or not the machine has been turned on
};
```

The actual `ComputerFactory` class returns a `Computer`, given a real world description of the object.

```
class ComputerFactory
{
public:
    static Computer *NewComputer(const std::string &description)
    {
        if(description == "laptop")
            return new Laptop;
        if(description == "desktop")
            return new Desktop;
        return NULL;
    }
};
```

Let's analyze the benefits of this design. First, there is a compilation benefit. If we move the interface `Computer` into a separate header file with the factory, we can then move the implementation of the `NewComputer()` function into a separate implementation file. Now the implementation file for `NewComputer()` is the only one that requires knowledge of the derived classes. Thus, if a change is made to any derived class of `Computer`, or a new `Computer` subtype is added, the implementation file for `NewComputer()` is the only file that needs to be recompiled. Everyone who uses the factory will only care about the interface, which should remain consistent throughout the life of the application.

Also, if a new class needs to be added, and the user is requesting objects through a user interface, no code calling the factory may need to change to support the additional computer type. The code using the factory would simply pass on the new string to the factory, and allow the factory to handle the new types entirely.

Imagine programming a video game, where you would like to add new types of enemies in the future, each of which has different AI functions and can update differently. By using a factory method, the controller of the program can call to the factory to create the enemies, without any dependency or knowledge of the actual types of enemies. Now, future developers can create new enemies, with new AI controls and new drawing member functions, add it to the factory, and create a level which calls the factory, asking for the enemies by name. Combine this method with an XML description of levels, and developers could create new levels without having to recompile their program. All this, thanks to the separation of creation of objects from the usage of objects.

Another example:

```
#include <stdexcept>
#include <iostream>
#include <memory>

class Pizza {
public:
    virtual int getPrice() const = 0;
};

class HamAndMushroomPizza : public Pizza {
public:
    virtual int getPrice() const { return 850; }
};

class DeluxePizza : public Pizza {
public:
    virtual int getPrice() const { return 1050; }
};

class HawaiianPizza : public Pizza {
public:
    virtual int getPrice() const { return 1150; }
};

class PizzaFactory {
public:
        enum PizzaType {
        HamMushroom,
        Deluxe,
        Hawaiian
    };

    static Pizza* createPizza(PizzaType pizzaType) {
        switch (pizzaType) {
            case HamMushroom:
                return new HamAndMushroomPizza();
            case Deluxe:
                return new DeluxePizza();
            case Hawaiian:
                return new HawaiianPizza();
        }
        throw "invalid pizza type.";
    }
};

/*
 * Create all available pizzas and print their prices
 */
void pizza_information( PizzaFactory::PizzaType pizzatype )
{
        Pizza* pizza = PizzaFactory::createPizza(pizzatype);
        std::cout << "Price of " << pizzatype << " is " << pizza->getPrice() <<
 std::endl;
        delete pizza;
}

int main ()
{
        pizza_information( PizzaFactory::HamMushroom );
        pizza_information( PizzaFactory::Deluxe );
        pizza_information( PizzaFactory::Hawaiian );
}
```

Prototype

A prototype pattern is used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used, for example, when the inherent cost of creating a new object in the standard way (e.g., using the `new` keyword) is prohibitively expensive for a given application.

Implementation: Declare an abstract base class that specifies a pure virtual `clone()` method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the `clone()` operation.

Here the client code first invokes the factory method. This factory method, depending on the parameter, finds out concrete class. On this concrete class call to the `clone()` method is called and the object is returned by the factory method.

- This is sample code which is a sample implementation of Prototype method. We have the detailed description of all the components here.
    - `Record` class, which is a pure virtual class that is having pure virtual method `clone()`.
    - `CarRecord`, `BikeRecord` and `PersonRecord` as concrete implementation of a `Record` class.
    - An [[C++ Programming/Programming Languages/C++/Code/Keywords/`enum`|`enum`]] RECORD_TYPE_en as one to one mapping of each concrete implementation of `Record` class.
    - `RecordFactory` class that is having `Factory` method `CreateRecord(…)`. This method requires an `enum` RECORD_TYPE_en as parameter and depending on this parameter it returns the concrete implementation of `Record` class.

```
/**
 * Implementation of Prototype Method
 **/
#include <iostream>
#include <map>
#include <string>

using namespace std;

enum RECORD_TYPE_en
{
  CAR,
  BIKE,
  PERSON
};

/**
 * Record is the Prototype
 */

class Record
{
  public :

    Record() {}

    ~Record() {}

    virtual Record* clone()=0;

    virtual void print()=0;
};
```

```
/**
 * CarRecord is a Concrete Prototype
 */

class CarRecord : public Record
{
  private:
    string m_carName;
    int m_ID;

  public:
    CarRecord(string carName, int ID)
      : Record(), m_carName(carName),
        m_ID(ID)
    {
    }

    CarRecord(CarRecord& carRecord)
      : Record()
    {
      m_carName = carRecord.m_carName;
      m_ID = carRecord.m_ID;
    }

    ~CarRecord() {}

    Record* clone()
    {
      return new CarRecord(*this);
    }

    void print()
    {
      cout << "Car Record" << endl
        << "Name  : " << m_carName << endl
        << "Number: " << m_ID << endl << endl;
    }
};


/**
 * BikeRecord is the Concrete Prototype
 */

class BikeRecord : public Record
{
  private :
    string m_bikeName;
    int m_ID;

  public :
    BikeRecord(string bikeName, int ID)
      : Record(), m_bikeName(bikeName),
        m_ID(ID)
    {
    }

    BikeRecord(BikeRecord& bikeRecord)
      : Record()
    {
      m_bikeName = bikeRecord.m_bikeName;
      m_ID = bikeRecord.m_ID;
    }

    ~BikeRecord() {}

    Record* clone()
    {
      return new BikeRecord(*this);
    }

    void print()
    {
      cout << "Bike Record" << endl
        << "Name  : " << m_bikeName << endl
        << "Number: " << m_ID << endl << endl;
    }
};


/**
 * PersonRecord is the Concrete Prototype
 */

class PersonRecord : public Record
{
  private :
    string m_personName;

    int m_age;

  public :
    PersonRecord(string personName, int age)
      : Record(), m_personName(personName),
        m_age(age)
    {
    }

    PersonRecord(PersonRecord& personRecord)
      : Record()
    {
      m_personName = personRecord.m_personName;
      m_age = personRecord.m_age;
    }

    ~PersonRecord() {}

    Record* clone()
    {
      return new PersonRecord(*this);
    }

  void print()
  {
    cout << "Person Record" << endl
      << "Name : " << m_personName << endl
      << "Age  : " << m_age << endl << endl ;
  }
};


/**
 * RecordFactory is the client
 */

class RecordFactory
{
  private :
    map<RECORD_TYPE_en, Record* > m_recordReference;

  public :
    RecordFactory()
    {
      m_recordReference[CAR]  = new CarRecord("Ferrari", 5050);
      m_recordReference[BIKE] = new BikeRecord("Yamaha", 2525);
      m_recordReference[PERSON] = new PersonRecord("Tom", 25);
    }

    ~RecordFactory()
    {
      delete m_recordReference[CAR];
      delete m_recordReference[BIKE];
      delete m_recordReference[PERSON];
    }
    Record* createRecord(RECORD_TYPE_en enType)
    {
      return m_recordReference[enType]->clone();
    }
};

int main()
{
  RecordFactory* poRecordFactory = new RecordFactory();

  Record* poRecord;
  poRecord = poRecordFactory->createRecord(CAR);
  poRecord->print();
  delete poRecord;

  poRecord = poRecordFactory->createRecord(BIKE);
  poRecord->print();
  delete poRecord;

  poRecord = poRecordFactory->createRecord(PERSON);
  poRecord->print();
  delete poRecord;

  delete poRecordFactory;
  return 0;
}
```

Another example:

To implement the pattern, declare an abstract base class that specifies a pure virtual `clone()` member function. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the `clone()` operation.

The client, instead of writing code that invokes the `new` operator on a hard-wired class name, calls the `clone()` member function on the prototype, calls a factory member function with a parameter designating the particular concrete derived class desired, or invokes the `clone()` member function through some mechanism provided by another design pattern.

```
class CPrototypeMonster
{
protected:
    CString           _name;
public:
    CPrototypeMonster();
    CPrototypeMonster( const CPrototypeMonster& copy );
    ~CPrototypeMonster();
    virtual CPrototypeMonster*    Clone() const=0; // This forces every derived
class to provide an overload for this function.
    void        Name( CString name );
    CString    Name() const;
};

class CGreenMonster : public CPrototypeMonster
{
protected:
    int         _numberOfArms;
    double      _slimeAvailable;
public:
    CGreenMonster();
    CGreenMonster( const CGreenMonster& copy );
    ~CGreenMonster();

    virtual CPrototypeMonster*    Clone() const;
    void  NumberOfArms( int numberOfArms );
    void  SlimeAvailable( double slimeAvailable );

    int         NumberOfArms() const;
    double      SlimeAvailable() const;
};

class CPurpleMonster : public CPrototypeMonster
{
protected:
    int         _intensityOfBadBreath;
    double      _lengthOfWhiplikeAntenna;
public:
    CPurpleMonster();
    CPurpleMonster( const CPurpleMonster& copy );
    ~CPurpleMonster();

    virtual CPrototypeMonster*    Clone() const;

    void  IntensityOfBadBreath( int intensityOfBadBreath );
    void  LengthOfWhiplikeAntenna( double lengthOfWhiplikeAntenna );

    int       IntensityOfBadBreath() const;
    double    LengthOfWhiplikeAntenna() const;
};

class CBellyMonster : public CPrototypeMonster
{
protected:
    double      _roomAvailableInBelly;
public:
    CBellyMonster();
    CBellyMonster( const CBellyMonster& copy );
    ~CBellyMonster();

    virtual CPrototypeMonster*    Clone() const;

    void       RoomAvailableInBelly( double roomAvailableInBelly );
    double     RoomAvailableInBelly() const;
};

CPrototypeMonster* CGreenMonster::Clone() const
{
    return new CGreenMonster(*this);
}

CPrototypeMonster* CPurpleMonster::Clone() const
{
    return new CPurpleMonster(*this);
}

CPrototypeMonster* CBellyMonster::Clone() const
{
    return new CBellyMonster(*this);
}
```

A client of one of the concrete monster classes only needs a reference (pointer) to a `CPrototypeMonster` class object to be able to call the 'Clone' function and create copies of that object. The function below demonstrates this concept:

```
void DoSomeStuffWithAMonster( const CPrototypeMonster* originalMonster )
{
    CPrototypeMonster* newMonster = originalMonster->Clone();
    ASSERT( newMonster );

    newMonster->Name("MyOwnMonster");
    // Add code doing all sorts of cool stuff with the monster.
    delete newMonster;
}
```

Now originalMonster can be passed as a pointer to CGreenMonster, CPurpleMonster or CBellyMonster.

Singleton

The term **Singleton** refers to an object that can only be instantiated once. This pattern is generally used where a global variable would have otherwise been used. The main advantage of the singleton is that its existence is guaranteed. Other advantages of the design pattern include the clarity, from the unique access, that the object used is not on the local stack. Some of the downfalls of the object include that, like a global variable, it can be hard to tell what chunk of code corrupted memory, when a bug is found, since everyone has access to it.

TODO
Other pros/cons of the use of singletons.

Let's take a look at how a Singleton differs from other variable types.

Like a global variable, the Singleton exists outside of the scope of any functions. Traditional implementation uses a static member function of the Singleton class, which will create a single instance of the Singleton class on the first call, and forever return that instance. The following code example illustrates the elements of a C++ singleton class, that simply stores a single string.

```
class StringSingleton
{
public:
    // Some accessor functions for the class, itself
    std::string GetString() const
    {return mString;}
    void SetString(const std::string &newStr)
    {mString = newStr;}

    // The magic function, which allows access to the class from anywhere
    // To get the value of the instance of the class, call:
    //     StringSingleton::Instance().GetString();
    static StringSingleton &Instance()
    {
       // This line only runs once, thus creating the only instance in existence

        static StringSingleton *instance = new StringSingleton;
       // dereferencing the variable here, saves the caller from having to use

        // the arrow operator, and removes tempation to try and delete the
        // returned instance.
        return *instance; // always returns the same instance
    }
```

```
 private:
     // We need to make some given functions private to finish the definition of
 the singleton
     StringSingleton(){} // default constructor available only to members or
friends of this class

     // Note that the next two functions are not given bodies, thus any attempt

     // to call them implicitly will return as compiler errors. This prevents
     // accidental copying of the only instance of the class.
     StringSingleton(const StringSingleton &old); // disallow copy constructor
     const StringSingleton &operator=(const StringSingleton &old); //disallow
assignment operator

     // Note that although this should be allowed,
     // some compilers may not implement private destructors
     // This prevents others from deleting our one single instance, which was
otherwise created on the heap
     ~StringSingleton(){}
 private: // private data for an instance of this class
     std::string mString;
 };
```

Variations of Singletons:

TODO
Discussion of Meyers Singleton and any other variations.

Applications of Singleton Class:

One common use of the singleton design pattern is for application configurations. Configurations may need to be accessible globally, and future expansions to the application configurations may be needed. The subset C's closest alternative would be to create a single global `struct`. This had the lack of clarity as to where this object was instantiated, as well as not guaranteeing the existence of the object.

Take, for example, the situation of another developer using your singleton inside the constructor of their object. Then, yet another developer decides to create an instance of the second class in the global scope. If you had simply used a global variable, the order of linking would then matter. Since your global will be accessed, possibly before main begins executing, there is no definition as to whether the global is initialized, or the constructor of the second class is called first. This behavior can then change with slight modifications to other areas of code, which would change order of global code execution. Such an error can be very hard to debug. But, with use of the singleton, the first time the object is accessed, the object will also be created. You now have an object which will always exist, in relation to being used, and will never exist if never used.

A second common use of this class is in updating old code to work in a new architecture. Since developers may have used to use globals liberally, pulling these into a single class and making it a singleton, can allow an intermediary step to bringing a structural program into an object oriented structure.

Another example:

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

/* Place holder for thread synchronization mutex */
class Mutex
{   /* placeholder for code to create, use, and free a mutex */
};

/* Place holder for thread synchronization lock */
class Lock
{   public:
        Lock(Mutex& m) : mutex(m) { /* placeholder code to acquire the mutex */
 }
        ~Lock() { /* placeholder code to release the mutex */ }
    private:
        Mutex & mutex;
};

class Singleton
{   public:
        static Singleton* GetInstance();
        int a;
        ~Singleton() { cout << "In Dtor" << endl; }

    private:
        Singleton(int _a) : a(_a) { cout << "In Ctor" << endl; }


        static Mutex mutex;

        // Not defined, to prevent copying
        Singleton(const Singleton& );
        Singleton& operator =(const Singleton& other);
};

Mutex Singleton::mutex;

Singleton* Singleton::GetInstance()
```

```
{
    Lock lock(mutex);

    cout << "Get Inst" << endl;

    // Initialized during first access
    static Singleton inst(1);

    return &inst;
}

int main()
{
    Singleton* singleton = Singleton::GetInstance();
    cout << "The value of the singleton: " << singleton->a << endl;
    return 0;
}
```

Note:
In the above example, the first call to `Singleton::GetInstance` will initialize the singleton instance. This example is for illustrative purposes only; for anything but a trivial example program, this code contains errors.

Structural Patterns

Adapter

Convert the interface of a class into another interface clients expect. **Adapter** lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge

The Bridge Pattern is used to separate out the interface from its implementation. Doing this gives the flexibility so that both can vary independently.

The following example will output:

```
API1.circle at 1:2 7.5
API2.circle at 5:7 27.5
```

```
#include <iostream>

using namespace std;

/* Implementor*/
class DrawingAPI {
  public:
   virtual void drawCircle(double x, double y, double radius) = 0;
   virtual ~DrawingAPI() {}
};

/* Concrete ImplementorA*/
class DrawingAPI1 : public DrawingAPI {
  public:
   void drawCircle(double x, double y, double radius) {
      cout << "API1.circle at " << x << ':' << y << ' ' << radius << endl;
   }
};

/* Concrete ImplementorB*/
class DrawingAPI2 : public DrawingAPI {
public:
   void drawCircle(double x, double y, double radius) {
      cout << "API2.circle at " << x << ':' << y << ' ' <<  radius << endl;
   }
};

/* Abstraction*/
class Shape {
  public:
   virtual ~Shape() {}
   virtual void draw() = 0;
   virtual void resizeByPercentage(double pct) = 0;
};

/* Refined Abstraction*/
class CircleShape : public Shape {
  public:
   CircleShape(double x, double y,double radius, DrawingAPI *drawingAPI) :
           m_x(x), m_y(y), m_radius(radius), m_drawingAPI(drawingAPI)
   {}
   void draw() {
      m_drawingAPI->drawCircle(m_x, m_y, m_radius);
   }
   void resizeByPercentage(double pct) {
      m_radius *= pct;
   }
  private:
   double m_x, m_y, m_radius;
   DrawingAPI *m_drawingAPI;
};

int main(void) {
   CircleShape circle1(1,2,3,new DrawingAPI1());
   CircleShape circle2(5,7,11,new DrawingAPI2());
   circle1.resizeByPercentage(2.5);
   circle2.resizeByPercentage(2.5);
   circle1.draw();
   circle2.draw();
   return 0;
}
```

Composite

Composite lets clients treat individual objects and compositions of objects uniformly. The Composite pattern can represent both the conditions. In this pattern, one can develop tree structures for representing part-whole hierarchies.

```
#include <vector>
#include <iostream> // std::cout
#include <memory> // std::auto_ptr
#include <algorithm> // std::for_each
#include <functional> // std::mem_fun
using namespace std;

class Graphic
{
public:
  virtual void print() const = 0;
  virtual ~Graphic() {}
};

class Ellipse : public Graphic
{
public:
  void print() const {
    cout << "Ellipse \n";
  }
};

class CompositeGraphic : public Graphic
{
public:
  void print() const {
    // for each element in graphicList_, call the print member function
    for_each(graphicList_.begin(), graphicList_.end(), mem_fun(&Graphic::print));

  }

  void add(Graphic *aGraphic) {
    graphicList_.push_back(aGraphic);
  }

private:
  vector<Graphic*>  graphicList_;
};

int main()
{
  // Initialize four ellipses
  const auto_ptr<Ellipse> ellipse1(new Ellipse());
  const auto_ptr<Ellipse> ellipse2(new Ellipse());
  const auto_ptr<Ellipse> ellipse3(new Ellipse());
  const auto_ptr<Ellipse> ellipse4(new Ellipse());

  // Initialize three composite graphics
  const auto_ptr<CompositeGraphic> graphic(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic1(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic2(new CompositeGraphic());

  // Composes the graphics
  graphic1->add(ellipse1.get());
  graphic1->add(ellipse2.get());
  graphic1->add(ellipse3.get());

  graphic2->add(ellipse4.get());

  graphic->add(graphic1.get());
  graphic->add(graphic2.get());

  // Prints the complete graphic (four times the string "Ellipse")
  graphic->print();
  return 0;
}
```

Decorator

The decorator pattern helps to attach additional behavior or responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. This is also called “Wrapper”.

Facade

The Facade Pattern hides the complexities of the system by providing an interface to the client from where the client can access the system on an unified interface. Facade defines a higher-level interface that makes the subsystem easier to use. For instance making one class method perform a complex process by calling several other classes.

Flyweight

It is the use of sharing mechanism by which you can avoid creating a large number of object instances to represent the entire system by using a smaller set fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight will act as an independent object in each context, becoming indistinguishable from an instance of the object that's not shared. To decide if some part of a program is a candidate for using Flyweights, consider whether it is possible to remove some data from the class and make it extrinsic.

Proxy

The Proxy Pattern will provide an object a surrogate or placeholder for another object to control access to it. It is used when you need to represent a complex object with a simpler one. If creation of an object is expensive, it can be postponed until the very need arises and meanwhile a simpler object can serve as a placeholder. This placeholder object is called the "Proxy" for the complex object.

Curiously Recurring Template

This technique is known more widely as a mixin. Mixins are described in the literature to be a powerful tool for expressing abstractions.

Interface-based Programming (IBP)

Interface-based programming is closely related with Modular Programming and Object-Oriented Programming, it defines the application as a collection of inter-coupled modules (interconnected and which plug into each other via interface). Modules can be unplugged, replaced, or upgraded, without the need of compromising the contents of other modules.

The total system complexity is greatly reduced. Interface Based Programming adds more to modular Programming in that it insists that Interfaces are to be added to these modules. The entire system is thus viewed as Components and the interfaces that helps them to co-act.

Interface-based Programming increases the *modularity* of the application and hence its maintainability at a later development cycles, especially when each module must be developed by different teams. It is a well-known methodology that has been around for a long time and it is a core technology behind frameworks such as CORBA.

This is particularly convenient when third parties develop additional components for the established system. They just have to develop components that satisfy the interface specified by the parent application vendor.

Thus the publisher of the interfaces assures that he will not change the interface and the subscriber agrees to implement the interface as whole without any deviation. An interface is therefore said to be a *Contractual agreement* and the programming paradigm based on this is termed as "interface based programming".

Behavioral Patterns

Chain of Responsibility

Chain of Responsibility pattern has the intent to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chains the receiving objects and passes the requests along the chain until an object handles it.

Command

Command pattern is an Object behavioral pattern that decouples sender and receiver by encapsulating a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo-able operations. It can also be thought as an object oriented equivalent of call back method.

Call Back: It is a function that is registered to be called at later point of time based on user actions.

```
#include <iostream>

using namespace std;
```

```
/*the Command interface*/
class Command
{
public:
        virtual void execute()=0;
};

/*Receiver class*/
class Light {

public:
        Light() {  }

        void turnOn()
        {
                cout << "The light is on" << endl;
        }

        void turnOff()
        {
                cout << "The light is off" << endl;
        }
};

/*the Command for turning on the light*/
class FlipUpCommand: public Command
{
public:

        FlipUpCommand(Light& light):theLight(light)
        {

        }

        virtual void execute()
        {
                theLight.turnOn();
        }

private:
        Light& theLight;
};

/*the Command for turning off the light*/
class FlipDownCommand: public Command
{
public:
        FlipDownCommand(Light& light) :theLight(light)
        {
        }
        virtual void execute()
        {
                theLight.turnOff();
        }
private:
        Light& theLight;
};

class Switch {
public:
        Switch(Command& flipUpCmd, Command& flipDownCmd)
        :flipUpCommand(flipUpCmd),flipDownCommand(flipDownCmd)
        {

        }

        void flipUp()
        {
                flipUpCommand.execute();
        }

        void flipDown()
        {
                flipDownCommand.execute();
        }

private:
        Command& flipUpCommand;
        Command& flipDownCommand;
};


/*The test class or client*/
int main()
{
        Light lamp;
        FlipUpCommand switchUp(lamp);
        FlipDownCommand switchDown(lamp);

        Switch s(switchUp, switchDown);
        s.flipUp();
        s.flipDown();
}
```

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator

The 'iterator' design pattern is used liberally within the STL for traversal of various containers. The full understanding of this will liberate a developer to create highly reusable and easily understandable data containers.

The basic idea of the iterator is that it permits the traversal of a container (like a pointer moving across an array). However, to get to the next element of a container, you need not know anything about how the container is constructed. This is the iterators job. By simply using the member functions provided by the iterator, you can move, in the intended order of the container, from the first element to the last element.

Let us start by considering a traditional single dimensional array with a pointer moving from the start to the end. This example assumes knowledge of pointer arithmetic. Note that the use of "it" or "itr," henceforth, is a short version of "iterator."

```
const int ARRAY_LEN = 42;
int *myArray = new int[ARRAY_LEN];
// Set the iterator to point to the first memory location of the array
int *arrayItr = myArray;
// Move through each element of the array, setting it equal to its position in
the array
for(int i = 0; i < ARRAY_LEN; ++i)
{
   // set the value of the current location in the array
   *arrayItr = i;
   // by incrementing the pointer, we move it to the next position in the array.

   // This is easy for a contiguous memory container, since pointer arithmetic

   // handles the traversal.
   ++arrayItr;
}
// Do not be messy, clean up after yourself
delete[] myArray;
```

This code works very quickly for arrays, but how would we traverse a linked list, when the memory is not contiguous? Consider the implementation of a rudimentary linked list as follows:

TODO
test compiling this, check for syntax errors.

```
class IteratorCannotMoveToNext{}; // Error class
class MyIntLList
{
public:
    // The Node class represents a single element in the linked list.
    // The node has a next node and a previous node, so that the user
    // may move from one position to the next, or step back a single
    // position. Notice that the traversal of a linked list is O(N),
    // as is searching, since the list is not ordered.
    class Node
    {
    public:
        Node():mNextNode(0),mPrevNode(0),mValue(0){}
        Node *mNextNode;
        Node *mPrevNode;
        int mValue;
    };
    MyIntLList():mSize(0)
    {}
    ~MyIntLList()
    {
        while(!Empty())
            pop_front();
    } // See expansion for further implementation;
    int Size() const {return mSize;}
    // Add this value to the end of the list
    void push_back(int value)
    {
        Node *newNode = new Node;
        newNode->mValue = value;
        newNode->mPrevNode = mTail;
        mTail->mNextNode = newNode;
        mTail = newNode;
        ++mSize;
    }
    // Remove the value from the beginning of the list
    void pop_front()
    {
        if(Empty())
            return;
        Node *tmpnode = mHead;
        mHead = mHead->mNextNode
        delete tmpnode;
        --mSize;
    }
    bool Empty()
    {return mSize == 0;}

    // This is where the iterator definition will go,
    // but lets finish the definition of the list, first

private:
    Node *mHead;
    Node *mTail;
    int mSize;
};
```

This linked list has non-contiguous memory, and is therefore not a candidate for pointer arithmetic. And we do not want to expose the internals of the list to other developers, forcing them to learn them, and keeping us from changing it.

This is where the iterator comes in. The common interface makes learning the usage of the container easier, and hides the traversal logic from other developers.

Let us examine the code for the iterator, itself.

```
    /*
     *  The iterator class knows the internals of the linked list, so that it
     *  may move from one element to the next. In this implementation, I have
     *  chosen the classic traversal method of overloading the increment
     *  operators. More thorough implementations of a bi-directional linked
     *  list would include decrement operators so that the iterator may move
     *  in the opposite direction.
     */
    class Iterator
    {
    public:
        Iterator(Node *position):mCurrNode(position){}
        // Prefix increment
        const Iterator &operator++()
        {
            if(mCurrNode == 0 || mCurrNode->mNextNode == 0)
                throw IteratorCannotMoveToNext();e
            mCurrNode = mCurrNode->mNextNode;
            return *this;
        }
        // Postfix increment
        Iterator operator++(int)
        {
            Iterator tempItr = *this;
            ++(*this);
            return tempItr;
        }
        // Dereferencing operator returns the current node, which should then
        // be dereferenced for the int. TODO: Check syntax for overloading
        // dereferencing operator
        Node * operator*()
        {return mCurrNode;}
        // TODO: implement arrow operator and clean up example usage following
    private:
        Node *mCurrNode;
    };
    // The following two functions make it possible to create
    // iterators for an instance of this class.
   // First position for iterators should be the first element in the container.

    Iterator Begin(){return Iterator(mHead);}
    // Final position for iterators should be one past the last element in the
container.
    Iterator End(){return Iterator(0);}
```

With this implementation, it is now possible, without knowledge of the size of the container or how its data is organized, to move through each element in order, manipulating or simply accessing the data. This is done through the accessors in the MyIntLList class, Begin() and End().

```
// Create a list
MyIntLList mylist;
// Add some items to the list
for(int i = 0; i < 10; ++i)
    myList.push_back(i);
// Move through the list, adding 42 to each item.
for(MyIntLList::Iterator it = myList.Begin(); it != myList.End(); ++it)
    (*it)->mValue += 42;
```

TODO

- Discussion of iterators in the STL, and the usefulness of iterators within the algorithms library.
- Iterators best practices
- Warnings on creation of and usage of

The following program gives the implementation of iterator design pattern with a generic template:

```
/***********************************************************************/
/* Iterator.h                                                          */
/***********************************************************************/
#ifndef MY_ITERATOR_HEADER
#define MY_ITERATOR_HEADER

#include <iterator>
#include <vector>
#include <set>

//////////////////////////////////////////////////////////////////////////
template<class T, class U>
class Iterator
{
public:
        typedef typename std::vector<T>::iterator iter_type;
        Iterator(U *pData):m_pData(pData){
                m_it = m_pData->m_data.begin();
        }

        void first()
        {
                m_it = m_pData->m_data.begin();
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }
private:
        U *m_pData;
        iter_type m_it;
};

template<class T, class U, class A>
class setIterator
{
```

```
public:
        typedef typename std::set<T,U>::iterator iter_type;

        setIterator(A *pData):m_pData(pData)
        {
                m_it = m_pData->m_data.begin();
        }

        void first()
        {
                m_it = m_pData->m_data.begin();
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }

private:
        A                       *m_pData;
        iter_type               m_it;
};
#endif
```

```
/***************************************************************************/
/* Aggregate.h                                                             */
/***************************************************************************/
#ifndef MY_DATACOLLECTION_HEADER
#define MY_DATACOLLECTION_HEADER
#include "Iterator.h"

template <class T>
class aggregate
{
        friend class Iterator<T, aggregate>;
public:
        void add(T a)
        {
                m_data.push_back(a);
        }

        Iterator<T, aggregate> *create_iterator()
        {
                return new Iterator<T, aggregate>(this);
        }


private:
        std::vector<T> m_data;
};
template <class T, class U>
class aggregateSet
{
        friend class setIterator<T, U, aggregateSet>;
public:
        void add(T a)
        {
                m_data.insert(a);
        }

        setIterator<T, U, aggregateSet> *create_iterator()
        {
                return new setIterator<T,U,aggregateSet>(this);
        }

        void Print()
        {
                copy(m_data.begin(), m_data.end(), std::ostream_iterat-
or<T>(std::cout, "\n"));
        }

private:
        std::set<T,U> m_data;
};

#endif
```

```
/**************************************************************************/
/* Iterator Test.cpp                                                      */
/**************************************************************************/
#include <iostream>
#include <string>
#include "Aggregate.h"
using namespace std;

class Money
```

```
{
public:
        Money(int a = 0): m_data(a) {}

        void SetMoney(int a)
        {
                m_data = a;
        }

        int GetMoney()
        {
                return m_data;
        }

private:
        int m_data;
};

class Name
{
public:
        Name(string name): m_name(name) {}

        const string &GetName() const
        {
                return m_name;
        }

        friend ostream &operator<<(ostream& out, Name name)
        {
                out << name.GetName();
                return out;
        }

private:
        string m_name;
};

struct NameLess
{
        bool operator()(const Name &lhs, const Name &rhs) const
        {
                return (lhs.GetName() < rhs.GetName());
        }
};

int main()
{
        //sample 1
        cout << "________________Iterator with
int______________________________________" << endl;
        aggregate<int> agg;

        for (int i = 0; i < 10; i++)
                agg.add(i);

        Iterator< int,aggregate<int> > *it = agg.create_iterator();
        for(it->first(); !it->isDone(); it->next())
                cout << *it->current() << endl;

        //sample 2
        aggregate<Money> agg2;
        Money a(100), b(1000), c(10000);
        agg2.add(a);
        agg2.add(b);
        agg2.add(c);

        cout << "________________Iterator with Class
Money______________________________" << endl;
        Iterator<Money, aggregate<Money> > *it2 = agg2.create_iterator();
        for (it2->first(); !it2->isDone(); it2->next())
                cout << it2->current()->GetMoney() << endl;

        //sample 3
        cout << "________________Set Iterator with Class
Name______________________________" << endl;

        aggregateSet<Name, NameLess> aset;
        aset.add(Name("Qmt"));
        aset.add(Name("Bmt"));
        aset.add(Name("Cmt"));
        aset.add(Name("Amt"));

        setIterator<Name, NameLess, aggregateSet<Name, NameLess> > *it3 =
aset.create_iterator();
        for (it3->first(); !it3->isDone(); it3->next())
                cout << (*it3->current()) << endl;
}
```

Console output:

```
________________Iterator with int______________________________________
0
1
2
3
4
```

```
5
6
7
8
9
________________Iterator with Class Money______________________________
100
1000
10000
________________Set Iterator with Class Name___________________________
Amt
Bmt
Cmt
Qmt
```

Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento

Without violating encapsulation the Memento Pattern will capture and externalize an object's internal state so that the object can be restored to this state later. Though the Gang of Four uses friend as a way to implement this pattern it is not the best design. It can also be implemented using PIMPL (pointer to implementation or opaque pointer). Best Use case is 'Undo-Redo' in an editor.

The Originator (the object to be saved) creates a snap-shot of itself as a Memento object, and passes that reference to the Caretaker object. The Caretaker object keeps the Memento until such a time as the Originator may want to revert to a previous state as recorded in the Memento object.

Observer

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem

In one place or many places in the application we need to be aware about a system event or an application state change. We'd like to have a standard way of subscribing to listening for system events and a standard way of notifying the interested parties. The notification should be automated after an interested party subscribed to the system event or application state change. There also should be a way to unsubscribe.

Forces

Observers and observables probably should be represented by objects. The observer objects will be notified by the observable objects.

Solution

After subscribing the listening objects will be notified by a way of method call.

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

// The Abstract Observer
class ObserverBoardInterface
{
public:
    virtual void update(float a,float b,float c) = 0;
};

// Abstract Interface for Displays
class DisplayBoardInterface
{
public:
    virtual void show() = 0;
};

// The Abstract Subject
class WeatherDataInterface
{
public:
    virtual void registerOb(ObserverBoardInterface* ob) = 0;
    virtual void removeOb(ObserverBoardInterface* ob) = 0;
    virtual void notifyOb() = 0;
};

// The Concrete Subject
class ParaWeatherData: public WeatherDataInterface
{
public:
    void SensorDataChange(float a,float b,float c)
    {
        m_humidity = a;
        m_temperature = b;
        m_pressure = c;
        notifyOb();
    }

    void registerOb(ObserverBoardInterface* ob)
    {
        m_obs.push_back(ob);
    }

    void removeOb(ObserverBoardInterface* ob)
    {
        m_obs.remove(ob);
    }
protected:
    void notifyOb()
    {
        list<ObserverBoardInterface*>::iterator pos = m_obs.begin();
        while (pos != m_obs.end())
        {
            ((ObserverBoardInterface* )(*pos))->update(m_humidity,m_temperat-
ure,m_pressure);
            (dynamic_cast<DisplayBoardInterface*>(*pos))->show();
            ++pos;
        }
    }

private:
    float        m_humidity;
    float        m_temperature;
    float        m_pressure;
    list<ObserverBoardInterface* > m_obs;
};

// A Concrete Observer
class CurrentConditionBoard : public ObserverBoardInterface, public DisplayBoardIn-
terface
{
public:
    CurrentConditionBoard(ParaWeatherData& a):m_data(a)
    {
        m_data.registerOb(this);
    }
    void show()
    {
        cout<<"_____CurrentConditionBoard_____"<<endl;
        cout<<"humidity: "<<m_h<<endl;
        cout<<"temperature: "<<m_t<<endl;
        cout<<"pressure: "<<m_p<<endl;
        cout<<"_______________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        m_h = h;
        m_t = t;
        m_p = p;
    }

private:
    float m_h;
    float m_t;
    float m_p;
    ParaWeatherData& m_data;
};

// A Concrete Observer
class StatisticBoard : public ObserverBoardInterface, public DisplayBoardInterface
{
public:
    StatisticBoard(ParaWeatherData& a):m_maxt(-
1000),m_mint(1000),m_avet(0),m_count(0),m_data(a)
    {
        m_data.registerOb(this);
    }

    void show()
    {
        cout<<"________StatisticBoard_________"<<endl;
        cout<<"lowest  temperature: "<<m_mint<<endl;
        cout<<"highest temperature: "<<m_maxt<<endl;
        cout<<"average temperature: "<<m_avet<<endl;
        cout<<"_______________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        ++m_count;
        if (t>m_maxt)
        {
            m_maxt = t;
        }
        if (t<m_mint)
        {
            m_mint = t;
        }
        m_avet = (m_avet * (m_count-1) + t)/m_count;
    }
```

```
private:
    float m_maxt;
    float  m_mint;
    float m_avet;
    int m_count;
    ParaWeatherData& m_data;
};


int main(int argc, char *argv[])
{

    ParaWeatherData * wdata = new ParaWeatherData;
    CurrentConditionBoard* currentB = new CurrentConditionBoard(*wdata);
    StatisticBoard* statisticB = new StatisticBoard(*wdata);

    wdata->SensorDataChange(10.2, 28.2, 1001);
    wdata->SensorDataChange(12, 30.12, 1003);
    wdata->SensorDataChange(10.2, 26, 806);
    wdata->SensorDataChange(10.3, 35.9, 900);

    wdata->removeOb(currentB);

    wdata->SensorDataChange(100, 40, 1900);

    delete statisticB;
    delete currentB;
    delete wdata;

    return 0;
}
```

State

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear as having changed its class.

Strategy

Defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients who use it.

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

```
class StrategyInterface
{
    public:
        virtual void execute() const = 0;
};

class ConcreteStrategyA: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyA execute method" << endl;
        }
};

class ConcreteStrategyB: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyB execute method" << endl;
        }
};

class ConcreteStrategyC: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyC execute method" << endl;
        }
};

class Context
{
    private:
        StrategyInterface * strategy_;

    public:
        explicit Context(StrategyInterface *strategy):strategy_(strategy)
        {
        }

        void set_strategy(StrategyInterface *strategy)
        {
            strategy_ = strategy;
        }
        void execute() const
        {
            strategy_->execute();
        }
};

int main(int argc, char *argv[])
{
    ConcreteStrategyA concreteStrategyA;
    ConcreteStrategyB concreteStrategyB;
    ConcreteStrategyC concreteStrategyC;

    Context contextA(&concreteStrategyA);
    Context contextB(&concreteStrategyB);
    Context contextC(&concreteStrategyC);

    contextA.execute(); // output: "Called ConcreteStrategyA execute method"
    contextB.execute(); // output: "Called ConcreteStrategyB execute method"
    contextC.execute(); // output: "Called ConcreteStrategyC execute method"

    contextA.set_strategy(&concreteStrategyB);
    contextA.execute(); // output: "Called ConcreteStrategyB execute method"
    contextA.set_strategy(&concreteStrategyC);
    contextA.execute(); // output: "Called ConcreteStrategyC execute method"

    return 0;
}
```

Template Method

By defining a skeleton of an algorithm in an operation, deferring some steps to subclasses, the Template Method lets subclasses redefine certain steps of that algorithm without changing the algorithms structure.

Visitor

The Visitor Pattern will represent an operation to be performed on the elements of an object structure by letting you define a new operation without changing the classes of the elements on which it operates.

```
#include <string>
#include <iostream>
#include <vector>
```

```
using namespace std;

class Wheel;
class Engine;
class Body;
class Car;

// interface to all car 'parts'
struct CarElementVisitor
{
  virtual void visit(Wheel& wheel) const = 0;
  virtual void visit(Engine& engine) const = 0;
  virtual void visit(Body& body) const = 0;

  virtual void visitCar(Car& car) const = 0;
  virtual ~CarElementVisitor() {}
};

// interface to one part
struct CarElement
{
  virtual void accept(const CarElementVisitor& visitor) = 0;
  virtual ~CarElement() {}
};

// wheel element, there are four wheels with unique names
class Wheel : public CarElement
{
public:
  explicit Wheel(const string& name) :
    name_(name)
  {
  }
  const string& getName() const
  {
    return name_;
  }
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
private:
    string name_;
};

// engine
class Engine : public CarElement
{
```

```
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
};

// body
class Body : public CarElement
{
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
};

// car, all car elements(parts) together
class Car
{
public:
  vector<CarElement*>& getElements()
  {
    return elements_;
  }
  Car()
  {
    // assume that neither push_back nor Wheel(const string&) may throw
    elements_.push_back( new Wheel("front left") );
    elements_.push_back( new Wheel("front right") );
    elements_.push_back( new Wheel("back left") );
    elements_.push_back( new Wheel("back right") );
    elements_.push_back( new Body() );
    elements_.push_back( new Engine() );
  }
  ~Car()
  {
    for(vector<CarElement*>::iterator it = elements_.begin();
      it != elements_.end(); ++it)
    {
      delete *it;
    }
  }
private:
  vector<CarElement*> elements_;
};

// PrintVisitor and DoVisitor show by using a different implementation the Car
class is unchanged
```

```
// even though the algorithm is different in PrintVisitor and DoVisitor.
class CarElementPrintVisitor : public CarElementVisitor
{
public:
  void visit(Wheel& wheel) const
  {
    cout << "Visiting " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Visiting engine" << endl;
  }
  void visit(Body& body) const
  {
    cout << "Visiting body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Visiting car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);        // this issues the callback i.e. to this from
 the element
    }
    cout << "Visited car" << endl;
  }
};

class CarElementDoVisitor : public CarElementVisitor
{
public:
  // these are specific implementations added to the original object without
modifying the original struct
  void visit(Wheel& wheel) const
  {
    cout << "Kicking my " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Starting my engine" << endl;
  }
  void visit(Body& body) const
  {
    cout << "Moving my body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Starting my car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);        // this issues the callback i.e. to this from
 the element
    }
    cout << "Stopped car" << endl;
  }
};

int main()
{
  Car car;
  CarElementPrintVisitor printVisitor;
  CarElementDoVisitor doVisitor;

  printVisitor.visitCar(car);
  doVisitor.visitCar(car);

  return 0;
}
```

Model-View-Controller (MVC)

A pattern often used by applications that need the ability to maintain multiple views of the same data. The model-view-controller pattern was until recently a very common pattern especially for graphic user interlace programming, it splits the code in 3 pieces. The model, the view, and the controller.

The Model is the actual data representation (for example, Array vs Linked List) or other objects representing a database. The View is an interface to reading the model or a fat client GUI. The Controller provides the interface of changing or modifying the data, and then selecting the "Next Best View" (NBV).

Newcomers will probably see this "MVC" model as wasteful, mainly because you are working with many extra objects at runtime, when it seems like one giant object will do. But the secret to the MVC pattern is not writing the code, but in maintaining it, and allowing people to modify the code without changing much else. Also, keep in mind, that different developers have different strengths and weaknesses, so team building around MVC is easier. Imagine a View Team that is responsible for great views, a Model Team that knows a lot about data, and a Controller Team that see the big picture of application flow, handing requests, working with the model, and selecting the most appropriate next view for that client.

TODO
Erm, someone please come up with a better example than the following... I can not think of any

For example: A naive central database can be organized using only a "model", for example, a straight array. However, later on, it may be more applicable to use a linked list. All array accesses will have to be remade into their respective Linked List form (for example, you would change myarray[5] into mylist.at(5) or whatever is equivalent in the language you use).

Well, if we followed the MVC pattern, the central database would be accessed using some sort of a function, for example, myarray.at(5). If we change the model from an array to a linked list, all we have to do is change the view with the model, and the whole program is changed. Keep the interface the same but change the underpinnings of it. This would allow us to make optimizations more freely and quickly than before.

One of the great advantages of the Model-View-Controller Pattern is obviously the ability to reuse the application's logic (which is implemented in the model) when implementing a different view. A good example is found in web development, where a common task is to implement an external API inside of an existing piece of software. If the MVC pattern has cleanly been followed, this only requires modification to the controller, which can have the ability to render different types of views dependent on the content type requested by the user agent.

Creational Patterns

Creational Patterns

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

In this section of the book we assume that the reader has enough familiarity with functions, global variables, stack vs. heap, classes, pointers, and static member functions as introduced before.

As we will see there are several creational design patterns, and all will deal with a specific implementation task, that will create a higher level of abstraction to the code base, we will now cover each one.

Builder

The Builder Creational Pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.

Problem

We want to construct a complex object, however we do not want to have a complex constructor member or one that would need many arguments.

Solution

Define an intermediate object whose member functions define the desired object part by part before the object is available to the client. Build Pattern lets us defer the construction of the object until all the options for creation have been specified.

```
#include <string>
#include <iostream>

using namespace std;
```

```
// "Product"
class Pizza
{
        public:
                void setDough(const string& dough)
                {
                        m_dough = dough;
                }
                void setSauce(const string& sauce)
                {
                        m_sauce = sauce;
                }
                void setTopping(const string& topping)
                {
                        m_topping = topping;
                }
                void open() const
                {
                       cout << "Pizza with " << m_dough << " dough, " << m_sauce
 << " sauce and "
                        << m_topping << " topping. Mmm." << endl;
                }
        private:
                string m_dough;
                string m_sauce;
                string m_topping;
};

// "Abstract Builder"
class PizzaBuilder
{
        public:
                Pizza* getPizza()
                {
                        return m_pizza;
                }
                void createNewPizzaProduct()
                {
                        m_pizza = new Pizza;
                }
                virtual void buildDough() = 0;
                virtual void buildSauce() = 0;
                virtual void buildTopping() = 0;
        protected:
                Pizza* m_pizza;
};

//------------------------------------------------------------------
```

```
class HawaiianPizzaBuilder : public PizzaBuilder
{
        public:
                virtual void buildDough()
                {
                        m_pizza->setDough("cross");
                }
                virtual void buildSauce()
                {
                        m_pizza->setSauce("mild");
                }
                virtual void buildTopping()
                {
                        m_pizza->setTopping("ham+pineapple");
                }
};

class SpicyPizzaBuilder : public PizzaBuilder
{
        public:
                virtual void buildDough()
                {
                        m_pizza->setDough("pan baked");
                }
                virtual void buildSauce()
                {
                        m_pizza->setSauce("hot");
                }
                virtual void buildTopping()
                {
                        m_pizza->setTopping("pepperoni+salami");
                }
};

//------------------------------------------------------------------

class Cook
{
        public:
                void setPizzaBuilder(PizzaBuilder* pb)
                {
                        m_pizzaBuilder = pb;
                }
                Pizza* getPizza()
                {
                        return m_pizzaBuilder->getPizza();
                }
                void constructPizza()
                {
                        m_pizzaBuilder->createNewPizzaProduct();
                        m_pizzaBuilder->buildDough();
                        m_pizzaBuilder->buildSauce();
                        m_pizzaBuilder->buildTopping();
                }
        private:
                PizzaBuilder* m_pizzaBuilder;
};

int main()
{
        Cook cook;
        PizzaBuilder* hawaiianPizzaBuilder = new HawaiianPizzaBuilder;
        PizzaBuilder* spicyPizzaBuilder   = new SpicyPizzaBuilder;

        cook.setPizzaBuilder(hawaiianPizzaBuilder);
        cook.constructPizza();

        Pizza* hawaiian = cook.getPizza();
        hawaiian->open();

        cook.setPizzaBuilder(spicyPizzaBuilder);
        cook.constructPizza();

        Pizza* spicy = cook.getPizza();
        spicy->open();

        delete hawaiianPizzaBuilder;
        delete spicyPizzaBuilder;
        delete hawaiian;
        delete spicy;
}
```

Factory

Definition: A utility class that creates an instance of a class from a family of derived classes

Abstract Factory

Definition: A utility class that creates an instance of several families of classes. It can also return a factory for a certain group.

Factory Method

The Factory Design Pattern is useful in a situation that requires the creation of many different types of objects, all derived from a common base type. The Factory Method defines a method for creating the objects, which subclasses can then override to specify the derived type that will be created. Thus, at run time, the Factory Method can be passed a description of a desired object (e.g., a string read from user input) and return a base class pointer to a new instance of that object. The pattern works best when a well-designed interface is used for the base class, so there is no need to cast the returned object.

Problem

We want to decide at run time what object is to be created based on some configuration or application parameter. When we write the code, we do not know what class should be instantiated.

Solution

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

In the following example, a factory method is used to create laptop or desktop computer objects at run time.

Let's start by defining `Computer`, which is an abstract base class (interface) and its derived classes: `Laptop` and `Desktop`.

```
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;
};
class Laptop: public Computer
{
public:
    virtual void Run(){mHibernating = false;}
    virtual void Stop(){mHibernating = true;}
private:
    bool mHibernating; // Whether or not the machine is hibernating
};
class Desktop: public Computer
{
public:
    virtual void Run(){mOn = true;}
    virtual void Stop(){mOn = false;}
private:
    bool mOn; // Whether or not the machine has been turned on
};
```

The actual `ComputerFactory` class returns a `Computer`, given a real world description of the object.

```
class ComputerFactory
{
public:
    static Computer *NewComputer(const std::string &description)
    {
        if(description == "laptop")
            return new Laptop;
        if(description == "desktop")
            return new Desktop;
        return NULL;
    }
};
```

Let's analyze the benefits of this design. First, there is a compilation benefit. If we move the interface `Computer` into a separate header file with the factory, we can then move the implementation of the `NewComputer()` function into a separate implementation file. Now the implementation file for `NewComputer()` is the only one that requires knowledge of the derived classes. Thus, if a change is made to any derived class of `Computer`, or a new `Computer` subtype is added, the implementation file for `NewComputer()` is the only file that needs to be recompiled. Everyone who uses the factory will only care about the interface, which should remain consistent throughout the life of the application.

Also, if a new class needs to be added, and the user is requesting objects through a user interface, no code calling the factory may need to change to support the additional computer type. The code using the factory would simply pass on the new string to the factory, and allow the factory to handle the new types entirely.

Imagine programming a video game, where you would like to add new types of enemies in the future, each of which has different AI functions and can update differently. By using a factory method, the controller of the program can call to the factory to create the enemies, without any dependency or knowledge of the actual types of enemies. Now, future developers can create new enemies, with new AI controls and new drawing member functions, add it to the factory, and create a level which calls the factory, asking for the enemies by name. Combine this method with an XML description of levels, and developers could create new levels without having to recompile their program. All this, thanks to the separation of creation of objects from the usage of objects.

Another example:

```
#include <stdexcept>
#include <iostream>
#include <memory>

class Pizza {
public:
    virtual int getPrice() const = 0;
};

class HamAndMushroomPizza : public Pizza {
public:
    virtual int getPrice() const { return 850; }
};

class DeluxePizza : public Pizza {
public:
    virtual int getPrice() const { return 1050; }
};

class HawaiianPizza : public Pizza {
public:
    virtual int getPrice() const { return 1150; }
};

class PizzaFactory {
public:
        enum PizzaType {
        HamMushroom,
        Deluxe,
        Hawaiian
    };

    static Pizza* createPizza(PizzaType pizzaType) {
        switch (pizzaType) {
            case HamMushroom:
                return new HamAndMushroomPizza();
            case Deluxe:
                return new DeluxePizza();
            case Hawaiian:
                return new HawaiianPizza();
        }
        throw "invalid pizza type.";
    }
};

/*
 * Create all available pizzas and print their prices
 */
void pizza_information( PizzaFactory::PizzaType pizzatype )
{
        Pizza* pizza = PizzaFactory::createPizza(pizzatype);
        std::cout << "Price of " << pizzatype << " is " << pizza->getPrice() <<
 std::endl;
        delete pizza;
}

int main ()
{
        pizza_information( PizzaFactory::HamMushroom );
        pizza_information( PizzaFactory::Deluxe );
        pizza_information( PizzaFactory::Hawaiian );
}
```

Prototype

A prototype pattern is used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used, for example, when the inherent cost of creating a new object in the standard way (e.g., using the `new` keyword) is prohibitively expensive for a given application.

Implementation: Declare an abstract base class that specifies a pure virtual `clone()` method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the `clone()` operation.

Here the client code first invokes the factory method. This factory method, depending on the parameter, finds out concrete class. On this concrete class call to the `clone()` method is called and the object is returned by the factory method.

- This is sample code which is a sample implementation of Prototype method. We have the detailed description of all the components here.
  - `Record` class, which is a pure virtual class that is having pure virtual method `clone()`.
  - `CarRecord`, `BikeRecord` and `PersonRecord` as concrete implementation of a `Record` class.
  - An [[C++ Programming/Programming Languages/C++/Code/Keywords/`enum`|`enum`]] RECORD_TYPE_en as one to one mapping of each concrete implementation of `Record` class.

- RecordFactory class that is having Factory method CreateRecord(…). This method requires an enum RECORD_TYPE_en as parameter and depending on this parameter it returns the concrete implementation of Record class.

```
/**
 * Implementation of Prototype Method
 **/
#include <iostream>
#include <map>
#include <string>

using namespace std;

enum RECORD_TYPE_en
{
  CAR,
  BIKE,
  PERSON
};

/**
 * Record is the Prototype
 */

class Record
{
  public :

    Record() {}

    ~Record() {}

    virtual Record* clone()=0;

    virtual void print()=0;
};

/**
 * CarRecord is a Concrete Prototype
 */

class CarRecord : public Record
{
  private:
    string m_carName;
    int m_ID;

  public:
    CarRecord(string carName, int ID)
      : Record(), m_carName(carName),
        m_ID(ID)
    {
    }

    CarRecord(CarRecord& carRecord)
      : Record()
    {
      m_carName = carRecord.m_carName;
      m_ID = carRecord.m_ID;
    }

    ~CarRecord() {}

    Record* clone()
    {
      return new CarRecord(*this);
    }

    void print()
    {
      cout << "Car Record" << endl
        << "Name  : " << m_carName << endl
        << "Number: " << m_ID << endl << endl;
    }
};


/**
 * BikeRecord is the Concrete Prototype
 */

class BikeRecord : public Record
{
  private :
    string m_bikeName;

    int m_ID;

  public :
    BikeRecord(string bikeName, int ID)
      : Record(), m_bikeName(bikeName),
        m_ID(ID)
    {
    }

    BikeRecord(BikeRecord& bikeRecord)
      : Record()
    {
      m_bikeName = bikeRecord.m_bikeName;
      m_ID = bikeRecord.m_ID;
    }

    ~BikeRecord() {}

    Record* clone()
    {
      return new BikeRecord(*this);
    }

    void print()
    {
      cout << "Bike Record" << endl
        << "Name  : " << m_bikeName << endl
        << "Number: " << m_ID << endl << endl;
    }
};


/**
 * PersonRecord is the Concrete Prototype
 */

class PersonRecord : public Record
{
  private :
    string m_personName;

    int m_age;

  public :
    PersonRecord(string personName, int age)
      : Record(), m_personName(personName),
        m_age(age)
    {
    }

    PersonRecord(PersonRecord& personRecord)
      : Record()
    {
      m_personName = personRecord.m_personName;
      m_age = personRecord.m_age;
    }

    ~PersonRecord() {}

    Record* clone()
    {
      return new PersonRecord(*this);
    }

  void print()
  {
    cout << "Person Record" << endl
      << "Name : " << m_personName << endl
      << "Age  : " << m_age << endl << endl ;
  }
};


/**
 * RecordFactory is the client
 */

class RecordFactory
{
  private :
    map<RECORD_TYPE_en, Record* > m_recordReference;

  public :
    RecordFactory()
    {
      m_recordReference[CAR]  = new CarRecord("Ferrari", 5050);
      m_recordReference[BIKE] = new BikeRecord("Yamaha", 2525);
      m_recordReference[PERSON] = new PersonRecord("Tom", 25);
    }

    ~RecordFactory()
    {
      delete m_recordReference[CAR];
      delete m_recordReference[BIKE];
      delete m_recordReference[PERSON];
    }

    Record* createRecord(RECORD_TYPE_en enType)
    {
      return m_recordReference[enType]->clone();
    }
};

int main()
{
  RecordFactory* poRecordFactory = new RecordFactory();

  Record* poRecord;
  poRecord = poRecordFactory->createRecord(CAR);
```

```
    poRecord->print();
    delete poRecord;

    poRecord = poRecordFactory->createRecord(BIKE);
    poRecord->print();
    delete poRecord;

    poRecord = poRecordFactory->createRecord(PERSON);
    poRecord->print();
    delete poRecord;

    delete poRecordFactory;
    return 0;
  }
```

Another example:

To implement the pattern, declare an abstract base class that specifies a pure virtual `clone()` member function. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the `clone()` operation.

The client, instead of writing code that invokes the `new` operator on a hard-wired class name, calls the `clone()` member function on the prototype, calls a factory member function with a parameter designating the particular concrete derived class desired, or invokes the `clone()` member function through some mechanism provided by another design pattern.

```
class CPrototypeMonster
{
protected:
    CString           _name;
public:
    CPrototypeMonster();
    CPrototypeMonster( const CPrototypeMonster& copy );
    ~CPrototypeMonster();

    virtual CPrototypeMonster*    Clone() const=0; // This forces every derived
class to provide an overload for this function.
    void        Name( CString name );
    CString    Name() const;
};

class CGreenMonster : public CPrototypeMonster
{
protected:
    int          _numberOfArms;
    double       _slimeAvailable;
```

```
public:
    CGreenMonster();
    CGreenMonster( const CGreenMonster& copy );
    ~CGreenMonster();

    virtual CPrototypeMonster*    Clone() const;
    void  NumberOfArms( int numberOfArms );
    void  SlimeAvailable( double slimeAvailable );

    int         NumberOfArms() const;
    double      SlimeAvailable() const;
};

class CPurpleMonster : public CPrototypeMonster
{
protected:
    int         _intensityOfBadBreath;
    double      _lengthOfWhiplikeAntenna;
public:
    CPurpleMonster();
    CPurpleMonster( const CPurpleMonster& copy );
    ~CPurpleMonster();

    virtual CPrototypeMonster*    Clone() const;

    void  IntensityOfBadBreath( int intensityOfBadBreath );
    void  LengthOfWhiplikeAntenna( double lengthOfWhiplikeAntenna );

    int       IntensityOfBadBreath() const;
    double    LengthOfWhiplikeAntenna() const;
};

class CBellyMonster : public CPrototypeMonster
{
protected:
    double      _roomAvailableInBelly;
public:
    CBellyMonster();
    CBellyMonster( const CBellyMonster& copy );
    ~CBellyMonster();

    virtual CPrototypeMonster*    Clone() const;

    void       RoomAvailableInBelly( double roomAvailableInBelly );
    double     RoomAvailableInBelly() const;
};

CPrototypeMonster* CGreenMonster::Clone() const
{
    return new CGreenMonster(*this);
}

CPrototypeMonster* CPurpleMonster::Clone() const
{
    return new CPurpleMonster(*this);
}

CPrototypeMonster* CBellyMonster::Clone() const
{
    return new CBellyMonster(*this);
}
```

A client of one of the concrete monster classes only needs a reference (pointer) to a `CPrototypeMonster` class object to be able to call the ‘Clone’ function and create copies of that object. The function below demonstrates this concept:

```
void DoSomeStuffWithAMonster( const CPrototypeMonster* originalMonster )
{
    CPrototypeMonster* newMonster = originalMonster->Clone();
    ASSERT( newMonster );

    newMonster->Name("MyOwnMonster");
    // Add code doing all sorts of cool stuff with the monster.
    delete newMonster;
}
```

Now originalMonster can be passed as a pointer to CGreenMonster, CPurpleMonster or CBellyMonster.

Singleton

The term **Singleton** refers to an object that can only be instantiated once. This pattern is generally used where a global variable would have otherwise been used. The main advantage of the singleton is that its existence is guaranteed. Other advantages of the design pattern include the clarity, from the unique access, that the object used is not on the local stack. Some of the downfalls of the object include that, like a global variable, it can be hard to tell what chunk of code corrupted memory, when a bug is found, since everyone has access to it.

TODO
Other pros/cons of the use of singletons.

Let's take a look at how a Singleton differs from other variable types.

Like a global variable, the Singleton exists outside of the scope of any functions. Traditional implementation uses a static member function of the Singleton class, which will create a single instance of the Singleton class on the first call, and forever return that instance. The following code example illustrates the elements of a C++ singleton class, that simply stores a single string.

```
 class StringSingleton
 {
 public:
     // Some accessor functions for the class, itself
     std::string GetString() const
     {return mString;}
     void SetString(const std::string &newStr)
     {mString = newStr;}

     // The magic function, which allows access to the class from anywhere
     // To get the value of the instance of the class, call:
     //     StringSingleton::Instance().GetString();
     static StringSingleton &Instance()
     {
        // This line only runs once, thus creating the only instance in existence

         static StringSingleton *instance = new StringSingleton;
        // dereferencing the variable here, saves the caller from having to use

         // the arrow operator, and removes tempation to try and delete the
         // returned instance.
         return *instance; // always returns the same instance
     }

 private:
     // We need to make some given functions private to finish the definition of
 the singleton
     StringSingleton(){} // default constructor available only to members or
friends of this class

     // Note that the next two functions are not given bodies, thus any attempt

     // to call them implicitly will return as compiler errors. This prevents
     // accidental copying of the only instance of the class.
     StringSingleton(const StringSingleton &old); // disallow copy constructor
     const StringSingleton &operator=(const StringSingleton &old); //disallow
assignment operator

     // Note that although this should be allowed,
     // some compilers may not implement private destructors
     // This prevents others from deleting our one single instance, which was
otherwise created on the heap
     ~StringSingleton(){}
 private: // private data for an instance of this class
     std::string mString;
 };
```

Variations of Singletons:

TODO
Discussion of Meyers Singleton and any other variations.

Applications of Singleton Class:

One common use of the singleton design pattern is for application configurations. Configurations may need to be accessible globally, and future expansions to the application configurations may be needed. The subset C's closest alternative would be to create a single global *`struct`*. This had the lack of clarity as to where this object was instantiated, as well as not guaranteeing the existence of the object.

Take, for example, the situation of another developer using your singleton inside the constructor of their object. Then, yet another developer decides to create an instance of the second class in the global scope. If you had simply used a global variable, the order of linking would then matter. Since your global will be accessed, possibly before main begins executing, there is no definition as to whether the global is initialized, or the constructor of the second class is called first. This behavior can then change with slight modifications to other areas of code, which would change order of global code execution. Such an error can be very hard to debug. But, with use of the singleton, the first time the object is accessed, the object will also be created. You now have an object which will always exist, in relation to being used, and will never exist if never used.

A second common use of this class is in updating old code to work in a new architecture. Since developers may have used to use globals liberally, pulling these into a single class and making it a singleton, can allow an intermediary step to bringing a structural program into an object oriented structure.

Another example:

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

/* Place holder for thread synchronization mutex */
class Mutex
{   /* placeholder for code to create, use, and free a mutex */
};

/* Place holder for thread synchronization lock */
class Lock
{   public:
        Lock(Mutex& m) : mutex(m) { /* placeholder code to acquire the mutex */
 }
        ~Lock() { /* placeholder code to release the mutex */ }
    private:
        Mutex & mutex;
};

class Singleton
{   public:
        static Singleton* GetInstance();
        int a;
        ~Singleton() { cout << "In Dtor" << endl; }

    private:
        Singleton(int _a) : a(_a) { cout << "In Ctor" << endl; }


        static Mutex mutex;

        // Not defined, to prevent copying
        Singleton(const Singleton& );
        Singleton& operator =(const Singleton& other);
};

Mutex Singleton::mutex;

Singleton* Singleton::GetInstance()
{
    Lock lock(mutex);

    cout << "Get Inst" << endl;

    // Initialized during first access
    static Singleton inst(1);

    return &inst;
}

int main()
{
    Singleton* singleton = Singleton::GetInstance();
    cout << "The value of the singleton: " << singleton->a << endl;
    return 0;
}
```

Note:
In the above example, the first call to `Singleton::GetInstance` will initialize the singleton instance. This example is for illustrative purposes only; for anything but a trivial example program, this code contains errors.

Structural Patterns

Structural Patterns

Adapter

Convert the interface of a class into another interface clients expect. **Adapter** lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge

The Bridge Pattern is used to separate out the interface from its implementation. Doing this gives the flexibility so that both can vary independently.

The following example will output:

```
API1.circle at 1:2 7.5
API2.circle at 5:7 27.5
```

```
#include <iostream>

using namespace std;

/* Implementor*/
class DrawingAPI {
  public:
   virtual void drawCircle(double x, double y, double radius) = 0;
   virtual ~DrawingAPI() {}
};

/* Concrete ImplementorA*/
class DrawingAPI1 : public DrawingAPI {
  public:
   void drawCircle(double x, double y, double radius) {
      cout << "API1.circle at " << x << ':' << y << ' ' << radius << endl;
```

```
    }
};

/* Concrete ImplementorB*/
class DrawingAPI2 : public DrawingAPI {
public:
   void drawCircle(double x, double y, double radius) {
      cout << "API2.circle at " << x << ':' << y << ' ' <<  radius << endl;
   }
};

/* Abstraction*/
class Shape {
  public:
   virtual ~Shape() {}
   virtual void draw() = 0;
   virtual void resizeByPercentage(double pct) = 0;
};

/* Refined Abstraction*/
class CircleShape : public Shape {
  public:
   CircleShape(double x, double y,double radius, DrawingAPI *drawingAPI) :
           m_x(x), m_y(y), m_radius(radius), m_drawingAPI(drawingAPI)
   {}
   void draw() {
      m_drawingAPI->drawCircle(m_x, m_y, m_radius);
   }
   void resizeByPercentage(double pct) {
      m_radius *= pct;
   }
  private:
   double m_x, m_y, m_radius;
   DrawingAPI *m_drawingAPI;
};

int main(void) {
   CircleShape circle1(1,2,3,new DrawingAPI1());
   CircleShape circle2(5,7,11,new DrawingAPI2());
   circle1.resizeByPercentage(2.5);
   circle2.resizeByPercentage(2.5);
   circle1.draw();
   circle2.draw();
   return 0;
}
```

Composite

Composite lets clients treat individual objects and compositions of objects uniformly. The Composite pattern can represent both the conditions. In this pattern, one can develop tree structures for representing part-whole hierarchies.

```
#include <vector>
#include <iostream> // std::cout
#include <memory> // std::auto_ptr
#include <algorithm> // std::for_each
#include <functional> // std::mem_fun
using namespace std;

class Graphic
{
public:
  virtual void print() const = 0;
  virtual ~Graphic() {}
};

class Ellipse : public Graphic
{
public:
  void print() const {
    cout << "Ellipse \n";
  }
};

class CompositeGraphic : public Graphic
{
public:
  void print() const {
    // for each element in graphicList_, call the print member function
   for_each(graphicList_.begin(), graphicList_.end(), mem_fun(&Graphic::print));

  }

  void add(Graphic *aGraphic) {
    graphicList_.push_back(aGraphic);
  }

private:
  vector<Graphic*>  graphicList_;
};

int main()
{
  // Initialize four ellipses
  const auto_ptr<Ellipse> ellipse1(new Ellipse());
  const auto_ptr<Ellipse> ellipse2(new Ellipse());
  const auto_ptr<Ellipse> ellipse3(new Ellipse());
  const auto_ptr<Ellipse> ellipse4(new Ellipse());

  // Initialize three composite graphics
  const auto_ptr<CompositeGraphic> graphic(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic1(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic2(new CompositeGraphic());

  // Composes the graphics
  graphic1->add(ellipse1.get());
  graphic1->add(ellipse2.get());
  graphic1->add(ellipse3.get());

  graphic2->add(ellipse4.get());

  graphic->add(graphic1.get());
  graphic->add(graphic2.get());

  // Prints the complete graphic (four times the string "Ellipse")
  graphic->print();
  return 0;
}
```

Decorator

The decorator pattern helps to attach additional behavior or responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. This is also called “Wrapper”.

Facade

The Facade Pattern hides the complexities of the system by providing an interface to the client from where the client can access the system on an unified interface. Facade defines a higher-level interface that makes the subsystem easier to use. For instance making one class method perform a complex process by calling several other classes.

Flyweight

It is the use of sharing mechanism by which you can avoid creating a large number of object instances to represent the entire system by using a smaller set fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight will act as an independent object in each context, becoming indistinguishable from an instance of the object that's not shared. To decide if some part of a program is a candidate for using Flyweights, consider whether it is possible to remove some data from the class and make it extrinsic.

Proxy

The Proxy Pattern will provide an object a surrogate or placeholder for another object to control access to it. It is used when you need to represent a complex object with a simpler one. If creation of an object is expensive, it can be postponed until the very need arises and meanwhile a simpler object can serve as a placeholder. This placeholder object is called the "Proxy" for the complex object.

Curiously Recurring Template

This technique is known more widely as a mixin. Mixins are described in the literature to be a powerful tool for expressing abstractions.

Interface-based Programming (IBP)

Interface-based programming is closely related with Modular Programming and Object-Oriented Programming, it defines the application as a collection of inter-coupled modules (interconnected and which plug into each other via interface). Modules can be unplugged, replaced, or upgraded, without the need of compromising the contents of other modules.

The total system complexity is greatly reduced. Interface Based Programming adds more to modular Programming in that it insists that Interfaces are to be added to these modules. The entire system is thus viewed as Components and the interfaces that helps them to co-act.

Interface-based Programming increases the *modularity* of the application and hence its maintainability at a later development cycles, especially when each module must be developed by different teams. It is a well-known methodology that has been around for a long time and it is a core technology behind frameworks such as CORBA.

This is particularly convenient when third parties develop additional components for the established system. They just have to develop components that satisfy the interface specified by the parent application vendor.

Thus the publisher of the interfaces assures that he will not change the interface and the subscriber agrees to implement the interface as whole without any deviation. An interface is therefore said to be a *Contractual agreement* and the programming paradigm based on this is termed as "interface based programming".

Behavioral Patterns

Behavioral Patterns

Chain of Responsibility

Chain of Responsibility pattern has the intent to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chains the receiving objects and passes the requests along the chain until an object handles it.

Command

Command pattern is an Object behavioral pattern that decouples sender and receiver by encapsulating a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo-able operations. It can also be thought as an object oriented equivalent of call back method.

Call Back: It is a function that is registered to be called at later point of time based on user actions.

```
#include <iostream>

using namespace std;

/*the Command interface*/
class Command
{
public:
        virtual void execute()=0;
};

/*Receiver class*/
class Light {

public:
        Light() {  }
        void turnOn()
        {
                cout << "The light is on" << endl;
        }

        void turnOff()
        {
                cout << "The light is off" << endl;
        }
};

/*the Command for turning on the light*/
class FlipUpCommand: public Command
{
public:

        FlipUpCommand(Light& light):theLight(light)
        {

        }

        virtual void execute()
        {
                theLight.turnOn();
        }

private:
        Light& theLight;
};

/*the Command for turning off the light*/
class FlipDownCommand: public Command
{
public:
        FlipDownCommand(Light& light) :theLight(light)
        {

        }
        virtual void execute()
        {
                theLight.turnOff();
        }
private:
        Light& theLight;
};

class Switch {
public:
        Switch(Command& flipUpCmd, Command& flipDownCmd)
        :flipUpCommand(flipUpCmd),flipDownCommand(flipDownCmd)
        {

        }

        void flipUp()
        {
                flipUpCommand.execute();
        }

        void flipDown()
        {
                flipDownCommand.execute();
        }

private:
        Command& flipUpCommand;
        Command& flipDownCommand;
};


/*The test class or client*/
int main()
{
        Light lamp;
        FlipUpCommand switchUp(lamp);
        FlipDownCommand switchDown(lamp);

        Switch s(switchUp, switchDown);
        s.flipUp();
        s.flipDown();
}
```

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator

The 'iterator' design pattern is used liberally within the STL for traversal of various containers. The full understanding of this will liberate a developer to create highly reusable and easily understandable data containers.

The basic idea of the iterator is that it permits the traversal of a container (like a pointer moving across an array). However, to get to the next element of a container, you need not know anything about how the container is constructed. This is the iterators job. By simply using the member functions provided by the iterator, you can move, in the intended order of the container, from the first element to the last element.

Let us start by considering a traditional single dimensional array with a pointer moving from the start to the end. This example assumes knowledge of pointer arithmetic. Note that the use of "it" or "itr," henceforth, is a short version of "iterator."

```
const int ARRAY_LEN = 42;
int *myArray = new int[ARRAY_LEN];
// Set the iterator to point to the first memory location of the array
int *arrayItr = myArray;
// Move through each element of the array, setting it equal to its position in
the array
for(int i = 0; i < ARRAY_LEN; ++i)
{
   // set the value of the current location in the array
   *arrayItr = i;
   // by incrementing the pointer, we move it to the next position in the array.

   // This is easy for a contiguous memory container, since pointer arithmetic

   // handles the traversal.
   ++arrayItr;
}
// Do not be messy, clean up after yourself
delete[] myArray;
```

This code works very quickly for arrays, but how would we traverse a linked list, when the memory is not contiguous? Consider the implementation of a rudimentary linked list as follows:

TODO
test compiling this, check for syntax errors.

```
class IteratorCannotMoveToNext{}; // Error class
class MyIntLList
{
public:
    // The Node class represents a single element in the linked list.
    // The node has a next node and a previous node, so that the user
    // may move from one position to the next, or step back a single
    // position. Notice that the traversal of a linked list is O(N),
    // as is searching, since the list is not ordered.
    class Node
    {
    public:
        Node():mNextNode(0),mPrevNode(0),mValue(0){}
        Node *mNextNode;
        Node *mPrevNode;
        int mValue;
    };
    MyIntLList():mSize(0)
    {}
    ~MyIntLList()
    {
        while(!Empty())
            pop_front();
    } // See expansion for further implementation;
    int Size() const {return mSize;}
    // Add this value to the end of the list
    void push_back(int value)
    {
        Node *newNode = new Node;
        newNode->mValue = value;
        newNode->mPrevNode = mTail;
        mTail->mNextNode = newNode;
        mTail = newNode;
        ++mSize;
    }
    // Remove the value from the beginning of the list
    void pop_front()
    {
        if(Empty())
            return;
        Node *tmpnode = mHead;
        mHead = mHead->mNextNode
        delete tmpnode;
        --mSize;
    }
    bool Empty()
    {return mSize == 0;}

    // This is where the iterator definition will go,
    // but lets finish the definition of the list, first

private:
    Node *mHead;
    Node *mTail;
    int mSize;
};
```

This linked list has non-contiguous memory, and is therefore not a candidate for pointer arithmetic. And we do not want to expose the internals of the list to other developers, forcing them to learn them, and keeping us from changing it.

This is where the iterator comes in. The common interface makes learning the usage of the container easier, and hides the traversal logic from other developers.

Let us examine the code for the iterator, itself.

```
    /*
     *  The iterator class knows the internals of the linked list, so that it
     *  may move from one element to the next. In this implementation, I have
     *  chosen the classic traversal method of overloading the increment
     *  operators. More thorough implementations of a bi-directional linked
     *  list would include decrement operators so that the iterator may move
     *  in the opposite direction.
     */
    class Iterator
    {
    public:
        Iterator(Node *position):mCurrNode(position){}
        // Prefix increment
        const Iterator &operator++()
        {
            if(mCurrNode == 0 || mCurrNode->mNextNode == 0)
                throw IteratorCannotMoveToNext();e
            mCurrNode = mCurrNode->mNextNode;
            return *this;
        }
        // Postfix increment
        Iterator operator++(int)
        {
            Iterator tempItr = *this;
            ++(*this);
            return tempItr;
        }
        // Dereferencing operator returns the current node, which should then
        // be dereferenced for the int. TODO: Check syntax for overloading
        // dereferencing operator
```

```
          Node * operator*()
          {return mCurrNode;}
          // TODO: implement arrow operator and clean up example usage following
      private:
          Node *mCurrNode;
      };
      // The following two functions make it possible to create
      // iterators for an instance of this class.
     // First position for iterators should be the first element in the container.

      Iterator Begin(){return Iterator(mHead);}
      // Final position for iterators should be one past the last element in the
  container.
      Iterator End(){return Iterator(0);}
```

With this implementation, it is now possible, without knowledge of the size of the container or how its data is organized, to move through each element in order, manipulating or simply accessing the data. This is done through the accessors in the MyIntLList class, Begin() and End().

```
// Create a list
MyIntLList mylist;
// Add some items to the list
for(int i = 0; i < 10; ++i)
    myList.push_back(i);
// Move through the list, adding 42 to each item.
for(MyIntLList::Iterator it = myList.Begin(); it != myList.End(); ++it)
    (*it)->mValue += 42;
```

TODO

- Discussion of iterators in the STL, and the usefulness of iterators within the algorithms library.
- Iterators best practices
- Warnings on creation of and usage of

The following program gives the implementation of iterator design pattern with a generic template:

```
/*************************************************************************/
/* Iterator.h                                                            */
/*************************************************************************/
#ifndef MY_ITERATOR_HEADER
#define MY_ITERATOR_HEADER
```

```
#include <iterator>
#include <vector>
#include <set>

//////////////////////////////////////////////////////////////////////////
template<class T, class U>
class Iterator
{
public:
        typedef typename std::vector<T>::iterator iter_type;
        Iterator(U *pData):m_pData(pData){
                m_it = m_pData->m_data.begin();
        }

        void first()
        {
                m_it = m_pData->m_data.begin();
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }
private:
        U *m_pData;
        iter_type m_it;
};

template<class T, class U, class A>
class setIterator
{
public:
        typedef typename std::set<T,U>::iterator iter_type;

        setIterator(A *pData):m_pData(pData)
        {
                m_it = m_pData->m_data.begin();
        }
        void first()
        {
                m_it = m_pData->m_data.begin();
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }

private:
        A                       *m_pData;
        iter_type               m_it;
};
#endif
```

```
/**************************************************************************/
/* Aggregate.h                                                            */
/**************************************************************************/
#ifndef MY_DATACOLLECTION_HEADER
#define MY_DATACOLLECTION_HEADER
#include "Iterator.h"

template <class T>
class aggregate
{
        friend class Iterator<T, aggregate>;
public:
        void add(T a)
        {
                m_data.push_back(a);
        }

        Iterator<T, aggregate> *create_iterator()
        {
                return new Iterator<T, aggregate>(this);
        }

private:
        std::vector<T> m_data;
};
template <class T, class U>
class aggregateSet
{
        friend class setIterator<T, U, aggregateSet>;
public:
        void add(T a)
        {
                m_data.insert(a);
        }

        setIterator<T, U, aggregateSet> *create_iterator()
        {
                return new setIterator<T,U,aggregateSet>(this);
        }

        void Print()
        {
                copy(m_data.begin(), m_data.end(), std::ostream_iterat-
or<T>(std::cout, "\n"));
        }

private:
        std::set<T,U> m_data;
};

#endif
```

```
/**************************************************************************/
/* Iterator Test.cpp                                                      */
/**************************************************************************/
#include <iostream>
#include <string>
#include "Aggregate.h"
using namespace std;

class Money
{
public:
        Money(int a = 0): m_data(a) {}

        void SetMoney(int a)
        {
                m_data = a;
        }
        int GetMoney()
        {
                return m_data;
        }

private:
        int m_data;
};

class Name
{
public:
        Name(string name): m_name(name) {}

        const string &GetName() const
        {
                return m_name;
        }

        friend ostream &operator<<(ostream& out, Name name)
        {
                out << name.GetName();
                return out;
        }

private:
        string m_name;
};

struct NameLess
{
        bool operator()(const Name &lhs, const Name &rhs) const
        {
                return (lhs.GetName() < rhs.GetName());
        }
};

int main()
{
        //sample 1
        cout << "________________Iterator with
int______________________________________" << endl;
        aggregate<int> agg;

        for (int i = 0; i < 10; i++)
                agg.add(i);

        Iterator< int,aggregate<int> > *it = agg.create_iterator();
        for(it->first(); !it->isDone(); it->next())
                cout << *it->current() << endl;

        //sample 2
        aggregate<Money> agg2;
        Money a(100), b(1000), c(10000);
        agg2.add(a);
        agg2.add(b);
        agg2.add(c);

        cout << "________________Iterator with Class
Money______________________________" << endl;
        Iterator<Money, aggregate<Money> > *it2 = agg2.create_iterator();
        for (it2->first(); !it2->isDone(); it2->next())
                cout << it2->current()->GetMoney() << endl;

        //sample 3
        cout << "________________Set Iterator with Class
Name______________________________" << endl;

        aggregateSet<Name, NameLess> aset;
        aset.add(Name("Qmt"));
        aset.add(Name("Bmt"));
        aset.add(Name("Cmt"));
        aset.add(Name("Amt"));

        setIterator<Name, NameLess, aggregateSet<Name, NameLess> > *it3 =
aset.create_iterator();
        for (it3->first(); !it3->isDone(); it3->next())
                cout << (*it3->current()) << endl;
}
```

Console output:

```
________________Iterator with int______________________________________
0
1
2
3
4
5
6
7
8
9
________________Iterator with Class Money______________________________
100
1000
```

```
10000
________________Set Iterator with Class Name___________________________
Amt
Bmt
Cmt
Qmt
```

Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento

Without violating encapsulation the Memento Pattern will capture and externalize an object’s internal state so that the object can be restored to this state later. Though the Gang of Four uses friend as a way to implement this pattern it is not the best design. It can also be implemented using PIMPL (pointer to implementation or opaque pointer). Best Use case is 'Undo-Redo' in an editor.

The Originator (the object to be saved) creates a snap-shot of itself as a Memento object, and passes that reference to the Caretaker object. The Caretaker object keeps the Memento until such a time as the Originator may want to revert to a previous state as recorded in the Memento object.

Observer

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem

In one place or many places in the application we need to be aware about a system event or an application state change. We'd like to have a standard way of subscribing to listening for system events and a standard way of notifying the interested parties. The notification should be automated after an interested party subscribed to the system event or application state change. There also should be a way to unsubscribe.

Forces

Observers and observables probably should be represented by objects. The observer objects will be notified by the observable objects.

Solution

After subscribing the listening objects will be notified by a way of method call.

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

// The Abstract Observer
class ObserverBoardInterface
{
public:
    virtual void update(float a,float b,float c) = 0;
};

// Abstract Interface for Displays
class DisplayBoardInterface
{
public:
    virtual void show() = 0;
};

// The Abstract Subject
class WeatherDataInterface
{
public:
    virtual void registerOb(ObserverBoardInterface* ob) = 0;
    virtual void removeOb(ObserverBoardInterface* ob) = 0;
    virtual void notifyOb() = 0;
};

// The Concrete Subject
class ParaWeatherData: public WeatherDataInterface
{
public:
    void SensorDataChange(float a,float b,float c)
    {
        m_humidity = a;
        m_temperature = b;
        m_pressure = c;
        notifyOb();
    }

    void registerOb(ObserverBoardInterface* ob)
    {
        m_obs.push_back(ob);
    }

    void removeOb(ObserverBoardInterface* ob)
    {
        m_obs.remove(ob);
    }
protected:
    void notifyOb()
    {
        list<ObserverBoardInterface*>::iterator pos = m_obs.begin();
        while (pos != m_obs.end())
        {
            ((ObserverBoardInterface* )(*pos))->update(m_humidity,m_temperat-
ure,m_pressure);
            (dynamic_cast<DisplayBoardInterface*>(*pos))->show();
            ++pos;
        }
    }

private:
    float        m_humidity;
    float        m_temperature;
    float        m_pressure;
    list<ObserverBoardInterface* > m_obs;
};

// A Concrete Observer
class CurrentConditionBoard : public ObserverBoardInterface, public DisplayBoardIn-
terface
{
public:
    CurrentConditionBoard(ParaWeatherData& a):m_data(a)
    {
        m_data.registerOb(this);
    }
    void show()
    {
        cout<<"_____CurrentConditionBoard_____"<<endl;
        cout<<"humidity: "<<m_h<<endl;
        cout<<"temperature: "<<m_t<<endl;
        cout<<"pressure: "<<m_p<<endl;
        cout<<"_______________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        m_h = h;
        m_t = t;
        m_p = p;
    }

private:
    float m_h;
    float m_t;
    float m_p;
    ParaWeatherData& m_data;
};

// A Concrete Observer
class StatisticBoard : public ObserverBoardInterface, public DisplayBoardInterface
{
public:
    StatisticBoard(ParaWeatherData& a):m_maxt(-
1000),m_mint(1000),m_avet(0),m_count(0),m_data(a)
    {
        m_data.registerOb(this);
    }

    void show()
    {
        cout<<"________StatisticBoard_________"<<endl;
        cout<<"lowest  temperature: "<<m_mint<<endl;
        cout<<"highest temperature: "<<m_maxt<<endl;
        cout<<"average temperature: "<<m_avet<<endl;
        cout<<"_______________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        ++m_count;
        if (t>m_maxt)
        {
            m_maxt = t;
        }
        if (t<m_mint)
        {
            m_mint = t;
        }
        m_avet = (m_avet * (m_count-1) + t)/m_count;
    }

private:
    float m_maxt;
    float  m_mint;
    float m_avet;
    int m_count;
    ParaWeatherData& m_data;
};


int main(int argc, char *argv[])
{

    ParaWeatherData * wdata = new ParaWeatherData;
    CurrentConditionBoard* currentB = new CurrentConditionBoard(*wdata);
    StatisticBoard* statisticB = new StatisticBoard(*wdata);

    wdata->SensorDataChange(10.2, 28.2, 1001);
    wdata->SensorDataChange(12, 30.12, 1003);
    wdata->SensorDataChange(10.2, 26, 806);
    wdata->SensorDataChange(10.3, 35.9, 900);

    wdata->removeOb(currentB);

    wdata->SensorDataChange(100, 40, 1900);

    delete statisticB;
    delete currentB;
    delete wdata;

    return 0;
}
```

State

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear as having changed its class.

Strategy

Defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients who use it.

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

class StrategyInterface
{
    public:
        virtual void execute() const = 0;
};

class ConcreteStrategyA: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyA execute method" << endl;
        }
};

class ConcreteStrategyB: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyB execute method" << endl;
        }
};

class ConcreteStrategyC: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyC execute method" << endl;
        }
};

class Context
{
    private:
        StrategyInterface * strategy_;

    public:
        explicit Context(StrategyInterface *strategy):strategy_(strategy)
        {
        }

        void set_strategy(StrategyInterface *strategy)
        {
            strategy_ = strategy;
        }

        void execute() const
        {
            strategy_->execute();
        }
};
```

```
int main(int argc, char *argv[])
{
    ConcreteStrategyA concreteStrategyA;
    ConcreteStrategyB concreteStrategyB;
    ConcreteStrategyC concreteStrategyC;

    Context contextA(&concreteStrategyA);
    Context contextB(&concreteStrategyB);
    Context contextC(&concreteStrategyC);

    contextA.execute(); // output: "Called ConcreteStrategyA execute method"
    contextB.execute(); // output: "Called ConcreteStrategyB execute method"
    contextC.execute(); // output: "Called ConcreteStrategyC execute method"

    contextA.set_strategy(&concreteStrategyB);
    contextA.execute(); // output: "Called ConcreteStrategyB execute method"
    contextA.set_strategy(&concreteStrategyC);
    contextA.execute(); // output: "Called ConcreteStrategyC execute method"

    return 0;
}
```

Template Method

By defining a skeleton of an algorithm in an operation, deferring some steps to subclasses, the Template Method lets subclasses redefine certain steps of that algorithm without changing the algorithms structure.

Visitor

The Visitor Pattern will represent an operation to be performed on the elements of an object structure by letting you define a new operation without changing the classes of the elements on which it operates.

```
#include <string>
#include <iostream>
#include <vector>

using namespace std;

class Wheel;
class Engine;
class Body;
class Car;

// interface to all car 'parts'
struct CarElementVisitor
{
  virtual void visit(Wheel& wheel) const = 0;
  virtual void visit(Engine& engine) const = 0;
  virtual void visit(Body& body) const = 0;

  virtual void visitCar(Car& car) const = 0;
  virtual ~CarElementVisitor() {}
};

// interface to one part
struct CarElement
{
  virtual void accept(const CarElementVisitor& visitor) = 0;
  virtual ~CarElement() {}
};

// wheel element, there are four wheels with unique names
class Wheel : public CarElement
{
public:
  explicit Wheel(const string& name) :
    name_(name)
  {
  }
  const string& getName() const
  {
    return name_;
  }
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
private:
    string name_;
};

// engine
class Engine : public CarElement
{
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
```

```
};

// body
class Body : public CarElement
{
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
};

// car, all car elements(parts) together
class Car
{
public:
  vector<CarElement*>& getElements()
  {
    return elements_;
  }
  Car()
  {
    // assume that neither push_back nor Wheel(const string&) may throw
    elements_.push_back( new Wheel("front left") );
    elements_.push_back( new Wheel("front right") );
    elements_.push_back( new Wheel("back left") );
    elements_.push_back( new Wheel("back right") );
    elements_.push_back( new Body() );
    elements_.push_back( new Engine() );
  }
  ~Car()
  {
    for(vector<CarElement*>::iterator it = elements_.begin();
      it != elements_.end(); ++it)
    {
      delete *it;
    }
  }
private:
  vector<CarElement*> elements_;
};

// PrintVisitor and DoVisitor show by using a different implementation the Car
class is unchanged
// even though the algorithm is different in PrintVisitor and DoVisitor.
class CarElementPrintVisitor : public CarElementVisitor
{
public:
  void visit(Wheel& wheel) const
  {
    cout << "Visiting " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Visiting engine" << endl;
  }
  void visit(Body& body) const
  {
    cout << "Visiting body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Visiting car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);        // this issues the callback i.e. to this from
 the element
    }
    cout << "Visited car" << endl;
  }
};

class CarElementDoVisitor : public CarElementVisitor
{
public:
  // these are specific implementations added to the original object without
modifying the original struct
  void visit(Wheel& wheel) const
  {
    cout << "Kicking my " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Starting my engine" << endl;
  }
  void visit(Body& body) const
  {
    cout << "Moving my body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Starting my car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);        // this issues the callback i.e. to this from
 the element
    }
    cout << "Stopped car" << endl;
  }
};

int main()
{
  Car car;
  CarElementPrintVisitor printVisitor;
  CarElementDoVisitor doVisitor;

  printVisitor.visitCar(car);
  doVisitor.visitCar(car);

  return 0;
}
```

Model-View-Controller (MVC)

A pattern often used by applications that need the ability to maintain multiple views of the same data. The model-view-controller pattern was until recently a very common pattern especially for graphic user interlace programming, it splits the code in 3 pieces. The model, the view, and the controller.

The Model is the actual data representation (for example, Array vs Linked List) or other objects representing a database. The View is an interface to reading the model or a fat client GUI. The Controller provides the interface of changing or modifying the data, and then selecting the "Next Best View" (NBV).

Newcomers will probably see this "MVC" model as wasteful, mainly because you are working with many extra objects at runtime, when it seems like one giant object will do. But the secret to the MVC pattern is not writing the code, but in maintaining it, and allowing people to modify the code without changing much else. Also, keep in mind, that different developers have different strengths and weaknesses, so team building around MVC is easier. Imagine a View Team that is responsible for great views, a Model Team that knows a lot about data, and a Controller Team that see the big picture of application flow, handing requests, working with the model, and selecting the most appropriate next view for that client.

TODO
Erm, someone please come up with a better example than the following... I can not think of any

For example: A naive central database can be organized using only a "model", for example, a straight array. However, later on, it may be more applicable to use a linked list. All array accesses will have to be remade into their respective Linked List form (for example, you would change myarray[5] into mylist.at(5) or whatever is equivalent in the language you use).

Well, if we followed the MVC pattern, the central database would be accessed using some sort of a function, for example, myarray.at(5). If we change the model from an array to a linked list, all we have to do is change the view with the model, and the whole program is changed. Keep the interface the same but change the underpinnings of it. This would allow us to make optimizations more freely and quickly than before.

One of the great advantages of the Model-View-Controller Pattern is obviously the ability to reuse the application's logic (which is implemented in the model) when implementing a different view. A good example is found in web development, where a common task is to implement an external API inside of an existing piece of software. If the MVC pattern has cleanly been followed, this only requires modification to the controller, which can have the ability to render different types of views dependent on the content type requested by the user agent.