Design Patterns

2008년 여름
박경신

Design Pattern Categories

- Creational Patterns
  - Initializing and configuring classes and objects
- Structural Patterns
  - Decoupling the interface and implementation of classes and objects
- Behavioral Patterns
  - Dynamic interactions among societies of classes and objects

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

Some Useful Programming Patterns

- Singleton – 단 하나의 인스턴스가 필요할 때
- Strategy – 알고리즘(변하는 요소)을 바꾸는 것
- Factory – 물건을 생산하는 것
- Composite – 내용물과 그릇을 동일시
- Decorator – 객체와 포장을 동일시
- Flyweight – 작은 객체를 공유
- Visitor – 구조 안을 돌아다니며 일하는 것
- Observer – 상태의 변화를 통지하는 것

Singleton

- 프로그램 실행 시
  - 하나의 클래스에 대한 인스턴스가 보통 여러 개 생성된다.
- 반드시 하나의 인스턴스만 생성되어야 하는 클래스도 있다.
  - 예: 컴퓨터 자체를 표현한 클래스, 윈도 시스템을 표현한 클래스
- 이 경우, 프로그래머가 new MyClass( )를 한번만 실행하면 된다.
- 그러나, 1개의 인스턴스만 생성되도록 프로그램에 표현하고 싶다면, Singleton 패턴을 사용한다.

Singleton

```
// declaration
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
// implementation
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance() {
    if (_instance == 0) {
            instance = new Singleton;
    }
    return _instance;
}
```

Singleton
Instance : Singleton
Singleton() Instance() : Singleton

Singleton

- Singleton 클래스
  - private static Singleton* _instance
    - Singleton 클래스의 인스턴스를 생성한다.
    - Static 이므로, Singleton 클래스를 로드할 때 한번만 실행된다.
    - private이므로, 외부에서 접근하지 못한다.
  - protected Singleton( ) 생성자
    - protected 이므로, 외부에서 new Singleton( )을 호출하지 못한다.
  - public static Singleton* Instance( )
    - Singleton 클래스의 유일한 하나의 인스턴스를 얻을 때 사용하는 메소드이다.

Strategy

- 전략
  - 적과 싸울 때의 책략
  - 군대를 움직일 때의 작전
  - 프로그래밍에서는 '알고리즘'
- Strategy 패턴
  - 알고리즘을 구현한 부분이 모두 교환 가능하도록 함
  - 알고리즘(전략, 작전, 책략)을 교체해서 동일한 문제를 다른 방법으로 해결하는 패턴

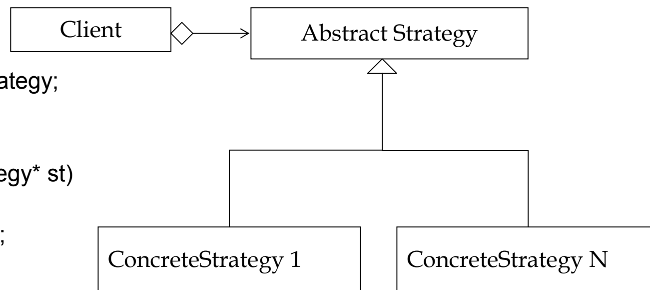
Strategy

- 다음은 a quick and dirty solution for creating an update routine that recalculates all AI를 보여주고 있다.

```
void recalc_AI()
{
    switch (state) {
        case FIGHT: recalc_fight(); break;
        case ESCAPE: recalc_escape(); break;
        case IDLE: recalc_idle(); break;
        case PATROL: recalc_patrol(); break;
    }
}
```

Strategy

```
class soldier {
public:
    soldier (strategy *);
    void recalc_AI();
    void change_strategy(strategy *);
private:
    point pos;
    float yaw;
    strategy* _thestrategy;
};

soldier::soldier(strategy* st)
{
    _thestrategy = st;
}
```



Strategy

```
void soldier::recalc_AI()
{
    _thestrategy->recalc_strategy(pos, yaw);
}

void soldier::change_strategy(strategy *st)
{
    _thestrategy = st;
}

class Strategy {
public:
    virtual int recalc_strategy(point, float) = 0;
protected:
    Strategy();
};
```


Strategy

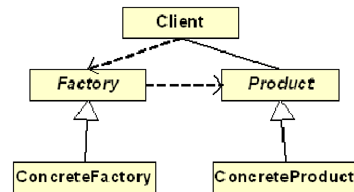
```
class FlightStrategy : public Strategy {
public:
    Strategy();
    virtual int recalc_strategy(point, float);
};

class IdleStrategy : public Strategy {
public:
    Strategy();
    virtual int recalc_strategy(point, float);
};

Soldier* s1 = new Soldier(new IdleStrategy);
s1.recalc_AI();
s1.change_strategy(new FlightStrategy);
s1.recalc_AI();
```


Factory

- Factory Method 패턴
  - Template Method를 변형한 패턴
  - 인스턴스 만드는 방법은 상위 클래스에서 결정하고
  - 인스턴스를 실제로 생성하는 일은 하위 클래스에서 결정한다.
  - '구체적인 제품 생성'을 '공장(Factory)'을 통해서 한다.

Factory

```
class Texture {};
class Mesh {};
class Item {};

class Factory {
public:
    Texture *CreateTexture();
    Mesh *CreateMesh();
    Item *CreateItem();
}

Texture* Factory::CreateTexture()
{
    return new Texture;
}
```



Factory

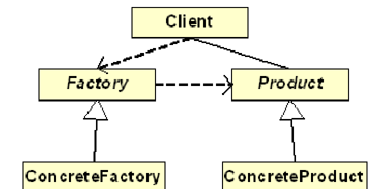
```
Mesh* Factory::CreateMesh()
{
    return new Mesh;
}

Item* Factory::CreateItem()
{
    return new Item;
}

Factory F;
Texture *t = F.CreateTexture();
```


Abstract Factory

- ‘추상적(abstract)’
  - 구체적으로 어떻게 구현되는지는 생각하지 않고 인터페이스(API)만을 주목하고 있는 상태
  - 예: 추상 메소드(abstract method)
    - 메소드의 본체는 쓰여져 있지 않고, 이름과 시그너처(인자의 형과 갯수, 반환경)만 정해져 있는 메소드
- 추상적인 공장에서 추상적인 부품을 조립하여 추상적인 제품을 만든다.
  - 하위 클래스에서 구체적인 구현을 수행한다.

Abstract Factory

```
class Product {};
class Texture: public Product() {};
class Mesh : public Product() {};
class Item: public Product() {};

typedef int ProductId;

#define TEXTURE 0
#define MESH 1
#define ITEM 2

class AbstractFactory {
public:
    Product* Create(ProductID);
}
```



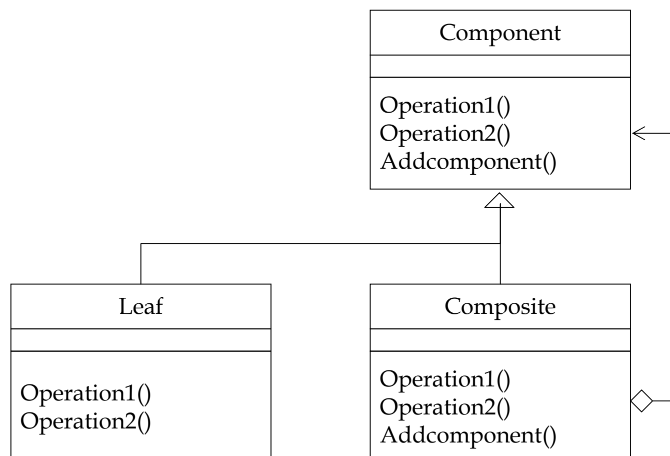
Abstract Factory

```
Product* AbstractFactory::Create(ProductID id)
{
    switch(id) {
            case TEXTURE return new Texture; break;
            case MESH return new Mesh; break;
            case ITEM return new Item; break;
    }
}
```


Composite

- 재귀적인 구조
  - 그릇 안에 내용물을 넣을 수도 있고, 작은 그릇을 넣을 수도 있다. 작은 그릇 안에는 더 작은 그릇이나 내용물을 넣을 수도 있다.
- Composite 패턴
  - 그릇과 내용물을 동일시해서 재귀적인 구조를 만드는 패턴
  - Composite이란 혼합물, 복합물이라는 뜻

Composite



Composite

- Leaf(잎사귀)
  - '내용물'에 해당되는 역할이다
  - 이 안에는 다른 것을 넣을 수 없다
- Composite(복합체)
  - '그릇'을 나타내는 역할이다
  - Leaf와 Composite을 넣을 수 있다
- Component
  - Leaf 역할과 Composite 역할을 동일시 하기 위한 역할이다.
  - Leaf와 Composite의 상위 클래스로 구현된다.
- Client(의뢰자)
  - Main 에서 의뢰하는 역할이다.

Composite

```
class Item {
private:
    string name;
public:
    Item(string s) : name(s) {}
    string GetName() const {return name;}
    virtual int GetSize() = 0;
    virtual void print (string prefix) const
    {
        cout << prefix << GetName() << endl;
    }
};
```


Composite

```
class File : public Item {
private:
    int size;
public:
    File(string n, int s) : Item(n), size(s) {}
    virtual int GetSize()  { return size; }
};

class Folder : public Item {
private:
    vector<Item*> items;
public:
    Folder(string n) : Item(n) {}
    void Add(Item* p)  { items.push_back(p); }
```


Composite

```
    virtual int GetSize()
    {
        int sz = 0;
        for (int i = 0; I < items.size(); ++i) {
            sz += items[i]->GetSize();
        }
        return sz;
    }
    virtual void print(string prefix) const
    {
        cout << prefix << GetName() << endl;
        prefix = prefix + string("\t");
        for (int i = 0; I < items.size(); ++i) {
            items[i]->print(prefix);
        }
    }
};
```

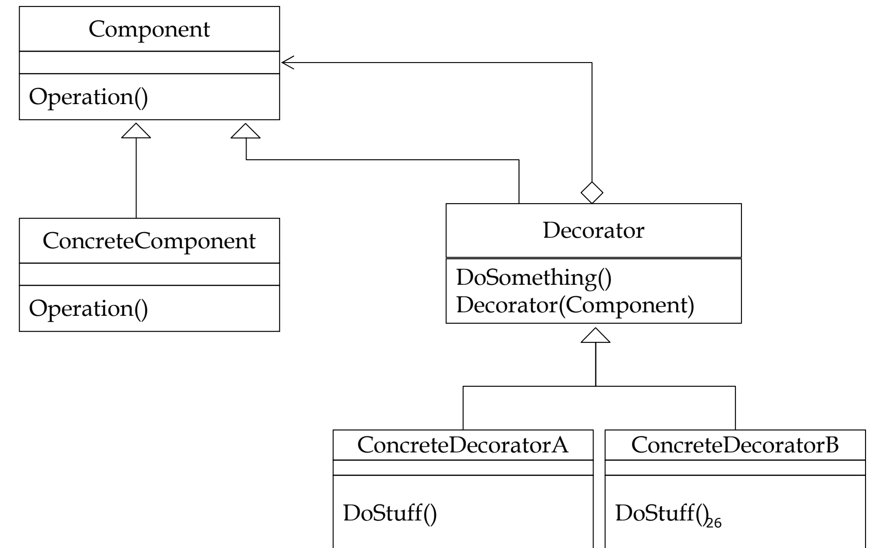

Composite

```
void main()
{
    Folder* R = new Folder("ROOT");
    Folder* A = new Folder("A");
    Folder* B = new Folder("B");
    R->Add(A);
    R->Add(B);
    R->Add(new File("a.txt", 10));
    A->Add(new File("b.txt", 20));
    B->Add(new File("c.txt", 30));
    cout << R->GetSize() << endl;
    R->print("");
}
```

Decorator

- 예: 케이크
  - 스펀지 케이크
  - 딸기를 얹으면 => 스트로베리 케이크
  - 화이트 초콜릿과 양초를 얹으면 => 생일 케이크
- Decorator 패턴
  - 중심이 되는 객체에, 장식과 같은 부가적인 기능을 하나씩 입혀서 좀 더 목적에 어울리는 객체를 만들자.
  - 내용물을 변경하지 않고, 새로운 장식을 계속해서 부착할 수 있다.
  - 포장되는 대상을 변경하지 않고, 기능을 추가할 수 있다.

Decorator



Decorator

```
class Widget {
public:
    virtual void draw() = 0;
    virtual ~Widget() {}
};

class TextField : public Widget {
private:
    int _width, _height;
public:
    TextField(int w, int h) { _width  = w; _height = h; }
    void draw() { cout << “TextField: “ << width << “, “ << height << ‘\n’; }
};
```


Decorator

```
class Decorator : public Widget {
private:
    Widget *_widget;
public:
    Decorator(Widget* w) { _widget = w; }
    ~Decorator() { delete _widget; }
    void draw() { _widget->draw(); }
};
class BorderDecorator : public Decorator { // ConcreteDecoratorA
public:
    BorderDecorator(Widget* w) : Decorator (w) {}
    void draw() { Decorator::draw(); cout << “ BorderDecorator “ << endl; }
};
class ScrollDecorator : public Decorator { // ConcreteDecoratorB
public:
    ScrollDecorator(Widget* w) : Decorator (w) {}
    void draw() { Decorator::draw(); cout << “ ScrollDecorator “ << endl; }
};
```

Decorator

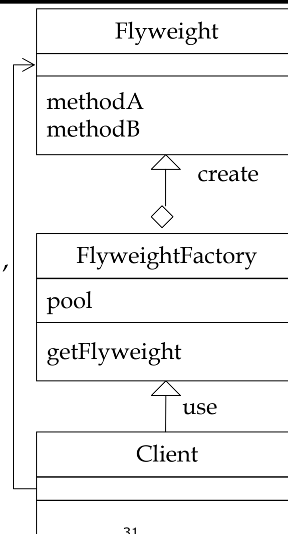
```
int main(void)
{
    Widget* aWidget = new BorderDecorator(new ScrollDecorator
                                                        (new TextField(80, 24)));
    aWidget->draw();
    delete aWidget;
}
```


Flyweight

- 가볍다 == 메모리의 사용량이 적다.
- new Something( )이 실행되면, Something 클래스의 인스턴스를 보관하기 위한 메모리가 할당된다.
- Flyweight 패턴
  - 인스턴스를 가능한 한 공유시켜서, 쓸데없이 new를 하지 않도록 한다.
  - 인스턴스가 필요할 때 마다 new를 하는 것이 아니라, 이미 만들어져 있는 인스턴스를 이용할 수 있으면 공유하자.

Flyweight

- Flyweight(플라이급)
  - 평소대로 취급하면 프로그램이 무거워져서(메모리 많이 차지해서) 공유하는게 훨씬 좋은 역할
- FlyweightFactory(플라이급의 공장)
  - Flyweight 역할을 만드는 공장의 역할
  - 공장을 사용해서 Flyweight 역할을 만들면, 인스턴스가 공유된다.
- Client(의뢰자)
  - FlyweightFactory를 사용해서 Flyweight 역할을 만들어내서 이용하는 역할



Flyweight

```
class BigChar {
private:
    char c;
    BigChar (char a) : c(a) {}
public:
    void Show() const { cout << "[[[[ " << c << " ]]]]" << endl; }
    friend class BigCharFactory;
};

class BigCharFactory {       // singleton & factory
private:
    map<char, BigChar*> pool;
    BigCharFactory() {}
public:
    static BigCharFactory & GetInstance() {
            static BigCharFactory _instance;
            return _instance;
    }
```

Flyweight

```
BigChar* CreateBigChar( char c )
 {
        if ( pool[c] == 0 )
                pool[c] = new BigChar(c);
        return pool[c];
 }
 void Reset()  // 또는 clean()
 {
        // map에 있는 모든 객체를 제거한다.
 }
 void Remove( char c )
 {
 }
};
```

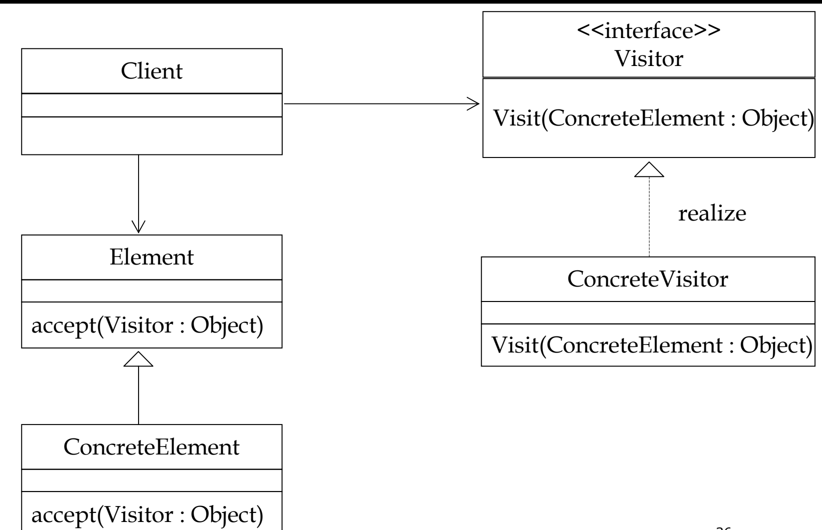

Flyweight

```
// Helper Macro 함수
BigChar* CreateChar( char c )
{
    return BigCharFactory::GetInstance().CreateBigChar(c);
}
void foo()
{
    BigChar*  p = CreateChar('c');
    cout << p << endl;
}
void main()
{
    BigChar* p = CreateChar('c');
    cout << p << endl;
    foo();
}
```

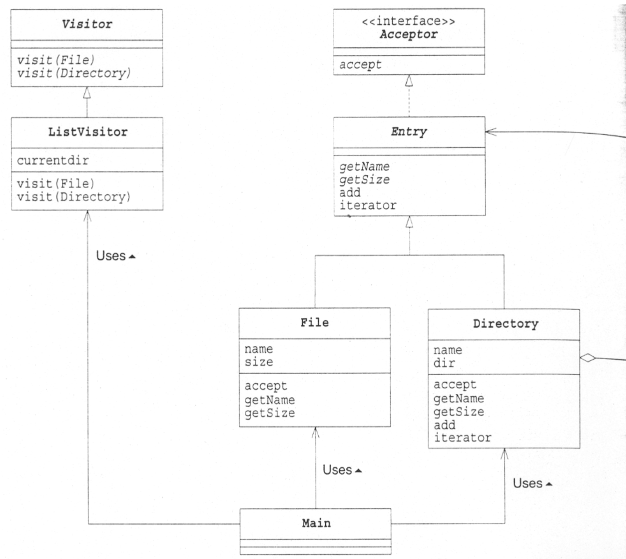

Visitor

- 데이터 구조 안에 저장되어 있는 많은 요소에 대해서, 무언가 "처리"해 나가고자 한다.
  - 얼핏 생각하면, 데이터 구조를 나타내고 있는 클래스 안에 "처리"를 기술해야 한다고 생각할 것이다.
  - 그러나, "처리"가 여러 종류라고 한다면?
    - 새로운 처리가 필요해질 때마다, 데이터 구조를 나타내는 클래스를 수정해야 한다.
- 데이터 구조와 처리를 분리하자.
  - 데이터 구조를 돌아다니는 "방문자"를 정의해서, 이 방문자가 "처리"를 담당하도록 하자.
  - 새로운 처리를 추가하고 싶을 때는, 새로운 "방문자"를 만든다.
- 데이터 구조는, 문을 두드리는 "방문자"를 받아들이기만 하면 된다.

Visitor



Visitor



Visitor

```
class Visitor {
protected:
    Visitor();
public:
    virtual void VisitElementA(ElementA*);
    virtual void VisitElementB(ElementB*);
};

class Element {
protected:
    Element();
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
}
```


Visitor

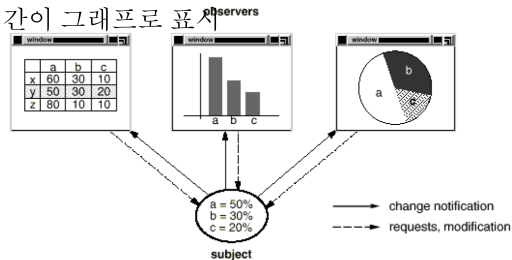
```
class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```


Visitor

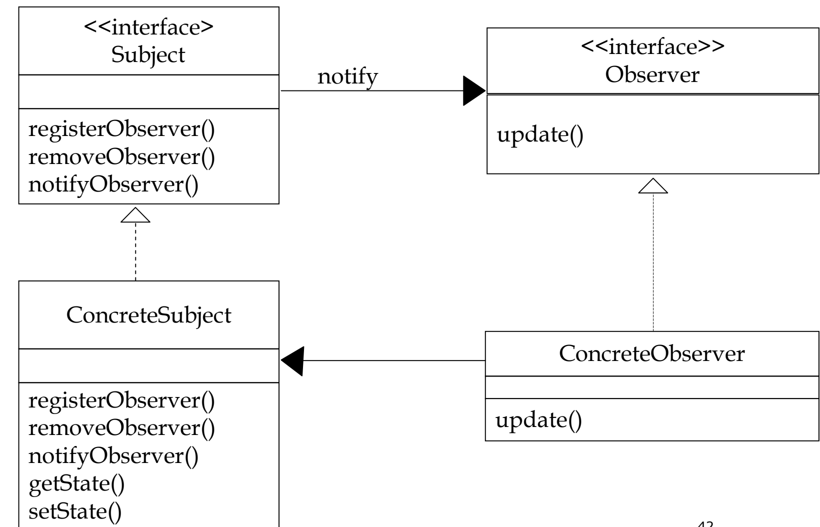
```
class CompositeElement : public Element {
private:
    List<Element*>* _children;
public:
    virtual void Accept(Visitor&);
};

void CompositeElement::Accept(Visitor& v) {
    ListIterator<Element*> i(_children);
    for (i.First(); !i.IsDone(); i.Next()) {
            i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}
```

Observer

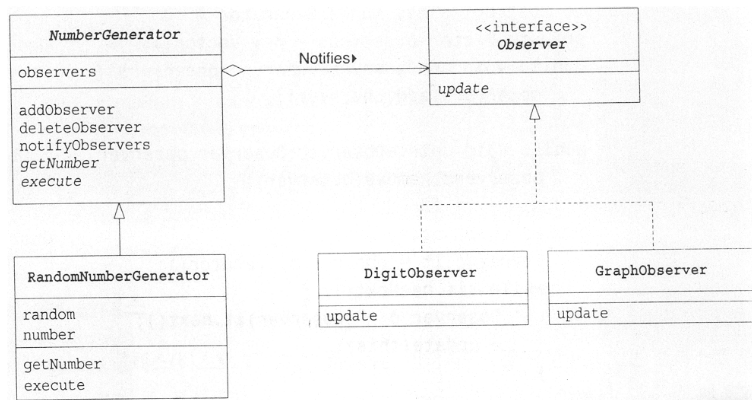
- Observer
  - 관찰 대상(Subject)의 상태(State)가 변하면, 관찰자(Observer)에게 통지(Notify)된다.
  - 객체의 상태 변화에 따른 처리를 기술할 때 유용하게 사용된다.
- 예: 수를 많이 생성하는 객체를 관찰자가 관찰해서, 그 값을 표시하는 프로그램
  - 관찰자의 종류에 따라 표시 방법이 다르다
  - DigitObserver: 값을 숫자로 표시
  - GraphObserver: 값을 간이 그래프로 표시



Observer



Observer



Observer

```
class Subject;
class Observer {
protected:
    Observer();
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
};

class Subject {
private:
    List<Observer*>* _observers;
protected:
    Subject();
public:
    virtual ~Subject();
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
};
```

Observer

```
void Subject::Attach(Observer* o) {
    _observers->Append(o);
}

void Subject::Detach(Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify() {
    ListIterator<Observer*> i(_observers);
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```


Observer

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();
    void Tick();
};

void ClockTimer::Tick() {
    // update internal time state
    Notify();
}
```


Observer

```
class DigitalClock : public Widget, public Observer {
private:
    ClockTimer* _subject;
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void Update(Subject *); // override Observer operation
    virtual void Draw();        // override Widget operation
};

DigitalClock::DigitalClock(ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}
```


Observer

```
DigitalClock::DigitalClock(ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

~DigitalClock::DigitalClock() {
    _subject->Detach(this);
}

void DigitalClock::Update(Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
    // digital clock draw
}
```

Observer

```
class AnalogClock : public Widget, public Observer {
private:
    ClockTimer* _subject;
public:
    AnalogClock(ClockTimer*);
    virtual ~AnalogClock();
    virtual void Update(Subject *); // override Observer operation
    virtual void Draw();       // override Widget operation
};

AnalogClock::AnalogClock(ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}
```


Observer

```
AnalogClock::AnalogClock(ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

~AnalogClock::AnalogClock() {
    _subject->Detach(this);
}

void AnalogClock::Update(Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
    // analog clock draw
}
```