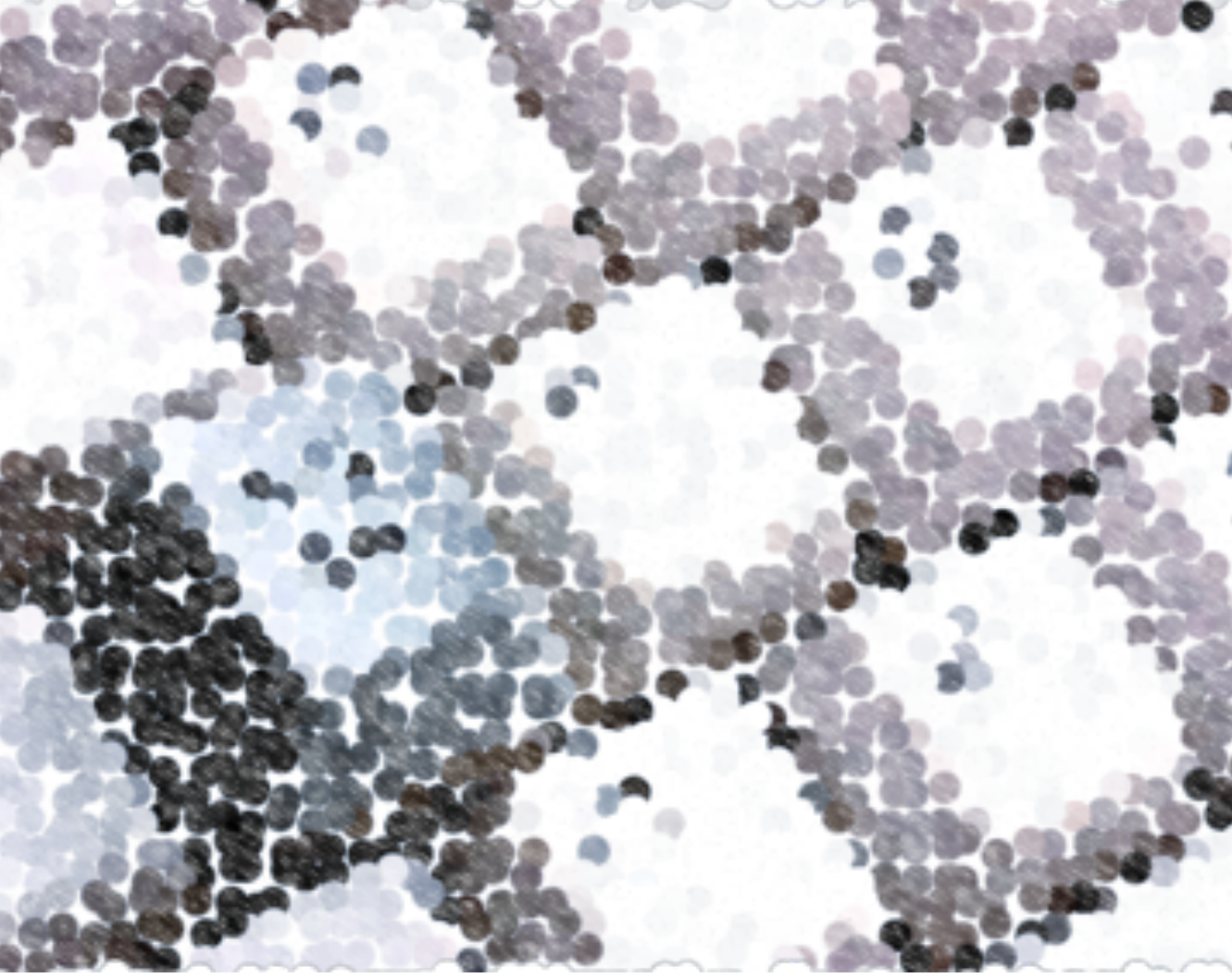


디자인 패턴
C++

윤찬식 | 아이오 교육센터

!m guru

과정 주요 내용

- 객체 지향 소프트웨어 설계 원칙
- GoF의 디자인 패턴 22가지
- C++ 주요 이디엄
- 다양한 C++ 오픈 소스에서 사용하는 다양한 설계 방법

강사 소개

윤찬식 강사
chansigi@ioacademy.co.kr
@Chansigi

참고 서적

- GoF의 디자인 패턴 개정판 (에릭 감마 외 3명, 김정아 역)
- Effective C++ 3판 (스콧 마이어스, 곽용재 역)
- More Effective C++ (스콧 마이어스, 곽용재 역)
- Effective STL (스콧 마이어스, 곽용재 역)
- C++ API 디자인 (마틴 레디, 천호민 역)
- Exceptional C++ (허브 셔터, 김동현 역)
- More Exceptional C++ (허브 셔터, 황은진 역)
- Exceptional C++ Style (허브 셔터, 류광 역)
- 일반적 프로그래밍과 STL (Matthew H. Austern, 류광 역)
- Modern C++ Design (안드레 알렉산드레스쿠, 이기형 역)
- C++ Network Programming(더글라스 C. 슈미트, 권태인 역)
- 실전 코드로 배우는 실용주의 디자인 패턴 (Allen Holub, 송지형 역)
- 패턴을 활용한 리팩터링 (조슈아 케리에브스키, 조성민 역)
- Effective Java 2판 (Joshua Bloch, 이병준 역)
- Java 언어로 배우는 디자인 패턴 입문 (YUKI HIROSHI)
- Java 언어로 배우는 리팩토링 입문(YUKI HIROSHI, 박건태 역)
- Refactoring (마틴 파울러, 윤성준 외 1명 역)

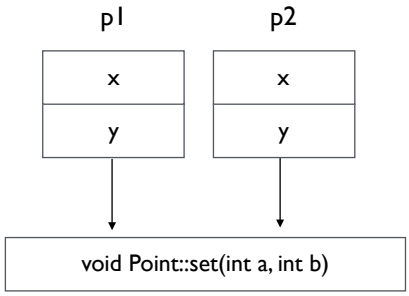
"가치를 설명하는 것은 광고가 아니라 제품이다."
"아이오 교육센터는 깊이 있고 좋은 강의를 최우선의 가치로 생각합니다."

thiscall

C++ 에서 멤버함수의 동작 원리를 이해하는 것은 중요합니다.

```
class Point {
  int x, y;

 public:
  void set(int a, int b) {
    x = a;
    y = b;
  }
};

int main() {
  Point p1, p2;
  p1.set(10, 20);
}
```

p1과 p2의 인스턴스는 x,y를 가진 형태로 메모리에 생성됩니다. 그때의 객체의 메모리 모습은 다음과 같습니다.



set 함수의 기계어 코드는 Point 객체가 모두 공유하도록 설계되어 있습니다. 그렇다면 p1.set(10, 20)의 호출에 있어서 멤버 함수는 어떤 객체를 대상으로 함수가 호출되었는지 알 수 있을까요?

```
int main() {
  Point p1, p2;
  p1.set(10, 20); // 이 순간의 정확한 원리를 생각해봅시다.
}
```

결국 p1.set은 set(&p1, 10, 20) 이라고 컴파일 됩니다. 멤버 함수의 호출은 첫번째 인자로 객체의 주소가 전달됨으로써 함수 내부에서 각 객체의 인스턴스 멤버 변수에 접근 가능합니다.

또한 클래스 내부의 함수 선언은 실제로는 다음과 같이 해석 됩니다.

```
void set(int a, int b) -> void set(Point* this, int a, int b)
{
  x = a;            -> this->x = a;
  y = b;            -> this->y = b;
}
```

정적 멤버 함수의 원리도 알아봅시다.

```
class Point {
  int x, y;

 public:
  static void foo(int a) { x = a; }
};

int main() { Point::foo(10); }
```

정적 멤버 함수는 인스턴스 생성 없이도 호출 가능하기 때문에, 결국 어떤 객체에 대해서 호출되었는지 구분할 필요가 없습니다. 즉 정적 멤버 함수 foo는 this가 필요하지 않습니다. 따라서 x=a는 this->x=a 로 해석될 것이고, 컴파일 에러가 발생합니다.

위의 원리를 잘 기억해놓고 아래 설계의 의도에 대해서 생각해봅시다.

```
class Sample {
  int data;

 protected:
  int on_foo() {
    cout << "foo" << endl;
    return data;
  }

 public:
  int foo() { return this ? on_foo() : 0; }
};
```

foo() 함수의 설계 의도는 무엇일까요? this가 널인 경우가 가능할까요?

```
int main() {
  // 메모리 할당에 실패해서 객체가 0이 되었다.
  Sample* p = 0;
  p->foo();
}
```

Sample 객체를 new 연산자를 통해 초기화 하는 것이 실패해서 객체 포인터가 NULL이 되었다고 가정해봅시다. p->foo()는 foo(&p) 로 컴파일러에 의해 번역될 것이고, 결국 foo의 첫번째 인자인 this로 전달됩니다. 즉 널 객체에 대해서 foo()가 호출될 경우, 잘못된 널 참조에 의해 프로세스가 종료되는 것을 방지하는 기법입니다. 위의 기법은 오픈소스 및 상용 라이브러리에서 널리 사용되었습니다.

```
// CWnd
_AFXWIN_INLINE CWnd::operator HWND() const
      { return this == NULL ? NULL : m_hWnd; }
_AFXWIN_INLINE HWND CWnd::GetSafeHwnd() const
      { return this == NULL ? NULL : m_hWnd; }
```

MFC(CWnd)

```
class IInterface : public virtual RefBase
{
public:
            IInterface();
            sp<IBinder>         asBinder();

protected:
    virtual                     ~IInterface();
    virtual IBinder*            onAsBinder() = 0;
};

sp<IBinder> IInterface::asBinder()
{
    return this ? onAsBinder() : NULL;
}
```

AOSP(IInterface.h / IInterface.cpp) - android-5.1.1_r9

하지만 널 객체에 대한 멤버 함수의 호출은 C++ 표준에서 정의하고 있지 않습니다. 많은 오픈 소스 및 라이브러리 내부에서 컴파일러의 구현에 의존한 코드를 작성하였던 것입니다. 따라서 위의 기법은 절대 사용하면 안됩니다. 최신 안드로이드 소스에서도 결국 위의 코드를 제거하고, 정적 멤버 함수를 이용한 방법으로 수정되었습니다.

```
// static
sp<IBinder> IInterface::asBinder(const IInterface* iface)
{
    if (iface == NULL) return NULL;
    return const_cast<IInterface*>(iface)->onAsBinder();
}
```

AOSP(IInterface.cpp) - android-l-preview_r2

핵심

- 멤버 함수 호출 원리
- 정적 멤버 함수 호출 원리
- 널 객체와 함수 호출

Thread

스레드를 캡슐화 해봅시다.

```
DWORD __stdcall foo(void* p) { return 0; }

int main() {
  CreateThread(0, 0, foo, "A", 0, 0);
  getchar();
}
```

Windows

```
void* foo(void* p) { return 0; }

int main() {
  pthread_t h;
  pthread_create(&h, 0, foo, (void*)"A");
  getchar();
}
```

Linux

하지만 C++98/03에서는 스레드를 만드는 방법을 제공하고 있지 않기 때문에, OS에서 제공하는 API를 사용해야 합니다. 이후의 예제는 Windows를 기준으로 진행하겠습니다.
운영체제에서 제공하는 API는 결국 C언어 함수로 구현되어 있습니다. 따라서 스레드를 생성하기 위해 필요한 절차와 각 함수 인자의 의미를 이해해야 합니다.

스레드를 추상화기 위해서는 CreateThread의 기본적인 사용법을 이해해야 합니다.

CreateThread(0, 0, **foo**, **"A"**, 0, 0);

foo : 스레드 생성후 수행할 함수의 주소
"A" : 스레드 시작 함수에 전달할 인자

하지만 스레드를 클래스로 추상화해서 라이브러리로 제공하면 사용자는 쉽게 스레드를 생성 가능합니다.

```
class MyThread : public Thread {
 public:
  virtual void threadLoop() { cout << "myThread" << endl; }
};
```

```
int main() {
  MyThread t;
  t.run(); // 이 순간 스레드가 생성되어 가상함수인 threadLoop()를 수행해야 합니다.

  getchar();
}
```

결국 Thread 클래스의 구현은 다음과 같습니다.

```
class Thread {
 public:
  void run() { CreateThread(0, 0, _threadLoop, this, 0, 0); }

  static DWORD __stdcall _threadLoop(void* user) {
    Thread* const self = static_cast<Thread*>(user);

    self->threadLoop();
    return 0;
  }

  virtual bool threadLoop() { return false; }
};
```

위의 코드에서 CreateThread의 세번째 인자로 전달되는 _threadLoop 함수가 정적 멤버 함수인 이유를 정확하게 이해해야 합니다. 정적 멤버 함수는 this가 없기 때문에 일반 함수의 시그니처와 동일합니다. 따라서 C언어 함수에서 함수 포인터를 전달해야 한다면, 정적 멤버 함수 포인터만 사용 가능합니다.

정적 멤버 함수 _threadLoop 안에서 가상 함수인 threadLoop를 호출하는 것이 불가능합니다. 왜냐하면 threadLoop()의 호출은 결국 this->threadLoop()로 해석되지만, 정적 멤버 함수인 _threadLoop 안에는 this가 존재하지 않습니다. 따라서 CreateThread의 네번째 인자로 멤버 함수인 run을 호출하는 시점에 유효한 this를 전달하여 Thread*의 타입으로 캐스팅해서 사용합니다.

실제로 안드로이드 프레임워크 내부의 Thread 클래스도 유사한 방법을 사용하고 있습니다.

```
class Thread : virtual public RefBase
{
public:
    virtual status_t    run(    const char* name = 0,
                                int32_t priority = PRIORITY_DEFAULT,
                                size_t stack = 0);
    ...
private:
    virtual bool        threadLoop() = 0;
    static  int             _threadLoop(void* user);
};
```

AOSP - Thread.h

```
int Thread::_threadLoop(void* user) {
  Thread* const self = static_cast<Thread*>(user);
  ...
  do {
    bool result;
    if (first) {
      ...
      if (result && !self->exitPending()) {
        result = self->threadLoop();
      }
    } else {
      result = self->threadLoop();
    }

    ...
  } while (strong != 0);
  ...
  return 0;
}
```

AOSP - Threads.cpp

chromium 오픈 소스 프로젝트 또한 다소 복잡하지만 동일한 방법을 통해 구현하고 있습니다.

```
bool Thread::Start() {
  Options options;
  return StartWithOptions(options);
}

bool Thread::StartWithOptions(const Options& options) {

  ...
  {
    AutoLock lock(thread_lock_);
    if (!PlatformThread::CreateWithPriority(options.stack_size,
                                            this, &thread_,
                                            options.priority)) {
      DLOG(ERROR) << "failed to create thread";
      message_loop_ = nullptr;
      return false;
    }
  }

  return true;
}
```

chromium - thread.cc

```
bool CreateThread(size_t stack_size,
                  bool joinable,
                  PlatformThread::Delegate* delegate,
                  PlatformThreadHandle* thread_handle,
                  ThreadPriority priority) {
  ...

  scoped_ptr<ThreadParams> params(new ThreadParams);
  params->delegate = delegate;
  params->joinable = joinable;
  params->priority = priority;

  pthread_t handle;
  int err = pthread_create(&handle, &attributes, ThreadFunc,
params.get());
  ...

  return success;
}

void* ThreadFunc(void* params) {
  base::InitOnThread();

  PlatformThread::Delegate* delegate = nullptr;

  {
    scoped_ptr<ThreadParams>
    thread_params(static_cast<ThreadParams*>(params));

    delegate = thread_params->delegate;
    ...
  }

  ...
  delegate->ThreadMain();

  ...
  base::TerminateOnThread();
  return NULL;
}
```

chromium - platform_thread_posix.cc

핵심

- 정적 멤버 함수와 일반 함수 포인터
- 정적 멤버 함수와 멤버 함수

Clock

타이머를 캡슐화 해봅시다.

```
void foo(int id)
{
    cout << "foo : " << id << endl;
}

int main()
{
    int n1 = IoSetTimer(1000, foo);
    int n2 = IoSetTimer(500, foo);

    IoProcessMessage();
}
```

IoSetTimer의 첫번째 인자로 ms를 지정하고, 두번째 인자로 시간이 만료되었을 경우 호출할 함수의 주소를 지정합니다. 위의 타이머는 자신만의 고유한 id를 리턴하게 되고, 사용자는 그 id로 타이머를 구분할수 있습니다.
이제부터 타이머를 클래스로 추상화해서 사용자가 인지하기 쉬운 문자열의 형태로 사용할 수 있도록 캡슐화해봅시다..

```
class Clock {
  string name;
 public:
  Clock(string s) : name(s) {}

  void start(int ms) {
    int id = IoSetTimer(ms, timerHandler);
  }

  static void timerHandler(int id) {
    cout << name << endl;
  }
};

int main() {
  Clock c1("AAA");
  Clock c2("\tBBB");

  c1.start(1000);
  c2.start(500);

  IoProcessMessage();
}
```

하지만 위의 코드에는 컴파일 에러가 발생합니다. 왜냐하면 IoSetTimer의 인자로 지정한 함수의 주소는 일반 멤버 함수가 아닌 정적 멤버 함수의 주소를 지정해야 합니다. 결국 정적 멤버 함수인 timerHandler 안에서 멤버 변수인 data에 접근할 수 없기 때문입니다. 하지만 CreateThread의 함수와는 달리 IoSetTimer 함수는 this를 전달할 수 있는 방법이 존재하지 않습니다. 결국 각 타이머가 만들어질때마다 리턴되는 타이머의 id에 따른 객체의 주소를 자료구조에 저장하고, timerHandler의 인자로 오는 id를 통해서 해당하는 객체의 주소, 즉 this를 얻어와야 합니다.

```
class Clock {
  string name;

  static map<int, Clock*> this_map;

 public:
  Clock(string s) : name(s) {}

  void start(int ms) {
    int id = IoSetTimer(ms, timerHandler);
    this_map[id] = this;
  }

  static void timerHandler(int id) {
    Clock* self = this_map[id];
    cout << self->name << endl;
  }
};

map<int, Clock*> Clock::this_map;
```

핵심

- 정적 멤버 함수와 일반 함수 포인터
- 객체의 주소를 저장하고, 얻어오는 방법.

Window

GUI 클래스 라이브러리의 설계에 대해 이야기 해봅시다.

```
int foo(int handle, int msg, int param1, int param2) {
  switch (msg) {
    case WM_LBUTTONDOWN:
      cout << "LButtonDown" << endl;
      break;
    case WM_KEYDOWN:
      cout << "KeyDown" << endl;
      break;
  }
  return 0;
}

int main() {
  int h1 = IoMakeWindow(foo);
  int h2 = IoMakeWindow(foo);

  IoProcessMessage();
}
```

IoMakeWindow 함수를 이용해 윈도우가 메세지를 처리하는 함수의 주소를 전달해주면, 윈도우를 생성할 수 있습니다. 메세지 처리 함수에서는 msg의 종류에 따라 분기하여 각 메세지를 처리하는 형태로 코드를 작성해야 합니다.

```
class MyWindow : public Window {
 public:
  virtual void OnLButtonDown() { cout << "LBUTTON" << endl; }
};

int main() {
  MyWindow w;
  w.Create();
  IoProcessMessage();
}
```

하지만 Window가 클래스 라이브러리로 제공된다면 사용자는 메세지 처리 과정에 대한 이해 없이도 쉽게 윈도우를 생성할 수 있습니다. 또한 자신이 처리하고자 하는 메세지에 해당하는 가상함수를 재정의하는 것으로 어렵지 않게 윈도우의 다양한 이벤트를 처리하는 것이 가능합니다.

결국 클래스 라이브러리의 Window 클래스는 다음과 같이 설계할 수 있습니다.

```
class Window {
  int handle;
  static map<int, Window*> this_map;

 public:
  void Create() {
    handle = IoMakeWindow(foo);
    this_map[handle] = this;
  }

  static int foo(int h, int msg, int param1, int param2) {
    Window* pThis = this_map[h];

    switch (msg) {
      case WM_LBUTTONDOWN:
        pThis->OnLButtonDown();
        break;
      case WM_KEYDOWN:
        pThis->OnKeyDown();
        break;
    }
    return 0;
  }

  virtual void OnLButtonDown() {}
  virtual void OnKeyDown() {}
};
map<int, Window*> Window::this_map;
```

위의 코드를 통해 기억해야 하는 사항은 다음과 같습니다.
첫번째. IoMakeWindow에 지정하는 메세지 처리 함수 foo가 static이어야 하는 이유를 이해해야 합니다.
두번째. 생성된 윈도우에 해당하는 객체의 주소를 handle을 키 값으로 하는 map에 저장해서, 정적 멤버 함수에 전달하는 방법과 이유를 이해해야 합니다.

핵심

- 정적 멤버 함수와 일반 함수 포인터
- 객체의 주소를 저장하고, 얻어오는 방법.

어댑터(Adapter)

클라이언트가 스택이라는 컨테이너 클래스를 요구합니다. 만약 우리에게 잘 설계된 연결 리스트가 있었다면 두 가지 접근 방식을 통해 클라이언트의 요구를 만족할 수 있습니다. 첫번째 방법은 새롭게 스택 클래스를 만들어서 제공하는 것입니다. 두번째 방법은 연결 리스트의 한쪽을 이용해서 데이터를 삽입하거나 삭제하면 어렵지 않게 스택을 제공할 수 있으니, 기존의 잘 설계된 리스트를 재사용하는 방법입니다. 당연히 잘 설계된 연결 리스트를 사용하는 것이 좋을 것 입니다. 새롭게 스택을 만들고 검증하는 작업은 기존의 클래스의 인터페이스를 변경해서 보여주는 것보다 훨씬 어렵고 많은 시간이 필요합니다.
이처럼 기존 클래스의 인터페이스를 변경해서 사용자가 요구하는 새로운 클래스처럼 보이게하는 패턴을 GoF의 디자인 패턴에서는 어댑터(Adapter) 패턴이라고 부릅니다.

GoF의 어댑터 패턴의 의도는 다음과 같습니다.
"클래스의 인터페이스를 사용자가 기대하는 다른 인터페이스로 변환하는 패턴으로, 호환성이 없는 인터페이스 때문에 함께 동작할 수 없는 클래스들이 함께 작동하도록 해 줍니다."

```
template <typename T>
class stack : public list<T> {
 public:
  inline void push(const T& a) { push_back(a); }
  inline void pop() { pop_back(); }
  inline T& top() { return back(); }
};
```

우리가 설계한 스택은 잘 설계된 리스트를 상속이라는 문법을 통해 재사용하고 있습니다. 하지만 위의 설계에는 문제가 있습니다. 스택을 사용하고자 하는 사용자는 스택의 함수 뿐 아니라 부모 클래스의 모든 인터페이스도 접근 가능합니다. 따라서 쉽게 스택을 깨뜨릴 수 있습니다. 클래스의 설계자는 클래스가 사용자에게 편의를 제공할 뿐 아니라 잘못 사용하지 못하도록 안전하게 만들어야 하는 책임이 있습니다. 스택은 부모의 구현과 인터페이스를 모두 물려받으면 안됩니다. 위의 문제를 해결하기 위해서는 C++ 접근 변경자를 통해 부모의 인터페이스에 대해서 은닉해야 합니다. private 상속은 부모로부터 구현은 물려받지만, 인터페이스는 물려받지 않겠다는 의도를 표현하는 C++의 기능입니다.

```
template <typename T>
class stack : private list<T> {
 public:
  inline void push(const T& a) { push_back(a); }
  inline void pop() { pop_back(); }
  inline T& top() { return back(); }
};
```

자바나 C#은 상속에 대해서 public 상속만을 지원하기 때문에, 이런 부분에 대해서는 추가적인 고려가 필요합니다. 결국 상속을 통한 재사용보다는 포함을 이용한 재사용을 이용하는 것이 안전한 설계가 됩니다. 하지만 안타깝게도 자바의 스택은 Vector<E>를 상속받은 형태로 어댑터를 구현한 형태로 스택을 구현하고 있을 뿐 아니라 많은 설계적 문제를 가지고 있습니다. 상속은 재사용에 있어서 편리함을 제공하지만, 잘못 사용되었을 경우 '깨지기 쉬운 기반 클래스' 문제의 원인이 됩니다. 기반 클래스는 기반 클래스만을 따로 떨어뜨려 놓고 안전하게 수정될 수 없으며, 모든 파생 클래스를 함께 살펴보고 테스트 해봐야 됩니다. 또한 기반 클래스를 사용하는 객체 뿐 아니라 파생 클래스를 사용해야 하는 클래스도 점검해 보아야 합니다. 단순히 기반 클래스를 수정했음에도 불구하고 전체 프로그램이 오작동할 수 있습니다. 결구 이것은 상속이 부모 클래스와 자식 클래스 간의 강한 결합 관계가 형성되기 때문입니다.

```
template <typename T, typename C = deque<T>>
class stack {
  C st;

 public:
  inline void push(const T& a) { st.push_back(a); }
  inline void pop() { st.pop_back(); }
  inline T& top() { return st.back(); }
};
```

실제 C++ STL의 스택이 설계된 방법입니다. 스택 어댑터라고 부릅니다. 템플릿 인자를 통해 사용자가 실제 내부적으로 사용할 컨테이너를 선택할 수 있도록 설계되어 있습니다. 또한 포함을 통해서 재사용을 구현하고 있기 때문에 상속으로 인한 불필요한 인터페이스 공개의 문제도 발생하지 않습니다.

핵심

- 어댑터 패턴
- private 상속의 철학

참고 자료

- GoF의 디자인 패턴: Adapter
- 실용주의 디자인 패턴: Chapter2. 인터페이스로 프로그래밍하기 그리고 몇 개의 생성패턴.
- 이펙티브 자바 2판 : 규칙 16. 계승하는 대신 구성하라.

도형 편집기로 배우는 객체지향 원리

도형을 추가하고 화면에 그리는 프로그램을 설계해 봅시다.

첫번째로 각각의 도형을 추상화하여 클래스로 관리하면 편리합니다.

```
class Rect {
 public:
  void draw() { cout << "Rect draw" << endl; }
};

class Circle {
 public:
  void draw() { cout << "Circle draw" << endl; }
};
```

하지만 도형을 각각의 독립적인 클래스로 설계하면, 컨테이너 같은 자료구조로 묶어서 관리할 수 없습니다. 서로 다른 클래스를 묶어서 관리하기 위해서는 결국 공통의 부모가 필요합니다. 모든 도형의 부모 클래스가 있다면, 모든 도형을 묶어서 관리할 수 있습니다.

```
class Shape {
 public:
  virtual ~Shape() {}
};

class Rect : public Shape {
 public:
  void draw() { cout << "Rect draw" << endl; }
};

class Circle : public Shape {
 public:
  void draw() { cout << "Circle draw" << endl; }
};
```

도형을 추가하고 화면에 그리는 기능에 대해서는 다음과 같이 간단하게 구현 가능합니다.

```
int main() {
  vector<Shape*> v;

  while (1) {
    int cmd;
    cin >> cmd;

    if (cmd == 1)
      v.push_back(new Rect);
    else if (cmd == 2)
      v.push_back(new Circle);
    else if (cmd == 9) {
      for (int i = 0; i < v.size(); i++) v[i]->draw();
    }
  }
}
```

하지만 위의 코드는 컴파일해보면 에러가 있습니다. 모든 자식 클래스가 구현하고 있는 draw()의 함수가 부모 클래스에 존재하지 않기 때문입니다. 모든 자식의 공통의 특징을 부모 포인터 또는 레퍼런스를 통해 사용하기 위해서는, 즉 다형성을 이용하기 위해서는 자식의 공통이 특징은 반드시 부모 클래스에도 있어야 합니다. 리스코프의 치환 원칙이라고 부릅니다.

```
class Shape {
 public:
  virtual ~Shape() {}
  virtual void draw() {}
};

class Rect : public Shape {
 public:
  void draw() { cout << "Rect draw" << endl; }
};

class Circle : public Shape {
 public:
  void draw() { cout << "Circle draw" << endl; }
};
```

또한 자식이 draw의 기능을 재정의하고 있기 때문에 가상 함수로 설계되어야 합니다. 스콧 마이어스의 이펙티브 C++에는 다음과 같은 격언이 있습니다. “가상 함수가 아닌 함수는 재정의 하지 말라.”

핵심

- 서로 다른 타입을 묶기 위해서는 공통의 부모가 필요하다.
- 리스코프 치환 원칙

참고 자료

- 이펙티브 C++ 3판 : 항목 36. 상속받은 비가상함수를 파생 클래스에서 재정의하는 것은 절대 금물
- Robert C. Martin, The Liskov Substitution Principle

도형 편집기로 배우는 객체지향 원리2

6장에서 완성한 도형 편집기에 몇 가지 기능을 더 추가해 봅시다.

첫번째 요구 사항은 도형 복제의 기능입니다.

```
int main() {
  vector<Shape*> v;

  while (1) {
    ...
    else if (cmd == 8) {
      cout << "몇번째 도형을 복사 할까요 >>";
      int k;
      cin >> k;
    } else if (cmd == 9) {
      for (int i = 0; i < v.size(); i++) {
        v[i]->draw();
      }
    }
}
```

복제하고 싶은 도형의 인덱스를 입력 받아서 복제하고자 합니다. 하지만 문제는 k번째 도형이 어떤 도형인지 알 수 있을까요? 도형의 타입을 정확히 알아야지만, 그 도형의 복사 생성자를 통해 복사를 할 수 있습니다. 하지만 벡터의 타입은 부모 타입으로 저장 되어 있기 때문에 절대 알 수 없습니다. 위의 문제를 해결하기 위한 가장 쉬운 해결책은 도형의 종류에 대한 추가적인 정보를 두고 각 도형이 어떤 도형인지 질의해서, 각 타입에 따른 도형의 복제를 구현하면 됩니다.

```
class Shape {
 public:
  int type;
  ...
};

class Rect : public Shape {
 public:
  Rect() { type = 1; }
  ...
};

class Circle : public Shape {
 public:
  Circle() { type = 2; }
  ...
};
```

```
int main() {
  vector<Shape*> v;

  while (1) {
    ... else if (cmd == 8) {
      cout << "몇번째 도형을 복사 할까요 >>";
      int k;
      cin >> k;

      switch (v[k]->type) {
        case 1:
          v.push_back(new Rect(*static_cast<Rect*>(v[i])));
          break;
        case 2:
          v.push_back(new Circle(*static_cast<Circle*>(v[i])));
          break;
     }

      ...
    }
}
```

위의 코드는 잘 동작하기는 하지만 문제가 있습니다. 각 도형의 종류에 따른 조건 분기 구문을 사용하고 있기 때문에, 만약 새로운 도형이 추가된다면 기존 코드는 계속해야 수정되어야 합니다. 새로운 기능이 추가될 때마다 기존 코드는 수정되어야 한다면 좋은 설계가 아닙니다. 객체 지향의 5대 원칙 중 하나인 개방 폐쇄의 원칙(Open Closed Principle) 입니다.
"모듈은 확장에는 열려 있어야 하고, 변경에는 닫혀 있어야 한다."

또한 마틴 파울러는 그의 저서 리팩토링에서 새로운 기능을 추가해야 하는데 프로그램의 코드가 새로운 기능을 추가하도록 구조화되어 있지 않은 경우에는 먼저 리팩토링을 해서 프로그램에 기능을 추가하게 쉽게 하고, 그 다음에 기능을 추가해야 한다고 했습니다. 그렇다면 어떻게 새로운 도형이 추가될때마다 도형 복제 기능에 대한 코드 수정이 발생하지 않도록 할 수 있을까요?

```
int main() {
  vector<Shape*> v;

  while (1) {
    ...
    else if (cmd == 8) {
      cout << "몇번째 도형을 복사 할까요 >>";
      int k;
      cin >> k;
    } else if (cmd == 9) {
      for (int i = 0; i < v.size(); i++) {
        v[i]->draw();
      }
    }
}
```

도형 편집기의 코드 중 새로운 도형이 추가되더라도 기존 코드에 수정이 발생하지 않는 곳이 있습니다. 바로 draw() 입니다. draw()는 가상함수이므로, v[i]에 어떤 도형이 있는지에 따라 그 도형의 draw

를 호출하는 다형성으로 구현되어 있습니다. 다형성은 OCP를 만족합니다. 리팩토링에는 “Replace type code with Polymorphism” 이라는 방법이 있습니다.

```
class Shape {
 ...
 public:
  virtual void draw() {}
  virtual Shape* clone() { return new Shape(*this); }
};

class Rect : public Shape {
 public:
  virtual void draw() { cout << "Rect Draw" << endl; }
  virtual Shape* clone() { return new Rect(*this); }
};

class Circle : public Shape {
 public:
  virtual void draw() { cout << "Circle Draw" << endl; }
  virtual Shape* clone() { return new Circle(*this); }
};


int main() {
  vector<Shape*> v;

  while (1) {
    ...
    else if (cmd == 8) {
      cout << "몇번째 도형을 복사 할까요 >>";
      int k;
      cin >> k;

      v.push_back(v[k]->clone());

      ...
    }
}
```

이제 v[k]에 어떤 타입의 도형이 있든지 상관없습니다. clone 이라는 가상함수를 통해 복제가 수행되기 때문에 실제 도형의 clone 함수가 호출됩니다. 그뿐 아니라 새로운 도형이 추가되어도 이제 복제에 대해서는 코드 수정이 필요하지 않습니다. 다형성은 OCP를 만족하기 때문입니다. 이렇게 다형성을 통해 객체 복제의 기능을 수행하는 것을 GoF의 디자인패턴에서는 프로토타입 패턴이라고 부릅니다.

GoF의 디자인 패턴에서 프토토타입의 의도는 다음과 같습니다.
“원형이 되는 인스턴스를 사용하여 생성할 객체의 종류를 명시하고, 이렇게 만든 견본을 복사해서 새로운 객체를 생성합니다.”

두번째 요구 사항은 draw()의 기능입니다. 화면에 그리는 기능이 다중 스레드에 안전하게 동작되도록 하기 위해서는 뮤텍스 등을 이용한 동기화가 필요합니다.

```
class Shape {
 ...
 protected:
  static mutex sLock;

 public:
  virtual void draw() {
    sLock.lock();
    cout << "Shape Draw" << endl;
    sLock.unlock();
  }
  virtual Shape* clone() { return new Shape(*this); }
};
```

하지만 draw는 가상함수이기 때문에, Shape의 draw뿐 아니라, 가상함수 draw를 재정의하고 있는 모든 클래스의 draw 함수에 대해서 동기화가 필요합니다.

```
class Shape {
 protected:
  static mutex sLock;

 public:
  virtual void draw() {
    sLock.lock();
    cout << "Shape Draw" << endl;
    sLock.unlock();
  }
  ...
};

class Rect : public Shape {
 public:
  virtual void draw() {
    sLock.lock();
    cout << "Rect Draw" << endl;
    sLock.unlock();
  }
  ...
};

class Circle : public Shape {
 public:
  virtual void draw() {
    sLock.lock();
    cout << "Circle Draw" << endl;
    sLock.unlock();
  }
  ...
};
```

하지만 위의 코드를 보면 중복된 코드(Duplicated Code)가 있습니다. 만약 이후버전에서 더이상 화면에 그리는 draw의 기능이 스레드 안정성이 필요하지 않다면, 동기화가 사용된 모든 부분을 찾아서 수

정해주어야 합니다. 결국 중복된 코드는 코드의 유지 보수를 힘들게 합니다. 리팩토링에서도 나쁜 코드의 첫번째 냄새로 중복된 코드를 말합니다. 그뿐 아니라 앤드류 헌트의 실용주의 프로그래머에는 "DRY(Don't Repeat Yourself) – 반복하지 말라"의 원칙이 있습니다.

draw의 함수 안에서 동기화의 정책은 변하지 않습니다. 하지만 도형에 따라 어떤 도형을 그릴지는 변해야 합니다. 변하는 것과 변하지 않는 것은 분리되어야 합니다.

```
class Shape {

 protected:
  static mutex sLock;

 public:
  void draw() {
    sLock.lock();
    drawImpl();
    sLock.unlock();
  }

 private:
  virtual void drawImpl() {}
};


class Rect : public Shape {
private:
  virtual void drawImpl() { cout << "Rect Draw" << endl; }
  ...
};

class Circle : public Shape {
private:
  virtual void drawImpl() { cout << "Circle Draw" << endl; }
  ...
};
```

변하지 않는 전체 알고리즘은 부모가 비가상함수로 제공하고 변해야 하는 부분만 가상함수화해서 자식의 변경할 수 있도록 했습니다. 사용자는 draw 함수를 호출할 것이고, 비가상함수이기 때문에 실제 도형의 타입에 상관없이 부모의 draw 함수가 호출됩니다. draw 함수에서 실제 도형을 그리는 drawImpl 가상함수가 호출되기 전에 잠금 동작을 수행하고, 도형을 그린 후에 잠금을 푸는 동작을 수행합니다. 이와 같은 설계 방법을 GoF의 디자인 패턴에서는 템플릿 메소드(Template Method) 패턴이라고 합니다. C++에서는 비가상 함수 인터페이스(NVI, Non Virtual Interface)라는 관용구로 많이 알려져 있습니다.

핵심

- 기능이 추가되어도 기존 코드는 수정되면 안된다.- Open Closed Principle
- 다형성은 OCP를 만족한다.
- Replace type code with polymorphism
- Prototype 패턴
- DRY(Don't repeat yourself)
- Template Method 패턴

참고 자료

- 이펙티브 C++ 3판 : 항목 35. 가상 함수 대신 쓸 것들도 생각해 두는 자세를 시시때때로 길러 두자.
- Robert C. Martin, The Open-Closed Principle
- GoF 디자인 패턴 : Prototype, Template Method
- Refactoring : 조건문을 다형성으로 바꾸기, 코드 속의 나쁜 냄새
- 실용주의 프로그래머: DRY(Don't Repeat Yourself)

인터페이스 기반 설계

```
class Camera
{
public:
    void take() { cout << "take picture with Camera" << endl; }
};

class Person
{
public:
    void takePicture(Camera* c) { c->take(); }
};

int main()
{
    Person p;
    Camera c;

    p.takePicture(&c);
}
```

위의 코드에서 Person 클래스는 사진을 찍는 기능을 Camera 객체를 이용하는 형태로 구현하고 있습니다. 객체 지향 설계에서 기능을 직접 구현하는 것이 아니라 이미 그 기능을 제공하는 다른 객체에게 위임하는 방법은 많이 사용됩니다.
만약 사진을 찍는 기능이 Camera 뿐 아니라 다른 클래스 SmartPhone 에도 제공된다면, Person 클래스는 기존 코드의 수정 없이 그 클래스를 이용 가능할까요? 불가능합니다. Person 클래스의 takePicture 함수는 Camera 라는 구체 클래스에 의존하고 있기 때문입니다. 이러한 형태의 설계는 결국 강한 결합(Tightly Coupling)을 만듭니다. 강한 결합은 교체가 불가능합니다.

교체 가능한 설계를 위해서는 먼저 카메라와 카메라 사용자 간의 규칙 즉 인터페이스를 먼저 설계해야 합니다.

```
struct ICamera {
  virtual void take() = 0;
  virtual ~ICamera() {}
};
```

이제 사용자는 진짜 카메라가 없어도 규칙대로만 사용하면 됩니다.

```
class Person {
 public:
  void takePicture(ICamera* p) { p->take(); }
};
```

모든 카메라 설계자는 반드시 약속된 ICamera 인터페이스를 구현해야 합니다.

```
class Camera : public ICamera {
 public:
  void take() { cout << "take picture with Camera" << endl; }
};

class SmartPhone : public ICamera {
 public:
  void take() { cout << "take picture with SP" << endl; }
};
```

```
int main() {
  Person p;
  SmartPhone sp;
  Camera c;

  p.takePicture(&sp);
  p.takePicture(&c);
}
```

이제 새로운 카메라가 추가된다고 하더라도 , Person의 클래스는 수정될 필요 없습니다. 하나의 클래스가 다른 클래스를 사용할 때 규칙을 정의한 기반 클래스(추상 클래스 또는 인터페이스)에 의존한다면 객체와 객체간에 약한 결합이 형성되고, 이것은 교체 가능한 유연한 설계를 만듭니다.

객체 지향 5대 원칙 중 의존 관계 역전의 원칙이 있습니다.
“클라이언트는 구체 클래스에 의존하는 것이 아닌 인터페이스나 추상 클래스에 의존해야 한다.”
“구현은 인터페이스보다 변하기 쉽다. 따라서 클라이언트는 변화에 민감한 구체 클래스를 상대하기보다는 인터페이스를 정의해서, 구현이 변화된다 하더라도 변화의 충격에서 자유로울 수 있도록 클라이언트를 구체 클래스와 분리시켜야 한다.”

핵심

- 의존 관계 역전 원칙(Dependency Inversion Principle)
- 강한 결합(Tightly Coupling) : 교체 불가능한 경직된 디자인
- 약한 결합(Loosely Coupling) : 교체 가능한 유연한 디자인

참고 자료

- 실용주의 디자인 패턴 : 0장.소프트웨어 설계의 고고학
- Robert C. Martin, The Dependency Inversion Principle

공통성과 가변성의 분리 1

이번 장에서는 공통성과 가변성의 분리라는 객체지향 소프트웨어 설계 방법에 대해서 정리해 봅시다.

LineEdit 라는 엔터가 입력될 때까지 문자열을 입력받는 GUI 도구를 간단하게 구현해 보겠습니다.

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;

class LineEdit {
  string data;

 public:
  string getData() {
    data.clear();

    while (1) {
      char c = getch();

      if (c == 13) break;

      if (isdigit(c)) {
        data.push_back(c);
        cout << c;
      }
    }
    cout << endl;
    return data;
  }
};
```

위의 LineEdit의 클래스는 숫자 입력에 대해서 한줄의 입력을 받는 형태로 동작하게 됩니다. 하지만 만약 다른 목적의 LineEdit가 필요하다면 위의 LineEdit는 재사용 될 수 없습니다. LineEdit 의 정책을 결정하는 코드에 대해서는 변경할 수 없기 때문입니다. 새로운 정책의 LineEdit가 필요할 때마다 기존 LineEdit의 코드는 변경되어야만 할 것입니다. 즉 OCP 를 만족하지 못합니다.

GoF의 디자인 패턴에는 다음과 같은 말이 있습니다.

> Consider what should be variable in your design. This approach is the opposite of focusing on the cause of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulat- ing the concept that varies, a theme of many design patterns.

Design Patterns : Addison-Wesley, 1995, p. 29

"설계에서 무엇이 변화될 수 있는지 고려하자. 이 접근법은 재설계의 원인에 초점을 맞추는 것과 반대되는 것이다. 설계에 변경을 강요하는 것이 무엇인지에 대해 고려하기 보다는, 재설계 없이 변경시킬 수 있는 것이 무엇인지 고려하자. 여기서의 초점은 많은 디자인 패턴의 주제인 변화하는 개념을 캡슐화하는 것이다"

위의 말은 결국 변해야 하는 것이 변하지 않는 것과 함께 있다면, 분리해야 한다고 할 수 있습니다. 즉 재사용 가능한 클래스를 설계하기 위해서는 공통성에서 가변성을 분리해야 합니다.
결국 LineEdit 에서 정책은 변경 가능 해야 합니다. 그 정책은 isdigit(c) 라는 함수에 의해서 결정됩니다. 그 외의 코드는 LineEdit의 공통된 기능은 변하지 않습니다.

```
string getData() {
  data.clear();

  while (1) {
    char c = getch();

    if (c == 13) break;

    if (isdigit(c)) {
      data.push_back(c);
      cout << c;
    }
  }
  cout << endl;
  return data;
}
```

객체 지향 설계에서 변하는 것, 가변성을 캡슐화하는 방법은 2가지가 있습니다.
첫번째 방법은 변하는 것을 가상 함수로 캡슐화해서 자식 클래스에서 오버라이딩을 통해 동작을 재정의하는 방법 입니다.

```
virtual bool validate(char c) { return isdigit(c); }

string getData() {
  data.clear();

  while (1) {
    char c = getch();

    if (c == 13) break;

    if (validate(c)) {
      data.push_back(c);
      cout << c;
    }
  }
  cout << endl;
  return data;
}
```

이제 새로운 정책의 LineEdit 가 필요하다면 LineEdit 를 상속 받아서 정책을 결정하는 가상함수인 validate() 를 오버라이딩 하면 됩니다. 더 이상 기존 LineEdit에 대한 수정은 필요하지 않습니다.
이렇게 변하지 않는 전체 알고리즘은 부모 클래스가 비 가상함수를 통해 제공하고, 변하는 부분은 가상 함수를 통해 분리하는 설계 기법을 GoF의 디자인 패턴에서는 템플릿 메서드(Template Method) 패턴이라 부릅니다.

GoF의 디자인 패턴에서 템플릿 메서드의 의도는 다음과 같습니다.
“객체의 연산에는 알고리즘의 뼈대만을 정의하고 각 단계에서 수행할 구체적 처리는 서브 클래스쪽으로 미루는 패턴입니다. 알고리즘 구조 자체는 그대로 놔둔 채 알고리즘 각 단계의 처리를 서브 클래스에서 재정의할 수 있게 합니다.”

템플릿 메서드를 통해 재사용 가능한 LineEdit 를 만들었지만, 가상 함수를 통해 정책을 재정의할 경우 다음과 같은 한계가 있습니다.

첫번째 한계. 실행 시간에 LineEdit의 정책을 변경하는 것이 불가능합니다.
가상함수를 통해 정책을 변경하는 것은 결국 객체에 대한 변경이 아닌, 클래스 자체에 대한 변경입니다. 변경된 정책을 사용하기 위해서는 결국 새로운 클래스를 정의해야 합니다.

두번째 한계. 정책을 재사용할 수 없습니다.
만약 텍스트 처리에 관한 정책이 다른 종류의 클래스에서 필요로 한다면, 재사용 될 수 없습니다.
정책에 대한 재사용을 위해서는 다른 방법을 사용해야 합니다.

공통성과 가변성을 분리하는 2번째 방법은 변하는 것을 다른 클래스로 분리하는 것 입니다. 하지만 그 클래스는 교체 가능해야하므로 구체 클래스로 분리하는 것이 아니라 인터페이스 기반 클래스로 분리해야 합니다.

```
class LineEdit {
  string data;

  IValidator* pValidator;

 public:
  LineEdit() : pValidator(0) {}
  void setValidator(IValidator* p) { pValidator = p; }

  string getData() {
    data.clear();

    while (1) {
      char c = getch();

      if (c == 13
        && (pValidator == 0 || pValidator->iscomplete(data)))
        break;

      if (pValidator == 0 || pValidator->validate(data, c)) {
        data.push_back(c);
        cout << c;
      }
    }
    cout << endl;
    return data;
  }
};
```

```
struct IValidator {
  virtual bool validate(string s, char c) = 0;
  virtual bool iscomplete(string s) { return true; }
  virtual ~IValidator() {}
};
```

이제 다양한 정책을 가진 클래스를 제공하기 위해서는 IValidator의 인터페이스를 구현하는 클래스를 제공하면 됩니다.

```
class LimitDigitValidator : public IValidator {
  int value;

 public:
  LimitDigitValidator(int n) : value(n) {}

  virtual bool validate(string s, char c) {
    return s.size() < value && isdigit(c);
  }
  virtual bool iscomplete(string s) { return s.size() == value; }
};
```

```
int main() {
  LineEdit edit;
  LimitDigitValidator v(5);
  edit.setValidator(&v);

  while (1) {
    string s = edit.getData();
    cout << s << endl;
  }
}
```

인터페이스 기반의 클래스로 정책을 분리하게 되면 더이상 클래스에 대한 정책의 변경이 아닌 객체에 대한 정책의 변경이 가능합니다. 실행 시간에 LineEdit의 정책을 변경하는 것이 가능하고, 그뿐만 아니라 IValidator의 인터페이스를 통해 정책을 사용하는 모든 클래스에서 정책을 재사용 가능합니다. 인터페이스 기반의 클래스로 정책의 변경을 가능하게 하는 설계 기법을 GoF의 디자인 패턴에서는 전략 패턴이라고 부릅니다.

GoF의 디자인 패턴에서 전략 패턴의 의도는 다음과 같습니다.
“동일 계열의 알고리즘군을 정의하고, 각각의 알고리즘을 캡슐화하며, 이들을 상호 교환이 가능하도록 만드는 패턴입니다. 알고리즘을 사용하는 사용자와 상관없이 독립적으로 알고리즘을 다양하게 변경할 수 있게 합니다.”

핵심

- 공통성과 가변성을 분리하는 2가지 방법.

참고 자료

- GoF의 디자인 패턴 – Template Method, Starategy
- Design Patterns Explained

공통성과 가변성의 분리 2

이번 장에서는 공통성과 가변성의 분리를 사용하는 다른 설계 방법에 대해서 정리해봅시다.

```
template <typename T>
class List {
 public:
  void push_front(const T& a) {
    // 구현 생략.
  }
};

// 전역변수!
List<int> st;
```

List<> 라는 컨테이너 클래스를 설계한다고 생각해봅시다. 하지만 List가 만약 전역적으로 사용되어야 한다면 push_front의 연산을 비롯한 List를 조작하는 연산이 스레드 안전하게 동작하도록 동기화의 구문을 작성해주어야 합니다.

```
template <typename T>
class List {
 public:
  void push_front(const T& a) {
    mutex.lock();
    //...
    mutex.unlock();
  }
};
```

하지만 위처럼 동기화 코드를 작성하게 되면, 단일 스레드에서 List<>를 사용하는 사용자 입장에서는 동기화에 대한 코드로 인한 성능 저하가 있습니다. 결국 동기화에 대한 정책은 분리되어야 합니다. 단일 스레드 및 다중 스레드의 전략은 List뿐 아니라 다른 컨테이너 클래스에서 사용 가능해야 하므로 인터페이스 기반 클래스로 분리하는 전략 패턴을 적용하겠습니다.

```
struct ISync {
  virtual void lock() = 0;
  virtual void unlock() = 0;
  virtual ~ISync() {}
};
```

다중 스레드의 전략과 단일 스레드의 전략은 ISync의 인터페이스를 구현하는 정책 클래스로 제공하면 됩니다.

```
class NoLock : public ISync {
public:
  virtual void lock() {}
  virtual void nolock() {}
};

class Lock : public ISync {
public:
  virtual void lock() {
    // mutex.lock();
  }

  virtual void unlock() {
    // mutex.unlock();
  }
};
```

프로그램 상에서 실제 스레드 안전하게 동작할 필요하다면 실행 시간에 setSync를 통해 동기화의 전략을 변경 가능합니다.

```
int main() {
  List<int> st;
  st.setSync(new MultiThread);

  // 스레드 안전하다.!
  st.push_front(10);

  st.setSync(new SingleThread);
  // 동기화가 없기 때문에, 훨씬 빠르게 동작한다.!
  st.push_front(20);
}
```

동기화의 정책을 인터페이스 기반의 클래스를 통해 결정하는 것은 결국 인터페이스의 가상 함수 호출을 통해 결정되기 때문에 성능의 저하가 있습니다. push_front 처럼 빈번하게 호출되는 함수라면 분명 함수 호출에 대한 오버헤드를 무시할 수 없습니다. 또한 List 같은 컨테이너의 정책은 굳이 실행 시간에 변경되는 것도 유용하지 않습니다. 만약 실행 시간에 정책이 변경될 필요가 없고, 함수 호출에 따른 성능 저하가 걱정된다면, C++ 진영의 다른 설계를 적용하면 됩니다.

```
class SingleThread {
 public:
  inline void lock() {}
  inline void unlock() {}
};

template <typename T, typename ThreadModel = SingleThread>
class List {
  ThreadModel tm;

 public:
  void push_front(const T& a) {
    tm.Lock();
    //......
    tm.Unlock();
  }
};

List<int, SingleThread> st;
List<int, MultiThread> mt;
```

이제 List의 동기화 정책은 ThreadModel 이라는 템플릿 인자에 의해 컴파일 타임에 결정됩니다. 더이상 실행 시간에 정책을 교체하는 것이 불가능합니다. 하지만 정책 클래스는 더이상 인터페이스 기반의 클래스가 아니기 때문에 lock과 unlock의 함수가 가상 함수가 아닙니다. 가상 함수가 아닌 멤버 함수는 컴파일 타임에 인라인화가 가능하기 때문에, lock과 unlock의 함수 호출에 따른 성능의 저하가 전혀 없습니다.

List처럼 템플릿 인자로 정책을 결정하는 설계 기법을 단위 전략(Policy base) 라고 부릅니다. GoF의 디자인 패턴에서는 매개변수화 타입(parameterized type)을 이용한 재사용 기법으로 언급되어 있습니다.

핵심

- 템플릿 기반 클래스로 분리하는 단위 전략 설계의 특징
- 전략 패턴을 통한 정책 교체의 장점, 단점

참고 자료

- GoF의 디자인 패턴 - Starategy
- Modern C++ Design

함수와 전략

이번 장에서는 공통성과 가변성의 분리를 사용하는 다른 설계 방법에 대해서 정리해봅시다.

```
void sort(int* x, int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + i; j < n; j++) {
      if (x[i] > x[j]) swap(x[i], x[j]);
    }
  }
}

int main() {
  int x[10] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

  sort(x, 10);
}
```

위의 코드는 버블 정렬의 코드 입니다. 입력으로 주어진 int 배열을 오름차순으로 정렬합니다. 하지만 오름차순 정렬이라는 정책이 고정되어 있기 때문에, 이 함수는 내림차순에 정책에 대해서는 재사용이 불가능합니다. 결국 일반 함수에서도 변하지 않는 전체 알고리즘에서도 변해야 하는 정책 코드는 분리되어야 합니다.

일반함수에 변하는 것을 분리하는 방법은 변하는 것을 함수 인자, 즉 함수 포인터로 분리하면 됩니다.

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

void sort(int* x, int n, bool (*cmp)(int, int)) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + i; j < n; j++) {
      if (cmp(x[i], x[j])) swap(x[i], x[j]);
    }
  }
}

bool cmp1(int a, int b) { return a < b; }
bool cmp2(int a, int b) { return a > b; }
int main() {
  int x[10] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

  sort(x, 10, cmp2);
}
```

정렬 알고리즘에서 오름차순과 내림차순의 정책을 결정하는 문장을 함수 포인터를 통해서 외부에서 사용자가 결정할 수 있게 되었습니다. 공통성과 가변성의 분리는 단순히 클래스를 설계할 뿐 아니라 일반적인 함수를 설계할 때도 적용되어야 합니다. C언어의 qsort 함수가 바로 그러한 원리에 의해서 설계되어 있습니다.

하지만 정책을 함수 포인터에 의해서 결정하는 것은 정책을 문장으로 고정해서 사용하는 것에 비해 유연하게 동작하겠지만 성능적인 차이는 분명합니다. 일반적으로 정렬 알고리즘은 데이터의 크기가 늘어남에 따라 많은 비교를 수행하게 될 것이고, 비교를 위해서는 함수 포인터를 통한 함수 호출의 성능 저하가 발생할 수 밖에 없습니다.

인라인 함수를 통해서 함수 포인터에 대한 호출의 성능 저하를 해결할 수 있을 것 같지만, 안타깝게도 인라인 기능은 함수 포인터에 대해서 동작하지 않습니다.

```
       int add1(int a, int b) { return a + b; }
inline int add2(int a, int b) { return a + b; }

int main() {
  int n1 = add1(1, 2);
  int n2 = add2(1, 2);
}
```

위 코드를 컴파일하면 다음과 같은 기계어가 생성됩니다.

```
; int n1 = add1(1, 2);
      push  2
      push  1
      call  ?add1@@YAHHH@Z                        ; add1
      add   esp, 8
      mov   DWORD PTR _n1$[ebp], eax

; int n2 = add2(1, 2);
      mov   eax, 1
      add   eax, 2
      mov   DWORD PTR _n2$[ebp], eax
```

일반적으로 인라인 함수를 사용할 경우 일반 함수와 달리 호출을 수행하지 않는 것을 확인할 수 있습니다. 기계어 제어가 다른 코드로 옮겨가지 않기 때문에 빠르게 일반적인 함수에 비해 빠르게 동작할 수 있습니다. 하지만 위의 인라인 기능이 함수 포인터를 통한 호출에서는 제대로 동작하지 않습니다.

```
    int(*f)(int, int);
    f = &add2;

    int n = 0;
    cin >> n;
    if (n == 0)
        f = &add1;

    int n3 = f(1, 2);
}
```

인라인 함수를 함수 포인터를 통해 호출하면 인라인 치환이 동작하지 않습니다. 인라인 함수는 컴파일 시간에 함수의 호출을 기계어 치환해주는 문법입니다. 하지만 함수 포인터 변수는 실행 시간에 변경이 가능하므로 컴파일러가 어떤 함수를 가르키고 있을지 알 수 없습니다. 즉 인라인 치환이 불가능합니다.

핵심

- 함수에서 정책을 분리하는 방법
- 인라인 함수와 함수 포인터의 관계

함수 객체와 전략

C++ 에서는 함수에서 함수 뿐 아니라 함수 객체를 통해서도 정책을 분리할 수 있습니다.

우선 함수 객체에 대해서 알아봅시다.

```
struct Plus {
  int operator()(int a, int b) { return a + b; }
};

int main() {
  Plus p;

  int n = p(1, 2);
  cout << n << endl;
}
```

Plus는 분명 클래스이지만 객체 생성후 호출 연산자를 통해서 마치 함수처럼 사용 가능합니다. 함수처럼 호출 가능하기 때문에 앞장의 sort 함수의 정책을 함수 포인터 대신 함수 객체를 통해 분리할 수 있습니다.

```
struct Less {
  inline bool operator()(int a, int b) { return a < b; }
};

void sort(int* x, int n, Less cmp) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + i; j < n; j++) {
      if (cmp(x[i], x[j])) swap(x[i], x[j]);
    }
  }
}

int main() {
  int x[10] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

  Less l;
  sort(x, 10, l);
}
```

sort의 정책은 Less 라는 타입으로 결정되어 있기 때문에, 컴파일러가 어떤 함수를 바인딩할지에 대해서 컴파일 시간에 결정 가능합니다. 따라서 인라인이 가능하게 되었습니다.

하지만 함수 객체는 함수 포인터와 달리 자신만의 타입이 있습니다. 즉 함수의 타입이 함수의 시그니처에 의해서 결정되는 것이 아니라 함수 객체 타입에 의해 결정되기 때문에 위의 sort 구현은 Less 타임에 대한 함수 객체만 사용 가능합니다. 또한 일반 함수에 대해서도 사용이 불가능합니다.

```
template <typename T>
void sort(int* x, int n, T cmp) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + i; j < n; j++) {
      if (cmp(x[i], x[j])) swap(x[i], x[j]);
    }
  }
}

int main() {
  int x[10] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};
  Less l; Greater g;

  sort(x, 10, l);
  sort(x, 10, g);
}
```

정책을 교환해야 되는 함수를 템플릿을 적용하면, 다양한 함수 객체를 정책으로 사용 할 수 있습니다. 또한 인라인도 가능합니다. 하지만 정책을 교환할 때마다 컴파일 시간에 해당하는 정책의 함수 객체 타입으로 코드를 생성해주기 때문에 많은 정책을 사용하고 있다면 그만큼의 코드 메모리를 사용해야 합니다.

성능적으로 뛰어난 함수 객체는 일반적인 함수 구현에 비해 사용하기가 번거롭습니다. 함수 객체 타입의 이름을 결정해야되고, 함수 객체의 이름도 고려해야 합니다. 또한 일반 함수보다 타이핑이 더 많습니다. C++11 에서는 람다 표현식이라는 문법을 통해 함수 객체를 쉽고 편리하게 사용 가능합니다.

```
sort(x, 10, [](int a, int b) { return a < b; });
sort(x, 10, [](int a, int b) { return a > b; });
```

C++에서는 함수에 변해야 하는 정책이 존재한다면, 템플릿을 통해 정책을 분리합니다. 그렇게하면 다양한 함수 객체에 대해서 정책을 교환하는 것이 가능합니다. 또한 템플릿 함수이기 때문에 정책을 일반 함수로도 사용 가능합니다. 정책을 일반 함수를 통해 구현하고 함수 포인터를 통해 교체한다면, 생성해야 되는 함수의 구현은 한개뿐입니다. 즉 템플릿에 의한 코드 메모리 낭비가 없습니다. 하지만 인라인화가 되지 않기 때문에 호출에 따른 성능의 저하가 있습니다. 반면에 함수 객체를 사용한다면 정책을 교환할때마다 그에 따른 코드를 생성하는 비용을 감수해야하지만 인라인화가 가능하기 때문에 빠르게 동작합니다.

핵심

- C++ 정책 분리 방법
- 함수 객체(람다 표현식) vs 함수 포인터

Framework

이번 장에서는 프레임워크 설계에 대해서 알아봅시다. 먼저 프레임워크가 라이브러리와 다른 점에서 대해서 이해해야 합니다. 라이브러리의 일반적 정의는 프로그램 개발에 필요한 도구를 의미합니다. 따라서 라이브러리의 사용자는 자유롭게 그 도구를 통해 원하는 기능을 구현할 수 있습니다. 하지만 프레임워크는 단순히 도구만을 제공하는 것이 아니라 미리 정의된 실행 흐름이 존재합니다. 그러므로 프레임워크의 사용자는 새로운 기능을 추가하기 위해서는 프레임워크에서 정의한 실행 흐름에 맞게 정의해야 합니다. 마치 안드로이드 프레임워크에서 "액티비티의 터치 이벤트를 추가하기 위해서는 Activity의 onTouch 메소드를 재정의 해야 한다". 또는 MFC에서 "프로그램이 처음 시작되었을 때 d 애플리케이션의 초기화 작업은 WinApp의 InitInstance 함수를 재정의해야 한다" 같은 규칙이 존재하는 것과 유사합니다. 그 외의 방법으로는 터치 이벤트를 추가하거나 애플리케이션 초기화 작업을 수행하는 것이 불가능합니다.

```
class CMyApp : public CWinApp {
 public:
  virtual bool InitInstance() {
    cout << "프로그램 시작" << endl;
    return true;
  }

  virtual int ExitInstance() {
    cout << "프로그램 종료" << endl;
    return 0;
  }
};

CMyApp theApp;
```

MFC에서 사용자들은 프로그램의 시작은 가상 함수 InitInstance가 호출되고, 프로그램의 종료전에 가상함수 ExitInstance가 호출된다고 생각합니다. 하지만 C++ 프로그램은 프로그램의 시작과 끝은 main함수를 통해서 처리됩니다. 그렇다면 MFC에서는 main함수가 존재하지 않는 것일까요? MFC에서는 main함수를 라이브러리 안에 숨김으로써 사용자에게 미리 정의된 흐름속에서 프로그래밍을 할 수 있도록 하고 있습니다.

라이브러리 내의 실제 코드를 직접 구현해보면 다음과 같습니다.

```
class CWinApp;

CWinApp* g_app;

class CWinApp {
 public:
  CWinApp() { g_app = this; }
  virtual bool InitInstance() { return false; }
  virtual int ExitInstance() { return 0; }
  virtual int Run() { return 0; }
};

CWinApp* AfxGetApp() { return g_app; }

int main() {
  CWinApp* pApp = AfxGetApp();

  if (pApp->InitInstance())
      pApp->Run();
  pApp->ExitInstance();
}
```

라이브러리의 사용자는 3가지의 규칙을 지켜야 합니다.
첫번째 규칙은 CWinApp의 파생 클래스를 만들어야 합니다. 두번째 규칙은 가상함수 InitInstance를 재정의해서 초기에 하고 싶은 일을 수행합니다. 세번째 규칙은 사용자가 만들 클래스를 전역적으로 한 개 생성해야 합니다.
자식 클래스의 전역적인 생성은 main함수가 호출되기 이전에 부모 클래스의 생성자를 호출합니다. 부모 클래스의 생성자에서 전역 변수 g_app에 자식 클래스의 인스턴스의 주소를 저장합니다. main에서는 AfxGetApp 전역 함수를 통해 자식 클래스 인스턴스의 주소를 얻어오게 되고, 가상 함수 InitInstance를 통해 애플리케이션의 초기화 과정을 수행합니다. 초기화 과정이 성공하게 되면 Run 가상 함수를 호출합니다. 모든 수행이 끝나면 ExitInstance 가상 함수를 호출합니다. 위의 설계는 결국 전체적인 흐름은 라이브러리안에 두고, 각 단계를 가상함수화 해서 자식이 재정의 할 수 있도록 하는 템플릿 메소드 패턴입니다. MFC, wxWidget, 안드로이드 앱, 아이폰 앱 등 많은 프레임워크에서 템플릿 메소드 패턴을 사용해 프레임워크의 흐름을 제공하고, 가상함수(메소드) 재정의를 통해 기능을 확장하도록 하는 설계가 적용되어 있습니다.

핵심

- 라이브러리와 프레임워크 차이
- 템플릿 메소드

참고 자료

- GoF의 디자인 패턴 – Template Method

Menu

이번 장에서는 메뉴를 설계해봅시다.
메뉴는 두 개의 타입으로 추상화할 수 있습니다.

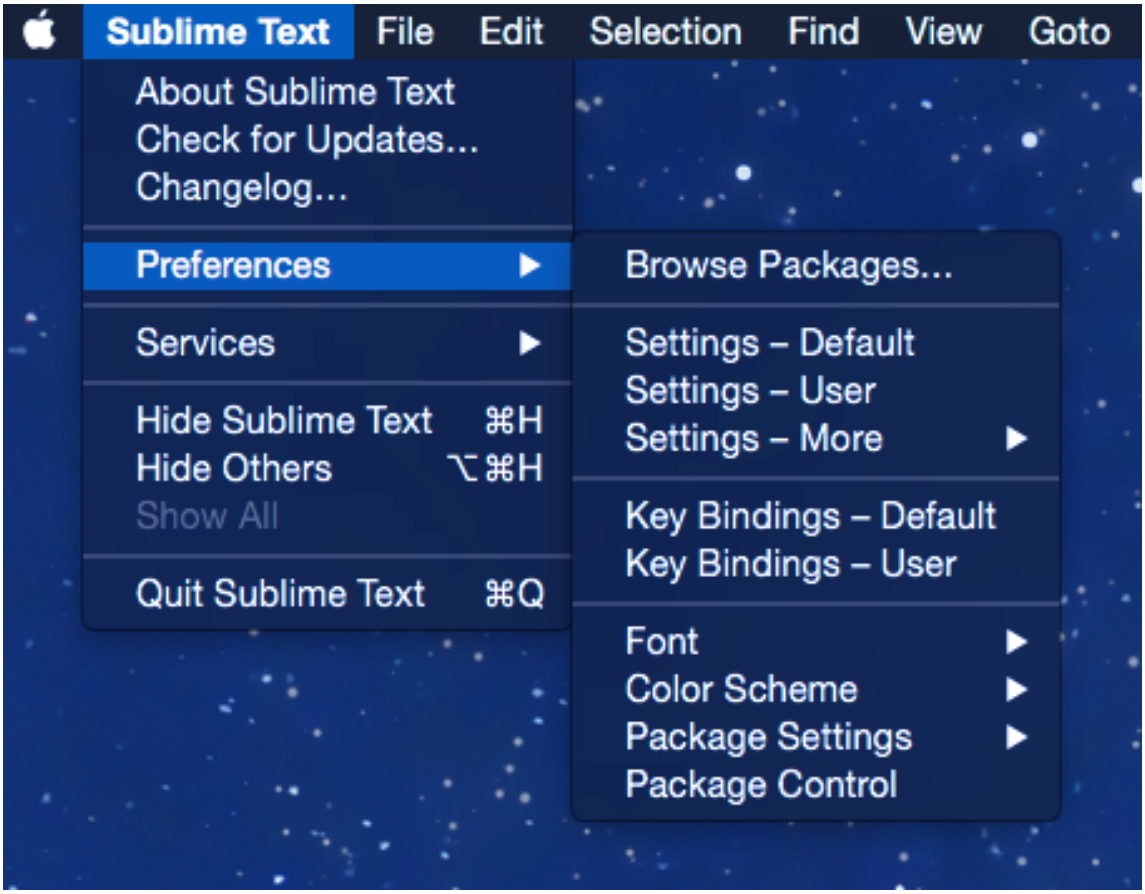
첫번째 타입은 MenuItem 으로서 제목을 가지고 있고, 메뉴 이벤트가 발생하였을 때, 어떤 메뉴에서 이벤트가 발생했는지를 구분할 수 있도록 id가 필요합니다. command 함수는 메뉴가 선택되었을 때 이벤트를 외부로 전파하는 기능을 수행합니다. 여기서는 선택되었다는 사실만 출력하도록 하겠습니다.

```
class MenuItem {
  string title;
  int id;

 public:
  MenuItem(string s, int n) : title(s), id(n) {}

  void command() {
    cout << title << "메뉴가 선택됨" << endl;
    getch();
  }
};
```

두번째 타입은 PopupMenu입니다.

```
class PopupMenu {
  string title;
  vector <???> v;
 public:
  PopupMenu(string s) : title(s) {}

  void addMenu(??? p) { v.push_back(p);}
  void command() {
      ...
  }
};
```

PopupMenu 메뉴는 addMenu 함수를 통해 다른 메뉴를 포함하는 기능을 제공합니다. 또한 command 함수를 통해 메뉴가 선택되었을 경우 자신이 가지고 있는 모든 메뉴를 보여주는 기능을 수행합니다.
그렇다면 메뉴를 포함하기 위한 자료구조인 벡터의 원소 타입과 addMenu의 인자 타입은 무엇이 되어야 할까요?



PopupMenu가 포함하는 메뉴는 MenuItem 타입 뿐 아니라 PopupMenu 타입도 포함할 수 있어야 합니다. PopupMenu와 MenuItem을 묶을 수 있어야 합니다. 객체 지향 설계에서 서로 다른 타입의 클래스 A와 B를 묶어서 관리 하기 위해서는 공통의 부모가 필요합니다. 즉 MenuItem과 PopupMenu의 공통의 부모 BaseItem을 설계하고, 공통의 속성을 부모 클래스로 관리하도록 하겠습니다.

```
class BaseMenu {
  string title;

 public:
  BaseMenu(string s) : title(s) {}
  virtual ~BaseMenu() {}

  string getTitle() const { return title; }
  virtual void command() = 0;
};
```

이제 MenuItem과 PopupMenu는 BaseMenu의 자식입니다. 또한 title 속성은 자식의 공통된 속성이므로 부모 클래스의 속성으로 관리됩니다. 공통의 인터페이스 command 함수는 BaseMenu가 굳이 구현해줄 필요가 없고, 자식 클래스에 의해서 제공되어야 하는 기능이므로 순수 가상 함수로 설계하였습니다. BaseMenu는 순수 가상 함수를 1개 이상 가지고 있는 추상 클래스 입니다. 만약 자식 클래스

는 command 함수를 제공하지 않으면 추상 클래스가 되고 인스턴스화 할 수 없습니다. 부모 입장에서는 구현해줄 필요가 없고, 자식이 반드시 만들어야 한다면 순수 가상 함수가 유용합니다.

```
class MenuItem : public BaseMenu {
  int id;

 public:
  MenuItem(string s, int n) : BaseMenu(s), id(n) {}

  void command() {
    cout << getTitle() << "메뉴가 선택됨" << endl;
    getch();
  }
};

class PopupMenu : public BaseMenu {
  vector<BaseMenu*> v;

 public:
  PopupMenu(string s) : BaseMenu(s) {}
  void addMenu(BaseMenu* p) { v.push_back(p); }
  void command() {
    while (1) {
      clrscr();

      int sz = v.size();
      for (int i = 0; i < sz; i++)
        cout << i + 1 << ". " << v[i]->getTitle() << endl;
      cout << sz + 1 << ". 상위 메뉴로" << endl;

      cout << "메뉴를 선택하세요 >>";
      int cmd;
      cin >> cmd;

      if (cmd == sz + 1) break;
      if (cmd < 1 || cmd > sz + 1) continue;

      v[cmd - 1]->command();
    }
  }
};
```

이제 PopupMenu는 BaseMenu*에 대해서 포함하는 것이 가능합니다. 이것은 MenuItem 타입에 대한 포함 뿐 아니라 PopupMenu 타입, 즉 자신의 타입에 대한 포함도 가능합니다. 이런식의 구성 방법을 디자인 패턴에서는 재귀적 합성(recursive composition) 기법이라고 부릅니다. 또한 재귀적 합성을 통한 계층적으로 구조화된 정보를 보여줄 수 있습니다. 마치 우리의 메뉴 시스템처럼 말입니다. 또한 위와 같은 재귀적 합성을 통해서 복합 객체를 구성하는 패턴을 GoF의 디자인 패턴에서는 컴포지트 패턴(Composite Pattern)이라고 부릅니다.

GoF의 디자인 패턴에서 컴포지트 패턴의 의도는 다음과 같습니다.

“객체들의 관계를 트리 구조로 구성하여 부분-전체 계층을 표현하는 패턴으로, 사용자가 단일 객체와 복합 객체 모두 동일하게 다루도록 합니다.”

위의 메뉴를 사용하는 사용자 코드는 다음과 같습니다.

```
int main() {
  PopupMenu* menubar = new PopupMenu("MENUBAR");
  PopupMenu* p1 = new PopupMenu("화면 설정");
  PopupMenu* p2 = new PopupMenu("소리 설정");

  menubar->addMenu(p1);
  p1->addMenu(p2);

  p1->addMenu(new MenuItem("해상도 설정", 11));
  p1->addMenu(new MenuItem("색상 설정", 12));
  p1->addMenu(new MenuItem("기타 설정", 13));

  p2->addMenu(new MenuItem("볼륨 설정", 21));
  p2->addMenu(new MenuItem("음색 설정", 22));

  menubar->command();
}
```

컴포지트 패턴은 두개의 특징이 있습니다.
첫번째 특징은 복합 객체(PopupMenu)는 단일 객체(MenuItem) 뿐 아니라 복합 객체도 포함할 수 있어야 합니다. 즉 단일 객체와 복합 객체은 공통의 부모(BaseMenu)가 필요합니다. 두번째 특징은 복합 객체의 사용법이 동일해야 합니다. 또한 그 사용법은 부모 클래스에 기반해야 합니다.(command)

핵심

- Composite 패턴

참고 자료

- GoF의 디자인 패턴 - Composite

Menu Event

이번 장에서는 C++에서 이벤트를 처리하는 방법에 대해서 정리해 봅시다.

```
class MenuItem {
  int id;

 public:
  MenuItem(int n) : id(n) {}

  virtual void command() {
    ...
  }
};

int main()
{
    MenuItem m1(11), m2(12);
    m1.command();
}
```

메뉴가 선택되면 command 함수가 호출될 것입니다. 여기서 메뉴는 자신이 선택되었다는 사실을 다시 외부로 알려야 합니다. "객체가 외부로 이벤트를 발생시킨다." 라고 표현합니다. 객체 지향 프로그래밍 언어에서 이벤트를 처리하기 위한 설계는 크게 두가지 입니다. 첫 번째는 안드로이드 앱 등 자바 기반 프레임워크에서 많이 사용되는 인터페이스 기반 리스너 설계 방식입니다. 두 번째는 C#, Objective-C, Qt 등이 사용하고 있는 함수 포인터 기반의 설계 방식입니다.

먼저 인터페이스 기반 설계 방식을 이용한 이벤트 처리 방법입니다.

```
struct IMenuListener {
  virtual void onCommand(int id) = 0;
  virtual ~IMenuListener() {}
};

class MenuItem {
  int id;
  IMenuListener* pListener;

 public:
  MenuItem(int n) : id(n), pListener(0) {}
  void setListener(IMenuListener* p) { pListener = p; }

  virtual void command() { pListener->onCommand(id); }
};
```

이제 MenuItem은 IMenuListener의 인터페이스를 구현하는 클래스를 등록 가능합니다. 메뉴가 선택되면 등록된 객체의 onCommand 함수를 통해 이벤트가 발생했다는 사실을 알려줍니다.

```
#define DIALOG_CLOSE 10
#define DIALOG_OPEN_FILE 11

class Dialog : public IMenuListener {
 public:
  virtual void onCommand(int id) {
    switch (id) {
      case DIALOG_CLOSE:
        close();
        break;
      case DIALOG_OPEN_FILE:
        open();
        break;
    }
  }

  ...
};

int main() {
  MenuItem m1(DIALOG_CLOSE), m2(DIALOG_OPEN_FILE);

  Dialog dlg;
  m1.setListener(&dlg);
  m2.setListener(&dlg);

  m1.command();
}
```

인터페이스 기반 리스너의 가장 큰 문제는 이벤트를 받는 객체가 여러개의 객체로부터 이벤트를 받을 경우 어떤 객체로부터 이벤트가 발생되었는지 알 수 없습니다. 모든 객체가 동일하게 onCommand를 호출하기 때문입니다. 결국 id 같은 추가적인 정보를 통해 어떤 객체로부터 이벤트가 발생했는지를 판단 해야 합니다. 이러한 형태의 조건 분기 코드는 다이얼로그가 많은 메뉴를 가질 수록 점점 더 복잡해질뿐 아니라 새로운 메뉴가 추가될 때마다 수정되어야 합니다. 그러나 함수 포인터 기반의 이벤트 처리 방식은 위의 문제를 해결할 수 있습니다. 즉 메뉴 아이템에 객체를 등록하는 것이 아니라 어떤 함수를 호출할지를 등록한다면 객체를 구분할 필요가 없습니다.

```
class Dialog {
 public:
  void close() { cout << "Dialog close" << endl; }
  void open() { cout << "Dialog open" << endl; }
};
```

```
class MenuItem {
  void (Dialog::*handler)();
  Dialog* object;

 public:
  void setHandler(void (Dialog::*h)(), T* o) {
    handler = h;
    object = o;
  }

  virtual void command() { (object->*handler)(); }
};

int main() {
  Dialog dlg;
  MenuItem m1(11), m2(12);
  m1.setHandler(&Dialog::close, &dlg);
  m2.setHandler(&Dialog::open, &dlg);

  m1.command();
  m2.command();
}
```

MenuItem은 Dialog의 함수를 등록할 수 있도록 재설계되었습니다. 많은 메뉴를 Dialog가 가지고 있다고 하더라도 이제 MenuItem의 클래스의 command에는 조건 분기와 같은 코드는 더 이상 필요하지 않습니다. 하지만 위의 MenuItem은 몇가지 문제를 가지고 있습니다. MenuItem이 등록할 수 있는 함수는 Dialog의 멤버 함수만 가능합니다. 또 다른 문제는 일반적인 함수를 등록하는 것이 불가능합니다. C++에서는 일반 함수 포인터와 멤버 함수 포인터는 다릅니다. 그뿐 아니라 멤버 함수 포인터는 클래스 타입의 정보를 포함하는 형태로 만들어야 합니다. 즉 C++에는 모든 클래스의 멤버 함수 주소를 담을 수 있을 뿐 아니라 일반 함수의 주소도 담을 수 있는 범용적인 함수 포인터 타입이 존재하지 않습니다. C#에는 델리게이트가 있고, Objective-C에는 셀렉터라는 범용 함수 포인터가 제공되기 때문에 이러한 문제가 발생하지 않습니다.

지금부터 모든 타입의 멤버 함수의 주소를 담을 수 있을뿐 아니라 일반 함수의 주소도 담을 수 있는 범용 함수 포인터 타입을 설계해봅시다.
만드는 과정은 다음과 같습니다.
첫번째. 일반 함수 포인터를 클래스로 래핑합니다.
두번째. 멤버 함수 포인터를 클래스로 래핑합니다.
세번째. 일반 함수 포인터와 멤버 함수 포인터를 동일시 하기 위해 공통의 부모를 설계합니다.

언어적으로 제공되는 일반 함수 포인터와 멤버 함수 포인터를 그냥 묶을 수는 없습니다. 묶기 위해서 각각을 객체로 래핑해야 합니다.

먼저 일반 함수 포인터의 기능을 하는 새로운 타입을 설계합시다.

```
class FunctionCommand {
  typedef void (*HANDLER)();
  HANDLER handler;

 public:
  FunctionCommand(HANDLER h) : handler(h) {}
  void execute() { handler(); }
};

void foo() { cout << "foo" << endl; }

int main() {
  FunctionCommand fc(&foo);
  fc.execute();
}
```

다음은 멤버 함수 포인터의 기능을 하는 새로운 타입을 설계합시다. 하지만 모든 클래스 타입의 멤버 함수 주소를 저장 가능해야 하므로 템플릿으로 설계해야 합니다.

```
template <typename T>
class MemberCommand {
  typedef void (T::*HANDLER)();
  HANDLER handler;
  T* object;

 public:
  MemberCommand(HANDLER h, T* p) : handler(h), object(p) {}
  void execute() { (object->*handler)(); }
};

class Dialog {
public:
    void close() { cout << "Dialog close" << endl; }
};

int main() {
  Dialog dlg;
  MemberCommand<Dialog> mc(&Dialog::close, &dlg);
  mc.execute();
}
```

일반 함수 포인터를 다루는 FunctionCommand와 모든 클래스 타입의 멤버 함수 포인터를 다루는 MemberCommand를 동시에 다루기 위해서는 "A와 B를 묶기 위해서는 공통의 부모가 필요하다"라는 설계 원칙에 의해 공통의 부모 클래스 즉 인터페이스를 제공하면 됩니다.

```
struct ICommand {
  virtual void execute() = 0;
  virtual ~ICommand() {}
};
```

```
class FunctionCommand : public ICommand {
    ...
};

template <typename T>
class MemberCommand : public ICommand {
    ...
};

int main() {
  ICommand* p = NULL;

  p = new FunctionCommand(&foo);
  p->execute();

  Dialog dlg;
  p = new MemberCommand<Dialog>(&Dialog::close, &dlg);

  p->execute();
}
```

ICommand 타입을 통해 일반 함수의 주소를 저장할 수 있을뿐 아니라 모든 클래스의 멤버 함수의 주소를 저장할 수 있습니다. 하지만 클래스 템플릿인 MemberCommand를 만들기가 불편하기 때문에 좀 더 쉽게 사용할 수 있도록 유틸리티 함수를 제공합니다. 일반적으로 암시적 추론이 불가능한 클래스 템플릿을 쉽게 사용할 수 있도록 함수 템플릿을 제공하면 편리하게 사용하는 것이 가능합니다.

```
template <typename T>
MemberCommand<T>* cmd(void (T::*f)(), T* p) {
  return new MemberCommand<T>(f, p);
}
```

라이브러리의 일관성을 위해 FunctionCommand를 만드는 함수도 제공합니다.

```
FunctionCommand* cmd(void (*f)()) { return new FunctionCommand(f); }
```

사용자는 이제 직접 FunctionCommand 또는 MemberCommand를 사용하는 것이 아니라, cmd라는 일관적인 방법을 통해 함수의 주소를 저장할 수 있습니다.

```
int main() {
  ICommand* p = NULL;

  Dialog dlg;
  p = cmd(&foo);
  p->execute();

  p = cmd(&Dialog::close, &dlg);
  p->execute();
}
```

최종적인 MenuItem의 모습은 다음과 같습니다.

```
class MenuItem {
  ICommand* pCommand;

 public:
  void setCommand(ICommand* p) { pCommand = p; }
  virtual void command() { pCommand->execute(); }
};

int main() {
  MenuItem m;
  m.setCommand(cmd(&foo));
  m.command();

  Dialog dlg;
  m.setCommand(cmd(&Dialog::close, &dlg));
  m.command();
}
```

핵심

- 인터페이스 기반의 리스너를 통한 이벤트 처리 방법
- 함수 포인터 기반의 이벤트 처리 방법

메세지 맵

4장에서 구현한 윈도우 클래스는 메세지를 가상 함수 기반으로 자식이 재정의하는 형태로 설계되어 있습니다. 따라서 윈도우 메세지는 Window에 가상 함수의 형태로 제공되어야 합니다. 그래야 자식 클래스에서 원하는 메세지를 재정의해서 처리할 수 있습니다.

```
class Window {
  int handle;
  static map<int, Window*> this_map;

 public:
  void create() {
    handle = IoMakeWindow(foo);
    this_map[handle] = this;
  }

  static int foo(int handle, int msg, int param1, int param2) {
    Window* self = this_map[handle];
    switch (msg) {
      case WM_LBUTTONDOWN:
        self->onLButtonDown();
        break;
      case WM_KEYDOWN:
        self->onKeyDown();
        break;
        ...
    }
    return 0;
  }

  virtual void onLButtonDown() {}
  virtual void onKeyDown() {}
  ...
};

map<int, Window*> Window::this_map;

class MyWindow : public Window {
 public:
  virtual void onLButtonDown() { cout << "lbutton" << endl; }
};
```

C++에서는 클래스가 가상 함수를 한개 이상 가지고 있다면, 가상 함수를 실행시간에 호출하기 위한 테이블을 유지해야 합니다. 가상 함수 테이블은 클래스가 가지고 있는 모든 가상 함수의 주소를 저장

합니다. 따라서 가상 함수의 개수에 비례합니다. 그뿐 아니라 가상 함수를 정의하고 있는 클래스를 상속받는 새로운 클래스가 추가될 때마다 부모의 가상 함수 테이블을 복제해야 합니다.
그런데 마이크로소프트 윈도우즈가 정의하고 있는 윈도우 메세지의 종류는 대략 1000개가 넘습니다. 이 모든 메세지를 가상함수 기반으로 처리하기 위해서는 부모 클래스에서 모든 메세지에 대한 가상 함수를 제공하여야 합니다. 이것은 가상 함수 테이블의 크기가 1000개 이상의 멤버 함수의 주소를 담을 수 있는 크기로 형성되어야 하는 것을 의미하고, 윈도우의 자식 클래스가 추가됨에 따라 그 배의 메모리가 필요하다는 것의 의미합니다.

윈도우의 모든 메세지를 가상 함수 기반으로 처리하는 것의 비용을 줄이기 위해서, MFC와 wxWidget에서는 메세지 맵을 통한 윈도우 메세지 처리를 사용하고 있습니다.

메세지 맵을 적용한 Window 클래스는 다음과 같습니다.

```
class Window;
struct AFX_MSG {
  int message;
  void (Window::*handler)();
};

class Window {
  int handle;
  static map<int, Window*> this_map;

 public:
  ...

  virtual AFX_MSG* GetMessageMap() { return 0; }

  static int foo(int handle, int msg, int param1, int param2) {
    Window* self = this_map[handle];
    if (self == 0) return 0;

    AFX_MSG* msgArray = self->GetMessageMap();
    if (msgArray == 0) return 0;

    for (; msgArray->message != 0; msgArray++) {
      if (msgArray->message == msg) {
        void (Window::*f)() = msgArray->handler;
        (self->*f)();
      }
    }

    return 0;
  }
};
```

이제는 모든 메세지에 대해서 가상함수를 미리 정의하는 것이 아니라, Window를 상속받는 자식 윈도우가 처리할 메세지와 함수의 주소를 저장하고 있는 배열을 만들어서 제공하면, 부모 윈도우가 GetMessageMap 가상 함수를 통해서 자식이 정의한 메세지 맵을 얻어와, 처리해주는 형태로 구현되어 있습니다. 이제 부모 윈도우에는 모든 윈도우 메세지에 대한 가상함수가 필요 하지 않습니다. GetMessageMap 가상 함수 하나만 있으면 됩니다.

자식 윈도우는 자신이 처리할 메세지의 번호와 함수의 주소를 담은 배열을 만들어서 리턴합니다.

```
class MyWindow : public Window {
 public:
  virtual AFX_MSG* GetMessageMap() {
    typedef void (Window::*HANDLER)();

    static AFX_MSG msgMap[] = {
        {WM_LBUTTONDOWN,
            static_cast<HANDLER>(&MyWindow::onLButtonDown)},
        {0, 0},
    };

    return msgMap;
  }

  void onLButtonDown() { cout << "lbutton" << endl; }
};
```

이런식으로 사용자가 메세지 맵을 만들어서 리턴하도록 하면, 불편합니다. 자동적으로 GetMessageMap을 생성해주는 매크로를 제공하면 편리하게 사용할 수 있습니다.

```
#define DECLARE_MESSAGE_MAP() virtual AFX_MSG* GetMessageMap();

#define BEGIN_MESSAGE_MAP(classname)    \
  AFX_MSG* classname::GetMessageMap() { \
    typedef void (Window::*HANDLER)();  \
    static AFX_MSG msgMap[] = {

#define ADD_MAP(message, handler)             \
  { message, static_cast<HANDLER>(handler) },

#define END_MESSAGE_MAP() \
    { 0, 0},              \
  };                      \
  return msgMap;          \
}
```

이제 자식 윈도우를 만들 때에 직접 메세지 맵을 구현하는 것이 아니라, 매크로를 이용한다면 쉽게 메세지 맵을 제공하는 것이 가능합니다.

```
class MyWindow : public Window {
 public:
  void onLButtonDown() { cout << "LBUTTON" << endl; }

  DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(MyWindow)
ADD_MAP(WM_LBUTTONDOWN, &MyWindow::onLButtonDown)
END_MESSAGE_MAP()
```

핵심

- 메세지 맵을 통한 윈도우 메세지 처리 방법.
- 매크로를 이용한 복잡한 코드를 자동적으로 생성해주는 방법.

CRTP

다음은 WTL(Windows Template Library)에서 사용하고 있는 메세지 처리 방법입니다.

```
template <typename T>
class Window {
 public:
  void msgLoop() {
    ...
    (static_cast<T*>(this))->onClick();
  }

  void onClick() {}
};

class MyWindow : public Window<MyWindow> {
 public:
  void onClick() { cout << "My Window onClick" << endl; }
};

int main() {
  MyWindow w;
  w.msgLoop();
}
```

이처럼 템플릿으로 작성된 기본 클래스에 템플릿 인자로 자식 클래스의 인자로 자식 클래스의 이름을 전달하는 기술을 CRTP(Curiosly Recurring Template Pattern)라고 합니다. WTL에서는 비가상함수가 가상함수처럼 동작하도록 만드는 형태로 사용되었지만, 다양한 활용이 가능합니다.

다음은 객체의 생성을 N개로 제한하는 기술 입니다.

```
template <typename T, int N = 1>
class LimitMaxObject {
 public:
  LimitMaxObject() {
    if (++count > N) throw 1;
  }
  ~LimitMaxObject() { --count; }
  static int count;
};
template <typename T, int N>
int LimitMaxObject<T, N>::count = 0;
```

만약 Mouse 라는 객체를 5개만 만들고 싶다면 다음과 같이 만들면 됩니다.

```
class Mouse : public LimitMaxObject<Mouse, 5> {
};
```

LimitMaxObject는 자식의 타입을 통해, 모든 자식 클래스가 별도의 static 멤버 변수를 물려 받도록 설계되어 있습니다.

핵심

- CRTP(Curiosly Recurring Template Pattern)
- 비 가상 함수를 가상 함수처럼 동작하게 만드는 기술.
- 객체의 개수를 N개로 제한하는 기술.

RTTI

이번 장에서는 RTTI(Run Time Type Information)에 대해서 정리해 봅시다.

```
class Animal {};

class Dog : public Animal {
 public:
  int color;
};

void foo(Animal* p) {
  // p가 Dog인지 조사하고 싶다.
}
```

foo 함수의 인자로 전달되는 포인터 p가 가르키는 타입이 Dog인지 조사하고 싶다면, RTTI를 사용하면 됩니다.

```
void foo(Animal* p) {
  const type_info& t1 = typeid(*p);
  const type_info& t2 = typeid(Dog);

  if (t1 == t2) {
    cout << "p는 Dog 입니다." << endl;
  } else {
    cout << "p는 Animal 입니다." << endl;
  }
}
```

C++에서는 typeid 연산자를 통해 객체와 클래스로부터 type_info 구조체를 얻어올 수 있습니다. p가 Dog 타입을 가르키고 있는지 조사하고 싶다면 객체로부터 꺼내온 type_info 구조체를 Dog 클래스로부터 꺼내온 type_info 구조체와 비교하면 됩니다. 하지만 위의 소스는 제대로 동작하지 않습니다. 왜냐하면 type_info 구조체는 가상 함수 테이블을 통해 관리되기 때문입니다.

```
class Animal {
  virtual ~Animal() {}
};
```

만약 foo 함수에서 p 타입이 Dog라면, Dog가 가지고 있는 고유한 속성인 color에 접근하고자 한다면 캐스팅이 필요합니다. 하지만 잘못된 다운 캐스팅을 조사하기 위해서는 실행 시간에 RTTI를 이용해서 캐스팅을 수행하는 dynamic_cast를 사용해야 합니다.

```
void foo(Animal* p) {
  Dog* pDog = dynamic_cast<Dog*>(p);
  if (pDog == 0)
    cout << "Animal" << endl;
  else
    cout << "Dog" << endl;
}
```

dynamic_cast도 마찬가지로 type_info를 이용하기 때문에, 객체 내부의 가상 함수 테이블이 있어야 합니다. 또한 실행 시간에 타입에 대한 정보를 확인하고, 아니라면 널을 리턴해주는 형태로 동작하는 로직이 포함되어 있기 때문에 느립니다. 꼭 필요한 경우만 사용해야 합니다. 만약 함수의 인자로 오는 p가 특정 자식 타입이라는 것이 확실하다면 컴파일 타임에 캐스팅을 수행하는 static_cast를 사용하면 됩니다. 하지만 static_cast는 잘못된 다운 캐스팅인지 알 수 없기 때문에 주의 해야 합니다.

핵심

- 실행 시간에 특정 타입에 대해서 조사하는 방법.
- 가상 함수 테이블의 type_info 객체를 통해 조사
- typeid 연산자
- static_cast<>
- dynamic_cast<>

참고자료

- 모어 이펙티브 C++ : 항목 24: 가상 함수, 다중 상속, 가상 기본 클래스, RTTI에 들어가는 비용을 제대로 파악하자.

RTTI 구현하기

RTTI의 핵심은 type_info 입니다. 클래스와 객체가 공유하는 정적 멤버 변수를 이용하면 직접 RTTI를 구현하는 것이 가능합니다.

```
struct CRuntimeClass {
  string name;
  string author;
  string version;
};
```

CRuntimeClass를 타입의 정보로 활용하기 위해서는 모든 클래스는 아래 세 가지 규칙을 지켜야만 합니다.
첫번째 규칙. 모든 클래스는 정적 멤버 변수인 CRuntimeClass가 있어야 합니다.
두번째 규칙. 정적 멤버 변수의 이름은 “class클래스이름” 으로 합니다.
세번째 규칙. 정적 멤버 변수를 리턴하는 가상함수 GetRuntimeClass가 있어야 합니다.

```
class Animal {
 public:
  static CRuntimeClass classAnimal;
  virtual CRuntimeClass* GetRuntimeClass() { return &classAnimal; }
};
CRuntimeClass Animal::classAnimal = {"Animal"};

class Dog : public Animal {
 public:
  static CRuntimeClass classDog;
  virtual CRuntimeClass* GetRuntimeClass() { return &classDog; }
};
CRuntimeClass Dog::classDog = {"Dog"};
```

함수의 인자로 오는 p가 Dog 타입인지 조사하기 위해서는 다음과 같이 사용하면 됩니다.

```
void foo(Animal* p) {
  if (p->GetRuntimeClass() == &Dog::classDog) {
    cout << "p는 Dog 입니다." << endl;
  }
}
```

하지만 위의 규칙을 사용자가 직접 구현하는 것은 쉽지 않습니다. 편하게 사용할 수 있도록 자동적으로 코드를 생성해주는 매크로를 제공하는 것이 좋습니다.

```
#define DECLARE_DYNAMIC(classname)       \
 public:                                 \
  static CRuntimeClass class##classname; \
  virtual CRuntimeClass* GetRuntimeClass();

#define IMPLEMENT_DYNAMIC(classname)                          \
  CRuntimeClass classname::class##classname = {#classname}; \
  CRuntimeClass* classname::GetRuntimeClass() { return
&class##classname; }

#define RUNTIME_CLASS(classname) &(classname::class##classname)
```

```
class Animal {
  DECLARE_DYNAMIC(Animal)
};
IMPLEMENT_DYNAMIC(Animal)

class Dog : public Animal {
  DECLARE_DYNAMIC(Dog)
};
IMPLEMENT_DYNAMIC(Dog)

void foo(Animal* p) {
  if (p->GetRuntimeClass() == RUNTIME_CLASS(Dog)) {
    cout << "p는 Dog 입니다.";
  }
}
```

핵심

- 타입의 정보를 가진 정적 멤버 변수
- 정적 멤버 변수를 참조를 리턴하는 가상 함수

컨테이너

이번 장에서는 객체 지향 라이브러리에서 제공하는 컨테이너를 설계하는 다양한 방법에 대해서 정리해 봅시다.

```
struct node {
  int data;
  node* next;
  node(int a, node* n) : data(a), next(n) {}
};

class slist {
  node* head;

 public:
  slist() : head(0) {}

  void push_front(int a) { head = new node(a, head); }
  int front() { return head->data; }
};

int main() {
  slist s;
  s.push_front(10);
  s.push_front(20);
  s.push_front(30);
  cout << s.front() << endl;
}
```

다음은 단일 연결리스트 기반의 컨테이너 입니다. 하지만 위의 컨테이너는 정수 타입만을 담을 수 있습니다. 컨테이너는 한가지 타입 뿐 아니라 다양한 타입을 담을 수 있어야 합니다. 객체 지향 라이브러리에서 다양한 타입을 담을 수 있도록 하는 컨테이너의 설계 방법은 2가지 입니다.
첫번째 설계 방법은 오브젝트 포인터 기반의 컨테이너 입니다.
오브젝트 기반의 컨테이너에서 전제되어야 하는 것은 라이브러리 내의 모든 클래스가 오브젝트 클래스의 자식 클래스이어야 합니다. 자바나 C# 등의 언어에서는 클래스를 만들면 자동적으로 오브젝트 클래스의 자식입니다. 하지만 C++에서는 명시적으로 지정해주어야 합니다.

```
class object
{
public:
    virtual ~object() {}
};
```

```
class Dialog : public object {};
class Point : public object {};
class Rect : public object {};
```

```
struct node {
  object* data;
  node* next;
  node(object* a, node* n) : data(a), next(n) {}
};

class slist {
  node* head;

 public:
  slist() : head(0) {}

  void push_front(object* a) { head = new node(a, head); }
  object* front() { return head->data; }
};

int main() {
  slist s;
  s.push_front(new Point);
  s.push_front(new Point);
  s.push_front(new Point);
  s.push_front(new Dialog);

  Dialog* n = dynamic_cast<Dialog*>(s.front());
}
```

오브젝트 포인터 기반의 컨테이너는 오브젝트의 자식 타입인 모든 객체를 컨테이너에 저장하는 것이 가능합니다. 하지만 이러한 오브젝트 기반의 컨테이너는 몇가지 문제점이 있습니다. 첫번째 문제점은 컨테이너에서 데이터를 꺼낼 때 캐스팅이 필요합니다. 두번째 문제점은 실수로 다른 타입을 집어 넣어도 컴파일 에러가 발생하지 않습니다. 세번째 문제점은 오브젝트 클래스의 자식 타입만 저장 가능하기 때문에 기본 타입(원시 타입)은 저장이 불가능합니다. 자바에서는 이러한 문제를 해결하기 위해서 기본 타입에 대한 래퍼 클래스 타입을 제공합니다. 다음은 오브젝트 포인터 기반의 컨테이너가 가지고 있는 타입 안정성의 문제를 해결하는 템플릿 기반의 컨테이너를 살펴 봅시다.

```
template <typename T>
struct node {
  T data;
  node* next;
  node(const T& a, node* n) : data(a), next(n) {}
};

template <typename T>
class slist {
  node<T>* head;

 public:
  slist() : head(0) {}

  void push_front(const T& a) { head = new node<T>(a, head); }
  T& front() { return head->data; }
};
```

```
int main() {
  slist<int> s;
  s.push_front(10);
```

템플릿 기반 컨테이너는 타입 안정성이 뛰어나고, 객체 뿐 아니라 표준 타입에 대해서 저장이 가능합니다. 잘못된 타입을 집어넣었을 경우 컴파일 에러가 발생합니다. 또한 꺼낼 때 캐스팅이 필요하지 않습니다. 하지만 템플릿 기반 컨테이너는 실제 타입으로 코드를 생성하므로 여러 가지 타입에 대해서 사용하면 slist의 기계어 코드가 많아집니다. 즉 코드 메모리에 오버헤드가 있습니다. 반대로 오브젝트 포인터 기반의 컨테이너는 하나의 구현으로 모든 타입을 저장하기 때문에 코드 메모리에 대한 오버헤드가 존재하지 않습니다.

그렇다면 모바일 환경에서는 어떤 컨테이너 설계를 사용하는 것이 좋을까요? 타입 안정성이 뛰어난 템플릿 기반의 컨테이너는 사용자가 사용하기는 편리하겠지만 제한된 메모리를 가지고 있는 모바일이나 임베디드에는 적합하지 않습니다. 오브젝트 포인터 기반의 컨테이너는 코드 메모리의 사용량은 적지만 많은 문제점을 가지고 있습니다.

다음은 모바일 플랫폼에서 사용하고 있는 Thin Template 기반의 컨테이너 설계 입니다.

```
struct node {
  void* data;
  node* next;
  node(void* a, node* n) : data(a), next(n) {}
};

class slistImpl {
  node* head;

 public:
  slistImpl() : head(0) {}
  void push_front(void* a) { head = new node(a, head); }
  void* front() { return head->data; }
};

template <typename T>
class slist : private slistImpl {
 public:
  inline void push_front(const T& a) {
    slistImpl::push_front(const_cast<T*>(&a));
  }

  inline T& front() { return
*(static_cast<T*>(slistImpl::front())); }
};
```

내부적인 구현은 slistImpl을 통해서 데이터를 void*로 저장합니다. 하지만 void*를 사용자가 직접 다루면 캐스팅에 대한 부담이 있기 때문에, 템플릿 인터페이스를 통해 사용자가 편리할 수 있도록 합니다. 템플릿 인터페이스인 slist는 인라인 함수로 구성되어 있기 때문에, 호출하는 코드가 코드 메모리에 필요하지 않습니다. 즉 코드 메모리 오버헤드가 존재하지 않습니다.
안드로이드 프레임워크 내부의 Vector 클래스가 Thin Template의 설계를 적용한 컨테이너로 구현되어 있습니다.

```
template <class TYPE>
class Vector : private VectorImpl {
  ...
  //! returns number of items in the vector
  inline size_t size() const { return VectorImpl::size(); }
  //! returns whether or not the vector is empty
  inline bool isEmpty() const { return VectorImpl::isEmpty(); }

  inline ssize_t setCapacity(size_t size) {
    return VectorImpl::setCapacity(size);
  }
  ...
};
```

AOSP - Vector.h

```
class VectorImpl {
 public:
  enum {  // flags passed to the ctor
    HAS_TRIVIAL_CTOR = 0x00000001,
    HAS_TRIVIAL_DTOR = 0x00000002,
    HAS_TRIVIAL_COPY = 0x00000004,
  };

  VectorImpl(size_t itemSize, uint32_t flags);
  VectorImpl(const VectorImpl& rhs);
  virtual ~VectorImpl();

  ...
 protected:
  // These 2 fields are exposed in the inlines below,
  // so they're set in stone.
  void* mStorage;  // base address of the vector
  size_t mCount;   // number of items

  const uint32_t mFlags;
  const size_t mItemSize;
};
```

AOSP - VectorImpl.h

핵심

- 오브젝트 포인터 기반의 컨테이너의 장점과 단점
- 템플릿 기반의 컨테이너의 장점과 단점
- Thin Template

스마트 포인터

이번 장에서는 스마트 포인터의 원리에 대해서 이야기 해봅시다.

```
class Car {
 public:
  void go() { cout << "Car go" << endl; }
  ~Car() { cout << "Car 파괴" << endl; }
};

int main() {
  Car* p = new Car;
  p->go();

  delete p;
}
```

특정 객체를 사용할 때, 동적으로 객체의 메모리를 할당했다면 사용 후에 꼭 객체의 메모리를 해지해야 합니다. Car 객체의 메모리를 자동적으로 해주는 클래스를 도입하면 편리하게 사용할 수 있습니다.

```
class Ptr
{
      Car* obj;
public:
      Ptr( Car* p = 0 ) : obj(p) {}
      ~Ptr() { delete obj; }
};

int main() {
  Ptr p = new Car;
}
```

Ptr은 소멸자를 통해서 자신이 가르키고 있는 객체의 delete를 호출해 주기 때문에, 명시적으로 delete 할 필요가 없습니다. 하지만 진짜 포인터처럼 사용하기 위해서는 Ptr 타입을 통해서 Car 객체의 멤버 함수를 호출하거나, 참조할 수 있어야 합니다.

```
class Ptr {
  ...
  Car* operator->() { return obj; }
  Car& operator*() { return *obj; }
};

int main() {
  Ptr p = new Car;
  p->go();
  (*p).go();
}
```

Ptr이 Car객체에 대해서만 가르키는 것이 아니라, 모든 타입에 대해서도 가르킬 수 있도록 템플릿으로 만들어줍니다.

```
template <typename T>
class Ptr {
  T* obj;

 public:
  Ptr(T* p = 0) : obj(p) {}
  ~Ptr() { delete obj; }
  T* operator->() { return obj; }
  T& operator*() { return *obj; }
};

int main() {
  Ptr<Car> p = new Car;
  p->Go();

  (*p).Go();
}
```

이렇게 임의의 객체가 다른 클래스의 포인터의 역활을 수행하는 것을 스마트 포인터라고 합니다. 스마트 포인터의 구현 원리는 -> 연산자와 * 연산자를 재정의해서 포인터처럼 보이게 만드는 것입니다. 스마트 포인터의 장점은 진짜 포인터가 아니라 객체이기 때문에 생성/소멸/복사/대입의 모든 과정에 대해서 마음대로 제어하는 것이 가능합니다. 스마트 포인터의 대표적인 활용은 소멸자에서의 자동 삭제 기능입니다. 또한 C++표준에서는 다양한 스마트 포인터를 제공하고 있습니다. 예를 들면 참조 계수 기반으로 객체를 관리해주는 shared_ptr<> 등이 있습니다.

핵심

- 스마트 포인터의 구현 원리

일반화 프로그래밍

C++ 라이브러리의 설계를 이해하기 위해서는 반드시 일반화(Generic)를 이해해야 합니다. 이번 장에서는 C언어의 함수가 C++로 오면서 어떻게 일반화 될 수 있는지에 대해서 알아봅시다.

다음은 C언어의 대표적인 선형 탐색 알고리즘 함수인 strchr 입니다. 문자열 안에서 검색하고자 하는 문자가 존재하는지 확인할 수 있습니다. 검색이 성공한다면 해당 문자의 위치를 가르키는 포인터를 리턴하고, 실패한다면 널 포인터를 리턴합니다.

```
char* xstrchr(char* s, char c) {
  while (*s != '\0' && *s != c)
      ++s;
  return *s == c ? s : 0;
}

int main() {
  char s[] = "abcdefg";
  char* p = xstrchr(s, 'c');
  if (p ! = 0) cout << *p << endl;
}
```

C++관점에서 위의 함수는 일반적이지 않습니다. 선형적인 구간에서 탐색을 수행하는 알고리즘은 부분 문자열이든 또는 다른 타입의 배열이든 다르지 않습니다. C 언어에서는 별도의 함수로 설계되어 있었지만, C++ 언어에서는 이 모든 것들이 하나의 함수를 통해서 이루어져야 한다고 믿었습니다. 이제부터 일반화의 설계를 적용해서 xstrchr의 함수를 일반화 해봅시다.

먼저 xstrchr은 널문자로 끝나는 문자열 구간에서 검색하도록 설계되었기 때문에, 부분 문자열에 대한 검색이 불가능합니다. 첫번째로 부분 문자열의 검색이 가능하도록 검색 구간의 일반화를 적용하겠습니다.

```
char* xstrchr(char* first, char* last, char value) {
  while (first != last && *first != value)
      ++first;

  return first;
}
```

이제 xstrchr 함수는 더 이상 널 문자로 끝나는 문자열 시작을 인자로 받는 것이 아니라, 검색 구간의 시작과 끝 다음을 가르키는 포인터를 인자로 받기 때문에 부분 문자열에 대한 검색이 가능합니다. 또한 널 포인터로 실패를 표현하는 것이 아니라 first가 last에 도달하는 것으로 실패를 표현하도록 설계하였습니다.

```
int main() {
  char s[] = "abcdefg";

  char* p = xstrchr(s, s + 4, 'c');
  if (p != s + 4)
      cout << *p << endl;
}
```

마지막으로 함수가 문자열에서만 검색이 아닌, 모든 타입에 대해서 검색이 가능하도록 템플릿을 도입하여 검색 대상 타입에 대한 일반화 설계를 적용합니다. 모든 타입에 대해서 검색이 가능하므로 함수의 이름도 xfind라고 하겠습니다.

```
template <typename T>
T* xfind(T* first, T* last, T value) {
  while (first != last && *first != value) ++first;

  return first;
}
```

하지만 위의 형태로 템플릿 인자를 설계하면, 문제가 있습니다. 바로 구간의 타입과 찾는 요소의 타입이 연관되어 있기 때문에, double 배열에서 int 타입을 찾는 것이 불가능합니다. 따라서 구간을 표현하는 타입과 구간의 요소 타입이 별도의 템플릿 인자로 분리되어야 합니다.

```
template <typename T, typename F>
T* xfind(T* first, T* last, F value) {
  while (first != last && *first != value) ++first;

  return first;
}
```

구간을 표현하는 타입이 T*으로 제한되어 있으면 진짜 포인터만 사용할 수 있습니다. 객체의 포인터의 역활을 수행하는 스마트 포인터는 사용할 수 없습니다.

```
template <typename T, typename F>
T xfind(T first, T last, F value) {
  while (first != last && *first != value) ++first;

  return first;
}
```

핵심

- 일반화 프로그래밍 설계 철학
- 검색 구간의 일반화
- 검색 대상 타입의 일반화

참고 자료

일반적 프로그래밍과 STL

반복자

GoF의 디자인 패턴에서 반복자 패턴의 의도는 다음과 같습니다.
“내부 표현부를 노출하지 않고 어떤 객체의 집합에 속한 원소들을 순차적으로 접근할 수 있는 방법을 제공하는 패턴입니다.”

디자인 패턴에서 말하는 반복자의 의도는 내부 요소를 열거할 수 있는 방법을 제공하는 것 입니다. 하지만 C++에서는 내부 요소를 열거하는 기능을 제공할 뿐 아니라 일반적 알고리즘 함수들과 컨테이너를 연결해주는 경계면의 역활을 수행합니다.

```
template <typename T, typename F>
T xfind(T first, T last, F value) {
  while (first != last && *first != value)
      ++first;

  return first;
}
```

이전 장에서 설계한 일반적 알고리즘 함수인 xfind는 배열에 대해서만 동작하도록 설계된 것이 아닙니다. 컨테이너의 내부 구간을 표현하는 타입 T가 find 함수가 내부적으로 사용하는 몇가지 연산(!=, *, ++) 등을 연산자 오버로딩을 통해 만족한다면, 컨테이너의 내부 요소를 가르키는 스마트 포인터, 즉 반복자에 대해서도 사용이 가능합니다.

```
template <typename T>
struct node {
  T data;
  node* next;
  node(const T& a, node* n) : data(a), next(n) {}
};

template <typename T>
class slist {
  node<T>* head;

 public:
  slist() : head(0) {}

  void push_front(const T& a) { head = new node<T>(a, head); }
  T& front() { return head->data; }
};
```

이번 장에서는 템플릿 기반 컨테이너에 대해서 내부 요소를 열거하는 기능과 STL 일반화 알고리즘 함수에 대해서도 동작가능한 반복자를 만들어 봅시다.

```
template <typename T>
class slist_iterator {
  node<T>* current;

 public:
  inline slist_iterator(node<T>* p = 0) : current(p) {}

  inline T& operator*() { return current->data; }
  inline slist_iterator& operator++() {
    current = current->next;
    return *this;
  }

  inline bool operator==(const slist_iterator& t) {
    return current == t.current;
  }

  inline bool operator!=(const slist_iterator& t) {
    return current != t.current;
  }
};
```

STL 스타일의 반복자는 구간을 이동하거나 참조하거나, 비교하는 모든 연산이 연산자 재정의를 통해서 제공되어야 합니다. 그래야만 일반적 알고리즘 함수에 대해서 사용이 가능합니다. 다음으로 모든 컨테이너 설계자는 자신의 시작을 가르키는 반복자와 끝 다음을 가르키는 반복자를 제공하는 함수인 begin과 end를 제공해야 합니다. 또한 구체 타입인 slist_iterator가 아닌 일관된 방법으로 반복자를 다룰 수 있도록 자신의 설계한 반복자를 iterator라는 이름으로 외부에 알려주어야 합니다.

```
template <typename T>
class slist {
  Node<T>* head;

 public:
  typedef slist_iterator<T> iterator;

  slist() : head(0) {}

  void push_front(const T& a) { head = new Node<T>(a, head); }
  T& front() { return head->data; }

  iterator begin() { return iterator(head); }
  iterator end()   { return iterator(0);    }
};

int main() {
  slist<int> s;
  s.push_front(10);
  s.push_front(20);
  s.push_front(30);

  slist<int>::iterator p = xfind(s.begin(), s.end(), 20);
  if (p != s.end()) cout << *p << endl;
}
```

C++에서 반복자를 설계하는 다른 방법은 Java 또는 C# 에서 제공하고 있는 인터페이스 기반의 설계입니다. 인터페이스 기반의 설계는 반복자가 제공하고 있는 모든 연산이 인터페이스로 미리 약속된 형태로 제공됩니다. 그리고 반복자를 다음 위치로 이동하는 기능과 요소에 참조하는 기능은 연산자 재정의가 아닌 인터페이스의 가상 함수로 제공합니다.

```
template <typename T>
struct IEnumerator {
  virtual bool MoveNext() = 0;
  virtual T& GetObject() = 0;

  virtual ~IEnumerator() {}
};

template <typename T>
struct slist_enumerator : public IEnumerator<T> {
  node<T>* current;

 public:
  slist_enumerator(node<T>* p = 0) : current(p) {}

  virtual bool MoveNext() {
    current = current->next;
    return current != 0;
  }

  virtual T& GetObject() { return current->data; }
};
```

반복자를 제공하는 컨테이너의 책임 또한 인터페이스를 통해 미리 약속되어야 합니다.

```
template <typename T>
struct IEnumerable {
  virtual IEnumerator<T>* GetEnumeratorN() = 0;
  virtual ~IEnumerable() {}
};

template <typename T>
class slist : public IEnumerable<T> {
  node<T>* head;

 public:
  slist() : head(0) {}

  virtual IEnumerator<T>* GetEnumeratorN() {
    return new slist_enumerator<T>(head);
  }

  void push_front(const T& a) { head = new node<T>(a, head); }
  T& front() { return head->data; }
};
```

하지만 이러한 인터페이스 기반의 반복자는 연산자 재정의를 통해 반복자의 기능을 제공하는 것이 아니라 가상 함수를 통해 제공하고 있기 때문에 STL의 일반화 알고리즘 함수를 이용할 수 없습니다. 별도의 함수가 제공되어야 합니다.

```
template <typename T>
void Show(IEnumerator<T>* p) {
  do {
    cout << p->GetObject() << endl;
  } while (p->MoveNext());
}

int main() {
  slist<int> s;
  s.push_front(10);
  s.push_front(20);
  s.push_front(30);

  IEnumerator<int>* p = s.GetEnumeratorN();
  Show(s.GetEnumeratorN());
}
```

핵심

- STL 기반 반복자 특징
- 인터페이스 기반 반복자 특징

참고 자료

- 일반적 프로그래밍과 STL

방문자

```
template <typename T>
struct node {
  T data;
  node* next;
  node(const T& a, node* n) : data(a), next(n) {}
};

template <typename T>
class slist {
  node<T>* head;

 public:
  slist() : head(0) {}
  void push_front(const T& a) { head = new node<T>(a, head); }
  T& front() { return head->data; }
};

int main() {
  slist<int> s;
  s.push_front(10);
  s.push_front(20);
  s.push_front(30);

  cout << s.front() << endl;
}
```

s안에 있는 모든 요소를 2배로 하고 싶다면 가장 쉬운 방법은 루프를 수행하면서 모든 요소를 2배로 해서 다시 넣으면 됩니다. 하지만 이러한 작업이 반복된다면 멤버 함수로 추가하는 것이 좋습니다. 이러한 작업이 slist 뿐 아니라 모든 컨테이너에 추가되어야 한다면 방문자 패턴을 사용하는 것이 좋습니다. GoF의 디자인 패턴에서 방문자 패턴의 의도는 다음과 같습니다.
"객체 구조를 이루는 원소에 대해 수행할 연산을 표현한느 패턴으로, 연산을 적용할 원소의 클래스를 변경하지 않고도 새로운 연산을 정의할 수 있게 합니다."

방문자 패턴은 두 개의 인터페이스가 필요합니다. 하나는 요소에 연산을 수행하는 방문자의 인터페이스와 또 다른 하나는 방문자를 받아들이는 수락자의 인터페이스가 필요합니다.

```
template <typename T>
struct IVisitor {
  virtual void visit(T& a) = 0;
  virtual ~IVisitor() {}
};
```

```
template <typename T>
struct IAcceptor {
  virtual void accept(IVisitor<T>* visitor) = 0;
  virtual ~IAcceptor() {}
};
```

컨테이너에서 구현해야 하는 인터페이스를 방문자를 받아들이는 수락자의 인터페이스 입니다.

```
template <typename T>
class slist : public IAcceptor<T> {
  node<T>* head;

 public:
  virtual void accept(IVisitor<T>* visitor) {
    node<T>* current = head;

    while (current) {
      visitor->visit(current->data);
      current = current->next;
    }
  }

  slist() : head(0) {}

  void push_front(const T& a) { head = new node<T>(a, head); }
    T& front() { return head->data; }
};
```

accept 함수에서 컨테이너를 순회하면서 모든 요소에 대해서 방문자에게 전달합니다. 마지막으로 다양한 방문자의 인터페이스를 구현하는 다양한 방문자를 제공하면 됩니다.

```
template <typename T>
class TwiceVisitor : public IVisitor<T> {
 public:
  virtual void visit(T& a) { a *= 2; }
};

template <typename T>
class ShowVisitor : public IVisitor<T> {
 public:
  virtual void visit(T& a) { cout << a << endl; }
};

int main() {
  ...
  TwiceVisitor<int> tv;
  s.accept(&tv);

  ShowVisitor<int> sv;
  s.accept(&sv);

  cout << s.front() << endl;
}
```

방문자 패턴을 사용하면 컨테이너를 수정하지 않고도 다양한 기능을 추가하는 것이 가능합니다. 뿐만 아니라 수행되어야 하는 연산은 수락자의 인터페이스를 구현하고 있는 기존 컨테이너의 수정 없이 연산을 수행하는 것이 가능합니다.
방문자 패턴은 복합 객체에 기능을 추가할 때도 유용하게 사용 가능합니다. 이전에 만들었던 메뉴 시스템에 방문자 패턴을 적용해 봅시다.

```
struct IMenuVisitor {
  virtual void visit(PopupMenu* p) = 0;
  virtual void visit(MenuItem* p) = 0;

  virtual ~IMenuVisitor() {}
};

struct IMenuAcceptor {
  virtual void accept(IMenuVisitor* visitor) = 0;
  virtual ~IMenuAcceptor() {}
};

class BaseMenu : public IMenuAcceptor {
  string title;
 public:
  BaseMenu(string s) : title(s) {}
  string getTitle() const { return title; }
  ...
};

class MenuItem : public BaseMenu {
 ...
 public:
  virtual void accept(IMenuVisitor* p) {
    p->visit(this);
  }
 ...
};

class PopupMenu : public BaseMenu {
  vector<BaseMenu*> v;
  ...
 public:
  virtual void accept(IMenuVisitor* p) {
    p->visit(this);

    for (int i = 0; i < v.size(); ++i) v[i]->accept(p);
  }
  ...
};

class TitleDecoratorVisitor : public IMenuVisitor {
 public:
  virtual void visit(MenuItem* p) {}
  virtual void visit(PopupMenu* p) {
    string s = p->getTitle() + "    >";
    p->setTitle(s);
  }
};
```

타이틀을 수정하고자 하는 방문자를 적용하는데 문제가 발생했습니다. 메뉴에서 타이틀을 수정할 수 있는 함수를 제공해주고 있지 않기 때문입니다. 이처럼 방문자를 적용하기 위해서는 방문자를 받아들이는 수락자는 모든 내부의 필드에 대해서 방문자에게 공개할 수 밖에 없습니다. 즉 방문자를 통해 초기 설계의 캡슐화의 전략에 위배가 발생할 수 있습니다.

핵심

방문자 패턴

참고 자료

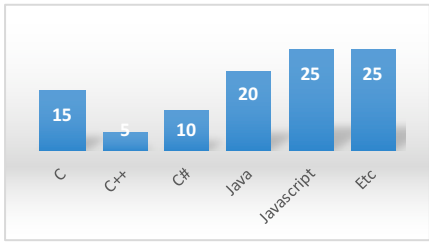
GoF 디자인 패턴: 방문자 패턴

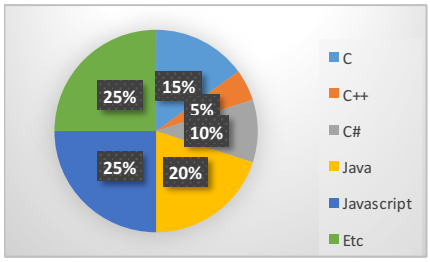
관찰자

이번 장에서는 GoF의 관찰자 패턴에 대해서 정리해 봅시다. GoF의 관찰자 패턴은 다음과 같은 의도를 가지고 있습니다.
“객체들 사이에 일 대 다의 의존 관계를 정의해 두어, 어떤 객체의 상태가 변할 때 그 객체에 의존성을 가진 다른 객체들이 그 변화를 통지받고 자동으로 갱신 될 수 있게 만드는 패턴입니다.”

다음과 같은 데이터가 있고 데이터를 효과적으로 표현하는 그래프가 있습니다.

C	15
C++	5
C#	10
Java	20
Javascript	25
Etc	25





각각의 그래프가 표로부터 데이터를 얻어오기 위해 주기적으로 질의하는 것은 비효율적입니다. 관찰자 패턴에서는 표가 데이터가 변경되었을 때, 등록된 표 객체에게 데이터가 변경되었다는 사실을 알려줍니다.
관찰자의 인터페이스는 다음과 같습니다. 데이터가 변경되었을 때 표 객체에서 등록된 객체에게 데이터가 변경되었다는 사실을 onUpdate 호출을 통해 알려줍니다.

```
struct IObserver {
  virtual void onUpdate(void* data) = 0;
  virtual ~IObserver() {}
};
```

```
class Subject {
  vector<IObserver*> v;
 public:
  void attach(IObserver* p) {
    v.push_back(p);
  }
  void detach(IObserver* p) {}
  void notify(void* p) {
    for (int i = 0; i < v.size(); ++i) v[i]->onUpdate(p);
  }
};

class Table : public Subject {
  int data[5];

 public:
  Table() { memset(data, 0, sizeof(data)); }

  void edit() {
    while (1) {
      int index;
      cout << "index : ";
      cin >> index;
      cout << "data  : ";
      cin >> data[index];

      notify(data);
    }
  }
};

class PieGraph : public IObserver {
 public:
  virtual void onUpdate(void* p) {
    ...
  }
};

class BarGraph : public IObserver {
 public:
  virtual void onUpdate(void* p) {
    ...
  }
};

int main() {
  Table table;
  PieGraph pg;
  BarGraph bg;

  table.attach(&pg);
  table.attach(&bg);
  table.edit();
}
```

지금부터는 그래프의 제작자가 그래프를 동적 로딩 라이브러리, 즉 별도의 모듈로 제공한다면 어떤 설계가 적용되어야 하는지에 대해서 이야기 해봅시다.

다음은 동적 로딩 라이브러리로 제공되는 그래프의 소스 입니다.

```
// SampleGraph.cpp
#include "IObserver.h"

class SampleGraph : public IObserver {
 public:
  void OnUpdate(void* p) {
    int* data = static_cast<int*>(p);
    cout << "******* Sample Graph *********" << endl;
    for (int i = 0; i < 5; i++) {
      cout << i << " : " << data[i] << endl;
    }
  }
};

extern "C" __declspec(dllexport) IObserver* CreateGraph() {
  return new SampleGraph;
}
```

SampleGraph라는 실제 그래프의 이름은 현재 동적 로딩 라이브러리를 만드는 사람만 알고 있습니다. 실행 파일에서는 절대 이름을 알 수 없습니다. 클래스의 이름을 모르면 절대 객체를 생성할 수 없습니다. 따라서 동적 로딩 라이브러리내에서 객체를 생성하기 위한 함수 하나가 약속되어야 합니다. 위의 예제에서는 CreateGraph 함수가 그 역활을 담당합니다. 그리고 CreateGraph가 리턴하는 타입은 구체 타입인 SampleGraph가 아닙니다. 약속된 관찰자 인터페이스 형태로 리턴해야 합니다. 그래야만 모듈과 실행 파일 간에 느슨한 결합이 형성되고 SampleGraph의 구현이 바뀌어도 실행 파일은 재컴파일 될 필요가 없습니다.

이제는 관찰자를 관리하는 Subject 클래스에서 약속된 폴더 안의 그래프가 구현되어 있는 동적 로딩 라이브러리 안에서 그래프를 생성하는 함수를 찾아서 생성하고, 관찰자로 등록시켜 주면 됩니다.

```
class Subject {
 public:
  Subject() { IoEnumFiles("D:\\PlugIn", "*.dll", LoadModule, this); }

  static int LoadModule(string name, void* param) {
    cout << name << endl;
    void* addr = IoLoadModule(name);
    typedef IObserver* (*FP)();

    FP f = (FP)IoGetFunctionAddress(addr, "CreateGraph");
    IObserver* p = f();
    Subject* pThis = static_cast<Subject*>(param);
    pThis->attach(p);
    return 1;
  }
  ...
};
```

핵심

- 관찰자 패턴의 의도
- 플러그인의 원리

참고 자료

- GoF 디자인 패턴 : 관찰자 패턴

통보 센터

통보 센터는 프로그램 상의 많은 이벤트와 이벤트 핸들러의 코드를 통합적으로 관리하는 기능을 제공하는 객체로서 매킨토시 라이브러리인 코코아가 제공하는 기능입니다.

C++에서 콜백을 처리하는 방법은 두가지 모양이 있습니다. 첫번째는 인터페이스 기반으로 객체를 등록하는 설계 방법과 함수 포인터 기반으로 함수를 등록하는 두번째 모양이 있습니다.
여기서는 C++11의 범용 함수 포인터 클래스 타입인 function<>을 이용한 함수 포인터 기반의 설계를 적용하였습니다.

```
void foo() { cout << "foo" << endl; }
void goo(int a) { cout << "goo" << endl; }

class NotificationCenter {
  typedef function<void()> HANDLER;
  map<string, vector<HANDLER>> notif_map;

 public:
  void addObserver(string key, HANDLER handler) {
    notif_map[key].push_back(handler);
  }

  void postNotificationWithName(string key) {
    vector<HANDLER>& v = notif_map[key];
    for (int i = 0; i < v.size(); i++) v[i]();
  }

  static NotificationCenter& defaultCenter() {
    static NotificationCenter instance;
    return instance;
  }
};

int main() {
  NotificationCenter& globalCenter =
NotificationCenter::defaultCenter();

  NotificationCenter nc;
  nc.addObserver("LOWBATTERY", &foo);
  nc.addObserver("LOWBATTERY", bind(&goo, 5));

  nc.postNotificationWithName("LOWBATTERY");
}
```

통보 센터는 복잡한 객체와 객체간의 관계를 캡슐화하는 중재자의 역활을 수행합니다. GoF의 디자인 패턴에서 중재자 패턴의 의도는 다음과 같습니다.

"한 집합에 속해있는 객체들의 상호 작용을 캡슐화하는 객체를 정의하는 패턴입니다. 객체들이 직접 서로를 참조하지 않도록 함으로써 객체들 사이의 느슨한 결합을 촉진시키며, 개발자가 객체들의 상호 작용을 독립적으로 다양화 시킬 수 있게 만듭니다."

핵심

중재자 패턴

참고 자료

GoF 디자인 패턴: 중재자 패턴
Cocoa Design Patterns: Chapter 14. Notification

책임의 전가

4장의 윈도우 클래스에 자식 윈도우를 포함할 수 있는 기능을 추가해봅시다. 하나의 윈도우는 여러 개의 자식 윈도우를 가질 수 있고, 한 개의 부모 윈도우를 가질 수 있습니다. 하지만 실제로 부모 윈도우에 자식 윈도우를 붙이는 기능은 C언어 함수를 통해 구현되어야 합니다.

```
class Window {
  int handle;
  static map<int, Window*> this_map;

  Window* parent;
  vector<Window*> children;

 public:
  Window() : parent(0) {}

  void addChild(Window* p) {
    p->parent = this;
    children.push_back(p);

    IoAddChild(this->handle, p->handle);
  }
};

class MyWindow : public Window {
 public:
  virtual void onLButtonDown() { cout << "lbutton" << endl; }
};

class ImageView : public Window {
 public:
  virtual void onLButtonDown() { cout << "Image view Touch!!!" <<
endl; }
};

int main() {
  MyWindow w;
  ImageView imageView;

  imageView.create();
  w.create();
  w.addChild(&imageView);
  IoProcessMessage();
}
```

이제 부모 윈도우와 자식 윈도우의 관계로 설정되었기 때문에, 독립된 윈도우가 두개 생성되는 것이 아니라 부모 윈도우 안에 자식 윈도우가 포함된 형태로 윈도우가 생성됩니다. 만약 자식 윈도우가 부

모 윈도우를 완벽하게 덮은 형태로 UI가 제공된다면, 절대 부모 윈도우의 클릭 이벤트는 절대 발생하지 않습니다. 위의 문제를 해결하기 위해서는 이벤트가 발생하였을 때, 해당하는 이벤트가 자식 윈도우가 처리해야 할 이벤트가 아니라면 부모 윈도우에게 전달해주어야 합니다.

```
class Window {
  ...
  static int foo(int handle, int msg, int param1, int param2) {
    Window* self = this_map[handle];
    switch (msg) {
      case WM_LBUTTONDOWN:
        self->fireLButtonDown();
        break;
      case WM_KEYDOWN:
        self->onKeyDown();
        break;
    }
    return 0;
  }

  void fireLButtonDown() {
    if (onLButtonDown() == true) return;
    if (parent != 0) parent->fireLButtonDown();
  }

  virtual bool onLButtonDown() { return false; }
  virtual void onKeyDown() {}
};

map<int, Window*> Window::this_map;

class MyWindow : public Window {
 public:
  virtual bool onLButtonDown() {
    cout << "lbutton" << endl;
    return true;
  }
};

class ImageView : public Window {
 public:
  virtual bool onLButtonDown() {
    cout << "Image view Touch!!!" << endl;
    return false;
  }
};
```

만약 이벤트가 발생했을 때, 그 이벤트가 처리되어야 하는 다음 객체에게 전달해주는 패턴을 책임의 전가 패턴이라고 부릅니다. GoF의 디자인 패턴에서 책임의 전가 패턴의 의도는 다음과 같습니다.
“요청을 처리할 수 있는 기회를 하나 이상의 객체에게 부여하여 요청을 보내는 객체와 그 요청을 받는 객체 사이의 결합을 피하는 패턴입니다. 요청을 받을 수 있는 객체를 연쇄적으로 묶고, 실제 요청을 처리할 객체를 만날 때까지 객체 고리를 따라서 요청을 전달합니다.”

책임의 전가 패턴은 많은 GUI 라이브러리에서 보편적으로 구현되어 있습니다. 그 중 MFC의 메뉴 메세지 View, Document, Frame, App 순서로 전달됩니다. 마찬가지로 iOS, OS X 계열의 코코아 라이브러리의 UI 이벤트가 리스폰더 체인이라는 이름의 책임의 전가 패턴으로 구현되어 있습니다.

핵심

- 책임의 전가 패턴.

참고 자료

- GoF 디자인 패턴: 책임의 전가 패턴
- Cocoa Design Pattern: Chapter 18.리스폰더 체인

싱글톤(Singleton)

이 장에서는 C++ 에서 싱글톤을 만드는 기술을 정리해봅시다.

싱글톤은 GoF 의 디자인 패턴중 가장 유명한 패턴입니다.
“클래스 인스턴스는 오직 하나임을 보장하며
이에 대한 접근은 어디에서든지 하나로만 통일하여 제공한다.”

C++ 에서 싱글톤을 만드는 가장 일반적인 방법은 Effective C++ 에서 스콧 마이어스에 의해 언급된 정적 멤버 함수 내부에 객체를 만드는 것 입니다. 그의 이름을 따서 마이어스의 싱글톤(Meyers’s Singleton) 이라고 부릅니다.

Cursor 라는 객체를 프로그램 상에서 오직 한개만을 생성하고 싶다면 아래와 같이 구현 가능합니다.

```
class Cursor {
 private:
  Cursor() {}

  Cursor(const Cursor&);
  void operator=(const Cursor&);

 public:
  static Cursor& getInstance() {
    static Cursor instance;
    return instance;
  }
};
```

싱글톤을 만드는 방법은 규칙 3가지가 중요합니다.
첫번째 규칙. private 생성자
두번째 규칙. 오직 한개의 인스턴스의 참조만을 리턴하는 정적 멤버 함수
세번째 규칙. 복사와 대입 금지

첫번째 규칙인 private 생성자는 클래스 외부에서 혹시 모를 객체의 생성을 방지하고, 두번째 규칙은 내부 정적 객체로 생성된 유일한 인스턴스에 접근하는 인터페이스를 제공합니다. 마지막 규칙은 C++ 에서 자동적으로 생성되는 복사 생성자와 대입 연산자의 선언만을 두는 복사 금지 기법을 통해 클래스 외부에서 객체의 복사를 방지하며, 클래스 내부에서 혹시 모를 복사와 대입을 방지하기 위해 선언만 두는 형태로 구현하는 것입니다. 만약 클래스 내부에서 문제가 발생한다면 링킹 에러를 통해 싱글톤이 깨지는 것을 방지할 수 있습니다.

C++ 에서 복사와 대입을 선언만을 통해 방지하는 기법은 C++11 에서는 delete function 이라는 문법을 통해 구현할 수 있습니다.

```
class Cursor {
 private:
  Cursor() {}

  Cursor(const Cursor&) = delete;
  void operator=(const Cursor&) = delete;

 public:
  static Cursor& getInstance() {
    static Cursor instance;
    return instance;
  }
};
```

delete function 을 통해 자동적으로 생성하는 복사 생성자와 대입 연산자를 삭제할 수 있으며, 클래스 외부에서의 접근 뿐 아니라 클래스 내부에서 발생되는 복사 및 대입 연산자의 접근을 컴파일 에러를 통해서 검출할 수 있습니다.

위의 싱글톤을 생성하는 규칙과 복사 금지의 코드를 구현하는 방법은 항상 동일하기 때문에, 매크로를 통해서 일반화한다면 또 다른 싱글톤 또는 복사 금지가 필요할 때 유용하게 사용할 수 있습니다.

```
#define MAKE_NOCOPY(classname)           \
 private:                                \
  classname(const classname&) = delete; \
  void operator=(const classname&) = delete;

#define MAKE_SINGLETON(classname)    \
 private:                            \
  MAKE_NOCOPY(classname)             \
  classname() {}                     \
                                     \
 public:                             \
  static classname& getInstance() { \
    static classname instance;       \
    return instance;                 \
  }

class Mouse {
  MAKE_SINGLETON(Mouse)
};
```

마이어스의 싱글톤은 C++11/14 부터는 static 정적 객체의 초기화 과정에 대한 다중 스레드 안정성 까지 보장되기 때문에 안전하게 사용할 수 있습니다.

> If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.
>
> §6.7 [stmt.dcl]

또한 GCC 컴파일러는 별도의 컴파일 옵션(-fthreadsafe-statics)을 통해 함수 내부의 정적 객체에 대한 스레드 안정성 보장을 위한 기능도 제공하고 있습니다. 하지만 C++98/03 표준에서는 스레드 안전성에 대한 언급이 존재하지 않기 때문에 스레드 안전성에 대한 추가적인 고려가 필요합니다.

싱글톤의 생성 시점에 스레드 안전성을 보장하기 위해서 가장 쉬운 접근 방법은 API design for C++에서 언급된 정적 초기화 방법이나 명시적 API 초기화 방법을 사용하는 것도 좋은 방법이 될 수 있습니다.

```
// main이 처음 호출되었을 시점에는 단일 스레드 이므로 안전하다.
int main() {
  Cursor::getInstance();
}
```

많은 C++ 기반의 오픈 소스도 마이어스의 싱글톤을 사용하고 있습니다.

```
CSSValuePool& CSSValuePool::singleton()
{
    static NeverDestroyed<CSSValuePool> pool;
    return pool;
}
```

Webkit(./Source/WebCore/css/CSSValuePool.cpp)

```
/* static */
OpenSLESProvider& OpenSLESProvider::getInstance()
{
  static OpenSLESProvider instance;
  return instance;
}
```

Firefox(./dom/media/systemservices/OpenSLESProvider.cpp)

Cursor 싱글톤 인스턴스를 힙에 생성하기 위해서는 다음과 같이 구현할 수 있습니다.

```
class Cursor {
 private:
  Cursor() {}
  Cursor(const Cursor&);
  void operator=(const Cursor&);

  static Cursor* sInstance;

 public:
  static Cursor& getInstance() {
    if (sInstance == 0) sInstance = new Cursor;
    return *sInstance;
  }
};

// static 멤버 변수는 반드시 외부에 선언을 해야 합니다.
Cursor* Cursor::sInstance = 0;

int main() { Cursor& c1 = Cursor::getInstance(); }
```

하지만 위의 싱글톤은 멀티 스레드에서 getInstance() 가 호출될 경우 인스턴스가 여러개 생성되는 문제점이 있습니다. 따라서 Mutex 를 통해 sInstance 의 인스턴스를 체크하고 new 를 통한 객체 생성을 보호해주어야 합니다.

```
class Mutex {
 public:
  void lock() { cout << "Mutex Lock" << endl; }
  void unlock() { cout << "Mutex Unlock" << endl; }
};
```

```
class Cursor {
 private:
  static Cursor* sInstance;
  static Mutex sLock;

 public:
  static Cursor& getInstance() {
    sLock.lock();
    if (sInstance == 0) sInstance = new Cursor;
    sLock.unlock();
    return *sInstance;
  }
};

Mutex Cursor::sLock;
Cursor* Cursor::sInstance = 0;
```

Mutex 를 통해 싱글톤의 객체 생성 시점을 보호해준다면, 다중 스레드 상에서 객체가 여러개가 생성되는 문제를 방지할 수 있습니다. 하지만 위처럼 뮤텍스를 lock 과 unlock 의 명시적 호출의 형태로

사용한다면 예외가 발생할 경우 데드락의 위험이 있습니다. Cursor 의 인스턴스를 할당하는 new 연산은 가용한 메모리가 없다면 std::bad_alloc 이라는 예외가 발생하게 됩니다. 예외가 발생하게 되면 뮤텍스의 잠금을 푸는 코드가 실행되지 않고 getInstance() 를 빠져나가게 됩니다. 만약 다른 스레드에 의해서 getInstance 가 호출된다면 뮤텍스의 잠금은 영원히 해지할 수 없기 때문에 데드락에 빠지게 될 것입니다.

C++ 에서는 뮤텍스 같은 사용 후 꼭 해지되어야 하는 자원에 대해서는 명시적인 잠금과 해지를 사용하기보다는 RAII(Resource Acquisition Is Initialization) 을 통해 사용하는 것이 일반적입니다.

```
class Mutex {
 public:
  void lock() { cout << "Mutex lock" << endl; }
  void unlock() { cout << "Mutex unlock" << endl; }

  class Autolock {
    Mutex& mLock;

   public:
    Autolock(Mutex& m) : mLock(m) { mLock.lock(); }
    ~Autolock() { mLock.unlock(); }
  };
};
```

```
static Cursor& getInstance() {
  Mutex::Autolock al(sLock);
  if (sInstance == 0) sInstance = new Cursor;

  return *sInstance;
}
```

위의 방법을 통해 뮤텍스를 사용한다면, al 이라는 지역 객체의 생성자를 통해 lock 이 호출되고 소멸자를 통해 unlock 이 자동적으로 호출됩니다. 또한 객체를 초기화하는 중 예외가 발생한다고 해도 C++ 표준에서는 예외 발생으로 인해 함수를 빠져나갈 때, 지역 객체에 대한 소멸자 호출을 보장하고 있기 때문에 al의 소멸자는 정상적으로 호출될 것 입니다. 따라서 데드락은 발생하지 않을 것입니다. 이처럼 생성자와 소멸자를 통해 자원을 관리하는 RAII 는 자원 해지를 편리하게 사용할 수 있을 뿐 아니라 예외가 발생하였을 경우 자원 누수 및 데드락을 방지하는데 유용하게 사용됩니다.

싱글톤의 getInstance 를 뮤텍스를 통해 동기화하면 다중 스레드에 의해 객체가 여러개 생성되는 것을 방지할 수는 있지만, 싱글톤 인스턴스에 접근하기 위해서는 결국 뮤텍스 동기화가 필요하므로 성능적으로 저하가 있습니다. 그래서 생성 이후에 불필요한 뮤텍스 동기화를 방지하기 위해 사용하는 방법이 있습니다.

```
static Cursor& getInstance() {
  if (sInstance == 0) {
    Mutex::Autolock al(sLock);
    if (sInstance == 0) sInstance = new Cursor;
  }
  return *sInstance;
}
```

위의 방법은 흔히 Double Checked Locking Pattern(DCLP) 로 알려져 있습니다. 하지만 위의 방법은 멀티 프로세서 머신에서 CPU의 명령어 비순차 처리에 의해 제대로 동작하지 않습니다.
(참고 : C++ and the Perils of Double-Checked Locking, 2004)
결국 DCLP를 제대로 동작하게 위한 방법은 C++98/03 에서는 제공되지 않기 때문에, 절대 사용하면 안됩니다. 다행스럽게도 C++11/14 에서는 메모리 장벽에 대한 기능이 제공되기 때문에 DCLP 를 안전하게 구현 가능하지만, 아래의 방법을 통해 힙에 생성하는 싱글톤을 좀 더 쉽게 구현할 수 있습니다.

```
static Cursor& getInstance() {
  static Cursor* instance = new Cursor;
  return *instance;
}
```

C++11/14 Heap Singleton

안드로이드 프레임워크는 CRTP(Curiosly Recurring Template Pattern)을 통해 힙에 생성하는 싱글톤을 일반화하여 사용하고 있습니다.

```
template <typename TYPE>
class ANDROID_API Singleton
{
public:
    static TYPE& getInstance() {
        Mutex::Autolock _l(sLock);
        TYPE* instance = sInstance;
        if (instance == 0) {
            instance = new TYPE();
            sInstance = instance;
        }
        return *instance;
    }

    static bool hasInstance() {
        Mutex::Autolock _l(sLock);
        return sInstance != 0;
    }

protected:
    ~Singleton() { };
    Singleton() { };

private:
    Singleton(const Singleton&);
    Singleton& operator = (const Singleton&);
    static Mutex sLock;
    static TYPE* sInstance;
};
```

AOSP Singleton

```
class Cursor : public Singleton<Cursor> {

};
```

싱글톤은 프로그램 상에서 하나만 존재하지 않는 자원을 표현할 때 유용하게 사용되는 패턴입니다. 또한 여러 리소스에 접근하기 위한 하나의 포인트를 제공하는 매니저 클래스를 구현할 때도 유용하게 사용됩니다. 하지만 싱글톤은 편리함을 제공하는 대신 클래스 간의 결합도를 높이기 때문에, 이후에 프로그램을 리팩토링 하거나 격리된 부분을 테스트 하기 위한 단위 테스트 코드 작성도 어렵게 만듭니다. ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ

핵심

- 싱글톤을 만드는 규칙 3가지
- 힙에 만드는 싱글톤
- 스레드 안전성에 대한 고려
- C++11/14 싱글톤

참고 자료

- Effective C++ 항목4: 객체를 사용하기 전에 반드시 그 객체를 초기화하자.
- API design for C++ 3.2 싱글톤
- C++ 표준 문서 §6.7 [stmt.dcl]
- http://gameprogrammingpatterns.com/singleton.html
- http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
- http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/

Command

다음은 도형 편집기에 실행 취소(Undo)의 기능을 구현해 봅시다.
실행 취소의 명령은 가장 최근에 실행한 명령에 대해서 반대의 동작을 수행하는 형태로 구현할 수 있습니다. 결국 실행 취소의 기능은 명령을 스택에 저장해야 합니다. 하지만 어떠한 동작을 수행하는 각각의 명령은 스택에 저장하는 것이 불가능합니다. 명령을 스택에 저장하기 위해서는 객체로 캡슐화해야 합니다. 먼저 모든 명령의 인터페이스를 설계합니다.

```
struct ICommand {
  virtual void execute() = 0;
  virtual bool canUndo() { return false; }
  virtual void undo() {}
  virtual ~ICommand() {}
};
```

이제 기존의 도형 편집기의 모든 명령을 ICommand의 인터페이스를 구현하는 형태로 리팩토링 해줍니다.

```
class AddRectCommand : public ICommand {
  vector<Shape*>& v;
 public:
  AddRectCommand(vector<Shape*>& p) : v(p) {}

  virtual void execute() { v.push_back(new Rect); }
  virtual bool canUndo() { return true; }
  virtual void undo() {
    Shape* p = v.back();
    delete p;
    v.pop_back();
  }
};

class AddCircleCommand : public ICommand {
  vector<Shape*>& v;
 public:
  AddCircleCommand(vector<Shape*>& p) : v(p) {}

  virtual void execute() { v.push_back(new Circle); }
  virtual bool canUndo() { return true; }
  virtual void undo() {
    Shape* p = v.back();
    delete p;
    v.pop_back();
  }
};
```

모든 명령은 스택에 저장이 가능하기 때문에 실행 취소 기능은 어렵지 않게 구현할 수 있습니다.

```
int main() {
  vector<Shape*> v;
  stack<ICommand*> undo;
  while (1) {
   ...
   } else if (cmd == 0) {
      command = undo.top();
      undo.pop();

      command->undo();
      delete command;
      command = 0;
    }

    if (command != 0) {
      command->execute();

      if (command->canUndo())
        undo.push(command);
      else
        delete command;
    }
  }
}
```

이처럼 명령을 객체로 캡슐화해서 실행 취소의 기능을 제공하는 디자인 패턴을 커맨드 패턴이라고 합니다. GoF의 디자인 패턴에서 커맨드 패턴의 의도는 다음과 같습니다.
“요청을 객체의 형태로 캡슐화하여 서로 요청이 다른 사용자의 매개변수화, 요청 저장 또는 로깅, 그리고 연산의 취소를 지원하게 만드는 패턴 입니다.”

핵심

- 커맨드 패턴의 구현 원리: 명령의 캡슐화

참고 자료

- GoF 디자인 패턴: 커맨드 패턴

Factory Method

앞 장의 예제에서 AddRectCommand와 AddCircleCommand의 구현은 중복되는 코드가 많습니다.

```
class AddRectCommand : public ICommand {
  vector<Shape*>& v;

 public:
  AddRectCommand(vector<Shape*>& p) : v(p) {}

  virtual void execute() { v.push_back(new Rect); }
  virtual bool canUndo() { return true; }
  virtual void undo() {
    Shape* p = v.back();
    delete p;
    v.pop_back();
  }
};

class AddCircleCommand : public ICommand {
  vector<Shape*>& v;

 public:
  AddCircleCommand(vector<Shape*>& p) : v(p) {}

  virtual void execute() { v.push_back(new Circle); }
  virtual bool canUndo() { return true; }
  virtual void undo() {
    Shape* p = v.back();
    delete p;
    v.pop_back();
  }
};
```

중복 코드는 유지보수를 나쁘게 만들기 때문에, 공통의 특징을 묶은 부모 클래스를 설계하겠습니다.

```
class AddCommand : public ICommand {
  vector<Shape*>& v;
 public:
  AddCommand(vector<Shape*>& p) : v(p) {}
  virtual bool canUndo() { return true; }
  virtual void undo() {
    Shape* p = v.back();
    delete p;
    v.pop_back();
  }
  virtual void execute() { v.push_back(createShape()); }
  virtual Shape* createShape() = 0;
};
```

이제 도형을 추가하는 명령은 AddCommand를 상속 받고, 어떤 도형을 생성할 것인지에 대한 createShape 가상 함수만 재정의하면 됩니다.

```
class AddRectCommand : public AddCommand {
 public:
  AddRectCommand(vector<Shape*>& p) : AddCommand(p) {}
  virtual Shape* createShape() { return new Rect; }
};

class AddCircleCommand : public AddCommand {
 public:
  AddCircleCommand(vector<Shape*>& p) : AddCommand(p) {}
  virtual Shape* createShape() { return new Circle; }
};
```

위의 설계는 변하지 않는 부분은 부모 클래스에서 제공하고, 변하지 않는 부분은 가상함수로 자식이 재정의하는 형태의 템플릿 메소드 패턴과 유사하지만 위의 패턴은 GoF의 디자인 패턴에서 팩토리 메소드 패턴이라고 부릅니다. GoF의 팩토리 메소드 패턴의 의도는 다음과 같습니다.

"객체를 생성하는 인터페이스는 미리 정의하되, 인스턴스를 만들 클래스의 결정은 서브 클래스 쪽에서 내리는 패턴입니다. 팩토리 메소드 패턴에서는 클래스의 인스턴스를 만드는 시점을 서브 클래스로 미룹니다."

팩토리 메소드 패턴은 템플릿 메소드 패턴과 유사합니다. 하지만 디자인 패턴의 이름을 결정하는 것은 설계 의도에 의해서 결정됩니다. 변하는 것이 알고리즘이나 정책이라면 템플릿 메소드 패턴이고, 변하는 것이 객체 생성에 관련된 부분이라면 팩토리 메소드 패턴입니다.

핵심

- 템플릿 메소드 패턴
- 팩토리 메소드 패턴

참고 자료

- GoF 디자인 패턴: 팩토리 패턴

디자인 패턴 카탈로그

Abstract Factory

생성 패턴

의도

상세화된 서브 클래스를 정의 하지 않고도 서로 관련성이 있거나 독립적인 여러 객체의 군을 생성하기 위한 인터페이스를 제공한다.

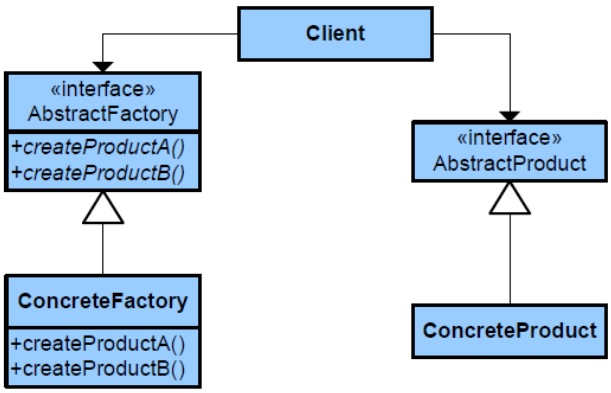
다른 이름

Kit

활용

- 생성되고 구성되고 표현되는 방식과 무관하게 시스템을 독립적으로 만들고자 할 때
- 하나 이상의 제품군들 중 하나를 선택해서 시스템을 설정해야 하고 한번 구성한 제품을 다른 것으로 대체 할 수 있을 때
- 관련된 객체군을 함께 사용해서 시스템을 설계하고, 이 제품이 갖는 제약 사항을 따라야 할 때
- 제품에 대한 클래스 라이브러리를 제공하고, 그들의 구현이 아닌 인터페이스를 표현하고 싶을 때

다이어그램



결과

- 구체적인 클래스를 분리 한다.
- 제품군 을 쉽게 대체할 수 있도록 한다
- 제품간의 일관성을 증진한다.
- 새로운 종류의 제품을 제공하기 어렵다.

Builder

생성 패턴

의도

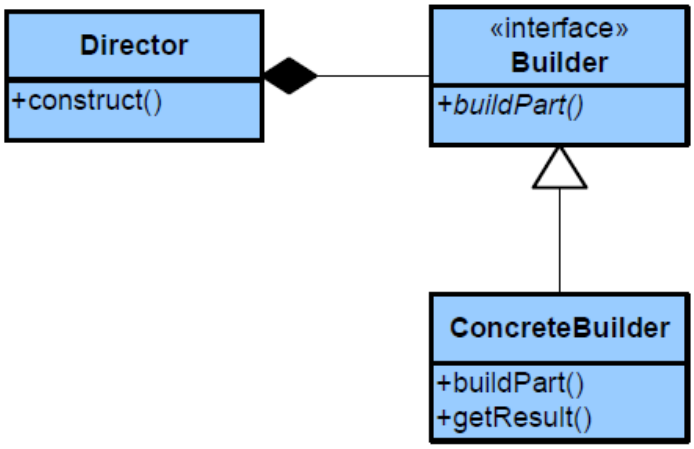
> 복잡한 객체를 생성하는 방법과 표현하는 방법을 정의하는 클래스를 별도로 분리하여 서로 다른 표현이라도 이를 생성할 수 있는 동일한 구축 공정을 제공할 수 있도록 한다.

다른 이름

활용

> - 복합 객체의 생성 알고리즘이 이를 합성하는 요소 객체들이 무엇인지 이들의 조립 방법에 독립적일 때
> - 합성할 객체들의 표현이 서로 다르더라도 구축 공정이 이를 지원해야 할 때

다이어그램



결과

> - 제품에 대한 내부 표현을 다양하게 변화할 수 있다.
> - 생성과 표현에 필요한 코드를 분리 한다.
> - 복합 객체를 생성하는 공정을 좀더 세밀하게 나눌 수 있다.

Factory Method

생성 패턴

의도

> 객체를 생성하기 위해 인터페이스를 정의 하지만, 어떤 클래스의 인스턴스를 생성할 지에 대한 결정은 서브 클래스가 한다. Factory Method 패턴에서는 클래스의 인스턴스를 만드는 시점을 서브 클래스로 미룬다.

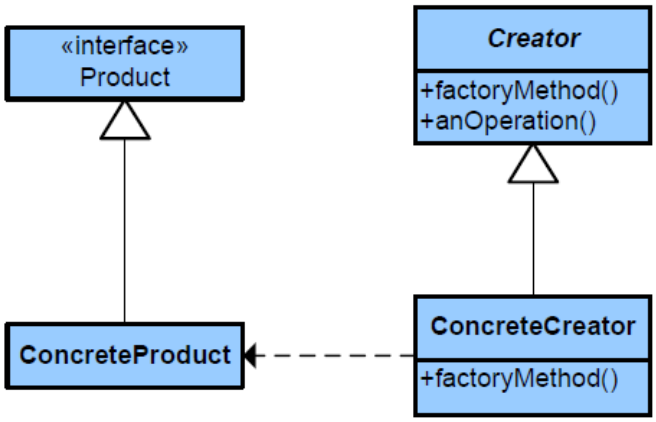
다른 이름

> 가상 생성자(Virtual Constructor)

활용

> - 생성할 객체 타입을 예측할 수 없을 때
> - 생성할 객체를 기술하는 책임을 서브클래스에 정의하고자 하는 경우
> - 객체 생성의 책임을 서브클래스에 위임 시키고 서브 클래스에 대한 정보를 은닉하고자 하는 경우

다이어그램

> 
>

결과

> - 서브클래스에 대한 훅(hook) 메소드를 제공한다.
> - 병렬적 클래스 계층도를 연결하는 역할을 담당한다.

Prototype

생성 패턴

의도

> 견본적(prototypical) 인스턴스를 사용하여 생성할 객체의 종류를 명시하고 이렇게 만들어진 견본을 복사하여 새로운 객체를 생성한다.

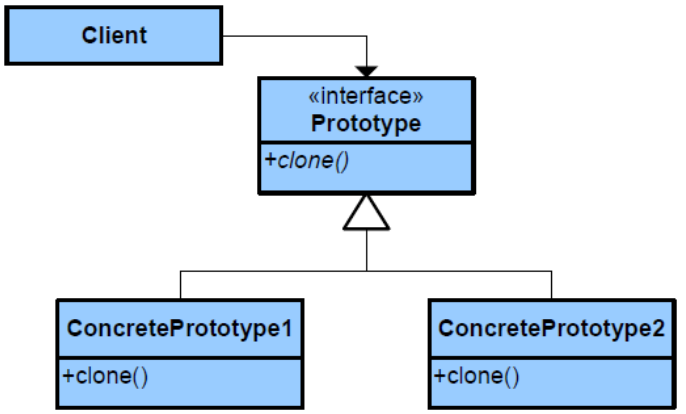
다른 이름

> 가상 복사 생성자(Virtual Copy Constructor)

활용

> • 런 타임시 만들 인스턴스의 클래스를 명세할 수 있을 때
> • 클래스 계층도와 병렬성을 갖는 팩토리 클래스의 계층을 피해야 할 경우
> • 클래스의 인스턴스들 이 서로 다른 상태 조합중에 어느 하나를 가질 때 사용한다.

다이어그램



결과

> • 실행시간에 새로운 제품을 삽입하고 삭제 할 수 있다.
> • 값들을 다양화함으로써 새로운 객체를 명세 한다.
> • 구조를 다양화함으로써 새로운 객체를 정의 할 수 있다.
> • 서브 클래스의 수를 줄인다.
> • 동적으로 생성된 클래스에 따라 애플리케이션을 형성할 수 있다.

Singleton

생성 패턴

의도

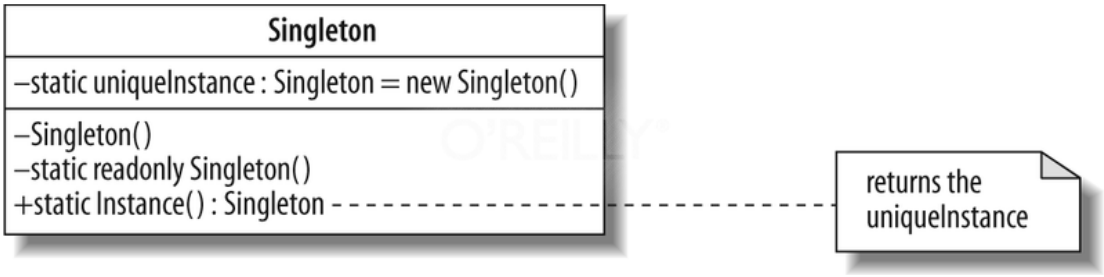
클래스의 인스턴스는 오직 하나임을 보장하며 이에 대한 접근은 어디에서든지 하나로만 통일하여 제공한다.

다른 이름

활용

- 클래스의 인스턴스가 오직 하나여야 함을 보장하고, 잘 정의된 접근 방식에 의해 모든 클라이언트가 접근할 수 있도록 해야 할 때.
- 유일하게 존재하는 인스턴스가 상속에 의해 확장 되어야 할 때, 클라이언트는 코드의 수정 없이 확장된 서브 클래스의 인스턴스를 사용 할 수 있어야 할 때

다이어그램



결과

- 유일하게 존재하는 인스턴스로의 접근을 통제할 수 있다.
- 변수 영역을 줄인다
- 오퍼레이션의 정제를 가능하게 한다.
- 인스턴스의 개수를 변경하기가 자유롭다
- 클래스 오퍼레이션을 사용하는 것보다 훨씬 유연한 방법이다.(Mono-State Pattern)
- C++의 정적 멤버 함수는 가상 함수 일 수 없다.

Adapter

구조 패턴

의도

> 클래스의 인터페이스를 클라이언트가 기대하는 형태의 인터페이스로 변환 한다. Adapter 패턴은 서로 일치 하지 않은 인터페이스를 갖는 클래스들을 함께 동작 시킨다.

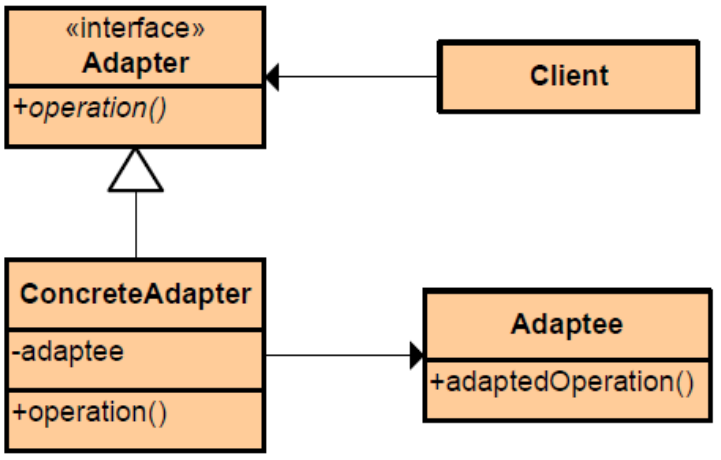
다른 이름

> Wrapper

활용

> - 기존의 클래스를 사용해야 하나 인터페이스가 수정되어야 하는 경우
> - 이미 만들어진 것을 재사용하고자 하나 이 재사용 가능한 라이브러리를 수정할 수 없는 경우.
> - 객체 생성의 책임을 서브클래스에 위임 시키고 서브 클래스에 대한 정보를 은닉하고자 하는 경우

다이어그램

> 
>

결과

> - Adapter 클래스가 Adaptee 클래스에 맞게 인터페이스를 Target 클래스로 변형한다. 또한 Adapter 클래스는 Adaptee 클래스의 행위를 재정의할 수 있다.

Bridge

구조 패턴

의도

> 구현과 추상화 개념을 분리하여 각각을 독립적으로 변형 할 수 있게 한다.

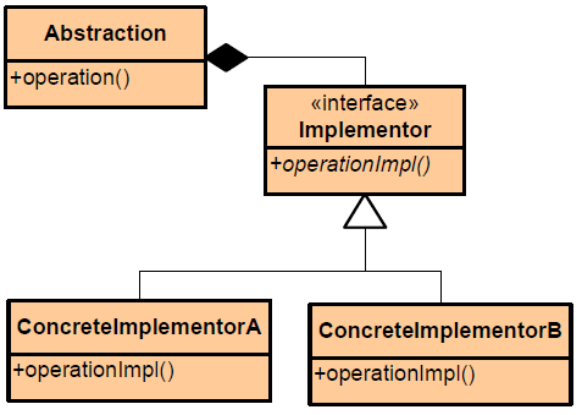
다른 이름

> Handle/Body

활용

> - 추상화 개념과 이에 대한 구현간의 종속성을 없애고 싶을 때
> - 추상화 개념과 구현 모두가 독립적으로 상속에 의해 확장 가능할 경우.
> - 추상화 개념에 대한 구현 내용을 변경하는 것은 다른 관련 프로그램에 아무런 영향을 주지 않는다.
> - C++ 사용자들은 클라이언트로부터 구현을 완벽하게 은닉하길 원한다.
> - 클래스 계층도 에서 클래스의 수가 급증하는 것을 방지 하고자 할 때
> - 여러 객체들에 걸쳐 구현을 공유하고자 할 때 또 이런 사실을 다른 코드에 공개 하고 싶지 않을 때

다이어그램

> 
>

결과

> - 인터페이스와 구현의 결합도 약화
> - 확장성 재고. Abstraction 과 Implementor를 독립적으로 확장할 수 있다.
> - 구현 사항을 은닉할 수 있다.

Composite

구조 패턴

의도

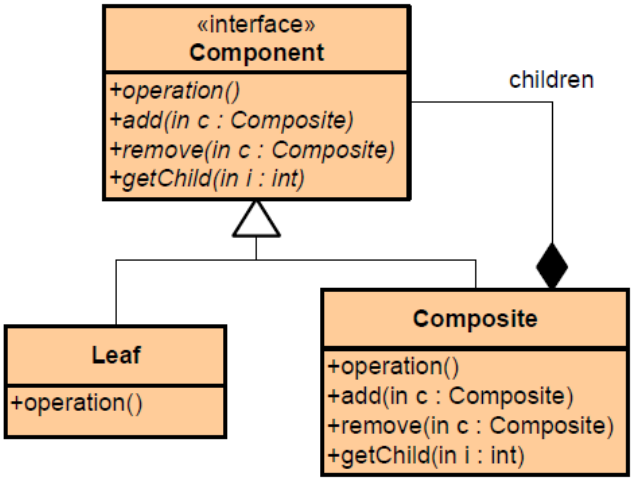
부분과 전체의 계층을 표현하기 위해 복합 객체를 트리 구조로 만든다.
Composite 패턴은 클라이언트로 하려면 개별 객체와 복합 객체를 모두 동일하게 다룰 수 있도록 한다.

다른 이름

활용

- 부분-집합 관계의 계층도를 표현해야 하고, 복합 관계와 단일 객체 간의 사용 방법에 차이를 두고 싶지 않을 때 사용한다.

다이어그램



결과

- 기본 객체와 복합 객체로 합성된 클래스로 하나의 일관된 계층도를 정의 한다.
- 프레임워크 사용자의 코드를 간단히 할 수 있다.
- 새로운 요소들을 쉽게 추가할 수 있다
- 범용성 있는 설계로 만들 수 있다.

Decorator

구조 패턴

의도

객체에 동적으로 새로운 서비스를 추가 할 있게 한다. Decorator 패턴은 기능의 추가를 위해서 서브 클래스를 생성하는 것보다 융통성 있는 방법을 제공한다.

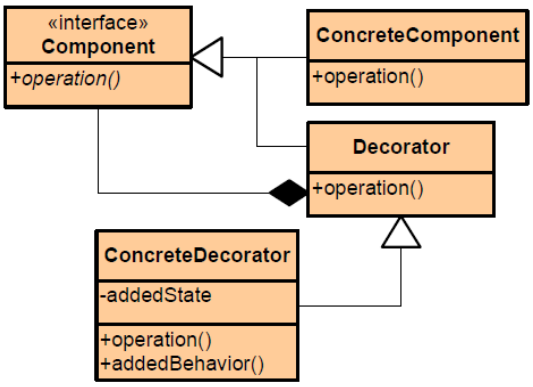
다른 이름

Wrapper

활용

- 동적으로 투명하게(transparent) 객체에 새로운 서비스를 추가하는 방법이며, 다른 객체에 어떠한 영향도 주지 않게 된다.
- 제거될 수 있는 서비스 일 때
- 실제 상속에 의해 서브클래스를 계속 만드는 방법이 실질적이지 못할 때 유용하다.

다이어그램



결과

- 단순한 상속보다 설계의 융통성을 증대 시킬 수 있다.
- 지금 예상하지 못한 특성들을 한꺼번에 다 개발하기 위해 고민하고 노력하기 보다는 지속적인 Decorator 객체를 통해서 발견하지 못하고 누락된 서비스들을 추가할 수 있다.
- Decorator와 Component가 동일한 것은 아니다.
- Decorator를 사용함으로써 작은 규모의 객체들을 많이 생성한다.

Facade

구조 패턴

의도

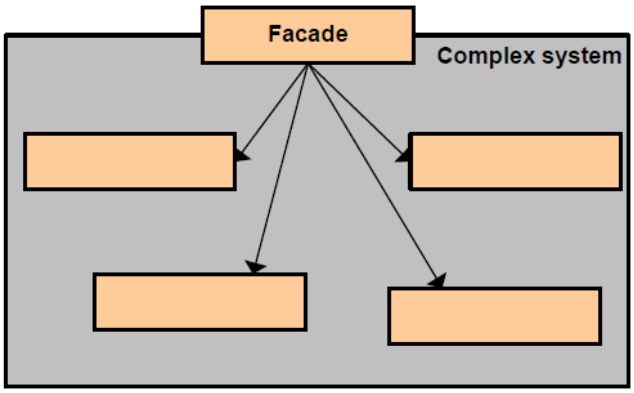
서브 시스템을 합성하는 다수의 객체들의 인터페이스 집합에 대해 일관된 하나의 인터페이스를 제공 할 수 있게 할 수 있게 한다. Facade 는 서브시스템을 사용하기 쉽게 하기 위한 포괄적 개념의 인터페이스를 정의 한다.

다른 이름

활용

- 복잡한 서브 시스템에 대한 단순한 인터페이스 제공이 필요할 때 유용하다.
- 클라이언트와 구현 클래스 간에 너무 많은 종속성이 존재 할 때 Facade의 사용으로 클라이언트와 다른 서브시스템간의 결합도를 줄일 수 있다.
- 서브 시스템들의 계층화를 이루고자 할 때, Façade 는 각 서브시스템의 계층별 접근점을 제공한다.

다이어그램



결과

- 서브 시스템의 구성 요소를 보호 할 수 있다. 이로써 클라이언트가 다루어야 할 객체의 수가 줄어든다.
- 서브 시스템과 클라이언트 코드 간의 결합도 를 줄일 수 있다.
- 서브 시스템 클래스를 사용하는 것을 완전히 막지는 않는다.

Flyweight

구조 패턴

의도

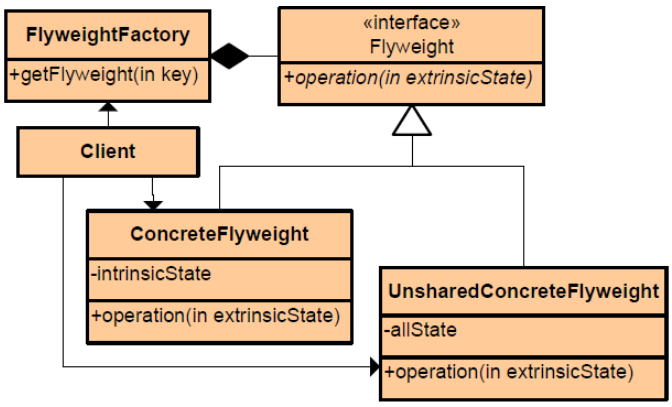
작은 크기의 객체들이 여러 개 있는 경우, 객체를 효과적으로 사용하는 방법으로 객체를 공유하게 한다.

다른 이름

활용

- 어플리케이션이 대량의 객체를 사용해야 할 때
- 객체의 수가 너무 많아져 저장 비용이 너무 높아 질 때
- 대부분의 객체 상태를 부가적인 것으로 만들 수 있을 때
- 부가적인 속성들을 제거한 후 객체들을 조사해 보니, 객체의 많은 묶음이 비교적 적은 수의 공유된 객체로 대체될 수 있을 때
- 어플리케이션이 객체 식별자에 비중을 두지 않은 경우

다이어그램



결과

- 공유해야 하는 인스턴스의 전체 수를 줄일 수 있다.
- 객체별 본질적 상태의 양을 줄일 수 있다.
- 부가적인 상태는 연산되거나 저장될 수 있다.

Proxy

구조 패턴

의도

> 다른 객체에 접근하기 위해 중간 대리 역할을 하는 객체를 둔다.

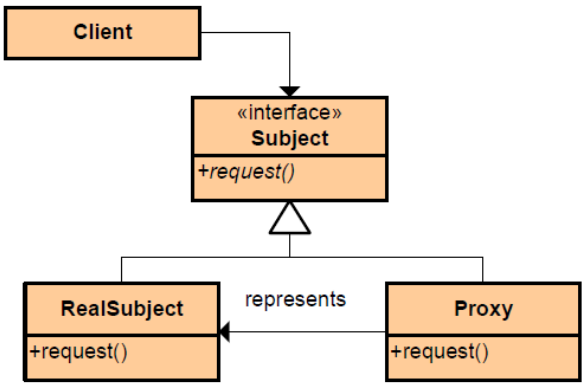
다른 이름

> Surrogate

활용

> - 원격지(Remote) Proxy 는 서로 다른 주소 공간에 존재하는 객체에 대한 지역적 표현으로 사용된다. (Proxy - Stub)
> - 가상(Virtual) Proxy는 요청이 있을 때만 필요한 복잡한 객체를 생성한다.
> - 보호용(Protect) Proxy는 원래 객체에 대한 실제 접근을 제어 한다.
> - 스마트 참조는(Smart reference) 객체로 접근이 일어날 때 추가적인 행동을 수행하는 노출된 포인터를 대신한다.

다이어그램

> 
>

결과

> - 원격지 프록시는 객체가 다른 주소에 존재한다는 사실을 숨길 수 있습니다.
> - 가상 프록시는 요구에 따라 객체를 생성하는 등 처리를 최적화할 수 있습니다.
> - 보호형 프록시 및 스마트 참조자는 객체가 접근할 때마다 추가 관리를 책임집니다. 객체를 생성할 것인지 삭제할 것인지를 관리합니다.

Chain Of Responsibility

행위 패턴

의도

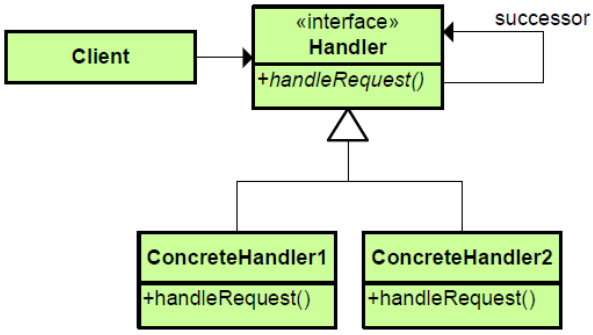
> 요청을 처리할 수 있는 기회를 하나 이상의 객체에 부여함으로써 요청하는 객체와 처리하는 객체 사이의 결합도를 없애려는 것이다. 요청을 해결할 객체를 만날 때 까지 고리를 따라서 요청을 전달한다.

다른 이름

활용

> - 하나 이상의 객체가 요청을 처리해야 하는 경우, 핸들러가 누가 선행자 인지를 모를 때, 다음 핸들러는 자동으로 결정된다.
> - 메시지를 받을 객체를 명시하지 않은 채 여러 객체 중 하나에게 처리를 요청하고 싶을 때
> - 요청을 처리할 객체 집합을 동적으로 정의하고자 할 때

다이어그램



결과

> - 객체들 간의 행위적 결합도가 적어 진다.
> - 객체에게 책임성을 할당하는데 있어 응용력을 높일 수 있다.
> - 메시지 수신을 보장할 수는 없다.

Command

행위 패턴

의도

> 요청을 객체로 캡슐화 함으로써 서로 다른 요청으로 클라이언트를 파라미터화 하고, 요청을 저장하거나 기록을 남겨서 오퍼레이션의 취소도 가능하게 한다.

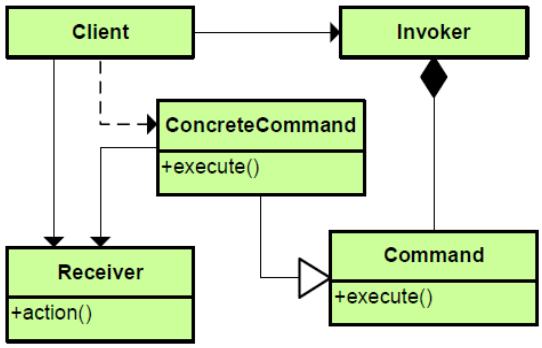
다른 이름

> Action, Transaction

활용

> - 수행할 행위 자체를 객체로 파라미터화 하고자 할 때. 절차 지향 프로그램의 Callback 함수 개념
> - 서로 다른 시간에 요청을 명시하고, 저장하고, 수행할 수 있다.
> - 명령어 객체 자체에 실행 취소에 필요한 상태를 저장할 수 있다.
> - 변경 과정에 대한 로그를 남겨 두면 시스템이 고장 났을 때 원상태로 복구가 가능하다.
> - 기본적인 오퍼레이션의 조합으로 좀더 고난위도의 오퍼레이션을 구성할 수 있다

다이어그램

> 
>

결과

> - Command는 오퍼레이션을 호출하는 객체와 오퍼레이션을 수행방법을 구현하는 객체를 분리한다.
> - Command 자체도 클래스로서 다른 객체와 같은 방식으로 조작되고 확장할 수 있다.
> - 명령어를 조합해서 다른 명령어를 만들 수 있다.
> - 새로운 명령을 추가하기 쉽다.

Iterator

행위 패턴

의도

> 복합 객체 요소들의 내부 표현 방식을 공개하지 않고도 순차적으로 접근할 수 있는 방법을 제공한다.

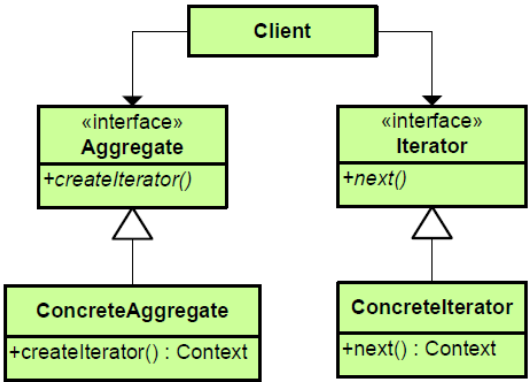
다른 이름

> Cursor

활용

> • 객체 내부의 표현 방식을 모르고도 집합 객체의 각 요소들을 순회 할 수 있다.
> • 집합 객체를 순회 하는 다양한 방법을 제시 할 수 있다.
> • 서로 다른 집합 객체 구조에 대해서도 동일한 방법으로 순회 할 수 있다.

다이어그램



결과

> • 집합 객체에 대한 다양한 순회 방법을 제공한다.
> • Iterator는 Aggregate 클래스의 인터페이스를 단순화 할 수 있다.
> • 집합 객체에 대해 하나 이상의 순회 방법을 정의 할 수 있다.

Mediator

행위 패턴

의도

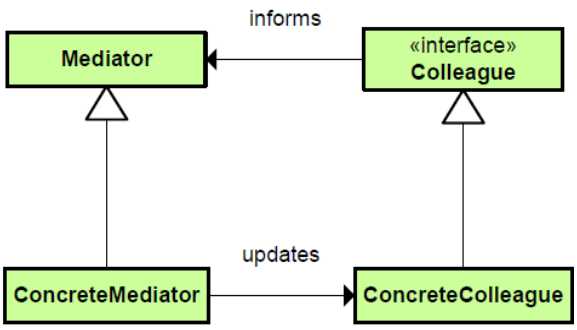
객체들 간의 상호 작용을 객체로 캡슐화 한다. 객체들간의 참조 관계를 객체에서 분리 함으로써 상호작용만을 독립적으로 다양하게 확대 해석할 수 있다.

다른 이름

활용

- 여러 객체가 잘 정의된 형태이기는 하지만 복잡한 상호 관계를 가지는 경우, 객체들 간의 의존성을 잘 이해하기 어려울 때
- 객체의 재사용이 다른 객체와의 연결 관계의 복잡함으로 인해 방해를 받을 때
- 여러 클래스에 분산된 행위들이 상속 없이 수정되어야 할 때

다이어그램



결과

- 서브 클래스의 수를 제한 한다.
- Colleague들 사이의 종속성을 줄인다.
- 객체의 프로토콜을 단순화 하는 장점이 있다.
- 객체들 간의 협력 방법을 하나의 클래스로 추상화 한다.
- 통제의 집중화가 이루어 진다.

Memento

행위 패턴

의도

> 캡슐화를 위해 하지 않고 객체 내부 상태를 캡슐화 하여, 나중에 객체가 이 상태로 복구 가능하게 한다.

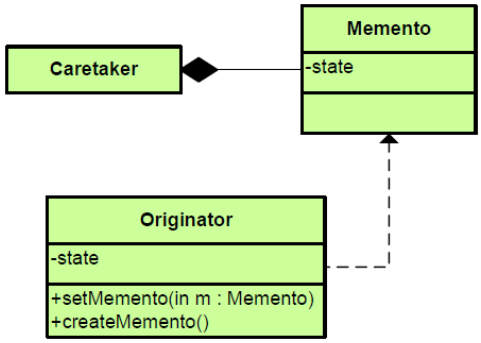
다른 이름

> Token

활용

> - 각 시점에서의 객체 상태를 저장한 후 나중에 이 상태로 복구해야 할 때
> - 상태를 얻는데 필요한 직접 인터페이스는 객체의 자세한 구현 내용을 드러나게 하고 객체의 캡슐화를 위배하는 것이므로, 이를 해결할 때.

다이어그램

> 
>

결과

> - 캡슐화된 경계를 유지 할 수 있다
> - Originator 클래스를 단순화 할 수 있다.
> - 메멘토의 사용으로 더 많은 비용을 들여야 할 수도 있다.
> - 좁은 범위의 인터페이스와 넓은 범위의 인터페이스를 정의 해야 한다.
> - 메멘토를 다루기 위해 감추어진 비용이 존재 한다.

Observer

행위 패턴

의도

> 객체 사이의 1:N의 종속성을 정의 하고 한 객체의 상태가 변하면 종속된 다른 객체에 통보가 가고 자동으로 수정이 일어 나게 한다.

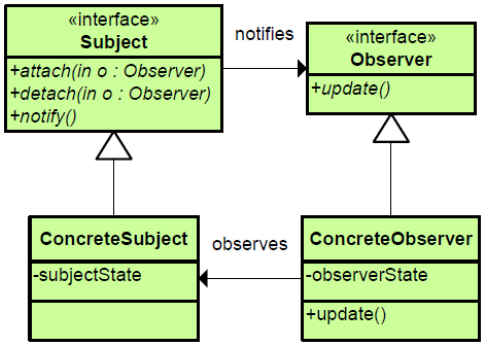
다른 이름

> Dependent, Publish-Subscribe

활용

> - 추상화 개념이 두 가지의 측면을 갖고 하나가 다른 하나에 종속적일 때, 이런 종속 관계를 하나의 객체로 분리시켜 이들 각각을 재 사용할 수 있다.
> - 한 객체에 가해진 변경으로 다른 객체를 변경해야 할 때 프로그래머들은 얼마나 많은 객체들이 변경되어야 하는지를 몰라도 된다.
> - 객체는 다른 객체에 변화를 통보 할 수 있는데, 변화에 관심 있어 하는 객체들이 누구 인지에 대한 가정 없이도 이루어져야 할 때

다이어그램

> 
>

결과

> - Subject와 Observer 클래스 간에는 추상화된 결합도 만이 존재 한다.
> - Broadcast 방식의 교류를 가능하게 한다.
> - 예측하지 못한 수정

State

행위 패턴

의도

> 객체 자신의 내부 상태에 따라 행위를 변경하도록 한다. 객체는 마치 클래스를 바꾸는 것처럼 보인다.

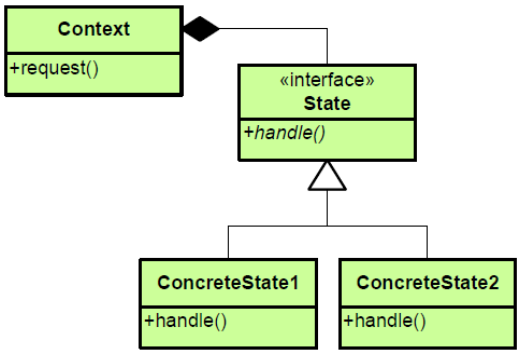
다른 이름

> Object for State

활용

> - 객체의 행위는 상태에 따라 달라 질 수 있기 때문에, 객체의 상태에 따라서 런타임시 행위가 바뀌어야 한다.
> - 객체에 상태에 따라 수많은 조건 문장을 갖도록 오퍼레이션을 구현 할 수 있다. 이때 객체의 상태를 별도의 객체로 정의 함으로써 행위는 다른 객체와 상관없이 다양화 될 수 있다.

다이어그램

> 
>

결과

> - 상태에 따른 행위를 국지화 하여 서로 다른 상태에 대한 행위를 별도의 객체로 관리 한다.
> - 상태 전이 규칙을 명확하게 만든다.
> - 상태 객체는 공유 될 수 있다.

Strategy

행위 패턴

의도

> 다양한 알고리즘이 존재 하면 이들 각각을 하나의 클래스로 캡슐화하여 알고리즘의 대체가 가능하도록 한다. Strategy 패턴을 이용하면 클라이언트와 독립적인 다양한 알고리즘으로 변형할 수 있다. 알고리즘을 바꾸더라도 클라이언트는 아무런 변경을 할 필요가 없다.

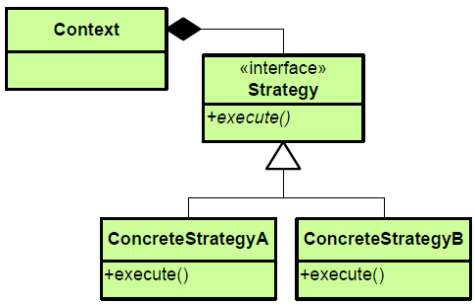
다른 이름

> Policy

활용

> - 행위들이 조금씩 다를 뿐 개념적으로 관련된 많은 클래스들이 존재 하는 경우, 각각의 서로 다른 행위 별로 클래스를 작성한다.
> - 알고리즘의 변형이 필요한 경우 사용할 수 있다.
> - 사용자가 모르고 있는 데이터를 사용해야 하는 알고리즘이 있을 때도 필요 하다.
> - 많은 행위를 정의하기 위해 클래스 안에 복잡한 다중 조건문을 사용해야 하는 경우 이런 선택문보다는 Strategy 클래스로 만드는 것이 바람직하다.

다이어그램

> 
>

결과

> - 관련 알고리즘 군을 형성한다.
> - 서브 클래싱을 사용하지 않는 다른 방법이다.
> - 조건문을 없앨 수 있다.
> - 구현의 선택이 가능하다.
> - Strategy 와 Context 클래스 사이에 과다한 메시지가 전송된다.
> - 객체 수가 증가한다.

Template Method

행위 패턴

의도

> 오퍼레이션에는 알고리즘의 처리 과정만을 정의 하고 각 단계에서 수행할 구체적인 처리는 서브 클래스 에서 정의 한다. Template Method 패턴은 알고리즘의 처리과정은 변경하지 않고 알고리즘 각 단계의 처리를 서브클래스에서 재정의 할 수 있게 한다.

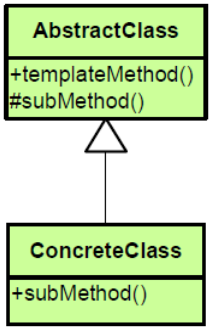
다른 이름

> NVI(Non Virtual Interface)

활용

> - 알고리즘의 변하지 않은 부분을 한 번 정의하고 다양해 질 수 있는 부분을 서브 클래스로 정의할 수 있도록 구현하고자 할 때
> - 서브 클래스 사이의 공통적인 행위를 추출하여 하나의 공통 클래스로 정의할 때
> - 서브 클래스의 확장을 제어할 수 있다.

다이어그램

> 
>

결과

> - 코드 재사용을 위한 기술
> - 라이브러리를 설계할 때 공통 부분을 분리하는 중요한 디자인 기술.

Visitor

행위 패턴

의도

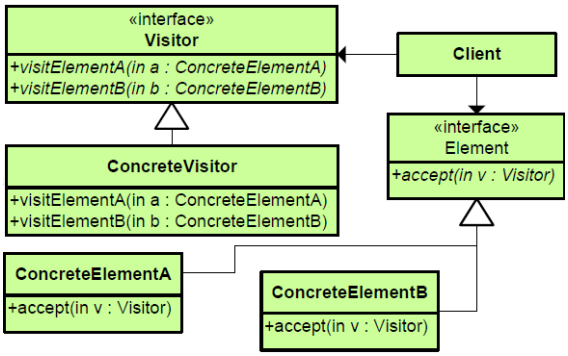
객체 구조에 속한 요소에 수행될 오퍼레이션을 정의 하는 객체. Visitor 패턴은 처리되어야 하는 요소에 대한 클래스를 변경하지 않고 새로운 오퍼레이션을 정의할 수 있게 한다.

다른 이름

활용

- 객체 구조가 다른 인터페이스를 가진 클래스들을 포함하고 있어서 구체적 클래스에 따라 이들 오퍼레이션을 수행하고자 할 때
- 구별되고 관리되지 않은 많은 오퍼레이션이 객체에 수행될 필요가 있지만, 오퍼레이션으로 인해 클래스들을 복잡하게 하고 싶지 않을 때.
- 객체 구조를 정의한 클래스는 거의 변하지 않지만 전체 구조에 걸쳐 새로운 오퍼레이션을 추가하고 싶을 때.

다이어그램



결과

- 새로운 오퍼레이션을 쉽게 추가 한다.
- 방문자를 통해 관련된 오퍼레이션을 하나로 모아 관련되지 않은 오퍼레이션과 분리 할 수 있다
- 새로운 ConcreteElement를 추가하기 어렵다.
- 상태를 누적할 수 있다.
- 캡슐화 전략을 위배할 수 있다.

Interpreter

행위 패턴

의도

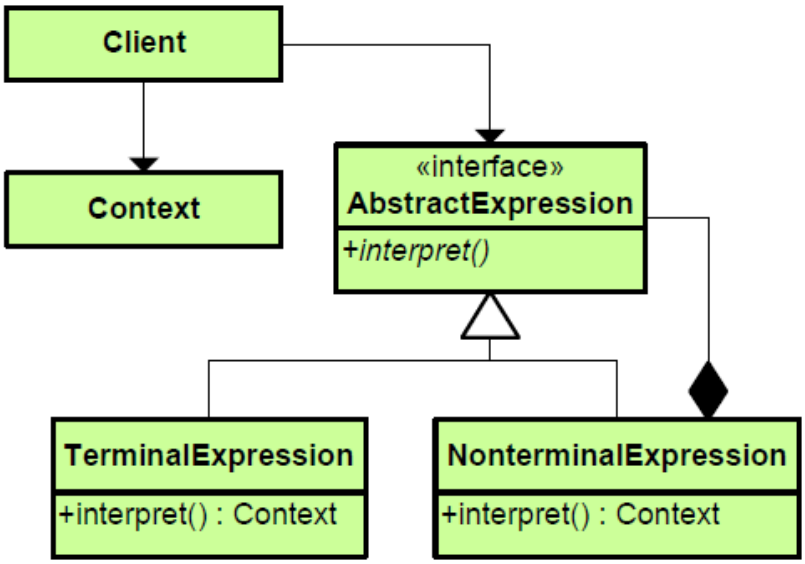
> 언어에 따라서 문법에 대한 표현을 정의 한다. 또 언어의 문장을 해석하기 위해 정의에 표현에 기반하여 분석기를 정의 한다.

다른 이름

활용

> • 정의할 언어의 문법이 간단한 경우
> • 효율성은 별로 고려 사항이 되지 않는다.

다이어그램



결과

> • 문법의 변경과 확장이 쉽다
> • 문법의 구현이 용이하다.
> • 복잡한 문법은 관리하기 어렵다.
> • 표현식을 해석하는 새로운 방법을 추가할 수 있다.