주요 디자인 패턴

객체 지향 설계

* 소프트웨어로 해결하고자 하는 문제를 다룸.
* 동시에 재설계 없이, 또는 재설계를 최소화하면서 요구 사항의 변화를 수용.

디자인 패턴

* 객체 지향 설계를 하다보면 이전과 비슷한 상황에서 사용했던 설계를 재사용.
* 이런 설계는 특정 상황에 맞는 해결책을 빠르게 찾을 수 있도록 도와줌.
* 반복적으로 사용되는 설계는 클래스, 객체의 구성, 객체 간 메시지 흐름에서 일정 패턴을 갖는다.

디자인 패턴으로 얻는 이득

* 상황에 맞는 올바른 설계를 더 빠르게 적용
* 각 패턴의 장단점을 통해서 설계를 선택하는데 도움.
* 설계 패턴에 이름을 붙임으로써 시스템의 문서화, 이해, 유지 보수에 도움

디자인 패턴 책 추천 : GoF의 디자인 패턴

자주 사용되는 패턴

* 전략 패턴 / 템플릿 메서드 패턴 / 상태 패턴
* 데코레이터 패턴 / 프록시 패턴 / 어댑터 패턴
* 옵저버 패턴 / 미디에이터 패턴 / 파사드 패턴
* 추상 팩토리 패턴 / 컴포지트 패턴
* 널(Null) 객체 패턴

1. **전략(Strategy) 패턴**

|  |
| --- |
| **리스트 7.1 가격 계산 모듈에 할인 정책 구현이 포함된 코드**  public class Calculator {  public int calculate(boolean firstGuest, List<Item> items) {  int sum = 0;  for (Item item : items) {  if (firstGuest)  sum += (int) (item.getPrice() \* 0.9); // 첫 손님 10% 할인  else if (! item.isFresh())  sum += (int) (item.getPrice() \* 0.8); // 덜 신선한 것 20% 할인  else  sum += item.getPrice();  }  return sum;  }  } |

위 코드의 문제점

* 서로 다른 계산 정책들이 한 코드에 섞여 있어, 정책이 추가될수록 코드 분석을 어렵게 만든다.
* 가격 정책이 추가될 때마다 calculate 메서드를 수정하는 것이 점점 어려워진다.

해결 방법 1: 가격 할인 정책을 별도 객체로 분리

* DiscountStategy 인터페이스: 상품의 할인 금액 계산 추상화
* 각 콘크리트 클래스: 상황에 맞는 할인 계산 알고리즘 제공
* Calculator 클래스: 가격 합산 계산의 책임

DiscountStategy를 전략(Strategy), Calculator를 콘텍스트(Context)라고 부른다.

특정 콘텍스트에서 알고리즘(전략)을 별도로 분리하는 설계 방법이 전략 패턴.

콘텍스트는 사용할 전략을 직접 선택하지 않는다.

콘텍스트의 클라이언트가 콘텍스트에 사용할 전략을 전달. (DI 방식)

어떤 메서드를 제공할 지는 콘텍스트가 전략을 어떻게 사용하느냐에 따라 달라진다.

|  |
| --- |
| **리스트 7.2 전략 패턴을 적용한 Calculator의 구현**  public class Calculator {  private DiscountStrategy discountStrategy;  public Calculator(DiscountStrategy discountStrategy) {  this.discountStrategy = discountStrategy;  }  public int calculate(List<Item> items) {  int sum = 0;  for (Item item : items) {  sum += discountStrategy.getDiscountPrice(item);  }  return sum;  }  } |

|  |
| --- |
| **리스트 7.3 Calculator에서 사용하는 전략 인터페이스**  interface DiscountStrategy {  int getDiscountPrice(Item item);  } |

만약 새로운 정책이 추가된다면 DiscountStrategy에 새로운 메서드가 추가되거나 새로운 정책을 위한 별도 인터페이스로 분리할 수 있다.

전략 객체는 콘텍스트를 사용하는 클라이언트에서 직접 생성.

|  |
| --- |
| **리스트 7.4 DiscountStrategy 인터페이스를 구현한 콘크리트 클래스**  class FirstGuestDiscountStrategy implements DiscountStrategy {  public int getDiscountPrice(Item item) {  return (int) (item.getPrice() \* 0.9);  }  } |

|  |
| --- |
| **처리하는 코드 예**  private DiscountStrategy strategy;  public void onFirstGuestButtonClick() {  // 첫 손님 할인 버튼 누를 때 생성 됨  strategy = new FirstGuestDiscountStrategy();  }  public void onCalculationButtonClick() {  // 계산 버튼 누를 때 실행 됨  Calculator calculator = new Calculator(strategy);  int price = calculator.calculate(items);  } |

전략의 콘크리트 클래스와 클라이언트의 코드가 쌍을 이루기 때문에 유지보수 문제가 줄어든다.

클라이언트의 버튼 처리 코드에서 전략 객체를 직접 생성하는 것은 코드 이해를 높이고 코드 응집을 높여주는 효과를 갖는다.

콘텍스트 코드의 변경 없이 새로운 전략을 추가할 수 있다.

Calculator 클래스는 할인 정책 확장에는 열려 있고 변경에는 닫혀 있게 된다. (개방 폐쇄 원칙)

일반적으로 if-else로 구성된 코드 블록이 비슷한 기능을 수행하는 경우 전략 패턴을 적용함으로써 코드를 확장 가능하도록 변경할 수 있다.

전략 패턴은 동일한 기능의 알고리즘 변경이 필요할 때 사용된다.

1. **템플릿 메서드(Template Method) 패턴**

|  |  |
| --- | --- |
| public class DbAuthenticator {  private UserDao userDao = new UserDao();  public Auth authenticate(String id, String pw) {  // 사용자 정보로 인증 확인  User user = userDao.selectById(id);  boolean auth = user.equalPassword(pw);  // 인증 실패시 익셉션 발생  if (!auth)  throw createException();  // 인증 성공시, 인증 정보 제공  return new Auth(id, user.getName());  }  private AuthException createException() {  return new AuthException();  }  } | public class LdapAuthenticator {  LdapClient ldapClient = new LdapClient();  public Auth authenticate(String id, String pw) {  // 사용자 정보로 인증 확인  boolean lauth = ldapClient.authenticate(id, pw);  // 인증 실패시 익셉션 발생  if(!lauth)  throw createException();  // 인증 성공시, 인증 정보 제공  LdapContext ctx = ldapClient.find(id);  return new Auth(id, ctx.getAttribute("name"));  }  private AuthException createException() {  return new AuthException();  }  } |

문제 상황

* 완전히 동일한 절차를 가진 코드를 작성하게 될 때
* 절차 중 일부 과정의 구현만 다를 뿐 나머지 구현은 똑같을 때

실행 과정/단계는 동일한데 각 단계 중 일부의 구현이 다른 경우에 사용할 수 있는 패턴이 템플릿 메서드 패턴이다.

템플릿 메서드 패턴의 구성

* 실행 과정을 구현한 상위 클래스
  + 상위 클래스는 실행 과정을 구현한 메서드 제공
  + 메서드는 기능을 구현하는 데 필요한 각 단계 정의, 일부 단계는 추상 메서드를 호출하는 방식으로 구현 (추상 메서드는 구현이 다른 단계에 해당)
* 실행 과정의 일부 단계를 구현한 하위 클래스

위 문제에서 템플릿 메서드 패턴을 적용하여 상위 클래스를 작성

|  |
| --- |
| 리스트 7.5 템플릿 메서드를 제공하는 Authenticator  public abstract class Authenticator {  // 템플릿 메서드  public Auth authenticate(String id, String pw) {  if (!doAuthenticate(id, pw))  throw createException();  return createAuth(id);  }  protected abstract boolean doAuthenticate(String id, String pw);  private RuntimeException createException() {  throw new AuthException();  }  protected abstract Auth createAuth(String id);  } |

* authenticate() 메서드는 모든 하위 타입에서 동일하게 적용되는 실행 과정을 제공.
* 이 메서드를 템플릿 메서드라고 부른다.
* 상속받은 하위 클래스는 authenticate() 메서드에서 호출하는 다른 메서드만 알맞게 재정의해 주면 된다.

|  |
| --- |
| 리스트 7.6 템플릿 메서드 패턴을 적용한 LdapAuthenticator의 구현  public class LdapAuthenticator extends Authenticator {  LdapClient ldapClient = new LdapClient();  @Override  protected boolean doAuthenticate(String id, String pw) {  return ldapClient.authenticate(id, pw);  }  @Override  protected Auth createAuth(String id) {  LdapContext ctx = ldapClient.find(id);  return new Auth(id, ctx.getAttribute("name"));  }  } |

* 전체 실행 과정 구현을 제공하지 않고 일부 과정의 구현만 제공
* 전체 실행 과정은 상위 타입 Authenticator의 authenticate() 메서드에서 제공
* 동일한 실행 과정의 구현을 제공, 동시에 하위 타입에서 일부 단계를 구현
  + 각 타입에서 코드가 중복되는 것 방지

템플릿 메서드 패턴 특징

* 하위 클래스가 아닌 상위 클래스에서 실행 흐름 제어
* 템플릿 메서드의 경우 외부에 제공하는 기능에 해당되기 때문에 public 범위
* 내부 메서드는 템플릿 메서드에서만 호출되는 메서드에 하위 타입에서 재정의 할 수 있어야 하기 때문에 protected의 범위
* 추상 메서드의 경우 하위 클래스에서 반드시 구현해 주어야 하면 사용하고, 그렇지 않다면 확장 가능하게만 해놓는다.

훅(hook) 메서드 : 상위 클래스에서 실행 지점이 제어되고, 기본 구현 제공, 하위 클래스에서 알맞게 확장 할 수 있는 메서드

* 템플릿 메서드와 전략 패턴을 함께 사용하면 상속이 아닌 조립의 방식으로 템플릿 메서드 패턴 활용 가능. (예 : 스프링 프레임워크의 Template 클래스들)
* 템플릿 메서드 패턴과 전략 패턴을 조합하면 상속에 기반을 둔 템플릿 메서드 구현과 비교해서 유연하다.
* 상속을 통한 재사용은 클래스가 불필요하게 증가할 수 있고, 런타임에 교체할 수 없는 단점이 있다.
* 상속 방식의 경우 훅 메서드를 재정의하는 방법으로 하위 클래스에서 쉽게 확장 기능 제공하는 장점. 조립/위임 방식에서는 확장 기능을 제공하려면 구현이 다소 복잡해지는 단점.

1. **상태(State) 패턴**

조건에 따라 다른 코드를 실행해야 하는 코드

|  |
| --- |
| 리스트 7.7 조건문을 이용한 자판기 프로그램 구현  class VendingMachine {  public static enum State { NOCOIN, SELECTABLE }  private State state = State.NOCOIN;  public void insertCoin(int coin) {  switch (state) {  case NOCOIN:  increaseCoin(coin);  state = State.SELECTABLE;  break;  case SELECTABLE:  increaseCoin(coin);  }  }  public void select(int productId) {  switch (state) {  case NOCOIN:  // 아무 것도 하지 않음  break;  case SELECTABLE:  provideProduct(productId);  decreaseCoin();  if (hasNoCoin())  state = State.NOCOIN;  }  }  private void increaseCoin(int coin) {  }  private boolean hasNoCoin() {  return false;  }  private void decreaseCoin() {    }  private void provideProduct(int productId) {    }  } |

새로운 요구사항 추가 : 자판기에 제품이 없는 경우에는 동전을 넣으면 바로 동전을 되돌려 준다.

|  |
| --- |
| 리스트 7.8 조건문을 이용한 자판기 프로그램 구현에 기능 추가하기  public void insertCoin(int coin) {  switch (state) {  case NOCOIN:  increaseCoin(coin);  state = State.SELECTABLE;  break;  case SELECTABLE:  increaseCoin(coin);  break;  **case SOLDOUT:**  **returnCoin();**  }  }  public void select(int productId) {  switch (state) {  case NOCOIN:  // 아무 것도 하지 않음  break;  case SELECTABLE:  provideProduct(productId);  decreaseCoin();  if (hasNoCoin())  state = State.NOCOIN;  **case SOLDOUT:**  **// 아무 것도 하지 않음**  }  } |

* insertCoin(), select() 메서드는 동일한 구조의 조건문을 갖고 있다.
* 상태가 많아질수록 복잡해지는 조건문이 여러 코드에서 중복해서 출현, 코드 변경을 어렵게 만든다.

위 클래스의 조건문은 다음과 같은 의미를 갖고 있음 : 상태에 따라 동일한 기능 요청의 처리를 다르게 함.

기능이 상태에 따라 다르게 동작해야 할 때 사용할 수 있는 패턴이 상태 패턴이다.

상태를 별도 타입으로 분리하고, 각 상태 별로 알맞은 하위 타입을 구현한다.

상태 패턴에서 중요한 점은 상태 객체가 기능을 제공.

|  |
| --- |
| interface State {  void increaseCoin(int coin, VendingMachine vm);  void select(int productId, VendingMachine vm);  } |

* State 인터페이스는 동전 증가 처리, 제품 선택 처리를 할 수 있는 두 개의 메서드 정의
* 모든 상태에 동일하게 적용되는 기능

|  |
| --- |
| 리스트 7.9 상태 패턴을 적용한 VendingMachine 구현  class VendingMachine {  private State state;  public VendingMachine() {  state = new NoCoinState();  }  public void insertCoin(int coin) {  state.increaseCoin(coin, this);  }  public void select(int productId) {  state.select(productId, this);  }  public void changeState(State newState) {  this.state = newState;  }  } |

* 콘텍스트는 필드로 상태 객체를 갖고 있다.
* 클라이언트로부터 기능 실행 요청을 받으면, 상태 객체에 처리를 위임하는 방식
* 리스트 7.9는 State 객체에 처리를 위임하는 방식으로 동작

|  |
| --- |
| 리스트 7.10 동전 없음 상태일 때의 동작을 구현한 NoCoinState 클래스  class NoCoinState implements State {  public void increaseCoin(int coin, VendingMachine vm) {  vm.insertCoin(coin);  vm.changeState(new SelectableState());  }  public void select(int productId, VendingMachine vm) {  SoundUtil.beep();  }  } |

|  |
| --- |
| 리스트 7.11 음료 가능 상태의 동작 방식을 구현한 SelectableState 클래스  class SelectableState implements State {  public void increaseCoin(int coin, VendingMachine vm) {  vm.insertCoin(coin);  }  public void select(int productId, VendingMachine vm) {  vm.provideProduct(productId);  vm.decreaseCoin();  if (vm.hasNoCoin()) {  vm.changeState(new NoCoinState());  }  }  } |

* VendingMachine 클래스에 구현되어 있는 상태 별 동작 구현 코드가 각 상태의 구현 클래스로 이동
* VendingMachine 클래스의 코드 구현은 상태 객체에 위임하는 방식

상태 패턴의 장점

* 새로운 상태가 추가되어도 콘텍스트 코드가 받는 영향은 최소화
* 상태에 따른 동작을 구현한 코드가 각 상태 별로 구분, 상태 별 동작을 수정하기 쉽다.

상태 변경

* 상태 패턴을 적용할 때 콘텍스트의 상태 변경을 누가 하느냐 고려해야한다.
* 상태 변경의 주체 : 콘텍스트, 상태 객체 (앞의 예는 상태 객체를 통해 변경)

|  |
| --- |
| public void increaseCoin(int coin, VendingMachine vm) {  vm.insertCoin(coin);  **// 상태 객체에서 콘텍스트의 상태 변경**  **vm.changeState(new SelectableState());**  }` |
| public void select(int productId, VendingMachine vm) {  vm.provideProduct(productId);  vm.decreaseCoin();  **if (vm.hasNoCoin()) { // 상태 변경을 위해, vm 객체가 동전이 없는지 확인**  vm.changeState(new NoCoinState());  }  } |

콘텍스트가 직접 상태를 변경하도록 VendingMachine 클래스를 수정하면 다소 복잡해질 수 있다.

|  |
| --- |
| 리스트 7.12 VendingMachine에서 상태를 변경하도록 구현한 예  class VendingMachine {  private State state;  public VendingMachine() {  state = new NoCoinState();  }  public void insertCoin(int coin) {  state.increaseCoin(coin, this);  if (hasCoin())  **changeState(new SelectableState()); // 콘텍스트에서 상태 변경**  }  public void select(int productId) {  state.select(productId, this);  if (state.isSelectable() && hasNoCoin())  **changeState(new NoCoinState()); // 콘텍스트에서 상태 변경**  }    **private** void changeState(State newState) {  this.state = newState;  }  **private** boolean hasCoin() {  return false;  }  **private** boolean hasNoCoin() {  return !hasCoin();  }  } |
| // 콘텍스트가 상태를 변경하므로, 상태 객체는 자신이 할 작업만 처리한다.  public class SelectableState implements State {  public void select(int productId, VendingMachine vm) {  vm.provideProduct(productId);  vm.decreaseCoin();  }  } |

콘텍스트의 상태 변경을 누가 할지는 주어진 상황에 알맞게 정해야 함.

콘텍스트에서의 상태 변경 : 상태 개수가 적고, 상태 변경 규칙이 거의 바뀌지 않는 경우

상태 객체에서 상태 변경 : 콘텍스트에 영향을 주지 않고, 상태를 추가하거나 상태 변경 규칙을 바꿀수 있다. 여러 클래스에 상태 변경 규칙이 분산되어 있어서 상태 구현 클래스가 많아지면 규칙을 파악하기 어렵다. 상태 클래스에서 다른 상태 클래스에 대한 의존도 발생한다.

주어진 상황에 알맞은 방식을 선택해야 한다.

1. **데코레이터{Decorator} 패턴**

* 상속을 이용한 기능 확장 방법이 쉽긴 하지만, 다양한 조합의 기능 확장이 요구될 때 클래스가 불필요하게 증가한다. 또한 클래스가 증가하면서 계층 구조가 복잡해진다.
* 데코레이터 패턴은 상속이 아닌 위임을 하는 방식으로 기능을 확장해 나간다.

|  |
| --- |
| 리스트 7.13 데코레이터를 위한 기반 클래스인 Decorator의 구현  public abstract class Decorator implements FileOut {  private FileOut delegate; // 위임 대상  public Decorator(FileOut delegate) {  this.delegate = delegate;  }  protected void doDelegate(byte[] data) {  delegate.write(data); // delegate에 쓰기 위임  }  } |
| FileOut 인터페이스는 파일 출력 기능 정의, FileOutImpl은 실제 파일 출력 기능 구현  Decorator 클래스는 모든 데코레이터를 위한 기반 기능을 제공하는 추상 클래스.  doDelegate() 메서드는 생성자를 통해서 전달받은 FileOut 객체에 쓰기 기능을 위임 |

|  |
| --- |
| 리스트 7.14 EncryptionOut 클래스 구현  class EncryptionOut extends Decorator {  public EncryptionOut(FileOut delegate) {  super(delegate);  }  public void write(byte[] data) {  byte[] encryptedData = encrypt(data);  super.doDelegate(encryptedData);  }  } |

BufferedOut, EncryptionOut, ZipOut 클래스는 모두 데코레이터 클래스, Decorator 클래스를 상속받고 있다.

자신의 기능 수행 후 doDelegate() 메서드를 이용해서 파일 쓰기를 위임하도록 구현.

|  |
| --- |
| **데이터를 암호화해서 쓰는 기능이 필요한 곳의 코드**  FileOut delegate = new FileOutImpl();  FileOut fileOut = **new EncryptionOut(delegate);**  fileOut.write(data); |
| **동작 로직**  EncryptionOut의 write() 메서드 실행  EncryptionOut의 write() 메서드에서 데이터 암호화 (EncryptionOut 객체는 FileOutImpl 객체가 제공하는 파일 쓰기 기능에 암호화 기능을 추가해 주는 역할 수행)  FileOutImpl 객체의 write() 메서드에 암호화된 데이터 전달 |

기존 기능에 새로운 기능을 추가해 준다는 의미에서 EncryptionOut 객체를 데코레이터(decorator)라고 부른다.

데코레이터 패턴 장점

* 데코레이터를 조합하는 방식으로 기능 확장 가능
* 기능 적용 순서의 변경도 쉽다.

|  |
| --- |
| **데이터 압축 > 암호화 > 파일에 쓰기**  FileOut delegate = new FileOutImpl();  FileOut fileOut = **new EncryptionOut(new ZipOut(delegate));**  fileOut.write(data); |
| **버퍼 > 암호화 > 압축 > 파일 쓰기**  FileOut fileOut = **new BufferedOut(EncryptionOut(new ZipOut(delegate)));** |
| **암호화 > 압축 > 버퍼 > 파일 쓰기**  FileOut fileOut = **new EncryptionOut(new ZipOut(new BufferedOut(delegate)));** |

* 데코레이터 패턴을 사용하면 각 확장 기능들의 구현이 별도의 클래스로 분리, 서로 영향 없이 변경할 수 있도록 만들어 준다.
* 데코레이터 패턴은 단일 책임 원칙을 지킬 수 있도록 만들어 준다.
* 매우 흔하게 사용되는 패턴
* 스프링 프레임워크의 경우 트랜잭션 처리를 위해 데코레이터 패턴 사용

데코레이터 대상이 되는 타입의 기능 개수에 대해 고려해야 함.

* 앞의 예는 write() 메서드 한 개라 구현이 간단했지만, 메서드가 많아지면 데코레이터의 구현도 복잡해진다.

데코레이터 객체가 비정상적으로 동작할 때 어떻게 처리할 것인지 고려해야 함

데코레이터의 단점

* 사용자 입장에서 데코레이터 객체와 실제 구현 객체의 구분이 되지 않음, 코드만으로는 기능이 어떻게 동작하는지 이해하기 어려움

|  |
| --- |
| public class ImageSource {  public void writeTo(FileOut out) {  out.write(imageData);  }  } |
| writeTo() 메서드는 파라미터로 전달받은 FileOut 객체를 사용  writeTo() 메서드는 이 FileOut 객체가 단순히 파일에 쓰기만 하는지 압축을 하는지 등의 여부를 알 수 없다.  실제로 객체가 어떻게 동작하는지 알려면 런타임에 생성된 객체의 구조를 이해해야 함. |

1. **프록시(proxy) 패턴**

실제 객체를 대신하는 프록시 객체를 사용해서 실제 객체의 생성이나 접근 등을 제어할 수 있도록 해 주는 패턴

스크롤 화면에서 ListUI 변경 없이 이미지 로딩 방식을 교체할 수 있도록 해 주는 패턴

|  |
| --- |
| interface Image {  void draw();  } |
| 리스트 7.15 프록시 기능을 수행하는 ProxyImage 구현  class ProxyImage implements Image {  private String path;  private RealImage image;  public ProxyImage(String path) {  this.path = path;  }  public void draw() {  if (image == null) {  **image = new RealImage(path); // 최초 접근 시 객체 생성**  }  **image.draw(); // RealImage 객체에 위임**  }  } |

ProxyImage 클래스가 프록시 패턴에서 프록시 역할을 한다.

ListUI 클래스는 Image 타입을 사용하기 때문에 실제 타입이 RealImage 인지 ProxyImage인지 모른다.

Image 타입의 draw()를 이용해서 이미지를 그려 달라고 할 뿐.

|  |
| --- |
| public class ListUI {  private List<Image> images;  public ListUI(List<Image> images) {  this.images = images;  }  public void onScroll(int start, int end) {  // 스크롤 시, 화면에 표시되는 이미지를 표시  for (int i = start; i <= end; i++) {  Image image = images.get(i);  image.draw();  }  }  } |
| images 필드에 보관된 Image 객체의 실제 타입이 ProxyImage인 경우  onScroll에서 ProxyImage 객체에 최초 draw() 메서드가 실행.  RealImage 객체 생성.  ProxyImage 객체의 draw() 메서드가 호출되기 전에는 RealImage 객체가 생성되지 않음  메모리에 이미지 데이터를 로딩하지 않음.  메모리를 낭비하는 상황 방지.  ListUI 클래스는 이미지가 언제 로딩되는지 알 필요가 없음.  이미지 로딩 정책을 변경해도 ListUI 클래스의 코드는 영향 받지 않음. |

|  |
| --- |
| 예) 상위 4개는 바로 이미지를 로딩, 나머지는 화면에 보여지는 순간에 로딩  RealImage 객체와 ProxyImage 객체를 섞어서 ListUI에 전달해 주면 된다.  class Proxy {  public static void main(String[] args) {  List<String> paths = new ArrayList<String>(); // 이미지 경로 목록을 가져옴  List<Image> images = new ArrayList<Image>(paths.size());  for (int i=0; i<paths.size(); i++) {  if (i <= 4)  images.add(new RealImage(paths.get(i)));  else  images.add(new ProxyImage(paths.get(i)));  }  **// 이미지 로딩 정책의 변경이 ListUI에 영향을 주지 않는다.**  ListUI listUI = new ListUI(images);  }  } |

가상 프록시(virtual proxy) : ProxyImage처럼 필요한 순간에 실제 객체를 생성해 주는 프록시

보호 프록시(protection proxy) : 실제 객체에 대한 접근을 제어하는 프록시, 접근 권한이 있는 경우에만 실제 객체의 메서드를 실행하는 방식으로 구현.

원격 프록시(remote proxy) : 다른 프로세스에 존재하는 객체에 접근할 때 사용되는 프록시, 내부적으로 IPC(Inter process communication)이나 TCP 통신을 이용해서 다른 프로세스의 객체를 실행. (예 : 자바의 RMI(Remote Method Invocation)

고려할 점

* 실제 객체를 누가 생성할 것인가?
  + 가상 프록시에서 실제 생성할 객체의 타입을 사용
  + 보호 프록시는 객체를 생성할 때 실제 객체를 전달. 실제 객체의 타입을 알 필요 없이 추상 타입 사용
* 위임 방식이 아닌 상속을 사용해서 구현 가능.
  + 특정 기능은 관리자만 실행할 수 있어야 한다고 할 경우 보호 프록시를 사용할 수 있음. 이때 보호 프록시는 다음과 같이 상위 클래스의 메서드를 재정의하는 방법으로 구현 가능

|  |
| --- |
| public class ProtectedService extends Service {  @Override  public void someMethod() {  if (! CurrentContext.getAuth().isAdmin())  throw new AccessDeniedException();  super.someMethod();  }  } |

* + 상속 방식을 사용하면 위임 방식에 비해 구조가 단순해서 구현이 쉬움
  + 상속 방식의 프록시는 객체를 생성하는 순간 실제 객체가 생성되기 때문에, 가상 프록시를 구현하기에는 적합하지 않음.

데코레이터 패턴과 프록시 패턴 차이

* 데코레이터 패턴은 기존 객체의 기능을 확장하는데 초점.
* 프록시 패턴은 실제 객체에 대한 접근을 제어하는데 초점.

1. **어댑터(Adapter) 패턴**

클라이언트가 요구하는 인터페이스와 재사용하려는 모듈의 인터페이스가 일치하지 않을 때 사용할 수 있는 패턴이 어댑터(Adapter) 패턴.

|  |
| --- |
| 리스트 7.16 TolrClient를 SearchService에 맞춰주는 객체 위임 방식 어댑터 구현  class SearchServiceTolrAdapter implements SearchService {  private TolrClient tolrClient = new TolrClient();  public SearchResult search(String keyword) {  // keyword를 tolrClient가 요구하는 형식으로 변환  TolrQuery tolrQuery = new TolrQuery(keyword);  // TolrClient 기능 실행  QueryResponse response = **tolrClient.query(tolrQuery);**  // TolrClient의 결과를 SearchResult로 변환  SearchResult result = convertToResult(response);  return result;  }  private SearchResult convertToResult(QueryResponse response) {  List<TolrDocument> tolrDocs = response.getDocumentList().getDocuments();  List<SearchDocument> docs = new ArrayList<SearchDocument>();  for (TolrDocument tolrDoc : tolrDocs) {  docs.add(new SearchDocument(tolrDoc.getId()));  }  return new SearchResult(docs);  }  } |
| SearchServiceTolrAdapter 클래스는 SearchService 인터페이스 구현  WebSearchRequestHandler 클래스의 코드 수정 없이 DB 기반 통합 검색에서 TolrClient를 이용한 통합 검색으로 구현 변경 가능 |

대표 예 : SLF4J(로깅 API)

* SLF4J가 제공하는 인터페이스와 각 로깅 프레임워크(log4j, 자바 로깅, LogBack)를 맞춰 주기 위해 어댑터 사용

어댑터 패턴은 개방 폐쇄 원칙을 따를 수 있도록 도와준다.

SearchServiceTolrAdapter 클래스는 TolrClient를 조립하는 방법으로 구현, TolrClient를 상속받는 방법으로 구현도 가능

상속을 이용하면 SearchServiceTolrAdapter 클래스의 search() 메서드는 상위 클래스 TolrClient에 정의된 메서드를 호출.

|  |
| --- |
| 리스트 7.17 TolrClient를 SearchService에 맞춰주는 클래스 상속 방식 어댑터 구현  class SearchServiceTolrAdapter **extends TolrClient** implements SearchService {  public SearchResult search(String keyword) {  // keyword를 tolrClient가 요구하는 형식으로 변환  TolrQuery tolrQuery = new TolrQuery(keyword);  // TolrClient 기능 실행  QueryResponse response = **super.query(tolrQuery);**  // TolrClient의 결과를 SearchResult로 변환  SearchResult result = convertToResult(response);  return result;  }  private SearchResult convertToResult(QueryResponse response) {  List<TolrDocument> tolrDocs = response.getDocumentList().getDocuments();  List<SearchDocument> docs = new ArrayList<SearchDocument>();  for (TolrDocument tolrDoc : tolrDocs) {  docs.add(new SearchDocument(tolrDoc.getId()));  }  return new SearchResult(docs);  }  } |

클라이언트가 사용하는 SearchService가 인터페이스가 아닌 일부 구현이 포함된 추상 클래스라면, 자바와 같이 단일 상속만을 지원하는 언어에서는 클래스 상속을 이용한 어댑터 구현에 제약을 받게 된다.

1. **옵저버(Observer) 패턴**

|  |
| --- |
| 웹 사이트의 상태를 확인해서 상태가 안좋으면 모니터링 담당자에게 이메일로 통지해주는 시스템  상태를 확인하는 StatusChecker 클래스  public class StatusChecker {  private EmailSender emailSender;  public void check() {  Status status = loadStatus();  if(status.isNotNormal())  emailSender.sendEmail(status);  }  } |
| 긴급한 메시지는 SMS로 바로 알려주는 기능을 추가해달라는 요구가 들어옴  public class StatusChecker {  private EmailSender emailSender;  private SmsSender smsSender;  public void check() {  Status status = loadStatus();  if(status.isNotNormal()) {  emailSender.sendEmail(status);  smsSender.sendSms(status);  }  }  } |

시스템의 문제 상황을 알려주는 방식이 추가될 때마다, StatusChecker 클래스도 변경된다.

상태가 변경될 때 정해지지 않은 임의의 객체에게 변경 사실을 알려준다.

한 객체의 상태 변화를 정해지지 않은 여러 다른 객체에 통지하고 싶을 때 사용되는 패턴이 옵저버(Observer) 패턴이다.

옵저버 패턴

* 주제(Subject) 객체
  + 옵저버 목록을 관리, 옵저버를 등록, 제거할수 있는 메서드 제공 (예 : add(), remove() 메서드)
  + 상태의 변경이 발생되면 등록된 옵저버에 변경 내역을 알린다. (예 : notifyStatus() 메서드가 -> onAbnormalStatus() 메서드 호출)
* 옵저버(Observer) 객체

|  |
| --- |
| 리스트 7.18 옵저버 패턴에서 주제에 해당하는 클래스의 구현  public abstract class StatusSubject {  private List<StatusObserver> observers = new ArrayList<>();  public void add(StatusObserver observer) {  observers.add(observer);  }  public void remove(StatusObserver observer) {  observers.remove(observer);  }  public void notifyStatus(Status status) {  for (StatusObserver observer : observers)  observer.onAbnormalStatus(status);  }  } |
| 옵저버 객체의 메서드를 호출하는 방식으로 상태에 변화가 생겼음을 옵저버 객체에게 알린다. |

|  |
| --- |
| 리스트 7.19 옵저버에게 통지가 필요한 콘크리트 클래스의 구현  public class StatusChecker extends StatusSubject {  public void check() {  Status status = loadStatus();  if(status.isNotNormal()) {  super.notifyStatus(status);  }  }  } |
| 비정상 상태가 감지되면 상위 클래스의 notifyStatus() 메서드 호출  등록된 옵저버 객체들에 상태 값을 전달 |

|  |
| --- |
| 리스트 7.20 옵저버 인터페이스  public interface StatusObserver {  void onAbnormalStatus(Status status);  } |

|  |
| --- |
| 리스트 7.21 콘크리트 옵저버 클래스의 구현 예  public class StatusEmailSender implements StatusObserver {  @Override  public void onAbnormalStatus(Status status) {  sendEmail(status);  }  private void sendEmail(Status status) {  // 이메일 전송 코드  }  } |

주제 객체에서 내용을 통지 받으려면, 옵저버 객체를 주제 객체에 등록해야 한다.

|  |
| --- |
| public class Observer {  public static void main(String[] args) {  StatusChecker checker = new StatusChecker();  checker.add(new StatusEmailSender()); // 옵저버로 등록  }  } |

옵저버 패턴 장점

* 주제 클래스 변경 없이 상태 변경을 통지 받을 옵저버를 추가할 수 있다.

옵저버 객체가 주제 객체의 상태에 따라 기능을 수행해야 할 수도 있다.

예를 들어 장애 상태에서만 수행하고, 비정상 상태인 경우에는 수행하지 않도록 하는 것.

현재 코드는 옵저버 객체에서 주제 객체에서 넘어온 상태 값을 확인해서 수행했다.

|  |
| --- |
| 옵저버는 파라미터로 전달받은 상태 값을 사용  public class FaultStatusSMSSender implements StatusObserver {  @Override  public void onAbnormalStatus(Status status) {  if (status.isFault()) // 전달받은 상태 값을 사용  sendSMS(status);  }  private void sendSMS(Status status) {  }  } |
| 전달받은 status 객체로도 원하는 기능 구현 가능  하지만, 경우에 따라서 전달받은 객체만으로는 기능 구현이 어려울 수도 있다. |

이런 경우 옵저버 객체에서 콘크리트 주제 객체에 직접 접근하는 방법을 사용하기도 한다.

|  |
| --- |
| public class SpecialStatusObserver implements StatusObserver {  private StatusChecker statusChecker;  private Siren siren;  public SpecialStatusObserver(StatusChecker statusChecker) {  this.statusChecker = statusChecker;  }  @Override  public void onAbnormalStatus(Status status) {  // 특정 타임의 주제 객체에 접근  if (status.isFault() && **statusChecker.isContinuousFault()**)  siren.begin();  }  } |
| status 파라미터와 statusChecker 필드를 이용해서 사이렌의 실행 조건을 판단  SpecialStatusObserver 클래스에서 StatusChecker 클래스로의 의존 발생  콘크리트 옵저버 클래스(SpecialStatusObserver)는 필요에 따라 특정한 콘크리트 주제 클래스(StatusChecker)에 의존하게 됨. |

옵저버 패턴이 가장 많이 사용되는 영역 : GUI 프로그래밍 영역

버튼(주제 객체) -> 로그인 기능 호출(옵저버) -> 로그인

안드로이드에서의 예시

예) Button 객체에 등록하는 OnClickListener 인터페이스가 옵저버 인터페이스가 된다.

한 개의 옵저버 객체를 여러 주제 객체에 등록할 수도 있다.

예) 로그인 버튼, 로그아웃 버튼에 같은 OnClickListener 객체를 연결

한 옵저버 객체에서 여러 주제 객체를 등록하면, 각 주제 객체를 구분할 수 있는 방법 필요.

OnClickListener 객체는 onClick() 메서드에서 주제 객체인 Button 객체를 구분하기 위해 ID 값을 사용. ID 값 외에 객체 레퍼런스도 사용 가능

StatusChecker, 안드로이드 예제는 모두 주제 객체를 위한 추상 타입을 제공

StatusCheck -> StatusSubject 추상 클래스

안드로이드의 Button -> View

두 상위 클래스는 모두 옵저버 객체를 관리하기 위한 기능을 제공

한 주제에 대한 다양한 구현 클래스가 존재한다면 위의 내용처럼 옵저버 객체 관리 및 통지 기능을 제공하는 추상 클래스를 제공, 불필요하게 동일한 코드가 여러 주제 클래스에서 중복되는 것 방지.

주제 클래스가 한 개뿐이라면, 옵저버 관리를 위한 추상 클래스를 따로 만들 필요는 없다.

옵저버 패턴 구현 고려 사항

* 주제 객체의 통지 기능 실행 주체
  + 앞의 예는 옵저버에 통지하는 주체가 StatusChecker 클래스
  + 필요에 따라 StatusChecker를 사용하는 코드에서 통지 기능을 수행할 수도 있다.
    - 예) StatusChecker 객체로부터 상태 정보를 읽어와 모두 비정상인 경우에만 통지
  + 주제 객체의 상태가 바뀔 때마다 옵저버에게 통지를 해야 한다면, 주제 객체에서 직접 통지 기능을 실행하는 것이 구현에 유리.
  + 한 개 이상의 주제 객체의 연속적인 상태 변경 이후, 옵저버에게 통지 해야 한다면, 주제 객체의 상태를 변경하는 코드에서 통지 기능을 실행해 주도록 구현하는 것이 통지 시점을 관리하기 수월
* 옵저버 인터페이스의 분리
  + 한 주제 객체가 통지할 수 있는 상태 변경 내역의 종류가 다양한 경우에는 각 종류 별로 옵저버 인터페이스를 분리해서 구현
    - 모든 종류의 상태 변경을 하나의 옵저버 인터페이스로 처리할 경우, 옵저버 인터페이스는 거대해질 것 -> 불필요한 코드를 만들어야 함.
* 통지 시점에서의 주제 객체 상태
  + 주제 객체의 상태에 결함이 없어야 한다.

|  |
| --- |
| public class AnySubject extends SomeSubject {  @Override  public void changeState(int newValue) {  super.changeState(newValue); // 상위 changeState()에서 옵저버에 통지  // 아래 코드가 실행되기 전에 옵저버가 상태를 조회  if (isStateSome()) {  state += newValue;  }  }  } |
| 상위 타입의 changeStatus() 메서드에서 옵저버에 통지를 하게 되면,  옵저버 객체는 완전하지 못한 상태 값을 조회하게 된다. |

* + 옵저버 객체가 올바르지 않은 상태 값을 사용하게 되는 문제가 발생하지 않도록 하는 방법
    - 상태 변경과 통지 기능에 템플릿 메서드 패턴을 적용하는 것

|  |
| --- |
| // 상위 클래스  public class SomeSubject {  // 템플릿 메서드로 구현  public void changeState(int newState) {  internalChangeState(newState);  notifyObserver();  }  } |
| // 하위 클래스  public class AnySubject extends SomeSubject {  // internalChangeState() 메서드 실행 이후에, 옵저버에 통지  @Override  public void internalChangeState(int newValue) {  super.internalChangeState(newValue);  if (isStateSome()) {  state += newValue;  }  }  } |
| 상위 클래스의 changeState() 메서드 > internalChangeState() 메서드 호출  notifyObserver() 메서드 호출 > 옵저버에 상태 변화 통지  옵저버는 완전한 상태 값을 사용할 수 있게 됨. |

* 옵저버 객체의 실행 제약 조건
  + - 제약 규칙을 정해야 한다.

|  |
| --- |
| private void notifyToObserver() {  for (StatusObserver o : observers) {  o.onStatusChange();  }  }  public void changeState(int newState) {  internalChangeState(newState);  notifyToObserver();  } |
| onStatusChange() 메서드 실행 시간이 10분 이상 걸리면, 종료 될 때까지 100분 걸림  한 개의 옵저버로 인해 다른 옵저버의 실행이 지연되는 상황이 발생 |

* + - 옵저버 인터페이스를 정할 때에는 옵저버 메서드의 실행 제한에 대한 명확한 기준이 필요.
    - 안드로이드는 사용자 이벤트에 대해 5초 이상 프로그램이 응답하지 않으면 ANR(Application Not Responding) 대화 상자를 띄움

이 외 고려 사항

* 옵저버 객체에서 주제 객체의 상태를 다시 변경 할 경우 어떻게 구현할 것인가
* 옵저버 자체를 비동기로 실행하는 문제

1. **미디에이터(Mediator) 패턴**

* 각 객체들이 직접 메시지를 주고 받는 대신, 중간에 중계 역할을 수행하는 미디에이터 객체를 두고 미디에이터를 통해서 각 객체들이 간접적으로 메시지를 주고 받을 수 있도록 하는 것.

비디오 플레이어의 주요 기능

* 동영상 목록에서 특정 제목을 클릭하면, 제목에 해당하는 동영상이 플레이 영역에 표시되고, 제목 표시 영역에 해당 제목을 표시.
* 영상 제어 영역에서 재생/멈춤 버튼을 누르면 영상 플레이 영역은 영상을 재생하거나 멈추고, 시간 이동 조작을 하면 영상 플레이 영역은 알맞은 시점으로 이동해서 영상을 재생.
* 영상 제어 영역에서 다음/이전 버튼을 누르면 영상 목록에서 다음이나 이전 영상을 재생.
* 영상 플레이 화면을 터치하면 플레이가 멈추고 다시 터치하면 재생. 이때 영상 제어 부분의 UI도 중지/시작 모양으로 알맞게 변경.

책임에 맞게 클래스 분리 : MediaController, VideoPlayer, VideoListUI, TitleUI

단점

* 재사용 어려움
* 함께 사용되는 클래스가 증가 > 개별 클래스의 수정이 어려움
  + 플레이 화면 좌측에 시스템이 추천한 영상 목록을 표시 : CaptionUI 추가
  + 영상 자막 존재 시, 자막 표시 : RecommendListUI
  + 재사용이 높아질 거라 예상 > 단일 구조가 되어버림
  + 객체 간의 의존이 직접 연결되어 있음

객체 간의 메시지 흐름을 각 클래스에 직접적인 의존으로 구현 > 개별 클래스의 재사용 어려움 > 메시지 흐름을 변경하려면 관련된 클래스들을 모두 변경해야 하는 문제

미디에이터 패턴 적용

* VideoMediator 클래스 생성
* VideoPlayer, MediaController, VideoListUI, TitleUI는 협업 대상(Colleague) 역할 수행
* 협업 대상 객체들은 미디에이터 객체를 통해 간접적으로 연결

|  |
| --- |
| 리스트 7.22 협업 객체는 미디에이터에 요청한다.  public class VideoListUI {  private VideoMediator videoMediator;  public void onSelectedItem(int selectedIdx) {  VideoInfo videoInfo = videoList.get(selectedIdx);  videoMediator.selectVideo(videoInfo.getFile());  }  } |
| VideoMediator 클래스에 대한 의존만 갖고 있음.  해당 객체는 목록에서 특정 비디오가 선택되면, VideoMediator 객체의 메서드 호출 |

|  |
| --- |
| 리스트 7.23 요청을 받은 미디에이터 객체는 협업 객체들에게 요청을 전달한다.  public class VideoMediator {  private VideoPlayer videoPlayer;  private TitleUI titleUI;    public void selectVideo(File videoFile) {  // 미디에이터는 다른 협업 객체에게 요청을 전달  videoPlayer.play(videoFile);  titleUI.setTitle(videoFile);  }  } |

다른 협업 객체들도 모든 요청을 미디에이터에 전달.

미디에이터는 요청을 처리할 알맞은 객체를 실행

협업 객체가 서로 알 필요 없이 미디에이터가 각 객체 간의 메시지 흐름을 제어.

새로운 객체가 추가되어도 미디에이터만 수정.

옵저버 패턴을 통해 협업 클래스의 재사용을 높일 수 있다.

협업 클래스는 미디에이터에만 의존하거나 옵저버 패턴을 이용해 미디에이터, 다른 협업 클래스에 의존하지 않게 되면서, 수정, 확장, 재사용하기가 쉬워진다.

미디에이터에 흐름 제어 코드가 모여있어, 전체 메시지 흐름을 이해하기 쉽고, 수정, 확장하는 것을 상대적으로 쉽게 만들어 준다.

미디에이터 패턴의 단점

* 협업 클래스의 개수가 증가할수록 미디에이터 코드는 복잡해짐.
* 복잡해지면서 유지보수하기가 어려워짐.

추상 미디에이터 클래스 : 협업 객체 간의 동일한 메시지 흐름이 서로 다른 기능에서도 반복되서 사용될 경우, 추상 미디에이터 클래스를 사용. 재사용을 높일 수 있다.

예) MP3 플레이어, 비디오 플레이어 > 미디어 제어기, 플레이 목록, 제목 표시가 비슷함 > 추상 미디에이터 클래스를 만들고 > 하위 미디에이터를 만듬 > 객체 간의 협업 흐름 재사용

|  |
| --- |
| 리스트 7.24 추상 미디에이터 클래스는 협업 객체 간 메시지 흐름을 재사용할 수 있도록 도와준다.  public abstract class PlayerMediator implements ControllerObserver {  private MediaController mediaController;  private TitleUI titleUI;  public PlayerMediator() {  this.mediaController = new MediaController();  this.mediaController.addObserver(this);  this.titleUI = new TitleUI();  }    public void select(File file) {  titleUI.setTitle(file);  }  } |
| select() 메서드는 구현을 제공  volumeChanged() 메서드는 추상 메서드  volumeChanged() 의 구현은 제공하지 않음  MediaController 객체의 볼륨 조절 이벤트 발생 시 volumeChanged() 메서드 호출  하위 클래스에서는 select() 메서드를 재사용해서 기능을 확장.  volumeChanged() 메서드는 추상 메서드이므로 흐름 제어를 재사용하기보다는 하위 클래스에서 알맞게 기능을 구현 |

|  |
| --- |
| PlayerMediator 클래스를 재사용하면서 필요한 기능 확장  public class VideoPlayerMediator **extends PlayerMediator** {  private VideoPlayer videoPlayer;  public VideoPlayerMediator() {  super();  this.videoPlayer = new VideoPlayer();  }  @Override  public void select(File file) {  videoPlayer.play(file);  **super.select(file); // 상위 미디에이터에 정의된 협업 기능 재사용**  }  // 하위 미디에이터에서 새로운 협업 기능 구현  public void volumeChanged(int volume) {  videoPlayer.changeVolume(volume);  }  } |
| select() 메서드 재정의  동시에 비디오 플레이를 위한 객체와의 협업 기능 추가  ListUI, MediaController, TitleUI 객체 간의 메시지 흐름 재사용 > 비디오 관련 기능 확장 |

1. **파사드(Facade) 패턴**

기본 : GUIViewer 클래스가 EmpDao(직원 정보), ResumeDao(직원 이력 정보), EvaluationDao(직원 평가 정보)를 읽어온다.

요구사항 추가 : xml, 엑셀에서도 동일하게 보여질 수 있도록 하기.

수정 : XMLExporter, ExcelExporter가 GUIViewer와 동일한 작업을 하게 된다.

문제점

* XMLExporter, ExcelExporter, GUIViewer 사이에서 코드 중복 발생
* 동일한 코드를 이용해 세 Dao에서 데이터 추출
* 코드가 완전히 같진 않고 미세하기 조금씩 다름
* 나중에 변경해 주어야 할 때 미세한 차이점으로 인해 버그가 생길 수 있음
* 세 Dao가 제공하는 데이터를 통합한 하나의 데이터인데 각각에 직접적인 의존을 하고 있음
* Dao의 인터페이스에 일부 변화가 발생하면 Dao를 직접적으로 사용하고 있는 뷰어에 모두 영향을 미친다.

코드 중복, 직접적인 의존을 해결해주는 패턴이 파사드(Facade) 패턴.

파사드 패턴은 서브 시스템을 감춰 주는 상위 수준의 인터페이스를 제공

EmpReportDaoFacade를 생성함으로써 클라이언트와 서브 시스템은 해당 객체와 연결된다.

|  |  |
| --- | --- |
| public class GuiViewer {  public void display() {  Emp emp = empDao.select(id);  Resume resume = resumeDao.select(id);  Evaluation eval = evaluationDao.select(id);  }  } | public class ExcelExporter {  public void export(File file) {  Emp emp = empDao.select(id);  Resume resume = resumeDao.select(id);  Evaluation eval = evaluationDao.select(id);  }  } |
| 파사드 패턴 적용 전 | |

|  |  |
| --- | --- |
| public class GuiViewer {  public void display() {  EmpReport rep = empReportDaoFacade.select(id);  }  } | public class ExcelExporter {  public void export(File file) {  EmpReport rep = empReportDaoFacade.select(id);  }  } |
| 파사드 패턴 적용 후 | |

파사드 패턴 적용 후 코드가 간결해지고, 클라이언트와 서브 시스템 간의 직접적인 의존을 제거했다.

클라이언트는 파사드에만 의존

서브 시스템이 변경되도 파사드에만 영향을 준다.

파사드를 인터페이스로 정의하면, 클라이언트의 변경 없이 서브 시스템 자체를 변경 가능

여러 클라이언트에 중복된 서브 시스템 사용을 파사드로 추상화할 뿐.

보다 세밀한 제어가 필요한 경우에는 서브 시스템에 직접 접근하는 방식도 선택 가능.

파사드를 통해 서브 시스템의 상세한 구현을 캡슐화.

1. **추상 팩토리(Abstract Factory) 패턴**

비행기 게임 (슈팅 게임) 구현

* 여러 종류의 적 출현
* 한 단계의 끝에 보스 출현
* 보스 잡으면 다음 단계로
* 중간에 장애물이 출현
* 그 외 다양한 적기들, 공격력, 방어력이 다른 적기들 등등

클래스

* Boss
* SmallFlight
* Obstacle
* Stage : 단계에 따라 적기, 장애물, 보스 생성

|  |
| --- |
| public class Stage {  private void createEnemies() {  for (int i=0; i<=ENEMY\_COUNT; i++) {  if (stageLevel == 1) {  enemies[i] = new DashSmallFlight(1, 1); // 공력/수비력 1  } else if (stageLevel == 2) {  enemies[i] = new MissileSmallFlight(1, 1);  }  }  if (stageLevel == 1) {  boss = new StrongAttackBoss(1, 10);  } else if (stageLevel == 2) {  boss = new CloningBoss(5, 20);  }  }  private void createObstable() {  for (int i=0; i<OBSTACLE\_COUNT; i++) {  if (stageLevel == 1) {  obstacles[i] = new RockObstacle();  } else {  obstacles[i] = new BombObstacle();  }  }  }  } |

위 코드의 문제점

* 적기, 보스, 장애물 생성이 Stage 클래스에 포함되어 있다.
* 클래스에 다양한 변수들이 발생하면 Stage 클래스도 같이 수정해줘야 한다.
* 중첩, 연속된 조건문이 존재한다.

해결법

* Stage 클래스로부터 객체 생성 책임을 분리한다.
* 이 때 사용되는 패턴이 추상 팩토리 패턴

추상 팩토리 패턴에서는 관련된 객체 군을 생성하는 책임을 갖는 타입을 별도로 분리

SmallFlight, Boss, Obstacle 객체를 생성해 주는 책임을 갖는 EnemyFactory 타입 추가

EnemyFactory

* SmallFlight, Boss, Obstacle 객체를 생성하는 메서드 정의
* 객체 생성 메서드를 선언하는 추상 타입 팩토리
* SmallFlight, Boss, Obstacle은 제품 타입
* EnemyFactory.getFactory() 메서드는 정적 메서드. 전달 받은 레벨에 따라 알맞은 EnemyFactory 객체 리턴

|  |
| --- |
| 리스트 7.25 추상 팩토리 패턴의 추상 팩토리 구현 예  public abstract class EnemyFactory {  public static EnemyFactory getFactory(int level) {  if (level == 1) {  return new EasyStageEnemyFactory();  } else {  return new HardEnemyFactory();  }  }    // 객체 생성을 위한 팩토리 메서드  public abstract Boss createBoss();  public abstract SmallFlight createSmallFlight();  public abstract Obstacle createObstacle();  } |

|  |
| --- |
| 리스트 7.26 콘크리트 팩토리 클래스 구현 예  public class EasyStageEnemyFactory extends EnemyFactory {  @Override  public Boss createBoss() {  return new StrongAttackBoss();  }  @Override  public SmallFlight createSmallFlight() {  return new DashSmallFlight();  }  @Override  public Obstacle createObstacle() {  return new RockObstacle();  }  }  public class HardEnemyFactory extends EnemyFactory {  @Override  public Boss createBoss() {  return new CloningBoss();  }  @Override  public SmallFlight createSmallFlight() {  return new MissileSmallFlight();  }  @Override  public Obstacle createObstacle() {  return new BombObstacle();  }  } |

|  |
| --- |
| Stage 클래스는 추상 팩토리 타입인 EnemyFactory를 이용해서 객체 생성  public class Stage {  public Stage(int level) {  // EnemyFactory 객체를 구함  **enemyFactory = EnemyFactory.getFactory(level);**  }  private void createEnemies() {  for (int i = 0; i <= ENEMY\_COUNT; i++)  enemies[i] = enemyFactory.createSmallFlight();  boss = enemyFactory.createBoss();  }  private void createObstable() {  for (int i = 0; i < OBSTACLE\_COUNT; i++)  obstacles[i] = enemyFactory.createObstacle();  }  } |
| 콘크리트 클래스 사용 안함  추상 타입만 사용 |

적 생성 규칙 변경 시, 새로운 팩토리 클래스를 만들면 됨

DI를 사용해도 됨 > 생성자나 설정 메서드를 통해 객체를 전달받게 됨 > getFactory() 메서드를 정의할 필요가 없어짐 > EnemyFactory 추상 클래스를 인터페이스로 전환 가능

|  |
| --- |
| public class Stage {  private EnemyFactory enemyFactory;  //DI를 적용하면 팩토리를 구하는 기능을 EnemyFactory에 구현할 필요가 없음  public Stage(int level, EnemyFactory enemyFactory) {  this.level = level;  this.enemyFactory = enemyFactory;  }  } |

추상 팩토리 패턴 장점

* 클라이언트에 영향을 주지 않으면서 사용할 제품(객체) 군을 교체할수 있다

팩토리가 생성하는 객체가 늘 동일한 상태면, 프로토타입 방식으로 팩토리를 구현

|  |
| --- |
| public class Factory {  private ProductA productAProto;  private ProductB productBProto;  public Factory(ProductA productAProto, ProductB productBProto) {  this.productAProto = productAProto;  this.productBProto = productBProto;  }  public ProductA createA() {  return (ProductA) productAProto.clone();  }    public ProductB createB() {  return (ProductB) productBProto.clone();  }  } |
| 프로토타입 방식은 생성할 객체의 원형 객체를 등록  객체 생성 요청이 있으면 원형 객체를 복제해서 생성 |

프로토타입 방식의 팩토리를 사용하면, 객체 군 마다 팩토리 객체를 생성해주면 됨.

|  |
| --- |
| // 객체 군 1을 위한 팩토리 객체  Factory family1Factory = new Factory(new HighProductA(), new HighProductB());  ProductA a = family1Factory.createA(); // HighProductA 객체 복제본 생성  // 객체 군 2를 위한 팩토리 객체  Factory family2Factory = new Factory(new LowProductA(), new LowProductB());  ProductB b = family2Factory.createB(); // LowProductB 객체 복제본 생성 |

프로토타입 방식은 추상 팩토리 타입과 콘크리트 팩토리 클래스를 따로 만들 필요가 없어 구현은 쉽다. 다만 제품 객체의 생성 규칙이 복잡할 경우 적용할 수 없는 한계가 있다.

대표적인 추상 팩토리 패턴을 적용한 예

* 자바의 JDBC API

1. **컴포지트(Composite) 패턴**

상황>

빌딩의 장비들의 전원 관리 제어 프로그램 개발

개별 장비의 전원을 켜고 끄는 기능을 제공하는 인터페이스

장비 별로 알맞은 콘크리트 클래스 구현

장비들을 하나로 묶어서 관리할 수 있도록 DeviceGroup 클래스 추가

장비들의 전원을 제어하는 코드는 다음과 같이 Device 타입과 DeviceList 타입을 구분해서 처리

|  |
| --- |
| public class PowerController {  public void turnOn(Long deviceId) {  Device device = findDeviceById(deviceId);  device.turnOn();  }    **// turnGroupOn()과 turnOn()은 개별/그룹 차이를 빼면 동일한 기능이다.**  public void turnGroupOn(Long deviceId) {  DeviceGroup group = findGroupDeviceById(deviceId);  group.turnAllOn();  }  } |

단점 : PowerController 입장에서는 장비, 장비 그룹이 동일한 동작을 한다. 그러나 처리는 따로.

새로운 기능이 추가될 경우 두 메서드에 동일한 기능을 추가해야 한다.

동일한 코드 중복 > 복잡도 증가 > 코드 수정 및 확장 어려움

컴포지트(Composite) 패턴

* 전체와 부분을 구성하는 클래스가 동일 인터페이스를 구현하도록 만드는 것

컴포지트 패턴의 책임

* 컴포넌트 그룹을 관리한다.
* 컴포지트에 기능 실행을 요청하면, 컴포지트는 포함하고 있는 컴포넌트들에게 기능 실행 요청을 위임한다.

|  |
| --- |
| 리스트 7.27 컴포지트에 해당하는 DeviceGroup 클래스의 구현  public class DeviceGroup implements Device {  private List<Device> devices = new ArrayList<>();  public void addDevice(Device d) {  devices.add(d);  }  public void removeDevice(Device d) {  devices.remove(d);  }  @Override  public void turnOn() {  for (Device device : devices) {  device.turnOn(); // 관리하는 Device 객체들에게 실행 위임  }  }  @Override  public void turnOff() {  for (Device device : devices) {  device.turnOff(); // 관리하는 Device 객체들에게 실행 위임  }  }  } |
| 실행 예시)  Device device1 = ...  Device device2 = ...  DeviceGroup group = new DeviceGroup();  group.addDevice(device1);  group.addDevice(device2);  group.turnOn(); // device1과 device2의 turnOn() 실행 |

|  |
| --- |
| Device 타입만을 이용해서 전원 관리 가능해짐  public class PowerController {  public void turnOn(Long deviceId) {  Device device = findDeviceById(deviceId);  device.turnOn();  }  // 이제, 아래 메서드는 필요 없음  ~~public void turnGroupOn(Long deviceId) {~~  ~~DeviceGroup group = findGroupDeviceById(deviceId);~~  ~~group.turnAllOn();~~  ~~}~~  } |

컴포지트 자체도 컴포넌트라서 컴포지트에 다른 컴포지트 등록이 가능.

|  |
| --- |
| // 각 층의 Light 객체를 모은 그룹 생성  DeviceGroup firstFloorLightGroup = ...; // 1층의 모든 Light를 포함  DeviceGroup secondFloorLightGroup = ...; // 2층의 모든 Light를 포함  ...  // 각 층의 DeviceGroup 객체를 하나더 DeviceGroup에 다시 등록  DeviceGroup allLightGroup = new DeviceGroup();  allLightGroup.add(firstFloorLightGroup);  allLightGroup.add(secondFloorLightGroup);  ...  allLightGroup.turnOff(); // 모든 층의 Light 객체의 turnOff() 실행 |

컴포지트 패턴 구현시 고려사항

* 컴포넌트를 관리하는 인터페이스를 어디서 구현할 것인가

컴포지트 패턴의 장점 : 하나의 클라이언트가 컴포지트와 컴포넌트를 구분하지 않고 컴포넌트 인터페이스만으로 프로그래밍 할 수 있게 돕는다는 점

|  |
| --- |
| public void addDeviceTo(Device device, Integer toDeviceId) {  Device composite = findDevice(toDeviceId);  composite.addDevice(device);  } |
| // Device에 addDevice()와 removeDevice()에 대한 기본 기능 구현  public abstrace class Device {  public void addDevice(Device d) {  **throw new CanNotAddException(“추가할 수 없음”);**  }  public void removeDevice(Device d) {  // 아무 것도 하지 않음  }  public abstract void turnOn();  public abstract void turnOff();  } |
| // GroupDevice에서 알맞게 재정의  public class GroupDevice extends Device {  @Override  public void addDevice(Device d) {  devices.add(d);  }  @Override  public void removeDevice(Device d) {  devices.remove(d);  }  ...  } |
| public void addDeviceTo(Device device, Integer toDeviceId) {  Device composite = findDevice(toDeviceId);  try {  composite.addDevice(device);  } **catch(CanNotAddException ex) {**  **... // 추가할 수 없는 경우의 처리**  **}**  } |

익셉션을 발생시키는 방법보다 괜찮은 방법은, 컴포넌트에 추가할 수 있는지의 여부를 판단해주는 기능을 Device 타입에 정의하는 것.

|  |
| --- |
| // Device에 canContain() 추가  public abstract class Device {  ...  **public boolean canContain(Device devices) {**  **return false;**  **}**  } |
| // GroupDevice에서 알맞게 재정의  public class GroupDevice extends Device {  ...  **@Override**  **public void canContain(Device devices) {**  **return true;**  **}**  } |
| public void addDeviceTo(Device device, Integer toDeviceId) {  Device composite = findDevice(toDeviceId);  **if (composite.canContain(device)) {**  **composite.addDevice(device);**  **return;**  **}**  ... // 추가할 수 없는 경우의 처리  } |
| canContain() 메서드는 객체 등록 여부뿐만 아니라 컴포지트에서 포함할 객체를 제한할 때에도 사용할 수 있다.  public class OnlyAirconContainingGroup extends GroupDevice {  @Override  public boolean canContain(Device device) {  return (device instanceof Aircon);  }  } |

1. 널(Null) 객체 패턴

**고객의 상태(장기 고객, 신규 고객)에 따라 특별 할인**

사용 요금 명세서를 생성하는 기능 > 명세서 상세 내역에 특별 할인 기능 추가

|  |
| --- |
| public Bill createBill(Customer customer) {  Bill bill = new Bill();  // ... 사용 내역 추가  bill.addItem(new Item("기본사용요금", price));  bill.addItem(new Item("할부금", somePrice));    // 특별 할인 내역 추가  SpecialDiscount specialDiscount = specialDiscountFactory.create(customer);  if (**specialDiscount != null**) { // 특별 할인 대상인 경우만 처리  specialDiscount.addDetailTo(bill);  }  } |

개발자가 null 검사 코드를 빼먹기 쉽다는 단점이 있다.

여러 코드에서 한 객체에 대한 null 검사를 하게 되면 null 검사 코드를 누락하기 쉽다.

NullPointerException을 발생시킬 가능성을 높여준다.

널 객체 패턴

* null을 리턴하지 않고 null을 대신할 객체를 리턴함으로써 null 검사 코드를 없앨 수 있도록 한다.

널 객체 패턴 구현

* null 대신 사용될 클래스를 구현한다. 이 클래스는 상위 타입을 상속받으며, 아무 기능도 수행하지 않는다.
* null을 리턴하는 대신, null을 대체할 클래스의 객체를 리턴한다.

|  |
| --- |
| 리스트 7.28 널 객체 패턴에서 널 객체에 해당하는 클래스 구현  public class NullSpecialDiscount extends SpecialDiscount {  @Override  public void addDetailTo(Bill bill) {  // 아무 것도 하지 않음  }  } |
| public class SpecialDiscountFactory {  public SpecialDiscount create(Customer customer) {  if (checkNewCustomer(customer))  return new NewCustomerSpecialDiscount();  ... // 다른 코드 실행  // 특별 할인 혜택이 없을 때, null 대신 NullSpecialDiscount 객체 리턴  **return new NullSpecialDiscount();**  }  } |
| public Bill createBill(Customer customer) {  Bill bill = new Bill();  // ... 사용 내역 추가  bill.addItem(new Item("기본사용요금", price));  bill.addItem(new Item("할부금", somePrice));  // 특별 할인 내역 추가  **// 특별 할인 대상이 아닐 경우 NullSpecialDiscount 객체 리턴**  SpecialDiscount specialDiscount = specialDiscountFactory.create(customer);  specialDiscount.addDetailTo(bill); **// null 검사 불필요**  return bill;  } |

널 객체 패턴 장점

* null 검사 코드를 사용할 필요가 없음
* 그래서 코드가 간결해짐 > 가독성 증가 > 코드 수정 쉬움