GRASP: 책임할당에 기반한 객체 설계 원칙

목표

- 책임할당 패턴을 정의 한다.
- 다섯 가지의 GRASP패턴을 적용하는 방법을 학습한다.

개요

- “객체설계”란…
  - 요구사항을 식별하고 도메인 모델을 생성한 후에, 적절한 소프트웨어 클래스에 메소드들을 추가하고, 요구사항(기능)을 수행하기 위하여 객체들간의 메시지 전달을 정의하라.
- How?
  - 어떤 객체가 어떤 책임을 감당하는가?
  - ==어떤 메소드가 어떤 객체에 속하는가,
  - 객체들이 어떻게 교류하여 일을 수행하는가 결정.
  - “객체설계 단계”는 객체지향 설계의 핵심 단계
- GRASP: 일반적인 책임할당 패턴.
  - General Responsibility Assignment Software Pattern
  - 기본 객체설계 학습을 위한 방법론

1. UML 대 설계 원칙

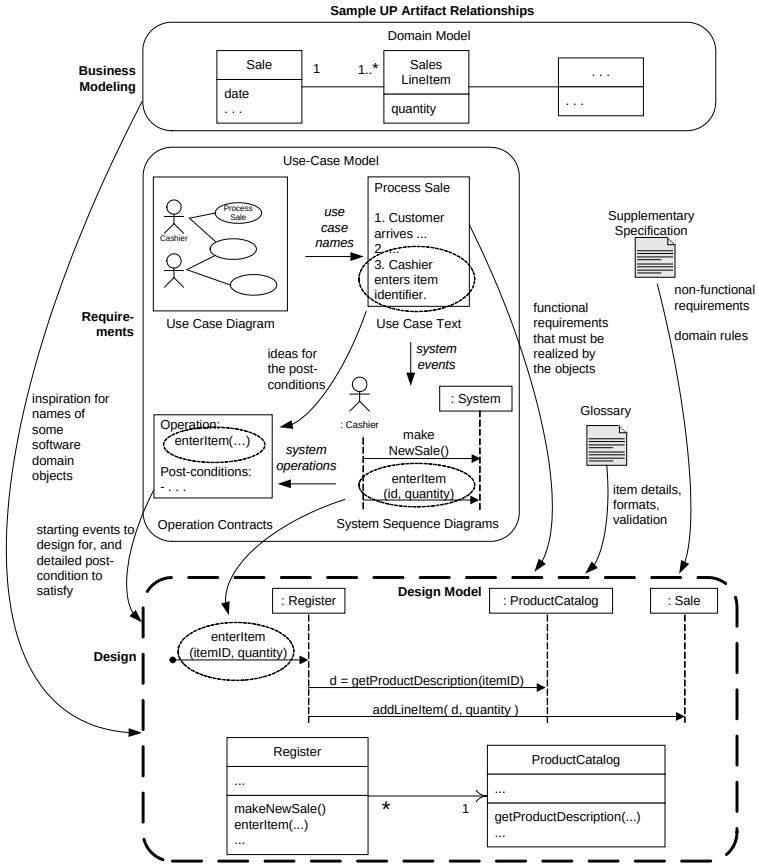
- 가장 중요한 설계 도구
  - 소프트웨어 개발에 있어 가장 중요한 설계 도구는 설계 원칙에 따라 잘 교육된 마음가짐이다.
- UML은 단지 표기법에 불과할 뿐...
  - UML 표기법을 알았다고 해서 객체설계를 할 수 있는 것은 아니다.
  - 객체 설계의 원리를 알아야 함.

2. 객체 설계: 입력, 출력, 활동

- 객체 설계 후의 출력
  - UML 순차도, 설계 클래스도
- 객체 설계를 위한 입력 (선택적)
  - UC text, SSD, 시스템 연산 약정
  - 보충명세서
  - 논리적 아키텍처
  - 도메인 모델
  - 용어집
- 객체 설계 활동
  - 테스트 주도 개발(코딩 즉시 테스트), UML 모델링, CRC 카드 모델링
  - 동적, 정적 모델링
    - GRASP, GoF 패턴을 적용
  - 책임주도 설계 (RDD: responsibility driven design)
    - 메타포어

RDD, GRASP, GoF 디자인 패턴

산출물의 관계


Sample UP Artifact Relationships
Business Modeling
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
Use-Case Model
Process Sale
Cashier
use case names
1. Customer arrives ...
2. ...
3. Cashier enters item identifier.
Use Case Diagram
Use Case Text
Require-ments
Supplementary Specification
non-functional requirements
domain rules
functional requirements that must be realized by the objects
ideas for the post-conditions
system events
: System
: Cashier
make NewSale()
enterItem (id, quantity)
Operation:
enterItem(...)
Post-conditions:
- . . .
system operations
Glossary
item details, formats, validation
inspiration for names of some software domain objects
Operation Contracts
System Sequence Diagrams
starting events to design for, and detailed post-condition to satisfy
Design Model
Design
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
d = getProductDescription(itemID)
addLineItem(d, quantity)
Register
...
makeNewSale()
enterItem(...)
...
*
1
ProductCatalog
...
getProductDescription(...)
...

3. 책임과 책임주도 설계

- 소프트웨어 객체 또는 컴포넌트 개발의 사고방식
  - 책임, 역할, 협력
  - 책임주도설계 (RDD: Responsibility Driven Design): 객체의 설계 시 책임 할당을 위주로 설계 하는 사고방식
- 책임(responsibility)
  - 분류자(클래스,컴포넌트,UC등)의 약정 혹은 의무
  - 책임의 종류
    - 인지책임: 아는 것
    - 실행책임: 하는 것
  - 실행 책임(doing) – 할 책임
    - 객체를 생성하거나 계산을 하는 것과 같이 무엇을 하는 것
    - 다른 객체의 행동을 시작시키는 것
    - 다른 객체의 활동을 제어하거나 조정하는 것
  - 인지책임 (knowing) – 알 책임
    - 전용의(private) 비공개 데이터에 관해 아는 것
    - 관련된 객체에 관하여 아는 것
    - 자신이 도출하거나 계산할 수 있는 결과에 관하여 아는 것.

책임과 객체

- 책임할당
  - 설계 시에 객체의 클래스에 할당시킴
  - “Sale” 객체는 “SalesLineItem”을 생성할 책임이 있다.(실행책임)
  - “Sale“은 판매합계를 알고 있을 책임이 있다. (인지 책임)
  - ➔ 도메인 모델로부터 추론

RDD는 메타포어이다.

RDD는 객체지향 소프트웨어 설계를 고찰하기 위한 일반적인 메타포어이다. 소프트웨어 객체를 어떤 일을 하기 위해서 다른 사람과 협력해야 하는 책임을 가진 사람으로 생각하자. RDD는 객체지향 설계를 책임을 가진 객체들이 서로 협력하는 공동체로서 바라보게 해 준다.

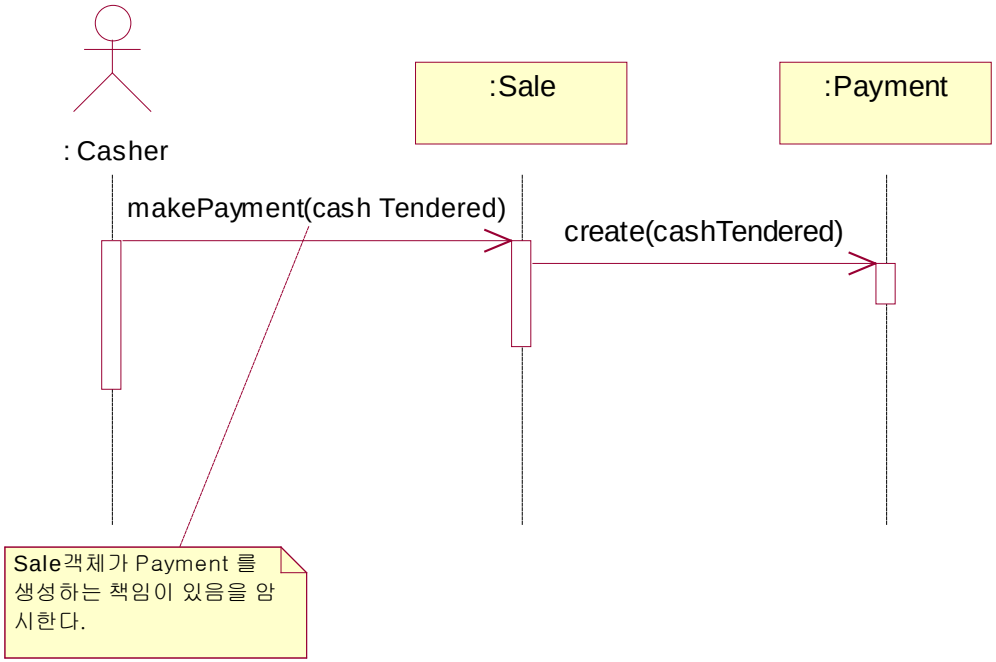
Metaphor = 은유(隱喩), 만물이 미소 지었다.

4. GRASP: 기본적인 객체지향 설계를 위한 조직적 접근

- GRASP 패턴을 적용하는 방법을 이해하는 것이 목표
- 일단 이해한 후에 세부적인 것은 잊어도 된다.

5. 책임과 교류도, GRASP

- 책임을 객체에 할당하는 기본원리
  - 개발하는 도중에도 항상 고려해야 하는 사항
    - 어떤 객체가 어떤 책임을 지도록 할 것인가. (GRASP 패턴)
    - ➔ 책임할당패턴을 적용



6. 책임할당 “패턴”

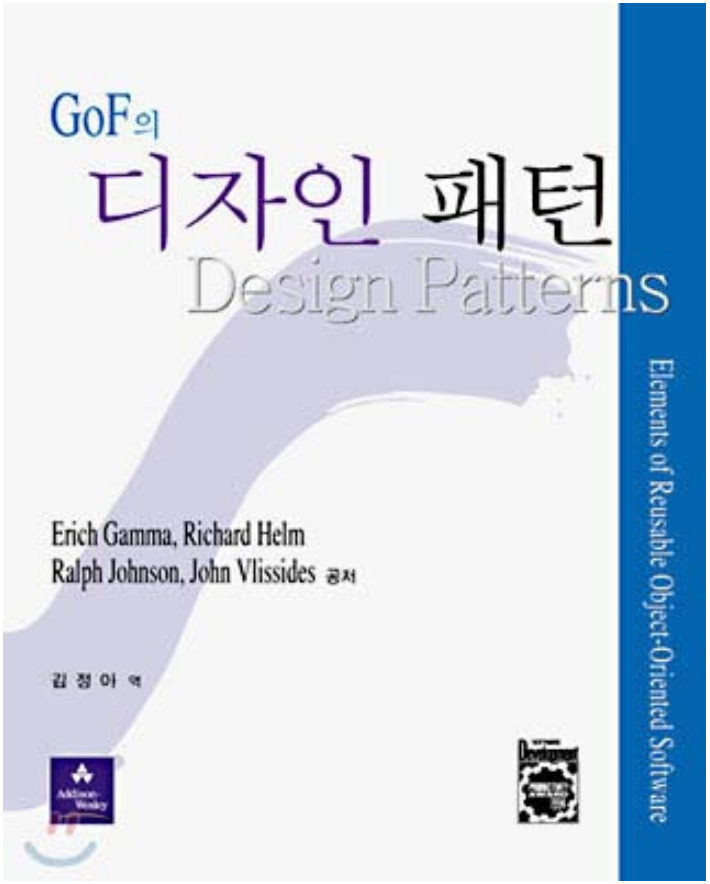
- 패턴 기술 형식 (예)
  - 패턴명: Information Expert
  - 문제: 책임을 객체에 할당하는 기본 원리는 무엇인가.
  - 해결책: 일을 수행하는 데 필요한 정보를 갖고 있는 클래스에게 책임을 할당.
- 패턴의 활용
  - 어떤 특정 범주의 문제가 주어졌을 때, 여러 개의 패턴들이 어떻게 책임을 객체에 할당해야 하는가에 관한 가이드라인 제공

패턴은 새로운 문맥에서 적용될 수 있는 이름이 붙여진 한 쌍의 문제/해결책이다. 또한 패턴 적용방법에 대한 조언과 다각도의 영향에 대한 설명을 제공.

패턴의 “반복성”과 “명칭”

- 반복성
  - 패턴이란 용어 자체가 반복적인 일의 형태를 암시.
  - 새로운 아이디어보다는 기존에 널리 적용되고 있는 기본 원리를 요약, 정리한 것.
  - 따라서, IT전문가에게는 패턴이 새로운 것이 아니라 이미 익숙한 것으로 느낌.
- 패턴이름
  - 문제/해결책의 이름을 구체화하여 쉽게 이해하고 기억하게 도움.
  - 의사소통을 원활하게 함.
- “새로운 패턴” 이란 말은 모순이다.
  - 패턴은 오랫동안 반복되어 온 것을 기술하는 말이다.
  - 따라서 “새로운 패턴”보다는 “신규로 체계화된 패턴”이란 말이 적절

GoF의 디자인 패턴



- 소프트웨어 패턴 용어의 최초 사용
  - 캔트 백, 1980 (XP의 창시자)
- 디자인 패턴, GoF, 1994
  - Adapter 패턴 등 23개 패턴 기술
  - 네 사람에 의해 쓰여짐 (gang of 4)
  - 입문서가 아니므로 UML 및 프로그래밍 언어에 조예가 있어야 함.

Applying UML and Patterns

- GRASP (General Responsibility Assignment Software Patterns or Principles)
  - 원칙인가 패턴인가
    - GRASP는 원칙에 가깝지 않는가?
    - GRASP을 패턴 기술 형식에 맞추어 기술하였음.
  - “Design Pattern” 의 말
    - “한 사람의 패턴은 다른 사람의 기본적인 구성요소가 될 수 있다.”

- 본 책의 목표
  - GRASP 책임할당 패턴 및 GoF 설계 패턴을 모두 익혀 적용할 수 있는 능력 배양!

7. 어디쯤 왔나?

- 1. 반복적 프로세스의 배경 기술
- 2. RDD: 책임을 가지고 서로 협력하는 객체
- 3. 객체지향 설계 아이디어 제공: GRASP, GoF.
- 4. 시각적 모델링 도구: UML

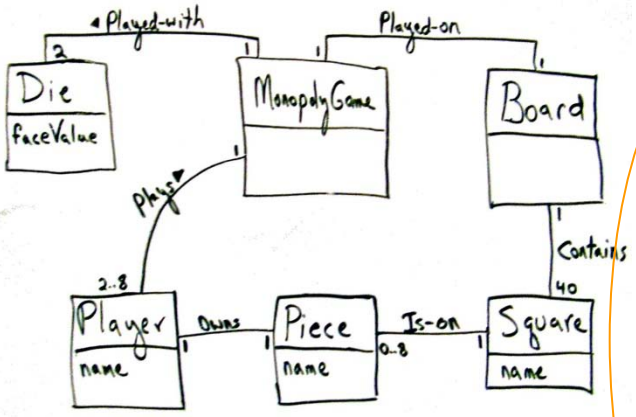
8. GRASP: 일반적인 책임할당 패턴

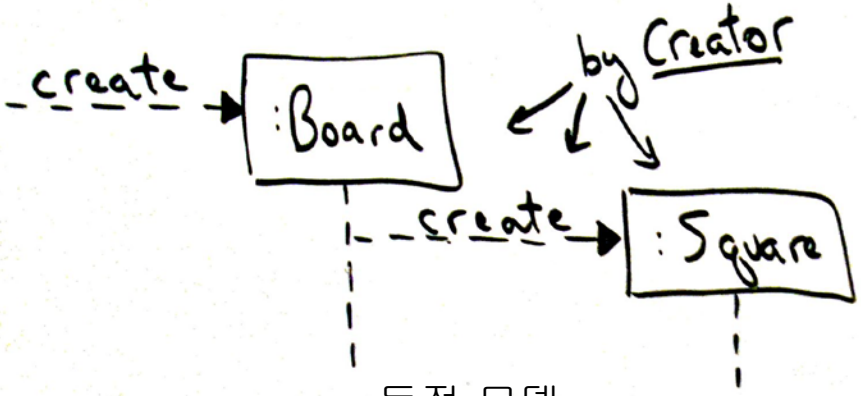
- GRASP 패턴
  - GRASP 패턴은 “객체 설계와 책임 할당에 관한 기본적인 원리”를, 패턴 형식에 맞추어 기술한 것.
- 5-GRASP 패턴
  - 전문가패턴 Information Expert
  - 생성자패턴 Creator
  - 고응집성패턴 High Cohesion
  - 저결합성패턴 Low Coupling
  - 제어기패턴 Controller
- 추가 GRASP 패턴
  - 다형성패턴 Polymorphism
  - 순수가공패턴 pure fabrication
  - 간접패턴 indirection
  - 보호변형패턴 protected variations

GRASP 패턴의 간단한 적용 예 1 – 생성자 패턴

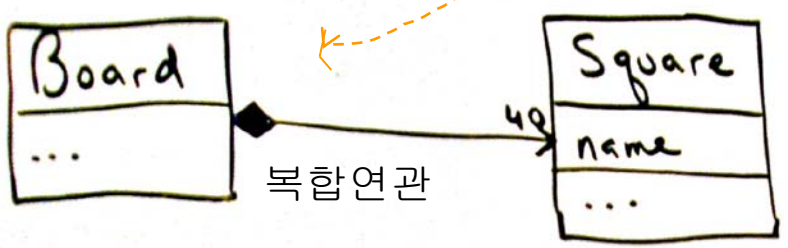
- 문제 : 누가 객체 Square를 생성하는가?

패턴명	Creator
문제	누가 객체 A를 생성하는가?
해결책	다음의 경우에 클래스 B에게 객체 A를 생성하는 책임을 할당한다. (많을 수록 좋다) -B가 A를 포함하거나, 구성적으로 합친다. -B가 A를 기록한다. -B가 A를 밀접하게 사용한다. -B가 A를 초기화시키는 데 필요한 데이터를 갖고 있다.





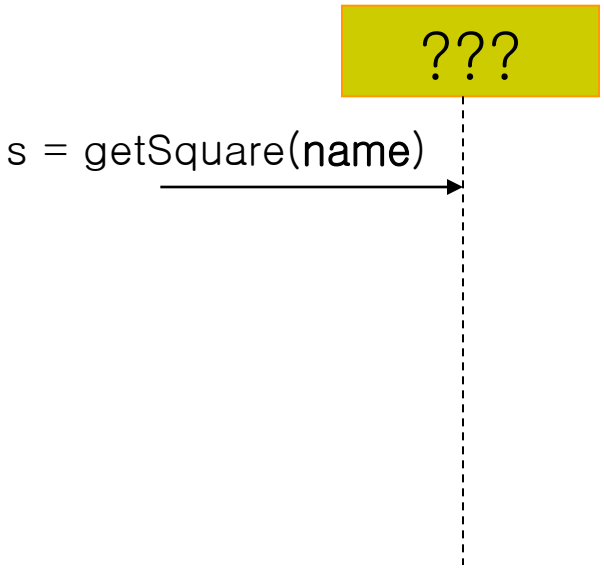
동적 모델

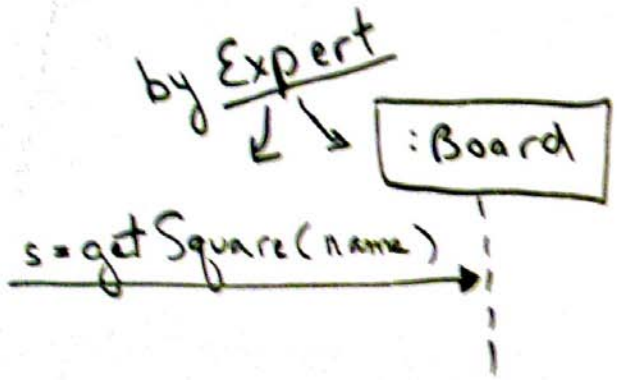


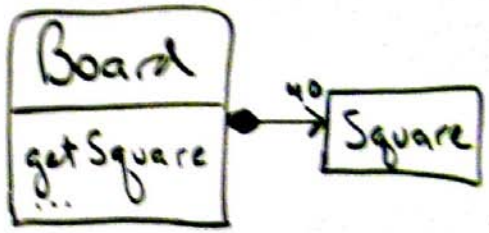
복합연관

GRASP 패턴의 간단한 적용 예 2 – 정보전문가

- getSquare(name) 메소드를 누가 지원하나?
- 문제: 키 값이 주어졌을 때 누가 Square 객체에 관해 알고 있는가?
- 해결책: 책임을 수행하는 데 필요한 정보를 갖고 있는 클래스에게 그 책임을 할당하라.

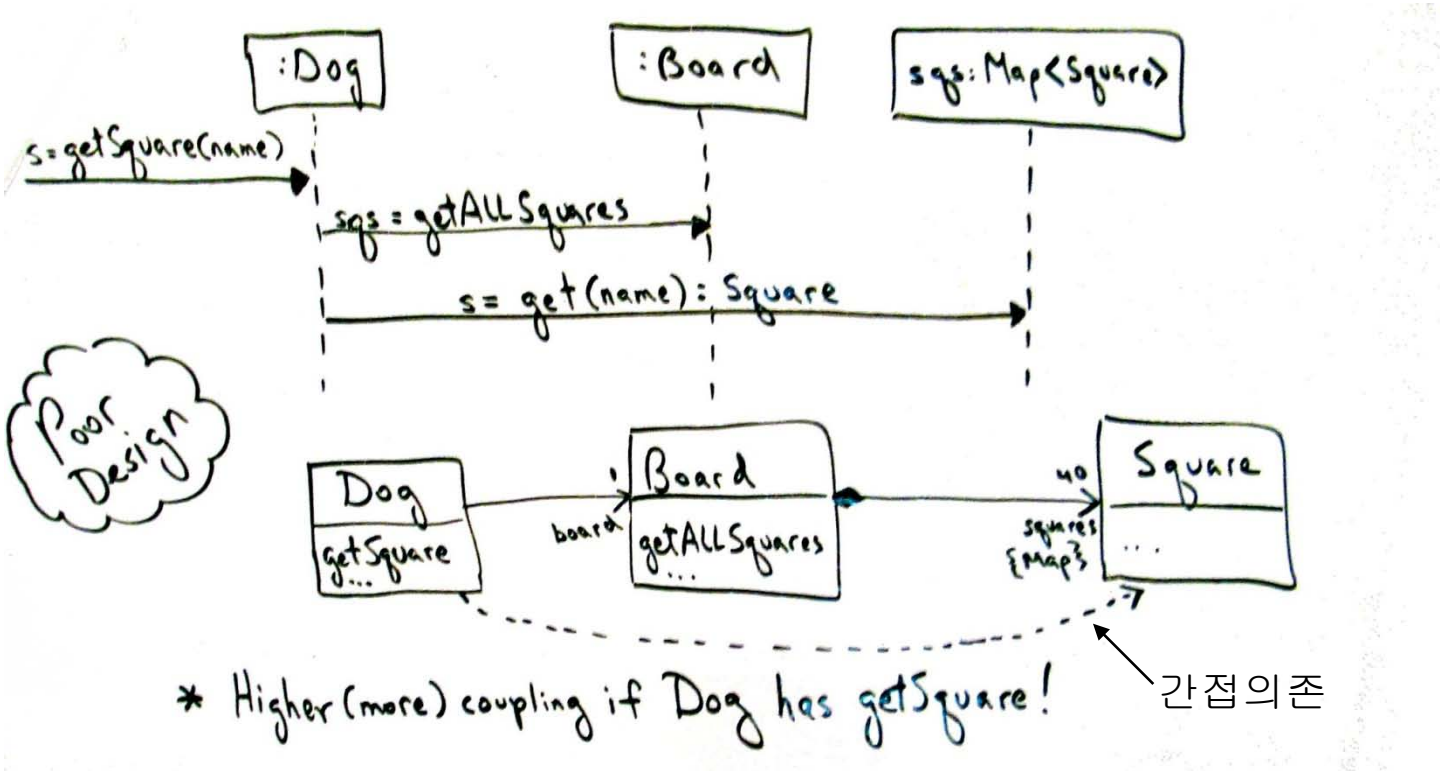






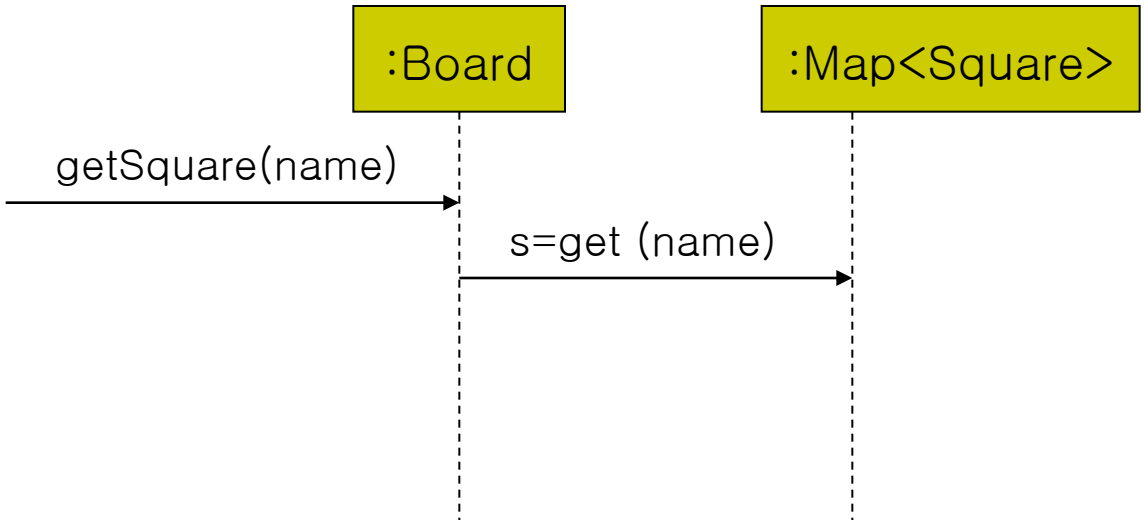
GRASP 패턴의 간단한 적용 예 3 – 저결합 패턴

- 질문: 왜 Dog 보다는 Board 인가?
- 문제: 변화의 영향을 어떻게 줄일 것인가?
- 해결책: (불필요한)결합도가 낮아지도록 책임을 할당하라. 대안을 평가하기 위한 원리로 이 원칙을 실행하라.



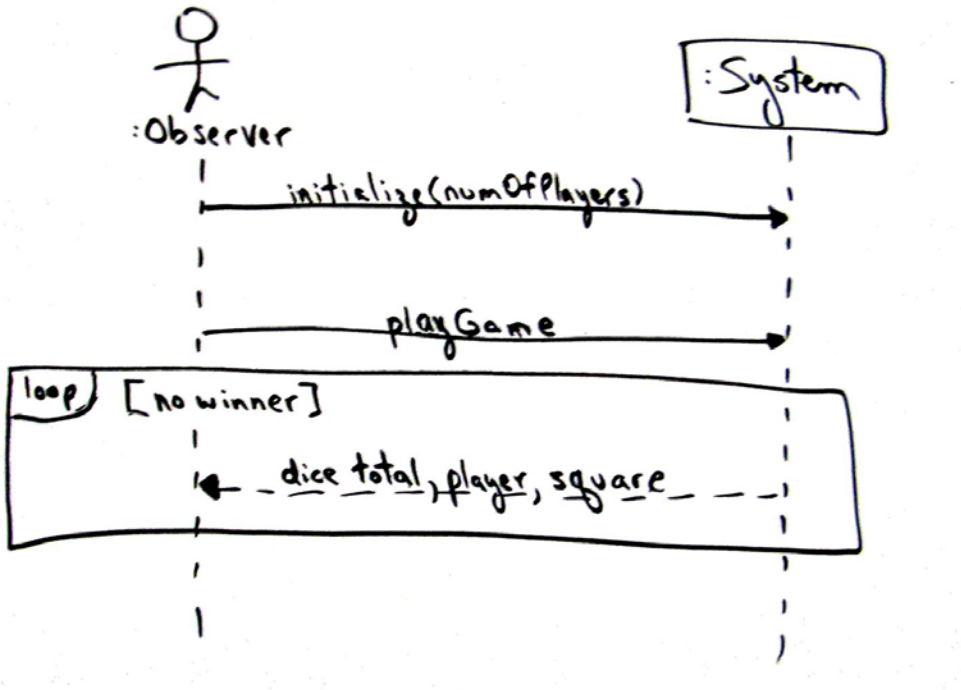
GRASP 패턴의 간단한 적용 예 3 – 저결합 패턴 2

- 핵심사항: Expert 는 Low Coupling 을 지원한다.

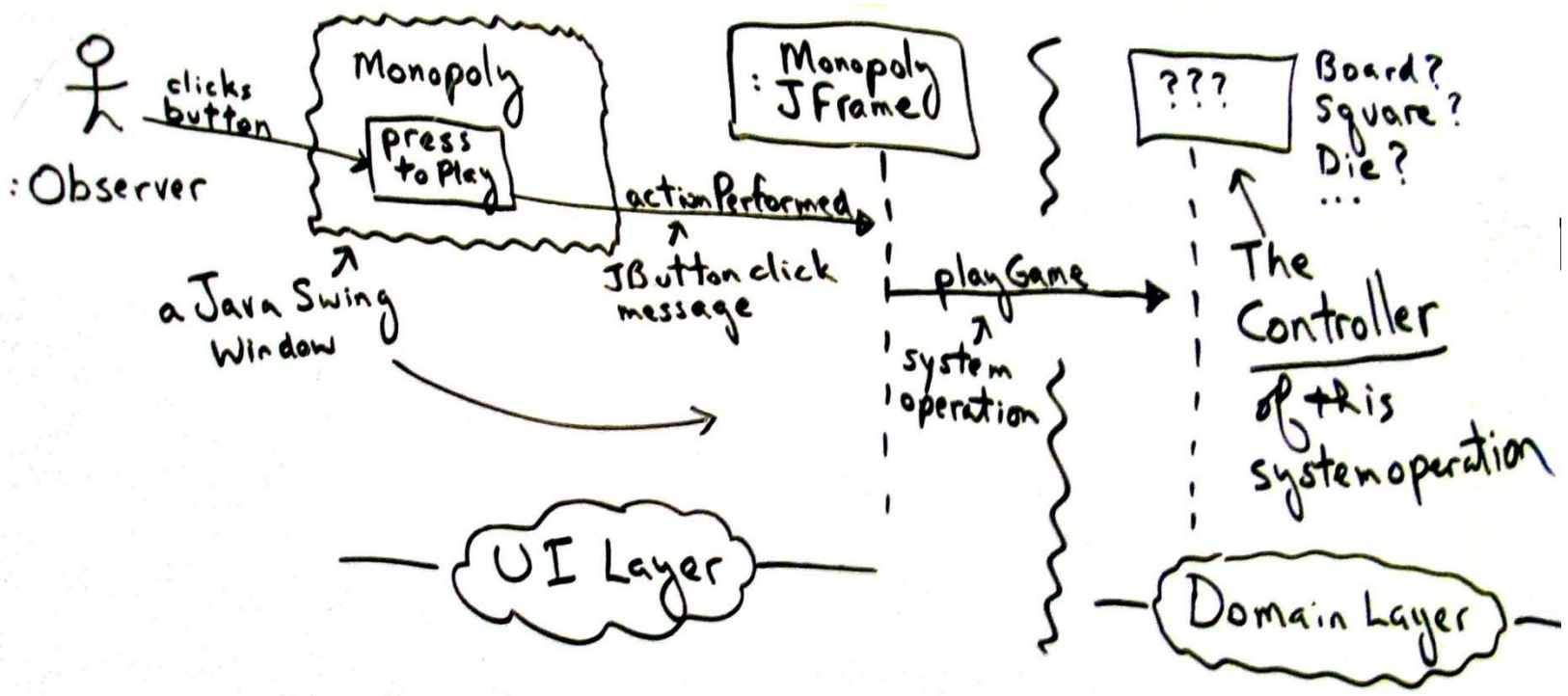


GRASP 패턴의 간단한 적용 예 4 – 제어기 패턴

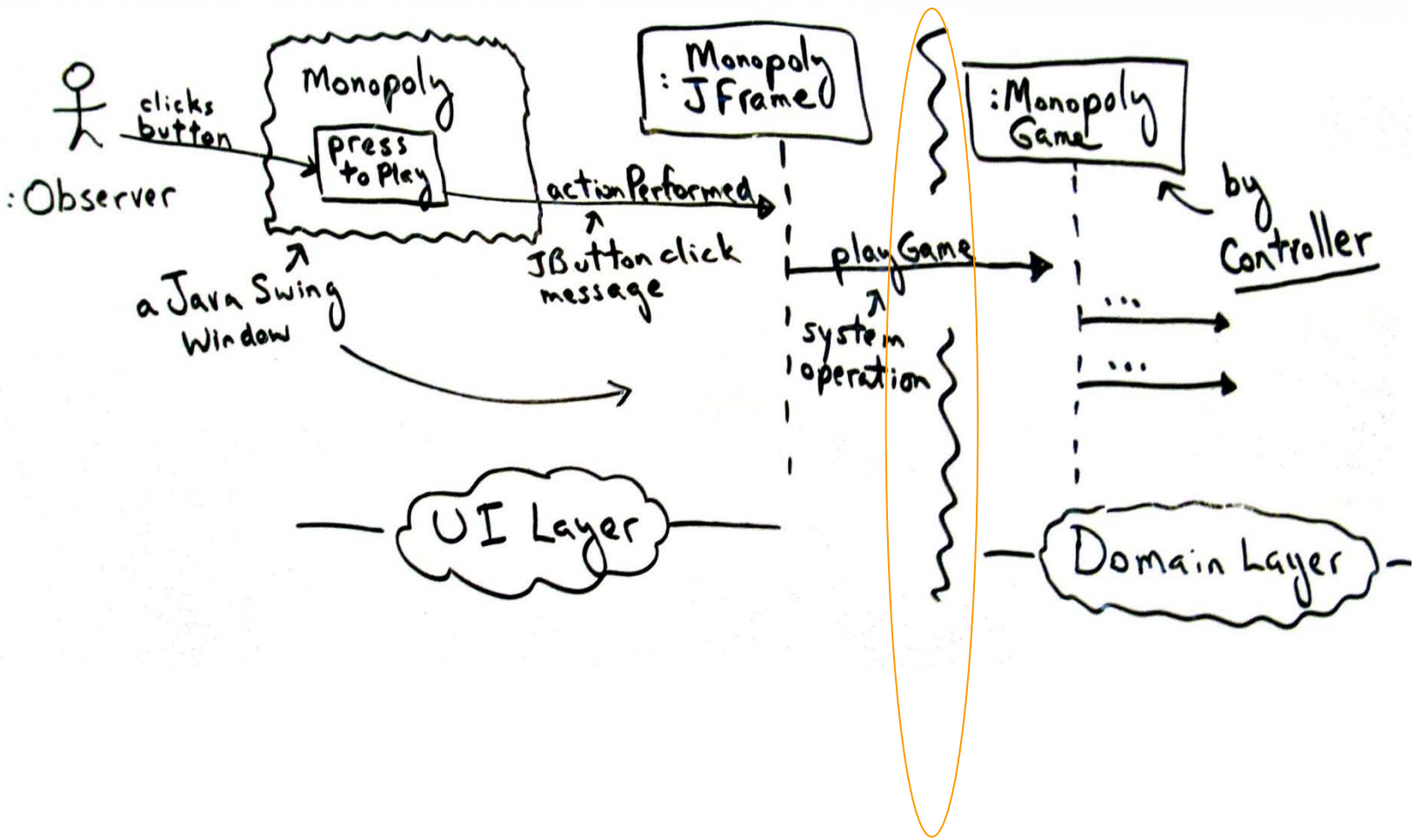
- MVC 모델 : 모델-뷰 분리 원칙
  - UI 객체들은 데이터를 다루지도 않고 상호작용의 순서도 다루지 않는다. 다만 주어진 데이터를 어떻게 표현하고, 어떻게 입력을 받는가에 책임을 질 뿐이다.
- 문제: UI 계층보다 상위의 객체 중 어떤 객체가 최초로 시스템 연산을 받아 제어하는가?
- 해결책: "시스템", "서브시스템: 객체 또는 UC 등의 시나리오를 진행하는 객체가 제어.



GRASP 패턴의 간단한 적용 예 4 – 제어기 패턴 2

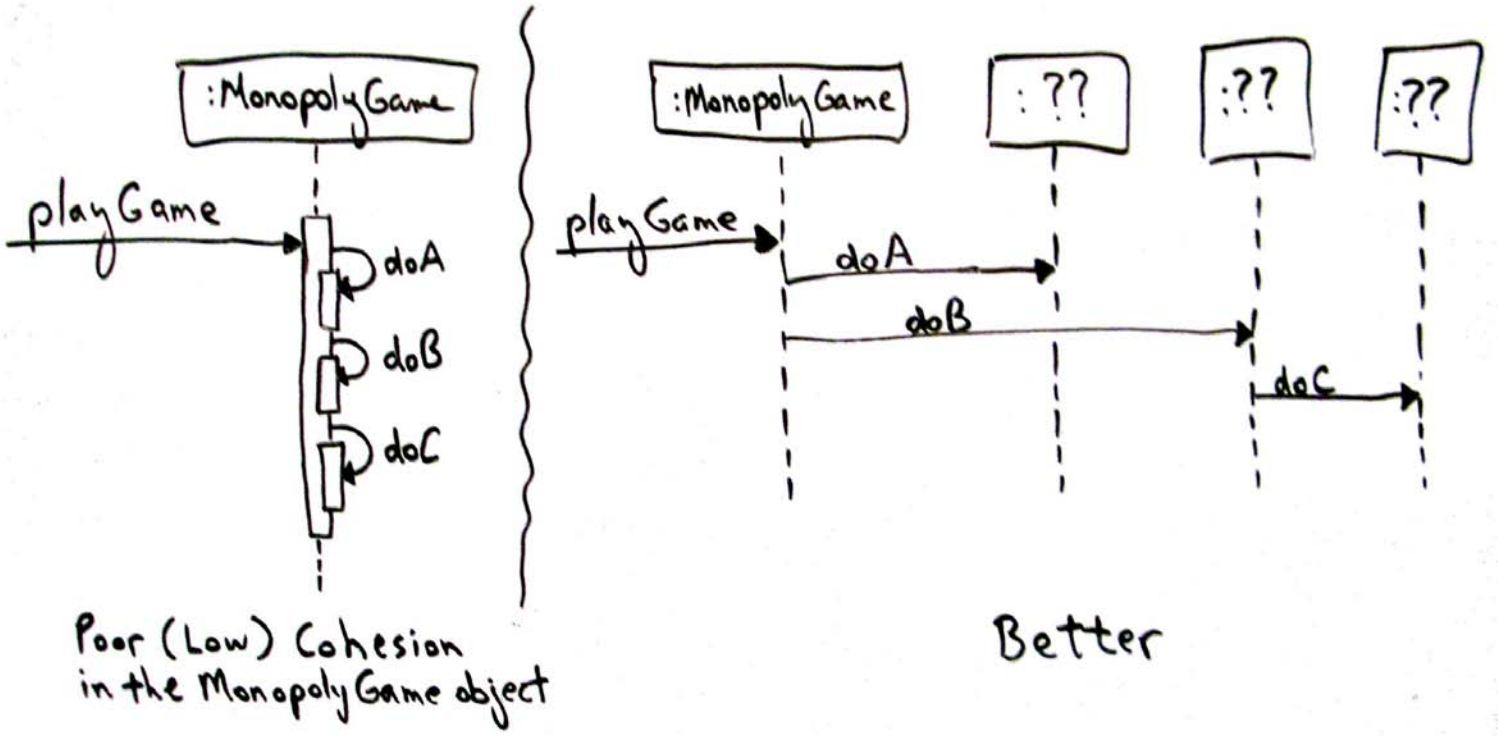


GRASP 패턴의 간단한 적용 예 4 – 제어기 패턴 3



GRASP 패턴의 간단한 적용 예 5 – 고응집 패턴

- 문제: 객체를 이해하기 쉽고 유지 보수하게 쉽게 할 것이며, 그것의 부작용인 Low Coupling을 어떻게 해결할 것인가?
- 해결책: 응집도가 높도록 책임을 객체에 할당하라.



9. 객체 설계에 GRASP 적용하기.

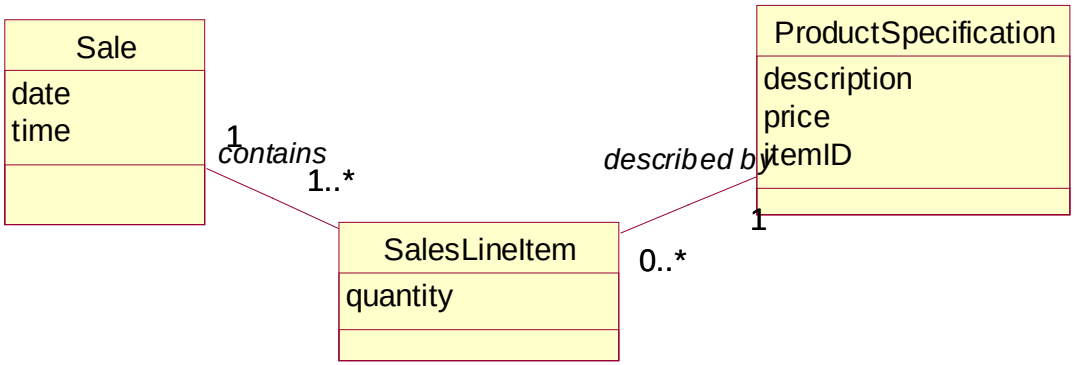
- 객체지향 설계하기 위해 우선 이 원칙을 ‘grasping’ 하는 것이 중요하다.
- 9가지 패턴
- 5-GRASP 패턴
  - 전문가패턴 Information Expert
  - 생성자패턴 Creator
  - 고응집성패턴 High Cohesion
  - 저결합성패턴 Low Coupling
  - 제어기패턴 Controller
- 추가 GRASP 패턴
  - 다형성패턴 Polymorphism
  - 순수가공패턴 pure fabrication
  - 간접패턴 indirection
  - 보호변형패턴 protected variations

10. 정보 패턴

- 패턴명: 정보 (혹은 전문가) 패턴
- 해결책
  - *책임을 수행하는 데 필요한 정보를 가진 클래스에게 책임을 지우라.*
- 문제
  - 책임을 객체에 할당하는 일반적인 원리는?

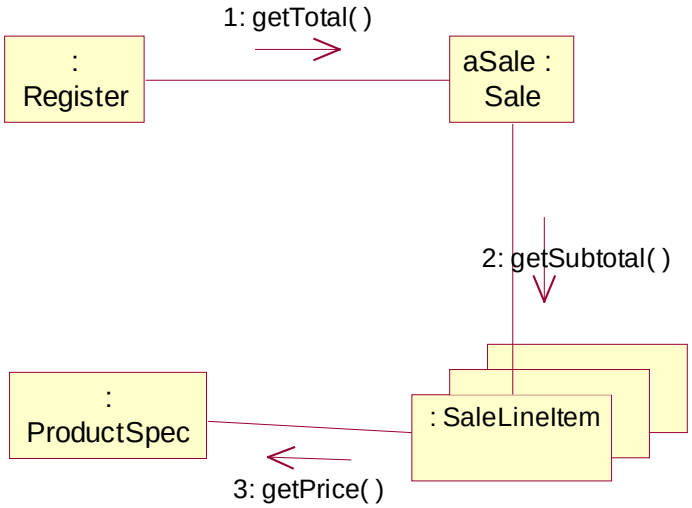
정보 패턴: 예

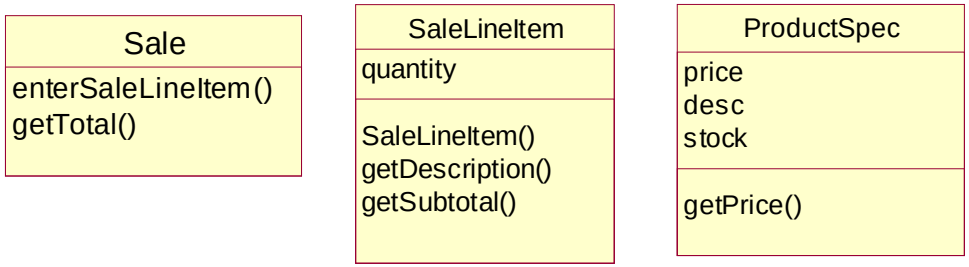
- <예제>
  - 문제: NextGen POS에서 판매총액을 알 책임은 누가 져야 하는가?
  - 해답: 도메인 또는 설계 모델을 고려
    - (1) 기존의 설계 모델에 적합한 클래스가 있는지 먼저 고려하고
    - (2) 그 후 도메인 모델에서 적절한 클래스를 발견하여 설계 객체로 전환시킨다.
  - Sale은 총액을 내기 위한 SalesLineItem 들과 ProductSpecification을 간접적으로 알고 있으므로 판매총액을 산출하는 책임을 질 만 하다.



NextPOS: 판매총액 계산 책임 할당

- Sale 객체가 판매총액 책임
  - 정보 패턴 적용





설계 클래스	책임
Sale	판매 총액을 안다
SalesLineItem	라인 아이템의 소계를 안다
ProductSpecification	제품의 가격을 안다

정보 패턴의 고찰

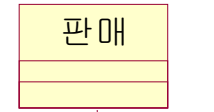
- 책임 수행을 위한 협력
  - Sale 객체가 책임을 지지만 실제는 3개의 객체의 협력 필요
  - 객체는 생명이 있다.
- 금기사항
  - 결합도와 응집도의 문제 때문에 바람직하지 않을 수 있다.
  - 예) Sale 을 데이터베이스에 저장하는 책임.
    - 정보패턴에 의하면 Sale자신이 DB에 접근해야 함
    - 하지만, Sale 본래의 책임에 집중하기 어려움.
    - 따라서, 전문 DB 클래스에 할당하는 것이 바람직.
- 이점
  - 캡슐화, 낮은 결합도, 높은 응집도 가능, 유지보수 용이
- 관련패턴
  - 저결합 패턴 Low Coupling
  - 고응집 패턴 High Cohesion

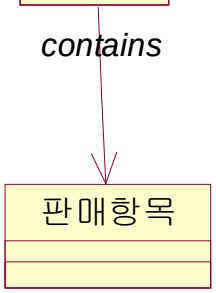
11. 생성 패턴 (Creator Pattern)

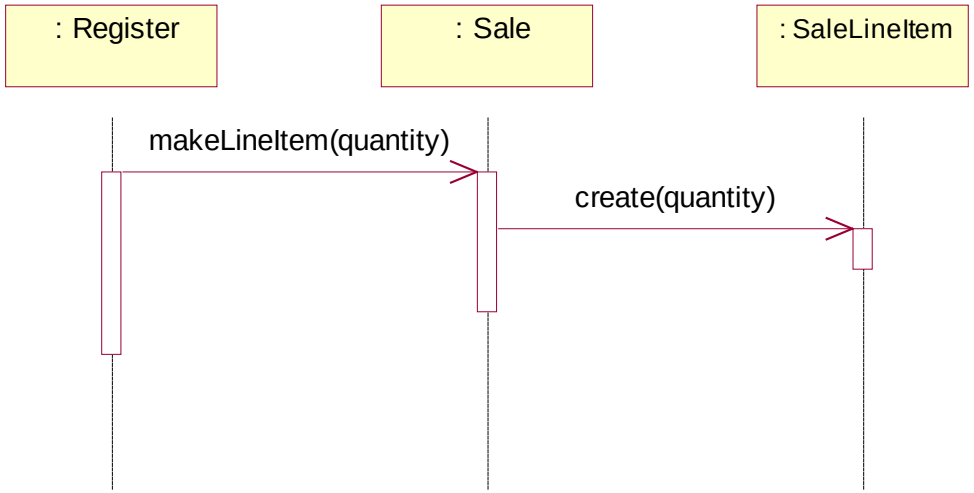
- 패턴명칭: 생성
- 해결책:
  - *다음 중 하나에 해당된다면 클래스 A의 객체를 생성하는 책임을 클래스 B에 할당하라.*
    - B가 A 객체들로 구성된다.
    - B가 A객체들을 포함한다.
    - B가 A객체를 가깝게 사용한다.
    - B가 A객체를 기록한다.
    - A가 생성될 때 A에게 전달되는 초기화 데이터를 B가 가지고 있다. (그러므로 B는 A에 생성에 관한 정보를 가지고 있다.)
- 문제:
  - 누가 클래스의 새로운 객체를 생성할 책임을 지는가.

생성패턴 예

- NextGen POS
  - 문제: POS에서 누가 SaleLineItem을 생성해야 하는가.
  - 해답: 도메인 모델을 고려하면 생성패턴에 의해 SaleLineItem 객체들로 구성되거나 이것들을 포함하는 클래스는 Sale이다. 따라서, “Sale” 이 SalesLineItem 객체를 생성할 책임을 갖는 후보로 적합하다.







생성패턴 논의

- 핵심
  - 생성된 객체와 연결될 필요가 있는 객체 찾기
  - 집합 (aggregation), 컨테이너(container), 기록(recorder)
- 금기사항
  - 성능상의 이유로, Factory라는 도우미 클래스에 위임 가능 경우.
- 이점
  - 낮은 결합도가 지원됨.
  - 생성 객체에 대한 가시성을 생성하는 객체가 이미 가지므로
- 관련패턴
  - 저결합
  - 팩토리
  - 전체-부분

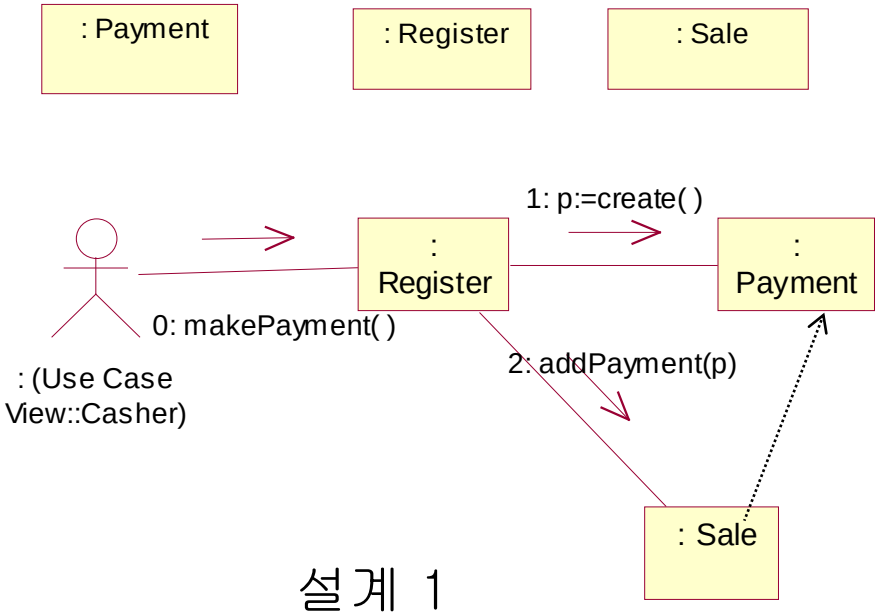
12. 저결합 패턴

- 패턴명: 저결합 (low coupling)
- 해결책:
  - *결합도가 낮게 유지되도록 책임을 할당한다.*
- 문　제:
  - 어떻게 의존성을 적게 하고, 변화의 영향을 적게 하면 재사용성을 증가시킬 것인가.
- 결합도?
  - *한 요소가 다른 요소와 얼마나 연관 되었는가에 대한 척도.*
    - 관련된 다른 클래스들이 수정되면 그 클래스의 내부도 수정해야 한다.
    - 그 클래스만 가지고 이해하기 어렵다.
    - 의존하는 클래스들이 있으므로 그 클래스 자체만으로는 재사용이 어렵다.

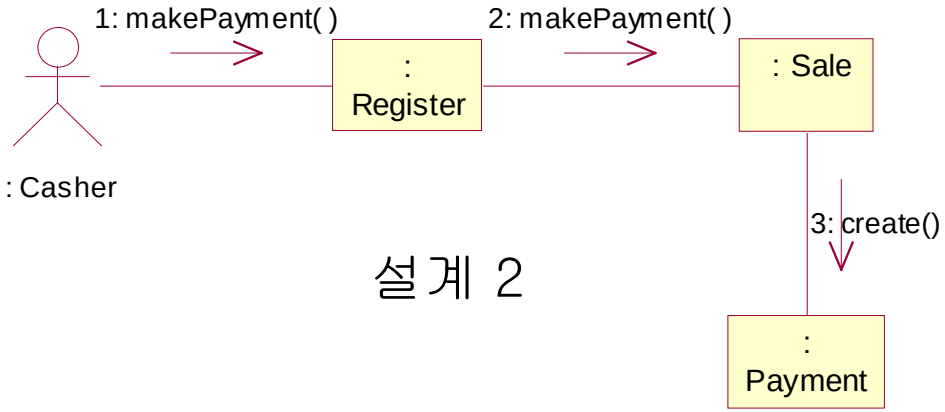
저결합 패턴의 예

- POS 예제
  - 문제: Payment 객체와 Sale객체를 연관시킬 필요가 있다. 어떤 클래스가 이에 대한 책임이 있는가?
  - 해답:
    - 실세계에서 Register 객체가 Payment를 기록하기 때문에 생성패턴에 의해 Register 객체는 Payment 객체의 생성자가 되는 것이 바람직하다.
    - (설계1) Payment 객체를 생성한 후 Sale의 addPayment(pay) 메소드를 통해 Payment 와 Sale을 결합시킨다. (결합도 1 증가)
    - (설계2) 그러나, Sale이 Payment 객체를 생성하는 경우 결합도가 증가하지 않는다. (저결합 패턴) 따라서 결합도 측면에서는 설계 2가 더 낫다.

저결합 패턴 예-그림



설계 1



설계 2

저결합 패턴 논의

- 결합도
  - C++/Java/C# – Type X에서 Type Y로의 결합형태
    - X는 Y의 객체 혹은 Y자체를 참조하는 “속성” 보유
    - X의 객체는 Y 객체의 서비스를 “호출”
    - X는 Y의 객체 또는 Y자체를 참조하는 “메소드” 보유. 이 경우 전형적으로 메소드가 Y타입의 매개변수나 지역변수를 포함하거나, 메시지로부터 반환되는 객체가 Y타입이다.
    - X는 Y의 직접적 또는 간접적 서브클래스이다.
    - Y는 인터페이스고 X는 Y를 구현한다.
  - 저결합은 책임할당으로 결합도가 증가하지 않도록 한다.
  - 극단적인 저결합도는 클래스간 결합이 제로인 경우인데, 이는 바람직하지 않다.
  - 저결합 패턴은 결합을 최소로 줄이자는 패턴임.
- 금기사항
  - 안정되었거나 널리 사용되는 요소의 높은 결합도는 문제되지않음.

저결합 패턴 논의(2)

- 이점
  - 다른 컴포넌트의 변화에 의해 영향 받지 않음.
  - 독립적으로 이해하기 쉽다.
  - 재사용이 가능하다.
- 배경
  - 결합도와 응집도는 설계에 있어 기본적인 원리이므로 모든 소프트웨어 개발자가 인지하고 적용해야 함.
- 관련패턴
  - Protected Variations

13. 고응집 패턴

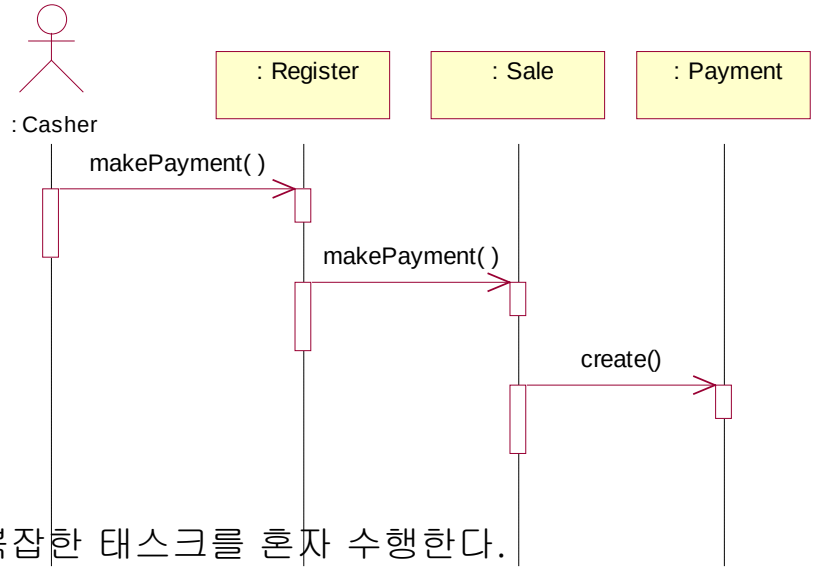
- 패턴명: 고응집(high cohesion)
- 해결책:
  - 응집도가 높게 유지되도록 책임을 할당하라.
- 문제점:
  - 어떻게 복잡성을 관리할 수 있는 수준으로 유지할 것인가.
- 응집도 (기능적 응집도)
  - 한 요소의 책임들이 서로 관련되고 집중되어 있는가에 대한 척도
  - 낮은 응집도: 관련 없는 책임을 많이 가지고 다양한 일을 하는 클래스.
    - 이해하기 어렵다.
    - 재사용이 어렵다.
    - 유지보수가 어렵다.
    - 민감하며 변화의 영향을 지속적으로 받는다.

고응집 패턴 예

- POS 예제
  - 문제: Payment 객체와 Sale객체를 연관시킬 필요가 있다. 어떤 클래스가 이에 대한 책임이 있는가?
  - 해답:
    - 실세계에서 Register 객체가 Payment를 기록하기 때문에 생성패턴에 의하면 Register 객체는 Payment 객체의 생성자가 되는 것이 바람직하다.
    - 그렇게 하면 Register 객체는 새로운 Payment객체를 매개변수로 addPayment를 통해 Sale로 보낼 수 있다. 즉, Register는 makePayment 시스템 연산을 수행할 책임을 "부분적으로" 진다.
    - 그러나 Register 클래스가 다른 더 많은 책임을 부여한다면, 이 클래스는 더 많은 일을 떠 안게 되고 따라서 응집도가 떨어진다.
    - 따라서 Payement 생성에 관한 책임과 연관에 관한 책임을 Sale에게 지운다면 Register는 높은 응집도를 유지할 것이다. (두 번째 설계가 바람직)

고응집 패턴 논의

- 고응집도
  - 모든 설계 시 고려사항
  - 평가원리
  - 응집도의 수준
    - 최저응집도
      - 한 클래스가 다양한 기능 영역에서 많은 책임이 있다.
    - 저응집도
      - 한클래스가 하나의 기능영역에서 복잡한 태스크를 혼자 수행한다.
    - 보통응집도
      - 한 클래스가 그 클래스 개념과 논리적으로 관련되지만 서로는 관련되지 않은 몇몇 다른 영역들에서 간단하고 유일한 책임을 진다.
    - 고응집도
      - 한클래스가 하나의 기능영역에서 적당한 책임을 수행하며, 태스크를 수행하기 위해 다른 클래스와 협력한다.



고응집성: 모듈화된 설계

- 모듈화된 설계란
  - 모듈성(modularity)는 응집도가 있고 결합도가 낮은 모듈들로 분해된 시스템이 갖는 특성이다.
- 응집도와 결합도:음과 양
  - 나쁜 응집도 → 나쁜 결합도 초래하며 그 역도 마찬가지.
  - 응집도와 결합도는 상호 의존적.
- 금기사항
  - 낮은 응집도가 정당화되는 몇몇 경우가 있음.
    - 유지보수를 한 사람에게 몰아주기 위해
    - 분산서버에서 성능상 하나의 특정 시스템으로 처리할 때.
- 이점
  - 설계의 명료성과 이해의 용이성 향상
  - 유지보수와 개선이 단순.
  - 밀접한 기능을 세분화함으로 재사용성이 증가.

14.제어기 패턴

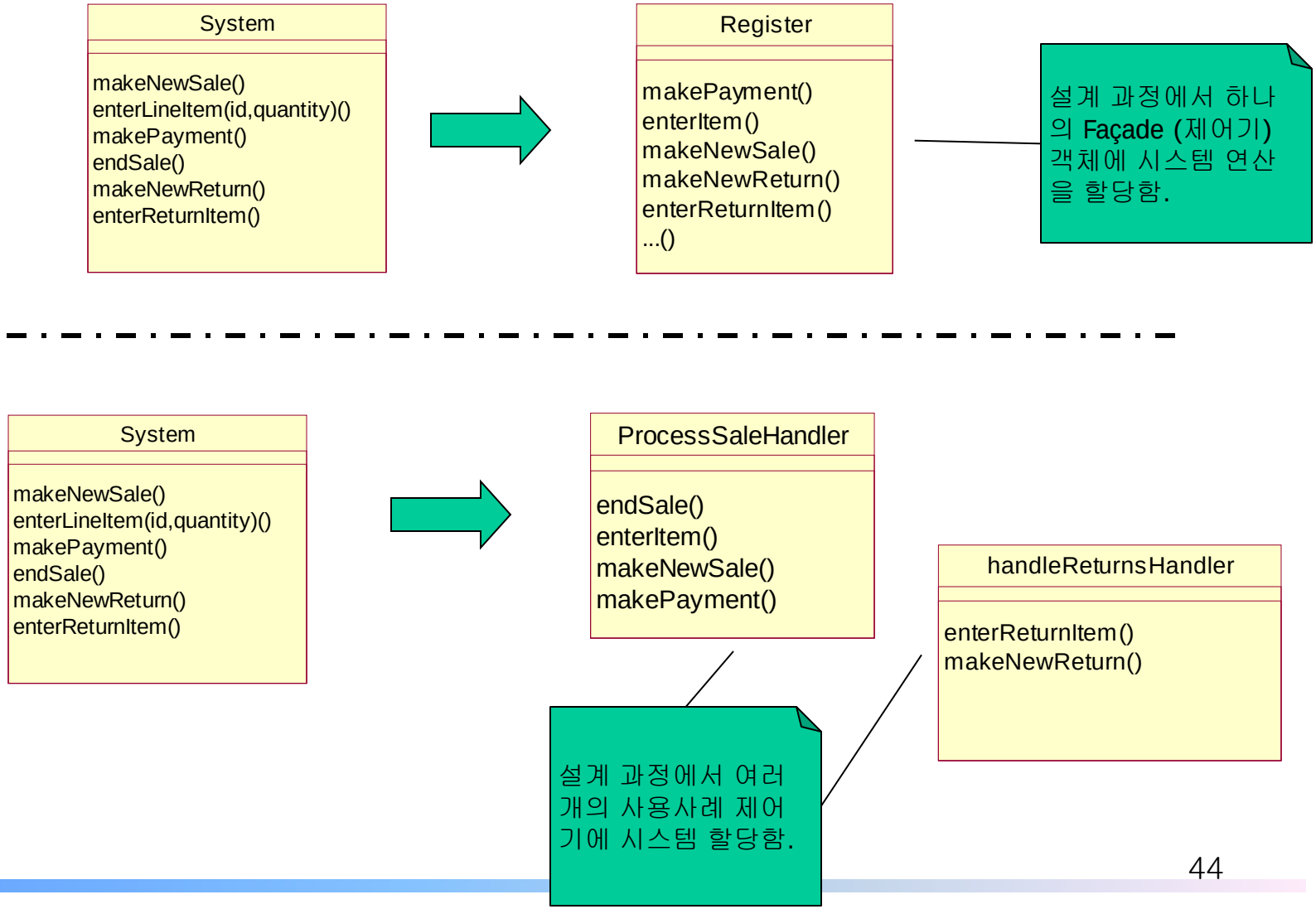
- 패턴명: 제어기(controller)
- 해결책:
  - 시스템 이벤트 메시지를 받거나 다루는 책임을 다음 클래스로 할당.
    - 전체 시스템, 장치, 혹은 서브시스템을 나타내는 클래스 (façade 컨트롤러)
    - 시스템 이벤트가 시나리오를 나타내는 클래스. <사용사례이름>Handler, <사용사례이름>Coordinator, 혹은 <UC>Session
      - 같은 사용사례 시나리오 안의 모든 시스템 사건에 대해 같은 제어 클래스를 사용할 것.
      - 비공식적으로 하나의 세션은 액터와의 대화에 의한 인스턴스임.
- 문제:
  - 누가 입력 시스템 이벤트를 처리할 책임이 있는가.

제어기 패턴의 예

- POS 예제
  - 문제:판매에서 액터로부터의 입력을 어떤 클래스가 처리할 것인가.
  - 해답:
    - 전문적인 제어기 클래스를 사용. 시나리오 핸들러나 세션 클래스를 정의하여 모든 사건의 처리 프로세스를 위임. UI클래스가 사건처리를 직접 해서는 안되고 Handler에게 위임하라.
    - 하나의 핸들러에서 처리할 것이 많으면 관련 있는 서브프로세스 별로 핸들러를 쪼개어 정의하라.

분석과정에서 시스템객체(:system)가 시스템 연산을 수행할 책임을 지지만, 설계 과정에서는 제어기 클래스들이 시스템 연산에 대한 책임을 분할 및 할당을 받는다.

제어기 패턴: 시스템/제어기 객체

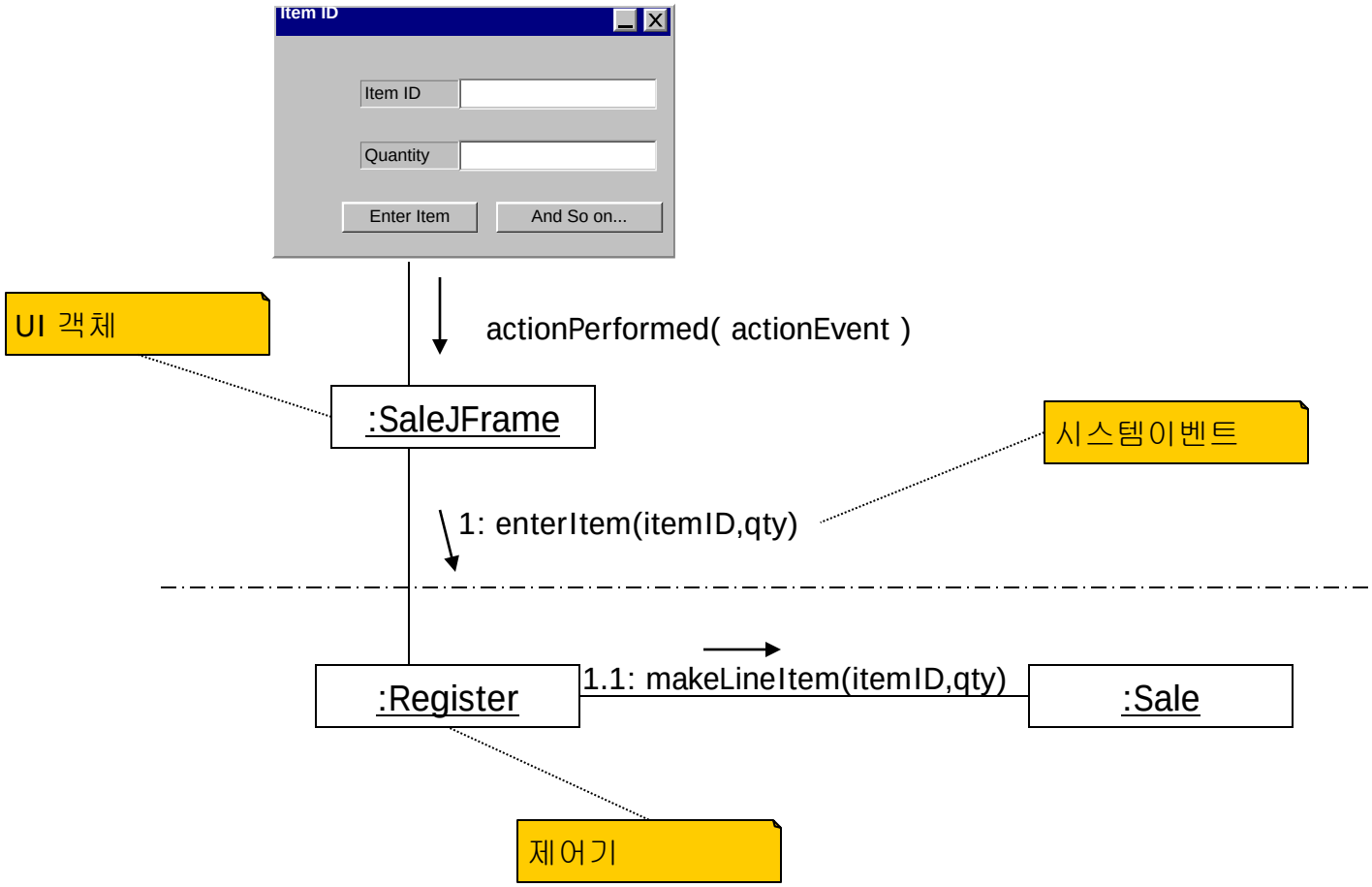


제어기 패턴 논의

- 제어기 패턴의 핵심
  - UI 계층의 클래스로부터 서비스 요청을 받고, 일반적으로 이를 다른 객체들에게 위임함으로써 임무를 수행
  - EJB : 서버 세션 빈
  - JSP: 서버 서블렛
  - Swing: 클라이언트측 객체
- UP
  - boundary객체: 액터-시스템 인터페이스
  - Entity객체: 응용프로그램에 독립적인 도메인 소프트웨어 객체
  - Control객체: boundary객체로부터의 이벤트를 여러 Entity 객체들에게 위임하여 사건처리 과정을 담당함. (제어기 패턴)
- 논점 및 리뷰
  - 팽창된 제어기 조심!
  - 인터페이스 객체는 시스템 사건을 처리하지 않음.

사용사례제어기
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

제어기 패턴: 바람직한 결합



제어기 패턴 - 논의

- 메시지 처리 시스템과 명령어 패턴
  - 서버 또는 메시지 처리 시스템
    - 교환기
    - 인터페이스나 제어기 설계는 달라짐
    - Command 패턴 또는 Command Processor 패턴
  - 관련된 패턴
    - Command – 각 메시지를 전담 Command객체에서 처리
    - Façade – 제어기 패턴은 일종의 Façade 이다.
    - Layers – Presentation / Domain Layer
    - Pure Fabrication – 도메인 모델과는 상관 없이 설계자가 제작한 소프트웨어 클래스. 사용사례 제어기는 일종의 Pure Fabrication 패턴임.

객체 설계와 CRC 카드

- Kent Back and Ward Cunningham
  - 객체 설계자가 책임할당과 상호 협력의 관점에서 더 추상적으로 생각하도록 장려하는 것, 패턴의 사용을 장려
  - CRC 카드는 인덱스 카드로, 클래스 이름, 책임, 협력 클래스 목록을 담고 있음.
  - 소규모 그룹 CRC 카드 작업
  - 책임할당과 협력의 결과를 기록하기 위한 카드.
- 참고 자료
  - 책임주도 설계
    - Tecktronix : Kent Beck 등.
    - Designning Object-oriented Software [www90]