Distributed Vending Machine

OOPT Phase 2050. Construct
OOI, 최종 발표

TEAM 6

김성환, 201911243

김지환, 201911560

이송헌, 201911200

조찬형, 201811223

Table of Contents

1. Refine OOPT 2040: OOD – SD 변경

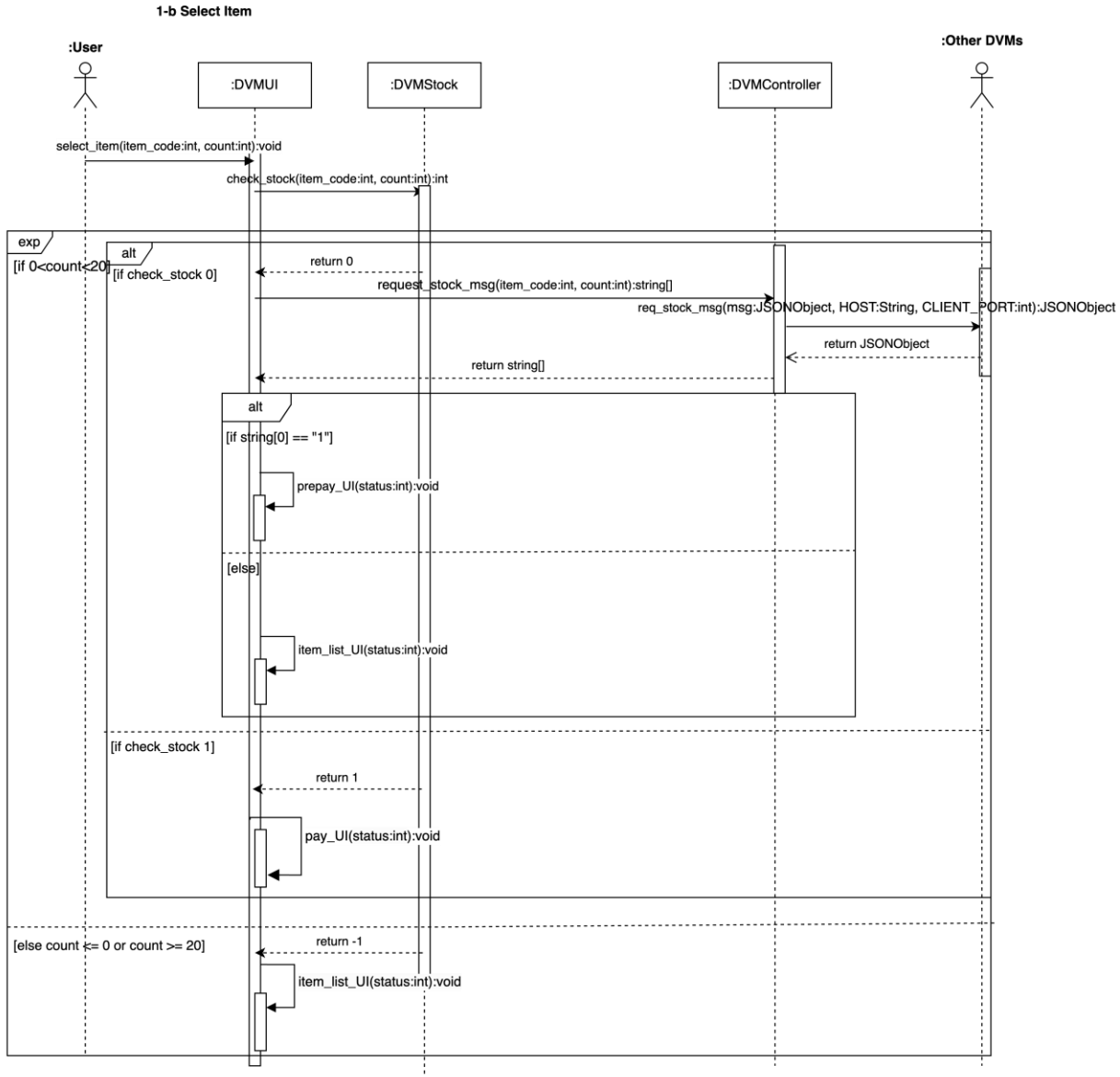
- 예외 처리. 특히 UI, 통신, 개수 등의 예외가 발생해도 user 입장에서 프로그램의 중지 없이 지속 서비스하기 위한 예외 처리들
- 실제 UI에서 표현해야 할 글을 위한 추가 함수 혹은 구현
- 여러 기능을 담당하는 함수를 하나의 기능만 구현하는 함수로 리팩토링

1. Refine OOPT 2040: OOD – SD 변경

1-b Select Item

0 ~ 20개까지 구매가능한 제약상황 추가

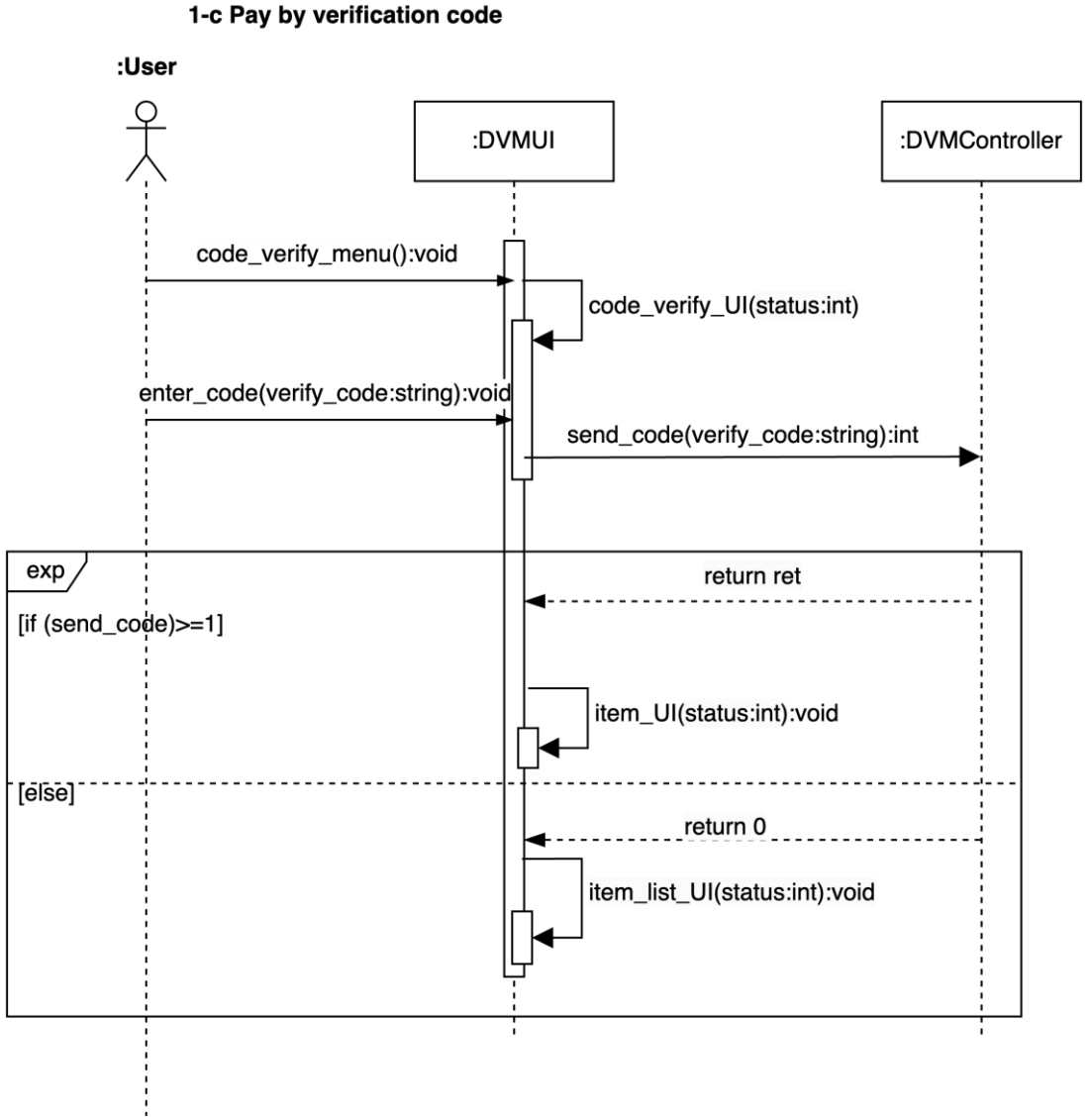
상태가 2개인 경우 UI에 충분한 정보 출력하지 못해 수정



1. Refine OOPT 2040: OOD – SD 변경

1-c Pay by Verification Code

Send_code에서 결제가 된 뒤
음료의 코드, 개수를 받기 위해
boolean이 아닌 int로 변환



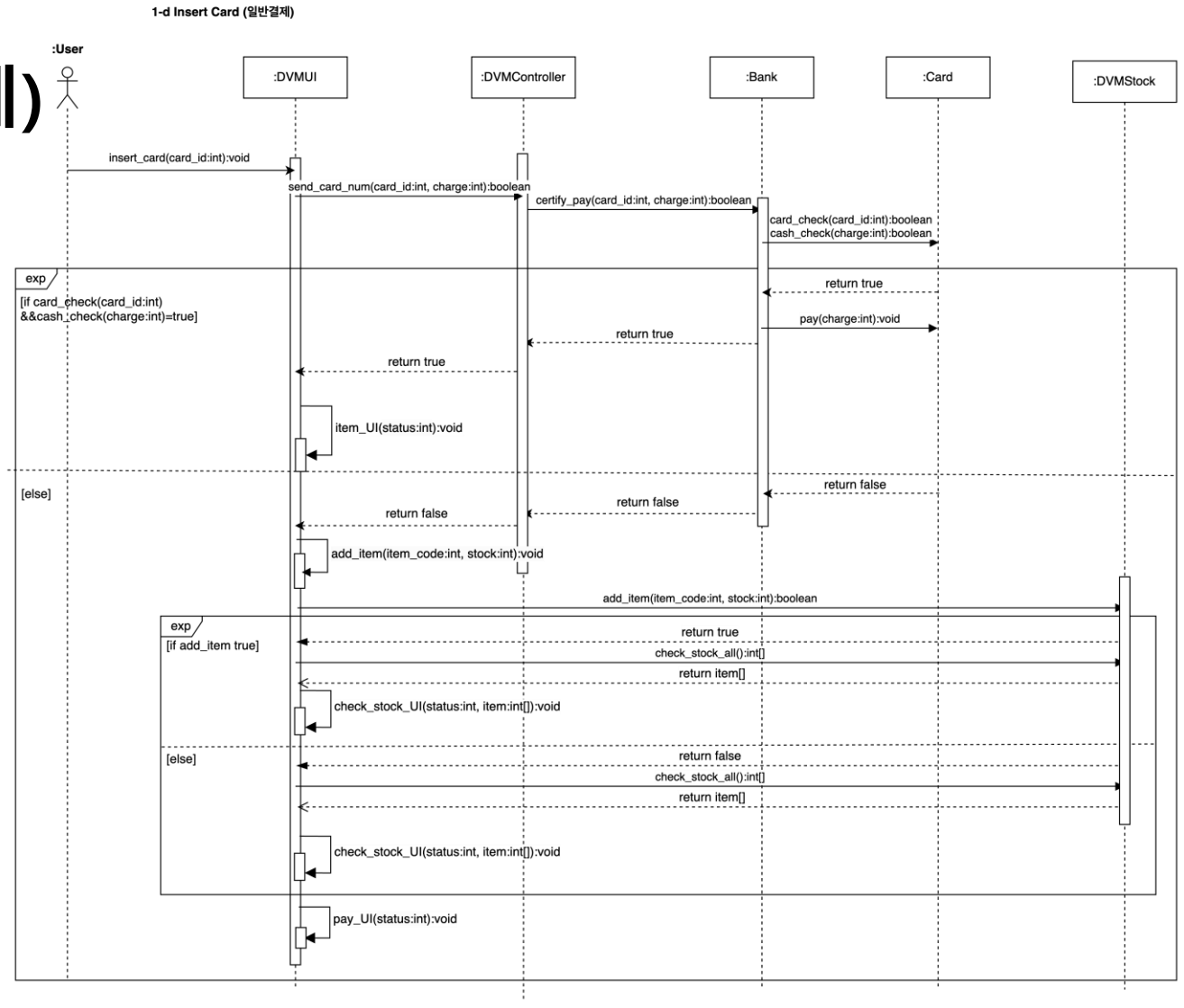
1. Refine OOPT 2040: OOD – SD 변경

1-d Insert Card(일반 결제)

카드 금액 체크가 추가

결제가 실패되는 경우 에러를

롤백을 위한 함수 추가

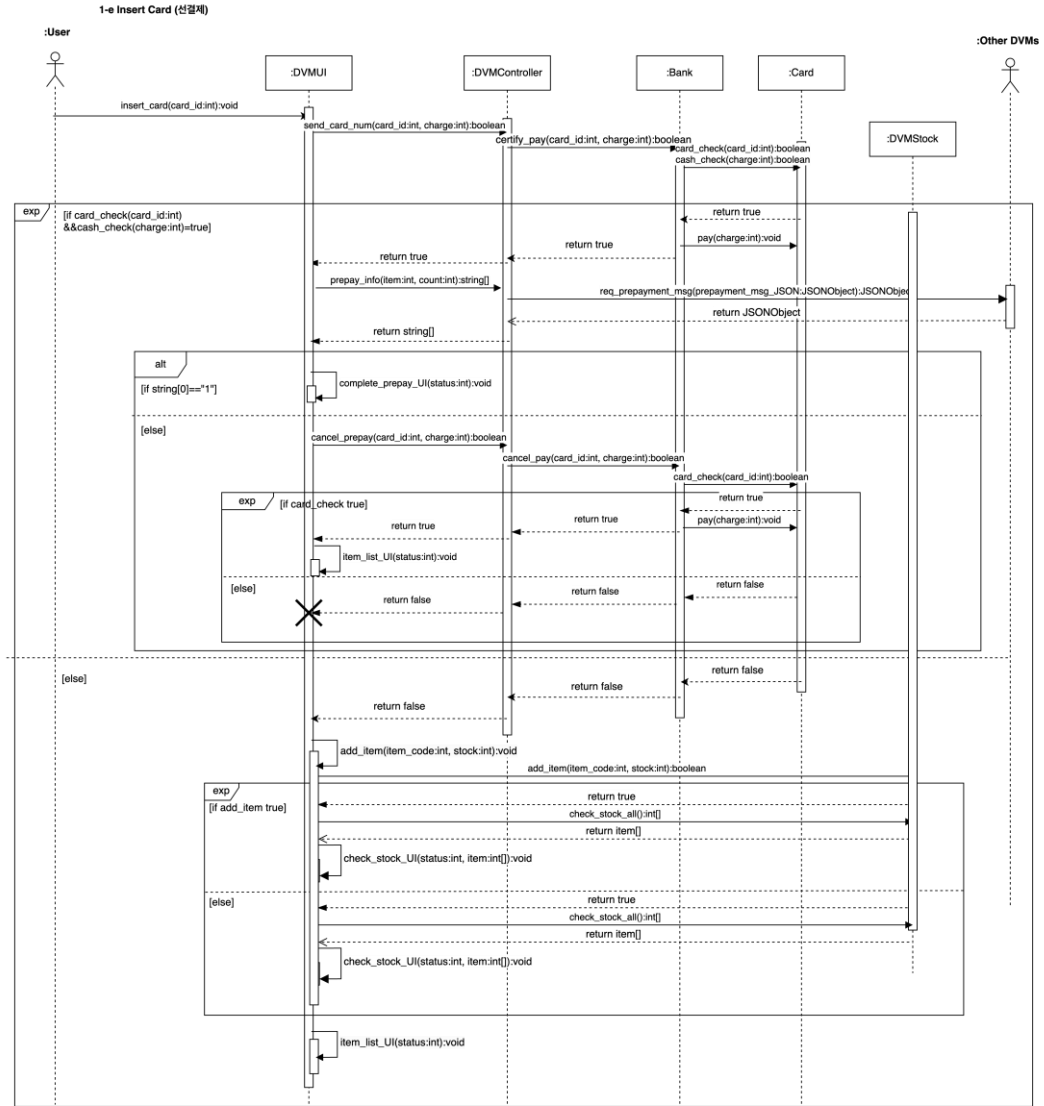


1. Refine OOPT 2040: OOD – SD 변경 변경

1-e Insert Card(선결제)

선결제 중 결제가 실패하는 경우 카드 정보에 문제가 생겨 결제 취소가 되지 않을 경우 강제 종료

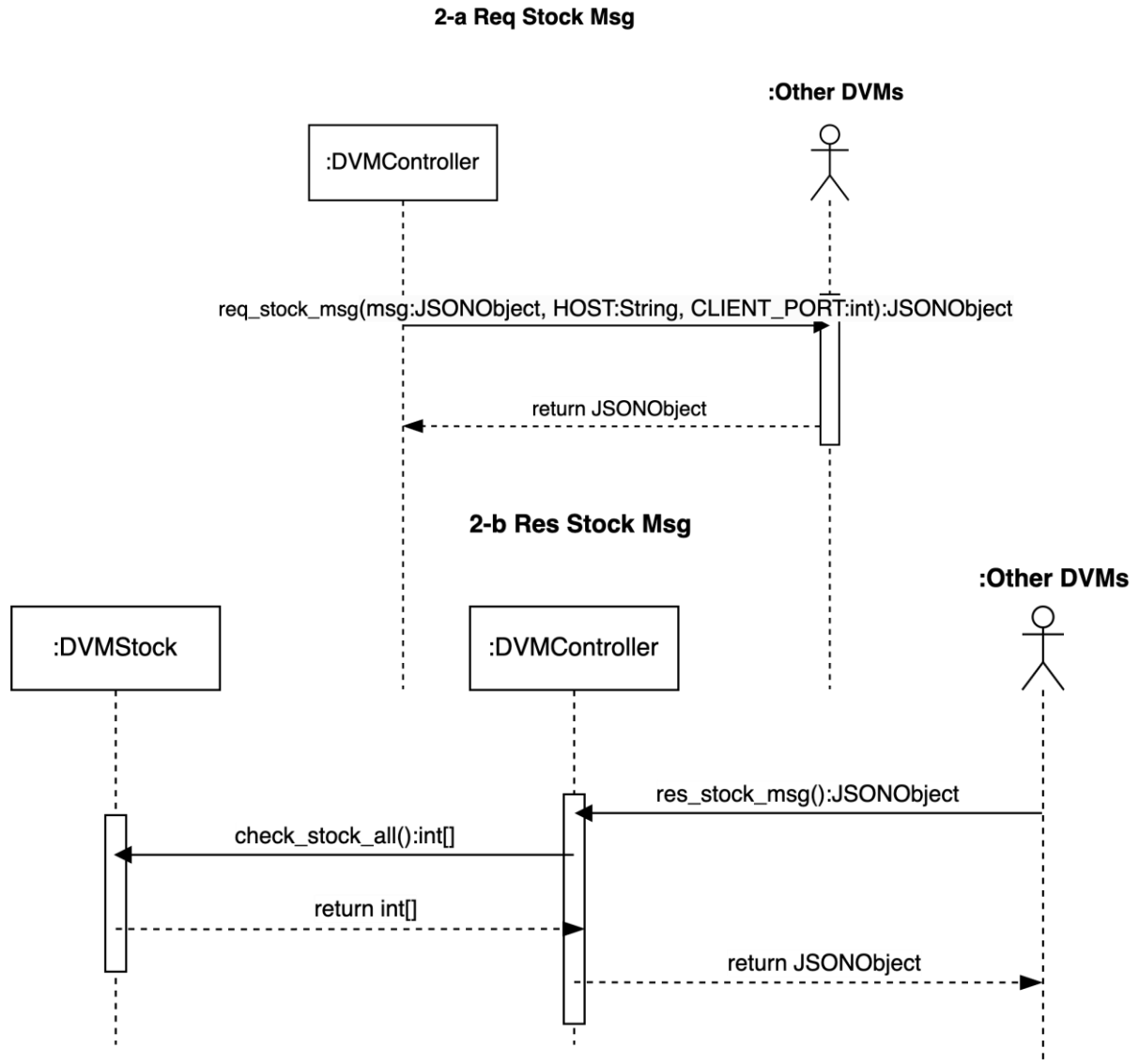
카드 잔액 체크하는 부분 신설



1. Refine OOPT 2040: OOD – SD 변경

2-a, b Req / Res Stock Msg

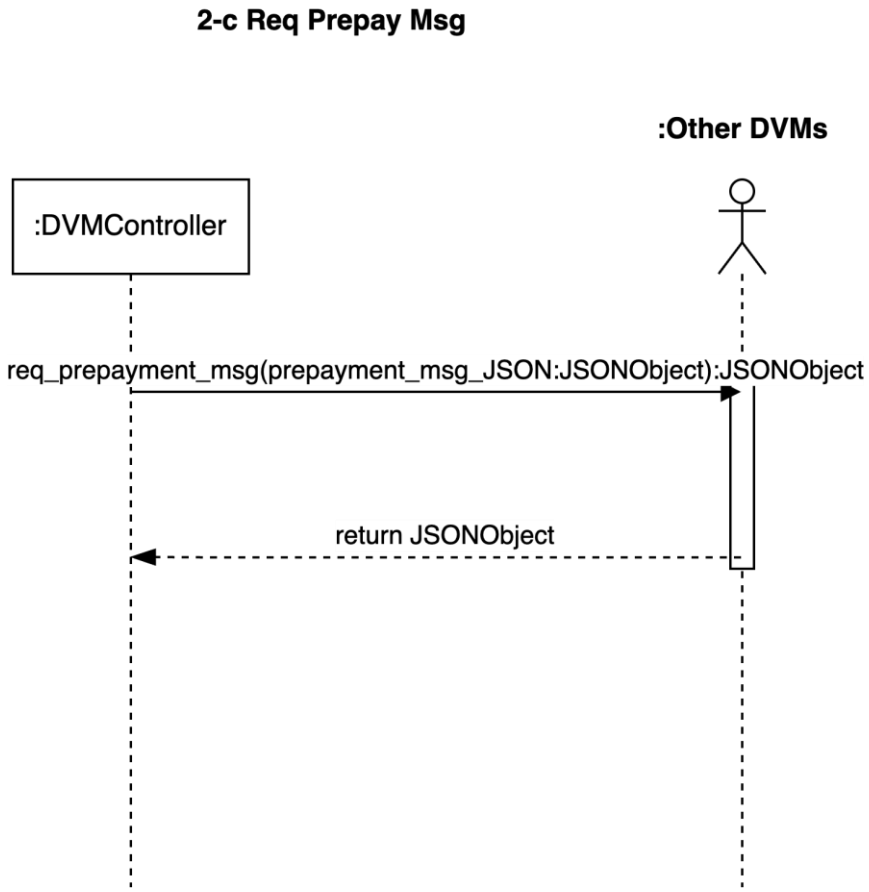
JSONObject를 담아서 보내는 것으로
구체화



1. Refine OOPT 2040: OOD – SD 변경

2-c Req Prepay Msg

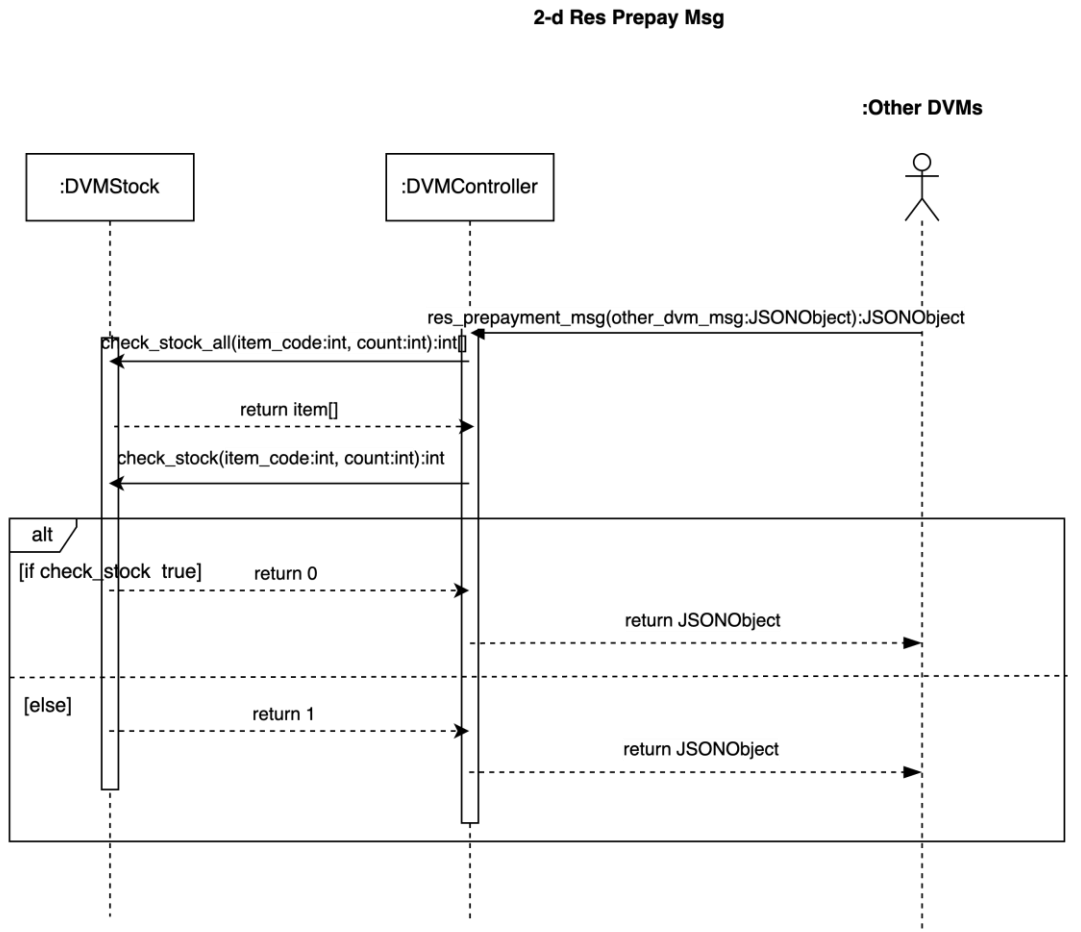
JSONObject를 담아서 보내는 것으로
구체화



1. Refine OOPT 2040: OOD – SD 변경

2-d Res Prepay Msg

Controller에서 받은
JSONObject를 가지고
아이템 리스트를 먼저 확인한 다음
작동하는 방식으로 변경

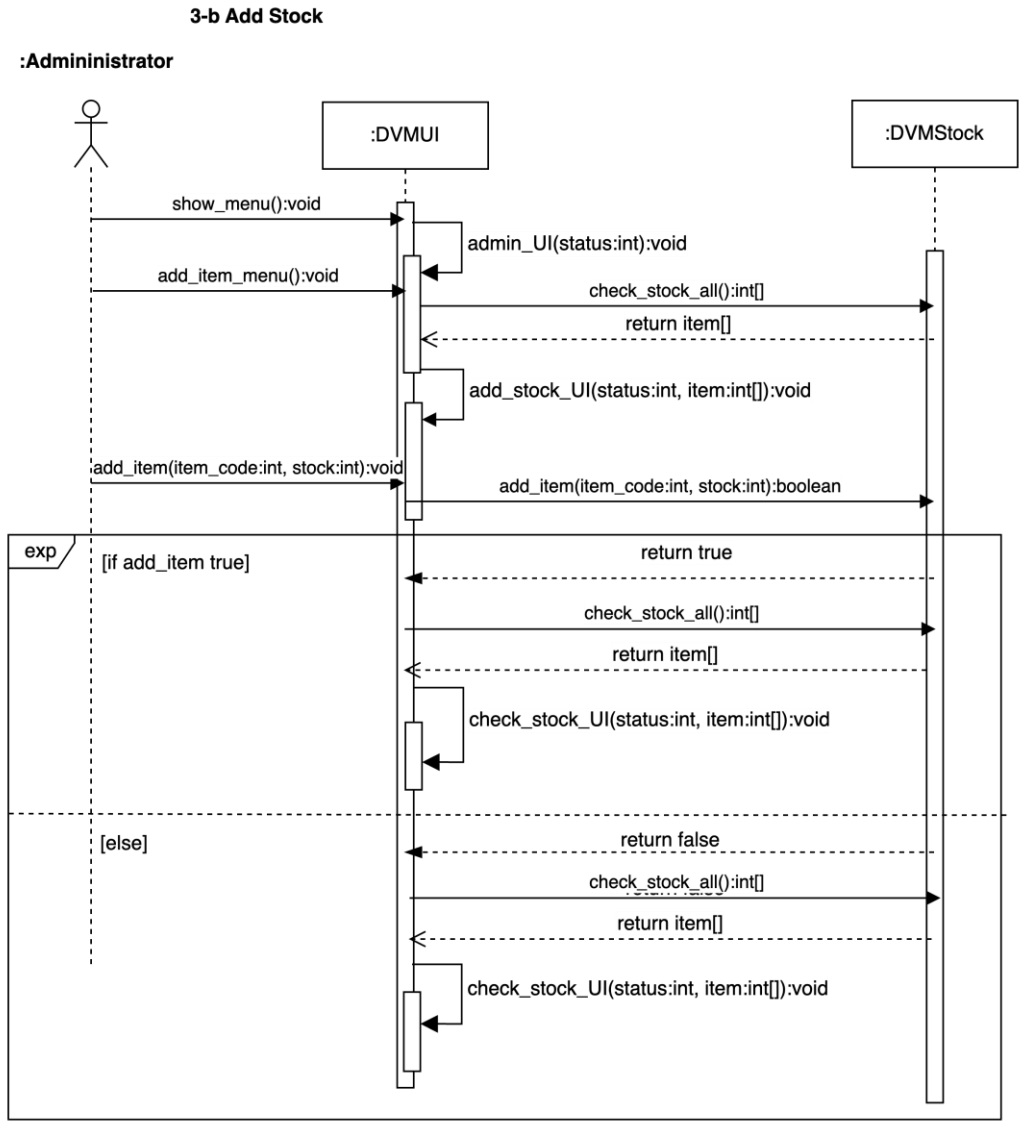


1. Refine OOPT 2040: OOD – SD 변경

3-b Add Stock

재고를 추가하려 할 때 재고 현황을 알기 위한 함수 추가

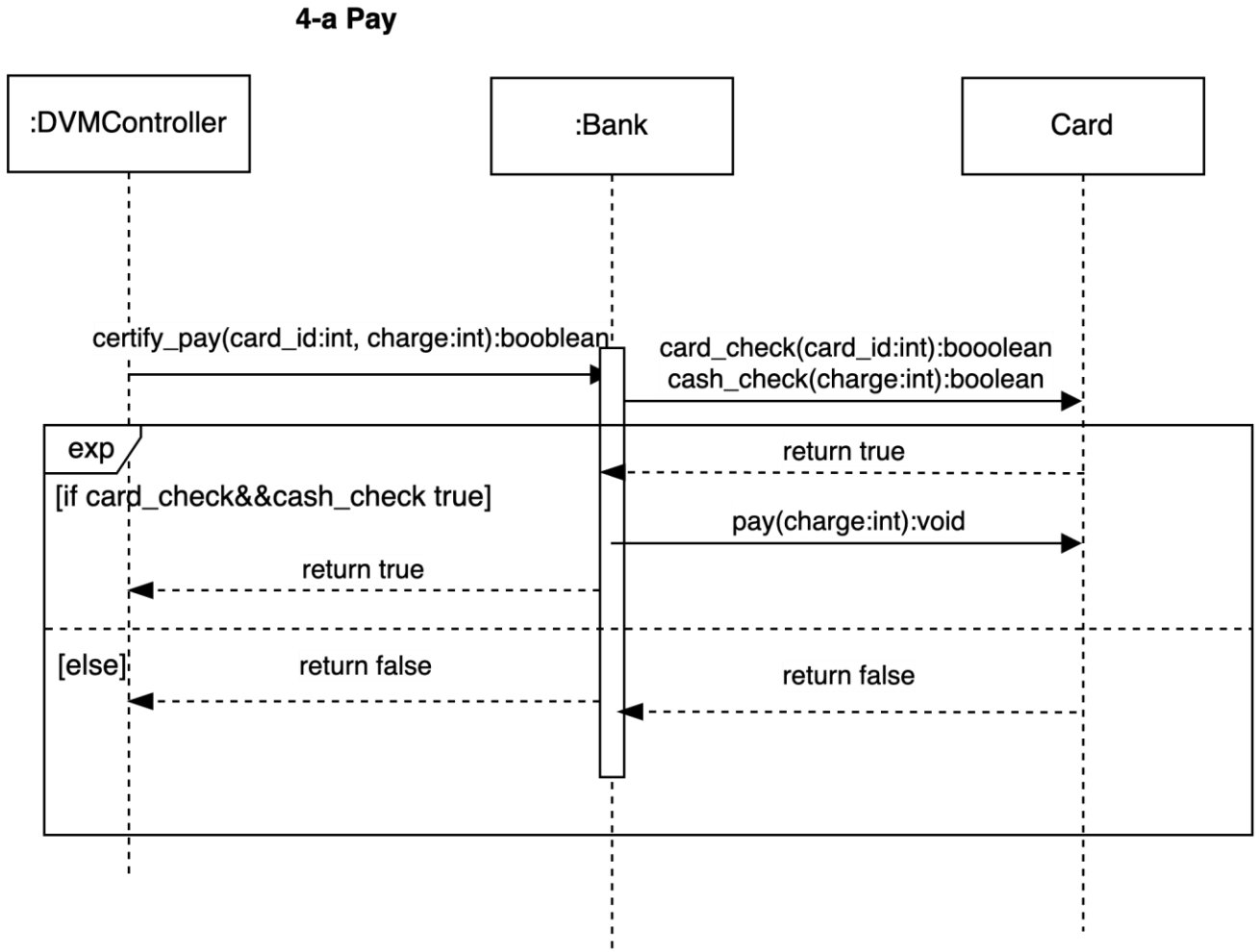
재고를 추가한 뒤, 추가된 재고를 확인하기 위한 함수 추가



1. Refine OOPT 2040: OOD – SD 변경

4-a Pay

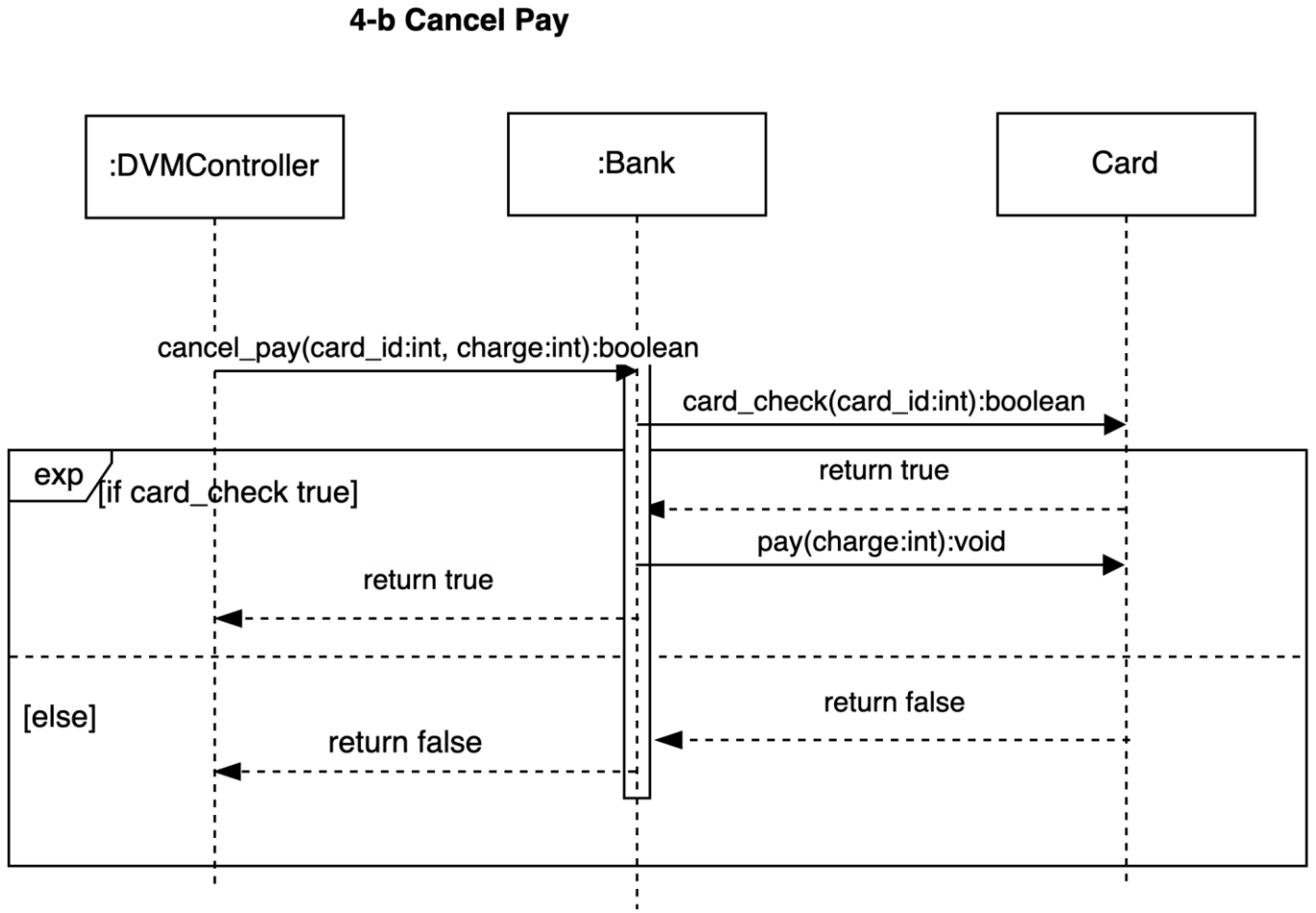
두가지 기능을 하는 함수를
card_check / cash_check함수
두가지로 나눠 재사용성 높임



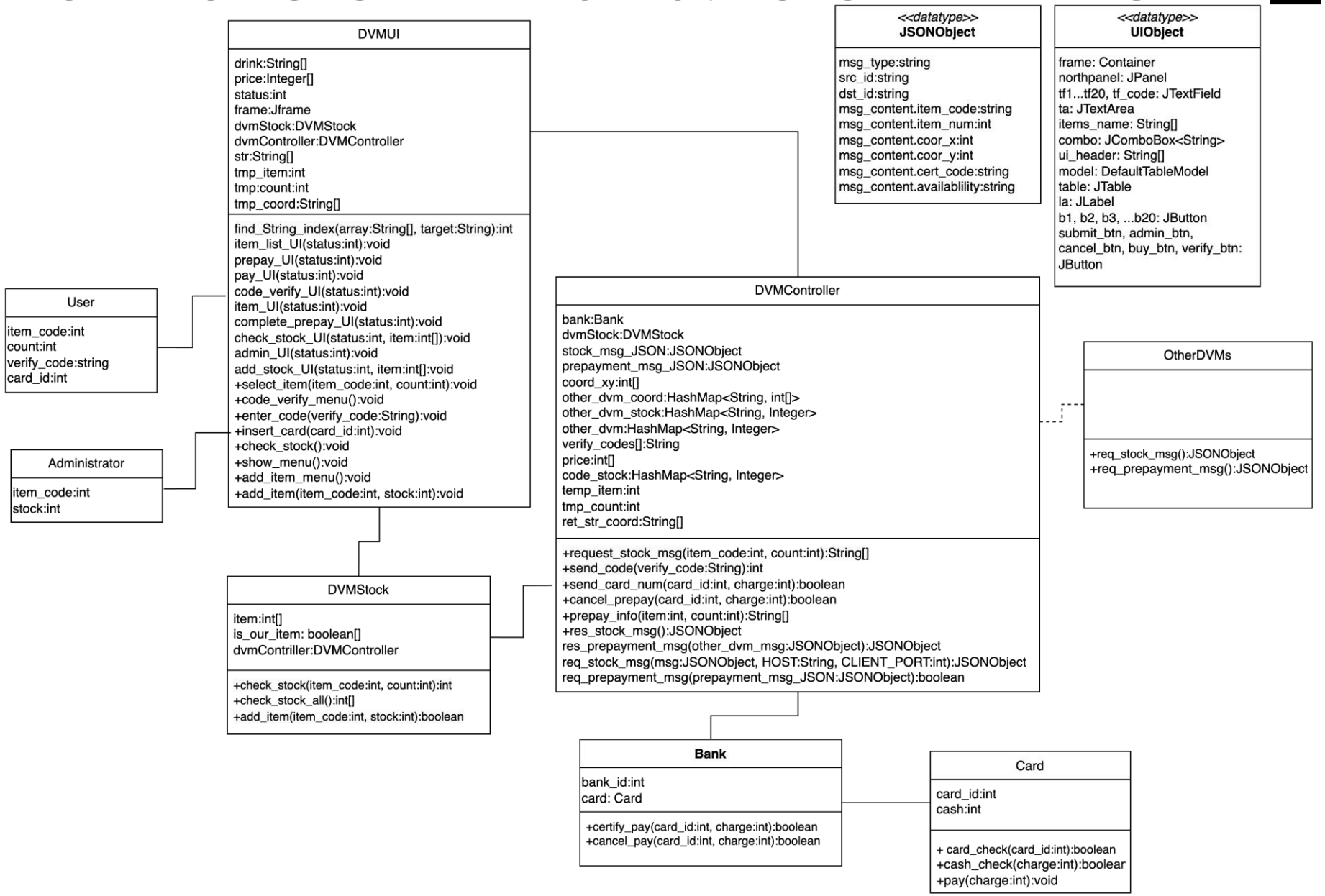
1. Refine OOPT 2040: OOD – SD 변경

4-b Cancel Pay

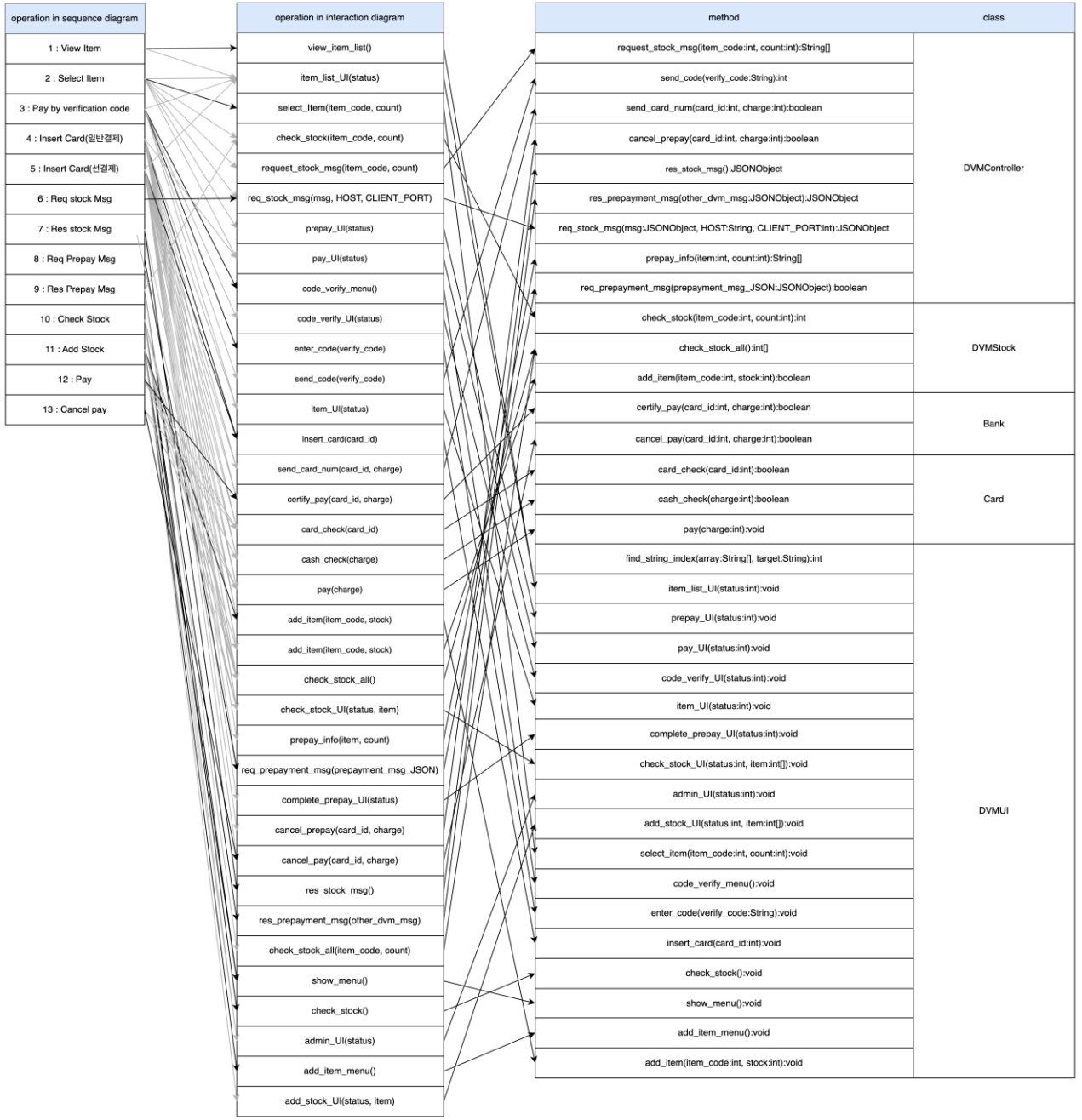
기존에 생략되어 있던 Card
오브젝트 추가
누락되어 있는 로직(pay, card_check)
추가



1. Refine OOPT 2040: OOD – DCD 변경



1. Refine OOPT 2040: OOD – Traceability 변경



2. Unit Test Result

- 각각의 Unit Test는 class 중 DVM UI을 제외한 클래스에 대해 Unit Test를 실행했다.

- 작성한 Unit Test 코드 45개에 대한 커버리지 100%를 만족하였다.

Test Summary

45 tests	0 failures	0 ignored	0.036s duration

100% successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
default-package	45	0	0	0.036s	100%

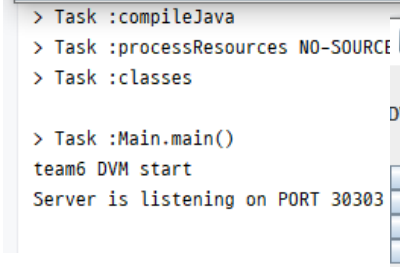
Generated by Gradle 8.2 at 2024. 6. 9. 오전 12:41:54

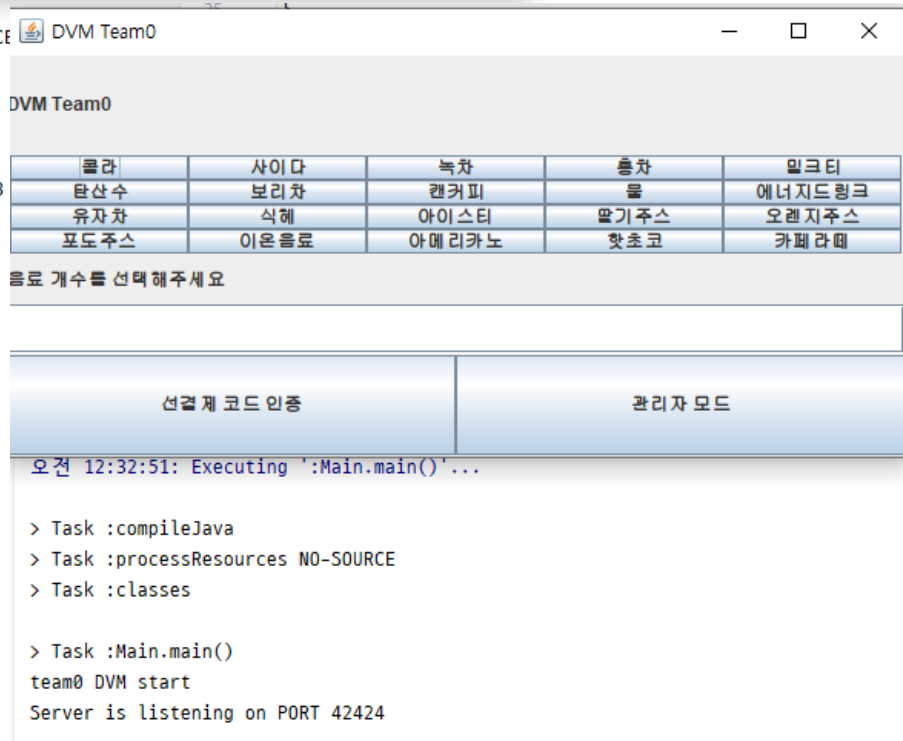
Class	Tests	Failures	Ignored	Duration	Success rate
BankTest	2	0	0	0.007s	100%
CardTest	3	0	0	0.001s	100%
DVMControllerTest	6	0	0	0.020s	100%
DVMStockTest	3	0	0	0.001s	100%
MainTest	31	0	0	0.007s	100%

3. System Operation Demo & System Test Scenario, Result

- Localhost, 포트를 30303(Team6), 42424(Team0)로 다르게 Test 진행
  - 가지고 있는 Item은 Port30303: 1~7, Port42424: 8~14

- GUI까지 통합된 System Test를 진행하며 Demo를 시연

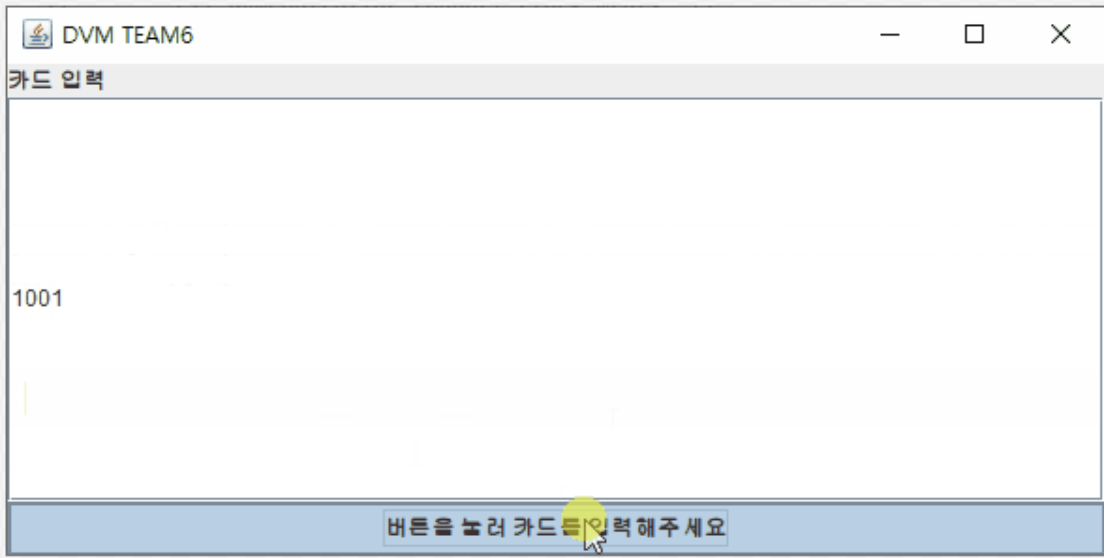


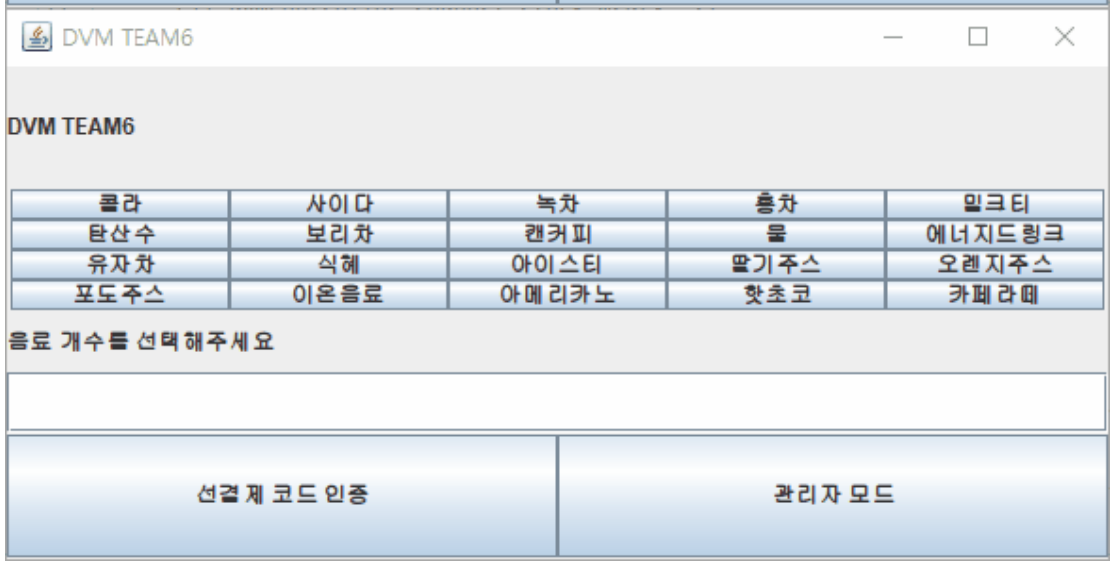


3. System Operation Demo & System Test Scenario, Result

1-a View Item, 1-b Select Item

- 모든 음료 메뉴가 정상적으로 출력되면 음료 선택 가능하다.
- 정상 범위 숫자를 구매 요청하면 카드 입력 화면으로 넘어간다.
- 정상 범위 숫자가 아니면 작동하지 않는다.

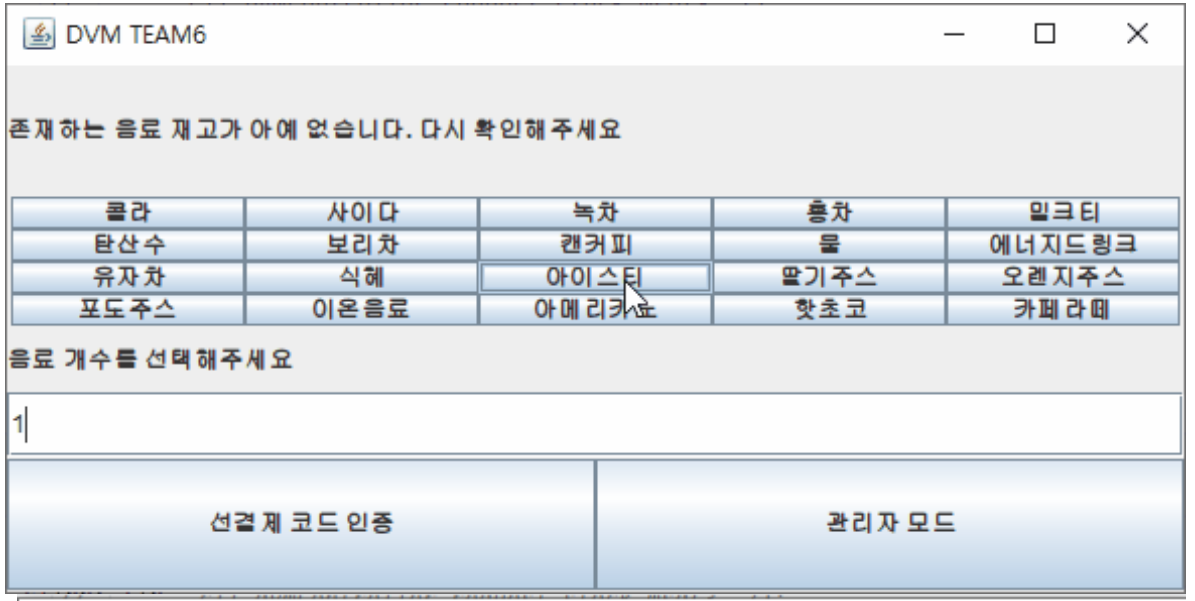


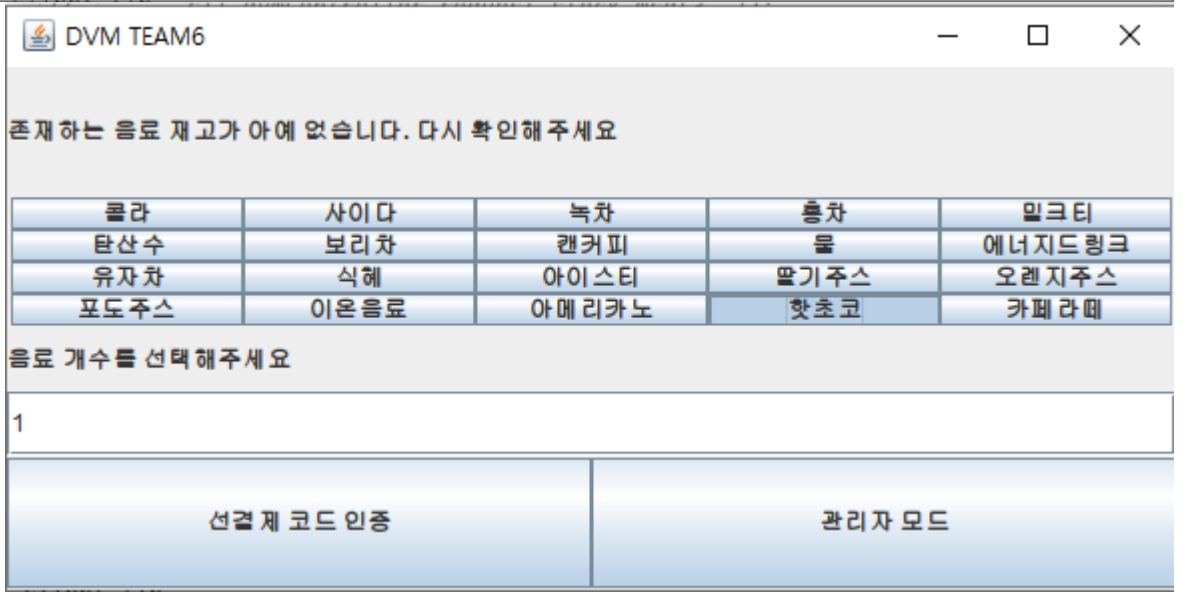


3. System Operation Demo & System Test Scenario, Result

1-b Select Item

- 유저가 선택한 음료가 우리 DVM에 존재할 경우 결제 화면으로 바로 이동한다. (앞 슬라이드)
- 유저 선택 음료가 다른 DVM에서 제공 가능한 경우 선결제 여부를 물어본다.
  - 직접 구매를 선택할 경우 다른 DVM 이름과 위치를 알려준다.
- 유저가 선택한 음료를 제공할 수 없는 경우 초기 화면에서 재고 없음을 알린다.

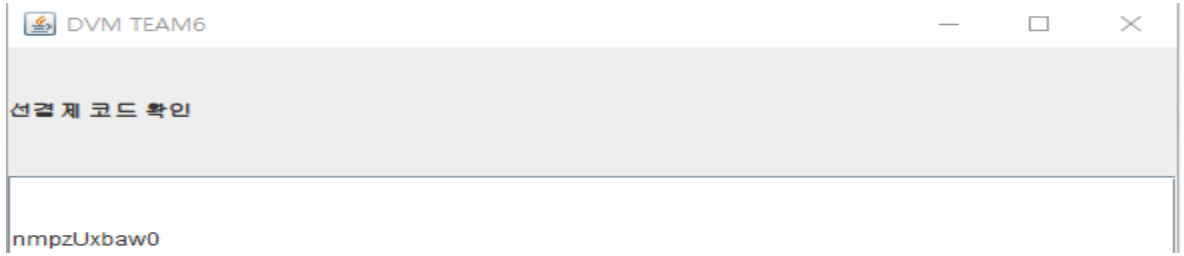




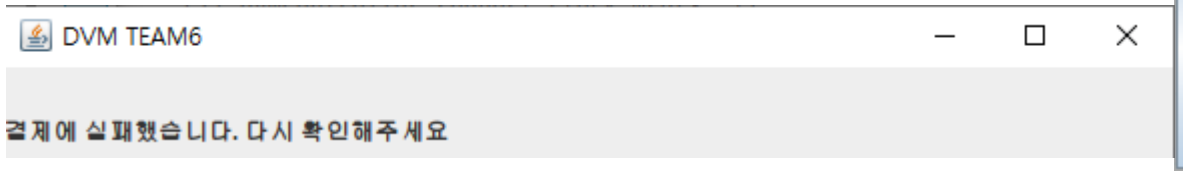
3. System Operation Demo & System Test Scenario, Result

1-c Pay by verification code

- 유저가 올바른 인증 코드를 입력하면, 해당 코드와 맞는 개수의 음료를 제공한다.



- 유저가 올바르지 않은 인증 코드를 입력하면 음료를 제공하지 않고 오류 메시지 띄운다.



3. System Operation Demo & System Test Scenario, Result

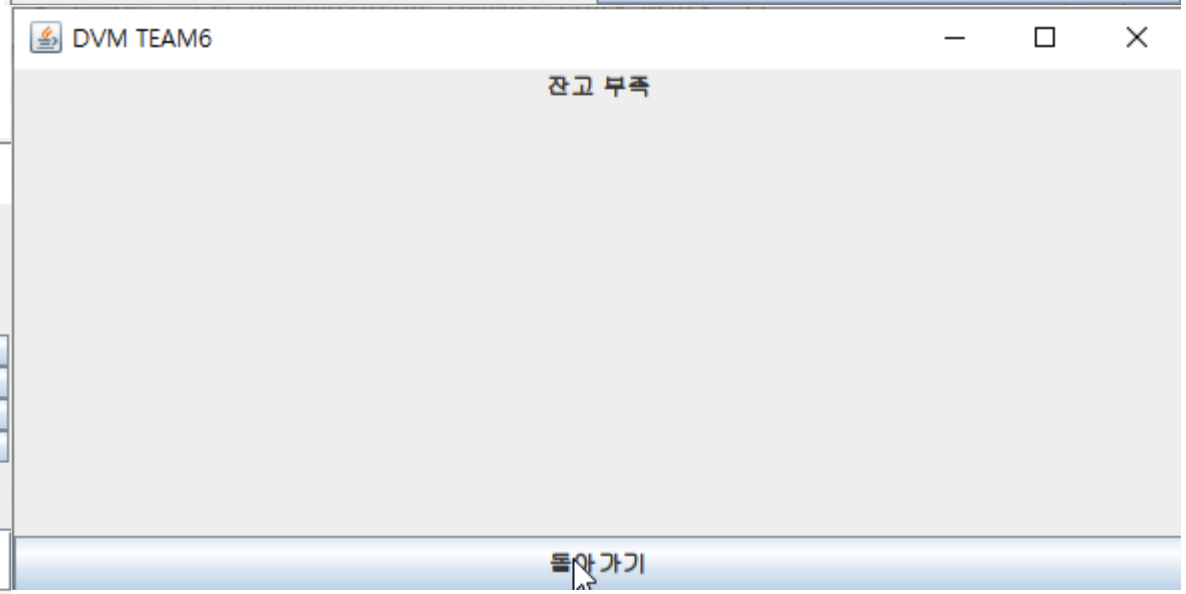
1-d Insert Card(일반결제)

- 유저가 입력한 카드 번호가 올바른 경우, 결제되고 음료를 제공한다.
- 카드 번호가 올바르지 않거나 잔액이 부족한 경우, 결제되지 않고 “잔고 부족 “ 메시지를 띄운 후 음료 선택 화면으로 돌아간다.

3. System Operation Demo & System Test Scenario, Result

1-e Insert Card(선결제)

- 유저가 입력한 카드 번호가 올바른 경우, 선결제되고 다른 DVM 위치와 코드를 제공한다.
- 카드 번호가 올바르지 않거나 잔액이 부족한 경우, 결제되지 않고 “잔고 부족 “ 메시지를 띄운 후 음료 선택 화면으로 돌아간다.



3. System Operation Demo & System Test Scenario, Result

2-a Req Stock Msg

- 다른 DVM에 음료 재고 정보를 요청하는 msg를 보낸다.
  - 다른 DVM의 server가 잘 작동 중이라면, 적절한 재고 정보 응답을 받는다.

```
Client : send to server{"src_id":"Team6","msg_content":{"item_code":"08","item_num":"1"},"msg_type":"req_stock","dst_id":"0"}
Client : server res receive!!{"src_id":"Team0","msg_content":{"item_code":"08","coor_x":27,"coor_y":80,"item_num":"99"},"msg_type":"resp_stock","dst_id":"0"}
client fin
```

- 다른 DVM에서 보낸 응답이 msg API 규격에 맞지 않는 경우 예외 처리를 통해 응답값은 무시한다.
- 다른 DVM와 정보 통신에 실패한 경우 예외 처리를 통해 무시한다.

3. System Operation Demo & System Test Scenario, Result

2-b Res Stock Msg

- 다른 DVM에서 음료 재고 정보를 요청하는 msg를 받는다.
  - 다른 DVM의 client가 잘 작동 중이라면, 적절한 재고 정보 응답을 우리가 다시 보낸다.

```
Server : receive msg {"src_id":"Team0","msg_content":{"item_code":"01","item_num":"4"},"msg_type":"req_stock","dst_id":"0"}
{"src_id":"Team6","msg_content":{"item_code":"01","coor_x":"27","coor_y":"80","item_num":"96"},"msg_type":"resp_stock","dst_id":"0"}
server end
```

- 다른 DVM에서 보낸 요청이 msg API 규격에 맞지 않는 경우 예외 처리를 통해 무시한다.
- 다른 DVM와 정보 통신에 실패한 경우 예외 처리를 통해 무시한다.

3. System Operation Demo & System Test Scenario, Result

2-c Req Prepay Msg

- 다른 DVM에 음료 재고가 남아있다면, 선결제 가능 여부를 요청한다.
  - 다른 DVM의 server가 잘 작동 중이라면, 적절한 선결제 정보 응답을 받는다.

```
Client : send to server{"src_id":"Team0","msg_content":{"item_code":"01","cert_code":"z9JPkBxan4","item_num":"4"},"msg_type":"req_prepay",
 "dst_id":"Team6"}
Client : server res receive!!{"src_id":"Team6","msg_content":{"item_code":"01","availability":"T","item_num":"92"},"msg_type":"resp_prepay",
 "dst_id":"Team0"}
client fin
```

- 다른 DVM에서 보낸 응답이 msg API 규격에 맞지 않는 경우 예외 처리를 통해 무시한다.
- 다른 DVM와 정보 통신에 실패한 경우 예외 처리를 통해 무시한다.

3. System Operation Demo & System Test Scenario, Result

2-d Res Prepay Msg

- 다른 DVM에서 음료 선결제 여부 요청을 받는다.
    - 다른 DVM의 client가 잘 작동 중이라면, 적절한 선결제 정보 응답을 우리가 다시 보낸다.

```
Server : receive msg {"src_id":"Team0","msg_content":{"item_code":"01","cert_code":"z9JPkBxan4","item_num":"4"},"msg_type":"req_prepay","dst_id":"Team6"}
server end
Client : server res receive!!{"src_id":"Team6","msg_content":{"item_code":"01","availability":"T","item_num":"92"},"msg_type":"resp_prepay",
 "dst_id":"Team0"}
client fin
```

- 다른 DVM에서 보낸 요청이 msg API 규격에 맞지 않는 경우 예외 처리를 통해 무시한다.
- 다른 DVM와 정보 통신에 실패한 경우 예외 처리를 통해 무시한다.

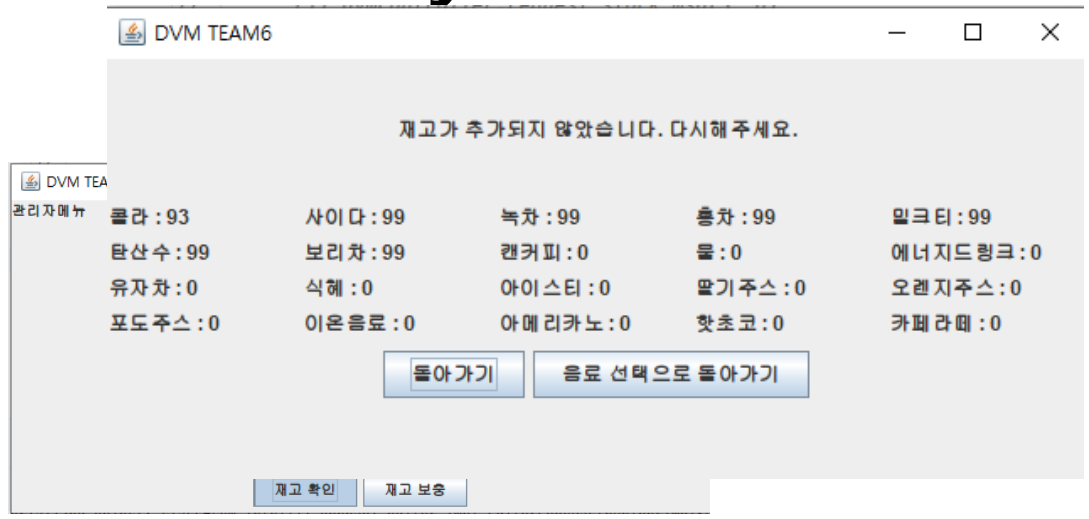
3. System Operation Demo & System Test Scenario, Result

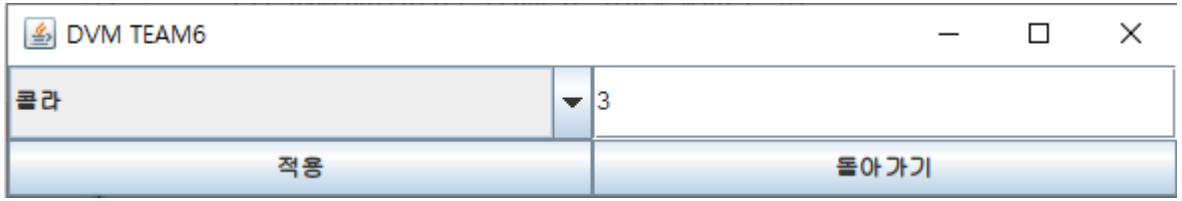
3-a Check Stock

- 재고가 존재하는지 확인 (개수 출력)
- 재고가 정상적으로 출력되는지 확인

3-b Add Stock

- 음료 재고 추가
- 추가된 개수만큼 화면에 반영
- 우리가 기존에 가진 7가지 음료만 추가 가능





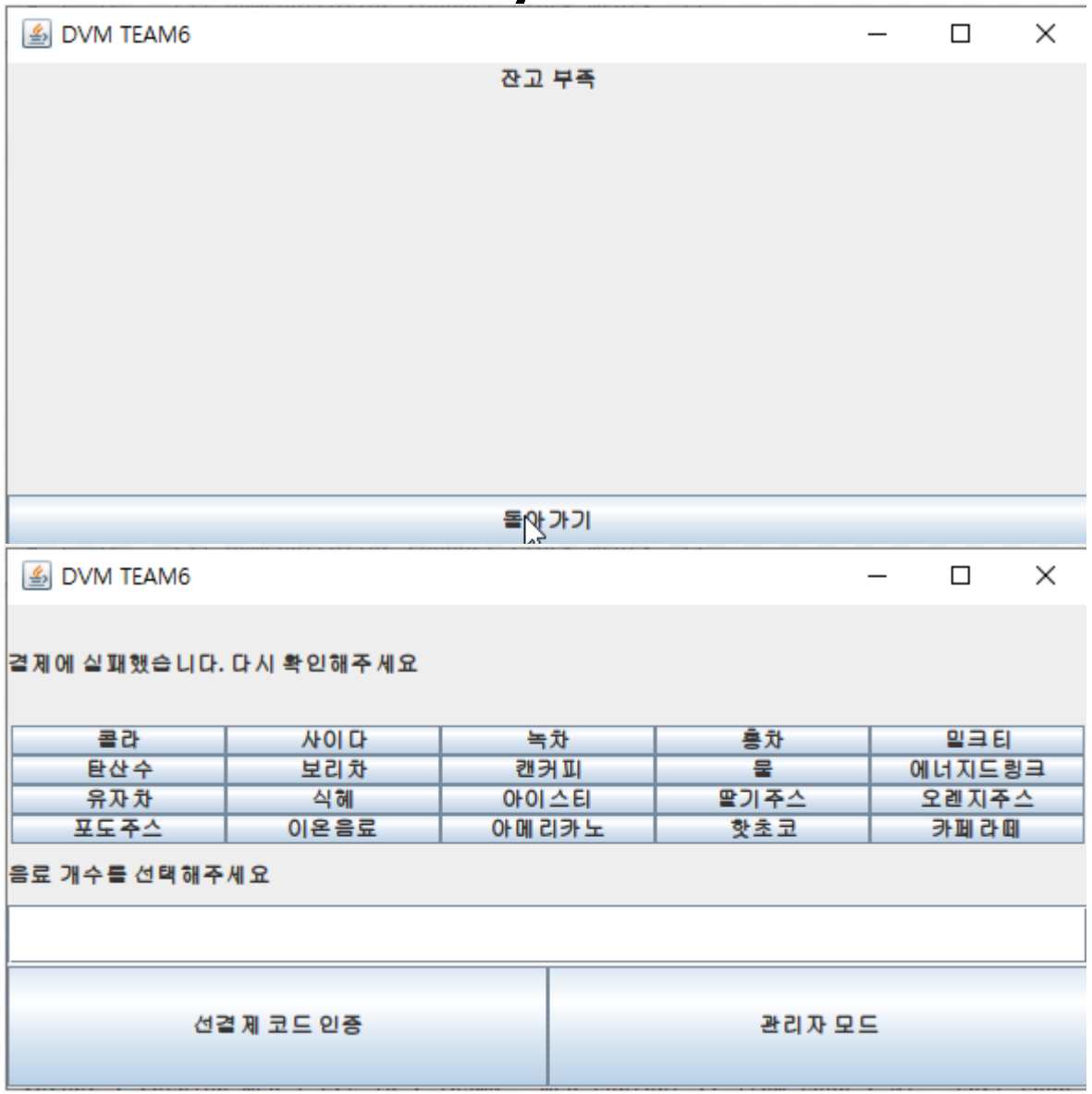
3. System Operation Demo & System Test Scenario, Result

4-a Pay

- 카드 잔액 충분한 경우 결제 완료
- 카드 잔액 없는 경우 “잔액 부족” 띄우고 초기 화면 돌아가기

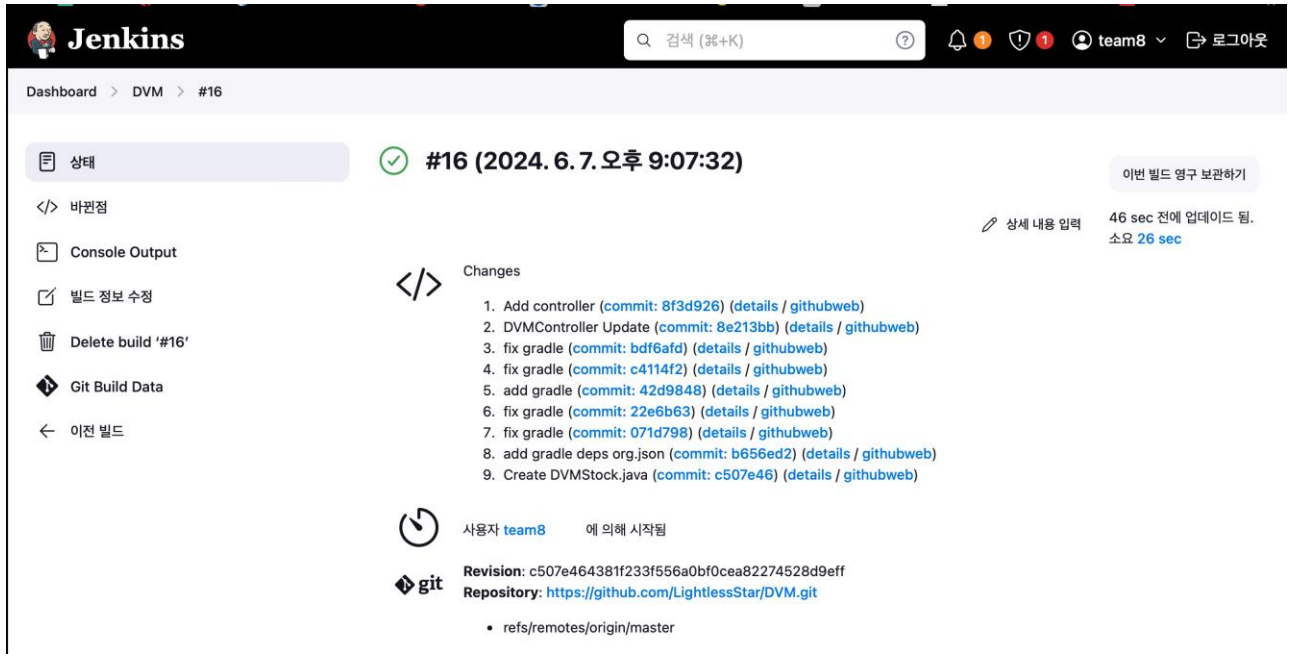
4-b Cancel Pay

- 선결제 실패 시, 자동으로 기존 결제 건이 취소
- 카드 정보 올바르지 않은 경우, 결제 실패 안내 메시지 띄우며 초기 화면으로 이동



4. CI/CD, CTIP, UT Development Environment

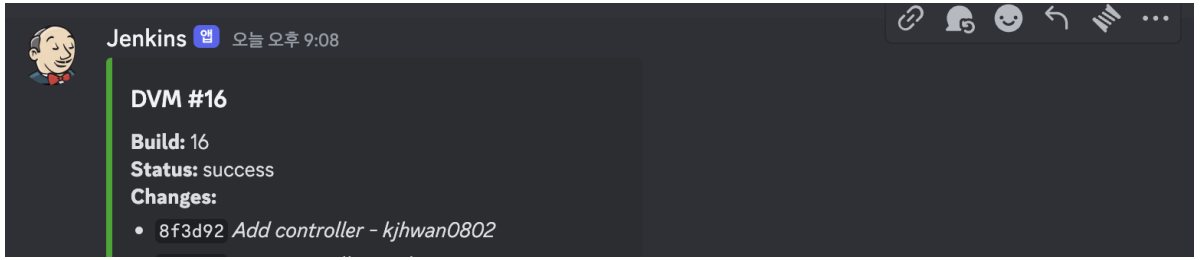
- On-premise 환경에서 도커 위에 Jenkins를 설치해 CI/CD를 구현함.

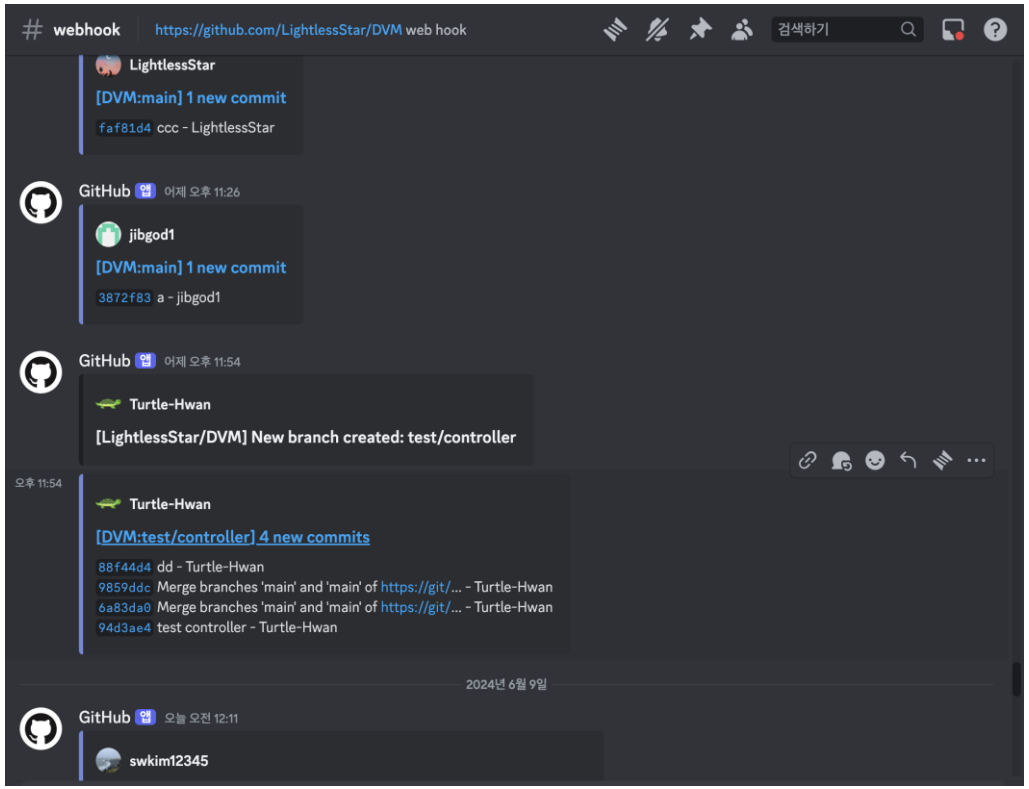


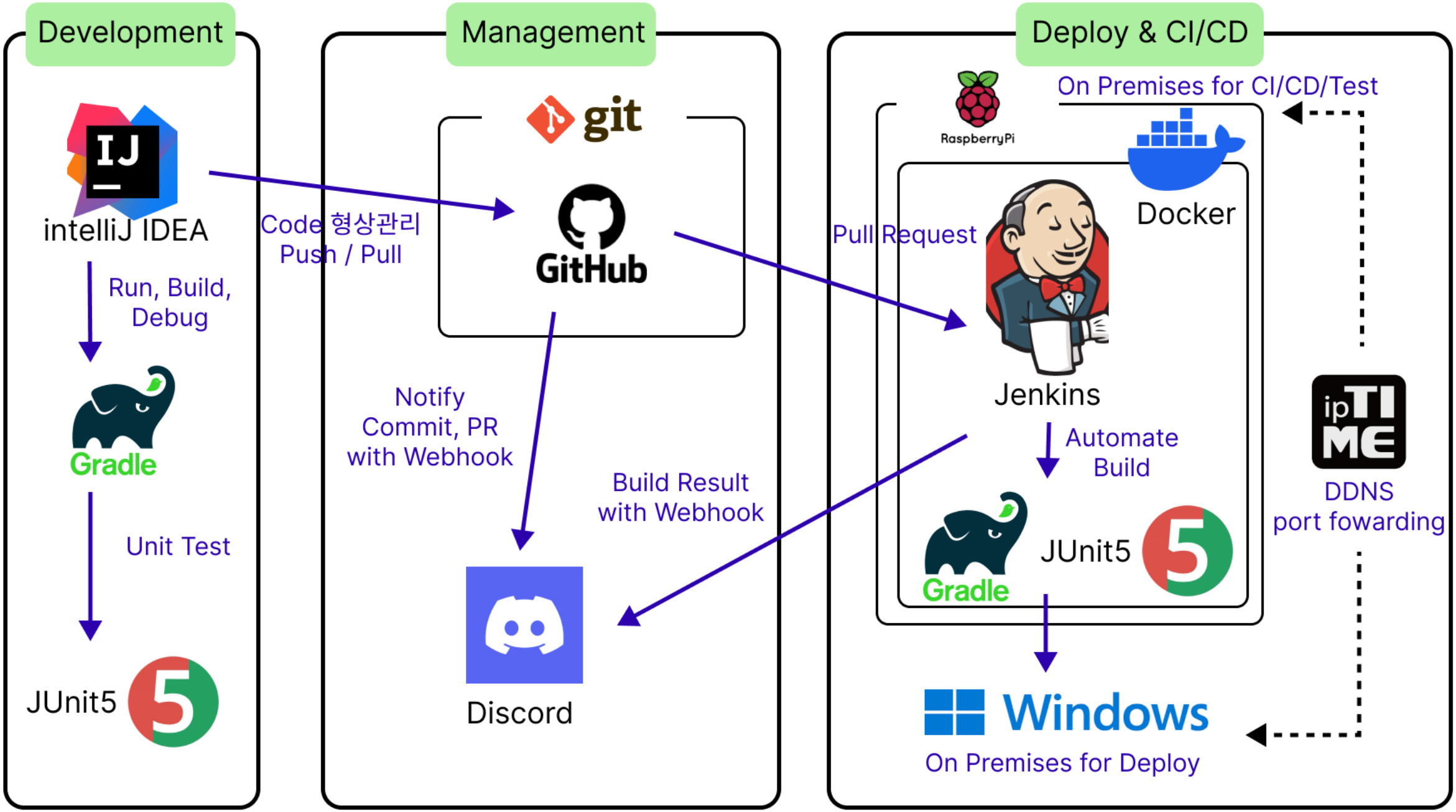
4. CI/CD, CTIP, UT Development Environment

- Github pull request와 Jenkins 빌드가 끝나게 되면 메시지를 Webhook으로 받을 수 있게 설정

- 빠르게 테스트의 결과를 확인






Development
Management
Deploy & CI/CD
intelliJ IDEA
Run, Build, Debug
Gradle
Unit Test
JUnit5
Code 형상관리
Push / Pull
git
GitHub
Notify
Commit, PR
with Webhook
Discord
Build Result
with Webhook
On Premises for CI/CD/Test
RaspberryPi
Docker
Pull Request
Jenkins
Automate
Build
Gradle
JUnit5
ipTIME
DDNS
port fowarding
Windows
On Premises for Deploy

5. 구현 시 예상보다 어려웠던 점

- 시간 부족
- CI/CD 환경 구축 : Webhook, token등록 등 많은 할 일 존재
- Socket 통신 시 Multi-Thread 구현
  - 하나의 DVM program이 다른 DVM 요청을 대기하는 동시에, 사용자 입력을 받아 다른 DVM에 요청을 보내야 하는 구조
- 미처 발견하지 못한 Architecture Risk가 남아있던 점
  - JSONObject 통신 시 String / Int 형식 미통일
  - Socket을 닫는 시점 불일치
- Iteration을 하지 못해 오류, 예외 처리가 힘들었던 점

5. 구현 시 예상보다 어려웠던 점

- 순환 참조 문제
  - DVMStock에서 DVMController instance 생성. 반대의 경우도 instance 생성해서 순환참조
  - DVMStock의 DVMController에 대한 의존성 없애도록 코드, SD 수정

- Class 의존성 문제
  - DVMStock이 DVMUI도 의존하고, DVMController도 의존하는데, 각각 instance를 생성해서 별개의 Stock을 가지는 문제
  - main에서 공통 instance 생성해서 넣어주는 것으로 해결

- 위 두 문제 모두 처음부터 디자인 패턴을 잘 적용해서 설계했었더라면 깔끔하게 해결할 수 있었겠지만 실력, 시간 부족으로 인한 어려움 및 아쉬움

5. 구현 시 예상보다 어려웠던 점

- 다양한 PFR 구현 방법
  - 구현을 할 때, 다른 DVM을 구현하는 팀 간의 소통이 없이 팀 별로 따로 구현을 진행하니 PFR을 해석하는 방식이 여러가지 존재
  - 각각의 PFR구현은 틀리지 않았으나, 통합할 때 오류 발생

6. 구현 시 예상보다 쉬웠던 점

- Swing 도입으로 인한 GUI의 빠른 구현
  - 비즈니스 로직이 없는 부분에서 다소 빠르게 구현 가능
  - 잘 몰랐던 객체 지향 언어임에도 불구하고, 언어적 리스크가 다소 해결

```
JLabel title = new JLabel( text: "DVM TEAM6");
contentPane.setLayout(new GridLayout( rows: 4,  cols: 1));
contentPane.removeAll();
//4 * 5 짜리 jpanel을 만들어 jframe에 부착하는 작업.
if (status == ERROR) {
    title.setText("결제에 실패했습니다. 다시 확인해주세요");
} else if (status == PREPAY) {
    title.setText("좌표는 " + tmp_coord[1] + "입니다  x : " + tmp_
} else if (status == NULLABLE) {
    title.setText("존재하는 음료 재고가 아예 없습니다. 다시 확인해주세요");
}
contentPane.add(title);
```

7. 객체지향개발방법론의 장단점

장점

- OOA, OOD를 미리 설계한 후 OOI를 진행하면 구현 시간이 짧아진다.
  팀원과 프로젝트에 대한 공통 도메인 지식을 공유하게 되어 협업이 원활해진다.
- UP 개발방법론을 따르면 risky drive하게 진행되어 Architecture risk와 Client risk가 해결된 채로 OOI가 진행되어 더 수월하다.

단점

- 디자인 패턴, 실제 개발에 관한 도메인 지식이 부족한 상태로 OOAD를 진행하면 OOI, code 구현 시 문제 및 수정 사항이 많이 발생할 수 있다.
- 미처 찾지 못한 architecture risk가 있다면 OOI 단계에서 큰 risk가 될 수 있다.
- 충분한 Iteration이 없는 상황에서는 OOD 설계 사항에 맞추어 구현하기 힘들고, 깔끔하지 않은 구현이 나올 수 있다.

결론 : 주어진 프로젝트의 기간과 비용에 맞게 Cost-effective 한 방법론을 선택하는 것이 가장 좋다.

8. 소감

- 김지환
  - 힘들지만 배워가는 것이 많았습니다. 코딩보다 문서화 시간이 길었지만 좋은 결과물이 나온 신기한 경험이었습니다.
- 김성환
  - 문서를 먼저 작성하는 것을 처음 해봐 어려웠지만, 가면 갈 수록 프로그램에 대해 이해도가 높아졌습니다.
  - 같은 PFR 구현에도 다양한 구현방법이 존재하고, 해결방법이 다르다는 것,
    - 협업의 중요성, risk analysis의 중요성을 느꼈습니다.
- 이송헌
  - 실제 프로젝트를 진행할 때 필요한 문서와 디자인들을 공부하고 그려보는 방법을 배운 유익한 수업이었습니다. 또한 객체 지향적인 프로그램이란 무엇인가에 대해 고찰하고 직접 구현까지 할 수 있었던 경험은 정말 좋았습니다. 앞으로도 수업에서 배운 내용을 잊지 않고 적용해나가도록 하겠습니다.
- 조찬형
  - 수업을 통해 문서로 정리하고 코딩하는 방법을 배운 덕분에, 팀원들과 생각을 더 잘 공유할 수 있게 되었고 협업에도 큰 도움이 되었습니다.

감사합니다