SW설계공학
프로젝트 결과 보고서

제목: 데일리 스케줄러 (Daily Scheduler)

소속 학과	컴퓨터소프트웨어공학과
제출일	2021년 06월 05일
1분반 2조	20163341 김동영 ,20173217 안대현, 20183139 김영주, 20193215 조정윤 20193200 진예림

목 차

1. 프로젝트 개요

1.1 비전

빠르게 돌아가는 사회 속에서 스케줄을 효율적으로 관리하는 것은 매우 중요하다. 그러나 계속해서 생겨나는 일정과 반복되는 일정 등을 체계적으로 관리하고 실행하기는 쉽지 않은 일이다. 이 때문에 다양한 일정을 손쉽게 관리할 수 있는 스케줄러가 필요하다.

데일리 스케줄러는 처음 스케줄러를 사용하는 사람도 쉽게 사용할 수 있도록 간단한 기능을 통해 일정 추가 및 수정, 삭제를 할 수 있도록 하며 그룹 지정, 백그라운드 지정, 중요도 설정 등의 기능을 제공함으로써 많은 양의 일정도 체계적으로 관리할 수 있도록 한다.

1.2 문제 기술

연번	해결하려는 문제	관련 기술
F-01	로그인 기능을 지원해야 한다.	사용자가 로그인한 정보의 정확성을 확인한다.
F-02	회원가입 기능을 지원해야 한다.	사용자가 회원가입한 정보와 DB의 기존 정보와 중복성을 비교한다.
F-03	회원 정보를 수정할 수 있다.	회원 정보를 수정하면 해당 정보가 서버와 연결된 데이터베이스에도 수정된다.
F-04	회원 탈퇴를 할 수 있다.	회원 탈퇴를 하면 연결된 데이터베이스의 데이터를 모두 삭제한다.
F-05	스케줄을 등록할 수 있다.	스케줄을 등록하면 해당 정보가 서버와 연결된 데이터베이스에 저장된다.
F-06	스케줄 기간을 설정할 수 있다.	스케줄 기간을 입력하면 해당 정보가 서버와 연결된 데이터베이스에 입력된 스케줄 기간이 저장된다.
F-07	스케줄에 간단한 메모를 추가할 수 있다.	메모를 추가하면 서버와 연결된 데이터베이스에 해당 메모를 저장한다.

F-08	스케줄의 중요도를 설정할 수 있다.	중요도를 설정하면 서버와 연결된 데이터베이스에 선택한 중요도를 저장한다.
F-09	등록된 스케줄 내용을 변경할 수 있다.	스케줄 내용을 변경하면 해당 정보가 서버와 연결된 데이터베이스에 변경되어 저장된다.
F-10	등록된 일정을 삭제할 수 있다.	스케줄을 삭제하면 해당 정보가 서버와 연결된 데이터베이스에 삭제된다.
F-11	중요한 스케줄을 고정시키거나 즐겨찾기를 할 수 있다.	스케줄을 고정시키면 서버와 연결된 데이터베이스에 해당 스케줄을 고정시켜 화면에 보여준다.
F-12	스케줄을 종료하는 체크 기능을 제공해야 한다.	스케줄을 종료하여 체크하면 서버와 연결된 데이터베이스에 해당 스케줄의 종료에 대한 체크를 저장한다.
F-13	스케줄 등록 및 확인 화면의 색상을 지정하여 구분할 수 있다.	각각의 스케줄 화면의 색상을 선택하여 변경하면 스케줄 화면의 색상을 변경하여 나타낸다.

1.3 특징 목록

연번	우선순위	특징 목록	유형
F-01	1	로그인 기능을 지원해야 한다.	기능
F-01-01	-	회원 정보를 외부의 서버에서 관리한다.	비기능
F-01-02	-	로그인 성공 시 사용자 정보를 불러온다.	비기능
F-01-03	-	비밀번호 찾기가 가능해야 한다.	비기능
F-02	1	회원가입 기능을 지원해야 한다.	기능

F-02-01	-	회원 정보 수정을 할 수 있다.	기능
F-02-02	-	회원 탈퇴를 할 수 있다.	기능
F-03	1	스케줄을 등록할 수 있다.	기능
F-04	1	스케줄 기간을 설정할 수 있다.	기능
F-05	2	스케줄에 간단한 메모를 추가할 수 있다.	기능
F-06	2	스케줄의 중요도를 설정할 수 있다.	기능
F-07	1	등록된 스케줄 내용을 수정할 수 있다.	기능
F-08	2	등록된 일정을 삭제할 수 있다.	기능
F-09-01	-	다수의 일정을 한번에 삭제할 수 있어야 한다.	비기능
F-10	2	중요한 스케줄을 고정시키거나 즐겨찾기를 할 수 있다.	기능
F-11	2	스케줄을 종료하는 체크 기능을 제공해야 한다.	기능
F-12	3	스케줄 등록 및 확인 배경화면의 색상을 지정하여 구분할 수 있다.	기능

1.4 관련 기술

연번	특징 목록	관련 기술
F-01	로그인 기능을 지원해야 한다.	신뢰할 수 있는 회원을 인증하고, 인증 정보를 일정 기간 동안 유지할 수 있도록 지원한다.
F-02	회원가입 기능을 지원해야 한다.	회원가입 기능으로 스케줄러를 사용하는 모든 사용자의 정보를 데이터베이스에 저장한다.
F-03	회원 정보를 수정할 수 있다.	저장된 회원 정보를 선택하여 수정할 수 있도록 한다.
F-04	회원 탈퇴를 할 수 있다.	회원 탈퇴를 하게 되면, DB에 저장된 모든 데이터가 삭제되어야 한다.
F-05	스케줄을 등록할 수 있다.	N/A
F-06	스케줄 기간을 설정할 수 있다.	N/A
F-07	스케줄에 간단한 메모를 추가할 수 있다.	N/A
F-08	스케줄의 중요도를 설정할 수 있다	N/A
F-09	등록된 스케줄 내용을 변경할 수 있다.	기존 데이터가 새로운 데이터로 변경되어 저장되어야 한다.
F-10	등록된 일정을 삭제할 수 있다.	DB에 저장된 일정도 삭제되어야 한다.
F-11	중요한 스케줄을 고정시키거나 즐겨찾기를 할 수 있다.	관련된 스케줄을 상단에 띄우는 등 순서를 재배치할 수 있어야 한다.
F-12	스케줄을 종료하는 체크 기능을 제공해야 한다.	N/A
F-13	스케줄 등록화면의 색상을 변경할 수 있다.	N/A

1.5 이해 당사자 요구사항

연번	이해 당사자	요구사항
St-01	스케줄러 기획자	기본적으로 지원되는 일정 생성 양식 외에 사용자의 기호에 따라 쉽게 일정을 추가할 수 있도록 기획하여야 한다.
St-02	스케줄러 사용자	스케줄러를 사용하려면, 회원가입을 해야 한다.
		회원 정보를 변경할 수 있다.
		스케줄 일정과 내용 변경이 유연해야 한다.
		중요한 스케줄을 따로 모아 볼 수 있다.
		스케줄러 배경 색상을 변경할 수 있다.
St-03	개발자	· 기능이 변경되거나 추가되는 경우 기존 스케줄에 오류가 생기지 않도록 해야 한다. · 등록된 데이터를 저장할 수 있는 서버를 구현한다.

2. 시스템 분석

2.1* 유스케이스 "회원가입" (UC-WM100)

관련 요구사항	SFR-WM101, SFR-WM102, , SFR-WM107		
액터	사용자		
목적	시스템을 사용하기 위해 회원가입을 한다.		
개요	시스템을 사용하기 위해 회원가입을 한다. 회원 정보를 입력 후 회원가입 버튼을 누르면 데이터베이스에 정보가 기록된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 시스템에 접속한다.	
	2		시스템은 **회원가입 화면**을 보여준다.
	3	사용자는 ID, 비밀번호, 성, 이름, 휴대폰 번호, 이메일, 사진을 입력한 뒤, 회원가입 버튼을 클릭한다.	
	4		시스템은 사용자의 회원 ID의 중복성을 확인한다. (E-4)
	5		시스템은 사용자의 회원 정보를 DB에 저장한다.
	6		시스템은 회원 가입 성공 알림을 표시한다.
예외 흐름	**E-4-01. 중복된 ID**		
	1	사용자는 회원 정보를 입력한 후 회원가입 버튼을 클릭한다.	
	2		시스템은 사용자가 입력한 ID가 기존 DB에 있는지 확인하고, 가입 실패 원인을 분석한다.
	3		시스템은 **이미 가입되어 있는 ID로 인한 회원가입 실패 화면**을 보여준다.

2.2* 유스케이스 "회원 관리" (UC-WM200)

관련 요구 사항	SFR-WM103		
액터	사용자		
목적	회원 정보 수정 또는 삭제를 위해 메뉴를 선택할 수 있다.		
개요	회원 정보 수정 또는 삭제하기 버튼을 클릭하면 해당 화면이 표시된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 회원 관리를 원할 때 유스케이스가 시작된다.	
	2	사용자가 회원 정보 수정 메뉴를 선택한다.	
	3		Change_Leave 화면을 보여준다.
	4	사용자가 회원수정 버튼을 선택한다.	
	5		회원 정보 수정 유스케이스**(UC-201)**를 실행한다.
	6	사용자가 회원 삭제 메뉴를 선택한다.	
	7		회원 삭제 유스케이스**(UC-202)**를 실행한다.

2.3* 유스케이스 "회원 정보 수정" (UC-WM201)

관련 요구사항	SFR-WM105, SFR-WM106, , SFR-WM107		
액터	사용자		
목적	회원 정보를 수정한다.		
개요	회원 정보를 수정하기 위해, 수정할 값을 입력한다. DB 의 값을 입력된 값으로 수정한다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 회원 관리를 원할 때 유스케이스가 시작된다.	
	2		시스템은 회원 정보 수정 화면을 보여준다.
	3	사용자는 비밀번호, 성, 이름, 휴대폰 번호, 이메일, 사진중 수정할 값을 선택적으로 입력한 후, 수정하기 버튼을 클릭한다.	
	4		시스템은 DB에 기록된 기존 값을 새로운 값으로 수정한다.
	5		시스템은 회원 정보 수정 성공 알림을 표시한다.

2.4* 유스케이스 "회원 삭제" (UC-WM202)

관련 요구사항	SFR-WM104, SFR-WM106		
액터	사용자		
목적	회원 탈퇴를 한다.		
개요	회원 탈퇴를 위해 탈퇴 버튼을 클릭한다. 시스템은 기록된 사용자 정보를 DB에서 삭제한다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 회원 삭제 버튼을 클릭하면 유스케이스가 시작된다.	
	2	현재 사용하고 있는 비밀번호를 입력한다.(E-2)	
	3		시스템은 비밀번호가 맞는지 비교한다.
	4		시스템은 입력된 비밀번호가 맞다면 DB에 기록된 회원 정보를 삭제한다.
	5		시스템은 회원 삭제 성공 알림을 표시한다.
	6		시스템은 화면을 로그인 화면으로 전환시킨다.
예외 흐름	**E-2-01. 틀린 암호**		
	1	사용자는 틀린 사용자 암호를 입력한다.	
	2		시스템은 암호의 정확성을 확인한다.
	3		시스템은 **유효하지 않은 암호로 인한 로그인 실패 화면**을 보여준다.

2.5* 유스케이스 "로그인" (UC-WM300)

관련 요구사항	SFR-WM201, SFR-WM202, SFR-WM203, SFR-WM204		
액터	사용자		
목적	시스템에 로그인한다.		
개요	사용자 ID와 암호를 입력하여 시스템에 로그인한다. 사용자 ID와 암호가 틀린 경우 실패 원인을 알려준다. 로그인 성공 시 메인 화면을 보여준다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 시스템에 접속한다.	
	2		시스템은 **로그인 화면**을 보여준다.
	3	사용자는 정확한 사용자 ID와 암호를 입력하고 로그인한다. **(E-3)**	
	4		사용자 ID와 암호의 정확성을 확인한 후, JWT 토큰을 생성한다.
	5		로그인 성공 시 메인 화면을 보여준다.
예외 흐름	**E-3-01. 틀린 사용자 ID**		
	1	사용자는 틀린 사용자 ID와 정확한 암호를 입력하고 로그인한다.	
	2		시스템은 사용자 ID와 암호의 정확성을 확인하고, 로그인 실패 원인을 분석한다.
	3		시스템은 **유효하지 않은 ID로 인한 로그인 실패 화면**을 보여준다.
예외 흐름	**E-3-02. 틀린 암호**		
	1	사용자는 정확한 사용자 ID와 틀린 암호를 입력하고 로그인한다.	
	2		시스템은 사용자 ID와 암호의 정확성을 확인하고, 로그인 실패 원인을 분석한다.
	3		시스템은 **유효하지 않은 암호로 인한 로그인 실패 화면**을 보여준다.

2.6* 유스케이스 "비밀번호 변경" (UC-WM301)

관련 요구사항	SFR-WM205, SFR-WM206		
액터	사용자		
목적	사용자 비밀번호를 찾는다.		
개요	비밀번호를 찾기 위해 사용자 ID를 입력한다. ID의 정확성과 비밀번호 동일여부를 확인하고 비밀번호를 재설정시킨다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 비밀번호 찾기를 누르면 유스케이스가 시작된다.	
	2		시스템은 **비밀번호 찾기 화면**을 보여준다.
	3	사용자는 정확한 사용자 ID 와 새 비밀번호 및 비밀번호를 재입력한다. **(E-3)**	
	4		사용자 ID의 정확성과 비밀번호와 재입력한 비밀번호의 동일여부를 확인한다.
	5		정보가 일치하면 DB에서 사용자의 비밀번호를 불러와 화면에 표시한다.
예외 흐름	**E-3-01. 틀린 사용자 ID**		
	1	사용자는 틀린 사용자 ID와 새 비밀번호를 입력한다.	
	2		시스템은 DB에 저장되어 있는 사용자 ID와 사용자가 입력한 ID를 비교하여, 비밀번호 찾기 실패 원인을 분석한다.
	3		시스템은 **유효하지 않은 ID로 인한 비밀번호 변경 실패 화면**을 보여준다.
예외 흐름	**E-3-02. 틀린 재입력 암호**		
	1	사용자는 정확한 사용자 ID, 새 비밀번호 입력 후 비밀번호 재입력시 틀린 비밀번호를 입력하고 비밀번호 찾기를 클릭한다.	
	2		시스템은 비밀번호 재입력시 동일 여부를 체크하고, 비밀번호 찾기 실패 원인을 분석한다.
	3		시스템은 **틀린 비멀번호로 인한 비밀번호 변경 실패 화면**을 보여준다.

2.7* 유스케이스 "스케줄 추가" (UC-WM400)

관련 요구사항	SFR-WM302, SFR-WM303, SFR-WM304, SFR-WM305, SFR-WM306, SFR-WM307		
액터	사용자		
목적	스케줄(task)을 추가한다.		
개요	스케줄을 추가하기 위해 필요한 정보를 입력한 후 , 추가 버튼을 누르면 해당 정보가 서버와 연결된 데이터베이스에 저장된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 로그인 화면에서 로그인이 성고하면 task를 추가하고자 할 때 유스케이스가 시작된다.	
	2		시스템은 task 화면을 보여준다.
	3	사용자는 스케줄을 추가하기 위해 task를 추가하기 위한 버튼을 클릭한다.	
	4		시스템은 task를 추가하기 위한 팝업창을 보여준다.
	5	사용자는 task의 제목, 메모, 중요도, 시작 날짜, 끝 날짜에 해당하는 값을 입력한 후 추가 버튼을 클릭한다.	
	6		시스템은 서버에 연결된 DB에 해당 스케줄 정보를 저장한다.
	7		시스템은 스케줄 추가 성공 알림을 보여준다.

2.8* 유스케이스 "스케줄 수정" (UC-WM401)

관련 요구사항	SFR-WM302, SFR-WM303, SFR-WM304, SFR-WM305, SFR-WM306, SFR-WM308, SFR-WM309		
액터	사용자		
목적	스케줄(task)을 수정한다.		
개요	스케줄을 수정하기 위해 필요한 정보를 입력한 후, 수정 버튼을 누르면 데이터베이스에 기록된 기존 정보가 수정된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 task 를 수정하고자 할 때 유스케이스가 시작된다.	
	2	사용자는 스케줄을 수정하기 위해 task 를 수정하기 위한 버튼을 클릭한다.	
	3		시스템은 task를 수정하기 위한 팝업창을 보여준다.
	4	사용자는 수정할 task의 제목을 고른 후 메모, 중요도, 시작 날짜, 끝 날짜에 해당하는 값 중에 수정하고 싶은 값을 선택적으로 입력한 후 수정하기 버튼을 클릭한다.	
	5		시스템은 서버에 연결된 DB에서 기존 스케줄 값을 수정된 값으로 변경한다.
	6		시스템은 스케줄 수정 성공 알림을 보여준다.

2.9* 유스케이스 "스케줄 삭제" (UC-WM402)

관련 요구사항	SFR-WM308, SFR-WM309		
액터	사용자		
목적	스케줄(task)을 삭제한다.		
개요	스케줄을 삭제하기 위해 삭제할 스케줄을 선택한 후, 삭제하기 버튼을 누르면 데이터베이스에 기록된 스케줄 정보가 삭제된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 삭제시키고자 하는 스케줄을 선택할 때 유스케이스가 시작된다.	
	2	사용자는 스케줄을 삭제하기 위해 task 를 수정하기 위한 버튼을 클릭한다.	
	3		시스템은 task를 삭제하기 위한 팝업창을 보여준다.
	4	사용자는 스케줄을 삭제할 task 의 제목을 선택하여 삭제하기 버튼을 클릭한다	
	5		시스템은 서버에 연결된 DB에서 해당 스케줄 레코드를 삭제한다.
	6		시스템은 스케줄 삭제 성공 알림을 보여준다.

2.10* 유스케이스 "스케줄 확인" (UC-WM403)

관련 요구사항	SFR-WM310		
액터	사용자		
목적	스케줄(task)을 확인한다.		
개요	스케줄을 확인하기 위해 확인 버튼을 누르면 해당 정보가 서버와 연결된 데이터베이스에 저장되어 있던 값들이 불러와진다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 스케줄(task)를 보고자 할 때 유스케이스가 시작된다.	
	2	사용자는 등록한 task 를 보기 위한 버튼을 클릭한다.	
	3		시스템은 DB에 저장된 task 를 보여준다.

2.11* 유스케이스 “todo 추가" (UC-WM500)

관련 요구사항	SFR-WM302, SFR-WM303, SFR-WM304, SFR-WM305, SFR-WM306, SFR-WM307		
액터	사용자		
목적	todo를 추가한다.		
개요	todo를 추가하기 위해 필요한 정보를 입력한 후, 추가 버튼을 누르면 해당 정보가 서버와 연결된 데이터베이스에 저장된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 todo 화면에서 todo를 추가하고자 할 때 유스케이스가 시작된다.	
	2		시스템은 todo 화면을 보여준다..
	3	사용자는 todo를 추가하기 위한 버튼을 클릭한다.	
	4		시스템은 todo를 추가하기 위한 팝업창을 보여준다.
	5	사용자는 todo의 제목, 메모, 중요도, 그룹, 상단 고정 여부에 해당하는 값을 입력한 후 추가 버튼을 클릭한다.	
	6		시스템은 서버에 연결된 DB에 해당 todo 정보를 저장한다.
	7		시스템은 todo 추가 성공 알림을 보여준다.

2.12* 유스케이스 “todo 수정" (UC-WM501)

관련 요구사항	SFR-WM302, SFR-WM303, SFR-WM304, SFR-WM305, SFR-WM306, SFR-WM308, SFR-WM309		
액터	사용자		
목적	todo를 수정한다.		
개요	todo를 수정하기 위해 필요한 정보를 입력한 후, 수정 버튼을 누르면 데이터베이스에 기록된 기존 정보가 수정된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 todo를 수정하고자 할 때 유스케이스가 시작된다.	
	2	사용자는 스케줄을 수정하기 위해 todo 를 수정하기 위한 버튼을 클릭한다.	
	3		시스템은 todo 를 수정하기 위한 팝업창을 보여준다.
	4	사용자는 수정할 todo의 제목을 고른 후 메모, 중요도, 그룹, 상단 고정 여부, 완료 여부에 해당하는 값 중에 수정하고 싶은 값을 선택적으로 입력한 후 수정하기 버튼을 클릭한다.	
	5		시스템은 서버에 연결된 DB에서 기존 todo 값을 수정된 값으로 변경한다.
	6		시스템은 todo 수정 성공 알림을 보여준다.

2.13* 유스케이스 "todo 삭제" (UC-WM502)

관련 요구 사항	SFR-WM308, SFR-WM309		
액터	사용자		
목적	todo를 삭제한다.		
개요	todo를 삭제하기 위해 삭제할 todo를 선택한 후, 삭제하기 버튼을 누르면 데이터베이스에 기록된 todo 정보가 삭제된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 todo를 삭제시키고자 할 때 유스케이스가 시작된다.	
	2	사용자는 스케줄을 삭제하기 위해 todo 를 수정하기 위한 버튼을 클릭한다.	
	3		시스템은 todo를 삭제하기 위한 팝업창을 보여준다.
	4	사용자는 스케줄을 삭제할 todo 의 제목을 선택하여 삭제하기 버튼을 클릭한다	
	5		시스템은 서버에 연결된 DB에서 해당 todo 레코드를 삭제한다
	6		시스템은 todo 삭제 성공 알림을 보여준다.

2.14* 유스케이스 "todo 확인" (UC-WM503)

관련 요구 사항	SFR-WM310		
액터	사용자		
목적	todo를 확인한다.		
개요	todo를 확인하기 위해 확인 버튼을 누르면 해당 정보가 서버와 연결된 데이터베이스에 저장되어 있던 값들이 불러와진다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 스케줄(todo)를 보고자 할 때 유스케이스가 시작된다.	
	2	사용자는 등록한 todo 를 보기 위한 버튼을 클릭한다.	
	3		시스템은 DB에 저장된 todo 를 보여준다.

2. 15* 유스케이스 "중요 일정 확인" (UC-WM600)

<table>
<tr><td>관련 요구사항</td><td colspan="3">SFR-WM311</td></tr>
<tr><td>액터</td><td colspan="3">사용자</td></tr>
<tr><td>목적</td><td colspan="3">스케줄과 todo 를 중요도 순서로 확인한다.</td></tr>
<tr><td>개요</td><td colspan="3">중요도 별 및 상단 고정된 일정을 확인하기 위해, DB에서 일정을 불러와 importance 화면에 중요도 순으로 표시한다.</td></tr>
<tr><td></td><td colspan="3"></td></tr>
<tr><td rowspan="4">주흐름
(대안 흐름 포함)</td><td>단계</td><td>액터</td><td>시스템</td></tr>
<tr><td>1</td><td>사용자가 일정을 확인하기 위해 showimportance 탭을 클릭하면 유스케이스가 시작된다.</td><td></td></tr>
<tr><td>2</td><td></td><td>시스템은 DB에서 일정을 불러와 중요도 순서로 정렬하고, 상단 고정된 일정을 별도로 분리하여 제일 앞에 위치시킨다.</td></tr>
<tr><td>3</td><td></td><td>시스템은 정렬한 일정을 화면에 표시한다.</td></tr>
</table>

2. 16* 유스케이스 "중요 화면에서 스케줄(task) 추가" (UC-WM700)

관련 요구사항	SFR-WM302, SFR-WM303, SFR-WM304, SFR-WM305, SFR-WM306, SFR-WM307		
액터	사용자		
목적	Importance_task를 추가한다.		
개요	importance_task를 추가하기 위해 필요한 정보를 입력한 후, 추가 버튼을 누르면 해당 정보가 서버와 연결된 데이터베이스에 저장된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 showimportance 화면에서 task를 추가하고자 할 때 유스케이스가 시작된다.	
	2		시스템은 showimportance 화면을 보여준다.
	3	사용자는 task를 추가하기 위한 버튼을 클릭한다.	
	4		시스템은 task를 추가하기 위한 팝업창을 보여준다.
	5	사용자는 task의 제목, 메모, 중요도, 시작 날짜, 마감 날짜에 해당하는 값을 입력한 후 추가 버튼을 클릭한다.	
	6		시스템은 서버에 연결된 DB에 해당 스케줄 정보를 저장한다.
	7		시스템은 스케줄 추가 성공 알림을 보여준다.

2. 17* 유스케이스 "중요 화면에서 스케줄(todo) 추가" (UC-WM701)

관련 요구사항	SFR-WM302, SFR-WM303, SFR-WM304, SFR-WM305, SFR-WM306, SFR-WM307		
액터	사용자		
목적	Importance_todo을 추가한다.		
개요	todo를 추가하기 위해 필요한 정보를 입력한 후, 추가 버튼을 누르면 해당 정보가 서버와 연결된 데이터베이스에 저장된다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 showimportance 화면에서 todo를 추가하고자 할 때 유스케이스가 시작된다.	
	2		시스템은 showimportance 화면을 보여준다.
	3	사용자는 todo를 추가하기 위한 버튼을 클릭한다.	
	4		시스템은 todo를 추가하기 위한 팝업창을 보여준다.
	5	사용자는 todo의 제목, 메모, 중요도, 시작 날짜, 마감 날짜에 해당하는 값을 입력한 후 추가 버튼을 클릭한다.	
	6		시스템은 서버에 연결된 DB에 해당 스케줄 정보를 저장한다.
	7		시스템은 스케줄 추가 성공 알림을 보여준다.

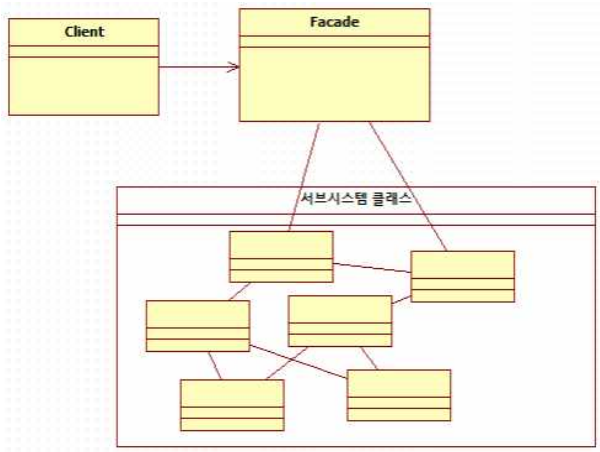
2. 18* 유스케이스 "TODO 배경화면 변경" (UC-WM800)

관련 요구사항	SFR-WM301		
액터	사용자		
목적	각각의 배경을 따로 변경할 수 있다.		
개요	배경 변경을 선택해서 showImportance, showTodo, showTask의 각 화면을 다르게 변경한다.		
주흐름 (대안 흐름 포함)	단계	액터	시스템
	1	사용자가 배경을 변경하기 위해 backgorund color를 클릭하면 유스케이스가 시작된다.	
	2		시스템은 배경화면 색깔 목록을 불러온다.
	3	사용자는 원하는 배경화면 색을 선택한다.	
	4		시스템은 선택된 배경화면 색을 화면에 표시한다.

3. 시스템 설계 및 구현

3.0.1 퍼사드 패턴(Facade Pattern)

퍼사드 패턴은 '건물의 정면'을 뜻하는 단어로 어떠한 소프트웨어의 다른 커다란 코드 부분에 대하여 간략화된 인터페이스를 제공해주는 디자인 패턴이다. 객체는 클라이언트의 요청이 발생했을 때, 서브시스템 내의 특정한 객체에 요청을 전달하는 역할을 하며, 복잡한 소프트웨어 바깥쪽의 코드가 라이브러리의 안쪽 코드에 의존하는 일을 감소시켜 주고, 복잡한 소프트웨어를 사용할 수 있게 간단한 인터페이스를 제공해준다.

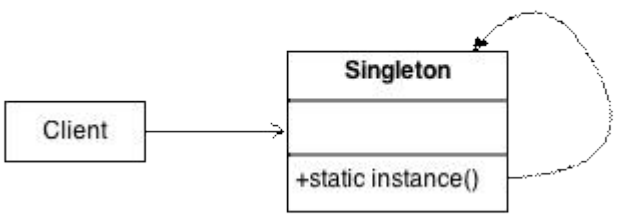


3.0.2 싱글톤 패턴 (Singleton Pattern)

생성자가 여러 차례 호출되더라도 실제로 생성되는 객체는 하나이고 최초 생성 이후에 호출된 생성자는 최초의 생성자가 생성한 객체를 리턴한다. 즉 해당 클래스의 인스턴스가 하나만 만들어지고, 그 인스턴스에 대한 전역 접근할 수 있게 하는 패턴이다. 인스턴스 단 한 개만 존재하는 것을 보증하는 디자인 패턴으로 주로 시스템 전반에 걸쳐서 특정한 자원 (Object, Moudel, Component) 이 공유될 때 사용된다.

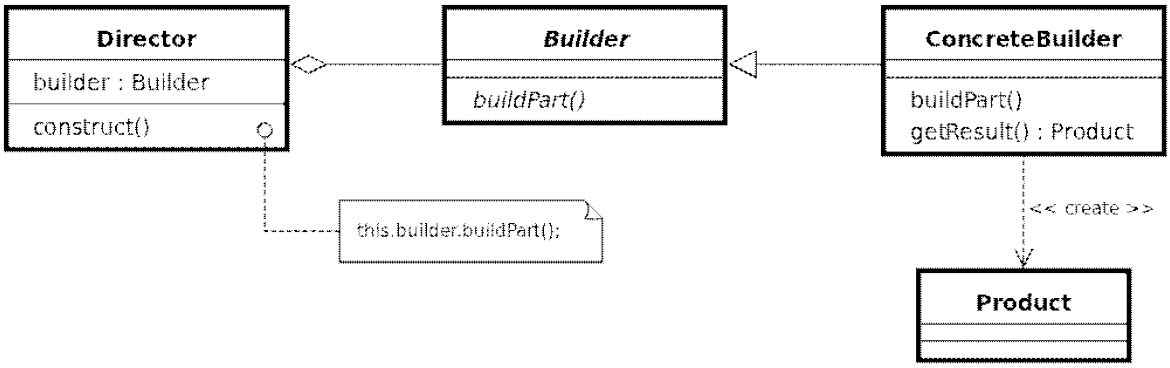
- 하나의 객체를 만들어 어느 곳에서나 접근할 수 있도록 함

규칙
1) private 멤버 변수로 자기 자신의 클래스의 인스턴스를 가짐
2) private 생성자를 지정, 외부에서 절대로 인스턴스를 생성하지 못하게 함
3) 생성된 인스턴스를 리턴하는 클래스 (static) 메서드 정의



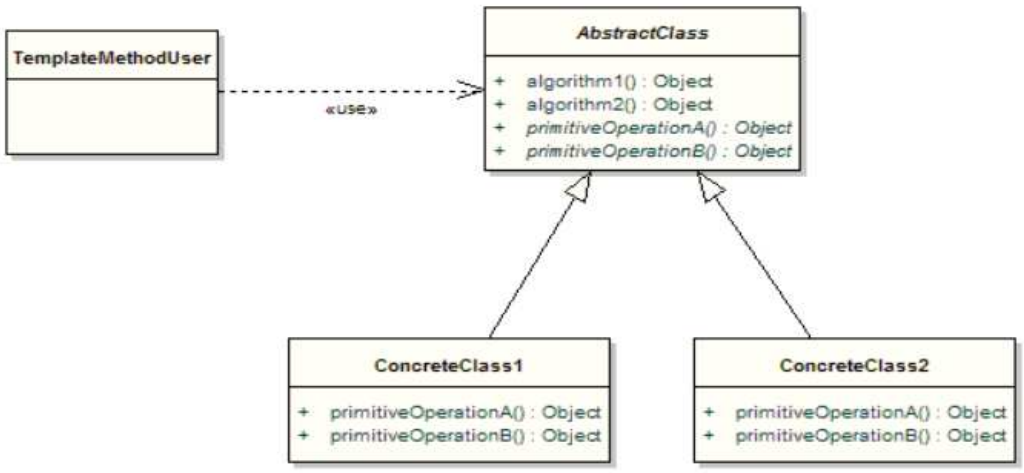
3.0.3 빌더 패턴 (Builder Pattern)

점층적 생성자 패턴은 매개 변수가 추가될수록 코드 수정이 복잡해지고, 가독성이 저하된다. 이를 개선하기 위해 고안된 자바빈 패턴은 1회의 호출로 객체 생성이 끝나지 않기 때문에 일관성이 깨지게 된다. 이러한 문제를 해결하기 위한 방법이 빌더 패턴이다. 빌더 패턴을 적용하면 한 번에 객체를 생성하기 때문에 객체 일관성이 깨지지 않으며, 각 인자가 어떤 의미인지 알아보기 쉽다. 즉 객체 생성에 필요한 파라미터의 의미를 코드 단에서 명확히 알 수 있어 가독성이 좋으며 생성에 필요한 파라미터가 추가될 때 마다 생성자 오버로딩을 하지 않아도 된다.



3.0.4 템플릿 매서드 패턴 (Template Method Pattern)

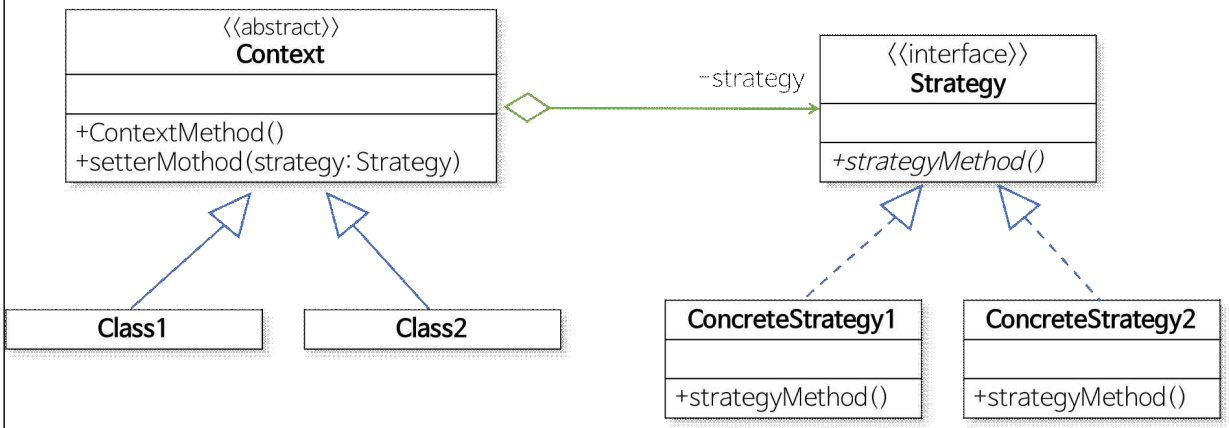
어떤 작업을 처리하는 일부분을 서브 클래스로 캡슐화하여 전체 일을 수행하는 구조는 바꾸지 않으면서 특정 단계에서 수행하는 내역을 바꾸는 패턴이다. 템플릿 메소드 패턴은 어떤 소스코드상의 알고리즘에서 특정 환경 또는 상황에 맞게 또는 변경을 해야 할 경우 유용하여 전체적으로 동일하면서 부분적으로는 다른 구문으로 구성된 메서드의 코드 중복을 최소화할 수 있다.. 추상클래스와 구현클래스로 작성할 수 있으며, 메인이 되는 로직 부분은 추상클래스의 일반 메소드로 선언해서 사용한다. 즉, 구현별로 달라질 수 있는 메소드들은 구현 클래스에서 선언 후 호출하는 방식으로 사용한다.



3.0.5 스트래티지 패턴(Strategy Pattern)

행위를 클래스로 캡슐화해 동적으로 행위를 자유롭게 바꿀 수 있게 해주는 패턴으로 같은 문제를 해결하는 여러 알고리즘이 클래스별로 캡슐화되어 있고 이들이 필요할 때 교체할 수 있도록 함으로써 동일한 문제를 다른 알고리즘으로 해결할 수 있게 하는 디자인 패턴이다. 즉, 전략을 쉽게 바꿀 수 있도록 해주는 디자인 패턴이다.

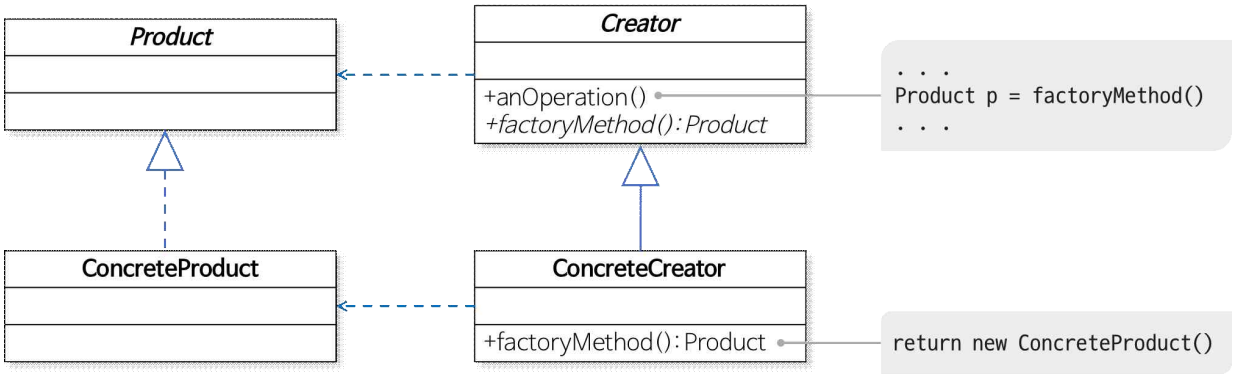
- 전략 : 어떤 목적을 달성하기 위해 일을 수행하는 방식, 비지니스 규칙, 문제를 해결하는 알고리즘 등



- Strategy
  인터페이스나 추상 클래스로 외부에서 동일한 방식으로 알고리즘을 호출하는 방법을 명시한다.
- ConcreteStrategy
  스트래티지 패턴에서 명시한 알고리즘을 실제로 구현한 클래스이다.
- Context
  스트래티지 패턴을 이용하는 역할을 수행한다.
  필요에 따라 동적으로 구체적인 전략을 바꿀 수 있도록 setter 메서드(집약 관계)를 제공한다.

3.0.6 팩토리 매서드 패턴 (Factory Method Pattern)

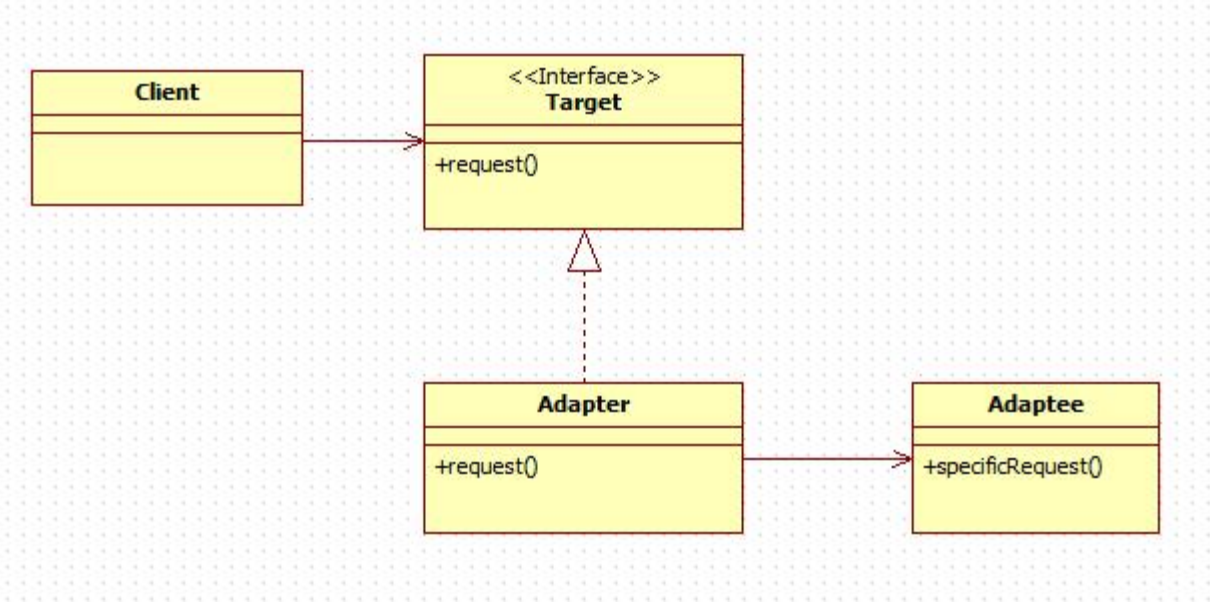
객체 생성 처리를 서브 클래스로 분리하여 처리하도록 캡슐화하는 패턴으로 객체의 생성 코드를 별도의 클래스/메서드로 분리함으로써 객체 생성의 변화에 대비하는데 유용하다. 팩토리 메서드 패턴의 팩토리 메서드는 객체를 생성해서 반환하는 것을 말하며 결과 값이 객체이다. 이를 사용하는 이유는 클래스의 생성과 사용의 처리로직을 분리하여 결합도를 낮추기 위함이다. 직접 객체를 생성해 사용하는 것을 방지하고 서브 클래스에 생성 로직을 위임함으로써 보다 효율적인 코드 제어를 할 수 있고 의존성을 제거한다.



- Product : 팩토리 메서드로 생성될 객체의 공통 인터페이스
- ConcreteProduct :구체적으로 객체가 생성되는 클래스
- Creator : 팩토리 메서드를 갖는 클래스
- ConcreteCreator : 팩토리 메서드를 구현하는 클래스로 ConcreteProduct 객체를 생성

3.0.7 어댑터 패턴 (Adapter Pattern)

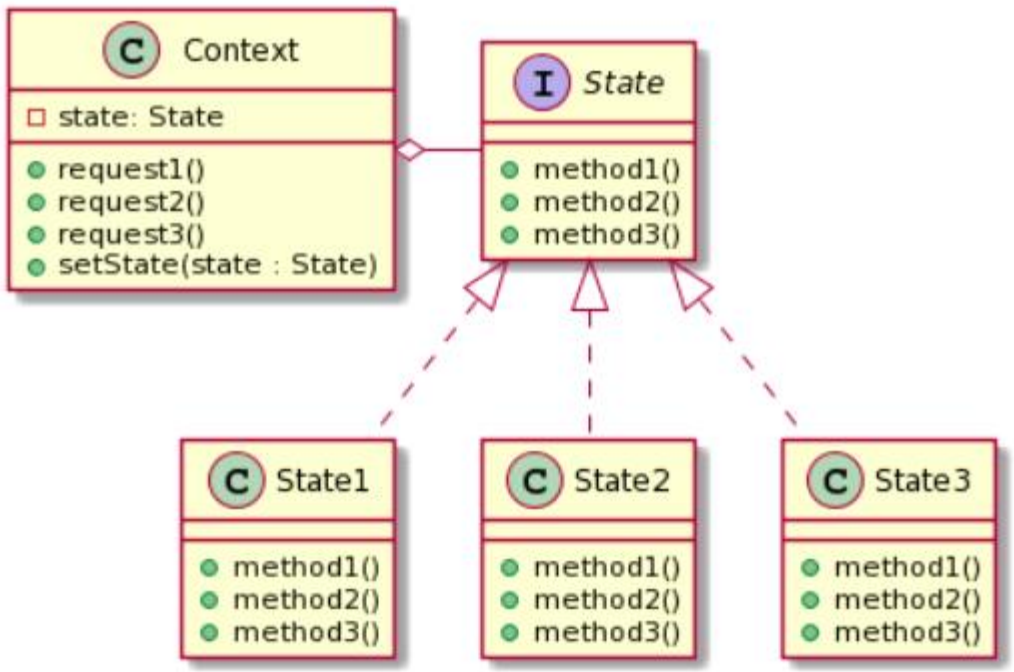
어댑터 패턴은 클래스의 인터페이스를 사용자가 기대하는 다른 인터페이스로 변환하는 패턴으로, 호환성이 없는 인터페이스 때문에 함께 동작할 수 없는 클래스들이 함께 동작하도록 해준다. 호환되지 않는 인터페이스를 사용하는 클라이언트를 그대로 활용할 수 있으며 클라이언트와 구현된 인터페이스를 분리하여 향후 인터페이스가 바뀌더라도 그 변경 내역은 어댑터에 캡슐화 되기 때문에 클라이언트는 바뀔 필요가 없어진다. 즉 기존 코드를 변경하지 않아도 되며 이 대문에 클래스 재활용성을 증가시킬 수 있다.



- Target : Adapter 가 구현하는 인터페이스이며 Client 는 Target 을 통해 Adaptee 기능을 사용
- Adapter : Adaptee 가 Target 에 호환될 수 있도록 연결해주는 역할
- Adaptee : Adapter 를 통해 Target 에 호환시킬 기능
- Client : Target 인터페이스를 사용함

3.0.8 스테이트 패턴 (State Pattern)

스테이트 패턴은 객체가 특정 상태에 따라 행위를 달리하는 상황에서 자신이 직접 상태를 체크하여 상태에 따라 행위를 호출하지 않고 상태를 객체화하여 상태가 행동을 할 수 있도록 위임하는 패턴이다. 객체의 특정 상태를 클래스로 선언하고, 클래스에는 해당 상태에서 할 수 있는 행위들을 메서드로 정의하여 각 상태 클래스들을 인터페이스로 캡슐화하여 클라이언트에서 인터페이스를 호출하는 방식이다.



- State : 시스템의 모든 상태에 공통 인터페이스를 제공한다.
  이 인터페이스를 실체화한 어떤 상태 클래스도 기존 상태 클래스를 대신해 교체해서 사용한다.
- State1, State2, State3 : Context 객체가 요청한 작업을 자신의 방식으로 실제 실행한다.
  대부분의 경우 다음 상태를 결정해 상태 변경을
  Context 객체에 요청하는 역할도 수행한다.
- Context : 여러 가지 내부 상태를 가질 수 있는 클래스
  request()가 호출되면 상태 객체에거 그 작업을 위임한다.

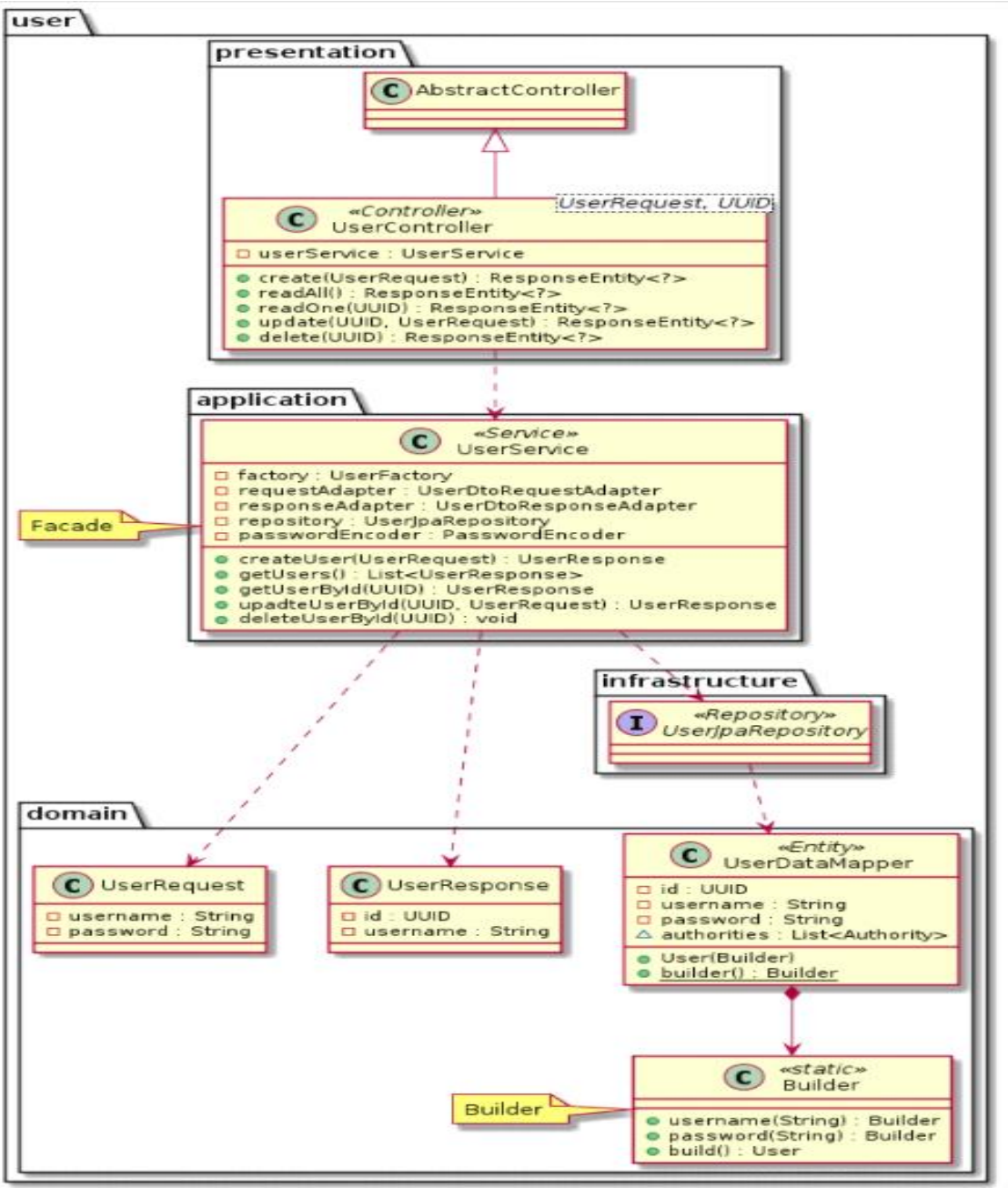
3.1 퍼사드 패턴 적용

3.1.1 문제점

2.1, 2.2, 2.3절에서 기술한 유스케이스 회원 가입과 회원 정보 수정 및 삭제를 할 시에 정보를 서버에 전송해야 한다. 해당 기능을 실행하기 위해서는 많은 과정들 또는 많은 서브 시스템을 거쳐야 하는데 사용자가 직접 하기엔 많은 번거로움과 어려움이 있다.

3.1.2 해결 방안

해당 문제를 해결 하기 위해 퍼사드 패턴을 적용하였다. 퍼사드 패턴을 이용하면 클라이언트가 요청을 할 때에 객체는 서브시스템 내의 특정한 객체에 요청만 하면 되고, 복잡한 소프트웨어를 사용할 수 있게 간단한 인터페이스를 제공함으로써 클라이언트가 보다 쉽게 사용할 수 있다.



3.1.3 관련 코드 설명

사용자가 회원 가입을 하려는 상황이라면 클라이언트에서 users 요청이 들어왔을 때, 여러 다른 로직을 직접 실행시키지 않고 create 함수를 실행시켜 UserService의 createUser 함수를 호출하도록 한다. createUser 함수는 회원 비밀번호를 암호화한 후 다른 정보와 캡슐화하여 데이터베이스에 저장한다. 만일 회원을 생성하는 내용을 바꿔야 할 일이 있더라도 클라이언트의 코드는 변화 없이 내용을 수정할 수 있다. 회원정보 검색, 수정, 삭제 역시 복잡한 과정을 알 필요 없이 수행할 수 있다.

UserController

```
@RequestMapping("/users")
@RepositoryRestController
public class UserController extends AbstractController<UserRequest, UUID> {

  private final UserService service;

  @Autowired
  public UserController(UserService service) {
    this.service = service;
  }

  @ResponseBody
  @PostMapping
  public ResponseEntity<?> create(@RequestBody final UserRequest request) {
    return ResponseEntity.ok(service.createUser(request));
  }

  @ResponseBody
  @GetMapping
  public ResponseEntity<?> readAll() {
    List<UserResponse> response = service.getUsers();

    return ResponseEntity.ok(response);
  }

  @ResponseBody
  @GetMapping("/{id}")
  public ResponseEntity<?> readOne(@PathVariable final UUID id) {
    UserResponse response = service.getUserById(id);

    return ResponseEntity.ok(response);
  }
```

UserService

```
@Transactional
public UserResponse createUser(final UserRequest newUser) {
  String hashedPassword = passwordEncoder.encode(newUser.getPassword());

  User user = factory.create(newUser.getUsername(), hashedPassword,
                             newUser.getFirstName(), newUser.getLastName(),
                             newUser.getPhoneNumber(), newUser.getEMail() );

  UserDataMapper dataMapper = requestAdapter.getEntity(user);
  dataMapper = repository.save(dataMapper);

  return responseAdapter.getEntity(dataMapper);
}

public List<UserResponse> getUsers() {
  return repository.findAll().stream()
    .map(responseAdapter::getEntity)
    .collect(Collectors.toList());
}

public UserResponse getUserById(UUID id) {
  return repository.findById(id)
    .map(responseAdapter::getEntity)
    .orElseThrow(() -> new RuntimeException(String.format("User id %s is none", id)));
}
```

3.2 싱글톤 패턴 적용

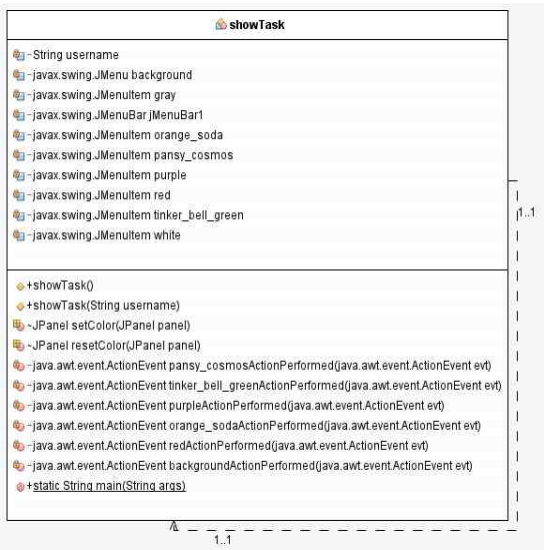
3.2.1 문제점

2.18에서 기술한 각 화면에 따른 배경화면 변경 유스케이스를 사용하려면 개별적으로 소스코드가 적용 되어야 한다. 만약 다른 각각의 화면의 배경색을 바꾸려고 했는데 모두가 똑같이 적용이 된다면 원하는 것처럼 개별의 배경색을 지정해 줄 수 없다.

3.2.2 해결 방안

해당 문제를 해결하기 위하여 싱글톤 패턴을 사용하였다. 사용자가 생성자를 여러 차례 호출하여 배경을 여러번 바꾸더라도 하나의 생성자가 리턴되기 때문에 각 화면에 원하는 배경색으로 변경할 수 있다.

▲ **showTodo 싱글톤**



▲ **showTask 싱글톤**

▲ **showImportance 싱글톤**

3.2.3 관련 코드 설명

static이나 instance를 사용하여 적용을 해야 하지만 JFrame 내에 자바 스윙을 이용하기 때문에 스윙을 기준으로 패턴을 적용했다. JMenuBar라는 백그라운드 컬러 메뉴를 만든다. 그리고 하위 선택지들로 컬러를 선택할 수 있게 해놓았다. 사용자가 직접 숫자를 입력하지 않고 마우스 클릭만으로 원하는 각 화면에 따라 배경색을 바꿀 수 있다.

```
private void pansy_cosmosActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    panel2.setBackground(new Color(245, 208, 221));
}

private void tinker_bell_greenActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    panel2.setBackground(new Color(191, 218, 131));
}

private void purpleActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    panel2.setBackground(new Color(153, 108, 209));
}

private void orange_sodaActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    panel2.setBackground(new Color(249, 194, 112));
}

private void redActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    panel2.setBackground(new Color(224, 106, 78));
}

private void backgroundActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
```

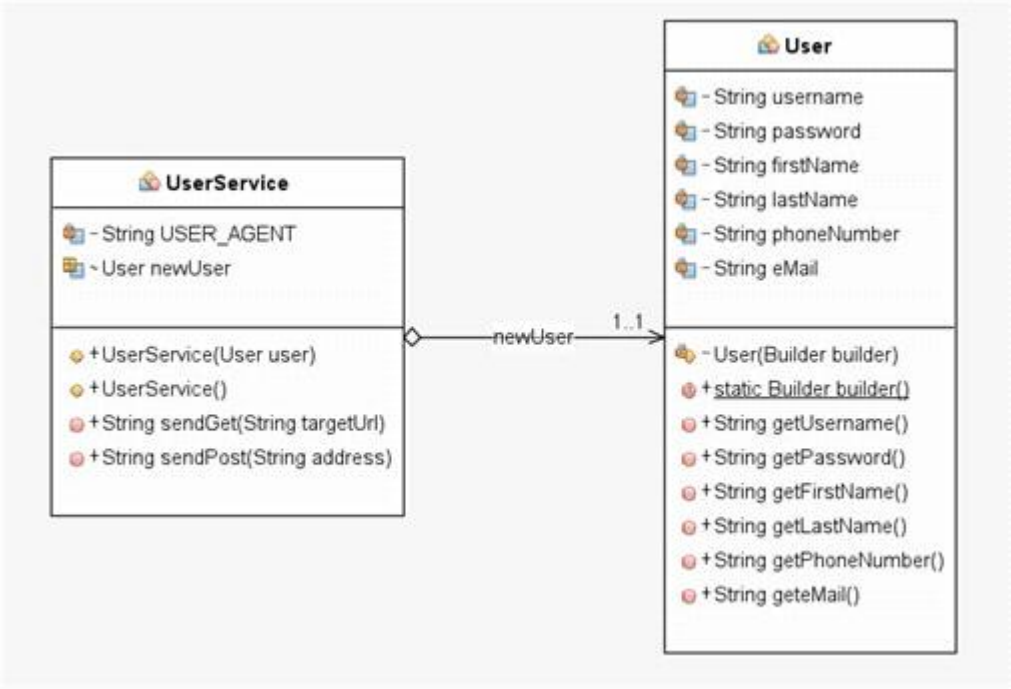
3.3 빌더 패턴 적용

3.3.1 문제점

2.1, 2.3절에서 기술한 유스케이스에서 회원 가입과 회원 정보 수정 시, 사용자의 정보를 서버로 전송하여 데이터베이스에 기록해야 한다. userService에서 사용자의 정보를 서버로 전송하기 위해 User의 값을 불러와야 하는데 필요한 매개 변수를 불러오기 위해 모든 생성자를 정의하기에는 코드가 복잡해질 수 있고, 입력 값이 추후에 추가된다면 매개 변수의 의미 오역에 의한 오류가 발생할 수도 있다.

3.3.2 해결 방안

해당 문제를 해결하기 위하여 빌더 패턴을 사용하였다. 사용자 객체를 생성할 때, 일일이 값을 입력하여 생성하지 않고 빌더를 통해 한 번에 객체를 생성하여 데이터의 일관성을 유지하고, 가독성을 높여 매개변수에 값이 잘못 들어가 오류가 생길 상황을 방지하였다.



3.3.3 관련 코드 설명

사용자 정보를 객체화하여 다루기 위해 사용하는 User 클래스이다. Builder를 통해 사용자 정보를 알아보기 쉽게 객체화할 수 있다. 또한, 사용자가 입력해야 할 값이 추가되거나 선택적인 입력 값이 생길 경우에도 유연하게 변수를 추가할 수 있다. 아래 코드 중 Register 클래스에서 회원가입을 위해 정보를 서버에 보내기 위해 사용자 정보를 build로 객체화한 것을 볼 수 있다.

User 클래스

```
public class User {

    private final String username;
    private final String password;
    private final String firstName;
    private final String lastName;
    private final String phoneNumber;
    private final String eMail;

    public static class Builder {

        private String username;
        private String password;
        private String firstName;
        private String lastName;
        private String phoneNumber;
        private String eMail;

        public Builder(String username) {
            this.username = username;
        }

        public Builder(String username, String password, String firstName, String lastName, String phoneNumber, String eMail) {
            this.username = username;
            this.password = password;
            this.firstName = firstName;
            this.lastName = lastName;
            this.phoneNumber = phoneNumber;
            this.eMail = eMail;
        }

        public Builder username(String value) {
            username = value;
            return this;
        }

        public Builder password(String value) {
            password = value;
            return this;
        }
```

Register 클래스

```
if(!password.getText().equals(confirm_password.getText())) {
            confirm_password.setText("");
            confirm_password.requestFocus();
            return "pm";
        } else {
            User user = User.builder()
            .username(user_name.getText())
            .password(password.getText())
            .firstName(first_name.getText())
            .lastName(last_name.getText())
            .phoneNumber(phone_number.getText())
            .eMail(e_mail.getText())
            .build();

            UserService service = new UserService(user);
```

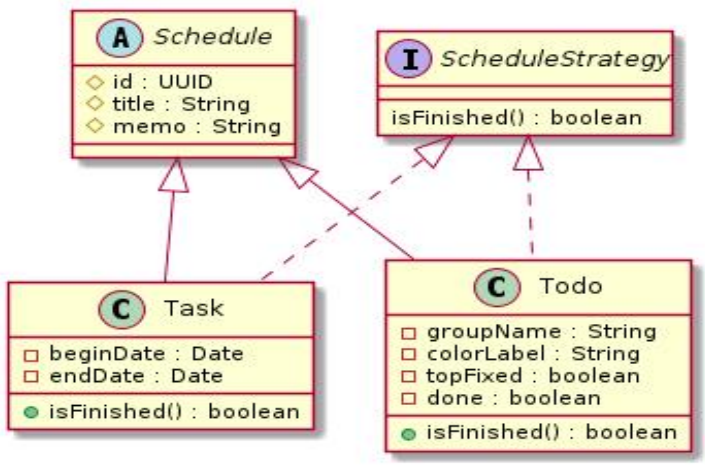
3.4 스트레티지 패턴 적용

3.4.1 문제점

2.12절에서 기술한 유스케이스에서 Todo 일정을 변경할 때, 사용자는 해당 일정이 끝났음을 체크할 수 있다. Todo 일정과 Task 일정은 일정이 끝난 경우를 지정해주어야 한다. 그러나 Todo는 사용자가 해당 일정을 끝냈다고 done을 체크 해주어야 일정이 끝나고, Task는 사용자가 입력했던 기간 중 endDate가 오늘 날짜를 지난 경우 일정이 끝이난다. 만일 일정의 끝을 체크하는 기준이 달라지거나 새로운 알고리즘이 생긴다면 해당 알고리즘을 슈퍼 클래스에 생성하여 상속의 방법으로 해결하기에는 의도치 않은 클래스에도 알고리즘이 적용되는 문제가 발생할 수 있다.

3.4.2 해결방안

상속으로 인해 발생할 수 있는 문제를 해결하기 위해 스트레티지 패턴을 적용하여 시스템에서 달라지는 부분을 찾아 분리시켰다. Schedule에서는 일정이 끝났는지를 확인하는 isFinished가 바뀌는 부분이기 때문에 달라지지 않는 부분으로부터 분리시켜, ScheduleStrategy 인터페이스에 세팅하여 구현하였다.



3.4.3 관련 코드 설명

Schedule 클래스에는 id, title, memo가 들어있고, 일정이 끝났는지 확인해주는 알고리즘을 ScheduleStrategy 인터페이스에서 상속받는다.

Task 클래스에는 endDate가 오늘 날짜를 지난 경우 일정이 끝나는 알고리즘을 적용하고, Todo 클래스에는 사용자가 입력한 done의 bool 값으로 일정을 끝내는 알고리즘을 적용한다.

```
@startuml strategy

abstract class Schedule {
  #id : UUID
  #title : String
  #memo : String
}

interface ScheduleStrategy {
  isFinished() : boolean
}

class Task extends Schedule implements ScheduleStrategy {
  -beginDate : Date
  -endDate : Date
  +isFinished() : boolean
}

class Todo extends Schedule implements ScheduleStrategy {
  -groupName : String
  -colorLabel : String
  -topFixed : boolean
  -done : boolean
  +isFinished() : boolean
}


@enduml
```

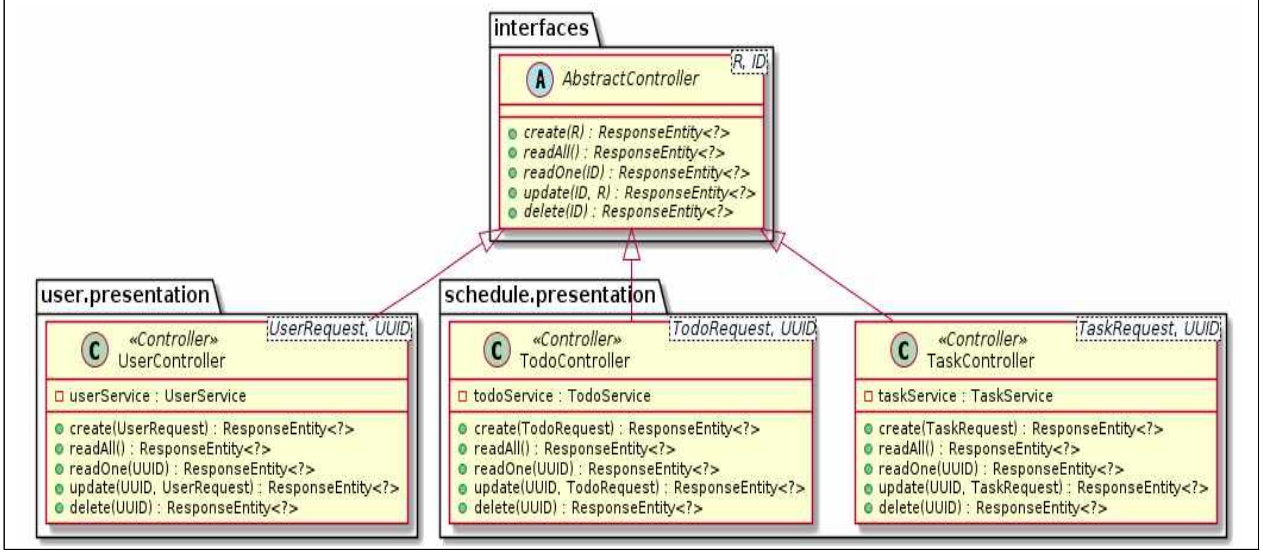
3.5 템플릿 매서드 패턴 적용

3.5.1 문제점

2.1, 2.7, 2.11절에 기술한 유스케이스를 보면 사용자(User), Todo일정, Task일정마다 CRUD 처리가 필요한 것을 확인할 수 있다. 해당 기능을 담당하는 세 가지의 Controller는 세부적인 곳에서는 차이가 있지만 전체적인 틀이 비슷하기 때문에 코드의 중복이 발생하고 있고, CRUD 처리가 필요한 새로운 카테고리가 생길 경우마다 코드가 반복될 것이다.

3.5.2 해결방안

동일한 구조를 가지는 Controller의 중복되는 코드를 줄이기 위해 일정한 틀을 AbstractController에 갖추어두고, 해당 추상 클래스를 상속받아 UserController, TodoController, TaskController에서 각 역할에 맞게 사용한다.



3.5.3 관련 코드 설명

AbstactController를 생성하여 CRUD 처리에 필요한 메소드를 정의한다. user, todo, task 외에 데이터를 저장할 카테고리가 추가된다면 AbstractController 클래스를 상속받아서 CRUD 처리 메소드를 쉽게 사용할 수 있고, 중복되는 코드를 줄일 수 있다.

user

사용자와 관련된 데이터의 CRUD를 위한 UserController이다. AbstractController를 상속받아 해당 메소드를 사용하여 UserService를 처리한다.

```
package interfaces {
  abstract class AbstractController<R, ID> {
    {abstract} +create(R) : ResponseEntity<?>
    {abstract} +readAll() : ResponseEntity<?>
    {abstract} +readOne(ID) : ResponseEntity<?>
    {abstract} +update(ID, R) : ResponseEntity<?>
    {abstract} +delete(ID) : ResponseEntity<?>
  }
}
```

schedule

Task와 Todo 역시 데이터를 처리하기 위해 AbstractController를 상속받아서 CRUD와 관련된 메소드를 각 역할에 맞게 사용한다.

```
package user.presentation {
  class UserController<UserRequest, UUID> <<Controller>>  extends AbstractController {
    -userService : UserService
    +create(UserRequest) : ResponseEntity<?>
    +readAll() : ResponseEntity<?>
    +readOne(UUID) : ResponseEntity<?>
    +update(UUID, UserRequest) : ResponseEntity<?>
    +delete(UUID) : ResponseEntity<?>
  }
}
```

```
package schedule.presentation {
  class TaskController<TaskRequest, UUID> <<Controller>>  extends AbstractController {
    -taskService : TaskService
    +create(TaskRequest) : ResponseEntity<?>
    +readAll() : ResponseEntity<?>
    +readOne(UUID) : ResponseEntity<?>
    +update(UUID, TaskRequest) : ResponseEntity<?>
    +delete(UUID) : ResponseEntity<?>
  }

  class TodoController<TodoRequest, UUID> <<Controller>>  extends AbstractController {
    -todoService : TodoService
    +create(TodoRequest) : ResponseEntity<?>
    +readAll() : ResponseEntity<?>
    +readOne(UUID) : ResponseEntity<?>
    +update(UUID, TodoRequest) : ResponseEntity<?>
    +delete(UUID) : ResponseEntity<?>

  }
}
```

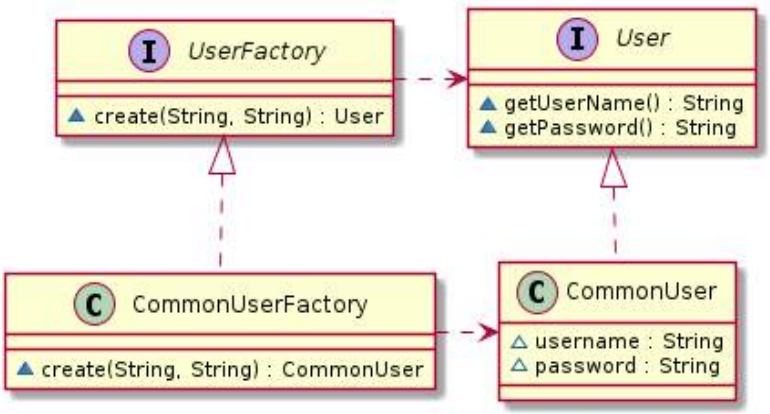
3.6 팩토리 매서드 패턴 적용

3.6.1 문제점

2.1절에 기술한 유스케이스를 보면 사용자가 회원 가입을 하였을 때, 관련 정보가 데이터베이스에 저장되어야 한다. 서버에서 받은 회원 정보를 데이터베이스에 저장하기 위해 구상 클래스를 바탕으로 정보를 객체화하면 새로운 구상 클래스가 추가될 때마다 코드를 고쳐야 한다.

3.6.2 해결 방안

팩토리 메서드 패턴을 사용하여 직접적으로 객체를 생성하지 않고 팩토리 메서드를 통해 위임하여 클래스 간의 결합도를 낮춘다. 객체를 생성하는 코드 부분을 분리 시켰기 때문에 객체를 추가/수정이 일어나더라도 객체를 생성하는 코드만 건들면 되므로 클래스에 변경이 생겼을 때 다른 클래스 영향을 덜 주기 때문이다. 그리고 객체생성 코드를 전부 하나의 객체 또는 메소드에 구현하기 때문에 코드에 중복되는 내용을 제거할 수 있고, 수정에도 용이하다.



3.6.3 관련 코드 설명

CommonUser 클래스는 User 인터페이스를 상속받아 회원 정보와 관련된 객체를 생성한다. CommonUserFactory 클래스는 UserFactory 인터페이스를 상속받아 create 메서드를 오버라이드한다. 반환 값으로 CommonUser의 객체를 생성하여 반환한다. 회원 정보로 비밀번호, 이메일과 같은 새로운 값이 추가될 경우 객체를 생성하는 CommonUser를 수정하면 된다.

```
interface User {
  ~getUserName() : String
  ~getPassword() : String
}

interface UserFactory {
  ~create(String, String) : User
}

class CommonUser implements User {
  ~username : String
  ~password : String
}

class CommonUserFactory implements UserFactory {
  ~create(String, String) : CommonUser
}

UserFactory .> User
CommonUserFactory .> CommonUser
```

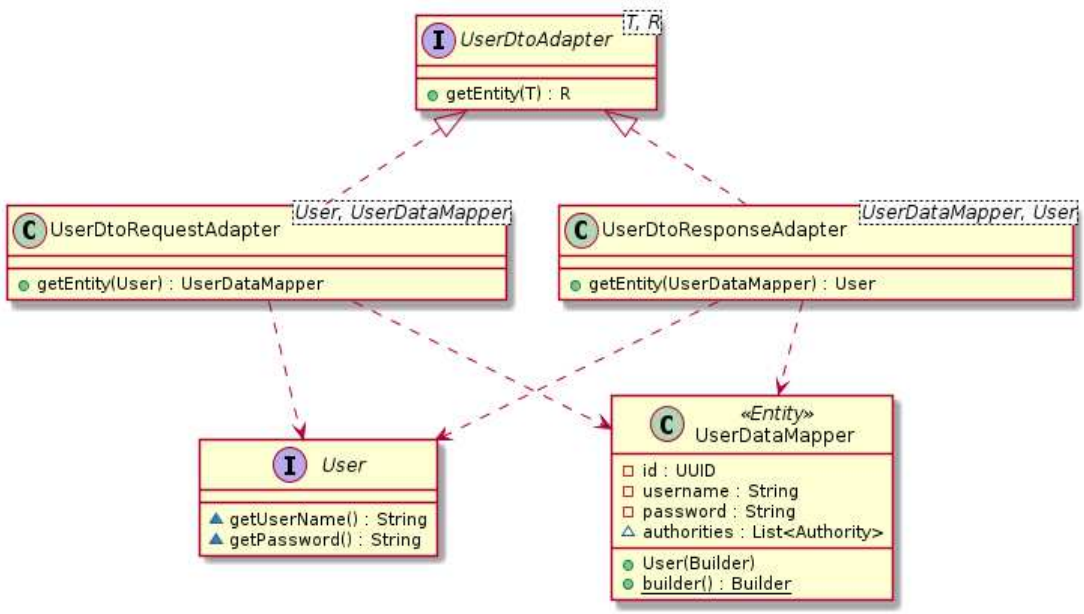
3.7 어댑터 패턴 적용

3.7.1 문제점

2.1절에 기술한 유스케이스를 보면 사용자가 회원 가입을 하였을 때, 관련 정보가 데이터베이스에 저장되어야 한다. 이 때, user 객체를 클라이언트가 기대하는 인터페이스로 변환하여 사용할 수 있게 하려면 Target 인터페이스와 호환되도록 클래스를 수정해야 하지만, 비효율적이고 오류를 일으키는 원인이 될 수 있다.

3.7.2 해결 방안

Adaptee 클래스가 Target 인터페이스와 호환되도록 어댑터 패턴을 적용하여 사이에 UserDtoRequestAdapter와 UserDtoResponseAdapter를 두었다.



3.7.3 관련 코드 설명

UserDtoAdapter 인터페이스를 생성하고, UserDtoRequestAdapter와 UserDtoResponseAdapter를 해당 인터페이스에서 상속받는다. UserDtoRequstAapter는 타겟인 User에 맞추어 객체를 변환하여 주고, UserDtoResponseAdapter는 타겟인 UserDataMapper에 맞추어 객체를 반환한다.

UserDtoAdapter

```
public interface UserDtoAdapter<T, R> {

  R getEntity(T user);
}
```

UserDtoRequestAdapter

```
public class UserDtoRequestAdapter implements UserDtoAdapter<User, UserDataMapper> {

  @Override
  public UserDataMapper getEntity(User user) {
    return UserDataMapper.builder()
      .username(user.getUsername())
      .password(user.getPassword())
      .firstName(user.getFirstName())
      .lastName(user.getLastName())
      .phoneNumber(user.getPhoneNumber())
      .eMail(user.getEMail())
      .build();
  }
}
```

UserDtoResponseAdapter

```
public class UserDtoResponseAdapter implements UserDtoAdapter<UserDataMapper, UserResponse> {

  @Override
  public UserResponse getEntity(UserDataMapper user) {
    return new UserResponse(user.getId(), user.getUsername(),
                            user.getFirstName(), user.getLastName(),
                            user.getPhoneNumber(), user.getEMail());
  }
}
```

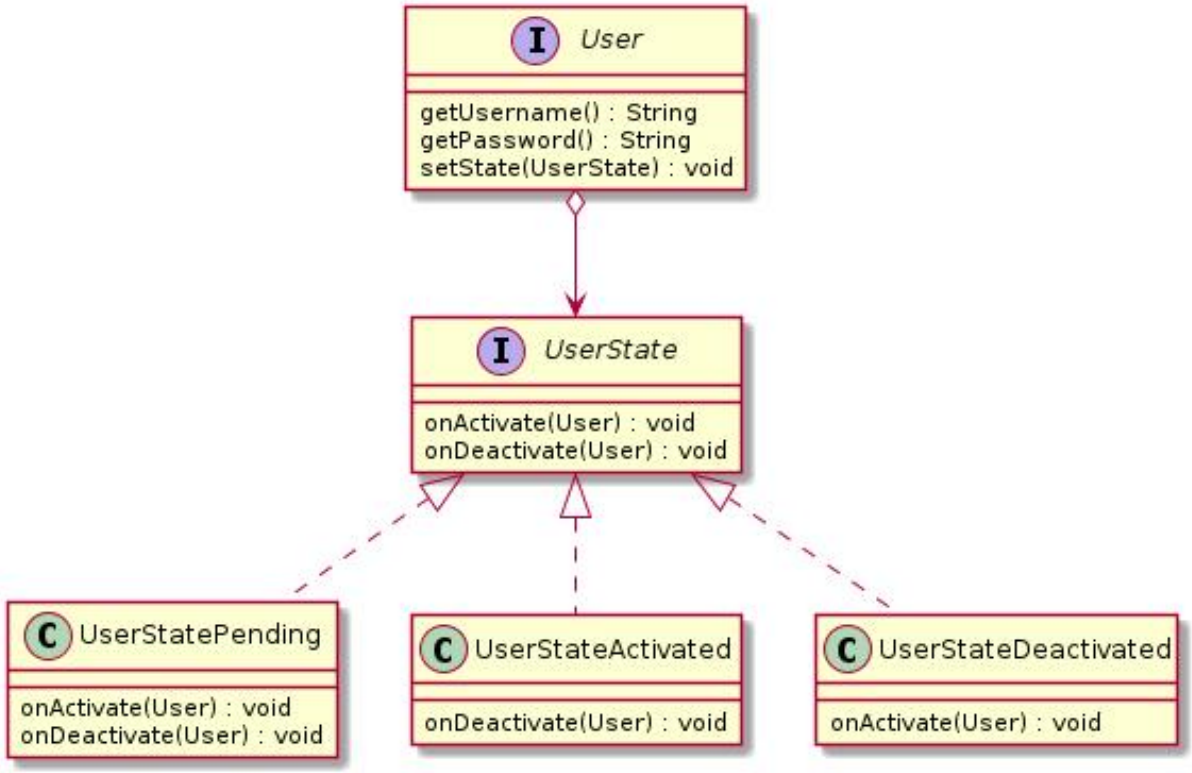
3.8 스테이트 패턴 적용

3.8.1 문제점

2.1절에 기술한 유스케이스를 보면 사용자 정보를 데이터베이스에 기록할 수 있어야 한다. 이때, 사용자의 상태에 따라 객체를 다르게 처리해야 할 일이 생길 경우 if문이나 switch문을 사용하여 처리할 경우 신규 상태가 발생 했을 때 코드를 전부 수정해야 하는 문제가 생긴다.

3.8.2 해결 방안

객체의 상태 정보를 저장하기 위해 state 패턴을 적용하였다. user 객체가 활동 중인지, 활동 중이지 않은지, 그리고 보류 중인지의 상태에 따라 처리를 할 수 있다.



3.8.3 관련 코드 설명

UserState 인터페이스

```
public interface UserState {

  void onActivate(User user);

  void onDeactivate(User user);
}
```

UserState 인터페이스를 생성하고 onActivate와 onDeactivate 메소드를 추가한다.

UserStateActivated 클래스

```
public class UserStateActivated implements UserState {

  @Override
  public void onActivate(User user) {
  }

  @Override
  public void onDeactivate(User user) {
    user.setState(new UserStateDeactivated());
  }
}
```

활동 중인 객체를 비활동 중인 상태로 설정한다.

UserStateDeactivated 클래스

```
public class UserStateDeactivated implements UserState {

  @Override
  public void onActivate(User user) {
    user.setState(new UserStateActivated());
  }

  @Override
  public void onDeactivate(User user) {
  }
}
```

비활동 중인 객체를 활동 중인 상태로 설정한다.

UserStatePending 클래스

```
public class UserStatePending implements UserState {

  @Override
  public void onActivate(User user) {
    user.setState(new UserStateActivated());
  }

  @Override
  public void onDeactivate(User user) {
    user.setState(new UserStateDeactivated());
  }
}
```

UserState를 상속받은 UserStatePending 클래스는 user 객체를 대기 상태로 설정한다.

4. 시스템 테스트

4.1 퍼사드 패턴 테스트

4.1.1 관련 테스트 코드

회원 가입, 정보 수정, 탈퇴 기능을 할 때 사용자는 간단한 인터페이스를 통해 복잡한 로직과 연결 과정을 알 필요 없이 원하는 기능을 실행할 수 있는지 확인한다.

- UserController

```
@ResponseBody
@PostMapping
public ResponseEntity<?> create(@RequestBody final UserRequest request) {
  return ResponseEntity.ok(service.createUser(request));
}

@ResponseBody
@PostMapping("/patch/userInfo/{id}")
public ResponseEntity<?> updateInfo(@PathVariable final UUID id,
  @RequestBody final UserRequest request) {
  UserResponse response = service.updateUserInfoById(id, request);

  return ResponseEntity.ok(response);
}
@ResponseBody
@GetMapping("/delete/{id}")
public ResponseEntity<?> delete(@PathVariable final UUID id) {
  service.deleteUserById(id);

  return ResponseEntity.noContent().build();
}
```

- UserService

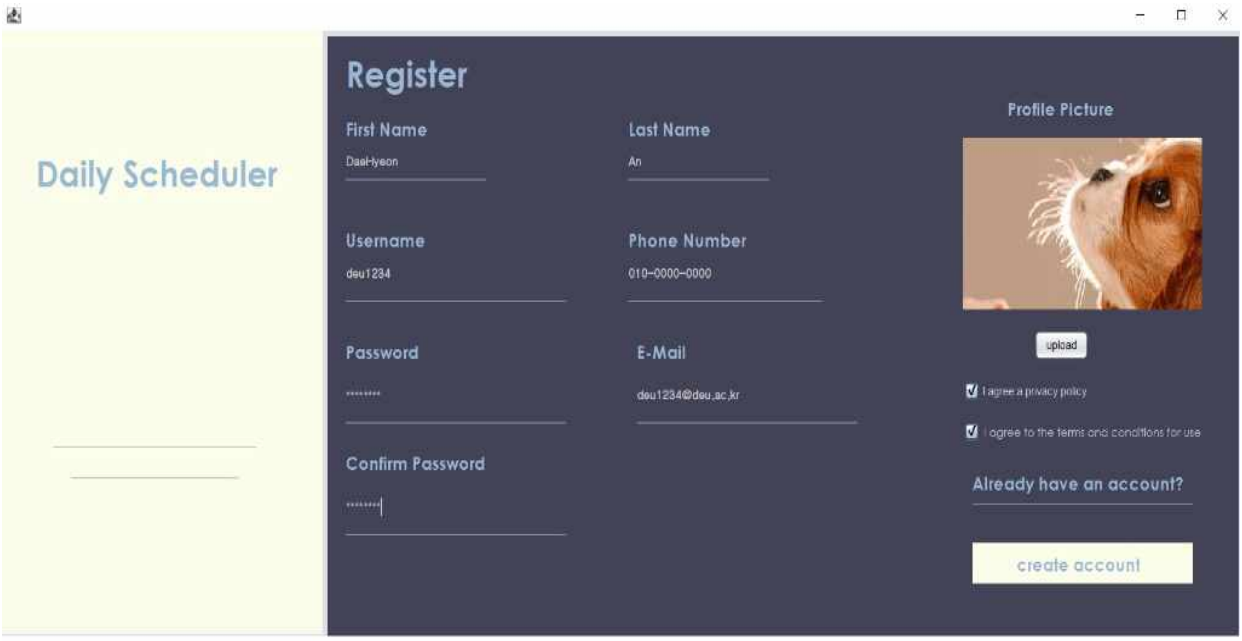
```
@Transactional
public UserResponse createUser(final UserRequest newUser) {
  String hashedPassword = passwordEncoder.encode(newUser.getPassword());

  User user = factory.create(newUser.getUsername(), hashedPassword,
                             newUser.getFirstName(), newUser.getLastName(),
                             newUser.getPhoneNumber(), newUser.getEMail() );

  UserDataMapper dataMapper = requestAdapter.getEntity(user);
  dataMapper = repository.save(dataMapper);

  return responseAdapter.getEntity(dataMapper);
}
```

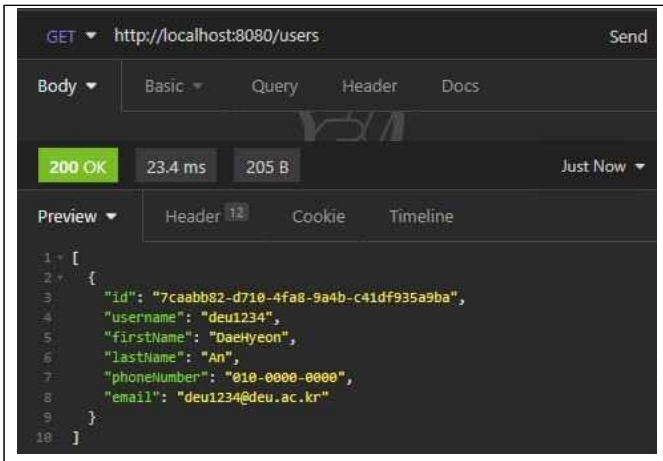
```java
@Transactional
public UserResponse updateUserInfoById(UUID id, UserRequest newUser) {
  UserDataMapper updatedUser = repository.findById(id)
    .map(user -> {
      user.setPassword(passwordEncoder.encode(newUser.getPassword()));
      user.setFirstName(newUser.getFirstName());
      user.setLastName(newUser.getLastName());
      user.setPhoneNumber(newUser.getPhoneNumber());
      user.setEMail(newUser.getEMail());
      return repository.save(user);
    }).orElseThrow(() -> new RuntimeException(String.format("User id %s is none", id)));

  return responseAdapter.getEntity(updatedUser);
}
```

```java
@Transactional
public void deleteUserById(UUID id) {
  repository.deleteById(id);
}
```

4.1.2 테스트 실행 결과

- 회원 가입 화면

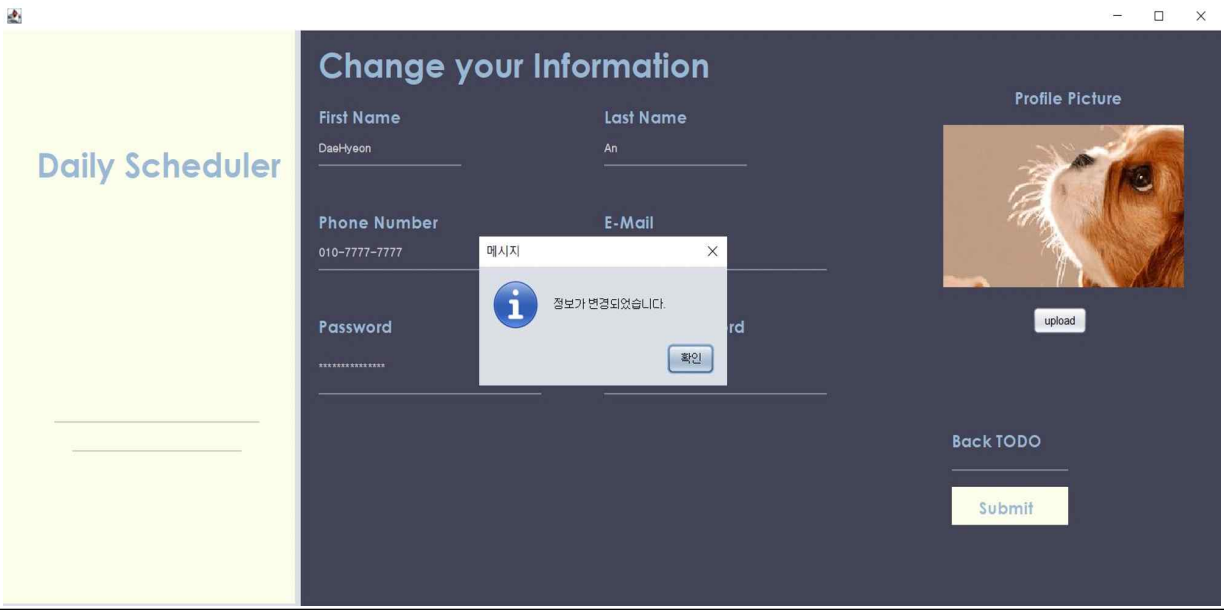


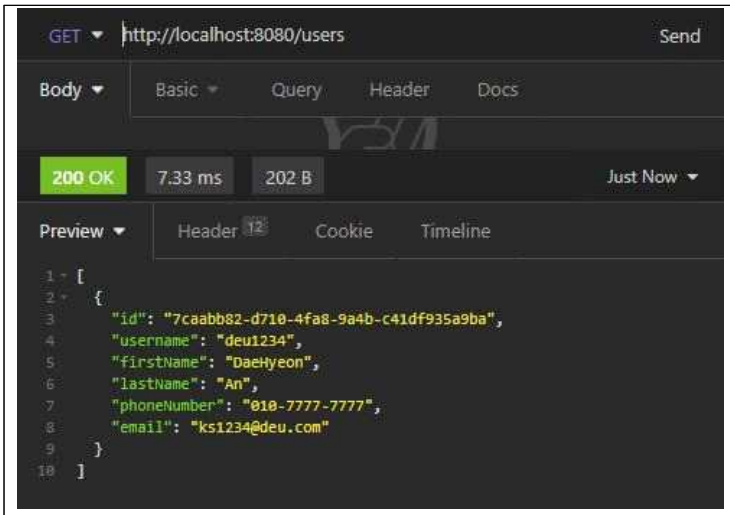


GUI에서 회원 정보를 입력하고, 'create account' 버튼을 누르면 사용자는 자신의 정보가 어떤 복잡한 과정을 거쳐서 데이터베이스에 저장되는지 알 필요 없이 계정 추가 기능을 사용할 수 있다.

Insomnia에서 http://localhost:8080/users의 주소로 GET 요청을 보낸 경우, 데이터베이스에 저장된 회원정보가 반환된다. 식별자로 UUID를 자동 생성하여 같이 출력하도록 하였고, password는 보안상의 이유로 출력하지 않도록 했다.

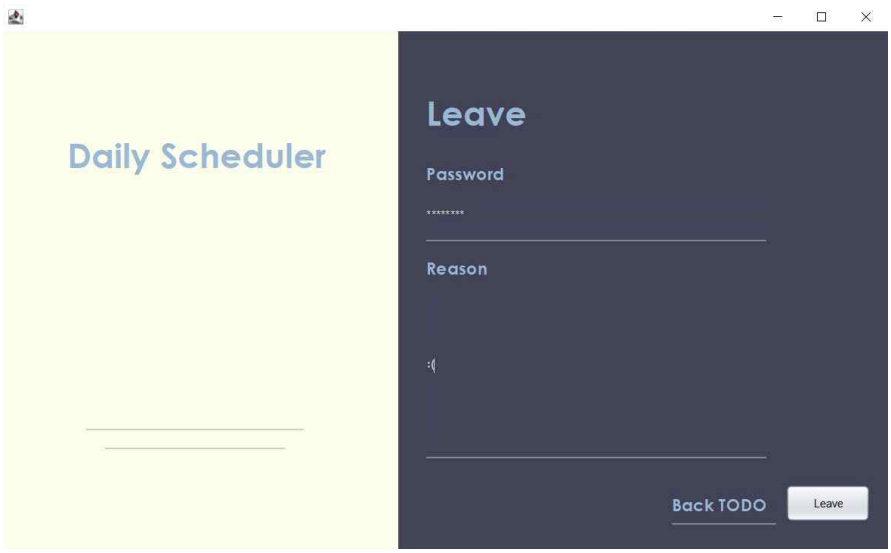
- 회원 정보 수정

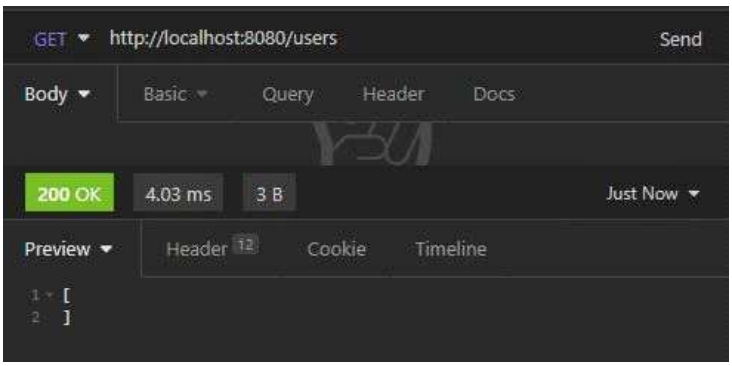




회원 정보를 변경할 때, 사용자는 자신의 정보를 일일이 찾아서 DB 값을 바꿔줄 필요 없이 수정 기능을 이용할 수 있다.

- **탈퇴**





비밀번호를 입력하고 Leave 버튼을 클릭하면 해당 회원 정보가 삭제된다.

4.2 싱글톤 패턴 테스트

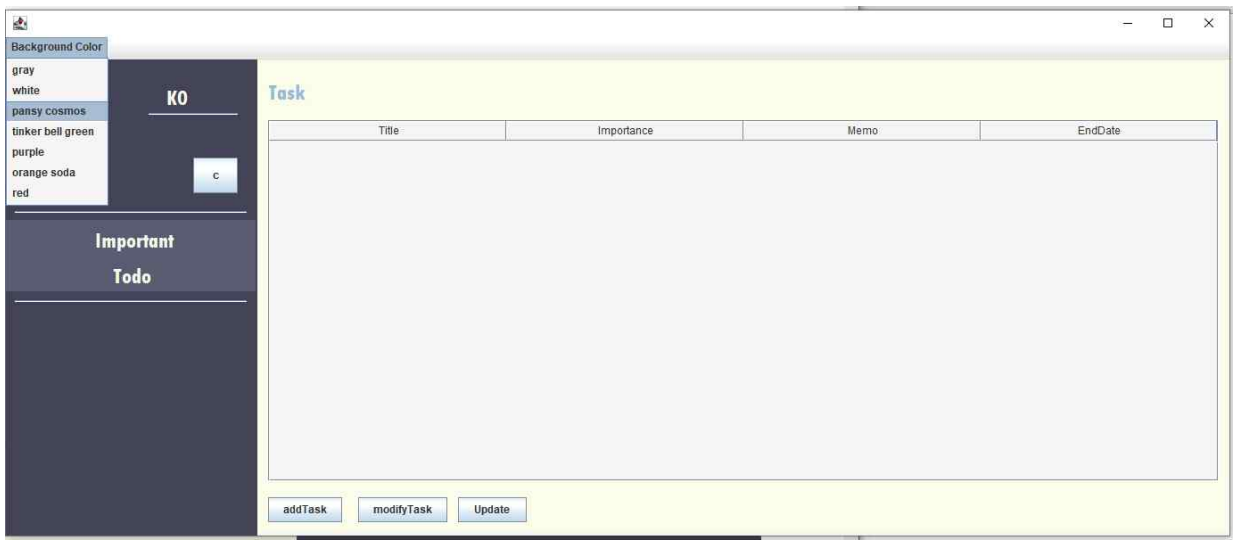
4.2.1 관련 테스트 코드

```
private void purpleActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    panel2.setBackground(new Color(153, 108, 209));
    System.out.println("Importance 보라색으로 화면 변경 성공!");
}
```

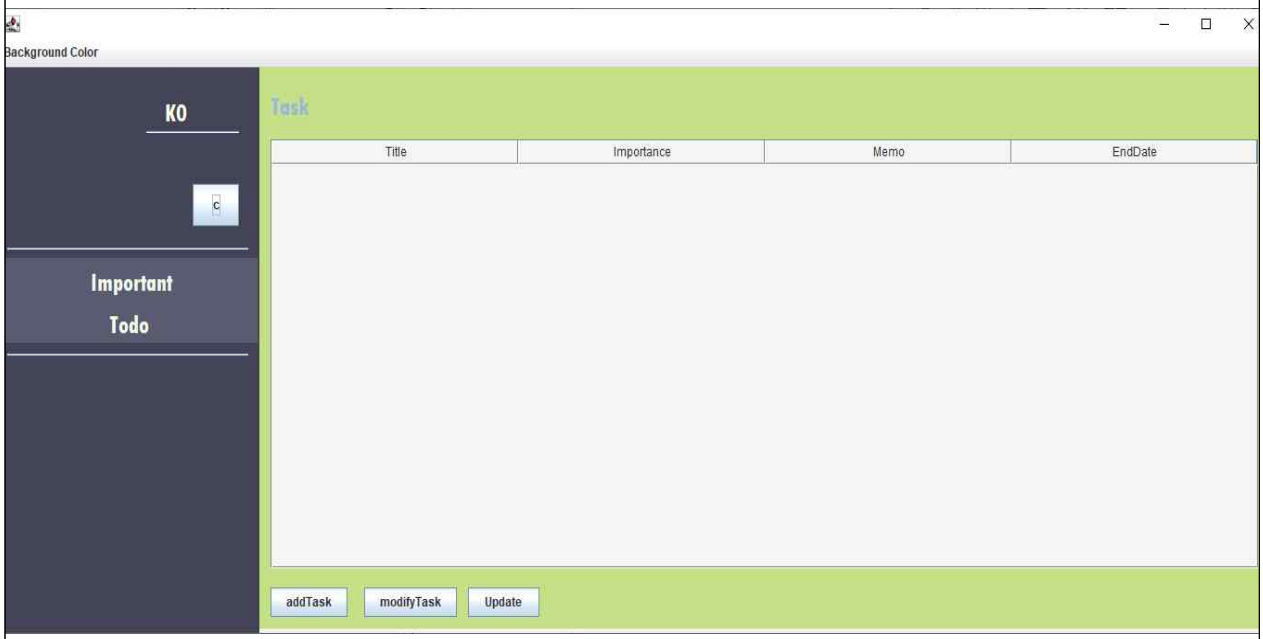
눈으로 바로 확인할 수 있지만 잘 돌아가는 것을 확인하기 위해 임의로 println을 importance의 purpleActionPerformed에 넣어 주었다.

4.2.2 테스트 실행 결과

- 색깔 선택 메뉴 바



- 선택지에 따른 바뀐 배경 화면



- 테스트 코드 확인

] --- exec-maven-plugin:3.0.0:exec (default-cli) @ daily-scheduler ---
. Importance 보라색으로 화면 변경 성공!

4.3 빌더 패턴 테스트

4.3.1 관련 테스트 코드

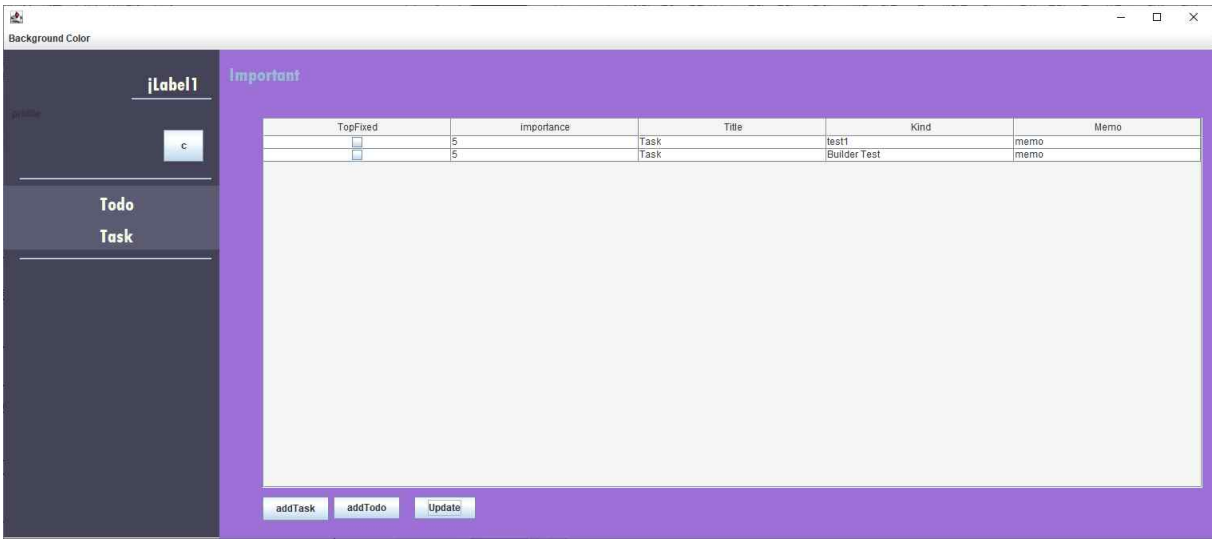
```
SimpleDateFormat df = new SimpleDateFormat("yyyyMMddHHmmss");
String start = df.format(startDate.getDate()) + startHour.getSelectedItem()+startMin.getSelectedItem();
String end = df.format(endDate.getDate()) + endHour.getSelectedItem()+endMin.getSelectedItem();

Task task = Task.builder()
    .title(taskTitle.getText())
    .memo(write.getText())
    .importance(Integer.parseInt(importanceBox.getSelectedItem().toString()))
    .startDate(start)
    .endDate(end)
    .build();

TaskService service = new TaskService(task);
```

```
Todo todo = Todo.builder()
.title(todoTitle.getText())
.memo(write.getText())
.importance(Integer.parseInt(importanceBox.getSelectedItem().toString()))
.topFixed(topFixed.isSelected())
.done(false)
.build();

TodoService service = new TodoService(todo);
```

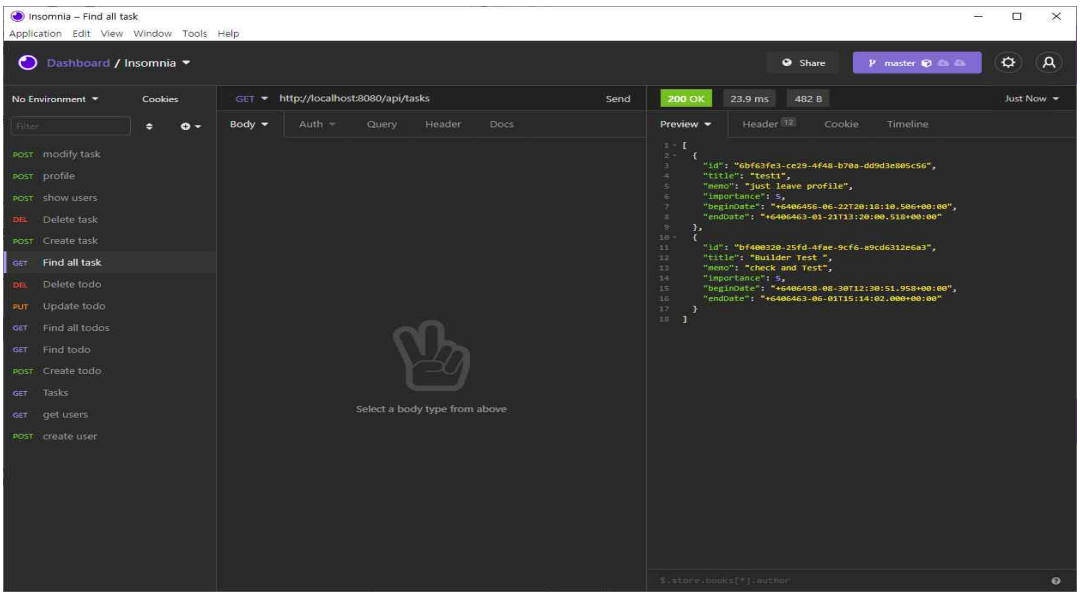
앞에서는 회원가입과 로그인 위주로 설명하였는데 빌더 패턴을 추가 되는 할 일과 스케줄에도 적용을 하였다. 해당되는 소스는 위와 같이 설명된다. 또한, 잘 적용이 되는지 Insomnia 프로그램을 이용하여 추가한 내용들을 확인해 보았다.

4.3.2 테스트 실행 결과

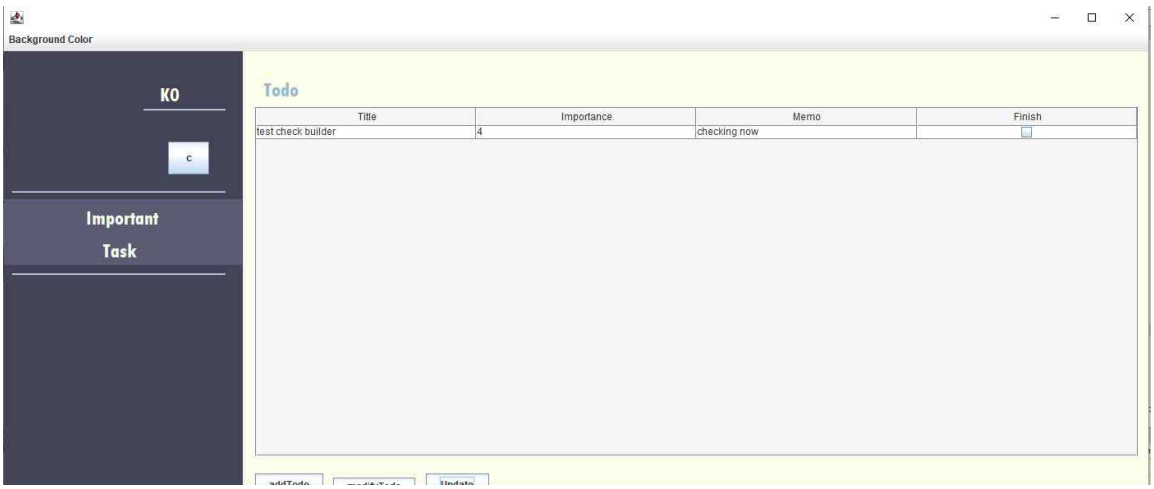
- Task에서 입력 한 내용



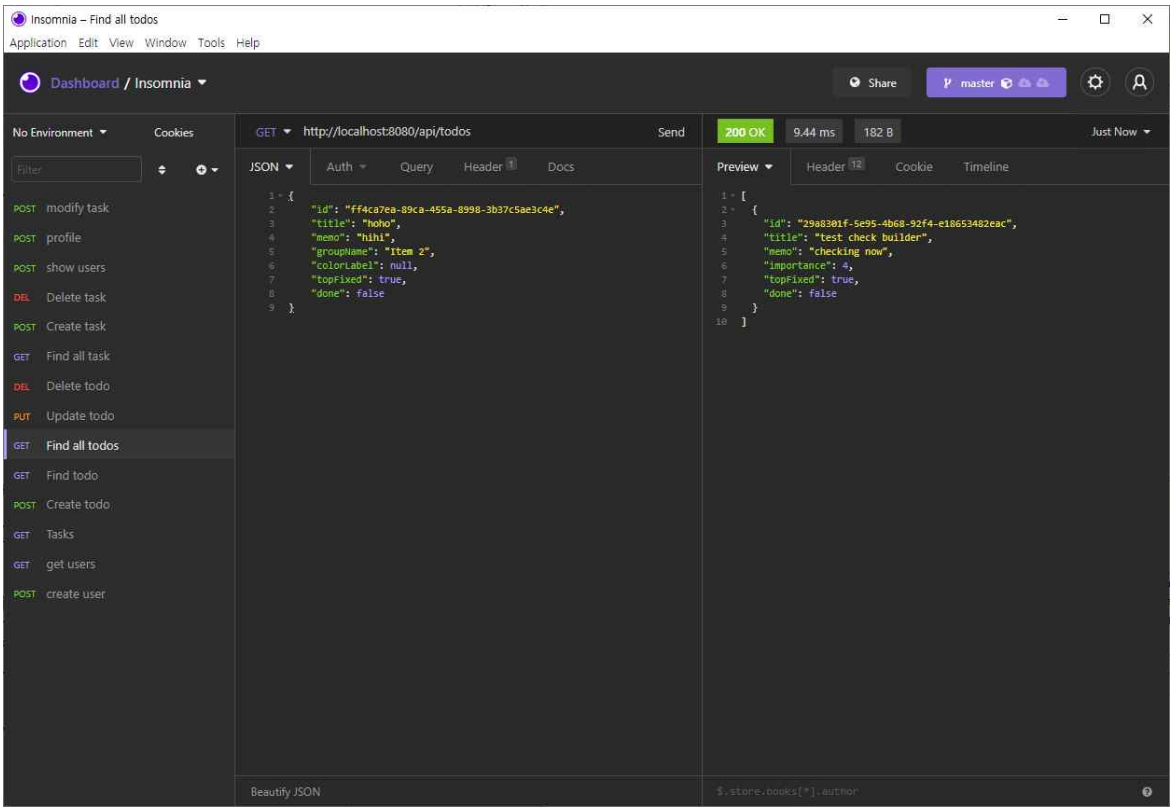
- Insomnia에서 확인한 입력받은 내용



- **TODO에서 입력한 내용**



- **Insomnia에서 확인한 입력받은 내용**



4.4 템플릿 메서드 패턴 테스트

4.4.1 관련 테스트 코드

AbstractController에게 상속받은 UserController, TodoController, TaskController의 create 메소드를 실행시켜 본다.

-UserController 클래스

```
import java.util.List;

@RequestMapping("/users")
@RepositoryRestController
public class UserController extends AbstractController<UserRequest, UUID> {

  private final UserService service;

  @Autowired
  public UserController(UserService service) {
    this.service = service;
  }

  @ResponseBody
  @PostMapping
  public ResponseEntity<?> create(@RequestBody final UserRequest request) {
    return ResponseEntity.ok(service.createUser(request));
  }
```

-TodoController 클래스

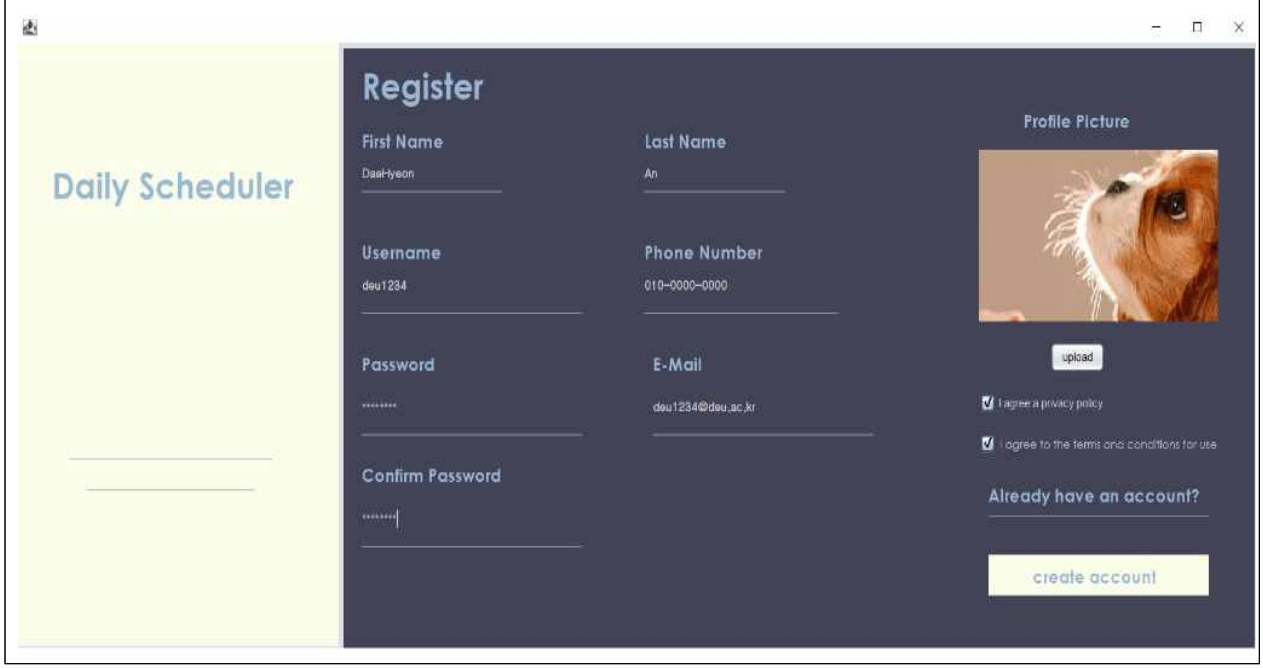
```
import java.util.List;

@RequestMapping("/todos")
@RepositoryRestController
public class TodoController extends AbstractController<TodoRequest, UUID> {

  private final TodoService service;

  @Autowired
  public TodoController(TodoService service) {
    this.service = service;
  }

  @Override
  @ResponseBody
  @PostMapping
  public ResponseEntity<?> create(@RequestBody TodoRequest request) {
    return ResponseEntity.ok(service.createTodo(request));
  }
```

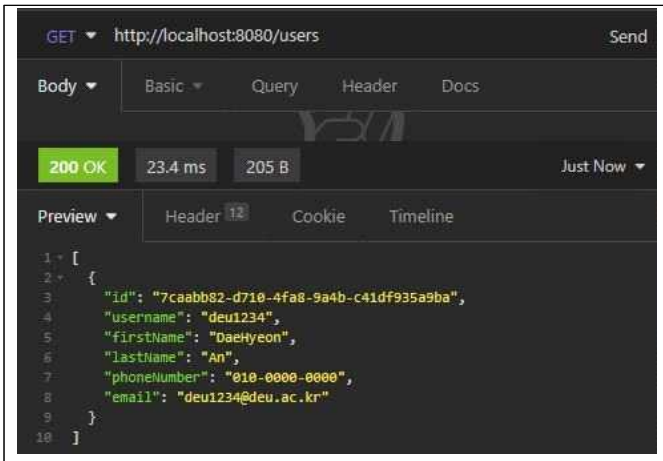
-TaskController 클래스

```
import java.util.List;

@RequestMapping("/tasks")
@RepositoryRestController
public class TaskController extends AbstractController<TaskRequest, UUID> {

  private final TaskService service;

  @Autowired
  public TaskController(TaskService service) {
    this.service = service;
  }

  @Override
  @ResponseBody
  @PostMapping
  public ResponseEntity<?> create(@RequestBody TaskRequest request) {
    return ResponseEntity.ok(service.createTask(request));
  }
```

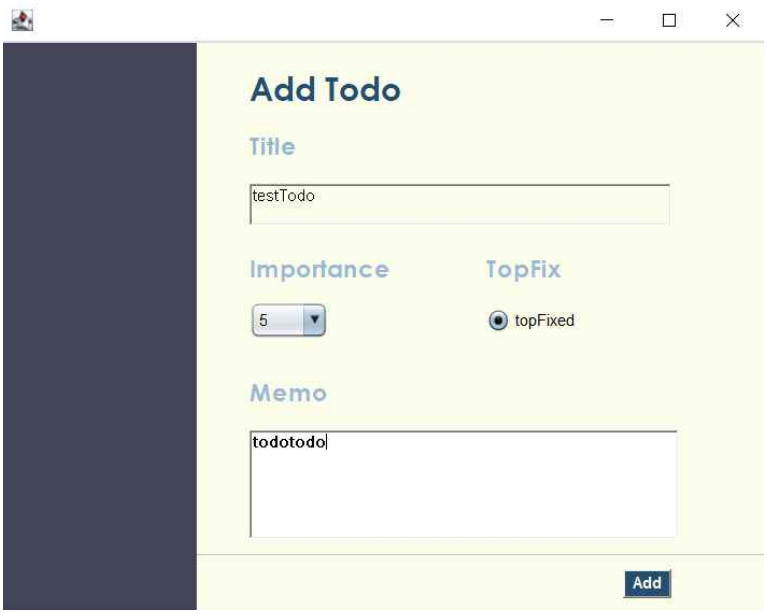
4.4.2 테스트 실행 결과

- 회원 가입



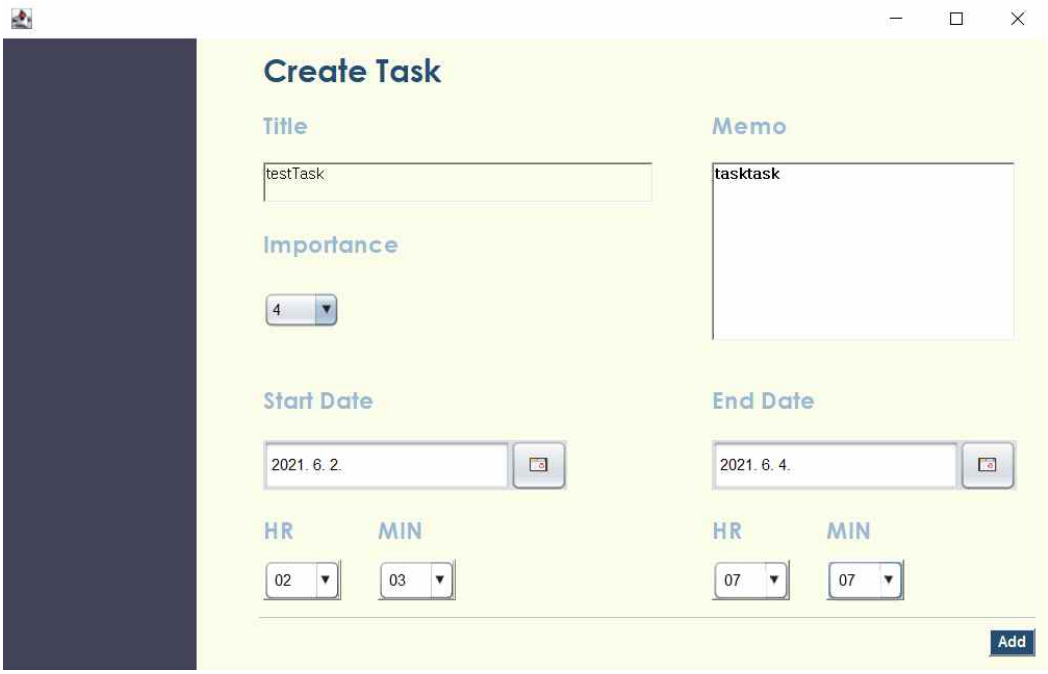

GET http://localhost:8080/users
Send
Body
Basic
Query
Header
Docs
200 OK
23.4 ms
205 B
Just Now
Preview
Header 12
Cookie
Timeline
[
{
"id": "7caabb82-d710-4fa8-9a4b-c41df935a9ba",
"username": "deu1234",
"firstName": "DaeHyeon",
"lastName": "An",
"phoneNumber": "010-0000-0000",
"email": "deu1234@deu.ac.kr"
}
]

- **todo 추가**


Add Todo
Title
testTodo
Importance
TopFix
5
topFixed
Memo
todotodo
Add


[
{
"id": "5b198fa7-dd12-4fd5-8924-2386085e062e",
"title": "testTodo",
"memo": "todotodo",
"importance": 5,
"topFixed": true,
"done": false
}
]

- **task 추가**




```
[
  {
    "id": "4b51a618-43ed-4805-a833-9c0fd8a79476",
    "title": "testTask",
    "memo": "tasktask",
    "importance": 4,
    "beginDate": "+6406454-10-26T07:06:40.203+00:00",
    "endDate": "+6406455-07-05T15:02:20.707+00:00"
  }
]
```

4.5 스트래티지 패턴 테스트

4.5.1 관련 테스트 코드

사용자는 해당 일정이 끝났음을 체크할 수 있는데 task 는 사용자의 End Date 를 보고 완료일을 체크하고, todo 는 실제로 사용자가 완료를 체크하여 일정을 종료한다. 이는 해당 소스를 통해 알 수 있고, 잘 적용이 되었는지 확인해보았다.

```
@Override
public boolean isFinished() {
  return new Date().compareTo(endDate) > 0;
}
```

```
String end_dt1 = "2020-05-26 00:00:00.0";
String end_dt2 = "2022-05-26 00:00:00.0";


int compare1 = endDate1.compareTo(today);
int compare2 = endDate2.compareTo(today);

if(compare1 >= 0) {
    System.out.println("Undone");
} else {
    System.out.println("Finished");
}
if(compare2 >= 0) {
    System.out.println("Undone");
} else {
    System.out.println("Finished");
}
```

- task의 isFinished() => 테스트를 위해 end_dt에는 2019년도의 날짜를 입력하고, end_dt2에는 2021년도 7월의 날짜를 입력한 뒤. 시스템의 오늘 날짜와 비교하여 해당 일정이 끝났는지 아닌지를 출력하도록 하였다.

```
@Override
public boolean isFinished() {
  return this.done;
}
```

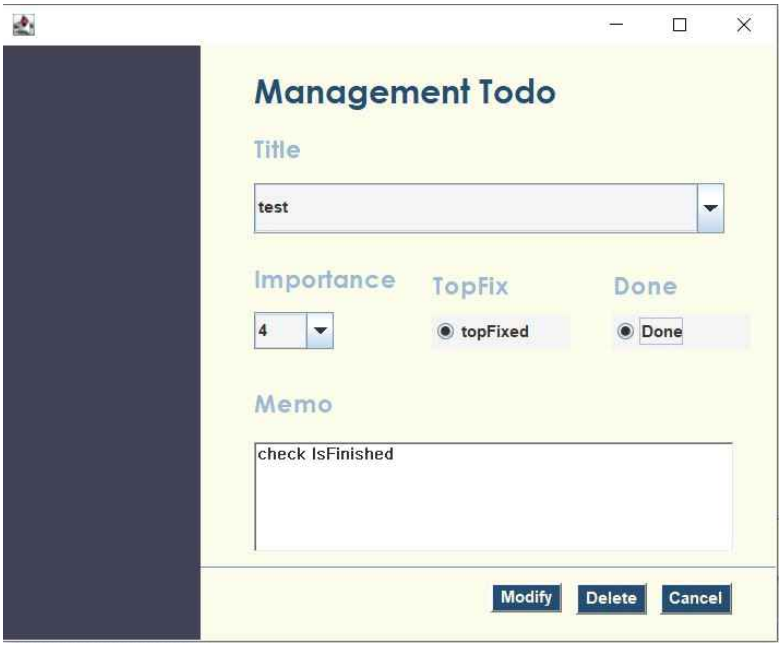
- todo의 isFinished()

4.5.2 테스트 실행 결과

- 오늘 날짜를 기준으로 비교한 task의 일정 종료 여부

```
Finished
Undone
```

- todo의 done을 체크



- 완료 체크


```
[
  {
    "id": "7d6e4e3d-1d96-4fc6-8f30-41e2896abe60",
    "title": "test",
    "memo": "check IsFinished",
    "importance": 4,
    "topFixed": true,
    "done": true
  }
]
```

4.6 팩토리 매서드 패턴 테스트

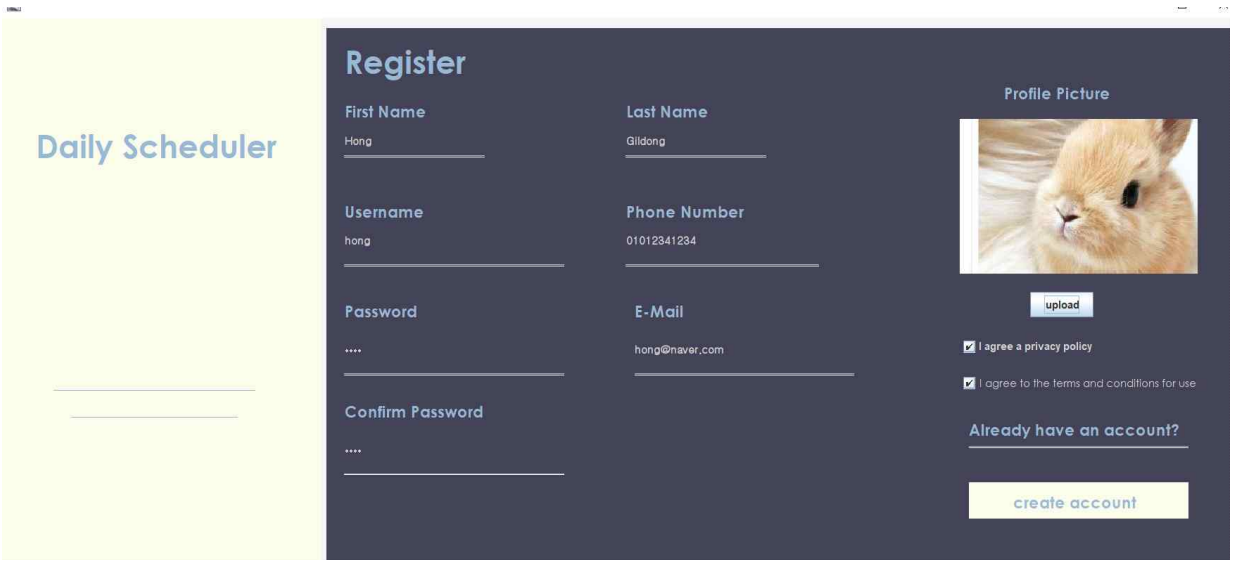
4.6.1 관련 테스트 코드

```
private final UserFactory factory = new CommonUserFactory();

@Transactional
public UserResponse createUser(final UserRequest newUser) {
  String hashedPassword = passwordEncoder.encode(newUser.getPassword());

  User user = factory.create(newUser.getUsername(), hashedPassword,
                             newUser.getFirstName(), newUser.getLastName(),
                             newUser.getPhoneNumber(), newUser.getEMail() );

  UserDataMapper dataMapper = requestAdapter.getEntity(user);
  dataMapper = repository.save(dataMapper);

  return responseAdapter.getEntity(dataMapper);
}
```

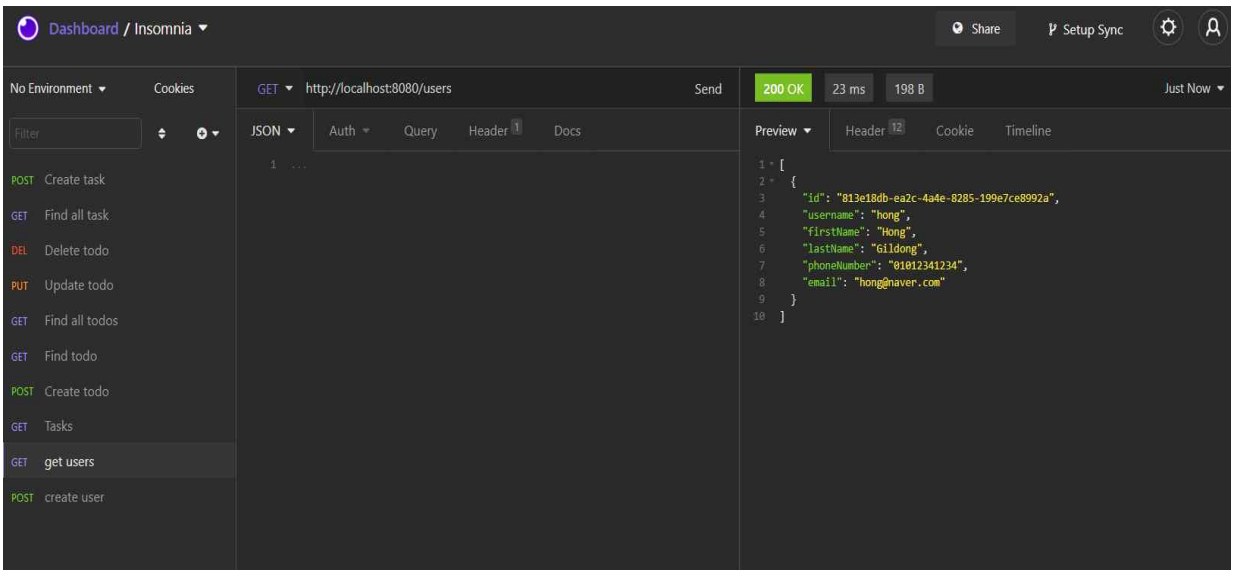
사용자 계정을 추가할 때, 팩토리로 생성된 user 객체의 값이 DB에 올바르게 저장되는지 확인한다.

4.6.2 테스트 실행 결과

- 사용자 회원가입



- 확인 (DB 에 저장)


Dashboard / Insomnia
Share
Setup Sync
No Environment
Cookies
GET
http://localhost:8080/users
Send
200 OK
23 ms
198 B
Just Now
Filter
JSON
Auth
Query
Header 1
Docs
Preview
Header 12
Cookie
Timeline
POST Create task
GET Find all task
DEL Delete todo
PUT Update todo
GET Find all todos
GET Find todo
POST Create todo
GET Tasks
GET get users
POST create user
[
{
"id": "813e18db-ea2c-4a4e-8285-199e7ce8992a",
"username": "hong",
"firstName": "Hong",
"lastName": "Gildong",
"phoneNumber": "01012341234",
"email": "hong@naver.com"
}
]

4.7 어댑터 패턴 테스트

4.7.1 관련 테스트 코드

```
private final UserFactory factory = new CommonUserFactory();

@Transactional
public UserResponse createUser(final UserRequest newUser) {
  String hashedPassword = passwordEncoder.encode(newUser.getPassword());

  User user = factory.create(newUser.getUsername(), hashedPassword,
                             newUser.getFirstName(), newUser.getLastName(),
                             newUser.getPhoneNumber(), newUser.getEMail() );

  UserDataMapper dataMapper = requestAdapter.getEntity(user);
  dataMapper = repository.save(dataMapper);

  return responseAdapter.getEntity(dataMapper);
}
```

user 객체가 어댑터를 통해 dataMapper와 호환가능한 객체로 되는지 확인하기 위해 회원 가입 시 createUser의 return 값을 확인한다.

4.7.2 테스트 실행 결과


```
{
  "id" : "5753d976-f925-491e-b8e8-46e9d2652134",
  "username" : "aaa1234",
  "firstName" : "DaeHyeon",
  "lastName" : "An",
  "phoneNumber" : "010-0000-0000",
  "email" : "aaa122312@deu.ac.kr"
}
```


회원 정보를 입력 후 가입하기 버튼을 누르면 해당 정보가 responseAdapter를 거쳐서 잘 반환되는 것을 확인 할 수 있다.

4.8 스테이트 패턴 테스트

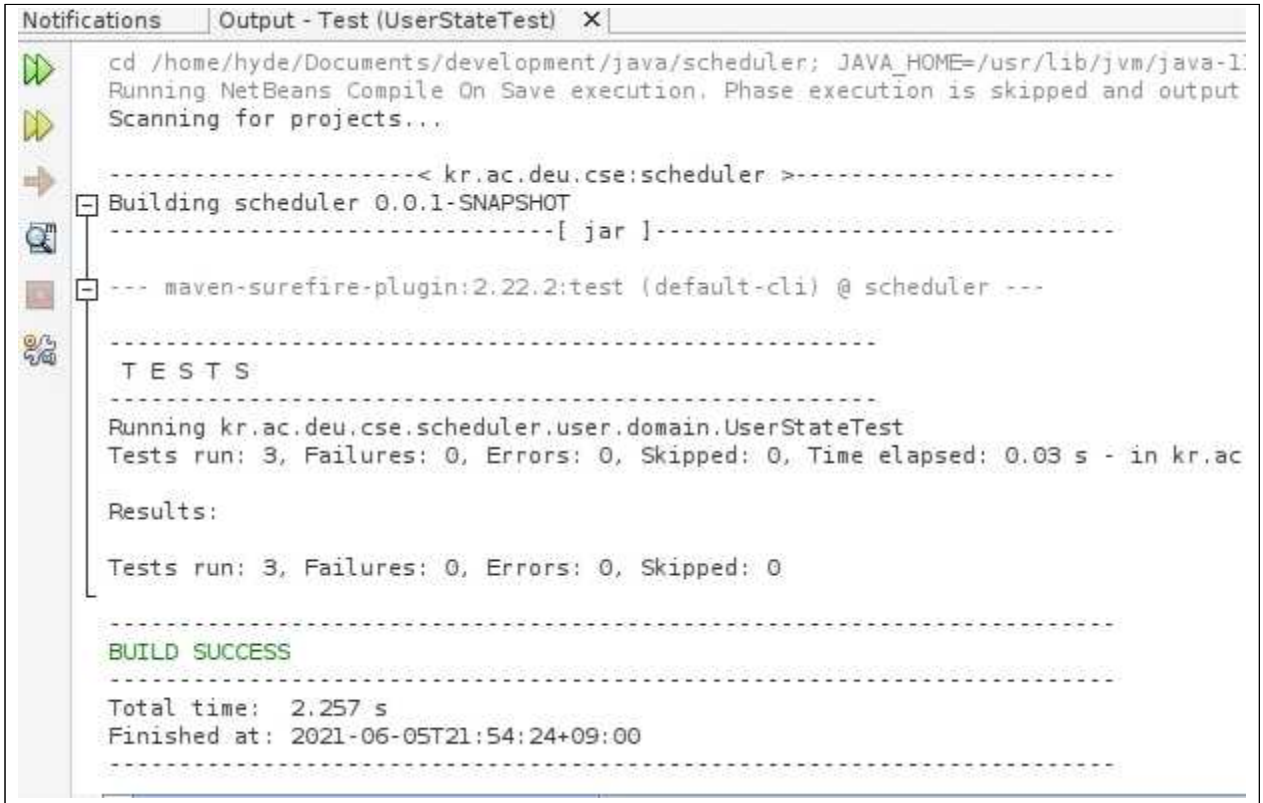
4.8.1 관련 테스트 코드

```
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class UserStateTest {

  UserFactory factory = new CommonUserFactory();
  User user;

  @BeforeEach
  void setUp() {
    user = factory.create(
      "testUser",
      "password",
      "first",
      "last",
      "010-0000-0000",
      "test@example.org"
    );
  }
```

```
  @Test
  void firstCreatedUserStateIsPending() {
    assertTrue(user.getState() instanceof UserStatePending);
  }

  @Test
  void changeUserStateActivated() {
    user.getState().onActivate(user);

    assertTrue(user.getState() instanceof UserStateActivated);
  }

  @Test
  void changeUserStateDeactivated() {
    user.getState().onDeactivate(user);

    assertTrue(user.getState() instanceof UserStateDeactivated);
  }
}
```

팩토리에서 생성된 user 객체의 상태 변화 테스트를 진행했다.

4.8.2 테스트 실행 결과



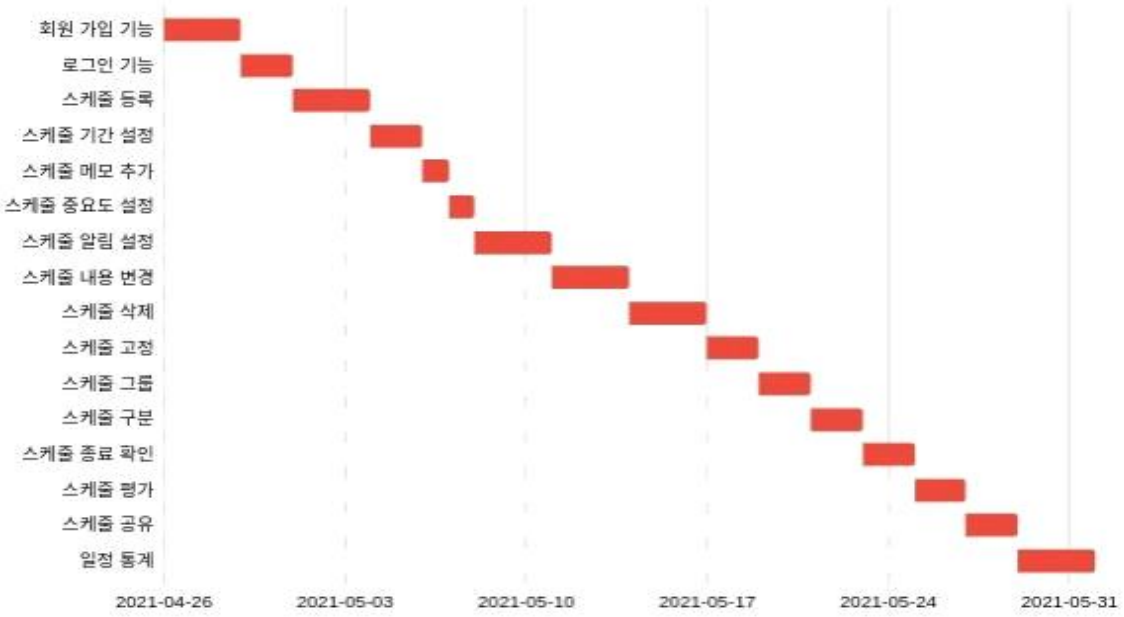
user 객체의 상태가 문제 없이 전환되었고, 해당 패턴을 사용하여 객체의 상태에 따른 if 또는 switch의 조건문을 줄여 코드를 간결화하고 가독성을 높이는 효과를 불러올 수 있다는 사실을 알게 되었다.

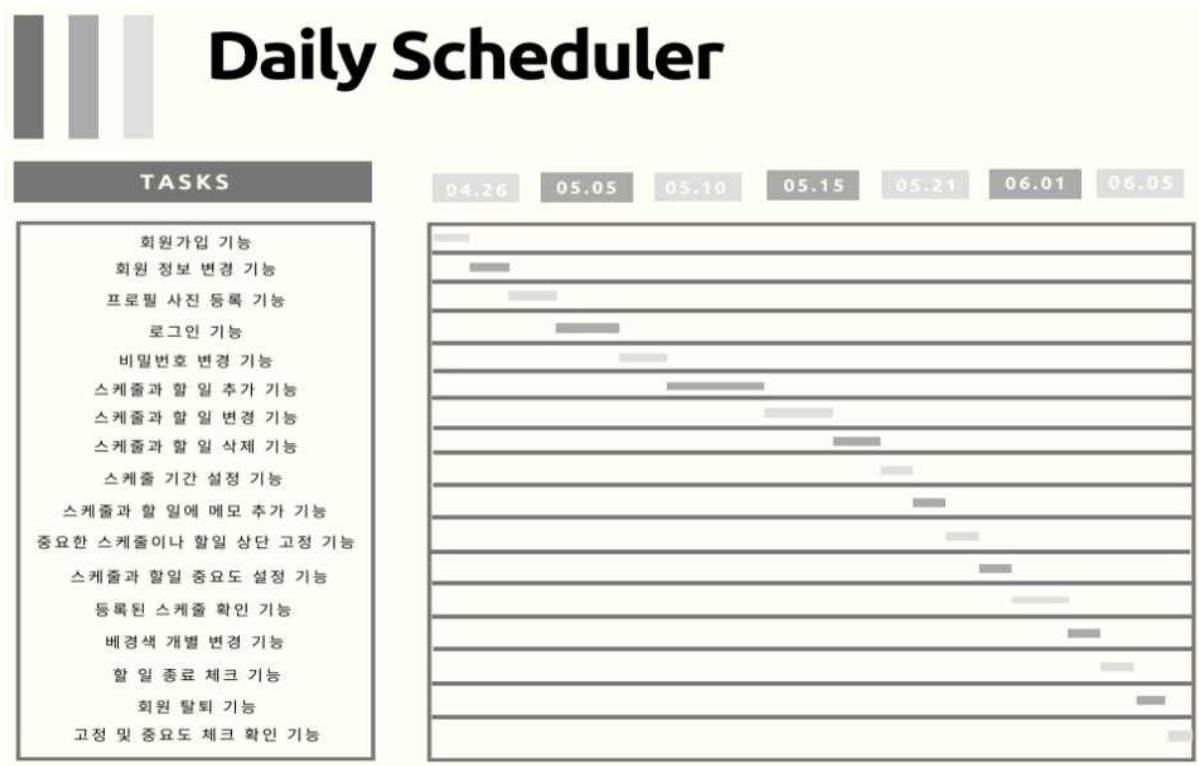
5. 프로젝트 결과

5.1 프로젝트 완성도

추적 가능성 표			완성도	우수성
요구사항	빌더 패턴	SFR-WM 101	100 %	상
	어댑터 패턴	SFR-WM 102	100 %	상
	팩토리 메서드 패턴	SFR-WM 102	100 %	상
	싱글톤 패턴	SFR-WM 301	100 %	중
	스테이트 패턴	SFR-WM 102	50 %	하
	스트레티지 패턴	SFR-WM 302	100 %	상
		SFR-WM 307	100 %	상
	템플릿 메서드 패턴	SFR-WM 302	100 %	상
		SFR-WM 307	100 %	상
	퍼사드 패턴	SFR-WM 102	100 %	상
		SFR-WM 106	100 %	상
자체 완성도 평가			95.5%	상

5.2 일정 계획 평가





위 간트 차트와 아래 간트 차트를 비교해보면 시작은 동일하지만, 최종 종료 날짜가 다르다. 하루 4시간을 기준으로 작업을 하였다. 개발을 하면서 몇몇 기능들이 삭제되고 새로 추가가 되어서 날짜는 예상보다 5일 정도 차이나고, 그 외에는 비슷하게 진행이 되었다. 과제 제출 기간과 시험 기간을 고려하여 개발을 하였다.

5.3 형상 관리 평가

Branches: Show All ☑ Show Remote Branches

Graph	Description	Date	Author	Com···
	main origin patch userinfo	4 Jun 2021 17:19	ads0070	821f9b5f
	change delete method	4 Jun 2021 15:23	ads0070	241371dc
	change delete http method	4 Jun 2021 14:20	ads0070	f3f899bf
	modify http method	4 Jun 2021 11:26	ads0070	98b9e970
	update todo	4 Jun 2021 10:12	ads0070	4c3fbee0
	Add profile Delete group_packages and colorLabel	4 Jun 2021 10:10	ads0070	45a409c0
	test: add user serivce test (#16)	4 Jun 2021 02:38	Dong-Young···	ca178334
	fix: fix h2 server config	4 Jun 2021 00:36	Dong-Young···	90d30b99
	fix: use interface instead of concrete class (#15)	4 Jun 2021 00:02	Dong-Young···	64ea80fe
	feat: add local h2 database (#14)	3 Jun 2021 23:50	Dong-Young···	a0338128
	modify controller	31 May 2021 07:32	ads0070	0c25d34c
	add group	29 May 2021 05:35	ads0070	5d87db···
	update schedule importance	28 May 2021 14:37	ads0070	1dd82400
	update user info	28 May 2021 13:41	ads0070	63ac5c42
	fix(user): update state pattern	27 May 2021 03:44	Dong-Young···	817b855a
	refactor(schedule): update strategy pattern	27 May 2021 03:24	Dong-Young···	d83dceb6
	refactor: apply template method	27 May 2021 03:13	Dong-Young···	dd25fa1c
	refactor: move uml files	27 May 2021 02:58	Dong-Young···	1532212e
	feat: add template method pattern (#13)	27 May 2021 02:53	Dong-Young···	bb6248e0
	style(user): fix code styles	27 May 2021 02:52	Dong-Young···	e770605c
	refactor(user): refactor architecture (#11)	27 May 2021 02:22	Dong-Young···	a94933c2
	feat: update UML	23 May 2021 15:34	Dong-Young···	29c8dba6
	feat(user): add UserStateFactory (#10)	23 May 2021 15:34	Dong-Young···	4d86263f
	feat: update UML (#8)	23 May 2021 12:00	Dong-Young···	095793fc
	feat(user): improve user state pattern (#9)	23 May 2021 02:08	Dong-Young···	81b2bb···
	fix(schedule): remove @SuperBuilder annotation (#6)	22 May 2021 21:44	Dong-Young···	64528b35
	build(docker): change conflicted port	22 May 2021 01:41	Dong-Young···	1c79f78d
	refactor(user): refactor user (#5)	22 May 2021 01:40	Dong-Young···	36f33d40
	refactor(schedule): update schedule package due t···	22 May 2021 00:40	Dong-Young···	1851b91e
	fix(schedule): fix schedule domain properties scope···	21 May 2021 23:43	Dong-Young···	29d6e806
	refactor(schedule): refactor schedule domain mode···	21 May 2021 21:04	Dong-Young···	153fdf89
	feat: add swagger config (#1)	19 May 2021 23:26	Dong-Young···	76033e27
	build(maven): add springfox maven for swagger	19 May 2021 23:26	Dong-Young···	27674e10
	schedule group	19 May 2021 19:46	ads0070	f678c1cf
	fix(user): remove toggle state method from user state	17 May 2021 04:44	Dong-Young···	b5ca9e18
	feat(user): update uml	17 May 2021 04:28	Dong-Young···	db24ac59
	build(docker): add plantuml server	17 May 2021 04:28	Dong-Young···	d0e84ddf

그림 84 서버 프로그램 git commit log

계획서 기준으로 개발을 시작했었다. 중간에 바뀐 내용도 백앤드와 프론트 앤드로 나뉘어서 서로 협업하면서 개발을 하였다. git을 이용하여 형상관리를 진행했다. 계속 맞추면서 개발을 진행 했기에 최종 마무리 시간까지 맞출 수 있었다.

5.4 설계 구성요소

항목	내 용
계획	주제 선정시 기본 주제(또는 자유 주제)를 바탕으로 설계 프로젝트의 최종 목표를 팀원 간의 합의에 의하여 도출하고, 프로젝트 주제에 대한 이해 및 해야 할 일을 계획할 수 있다.
	● 거의 매주 일요일 오후 4시를 고정적으로 회의를 잡아 일정 계획 및 서로 간의 피드백을 주고 받았다. ● 다음 주에는 어떤 작업을 할 것인지 모르는 점이 있다면 설명을 들으면서 진행하였다.
분석	문제 정의와 작업 분해를 통하여 디자인 패턴을 적용하기 위한 요구사항을 도출할 수 있다. (분석 모델링은 없음)
	● 회원가입 유스케이스 : 빌더 패턴 기능을 수행하기 위한 이벤트 흐름을 기술하였다. ● 회원 삭제 유스케이스 : 퍼사드 패턴 기능을 수행하기 위한 이벤트 흐름을 기술하였다. ● task 추가 유스케이스 : 팩토리메서드 패턴, 스트레티지 패턴과 빌더 패턴 기능을 수행하기 위한 이벤트 흐름을 기술하였다. ● todo 추가 유스케이스 : 팩토리메서드 패턴, 스트레티지 패턴과 빌더 패턴 기능을 수행하기 위한 이벤트 흐름을 기술하였다. ● Importance_task 추가 유스케이스 : 팩토리메서드 패턴, 스트레티지 패턴과 빌더 패턴 기능을 수행하기 위한 이벤트 흐름을 기술하였다. ● Importance_todo 추가 유스케이스 : 팩토리메서드 패턴, 스트레티지 패턴과 빌더 패턴 기능을 수행하기 위한 이벤트 흐름을 기술하였다. ● TODO 배경화면 변경 유스케이스 : 싱글톤 패턴 기능을 수행하기 위한 이벤트 흐름을 기술하였다.
상세 설계	요구사항으로부터 문제점을 발견하고 디자인 패턴을 적용하여 설계 모델링을 할 수 있다.
	● 빌더 패턴 : 값을 불러오기 위해 생성자를 정의에 코드가 복잡해지고 오류의 발생을 방지하기 위해 객체를 생성할 때, 일일이 값을 입력하여 생성하지 않고 빌더를 통해 한 번에 객체를 생성하여 데이터의 일관성 유지 및 가독성을 높여 매개변수에 값이 잘못 들어가 오류가 생길 상황을 방지하도록 구현하였다. ● 퍼사드 패턴 : 정보 전송 시 많은 시스템을 거쳐야하는 대신 클라이언트가 요청을 할 때에 객체는 서브시스템 내의 특정한 객체에 요청하고 복잡한 소프트웨어를 사용할 수 있게 간단한 인터페이스를 제공하도록 구현하여 클라이언트가 보다 쉽게 사용하도록 구현하였다. ● 어뎁터 패턴 : 객체를 클라이언트가 기대하는 인터페이스로 변환하여 사용할 수 있게 하기 위해 클래스를 수정하기에 비효율적이고 오류를 일으킬 수 있어 클래스가 인터페이스와 호환되도록 적용시키는 방식으로 구현하였다. ● 팩토리 메서드 패턴 : 직접적으로 객체를 생성하지 않고 팩토리 메서드 패턴을 통해 위임하여 클래스 간의 결합도를 낮춘다. 객체를 생성하는 코드 부분을 분리 시켰기 때문에 객체를 추가/수정이 일어나더라도 객체를 생성하는 코드만 건들면 되므로 클래스에 변경이 생겼을 때 다른 클래스 영향을 덜 준다. ● 스트레티지 패턴 : 기준이 달라지거나 새로운 알고리즘이 생겼을 때 해당 알고

	리즘을 슈퍼 클래스에 생성하여 상속으로 해결하기에 적용되는 문제의 발생을 대비하여 시스템에서 달라지는 부분을 찾아 분리하여 구현하였다. ● 싱글톤 패턴 : 사용자가 생성자를 여러 차례 호출하더라도 하나의 생성자가 리턴되게 구현함으로써 개별적으로 반응할 수 있도록 구현하였다. ● 템플릿 메서드 패턴 : 전체적인 알고리즘은 상위 클래스에서 구현하면서 다른 부분은 하위 클래스에서 구현할 수 있도록 함으로써 전체적인 알코리즘 코드를 재사용하는 데 유용하도록 구현할 수 있다. ● 스테이트 패턴 : 객체를 호출하여 시행하는 것이 아닌 상태를 객체화 하여 상태가 행동을 할 수 있도록 하는 상태를 구현하였다.
테스팅 (시험)	적용된 디자인 패턴이 올바르게 동작함을 보이기 위해 수행한 통합/시스템 테스팅을 수행할 수 있다.
	● 퍼사드 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과 보고서 4.1)을 수행하였음. ● 싱글톤 메서드 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과 보고서 4.2)을 수행하였음. ● 빌드 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과 보고서 4.3)을 수행하였음. ● 템플릿 메서드 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과 보고서 4.4)을 수행하였음. ● 스트레티지 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과 보고서 4.5)을 수행하였음. ● 팩토리 메서드 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과 보고서 4.6)을 수행하였음. ● 어뎁터 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과보고서 4.7)을 수행하였음. ● 스테이트 패턴을 구현하였음을 증명하기 위하여 시스템 테스팅(결과 보고서 4.8)을 수행하였음. ● 각자 작성한 코드를 합친 후 전체적 테스팅을 완료하였다.

5.5 현실적 제한조건

항목	내 용
생산성	코딩만 수행하는 기존 프로그래밍 기법의 한계를 극복하고 보다 좋은 구조를 가지는 프로그램을 제작하여 생산성을 높인다. (bouml, git 형상관리도구 활용 측면)
	● NetBeans IDE의 maven 빌드 도구를 사용하여 라이브러리 설치를 자동으로 하게 하였으며, FindBugs 정적 분석 도구를 사용하여 발견된 주요 결함(major 이상)을 제거하였다. ● 패턴을 적용할 때 bouml을 이용하여 클래스 미리 만들고, UML을 활용하여 프로그램을 구현하면 더욱 쉽고 빠르게 구현이 가능하다. ● Database를 이용하여 필요한 정보를 외부에 별도로 저장하고, 요청 시에만 불러와 사용함으로써 코드를 간결화하고, 데이터 접근이 용이하게 한다. ● Git을 이용하여 팀원 간의 코드를 공유함으로써 서로의 의견을 피드백하여 효율적인 프로젝트 처리가 가능하도록 한다.
산업표준	프로젝트를 수행하기 위하여 객체지향 분석, 설계의 기본 요소인 클래스 표기 방법인 UML 클래스 다이어그램을 활용할 수 있어야 하며, 적절한 SW 개발 방법론과 테스팅, 형상 관리 기법을 적용할 수 있어야 한다.
	● 개발 프로세스로 애자일 개발 방법론인 스크럼을 사용한다. ● 개발 환경은 NetBeans IDE(maven)를 사용한다. ● 개발 단계에서 테스팅시 ISO/IEC/IEEE 29119 SW 테스팅 국제 표준에서 정의하는 테스트 케이스 설계 기법(경계값 분석 등)을 적용한다. ● 객체지향 분석, 설계의 기본 요소인 클래스 표기 방법인 UML을 이해하고, 클래스의 구조를 디자인 패턴에 맞게 표현할 수 있어야 한다.
기타	개발 환경은 NetBeans IDE(maven)를 사용하여야 한다.
	● easyUML을 이용하여 클래스 다이어그램을 작성하였고, 작성한 클래스 다이어그램을 코드화하여 개발 단계에 코드 작성에 사용하였다. ● GUI를 사용하여 사용자에게 보다 편리하게 이용할 수 있게 구성하였다.