**Project#4 Final Report**

**Seat Arrangement Program**



**Team member**

**안준형(Leader) 김재형 김종욱 박찬주 유가람 안태원**

**INDEX**

a) Project Summary (주제 소개)

b) How to compile and execute (작동법 소개)

c) Description on what functions are implemented (기능 소개)

* UML, Algorithm, GUI

d) How you implemented (구현 방법)

e) How applied object oriented concepts (과제 내 OOP적 개념의 사용)

f) Conclusion (결론)

g) Execution result (결과 화면)

**(a) Project Summary**

이 프로젝트는 사용자들에게 좌석을 효율적으로 할당해주는 프로그램이다. Best Fit을 사용하고. 응집도(Coupling)이라는 개념을 도입하여, 효율적인 자리 배치를 하여 더욱 많은 사람이 좌석을 이용 가능하게 해주는 것이 프로그램의 목표이다.

**(b) How to compile and execute**

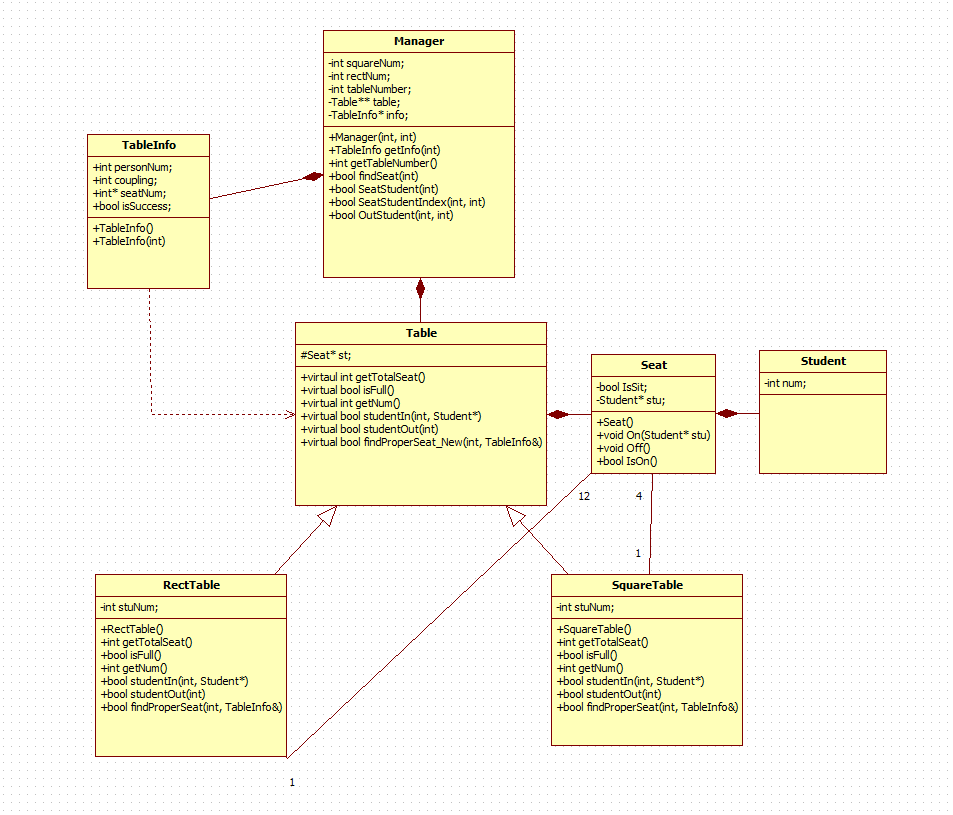
이 프로그램을 컴파일하기 위해서는 QT, VS2010이 컴퓨터에 설치되어있어야 한다.

프로그램을 작동하기 위해선, qt-project.org에서 Qt library (VS210 버전)와 Qt add-in(VS2010)을 설치한다.

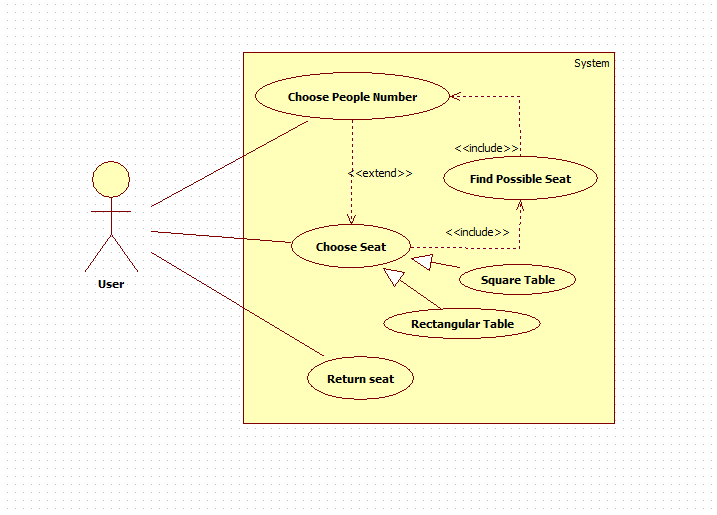
(단, 컴파일시 cmd .exe 실행에러가 뜨면, 압축 파일을 바탕화면에서 풀고 컴파일을 해야됩니다.)

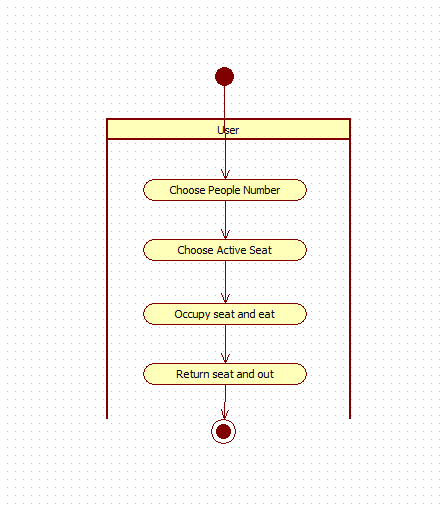
**(c) Description on what functions are implemented**

**1) UML**



**-> Class Diagram**

** -> Use-case Diagram**

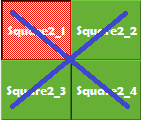
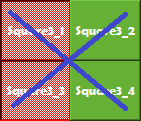
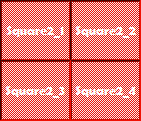
**-> Activity Diagram**

**2) Algorithm**

**◇Table 배치 가정**

**Square**(정사각형) Table

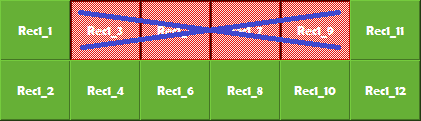
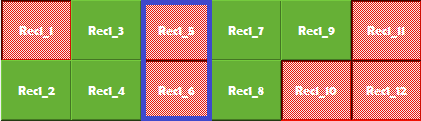
* **2~4명**만 앉을 수 있음.
* 누구라도 앉아 있는 경우, 추가로 접수 받지 않음. 즉, **테이블 자체를 할당**함.

A  B  C 

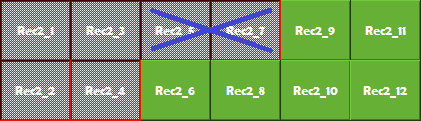
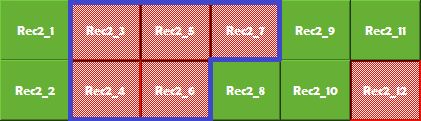
* A : 1명은 앉을 수 없음
* B : 테이블 자체를 할당해야 함

**Rect**(직사각형) Table

* **짝수**인 경우, **가로로 연속해서 앉히지 않음**.

A  B 

* A : 가로로 연속하여 앉히지 않음.
* B : 다음과 같이, 항상 마주보는 형태로 함.
* **홀수**인 경우, **마주보는 사람 없이 2명 이상을 가로로 연속해서 앉히지 않음**.

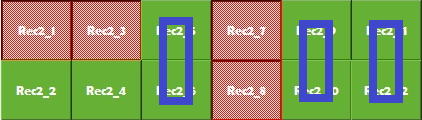
A  B 

* A : 마주보는 사람 없이 2명 이상을 가로로 배치하지 않음.
* B : **1명까진 허용**하면서, **가능한 마주보는 형태**로 자리를 배치함.

**◇자리를 찾는 알고리즘 – Best Fit, Coupling(응집도) 사용**

First Fit – Bin 중 조건에 만족하는 첫 번째 Bin을 찾으면, 해당 Bin을 사용

**Best Fit(채택)** – **조건에 만족하는 Bin 중, 최소 크기의 Bin**을 사용함

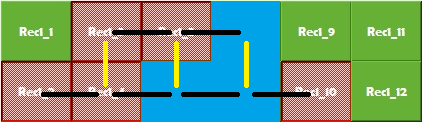
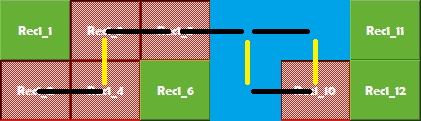
A  B 

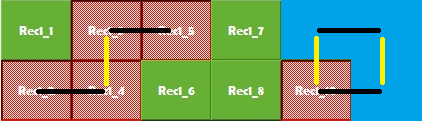
* A : Best Fit을 이용, Bin의 크기가 2인 곳을 찾을 수 있음.
* B : 크기가 2개인 Bin이 3개나 있지만, 아직 어느 곳에 넣는 것이 최적인지 알 수 없음
* **Best Fit만으로는 어떤 자리가 최적인지 알 수 없음**

**Coupling**(응집도)

* **Best Fit 만으로는 얻을 수 없는 정보를 얻기 위해 가정한** 도수
* 좌석에 앉아있는 사람을 기준으로, 주변에 사람이 얼마나 붙어있는지를 계산
* **Coupling이 최대가 되는 Bin**이 가장 적합함. (Best Fit 적용 후 Bin의 크기가 같을 경우)
* **가중치**를 두어, **세로로 인접한 경우**는, 가로로 인접한 경우보다 2배 더 큰 값을 갖는다.
* 응집도(최종) = 응집도(가로) + 응집도(세로) // 중복 계산은 하지 않음

Best Fit으로 도출되는 경우의 수 (Bin Size : 3)

ⅰ)  ⅱ) 

ⅲ) 

ⅰ)의 경우, 응집도 = 6(가로) + 3\*2(세로) = **12**

ⅱ)의 경우, 응집도 = 5(가로) + 3\*2(세로) = **11**

ⅲ)의 경우, 응집도 = 4(가로) + 3\*2(세로) = **10**

따라서, ⅰ)의 경우처럼 자리를 배치하는 것이 적합하다.

**3) GUI**

Vs2010 환경에서 GUI를 구현하기 위해 Opensource인 Qt를 이용하였다. 하나의 좌석은 QPushButton으로 구성되어 있으며 각 Seat이 정상적으로 동작하기 위해 manager.cpp에서 생성된 table 객체를 참조하였다.

Manager(int square, int rect) 생성자를 통해 square 개의 정사각형 테이블과 rect 개의 직사각형 테이블을 생성하는 Manager 객체 rest 를 만든다. 그 후 각각의 테이블과 GUI를 연동시킨다.

기본 화면 구성은 4명이 앉을 수 있는 테이블 3개, 12명이 기본인 테이블 2개로 구성되어 있으며 비어있다는 의미로 초록색 버튼과 사람이 앉아있는 빨간색 버튼으로 구분된다. 클릭 시, 비어 있을 경우 빨간색 버튼으로 사람이 들어있다고 인식하며 변하며, 그 반대의 경우 버튼이 초록색으로 되돌아오며 해당 좌석이 비게 된다.

if(ui.Square1\_1->isChecked()) { //클릭 했을 때 자리에 학생이 있으면

if(rest.SeatStudentIndex(0,0)) //학생을 앉힌다

ui.Square1\_1->setStyleSheet(QString::fromUtf8("QPushButton { color: white; background-color: rgb(208, 16, 5) }")); //버튼 색을 빨간색으로 바꾼다

}

else { //학생이 없으면

if(rest.OutStudent(0,0)) //학생을 자리에서 내보낸다

ui.Square1\_1->setStyleSheet(QString::fromUtf8("QPushButton { color: white; background-color: rgb(102, 176, 54) }")); //버튼 색을 초록색으로 바꾼다

}

}

하단에는 1~8의 숫자가 적힌 버튼이 있는데 이는 한번에 입장할 인원 수를 의미한다. 그 임의의 숫자가 적힌 버튼을 눌렀을 시, rest내부에 있는 rest.findSeat(int num) 함수를 통해 해당 인원만큼 자리에 앉을 수 있는 최적의 자리의 정보 값을 GUI에 넘겨준다. 그럼 최적의 자리는 화면 상에 하늘색 버튼으로 나타낸다. 사용자가 하늘색 좌석그룹 중 한 그룹을 선택하면 그 선택 값을 rest 인스턴스에서 받아와서 rest.SeatStudentIndex() 함수를 통해 해당 자리에 학생들을 앉히고 GUI에서는 좌석색깔을 빨간색으로 바꿔준다. 이 때 다른 초록색 버튼은 클릭할 수 없으며 이용자들은 추천된 자리들(최적의 자리) 중 하나를 골라서 앉을 수 있다. 4명이 앉은 테이블에는 입장 인원이 한 명인 경우 좌석의 효율성을 위해 추천해주지 않는다.

 => 기본 좌석 (사람이 앉지 않음)

=> 앉은 경우.

=>최적화된 추천 좌석



* 입장 인원 수 (최소 1명 ~ 최대 8명 입장 가능)

**(d) How we implemented (important implementation issues)**

**1)Structure**

OOP적인 관점에 맞춰 최대한 사용자가 이용하기 편하고 유지보수가 높아지는 방향으로 설계하였다. 실제 현실에서의 사물들을 객체화하고 객체의 속성, 행동들을 그대로 프로그래밍화 하였다. 그 예로 Seat 클래스의 On(Student\*) 함수를 보면

void Seat:: On(Student\* stu) { //학생이 자리에 앉음

isSit = true;

this->stu = stu;

}

에서 Student 클래스 인스턴스의 주소를 parameter로 받아 학생이 해당 자리에 앉는 행동을 그대로 표현하였고

또 다른 예로 Table 클래스의 studentIn(int) 함수를 보면

bool SquareTable::studentIn(int index, Student\* stu) {

if (index<0 || index >(SQUARE\_NUM-1)){ //잘못된 index 값

printf("studentIn함수에서 잘못된 index값\n");

return false;

}

if(st[index].IsOn() == true){ //index 번째의 자리에 학생이 앉아있으면

printf("studentIn함수에서 index 번째 자리에 학생이 있는데 앉으라 명령!\n");

return false;

}

else { //index 번째의 자리에 학생이 없으면

st[index].On(stu); //학생을 앉힘

stuNum++;

printf("square table %d th seat in\n", index);

return true;

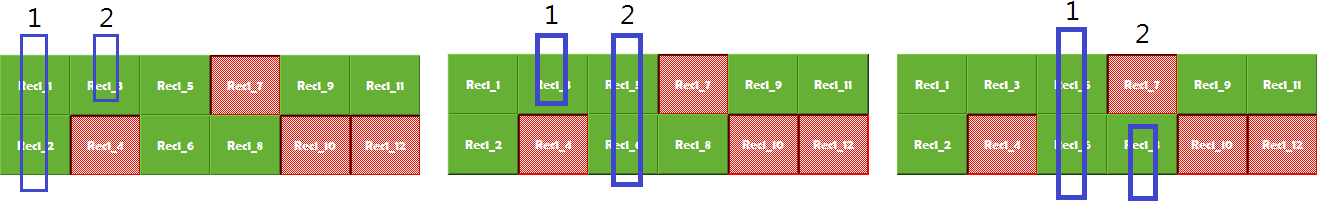
}

}

에서 int형 변수 index와 Student\*형 인스턴스 stu를 parameter로 받아 index 번 째의 자리에 학생을 앉히는 행동을 그대로 프로그램화 하였다.

또한 상속 및 Dynamic Binding 등을 사용하여 Maintenance가 높아지도록 하였고 이 외의 자세한 설명은 뒤의 **(e) Explain for how we applied object oriented concepts to the development for project.** 에서 마저 하도록 하겠다.

**2) Algorithm**



다음과 같이, x좌표값을 기준으로 오른쪽으로 선형검사를 한다.

앞에서 언급한 Bin의 조건에 맞지 않는 검색이 될 경우, 자동적으로 검색을 종료한다.

그에 해당하는 코드는 다음과 같다.

while(1){

// seatList[1+RECT\_NUM+1] 에는..

// 0 : 사람 수

// 1~12 : 좌석 자리

// 13 : 응집도

// 를 대입한 결과가 이 while문 실행 후 저장됨.

if(x==6) // 범위 외 검색을 시도하는 경우

break;

if(seatList[0] >= personNum)

break;

if(isFirstTry){ // 처음 검색을 시도하는 경우

if(isTableSeat[0][x] && isTableSeat[1][x]) // 두 자리 모두 안되는 경우

break;

else if(isTableSeat[0][x] && !isTableSeat[1][x]){ // 아래자리만 가능한 경우

isCornered = true; // 구석 자리를 할당했음을 표시

seatList[0]++;

seatList[seatList[0]] = 2\*x+1; // 자리 할당

isFirstTry = false;

}

else if(!isTableSeat[0][x] && isTableSeat[1][x]){ // 윗자리만 가능한 경우

isCornered = true; // 구석 자리를 할당했음을 표시

seatList[0]++;

seatList[seatList[0]] = 2\*x; // 자리 할당

isFirstTry = false;

}

else if(!isTableSeat[0][x] && !isTableSeat[1][x]){ // 둘 다 가능한 경우

seatList[0]++;

seatList[seatList[0]] = 2\*x;

seatList[0]++;

seatList[seatList[0]] = 2\*x+1; // 자리 할당 x2

isFirstTry = false;

}

x++; // 다음 검색으로..

}

else if(!isFirstTry){ // 처음 검색을 시도하는 경우가 아닌 경우

if(isTableSeat[0][x] && !isTableSeat[1][x]){ // 아래자리만 가능한 경우

if(isCornered) // 이미 구석 자리를 할당한 경우

break;

else if(!isCornered){ // 구석 자리를 할당하지 않은 경우

seatList[0]++;

seatList[seatList[0]] = 2\*x+1; // 자리 할당 후 종료

break;

}

}

else if(!isTableSeat[0][x] && isTableSeat[1][x]){ // 윗자리만 가능한 경우

if(isCornered) // 이미 구석 자리를 할당한 경우 종료

break;

else if(!isCornered){ // 구석 자리를 할당하지 않은 경우

seatList[0]++;

seatList[seatList[0]] = 2\*x; // 자리 할당 후 종료

break;

}

}

else if(isTableSeat[0][x] && isTableSeat[1][x]) // 두 자리 모두 안되는 경우

break; // 할당 종료

else if(!isTableSeat[0][x] && !isTableSeat[1][x]){ // 두 자리 모두 되는 경우

seatList[0]++;

seatList[seatList[0]] = 2\*x;

seatList[0]++;

seatList[seatList[0]] = 2\*x+1; // 자리 할당 x2

}

x++; // 다음 검색으로..

}

}

또한, 이렇게 얻은 Bin에 대한 응집도의 계산은 다음과 같다.

int coupling=0;

for(int i=0; i<6; i++){

if(isTableSeat[0][i]){ // (i, 0)이 사용 가능한 경우

if(isTableSeat[1][i]) // 밑 좌석이 사용 가능한 경우

coupling+=2; // 응집도 += 2 (세로로 인접)

if(i<5)

if(isTableSeat[0][i+1]) // 오른쪽 좌석이 사용 가능한 경우

coupling++; // 응집도 += 1 (가로로 인접)

}

}

for(int i=0; i<6; i++){

if(isTableSeat[1][i]) // (i, 1)이 사용 가능한 경우

if(i<5)

if(isTableSeat[1][i+1]) // 오른쪽 좌석이 사용 가능한 경우

coupling++; // 응집도 += 1 (가로로 인접)

}

**3) GUI**

GUI 환경에서 좌석 중 최적화된 좌석을 분석하여 화면상에 보여주게 하는 changeColor 함수가 핵심이다. 그 외에 GUI를 구성하는 함수들은 다음과 같다.

void changeColor();

void clearBlue(); //하늘색 좌석 선택 시, 다른 테이블의 하늘색 좌석들을 초록색으로 초기화 초록색으로 초기화

void checkedRed(); //초록색으로 초기화 시, 이미 선택된 좌석의 경우 빨간색 표시.

void inputClear(); // 8->1 순으로 input값 선택 시, 하늘색 좌석의 재 초기화.

//void setCheckable();

void checkableFalse(); //하늘색 좌석 & 빨간색 좌석 이외의 모든 초록색 좌석을 선택불가로 만듦. 선택불가로 만듦.

void checkableTrue(); //다시 모든 전 좌석을 선택 가능케 함(하늘색 좌석 선택 후 화면). 화면).

==🡺 changeColor()의 내부 구현

void practice ::changeColor() {

bool seatOccupied[12]; //최적화된 좌석을 확인하기 위한 배열

for(int loop=0; loop<12; loop++){

seatOccupied[loop] = false;

for(int loop2=0; loop2<5; loop2++){

possibleSeat[loop2][loop] = false;

//파란색으로 변한 것 카운트 하기위해 모두 false로 초기화 해줌.

}

}

if(rest.getInfo(0).isSuccess){ //0번 째 테이블에 최적화된 좌석의 유무 확인

int temp = rest.getInfo(0).personNum; //최적화된 좌석의 개수

for(int loop=0; loop<temp; loop++){

seatOccupied[rest.getInfo(0).seatNum[loop]] = true; //최적화된 좌석의 위치 표시

}

for(int loop=0; loop<SQUARE\_NUM; loop++){

if(seatOccupied[loop]){

// blue

if(loop==0) { //최적화된 좌석을 찾아 그 좌석의 색상을 하늘색으로 변경하여 최적화되었음을 표시 표시

ui.Square1\_1->setStyleSheet(QString::fromUtf8("QPushButton { color: white; background-color: rgb(73, 186, 254) }"));

possibleSeat[0][0]=true; //버튼 클릭 시에 해당좌석이 최적화되어 있는 지를 판별.

}

...if(loop==3)까지 동일 적용.

checkableFalse(); //하늘색 좌석 & 이미 선택된 좌석(빨간색) 외의 모든 초록색 좌석을 선택불가로 변형. 좌석을 선택불가로 변경.

**(e) Explain for how we applied object oriented concepts to the development for project.**

Table Class를 **Abstract Class**(추상 클래스)로 만들어서, RectTable과 SquareTable Class에 **Inheritance**(상속)하여 사용했다. 이로써 이 후에 추가로 테이블 형태를 추가할 때도 형식을 많이 바꾸지 않고 상속만 시킴으로써 해결 할 수 있도록 하였다. 또한, 상위 추상클래스의 **virtual function을 이용한 Dynamic Binding**이 일어날 수 있도록, Table을 관리하는 Class인 Manager Class에는, Table\* 형 변수가 들어있다.

class Manager{

private:

...

int squareNum;

int rectNum;

int tableNumber;

Table\*\* table;

...

};

class Table{

protected:

Seat\* st;

public:

virtual int getTotalSeat()=0;

virtual bool isFull()=0;

virtual int getNum()=0;

virtual bool studentIn(int , Student\*)=0;

virtual bool studentOut(int)=0;

virtual void findProperSeat\_New(int, TableInfo&)=0;

};

class SquareTable : public Table{

private:

int stuNum;

public:

SquareTable();

virtual int getTotalSeat();

virtual bool isFull();

virtual int getNum();

virtual bool studentIn(int index, Student\* stu);

virtual bool studentOut(int index);

virtual void findProperSeat\_New(int, TableInfo&);

};

class RectTable : public Table{

private:

int stuNum;

public:

RectTable();

virtual int getTotalSeat();

virtual bool isFull();

virtual int getNum();

virtual bool studentIn(int index, Student\* stu);

virtual bool studentOut(int index);

virtual void findProperSeat\_New(int, TableInfo&);

void bestFitBin(int personNum, int\* bin);

int calcCoupling(int\* seatLIst);

};

**Dynamic Binding**의 예로 Manager Class의 멤버함수 findSeat 함수를 보면

bool Manager::findSeat(int num) {

for(int i=0; i<tableNumber; i++) {

table[i]->findProperSeat\_New(num, info[i]); **//Dynamic Binding!**

}

for(int i=0; i<tableNumber; i++) { //한 테이블이라도 자리가 찾아지면 true 반환

if(info[i].isSuccess == true)

return true;

}

return false; //어떤 테이블에서도 자리를 못찾았으면 false 반환

}

이렇게 **Dynamic Binding**을 사용함으로써 Manager 클래스에서 Table 객체들을 사용 및 관리기 편하게 코드를 작성하였고 **maintenance**를 높였다(유지 및 보수가 쉽도록 했다.)

그 외에 OOP적 관점에서 실제 학생, 좌석, 테이블 등을 그대로 프로그래밍 해서 객체화했다.

class Student

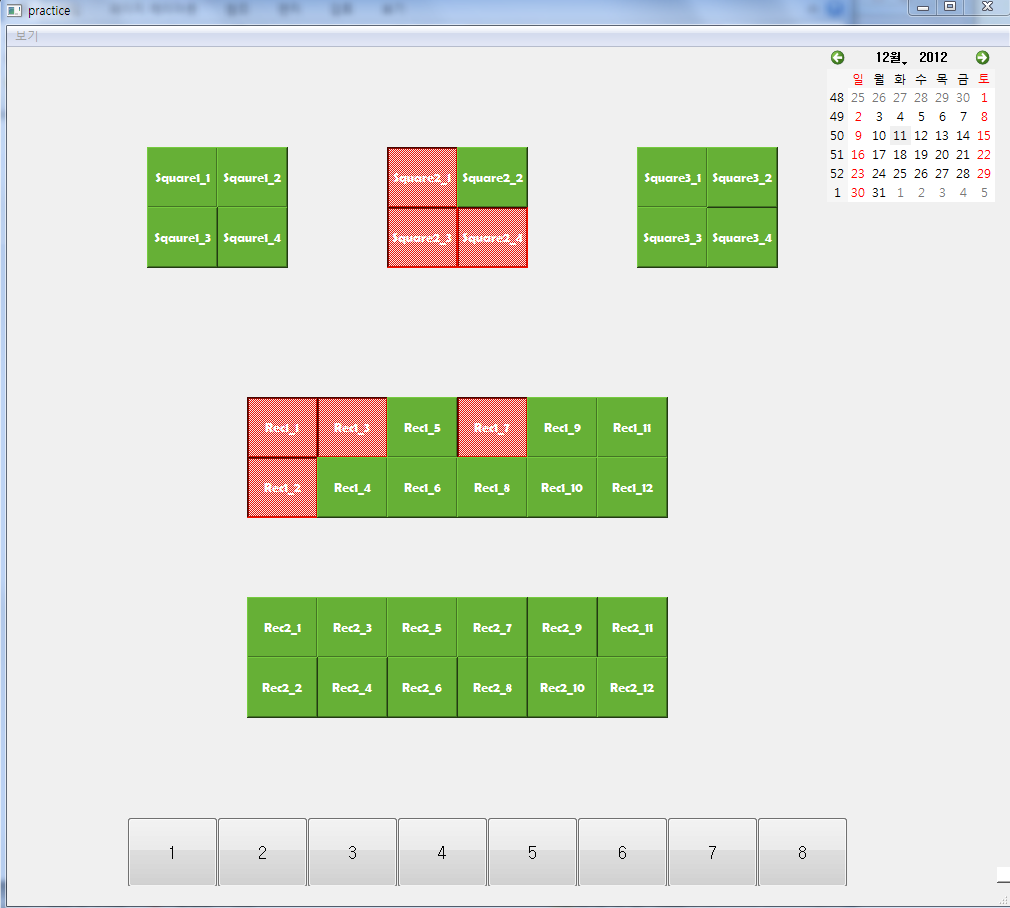
class Seat

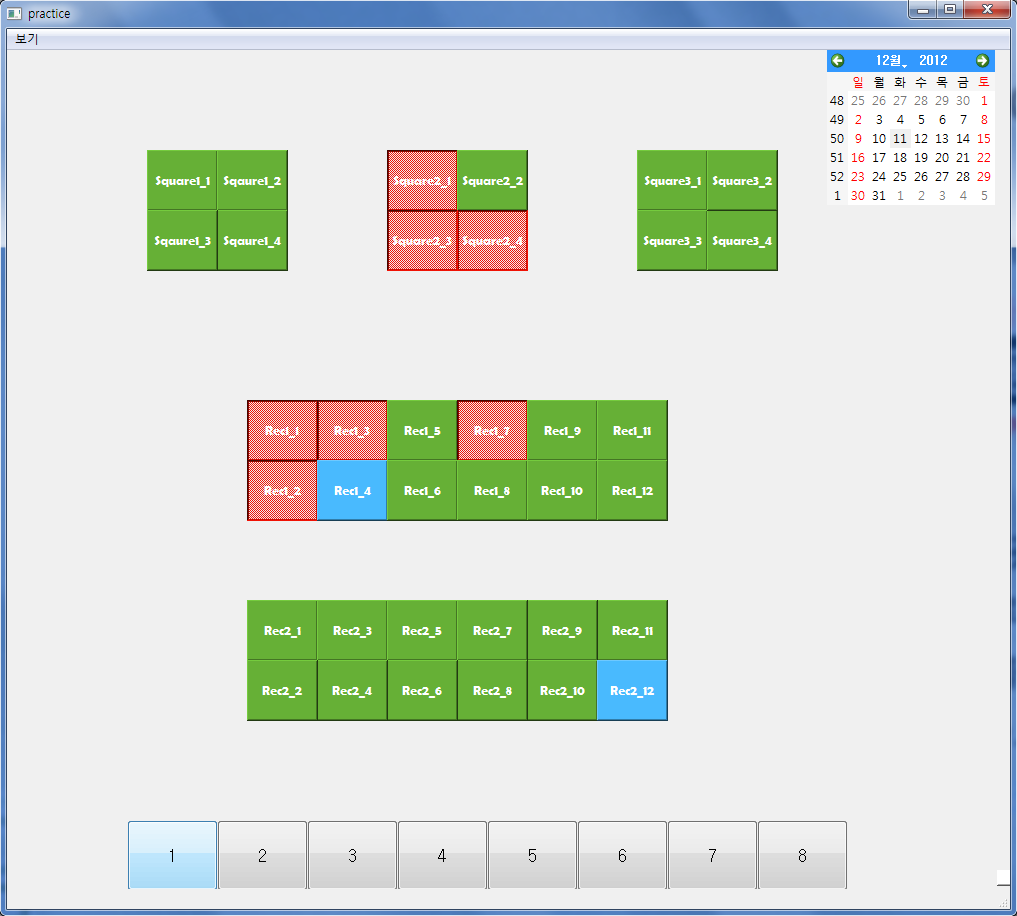
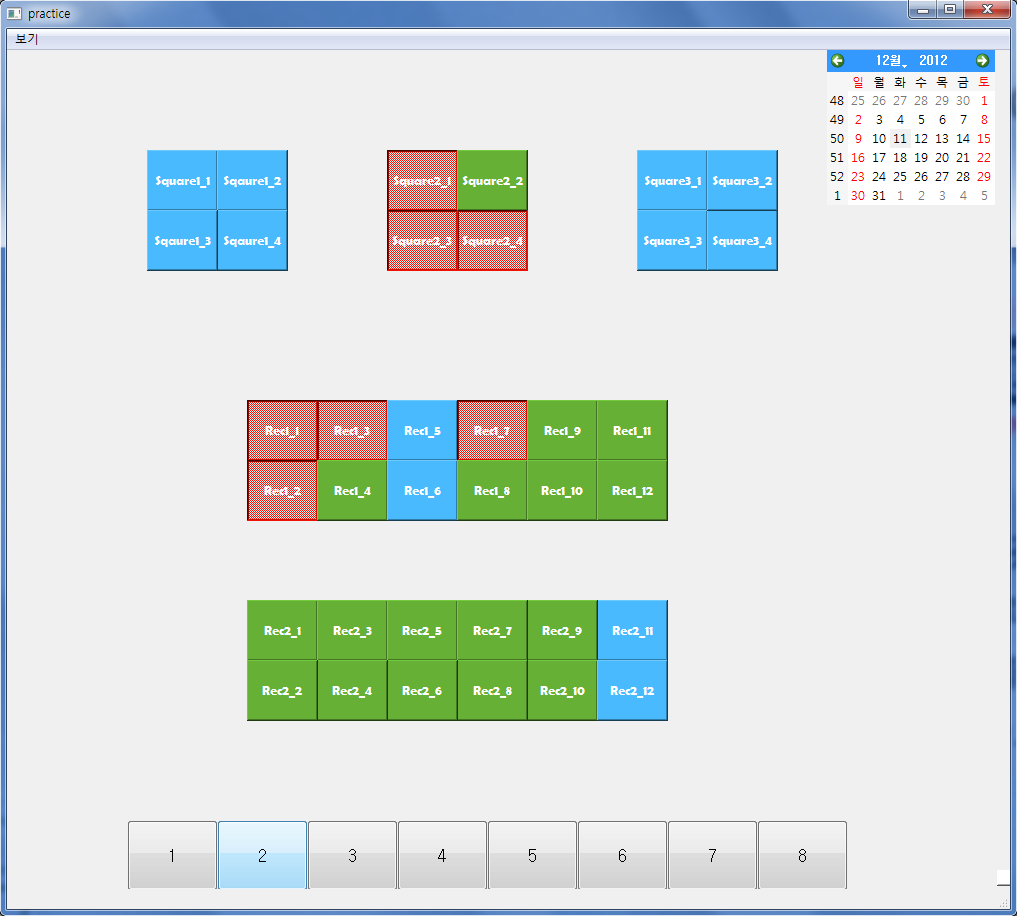
class Table

class Manager

또 테이블에 학생을 앉는 동작, 테이블에서 학생이 나가는 동작 등 실생활에서의 행동을 OOP적 관점에서 그대로 프로그래밍 하였고(Seat::On(), Seat::Off(), Table::studentIn(int, Student\*), Table::studentOut(int) 등등..) 세부 구현은 안으로 숨김으로 코드를 사용할 때 사용자가 이해하기 쉽고 사용하기 편하도록 프로그래밍 하였다. 또 멤버변수를 모두 private로 선언함으로써 클래스외부에서의 멤버변수 접근을 차단하고 정보보호기능을 높였다.

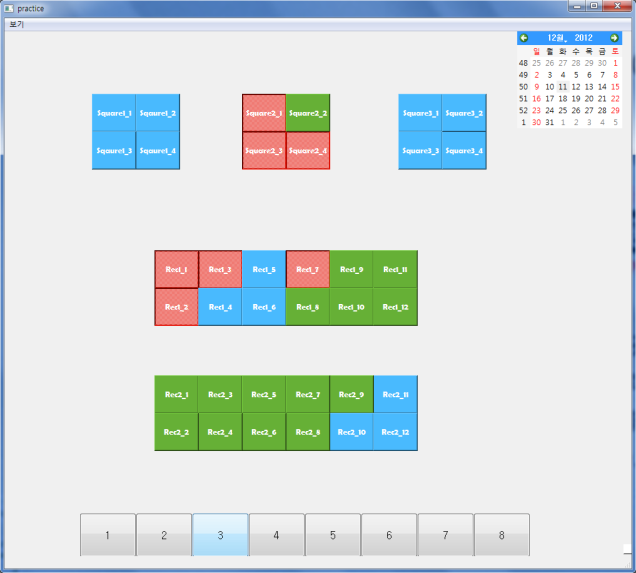
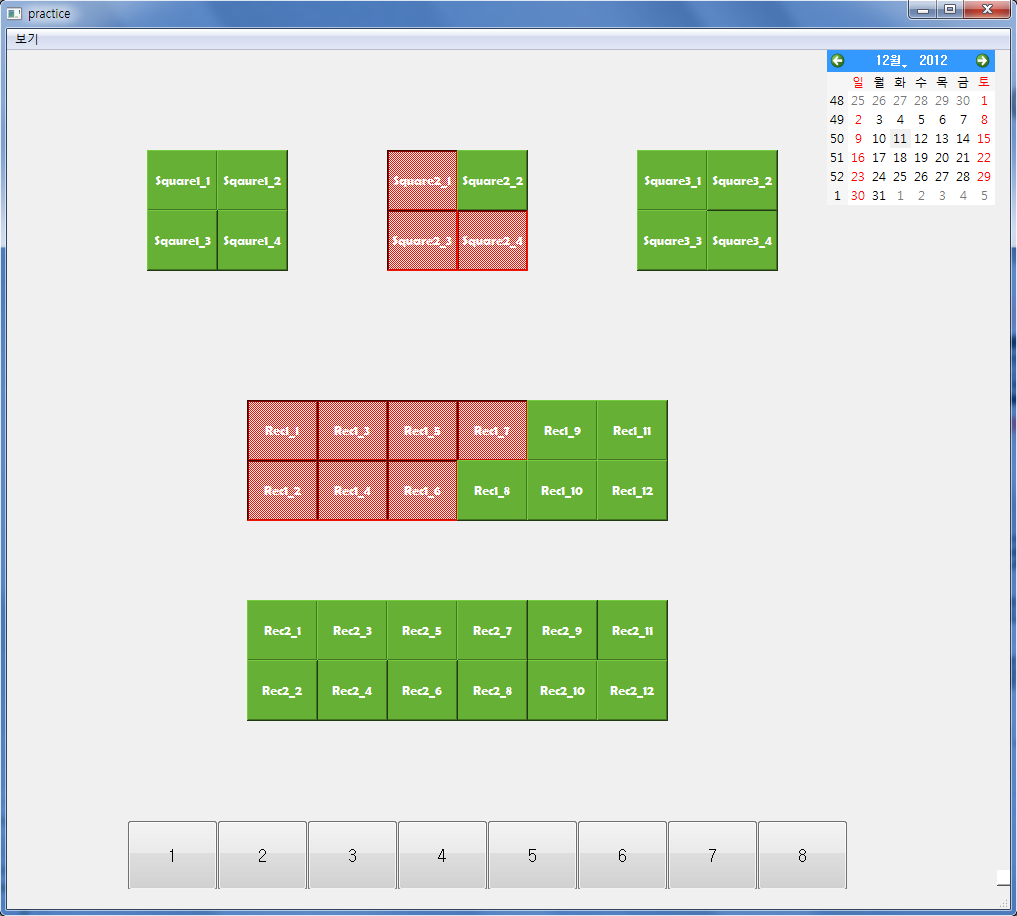
**(g) Execution results: show real examples of program execution**

 <- 초기 상태 가정

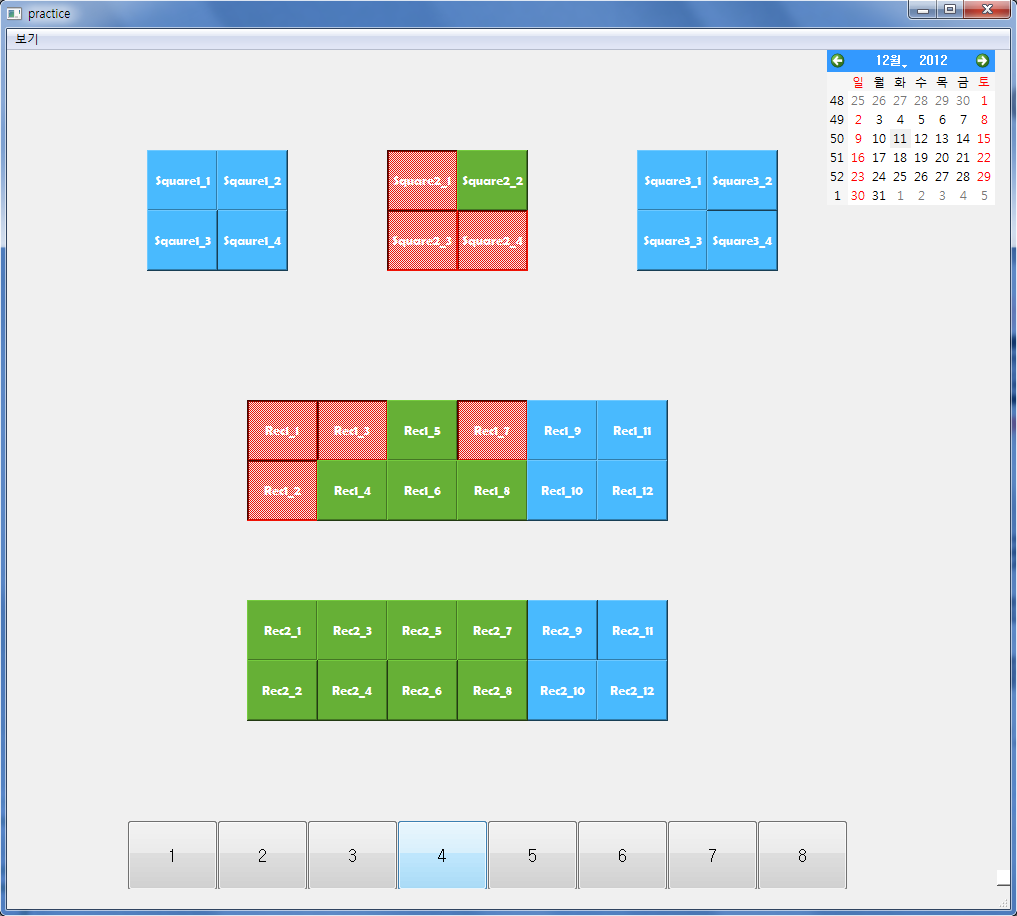
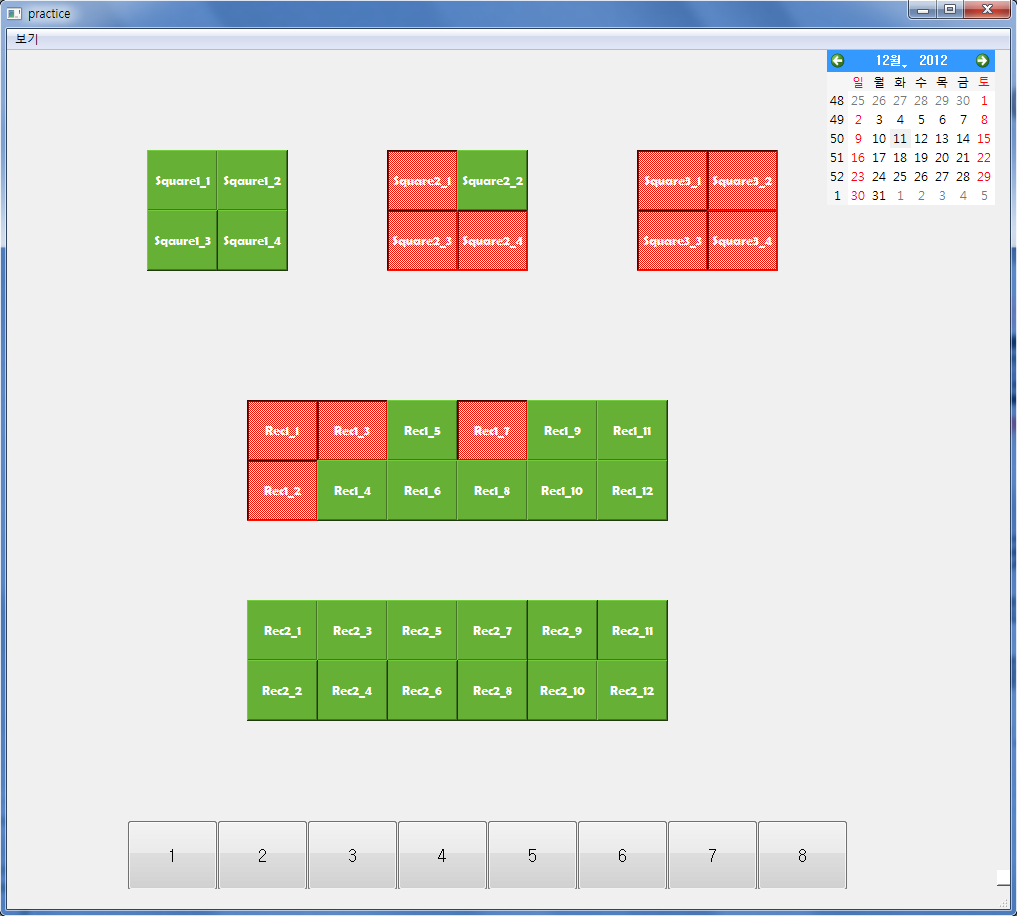
 

각각 1명, 2명을 눌렀을 때에 나오는 화면.

SquareTable은 1명일 때에는 할당되지 않으므로, 파란색으로 변하지 않는다. 1명의 경우, 1번째 RectTable에서 4번째 자리가 응집도가 가장 높아지므로, 해당 좌석을 파란색으로 변경한다. 2명의 경우, 5, 6번째 자리를 할당하는 것이 응집도가 가장 높으므로, 해당 좌석을 파란색으로 변경한다. 좌석이 파란색으로 변하지 않은 곳은, 클릭해봐도 다른 반응은 일어나지 않는다.

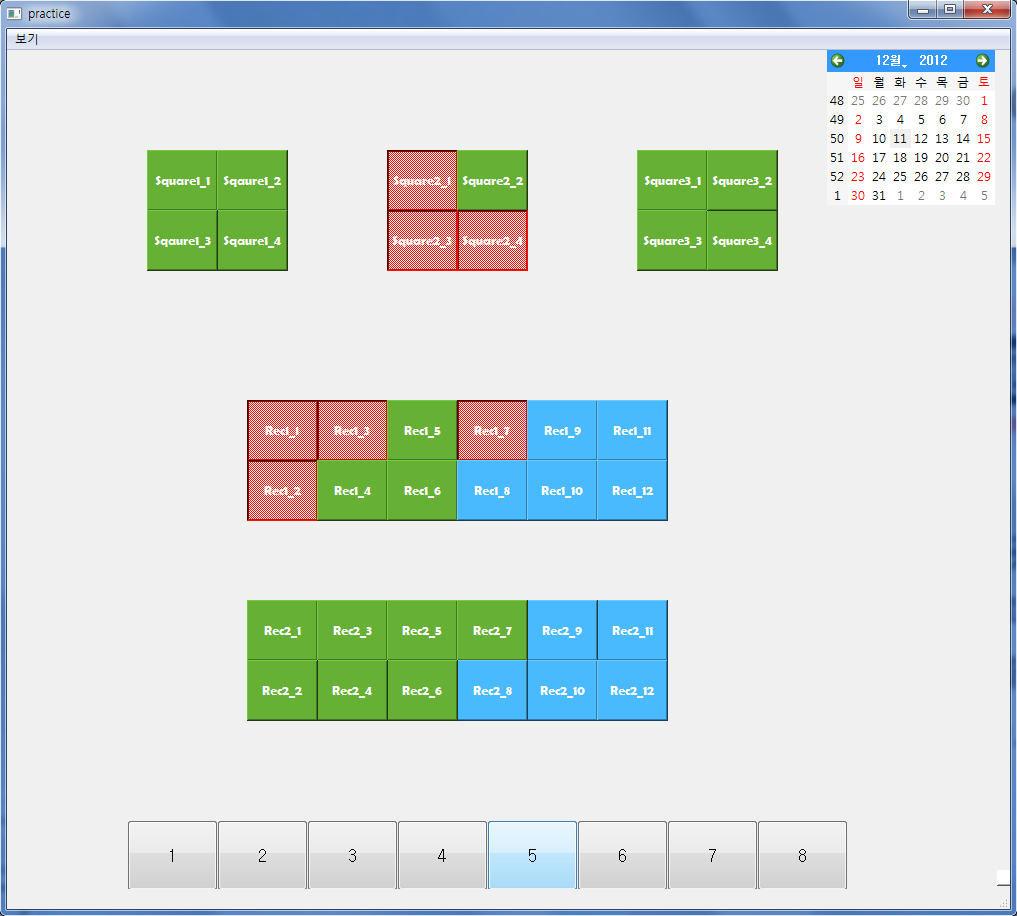
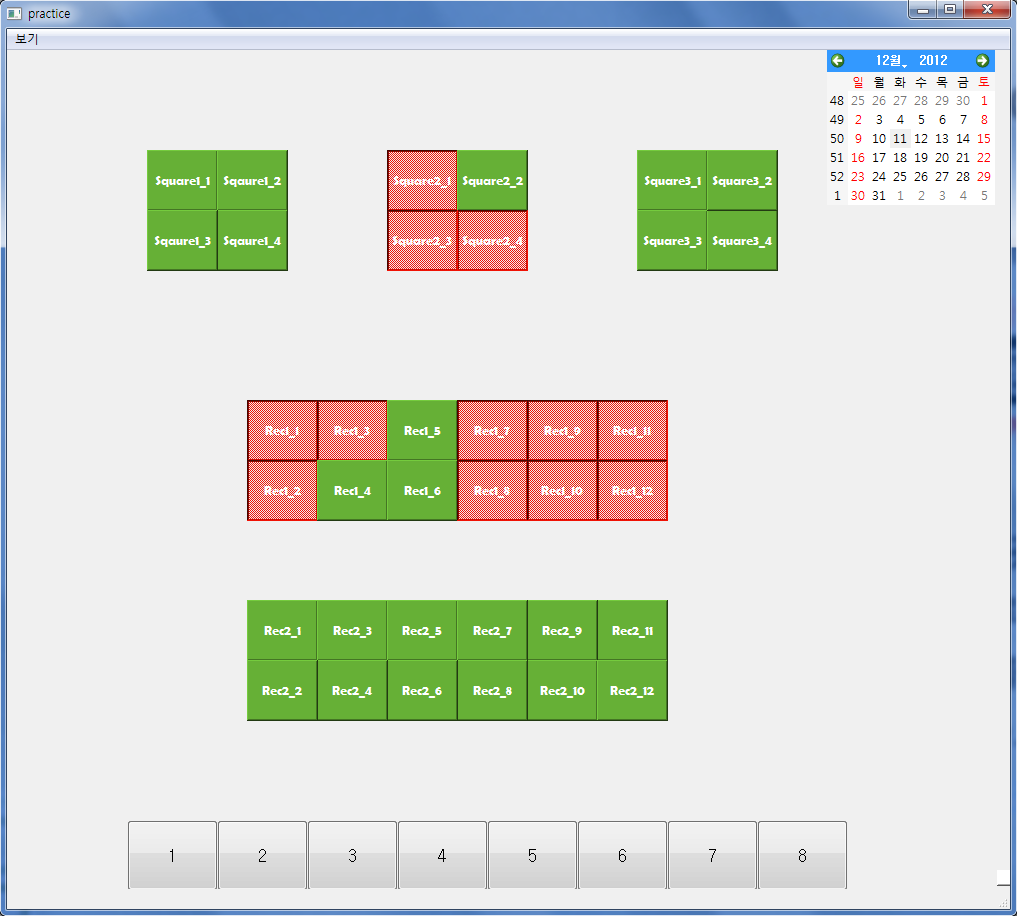
 

3명일 때의 경우이다. 첫 번째 RectTable의 파란색 좌석을 누르면, 이 그림과 같이 다른 곳의 지정은 풀리고, 파란색이었던 좌석을 빨간색으로 바꾸어, 자리를 할당해준다.

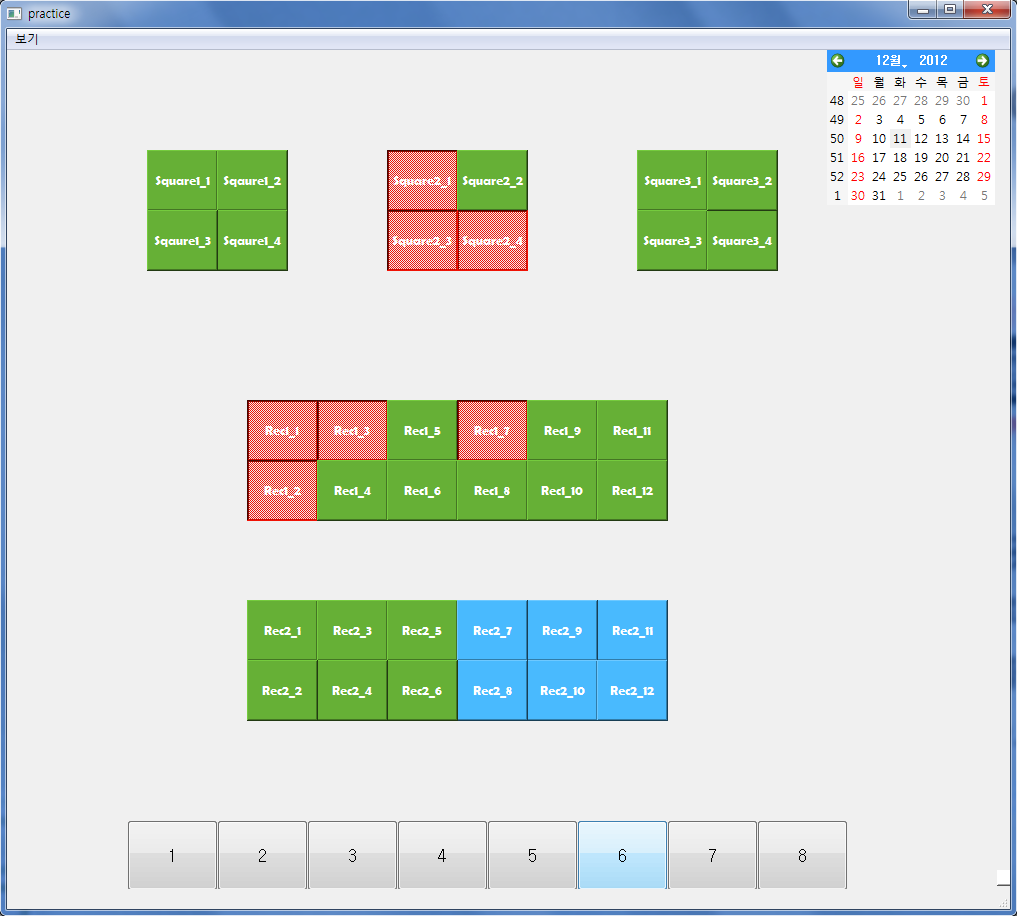
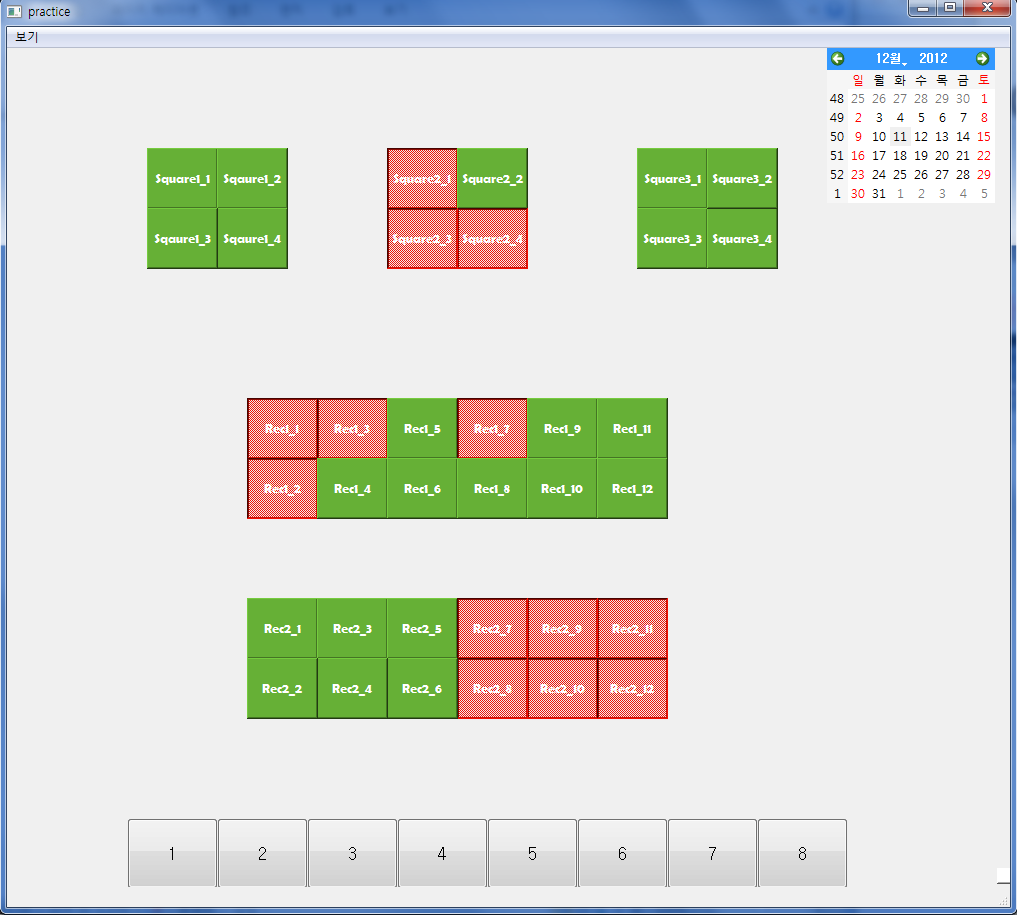
 

4명일 때의 경우이다. 3번째의 SquareTable에 좌석을 할당해준 모습이다. 첫 번째 RectTable에서, 좌석번호 4,5,6,8을 할당해주지 않은 것은, 응집도는 최대이나, 처음의 가정인 “짝수일 때에는 항상 직사각형의 형태로 자리를 할당한다” 를 어기기 때문이다.

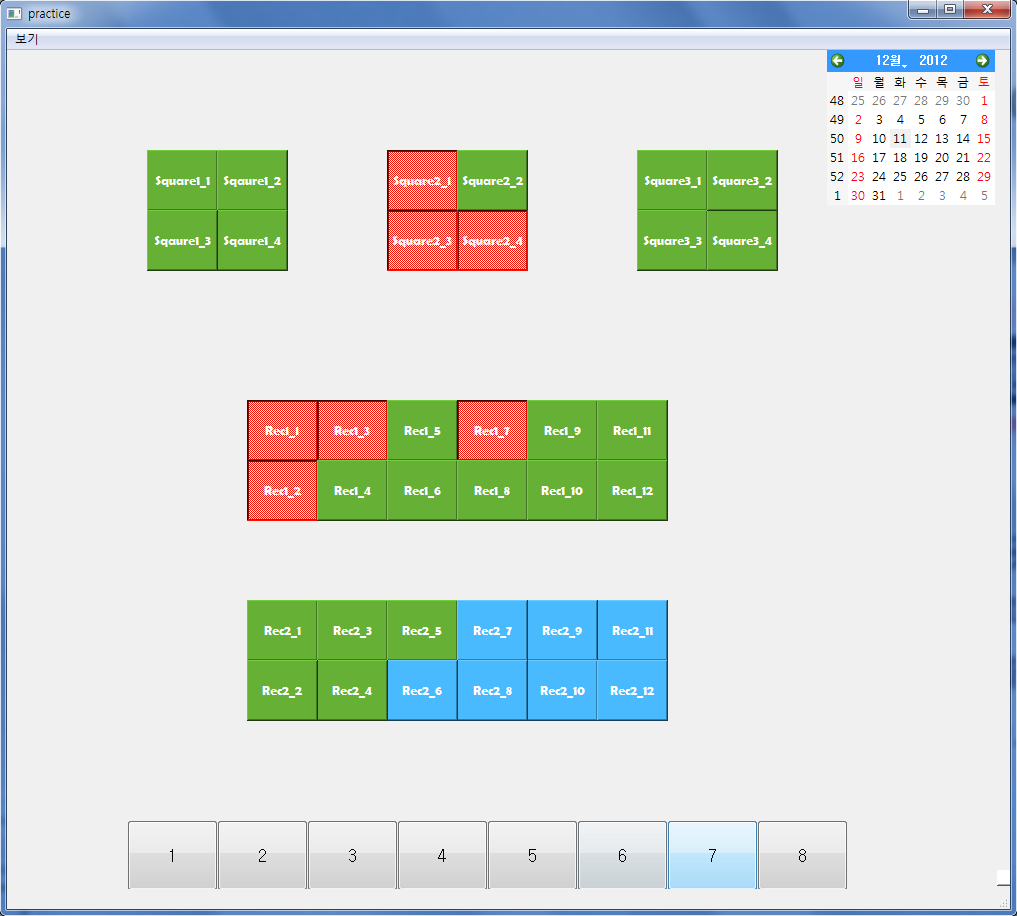
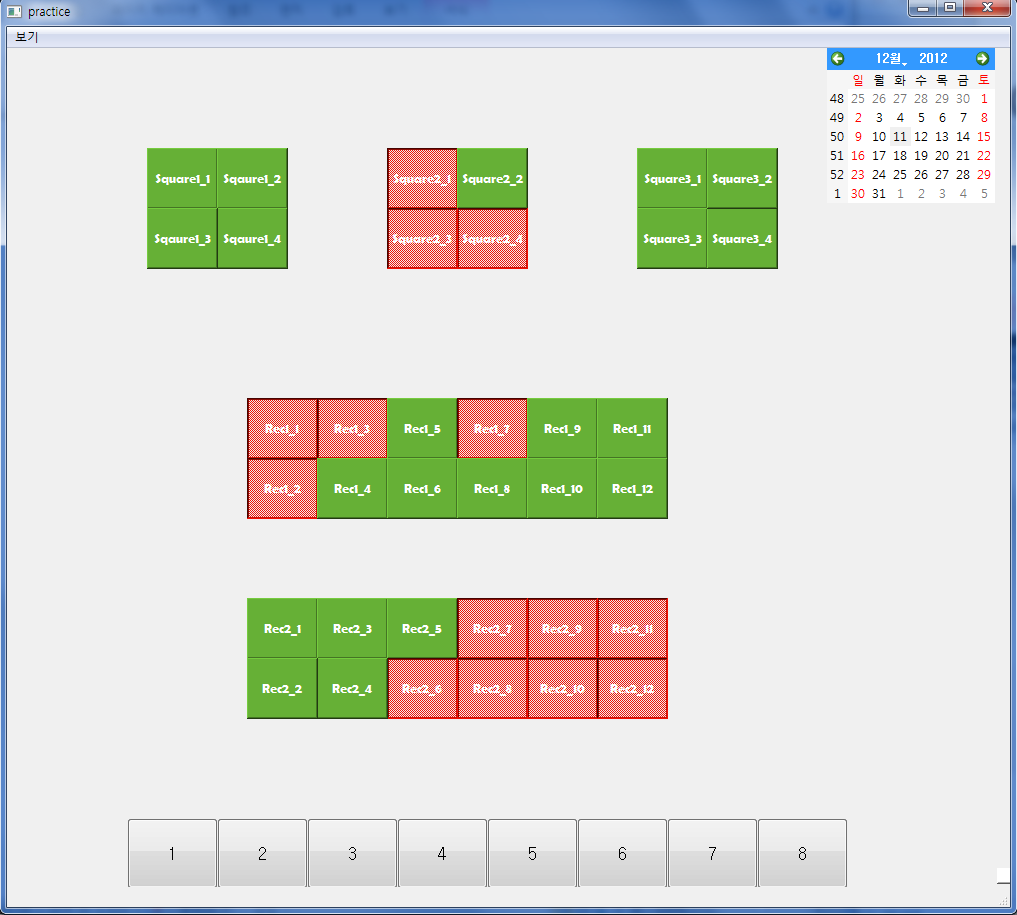
이 가정을 적용한 이유는, 실제 적용 사례에선 Table의 수가 2~3개만 있지 않을 것이고, 약 10~30개 이상의 Table이 있을 것이라고 생각했기 때문에, 비효율적인 배치방식은 출력 결과에서 제외하였기 때문이다.

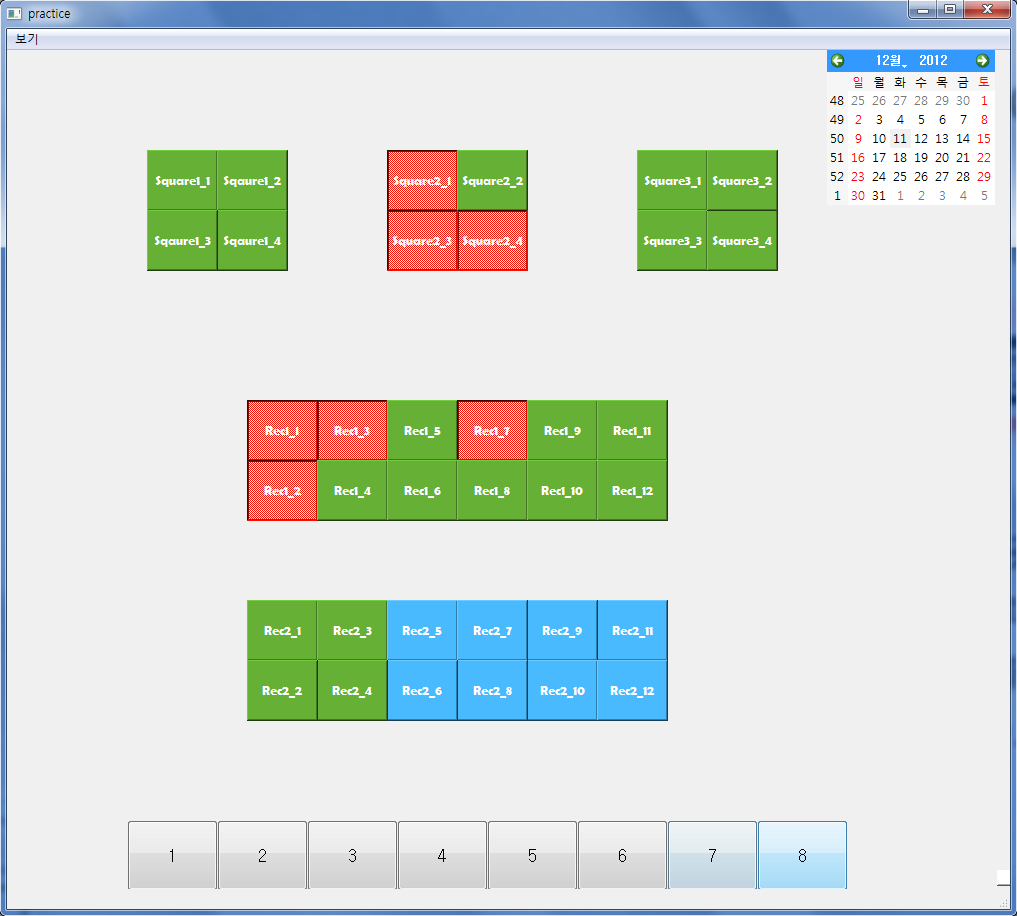
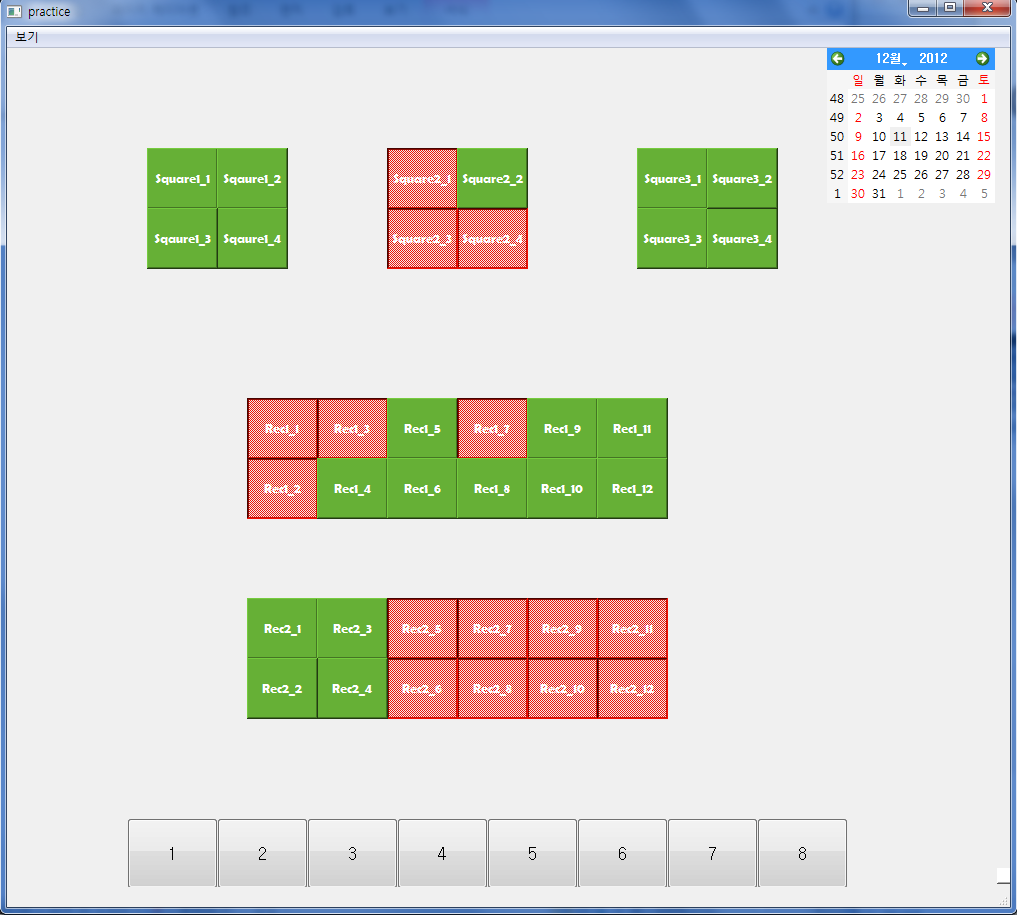
5명일 때의 경우이다. 사람 수가 4명을 넘으므로 SquareTable에는 할당되지 않고, RectTable 중 좌석이 여유가 있는 곳만 할당 가능한 좌석을 표시해준다. 예시에선 1번 RectTable에 할당해주었다.

6명일 때의 경우이다. 6개의 자리가 남에도 불구하고 1번째 RectTable에 할당 가능 좌석이 표시되지 않는 이유는, 마찬가지로 처음의 가정을 어기기 때문이다. 따라서 예시에선 2번 RectTable에 할당해주었다.

7명일 때의 경우이다. 마찬가지로 1번째 RectTable에는 7명 이상의 자리가 나지만, 처음의 가정 중 홀수의 경우인 “2명 이상 마주보는 사람 없이 가로로 배치하지 않는다” 를 어기기 때문이다. 따라서 예시에선 2번 RectTable에 할당해주었다.

8명일 때의 경우이다. 1번째 RectTable에는 가정을 어기므로 표시하지 않고, 2번 RectTable에만 할당 가능함을 볼 수 있다.

**(f) Conclusion**

조원들과 종종 교내에 있는 법학관 지하의 학생식당을 이용하면서 프로그램 구상을 해본 적이 있었습니다. 현 학생식당의 좌석은 점심 시간에 사람들이 드문드문 앉아있어 정작 음식은 받았지만 일행들과 같이 먹기 위해 자리에 서서 기다려야 하는 경우가 빈번했습니다. 그 때 당시는 어떻게 하면 시간의 효율성을 높이면서 법학관 좌석을 사람들에게 배치할 수 있을까에 대해 고민하였었는데, 이번 OOP Project 4 팀 과제를 통해 구현해볼 기회를 가질 수 있었습니다.

처음 구상했을 당시, 현 중앙대 중앙도서관에서 채택하고 있는 자리배석 프로그램의 인터페이스를 모델로 하여 프로그램을 구현하기로 하고 UML을 구상했습니다. 그러나 식당 배식 프로그램은 일반적인 자리배석 프로그램과는 달리 고려해야 될 변수들이 여럿 있었고 이를 해결하기 위해 많은 시간을 알고리즘을 짜는 데에 할애해야 했습니다.

GUI 부문도 난제였습니다. C++, Java 언어 중, C++을 VS2010 환경에서 개발하기로 한 저희 팀은 당장에 콘솔을 통한 도스화면 대신 윈도우 화면을 구현시켜줄 프로그램이 필요하였습니다. 그렇게 찾은 프로그램이 QT였고 이 역시 만만치 않았습니다. Library를 뒤져가면서 여러 시행착오를 겪었습니다.

이번 프로젝트에서 팀원들은 크게 Structure, Algorithm, GUI 3가지 파트로 작업을 분업해 프로젝트를 수행했습니다. 먼저 기본적인 Structure를 만들고, 이를 사용해 좌석을 효율적으로 배치해주는 Algorithm을 만들고 최종적으로 이를 GUI와 연동하여 자리 배석 프로그램을 완성하였습니다. 특히 Structure에서 객체지향의 관점에 맞게 코드를 짜는 것에 초점을 맞추었으며, 이를 위해 현실의 사물을 객체화하여 프로그래밍하고 상속과 Dynamic Binding 을 사용, maintenance를 높일 수 있는 방법을 기반으로 하여 코딩 함에 따라 OOP 철학에 충실하도록 노력하였습니다.