# REPORT


**7조 TOMTOM**

**12131819 육동현**

**12161613 왕현정**

**12163509 강인희**

**제출일 : 2018-11-27**

# 목차

1. 전략수립

2. 참여인원 및 역할분담

3. Agent 상세 설계 및 구현방법 기술

4. 시행착오 및 분석

5. 차후 개선 사항

6. 결론

# 1.전략수립

## 1) TAC-SCM에 대한 이해

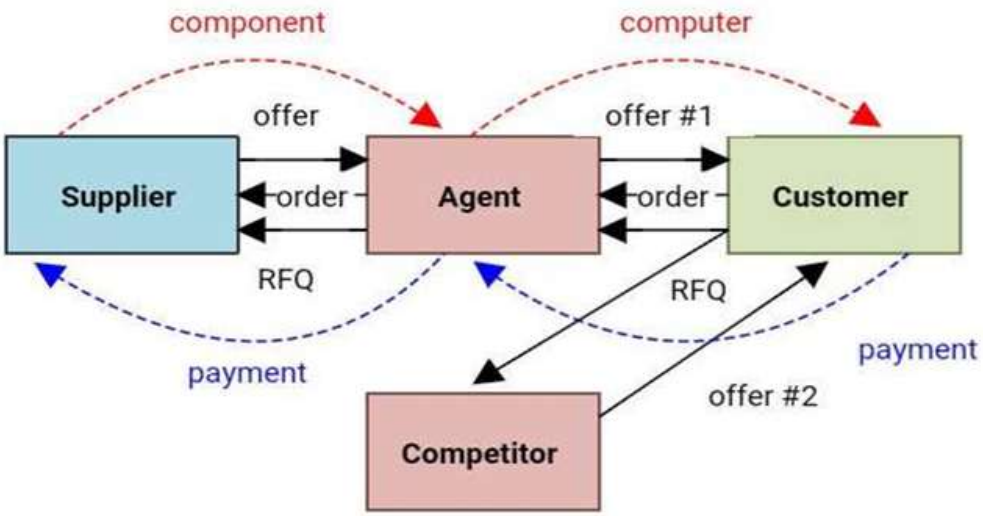
### A. Supply Chain이란?


공급사슬은 판매자와 구매자 관계의 연쇄로 이루어짐

TAC-SCM에서 우리는 컴퓨터 제조회사의 역할을 수행

Agent는 가격협상과 재고관리를 통해 이윤극대화를 추구함

### B. TAC-SCM의 프로세스 이해




RFQ(견적서) - 이거 사고 싶은데 누가 팔래?

Offer(제안서) - 저는 이 가격에 팔 수 있습니다!

Order(주문서) - 그래 너한테 살게!

주문정보흐름– (RFQ → Offer → Order)

### 2)기본전략 소개

저희 조는 Customer에서 제조회사를 선택할 때 가장 큰 고려요소는"가격"이라는 생각 하에 기본적으로 Customer의 주문성사율에 따라 가격을 조정해 나가는 방안을 채택하기로 전략방향을 정했습니다. 또한 저희는 초기에는 가격우위를 잡기 위해 초기에 가격을 낮추지만, 일정한 가격 이하로는 가격을 낮추지 않도록 설계하였습니다.
여기에서는 이렇게 간략하게 저희의 전략방향을 소개하도록 하겠고, 보다 자세한 구현 방법에 대해서는 뒤에서 자세히 다루겠습니다.

# 2.참여인원 및 역할분담

저희 7조는육동현, 왕현정, 강인희, 총 3명이 팀 프로젝트에 참여하였습니다.
육동현은 조장으로서 전체적인 프로그램 설계 및 팀 프로젝트 총괄을 담당하였고, 왕현정은 각 요인을 여러 차례 시뮬레이션하며 최적의 값을 갖도록 모델을 점차 다듬는 업무를 맡았습니다. 또한 강인희는 시뮬레이션 자료를 분석하여 추후 프로그램 개선에 피드백을 주는 역할과 보고서 작성업무를 맡았습니다.

# 3.Agent 상세설계 및 구현방법 기술

## 1) 프로그램의 기본적인 구조설명

A. 공장의 총 capacity는 2000으로 제한되어 있으므로 이를 넘지 않도록 컨트롤 하기 위해서 아래의 2가지 변수를 이용하여 공장가동률을 조정하였습니다.

```
private double unitProdCycle; // 단위 제품 만드는데 요구되는 사이클
private double totProdCycle; // 현재 전체 생산 사이클
```

B. PPT에서 주어진 자료를 근거로 SKU 별로 구매원가와 사이클에 대한 정보를 배열에 저장하여 의사결정에 활용할 수 있게 하였습니다.

```
/** Given value from PPT */
// 1) SKU에 따른 상품원가
//    SKU #1 = 필요부품{100, 200, 300, 400} = 1000 + 250 + 100 + 300 = 1650
private int[] SKU_price = {-1,1650,1750,1750,1850,2150,2250,2250,
                           2350,1650,1750,1750,1850,2150,2250,2250,2350};
// 2) SKU에 따른 필요 사이클
//    SKU #1 = 4 unitProdCycle (마찬가지로 뒤의 것도 구함)
private int[] SKU_unitProdCycle = {-1,4,5,5,6,5,6,6,7,4,5,5,6,5,6,6,7};
```

**C. 보낸 오퍼 수와 받은 오더의 비율을 통해서 저희가 다른 에이전트에 비해 가격경쟁력이 있는지 지속적으로 체크하며 할인율을 조정하기 위해 아래 변수를 이용하였습니다.**

```
private double[] offerLog; // 보낸 오퍼관리
private double[] orderLog; // 받은 오더관리
private double hitRatio; // 주문성사율 = 받은 오더/보낸 오퍼

// 목적: 다른 Agent보다 많은 order를 따 내야 높은 수익 달성가능
// order 많으면 많은 주문 땀, offer 많으면 견적서 보냈으나 주문 못 땀 -> hitRatio에 따라 할인율 조정
hitRatio = orderLog[transCnt-1]/offerLog[transCnt-2];

// 주문성사율이 100%이면 우리는 아쉬울게 없으므로 비싸게 받음
if (hitRatio == 1.0)
    priceDiscountFactor += 0.03;
else
{
    // 전략: 주문성사율이 높으면 할인율을 낮추고, 낮으면 할인율을 높이는 전략 -> 비율은 수능등급 이용
    if(hitRatio >= 0.96)
        priceDiscountFactor -= 0.001; // 1등급 컷
    else if(hitRatio >= 0.89)
        priceDiscountFactor -= 0.003; // 2등급 컷
    else if(hitRatio >= 0.77)
        priceDiscountFactor -= 0.010; // 3등급 컷
    else if(hitRatio >= 0.60)
        priceDiscountFactor -= 0.015; // 4등급 컷
    else if(hitRatio >= 0.40)
        priceDiscountFactor -= 0.017; // 5등급 컷
    else if(hitRatio >= 0.23)
        priceDiscountFactor -= 0.020; // 6등급 컷
    else if(hitRatio >= 0.11)
        priceDiscountFactor -= 0.025; // 7등급 컷
    else if(hitRatio >= 0.04)
        priceDiscountFactor -= 0.030; // 8등급 컷
    else
        priceDiscountFactor -= 0.050; // 9등급 컷
}
```

**offerLog와 orderLog를 이용하여 hitRatio를 조절하도록 설계하였습니다.**
**hitRatio가 너무 낮으면, 즉 offer는 많이 보냈는데 order는 적게 오면,**
**우리 쪽이 가격이 비싸다고 유추할 수 있으므로 가격을 조정하는 조치를 단행했습니다.**
**hitRatio가 1.0에 가까우면 우리 가격이 낮으므로 가격을 올리거나 가격 할인 폭을 줄이도록 했습니다.**

**D. 상품원가를 B에서 말씀 드린 배열에서 SKU에 따라 받아오고 RFQ를 선택할지 말지를 제어하기 위해서 selectFilter라는 변수를 이용하였습니다.**

```
private int basePrice; // 상품원가 (공급자로부터 구매한 부품가격)
private int selectFilter = 400; // RFQ 걸러내는 기준 #보내는 오퍼의 개수 조절
```

### 2) 코드 구현방법 기술

**Product에 따라서 저희가 얻을 이익부분을 고려하는 것이 가장 중요했기 때문에, 가격을 조정하는 부분이 가장 주요하게 다뤄져야 할 부분이라고 생각했습니다. 따라서 저희는 Customer로부터 주문을 받을 때, 이를 받아들일 선택의 기준을 아래와 같이 마련하였고 이를 시행착오를 통해 점차 다듬어가는 방향을 취했습니다.**

**a) 원가 x 마진비율 일정한 수준 이상일 때 접수**
**b) 마진이 150~ 400(고정 마진 값의 범위) 이상일 때만 접수**
**c) Range의 high /mid /low에 따라 할인 비율을 다르게 조정하여 주문 접수**
**d)Product의 가격대에 따라 할인 비율을 다르게 조정하여 주문 접수**

### 3) 시뮬레이션 예측

**"b) 마진이 150~400(고정 마진 값의 범위) 이상일 때만 접수"와 같이 고정된 값으로 설정하는 것보다 "a) 원가x 마진비율 일정한 수준 이상일때만 접수"**
**또는"c) Range의 high /mid /low에 따라 할인 비율을 다르게 조정하여 주문 접수" 같이 주문에 대한 할인 및 마진 비율을 달리 하는 것이 더 효율적일 것이라고 예측하였습니다.**

# 4. 시행착오 및 분석

## a) 원가x 마진비율 일정한 수준 이상일 때만 접수

```
selectFilter= (int)(basePrice*0.25);   // 원가의 0.25이상만 받기로 설정
```

**예상보다 시뮬레이션의 결과는 매우 불안정하였으며, 실행 중 오류도 빈번히 일어났습니다. 결과로 도출해낼 만한 규칙적 결과를 나타내지 못했으며 오히려 다른 방식을 사용할 때보다 결과적 수치는 항상 아쉬웠습니다.**

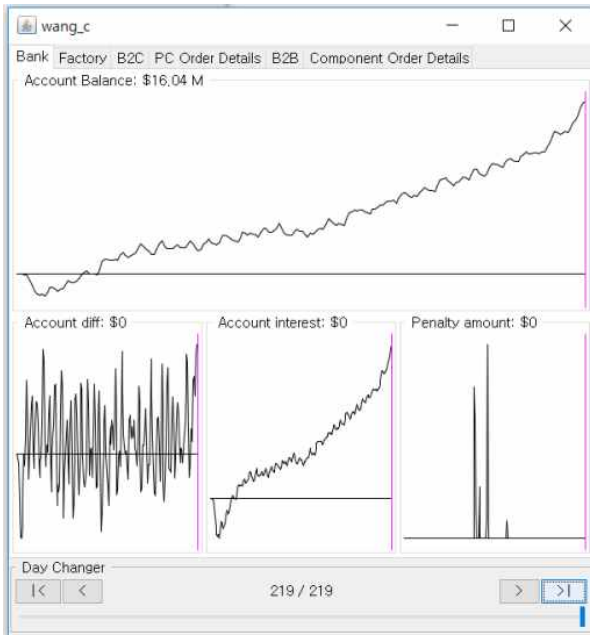
## b) 마진이 150~ 400(고정 마진 값의 범위) 이상일 때만 접수

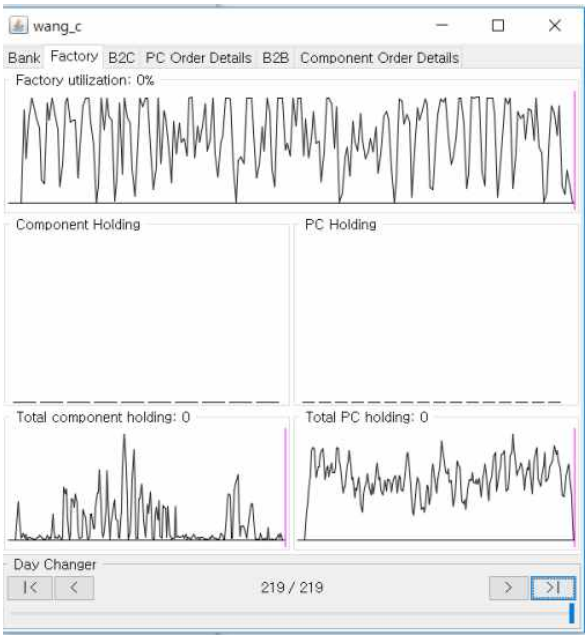
**150~400 사이의 다양한 값을 설정하여 실행해보았는데, 대표적으로 가장 결과값이 좋게 나타났던 150, 400, 2가지 case를 세부적으로 분석하겠습니다.**

**b-1) selectFilter=150, discount=0.02**

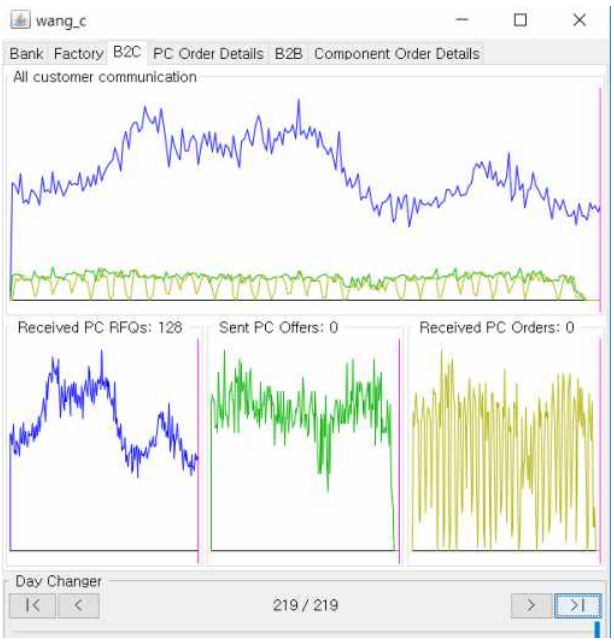
**(150은 고정 마진, 각 상품에 대한 할인율은 2%)**

**<bank><factory>**



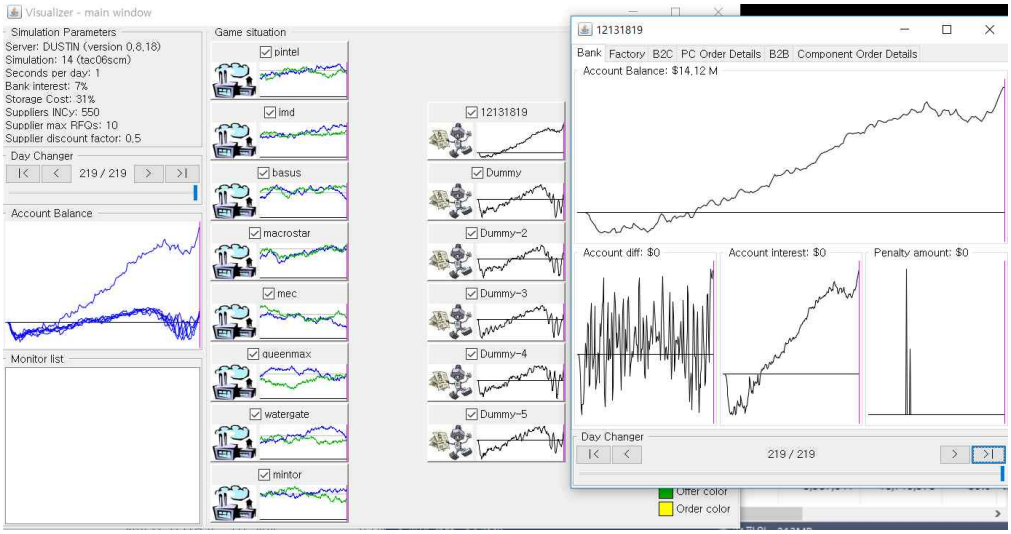

**<B2C>**




**b-2)selectFilter=400, discount=0.02**

**(400은 고정 마진, 각 상품에 대한 할인율은 2%)**

**<bank>**

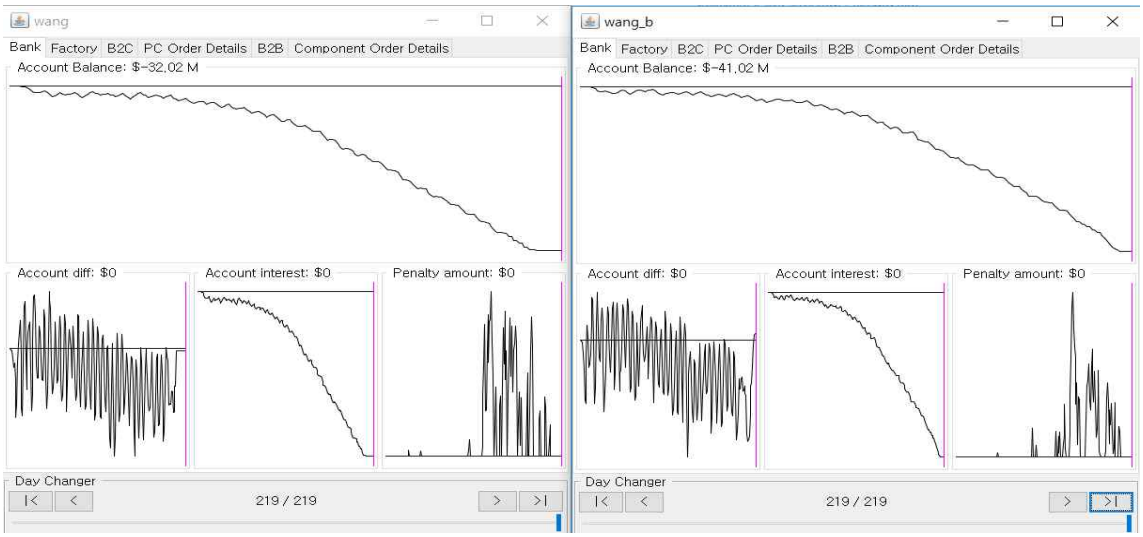



**<factory><B2C>**

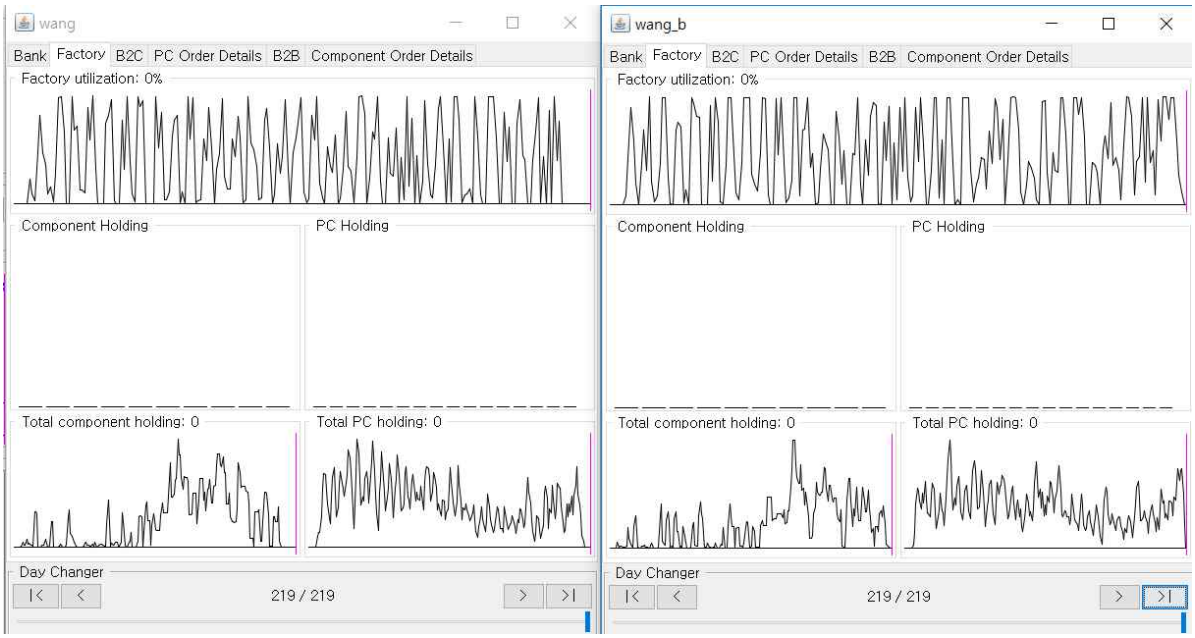

**b-3) 150과 400 동시 실행 비교**

왼쪽이a의 경우, 오른쪽이 b의 경우인데 400인 경우가 수익적인 면에서 우세한 모습이 나타났으며, 코드가 유사해서 진행상황은 비슷하게 나타나지만 400인 경우에 a의 경우보다 늦게 힘을 발휘하는 경향이 있지만 a의 penalty수치 빈도수보다 적은 경향이 있다는 것을 발견하였습니다.즉, 게임이 진행될수록 수익은 안정적으로 늘어나고, penalty의 양도 적은 것을 볼 수 있습니다.

<bank>




<factory>




<B2C>


## c)Range에 따라 할인비율을 다르게 적용

**High Range 를 주 타겟으로 하여 High Range 경우의 할인율을 가장 높게설정했습니다.**

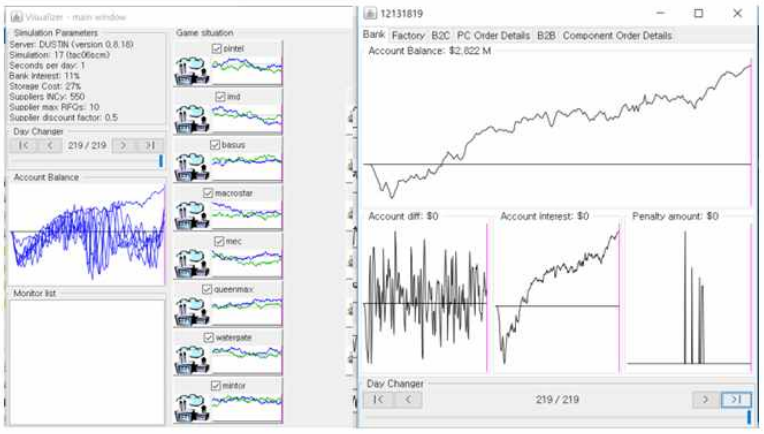
**그 다음으로 low 와 mid 를 0.05 씩 순차적으로 할인율을 낮추어 코드를 개발해보았습니다.그 후론 타겟설정을 Middle Range 로 바꿔서 실행해보았습니다.**

### c-1)high range 타겟

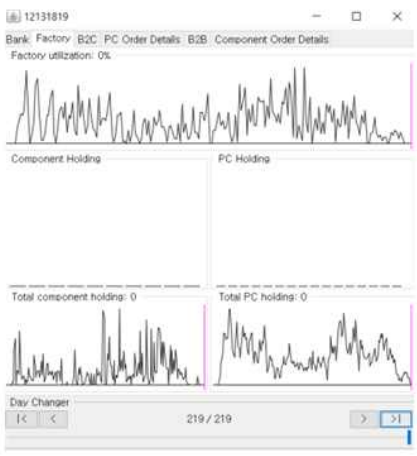
**selectFilter400 , discount 0.02 , low, mid, high 할인율 각각 0.2, 0.25, 0.15**

```
   Low Range
if(basePrice==SKU_price[1] || basePrice==SKU_price[2] || basePrice==SKU_price[9] || basePrice==SKU_price[10] || basePrice==SKU_price[11])
    selectFilter=(int)(basePrice * 0.2);
/* Mid Range */
 else if(basePrice==SKU_price[3] || basePrice==SKU_price[4] || basePrice==SKU_price[5] || basePrice==SKU_price[12] || basePrice==SKU_price[13] || basePrice==SKU_price[14])
    selectFilter=(int)(basePrice * 0.25);
/* High Range */
 else
    selectFilter=(int)(basePrice * 0.15);
```

<bank>




<factory>

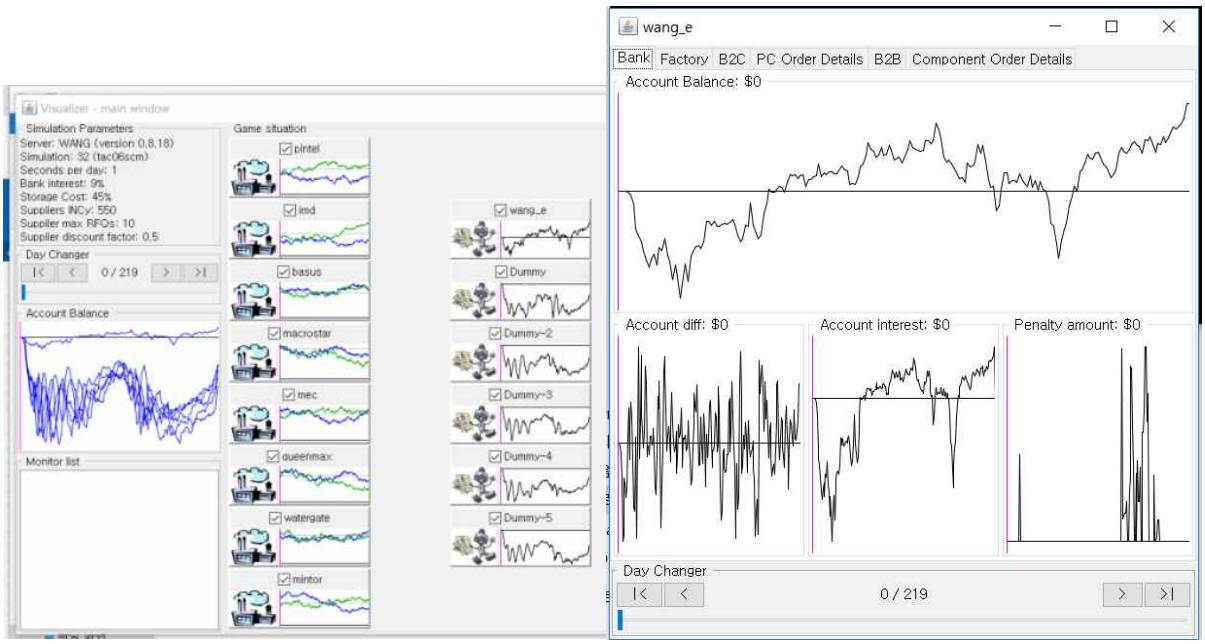


## c-2) middlerange타겟

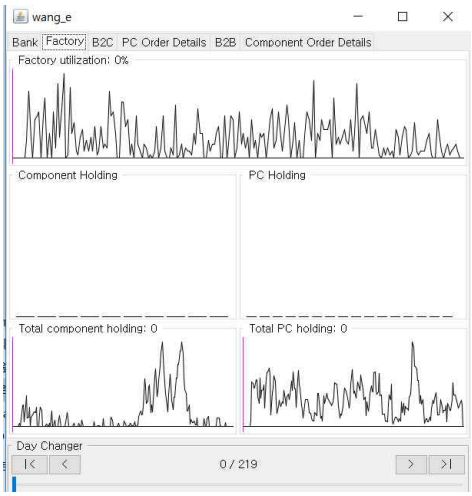
## selectFilter=400 , discount 0.02 , low, mid, high 할인율 각각 0.2, 0.25, 0.3

```
   Low Range
if(basePrice==SKU_price[1] || basePrice==SKU_price[2] || basePrice==SKU_price[9] || basePrice==SKU_price[10] || basePrice==SKU_price[11])
    selectFilter=(int)(basePrice * 0.2);
/* Mid Range */
 else if(basePrice==SKU_price[3] || basePrice==SKU_price[4] || basePrice==SKU_price[5] || basePrice==SKU_price[12] || basePrice==SKU_price[13] || basePrice==SKU_price[14]
    selectFilter=(int)(basePrice * 0.25);
/* High Range */
 else
    selectFilter=(int)(basePrice * 0.3);
```

<bank>




<factory>




수익은 각각 2.822M, 530.453으로 range의 주요 focus를 high range로 맞춘 경우 결과가 훨씬 좋았습니다.

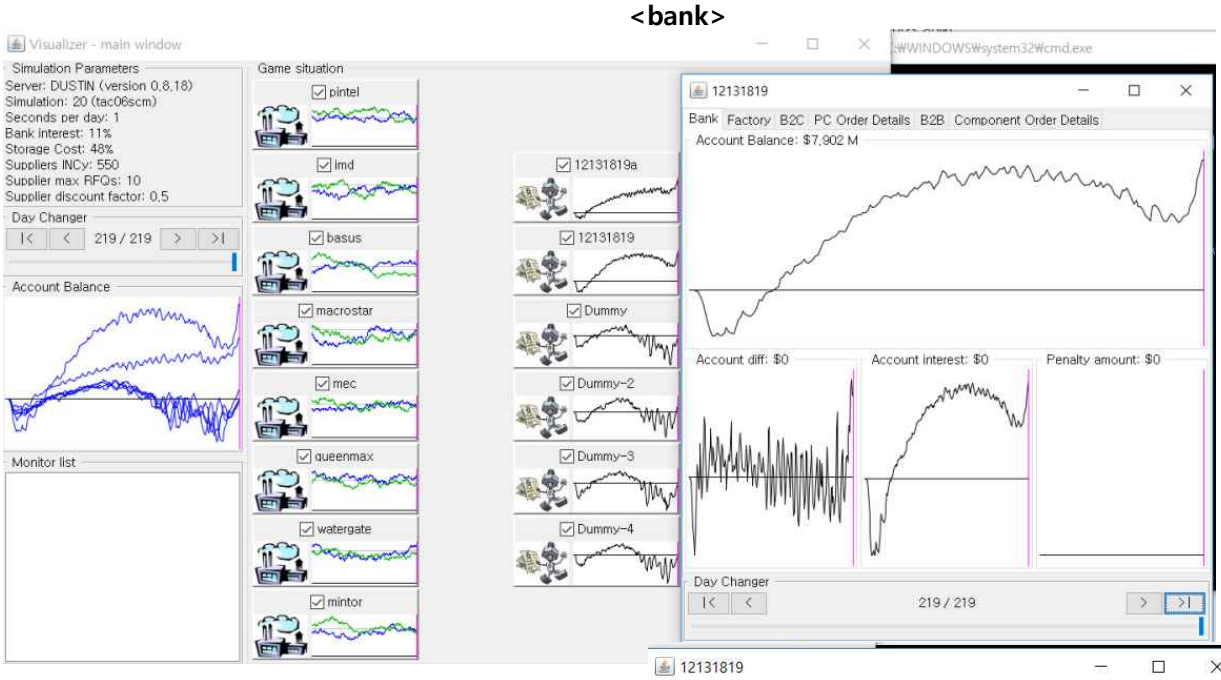
이로써 high에 초점을 두는 것이 유리하다는 결론이 나왔습니다.

## d) Product의 가격대에 따라 할인 비율을 다르게 조정하여 주문 접수

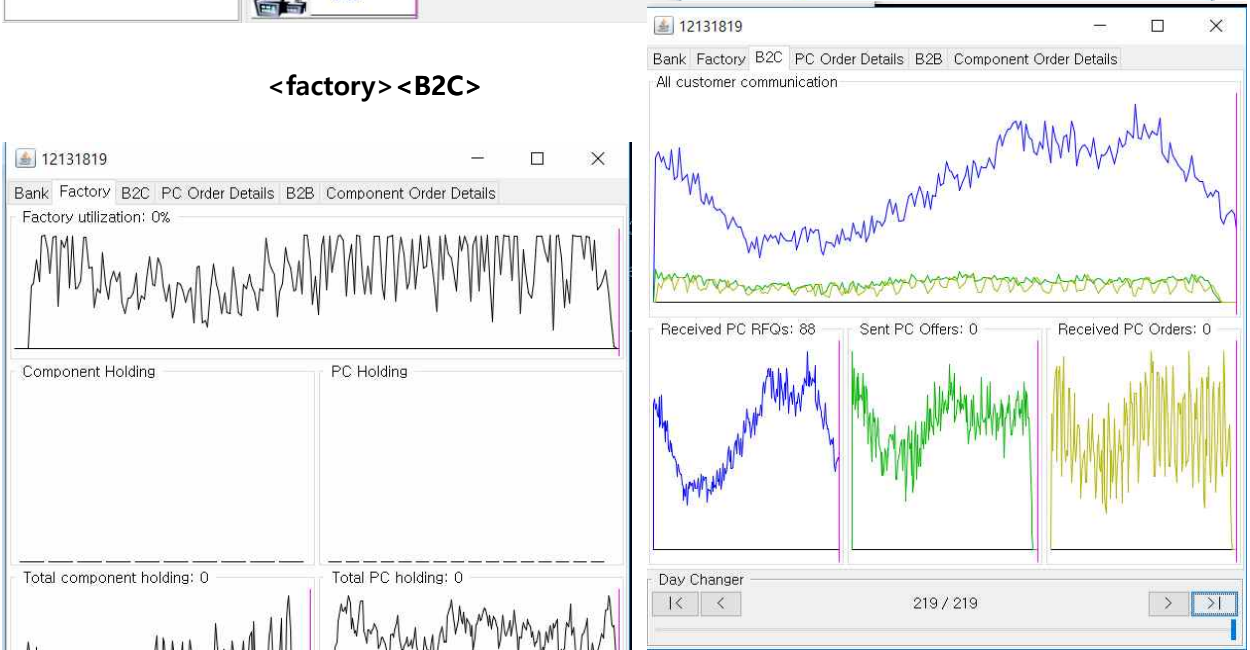
### 첫 번째 시행의 경우

```
    //low
if(basePrice >= 1650 && basePrice <= 1750)
 selectFilter=(int)(basePrice * 0.10);
    //mid
else if(basePrice > 1750 && basePrice < 2250)
  selectFilter=(int)(basePrice * 0.10);
  //high
else
 selectFilter=(int)(basePrice * 0.05);
```

<bank>



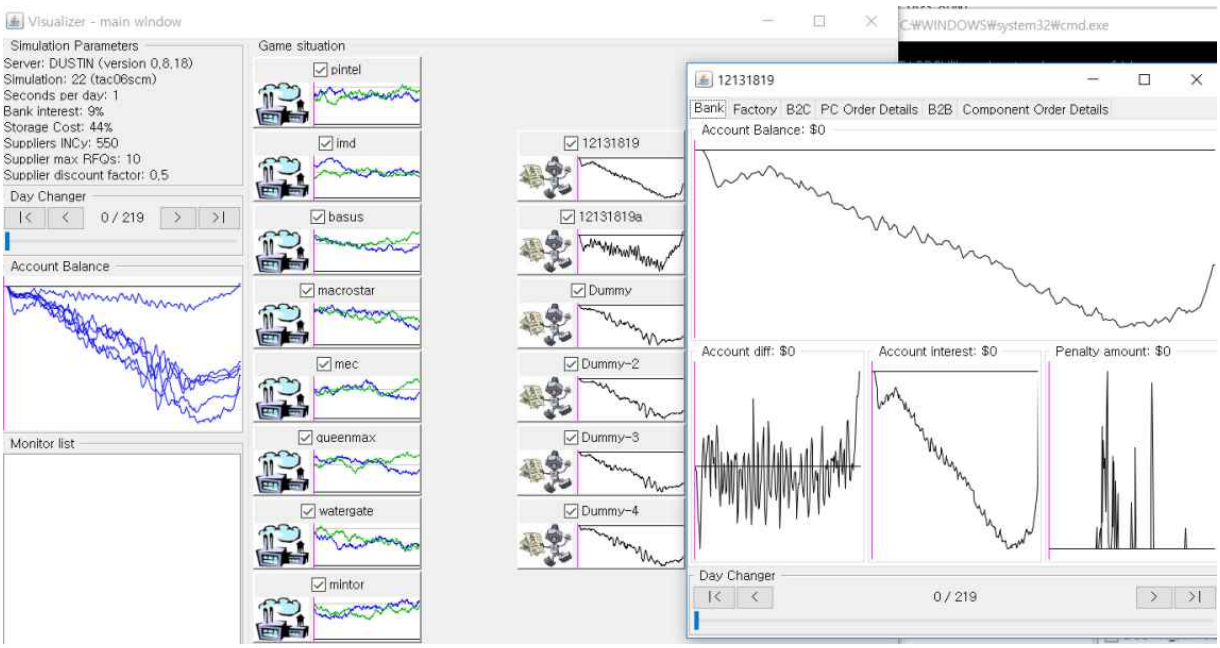

<factory><B2C>



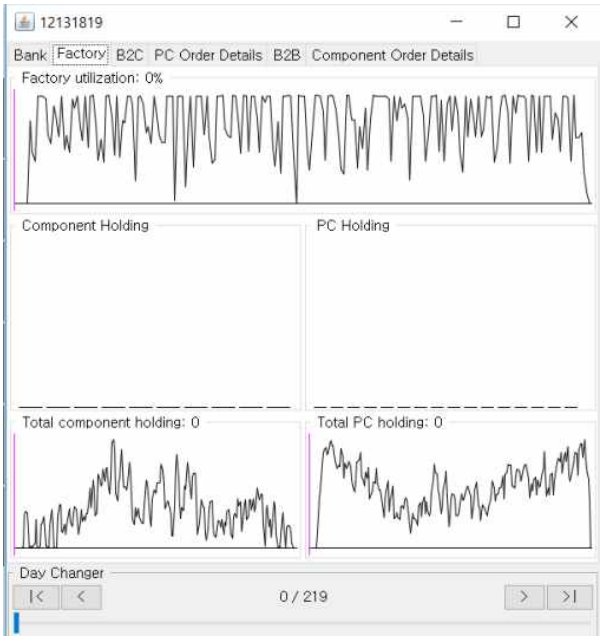
## 두 번째 시행 경우

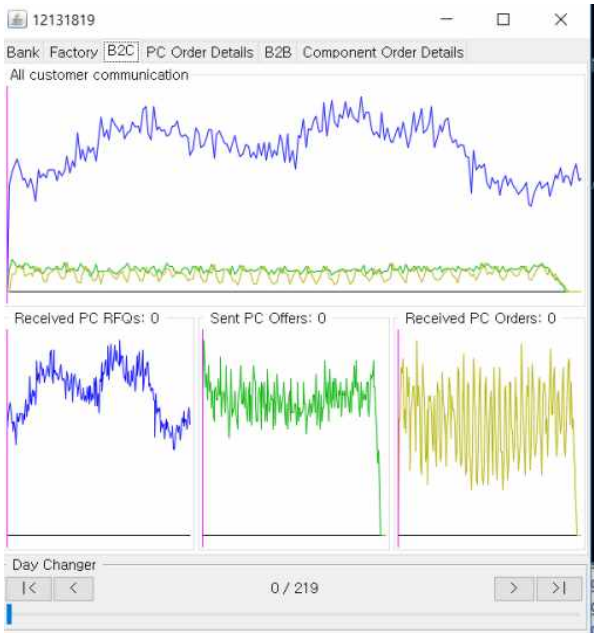
```
//low
if(basePrice >= 1650 && basePrice <= 1750)
   selectFilter=(int)(basePrice * 0.05);
//mid
else if(basePrice > 1750 && basePrice < 2250)
   selectFilter=(int)(basePrice * 0.01);
//high
else
   selectFilter=(int)(basePrice * 0.01);
```

**<bank>**



<factory><B2C>




**시행착오를 통한 결론**

# 5.추후 개선 방향

**주요 문제점으로는퍼포먼스가 가장 안정된 것을 찾으려 했지만 굉장히 불규칙적 양상으로 게임이 진행되었기 때문에 이를 발견하는 것이 쉽지 않았습니다.이로 인해패널티수치가 종종 높게 나타나는 경우가 있었고, 이로 인해 수익이 안정적으로 성장하지 못하는 문제점을 인지하게 되었습니다. 또한 너무 가격에만 초점을 둔 코드를 짜서인지재고관리를 비롯한 다른 부분은 관리가 덜되었던 부분은 한계점으로 남게 되었습니다. 추후 이 코드를 개선하기 위해서는 재고확보를 체크하는 코드를 좀 더 보완하는 방향으로 설계해야겠다는 점과 상황의 안정성을 유지할 수 있는 변수 및 코드를 추가적으로 구현하면 좋겠다는 생각이 들었습니다.**

# 6.결론

**공급사슬관리를 직접 코드로 구현해보고 시뮬레이션을 하면서 공급사슬을 관리하기 위해서는 많은 변수들을 고려해야 하고 최적화를 위해서는 여러 가지 요인을 적절히**

조절해야 최선의 결론을 얻을 수 있다는 것을 깨달았습니다. 또한 생각으로만 모델링 한 것이 실제 시뮬레이션 결과와 다른 점을 보면서 여러 가지 요소가 복합적으로 연관되어 있는 상황에서는 생각보다 최적화가 어렵다는 것을 깨달았습니다. 또한 이 시뮬레이션은 컴퓨터 제조회사만을 상정해서 만들었다고 전제되어 있고, 많은 요소가 주어진 상태로 시작하였지만 실제로는 여러 가지 산업이 존재하고 이에 따라 공급자, 소비자의 관계가 훨씬 더 복잡하게 얽혀있는 것을 고려하지 않았는데도 이렇게 공급사슬관리가 어려운 것을 보면서 실제로 공급사슬관리 소프트웨어, 가령 삼성 SDS 의 첼로시스템과 같은 시스템은 어떻게 구현하고 얼마나 정확한지 어떻게 동작하는 지 등 많은 의문이 들었습니다. 이번 기회를 통해 공급사슬관리의 복잡성과 발견적 해법의 중요성을 알게 되었습니다.