Smart DJ CoffeeMaker

OOAD & SASD FINAL.

T1

200611499 이낙원 , 200611521 최정명
200611460 김정태 , 200611481 송준현

1. OOAD 와 SASD

A. OOAD란?

시스템을 Object들 간의 상호작용을 통해 분석하고 설계하는 소프트웨어 개발 방법론이다. OOAD기법에서는 사용자의 요구사항을 통해 사용자가 시스템을 사용하는 방법인 Use Case를 도출해 내는 것으로부터 개발과정을 시작한다. OOAD의 최종 목표는 Class Diagram을 만드는 것인데 여기서 Class는 Object를 추상화한 형태라 할 수 있다. Class는 시스템 Domain에서 필요한 요소들을 골라 만들게 되는데 이때 Domain이 가지는 특성들을 Class의 attribute로 추가하여 실제 세계의 요소들을 소프트웨어 시스템 안으로 이식시킨다. Class는 attribute외에도 다른 Object들과 상호 작용하기 위해 Operation들을 포함하게 되는데, 이런 Operation을 찾아내기 위해 Sequence Diagram을 작성한다. Sequence Diagram작성을 위해서는 Use Case를 참조하여야 하며, Sequence Diagram에 표현된 Object간의 상호작용은 모두 Class Diagram에 나타나게 된다. 이렇게 OOAD기법에서는 시스템을 나타내기 위해 서로 다른 특성을 지닌 Diagram들을 이용하며, 다양한 Diagram을 작성 함으로서 Diagram만 완벽하게 그리면 자동적인 Code Generation이 가능할 정도로 시스템을 완벽하게 표현 할 수 있는 것이 특징이다. 위의 Diagram들을 표현하기 위해 UML(Unified Modeling Language)를 많이 사용하며 Collaboration Diagram, Activity Diagram, State Chart Diagram, Deployment Diagram, Package Diagram등 다양한 Diagram들을 사용할 수 있다.

B. OSP란?

OOAD 방법론의 한 종류로 Object Space Process의 약자. RUP(Rational Unified Process)을 기반으로 하여 대학교 교육과정을 위해 사용되는 개발 과정이다. OSP는 크게 3단계로 나누어져 있으며 1000 단계 Plan & Elaboration에서는 개발할 시스템의 요구사항을 도출하고 전체 개발 과정의 계획을 수립한다. 2000단계 Build에서는 OOA와 OOD 단계, 즉 요구사항을 분석하고, 시스템을 디자인한다. 3000단계 Deployment에서는 Design된 시스템을 사용 할 수 있도록 실제로 구현한다.

C. SASD란?

Structured Analysis and Design의 약자로, 구조적 분석 설계 방법론 이라고도 부른다. 시스템을 Data의 흐름을 통해 표현하는 것이 큰 특징이다. 특별한 개발 프로세스 지원 툴 없이 펜과 종이만 가지고도 개발을 진행 할 수 있다. 시스템과 데이터를 주고 받는 장치들을 정의하는 System Context Diagram과 Event List를 작성하고, 장치로부터 데이터를 받아 또 다른 장치로 내 보내는 데이터의 흐름을 DFD를 통해 나타낸다. 이 때 DFD에서 Controller부분은 단순한 데이터의 흐름으로 나타낼 수 없기 때문에 Finite State Machine을 통해 더 자세하게 표현해 준다. DFD에서 각 데이터의 흐름에 대해 정의하는 Data Dictionary와 DFD의 각 프로세스에 대해 설명하는 Process Description의 작성 또한 필요하다. 분석 단계를 통해 도출된 DFD를 실제 구현 가능한 형태로 모델링 하는 과정이 SD 과정인데 이를 위해 Structured Chart가 사용된다. Structured Chart를 작성하게 되면 이를 통해 바로 프로그램 코드의 작성이 가능하게 된다.

2. OSP 1000~2030 & SA 비교

A. OSP 1000~2033 과 SA Essential Model 의 Environment Model 비교

OSP 1000~ 2033까지 단계와 SA Essential Model Environment까지 최종목표는 System Context를 정의하는 것에 있다.

OSP의 첫 단계는 1000. Plan and Elaboration 단계이다. 해당 단계는 실제적인 시스템의 분석 단계가 아닌 프로젝트를 수행하기 전에 예산이나 인력, 프로젝트 완료 예정 시간, 그리고 요구 사항을 한번 점검함으로써 프로젝트를 수행하면서 생기는 문제점이나 요구 사항이 빠진 것이나 잘못된 것이 없는지 점검하는 것이다. 이 단계를 통해서 커피 메이커 프로젝트 동기와 목적 그리고 시스템의 요구 사항을 결정하고 기술하였다.

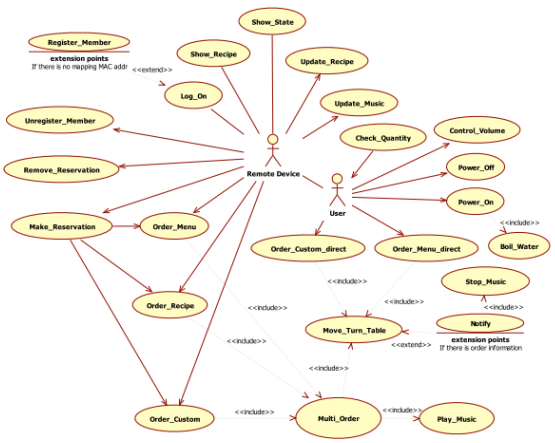
요구사항들의 Depth를 조절하는 부분에서 논란이 많이 생겼었는데, 하드웨어적인 부분의 경우 어디까지를 우리가 구현해야 할 레벨로 봐야 할 것인지를 정하는 것과 그것에 따라 다른 객체 부분들은 어느 레벨까지 구현을 할지를 판단하는 것이 어려웠다.

SA의 Essential Model Environment 같은 경우도 OSP의 해당 단계와 유사하게 수행하는데 여기서는 System의 Boundary, 시스템과 외부 장치간의 상호작용을 찾았다.

가장 처음에 수행되는 것은 "Statement of Purpose"로서 시스템의 기능을 간략하고 명확하게 서술하는 것이다. 이것은 1003단계의 기능적 요구 사항과 매우 유사한데, 차이점은 OSP같은 경우에서는 사용자에게 제공되는 시스템 자체의 기능을 자세히 기술하지만 SA 같은 경우는 시스템 자체적에서 행해지는 행동을 기술한 것이다. 그리고 SA단계를 수행하면서는 OSP처럼 자세히 기술하지 않았는데 그 이유는 Divide-Conquer를 하면서 기능도 나눠지기 때문이다. OSP에서는 이 과정에서 모든 요구 사항을 자세히 기술이 되었기 때문에 개발자의 입장에서 어떤 걸 고려해야 할지를 명확히 알 수 있었다.

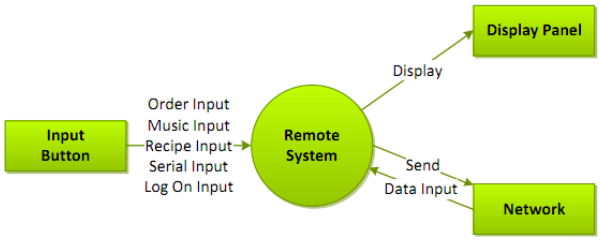
하지만 OSP에서 이 과정을 거치며 요구사항을 처음에 정확히 정의하는 것이 매우 힘들고 시간도 오래 걸렸다. 요구사항이 프로젝트 수행 중에 언제든지 변경이 되거나 추가될 수가 있었기 때문이다. 그렇기 때문에 요구사항을 추적표를 통해 관리를 해야 했으며 추적표 자체를 작성하는 것 또한 시간이 많이 소모되었다. 마지막으로 시스템의 흐름을 전체적으로 한 눈에 파악하기가 힘들었다.

OSP에서 UseCase를 사용함으로써 느껴진 장점은 Actor와 Actor가 사용하는 객체 간의 관계를 한 눈에 알 수 있다는 점이다. 객체를 사용함으로써 구조적 프로그래밍으로 구현이 가능하고 상속관계도 나타나기 때문에 나중에 구현할 때 코드의 재사용의 가능하다.



단점으로는 객체를 정의하고 구분하는 것이 어려웠다는 점이다. 어떤 속성을 객체에 포함하려고 할 때 이 속성이 해당 객체에 적절한지를 고려해야 하고 어느 객체에 포함하려고 해도 기준이 애매할 수가 있다. 그리고 UseCase를 통해서 시스템의 흐름을 알 수가 없었고 객체가 많아지면 각 객체간의 관계를 정의하는 것이 어려워지면서 또한 UseCase를 만드는 것이 어려워졌다. 결과적으로 시스템을 이해하는 것이 어려워질 수가 있다는 것이었다. 하지만 이러한 문제점을 보완하기 위해 Analyze단계로 넘어 가면서 2030단계에서 Plan과 UseCase를 다시 한번 더 점검을 하기 때문에 객체의 정의나 객체 간의 관계에서 잘못된 점을 수정할 수가 있었다.

SA의 Data Flow Diagram는 Top -Down 방식을 사용함으로써 처음에는 가장 최상위 관점으로 컨트롤과 인터페이스간의 상호 작용으로 시스템을 보기 때문에 구조도 간단하고 그리는 것도 매우 간편하였다.

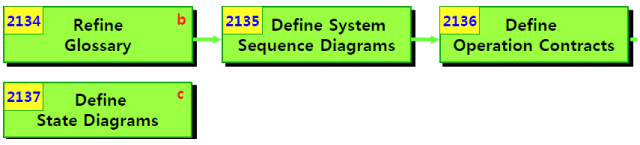


데이터의 흐름과 컨트롤과 인터페이스의 상호작용도 이벤트 리스트를 통해 보여 주기 때문에 사용자나 개발자 입장에서 시스템의 흐름과 구조를 이해하기 편하였고, 어느 정도 기준이 정해져 있기 때문에 컨트롤을 정의 함에 있어서 오류가 발생할 확률이 적었다. Data Flow Diagram의 단점으로는 컨트롤과 인터페이스를 정의할 때 이 부분이 완벽하게 정의되지 않으면 다음 레벨로 진행을 할 수가 없었으며 다음 레벨로 진행이 된다고 하여도 잘못된 정의는 분석과 디자인의 실패로 이어질 수 있기 때문에 처음 System Boundary를 정의 하는 것이 매우 중요하였다.

B. OSP 2034~ 와 SA Essential Model 의 Behavioral Model 비교

OSP에서는 2034 단계부터 이전 단계를 통해 도출된 용어들과 System Boundary들을 명확하게 정립하기 위해 Refine Glossary 단계와 System Sequence Diagram을 도출하는 단계를 거치게 된다.

System Sequence Diagram을 도출하는 과정에서 사용자와 시스템 간의 기능들과 System Boundary를 구분하게 되며, Define Operation Contracts 단계를 통해 Sequence Diagram을 통해 나온 시스템의 기능들에 대한 명세를 하였다..



OSP에서는 각 시스템의 기능을 명세할 때 하드웨어적인 부분을 표현하는 것이 어려웠는데, 어느 부분까지를 하드웨어로 정의하고 어느 부분을 시스템 구현 부분으로 나누어야 할 지를 정의하는 것이 힘들었다.

Define State Diagram 단계를 통해 도출된 각 use-case에서 각 상태에 대한 시스템 이벤트들을 정의하게 되는데, OSP의 State Diagram에서는 객체(Object)가 해당 이벤트를 발생시키는 시점과 객체의 상태가 이벤트에 따라 어떻게 변하는지를 나타내게 된다.

Structured Analysis 에서 또한 Behavior Model을 정립함으로써 시스템의 내부 Behavior와 시스템 내에서 사용되는 Data에 대해 명세를 하게 되는데 여기서 Data Dictionary를 명세하였다. OSP에서도 정의된 클래스의 Term에 대한 명세를 하였는데, 마찬가지로 SA에서도 비슷한 과정을 수행하였다. 이것을 통해 시스템에서 사용되는 데이터들에 대한 실제 해당 데이터나 자료구조의 자료형, 값의 범위, 식별자를 정의하게 되며, 여기서 정의된 Data 명세를 통해 프로세스들 간의 전달되는 인자 또한 정의하였다. Data를 정의할 때 저장소의 표현이 상당히 애매하였는데, 구조체나 지역 변수 등 실제 코드에서 쓰이는 개념들을 데이터 저장소에 매핑을 하는 것이 다소 어려웠다.

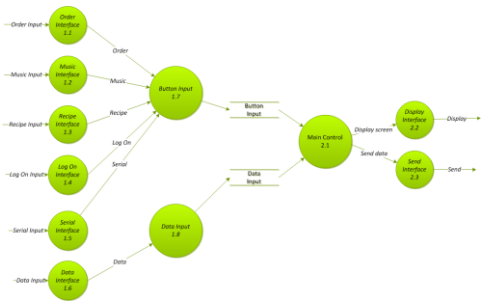
DFD에서 표현을 할 때 입력값이나 출력값의 종류 또한 결정해주어야 했기에 Data 대한 명세 뿐만 아니라 정의된 Event에 대해 해당 Event가 동기, 비동기적인지 또한 정하였다.

OSP에서 해당 단계에서 각 시스템 기능들에 대한 개별적인 명세와 해당 되는 이벤트에 대해 집중하여 명세를 하였지만 SA에서 Behavior Model를 정립하는 단계에서는 각 시스템에 대한 기능도 다루는 것보다는 각 기능(프로세스) 간의 관계와 각 프로세스들 간에 교환되는 Data의 흐름에 초점을 두고 모델을 정립하였다.

이로 인해 자연스레 Data나 Object 자체에 대한 것보다는 각 프로세스들간의 호출관계나 연결관계를 위주로 설계를 하게 된다.

이것들을 Structured Analysis에서는 Data Flow Diagram을 작성함으로써 표현을 하였는데, Divide and Conquer 시점으로 시스템 기능(프로세스)들 사이에 표현되는 Data 흐름을 개괄적인 시점에서 세부적인 시점으로 명세를 하였다.

OSP에서 Sequence Diagram을 사용자가 시스템에 요구하는(시스템이 사용자에게 제공하는)기능으로 생각하고 명세를 한 것에 반해, DFD에서는 Data 흐름에 따른 각 프로세스들의 관계들이 표시를 하게 되었다. 이로 인해 시스템의 기능을 단계별로 쉽게 파악할 수 있고, 전달되는 Data 흐름의 파악이 쉬워진다는 장점이 있었지만 사용자가 보기에 이해하기 어렵고, 시간적인 개념들을 표현하기 힘들다는 단점이 있었다.



Reference No.	2.1.4
Name	Send Order
Input	Button Input, Trigger, Tick
Output	Send Data
Process Description	Button Input의 What, Menu, Milk, Water, Syrup, ID값을 Send Data에 대입한후 F는 1로, path를 NULL로 설정하여 다음 프로세스로 전송한다.

DFD를 작성한 뒤에는 DFD에서 표현된 각 Process들에 대한 명세를 하는 Process Specification 단계를 거치게 된다. 각 프로세스가 전달받은 Data를 어떤 로직으로 처리하고 Output을 내보내는지에 대한 자세한 과정을 명세하게 되는데, 내부에서 사용되는 Data 타입과 호출시 전달되는 인자들을 세부적으로 표시하게 되는데 따로 툴들을 사용해서 정의한 Data 정보를 참조하기가 힘들고 소스 코드로 실제적으로 구현되는 부분이라 사용자나 설계자가 직관적으로 이해하기에는 다소 어려운 점이 있었다.

각 프로세스들이 내부적인 컨트롤러를 통해 호출되는 관계와 기능을 수행하게 되는 조건들을 표현하기 위해 OSP와 유사한 State Diagram을 도출하게 되는데, 시스템에서 표현되는 상태들을 정의하여 Finite State Machine에 표현을 하였다. FSM을 작성하면서 한 시스템 내에 여러 개의 컨트롤러를 정의한 것으로 인해 각 컨트롤러간의 동기화를 맞추고 표현하는 것이 어려웠었다.

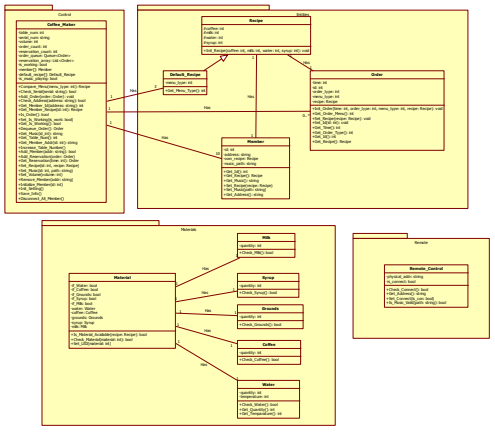
OSP의 State Machine이 각 객체에 대한 이벤트가 발생하는 시점에 초점을 맞춘 것이라 하면 Structured Analysis(SA)에서의 FSM은 Data 흐름에 해당되는 각 프로세스가 수행되는 시점에 초점을 둔다. Data 흐름에 따른 프로세스와 시스템의 상태 변화가 FSM에 나타나게 되었지만, OSP와 달리 해당 State의 전이가 언제, 얼마나 일어나는 지는 FSM에서 알 수가 없어 예약 부분이나, 컨트롤러간의 동기화 문제를 정의하는데 어려웠다.

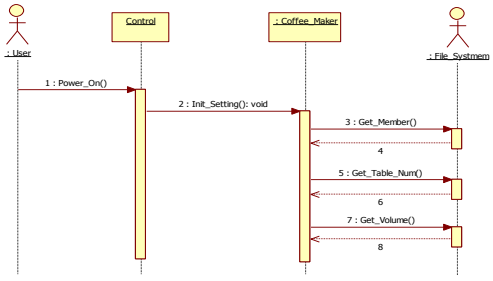
OSP의 2034 ~ 2037단계, SA - Behavior Model 정립 두 단계 모두 각 시스템에 해당되는 기능을 정의하고, System Boundary를 명확하게 한다는 점에서 공통점이 있었지만, 각각의 단계들이 초점을 두는 방식이 조금씩 다르다는 걸 느낄 수 있었다.

3. OSP 2040 & SD 비교

A. OSP 2040 Design & SD Implement Model 비교

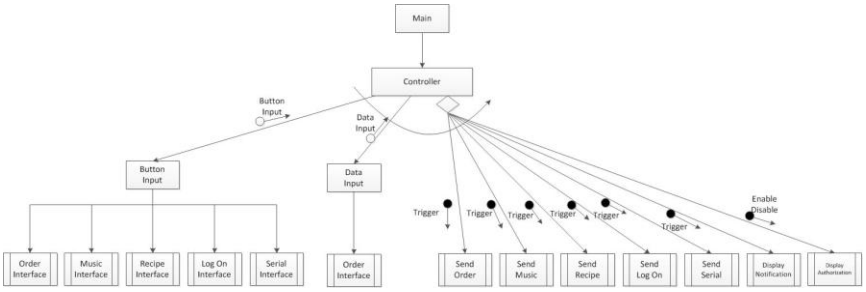
OSP 2040단계는 OSP 2010 Plan & Elaboration에서 나온 requirement들을 OSP 2030 Analyze단계에서 Use Case로 뽑아내고, 그 Use Case를 바탕으로 2041 단계에서 Use Case들이 실제로 각각 어떻게 동작을 하는지를 정의하고, 전 단계에서 나온 System Sequence Diagram과 정의된 Use case를 바탕으로 각각의 객체들이 어떻게 상호작용하는지를 2044단계 Define Interaction Diagram에서 Sequence Diagram을 통해서 나타냈고, 전 단계 2033 에서 정의한 Domain Model을 바탕으로 2045단계에서 Class Diagram을 Design하였다.





OOD에서 use case description을 만들 때 OOA에서 만들어진 Use Case와 객체들을 가지고 시스템을 만드는 방법이다. 어떤 기능인지만 정의 되어있던, Use Case들을 가지고 실제 이 Use Case가 객체들 사이에서 어떤 순서로 진행되는지를 정의하고, 이 때에 alternative 또는 exceptional한 단계가 있을 경우 포함 시켜줘야 한다. 이 단계를 진행할 때 실제 어떤 순서로 어떻게 동작해야 하는지를 정의 해야 하기 때문에 할 것이 많고 시간이 오래 걸린다. 그리고 sequence diagram을 그릴 때는 use case description을 보면서 그리면 되므로 많은 노력이 들지 않고, class diagram은 sequence diagram을 그리면서 각각의 객체에 function들과 attribute들을 정의 하면서 그릴 수 있으므로, 따로 그리는 시간이 필요하지 않다. 따라서 OSP 2040단계에서 가장 시간이 많이 할당되는 부분은 real use case를 만드는 것으로 이것이 완성되면 나머지 단계는 그리 어렵지 않았었다.

SD 단계는 SA단계에서 그린 Data Flow Diagram(DFD)을 바탕으로 시스템전체의 Call 관계를 정의한 Structured Diagram을 그리는 것이 목적이다.



SD 단계에서 Structured Diagram을 그릴 때 이전 단계에서 그렸던 DFD를 그대로 따라가면 된다. DFD에서의 controller와 interface들의 Trigger와 Enable/Disable 그리고 Data들의 흐름을 나타내면서 시스템의 명령이 어떻게 전달되는지를 나타낸다. SD에서는 SA에서의 DFD를 그대로 따라가면 되므로 단계를 쉽고 빠르게 끝낼 수 있었다.

SD방법론은 여러 번의 cycle을 돌지 않고 한번에 끝낸다. 그러나 divide & conquer방법으로 점진적으로 설계해 나가기 때문에 한번의 cycle을 통해 원하던 기능들을 모두 설계할 수 있었다. SD는 structured diagram만 그리면 전체 시스템의 data흐름을 파악할 수 있기 때문에, 많은 수의 diagram을 그려야 했던 OOAD개발 방법론보다 설계할 때 시간이 적게 소요되었다.

4. 구현단계

A. OSP 구현

우리는 이번 프로젝트를 진행 하면서 실제 구현을 하지는 않았지만, Diagram을 통해 code generation까지 해 보았다. OSP의 단계들을 충실히 따라 왔다면 실제로 code generation을 했을 때 각 Class의 형태는 매우 정확하게 생성 되는 것을 확인할 수 있었다. UML자체가 매우 다양한 Diagram들을 제공해주고 있기 때문에, 만약 더 좋은 개발 툴을 사용해서 필요한 Diagram을 모두 작성한다면, 따로 개발자가 코딩 할 필요 없이 완전한 code의 생성도 가능 할 것이다. 역시 문제는 그려야 하는 Diagram이 너무 많다는 데 있다. 기본적으로 Class Diagram 과 Sequence Diagram만으로는 Class의 형태만 정의할 수 있고, Activity Diagram과 State Chart Diagram, Package Diagram, Deployment Diagram 등의 Diagram을 모두 그려야 하는 것은 물론이고, 모든 Diagram이 톱니바퀴처럼 모두 맞물려 있어야 하기 때문에 직접 코딩 하는 것 이상의 작업이 필요할 수도 있다. 그래서 꼭 모든 Diagram을 작성할 필요 없이 원하는 Diagram을 선택해서 사용할 수 있다는 점은 설계자에게 유연성 있는 적용을 가능하게 한다. 단 Diagram의 종류가 줄어들면 시스템을 표현하는 방법이 제한적인 만큼 코드로 직접 구현 할 때 고려해야 할 부분이 많아진다. SD 과정을 통해 나온 Structured Chart가 바로 코드로 변환이 가능한 것에 비하면 실제 구현에 있어 OOAD가 더 어려울 수 있다.

B. SASD 구현

SASD과정을 거쳐서 도출해낸 Structured Chart는 바로 code로 변환이 가능할 정도로 모듈의 호출 관계를 정확하게 나타내고 있다. 그래서 실제로 code작성을 할 때 막힘 없이 빠른 code화가 가능했다. 그리고 FSM을 참조하게 되면 더욱 정교하게 모듈의 호출을 나태 낼 수 있다. 하지만 SASD과정에서는 각 모듈에 대한 설명은 Process Specification과 Data Dictionary에 만 표현되어 있기 때문에 세부적인 각 모듈을 구현하기 위해서는 프로그래머가 고려해야 할 부분이 상당하다. 그리고 OOAD처럼 표현을 위한 Diagram의 종류가 많지 않기 때문에 자동 툴을 사용 하더라도 완벽한 code를 생성해 내는 것은 불가능하다.

C. 구현 단계에서의 OSP & SASD 비교

OOAD는 많은 Diagram을 사용 할 수 있기 때문에 SASD에 비해 더 자세하게 표현이 가능하고, 따라서 자동 code generation을 사용할 수 있다. SASD는 전체적인 모듈의 호출은 Structured Chart를 통해 쉽게 구현 할 수 있지만, 세부 모듈을 구현하기 위해서는 코딩을 하는 프로그래머가 고려할 사항이 많다. 하지만 설계자와 구현자가 다를 경우 설계 과정에서 세부 사항에 제약을 주지 않고. 구현하는 사람의 재량에 따라 구현이 가능하다는 점에서 유연성을 가지고 있다. OOAD에서도 이런 유연성을 가지고 있어 많은 Diagram중에 설계자가 표현 하고자 하는 부분만 골라서 작성 함으로서 구현자가 세부 사항을 선택 하는데 제약을 주지 않도록 할 수 있다. Diagram을 어떻게 선택, 작성하느냐에 따라서 제약사항을 조절할 수 있다는 점에서 OOAD가 좀더 유연한 설계 방법일 수 있다.

5. Summary

SASD의 장점으로 사용자 입장에서 시스템의 흐름을 파악하기 쉬웠다. 또한 프로젝트를 수행하며 요구사항과 관련하여 작성을 해야 하는 문서가 매우 적었기 때문에 시간 소요가 OSP보다는 적게 소모되었다. SASD의 경우 컴퓨터 구조에 기반하여 과정들이 수행되기 때문에 OOAD보다 전체 시스템을 설계하고 이해하는데 편리하였다. 반면에 요구 사항 명세가 OOAD에 비해 약한 부분이 존재하여 개발 도중에 요구사항 변경이 발생하였을 때의 리스크가 다소 크게 작용할 것 같았다.

여러 시스템들이 연동되거나 다수의 컨트롤러가 설계되는 대규모 시스템을 개발할 때에는 여러 시스템들간의 연관관계를 표시하거나 컨트롤러의 동기화 부분을 표현하는 것에서 SASD보다는 OOAD 개발 방법론이 조금 더 적합해 보였다.

또한 OOAD에 비해 개발과정에서 사용할 툴들이 많이 없어서 Data Dictionary, 프로세스 명세와 관련된 내용들을 체계적으로 참조하고 관리하기가 힘들었다.

OSP의 장점 중의 하나는 여러 번의 Cycle을 돌 수 있다는 것이었다. 여러 번의 Cycle을 돌면서 설계 하므로 첫 번째 Cycle에서 미쳐 고려하지 못했던 부분들을 두 번째 Cycle에서 고려해서 설계할 수 있었고 잘못된 부분을 수정 할 수 있었다. 그리고 두 번째 Cycle을 돌 때 시스템 요구사항의 변화가 있었음에도 불구하고, 첫 번째 Cycle에서 나온 결과물의 거의 모든 부분을 그대로 사용할 수 있었다. 이러한 점을 통해 시스템을 점진적으로 개발할 수 있었다.

하지만 OOAD개발 방법론은 많은 Diagram을 지원하기 때문에 어떤 Diagram을 그려야 하는지 선택하는 것이 어려웠고, 한 두 개의 Diagram을 통해 실제 프로그램 작성이 어렵고 설계자가 원하는 모든 것을 표현할 수 없기 때문에 구현하는 사람이 이해할 수 있도록 여러 개의 Diagram을 그려야 했기 때문에 설계하는데 시간이 더 많이 소요되었다.
하지만 SASD의 FSM에서 표현되는 시스템의 실시간성이 OOAD에서는 표현하기가 힘들었다.

공통점은 전체 시스템의 개발과정이 Sequence한 설계방법론으로서 순차적으로 단계가 진행된다는 점이다. OODA에서는 이전의 단계 즉 OOA에서 만들었던 Diagram과 Description을 통해서 OOD에서 실제 코딩을 하기 위한 Sequence Diagram과 Class Diagram을 만들었고, SASD에서는 SA에서 순차적으로 만들었던 DFD를 통해서 Structured Diagram을 만들었다.

또한 이 두 방법 모두 실제 코딩을 위한 디자인 방법이라는 것이다. 따라서 OSP에서 나온 결과물인 Class Diagram과 Sequence Diagram, 그 외의 다양한 Diagram을 통해서 실제 프로그램을 개발하는 개발자가 설계대로 프로그램을 구현하면 되고, SASD또한 Structured Diagram과 Finite State Diagram을 통해서 설계된 대로 프로그램을 구현 할 수 있다.

OOAD와 SASD 개발방법론 모두 요구사항 분석단계가 가장 중요했다. OOAD를 하면서 가장 많은 시간을 할당하고 설계도중에 가장 많이 바뀐 부분이 요구사항 부분이었다. 요구사항이 명확하면 설계의 방향이 명확해지고, 사용자가 원하는 프로그램을 만들 수 있다.

그리고 OOAD 설계를 할 때 요구사항을 명확히 했기 때문에, SASD 설계할 때 많은 도움이 되었다. SASD설계 방법론은 여러 번의 Cycle을 도는 개발 방법론이 아니기 때문에 요구사항 분석이 명확하지 않으면, 시스템 설계를 처음부터 다시 해야 하는 상황이 생긴다. 하지만 요구사항이 명확했기 때문에 쉽게 설계할 수 있었다. OOAD 개발방법론도 요구사항을 통해 Use case를 만들기 때문에 요구사항 분석단계는 모든 OOAD, SASD 두 가지 방법 모두에서 가장 먼저 해야 할 중요한 단계였다.

6. 각 팀원 별 소감

A. 김정태

이번 프로젝트를 하면서 OOAD같은 경우는 객체지향적 구조로 하기 때문에 하면서 객체로 나누는 과정에 있어서 그렇게 어렵지 않았지만 시스템의 흐름을 알 수 없기 때문에 Sequence Diagram을 그리는 과정에 있어서 어려운 점이 있었고 SASD에서는 해야 할 문서도 매우 적기 때문에 OOAD보다 시간이 덜 걸릴 것이라 예상했지만 시스템이 레벨이 올라가면서 컨트롤 개수도 많아지고 프로세스도 많아 지면서 OOAD만큼 시간이 걸렸다. 교수님 말씀처럼 소프트웨어 공학은 정답이 없고 다 장단점이 있는 것 같다. ,

B. 송준현

지난 학기 때 소프트웨어 공학 수업을 들으며 SASD방법론에 대한 프로젝트를 수행해봤지만 OOAD와 관련된 내용은 실습해보지 않았는데, 많이 헤맸지만 팀원들에게 많이 배우고 고생해가면서 두 개발 방법론에 대해 정말 많은 것을 알게 되었다.
어느 것이 좋다 안 좋다가 아니라 이러이러한 시스템에는 OOAD, SASD 이렇게 시스템의 특성에 따라 개발론을 적용하여 시스템을 설계해 나가야 한다는 것을 깨닫게 되었고, 소프트웨어 개발에서 실제 코딩을 하는 구현 파트보다는 그 이전에 시스템을 분석하고 설계하는 단계가 어마어마하게 더 중요하다는 것을 몸으로 느끼게 해 준 수업이었다.

C. 이낙원

먼저 소프트웨어 공학이 정말 어려운 과목이라는 걸 뼈저리게 느꼈다. 다른 전공 과목들이 답이 있는 수학 문제 같은 느낌이라면 소프트웨어 공학은 답이 없는 문제를 푸는 것 같았다. 표기법에 제약 사항도 많고, 아직은 각 방법론에 익숙하지 않다 보니, 제대로 진행하고 있는 것인지 확신을 가질 수 없었다. 특히 OOAD같은 경우 UML의 정확한 사용법을 모른 상태에서 진행 하다 보니 문서 작업에 많은 시간이 소요 됐다. 두 방법론 모두 작성할 문서도 상당히 많고, 시간도 너무 많이 소요돼서인지 아직은 소프트웨어 개발 방법론이 효율적인 개발을 가능하게 하는지는 잘 와 닿지 않는 듯 하다.

D. 최정명

SASD와 OOAD개발방법론은 둘 다 소프트웨어를 만들기 위한 방법이지만 방법자체는 매우 다르다는 것을 알게 되었고, 또 각각의 개발방법론이 어떤 상황에서 어떠한 장단점이 있는지를 직접 체험할 수 있었다. 그리고 두 개의 각기 다른 개발 방법론으로 소프트웨어 설계를 해 보았기 때문에 나중에 다른 소프트웨어 개발 방법론으로 개발을 하게 되더라도 다른 사람들에 비해 빠르게 그 개발방법론에 적응을 할 수 있을 것 같다.