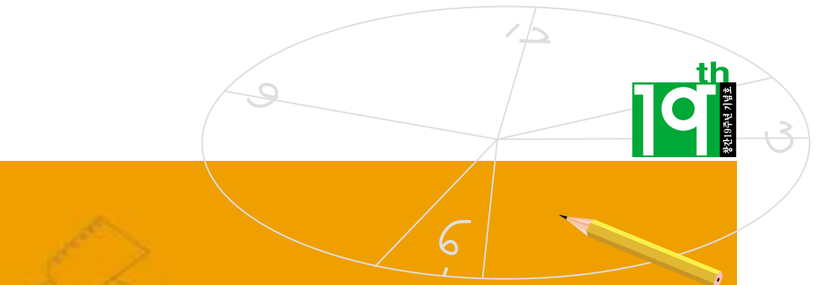
특집 2-1부

난 이렇게 개발한다 1

개발 속도 향상 비법은 '정공법'에 담겨 있다



개발 속도 향상을 위한 왕도는 없다고 본다. 정도를 따라서 개발자의 실력을 높이는 것이 궁극적으로 개발 속도를 향상시킬 수 있는 방법이 될 것이다. 여기서는 필자가 경험하고 느낀 것을 바탕으로 실제 개발시 사용하고 있는 방법에 대해 얘기해 보려 한다.

프로그램 개발 속도의 향상은 쉽게 얘기할 수 있는 문제는 아니다. 일단 어려운 점은 확실한 개발 속도를 높이는 비법이 존재하지 않는다는 점이고, 특별한 방법이 존재한다고 하더라도 각 개발자마다 적용할 수 있는 정도가 다르고 실제 효과도 다를 것이라는 점이다. 필자는 이런 점에서 개발 속도 향상에 있어 개인적으로 '정공법'이 가장 좋은 방법이라고 생각한다. 정공법이란 말 그대로 일반적으로 개발할 때 하는 방법 그대로를 말한다. 즉, 분석/설계 우선의 개발, 효과적인 코딩, 효율적인 디버깅 그리고 개인 스케쥴 관리 등을 통해 프로그램을 개발하는 방법이 바로 정공법이다. 그리고 부연 설명을 덧붙인다면 정공법을 통한 개발 속도 향상이란 개발자 스스로의 내공을 높이는 것을 말한다. 궁극적으로 개발 속도 향상을 이뤄내기 위해서는 개발자 스스로의 역량을 높이는 것이 가장 좋은 방법이고 바람직하지 않을까 생각한다. 물론 단기적으로 볼 때 IDE 툴을 도입해서 코딩을 쉽게 한다던가 효율적인 디버깅 전용 도구를 사용한다면 개발 속도 향상에 큰 도움이 될 것이다. 그렇지만 결국 IDE 툴을 사용하더라도 우선적으로 그 툴을 잘 다룰 수 있도록 훈련해야 하고, 그러한 훈련 자체가 바로 개발자 자신의 실력을 업그레이드 하는 것과 같지 않을까.

필자가 하려는 얘기는 실제 내 자신이 어떤 방식으로 개발하는지 따라가 보면서, 정공법이 개발 속도 향상에 어떻게 영향을 미치는 지에 대해 개인적인 생각을 얘기해 보려고 한다.

물론 정공법이라는 것도 결국 내가 하고 있는 개발 방법을 얘기하는 것이지만, 그것이 일반적인 개발 방법론이 아니라는 것은 먼저 말해 둔다. 다만 이 글을 읽으면서 각자 갖고 있는 정공법과 비교해 보고 판단하기를 바란다.

필자의 경우에는 개발할 때 다음과 같은 순서로 진행을 한다. 「기능 분석/기능 설계 | 코딩 | 테스트/디버깅」. 특별히 다른 사람이 사용하고 있는 개발 순서와 다른 점은 하나도 없을 것이다. 이러한 순서대로 개발의 흐름을 따라가 보면서 얘기를 해보고자 한다.

윤남영 skywatch@ntechs.com

현재 다음 기술에서 자바 개발자로 활동하고 있으며, 프레임워크와 아키텍처 쪽에 관심을 갖고 있다. 농구를 좋아해서 휴일에는 4~5시간을 할애할 정도의 농구광. 애니메이션과 만화 또한 무척 좋아한다고.

코딩 이전의 문서화는 필수

프로그램을 개발할 때 가장 먼저 할 일은 '무얼 만들까'란 것이다. 내가 맡은 부분이 독립된 별도의 프로그램일 수도 있고 아니면 공통으로 사용하는 특정 모듈일 수도 있다. 또 특정한 비즈니스 로직을 구현하는 것일 수도 있다. 그 어떤 개발이든 간에 먼저 해야 할 일은 만들고자 하는 기능을 정의하는 일이다. 즉, 기능 분석이 우선이다. 개발을 하다 보면 만들고자 하는 기능에서 벗어난 기능을 만들고 있는 경우가 종종 생기게 된다. 그러나 기능 분석을 통해 만들고자 하는 기능을 확실히 정의해 놓고 내친 김에 문서화까지 해놓는다면 아마 그러한 일은 일어나지 않을 것이다. 필자의 경우에는 프로그램을 개발할 때 먼저 기능 분석을 하고 그 분석한 내용을 문서화한다. 즉, 기능 분석 문서를 작성한다. 때에 따라서는 그 기능 분석 문서에 UML을 사용해 도식화하고, UseCase 분석을 해서 화려하게 그려 놓기도 하고, 때로는 그냥 간단한 워드 문서를 만들어서 사용하기도 한다. UML을 사용하든지 간단한 형식을 사용하든지 중요한 점은 문서화를 한다는 점이다.

객체지향 방법론으로 개발을 하다 보면 초반에 만들어야 할 문서가 엄청 많다. UML과 관련해서 사용하는 UseCase 다이어그램, UseCase 디스크립션, 시퀀스 다이어그램, 액티비티 다이어그램 등 문서가 꽤 많고 다양하다. 정석대로 다 하려고 한다면 초반 문서화에만 아마 전체 기간의 10%~20%는 필요할 것이다. 실질적으로는 좋은 방법론이지만 실제 프로젝트 초반에 도입을 하기에는 쉽지가 않다. 또 현실적으로 UML 쪽 문서화에 대해서 훈련이 된 개발자들도 그리 많지는 않다. 또한 경험상 그 정도의 문서가 개발 단계에서 다 필요하지는 않다. 따라서 필요한 문서들만을 골라서 초기 문서화를 하는 것이 좋다.

보통 개발하다 보면 노트에다 구현해야 할 기능을 간단히 요약해서 써 놓거나, 머리 속에만 담고 개발을 하는 경우를 종종 보게 된다. 그러나 이런 방식으로 개발하다 보면 기능이 많을 경우에는 몇몇 기능은 빼 놓거나, 원래 의도와 맞지 않는 경우가 생기는 것을 쉽게 볼 수 있다. 문서화는 어려운 것이 아니다. 그리고 시간이 많이 들어가는 것도 아니다. 대부분의 개발자들은 문서화에 대해서 필요하지만 귀찮은 작업 정도로 여긴다. 그리고 해야 하기는 하지만 대충대충 넘어가고 나중에 하려고 하는 사람들도 꽤 많다. 그러나 일단 문서화는 나중에 하게 되면 산출물로서의 가치는 있을지 몰라도 개발 문서의 원래 목적인 개발에 도움이 되는 실제 가치는 전혀 얻을 수 없다. 그리고 기능 분석한 것을 문서화하는 것은 의외로 쉬운 작업이다. 또 문서화를 우선시할 경우 좋은 점은 개발을 하기 전에 팀장이나 고객들에게 확인을 얻을 수 있다는 점이다. 그러다 보면 기껏 개발을 해 놓고 나중에 다시 만드는 일은 벌어지지 않는다.

기능은 말로 풀어쓴다

기능 분석에 대한 문서화 작업이란 것은 기능에 대해서 본인이 이해한 것을 말로 풀어 쓰는 작업이라고 생각한다. 어쩌면 편견인지 모르지만 불행히도 공대 출신인 사람들은(물론 글을 쓰고 있는 필자도 공대 출신이다) 번호를 매겨서 첫째 그리고 둘째 하는 식으로 정의하는 것에 익숙한 경향이 있다. 즉, 분류를 하고 번호를 붙여서 정리하는 것에 익숙하다는 얘기이다. 그러나 기능 분석에 대한 문서화에서는 그런 분류는 필요 없다. 중요한 것은 말이다. 기능을 말로 풀어서 쉽게 읽을 수 있도록 하는 것이 오히려 효과적이다. 그러면 어떤 식으로 기능을 말로 풀어서 얘기할 수 있는지 알아보도록 하자.

간단히 스케쥴러를 만든다고 가정해보자. 스케쥴러는 등록된 작업 정보를 갖고 정해진 시간에 그 작업을 실행시켜 주는 기능을 하는 프로그램이다. 자세히 분석해보면 아마도 스케쥴러의 세부적인 기능들이 도출될 것이다. 예를 들어 간단하게 말로 만들어 보자.

"스케쥴러는 사용자가 설정해 놓은 특정한 작업에 대해서 정해진 시간에 실행시켜 주는 기능을 가진다"

그렇지만 이건 너무 간단하다. 읽는 사람이 쉽게 알 수는 있지만 기능을 표현하고 있지는 못하다. 조금 더 말을 길게 만들어 보자.

"사용자는 특정한 시간에 실행시키기를 원하는 작업을 스케쥴러에 등록할 수 있고, 등록할 때는 실행하고자 하는 작업의 실행 커맨드와 파라미터를 미리 등록한다. 등록된 정보는 프로그램 상에 별도로 저장된다. 스케쥴러는 주기적으로 등록이 된 스케쥴 정보를 검사하며 실행할 시간이 된 스케쥴에 대해서는 실행할 작업의 커맨드와 파라미터 정보를 가져와서 실행을 해준다."

좀더 말이 늘어났다. 그리고 추가적으로 작업 또는 스케쥴이란 실행 커맨드와 파라미터라는 정보로 이뤄져 있다는 것이 새롭게 들어갔다. 이런 방식으로 계속해서 말을 늘려가면서 기능에 대해 개발자 자신이 이해한 내용을 정리해 놓은 것이 바로 기능 분석 문서이다.

기능 분석 문서를 먼저 작성하게 되면 구현해야 할 기능에 대해서 완전히 이해를 하게 된다. 그렇게 되면 기능을 구현하다가

다시 분석 단계로 돌아오는 경우가 생기지 않게 되며, 개발하는 동안 이 기능 분석 문서는 개발 가이드로 계속해서 참조가 가능할 것이다. 즉, 구현할 기능에 대해서 잘못 이해함에 따라서 오는 개발 시간의 지연이 없을 것이다. 그리고 의외로 쉽다는 사실을 말하고 싶다. 오히려 첫째 그리고 둘째라는 획일적인 방식으로 번호를 매겨가면서 분석하는 것보다 훨씬 빠르고 쉽게 분석할 수 있을 것이다.

단순하고 간단한 설계가 원칙

분석을 했으면 이제 다음 단계로 넘어가 보자. 설계는 프로그램의 청사진을 만드는 단계이므로 세심한 공이 들어가야 하고, 그만큼 어려운 작업이다. 예전에 설계의 중요성에 대해 설명한 글에서 "설계에는 40%의 시간을 할당하고, 구현에는 60%를 할당하라"는 글을 본 기억이 있다. 맞는 말이다. 설계의 비중은 그만큼 크다. 그렇지만 개인적인 경험으로는 그 40%의 시간을 모두 코딩 이전에 몽땅 투자해 버리는 것은 그리 좋은 방법은 아니라고 생각한다. XP 쪽에서는 단순한 설계를 강조한다. 특히 XP에서는 사람들이 모여서 10~30분 동안에 설계를 하라고 추천한다. 개인적으로는 충분히 수용할 만한 생각인 것 같다. 물론 설계는 중요하다. 그러나 설계에서 너무 많은 시간을 소비하는 것은 개발 속도를 늦추는 원인이 된다. 설계 단계에서 너무 많은 것을 생각해서는 안 된다. 또 너무 복잡하게 생각하는 것도 좋은 생각은 아니다.

보통 프로그램을 개발하다 보면 처음의 설계가 마지막까지 이어지는 경우를 본 기억이 없다. 실제 코딩하다 보면 또는 새로 추가되는 기능과 제약 사항을 수용하다 보면 설계는 당연히 변경되기 마련이다. 처음에 너무 거창하게 설계하면 이러한 변경되는 부분에 대한 수용이 상당한 부담으로 다가 온다. 또한 처음에 너무 완벽한 설계를 하기 위해서 투자한 시간 때문에 그 이후의 설계 변경에 할당할 수 있는 시간이 줄어들게 된다. 즉, 전체적으로 설계에 투자하는 시간이 줄어든다.

그렇다고 대충대충 설계를 끝내라는 말은 절대 아니다. RUP(Rational Unified Processor)나 XP에서는 기본적으로 Iteration이란 것을 강조하며 기본으로 삼고 있다. Iteration은 지속적인 반복을 위한 단위를 의미하는 말이다. 즉, 기능을 시나리오 단위로 나누고 점진적으로 개발하는 것을 말한다. 이는 설계에도 포함되는 얘기라고 생각한다. 처음에 설계한 것을 매번 Iteration마다 반복적으로 검증하고 점진적으로 발전시켜 나가는 것이다. 즉, 리팩토링(Refactoring)을 통한 점진적인 개선을 염두에 두는 것이다. 물론 이러한 점진적인 설계에서도 전체적인 큰 틀은 변화하면 안된다(때로는 변화할 수도 있겠지만 가급적이면 이러한 일이 벌어지지 않도록 해야 한다). 지나친 변화는 오히려 더 큰 부담으로 작용하게 되고 큰 틀이 변화하면 내부에 대부분의 것들이 변화해야 하므로 주의해야 한다. 바로 설계할 때 차후 수정할 것을 염두에 두어야 한다는 점이다. 이 점은 생각보다 중요하다. 그래서 적용하는 것이 바로 큰 틀 구조인 아키텍처 또는 프레임워크이다.

그러나 실제 설계할 때 단순하게 한다는 것은 쉬운 일이 아니다. 보통 시스템을 설계하다 보면 이런 저런 생각을 많이 하게 되고, 그러한 생각들이 반영되면 시스템의 복잡도는 기하급수적으로 증가하게 된다. 충분히 단순하다는 것을 XP에서는 "첫째 모든 테스트를 실행한다, 둘째 모든 아이디어를 표현한다, 셋째 중복된 코드를 포함하지 않는다, 넷째 최소한의 클래스와 메쏘드를 가진다"라는 네 가지 표현을 뜻한다고 말한다. 충분히 단순한 설계의 경우에는 나중에 변경이 있더라도 간단하게 그 변화를 적용할 수 있다. 특히 "중복된 코드를 포함하지 않는다"라는 뜻은 나중에 코드를 추가할 때 한 곳만 고치면 되기 때문에 여러 부분에서 고칠 필요가 없다. 이렇게 하기 위해서는 단순한 설계만을 갖고 되는 것은 아니며 충분한 리팩토링을 거쳐야 한다.

코딩의 기본은 짧게

이제 코딩에 관한 이야기를 해보자. 실제 개발 도중에 코딩에서 원래 의도한 것 이상의 시간이 걸리는 경우가 많다. 그러나 실제로 그 이유를 보면 실제 기능을 수행하는 메쏘드를 만드는 시간이 많이 소요되는 것이 아니라 이것저것 메쏘드를 엮고, 재설계를 하는 시간이 많이 걸리는 것을 볼 수 있다. 또한 복잡한 설계를 함에 따라서 각 모듈 간의 관계가 서로 얽히게 되서 시간이 오래 걸리는 경우가 많다. 즉, 순수하게 특정 메쏘드나 함수를 추가하는 시간은 그렇게 오래 걸리지 않는다. 물론 코딩 도중에 기술적인 한계 때문에 시간이 많이 소요되는 경우가 있는데, 이에 대한 고려는 설계 단계에서 고려해야 하고 실제 일을 분배할 때 고려해 진행되어야 할 사항이다. 즉, 프로젝트나 프로그램을 개발할 때, 서로 담당할 모듈을 분배하는 시점에서 기술력에 맞도록 분배가 되어야 한다.

코딩을 짧게 하는 것에 대한 특별한 방법은 없다. 일단 코딩은 설계의 영향을 많이 받게 되고, 실제 코딩에 있어서도 어떤 식으로 구현을 하는가에 따라서 많이 달라지기 때문이다. 가급적이면 하나의 메쏘드에 여러 가지 생각을 담지 않는 것이 좋다. 하나의 메쏘드에는 하나의 로직만 존재하도록 해주고, 여러 개의 로직이 필요한 경우에도 실제 로직을 수행하는 부분들은 별도의 메쏘드

javadoc 만들기

보통 Java Document(Javadoc)라는 것은 아마 자바를 개발하는 개발자들은 매일 같이 보아 왔으니까 알고 있을 것이다. 바로 자바의 API 도큐먼트를 말하는 것이다. 자바에서는 SDK를 설치할 때 자동으로 javadoc이라는 실행파일이 설치가 된다. 이 프로그램이 하는 일은 소스를 분석해서 javadoc 형태의 코멘트를 갖고 자동으로 API 도큐먼트를 만들어 주는 일이다. 관련해서는 SDK의 도큐먼트에 보면 'Tool Docs' 항목으로 들어가서 'Java Doc 1.xx' 항목을 보면 관련 문서를 볼 수 있다.

Javadoc을 만들기 위해서는 소스에 javadoc 스타일의 코멘트를 넣어 주어야 한다. 기본적으로 두 가지가 있다. 첫 번째는 클래스 디스크립션(class description)이고, 또 하나는 메쏘드 디스크립션(method description)이다. 먼저 클래스 디스크립션은 다음과 같은 식으로 넣어 주게 된다. 다음의 소스 코드는 실제 개발한 소스 코드의 일부이다.

```
/**
 * <br><b>Class Description</b><br>
 * NCpScSearch class는 일반적인 검색에 사용되는 쿼리 스트링(Query String)을
 * 만드는 기능을 제공해 주는 클래스이다.
 *
 * NCpScSearchBase를 상속 받아서 사용된다.
 * 일반 검색에서는 Multi-Field Range를 지원한다. 단, 원래의 의미로서의 Multi-
Field Range 검색이 아니고
 * Multi-Field Range Restriction(다중 필드 제약 검색)의 의미로서 다중 필드를
 * 처리해 준다. 즉, 필드 제약 경과 A라는 결과 집합이 나왔을 경우에 범위 제약이란
 * 그 결과 A 내에서 제약 조건에 해당하는 검색 결과들을 제외시키는 것을 말한다.
 .......
 */
```

이런 식으로 만들어 주면 〈화면 1〉과 같은 형식으로 문서가 만들어져서 보여진다.

두 번째 메쏘드 디스크립션은 다음과 같이 넣어 주게 된다.

```
/**
 * 현재까지 설정한 값들에 의한 실제 쿼리 스트링을 생성해서 가져온다.
 * 실제 쿼리 스트링을 현재까지 설정된 데이터를 기반으로 만들어 낸다.
 *
 * @return QISP Query Protocol string for search.
 * @exception NCpScException there is some error while
 * create query. Like some needed field data is not exist

 */
public String getQueryString()
        throws NCpScException, Exception
{
        .......
}
```

〈화면 1〉 만들어진 javadoc의 클래스 디스크립션 부분

〈화면 2〉 만들어진 javadoc의 메쏘드 부분

@return이나, @exception은 예약어로서 사용되며 실제로는 여러 개가 정의되어 있지만 몇 가지만 설명하면 다음과 같다.

- **@param** : Parameter에 대한 설명을 나타낸다. @param 〔파라미터 변수〕 〔설명〕 과 같이 써 준다.
- **@return** : Return되는 값에 대한 설명을 넣어주면 된다. @return 〔return 값 설명〕
- **@exception** : @exception 〔Exception 이름〕으로 써주며 특정 Exception이 발생할 경우의 상황이나 발생하는 원인에 대해서 써주면 된다.

앞의 코멘트를 갖고 실제 만들어진 javadoc은 〈화면 2〉와 같다.

개인적으로는 자바를 사용하는 개발자들은 javadoc을 반드시 이용하는 것을 권한다.

로 만들어 주는 것이 좋다. 그렇게 하면 메쏘드는 상당히 간결해진다. 간결하다는 것은 만들기가 쉬워지며 또 버그가 발생할 수 있는 가능성도 줄어든다.

코멘트 우선의 코딩, 가이드가 되어준다

개인적인 취향인지 모르겠지만 코딩할 때 먼저 코멘트를 달고 코딩하는 습관이 있다. 특정 클래스를 만들 때는 일단 기본 코멘트를 달아 준다. 대충 어떤 기능을 하는 클래스이며 저작자나 수정한 기록 등을 먼저 코멘트 형식으로 기입해 준다. 그리고 클래스 Description을 적어 준다. Description은 Javadoc 형태로 적어 준다(Javadoc 형태의 코딩에 대해서는 박스 기사를 참조하기 바란다). Description에는 현재 작성하는 클래스의 기능이나 역할에 대해서 간단하게 설명하며, 필요한 경우 클래스의 이용에 대한 코멘트를 달아 준다. 그리고 메쏘드를 코딩할 때에도 Javadoc 형태의 코멘트를 먼저 달아 준다. 즉, 어떤 파라미터를 입력하도록 되어 있고 수행 결과 어떤 값이 넘어 오게 되며, Exception을 throws할 경우에는 어떤 에러일 때 그러하는지에 대해서 먼저 코멘트를 달아 준다.

이런 식으로 코멘트를 먼저 만든 다음에 클래스나 메쏘드에 대한 코딩을 하게 되면 코멘트가 일종의 가이드 역할을 해준다. 즉, 설계에서는 세부적인 것을 다루지 않기 때문에 실제 코딩 단계에서는 간단한 클래스 다이어그램이나 간단한 클래스 정의서를 보고 코딩하게 마련이다. 이러한 것들을 더 자세히 정의한 것이 바로 코멘트가 되는 것이다. 특히 Javadoc 형태로 코멘트를 넣어 줄 경우에는 javadoc 형태의 문서로 바로 generation이 가능하며 개발하는 도중에 javadoc 문서를 보면서 코딩할 수 있다. 즉, 메쏘드의 정의를 코멘트를 통해 했으므로 그 정의한 대로 코딩하면 정확히 원하는 기능만을 구현할 수 있다. 또한 바로 문서화가 되므로 별도의 문서화가 필요 없이 바로 각 클래스나 메쏘드에 대한 문서를 얻을 수 있고, 개발하는 도중에 계속해서 참조해 사용할 수 있다. 이런 식으로 코멘트 우선의 코딩을 하다 보면 처음에는 좀 더디게 개발이 되지만 익숙해진 이후에는 개발 속도가 증가하는 밑거름이 된다.

이 코멘트 내가 작성한 거 맞나?

코멘트를 넣어 줄 때에 항상 염두에 두어야 할 사항이 있다. 코멘트는 누군가에게 보여주기 위한 글이다. 그 누군가는 팀 내의 다른 사람일 수도 있고 본인일 수도 있다. 아니면 유지보수를 위해 새로 입사한 신입사원일 수도 있는 것이다. 따라서 코멘트를 넣어 줄 때는 자신만이 알아 볼 수 있는 방식으로 코멘트를 넣어선 안 된다. 물론 개발자 본인만이 다시 본다고 가정한다면 그럴 수 있을지도 모른다. 그러나 경험상 본인이 작성한 프로그램의 소스도 2~3개월이 지나면 실제 코드를 보아야 기억이 나며, 6~7개월이 지나면 남이 만든 소스와 별로 다를 것이 없다고 느끼게 된다. 나중에 본인이 작성한 코드를 보고 그 코드를 만든 누군가(본인이지만 아마 잘 모를 것이다)를 원망하지 않으려면 쉽게 코멘트를 달아 주어야 한다.

그렇다고 코멘트를 너무 많이 다는 것 또한 좋지 않은 생각이다(여기서 말하는 코멘트는 앞에서 얘기한 javadoc 형태의 코멘트를 제외한 코멘드를 말하는 것이다). 지나친 코멘트는 소스의 가독성을 크게 해친다. 필자 또한 예전에는 거의 한 라인당 한 라인의 코멘트를 넣어 주곤 했다. 그러나 결국은 그럴 필요가 없다는 것을 알게 됐다. 메쏘드가 충분히 작고 쉽게 그 기능을 알아 볼 수 있다면 메쏘드 내의 코멘트는 사실상 필요가 없고 오히려 코멘트 때문에 실제 로직이 가려져 버리는 경우가 생기기 때문이다.

수정을 꺼리지 말자

코딩을 아무리 잘해 놓았다고 하더라도 개발 중간에 수정을 안 할 수는 없다. 특히 프로젝트를 수행하다 보면 고객의 요구사항이 바뀌는 경우가 많이 생긴다. 물론 개발 중간에 그런 식의 변화가 많이 생긴다면 좀 문제가 있겠지만 실질적으로 그러한 기능상의 변화는 종종 있는 경우이다. 그러한 변화가 많은 수정을 동반하는 경우도 있고 그렇지 않은 경우도 있다. 그렇지만 결국 문제는 개발자한테 돌아오게 마련이다. 간단한 수정의 경우에는 웃으면서 "뭐 그렇게 하지요"라고 할 수도 있지만, 수정의 범위가 많아지거나 수정할 내용이 난이도가 있다면 상당한 부담이 된다. 솔직히 그런 경우에는 먼저 짜증이 난다. 그렇지만 대부분의 경우에는 결국 수정하게 된다. 그럴 때는 확실하게 수정하는 것이 좋다. 대충 수정을 하거나, 간단한 편법으로 마감을 한다면 나중에 오히려 크게 수정하게 되는 경우가 생기게 된다. 즉, 수정을 꺼리지 말아야 한다는 얘기다. 오히려 그러한 수정을 꺼리다가 나중에 혼이 나는 경우를 종종 경험했었다.

XP에서 말하는 단순한 설계라는 의미는 이러한 경우를 위해서도 있는 말인 것 같다. 단순한 설계란 최소한의 중복을 갖게 되고, 그 결과 코딩 상에서도 최소한의 수정으로 기능을 변경할 수도 있다는 얘기가 된다. 또 뒷부분에서 말하겠지만 테스트 드리븐 디벨롭먼트(Test Driven Development, 이하 TDD)라는 것도 결국 수정이 일어났을 경우에 대처하기 쉽게 만들어 주는 하나의 방법으로 사용된다. 또 코딩을 할 때 기능 단위로 작게 만들고 모듈 간의 커플링을 최대한 줄이는 것도 수정을 하는데 편리하게 작용한다.

그렇지만 이미 만들어 놓은 것을 외적인 요인에 의해서 변경해야 하는 것은 상당한 스트레스를 받는 일이다. 이럴 때는 미리 처음부터 수정할 가능성을 염두에 두고 작업을 하는 것이 덜 스트레스를 받는 방법일 것이다. 만약 기능상의 수정에 대한 여파가 크다면 회의 시간에 강력히 주장을 하는 것이 좋다. 그리고 나서 결국 수정을 해야 할 경우에는 순순히 따라야 한다. 괜히 회의 시간에는 조용히 있다가 나중에 코딩을 하면서 불만을 토로하는 것은 전혀 도움이 되지 않는다. 오히려 자신에게는 감점 요인이 되고 스트레스에 의한 개발 지연의 원인이 될 것이다.

테스트 주도적인 개발?

XP에서는 TDD를 강조한다. TDD는 간단히 설명하면 먼저 테스트를 만들어 놓고 개발하라는 얘기이다. 즉, someMethod()라는 메쏘드를 만들기 전에 someMethod()를 테스트하는 프로그램을 만들고 someMethod()를 구현하라는 얘기이다. 그렇게 되면 someMethod()는 아직 작성되지 않는 기능이기 때문에 일단 테스트 프로그램에서는 컴파일 에러가 발생할 것이다(아니면 간단하게 blank method를 먼저 만들어 놓을 수도 있다). 그러면 이제 someMethod()를 구현한다. 그리고 테스트 프로그램을 돌려서 제대로 동작을 하는지 확인하고 원하는 결과 값이 나오지 않을 경우에는 다시 someMethod()를 수정하고 다시 테스트를 돌려 본다. 이런 식으로 개발하는 것을 XP에서는 TDD라고 말한다.

이런 식으로 만들어진 테스트들은 나중에 상당한 효력을 발휘한다. 특히 리팩토링한 이후에 이 테스트들을 전부 수행시켜 보면 리팩토링이 어떤 버그를 만들어 내지 않았는지 쉽게 확인이 가능하다. 실제로 프로그램을 개발하다 보면 버그를 수정하는 데 걸리는 시간이 상당하다. 즉, 잘 만들어진 테스트 프로그램들이 존재한다면 버그를 수정하는 데 걸리는 시간이 상당히 단축될 수 있다는 말이다. 또한 유지 보수나 모듈들의 통합을 추진할 때 무언가 수정이 필요하더라도 테스트들을 갖고 있다면 부담 없이 수정할 수 있다. 즉, 수정한 후에 테스트를 돌려보면 에러가 나는 부분을 쉽게 찾을 수 있기 때문이다.

그러나 개인적으로는 TDD가 좋은 방법인 것을 알지만 보통 개발을 하다 보면 완전히 테스트 주도적인 개발을 하기는 쉽지 않다. 그럴 때는 보통 특정 기능을 만든 이후에 테스트를 작성하게 된다. 물론 그렇게 하면 XP에서 말하는 것과는 다르게 되고 실제 코딩 상에서의 이점은 없어 보이지만, 개인적인 경험으로는 XP에서 말하는 테스트를 먼저 만드는 방식은 쉽게 적용이 힘들었다. 또 습관 때문인지는 몰라도 아무것도 없는 상태에서 무언가를 하는 것은 쉽지가 않다. 테스트를 하더라도 무언가를 가지고 하는 것이 훨씬 심적으로나 효율 면에서 좋다고 생각한다. 그렇지만 가급적이면 먼저이든지 아니면 나중이든지 테스트는 작성하는 것이 프로젝트에 도움이 된다.

버그를 쉽게 잡자

디버깅 이야기가 나온 김에 계속 이야기를 해보자. 개인적으로는 프로그램을 작성할 때 필요 없이 시간을 투자해야 하는 부분이 바로 디버깅이라고 생각한다. 디버깅이 필요가 없다는 말이 아니고 디버깅에 투자되는 시간이 아깝다는 얘기이다. 때로는 버그를 추적하는 데 하루 종일 걸리기도 한다.

예전에 개발했던 프로그램에서 특정 클래스에 static하게 Database connection 정보를 넣어서 사용했던 적이 있다(물론 좋지 않은 방법이고 필자가 작성한 모듈이 아니었다). 개발 당시에는 test1 서버를 사용했다. 그런데 이 프로그램을 검수하기 위해서 또 다른 테스트 서버를 만들었다. 이 서버는 test2라고 했

자신만의 리소스를 구축해라

개발할 때는 여러 자료를 참고하는 경우가 많다. 대부분의 경우 특별한 문제가 발생하거나 아니면 잘 모르는 기술을 사용해 코딩할 경우에는 반드시 리소스가 필요하다. 리소스는 특정한 코드 샘플이나 웹 사이트, 뉴스 그룹, 책, 라이브러리 등 다양하다.

개발을 하면서 참조하는 이러한 리소스들을 잘 모아서 분류해 놓으면 유용한 경우가 많다. 그리고 그것을 이용해 자신만의 라이브러리, 리소스를 만들어야 한다. 개발을 잘하는 사람들을 보면 반드시 그 사람들만의 라이브러리나 리소스가 있음을 볼 수 있다. 그래서 무언가 자료가 필요하거나 문제가 생길 경우 그 리소스들을 사용해 금방 해결하는 것을 볼 수 있다. 이러한 리소스는 한 순간에 만들어지는 것은 아니고 차근차근 점진적으로 만들어야 할 것이다.

또 이러한 리소스를 구축하기 쉬운 방법 중 하나는 매일 조금씩 시간을 내서 자료를 정리하는 것이 가장 좋다. 필자의 경우 하루에 한 번은 자주 방문하는 자바 및 기술 웹 사이트에 들어간다. 그리고 새로 올라오는 기사나 Q&A를 읽어 보고 필요한 경우 저장해 놓는다. 이러한 것들이 쌓여서 지금은 꽤 많은 자료를 모을 수 있었다. 중요한 것은 본인이 이러한 것을 직접 만드는 것이다. 다른 사람이 만들어 놓은 것은 다른 사람의 것이다. 내 것이 아니기 때문에 실제 필요한 경우에 레퍼런스로서 별 효력을 발휘하지 못한다. 또 그 안에 무엇이 들어 있는지 쉽게 파악하기가 힘들다. 다른 사람의 것을 참고하는 경우에는 그것을 가져 와서 자신의 것으로 만드는 시간이 반드시 필요하다는 점을 잊지 말아야 한다.

다. 이상하게도 멀쩡했던 프로그램이 test2로 옮긴 후부터 계속 에러를 발생했다. 그것도 테스트 결과 특정 operation 중에 시스템이 정지해 버리는 에러였다. 그 후 계속해서 문제가 될 만한 메쏘드들에 System.out.println을 심어 가면서 에러를 추적했다. 결론적으로 아까 말한 static하게 설정해 놓은 DB 정보가 틀린 것이다. 이 틀린 DB 정보를 갖고 connection을 얻어서 사용하는 부분에서 에러가 발생한 것이고, 이 에러를 적절하게 처리하지 못해서 발생한 에러였다. 너무 황당한 결론이었다. 결국 DB 정보를 수정하고 다시 시스템을 작동시키니까 훌륭하게 돌아갔다. 중요한 점은 메쏘드의 stack을 추적하는 동안 하나의 로그를 뿌려주는 코드를 보지 못했다는 것이다. 만약 DB connection을 얻어오는 부분에서 로그를 남기도록 했었다면 이 에러를 수정하는 데는 불과 몇 분만 있었어도 될 것이다.

디버깅에 투자하는 시간을 줄일 수 있는 최고의 방법은 물론 버그를 안 만들어 내는 것이겠지만, 버그 없는 프로그램은 있을 수 없다. 하다못해 가장 간단한 HelloWorld 프로그램도 환경에 따라서 버그를 발생할 수도 있다. 그러므로 최상의 방법은 버그를 쉽게 찾을 수 있도록 하는 것이다. 아마 개인마다 버그를 쉽게 찾기 위한 방법이 있을 것이다.

그 중에서도 가장 좋은 방법은 로그(log)를 이용하는 방법일 것이다. 로그를 남기는 방법은 가장 고전적이면서 많이 사용되는 방법이다. 가장 간단하게는 System.out.println("....")으로 화면에 보여주는 메시지도 로그이고 파일로 저장해 놓는 것도 로그이다. 로그를 남길 때는 로그를 남기는 위치와 관련 메시지를 잘 정의해서 사용해야 한다. 특히 로그에서 남겨 놓는 메시지는 다른 사람이 봐서 쉽게 이해할 수 있도록 해줘야 한다. 가끔 보면 로그로 'HAHA', 'SKYWATCH', 'Line 100' 등과 같이 이해할 수 없는 메시지를 보이도록 해서 사용하는 사람들을 봤다. 하지만 이것은 별 도움이 안 되는 메시지이다. 로그를 남길 때는 적어도 로그가 남는 위치, 남겨진 로그의 레벨, 관련 메시지 등을 남기면 좋다. 보통 내 경우에는 로그가 남는 위치는 Classname.methodname()와 같이 남겨 둔다. 동일 이름의 메쏘드가 또 있을 경우에는 파라미터의 타입까지 써주는 경우도 있다. 그리고 로그의 레벨은 현재 남긴 로그가 어떤 로그인지 나타내 준다. 내 경우에는 'FATAL', 'ERROR', 'INFO', 'DEBUG'로 분류해서 로그를 남긴다. 메시지에 대해서는 때에 따라서 필요한 내용을 보여준다. 어떤 때는 특정 변수의 값을 어떤 때는 Exception의 메시지를 보여준다.

때로는 실제 시스템을 릴리즈할 때에 로그를 안 남기도록 설정하거나 컴파일을 해서 릴리즈를 하는 경우가 있다. 개인적인 의견으로는 그다지 좋지 않은 방법이라고 생각한다. 릴리즈한 시스템은 물론 가급적이면 버그가 발생하지 말아야 하지만, 절대라는 것은 없다. 문제가 생겼을 때 로그가 없다면 그 문제의 원인조차 찾기 힘들다. 그리고 어떤 상황에서 그런 문제가 생기는지도 알 수가 없다. 퍼포먼스에 크게 민감한 시스템이 아니라면 로그는 반드시 남기는 것이 좋다고 생각한다. 그리고 시간이 되고 금전적인 능력이 된다면 디버깅 툴을 구입해서 익히는 것 역시 좋은 방법이다. 요즘 디버깅 툴은 상당히 막강한 능력을 발휘한다.

그리고 추가적으로 얘기하고 싶은 것은 로그를 남기는 것은 어느 정도 상위 레벨에서 남기는 것이 좋다. 여기서 말하는 레벨은 시스템의 단계를 말한다. 지나치게 로우 레벨의 모듈에서 로그를 남기는 것은 오히려 좋지 않은 방법이다. 로그를 적용하기 전에 먼저 어느 정도 레벨의 모듈에서 로그를 남길 것인지를 미리 정하고 그 레벨까지는 exception이나 error code 등을 활용해서 에러를 처리하는 것이 좋다. 즉, 비즈니스 로직을 담당하는 모듈에서만 로그를 남기기로 했다면 그 안에서 사용하는 하위 모듈에서는 로그를 남기지 말고 exception을 넘기는 것이 좋다. 예를 들어 SomeBeans class에서 로그를 처리하기로 했다고 하자. 그리고 SomeBeans에서는 FileManager라는 class를 사용해서 특정 정보를 파일에 저장한다고 하자. 그러면 FileManager에서는 파일에 대한 에러가 날 경우 적절한 Exception을 통해서 SomeBeans로 알려주기만 하면 되는 것이다. FileManager에서 로그를 남기지 말고 Exception을 통해서 파일이 존재하지 않아서 그러는 것인지, 파일에 대한 권한 때문인지를 알려주면 되는 것이다. 그리고 로그는 SomeBeans에서 남기면 된다. 이렇게 할 경우 전체적인 프로그램의 로그가 일관적으로 적용되어서 훨씬 유용하다.

상세한 스케쥴 관리, 나만의 노하우를 쌓자

"개인만의 스케쥴을 짜라. 단, 그 스케쥴은 가능하면 자세하게 짜는 것이 좋다." 그렇다고 일주일치 스케쥴을 모두 짜 놓고 그 스케쥴에 맞춰서 작업을 하겠다는 생각은 일찌감치 버리는 것이 좋다. 회사에서 개발하다 보면 종종 옆에서 태클(?)이 들어오는 경우가 많다. 또는 갑작스런 외근이라든지 아니면 회사 내에서 특정한 일 때문에 하루 정도 늦어지는 경우가 있다. 그렇게 되면 결국은 스케쥴을 다시 짜던가(물론 그 스케쥴을 짜는 데 많은 시간을 투자해야 할 것이다) 아니면 억지로 밤을 새서라도 밀고 나가는 것이다. 결국 두 가지 모두 좋은 결과를 가져오지는 않는다. 팀장한테 스케쥴이 안 맞는다고 한소리 듣거나 피곤에 지친 몸을 이끌고 한 주일을 지내야 하는 경우가 생길수도 있다.

필자의 경우에는 아침에 하루 일정을 계획한다. 대충 몇 시부터 몇 시까지는 어떤 기능을 구현하고 그 다음에는 어떤 작업을 하고 이런 식으로 보통 시간 단위로 계획을 작성한다. 내일 일은 걱정하지 않는다. 내일 일에 대한 일정은 내일 작성하면 된다. 어떻게 보면 무책임한 일이지만 의외로 효과가 높다. 즉, 하루에 해야 할 일을 자세하게 만들어 놨으니 그대로 따라가면 되는 것이다.

그래서 사용하는 것이 스토리 카드이다. 이는 원래 XP에서 빌려온 개념이다(예전 마소의 특집에 났던 것을 수정해서 사용하고 있다). XP에서 스토리 카드의 개념과는 물론 좀 틀리다. XP에서 말하는 스토리를 좀 더 작게 나누어 기입하기 때문이다. 보통 필자 스토리 카드에는 다음과 같은 내용들이 들어간다. XP에서 말하는 스토리는 그 자체로 완전한 기능을 말하지만, 필자의 경우에는 그 스토리를 좀더 작게 나눠 기입하기 때문이다. 즉, 스토리를 더 작은 세부 기능으로 나누고, 그 세부 기능을 모듈 등으로 나눠 기입한다(개인적으로는 그냥 작업이라고 한다. 예를 들어 xxx에 대한 테스트 작성, xxx에 대한 테스트와 같은 식으로 세분화한다. 그리고 가능하면 시간 레벨에서 스케쥴링을 한다).

1. 스토리 카드의 이름 - 보통 개발하고 있는 시스템의 이름이나 모듈 또는 기능의 이름을 넣는다.
2. 시작 날자, 마침 날자 - 이 스토리 카드를 시작한 날자와 끝낼 예상 날짜를 적는다.
3. 예상 소요 시간 - 스토리 카드 상의 작업을 모두 끝마치는 데 걸릴 예상 시간을 적는다.
4. 작업 요약 - 해야 할 작업들을 간단하게 요약해서 써놓는다.
5. 작업 - 실제 작업에 대한 내역을 적는다.
6. 내역 - 작업 내역(좀 상세하게 적는다)
7. 날짜 - 작업 날짜
8. 작업 시간 - 실제 작업한 시간, 계속해서 작업을 했을 경우에는 hh:mi~hh:mi와 같이 적지만 중간에 다른 작업을 했을 경우에는 실제 작업한 시간만을 적는다. 즉, hh:mi 1~hh:mi 1', hh:mi 2~hh:mi 2'와 같이 여러 개를 적어 준다.
9. 경과 시간 - 총 작업 시간
10. 난이도 - 난이도, 보통 간단한 문서 작업을 레벨 1로 코딩을 레벨 3, 4로 설계 등을 레벨 5로 해 1~5 단계로 적는다.

필자는 이런 식으로 작성한 후 사용해서 효과를 봤다. 특히 경과 시간의 경우에는 나중에 본인의 개발속도를 측정할 수 있는 자료가 되므로, 다른 모듈이나 시스템을 개발할 때 참고자료로 활용할 수 있다. 본인의 개발 속도를 측정할 수 있다는 것만으로도 이러한 스토리 카드는 충분히 적용할 만한 것이라고 생각한다. 또한 예상 시간과 실제 시간을 측정할 수 있으므로 개발자 본인에 대한 스스로의 평가를 내릴 수 있을 것이다. 부끄러운 얘기지만 아직 예상시간 내에 작업을 끝내 본적은 없다.

"자신만의 방법론을 찾아라"

개발 속도를 향상시키기 위한 특별한 왕도는 없다. 결국 개발 속도 향상을 위해서는 개발자 본인의 성장이 따라 붙어야 한다. IDE 툴을 사용해서 개발 속도를 높이는 것도 결국은 자신의 능력을 키워가는 것이기 때문이다. 여러 가지 툴을 사용하면 물론 개발 속도를 높이는 데 도움은 되겠지만 한계가 있게 된다. 즉, 프로그램의 기능 분석을 하고 설계를 하는 게 UML을 사용할 수는 있지만 UML만 쓴다고 기능 분석이나 설계가 자동으로 되는 것은 아니다.

앞에서 얘기한 많은 것들은 궁극적으로는 좋은 개발 습관을 갖기 위한 나만의 방법론을 말한 것이다. 물론 필자가 말한 것들은 필자가 느끼고 경험한 것을 바탕으로 스스로 실천하고 있는 방법론이기 때문에 다른 사람에게는 안 맞을 수도 있고 이해가 안되는 부분도 많을 것이다. 그렇지만 이런 식으로 개발하는 사람도 있다는 참고 자료로서 보고 자신만의 방법론을 만들어 가는데 이 글이 도움이 되었으면 한다. 중요한 것은 다시 한 번 강조하지만 자신 만의 속도 향상 비법을 만들어 가는 것이다.

정리 : 조규형 jokyu@sbmedia.co.kr

30%를 아껴라

스케쥴을 잡을 때 가장 좋은 것은 정확한 시간을 예측해서 잡는 것이다. 그렇지만 원래 정확한 예측은 불가능하다. 때로는 현재 하고 있는 일보다 우선 순위가 높은 일이 갑자기 들어와서 일을 할 경우도 있고, 기술적으로 문제가 생겨서 시간이 늦어질 수도 있다. 그렇다고 한 번 잡은 일정은 쉽게 늘려 주지 않는다. 그래서 개인적으로는 스케쥴을 잡을 때 본인의 능력의 70%~80% 정도로 일을 했을 때 마칠 수 있는 정도가 적당하다고 생각한다. 경험상 그렇게 일정을 잡아도 결국 100%로 일을 해야 끝나는 경우가 많았다. 그런데 만약 100%로 일정을 잡았다면 아마 매일 밤을 코드와 싸우면서 보내야 했을 것이다. 물론 시간이 정해진 상태에서 작업을 하는 SI 프로젝트의 경우는 아마 이렇게 하기도 힘들 것이다. 국내 현실상 심할 경우에는 주 7일 근무를 하는 불행한 경우도 있기 때문이다.