Hanbit eBook

Realtime 61

Thinking About

C/C++

프로그램 개발편

프로그래머가 몰랐던 프로그램의 동작 원리

박수현 지음

한빛미디어 Hanbit Media, Inc.

Thinking About

C/C++

프로그래머가 몰랐던 프로그램의 동작 원리

프로그램
개발편

Thinking about C/C++: 프로그래머가 몰랐던 프로그램의 동작 원리 프로그램개발편

초판발행 2014년 3월 27일

지은이 박수현 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-693-7 15000 / **정가** 9,900원

책임편집 배용석 / **기획·편집** 정지연
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 서은옥, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea
Copyright © 2014 박수현 & HANBIT Media, Inc.
이 책의 저작권은 박수현과 한빛미디어(주)에 있습니다.
저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_박수현

홍익대학교 컴퓨터 공학 학사부터 박사까지 마쳤으며, 현재 현대오토에버에 재직 중이다. 약 12년간에 걸친 홍대 생활로 인하여 잘 놀 것 같다는 오해를 자주 받고 있다. 사실 홍대 앞 번화가에 대해서는 잘 모르지만, 홍대 근처 어느 집에서 자장면을 시켜야 맛있는지는 조언해 줄 수 있다. 운영체제, 시스템 프로그래밍에 관심이 많다.

저자 서문

중고등학교 시절 나를 끊임없이 괴롭히던 한 가지 생각은 “이 많은 것들을 배워서 도대체 어디에 써먹는 것일까”였다. 물리학자도, 수학자도 될 생각이 없었기에 미적분은 그저 성적 유지를 위해서 공부하는 것에 불과했다. 대학교에 진학하고 컴퓨터 공학을 전공하면서도 이 생각은 그대로였다. 자료구조와 알고리즘 수업을 듣는다고 해서 멋있는 게임을 만들 수도 없었고, 컴퓨터 구조나 운영체제 수업을 듣는다고 임베디드 장치를 만들 수 없었기에 그 허탈감은 이루 말할 수 없었다.

대학원 진학을 고민하던 시절, 술자리에서 한 선배가 필자에게 질문을 던졌다. “넌 명색이 컴퓨터 공학과 졸업자인데 컴퓨터가 어떻게 부팅되는지 정확하게 설명할 수 있느냐”라고 말이다. 망치로 머리를 얻어맞은 느낌이었다. 비록 모든 내용을 다 배우지는 못했지만 4년 동안 컴퓨터 공학 전공 수업을 들으면서 한 부분씩은 배웠을 텐데 이들이 어떻게 연결되고 어떻게 동작하는지 이해하려는 시도조차 하지 않았다. 수동적으로 배우기만 하고 내가 만든 프로그램이 어떻게 동작하는지, 무엇이 문제인지 고민조차 해 보지 않은 것이다.

대학원에 진학하고 6년이 넘도록 컴퓨터 시스템에 관련된 공부를 하면서 가장 많이 받은 질문은 “왜 인기도 없는 시스템을 공부하느냐”였다. 답은 간단했다. 어떤 언어를 쓰고 어떤 프로그램을 만들고 어떤 운영체제를 사용하든지 그 근간은 시스템이라고 생각하기 때문이다. 아무리 능숙한 프로그래머라 할지라도 언어의 기본적인 개념을 이해하지 못하거나, 프로그램이 동작하는 근본 원리나 플랫폼의 세부 사항을 알지 못한다면 이는 능숙한 ‘코드 작성자’밖에 안 된다. 실무를 통해 겪었든 수업이나 책을 통해서 공부했든 자신이 몸담은 분야에 대한 기반 지식이 없다면 살아남기 힘들다고 생각한다. 기반 지식보다 더 문제인 것은 이미 배운 내용을 등한

시하고 사용하지 않는다는 점이다. 아마도 이는 필자가 느낀 것과 마찬가지로 이들을 어디에 써먹어야 할지, 왜 필요한지를 이해하지 못하기 때문이라고 생각한다.

이 책은 그래서 쓰기 시작했다. 컴퓨터 공학을 전공하지 않았거나 잘 모르는 사람이 보기에는 조금 어려울 수도 있고, 실무 경험이 많거나 컴퓨터에 대한 지식이 많은 사람이 보기에는 너무 쉬울지도 모른다. 책을 쓰는 데 많은 도움을 주신 분의 평가처럼 대상 독자층이 참으로 '애매모호'하다.

하지만 나는 이 책을 통해서 여러분이 배웠을지도 모르는 그 지식이 어떻게 사용될 수 있으며 왜 중요한지 꼭 이야기하고 싶었다. 한 줄의 코드 라인에 캐시의 철학을 담을 수 있는 사람이 진정한 프로그래머라고 생각한다. 그리고 이 책을 읽는 많은 분은 캐시가 무엇인지 그리고 어떻게 프로그램을 만드는 것인지 이미 다 알고 있다고 생각한다. 다만 왜 캐시가 중요하고, 캐시가 어떻게 프로그램에 영향을 미치는지 그리고 작성하는 코드가 캐시의 동작에 어떤 의미를 가지는지를 모를 뿐이라고 생각한다. 비단 캐시뿐만 아니라 이 책에서 설명하는 모든 부분은 익히 아는 내용일 것이다. 다만 이들이 여러분의 프로그램과 어떤 연관이 있는지 모를 뿐이다.

사실 이 책에서 이야기하는 내용은 몰라도 회사에서 일하고 프로그램을 만들고 돈을 버는 데에는 아무 지장이 없다. 다만 이 책을 다 읽고 덮을 때 즈음 여러분이 프로그램의 소스 코드뿐 아니라 그 외에 많은 부분을 둘러볼 수 있는 좀 더 넓은 시야를 갖게 되길 바란다.

마지막으로 이 책이 나오기까지 많은 도움을 주신 모든 분께 감사의 말을 전한다.

대상 독자 및 참고사항

초급	초중급	**중급**	중고급	고급

이 책은 C/C++ 등 프로그래밍 언어에 어느 정도 익숙하며 프로그램을 만들고 실행해 본 경험이 있는 중급 이상의 프로그래머를 대상으로 한다.

이 책에서는 주로 POSIX를 따르는 Unix 계열의 운영체제와 C/C++ 언어를 주로 다루고 있다. 이 책에 수록된 예제는 대부분 이해를 도우려고 작성된 슈도 코드 또는 스켈레톤 코드 형태이며 상당수가 불완전하다. 각 코드의 실행 환경은 대부분 설명 부분에 기재하였으나, 없는 경우 리눅스와 GCC를 사용하였다고 보아도 무방하다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위해 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 내려받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

차례

01 누구나 알기 쉬운 소프트웨어의 모든 것 1

1.1 소프트웨어의 미덕 1
1.2 소프트웨어의 구조 5
1.2.1 파이프와 필터 5
1.2.2 데이터 추상화와 객체 지향 구조 6
1.2.3 이벤트 기반 구조, 묵시적 호출 7
1.2.4 레이어 시스템 9
1.2.5 저장소 시스템 9
1.2.6 테이블 기반 인터프리터 10
1.3 소프트웨어 패턴 11
1.3.1 MVC 11
1.3.2 싱글턴 13
1.3.3 전략 패턴 15
1.4 디자인 패턴 17
1.5 애자일 개발 프로세스 21
1.6 소프트웨어 저작권 24
1.7 정리 28

02 C와 C++를 중심으로 본 프로그래밍 언어 29

2.1 이름이 비슷하다고 비슷한 언어일까 29
2.2 C와 C++는 무엇이 다른가 31
2.2.1 절차적 프로그래밍 언어 VS 객체 지향 프로그래밍 언어 31
2.2.2 절차적 프로그래밍 언어 – C 32

2.2.3 객체 지향 프로그래밍 언어 – C++ 32
2.2.4 C와 C++에 대한 엇갈린 평가 55
2.3 정리 59

03 멀티 코어 시대의 C와 C++ 60

3.1 Race Condition 62
3.2 자원 공유 문제 69
3.3 동기화 문제 74
3.4 멀티 코어와 성능 84
3.5 안전한 함수 사용 88
3.6 정리 93

04 C와 C++의 문제아, 함수 포인터 94

4.1 함수 포인터는 왜 쓸까 94
4.2 업콜과 다운콜 99
4.3 함수 포인터 사용법 109
4.4 인터페이스 설계하기 114
4.5 가상 함수 테이블 119
4.6 함수 포인터의 대체 방법 124
4.7 정리 127

1 | 누구나 알기 쉬운 소프트웨어의 모든 것

컴퓨터는 하드웨어와 소프트웨어의 조합이다. 소프트웨어에는 운영체제나 컴파일러와 같은 시스템 프로그램도 있고, 메모장이나 게임과 같은 응용 프로그램도 있다. 무엇을 만들든지 이들은 모두 소프트웨어다. 소프트웨어는 그 자체만의 구조가 있고 데이터가 있으며 코드가 있다. 어떤 언어를 사용하더라도 그 결과물을 하드웨어에서 실행할 수 있다면 소프트웨어라 할 수 있다. 물론 하드웨어를 직접 사용할 수도 있고, 자바 가상 머신이나 에뮬레이터 같은 가상의 하드웨어에서 실행할 수도 있으며 인터프리터를 통해서 실행할 수도 있다. 이 장에서는 소프트웨어가 무엇인지, 소프트웨어를 만들 때 무엇을 고려해야 할지 그리고 그 외에 소프트웨어에 관련된 다양한 요소들을 짚어보도록 하겠다.

1.1 소프트웨어의 미덕

소프트웨어는 그 종류에 따라서 갖추어야 할 요소들이 서로 다르다. 운영체제와 같은 시스템 소프트웨어는 안정성과 신뢰성이 높아야 하고, 좋은 성능을 보여야 한다. 응용 프로그램 소프트웨어는 사용하기 쉽고 다양한 기능을 제공하는 것이 좋을 것이다. 프로그래머가 UI[User Interface]나 UX[User eXperience]를 모두 책임지고 개발하지는 않지만 적어도 이들을 고려해서 코드를 작성하고 프로그램을 만들어야 할 것이다. 프로그래머는 소프트웨어의 내부를 책임지지만 소프트웨어를 바깥에서 바라보는 관점을 결코 무시해서는 안 된다.

그렇다면 좋은 소프트웨어란 무엇일까? 소프트웨어의 품질을 평가하는 방법은 매우 다양하므로 딱 잘라 말할 수는 없다. 누군가에게 매우 좋은 프로그램이 누군가에게는 저질 프로그램으로 느껴질 수도 있다. 하지만 소프트웨어라고 하면 누구나 공감할 수 있는 중요한 요소가 있으며 이들을 파악함으로써 여러분이 만드는 소프

트웨어가 어떤 부분이 부족한지, 어느 곳을 개선해야 하는지 생각할 수 있게 된다. 다음은 2011년 정보통신산업진흥원에서 작성한 '소프트웨어 기술성 평가기준 적용 가이드[01]'에 수록된 표로 상용 소프트웨어에 대한 평가 기준을 제시하고 있다.

[표 1-1] 소프트웨어 기술성 평가기준 적용 가이드(출처: 정보통신산업진흥원)

평가부문	평가항목	평가기준
기능성	기능구현 완전성	제안 요청서에서 요구하는 기능이 모두 구현되어 있는지 평가한다.
	기능구현 정확성	구현된 모든 기능이 정상적으로 동작하는지 평가한다.
	상호 운용성	제안 요청서에서 요구하는 다른 프로그램 또는 시스템과의 연동(데이터교환, 인터페이스 요구 충족 등) 가능 여부를 평가한다.
	보안성	인가되지 않은 사람이나 시스템의 접근을 방지하여 정보 및 데이터를 보호하는지 평가한다.
	표준 준수성	제안 요청서에서 요구하는 규제 또는 표준을 준수하여 개발되었는지 평가한다.
사용성	기능학습 용이성	도움말, 매뉴얼 등을 통해 제품 기능 정보를 제공하여 학습이 용이한지 평가한다.
	입출력 데이터 이해도	데이터 입출력 방법 및 절차가 편리하고 제안 요청서의 요구 내용에 적합한지 평가한다.
	사용자 인터페이스 조정가능성	사용자 요구조건에 맞게 화면구조(메뉴, 화면배치 등)를 변경할 수 있는지 평가한다.
	사용자 인터페이스 일관성	동일하거나 유사한 기능 수행을 위해 일관된 또는 통합된 인터페이스를 제공하는지 평가한다.
	진행 상태 파악 용이성	사용자가 수행하는 작업의 진행 상태를 쉽게 파악할 수 있는 화면을 제공하는지 평가한다.
	운영 절차 조정가능성	사용자 취향이나 습관에 맞게 운영 절차를 최적화할 수 있는 기능을 제공하는지 평가한다.
이식성	운영 환경 적합성	제안 요청서에서 요구하는 사용 환경에 설치 가능한지 평가한다.
	설치 제거 용이성	제품 설치나 제거 시 다운되거나 중지되는 현상이 발생하지 않는지 평가한다.
	하위 호환성	이전 버전이 있을 경우 이전 데이터를 사용할 수 있는지 평가한다.

01 https://www.nipa.kr/board/boardView.it?boardNo=101&contentNo=156&menuNo=30&gubn=&page=1&keywords=2011&fieldType=all

효율성	반응 시간	제안 요청서에서 요구하는 시스템 반응 시간 충족 정도를 평가한다.
	자원 사용률	제안 요청서에서 요구하는 부하요건 하에서 시스템 자원(CPU, 메모리, HDD 등) 사용의 적정성을 평가한다.
	처리율	제안 요청서에서 요구하는 부하요건 하에서 시스템이 처리할 수 있는 데이터 처리량을 평가한다.
유지보수성	문제 진단/ 해결 지원	오류가 발생했을 경우 오류를 해결할 수 있는 진단 기능을 제공하는지 평가한다.
	환경 설정 변경 가능성	시스템 확장 또는 효율적 운영을 위한 환경 설정 변경이 가능하고 변경이 용이한지 평가한다.
	업데이트 용이성	제품의 기능 또는 성능 향상을 위한 업데이트가 용이한지 평가한다.
	백업/복구 용이성	사용자가 원하는 시점에 시스템을 백업하고 필요 시 복원할 수 있는지 평가한다.
신뢰성	운용 안정성	시스템을 장시간 운용 시 안정적으로 동작하는지 평가한다.
	장애복구 용이성	시스템 장애 발생 시 복구가 용이하고 정상적으로 기능이 동작하는지 평가한다.
	서비스 지속성	시스템 장애 발생 시에도 지속적인 서비스가 가능한지 여부를 평가한다.
	데이터 회복성	시스템 장애 발생 시에도 데이터 소실 없이 유지 또는 복구 되는지 평가한다.
공급업체 지원	유지보수 지원	제품 사용상 문제가 발생하거나 제품 업데이트 필요 시 이를 지원할 수 있는 절차, 인력, 유지보수기간 등이 적절하며 라이선스 정책이 제안 요구사항을 충족시킬 수 있는지 평가한다.
	교육훈련 지원	구매할 제품의 사용방법 및 관리방법과 관련된 사용자와 관리자에 대해 지원되는 교육훈련 과정 및 교육 전담 인력 지원 평가한다.
	제품 신뢰도	GS인증 등 품질인증을 획득하였으며, 지적재산권과 관련한 법적인 문제가 없는지, 제품 개발 후 업그레이드가 있었는지 평가한다.
	직접생산 여부	단독 또는 공동수급체 구성을 통한 입찰참가 시, 입찰참가자가 중소소프트웨어사업자이며 중소기업 제품 구매촉진 및 판로지원에 관한 법률에 따른 직접생산확인증명서를 제출한 경우 최고 등급을 부여한다. 다만, 대기업인 소프트웨어사업자가 저작권법, 특허법 등 관련 법률에 따라 직접 생산을 증명하거나, 직접 생산하지 않는 중소소프트웨어사업자가 참여하는 경우에는 최고 등급보다 한 단계 하위 등급을 부여하고 이에 해당하지 아니하는 소프트웨어사업자는 최하위 등급을 부여한다.

이 평가 기준은 정부나 공공기관이 사용할 소프트웨어의 평가를 위한 것이기 때문에 모든 상황에 맞지는 않는다. 예를 들어 자동차 제어 시스템을 만드는 경우 사용자 인터페이스는 고려 대상에서 제외될 수 있다. 또한, 사용자 인터페이스나 기능 학습 용이성과 같은 항목은 프로그래머가 담당하는 부분이라기보다는 UI 디자이너나 소프트웨어 기획자가 처리할 항목에 더 가깝다. 하지만 이런 가이드 라인을 한 번쯤 읽어봄으로써 여러분이 만들고 있는 소프트웨어가 무엇이 부족하고 무엇이 불필요한지, 어떤 점을 개선할 것인지를 파악할 수 있다.

소프트웨어는 만드는 것만큼 그에 대한 평가 또한 중요하다. 어느 정도의 성능을 보이는지, 문제가 발생하지는 않는지를 파악해야만 그에 대한 대처가 가능하다. 소프트웨어 회사에서는 이런 이유로 제품의 안정성을 담당하고 확인하는 별개의 조직을 구성하여 운영하는 경우가 많다. 바로 QA[Quality Assurance] 혹은 QM[Quality Management] 팀이 바로 그것이다. 이들은 개발 중이거나 완료된 제품에 대해서 동작을 검증하고 시험하며 이 정보를 개발팀에 피드백하는 역할을 담당한다. 소프트웨어 테스팅은 단순히 소프트웨어를 구동하고 사용해 보는 것이 아니다. 과부하가 걸리거나, 예측할 수 없는 데이터를 주는 등 다양한 환경에서 구동해보고 이에 따른 소프트웨어 동작을 살펴본다. 어디서 어떤 문제가 발생하는지 다각적으로 연구하고 분석하기 때문에 대부분 QA는 실력 있는 엔지니어가 담당한다.

NOTE

프로그래머가 소프트웨어를 평가하게 되면, 프로그램을 작성할 때 생각한 설계나 동작 방식을 염두에 두고 테스트하기 때문에 오류를 찾기가 힘들다. 완제품이 발매되고 사람들이 사용하기 시작하면 다양한 환경에 노출된다. 완제품에서 치명적인 결함이 발견되면 고객들이 얼마나 불쾌하겠는가. 같은 제품이라도 사람에 따라서 사용 환경이 제각각이다. 이런 이유로 전문적인 QA 담당자가 필요한 것이다. 그들은 프로그래머가 감히 상상도 못 하는 다양한 방법으로 소프트웨어를 괴롭힌다. 그리고 사용자들이 평생 못 볼지도 모르는 기기묘묘한 버그들을 알려주기도 한다.

1.2 소프트웨어의 구조

사용자가 바라보는 소프트웨어는 어떤 형태일까? 사용자나 테스터에게 소프트웨어는 일종의 블랙박스다. 내부 구조나 기능이 어떻게 구현되어있는지 알 수 없고 알 필요도 없다. 사용자의 관심사는 소프트웨어가 얼마나 잘 동작하느냐지, 소스 코드가 얼마나 잘 정리되어 있고 유지 보수가 편한가가 아니다.

소프트웨어의 구조 및 설계, 소스 코드 작성은 오로지 프로그래머의 몫이다. 소프트웨어 구조가 외부에서 보았을 때 명백히 드러나는 경우가 있기는 하다. 대표적으로 서버-클라이언트 모델인데 소프트웨어를 사용하는 입장에서 소프트웨어가 어떤 서버에 접속해서 작업의 일부 또는 대부분을 처리하게 될 때 이 소프트웨어의 구조가 서버-클라이언트 모델에 기반을 두었음을 알 수 있다.

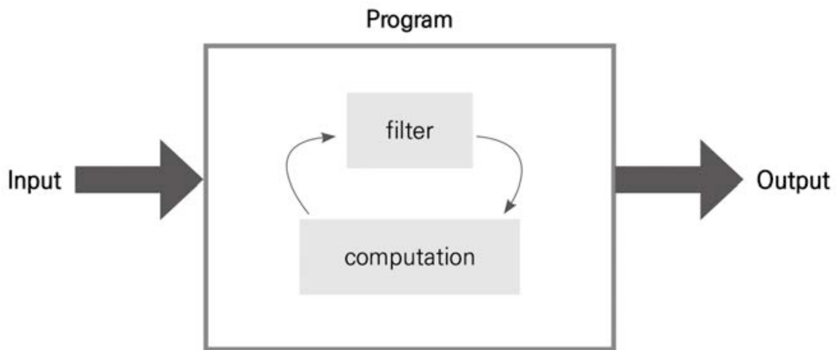
그렇다면 소프트웨어 구조는 어떤 일을 할까? 간단히 설명하면 소프트웨어 전체를 설계하고, 각 부분이 어떤 일을 담당할지를 정한다. 소프트웨어의 구조는 소프트웨어별로 천차만별이지만 크게 몇 가지 스타일로 나눌 수 있다. 1994년 데이비드 갈란David Garlan과 메리 쇼Mary Shaw가 작성한 「소프트웨어 구조 개론(An Introduction to Software Architecture)」 문서에 따르면 소프트웨어 구조는 크게 '파이프와 필터Pipes and Filters', '데이터 추상화와 객체 지향 구조Data Abstraction and Object-Oriented Organization', '이벤트 기반 구조, 묵시적 호출Event-based, Implicit Invocation', '레이어 시스템Layered System', '저장소 시스템Repositories', '테이블 기반 인터프리터Table Driven Interpreters'로 나눈다. 이 소프트웨어 구조 스타일은 단일 소프트웨어의 내부뿐 아니라 둘 이상의 소프트웨어 간 협력도 규정하고 있다. 여기에서는 각각의 스타일이 무엇인지를 간략히 살펴보도록 하겠다.

1.2.1 파이프와 필터Pipes and Filters

이 스타일은 간단히 말해서 프로그램에 어떤 입력이 주어지면 거기에 상응하는 출

력을 기대하는 것이다. 프로그램이 입력을 받아들여서 이를 가공하고 필터링해서 필요한 연산을 수행한 다음, 그 결과를 외부에 출력하는 방식이다. 연속적인 입력을 지속해서 처리하는 배치 작업Batch Job에 적합한 형태로, 대표적으로 유닉스의 셸이 이러한 구조로 되어있다. 파이프와 필터 스타일은 프로그램 설계가 단순하고 매우 직관적이며 필터를 추가함으로써 새로운 기능을 추가하고 확장하기가 쉽다. 따라서 재사용성이 극대화될 수 있다. 하지만 너무 단순해서 복잡한 프로그램을 설명하는 데는 적합하지 않다.

[그림 1-1] 파이프와 필터 구조도



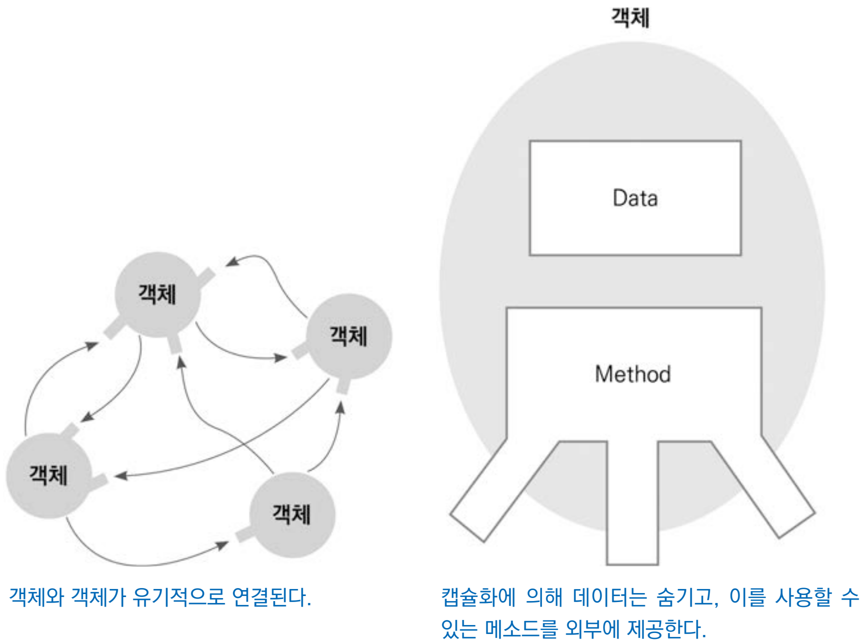
프로그램은 입력을 받아들이고 내부에 정의된 규칙에 따라 출력을 내보낸다. 아주 단순한 구조이며 연속적인 일을 처리하는 데 효과적이다.

1.2.2 데이터 추상화와 객체 지향 구조Data Abstraction and Object-Oriented Organization

객체 지향으로 프로그램을 설계하는 방식이다. 프로그램이 처리해야 하는 데이터와 이에 관련된 처리 부분을 하나의 모듈로 생각한다. 따라서 이 스타일은 각각의 모듈이 하나의 객체로 구성된다. 객체는 자신의 데이터를 책임지고, 자신을 사용할 수 있는 메소드를 외부에 제공한다. 객체 내부의 구현이나 데이터는 외부에 공개되지 않기 때문에 객체를 사용하기 위해서 객체의 상세를 알 필요가 없다는 것이 장점이다.

객체 지향 언어의 보급과 더불어 널리 사용되는 소프트웨어 구조 스타일 중 하나가 되었다. 이 스타일은 파이프와 필터 스타일보다 더 복잡한 양상을 띠며 객체가 변경되면 다른 객체를 수정해야 할 수도 있다는 점이 문제가 된다. 물론 수정 사항을 줄이려고 객체 지향 프로그래밍의 원칙이 제시되기도 하였지만, 근본적으로 객체 간의 의존성이 발생하고 구조가 복잡해지는 점은 피할 수 없다.

[그림 1-2] 데이터 추상화와 객체 지향 구조도



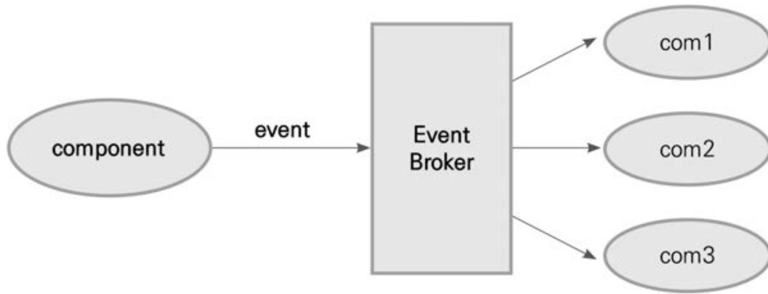
객체와 객체가 유기적으로 연결된다.

캡슐화에 의해 데이터는 숨기고, 이를 사용할 수 있는 메소드를 외부에 제공한다.

1.2.3 이벤트 기반 구조, 묵시적 호출Event-based, Implicit Invocation

소프트웨어를 여러 모듈이나 객체로 나누어 생각하면, 각 모듈이나 객체는 자신의 기능을 사용할 수 있는 함수 등을 외부에 제공하게 된다. 앞에서 설명한 데이터 추상화와 객체 지향 구조에서도 각 객체는 다른 객체가 사용할 수 있는 메소드를 외부에 제공하고, 직접 이 메소드를 호출하여 객체를 사용한다.

[그림 1-3] 이벤트 기반 구조도



이벤트 브로커는 외부에서 전달되는 이벤트 종류에 따라서 이를 처리할 함수나 요소를 호출한다.

그러나 이벤트 기반 구조는 이렇게 메소드나 함수를 직접 호출하는 것이 아니라, 처리할 기능을 모듈이나 요소에 알려주고 각 모듈이나 요소는 요청된 이벤트에 알맞은 함수를 수행하는 방식이다. 함수나 메소드를 직접 호출하지 않기 때문에 묵시적 호출이라는 용어가 함께 사용된다.

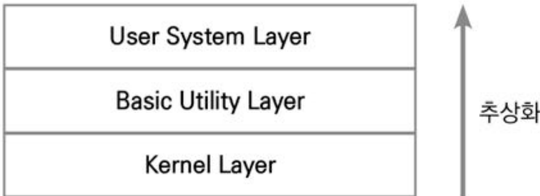
묵시적 호출을 사용하면 코드 재사용성이 매우 높아진다. 메소드나 함수의 이름 또는 구현이 변경되더라도, 이들을 사용하는 객체는 직접 호출이 아니라 이벤트 브로커를 통해서 묵시적으로 호출하므로 이들의 이름이나 실제 구현된 내용을 알 필요가 전혀 없다. 즉, 이벤트 브로커[Event Broker]를 사용하는 모든 요소는 수정할 필요가 없다는 것이다.

하지만 이벤트를 발생시킨 개체는 그에 대한 응답을 언제 받을 수 있는지 혹은 응답을 받을 수 있는 것인지는 전혀 알 수가 없다. 또한, 이벤트를 통해서 데이터를 주고받는 방식이 힘든 경우가 있다는 것이 또 다른 문제점이라 할 수 있다. 예를 들어 배치 작업처럼 연속적인 데이터를 처리해야 하는 프로그램은 프로그램이 실행되는 동안 별도의 외부 이벤트가 전혀 발생하지 않을 수도 있다. 이러한 프로그램은 이벤트 기반 구조로 작성되면 필요한 함수를 적시에 호출할 수 없는 상황이 발생한다.

1.2.4 레이어 시스템Layered System

레이어 시스템은 소프트웨어를 여러 계층으로 나누고, 각 계층은 상위 계층에게 필요한 메소드나 함수를 제공하며 하위 계층의 기능을 사용하는 방식이다. 하위 계층에서 상위 계층으로 갈수록 더 높은 수준의 추상화를 제공해준다. 대표적으로 네트워크의 OSI 7 Layer를 들 수 있다.

[그림 1-4] 레이어 시스템 구조도



상위 계층으로 갈수록 추상화가 심해지며, 하위 계층은 구체적인 구현을 담당한다.

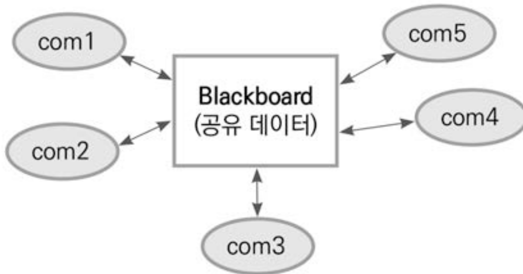
이 레이어 시스템은 높은 수준의 추상화를 제공할 수 있으며 각 계층에서 수정 사항이 생길 경우 그 영향은 최대 두 개의 계층(상부와 하부)에만 미치기 때문에 재사용성이 커진다. 하지만 모든 시스템을 계층 구조로 설명하기 힘들고, 각 계층에서 필요한 적절한 수준의 추상화를 구현하기 힘든 경우도 많다.

1.2.5 저장소 시스템Repositories

여러 개체 또는 모듈이 단일 저장소를 사용하는 구조다. 칠판Blackboard 구조라고도 하는데 여러 개체 또는 모듈은 하나의 사용자로서 칠판에 자유롭게 데이터를 기록하고 사용하는 구조다. 단일 저장소를 사용하는 모든 모듈이나 개체는 서로 데이터를 자유롭게 공유한다. 음성이나 패턴 인식 분야에서 많이 사용된다. 데이터를 조작하고 관리하는 제어 모듈이 별도로 존재하지 않으며, 각 개체는 서로 연관될 수도 있고 완전히 독립적으로 동작할 수도 있다. 이를테면 통합 개발 환경IDE은 동일한 소스 코드나 프로젝트에 대해서 코드를 관리하는 모듈과 코드를 편집하는 모듈,

그 외에 환경을 관리하는 모듈로 구성되어 있기 때문에 저장소 구조라고 할 수 있겠다.

[그림 1-5] 저장소 시스템 구조도

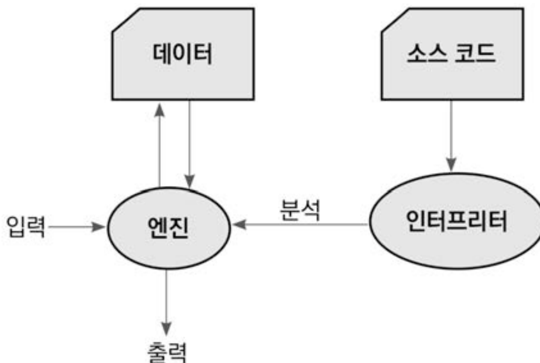


칠판은 각 요소가 데이터를 자유롭게 읽거나 쓸 수 있는 공유되는 데이터 영역이 된다. 단일 데이터에 대한 다양한 표현이나 조작에 적합한 구조라 할 수 있다.

1.2.6 테이블 기반 인터프리터Table Driven Interpreters

소프트웨어를 실행하는 환경에 인터프리터와 같은 가상 머신이 도입되는 경우다. 소프트웨어 자체보다는 가상 머신이나 인터프리터를 설계할 때 사용된다.

[그림 1-6] 테이블 기반 인터프리터 구조도



소스 코드 자체는 다른 구조에 따라 설계될 수 있으나, 이 소스 코드가 인터프리터에 의해 해석되고 실행되면 인터프리터 구조라 할 수 있다. 자바스크립트나 베이식이 대표적인 예이다.

이 외에도 소프트웨어 구조는 분산 처리 구조나 특정 도메인에 특화된 구조, 상태 전이 시스템 등 여러 방식이 있다. 소프트웨어 구조는 정해진 정답이 없으며 두 가지 이상이 혼합되어 구성될 수도 있다.

1.3 소프트웨어 패턴

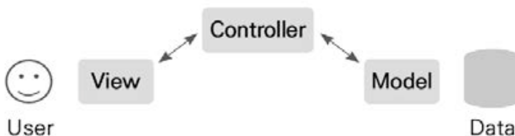
소프트웨어 구조는 스타일 외에도 패턴Pattern이라는 것이 있다. 스타일과 패턴을 독립적인 개념으로 보는 경우도 있고, 패턴이 스타일의 구체적인 형태로 보는 경우도 있지만, 이는 어디까지나 개인의 취향이기 때문에 스타일과 패턴을 딱딱하게 정의할 필요 없이 두 가지를 자유롭게 사용할 수 있다. 소프트웨어 구조 패턴은 디자인 패턴과도 유사하거나 혼용되는 부분이 많으므로 공부해 두는 것이 좋다.

여기에서 모든 패턴을 전부 설명할 수는 없으므로 널리 사용되는 몇 가지 패턴을 살펴보도록 하겠다.

1.3.1 MVCModel-View-Controller

GUI 프로그램에서 널리 사용되는 패턴이다. 이 패턴은 프로그램을 세 부분으로 나누어 구성하는데 이름에서도 알 수 있듯이 모델Model과 뷰View 그리고 컨트롤러Controller다. 모델은 쉽게 생각해서 프로그램이 사용하는 데이터와 데이터를 처리하는 부분을 의미한다. 컨트롤러는 사용자 입력 등에 반응해서 프로그램을 전체적으로 관리하고 제어하는 부분이고, 뷰는 사용자에게 표시되는 화면이라고 생각할 수 있다.

[그림 1-7] MVC 구조도



모델은 데이터, 뷰는 사용자에게 표시되는 화면 그리고 컨트롤러는 뷰와 데이터를 연결하는 로직 부분이다.

대부분의 소프트웨어 구조의 스타일이나 패턴이 그러하듯이 MVC 패턴 역시 각 요소의 재사용성을 극대화하고 유지 보수를 쉽게 하는 데 그 목적이 있다. 웹 페이지를 예로 들면, 서버에 있는 데이터베이스나 멀티미디어 파일들이 모델에 해당하며 컨트롤러는 사용자로부터 입력을 받아서 처리하고 모델을 가공하고 제어하는 역할을 담당하는 서버-사이드 스크립트를 생각할 수 있다.

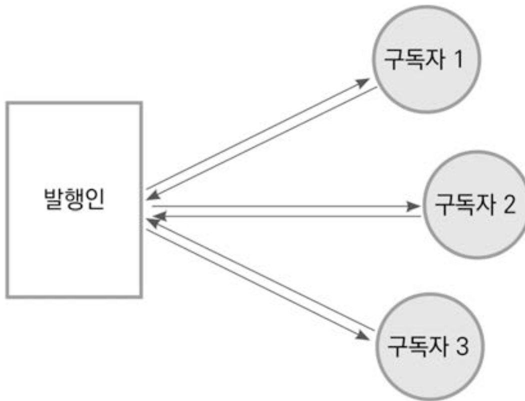
뷰는 서버에서 전송되는 HTML 파일을 가공하여 웹 브라우저에 내용을 출력하는 HTML이나 자바스크립트를 생각할 수 있다. 이렇게 분리하면 여러 개의 뷰를 만드는 것이 쉬워진다. 예를 들어 데스크톱에서 웹 페이지에 접근할 때와 스마트폰과 같은 모바일 장치에서 접근할 때 서로 다른 화면을 보여주도록 구성할 수 있다. 이 경우 두 개 이상의 서비스 루틴을 작성할 필요 없이 뷰 요소만 두 개 만들면 컨트롤러나 모델은 그대로 재사용이 가능하다.

하지만 MVC 패턴은 뷰와 모델을 구분하지 못하는 의존성 문제가 있다. 모델이 변경되면 당연히 뷰가 변경되어야 하나 구조적으로 모델은 뷰에 직접 갱신 내용을 알려주지 못한다. 그러다 보니 모델 변경에 맞추어 뷰를 변경하려고 이 둘을 한데 모아서 구현하는 일이 발생하기 시작한다. 그래서 모델 구조가 변경되면 뷰 또한 변경되어야 한다는 심각한 의존성 문제가 발생한다.

이 문제를 해결하기 위해서 옵저버 패턴Observer Pattern이 제시되었다. 발행-구독 모델이라고도 하는데 이 패턴은 한 객체의 상태가 변경되면 그 객체에 의존하는 다른 객체들에 메시지가 전달되며 자동으로 내용이 갱신되는 방식이다. 대개 객체와 객체 간에 일대다One-to-Many 관계가 형성된다. 예를 들어, 신문사가 신문을 발행하면 이 신문을 받아보는 여러 구독자가 신문의 내용을 읽을 수 있다. 여기서 하나의 신문사는 둘 이상의 구독자와 의존성을 형성하게 된다.

옵저버 패턴은 이벤트 기반 구조, 묵시적 호출 스타일과 비슷한 부분을 보인다. 변경에 관련된 이벤트가 발생하면 옵저버가 이벤트에 해당하는 일을 자유롭게 처리한다.

[그림 1-8] 옵저버 패턴 구조도



발행인에서 발생한 이벤트를 옵저버(구독자)들에게 전달한다.

재미있는 점은 옵저버가 실제로 관찰하지는 않는다는 것이다. 이벤트를 처리하는 곳이 옵저버로 등록된 객체들에 이벤트가 발생했음을 알려주므로 정확히 말하면 리스너Listener 정도의 용어가 적당하지 않을까 생각한다.

1.3.2 싱글턴Singleton

어떤 객체를 단 하나만 가지고 사용하도록 강제하는 구조다. 예를 들어 다음 코드를 살펴보자.

```
     #include <iostream>
     class Singleton
     {
     private:
        static Singleton *instance;
01      Singleton(void){}
     public:
        ~Singleton(void){}
```

```
    public:
02      static Singleton *getInstance()
03      {
04          if (instance == NULL)
05          {
06              instance = new Singleton();
07          }
08          return instance;
        }

    private:
    };

09  Singleton *Singleton::instance = NULL;

    int main()
    {
10      Singleton *singletonA = Singleton::getInstance();
11      Singleton *singletonB = Singleton::getInstance();
12      Singleton *singletonC = Singleton::getInstance();

        std::cout << singletonA << "\t" << singletonB << "\t" << singletonC <<
        "\n";
        return 0;
    }
```

---

01 일반적인 생성자와 달리 private이다. 따라서 일반적인 방법으로 객체를 만들 수 없다.

02 ~ 08 외부에서 객체를 얻거나 생성하기 위해 호출하게 되는 정적 메소드다. 이 메소드를 통해 단 하나의 객체를 만들고 유지한다.

09 정적 멤버 변수에 대한 초기화가 필요하다.

10 ~ 11 동일한 객체에 대한 참조를 얻게 된다.

[실행 결과]

```
$ g++ singleton.cpp
$ ./a.out
0x8fde008 0x8fde008 0x8fde008
```

우선 코드에서 주목해야 할 부분은 객체의 생성자 private이다. 즉, 객체를 생성할 때 생성자를 호출할 수 없으므로 객체를 생성할 수 없다. 대신 getInstance() 메소드를 사용해서 이미 생성된 객체를 반환하거나 객체가 없다면 내부적으로 객체를 생성하고 저장한다. 그 결과 프로그램의 모든 곳에서 동일한 객체에 접근하는 것을 볼 수 있다. 대개 프로그램 전체에서 단 하나만 존재해야 하는 객체는 싱글턴으로 구현하는 경우가 많다.

1.3.3 전략 패턴Strategy Pattern

정책 패턴Policy Pattern이라고도 불리는데 여러 알고리즘을 캡슐화해서 구현하고, 실행 시간에 이들 중 하나를 선택하거나 기존의 알고리즘을 다른 알고리즘으로 교체하여 실행할 수 있도록 구성하는 것을 의미한다. 즉, 정책에 포함되는 것은 캡슐화되어 구현된 여러 알고리즘 모듈이며, 프로그램은 실행 시간 동안 정책에 있는 알고리즘 중 하나를 선택하여 동작할 수 있다. 다음 코드를 살펴보자.

```
     class Algorithm
     {
     public:
        virtual int cal(int a, int b) = 0;
     };

01   class Add : public Algorithm
02   {
```

```
03  public:
04      int cal(int a, int b){ return a + b;}
05  };

06  class Sub : public Algorithm
07  {
08      int cal(int a, int b) { return a - b; }
09  };

10  class Mult : public Algorithm
11  {
12  public:
13      int cal(int a, int b) { return a * b; }
14  }

15  class Div : public Algorithm
16  {
17  public:
18      int cal(int a, int b) { return a / b; }
19  }

20  class Strategy
21  {
22  private:
23      Algorithm strategy;
24  public:
25      Strategy(Algorithm src) { strategy = src; }
26      int doStrategy(int a, int b) { return strategy.cal(a, b); }
27  };

    int main()
    {
```

```
28      int resultA, resultB, resultC, resultD;
29      Strategy *str;
30      str = new Strategy(new Add());
31      resultA = str->doStrategy(20, 10);
32      str = new Strategy(new Sub());
33      resultB = str->doStrategy(20, 10);
34      str = new Strategy(new Mult());
35      resultC = str->doStrategy(20, 10);
36      str = new Strategy(new Div());
37      resultD = str->doStrategy(20, 10);
        return 0;
    }
```

01 ~ 19 Algorithm 클래스의 자식 클래스다. 구체적인 알고리즘(혹은 정책)을 구현한다.
20 ~ 27 정책을 포함하고 사용할 목적으로 작성된 클래스다. 구현 자체는 델리게이트와 비슷하다.
28 ~ 37 각 정책(또는 알고리즘)을 가지는 클래스 객체를 만들어서 상황에 따라 다른 로직을 사용한다.

프로그램에서는 Add나 Sub, Mult, Div 클래스의 cal 메소드를 직접 호출하지 않으며 모든 객체 호출은 Strategy 클래스를 통해서 이루어진다.

1.4 디자인 패턴

소프트웨어 구조 패턴Architectural Pattern 외에도 디자인 패턴Design Pattern이 있다. 디자인 패턴에서 빼놓을 수 없는 사람들이 바로 GoFGang of Four인 에릭 감마Gamma, Erich 리처드 헬름Richard Helm, 랠프 존슨Ralph Johnson 그리고 존 블라시디스John Vlissides 네 명이다. 이들은 『GoF의 디자인 패턴(Design Patterns: Elements of Reusable Object-Oriented Software), PTG, 2007』이라는 책에서 처음으로 디자인 패턴이라는 용어를 사용하였다. 디자인 패턴은 사실 이를 곧바로 소스 코드로 만들 수 있는 완전한 명세서 형태는 아니다. 다만 일반적으로 소프트웨어를 만드는 과정에서 바람직하거나 권장할 만한 디자인을 모아서 정리하고, 프로그래머가 어떤 패

턴을 따를 때 전반적인 소스 코드 관리나 구현이 좀 더 용이하거나 유지 보수가 쉬워진다는 장점을 가지게 된다.

최근 패턴이 너무 많아져서 오히려 독이 되는 경우도 있지만, 기본적인 개념에 대해서는 알아두는 것이 좋다. 특히 일부 프레임워크는 디자인 패턴을 그대로 옮겨와서 구현하고 지원하는 경우도 있기 때문에 이러한 프레임워크를 이해하고 사용하기 위해서는 기반을 두는 디자인 패턴을 잘 숙지해야 한다.

디자인 패턴은 크게 생성 패턴Creational Pattern과 구조적 패턴Structural Pattern, 행위 패턴Behavioral Pattern 그리고 병행 패턴Concurrency Pattern으로 나눌 수 있다.

각 범주에 해당하는 패턴은 다음 표와 같다.

[표 1-2] 범주별 디자인 패턴

범주	종류
생성 패턴	Abstract factory, Builder, Factory method, Lazy initialization, Multiton, Object pool, Prototype, Resource acquisition is initialization, Singleton
구조적 패턴	Adapter(Wrapper or Translator), Bridge, Composite, Decorator, Façade, Flyweight, Front controller, Module, Proxy
행위 패턴	Blackboard, Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null object, Observer(Publish/subscribe), Servant, Specification, State, Strategy, Template method, Visitor
병행 패턴	Active object, Balking, Binding properties, Double-checked locking, Eventbased asynchronous, Guarded suspension, Lock, Messaging design pattern(MDP), Monitor object, Reactor, Read-write lock, Scheduler, Thread pool, Thread-specific storage

표에서 앞에 언급한 소프트웨어 구조 패턴과 동일한 이름이 몇 가지 보인다. 사실 패턴이라는 것은 구조에 대한 정리 또는 정의이기 때문에 이것을 소프트웨어와 시스템 전반에 걸쳐서 적용할 것인지 혹은 프로그램의 특정 도메인에 적용할 것인지에 따라 구조 패턴이나 디자인 패턴으로 구분하게 된다. 결국, 둘 사이에 큰 차이점은 존재하지 않는다.

우선 생성 패턴은 디자인 패턴 중에서 객체 생성에 관련된 부분이다. 객체를 만들고 관리하는 방법에 따라서 프로그램이 단순해지기도 하고 복잡해지기도 한다. 예를 들어 앞에서 살펴본 싱글턴은 프로그램 전체에서 지정된 클래스에 대한 객체를 오직 하나만 생성하고 사용한다.

예를 들어, 공장Factory 패턴은 객체를 생성하고 관리하는 용도로 전체 객체를 만들어내는 Factory 클래스와 객체가 존재하며 개별적으로 객체를 생성하는 것이 아니라 모든 객체를 Factory 객체를 통해서 생성한다. 말 그대로 공장에서 물건을 찍어내고 사용하는 개념이다.

[그림 1-9] 공장의 컨베이어 벨트[02]

Factory는 말 그대로 공장의 컨베이어 벨트처럼 필요한 객체를 그때그때 만들어서 사용자에게 제공한다. 객체를 만들고 초기화하고 관리하는 것은 모두 Factory 객체에 의해 이루어지기 때문에 사람이 직접 만드는 것보다 실수나 오류가 훨씬 적다. 그렇다고 완전 자동화는 아니다. 설비도 결국 사람이 만드는 것이니까.

구조적 패턴은 프로그램의 각 요소가 어떤 구조로 연결되는지에 초점을 둔 디자인

---

02 Copyright 2009 Kay-Uwe Rosseburg. 이 이미지는 크리에이티브 커먼즈 〈저작자표시-일반 3.0 이용허락〉에 따라 이용할 수 있다.

패턴이다. 예를 들어 어댑터Adapter 패턴은 안드로이드가 사용하고 있는 패턴인데 서로 다른 두 개 이상의 인터페이스 또는 메소드 간 차이를 제거하는 것이 목적이다.

안드로이드의 버전이 올라가면서 기존에 제공하던 methodA( )가 사라지고, newMethod(A)를 대신 제공한다고 가정하자. 그러면 기존 안드로이드 기기에서 동작하던 프로그램은 상위 버전의 안드로이드에서는 동작하지 않는다. 이 상황을 해결하기 위해서 상위 버전의 안드로이드는 methodA( )를 제공하고, 이 메소드 내부에서는 newMethod(A)를 호출한다. 이때 상위 버전의 안드로이드가 제공하는 methodA( )가 바로 어댑터가 된다.

행위 패턴은 객체 간에 어떤 식으로 메시지가 오가는지에 초점을 맞춘 디자인 패턴이다. 표에서도 알 수 있듯이 이전에 살펴본 옵저버 패턴(발행-구독 모델)이 포함되어 있다. 옵저버 패턴은 관심 있는 이벤트에 객체를 등록하고, 실제 이벤트가 발생할 때 이 정보를 등록된 객체에 알려주는 방식이다. 즉, 객체로 메시지를 전달하는 방법으로 옵저버 패턴을 사용할 수 있다.

마지막으로 병행 패턴은 멀티프로그래밍 환경에 적용할 수 있는 디자인 패턴이다. 이 패턴은 대부분 동시에 여러 코드가 실행될 때 발생할 수 있는 문제를 디자인 패턴을 통해서 해결하고자 한다. 예를 들어, 모니터Monitor 패턴은 어떤 임계 구역을 감시하는 모니터가 존재하고, 모니터 때문에 임계 구역에는 여러 프로세스 혹은 스레드 중 단 하나만 진입할 수 있게 된다. 읽기-쓰기 잠금Read-write lock 패턴은 공유되는 데이터에 대한 접근을 관리하기 위한 패턴으로, 어떤 객체가 공유 데이터를 읽는 것은 몇 개의 객체가 동시에 읽든지 모두 허용하지만, 어느 하나의 객체라도 쓰기를 시도하는 경우 해당 객체를 제외한 모든 객체는 읽기와 쓰기가 금지된다.

소프트웨어 구조 패턴이든 디자인 패턴이든 알아두면 좋지만, 반드시 이들을 따를 필요는 없다. 패턴이 너무 많아졌으며, 프레임워크나 패턴을 따르지 않으면 능력 없는 프로그래머로 취급받는 일부 인식은 매우 부당하다고 생각한다. 이런 패턴이

나 프레임워크는 좀 더 효율적으로 소프트웨어를 만들고 유지 보수하자는 취지에서 시작한 것이지, 결코 소프트웨어를 만드는 데 굴레가 되어서는 안 된다. 스파게티 코드로 작성된 프로그램이라고 하여도 프로그래머가 잘 이해할 수 있고 유지 보수가 쉽다면, 이 자체도 좋은 소프트웨어라 할 수 있다. 물론 대부분은 그렇지 않겠지만 말이다.

1.5 애자일 개발 프로세스

소프트웨어 개발 프로세스 중에서도 애자일Agile에 대해서 잠깐 이야기해 보겠다.

사실 개발 프로세스에 대해서는 소프트웨어 공학 등에서 잘 정의가 되어 있고, 독자 중 상당수는 해당 수업을 듣거나 공부했을 것이다. 폭포수Waterfall 개발 프로세스나 프로토타입 모델링 등은 한 번쯤 들어보았을 것이다. 패턴과 마찬가지로 개발 프로세스에 대한 모델 역시 많은 소프트웨어 개발 회사나 팀 또는 개인이 겪었던 경험을 바탕으로 좀 더 효율적으로 소프트웨어를 개발하고 관리하고 유지 보수하고자 제시되는 것들이다.

하지만 문제는 소프트웨어 개발 기획과 소프트웨어 개발 사이의 괴리다. 누군가는 매우 철두철미하게 계획을 세워서 소프트웨어를 만들고 싶어 하고, 누구는 아무런 계획 없이 소프트웨어를 만들려고 할 것이다. 계획을 철두철미하게 세운다고 해서 좋은 소프트웨어가 일정에 맞추어서 완성되지 않고, 아무 계획 없이 만들어진 소프트웨어가 정말 높은 완성도를 가질 수도 있다. 하지만 대부분은 어느 정도 계획이 세워진 소프트웨어가 높은 완성도를 보이기 마련이다. 문제는 그 '적당한 계획'이 어느 수준인가다.

개발 프로세스 중에서 폭포수 모델은 완전한 계획에 기반을 둔 모델이다. 요구 사항을 분석하고, 소프트웨어를 디자인하고, 이에 기반을 두어 구현하고, 소프트웨어를 검증한 뒤 유지 보수한다. 이전 단계가 끝나면 그 단계로 다시 돌아가지 않는다.

따라서 계획이 잘못되면 그다음 모든 단계는 틀어지게 된다.

이 문제를 해결하기 위해서 반복 개발 프로세스Iterative and Incremental Development Process가 도입되었다. 이것은 앞선 단계에서 문제가 발생하게 되면 전체 프로세스를 재순환한다. 기획하고, 디자인하고, 구현하고, 검증한 뒤 문제가 생기면 기획을 수정하고, 디자인을 고치고, 코드를 바꾼 뒤 다시 검증하는 과정을 반복한다. 크게 문제는 없어 보이지만 이 모델 역시 기획과 디자인에 크게 의존한다.

애자일 개발 프로세스 모델이 괜찮다고 생각하는 가장 큰 이유는 전통적으로 문서에 기반을 둔 개발 모델이 아니라 코드에 기반을 둔 개발 모델이라는 점이다. IT에서 소프트웨어 개발은 자동차를 만들거나, 전자 기기를 설계하고 생산하는 것과는 다르다. 다른 공학적 생산도 요구 조건을 분석하고, 제품을 기획하고 디자인하며, 실제로 생산하고 검증하고 판매하는 과정을 거치기 때문에 개발 과정만 비교하면 큰 차이가 없다. 하지만 소프트웨어는 다른 공학적 개발 제품과 달리 변경이 쉽고, 기능의 추가나 삭제가 매우 쉽다. 요구 조건은 시간이 지남에 따라서 계속 변동이 생길 수 있고, 기존에 개발된 소프트웨어를 수정하고 재사용하거나 혹은 완전히 변경할 수도 있다. 새로운 버전의 소프트웨어는 이전 소프트웨어와의 호환을 염두에 둘 필요성도 있다. 새로운 자동차 모델이 나오면 핸들을 서로 바꿔 끼울 수 있는가? 모르는 일이다.

그들은 호환성에 대해서 거의 고려하지 않는다. 결국 소프트웨어는 다른 공학적 개발 프로세스와는 다른 방식의 접근이 필요하며 이러한 맥락에서 애자일 개발 프로세스 모델이 제시된 것이다.

애자일 개발 프로세스 모델을 딱 잘라서 설명하긴 힘들지만, 간단히 말해서 “Make it run and make it”이라고 할 수 있다. 장황한 계획이나 디자인 없이 우선 프로토타입을 개발하고, 검증하고 테스트하며 그 과정에서 새로운 기능을 추가하거나 요구사항을 반영하는 식으로 소프트웨어가 점점 커지는 방식이다. 사실 어떤 특정한

개발 방법론을 지칭하는 것이 아니라 소프트웨어 개발에 있어서 변동 사항을 매우 기민하게 받아들일 수 있는 모든 방법을 애자일 모델이라 할 수 있다.

[그림 1-10] 애자일 소프트웨어 개발 포스터[03]

익스트림 프로그래밍eXtreme Programming이 애자일 개발 모델의 좋은 예라 할 수 있다. 소규모 소프트웨어 개발에 특히 강점을 보이는 익스트림 프로그래밍은 소프트웨어 개발에서 전체 계획이나 디자인보다 소스 코드 자체를 문서로 생각하며, 개발 팀 전체가 아니라 개발을 담당하는 프로그래머 개개인을 중요시한다. 주어진 부분에 대해서 자유롭게 개발하고, 이들을 통합하여 테스트하고 검증한 뒤, 다시 프로그래머 각자가 수정 사항을 반영하고 통합하고 테스트하고 검증하는 과정을 반복

03 Copyright 2011 VersionOne, Inc. 이 이미지는 크리에이티브 커먼즈 〈저작자표시-동일조건변경허락 3.0〉에 따라 이용할 수 있다.

한다. 어떻게 보면 기존의 반복 개발 프로세스와 유사하지만, 그 규모가 작고 요소가 적어 수정 사항에 매우 민첩하게 반응할 수 있다.

애자일 개발 프로세스 모델 보급을 위해서 만들어진 애자일 연합은 2001년 '애자일 소프트웨어 개발 선언문Manifesto for Agile Software Development'을 발표하였다.

이 선언문이 담고 있는 내용은 다음과 같다.

- 프로세스와 도구를 넘어선 개인과 융합
- 복잡한 문서가 아닌 동작하는 소프트웨어
- 계약과 협상이 아닌 고객과의 협력
- 계획 준수가 아닌 변화에의 반응

결국, 개발 프로세스에서 프로그래머의 비중을 더 높이고, 문서를 통한 계약이나 복잡한 절차를 생략하고 소스 코드를 위주로 소프트웨어를 개발하는 데 더 집중하자는 뜻이 된다. 소프트웨어 사용 설명서 등은 제외하더라도 프로그램의 디자인이나 세부 명세 등은 별도의 문서가 아닌 소스 코드의 주석 또는 좋은 함수나 변수 이름으로도 충분히 설명할 수 있지만, 기존의 개발 프로세스 모델들은 문서 작성에 지나치게 의존하여 결국 배보다 배꼽이 더 큰 경향을 보이게 되었다. 프로그래머에게 문서 작업이 스트레스가 되고 개발 시간을 빼앗아 가며 결국 야근과 납품일 초과로 밤샘 작업을 강요하게 하는 악순환이 된다.

1.6 소프트웨어 저작권

지금까지의 내용과는 약간 동떨어진 이야기이지만 소프트웨어에서 저작권을 빼놓고 이야기할 수는 없다. 소프트웨어를 보호하기 위해서도 저작권이 중요하지만, 반대로 소프트웨어를 만드는 데 사용하는 라이브러리 등이 어떤 라이선스를 따르느냐에 따라 골치 아픈 문제가 발생할 수도 있기 때문이다.

소프트웨어에 대한 특허를 허용하는 나라도 있지만, 국내에서는 소프트웨어에 대한 특허 출원은 불가능하며 저작권을 통해서 법적 권리를 주장할 수 있다. 많은 나라에서도 국내와 마찬가지로 라이선스 형식으로 소프트웨어에 대한 권리를 주장한다. 특히 오픈 소스 진영이 커지고 많은 오픈 소스 기반 소프트웨어가 개발되면서 이들을 사용할 때 라이선스 분쟁이 심심찮게 발생하는 것을 볼 수가 있다. 아마도 가장 많이 맞닥트리는 라이선스가 GNU GPLGeneral Public License가 아닐까 한다[04]. 자유 소프트웨어 재단Free Software Foundation이 진행하는 GNU Project의 산출물에 대한 관리를 위해서 제시되었으며 현재에는 GPL 버전 3까지 작성된 상태다. GPL 외에도 오픈 소스 라이선스에는 LGPL, BSD License, Apache License, MPLMozilla Public License가 있다. 오픈 소스 라이선스는 일반적으로 아래와 같은 특성을 가진다.

- 소프트웨어를 자유롭게 사용할 수 있다.
- 소프트웨어를 자유롭게 복제할 수 있으며, 일정 조건을 만족하면 재배포할 수도 있다.
- 소프트웨어를 자유롭게 수정하여 사용할 수 있으며, 일정 조건을 만족하면 수정된 내용을 재배포할 수도 있다.
- 소프트웨어 소스 코드에 자유롭게 접근하고 사용할 수 있다.

결국, 사용자 입장에서는 어떻게 사용하더라도 큰 문제가 발생하지 않는다. 하지만 오픈 소스 라이선스를 따르는 소프트웨어나 라이브러리를 가져와서 개발을 진행할 때 라이선스가 문제를 발생시키지 않는지 반드시 고려해야 한다. 오픈 소스 라이선스를 따를 때 지켜야 하는 사항은 다음과 같다.

- 라이선스 관련 문구를 유지
- 동일한 소프트웨어 이름 사용 금지
- 서로 다른 라이선스의 조합

04 볼륨 라이선스나 OEM, 패키지 라이선스와 같은 소프트웨어 사용에 관련한 라이선스는 설명하지 않는다. 관심 있는 것은 소프트웨어의 생산이지 소비가 아니기 때문이다.

‘서로 다른 라이선스의 조합’은 예를 들어 프로그램을 만들 때 GPL을 따르는 라이브러리 A와 BSD를 따르는 라이브러리 B를 가져와서 사용할 경우 GPL과 BSD를 혼합하여 라이선스로 명시할 수 있는지를 잘 판단해야 한다. 만약 GPL이나 BSD 모두 다른 라이선스와의 혼용을 금지하는 경우 이 두 라이브러리를 함께 사용하는 것은 (법적으로) 문제가 있다.

오픈 소스 소프트웨어를 가져와서 사용할 때 회사나 프로그래머 입장에서 특히 생각해 보아야 하는 문제는 소스 코드 공개 의무다. 오픈 소스 라이선스 중에서 가장 엄격한 축에 속하는 GPL의 경우 소프트웨어를 수정하거나 다른 소프트웨어와 링크할 때에는 관련 소스 코드를 전부 공개해야 한다. 또한, 이렇게 만들어진 소프트웨어를 바이너리 형태로 제공할 때는 소스 코드를 함께 배포하거나 혹은 소스 코드를 받는 방법을 함께 제공해야 한다. 리눅스 커널의 경우 GPL 버전 2.0을 따르고 있으므로 리눅스 커널을 사용하는 모든 제품은 반드시 해당 제품에 사용된 리눅스 커널 소스 코드를 제공해야 하는 의무가 있다.

GPL이 엄격하다 보니 프로그래머 입장에서 오픈 소스 라이브러리 사용을 꺼리는 경향이 있었다. 이를 해결하기 위해서 GNU에서는 LGPL^Lesser General Public License^를 만들었다. LGPL에 따르면 라이브러리를 동적 링크^Dynamic link^할 경우에는 소스 코드나 오브젝트 코드를 모두 제공할 필요가 없다. BSD 라이선스는 소스 코드 공개 의무가 아예 없으므로 상용 소프트웨어 제작에 BSD 라이선스를 따르는 소프트웨어를 사용하는 데 있어서 아무런 제약이 없다. Apache 라이선스도 BSD 라이선스와 유사하게 소스 코드 공개 의무가 없다. MPL 라이선스는 GPL과 달리 공개해야 할 소스 코드의 범위가 명확히 명시되어 있다. 기존 소프트웨어에 포함된 파일이 수정된 경우 소스 코드를 공개해야 하지만 새로운 소스 파일은 공개할 필요가 없다.

많은 경우 라이선스에 대한 부분을 무시하고 오픈 소스 라이브러리를 가져와서 사용하는데, 오픈 소스 이니셔티브 쪽에서는 이러한 라이선스 위반 사례에 대해서 법

적 대응을 하기 시작하였다. 아이폰에서 동영상 재생을 위해 개발된 AVPlayer라는 앱이 있는데 이 앱은 어떤 형태로 인코딩된 동영상도 다시 인코딩할 필요 없이 아이폰에서 그대로 재생해서 볼 수 있다는 점을 내세워 상당한 인기를 끌었다. 이 앱은 FFmpeg 라이브러리를 가져와서 사용했다. FFmpeg 라이브러리는 오디오와 비디오 스트림 재생을 위해 개발되었으며 LGPL 라이선스를 따른다.

문제는 바로 아이폰에서 사용하는 iOS 운영체제의 특성인데 이 운영체제는 앱에 대한 동적 링크를 허용하지 않으며 모두 정적 링크만 사용해야 한다. LGPL에 따르면 정적 링크인 경우 라이브러리 함수를 호출하는 부분의 소스 코드 그리고 그 외의 부분에 대한 오브젝트 코드를 제공해야 한다. 결국, 아이폰에서 FFmpeg 라이브러리를 사용하기 위해서는 함수 호출 부분에 대한 소스 그리고 나머지 앱의 오브젝트 코드를 공개해야만 한다. 물론 라이선스를 위반하였지만, 이는 상용으로 앱을 팔아서가 아니라 공개 의무를 이행하지 않았기 때문이다.

프로그램의 소스 코드가 매우 중요해서 공개해서는 안 되는 경우 BSD나 Apache와 같이 소스 코드 공개 의무가 없는 라이선스 라이브러리를 사용하길 권장한다.

라이선스 부분은 법적인 문제이기 때문에 일반 프로그래머가 이해하기는 쉬운 일이 아니다. 실제 라이선스 위반 문제가 발생하고 법적 대응이 발생하였을 때 그 문제는 프로그래머가 처리할 수 있는 수준이 아닐 것이다. 가장 바람직한 것은 미래에 발생할 수도 있는 라이선스 문제를 미리 방지하는 것이며 소프트웨어 개발 시 상용 혹은 오픈 소스 라이브러리 또는 소프트웨어를 사용할 때 반드시 라이선스 조항을 꼼꼼히 확인하고 사용해야 한다. 그리고 개발되는 소프트웨어가 환경에 따라서 라이선스 조항을 우회할 수도 혹은 그렇지 않을 수도 있으니 이 점을 항상 확인하고 염두에 두어야 한다. 가장 좋은 것은 모든 것을 직접 만들고 사용하는 것이지만, 여건이 그렇지 못하니 어쩔 수 없다. 가져와서 사용할 것은 사용하되 문제가 생길 상황을 만들어서는 안 된다.

1.7 정리

언어나 패턴, 개발 방법론 모두 프로그램을 좀 더 쉽고 효율적으로 작성하고, 유지보수가 쉽고 완성도 높은 소프트웨어를 만드는 데 목적이 있다. 소프트웨어의 종류가 다양해지고 그 목적 또한 천차만별이다 보니 언어도 많아지고 패턴이나 개발 방법론은 홍수라 해도 과언이 아닐 정도다. 항상 새로운 프레임워크나 라이브러리, 디자인 패턴을 따라가기에만 급급하고 요구 사항이나 디자인, 기획에 관련된 문서를 작성하느라 정작 코드는 한 줄도 작성하지 못하는 경우도 허다할 것이다. 의존 관계가 역전되어 버린 것이다.

소프트웨어는 결국 소스 코드로 이루어지는 것들이다. 소스 코드가 작성되지 않았는데 문서가 백만 장이면 무엇하겠는가! 패턴이고 언어고 개발 방법론이고, 소프트웨어를 만드는 데 방해가 된다면 빛 좋은 개살구일 뿐이다. 항상 공부하되, 이들에 얽매이지는 말자. 자신만의 패턴을 만들 수도 있고, 회사만의 개발 방법론이 있을 수도 있다. 소프트웨어를 만드는 주인공은 바로 자신임을 잊지 말자.