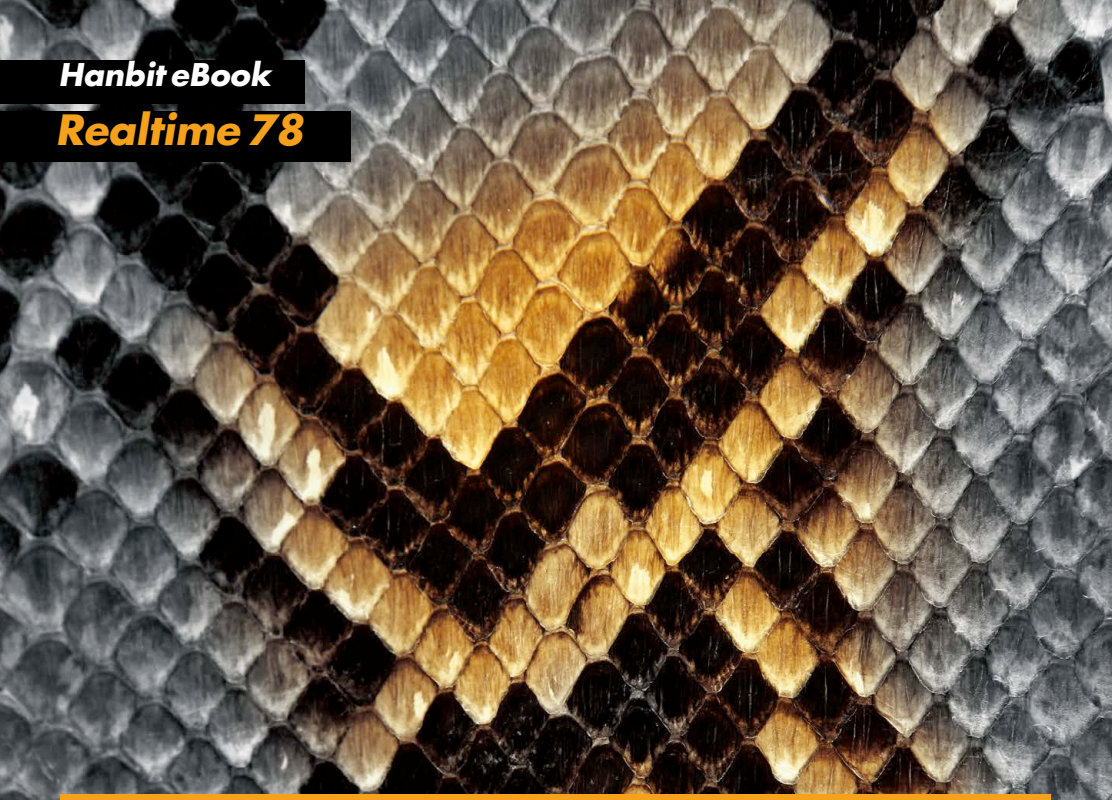


Hanbit eBook

Realtime 78

레거시 코드를 클린 코드로

누구나 쉽게,

리팩토링

신정호, 이규일, 박승규, 김태환, 정승욱 지음

한빛미디어 Hanbit Media, Inc.

레거시 코드를 클린 코드로

누구나 쉽게, 리팩토링

레거시 코드를 클린 코드로 **누구나 쉽게, 리팩토링**

초판발행 2014년 11월 18일

지은이 신정호, 이규일, 박승규, 김태환, 정승욱 / **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 978-89-6848-674-6 15000 / **정가** 12,000원

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

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 서문

현업에서 가장 많이 하는 일 중 하나는 기존 코드(레거시 코드)를 파악하는 것입니다. 즉, 작동하는 레거시 코드를 본다는 거죠. 시간이 어느 정도 흐르면 파악된 레거시 코드에 기능을 추가하거나 수정하는 일을 하게 됩니다. 하지만 많은 개발자가 현업에서 레거시 코드를 다룰 때 어려움을 토로합니다. 이는 레거시 코드를 개선하거나 활용하는 데 적지 않은 시간과 노력이 들고, 치미는 울화를 참으며 생기는 스트레스 때문입니다.

모든 레거시 코드가 그렇지는 않지만 대부분의 레거시 코드는 엉망일 것입니다. 또한 오랫동안 개선하지 않은 레거시 코드는 개선에 드는 일정을 산정할 수 없을 정도까지 되어 버리기도 합니다.

우리가 흔히 겪는 일의 하나는 자신의 코드를 자신이 알아보지 못하는 경우입니다. 레거시 코드의 개선 작업을 하는 가장 큰 이유가 여기 있습니다. 즉, 로직이나 기능을 구현한 코드를 다른 사람이 쉽게 파악하게 하기 위함입니다. 또 다른 이유는 기능의 확장이나 수정이 필요할 때 빠르고 안전하게 대응하기 위해서입니다.

물론 레거시 코드를 개선한다고 해서 이런 문제가 모두 사라지는 것은 아니지만 같은 문제가 발생하는 확률은 눈에 띄게 줄어들 것입니다. 그런데 레거시 코드의 개선은 많은 장점에도 불구하고 현업에서는 쉽게 행해지지 않습니다.

이유는 두 가지입니다. 첫 번째는 레거시 코드를 수정하면 예상치 못한 예외사항이 발생할 수 있기 때문입니다. 두 번째는 수정 작업에 많은 시간이 들기 때문입니다. 이를 개선하는 방법은 레거시 코드를 안전하고 빠르게 수정하는 것입니다.

지극히 형식적인 이야기이지만 다음의 간단한 방법을 수행해 본다면 결코 뜬구름

잡는 이야기가 아님을 알 수 있습니다. 먼저, 레거시 코드가 복잡하다면 TDD 또는 단위 테스트를 작성하여 구현체를 안전하게 수정합니다. 그리고 문제 해결 방법을 패턴화하여 구현체의 유형만 파악하면 쉽게 수정할 수 있도록 만드는 것입니다. 그래서 이 두 가지 방법으로 **안전하고 빠르게 레거시 코드 개선하기**라는 주제로 이 책을 집필하였습니다.

여러분은 이 책을 읽으면서 레거시 코드의 개선 방법과 경험을 얻게 됩니다. 그리고 비슷한 케이스가 자신의 코드에서 발견된다면 여기서 얻은 지식을 토대로 쉽게 개선할 수 있을 겁니다. 이것이 바로 문제 해결의 패턴화입니다.

그리고 복잡한 코드의 해결 과정에 적용한 TDD와 단위 테스트를 통해 여러분은 안전하고 쉽게 레거시 코드를 수정할 수 있는 방법을 알게 될 것입니다.

이 책을 쓰면서 집필에 참여한 모든 사람은 같은 글을 여러 번 읽어가며 수차례 회의를 통해 피드백하고 수정하기를 반복하였습니다. 조금이라도 더 여러분에게 이야기하고자 하는 바를 쉽게 전달하고 오해를 만들어내지 않기 위해서입니다.

물론 이 책은 레거시 코드를 좋은 코드로 리펙토링하는 모든 방법을 담고 있는 마법서가 아닙니다. 하지만 현업에서 느낀 개선할 수 있는 다양한 케이스를 알려주고 있습니다. 이를 통해 구현하는 코드의 크고 작은 문제점을 파악하는 시야를 넓혀줄 것입니다.

열정과 노력을 쏟아부은 책이 반드시 좋은 책이라는 법은 없지만, 좋은 책의 기본 조건은 열정과 노력이기에 최선을 다했고 열정을 쏟아부었습니다. 부디, 저희의 열정과 노력이 여러분께 조금이라도 도움이 되기를 간절히 기도합니다.

감사의 글

저자_ 신정호

가장 먼저 지혜와 용기를 주신 하나님께 감사드립니다. 쏟아지는 잠을 줄여가며 함께 집필에 몰두한 후배님들 감사합니다. 또한 책을 집필할 수 있도록 기회를 주신 한빛미디어 관계자 분들과 제가 몸담고 있는 다우기술 연구소 분들 그리고 신제품 개발팀, 조언을 아끼지 않고 독설을 퍼부어 주신 후배 전우균님과 장인선님 감사합니다. 불효자 아들의 건강을 걱정하시는 어머니와 아버지, 집필에 집중할 수 있도록 아낌없이 지원하신 장모님, 항상 저에게 비타민이 되어주는 아들 소명과 딸 소원, 그리고 엉뚱한 나의 모습도 사랑해 주고 응원해 준 나의 전부인 아내 이미라에게 고맙다는 말을 전합니다.

저자_ 이규일

이 책은 『TDD에 대한 오해와 진실, TDD이야기』(한빛미디어, 2013년)에 이어 두 번째 집필에 참여한 책입니다. 한 번 집필했으므로 이번에는 좀 더 수월하게 글을 쓸 수 있을 거라는 아주 잘못된 생각으로 시작하는 바람에, 글을 쓰는 동안 많은 어려움에 부딪치고 좌절하였습니다. 가장 먼저 이런 좌절을 이겨내고 끝까지 책의 집필을 마칠 수 있도록 당근과 채찍(?)을 주신 신정호 선배님과 동기에게 감사하다는 말을 전합니다. 또한 책을 쓴다는 핑계로 무관심했던 가족에게도 미안함과 고마움의 마음을 전합니다. 마지막으로 집필하는 동안 지쳐 있을 때마다 에너지를 공급해준 많은 친구와 저를 아끼고 사랑해 주신 많은 분께 감사의 마음을 전합니다.

저자_ 박승규

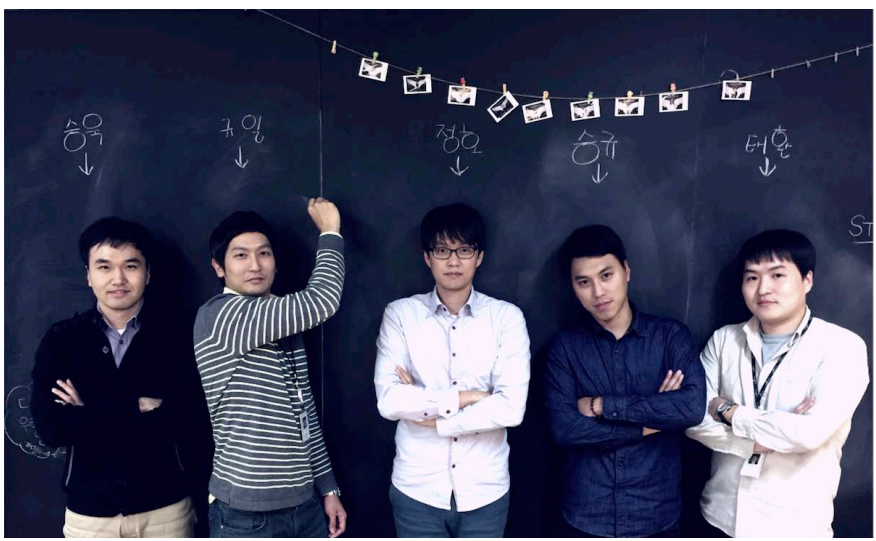
프로젝트 속에서만 개발을 생각하다가 밖에서 프로젝트를 볼 수 있는 시야를 가질 수 있게 해주신 신정호 선배님께 감사의 말을 전합니다. 개발자로 살면서 다른 위치에서 개발을 생각하며 고민할 수 있는 기회가 얼마나 많을지는 알 수 없지만, 개발 4년차에 이러한 기회를 얻고 참여하게 되어 매우 뜻깊은 시간이었습니다. 또한 글을 쓴다는 핑계로 소홀하였지만 짜증과 화를 내지 않고 응원해준 친구들, 개발에 대해 많은 생각을 하게 하고 능력이 향상되도록 도움을 주시는 다우기술 신제품 개발팀 분들, 항상 옆에서 응원해준 가족과 여자친구에게 고맙다는 말을 전합니다. 또한 이 책을 보시는 모든 분께 감사의 인사를 드립니다. 감사합니다.

저자_ 김태환

책을 쓸 수 있는 기회를 준 신정호 선배님께 감사합니다. 그리고 즐겁게 개발하도록 많은 도움을 주는 Groupware 개발팀에게 감사의 말을 전합니다. 책을 쓰는 과정은 어려웠지만 무사히 마무리할 수 있게 해준 10XP 분들과 항상 저를 응원해주는 많은 분께 감사의 인사를 드립니다.

저자_ 정승욱

새로운 경험에 도전할 수 있게 해준 공저자이자 선배 신정호님께 감사 인사를 드립니다. 함께 책을 쓰며 도와준 10XP 분들과 집필을 위해 배려해준 다우기술 신제품 개발팀 여러분, 리팩토링 공부에 많은 도움을 준 분들께 감사의 말씀을 전합니다.



이 책을 좀 더 쉽게 읽기 위한 방법

긴 코드를 다루는 개발 서적을 집중력을 유지하면서 계속 읽기란 매우 어렵습니다. 긴 코드를 읽다 보면 지루한 것은 물론이고, 이 코드가 무엇을 구현한 코드인지 금방 잊어버리곤 합니다. 또한, 필자의 의도가 무엇인지 되짚는 데 많은 시간이 걸리고 코드를 언급한 부분을 다시 읽어야 하는 번거로움이 생깁니다.

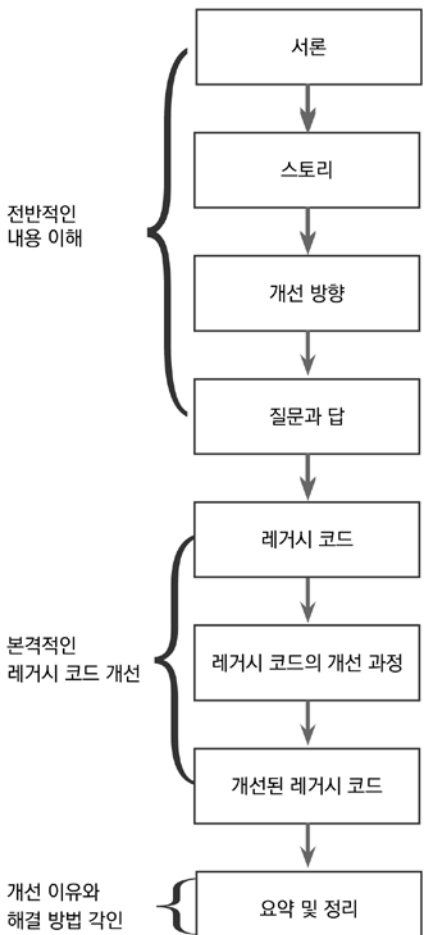
이 책 역시 "레거시 코드를 개선하는 방법"을 설명하기 때문에 앞에서 말한 따분하고 집중이 안 되는 긴 코드를 다뤄야 했습니다. 그리고 개발자인 저희에게 지식이나 생각을 글로 옮기는 것은 매우 어려운 작업이었습니다.

이런 어려움을 어떻게 극복하고 독자에게 좀 더 편하고 정확하게 지식을 공유할 수 있을지 많은 고민을 하게 되었고, 고민 끝에 다음의 방법으로 책에 녹여냈습니다.

1. 모든 장에서 일관된 규칙을 따를 것
2. 핵심 내용이나 어려운 내용은 박스 안으로 넣어서 잘 보이게 할 것
3. 레거시 코드와 개선의 포인트를 주기적으로 언급할 것
4. 이미지나 도표로 표현할 수 있는 것은 이미지나 도표를 사용할 것

이렇게 네 가지를 중점적으로 지키며 글을 썼습니다. 이 중 첫 번째를 잠깐 봐 주시기 바랍니다. 그래야 앞으로 볼 험난한(?) 글을 좀 더 잘 이해할 수 있습니다.

각 요소는 다음과 같은 의미가 있습니다.



서론

각 장에서 다루는 레거시 코드의 포괄적인 문제에 대하여 이야기하고, 문제의 원인과 발생하는 결과에 대해서도 이야기합니다.

스토리

서론에서 이야기한 문제를 구체적으로 다루기 위한 비즈니스 모델을 이야기합니다. 때로는 제반 사항이나 레거시 코드가 왜 그렇게 되었는지도 이야기합니다.

개선 방향

스토리에서 언급한 레거시 코드의 문제를 해결하는 방안을 대략적으로 설명합니다.

질문과 답

주제의 기반 지식을 다루거나 문제를 해결하는 과정에서 생겨난 또 다른 질문에 대해 간단하게 알아봅니다.

레거시 코드

스토리에서 언급한 문제를 보여주고 이를 설명합니다. 클래스, 메서드, 변수 등 많은 내용이 언급되지만, 주요 목적은 레거시 코드를 이해하는 것이므로 이를 중심으로 설명합니다.

레거시 코드의 개선 과정

문제가 된 레거시 코드를 개선하는 과정을 설명합니다. 복잡하거나 예외사항이 많이 발생할 것으로 판단되는 개선 과정에서는 단위 테스트나 TDD를 활용합니다.

개선된 레거시 코드

개선된 코드를 다시 한 번 정리합니다. 이를 통해 레거시 코드와 그 개선 과정을 머릿속에서 정리할 수 있게 도와줍니다.

요약 및 정리

레거시 코드의 개선 포인트와 개선의 이점에 대해서 이야기합니다. 그리고 "생각의 흐름"에서는 레거시 코드의 문제를 찾아내기 위한 인사이트(쉽게 구린내 맡는 방법)를 제공합니다.

총 8단계로 쓰여졌기 때문에 글을 읽기도 전에 뭔가 복잡하다고 느끼실 수도 있습니다. 복잡하게 생각하지 마세요. 우리가 흔히 알고 있는 글의 전개 방식인 "서론, 본론, 결론"과 같다고 보시면 됩니다. 그리고 이 사실을 잊고 읽어도 됩니다. 곧 반복되는 패턴에 적응할 것입니다.

저희가 고민 끝에 생각한 방법이 레거시 코드와 스토리를 별다른 고민 없이 손쉽게 읽는 데 도움이 되었으면 합니다. 그리고 이 책이 레거시 코드의 개선에 도움이 되면 좋겠습니다.

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 노력하였습니다.

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.

차례

final static 필드를 모아 놓아서 뚱뚱해진 클래스 개선하기

상수를 쉽게 관리하려고 만든 클래스가 오히려 힘들고 피곤하게 만듭니다.

유일한 값을 만들어 사용할 때 보통 상수를 활용합니다. 이렇게 만든 상수들은 개발 초기에는 각기 다른 클래스에 분산되어 있다가, 개발이 진행될수록 하나의 클래스에서 관리되도록 만듭니다. 이런 방법은 상수가 적은 구조에서는 유용하지만 기능이 점차 늘어나면 상수도 많아지면서 문제가 됩니다.

상수를 이용할 때 상수의 값이 무엇인지 관리 클래스에서 어렵게 찾아야 하고, 같은 값이지만 용도가 다르게 정의된 상수는 무엇인지 확인하기 힘들어집니다. 또한, 특정 기능을 담당하는 모듈을 만들어도 상수를 관리하는 클래스의 종속적 구조에서 벗어나지 못하게 됩니다.

결국, 기능이나 로직이 확장될수록 의도하지 않은 곳에서 상수를 사용하거나, 통합 관리 목적과는 다르게 진행됩니다. 상수의 통합 관리에 초점을 맞추려다 보니 상수

클래스에 종속되어 의존성이 높아지고, 분류하기 어려운 상수 때문에 유지보수가 어려운 구조가 됩니다.

스토리

호텔 서비스 시스템에서 잘못된 상수 관리는 어떤 결과로 나타날까요?

호텔 서비스의 체크인-아웃 모듈에 문제가 생겼습니다. 체크아웃 시 예치금 반환과 서비스 요금 결제가 동시에 이루어져야 하는데 이 부분에서 오류가 발생하였습니다. 오류의 원인을 찾아보니 다른 상수를 사용한 것으로 밝혀졌습니다. 로직을 수정하면서 상수를 관리하는 클래스의 비슷한 값을 가진 상수를 사용했는데, 상수 값이 변경되어 의도하지 않은 값이 사용되었다고 합니다. 주요 원인은 상수를 모아 놓은 클래스에 상수가 많아서 이들 값을 구별하기 힘들기 때문입니다. 따라서 이를 개선하려고 합니다.

개선 방향

상수 클래스를 분류하고 책임 클래스로 이동하자!

상수를 관리하는 클래스에는 다양한 로직에서 사용되는 상수들이 저장되어 있습니다. 예치금 반환과 서비스 요금 결제 기능을 추가하는 과정에서 다른 로직의 상수를 사용했는데, 용도는 다르지만 의도한 값과 같은 상수를 잘못 사용했습니다.

그런데 사용하는 상수가 포함된 로직을 수정하면서 상수 역시 변하게 되었고, 스토리에서 이야기한 오류가 발생한 것입니다. 이런 오류의 근본적 원인은 원하는 상수를 찾기 어려울 만큼 많은 상수가 모여 있기 때문입니다. 따라서 의도된 상수를 사용하도록 상수 관리 클래스에 정의된 상수들을 목적에 맞게 분류하고 필요에 따라서는 상수를 해당 객체로 이동시켜야 합니다.

질문과 답

Q. 상수를 관리하는 객체를 만드는 것이 좋지 않은 습관이라는 건가요?

A. 아니요. 상수 관리 객체는 필요합니다. 서비스의 데이터베이스 접속 정보와 같은 자료는 관리 객체에서 공통으로 접근하여 사용하는 것이 좋습니다. **그러나 관리하는 상수의 범위가 일정하지 않으면 유지보수성이 모호해집니다.** 특정 로직에 상수가 필요할 때마다 시스템 상수를 관리하는 곳에 상수를 추가하게 됩니다. 이렇게 코드량이 늘어나면 상수 관리 객체는 시스템 상수만을 취급하는 곳이 아니라 서비스의 부분에서만 사용하는 상수들과 섞이게 되는 거죠.

Q. 왜 상수를 사용 목적에 맞는 객체로 이동해야 하나요?

A. 상수를 이동하는 가장 큰 이유는 의존성이에요. 모듈이 별도의 애플리케이션으로 동작하는 경우를 생각해 보죠. 이때 모듈에서 사용하는 상수가 속한 클래스도 함께 이동됩니다. 그러면 의도하지 않는 상수들도 딸려오게 되죠. 이러한 의

존성은 구조적으로도 좋지 않아요. 그러니 사용 목적에 맞는 객체와 가깝게 상수를 이동하는 것이 좋습니다.

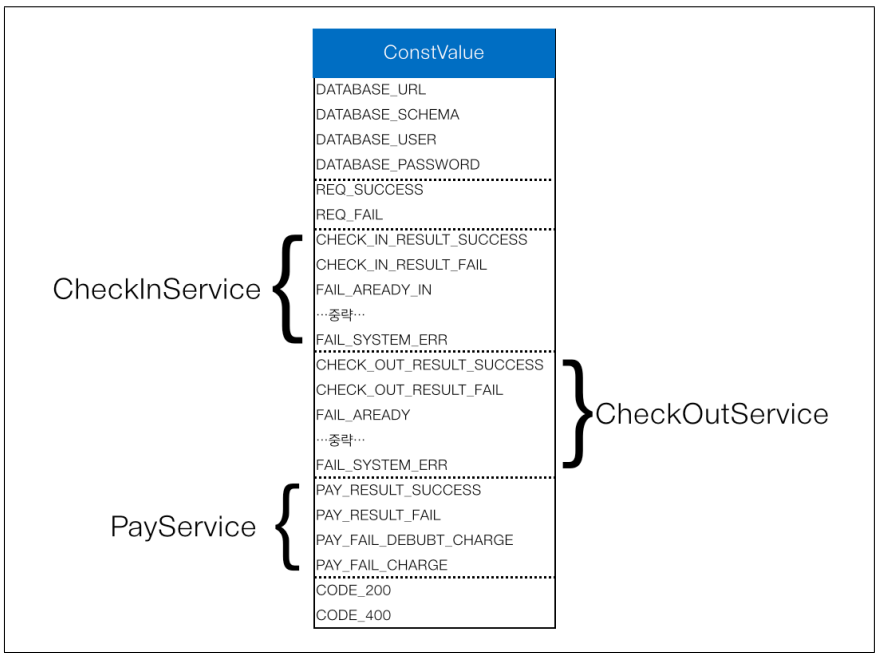
Q. 그러면 어떤 기준으로 상수를 이동해야 하나요?

A. 상수를 이용하는 객체를 기준으로 생각해 봅시다. 상수를 반환하기 위해 사용하는 객체가 있고 반환받아 사용하는 객체가 있어요. 제 경우에는 대개 반환받아 사용하는 객체가 좀 더 범용성이 있는 객체였어요. 반환하기 위해 사용하는 로직은 여러 곳에 분산되었지만, 반환받아 사용하는 곳은 한곳인 경우가 많았어요. 그래서 주로 상수를 사용하는 객체에 정의해서 사용했어요. 여러분도 다음 기준으로 생각해 보세요.

1. 상수를 사용하는 곳이 어느 패키지에 집중되어 있는지 살펴봅시다.
2. 상수를 사용하는 곳이 집중된 객체 중 제일 포괄적인 성격을 지닌 곳을 살펴봅시다.

레거시 코드

[그림 1] final static 필드를 모아 놓아서 뚱뚱해진 클래스



ConstValue 클래스

다른 로직에 필요한 상수들을 담고 있는 클래스로, 체크인-아웃 요청과 관련된 값들과 시스템 상수가 정의되어 있습니다. 이 상수 집합의 문제는 **시스템 상수와 일반 로직에서 사용하는 상수가 섞여 있다는 것**입니다. 일반 로직에서 사용할 상수를 추가할 때마다 이 클래스에 추가될 가능성이 높습니다. 유지보수를 할 때마다 이 객체는 더 많은 정보를 담게 되고, 어느 순간 내가 원하는 상수를 찾을 수 없는 상황이 올 수 있습니다. 즉, 잠재적인 위험 요소를 가지고 있습니다. 시스템 상수와 일반 로직 상수를 구분하고 분리해야 합니다.

```
public class ConstValues {
    public static final String DATABASE_URL = ‘example.co.kr’;
    public static final String DATABASE_SCHEMA = ‘example’;
    public static final String DATABASE_USER = ‘example1’;
    public static final String DATABASE_PASSWORD = ‘12341234’;

    public static final int REQ_SUCCESS = 10;
    public static final int REQ_FAIL = 19;

    public static final int CHECK_IN_RESULT_SUCCESS = 30;
    public static final int CHECK_IN_RESULT_FAIL = 39;

    public static final int CHECK_IN_FAIL_ALREADY_IN = 301;
    public static final int CHECK_IN_FAIL_NOT_RESERVED = 302;
    public static final int CHECK_IN_FAIL_DEBT_CHARGE = 303;
    public static final int CHECK_IN_FAIL_SYSTEM_ERR = 399;

    public static final int CHECK_OUT_RESULT_SUCCESS = 40;
    public static final int CHECK_OUT_RESULT_FAIL = 49;

    public static final int CHECK_OUT_FAIL_ALREADY_OUT = 401;
    public static final int CHECK_OUT_FAIL_NOT_CUSTOMER = 402;
    public static final int CHECK_OUT_FAIL_DEBT_RETURN = 403;
    public static final int CHECK_OUT_FAIL_CREDIT_CHARGE = 404;
    public static final int CHECK_OUT_FAIL_SYSTEM_ERR = 499;

    public static final int PAY_RESULT_SUCCESS = 50;
    public static final int PAY_RESULT_FAIL = 51;

    public static final int PAY_FAIL_DEUBT_CHARGE = 501;
    public static final int PAY_FAIL_DEUBT_RETURN = 502;
    public static final int PAY_FAIL_PAY_CHARGE = 503;

    public static final int RESULT_CODE_200 = 200;
    public static final int RESULT_CODE_400 = 400;
```

```
    public static final int RESULT_CODE_404 = 404;
    public static final int RESULT_CODE_500 = 500;
}
```

Result 클래스

호텔 체크인-아웃 때 처리되는 로직들의 결과값을 가진 클래스입니다.

```
public class Result {
    private int result;
    private int resultMsgType;

    public int getResult() {
        return result;
    }

    public void setResult(int result) {
        this.result = result;
    }

    public int getResultMsgType() {
        return resultMsgType;
    }

    public void setResultMsgType(int resultMsgType) {
        this.resultMsgType = resultMsgType;
    }
}
```

CheckInService 클래스

호텔 체크인 때 처리할 로직을 가진 클래스입니다. 처리의 성공과 실패에 따라 결과 객체를 반환합니다. 결과값은 `ConstValue` 객체에 정의된 값을 참조하고

ConstValues 클래스와 의존 관계입니다.

```
public class CheckInService {
    public Result checkIn(String name, String roomNo) {
        boolean isSuccess;
        // 중략

        Result result = new Result();
        if (isSuccess) {
            result.setResult(ConstValues.CHECK_IN_RESULT_SUCCESS);
        } else {
            result.setResult(ConstValues.CHECK_IN_RESULT_FAIL);
            result.setResultMsgType(ConstValues.CHECK_IN_FAIL_DEBT_CHARGE);
        }
        return result;
    }
}
```

CheckOutService 클래스

호텔 체크아웃 때 처리할 로직을 가진 클래스입니다. 처리의 성공과 실패에 따라 결과 객체를 반환합니다. 결과값은 ConstValue 객체에 정의된 값을 참조하고 ConstValues 클래스와 의존 관계입니다.

```
public class CheckOutService {
    public Result checkOut(String name, String roomNo) {
        boolean isSuccess;
        // 중략

        Result result = new Result();
        if (isSuccess) {
            result.setResult(ConstValues.CHECK_OUT_RESULT_SUCCESS);
```

```
        } else {
            result.setResult(ConstValues.CHECK_OUT_RESULT_FAIL);
            result.setResultMsgType(ConstValues.CHECK_OUT_FAIL_DEBT_RETURN);
        }
        return result;
    }
}
```

PayService 클래스

요금 결제 때 처리할 로직을 가진 클래스입니다. 처리의 성공과 실패에 따라 결과 객체를 반환합니다. 결과값은 `ConstValue` 객체에 정의된 값을 참조하고 `ConstValues` 클래스와 의존 관계입니다.

```
public class PayService {
    public Result payRoomCharge(String name, String roomNo, Date in, Date out) {
        boolean isSuccess;
        // 중략

        Result result = new Result();
        if (isSuccess) {
            result.setResult(ConstValues.PAY_RESULT_SUCCESS);
        } else {
            result.setResult(ConstValues.PAY_RESULT_FAIL);
            result.setResultMsgType(ConstValues.PAY_FAIL_DEUBT_CHARGE);
        }
        return result;
    }
}
```

레거시 코드의 개선 과정

ConstValue의 상수를 목적에 맞게 내부 클래스로 묶기

상수를 관리하는 클래스는 각 상수의 사용 목적에 따라 정의되었습니다. 이런 형태는 다른 목적의 상수들과 쉽게 섞입니다. 결국, 추가로 생기는 상수에 대해 추가할 위치를 정하기가 어려워지고 정의된 상수를 찾을 때에도 어려움을 겪게 됩니다.

따라서 각 상수를 목적에 맞게 내부 클래스로 범위를 다시 지정함으로써 상수의 추가, 수정, 검색이 쉽도록 만들었습니다. 일부 상수는 Enum 클래스로 적용하는 방법도 있습니다.

[ConstValue 클래스]

```
public class ConstValues {
    public class Database {
        public static final String URL = ‘example.co.kr’;
        public static final String SCHEMA = ‘example’;
        public static final String USER = ‘example1’;
        public static final String PASSWORD = ‘12341234’;
    }

    public static final int REQ_SUCCESS = 10;
    public static final int REQ_FAIL = 19;

    public class CheckIn {
        public static final int RESULT_SUCCESS = 30;
        public static final int RESULT_FAIL = 39;

        // 실패 원인
        public class Fail {
            public static final int ALREADY_IN = 301;
            public static final int NOT_RESERVED = 302;
            public static final int DEBT_CHARGE = 303;
            public static final int SYSTEM_ERR = 399;
```

```
        }
    }

    public class CheckOut {
        public static final int RESULT_SUCCESS = 40;
        public static final int RESULT_FAIL = 49;

        // 실패 원인
        public class Fail {
            public static final int ALREADY_OUT = 401;
            public static final int NOT_CUSTOMER = 402;
            public static final int DEBT_RETURN = 403;
            public static final int CREDIT_CHARGE = 404;
            public static final int SYSTEM_ERR = 499;
        }
    }

    public class Pay {
        public static final int RESULT_SUCCESS = 50;
        public static final int RESULT_FAIL = 51;

        // 실패 원인
        public class Fail {
            public static final int DEUBT_CHARGE = 501;
            public static final int DEUBT_RETURN = 502;
            public static final int CHARGE = 503;
        }
    }

    public class HttpStatus {
        public static final int CODE_200 = 200;
        public static final int CODE_400 = 400;
        public static final int CODE_404 = 404;
        public static final int CODE_500 = 500;
    }
}
```

ConstValue 내부 클래스의 이동

앞에서 ConstValue 내부 클래스를 목적에 맞는 클래스로 이동하였습니다. 이는 내부 클래스가 외부 클래스의 상수들과 연관이 없고, 내부 클래스의 사용 범위가 매우 한정적이기 때문입니다. 이런 경우에는 내부 클래스를 사용 목적에 맞는 곳과 가깝게 배치하여 검색이나 수정이 쉬워지도록 합니다. ConstValue, CheckIn, CheckOut, Pay 안의 상수들을 CheckInServer, CheckOutService, PayService로 이동하고 내부 클래스를 삭제합니다.

[ConstValue 클래스]

```
public class ConstValues {
     public static final int REQ_SUCCESS = 10;
     public static final int REQ_FAIL = 19;

     public class ResultCode {
         public static final int CODE_200 = 200;
         public static final int CODE_400 = 400;
         public static final int CODE_404 = 404;
         public static final int CODE_500 = 500;
     }
}

CheckInService Class
public class CheckInService {
     public static final int RESULT_SUCCESS = 30;
     public static final int RESULT_FAIL = 39;
     public static final int FAIL_ALREADY_IN = 301;
     public static final int FAIL_NOT_RESERVED = 302;
     public static final int FAIL_DEBT_CHARGE = 303;
     public static final int FAIL_SYSTEM_ERR = 399;

     public Result checkIn(String name, String roomNo) {
```

```
        boolean isSuccess;
        // 중략

        Result result = new Result();
        if (isSuccess) {
            result.setResult(ConstValues.CHECK_IN_RESULT_SUCCESS);
        } else {
            result.setResult(ConstValues.CHECK_IN_RESULT_FAIL);
            result.setResultMsgType(ConstValues.CHECK_IN_FAIL_DEBT_CHARGE);
        }
        return result;
    }
}
```

[CheckOutService 클래스]

```
public class CheckOutService {
    public static final int RESULT_SUCCESS = 40;
    public static final int RESULT_FAIL = 49;
    public static final int FAIL_ALREADY_OUT = 401;
    public static final int FAIL_NOT_CUSTOMER = 402;
    public static final int FAIL_DEBT_RETURN = 403;
    public static final int FAIL_CREDIT_CHARGE = 404;
    public static final int FAIL_SYSTEM_ERR = 499;

    public Result checkOut(String name, String roomNo) {
        boolean isSuccess;
        // 중략

        Result result = new Result();
        if (isSuccess) {
            result.setResult(ConstValues.CHECK_OUT_RESULT_SUCCESS);
        } else {
```

```
            result.setResult(ConstValues.CHECK_OUT_RESULT_FAIL);
            result.setResultMsgType(ConstValues.CHECK_OUT_FAIL_DEBT_RETURN);
        }
        return result;
    }
}
```

[PayService 클래스]

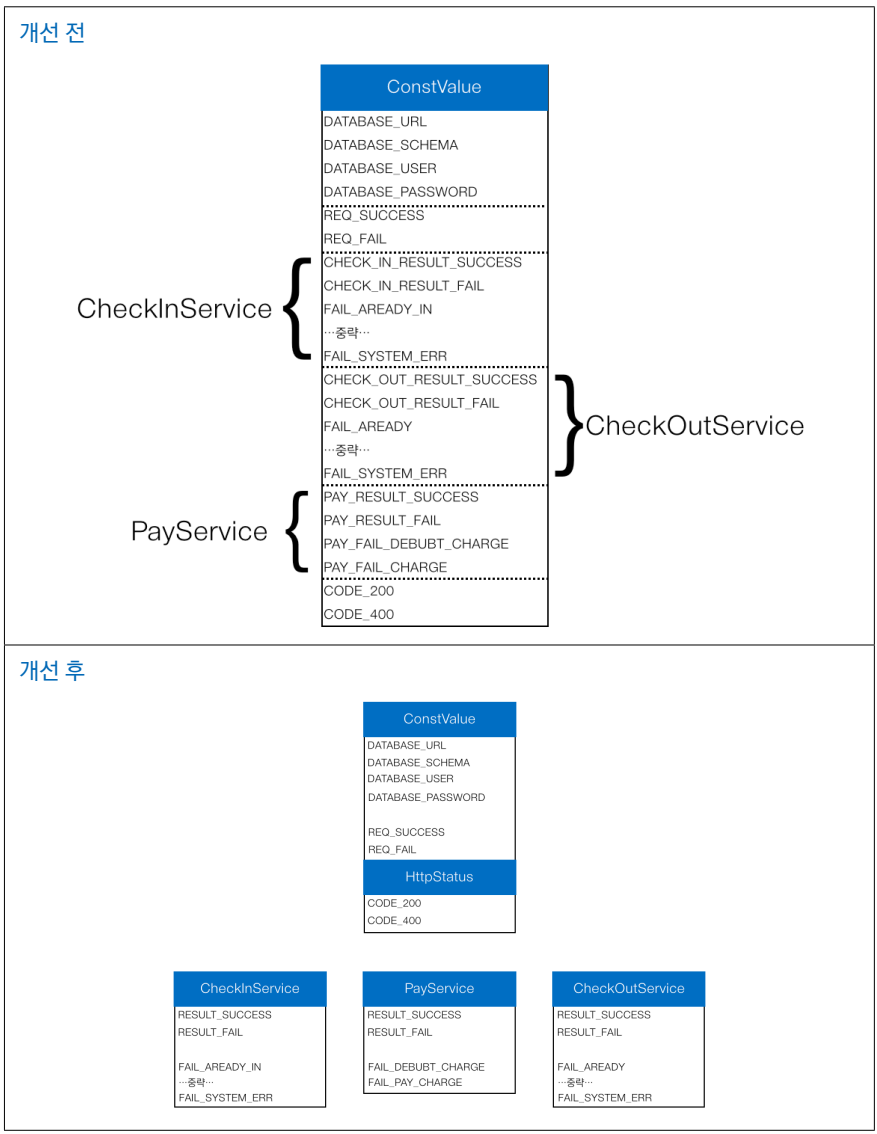
```
public class PayService {
    public static final int RESULT_SUCCESS = 50;
    public static final int RESULT_FAIL = 51;
    public static final int FAIL_DEUBT_CHARGE = 501;
    public static final int FAIL_DEUBT_RETURN = 502;
    public static final int FAIL_CHARGE = 503;
    // 중략

    public Result payRoomCharge(String name, String roomNo, Date in, Date out) {
        boolean isSuccess;
        // 중략

        Result result = new Result();
        if (isSuccess) {
            result.setResult(ConstValues.PAY_RESULT_SUCCESS);
        } else {
            result.setResult(ConstValues.PAY_RESULT_FAIL);
            result.setResultMsgType(ConstValues.PAY_FAIL_DEUBT_CHARGE);
        }
        return result;
    }
}
```

개선된 레거시 코드

[그림 2] 레거시 코드의 개선 전과 후 비교



ConstValue 클래스

시스템에 연관된 상수와 서비스 전역에서 사용할 공통 상수들만 남아 있습니다. `ConstValue` 클래스에서 관리되는 상수 중 특정 로직에 국한된 상수들을 사용 목적과 가까운 클래스로 이동하였습니다. 로직들은 이제 `ConstValue` 클래스를 참조하지 않습니다.

CheckInService / CheckOutService / PayService 클래스

`ConstValue` 클래스에 있는 상수 중 일부가 이동되어 추가되었습니다. 각 로직에서만 사용되던 상수들은 그에 맞는 `Service` 클래스들로 이동하여 `ConstValue` 클래스에 대한 의존성이 낮아지거나 없어졌습니다. 또한, 각 로직에서 필요한 상수는 로직과 관련된 클래스에서 찾고 추가할 수 있어서 유지보수성도 높아졌습니다.

요약 및 정리

상수를 관리하는 클래스를 만드는 것은 설계 초기에 매우 유용한 방법입니다. 하지만 상수가 많아지고 상수들의 사용 범위가 모호해지면 관리를 위한 상수 집합이 원하는 코드를 찾는 순간부터 독이 됩니다. 수많은 코드 중에서 구별해 내야 하는 어려움이 있기 때문이죠. 따라서 초기의 좋은 모습을 유지하기 위해서는 무조건적인 사용을 막아야 하고, 사용하려면 명확히 구분을 지어야 합니다. 특히, 시스템 관련 상수는 로직 상수와 섞이지 않도록 해야 하고, 로직 관련 상수는 되도록 로직과 가까운 곳에 있어야 합니다.

비대해진 상수 클래스를 개선하기 위한 생각의 흐름

1. 상수의 사용 범위가 같은지 확인합니다.
2. 상수의 사용 범위가 같은 상수들끼리 관리되는지 확인합니다.
3. 상수가 이를 사용하는 객체와 가까운 곳에 있는지 확인합니다.
4. 상수를 사용하는 곳이 어느 패키지에 집중되어 있는지를 확인합니다.
5. 상수를 사용하는 곳이 집중된 객체 중에서 제일 포괄적인 곳을 확인합니다.