프로그래밍 방식의 변천 과정

절차 지향에서 프레임워크 까지...

패러다임(Paradigm)

- 한 시대의 지식인들의 합의로 형성된 지식의 집합체
- 즉, 전문가들의 합의로 생성된 지식의 구조(체계)로서 동 시대 사람들의 견해나 사고에 영향을 준다.
- 그러나 어떤 집단이 갖고 있는 생각의 틀(방식)만을 뜻하는 것은 아니며 개개인이 주어진 조건에서 생각하는 방식 또한 패러다임이라고 말한다. (위키피디아 인용)

패러다임이 정립되기, 이전 이야기...

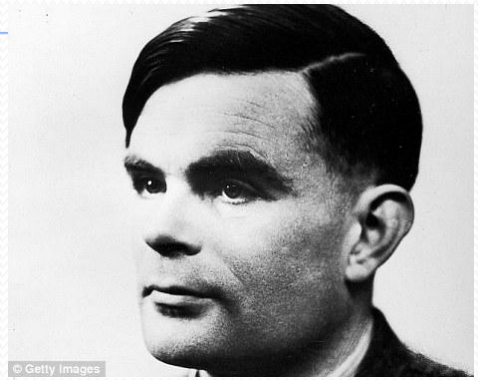
- 선구적인 프로그래머들은 거의 다 수학자 혹은 과학자!

- 서브루틴(subroutine)
- 루프(loop)
- 점프(jump)

그리고, "if - then" 구문까지...

소프트웨어의 기본 개념을 창안.

Aida, 러브레이스 백작 부인



- 수학자, 암호학의 천재
- 인공지능의 아버지
- 튜링 테스트 창안

앨런 튜링

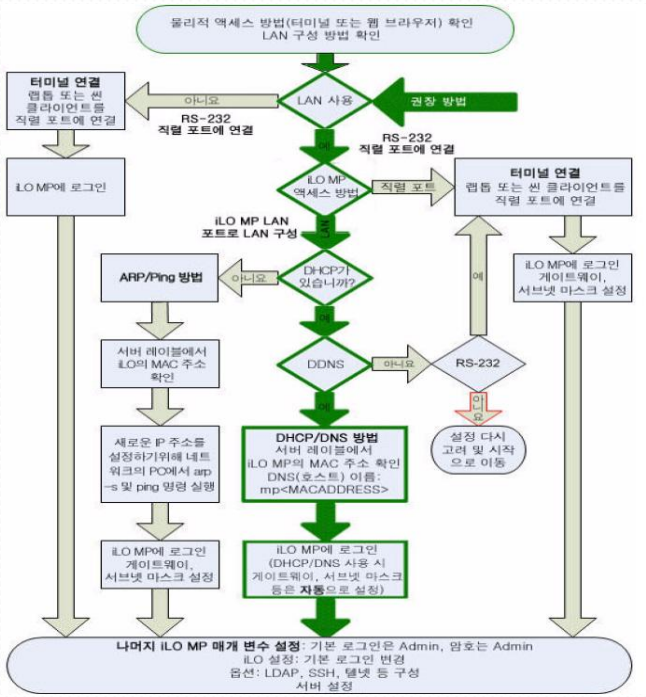
‘구조적 프로그래밍’ 패러다임

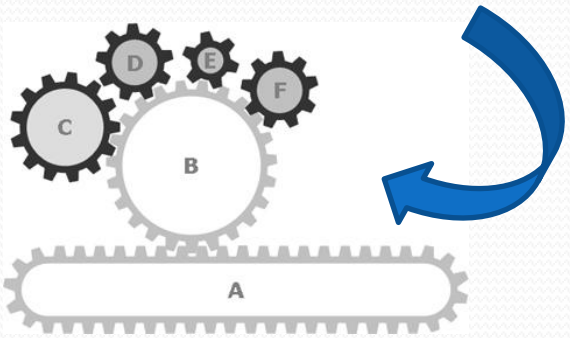
- 최초의 프로그래머들은 소프트웨어를 선형적(linear)으로 서술했습니다.
  그리고, 컴퓨터는 입력된 명령을 순서대로 실행합니다.

- 이런 방식은 사람이 논리적 사고가 필요한 문제를 ‘절차적’으로 풀기 때문에 지극히 자연스럽습니다.

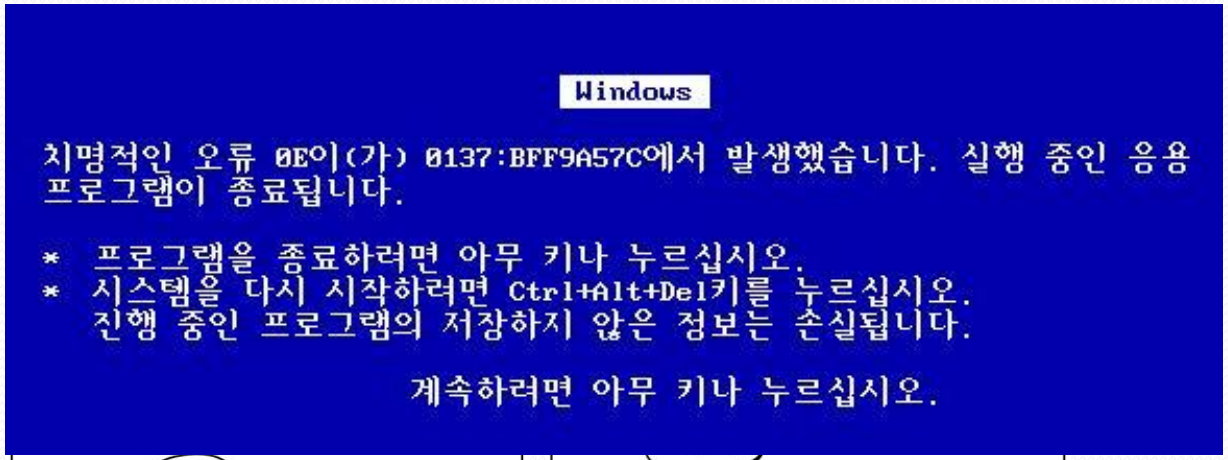
- 순서도(flow chart)와 구조적 프로그래밍(structured programming) 기법이 제안되고, 절차적 사고에 기반한 구조적 프로그래밍 패러다임이 정착됩니다.

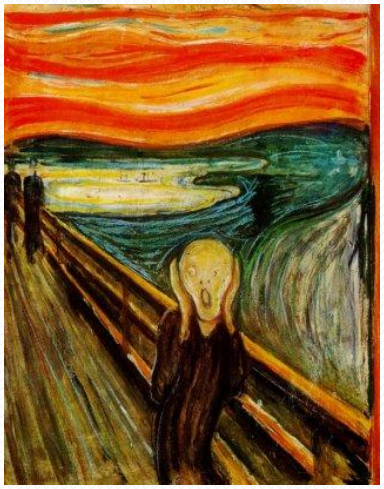
- 복잡한 문제를 하향식 구조(top-down)로 설계한 후 코딩하는 것이 일반화 되었습니다. (1960년 말)

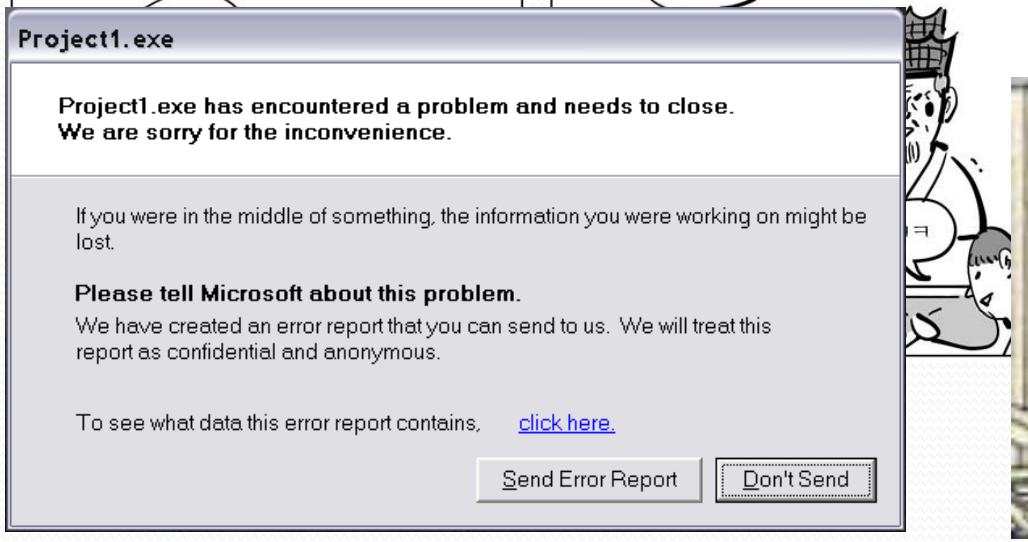




그런데, 일어나야 하지 말아야 할 일들이...







이런 사태를 '소프트웨어 위기'라고 합니다.

프로그래머들이
멍청해서...? 위기가 온 걸까요?

김현남 님의 카툰, [게임 회사 이야기] 중에서...

왜? 소프트웨어는 항상 '오동작' 하는가?

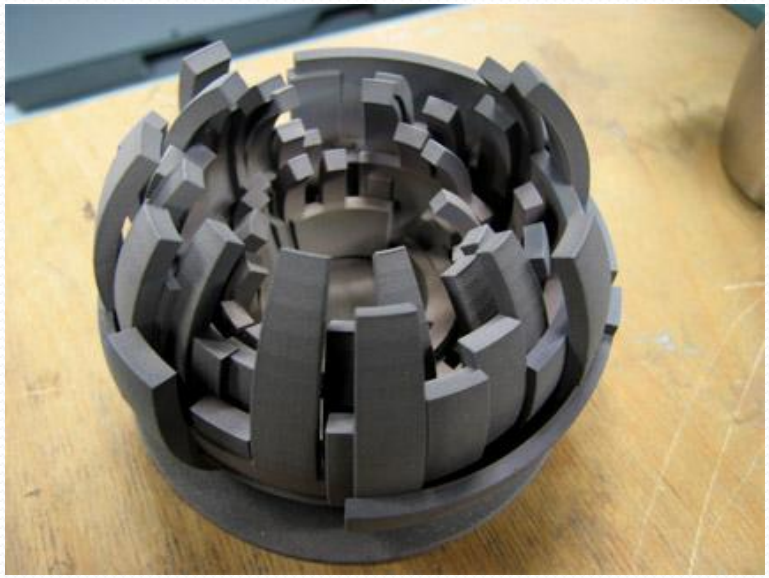
개발자가 의도한 것도 아니고, 테스트할 때는 문제가 없었다.

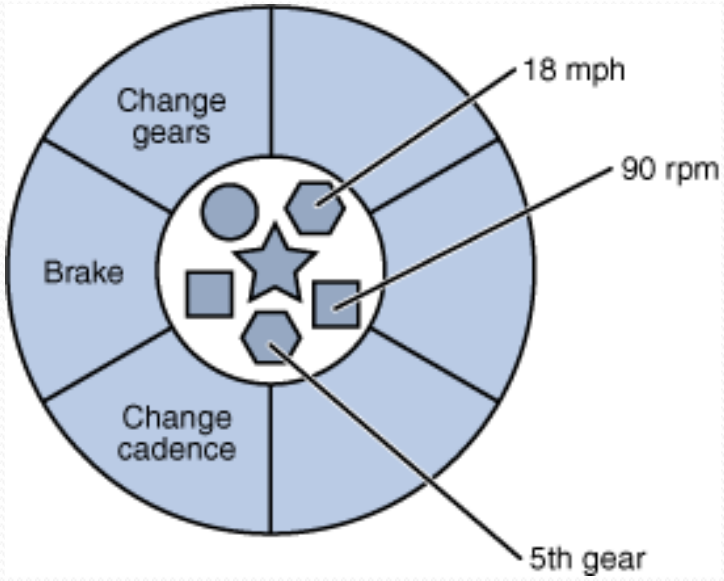
테스트 할 때는 정상적인 데이터를 이용해서 확인합니다.

하지만, 실제 운영할 때는 항상 정상 데이터가 들어온다고, 보장 못합니다.

(원래 계획대로 된다면, 야근은 줄어들 겁니다.)

개발자도, 코드도 죄가 없다? 그러면…





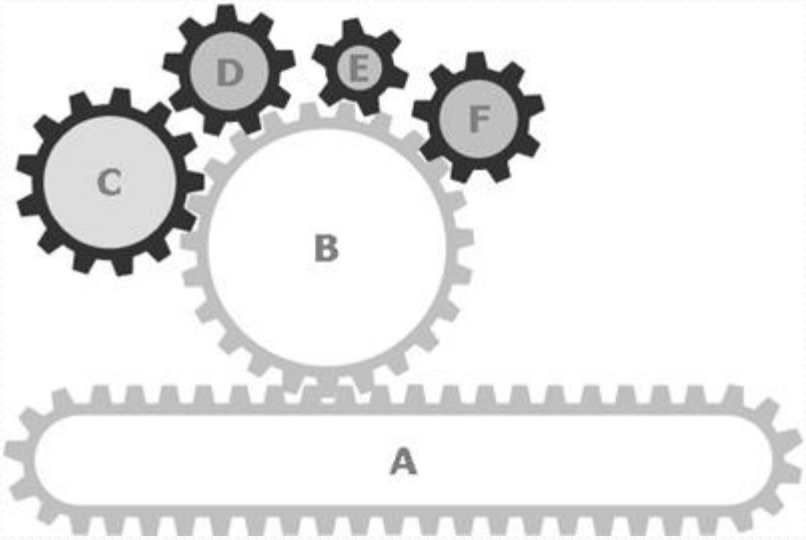
문제는 '데이터'인 것입니다. (Bug = caused by **illegal data**, may be…)

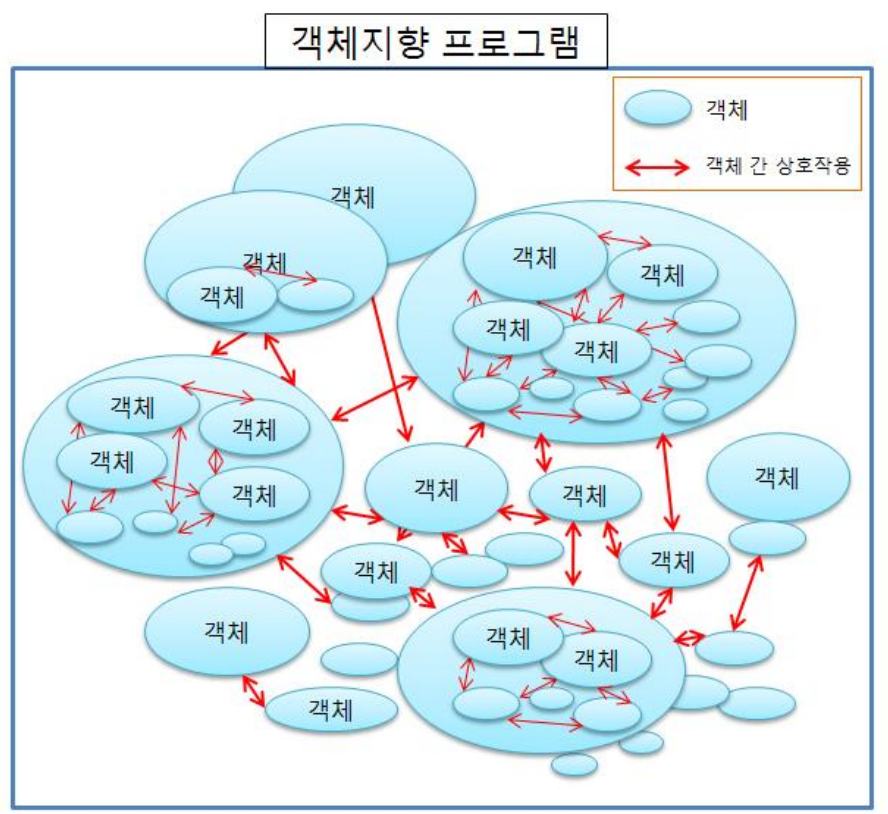
데이터가 '적합한 범위'를 벗어나지 못하게 해야 하는 것이죠.

절차(procedure)와 명령(code) 뿐만 아니라, **데이터를 중시해야 합니다.**

데이터를 보호하고, 수정하는 코드를 데이터와 합체시킵니다.

함수 단위가 아니라, 객체를 조립하는 방식





절차지향 프로그래밍에서...

객체지향 프로그래밍으로...

참고 : http://bopboy.tistory.com/725

소프트웨어 생산성 이슈

- 코딩은 가내수공업...

- 소프트웨어도 부품 조립 방식처럼 간편하게 만들 수 없을까?

그래서, 살림 좀 나아지고 계십니까?

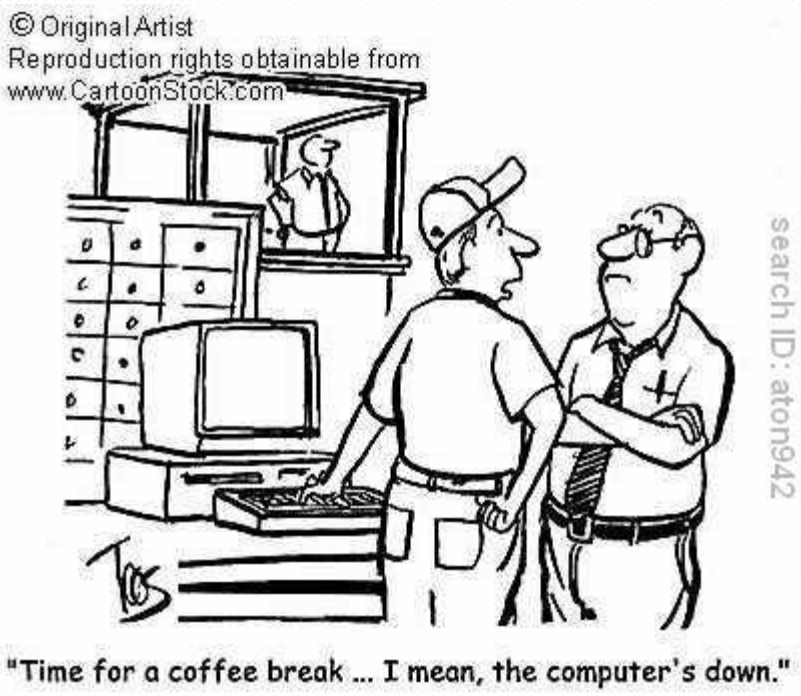
전자/기계/건설은 어떻게 수공업을 벗어난 걸까?

- 가내 수공업에서 조립 생산으로 이전하게 된 원리

  '조립 생산(assembly line)'을 참고!

- 전체는 부분의 합보다 크다.

  The whole is more than the sum of parts.
- (아리스토텔레스)



소프트웨어를 부품화 하기 위한 기법들

- 제품(whole product)을 조립하려면,
  블랙박스(black box) 형태의 부품(parts)이 필요.
- 객체(object)라는 부품을 제작하고 재사용하기 위한
  다양한 기법들이 소개되었죠.
- 하지만… 아무렇게나 쓰면 안됩니다.

객체지향의 다양한 원리

- 클래스 선언 및 정의(class declaration & definition)
- 상속(inheritance)과 구성(aggregation)
- 데이터 캡슐화(data encapsulation)
- 다형성 (polymorphism)
- 정보 은폐 (information hiding)
- 디자인 패턴 (design patterns)

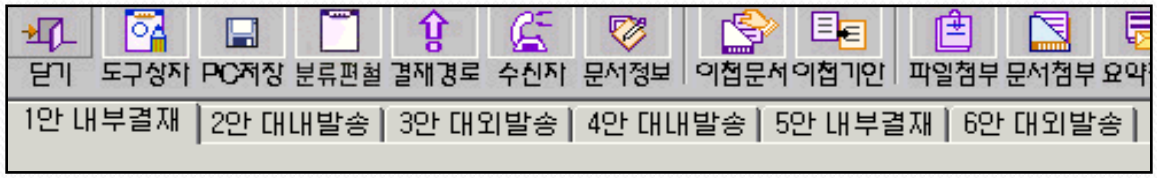
그런데 이런 기법들을 올바로 쓰고 계십니까?

상속(Inheritance)의 위험성

다양한 서식

행정자치부 사무 문서 규정에 따른 서식의 종류
내부결재, 대내발송, 대외발송, 일괄기안, 회계결의서 기안, 간이서식 등

동일한 툴바(기능)



유사한 워크 플로우

생산문서 흐름도

기안 → 결재 → 분류 및 편철 → 등록

접수문서 흐름도

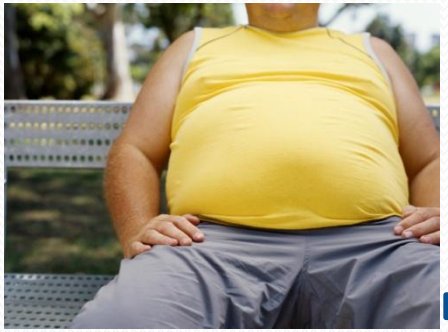
접수(기안) → 담당자 지정 → 선람(공람) → 분류 및 편철 → 등록

전자문서 시스템

설계 사례 !!!

- 다양한 서식을 지원해야 한다.
- 서식의 형태(form)은 유사하다.
- 서식 별로 거의 동일한 기능(menu)을 제공한다.
- 워크플로우(workflow)이 거의 일정하다.

뚱뚱한 상위 클래스와 허약한 하위 클래스



Fat parent class

- 공통 기능을 상위 클래스에서 선언.
- 전자문서 클래스에서 기능 다수 구현.
- 복잡도 증가, 상위 클래스 = 툴 박스
- Core 개발은 협업 불가능!
- 겉보기에는 재사용성과 응집도가 높아 보이지만, 유지보수가 아주 어렵다.

Thin child class

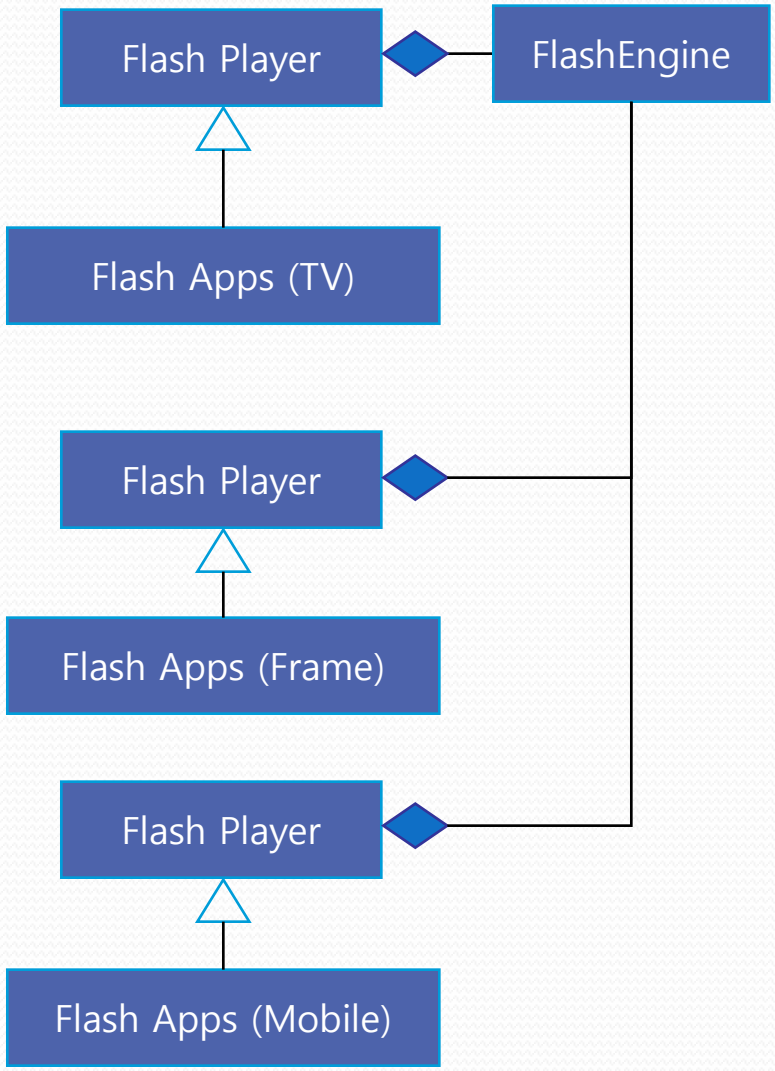
- Override 메소드로 인해 오히려 혼란 초래 (기본과 응용 메소드 호출 시 명칭이 동일함, 에러 로그 분석의 어려움)
- 상위 클래스에 대한 의존성 높음.
- 하위 클래스의 기능을 변경하고 싶은데 엉뚱하게 상위 클래스를 고쳐야 하는 경우가 발생하고, 상위 클래스의 기능ㅇ르 변경하면 다른 하위 클래스에서 버그가 발생하기도 한다. (무려 나비효과!)

깨지기 쉬운 상위 클래스 (Fragile base class)

Television

Digital Camera, Frame

Mobile devices



깨지는 상위 클래스

- **다양한 멀티미디어를 지원하는 플래시 플레이어 엔진 개발.**
- **FlashEngine + Linux + HAL (Hardware Abstraction Layer) 를 감싸는 공통 상위 클래스 작성**
- **이상과 현실은 달랐다.**
  - 하드웨어 마다 사운드 재생 방식이 다르고, 지원하는 동영상 디코더가 다르고, 비디오 버퍼 접근 방식이 다르더라.
- **상위 클래스 이름만 동일할 뿐, 코드는 일치하지 않는다. (재사용은 꿈이 되었고...)**
- **결론 : 합성(컴포지션)으로 설계 변경**

오늘 배운 나는 어제보다 낫고, 남들은 나보다 낫다.

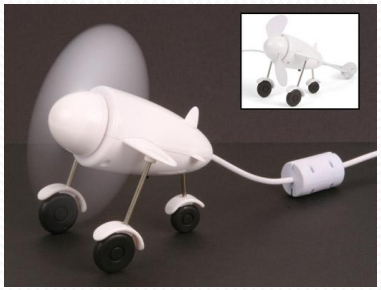
- 교과서에 나온다고, 무조건 좋은 게 아닙니다.
  합성은 상속에 비해 결합도(the degree of coupling)가 낮습니다.
  게다가, 부품의 교체도 가능합니다. **공부하세요!**

- '바퀴를 다시 발명 하지 말라' 는 명언이 있습니다.
  세상에는 이미 필요한 코드들이 오픈 소스와 프레임워크 형태로 나와 있습니다.

- 문제는 구슬이 서말이라도 꿰어야 보배라고,
  남들이 잘 만들어 둔 걸, 잘 가져다 쓰는 법도 배워야 합니다.

Spring Framework 의 효용성

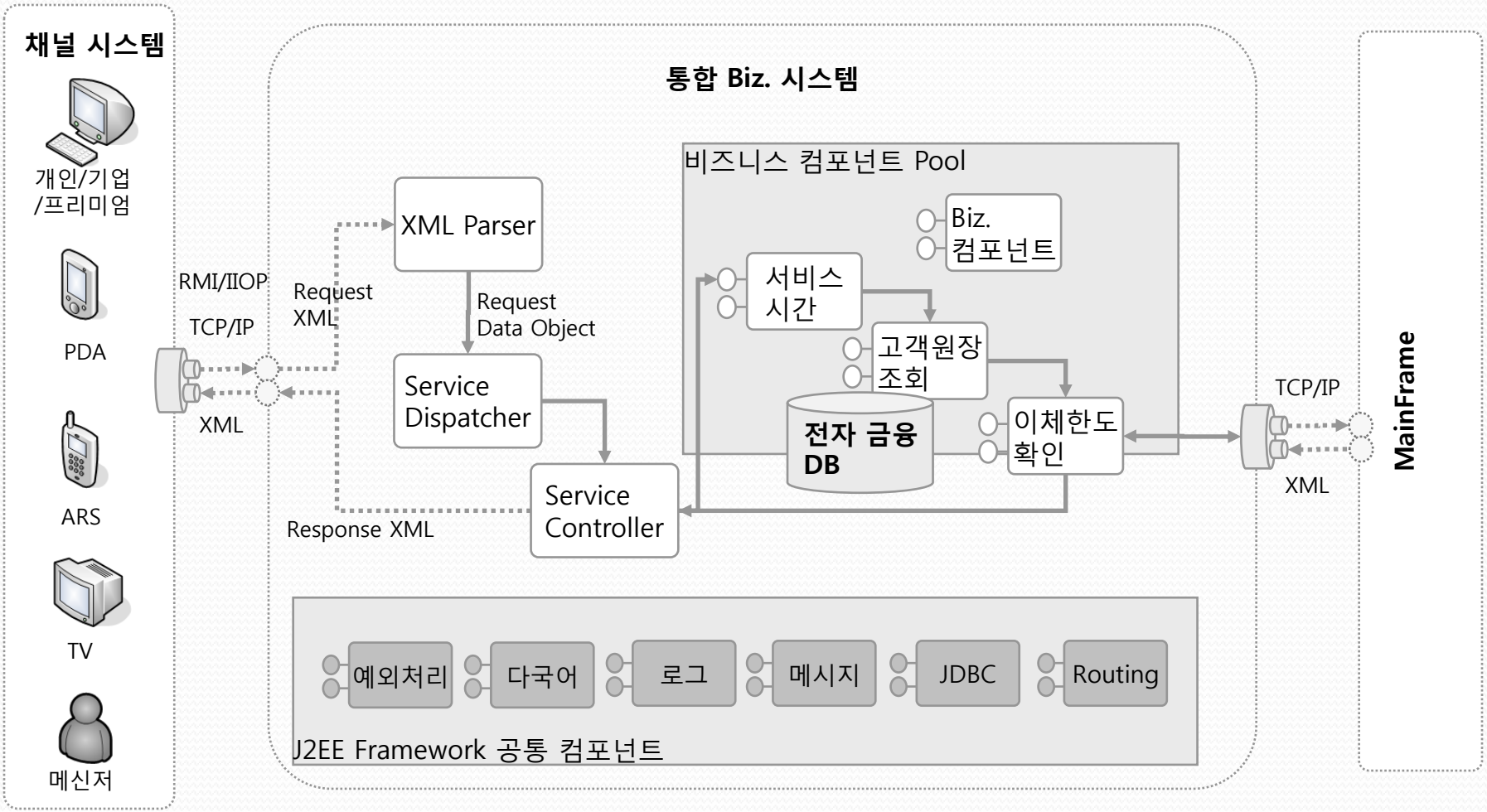
Spring Framework 는 소프트웨어 개발자를 위한 USB (포트)장치 입니다.



USB 포트만 있으면, 다양한 외부 장치를 연결할 수 있습니다.

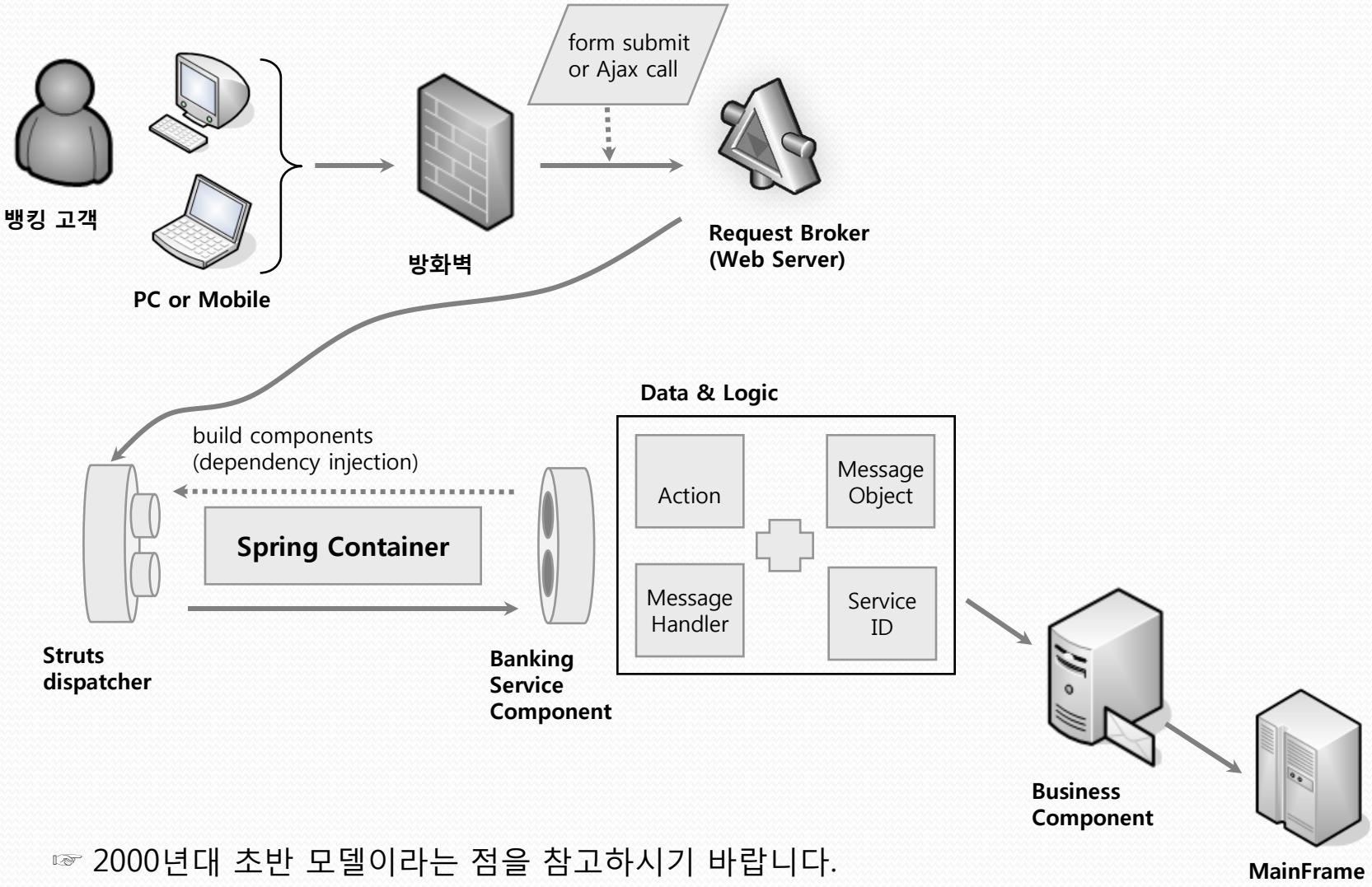
스프링 프레임워크를 '접착제(glue)' 프레임워크라고 부르기도 합니다.

금융 시스템 아키텍처 사례



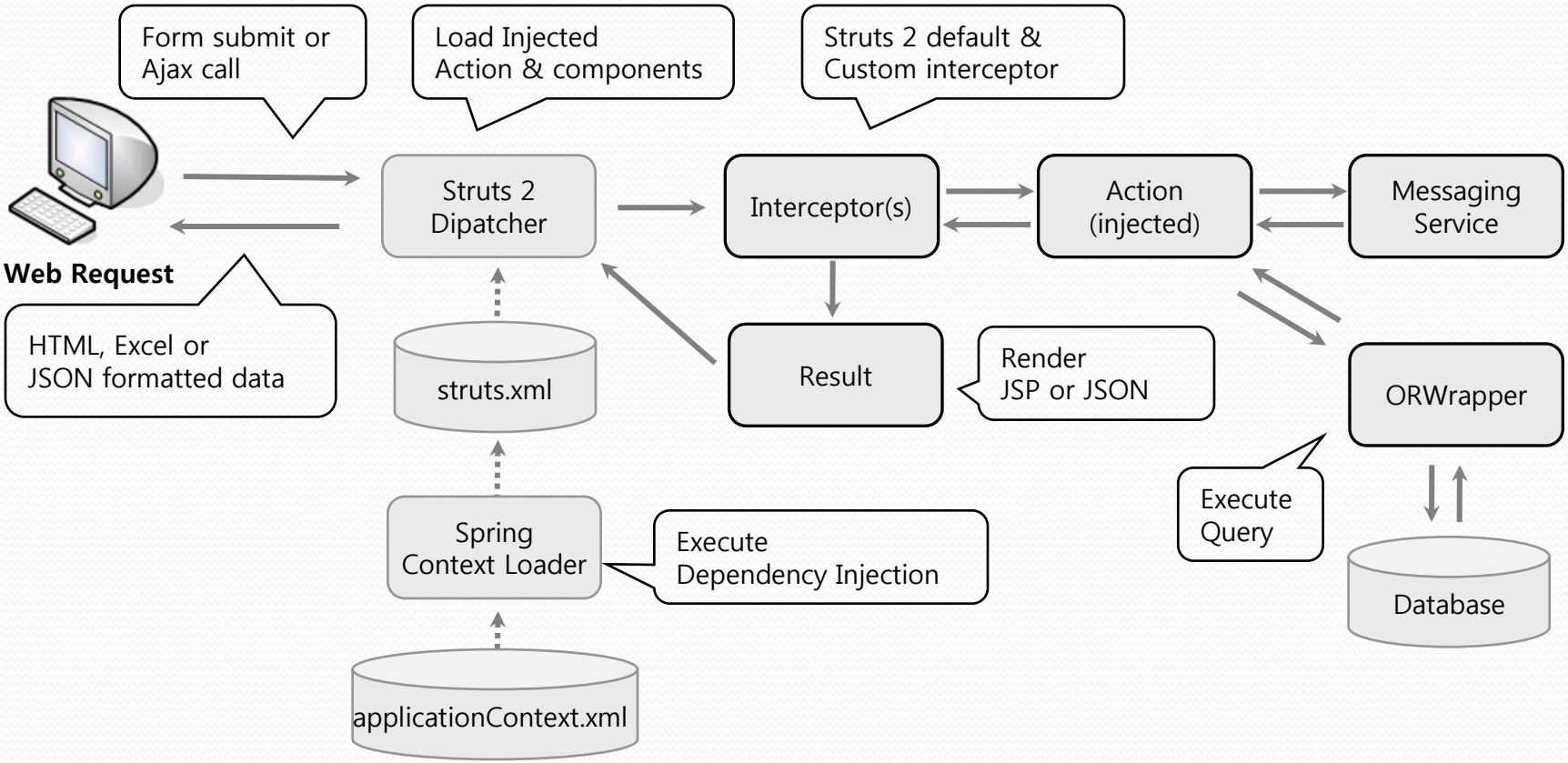
☞ 금융 시스템은 비즈니스 컴포넌트 종류가 정말 많습니다.

Service flow overview



☞ 2000년대 초반 모델이라는 점을 참고하시기 바랍니다.

Struts 2 & Spring 2 service flow



☞ Struts, Spring, ORM (iBatis) 등 다양한 프레임워크가 결합되고, 또 각 프레임워크는 비즈지스 컴퍼넌트를 조합해서 실행 시켜주는 역할을 담당합니다.

총 정리

- 소프트웨어 개발 패러다임은 '코드 중심'에서 '데이터 & 코드 결합'으로 이전되었습니다.
- 데이터와 코드가 결합된 객체가 발명됨으로써 '소프트웨어'를 조립할 수 있는 제대로 된 '부품'이 탄생하였습니다.
- 부품들을 수작업으로 혹은 강하게 결합하는 것은 여전히 수공업의 전통을 벗어나지 못하는 것입니다.
- 코드를 짜는 기법 뿐 아니라, 코드들을 조립하기 위한 방법을 공부해야 합니다. 또한 언제 어떤 기법을 사용하는지도 알아야 합니다.
- 오늘 배운 '나'는 어제의 '나'보다 낫고, 남들은 나보다 낫다.
  스스로 공부하고, 또 더 나은 건 남들의 경험을 배우는 겁니다.