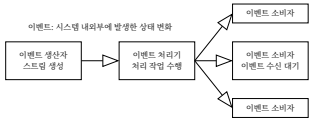
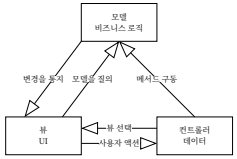
9. 아키텍처 스타일

- 스타일: 패턴보다 고수준, 개념적 추상적 양식. 모델-뷰-컨트롤러 분리
- 패턴: 문제 해결 위해 정의된 솔루션. UI-데이터보델-로직. 사이 복잡한 인터페이스 유지보스 힘

소프트웨어 아키텍처

- SW 구성요소, 요소 간 관계 표현 -한 아키텍처에 여러 스타일, 패턴 포함

전통 스타일

명칭	개요	장점	단점
클라-서버	클라-서버(자원) 분리	데이터 집중화, 보안, 모든 사용자 서비스 받	병목, 비용, 비강인성, 네트워크 의존
• 클라: 자원 사용-서버 접속 - 서버: 자원 관리-요청 처리 • 사용: 여러 지역, 공유자원(DB) 사용			
계층형(N-tier)	기능별 수직 분할 (계층에 연관 기능 배치)	추상, 캡슐, 높은 응집, 재사용, 인터페이스 유지(새 구현으로 대체 가능)	층 간 제한된 소통, 계층 명백히 구분 어려움
• 기능별 수직 분할(계층에 연관 기능 배치) • 사용처 ▸ 여러팀 계층 하나씩 담당할 때, 보안 여러 겹 강화 필요 할때 • 층 사이: 메시지 교환 - 각 층: 본인 역할 충실 • 1티어: 모놀리식			
이벤트 중심	이벤트 발생에 따라 시스템이 동작	시스템 변경 쉬움, 분선시스템에서 구현 b	이벤트 처리 시간 모름, 다른 구성요소 충
 • 이벤트 중심으로 구성, 이벤트 발생에 대응 • 사용처: 네트워크 상 분산된 컴포넌트 통합 • 비동기적(동기화 없이 이벤트 발생만 알림), 동기적(요청 받은 후 처리하고 응답 반환. 순차적)			
MVC(모델-뷰-컨트롤러)	UI, 로직, 데이터 분리 (보여주는것 처리하는 것 분리)	데이터 처리, display 각자 쉽게 변경	간단한 SW에 적용 시 더 복잡해짐
			
파이프 필터	필터사이 데이터 이동시키며 단계적 처리	단순, 재사용(변환 필터), 병렬	자원낭비, 의존성(선후행 작업 필수)
• 필터로 데이터 이동, 단계적으로 처리 - `ls | grep` • 출력이 다른 컴포넌트 입력 • 구성요소 > - 필터: 데이터 변환 수행 - 데이터 소스: 초기 입력 - 데이터 싱크: 아웃풋 저장 장소			
데이터 중심 아키텍처	공유 Data, 공유 Data 접근자(수정삭제추가)	중앙, 데이터 일관되게 관리 가능	공유D 문제는 전체 sys에 영향을 줌
P2P	서비스를 요청하는 클라이언트에 동시에 서비스를 제공. 서비스를 요청하는 클라이언트에 동시에 서비스 제공. 동일한 수신 전송D 양>대칭적 sys		

최근 스타일

명칭	상세
서비스 기반/지향	• 서비스: 플랫폼 독립적 표준 인터페이스>기업 업무 표현. 상호 조합 o ▸ 일반 모듈(타 모듈과 긴밀), 인터페이스는 플랫폼에 독립적(혼자o) • 서비스 기반 아키텍: 서비스 기반 구성. • 서비스 지향 아키텍: 서비스 중심 개발 운영 ▸ 여러 시스템에서 서비스 공유
마이크로서비스 아키	작은 독립적 단위로 동작하는 서비스 구동되도록. 공유보단 독립 실행
이벤트 중심 MSA	서비스간의 통신이 이벤트 중심으로 수행 + MSA
서버리스 아키텍	서버 관리 필요 X 서비스 구축 운영. 서버리스(서버 관리 x), cloud 활용

10. 디자인 패턴

- 아키텍 패턴=SW구조 패턴화(큰 뼈대) vs 디자인 패턴=구조 내 특정 문제 피하기 위한 패턴(코드)
- 패턴 사용 장점: 생산성 증가, 경험 전달 및 학습, 솔루션 찾기 논의 제거, 품질 향상
- 검증된 솔루션이므로 효과적인 문제 해결, 장황한 설명보다 효율적
- 패턴: 반복되는 현상을 추상화하여 정의한 것
- 전략 패턴 — 알고리즘 정의, 캡슐화 ⇒ 독립적으로 변경/사용

GoF, Gang of Four 패턴 — 23개의 SW 디자인 패턴

- 생성 패턴 — 생성, 초기화 관여.클래스의 인스턴스가 어떻게 생성되는가를 추상화
- 구조 패턴 — 클래스, 객체 합성. 클래스의 기본 구조를 확장해 더 큰 구조 제공하도록
- 행위 패턴 — 클래스, 객체 간 상호작용, 책임분산 방법. (객체에 주어지는 기능/책임과 관련)
- 패턴의 정의: (일반적)패턴 이름, 해결하려는 문제, 적용 대상, 구조, 적용 시 고려 사항, 제약 사항

전략패턴 SinUduck(행위패턴 ⊃ 전략패턴)

오리(부모) > 러버덕 상속(날지 않음)

- 코드 재사용 관점에서 오버라이드를 통해 해결한 것처럼 보이지만 유지보수 생각하면 비효율 >> 달라지는 부분과 달라지지 않는 부분 분리 후 캡슐화.(새로운 클래스) >> 달라진 부분만 고친다.
- 알고리즘군을 정의하고 캡슐화하여 각 알고리즘을 수정해 쓸 수 있도록 하는 패턴. 독립적 변경o

```
public interface Flybehavior{ public void fly(); }
public class FlyWithWings implements FlyBehavior {
  public void fly() { System.out.println("날고있어요!"); }
}
public class Duck {
  Flybehavior flyBehavior; public Duck{};
  public void performFly() { flyBehavior.fly(); }
}
public class MallardDuck extends Duck {
  public MallarDuck() { flyBehavior = new FlyWithWings(); }
}
```

싱글턴 패턴(⊂ 생성패턴) 초콜릿 보일러

클래스 인스턴스(객체)가 하나만 있도록 유지하는 패턴. 생성자 여러번 호출 but 객체 하나.

```
public class Chocolateboiler { private boolean empty, boiled;
private static Chocolateboiler uniqueInstance; private ChocolateBoiler(){empty=true;
boiled=flase; }
public static synchronized ChocolateBoiler getInstance(){if(uniqueInstance == null)
{ uniqueInstance = new ChocolateBoiler()}; return uniqueInstance;}}
```

Observer pattern(⊂ 행위 패턴) 신문사(subject)+구독자(observer. 다수) 일대다 관계

옵저버들의 목록을 객체에 등록하고 상태변화 시 메서드를 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 함. (옵저버 = 객체 상태 변화를 관찰하는 객체). 장점: 느슨한 결합력. 유연성.

```
public class WeatherData implements Subject {
  private List<Observer> observers; private float temp, humidity, pressure;
  public void refisterObserver(Observer o){observers.add(o);}
  public void removeObserver(Observer o){obervers.remove(o);}
  public void notifyObserver(Observer o){
    for(Observer observer : observers){ observer.update(temp, hum, pressure);}}}
public class CurrentCondition implements Observer DisplayElement { 변수 지정!
  public CurConditionDisplay(WeatherData weatherData){this.weatherData=weatherData;
weatherData.registerObserver(this);}
  public void update(float temp, float hum, float pressure){this.temp=temp;
this.hum=hum;display();}
  public void display(){} }
```

팩토리 메소드 패턴(⊂ 생성 패턴)

객체를 생성하기 위한 인터페이스를 정의하는 과정에서 어떤 클래스의 인스턴스를 만들지 서브클래스에서 결정.= 클래스 인스턴스를 만드는 일을 서브클래스가 수행. 팩토리= 객체생성 클래스

```
public class SimplePizzaFactory{
  public Pizza createPizza (String type){
    Pizza pizza = null; if(type.equals("cheese")) pizza = new CheesePizza();
elseif ...
}}
Pizza class pizzaStore{
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) this.factory=factory;
  public Pizza orderPizza(Stirng type){ Pizza pizza; pizza=factory.createPizza(type);
pizza.prepare(); }
} // 팩토리가 피자 타입에 따라 피자 객체를 만들어주므로 가게에서 피자 객체 생성할 필요 없음.
```

피사드 패턴 홈시어터.

- 서브시스템에 있는 일련의 인터페이스를 통합 인터페이스로 묶어 주는 패턴. 파사드 클래스에 생성하고자 하는 서브시스템(아래있는 클래스)을 모두 인스턴스 변수로 가져옴. 그 안에서 이 모든 서브시스템을 이용하는 걸 하나의 메소드로 처리.

어댑터 패턴

기존 `Duck{public void quack(), fly()}`, 업체 제공 클래스 `Turkey{goobble();fly();}`

- 특정 클래스 인터페이스를 클라이언트에서 요구하는 다른 인터페이스로 변환.

```
public class TurkeyAdapter implements Duck{
  Turkey turkey; public TurkeyAdapter(Turkey turkey){ this.turkey = turkey; }
  public void quack(){ turkey.boobble(); } public void fly(){ turkey.fly(); }
}
```

11. 코딩

KISS[3]	Keep it simple stupid/간결, 이해 쉬운 코드
YAGNI[3]	You ain't gonna need it/안 필요하면 구현하지 마
DRY[3]	Do not Repeat Yourself/중복 코드 피해. 반복시 추상화
좋은코드를 위한 규칙 1) Fuck the optimization, Readability first/가독성 우선	
2) Architecture first/아키텍처(대략적 구조) 우선 개발	
3) Test Coverage	테스트가 필요한 경우 • 최소 한달은 변경하지 않을 모듈, 마이크로서비스를 개발하는 경우 • 오픈소스 코드, 금융 관련 코드 작성하는 경우 • 코드 업데이트 시 테스트 업데이트까지 가능한 비용 있는 경우 필요없는 경우 • 홍보용 소프트웨어 개발하는 경우/ 작은 팀, 빠르게 변하는 코드 • 간단한 테스트 스크립트 작성 가능한 경우
4) Simple	• KISS 원리 지킬 것. >> 버그 감소, 디버깅 시간 감소 • 적정 수준으로 추상화 패턴 적용. >> 가독성, 유지보수
5) 주석은 보조	• 좋은 코드는 주석이 없어도 이해되는 코드 • 메서드 정의 + 사용법 >> 주석
6) 강한 응집력과 느슨한 결합력 추구 << 마이크로서비스 아키텍처활용.(다양한 환경 지원)	
7)Code reviews가 항상 좋은 건 아니다	• 내재된 오류를 찾는 것이 목적 • 개발 대상 응용 영역에도 친숙하다면 코드 리뷰 의미 없음
8) Refactoring does not work	• 체계성 무시하고 기능 실행에 초점 (추후 리팩토링 의도) >> 기술 부채로 돌아옴. • 프로토타입 단시간 개발 시 리팩토링 적용 여지는 있다.
9) 자동화 도구 사용	자동화를 통해 작업 시간 감소, 비용 절감, 품질 향상
10) 취미를 가져라	새로운 아이디어 개발의 기회. 생산성 향상, 개발 집중도 향상
11)여유시간에 새로운 거 배워라. 새로운 기술/환경 적용, 새로운 스타일 코드, 기술 변화 대비	

클린 코드 의 모듈화와 응집도는 상충적인 관계로 적정 수준 맞춰야 해.

잘 동작하는 코드
- 사이드케이스 대응
- 정확성 — 의도한 대로 동작
- 성능 — 적절한 자료구조와 알고리즘 ⇒효율
- 신뢰성(안전성) — 예상하지 않은 오류 처리
- 확장성 — 큰 규모 처리도 가능하도록 설계

유지보수 용이
- 가독성 — 명확한 변수, 함수명. 읽기 쉽게
- 모듈화 - 코드>작은 모듈. 각 모듈 특정 기능
- 응집도 - 모듈 내 구성요소들 관련있어야 해
- 재사용성 - (ex. func, class, lib)
- 유연성 - 코드는 변화에 대응할 수 있어야 해
- 문서화 - 충분한 문서화 필요. (ex.주석)

시큐어 코딩 SW 안전성, 신뢰성, 보안성 향상을 위한 규칙과 권고로 구성.
- 구현 단계 잠재적 보안 취약점 사전 제거 ⇒ 외부 공격으로부터 안전한 SW 개발.

정적 분석 도구 정적: 코드 실행 없음. 동적: 코드 실행

- SAT(Static Analysis Tool) : 정적으로 소스코드 분석> 문제점 찾아주는 도구
- Lint: 정적으로 문법과 스타일 체크하는 도구

오픈소스 활용 시 주의사항
- 라이선스 - 라이선스 권한 확인, 검토
- 악성코드
  1)보안 취약점 문제 점검 도구 활용 2)지속적인 관리: 취약점 발견 ⇒ 공지됨 — 정기 점검 필요

좋은 소프트웨어 코드의 6가지 공통점
- 가독성, 주석 well, 코드 구조 간결, 변경에 탄력적, 관리 가능, 제 기능을 해야한다.

11. 테스팅

검증과 검사
- 검증: 이전-현재 결과물이 동일한가 (요구사항=설계=코드)
- 검사: 산출물이 사용자의 요구사항을 충족하는가
- 정적활동 - 컴파일 타임 테스트(실행하지 않고 정적 형태로 V&V 실행) ex. 리뷰, 코드 리딩
  동적 활동 - 런타임 테스트(코드를 실행하며 과정 살핌) ← 주로 얘가 테스트
- 목적: 버그 결함 식별 위함 … 정확성, 신뢰성, 성능 평가와 향상
- 테스트활동
  입력 데이터 준비 ⇒ 실행 ⇒ 결과 모니터링 ⇒ 의도와 실제 비교 >>차이 발생 시 디버깅 수행

SW테스트 기본 원칙

- 테스팅: 결함이 존재함을 밝힘(결함이 무엇인지, 어디인지 알 수 없음)
- 디버깅 : SW의 결함 위치를 찾고 수정.

1) 테스팅은 결함이 존재함을 밝히는 활동. 빨리 많이 발견할 수록 좋다.
2) 완벽한 테스팅 불가능 … 가장 중요한 부분 중점 테스트
3) 개발 초기 단계부터 수행해야 함
4) 결함 집중 현상 인지 … 파레토 법칙(오류80%는 전체 모듈 20%에서 발견)
… 결함 발생한 모듈에서 계속 발생 가능성 높음
5) 오류 부재의 궤변 … 결함이 없어도 요구 만족 못하면 실패

SW 테스트 용어

- **Defect**[3]: 요구사항-기대결과와 실제의 차이 [일반적으로 3개 혼용]
- **Fault**[3]: 잘못된 논리, 데이터 상태 등으로 인한 잠재적 원인. ex. 잘못된 암호화 알고리즘 구현
- **Bug**[3]: 코딩 실수, 사용자-테스터에 의해 발견되고 개발팀에서 승인한 것.
- **테스트 절차/시나리오**: 테스트를 어떻게 수행할 지 개략적으로 기술하는 계획
  - **테스트 스크립트**: 스텝이 명시된 좀 더 구체화 된 시나리오
- **테스트 케이스**: 시나리오 실행을 위한 명확한 계획. 절차>입력>예상 결과
  입력 데이터 & 예상 결과의 쌍으로 정의. TC식별자, 테스트 항목, 입력, 결과, 기타사항 명세
- **자동 테스트 스크립트** : 수동 테스트 스크립트/테스트 케이스 실행을 자동화 (=테스트 코드)

테스트 오라클

- 테스트 오라클: 테스트 실행 결과 검증 메커니즘. 예상 결과와 비교. (테스트오라클 = 예상 결과)

오라클 생성을 위한 소스

테스터의 주관적 판단, 기존 유사 프로그램의 실행 결과 비교,
회귀 테스트에서 사용된 테스트 결과 사용(수정에 대한 테스트), 정형화된 수학 공식 통한 산출 등.

테스트 단계

- 단위 테스트:SW 구성 단위 중심 테스트. 테스트 드라이버, 스텁 필요.
  - driver:하위 모듈o, 상위 모듈x. 상향식. 없는 상위 모듈 인터페이스 역할.
  - stub:하위 모듈x, 상위 모듈o. 하향식. 하위에서 상위 호출 시 가짜 모듈 역할.
- 통합 테스트: 단위테스트가 종료된 모듈들을 하나씩 통합하며 수행하는 테스트 모듈
  - 비점증적 통합: 모든 모듈을 한 번에 통합 후 테스트. (결함 원인 식별 어렵)
  - 점증적 통합: 단위 모듈을 한 번에 하나씩 통합 후 테스트. (아직 통합 안 된 모듈의 스텁 필요)
    ex. 하향식 통합, 상향식 통합, 스레드 기반 통합, 샌드위치 통합, 핵심 모듈 기반 통합
- 시스템 테스트 : 개발자 환경의 통합 sw테스트. 하드웨어 환경 갖추어 테스트
- 인수 테스트 : 사용자 환경에서 개발된 sw 테스트. 데모형식 진행(요구 기능 하나씩 실행)
  - 알파 테스팅: 개발 조직 내부 사용자가 테스트
  - 베타 테스팅: 개발 조직 외부 사용자. (베타:일정 사용자, 감마:배포 후 다수의 사용자)
- 회귀 테스트: 변경(수정,추가)이 일어난 후 수행하는 테스트
  - 전수회귀(Reset All): 모든 TC, 추가 기능 TC 모두 테스트 >> 비용↑ 커버리지↑
  - 우선순위 기반 회귀 테스트: 변경부분 핵심 기능 위주로 테스트 >> 비용↓ 커버리지↓

12. 화이트박스 테스트(=구조적 테스트)

화이트 박스 테스트

- 소스코드가 오픈된 상태에서 수행. 개발자 관점 단위 테스팅
- 목표: 모든 실행 경로 확인 > 코드의 모든 분기 테스트, 논리적 오류 발견, 사용x 도달x 코드 식별
- 장점: 코딩된 동작 다룸, 객관적, 자동
- 주요 커버리지 종류 : 문장, 분기, 조건, 데이터흐름, MC/DC>> 전체 커버리지 최대화 목표

문장 커버리지 (수행된 문장의 수/전체 문장의 수)*100. 모든 문장은 최소 한번씩 거쳐가야한다.
분기(결정) 커버리지 DC (수행된 분기수/전체 분기수)*100.분기=전체조건식 한번씩 T/F모두 거쳐
- if문(T/F두 경로 한번씩), switch(모든 case, default문 선정), for/while(한번은 루프내부 실행)
- 100% 문장 커버리지 수행이 모든 분기를 포함하는 게 아니다.

조건 커버리지 CC (수행된 조건 수/전체 조건 수)*100. 각 조건식이 T/F모두 실행하도록 하기
- 진리표에 모두 작성

MC/DC 커버리지 조건식 판단을 위한 모든 조건에 대하여 최소한 한 번씩 참과 거짓이 고려되어야 하며, 각 조건이 전체 진리 값의 결정에 영향을 미친다는 것을 보여야 함.
- 조건 커버리지를 고려하는 TC 조합의 수를 줄이면서 개별 조건식에 대한 테스트를 수행할 수 있도록 고안된 기법.
- 각 독립 조건식이 전체 조건 판단에 영향을 미치는 경우만 선택. 아니면 TC에서 제외.

뮤테이션 기반 테스트

- 의도적으로 코드에 작은 변형(mutation) 후 테스트하는 기법. 오류 기반 테스트 중 하나.
- 랜덤한 위치에 코드를 변형하고 변형된 코드를 테스트하여 기존 테스트 케이스가 변형을 감지할 수 있는지를 확인. >> 이를 통해 테스트 케이스 효과를 검증하고 개선할 수 있음.
- ex) `return(x-y); >> M1: return(x+y;) M2: reurn(x-0;) M3: return(0+y);`
- 변형 생성자: &&>>||, 관계연산자 대체, 문장 삭제, 배열 참조 대체A[i]>A[i21]
- 장점: 미처 다루지 못한 예외 발견 가능. TC 품질 검증 가능.
- 단점: 많은 변형 생성, 시간 소모

디버깅 자동화

- 프로세스: 테스트 스위트 ⇒ 결함 지역성 ⇒ 패치 생성 ⇒ 패치 검증

결함 지역성, 결함 위치 식별 자동화 (fault localization)
- 테스트 커버리지 기반 결함 위치 식별
  - 스펙트럼 기반 결함 위치 식별: 실패 케이스 통과 + 성공 케이스 실패한 문장 ⇒ 결함 가능성 多
- 뮤테이션 테스팅 기반 결함 위치 식별
  - 뮤턴트 별 의심도 산출 ⇒ 결함 위치 식별. (뮤턴트 삽입 위치를 결함으로 의심)

패치 코드 생성 자동화 (patch generation)
- 뮤테이션 기반 프로그램 자동 수정 (결함 위치 식별 완료 가정)
  - 결함 코드를 변형 생성자 적용해(+ > -) 후보 패치 코드(변형코드의 해결코드) 생성
- 템플릿/패턴 기반 프로그램 자동 수정
- 딥러닝 기반 프로그램 자동 수정

테스트케이스 우선순위화
- 생성된 후보 패치들 중 기존 TC를 모두 통과할 수 있는 패치 선택해야
- 패치들의 결함 식별 가능한 TC 우선 선택 (일반적인 CI/CD 환경에서도 중요)
1. 수정된 라인 커버 테케 우선 실행
2. 테케 실행 시간 짧은것 우선 실행
3. 1+2를 모두 만족하는 테케 우선 실행(최적화 기법 적용)

13. 블랙박스 테스트(=기능적 테스트)

코드 없이 테스트 수행: 입출력만 안다. 세부 작동 원리 모름(사용자 입장 테스트)
- 목표 : 소프트웨어 기능적 요구 사항 중심 수행, '출력이 사용자의 요구 잘 따르나?'만 확인
- 장점 : 도메인에 집중 가능, 코드가 필요하지 않음

휴리스틱(경험적 발견된 방법/규칙) TC 생성
- 동치 분할: 효율적이나 정확도 한계
  - 입력 자료에 초점 맞춰 TC 생성, 검사. (↓아래 주로 명세서 입출력 값 기반)
  - 동일한 동작/결과 예상되는 동등 클래스로 분할 ⇒각 경우의 대표값(중앙값)으로 TC 도출
- 경계값 분석: 보다 정확하나 TC 수 증가
  - 동치 분할 후 동치 클래스 경계 값을 테스트 데이터로 선정
    - 입력 변수에 정의된 값의 특정 영역에 대한 경계값을 선정. ex.min-1, min, max, max+1
- 원인(입력)-결과(효과) 그래프 검사
  - 입력(원인) 효과(결과) 관계 확립 후 TC 생성
  - 의사결정 테이블 기반(동치클래스와 이들의 조합 클래스 구조를 테이블 형태로 표현)
    - 테이블 상단에 입력변수 조건 나열. 하단에 상단 조건에 의해 수행되는 동작 기술. 각 컬럼이 하나의 테스트 데이터가 된다.
    - 원인 결과 그래프: 왼쪽에 모든 조건 막대기 오른쪽에 모든 결과 막대기.
    - 중간에 연산자 원(∧). y/n 조건의 경우 오른쪽에 노드 하나만 쓴다. yes인 경우에 이어.

테스트 계획 수립 ⇒ 테스트 목적 및 목표 정의 ⇒ 테스트 종료 조건 정의 ⇒ 테스트 케이스 설계(요구사항 이해 ⇒ 입력 변수 도출 및 정의 ⇒ 입력값의 동치 분할 수행 ⇒ 경계값 분석 수행 ⇒ 테스트 데이터 식별자 부여 ⇒ 테스트 케이스별 예상 결과 생성) ⇒ 테스트 준비 ⇒ 테스트 케이스 기반 테스트 ⇒ 테스트 결과 평가 ⇒ 테스트 결과 반영 ⇒ 테스트 절차 개선

GUI 테스트

- 주로 블랙박스 테스트를 가정
- GUI, 명세에 부합하는가?
  - 컴포넌트를 인식 식별, 이벤트 실행, 입력 제공, 컴포넌트가 제공하는 기능 확인, 일관적인 표현
- 수동 테스트: 요구사항에 따라 직접 테스트
- 기록 및 재생: 자동화 도구 활용, 테스트 과정 녹화/재생
- 모델 기반 테스트: 시스템 동작에 대해 모델 구성, 모델 기반으로 테스트
- 모델 기반 테스트 종류
  - 행동 모델링 기반 테스트 자동화 (화면, 행동에 대한 커버리지 최대화)
  - 몽키 테스트: 임의의 입력 ⇒ 동작 확인 (예상 못한 경우 대응)
    - 무작위 입력이기에 버그 재현불가(>> 기록 및 재생), 시간 多, 정확성 보장 어려움.

그레이 박스 테스트

- 블랙박스 + 화이트박스.
- 내부 구조 일부만 알고 테스트: TC 생성은 내부 구조 활용, 테스트는 블랙박스로 진행

시나리오 기반 테스트

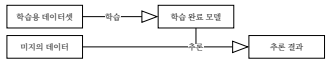
테스트 시나리오 생성 방법
1.아웃라인 방법: 기능의 연계성 고려해 시나리오 구성
- 사용자가 시스템 사용을 시작하는 입력 서비스 종료까지 순차적인 과정 식별
- 각 화면 단위 사용자 입력에 대해 TC 생성
- 테스트 결과는 화면 전환이 이루어지도록 구성
- 요구 사항 스펙 맞춰 테스트 입력의 결과를 예상 결과로 정의
- 각 화면에서 다음 화면으로 분기 가능한 모든 입력에 대한 예상 결과 생성
- 예상 결과대로 동작하는가?점검 > Pass Fail을 실제 결과에 기록

2. 유스케이스 방법: 시스템 액터 중심 시나리오 생성.
   - 시스템 수준의 테케 생성
   - 메인 시나리오에 기술된 스텝마다 하나의 TC 생성
   - 사용자 입력 기준 TC 생성, 입력에 대한 시스템의 반응을 예상 결과로 정의
   - 유스 케이스 예외 처리도 각가 테케로 정의
- 생성 방법 선정 기준
  - 테스터의 경험과 선호도, 시스템 요구사항에 주어진 형식, 시나리오 개발 위한 도구의 가용성

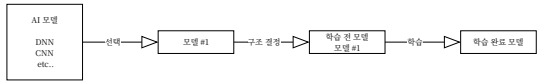
13. AISW 테스트

AI 모델 학습, 추론, 평가

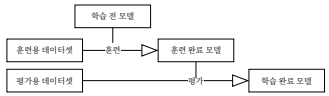
학습과 추론



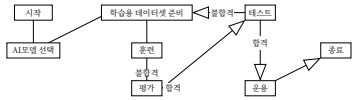
AI 모델과 학습



학습과 훈련, 평가



머신러닝의 절차



AISW 테스트 기법 AI 특성: ML을 이용해 SW 재구축은 가능. but 기존 테스트 적용 불가. 유사 입력이라도 다르게 계산&랜덤니스 >> AI에서는 동치 클래스의 적용이 어려움.

이름	과제	개요
메타모픽 테스트 기법	입력에 대한 출력의 정답 데이터(요구사항 명세)를 알 수 없음, 데이터 만들 수 없음. 입력 출력 메타모픽 관계	입력의 변화에 대해, 예측되는 출력의 변화 토대로 테스트 작성. 블랙박스 테스트 기법 포함.
뉴런 커버리지 테스트 기법	충분한 데이터셋으로 테스트되었는지 알 수 없음. 모든 노드를 활성화하는 test를 만드는 게 목표.	가능한 많은 뉴런 활성화되도록 테스트 데이터 생성 (문장커버리지와 유사)
최대 안전 반경 테스트 (강건성 테스트)	입력 데이터의 미세한 변화에 대해 출력 데이터가 크게 변화	크게 변화하지 않은 범위 구한다
커버리지 검증 기법	절대 엄수 조건이 있다.	절대 엄수 조건에 대해 모든 입력에 대해 만족하는지 검증

메타모픽 테스트:준비(x)>가공(x') > 실행 > y와 y' 관계 평가

강건성 테스트 최대 안전 반경 테스트(안전반경 계산해 평가/정량적 점수) 반경 높이도록 개선목표
- AISW 강건성: 입력 미세 변화로도 추론 결과 변화 X.(노이즈 추가에도 오류 X)
- 최대 안전 반경: AISW 강건성에 관한 지표.
  - 안전 반경: 입력 x의 결과가 y일 때, x로부터 거리 ε 내에 존재하는 임의 데이터에 대해 $\hat{x}$ 의 결과가 y인 경우의 ε
  - 최대 안전 반경: 가장 큰 값을 갖는 안전 반경
- 적대적 데이터: 강건성 체크하는 데이터. 일부로 틀리게끔 만든 데이터

뉴런 커버리지 테스트
- 기존 데이터 셋으로 사용되지 않고 있는 뉴런 확인>데이터셋 편향 의심>사용x 뉴런도 이용하게
- 뉴런 커버리지: 인공신경망 내 전체 뉴런 개수 대비 활성화된 뉴런의 비율(문장 커버리지와 유사)

시험범위 및 출제 문제

단원	단원명
9	아키텍처 스타일
10	디자인 패턴
11	코딩, 테스팅
12	화이트박스 테스트
13	블랙박스 테스트, AISW 테스트

유형	요약	배점

객	아키텍처 스타일	4
객	디자인 패턴	4
객	코딩	4
객	테스팅	4
객	AISW 테스트	4
주	아키텍처 스타일	6
주	디자인 패턴	6
주	테스팅	6
주	테스팅	6
주	AISW 테스트	6
서	디자인 패턴	10
서	코딩	10
서	화이트박스 테스트	10
서	화이트박스 테스트	10
서	블랙박스 테스트	10