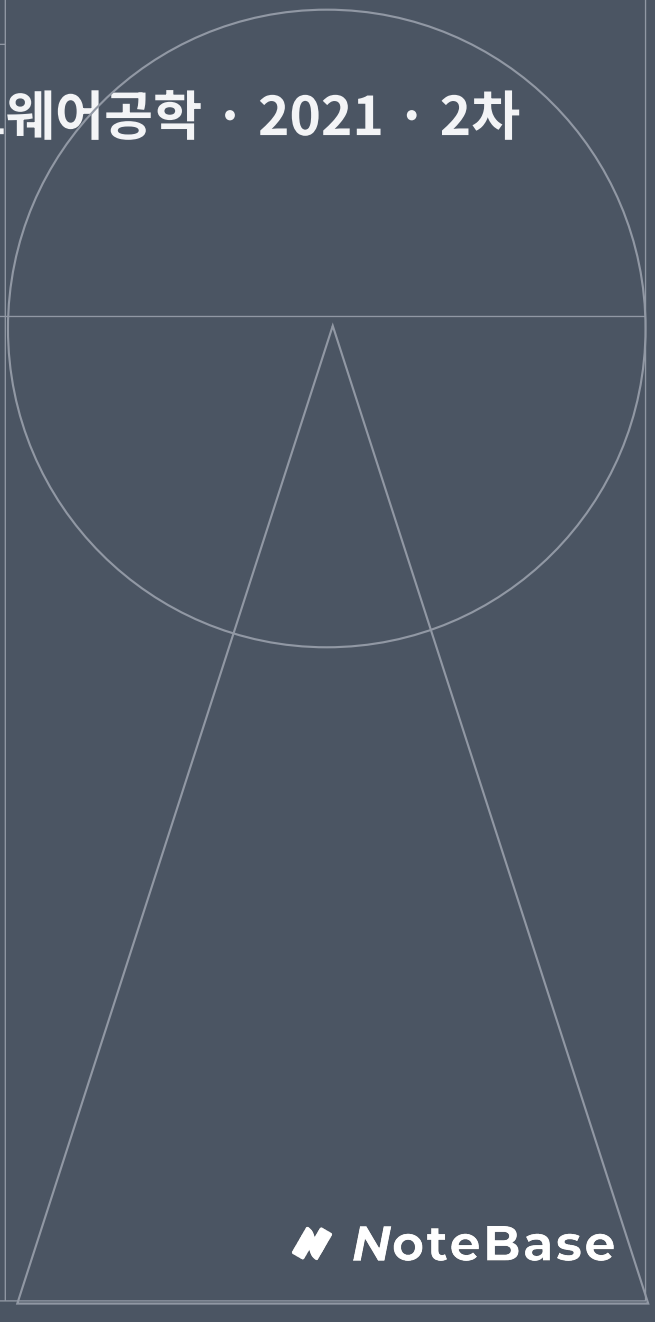
기출OX 해설집

공무원/국가직/7급 · 소프트웨어공학 · 2021 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 소프트웨어공학 · 2021 · 2차

공무원/국가직/7급 · 소프트웨어공학 · 2021 · 2차

문 1. 다음 특징을 가지는 소프트웨어 개발 방법론은?

◯ 리팩토링(refactoring)
◯ 사용자 스토리(user story)
◯ 소규모 릴리즈(small release)
◯ 짝 프로그래밍(pair programming)
◯ 테스트 우선 개발(test-first development)

① COCOMO
② 익스트림 프로그래밍(XP)
③ 통합 프로세스(UP)
④ 폭포수(waterfall) 모델

정답 2번

출제 포인트

제시된 다섯 가지 특징(리팩토링, 사용자 스토리, 소규모 릴리즈, 짝 프로그래밍, 테스트 우선 개발)이 모두 애자일 방법론 중에서도 익스트림 프로그래밍(XP)의 핵심 실천 방법(practice)이라는 점을 알아야 하는 문제이다. 즉 개별 실천 방법의 명칭을 그 실천 방법들이 속한 상위 방법론과 연결할 수 있는지를 묻는다.

핵심 개념

익스트림 프로그래밍(XP)은 켄트 벡이 제안한 대표적인 애자일 방법론으로, 변화하는 요구사항에 유연하게 대응하기 위해 짧은 개발 주기와 지속적인 피드백을 강조한다. XP의 대표적인 실천 방법에는 짝 프로그래밍(두 명이 한 컴퓨터로 함께 코딩), 테스트 우선 개발(코드 작성 전에 테스트 코드를 먼저 작성), 리팩토링(기능 변경 없이 코드 구조를 지속적으로 개선), 소규모 릴리즈(짧은 주기로 자주 배포), 사용자 스토리(사용자 요구사항을 간단한 카드 형태의 문장으로 기술), 공동 소유권, 지속적 통합, 단순한 설계 등이 있다. 제시된 다섯 특징은 모두 이 목록에 포함된다.

NoteBase

선지별 해설

① COCOMO — X. COCOMO(Constructive Cost Model)는 보엠이 제안한 소프트웨어 개발 비용과 기간을 예측하는 산정 모형이다. 리팩토링이나 짝 프로그래밍 같은 개발 실천 방법을 규정하는 방법론이 아니라 비용 산정 공식이므로 해당하지 않는다.
② 익스트림 프로그래밍(XP) — O. 리팩토링, 사용자 스토리, 소규모 릴리즈, 짝 프로그래밍, 테스트 우선 개발은 모두 XP의 핵심 실천 방법이므로 정답이다.
③ 통합 프로세스(UP) — X. UP(Unified Process)는 유스케이스 중심, 아키텍처 중심, 반복적·점진적 개발을 특징으로 하며 도입·상세·구축·전이 단계로 구성된다. 짝 프로그래밍이나 테스트 우선 개발과 같은 실천 방법을 규정하지 않으므로 해당하지 않는다.
④ 폭포수(waterfall) 모델 — X. 폭포수 모델은 요구분석부터 유지보수까지 각 단계를 순차적으로 진행하는 전통적인 예측형(계획 중심) 개발 모델로, 반복이나 점진적 릴리즈를 전제로 하지 않는다. 제시된 특징들과 정반대의 성격을 가지므로 해당하지 않는다.

오답 포인트

COCOMO를 방법론으로 착각하기 쉬운데, 이는 비용 산정 모형일 뿐 개발 실천 방법을 제시하지 않는다. 또한 UP을 애자일 방법론으로 혼동하는 경우가 있는데, UP은 반복적 개발을 하지만 짝 프로그래밍이나 테스트 우선 개발 같은 XP 고유의 실천 방법과는 구분된다.

다시 나오면 이렇게 판단

리팩토링, 짝 프로그래밍, 테스트 우선 개발, 소규모 릴리즈, 사용자 스토리 중 두세 개만 제시되어도 XP를 가리키는 것으로 판단한다. 반대로 스크럼이 언급되면 스프린트·백로그·데일리 스크럼 등 다른 용어가, UP이 언급되면 유스케이스·아키텍처 중심이라는 용어가 함께 나오는지 확인해 구분한다.

암기 포인트

XP 5대 실천 방법으로 리팩토링, 짝 프로그래밍, 테스트 우선 개발, 소규모 릴리즈, 사용자 스토리를 묶어서 외워 둔다. COCOMO는 비용 산정, UP은 유스케이스·아키텍처 중심 반복 개발, 폭포수는 순차적 계획 중심 모델이라는 한 줄 구분과 함께 기억해 둔다.

NoteBase

문 2. 다음 상황에 가장 적합한 소프트웨어 아키텍처 패턴은?

◯ 데이터 관리 컴포넌트, 인터랙션 제어 컴포넌트, 화면 관리 컴포넌트를 분리
◯ 하나의 데이터를 다양한 그래프로 보여 줄 수 있는 데스크탑 응용 프로그램 개발
◯ 데이터베이스 연동 부분과 사용자 웹 화면 제공 부분으로 나눈 웹 기반 응용 프로그램 개발

① MVC(Model-View-Controller)
② 리포지토리(repository)
③ 트랜잭션 처리(transaction processing)
④ 파이프 필터(pipe-and-filter)

정답 1번

출제 포인트

소프트웨어 아키텍처 패턴 중 MVC 패턴이 갖는 세 가지 특징을 서로 다른 상황 설명으로 나누어 제시하고, 이를 종합적으로 인식해 MVC 패턴을 골라낼 수 있는지를 묻는다.

핵심 개념

MVC(Model-View-Controller) 패턴은 데이터를 관리하는 모델, 사용자 입력과 흐름을 처리하는 컨트롤러, 화면 표시를 담당하는 뷰로 역할을 분리하는 아키텍처 패턴이다. 하나의 모델에 여러 개의 뷰를 연결할 수 있어 동일한 데이터를 그래프 등 다양한 형태로 동시에 보여줄 수 있고, 웹 애플리케이션에서는 데이터베이스 연동 부분(모델)과 사용자에게 화면을 제공하는 부분(뷰)으로 흔히 나뉜다. 리포지토리 패턴은 여러 컴포넌트가 공유하는 데이터 저장소를 중앙집중적으로 관리하는 패턴이고, 트랜잭션 처리 패턴은 요청을 처리 흐름별 트랜잭션 단위로 나눠 순차 실행하는 패턴이며, 파이프-필터 패턴은 데이터를 여러 필터를 거쳐 순차적으로 가공해 다음 단계로 전달하는 패턴이다.

선지별 해설

① MVC(Model-View-Controller) — O. 데이터 관리(모델)와 인터랙션 제어(컨트롤러), 화면 관리(뷰)를 분리하는 첫 번째 상황, 하나의 데이터를 여러 그래프(다중 뷰)로 보여주는 두 번째 상황, 데이터베이스 연동(모델)과 웹 화면 제공(뷰)으로 나누는 세 번째 상황 모두 MVC 패턴의 전형적인 특징이다. 정답.
② 리포지토리(repository) — X. 여러 컴포넌트가 공유하는 데이터 저장소를 중앙에서 관리하는 패턴으로, 인터랙션 제어·화면 관리 컴포넌트의 분리나 하나의 데이터를 다양한 뷰로 표시하는 특징과는 관련이 없다.
③ 트랜잭션 처리(transaction processing) — X. 여러 작업을 하나의 처리 단위로 묶어 순차 실행하는 패턴으로, 제시된 상황들이 설명하는 데이터·화면 분리 구조와는 다르다.
④ 파이프 필터(pipe-and-filter) — X. 데이터를 여러 필터를 거쳐 순차적으로 가공해 넘기는 흐름 중심 구조로, 모델과 뷰를 분리하고 하나의 데이터를 다양한 형태로 보여주는 상황과는 맞지 않는다.

NoteBase

오답 포인트

각 패턴의 정의를 따로따로만 외우면 여러 상황 설명이 동시에 제시될 때 무슨 패턴을 가리키는지 종합하지 못한다. 이 문제처럼 관심사 분리, 하나의 데이터에 대한 다중 표현, 데이터 처리부와 화면 제공부의 분리가 함께 제시되면 이 세 특징을 모두 만족하는 패턴이 무엇인지를 먼저 따져야 한다.

다시 나오면 이렇게 판단

상황 설명에 데이터 관리 부분과 화면(뷰) 부분이 나뉘어 있고, 하나의 데이터를 여러 형태로 동시에 보여줄 수 있다는 표현이 있으면 MVC를 우선 고려한다. 데이터 저장소를 공유·중앙관리하는 설명이면 리포지토리, 데이터를 순차적으로 가공해 넘기는 설명이면 파이프-필터, 작업을 단위로 묶어 처리하는 설명이면 트랜잭션 처리로 구분한다.

암기 포인트

MVC = 모델(데이터 관리)+뷰(화면 표시, 다중 뷰 가능)+컨트롤러(인터랙션 제어); 리포지토리 = 데이터 접근·저장소 중앙화; 파이프-필터 = 순차 가공 흐름; 트랜잭션 처리 = 작업 단위 순차 처리.

NoteBase

문 3. LOC(Lines Of Code) 기법에 의해 예측된 프로젝트의 총라인 수가 80,000 LOC이고, 투입 개발자 수는 8명, 개발자 1인당 월평균 생산성이 500 LOC일 경우, 개발에 소요되는 시간은?

① 10개월

② 16개월

③ 20개월

④ 40개월

정답 3번

출제 포인트

LOC(원시 코드 라인 수) 기법을 이용한 개발 기간(노력) 산정 방식을 아는지 묻는 문제이다. 총 라인 수와 투입 인원, 1인당 월평균 생산성이 주어졌을 때 전체 개발 소요 기간을 구하는 계산이 핵심이다.

조건 정리

예측된 총 라인 수(LOC) = 80,000 LOC. 투입 개발자 수 = 8명. 개발자 1인당 월평균 생산성 = 500 LOC/월.

사용할 공식

먼저 전체 개발팀의 월간 생산성(팀 생산성)을 구한다. 팀 월간 생산성 = 개발자 수 × 1인당 월평균 생산성
그다음 개발 소요 기간을 구한다. 개발 기간(개월) = 총 LOC팀 월간 생산성

계산 과정

팀 월간 생산성을 먼저 계산한다. 8명 × 500 LOC/월 = 4,000 LOC/월 이 값으로 총 라인 수를 나누어 소요 기간을 구한다. 80,000 LOC4,000 LOC/월 = 20 개월 따라서 이 프로젝트의 개발에 소요되는 시간은 20개월이다.

선지 판단

① 10개월 → X. 80,000을 4,000으로 나누면 20이 되므로 10개월은 팀 생산성을 8,000 LOC/월로 잘못 계산했을 때(예: 인원과 생산성을 곱이 아니라 다른 값으로 착각)나 나눗셈을 절반으로 잘못한 경우에 나오는 값이며 조건과 맞지 않는다.
② 16개월 → X. 80,000을 5,000으로 나눈 값으로, 1인당 생산성이나 인원수를 잘못 대입했을 때 나오는 값이며 주어진 조건(8명×500)과 일치하지 않는다.
③ 20개월 → O. 총 라인 수 80,000 LOC를 팀 전체의 월간 생산성 4,000 LOC(8명×500 LOC)로 나눈 값과 정확히 일치한다.
④ 40개월 → X. 인원수를 반영하지 않고 총 라인 수를 1인당 생산성(500 LOC)만으로 나누면 160개월이 되어야 하므로 이 값도 아니며, 조건을 절반만 반영하거나 다른 값과 혼동했을 때 나올 수 있는 오답이다.

NoteBase

오답 포인트

이 유형의 함정은 '총 라인 수 ÷ 1인당 생산성'만 계산하고 투입 인원수를 곱하는 과정을 빠뜨리는 것이다.
반드시 개발자 수와 1인당 생산성을 먼저 곱해 팀 전체의 월간 생산성을 구한 뒤, 그 값으로 총 라인 수를 나누어야 한다.

다시 나오면 이렇게 판단

LOC 기법 계산 문제가 나오면 가장 먼저 '전체 규모(LOC) ÷ 팀 전체 생산성'의 구조인지 확인한다. 팀 전체 생산성은 항상 '투입 인원수 × 1인당 생산성'으로 구한다는 점을 놓치지 않도록 한다.

암기 포인트

LOC 기법 개발 기간 = 총 LOC ÷ (투입 인원수 × 1인당 월평균 생산성). 인원수를 곱하는 것을 빠뜨리지 않는다.

NoteBase

문 4. 사용자 요구분석 명세서의 바람직한 특성으로 옳지 않은 것은?

① 서로 모순되는 부분이 없어야 한다.

② 고객과 개발자가 동의한 것이어야 한다.

③ 고객과 개발자 모두 쉽게 이해할 수 있어야 한다.

④ 시스템의 구조와 서브 시스템의 구조를 포함하여 기술해야 한다.

정답 4번

출제 포인트

사용자 요구분석 명세서(SRS, Software Requirements Specification)가 갖추어야 할 바람직한 특성을 정확히 알고 있는지 묻는 문제이다. 요구분석 명세서는 시스템이 '무엇을(what)' 해야 하는가를 기술하는 문서이며, '어떻게(how)' 구현할 것인가에 해당하는 설계 내용은 포함하지 않아야 한다는 점이 핵심이다.

핵심 개념

요구분석 명세서의 바람직한 특성으로는 일반적으로 정확성(정확해야 함), 명확성(모호하지 않아야 함), 완전성(빠짐없이 기술), 일관성(모순이 없어야 함), 검증 가능성(테스트로 확인 가능해야 함), 이해 가능성(고객과 개발자 모두 이해 가능), 수정 용이성 등이 꼽힌다. 반면 시스템 내부 구조나 서브 시스템 구조와 같은 설계적 요소는 요구분석 단계가 아니라 설계 단계에서 다루어지는 내용이므로, 요구분석 명세서에 포함시켜야 할 바람직한 특성이 아니다.

선지별 해설

① 서로 모순되는 부분이 없어야 한다. → O. 일관성(consistency)에 해당하며, 명세서 내 요구사항들 사이에 상충이 없어야 한다는 것은 바람직한 특성이 맞다.
② 고객과 개발자가 동의한 것이어야 한다. → O. 요구분석 명세서는 고객과 개발자가 합의한 내용을 담아야 하므로 바람직한 특성이다.
③ 고객과 개발자 모두 쉽게 이해할 수 있어야 한다. → O. 이해 가능성(understandability)에 해당하며, 명세서는 전문성이 다른 고객과 개발자가 함께 읽고 소통할 수 있어야 하므로 바람직한 특성이다.
④ 시스템의 구조와 서브 시스템의 구조를 포함하여 기술해야 한다. → X. 요구분석 명세서는 시스템이 무엇을 해야 하는지(요구사항)를 기술하는 것이지, 시스템이나 서브 시스템의 내부 구조를 기술하는 것은 설계 명세서의 역할이다. 요구분석 단계에서 구조적 설계 내용까지 포함시키면 요구사항과 설계가 뒤섞여 명세서의 목적에 어긋나므로, 바람직한 특성이 아니다.

NoteBase

오답 포인트

①②③은 모두 요구분석 명세서가 갖추어야 할 특성을 올바르게 서술하고 있어 옳은 것으로 헷갈리지 않는다. 반면 ④는 언뜻 보면 시스템을 자세히 기술하는 것이 바람직해 보여 옳은 특성으로 착각하기 쉬운데, '구조를 포함하여 기술'하는 것은 요구사항이 아니라 설계에 해당하는 내용이므로 요구분석 명세서의 바람직한 특성으로 볼 수 없다는 점을 놓치지 않아야 한다.

다시 나오면 이렇게 판단

요구분석 명세서 관련 문제는 선지에서 '무엇을 할 것인가(요구사항)'를 설명하는 표현인지, '어떻게 구현할 것인가(설계, 구조)'를 설명하는 표현인지부터 구분한다. 일관성·완전성·명확성·이해 가능성·검증 가능성처럼 요구사항 자체의 품질을 말하는 선지는 옳은 특성이고, 시스템 구조·모듈 구조·자료구조 등 구현·설계 요소를 담으라는 선지는 틀린 특성으로 판단한다.

암기 포인트

요구분석 명세서는 What(무엇을 해야 하는가)을 담고, How(어떻게 구현하는가·구조)는 담지 않는다는 원칙을 기억해 둔다. 바람직한 특성 키워드는 일관성·완전성·명확성(모호하지 않음)·이해 가능성·검증 가능성·수정 용이성으로 정리해 둔다.

NoteBase

문 5. 아키텍처의 4+1 관점에 대한 설명으로 옳지 않은 것은?

① 배치 관점은 시스템을 구성하는 처리 장치 간의 물리적 배치에 초점을 둔다.

② 프로세스 관점은 시스템에서 런타임에 상호 작용하는 프로세스들이 어떻게 구성되는지 보여 준다.

③ 구현 관점은 물리적 시스템에서 사용하는 소프트웨어 서브 시스템의 모듈이 어떻게 구조화되어 있는가에 관심을 둔다.

④ 유스케이스 관점은 시스템의 기능을 제공하기 위해 필요한 클래스나 컴포넌트의 종류, 그리고 이들의 관계에 초점을 둔다.

정답 4번

출제 포인트

이 문제는 Kruchten의 4+1 아키텍처 뷰 모델에서 각 관점(뷰)이 무엇을 표현하는지 정확히 구분할 수 있는지를 묻는다. 4+1 뷰는 논리 관점(Logical View), 프로세스 관점(Process View), 구현 관점(Development View), 배치 관점(Physical/Deployment View)의 네 가지 뷰와, 이 네 뷰를 하나로 엮어 검증하는 유스케이스 관점(시나리오, Scenarios)으로 구성된다. 각 뷰가 다루는 대상(처리 장치, 프로세스, 모듈, 클래스)과 유스케이스 관점의 역할을 헷갈리게 만든 선지가 정답이 된다.

핵심 개념

논리 관점은 시스템이 제공하는 기능을 클래스, 객체, 컴포넌트와 이들 간의 관계로 표현하는 뷰이다. 프로세스 관점은 런타임에 동작하는 프로세스·스레드들의 상호 작용과 동시성, 통신 구조를 보여 준다. 구현 관점(개발 관점)은 실제 개발 환경에서 소프트웨어 모듈(서브시스템, 라이브러리, 패키지)이 어떻게 구조화되고 관리되는지를 다룬다. 배치 관점(물리 관점)은 소프트웨어가 어떤 물리적 노드(서버, 처리 장치)에 배치되는지, 노드 간 물리적 연결을 다룬다. 유스케이스 관점은 별도의 정적 구조가 아니라, 주요 유스케이스(시나리오)를 통해 앞의 네 관점이 일관되고 완전한지를 검증하고 서로 연결하는 역할을 한다.

NoteBase

선지별 해설

① O - 배치 관점(물리 관점)은 시스템을 구성하는 처리 장치들 사이의 물리적 배치와 연결을 다루므로 옳은 설명이다.
② O - 프로세스 관점은 런타임에 상호 작용하는 프로세스들이 어떻게 구성되고 동작하는지를 보여 주므로 옳은 설명이다.
③ O - 구현 관점은 물리적 시스템에서 실제로 사용되는 소프트웨어 서브시스템(모듈)이 어떻게 구조화되어 있는지에 관심을 두므로 옳은 설명이다.
④ X - 이 설명은 유스케이스 관점이 아니라 논리 관점(Logical View)에 대한 설명이다. 유스케이스 관점은 클래스나 컴포넌트의 종류와 관계를 직접 규정하는 정적 구조 뷰가 아니라, 주요 시나리오를 통해 나머지 네 뷰의 설계가 실제 요구사항을 만족하는지 확인하고 통합하는 역할을 한다. 틀린 표현 → 올바른 표현: "유스케이스 관점은 클래스나 컴포넌트의 종류와 관계에 초점을 둔다" → "논리 관점이 클래스나 컴포넌트의 종류와 관계에 초점을 두며, 유스케이스 관점은 시나리오로 각 뷰를 검증·연결하는 역할을 한다."

오답 포인트

논리 관점과 유스케이스 관점을 혼동하기 쉽다. '클래스나 컴포넌트의 종류와 관계'라는 표현이 나오면 이는 정적 구조를 다루는 논리 관점의 특징이지, 시나리오 중심으로 뷰들을 검증하는 유스케이스 관점의 특징이 아니라는 점을 구분해야 한다.

다시 나오면 이렇게 판단

선지에서 각 관점의 이름과 설명이 짝지어져 나오면, 먼저 그 설명이 가리키는 대상(처리 장치=배치, 프로세스=프로세스, 모듈=구현, 클래스/컴포넌트 구조=논리, 시나리오=유스케이스)을 확인한 뒤 관점 이름과 대응이 맞는지 대조한다. 유스케이스 관점 자리에 논리 관점의 설명이 들어가 있는 유형이 반복 출제된다.

암기 포인트

4+1 뷰: 논리(클래스·컴포넌트 구조) / 프로세스(런타임 프로세스 상호작용) / 구현(개발 모듈 구조) / 배치(물리적 노드 배치) + 유스케이스(시나리오로 네 뷰를 검증·통합).

NoteBase

문 6. 다음 자바 코드에 나타난 것과 같이 동일한 이름의 메시지로 다른 구현을 호출할 수 있는 객체 지향 개념은?

```
Animal a;
a = new Dog( );
a.makeSound( ); // "멍멍" 출력함
a = new Cat( );
a.makeSound( ); // "야옹" 출력함
```

① 다형성

② 정보 은닉

③ 정적 바인딩

④ 캡슐화

정답 1번

출제 포인트

제시된 자바 코드는 상위 타입 참조 변수 Animal a에 서로 다른 하위 클래스(Dog, Cat)의 인스턴스를 대입하면서, 동일한 메시지 a.makeSound( )를 호출했을 때 실제로 실행되는 구현이 참조하는 객체의 실제 타입에 따라 달라지는 상황을 보여 준다. 이는 객체 지향의 다형성(polymorphism) 개념을 코드로 확인하는 문제이다.

핵심 개념

다형성은 동일한 이름의 메시지(메서드 호출)가 호출되는 객체의 실제 타입에 따라 서로 다른 구현으로 실행되는 성질이다. 자바에서는 상위 클래스나 인터페이스 타입의 참조 변수가 다양한 하위 클래스의 객체를 가리킬 수 있고, 메서드 호출 시점에 실제 객체의 타입에 따라 알맞은 메서드가 선택되어 실행된다(동적 바인딩에 의해 구현됨). 정보 은닉은 객체의 내부 데이터나 구현 세부사항을 외부로부터 감추는 것이고, 정적 바인딩은 컴파일 시점에 호출될 메서드가 이미 결정되는 방식으로 다형성과 반대되는 개념이며, 캡슐화는 데이터와 그 데이터를 다루는 메서드를 하나의 단위로 묶고 접근을 제한하는 것을 말한다.

NoteBase

선지별 해설

① 다형성 → O. 동일한 이름의 메시지 makeSound( )를 호출했지만 참조 변수 a가 실제로 가리키는 객체가 Dog일 때는 "멍멍"을, Cat일 때는 "야옹"을 출력하는 것은 동일한 메시지로 서로 다른 구현이 호출되는 다형성의 전형적인 예이다.
② 정보 은닉 → X. 정보 은닉은 객체 내부의 데이터나 구현을 외부에서 접근하지 못하도록 감추는 것으로, 이 코드에서 보여 주는 것은 감추는 것이 아니라 동일한 호출에 대해 다른 동작이 일어나는 현상이므로 해당하지 않는다.
③ 정적 바인딩 → X. 정적 바인딩은 컴파일 시점에 호출할 메서드가 확정되는 방식으로, 이 경우라면 a의 선언 타입인 Animal의 메서드로 고정되어 항상 같은 동작을 해야 한다. 하지만 코드에서는 실행 시점에 대입된 실제 객체(Dog, Cat)에 따라 다른 메서드가 실행되므로 정적 바인딩이 아니라 동적 바인딩에 의한 다형성이 나타난 것이다.
④ 캡슐화 → X. 캡슐화는 데이터와 메서드를 하나로 묶고 외부 접근을 제한하는 개념으로, 서로 다른 구현이 동일한 메시지로 호출되는 이 코드의 특징과는 직접적인 관련이 없다.

오답 포인트

정적 바인딩(③)을 다형성과 헷갈리기 쉬운데, 이 문제의 코드는 오히려 정적 바인딩과 반대되는 상황(실행 시점에 실제 객체 타입에 따라 다른 메서드가 실행됨)을 보여 준다는 점을 주의한다. 정보 은닉(②)과 캡슐화(④)는 접근 제어·데이터 보호와 관련된 개념으로, 동일 메시지에 대한 다른 구현 실행이라는 이 코드의 핵심 특징과는 무관하다.

다시 나오면 이렇게 판단

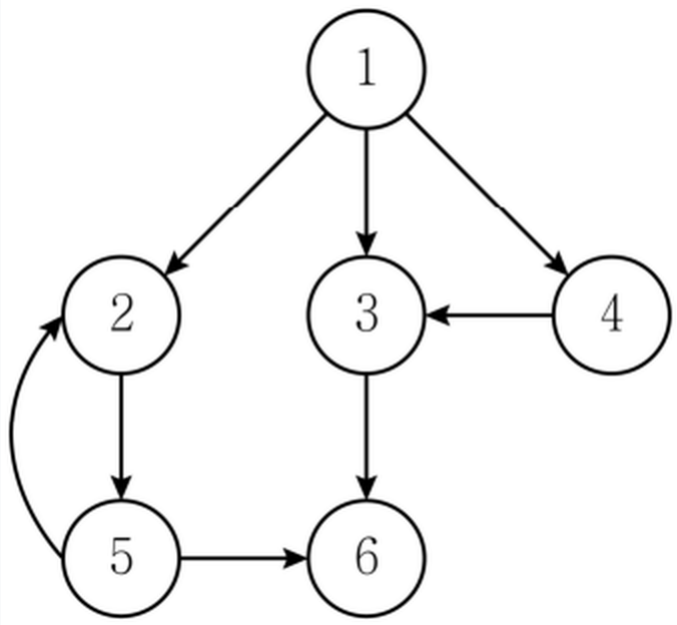
상위 타입 참조 변수에 여러 하위 클래스 객체를 번갈아 대입하고 동일한 메서드를 호출했을 때 실행 결과가 객체마다 달라지는 코드가 나오면, 곧바로 다형성을 떠올린다. 반대로 실행 결과가 항상 참조 변수의 선언 타입에 따라 고정된다면 정적 바인딩 쪽을 의심한다.

암기 포인트

다형성=같은 메시지, 다른 구현(실행 시점 실제 객체 타입에 따라 결정) / 정적 바인딩=컴파일 시점에 호출 메서드 고정(다형성과 반대) / 정보 은닉·캡슐화=데이터·구현을 감추고 묶는 것(이 코드와 무관).

NoteBase

문 7. 다음 제어흐름 그래프에 대한 순환 복잡도(cyclomatic complexity)는?



① 2

② 3

③ 4

④ 5

정답 3번

출제 포인트

제어흐름 그래프(control flow graph)가 주어졌을 때 McCabe의 순환 복잡도를 실제로 계산할 수 있는지를 묻는다. 그래프의 노드와 간선(에지) 개수를 정확히 세는 것이 핵심이다.

조건 정리

그림의 제어흐름 그래프를 노드와 간선으로 정리하면 다음과 같다. 노드: 1, 2, 3, 4, 5, 6 (총 6개). 간선: 1→2, 1→3, 1→4, 4→3, 2→5, 5→2, 5→6, 3→6 (총 8개).

사용할 공식

간선과 노드 수를 이용한 순환 복잡도 공식은 $V(G) = E - N + 2$이다. 여기서 E는 간선의 수, N은 노드의 수이며, 2는 연결된 그래프 1개를 기준으로 한 상수항이다(그래프 연결요소가 P개면 $V(G) = E - N + 2P$, 여기서는 $P = 1$).

NoteBase

계산 과정

노드 수 $N = 6$, 간선 수 $E = 8$이므로 $V(G) = E - N + 2 = 8 - 6 + 2 = 4$이다. 판정 노드(분기 노드) 기준으로 검산해도 같은 값이 나온다. 노드 1은 세 갈래(2, 3, 4)로 분기하는 판정 노드이고, 노드 5는 두 갈래(2, 6)로 분기하는 판정 노드이다. 이진 분기 노드 하나당 순환 복잡도가 1씩 증가한다고 보면, 노드 1의 3갈래 분기는 이진 분기 2개(+2)에 해당하고 노드 5의 2갈래 분기는 이진 분기 1개(+1)에 해당하여, 기본값 1에 3을 더해 역시 4가 된다.

선지 판단

① 2 → X, 간선 8개와 노드 6개를 반영하지 않고 분기 노드 수만 단순히 헤아리는 등 계산을 축소한 값이다.
② 3 → X, 노드 5에서 2로 되돌아가는 간선(5→2)을 빠뜨리고 계산하면 나오는 값으로, 실제 그래프의 간선 수를 과소 계산한 결과이다.
③ 4 → O, $E - N + 2 = 8 - 6 + 2 = 4$로 정확히 일치한다.
④ 5 → X, 간선이나 노드 수를 하나 더 많거나 적게 세었을 때 나올 수 있는 값으로, 실제 그래프와 맞지 않는다.

오답 포인트

이 문제에서 가장 흔한 실수는 노드 5에서 노드 2로 되돌아가는 간선(5→2)을 빠뜨리는 것이다. 이 간선을 놓치면 간선 수가 7개가 되어 $V(G) = 7 - 6 + 2 = 3$으로 계산되어 오답 ②를 고르게 된다. 그래프를 셀 때는 화살표(방향) 하나하나를 빠짐없이 세어야 하며, 순환 구조(2와 5 사이를 오가는 간선)가 있다는 점을 놓치지 않아야 한다.

다시 나오면 이렇게 판단

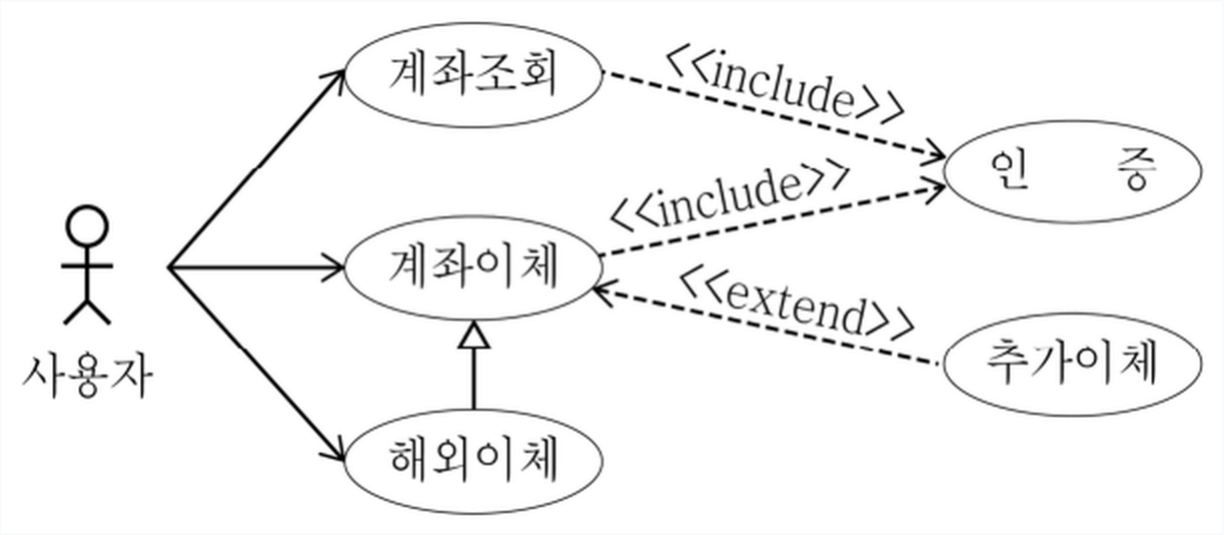
제어흐름 그래프 문제가 나오면 먼저 노드 수와 간선 수를 표로 정리하듯 하나씩 세어 N과 E를 확정한 뒤 $V(G) = E - N + 2$ 공식에 대입한다. 시간이 부족하면 판정(분기) 노드의 개수로 검산한다. 이진 분기 노드 수에 1을 더한 값이 순환 복잡도와 같아야 하며, 여러 갈래로 갈라지는 노드는 (갈래 수 - 1)만큼 반영한다.

암기 포인트

순환 복잡도 공식은 $V(G) = E - N + 2P$이며, 그래프가 하나로 연결되어 있으면 $P = 1$이라 $V(G) = E - N + 2$로 계산한다. 이 값은 판정 노드 수에 1을 더한 값과도 같고, 테스트 케이스를 설계할 때 필요한 독립 경로의 최소 개수를 의미한다.

NoteBase

문 8. 다음 유스케이스 다이어그램에 대한 설명으로 옳지 않은 것은?



① '계좌이체'는 '해외이체'보다 일반화된 유스케이스에 해당한다.

② '계좌이체' 유스케이스가 수행되면 '추가이체' 유스케이스도 수행된다.

③ '계좌조회', '계좌이체', '해외이체' 유스케이스는 '사용자' 액터에 의해 수행된다.

④ '인증' 유스케이스는 '계좌조회'와 '계좌이체' 유스케이스에 포함되어 수행된다.

정답 2번

출제 포인트

유스케이스 다이어그램에서 액터와 유스케이스의 관계, 그리고 include·extend·일반화(generalization) 세 관계의 표기와 의미 차이를 정확히 구분할 수 있는지를 묻는다. 다이어그램에서 계좌조회와 계좌이체는 각각 점선 화살표(<<include>>)로 인증을 가리키고, 해외이체는 속이 빈 삼각형 화살표로 계좌이체를 가리키며(일반화), 추가이체는 점선 화살표(<<extend>>)로 계좌이체를 가리킨다(확장). 각 관계의 화살표 방향과 필수 여부를 함께 읽어야 한다.

NoteBase

선지별 해설

① 해외이체에서 계좌이체 쪽으로 속이 빈 삼각형 화살표가 그려져 있는데, 이는 일반화 관계이며 화살표는 자식(하위) 유스케이스에서 부모(상위) 유스케이스 쪽으로 향한다. 즉 해외이체가 계좌이체를 특수화한 하위 유스케이스이고, 계좌이체가 더 일반화된 유스케이스이다. 옳은 설명이다.
② 추가이체는 <<extend>> 관계로 계좌이체를 확장한다. extend 관계는 확장 유스케이스(추가이체)가 특정 조건이 성립할 때에만 기본 유스케이스(계좌이체)의 실행 흐름에 선택적으로 추가되는 것이지, 계좌이체가 수행된다고 해서 추가이체가 항상 함께 수행되는 것이 아니다. include 관계처럼 반드시 함께 수행되는 것은 계좌조회·계좌이체와 인증 사이의 관계이다. 따라서 '계좌이체가 수행되면 추가이체도 수행된다'는 설명은 틀렸다 → 올바르게는 '추가이체는 특정 조건을 만족할 때에 한하여 선택적으로 수행될 수 있다'이다.
③ 다이어그램에서 사용자 액터로부터 계좌조회, 계좌이체, 해외이체 각각으로 실선 화살표가 이어져 있으므로 세 유스케이스 모두 사용자 액터에 의해 직접 수행된다. 옳은 설명이다.
④ 계좌조회와 계좌이체에서 인증 쪽으로 각각 <<include>> 점선 화살표가 그려져 있으므로, 인증 유스케이스는 계좌조회와 계좌이체 각각에 포함되어 그 실행 과정에서 반드시 함께 수행된다. 옳은 설명이다.

오답 포인트

extend 관계와 include 관계를 혼동하면 이 문항을 틀리기 쉽다. include는 기본 유스케이스가 실행될 때 포함된 유스케이스가 항상 함께 수행되는 필수 관계이고, extend는 확장 유스케이스가 특정 조건에서만 선택적으로 끼어드는 관계이다. 추가이체가 계좌이체를 extend한다고 해서 계좌이체 실행이 곧 추가이체 실행을 보장하지 않는다는 점을 놓치면 ②를 옳은 설명으로 오판하게 된다.

다시 나오면 이렇게 판단

화살표가 점선인지 실선인지, 그리고 화살표 끝에 <<include>>·<<extend>>·속이 빈 삼각형 중 어느 표기가 붙어 있는지부터 확인한다. <<include>>는 '항상 함께 수행', <<extend>>는 '조건에 따라 선택적으로 수행', 속이 빈 삼각형은 '화살표가 향하는 쪽이 더 일반화된 부모 유스케이스'로 각각 치환해서 선지를 검증하면 빠르게 오답을 가려낼 수 있다.

암기 포인트

include=필수 포함(항상 함께 수행), extend=조건부 확장(선택적 수행), 일반화 화살표(속이 빈 삼각형)는 자식→부모 방향이며 화살표가 가리키는 쪽이 더 일반화된 유스케이스이다.

NoteBase

문 9. 다음 설명에 해당하는 아키텍처는?

네트워크의 어떠한 노드라도 주어진 연산을 수행할 수 있는 비중앙집중적인(decentralized) 아키텍처로, 서버와 클라이언트의 구분이 없다. 별도의 중앙 서버 없이 사용자의 PC 사이에서 파일을 주고받기 위한 파일 공유 네트워크 등이 이에 해당한다.

① 마이크로서비스(microservice) 아키텍처

② 분산 컴포넌트(distributed component) 아키텍처

③ 서비스 지향(service-oriented) 아키텍처

④ 피어 투 피어(peer-to-peer) 아키텍처

정답 4번

출제 포인트

소프트웨어 아키텍처 스타일 중 '비중앙집중적(decentralized)'이고 노드 간 서버-클라이언트 구분이 없는 구조를 설명과 명칭으로 매칭하는 문제이다. 지문의 핵심 단서는 (1) 어떠한 노드라도 동일한 연산을 수행할 수 있다는 점, (2) 서버와 클라이언트의 역할 구분이 존재하지 않는다는 점, (3) 대표 사례로 파일 공유 네트워크가 제시된다는 점이다.

핵심 개념

피어 투 피어(P2P) 아키텍처는 네트워크에 참여하는 각 노드(피어)가 클라이언트이면서 동시에 서버 역할도 수행하는 구조이다. 중앙 서버 없이 피어들이 대등한 자격으로 직접 자원과 데이터를 주고받으며, 대표적인 응용이 파일 공유 네트워크이다. 마이크로서비스 아키텍처는 하나의 애플리케이션을 독립적으로 배포 가능한 작은 서비스 단위로 분할하는 구조로, 서비스 간에는 API를 통해 통신하지만 여전히 요청을 보내는 쪽과 처리하는 쪽의 역할이 구분된다. 분산 컴포넌트 아키텍처는 컴포넌트들이 네트워크상에 분산 배치되어 원격 호출로 상호작용하는 구조로, 컴포넌트 간 역할이 고정되어 있고 중앙 조정 요소가 존재할 수 있다. 서비스 지향 아키텍처(SOA)는 서비스 제공자와 서비스 요청자의 역할을 구분하고, 레지스트리 등 중개 요소를 통해 서비스를 검색·호출하는 구조이다.

NoteBase

선지별 해설

① 마이크로서비스 아키텍처 → X. 서비스를 잘게 나누어 독립 배포하는 구조이지만 서비스 호출자와 제공자의 역할이 구분되어 있어 서버-클라이언트 구분이 없다는 지문 설명과 맞지 않는다.
② 분산 컴포넌트 아키텍처 → X. 컴포넌트가 네트워크에 분산되어 있다는 점은 유사하나, 각 컴포넌트의 역할(요청자/제공자)이 고정되어 있어 '어떠한 노드라도 동일한 연산을 수행'한다는 대등성 설명과 맞지 않는다.
③ 서비스 지향 아키텍처 → X. 서비스 제공자와 요청자의 역할이 명확히 구분되고, 흔히 서비스 레지스트리와 같은 중앙 중개 요소를 두므로 비중앙집중적이라는 설명과 배치된다.
④ 피어 투 피어 아키텍처 → O. 모든 노드(피어)가 대등하게 클라이언트와 서버 역할을 동시에 수행하며 중앙 서버가 없고, 파일 공유 네트워크가 대표 사례라는 점이 지문과 정확히 일치한다.

오답 포인트

마이크로서비스나 SOA는 '분산되어 있다'는 인상 때문에 비중앙집중적 구조로 착각하기 쉽지만, 실제로는 서비스 요청자와 제공자의 역할이 뚜렷이 구분된다는 점에서 P2P와 다르다. 분산 컴포넌트 아키텍처 역시 컴포넌트가 네트워크에 흩어져 있을 뿐 노드 간 역할 대등성은 보장하지 않는다. 핵심은 '서버-클라이언트 구분이 없는가'와 '모든 노드가 동일한 연산을 수행할 수 있는가'라는 두 가지 조건이다.

다시 나오면 이렇게 판단

지문에 '서버와 클라이언트 구분이 없다', '모든 노드가 동등하다', '파일 공유' 같은 표현이 나오면 P2P 아키텍처를 먼저 떠올린다. 반대로 '서비스 제공자/요청자', '레지스트리', '작은 서비스 단위로 분할' 같은 표현이 나오면 SOA나 마이크로서비스 쪽으로 판단한다.

암기 포인트

P2P = 노드 대등(peer) + 서버/클라이언트 구분 없음 + 파일 공유 네트워크가 대표 사례. 마이크로서비스 = 기능 단위로 잘게 쪼갠 독립 서비스. SOA = 서비스 제공자-요청자 구분 + 레지스트리를 통한 검색·호출. 분산 컴포넌트 = 컴포넌트가 네트워크에 흩어져 원격 호출로 상호작용.

NoteBase

문 10. 다음 자바 코드에서 오류가 발생하는 문장은?

```
abstract class Shape {
  public void paint() { draw(); }
  abstract public void draw();
}
abstract class Circle extends Shape {
  public int radius;
  public Circle(int radius) {
    this.radius = radius; }
  double getArea() {
    return (3.14 * radius * radius); }
}
public class NamedCircle extends Circle {
  String name;
  public NamedCircle(int radius, String name) {
    super(radius);
    this.name = name;
  }
  public void draw() {
    System.out.println("반지름이 " +
                radius  + "인 원을 그리다.");
  }
  public static void main(String[] args) {
    Shape s;
    s = new Shape();
    NamedCircle w = new NamedCircle(5, "Ring");
    System.out.println(w.getArea());
    w.draw();
  }
}
```

① super(radius);

② Shape s;

③ s = new Shape();

④ System.out.println(w.getArea());

정답 3번

출제 포인트

이 문제는 자바의 추상 클래스(abstract class)가 인스턴스화될 수 없다는 규칙을 코드에서 찾아내는 문제이다. Shape, Circle이 모두 abstract로 선언되어 있고, 이 중 어떤 클래스를 직접 new로 생성하려는 문장에서 컴파일 오류가 발생하는지를 확인한다.

핵심 개념

abstract 클래스는 미완성 설계도와 같아서 자신의 객체를 직접 만들 수 없다. new 클래스명() 형태로 인스턴스를 생성하려면 그 클래스가 abstract가 아니어야 한다. Shape는 abstract class Shape로 선언되어 있으므로 new Shape()는 컴파일 오류이다. 반면 NamedCircle은 abstract 없이 public class NamedCircle extends Circle로 선언되어 있고, 부모의 추상 메서드 draw()도 재정의(override)했으므로 완전한 클래스이며 정상적으로 인스턴스화할 수 있다. 참조 변수 선언(Shape s;)이나 부모 생성자 호출(super(radius);) 자체는 인스턴스 생성이 아니므로 오류와 무관하다.

NoteBase

선지별 해설

① super(radius); → O(오류 아님). NamedCircle 생성자에서 부모 클래스 Circle의 생성자를 호출하는 문장으로, Circle(int radius) 생성자가 존재하므로 정상 동작한다.
② Shape s; → O(오류 아님). 참조 변수를 선언만 하는 문장이며, abstract 클래스라도 참조 변수 타입으로 사용하는 것은 아무 문제가 없다.
③ s = new Shape(); → X(오류 발생). Shape는 abstract class로 선언되어 있어 자신의 인스턴스를 직접 생성할 수 없다. new Shape()는 컴파일 타임 오류를 일으키는 문장이다.
④ System.out.println(w.getArea()); → O(오류 아님). w는 NamedCircle 타입 객체이고 NamedCircle은 Circle을 상속하며 getArea()를 그대로 물려받으므로 정상 호출된다.

오답 포인트

① 번과 ② 번을 오답으로 고르는 경우는 super(radius)나 참조 변수 선언 자체를 실행 가능한 객체 생성으로 착각했기 때문이다. 참조 변수 선언과 부모 생성자 호출은 인스턴스화가 아니다. ④ 번을 고르는 경우는 NamedCircle도 Circle을 상속받는 클래스이므로 혹시 추상 메서드가 남아 있어 오류가 나지 않을까 의심한 것인데, NamedCircle이 draw()를 완전히 재정의했으므로 더 이상 추상 메서드가 남아 있지 않아 정상적으로 인스턴스화되고 메서드 호출도 문제없다.

다시 나오면 이렇게 판단

코드에서 abstract로 선언된 클래스 이름을 먼저 표시해 두고, new 뒤에 그 클래스 이름이 그대로 오는 문장을 찾는다. new 뒤에 오는 클래스가 abstract가 아니고 상속받은 모든 추상 메서드를 재정의했다면 정상적인 생성문이다.

암기 포인트

abstract 클래스는 new로 직접 생성할 수 없다. 자식 클래스가 모든 추상 메서드를 구현(override)해야만 그 자식 클래스는 new로 생성 가능한 완전한 클래스가 된다.

NoteBase

문 11. 코드 인스펙션(code inspection)에 대한 설명으로 옳지 않은 것은?

① 품질 보증 활동으로 인식한다.

② 동적(dynamic) 테스트의 하나이다.

③ 회의 전에 프로그램 코드, 분석, 설계 등의 문서를 배포한다.

④ 검사 팀은 관련 전문가로 이루어지며 검사목록(checklist)을 제공한다.

정답 2번

출제 포인트

코드 인스펙션이 정적(static) 검토 기법이라는 점과 그 진행 절차(사전 배포, 회의 진행, 검사팀 구성, 체크리스트 활용)를 정확히 알고 있는지를 묻는다. 프로그램을 실제로 실행하지 않고 코드·문서를 사람이 읽고 검토하는 방식이라는 점이 핵심이다.

핵심 개념

코드 인스펙션은 소프트웨어 품질 검토(review) 기법 중 하나로, 개발자가 작성한 코드를 실행하지 않은 상태에서 동료·전문가로 구성된 검사팀이 미리 배포된 코드와 관련 문서(분석서, 설계서 등)를 검토 회의를 통해 점검하는 정적 분석(static analysis) 활동이다. 반면 동적 테스트는 프로그램을 실제로 실행시켜 입력값에 따른 출력 결과를 확인하는 기법(단위테스트, 통합테스트 등)으로, 코드 인스펙션과는 성격이 다르다.

선지별 해설

① 품질 보증 활동으로 인식한다. → O. 코드 인스펙션은 결함을 조기에 발견하여 품질을 높이기 위한 정형화된 품질 보증(SQA) 활동으로 다뤄진다.
② 동적(dynamic) 테스트의 하나이다. → X. 코드 인스펙션은 프로그램을 실행하지 않고 코드와 문서를 검토하는 정적(static) 검토 기법이다. "동적 테스트" → "정적 검토(정적 분석)"로 고쳐야 옳은 설명이 된다.
③ 회의 전에 프로그램 코드, 분석, 설계 등의 문서를 배포한다. → O. 검토 회의의 효율을 높이기 위해 참가자들에게 사전에 코드와 관련 분석·설계 문서를 배포하여 미리 검토하게 하는 것이 인스펙션의 일반적인 절차이다.
④ 검사 팀은 관련 전문가로 이루어지며 검사목록(checklist)을 제공한다. → O. 인스펙션은 사회자(진행자), 작성자, 검토자 등 관련 분야 전문가로 검사팀을 구성하며, 흔히 발생하는 결함 유형을 정리한 체크리스트를 활용하여 검토의 일관성과 효율을 높인다.

NoteBase

오답 포인트

"동적"이라는 단어만 보고 테스트 관련 기법이라 넘기지 말고, 코드 인스펙션이 프로그램을 실행하는지 여부를 먼저 따져야 한다. 코드 인스펙션·워크스루(walkthrough) 등 검토(review) 계열 기법은 모두 정적 기법이며, 실행을 통해 결함을 찾는 단위테스트·통합테스트·시스템테스트 등이 동적 기법에 해당한다는 점을 구분해서 기억해 둔다.

다시 나오면 이렇게 판단

선지에 "동적 테스트", "실행하여 확인" 같은 표현이 나오면 그 기법이 실제로 프로그램을 구동하는지부터 확인한다. 코드 인스펙션, 워크스루, 데스크 체킹 등은 실행 없이 검토만 하므로 전부 정적 기법으로 분류된다는 점을 기준으로 삼는다.

암기 포인트

정적 기법(실행 X): 코드 인스펙션, 워크스루, 데스크 체킹. 동적 기법(실행 O): 단위테스트, 통합테스트, 시스템테스트, 인수테스트. 코드 인스펙션 절차 요약: 사전 문서 배포 → 검토 회의 진행 → 체크리스트 기반 결함 식별 → 품질 보증 활동으로 관리된다.

NoteBase

문 12. 다음 설명과 용어를 바르게 연결한 것은?

(가) 코드만 남아있고 관련 문서가 없을 때, 프로그램을 분석하여 프로그램 구조, 자료 구조, 모듈 사이의 관계 등 상세한 설계 정보를 추출함
(나) 겉으로 보이는 동작이나 외부 행위를 바꾸지 않고 소프트웨어 내부 구조를 바꾸며 점진적으로 설계를 향상시키는 기법
(다) 기존 시스템을 이해하여 새로운 기능을 추가하거나 성능을 향상시키는 등의 새로운 형태로 변경하는 것으로, 단순히 설계정보를 끌어내는 것뿐만 아니라 이 정보를 사용하여 보다 나은 시스템을 만드는 것

① (가): 역공학, (나): 리팩토링, (다): 재공학
② (가): 역공학, (나): 프로토타이핑, (다): 재공학
③ (가): 재공학, (나): 리팩토링, (다): 역공학
④ (가): 재공학, (나): 프로토타이핑, (다): 역공학

정답 1번

출제 포인트

소프트웨어 재공학(software re-engineering) 범주에 속하는 세부 활동인 역공학, 리팩토링, 재공학의 정의를 서로 구분할 수 있는지 묻는 문제이다. 세 용어 모두 기존 소프트웨어를 대상으로 한다는 공통점이 있지만, 무엇을 입력으로 삼고 무엇을 산출물로 내는지가 서로 다르다.

핵심 개념

역공학(reverse engineering)은 소스 코드만 남아있고 설계 문서가 없을 때, 코드를 분석하여 프로그램 구조·자료 구조·모듈 관계 등 상위 수준의 설계 정보를 역으로 추출해 내는 활동이다. 리팩토링(refactoring)은 외부에서 관찰되는 동작(기능)은 그대로 유지한 채 내부 구조만 개선하여 점진적으로 품질을 높이는 기법이다. 재공학(re-engineering)은 역공학으로 추출한 설계 정보를 단순히 얻는 데서 그치지 않고, 그 정보를 활용해 기능을 추가하거나 성능을 향상시켜 기존 시스템을 더 나은 새로운 형태로 변경하는 활동이다. 즉 역공학이 재공학의 한 단계(분석 단계)이고, 재공학은 역공학의 결과물을 바탕으로 순공학(forward engineering)까지 이어지는 더 큰 개념이다. 참고로 프로토타이핑(prototyping)은 요구사항이 불명확할 때 시제품을 빠르게 만들어 사용자 피드백을 받는 개발 기법으로, 기존 코드를 분석·개선하는 활동과는 성격이 다르다.

NoteBase

선지별 해설

① (가) 역공학, (나) 리팩토링, (다) 재공학 → (가)는 코드만 남은 상태에서 설계 정보를 추출하는 역공학의 정의와 일치하고, (나)는 외부 동작을 바꾸지 않고 내부 구조만 점진적으로 개선한다는 점에서 리팩토링의 정의와 일치하며, (다)는 설계 정보 추출에 그치지 않고 이를 이용해 기능 추가·성능 향상까지 이루는 재공학의 정의와 정확히 일치한다. 세 설명 모두 옳게 연결되었다.
② (나)를 프로토타이핑으로 연결 → 프로토타이핑은 요구사항을 구체화하기 위해 시제품을 만드는 기법이지, 외부 동작을 유지하면서 내부 구조를 점진적으로 개선하는 활동이 아니므로 (나)의 설명과 맞지 않는다.
③ (가)를 재공학, (다)를 역공학으로 연결 → (가)의 설명(코드 분석을 통한 설계 정보 추출)은 역공학의 정의이지 재공학의 정의가 아니며, (다)의 설명(설계 정보를 활용한 개선·변경)은 재공학의 정의이지 역공학의 정의가 아니므로 (가)와 (다)가 서로 뒤바뀌었다.
④ (가)-재공학, (나)-프로토타이핑, (다)-역공학 → (가)와 (다)가 뒤바뀐 것에 더해 (나)까지 프로토타이핑으로 잘못 연결되어 세 항목 모두 오류이다.

오답 포인트

역공학과 재공학을 혼동하기 쉬운데, 역공학은 '분석해서 설계 정보를 뽑아내는 것'까지이고 재공학은 그 정보를 '활용해서 더 나은 시스템으로 바꾸는 것'까지 포함한다는 범위 차이로 구분해야 한다. 또한 리팩토링과 프로토타이핑은 둘 다 소프트웨어를 점진적으로 다듬는다는 인상을 줄 수 있지만, 리팩토링은 기존 시스템의 겉모습(기능)은 그대로 두고 내부만 개선하는 것이고 프로토타이핑은 요구사항 확정을 위한 시제품 제작이라는 점에서 목적 자체가 다르다.

다시 나오면 이렇게 판단

지문에 '문서가 없어 코드만 보고 설계 정보를 뽑아낸다'는 표현이 나오면 역공학, '외부 동작(기능)은 그대로인데 내부 구조만 바뀐다'는 표현이 나오면 리팩토링, '설계 정보를 뽑아내는 데 그치지 않고 기능 추가·성능 향상까지 이어진다'는 표현이 나오면 재공학으로 즉시 매칭한다.

암기 포인트

역공학=코드→설계정보 추출(분석만), 리팩토링=외부동작 유지+내부구조 개선,
재공학=역공학+개선·변경(활용까지), 프로토타이핑=요구사항 확정용 시제품 제작으로 성격이 다름.

NoteBase

문 13. UML 다이어그램에 대한 설명으로 옳지 않은 것은?

① 상태 다이어그램(state diagram)은 객체의 상태 변화를 나타낸다.

② 클래스 다이어그램(class diagram)은 클래스와 클래스 사이의 관계를 나타내는 정적인 모델이다.

③ 컴포넌트 다이어그램(component diagram)은 물리적인 노드와 커뮤니케이션 경로를 각각 육면체와 선으로 표시한다.

④ 활동 다이어그램(activity diagram)은 업무의 흐름을 표현하거나 유스케이스의 구체적인 흐름을 나타내기 위해 사용한다.

정답 3번

출제 포인트

UML 다이어그램의 각 종류가 무엇을 표현하는 모델인지, 그리고 정적 모델과 동적 모델을 구분해 알고 있는지를 묻는다. 특히 구조(정적) 다이어그램인 컴포넌트 다이어그램과 배치 다이어그램의 표현 대상을 서로 바꿔치기해 혼동을 유도하는 문제이다.

핵심 개념

UML 다이어그램은 크게 구조 다이어그램과 행위 다이어그램으로 나뉜다. 구조 다이어그램에는 클래스 다이어그램, 컴포넌트 다이어그램, 배치(deployment) 다이어그램 등이 있고, 행위 다이어그램에는 상태 다이어그램, 활동 다이어그램, 유스케이스 다이어그램 등이 있다. 컴포넌트 다이어그램은 소프트웨어 컴포넌트들 사이의 구조와 의존 관계, 인터페이스를 표현하는 다이어그램으로 컴포넌트를 사각형(또는 롤리팝 표기)으로 나타낸다. 반면 배치 다이어그램은 하드웨어의 물리적인 노드(서버, 장비 등)를 육면체(3차원 상자)로, 그 사이의 통신 경로를 선으로 나타내어 시스템의 실행 환경 구성을 보여 준다.

선지별 해설

① O - 상태 다이어그램은 하나의 객체가 생성부터 소멸까지 이벤트에 따라 상태가 어떻게 변화하는지를 나타내는 행위 다이어그램이므로 옳은 설명이다.
② O - 클래스 다이어그램은 클래스들과 그 사이의 관계(연관, 집합, 상속 등)를 나타내는 대표적인 정적(구조) 모델이므로 옳은 설명이다.
③ X - 물리적인 노드와 그 사이의 커뮤니케이션 경로를 육면체와 선으로 표시하는 것은 컴포넌트 다이어그램이 아니라 배치 다이어그램의 특징이다. 컴포넌트 다이어그램은 소프트웨어 컴포넌트 간의 구성과 의존 관계를 표현하는 다이어그램으로, 육면체로 물리적 노드를 표시하지 않는다. → 물리적인 노드와 커뮤니케이션 경로를 육면체와 선으로 표시하는 것은 배치 다이어그램이다.
④ O - 활동 다이어그램은 업무 처리 과정(업무 흐름)이나 유스케이스 내부의 구체적인 처리 흐름을 순서도와 유사한 형태로 표현하는 데 사용되므로 옳은 설명이다.

NoteBase

오답 포인트

컴포넌트 다이어그램과 배치 다이어그램을 혼동하기 쉽다. 컴포넌트 다이어그램은 '소프트웨어 구성 요소(컴포넌트) 간의 관계'를 다루고, 배치 다이어그램은 '하드웨어 노드에 소프트웨어 컴포넌트가 어떻게 배치되는지, 물리적 실행 환경'을 다룬다는 점을 구분해야 한다. 육면체(3차원 상자) 표기는 배치 다이어그램의 노드 표기라는 점을 기억해 둔다.

다시 나오면 이렇게 판단

선지에서 '물리적인 노드', '육면체', '실행 환경', '하드웨어'라는 표현이 나오면 배치 다이어그램을 가리키는 것이고, '컴포넌트', '인터페이스', '의존 관계'라는 표현이 나오면 컴포넌트 다이어그램을 가리키는 것으로 구분해 판단한다.

암기 포인트

배치 다이어그램 = 물리적 노드(육면체) + 통신 경로(선), 컴포넌트 다이어그램 = 소프트웨어 컴포넌트 + 인터페이스/의존 관계로 짝지어 외워 둔다.

NoteBase

문 14. 바람직한 설계에 대한 설명으로 옳지 않은 것은?

① 모듈 간의 결합도를 강하게 유지한다.

② 유지관리를 잘할 수 있도록 설계한다.

③ 분할과 정복의 원리를 적용하여 복잡도를 감소시킨다.

④ 변경하기 쉽고 변경으로 인한 영향도 최소화되도록 한다.

정답 1번

출제 포인트

바람직한 소프트웨어 설계가 갖추어야 할 기본 원칙(낮은 결합도, 높은 응집도, 분할과 정복, 유지보수 용이성)을 정확히 알고 있는지를 묻는 문제이다. 결합도와 응집도의 방향을 반대로 뒤집어 놓은 선지가 정답이 되는 전형적인 구성이다.

핵심 개념

결합도(coupling)는 모듈 간의 상호 의존 정도를 뜻하며, 바람직한 설계는 결합도를 낮게(약하게) 유지하여 모듈 간 의존성을 최소화한다. 응집도(cohesion)는 모듈 내부 요소들의 관련성을 뜻하며, 바람직한 설계는 응집도를 높게 유지한다. 분할과 정복(divide and conquer)은 큰 문제를 작은 모듈로 나누어 각 모듈의 복잡도를 낮추는 설계 원리이다. 좋은 설계는 유지보수가 쉽고, 변경이 발생했을 때 그 영향이 다른 모듈로 퍼지지 않도록(파급 효과 최소화) 만든다.

선지별 해설

① 모듈 간의 결합도를 강하게 유지한다. → X. 결합도는 강하게가 아니라 약하게(낮게) 유지하는 것이 바람직한 설계이다. 결합도가 강하면 한 모듈의 변경이 다른 모듈에 미치는 영향이 커져 유지보수가 어려워진다. 틀린 표현 → 올바른 표현: '결합도를 강하게 유지한다' → '결합도를 약하게(낮게) 유지한다'.
② 유지관리를 잘할 수 있도록 설계한다. → O. 유지보수성은 좋은 설계가 갖추어야 할 대표적인 품질 목표이다.
③ 분할과 정복의 원리를 적용하여 복잡도를 감소시킨다. → O. 큰 시스템을 여러 모듈로 나누어 각 모듈의 복잡도를 낮추는 것은 바람직한 설계의 기본 전략이다.
④ 변경하기 쉽고 변경으로 인한 영향도 최소화되도록 한다. → O. 설계 변경의 용이성과 파급 효과 최소화는 좋은 설계의 목표에 해당한다.

오답 포인트

결합도와 응집도를 헷갈리면 이 문제를 틀리게 된다. '강하게/약하게'라는 표현만 보고 무조건 좋다고 판단하지 말고, 결합도는 낮을수록, 응집도는 높을수록 좋은 설계라는 점을 짝지어 기억해야 한다.

NoteBase

다시 나오면 이렇게 판단

선지에 결합도·응집도가 나오면 먼저 어느 쪽을 말하는지 확인하고, 결합도는 '약하게/낮게', 응집도는 '강하게/높게'라는 방향이 맞는지부터 점검한다. 방향이 반대로 되어 있으면 바로 오답으로 판단한다.

암기 포인트

결합도는 낮게, 응집도는 높게. 좋은 설계 = 낮은 결합도 + 높은 응집도 + 분할과 정복 + 쉬운 유지보수·변경.

NoteBase

문 15. 요구사항 명세 기법 중 자연어 기반 비정형 명세에 대한 설명으로 옳은 것만을 모두 고르면?

ㄱ. 표현이 모호할 수 있고, 해석이 다를 수 있다.
ㄴ. 수학적 증명 기술을 이용하여 완전성을 검증할 수 있다.
ㄷ. 자연어를 기반으로 작성하기 때문에 쉽게 작성할 수 있다.

① ㄷ
② ㄱ, ㄴ
③ ㄱ, ㄷ
④ ㄴ, ㄷ

정답 3번

출제 포인트

요구사항 명세 기법은 크게 자연어 기반의 비정형 명세와 Z 명세, VDM 등 수학적 표기법에 기반한 정형 명세로 나뉜다. 이 문항은 비정형 명세의 특징(장점과 한계)을 정확히 구분해 낼 수 있는지를 묻는다.

핵심 개념

비정형(informal) 명세는 자연어 문장으로 요구사항을 서술하는 방식으로, 작성자와 독자 모두 별도의 표기법 학습 없이 쉽게 작성하고 읽을 수 있다는 장점이 있다. 반면 자연어 자체가 다의적이고 모호한 표현을 허용하므로 읽는 사람에 따라 해석이 달라질 수 있고, 문장으로는 논리적 완전성이나 일관성을 수학적으로 증명할 방법이 없다. 이와 달리 정형(formal) 명세는 수학적 표기와 형식논리를 사용하므로 정리·증명 기법으로 완전성과 무모순성을 검증할 수 있지만, 작성과 이해에 전문성이 필요해 작성이 어렵다.

선지별 해설

ㄱ. 표현이 모호할 수 있고, 해석이 다를 수 있다 → O. 자연어는 문장 구조상 중의적 표현이 발생할 수 있어 읽는 사람마다 다르게 해석할 위험이 있다. 이는 비정형 명세의 대표적인 단점이다.
ㄴ. 수학적 증명 기술을 이용하여 완전성을 검증할 수 있다 → X. 이는 정형 명세의 특징이다. 자연어 문장은 수학적 대상이 아니므로 증명 기법을 적용해 완전성이나 정확성을 검증할 수 없다.
ㄷ. 자연어를 기반으로 작성하기 때문에 쉽게 작성할 수 있다 → O. 별도의 표기법이나 기호 체계를 배우지 않아도 되므로 작성자 입장에서 접근이 쉽다는 것이 비정형 명세의 대표적인 장점이다.

오답 포인트

'수학적 증명으로 완전성을 검증한다'는 서술을 보면 정형 명세와 비정형 명세를 헷갈리기 쉽다. 비정형 명세는 검증 도구가 아니라 자연어 문장 자체이므로 증명 대상이 될 수 없다는 점을 기억해야 한다. ㄴ을 포함한 ②, ④는 이 때문에 오답이다.

NoteBase

다시 나오면 이렇게 판단

명세 기법 문제에서는 먼저 '자연어 기반인가, 수학적 표기 기반인가'를 구분한다. 자연어 기반(비정형)은 작성이 쉽지만 모호함이 남고, 수학적 표기 기반(정형)은 작성은 어렵지만 증명을 통한 검증이 가능하다는 대응 관계로 정리하면 선지 판단이 빨라진다.

암기 포인트

비정형 명세=자연어 → 작성 쉬움·해석 모호함·증명 불가. 정형 명세=수학적 표기 → 작성 어려움·표현 명확·증명(완전성 검증) 가능.

NoteBase

문 16. 다음 설명에 해당하는 코드 악취(code smell)는?

수정할 때마다 관련된 여러 클래스에서 수많은 자잘한 변경을 해야 하는 상황이 발생하여 여러 클래스를 동시에 수정해야 하는 상황

① 과잉 중개 메서드(middle man)
② 기능의 산재(shotgun surgery)
③ 메시지 체인(message chains)
④ 직무 유기 클래스(lazy class)

정답 2번

출제 포인트

리팩터링에서 다루는 대표적인 코드 악취(code smell)들의 정의를 구분할 수 있는지 묻는 문제이다. 지문은 '하나를 고치려면 관련된 여러 클래스에 걸쳐 자잘한 수정이 동시에 퍼져야 하는 상황'을 설명하고 있으며, 이는 하나의 책임(기능)이 여러 클래스로 흩어져 있을 때 나타나는 전형적인 증상이다.

핵심 개념

기능의 산재(shotgun surgery)는 한 가지 변경 요구가 발생했을 때 이를 반영하기 위해 여러 클래스에 조금씩 손을 대야 하는 악취로, 하나의 책임이 여러 클래스에 나뉘어 있어 응집도는 낮고 결합도는 높은 상태에서 생긴다. 과잉 중개 메서드(middle man)는 한 클래스가 자신의 일을 거의 다른 클래스에 위임만 하고 실제로는 별다른 일을 하지 않는 경우를 말한다. 메시지 체인(message chains)은 한 객체가 다른 객체를 얻고, 그 객체에서 또 다른 객체를 얻는 식으로 얻어오기 호출이 길게 연쇄되는 경우를 말한다. 직무 유기 클래스(lazy class)는 하는 일이 거의 없어서 더 이상 자신의 존재 값을 못 하는 클래스를 말한다.

선지별 해설

① 과잉 중개 메서드(middle man): X. 이는 한 클래스가 실제 작업 없이 다른 클래스에 위임만 반복하는 상황을 가리키며, 지문의 '여러 클래스를 동시에 수정해야 하는 상황'과는 무관하다.
② 기능의 산재(shotgun surgery): O. 하나의 변경이 여러 클래스에 걸친 자잘한 수정을 동시에 요구하는 상황을 정확히 가리키는 악취이므로 지문과 일치한다.
③ 메시지 체인(message chains): X. 이는 객체를 연쇄적으로 타고 들어가 호출하는 구조상의 문제이며, 수정이 여러 클래스로 퍼지는 상황을 설명하는 것이 아니다.
④ 직무 유기 클래스(lazy class): X. 이는 하는 일이 거의 없어 존재 의의가 희박한 클래스를 가리키는 것으로, 수정 시 여러 클래스가 동시에 바뀌어야 한다는 지문 내용과는 관련이 없다.

NoteBase

오답 포인트

과잉 중개 메서드와 메시지 체인은 모두 '클래스 간 관계'와 관련된 악취라서 기능의 산재와 혼동하기 쉽지만, 전자들은 위임·연쇄 호출 구조 자체가 문제인 반면 기능의 산재는 하나의 변경이 여러 클래스에 흩어져 반영되어야 한다는 '수정의 파급 범위'가 핵심이다. 지문에서 '수정할 때마다', '여러 클래스를 동시에 수정'이라는 표현이 나오면 기능의 산재를 먼저 떠올려야 한다.

다시 나오면 이렇게 판단

지문에서 변경(수정) 하나가 여러 클래스로 퍼져 나간다는 서술이 나오면 곧바로 기능의 산재로 판단한다. 반대로 한 클래스가 실질적 로직 없이 위임만 한다는 서술이면 과잉 중개 메서드, 객체를 연쇄적으로 얻어 오는 호출 체인이 문제라면 메시지 체인, 클래스가 거의 아무 일도 하지 않는다는 서술이면 직무 유기 클래스로 구분한다.

암기 포인트

기능의 산재(shotgun surgery) = 한 번의 수정이 여러 클래스로 산탄총처럼 흩어져 퍼진다는 이미지로 기억해 둔다.

NoteBase

문 17. 다음 소스 코드와 테스트 입력을 사용한 테스트 커버리지 설명으로 옳지 않은 것은? (단, 단축연산(short-circuit evaluation)은 수행하지 않는다)

문장 번호	소스 코드	테스트 입력
	void temp(int x, int y) {	
1	if(x < 0 && y >= 0)	
2	printf("%d",y);	T = {t1:(x=-2, y=3),
3	if(x >= 0)	t2:(x=2, y=0)}
4	printf("%d",x);	
	}	

① t2만 사용하면 문장(statement) 커버리지가 75%이다.

② T를 사용하면 문장 커버리지가 100%이다.

③ T를 사용하면 분기(branch) 커버리지가 100%이다.

④ T를 사용하면 조건(condition) 커버리지가 100%이다.

정답 4번

출제 포인트

테스트 케이스 집합이 주어졌을 때 문장 커버리지, 분기 커버리지, 조건 커버리지를 각각 계산할 수 있는지를 묻는 문제이다. 단축연산을 수행하지 않는다는 단서가 핵심인데, 이 경우 if문의 두 조건(x < 0, y >= 0)이 결합 연산자와 무관하게 항상 둘 다 개별적으로 평가되므로, 조건 커버리지는 최종 논리식의 참/거짓이 아니라 각 개별 조건이 참과 거짓을 모두 가졌는지로 따져야 한다.

NoteBase

조건 정리

코드는 다음과 같다. 문장1: if(x < 0 && y >= 0), 문장2: printf("%d", y), 문장3: if(x >= 0), 문장4: printf("%d", x). 테스트 케이스는 t1:(x=-2, y=3), t2:(x=2, y=0)이다.
t1 실행 흐름: x < 0은 참(-2<0), y >= 0은 참(3>=0)이므로 문장1의 논리식 결과는 참 → 문장2 실행. 문장3: x >= 0은 거짓(-2>=0 아님) → 문장4 미실행.
t2 실행 흐름: x < 0은 거짓(2<0 아님), y >= 0은 참(0>=0)이므로 문장1의 논리식 결과는 거짓(단축연산 없어도 &&는 하나라도 거짓이면 전체 거짓) → 문장2 미실행. 문장3: x >= 0은 참(2>=0) → 문장4 실행.

사용할 공식

문장 커버리지 = (실행된 문장 수) / (전체 문장 수) × 100. 분기 커버리지 = (실행된 분기(참/거짓) 수) / (전체 분기 수) × 100, 여기서 분기는 각 if문의 참 경로와 거짓 경로 각각을 의미한다. 조건 커버리지 = 개별 원자 조건 각각이 참과 거짓 값을 적어도 한 번씩 가졌는지를 확인하는 기준이며, and/or로 묶인 전체 논리식의 참/거짓과는 별개로 각 원자 조건 단위로 판단한다.

계산 과정

전체 문장은 4개(문장1, 문장2, 문장3, 문장4)이다.
t2만 사용: t2는 문장1, 문장3, 문장4를 실행하고 문장2는 실행하지 않는다. 실행된 문장 수 3개이므로 문장 커버리지 = 3/4 × 100 = 75%이다. → ①은 옳다.
T(t1+t2) 사용: t1이 문장1, 문장2, 문장3을 실행하고 t2가 문장1, 문장3, 문장4를 실행하므로 합치면 문장1~4가 모두 실행된다. 문장 커버리지 = 4/4 × 100 = 100%이다. → ②는 옳다.
분기 계산: 문장1의 분기는 참(t1이 담당)과 거짓(t2가 담당)이 모두 나타나고, 문장3의 분기도 거짓(t1: x=-2로 x>=0 거짓)과 참(t2: x=2로 x>=0 참)이 모두 나타난다. 두 if문의 참/거짓 분기가 T로 전부 실행되므로 분기 커버리지 = 100%이다. → ③은 옳다.
조건 계산: 원자 조건은 x < 0, y >= 0(문장1 내부), x >= 0(문장3)이다. x < 0은 t1에서 참, t2에서 거짓으로 참/거짓이 모두 나타난다. x >= 0(문장3)도 t1에서 거짓, t2에서 참으로 모두 나타난다. 그런데 y >= 0은 t1(y=3)에서 참, t2(y=0)에서도 참이므로 T 전체에서 단 한 번도 거짓 값을 갖지 못한다. 따라서 조건 커버리지는 100%가 아니다. → ④는 틀렸다.

선지 판단

① t2만 사용하면 문장 커버리지가 75%이다: O. t2는 문장1, 문장3, 문장4 3개를 실행하여 4개 중 3개, 75%이다.
② T를 사용하면 문장 커버리지가 100%이다: O. t1과 t2를 합치면 문장1~4가 모두 실행된다.
③ T를 사용하면 분기 커버리지가 100%이다: O. 문장1과 문장3의 참/거짓 분기가 t1, t2로 모두 실행된다.
④ T를 사용하면 조건 커버리지가 100%이다: X → 실제로는 100%가 아니다. y >= 0이라는 개별 조건이 t1, t2 어느 경우에도 거짓 값을 갖지 못하기 때문이다(항상 참으로만 평가됨).

NoteBase

오답 포인트

조건 커버리지를 분기 커버리지와 혼동하기 쉽다. 분기 커버리지는 if문 전체 논리식의 참/거짓 결과만 양쪽 다 나오면 충족되지만, 조건 커버리지는 그 논리식을 구성하는 개별 원자 조건 하나하나가 각각 참과 거짓 값을 모두 가져야 충족된다. 문장1의 논리식 (x<0 && y>=0)은 t1에서 참, t2에서 거짓으로 양쪽이 다 나와 분기는 만족되지만, 그 내부의 y>=0이라는 조건 자체는 t1, t2 모두에서 참으로만 평가되어 거짓 값이 한 번도 등장하지 않는다. 단축연산을 하지 않는다는 조건 때문에 && 연산자의 오른쪽 항인 y>=0도 매 테스트마다 반드시 평가되지만, 그 평가값이 우연히 항상 참으로만 나온 것이다.

다시 나오면 이렇게 판단

커버리지 문제를 보면 먼저 전체 문장 수를 센 다음, 각 테스트 케이스가 실제로 어느 문장을 실행하는지 흐름을 그려본다. 그다음 조건이 여러 개 결합된 if문이 있으면, 논리식 전체의 참/거짓(분기)과 그 내부 개별 조건 각각의 참/거짓(조건)을 분리해서 표로 확인한다. 단축연산을 하지 않는다는 단서가 있으면 결합된 모든 개별 조건이 매번 실제로 평가된다는 뜻이므로, 그 각 조건이 참/거짓을 모두 거쳤는지를 꼼꼼히 따져야 한다.

암기 포인트

문장 커버리지는 실행된 문장의 비율, 분기 커버리지는 각 분기(if의 참/거짓 경로)가 모두 실행되었는지, 조건 커버리지는 논리식을 구성하는 개별 원자 조건 하나하나가 참과 거짓을 모두 가졌는지로 구분해 기억해 둔다. 분기 커버리지 100%가 곧 조건 커버리지 100%를 의미하지 않는다는 점이 이 문제의 핵심이다.

NoteBase

문 18. UML 시퀀스 다이어그램에서 사용되지 않는 것은?

① 구획면(swimlane)

② 메시지(message)

③ 생명선(lifeline)

④ 활성화(activation)

정답 1번

출제 포인트

UML 시퀀스 다이어그램을 구성하는 표기 요소와, 그중 시퀀스 다이어그램이 아닌 다른 다이어그램에서 쓰이는 요소를 구별할 수 있는지 묻는다. 시퀀스 다이어그램은 객체들 사이의 메시지 교환을 시간 흐름에 따라 표현하는 상호작용 다이어그램이고, 구획면(swimlane)은 액티비티 다이어그램에서 작업의 책임 주체(행위자, 부서 등)를 구분하기 위해 사용하는 요소로 서로 다른 다이어그램에 속한다.

핵심 개념

생명선(lifeline)은 시퀀스 다이어그램에서 객체 하나를 나타내는 세로 점선으로, 그 객체가 존재하는 시간 축을 표현한다. 활성화(activation)는 생명선 위에 놓이는 얇은 사각형 막대로, 해당 객체가 특정 동작을 실제로 수행하고 있는 구간(제어를 갖고 있는 기간)을 나타낸다. 메시지(message)는 한 객체의 생명선에서 다른 객체의 생명선으로 이어지는 화살표로, 객체 간에 주고받는 함수 호출·응답 등을 표현한다. 이 세 요소는 모두 시퀀스 다이어그램의 기본 표기법이다. 반면 구획면(swimlane)은 여러 행위자나 처리 단위가 수행하는 작업 흐름을 열(또는 행) 단위로 나누어 보여 주는 표기로, 액티비티 다이어그램에서 업무 프로세스의 담당 주체를 구분할 때 사용되며 시퀀스 다이어그램에는 존재하지 않는 개념이다.

선지별 해설

① 구획면(swimlane): X, 시퀀스 다이어그램의 구성 요소가 아니다. 여러 담당 주체별로 처리 흐름을 나누어 보여 주는 액티비티 다이어그램의 표기 요소이므로 정답이다.
② 메시지(message): O, 시퀀스 다이어그램에서 객체 간의 함수 호출·응답을 화살표로 표현하는 핵심 요소이다.
③ 생명선(lifeline): O, 시퀀스 다이어그램에서 각 객체의 존재 기간을 나타내는 세로 점선으로, 반드시 있어야 하는 기본 요소이다.
④ 활성화(activation): O, 생명선 위에서 객체가 실제로 동작(제어)을 수행하는 구간을 표시하는 막대 표기로 시퀀스 다이어그램의 기본 요소이다.

NoteBase

오답 포인트

구획면(swimlane)이라는 용어를 액티비티 다이어그램이 아니라 시퀀스 다이어그램의 요소로 착각하기 쉽다. 시퀀스 다이어그램은 '누가(담당 주체)'보다 '어떤 순서로 메시지를 주고받는가'에 초점을 두므로 담당 주체를 나열해 구분하는 구획면 개념 자체가 필요하지 않다는 점을 기억해 둔다.

다시 나오면 이렇게 판단

선지에 생명선·활성화·메시지·프레임(콤비네이션 프래그먼트) 등 시퀀스 다이어그램 고유 표기가 섞여 나오면 그중 액티비티 다이어그램 전용 요소(구획면, 분기/병합, 시작·종료 노드 등)를 찾아 소거하는 방식으로 접근한다. 다이어그램별로 어떤 요소가 전용인지 짝지어 암기해 둔다.

암기 포인트

시퀀스 다이어그램=생명선·메시지·활성화(+실행 프레임). 액티비티 다이어그램=구획면(swimlane)·분기·병합·시작/종료 노드.

NoteBase

문 19. 다음 테스트 주도 개발(TDD)의 단계를 순서대로 바르게 나열한 것은?

(가) 무엇을 테스트할 것인지 생각한다.
(나) 테스트를 통과하도록 소스 코드를 작성한다.
(다) 테스트 코드를 작성한다.
(라) 소스 코드와 테스트 코드를 리팩토링한다.
(마) 구현해야 할 소스 코드가 남아 있으면 위의 단계를 반복한다.

① (가) - (나) - (다) - (라) - (마)
② (가) - (다) - (나) - (라) - (마)
③ (나) - (가) - (다) - (라) - (마)
④ (나) - (가) - (라) - (다) - (마)

정답 2번

출제 포인트

테스트 주도 개발(TDD, Test-Driven Development)의 표준 사이클인 Red-Green-Refactor 순서를 단계 설명만 보고 재구성할 수 있는지를 묻는다. TDD는 실제 구현 코드보다 테스트 코드를 먼저 작성하고, 그 테스트를 통과시키는 최소한의 코드를 만든 뒤 정리하는 반복 과정이다.

올바른 순서

1단계(가): 무엇을 테스트할 것인지 생각한다 — 요구사항을 검증 가능한 테스트 대상으로 구체화한다.
2단계(다): 테스트 코드를 작성한다 — 아직 통과하지 못하는(실패하는) 테스트를 먼저 만든다(Red).
3단계(나): 테스트를 통과하도록 소스 코드를 작성한다 — 테스트를 통과시키는 최소한의 구현을 추가한다(Green).
4단계(라): 소스 코드와 테스트 코드를 리팩토링한다 — 동작은 그대로 유지한 채 코드 구조를 개선한다(Refactor).
5단계(마): 구현해야 할 소스 코드가 남아 있으면 위 단계를 반복한다 — 남은 기능이 없어질 때까지 1~4단계를 순환한다.
따라서 올바른 순서는 (가) - (다) - (나) - (라) - (마)이다.

NoteBase

선지별 해설

① (가)-(나)-(다)-(라)-(마): X. 테스트 코드 작성(다)보다 소스 코드 작성(나)을 먼저 두어 순서가 뒤바뀌었다. TDD는 테스트를 먼저 작성한 뒤 그것을 통과시키는 코드를 작성하므로 (다)가 (나)보다 앞서야 한다.
② (가)-(다)-(나)-(라)-(마): O. 테스트 대상을 정하고(가), 실패하는 테스트를 먼저 작성한 뒤(다), 이를 통과시키는 코드를 작성하고(나), 리팩토링하며(라), 남은 작업이 있으면 반복한다(마)는 TDD의 기본 사이클과 정확히 일치한다.
③ (나)-(가)-(다)-(라)-(마): X. 무엇을 테스트할지 생각하기(가)도 전에 소스 코드부터 작성(나)하는 것은 TDD의 순서에 어긋난다. 테스트 대상 구상이 항상 가장 먼저 와야 한다.
④ (나)-(가)-(라)-(다)-(마): X. 소스 코드 작성(나)이 맨 앞에 오고 리팩토링(라)이 테스트 코드 작성(다)보다 먼저 오는 등 TDD의 순서와 전혀 맞지 않는다.

오답 포인트

이 문항의 함정은 '테스트 코드 작성'과 '소스 코드 작성'의 선후 관계를 헷갈리게 만드는 데 있다. TDD의 핵심은 '테스트가 실패하는 것을 먼저 확인한 뒤 구현한다'는 것이므로, 소스 코드(나)가 테스트 코드(다)보다 앞서 나오는 선지는 모두 TDD 원칙에 위배된다. 또한 '무엇을 테스트할지 생각하는 단계(가)'는 실제 테스트 코드를 작성하기 전에 반드시 선행되어야 하는 준비 단계임을 놓치지 않아야 한다.

다시 나오면 이렇게 판단

TDD 순서 문제는 Red-Green-Refactor라는 세 단어를 기준으로 삼는다. 실패하는 테스트 작성(Red) → 테스트를 통과시키는 최소 구현(Green) → 코드 개선(Refactor) → 남은 기능이 있으면 반복, 이 네 마디만 기억하면 선지에 나온 문장을 빠르게 각 단계에 대응시켜 순서를 판별할 수 있다.

암기 포인트

TDD 사이클: 테스트 대상 구상 → 테스트 코드 작성(Red) → 통과시키는 소스 코드 작성(Green) → 리팩토링(Refactor) → 남은 기능 있으면 반복. 줄여서 '생각-테스트-구현-정리-반복' 순으로 외워 둔다.

NoteBase

문 20. 다음 상황에 적합한 디자인 패턴은?

슈팅 모바일 게임을 개발하는데, 이 게임은 사용자가 화면을 터치하면 게임 캐릭터가 사용자가 터치한 곳으로 무기를 발사해 적을 공격하는 기능이 있다. 이는 세부적으로 '사용자 터치 인식→터치 한 방향으로 캐릭터를 회전→무기를 겨누는 동작 실행→무기 발사'순서로 실행되는 단계를 가지며, 각 단계별 동작은 별도 클래스의 메서드로 구현된다. 따라서 캐릭터 무기 발사를 구현하기 위해서는 클라이언트 코드가 각각의 클래스와 내부 메서드 사용 방법을 알아야 한다. 이 문제를 해결하기 위해 클라이언트 코드에게는 위의 복잡한 메서드 호출 과정을 숨기고, 위 과정을 대신 수행하는 높은 레벨의 인터페이스를 제공하고자 한다.

① 어댑터 패턴

② 추상 팩토리 패턴

③ 컴포지트 패턴

④ 퍼사드 패턴

정답 4번

출제 포인트

여러 클래스의 메서드를 순서대로 호출해야 하는 복잡한 서브시스템을, 클라이언트가 몰라도 되도록 하나의 높은 레벨 인터페이스로 감싸는 디자인 패턴을 구별하는 문제이다. 지문의 핵심은 '터치 인식 → 캐릭터 회전 → 무기 조준 → 무기 발사'라는 여러 클래스에 걸친 호출 절차를 클라이언트가 몰라도 되게 단순화한다는 점이며, 이는 구조 패턴 중 퍼사드(Facade) 패턴의 정의와 정확히 일치한다.

핵심 개념

퍼사드 패턴은 여러 서브시스템 클래스들의 복잡한 상호작용과 호출 순서를 캡슐화하여, 클라이언트에게 단순하고 통일된 하나의 인터페이스만 제공하는 구조 패턴이다. 클라이언트는 퍼사드 객체의 메서드 하나만 호출하면 되고, 내부에서 어떤 클래스의 어떤 메서드가 어떤 순서로 호출되는지는 알 필요가 없다. 어댑터 패턴은 서로 호환되지 않는 인터페이스를 변환해 연결하는 것이 목적이고, 추상 팩토리 패턴은 관련된 객체 계열을 구체 클래스를 몰라도 생성하게 해주는 생성 패턴이며, 컴포지트 패턴은 개별 객체와 객체의 집합을 동일하게 다루기 위해 트리 구조로 구성하는 패턴이다. 셋 다 지문의 '복잡한 호출 과정을 숨기고 대신 수행하는 단일 인터페이스 제공'이라는 요구와는 목적이 다르다.

NoteBase

선지별 해설

① 어댑터 패턴 → 서로 다른 인터페이스를 가진 클래스를 클라이언트가 원하는 인터페이스로 변환해 호환시키는 패턴으로, 지문에는 인터페이스 불일치 문제나 변환 요구가 나타나지 않으므로 해당하지 않는다.
② 추상 팩토리 패턴 → 관련 객체들의 집합(제품군)을 구체 클래스에 의존하지 않고 생성하는 생성 패턴으로, 지문은 객체 생성이 아니라 이미 존재하는 여러 클래스의 메서드 호출 순서를 감추는 문제이므로 맞지 않는다.
③ 컴포지트 패턴 → 개별 객체와 객체들의 집합을 동일한 방식으로 다루기 위한 트리 구조 패턴으로, 지문에는 부분-전체 계층 구조나 재귀적 구성 요구가 없으므로 해당하지 않는다.
④ 퍼사드 패턴 → 터치 인식, 캐릭터 회전, 무기 조준, 무기 발사라는 여러 클래스의 메서드 호출 절차를 클라이언트가 몰라도 되도록 하나의 높은 레벨 인터페이스로 감싸주므로 지문 상황에 정확히 부합한다.

오답 포인트

패턴 이름과 목적을 1대1로 정확히 매칭하지 못하면 '무언가를 감싼다'는 인상만으로 어댑터나 컴포지트를 고르는 오답이 나오기 쉽다. 어댑터는 인터페이스 변환, 컴포지트는 계층 구조 표현이 핵심이지 호출 절차 은닉이 핵심이 아님을 구분해야 한다.

다시 나오면 이렇게 판단

지문에서 '여러 클래스/메서드를 순서대로 호출해야 하는 복잡함을 감추고 하나의 단순한 인터페이스를 제공한다'는 표현이 나오면 퍼사드 패턴을 우선 떠올린다. 반대로 '인터페이스가 맞지 않아 변환이 필요하다'면 어댑터, '객체 생성 계열을 캡슐화한다'면 추상 팩토리, '부분과 전체를 동일하게 다룬다'면 컴포지트로 구분한다.

암기 포인트

퍼사드(Facade)=건물의 정면이라는 뜻처럼, 복잡한 서브시스템 앞에 단순한 얼굴 하나를 세워 클라이언트가 그 얼굴만 보고 쓰게 하는 패턴으로 기억해 둔다.

NoteBase

문 21. 다음 코드에 대해 기본 경로(basis path) 테스트를 수행할 때, 모든 독립적인 경로가 최소한 한 번은 실행되는 테스트 입력 집합은?

```
int max(int a, int b, int c) {
  int result = a;
  if (b > result)
    result = b;
  if (c > result)
    result = c;
  return result;
}
```

① (a=1, b=2, c=3)

② (a=1, b=1, c=1), (a=1, b=2, c=1)

③ (a=1, b=1, c=1), (a=1, b=1, c=2), (a=1, b=2, c=2)

④ (a=1, b=2, c=3), (a=1, b=3, c=2), (a=2, b=3, c=3)

정답 3번

출제 포인트

기본 경로 테스트(basis path testing)는 코드의 순환복잡도 $V(G)$를 구하고, $V(G)$개의 서로 독립적인(선형독립인) 실행 경로를 각각 최소 한 번씩 실행시키는 테스트 케이스 집합을 만드는 기법이다. 이 문제는 두 개의 if문으로 이루어진 코드에서 실제로 서로 다른 경로를 커버하는 입력 조합을 골라내는 능력을 묻는다.

조건 정리

코드는 result=a로 초기화한 뒤, if(b>result) result=b, if(c>result) result=c 두 개의 순차적(중첩 아님) 조건문을 거쳐 result를 반환한다. 결정문(if)이 2개이므로 코드 안에는 참/거짓 두 갈래를 갖는 분기점이 2개 있다.

NoteBase

사용할 공식

순환복잡도 V(G) = 결정노드 수 + 1이다. 이 코드는 if문이 2개이므로 $V(G) = 2 + 1 = 3$이며, 기본 경로 테스트를 만족하려면 서로 독립적인 경로 3개를 모두 실행하는 테스트 케이스가 필요하다.

계산 과정

두 if문의 참/거짓 조합은 이론상 (F,F), (F,T), (T,F), (T,T) 4가지가 가능하지만, $V(G) = 3$이므로 이 중 선형독립인 3가지 경로만 있으면 충분하다.
① (a=1,b=2,c=3): b(2)>result(1) 참 → result=2, 이어서 c(3)>result(2) 참 → 경로 (T,T) 하나만 실행된다.
② 첫 번째 케이스 (1,1,1): b(1)>1 거짓, c(1)>1 거짓 → (F,F). 두 번째 케이스 (1,2,1): b(2)>1 참 → result=2, c(1)>2 거짓 → (T,F). 두 경로만 커버되고 (F,T) 경로가 빠진다.
③ 첫 번째 케이스 (1,1,1): b(1)>1 거짓, c(1)>1 거짓 → (F,F). 두 번째 케이스 (1,1,2): b(1)>1 거짓, c(2)>1 참 → (F,T). 세 번째 케이스 (1,2,2): b(2)>1 참 → result=2, c(2)>2 거짓 → (T,F). 즉 (F,F), (F,T), (T,F) 세 가지 서로 다른 경로가 모두 실행된다.
④ 첫 번째 케이스 (1,2,3): (T,T)(위 ①과 동일 경로). 두 번째 케이스 (1,3,2): b(3)>1 참 → result=3, c(2)>3 거짓 → (T,F). 세 번째 케이스 (2,3,3): b(3)>2 참 → result=3, c(3)>3 거짓 → (T,F). 결국 (T,T)와 (T,F)만 커버되고 (T,F)는 중복되며 (F,F) 경로가 빠진다.

선지 판단

① (a=1,b=2,c=3): 실행 경로가 (T,T) 하나뿐이라 3개의 독립 경로를 커버하지 못한다. → 오답
② (a=1,b=1,c=1), (a=1,b=2,c=1): 커버되는 경로가 (F,F)와 (T,F) 두 가지뿐이며 (F,T) 경로가 빠져 있다. → 오답
③ (a=1,b=1,c=1), (a=1,b=1,c=2), (a=1,b=2,c=2): (F,F), (F,T), (T,F) 세 가지 독립 경로를 모두 커버한다. → 정답
④ (a=1,b=2,c=3), (a=1,b=3,c=2), (a=2,b=3,c=3): 커버되는 경로가 (T,T)와 (T,F)(두 번 중복)뿐이며 (F,F) 경로가 빠져 있다. → 오답

오답 포인트

테스트 케이스가 3개라고 해서 저절로 기본 경로 테스트 요건을 만족하는 것은 아니다. ④처럼 입력 값이 서로 달라 보여도 실제 분기 결과(T/F 조합)가 겹치면 독립적인 경로를 커버한 것이 아니다. 반대로 ②는 테스트가 2개뿐이라 애초에 $V(G) = 3$개의 경로를 다 채울 수 없다. 판단 기준은 입력 값의 개수가 아니라 실제로 실행되는 조건 분기(T/F)의 조합이다.

다시 나오면 이렇게 판단

먼저 코드 안의 결정문(if, while, for, case, 논리연산자 등) 개수를 세어 V(G) = 결정노드 수 + 1을 구한다. 그다음 각 선지의 입력값을 코드에 직접 대입해 매 결정문의 참/거짓 결과를 하나씩 표시하고, 선지 전체가 만들어내는 T/F 조합 경로들이 서로 다르며 $V(G)$개를 모두 채우는지 확인한다.

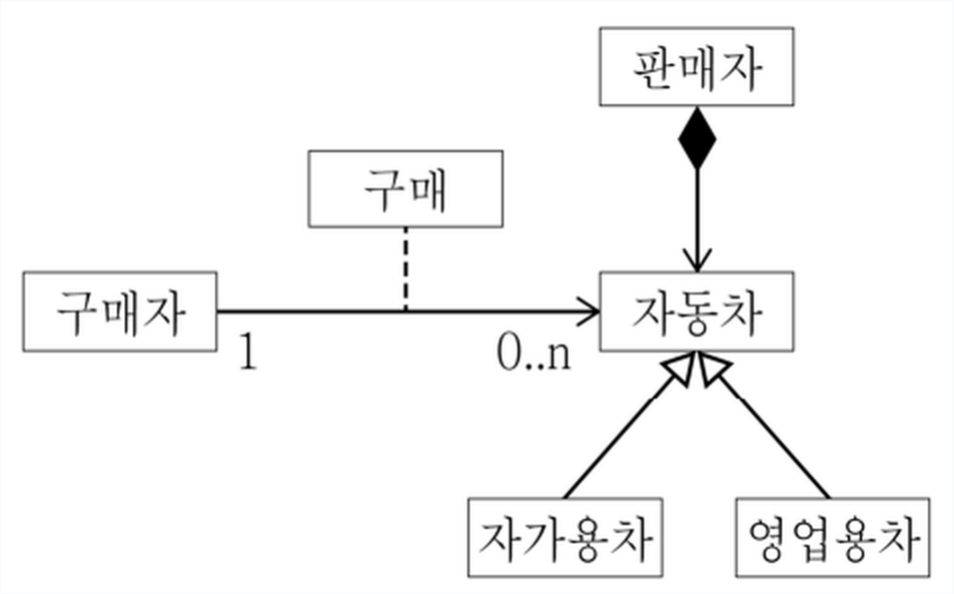
NoteBase

암기 포인트

순환복잡도 V(G) = 결정노드 수 + 1(또는 V(G) = 간선 수 - 노드 수 + 2)이며, 기본 경로 테스트에 필요한 최소 테스트 케이스 수는 $V(G)$와 같다. 판단은 입력값 자체가 아니라 각 결정문에서 실제로 실행되는 참/거짓 조합이 서로 다른가로 한다.

NoteBase

문 22. 다음 클래스 다이어그램에 대한 설명으로 옳은 것은?



① '자동차' 클래스는 '구매자' 클래스를 직접 참조한다.

② '자가용차', '영업용차' 클래스는 '자동차' 클래스와 실체화 관계를 갖는다.

③ '판매자' 객체가 소멸되어도 집약(aggregation)관계인 '자동차' 객체는 소멸되지 않는다.

④ '구매' 클래스는 '구매자' 클래스와 '자동차' 클래스의 연관관계에 속성이나 행위를 추가하기 위해 사용된다.

정답 4번

출제 포인트

UML 클래스 다이어그램에 쓰인 관계 표기(연관, 다중도, 연관 클래스, 컴포지션, 일반화)를 정확히 구분해 해석할 수 있는지를 묻는다. 다이어그램에는 구매자에서 자동차로 향하는 실선 화살표(다중도 1, 0..n)와 그 연관선에 점선으로 연결된 '구매' 클래스, 판매자에서 자동차로 향하는 채워진 다이아몬드, 자가용차·영업용차에서 자동차로 향하는 속이 빈 삼각형이 나타난다.

NoteBase

선지별 해설

① X: 화살표는 구매자에서 자동차 방향으로 그려져 있다. 이는 구매자 클래스가 자동차 클래스를 참조(탐색)한다는 뜻이며, 자동차 클래스가 구매자 클래스를 참조하는 것이 아니다. 참조의 주체와 대상이 반대로 서술되어 있다.
② X: 자가용차·영업용차와 자동차 사이의 표기는 속이 빈 삼각형과 실선으로 이루어진 일반화(상속) 관계이다. 실체화(realization) 관계는 속이 빈 삼각형에 점선을 사용하며 인터페이스와 구현 클래스 사이에 쓰이는 표기이므로 이 다이어그램과는 다르다.
③ X: 판매자와 자동차 사이는 속이 채워진 다이아몬드이므로 집약(aggregation)이 아니라 컴포지션(composition) 관계이다. 컴포지션은 전체 객체가 소멸되면 부분 객체도 함께 소멸되는 강한 소유 관계이므로, 판매자 객체가 소멸되면 자동차 객체도 함께 소멸된다. 관계의 명칭도 틀렸고 소멸 여부에 대한 설명도 반대로 되어 있다.
④ O: '구매' 클래스는 구매자와 자동차 사이의 연관관계 선에 점선으로 연결되어 있다. 이는 연관 클래스(association class)로, 두 클래스 사이의 연관관계 자체에 속성이나 행위(예: 구매일자, 구매금액 등)를 추가로 부여하고 싶을 때 사용하는 표기이다.

오답 포인트

다이어그램에서 화살표 방향과 참조 주체를 뒤바꿔 읽지 않도록 주의한다. 또한 겉모습이 비슷한 관계 표기(속이 빈 삼각형 실선=일반화, 속이 빈 삼각형 점선=실체화, 속이 빈 다이아몬드=집약, 속이 채워진 다이아몬드=컴포지션)를 서로 혼동하기 쉬우므로 선·화살표·다이아몬드 채움 여부를 하나씩 대조해서 판단해야 한다.

다시 나오면 이렇게 판단

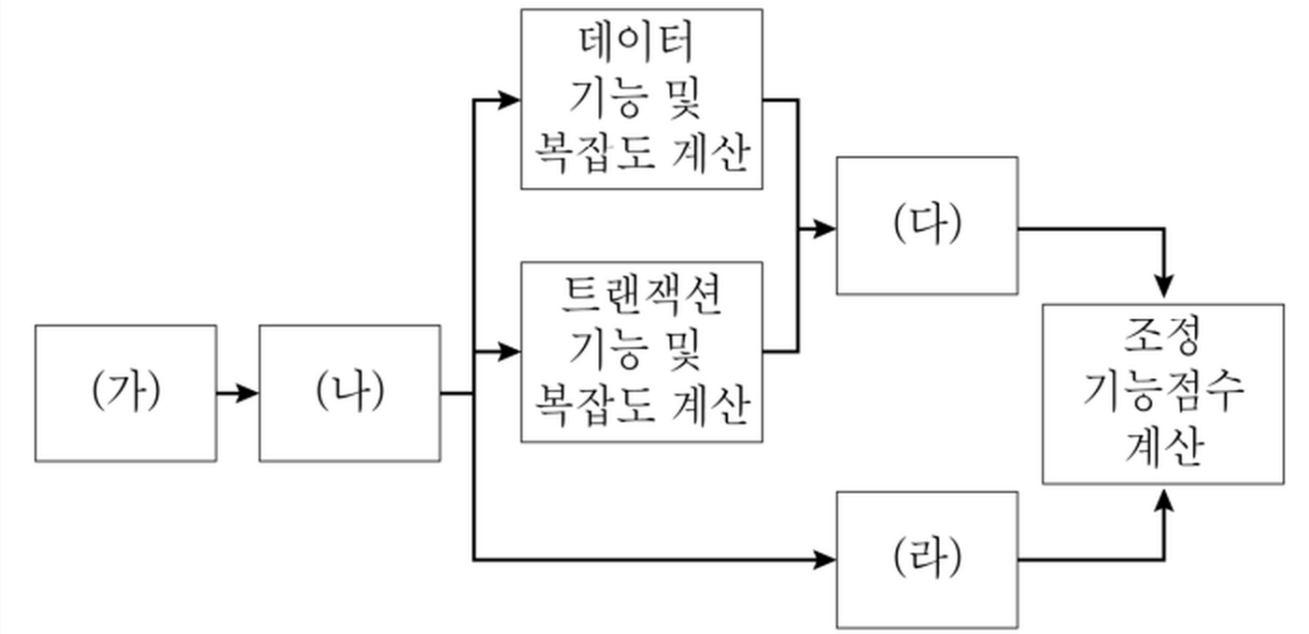
먼저 각 관계선의 끝 모양(화살표, 삼각형, 다이아몬드)과 선 종류(실선/점선)를 확인하고, 점선으로 별도 클래스가 매달려 있으면 연관 클래스인지부터 확인한다. 그다음 다이아몬드가 채워져 있는지(컴포지션) 비어 있는지(집약)를 확인하고, 삼각형이 실선(일반화)인지 점선(실체화)인지를 확인하면 선지를 빠르게 소거할 수 있다.

암기 포인트

일반화(상속)=속이 빈 삼각형+실선, 실체화=속이 빈 삼각형+점선, 집약=속이 빈 다이아몬드, 컴포지션=속이 채워진 다이아몬드(전체 소멸 시 부분도 소멸), 연관 클래스=연관선에 점선으로 매달린 클래스로 연관관계에 속성·행위를 추가할 때 사용한다.

NoteBase

문 23. 다음은 IFPUG에서 정의한 기능점수 산정 절차이다. (가) ~ (라)에 해당하는 것을 A ~ D에서 바르게 연결한 것은?



A. 계산 범위 및 애플리케이션 경계 식별
B. 기능점수 측정 유형 결정
C. 미조정 기능점수 계산
D. 조정 인자 계산

① (가): A, (나): B, (다): D, (라): C
② (가): B, (나): A, (다): C, (라): D
③ (가): B, (나): A, (다): D, (라): C
④ (가): D, (나): A, (다): C, (라): B

정답 2번

출제 포인트

IFPUG(국제 기능점수 사용자 그룹) 방식의 기능점수 산정 절차를 순서대로 정확히 알고 있는지를 묻는 문제이다. 그림의 순서도는 처음 두 단계(가), (나)를 거친 뒤 데이터 기능/트랜잭션 기능의 복잡도 계산이 이루어지고, 그 결과가 (다)로 모여 미조정 기능점수가 산출되며, (나)에서 갈라져 나온 (라)를 통해 조정 인자가 계산된 다음, 두 결과가 합쳐져 최종적으로 조정 기능점수가 계산되는 흐름을 보여 준다.

NoteBase

올바른 순서

IFPUG 기능점수 산정은 다음 순서로 진행된다.
1) 기능점수 측정 유형 결정(개발 프로젝트용/개선 프로젝트용/애플리케이션용 중 어떤 목적의 측정인지 먼저 정한다) — 그림의 (가)
2) 계산 범위 및 애플리케이션 경계 식별(측정 대상 시스템의 범위와 다른 시스템과의 경계를 정한다) — 그림의 (나)
3) 데이터 기능(내부논리파일·외부연계파일) 및 트랜잭션 기능(외부입력·외부출력·외부조회)의 복잡도를 각각 계산한다.
4) 3)의 결과를 합산하여 미조정 기능점수를 계산한다 — 그림의 (다)
5) (나)의 경계 식별 결과를 바탕으로 일반시스템특성 등을 평가하여 조정 인자(값 조정 계수)를 계산한다 — 그림의 (라)
6) 미조정 기능점수와 조정 인자를 결합해 조정 기능점수를 계산한다.
따라서 (가)=B(기능점수 측정 유형 결정), (나)=A(계산 범위 및 애플리케이션 경계 식별), (다)=C(미조정 기능점수 계산), (라)=D(조정 인자 계산)이다.

선지별 해설

① (가)A (나)B (다)D (라)C → X. (가)와 (나)의 순서가 실제와 반대이고, (다)와 (라)의 내용도 서로 바뀌어 있다. (다)는 데이터·트랜잭션 복잡도 계산 결과를 받아 미조정 기능점수(C)를 산출하는 단계이지 조정 인자(D)를 계산하는 단계가 아니다.
② (가)B (나)A (다)C (라)D → O. 측정 유형 결정(B)이 먼저이고 그 다음 경계 식별(A)이며, 데이터·트랜잭션 복잡도 계산 결과가 모이는 (다)는 미조정 기능점수 계산(C), 별도 경로로 갈라지는 (라)는 조정 인자 계산(D)이므로 모두 순서도의 흐름과 일치한다.
③ (가)B (나)A (다)D (라)C → X. (가), (나)는 맞게 배치했으나 (다)와 (라)가 서로 바뀌었다. (다)는 데이터·트랜잭션 기능 복잡도 계산 박스로부터 화살표를 받으므로 그 결과를 합산하는 미조정 기능점수 계산(C)이 되어야 하고, (라)는 별도 경로에서 조정 기능점수 계산으로 바로 들어가므로 조정 인자 계산(D)이 되어야 한다.
④ (가)D (나)A (다)C (라)B → X. (나), (다)는 맞으나 (가)에 조정 인자 계산(D)을, (라)에 측정 유형 결정(B)을 배치하여 절차의 시작 단계와 중간 단계가 뒤바뀌어 있다. 측정 유형 결정(B)은 절차의 맨 처음에 와야 하는 단계이다.

NoteBase

오답 포인트

이 문제의 함정은 (다)와 (라)를 헷갈리는 것이다. 그림에서 (다)는 데이터 기능 및 트랜잭션 기능의 복잡도 계산 박스 두 개로부터 화살표를 받아 모이는 위치에 있으므로 두 계산 결과를 합산하는 미조정 기능점수 계산(C)이 되어야 하고, (라)는 그 경로와 별도로 처음부터 곧바로 조정 기능점수 계산 박스로 이어지므로 조정 인자 계산(D)이 되어야 한다. 위치와 화살표의 방향을 무시하고 이름만 보고 짝을 맞추면 ①이나 ③처럼 (다)(라)를 뒤바꾸는 오류가 생긴다. 또한 (가)와 (나)의 순서도 유의해야 하는데, 애플리케이션 경계를 식별하기 전에 먼저 이번 측정이 어떤 목적(개발/개선/애플리케이션)의 기능점수 측정인지 유형을 결정하는 것이 IFPUG 절차의 원칙이다.

다시 나오면 이렇게 판단

순서도형 문제는 각 박스가 어디로부터 화살표를 받고 어디로 화살표를 보내는지부터 확인한다.
데이터·트랜잭션 기능 복잡도 계산 박스 둘로부터 동시에 화살표를 받는 박스는 그 두 결과를 합산하는 단계(미조정 기능점수 계산)이고, 처음부터 별도 경로로 바로 최종 결과 박스로 이어지는 단계는 조정 인자(가중치) 계산 단계라는 점을 기준으로 삼는다. 그리고 IFPUG 절차의 시작은 항상 측정 유형 결정 → 애플리케이션 경계 식별 순서임을 기억한다.

암기 포인트

IFPUG 기능점수 산정 6단계: 측정 유형 결정 → 애플리케이션 경계 식별 → 데이터/트랜잭션 기능 복잡도 계산 → 미조정 기능점수 계산 → 조정 인자(VAF) 계산 → 조정 기능점수 계산. 순서를 '유형→경계→복잡도→미조정→조정인자→조정점수'로 통째로 외워 둔다.

NoteBase

문 24. 다음은 '갑' 프로젝트의 계획 단계에서 수립한 CPM(critical path method) 네트워크의 작업 목록표이다. '갑' 프로젝트 완료에 필요한 최소 시간은?

작업	선행작업	소요기간(일)
A	D	15
B	A, D, E	10
C	E	15
D	-	5
E	-	10
F	A, B	20
G	B, C	15

① 40일

② 45일

③ 50일

④ 90일

정답 3번

출제 포인트

CPM(critical path method) 네트워크에서 각 작업의 선행 관계와 소요기간이 주어졌을 때, 각 작업의 최조착수시각(ES)과 최조종료시각(EF)을 앞에서부터 순차적으로 계산해 전체 프로젝트를 끝내는 데 필요한 최소 시간(주공정, critical path 길이)을 구할 수 있는지를 묻는다.

조건 정리

선행작업이 없는 작업은 D(5일), E(10일)이다. A는 D에 의존, B는 A·D·E 모두에 의존, C는 E에 의존, F는 A·B에 의존, G는 B·C에 의존한다. 한 작업은 자신의 모든 선행작업이 끝난 뒤에만 시작할 수 있으므로, 선행작업이 여럿이면 그 중 가장 늦게 끝나는 시점이 착수시각이 된다.

NoteBase

사용할 공식

각 작업의 최조종료시각은 EF = ES + 소요기간이며, 선행작업이 둘 이상인 작업의 최조착수시각은 ES = (선행작업들의 EF)이다. 프로젝트 전체 완료시간은 마지막 작업들의 *EF* 중 최댓값, 즉 max(*EF*)이다.

계산 과정

D: 선행작업 없음 → ES=0, EF=0+5=5
E: 선행작업 없음 → ES=0, EF=0+10=10
A: 선행 D → ES=EF(D)=5, EF=5+15=20
C: 선행 E → ES=EF(E)=10, EF=10+15=25
B: 선행 A, D, E → ES=max(20, 5, 10)=20, EF=20+10=30
F: 선행 A, B → ES=max(20, 30)=30, EF=30+20=50
G: 선행 B, C → ES=max(30, 25)=30, EF=30+15=45
프로젝트는 후행작업이 없는 F와 G에서 끝나므로, 완료시간은 max(EF(F), EF(G))=max(50, 45)=50일이다.

선지 판단

① 40일 → X. F 경로(D→A→B→F, 5+15+10+20=50)만으로도 40일을 넘어서므로 최소 소요시간이 될 수 없다.
② 45일 → X. G 경로의 최조종료시각(45일)에 해당하지만, 이는 F 경로(50일)보다 짧아 전체 프로젝트를 완료하는 시간이 아니다. 프로젝트는 모든 경로가 끝나야 완료되므로 가장 긴 경로 기준으로 정해진다.
③ 50일 → O. D→A→B→F 경로의 길이가 5+15+10+20=50일로 가장 길며, 이것이 주공정(critical path)이다.
④ 90일 → X. 모든 작업의 소요기간을 단순히 합산한 값(15+10+15+5+10+20+15=90)으로, 병렬로 진행 가능한 작업들을 고려하지 않은 오답이다.

오답 포인트

선행작업이 여러 개일 때 착수시각을 하나의 선행작업 EF만 보고 정하면 안 되며, 반드시 모든 선행작업 EF 중 최댓값을 취해야 한다. 또한 프로젝트 완료시간은 마지막 작업 하나의 EF가 아니라, 후행작업이 없는 모든 작업들의 EF 중 최댓값이라는 점에 유의한다. G의 EF(45)만 보고 답을 고르면 F 경로가 더 길다는 사실을 놓치게 된다.

다시 나오면 이렇게 판단

작업 목록표가 주어지면 먼저 선행작업이 없는 작업부터 ES=0으로 두고, 표의 순서와 무관하게 선행관계가 모두 확정된 작업부터 차례로 EF를 계산해 나간다. 마지막에는 후행작업으로 이어지지 않는 작업들의 EF를 모두 비교해 그 중 최댓값을 프로젝트 완료시간으로 답한다.

NoteBase

암기 포인트

ES(최조착수시각) = 모든 선행작업 EF의 최댓값, EF(최조종료시각) = ES + 소요기간, 프로젝트 완료시간 = 후행작업이 없는 작업들 EF의 최댓값(=주공정 길이)이다.

NoteBase

문 25. UML 시퀀스 다이어그램의 결합 프레그먼트(combined fragment)에서 사용하는 인터랙션 오퍼레이터에 대한 설명으로 옳은 것은?

① ref - 동시에 실행되는 상호작용을 표현한다.

② opt - 다른 다이어그램을 참조할 때 사용한다.

③ par - 특정 조건을 만족하는 경우에만 상호작용을 수행한다.

④ alt - 여러 상호작용 중 조건을 만족하는 상호작용만 수행하는데, 각 상호작용은 점선으로 분리한다.

정답 4번

출제 포인트

UML 시퀀스 다이어그램의 결합 프레그먼트(combined fragment)에 쓰이는 인터랙션 오퍼레이터 각각의 의미를 정확히 구분할 수 있는지를 묻는다. ref, opt, par, alt는 서로 역할이 다르므로 이름과 기능을 짝지어 외워 둘 필요가 있다.

핵심 개념

ref는 다른 시퀀스 다이어그램에 정의된 상호작용을 참조해 불러오는 오퍼레이터이다. opt는 조건을 만족할 때만 실행되는 단일 대안(옵션)을 표현한다. par는 여러 상호작용이 동시에(병렬로) 실행됨을 표현한다. alt는 여러 상호작용 중 조건을 만족하는 하나만 선택적으로 실행되며, 프레그먼트 안의 각 대안(operand)은 점선으로 구분된다.

선지별 해설

① ref - 동시에 실행되는 상호작용을 표현한다. (X) ref는 동시 실행이 아니라 다른 다이어그램에 정의된 상호작용을 참조하는 오퍼레이터이다. → 동시 실행을 표현하는 것은 par이다.
② opt - 다른 다이어그램을 참조할 때 사용한다. (X) 다른 다이어그램을 참조하는 것은 ref이다. opt는 조건을 만족하는 경우에만 실행되는 단일 대안을 표현한다.
③ par - 특정 조건을 만족하는 경우에만 상호작용을 수행한다. (X) 특정 조건을 만족하는 경우에만 수행되는 것은 opt(또는 alt)이며, par는 여러 상호작용이 동시에 병렬로 수행됨을 표현한다.
④ alt - 여러 상호작용 중 조건을 만족하는 상호작용만 수행하는데, 각 상호작용은 점선으로 분리한다. (O) alt는 여러 대안 중 조건을 만족하는 하나만 실행되는 오퍼레이터이며, 각 대안(operand)은 프레그먼트 내부에서 점선으로 구분되므로 설명이 정확하다.

오답 포인트

이 문제는 ref-동시실행, opt-참조, par-조건부실행 처럼 오퍼레이터 이름과 설명을 서로 뒤바꿔 놓은 함정이 핵심이다. 이름만 보고 기능을 추측하지 말고 각 오퍼레이터의 정의를 정확히 짝지어야 한다.

NoteBase

다시 나오면 이렇게 판단

선지에서 오퍼레이터 이름과 설명 문구가 나오면, 먼저 네 오퍼레이터의 정의를 각각 떠올린 뒤 이름-설명 쌍이 서로 바뀌어 있지 않은지부터 확인한다. alt는 '조건에 따른 선택 실행 + 점선 구분'이라는 두 특징을 함께 기억해 둔다.

암기 포인트

ref=참조(reference), opt=옵션(조건부 단일 실행), par=병렬(동시 실행), alt=대안(조건별 택일 실행, 점선으로 구분).

NoteBase