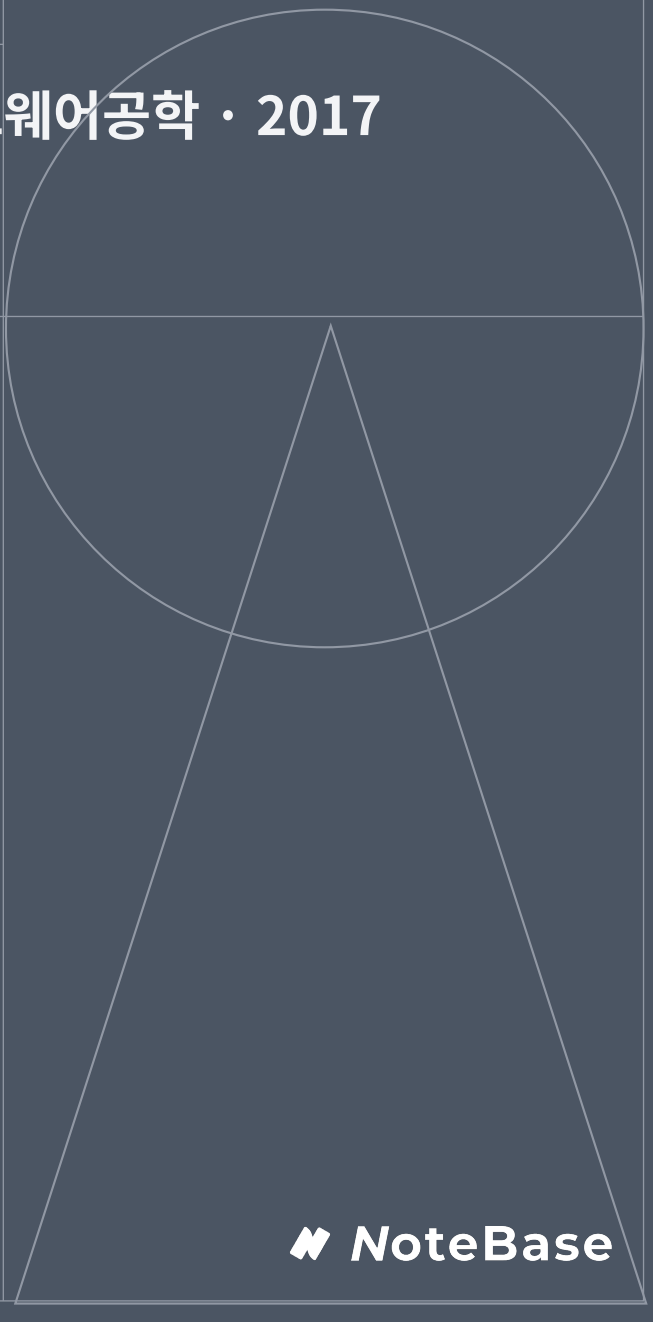
기출OX 해설집

공무원/국가직/7급 · 소프트웨어공학 · 2017



NoteBase

기출OX 해설집

공무원/국가직/7급 · 소프트웨어공학 · 2017

공무원/국가직/7급 · 소프트웨어공학 · 2017

문 1. 소프트웨어 요구분석 명세서(SRS)에 포함되는 내용이 아닌 것은?

① 개발 시스템의 목적 및 범위

② 아키텍처 및 인터페이스 명세

③ 자료흐름도, 자료사전, 소단위명세서

④ 기능적 요구사항 및 비기능적 요구사항

정답 2번

출제 포인트

소프트웨어 요구사항 명세서(SRS, Software Requirements Specification)에 실제로 담기는 항목이 무엇인지를 구분하는 문제이다. SRS는 '무엇을(what)' 만들 것인지를 기술하는 문서이지 '어떻게(how)' 구현할 것인지를 정하는 설계 문서가 아니라는 점이 핵심이다.

핵심 개념

SRS에는 시스템의 목적과 범위, 기능적 요구사항(시스템이 수행해야 할 기능)과 비기능적 요구사항(성능, 신뢰성, 보안 등 품질 속성), 그리고 요구사항을 구조화해 표현하는 자료흐름도(DFD)·자료사전·소단위명세서(minispec) 같은 분석 도구의 산출물이 포함된다. 반면 아키텍처 설계(모듈 구조, 계층 구성)와 인터페이스의 상세 명세는 요구분석 이후 단계인 설계 단계의 산출물(설계 명세서, SDD)에서 다루는 내용으로, 요구분석 명세서의 범위를 벗어난다.

선지별 해설

① 개발 시스템의 목적 및 범위: O, SRS는 서두에 시스템 개발의 목적과 적용 범위를 명시한다.
② 아키텍처 및 인터페이스 명세: X, 이는 요구분석 이후 설계 단계에서 다루는 내용이며 SRS의 구성 항목이 아니다 → SRS에는 '요구사항이 무엇인가'만 담기고 아키텍처·인터페이스의 구체적 설계는 설계 명세서에 담긴다.
③ 자료흐름도, 자료사전, 소단위명세서: O, 구조적 분석 기법으로 요구사항을 정리한 산출물로 SRS에 포함된다.
④ 기능적 요구사항 및 비기능적 요구사항: O, SRS의 가장 핵심적인 구성 요소이다.

NoteBase

오답 포인트

아키텍처나 인터페이스라는 단어가 나오면 무조건 요구분석 단계로 착각하기 쉬운데, 이는 요구사항을 바탕으로 시스템을 어떻게 구성할지 결정하는 설계 단계의 산출물이다. '무엇을 요구하는가'와 '어떻게 구현하는가'를 구분하지 못하면 오답을 고르게 된다.

다시 나오면 이렇게 판단

선지에 목적/범위, 기능적·비기능적 요구사항, DFD·자료사전·소단위명세서처럼 '요구사항 자체를 정리·기술'하는 항목이 나오면 SRS에 포함되는 내용으로 판단하고, 아키텍처 구조나 인터페이스 상세 설계처럼 '어떻게 구현할지'를 다루는 항목이 나오면 설계 단계 산출물로 분류해 SRS와 구분한다.

암기 포인트

SRS = 목적·범위 + 기능/비기능 요구사항 + DFD·자료사전·소단위명세서(구조적 분석 산출물).
아키텍처·인터페이스 설계는 SRS가 아니라 설계 명세서(SDD) 몫이라고 정리해 둔다.

NoteBase

문 2. 프로토타입 개발 모델에 대한 설명으로 옳지 않은 것은?

① 시스템 기능을 사용자에게 확인시킴으로써 개발자와 사용자간의 견해차이가 해결될 수 있다.

② 분석가나 개발자는 프로토타입을 이용하여 불완전하거나 일치하지 않는 요구사항을 발견할 수 있다.

③ 완전하지는 못하지만 작동하는 시스템을 만들어 기능성과 유용성을 관리자에게 보여줄 수 있다.

④ 고객의 요구사항을 초기에 구체적으로 기술하기 어렵고 중요한 문제점이 프로젝트의 후반부에 가서야 발견된다.

정답 4번

출제 포인트

프로토타입 개발 모델이 실제로 갖는 특성(사용자·개발자 간 견해차이 해소, 불완전·모순된 요구사항 발견, 관리자에게 실현 가능성을 보여주는 시제품 제작)과, 반대로 프로토타입을 만들지 않을 때 생기는 문제점(요구사항을 초기에 구체화하기 어렵고 문제가 프로젝트 후반부에야 드러남)을 구분할 수 있는지를 묻는다.

핵심 개념

프로토타입 모델은 실제 동작하는 시제품을 초기에 만들어 사용자에게 보여주고 피드백을 받는 개발 방식이다. 이를 통해 (1) 개발자와 사용자 사이의 요구사항 견해차이를 조기에 좁히고, (2) 문서만으로는 드러나지 않는 불완전하거나 모순된 요구사항을 실제 조작을 통해 찾아내며, (3) 완전하지 않아도 동작하는 시스템으로 기능성과 유용성을 관리자·이해관계자에게 시연할 수 있다는 장점을 가진다. 반면 ④에서 말하는 상황, 즉 요구사항을 초기에 구체적으로 기술하기 어렵고 중요한 문제점이 프로젝트 후반부에 가서야 발견되는 것은 프로토타입 없이 명세만으로 개발을 진행하는 전통적(폭포수) 방식에서 나타나는 문제이며, 프로토타입 모델은 바로 이 문제를 해결하기 위해 도입되는 기법이다. 따라서 ④는 프로토타입 모델 자체에 대한 설명이 아니라 프로토타입 모델이 필요한 이유(즉 프로토타입을 쓰지 않을 때의 단점)를 서술한 것이므로 옳지 않다.

선지별 해설

① O: 사용자가 프로토타입을 직접 확인하면서 개발자와 사용자 사이의 요구사항에 대한 견해차이가 해소될 수 있다는 것은 프로토타입 모델의 대표적인 장점이다.
② O: 분석가나 개발자는 프로토타입을 만들어 보는 과정에서 문서상으로는 드러나지 않던 불완전하거나 상호 모순되는 요구사항을 발견할 수 있다.
③ O: 프로토타입은 완전한 시스템은 아니지만 실제로 동작하므로, 기능성과 유용성을 관리자나 이해관계자에게 보여주는 시연 수단이 될 수 있다.
④ X: 요구사항을 초기에 구체적으로 기술하기 어렵고 중요한 문제점이 프로젝트 후반부에야 발견되는 것은 프로토타입 없이 진행하는 방식의 문제점이며, 프로토타입 모델은 이러한 문제를 조기에 해결하기 위한 기법이다 → 프로토타입 모델에 대한 옳은 설명이 아니라, 프로토타입을 쓰지 않을 때 생기는 문제 상황에 대한 설명이다.

NoteBase

오답 포인트

④를 프로토타입 모델의 특징으로 착각하기 쉬운 이유는 문장 자체가 소프트웨어 개발에서 실제로 발생하는 현상을 정확히 설명하고 있기 때문이다. 그러나 이 문제는 프로토타입 모델에 대한 옳지 않은 설명을 찾는 것이므로, 문장의 사실 여부가 아니라 그 문장이 프로토타입 모델의 장점을 설명하는지, 아니면 프로토타입 모델이 해결하려는 기존 문제를 설명하는지를 구분해야 한다.

다시 나오면 이렇게 판단

선지에 등장하는 각 문장이 프로토타입을 사용했을 때의 결과인지, 아니면 프로토타입을 사용하지 않았을 때의 문제인지를 먼저 구분한다. 프로토타입 모델 설명 문제에서는 '견해차이 해소', '불완전·모순 요구사항 발견', '관리자에게 시연 가능'이 정석적인 장점으로 반복 출제되고, '요구사항을 초기에 구체화하기 어렵다', '문제점이 후반부에 발견된다'는 표현은 프로토타입이 없을 때의 단점으로 뒤집혀 나오는 함정임을 기억한다.

암기 포인트

프로토타입 모델의 3대 장점은 견해차이 해소, 불완전/모순 요구사항 발견, 관리자 시연 가능이다. 반대로 '초기에 요구사항 구체화 어려움 + 문제점을 후반부에야 발견'은 프로토타입이 없을 때의 문제이지, 프로토타입 모델 자체의 설명이 아니다.

NoteBase

문 3. 코드의 정적 분석 도구를 통하여 찾을 수 있는 오류에 해당하지 않는 것은?

① 특정 모듈의 실행 시간 메모리 사용량 초과

② 데이터 정의의 충돌이나 잘못된 데이터 사용

③ 실행 경로가 없는 Dead Code

④ 원시 코드의 구조적 결함

정답 1번

출제 포인트

정적 분석(static analysis) 도구는 프로그램을 실행하지 않고 원시 코드 자체를 분석하여 오류를 찾아내는 기법이다. 이 문제는 정적 분석으로 발견 가능한 결함과, 프로그램을 실제로 실행해야만 알 수 있는 동적(runtime) 특성을 구분할 수 있는지를 묻는다.

핵심 개념

정적 분석은 소스 코드의 구조·문법·데이터 흐름을 검사해 코딩 표준 위반, 미사용 변수, 도달할 수 없는 코드, 변수의 정의·사용 충돌 등을 컴파일 단계에서 잡아낸다. 반면 실행 시간이나 메모리 사용량처럼 시스템 자원의 실제 소비량은 프로그램을 구동시켜 측정하는 동적 분석(성능 테스트, 프로파일링)의 영역이다. 즉 정적 분석은 코드를 '읽어서' 판단하고, 동적 분석은 코드를 '실행해서' 판단한다.

선지별 해설

① 특정 모듈의 실행 시간 메모리 사용량 초과 → 이는 프로그램을 실제로 실행시켜 자원 사용량을 측정해야 알 수 있는 동적 특성이므로, 코드를 실행하지 않는 정적 분석 도구로는 찾을 수 없다. 문제에서 요구하는 '해당하지 않는 것'이 바로 이 선지이다.
② 데이터 정의의 충돌이나 잘못된 데이터 사용 → 변수가 정의되지 않고 사용되거나, 서로 다른 자료형으로 충돌하는 등의 문제는 코드의 선언과 사용 흐름만 분석해도 발견되므로 정적 분석의 대표적 탐지 대상이다.
③ 실행 경로가 없는 Dead Code → 어떤 입력으로도 도달할 수 없는 코드(도달 불가능 코드)는 제어 흐름 구조만 분석해도 식별되므로 정적 분석으로 찾을 수 있다.
④ 원시 코드의 구조적 결함 → 코딩 규칙 위반, 중첩 구조 오류 등 코드 자체의 구조적 문제는 정적 분석 도구가 검사하는 핵심 항목이다.

오답 포인트

'분석 도구가 코드를 본다'는 사실만으로 모든 오류를 다 잡아낼 수 있다고 오해하면 안 된다. 실행 시간·메모리 사용량처럼 실제 구동 결과에 의존하는 값은 정적 분석의 대상이 아니라는 점을 놓치면 오답을 고르게 된다.

NoteBase

다시 나오면 이렇게 판단

선지에 '실행 시간', '메모리 사용량', '응답 속도'처럼 프로그램을 돌려봐야 알 수 있는 수치가 나오면 동적 분석의 영역으로 분류하고, '코드 구조', '데이터 흐름', '도달 불가능 코드'처럼 소스만 봐도 판단 가능한 항목은 정적 분석의 영역으로 분류한다.

암기 포인트

정적 분석=실행 없이 코드를 읽어 구조·흐름·표준 위반을 검사(Dead Code, 데이터 충돌, 구조적 결함). 동적 분석=실제 실행으로 성능·자원 사용량을 측정(실행 시간, 메모리 사용량).

NoteBase

문 4. 리팩토링에 대한 설명으로 옳지 않은 것은?

① 리팩토링의 대상은 읽기 어려운 코드, 중복된 로직의 코드, 복잡한 조건문이 있는 코드 등이 대표적인 것이다.

② 리팩토링은 실행 중인 프로그램의 기능 변경이 수반되어야 한다.

③ 리팩토링을 통해서 프로그램의 이해가 쉬워진다.

④ 리팩토링은 결함을 찾는 데 도움을 준다.

정답 2번

출제 포인트

리팩토링(refactoring)의 정의와 목적을 정확히 알고 있는지, 특히 리팩토링이 프로그램의 '외부 동작(기능)'은 그대로 유지하면서 '내부 구조'만 개선하는 작업이라는 핵심 원칙을 아는지를 묻는다.

핵심 개념

리팩토링은 소프트웨어의 겉으로 드러나는 기능(동작 결과)을 바꾸지 않으면서 내부의 코드 구조, 설계를 개선하는 작업이다. 즉 입력에 대한 출력 결과는 리팩토링 전후가 동일해야 하며, 새로운 기능을 추가하거나 기존 기능을 변경하는 것은 리팩토링의 범위에 포함되지 않는다. 대상이 되는 코드는 읽기 어려운 코드, 중복된 로직을 가진 코드, 지나치게 복잡한 조건문을 가진 코드 등이며, 이러한 코드를 개선하면 프로그램의 가독성과 이해도가 높아지고, 구조가 단순해지는 과정에서 숨어 있던 결함을 발견하는 데도 도움이 된다.

선지별 해설

① O: 리팩토링의 대상은 읽기 어려운 코드, 중복된 로직의 코드, 복잡한 조건문이 있는 코드 등이 대표적이라는 설명은 리팩토링의 일반적인 적용 대상에 대한 옳은 설명이다.
② X: 리팩토링은 실행 중인 프로그램의 '기능 변경이 수반되어야 한다'는 설명은 틀렸다. 리팩토링의 핵심 원칙은 오히려 기능(외부 동작)을 변경하지 않고 내부 구조만 개선하는 것이다. 틀린 표현 '기능 변경이 수반되어야 한다' → 올바른 표현 '기능(동작)은 변경되지 않아야 한다'.
③ O: 리팩토링을 통해 코드 구조가 단순해지고 중복이 제거되면서 프로그램을 이해하기 쉬워진다는 설명은 옳다.
④ O: 코드를 정리하고 구조를 개선하는 과정에서 숨어 있던 논리적 오류나 결함을 발견하기 쉬워지므로, 결함을 찾는 데 도움을 준다는 설명은 옳다.

오답 포인트

'기능 개선'과 '구조 개선'을 혼동하면 안 된다. 리팩토링은 프로그램이 하는 일(기능, 출력 결과)은 그대로 두고 코드를 작성하는 방식(구조)만 바꾸는 것이다. 기능을 추가하거나 변경하는 작업은 리팩토링이 아니라 별도의 기능 개발이나 유지보수 작업으로 분류된다.

NoteBase

다시 나오면 이렇게 판단

선지에 '기능 변경', '기능 개선', '새로운 기능 추가'와 같은 표현이 리팩토링의 목적으로 제시되면 옳지 않은 설명일 가능성이 높다고 판단한다. 반대로 '가독성 향상', '구조 단순화', '중복 제거', '이해도 향상', '결함 발견 용이' 등의 표현은 리팩토링의 옳은 효과로 판단한다.

암기 포인트

리팩토링 = 기능은 그대로, 구조만 개선. 대상은 읽기 어려운 코드·중복 로직·복잡한 조건문. 효과는 가독성 향상·이해 용이·결함 발견 도움으로 정리해 둔다.

NoteBase

문 5. 같은 유형의 소프트웨어 테스트 기법으로만 묶은 것은?

ㄱ. 기본 경로 테스트
ㄴ. 페어와이즈 테스트
ㄷ. 모델 기반 테스트
ㄹ. 분기 커버리지
ㅁ. 직교 배열 테스트

① ㄱ, ㄴ, ㄹ
② ㄱ, ㄷ, ㄹ
③ ㄴ, ㄷ, ㅁ
④ ㄱ, ㄴ, ㄷ, ㄹ

정답 3번

출제 포인트

소프트웨어 테스트 기법을 코드 구조를 기준으로 설계하는 구조기반(화이트박스) 기법과, 명세·입력조합을 기준으로 설계하는 명세기반(블랙박스) 기법으로 구분할 수 있는지를 묻는다. ㄱ~ㅁ 다섯 개 기법 중 같은 범주에 속하는 것끼리 묶는 것이 과제이다.

핵심 개념

구조기반(화이트박스) 테스트=프로그램 내부의 코드 구조(경로, 분기, 조건)를 기준으로 테스트 케이스를 설계한다. 기본 경로 테스트와 분기 커버리지가 여기에 속한다.
명세기반(블랙박스) 테스트=내부 구조를 보지 않고 입력과 출력, 요구사항 명세를 기준으로 테스트 케이스를 설계한다. 페어와이즈 테스트, 직교 배열 테스트, 모델 기반 테스트가 여기에 속한다.
페어와이즈 테스트=여러 입력 변수의 값 조합 중 두 변수 간 모든 쌍이 최소 한 번씩 나타나도록 조합을 줄여서 테스트하는 조합기반 기법이다.
직교 배열 테스트=직교 배열표를 이용해 변수 조합의 경우의 수를 최소화하면서도 균형 있게 조합을 뽑아내는 조합기반 기법으로, 페어와이즈와 같은 계열의 명세기반 조합 테스트 기법이다.
모델 기반 테스트=시스템의 동작을 상태 전이도, 유스케이스 등 모델로 표현하고 그 모델로부터 테스트 케이스를 도출하는 기법으로, 내부 코드가 아닌 외부에서 관찰되는 동작 명세를 기준으로 하므로 명세기반에 해당한다.
기본 경로 테스트=제어 흐름 그래프의 순환복잡도를 이용해 독립적인 실행 경로를 모두 테스트하도록 설계하는 구조기반 기법이다.
분기 커버리지=프로그램의 각 분기(조건문의 참/거짓 분기)가 적어도 한 번씩 실행되도록 테스트하는 구조기반 커버리지 기준이다.

NoteBase

선지별 해설

① ㄱ, ㄴ, ㄹ → X. 기본 경로 테스트(ㄱ)와 분기 커버리지(ㄹ)는 구조기반 기법이지만 페어와이즈 테스트(ㄴ)는 명세기반 조합 기법이므로 서로 다른 유형이 섞여 있다.
② ㄱ, ㄷ, ㄹ → X. 기본 경로 테스트(ㄱ)와 분기 커버리지(ㄹ)는 구조기반 기법인데 모델 기반 테스트(ㄷ)는 명세기반 기법이므로 유형이 섞여 있다.
③ ㄴ, ㄷ, ㅁ → O. 페어와이즈 테스트(ㄴ), 모델 기반 테스트(ㄷ), 직교 배열 테스트(ㅁ)는 모두 내부 코드 구조가 아니라 입력 조합이나 동작 명세를 기준으로 테스트를 설계하는 명세기반(블랙박스) 기법으로 유형이 동일하다.
④ ㄱ, ㄴ, ㄷ, ㄹ → X. 기본 경로 테스트(ㄱ)와 분기 커버리지(ㄹ)는 구조기반, 페어와이즈 테스트(ㄴ)와 모델 기반 테스트(ㄷ)는 명세기반이므로 두 유형이 섞여 있다.

오답 포인트

기본 경로 테스트와 분기 커버리지는 코드의 실행 경로·분기를 직접 대상으로 하는 구조기반 기법이라는 공통점을 놓치기 쉽다. 반대로 페어와이즈, 직교 배열, 모델 기반 테스트는 모두 내부 코드를 보지 않고 입력 조합이나 명세된 동작을 기준으로 테스트 케이스를 뽑아낸다는 공통점이 있다. '커버리지'라는 단어만 보고 모든 항목을 같은 계열로 오인하지 않도록 주의한다.

다시 나오면 이렇게 판단

보기에 제시된 각 기법이 코드 내부 구조(경로·분기·조건)를 대상으로 하는지, 아니면 입력 조합·명세된 동작을 대상으로 하는지를 먼저 표시해 두고, 같은 표시끼리 묶인 선지를 고른다.

암기 포인트

구조기반(화이트박스)=기본 경로 테스트, 분기 커버리지, 조건 커버리지 등 코드 흐름 기준.
명세기반(블랙박스)=페어와이즈 테스트, 직교 배열 테스트, 모델 기반 테스트, 동등 분할, 경계값 분석 등 입력·명세 기준.

NoteBase

문 6. 소프트웨어 개발 작업에 일관적이고 체계적인 구조(framework)를 제공하기 위하여 1995년에 ISO/IEC에서 제정한 소프트웨어 생명주기 공정 국제표준은?

① ISO/IEC 9126
② ISO/IEC 12119
③ ISO/IEC 12207
④ ISO/IEC 25000

정답 3번

출제 포인트

소프트웨어 생명주기(software life cycle) 관련 ISO/IEC 국제표준의 번호와 그 표준이 다루는 대상을 정확히 구분할 수 있는지를 묻는다. 특히 ISO/IEC 12207은 1995년에 제정된 표준으로, 소프트웨어의 개발·운영·유지보수 전 과정에 걸쳐 일관되고 체계적인 구조(framework)와 공정(process)을 정의한다는 점이 핵심 단서이다.

핵심 개념

ISO/IEC 12207=소프트웨어 생명주기 공정(life cycle processes) 표준. 소프트웨어의 취득·공급·개발·운영·유지보수에 필요한 공정, 활동, 작업을 체계적으로 정의하여 소프트웨어 개발 전반에 일관된 프레임워크를 제공한다. / ISO/IEC 9126=소프트웨어 제품 품질 특성(기능성·신뢰성·사용성·효율성·유지보수성·이식성)을 정의한 품질 평가 모델 표준이다. / ISO/IEC 12119=패키지 소프트웨어 제품에 대한 품질 요구사항 및 시험 표준이다. / ISO/IEC 25000=SQuaRE(Software Quality Requirements and Evaluation) 시리즈로, ISO/IEC 9126과 14598을 통합·발전시켜 소프트웨어 품질 요구사항과 평가 방법을 다루는 표준이다.

선지별 해설

① ISO/IEC 9126 → 소프트웨어 제품의 품질 특성을 정의하는 품질 모델 표준이며, 생명주기 공정 전반의 구조를 제공하는 표준이 아니므로 옳지 않다.
② ISO/IEC 12119 → 패키지 소프트웨어 제품의 품질 요구사항과 시험 방법에 관한 표준으로, 생명주기 공정 표준이 아니므로 옳지 않다.
③ ISO/IEC 12207 → 1995년 제정된 소프트웨어 생명주기 공정 국제표준으로, 소프트웨어 개발 작업 전반에 일관되고 체계적인 구조(framework)를 제공한다는 설명과 정확히 일치하므로 옳다.
④ ISO/IEC 25000 → 소프트웨어 품질 요구사항 및 평가(SQuaRE) 시리즈로 품질 관련 표준이며, 생명주기 공정 표준이 아니므로 옳지 않다.

NoteBase

오답 포인트

9126, 12119, 25000은 모두 '품질'과 관련된 표준이라는 공통점 때문에 생명주기 공정 표준인 12207과 혼동하기 쉽다. 문제에서 '일관적이고 체계적인 구조(framework)'와 '생명주기 공정'이라는 표현이 나오면 품질 표준이 아니라 공정(프로세스) 표준을 가리킨다는 점을 구분해야 한다.

다시 나오면 이렇게 판단

지문에 '생명주기', '공정(process)', '프레임워크'라는 단어가 함께 나오면 ISO/IEC 12207을 먼저 떠올린다. 반대로 '품질', '품질 특성', '품질 요구사항', '평가'라는 단어가 나오면 9126·12119·25000 계열임을 구분한 뒤, 대상이 제품 전반의 품질 특성이면 9126, 패키지 제품의 품질 요구사항·시험이면 12119, 통합 품질평가 체계(SQuaRE)이면 25000으로 세분한다.

암기 포인트

12207=생명주기 공정(1995), 9126=품질 특성, 12119=패키지 제품 품질, 25000=SQuaRE(품질 통합체계)로 번호와 주제를 짝지어 암기한다.

NoteBase

문 7. 소프트웨어 아키텍처의 4+1 관점(view)에 대한 설명으로 옳지 않은 것은?

① 유스케이스 관점에서는 외부 행위자에 의해 인식되는 시스템의 기능 요구사항을 보여주는 데 초점을 둔다.

② 논리 관점에서는 계층 구조, 제약 사항, 코드 재사용 등과 같은 시스템 구현을 위한 요건을 보여주는 데 초점을 둔다.

③ 프로세스 관점에서는 독자적인 제어 스레드를 가질 수 있는 액티브 클래스에 초점을 둔다.

④ 배치 관점에서는 물리적인 시스템을 구성하고 있는 각 부분들의 분산 형태와 설치에 초점을 둔다.

정답 2번

출제 포인트

Kruchten의 4+1 아키텍처 관점(view) 모델에서 각 관점(유스케이스, 논리, 프로세스, 개발, 배치)이 무엇을 표현하는지 정확히 구분할 수 있는지를 묻는 문제이다. 특히 논리 관점과 개발 관점의 역할을 혼동하지 않는 것이 핵심이다.

핵심 개념

4+1 뷰 모델은 시스템을 서로 다른 다섯 개의 관점으로 나누어 기술한다. 유스케이스 관점은 다른 네 관점을 하나로 묶는 시나리오 역할을 하며 외부 행위자가 인식하는 기능 요구사항을 보여준다. 논리 관점은 시스템이 사용자에게 제공하는 기능을 설계 관점에서 표현한 것으로, 주요 추상화(클래스, 객체)와 이들 간의 관계, 즉 객체 모델에 초점을 둔다. 개발 관점(구현 관점)은 계층 구조, 서브시스템 분할, 모듈 간 제약 사항, 코드 재사용과 배포 단위 등 소프트웨어 구현·관리 측면을 다룬다. 프로세스 관점은 동시성과 동기화를 다루며 독자적인 제어 흐름을 가지는 액티브 클래스(프로세스·스레드)에 초점을 둔다. 배치 관점(물리 관점)은 소프트웨어가 실행되는 하드웨어 노드에 어떻게 물리적으로 분산·설치되는지를 보여준다.

선지별 해설

① 유스케이스 관점에서는 외부 행위자에 의해 인식되는 시스템의 기능 요구사항을 보여주는 데 초점을 둔다. → O. 유스케이스 관점은 시나리오를 통해 시스템의 기능 요구사항을 외부 행위자의 시각에서 드러낸다.
② 논리 관점에서는 계층 구조, 제약 사항, 코드 재사용 등과 같은 시스템 구현을 위한 요건을 보여주는 데 초점을 둔다. → X. 이는 논리 관점이 아니라 개발(구현) 관점에 대한 설명이다. 논리 관점은 시스템의 기능을 지원하는 주요 클래스·객체와 그 관계(객체 모델)를 보여주는 데 초점을 두며, 계층 구조와 코드 재사용, 모듈 간 제약 사항은 개발 관점이 다루는 내용이다.
③ 프로세스 관점에서는 독자적인 제어 스레드를 가질 수 있는 액티브 클래스에 초점을 둔다. → O. 프로세스 관점은 시스템의 동시성, 동기화, 성능 등을 다루며 독립적으로 실행되는 액티브 클래스를 중심으로 기술한다.
④ 배치 관점에서는 물리적인 시스템을 구성하고 있는 각 부분들의 분산 형태와 설치에 초점을 둔다. → O. 배치 관점은 소프트웨어 요소가 물리적 하드웨어(노드) 위에 어떻게 분산·설치되는지를 보여준다.

NoteBase

오답 포인트

논리 관점과 개발 관점을 헷갈리는 것이 이 문항의 함정이다. 논리 관점은 '무엇을 하는가'에 해당하는 객체·클래스 설계(기능적 요구사항의 구조화)를 다루고, 개발 관점은 '어떻게 코드로 조직하는가'에 해당하는 계층·모듈·재사용·빌드 단위를 다룬다. 계층 구조, 제약 사항, 코드 재사용이라는 키워드가 나오면 개발 관점을 떠올려야 한다.

다시 나오면 이렇게 판단

지문에 계층 구조, 모듈 분할, 코드 재사용, 빌드·배포 단위 같은 키워드가 나오면 개발 관점으로, 클래스·객체·기능의 정적 구조가 나오면 논리 관점으로 판단한다. 동시성·스레드·프로세스 키워드는 프로세스 관점, 하드웨어 노드·분산·설치는 배치 관점, 행위자·시나리오·기능 요구사항은 유스케이스 관점으로 즉시 매칭한다.

암기 포인트

4+1 관점: 유스케이스(시나리오·행위자), 논리(객체·클래스 구조), 개발(계층·모듈·코드 재사용), 프로세스(동시성·액티브 클래스), 배치(하드웨어 분산·설치). '논리=객체 설계, 개발=코드 조직'으로 짝지어 외워 둔다.

NoteBase

문 8. 테스트 주도 개발(Test-driven development)은 애자일 기법에서 개발되는 증분과 해당 증분을 위한 테스트 코드를 함께 작성해 나가는 방법이다. 이에 대한 설명으로 옳지 않은 것은?

① 멀티 스레드를 사용하는 어플리케이션의 테스트를 위해 개발되었다.

② JUnit과 같은 자동화된 테스트 프레임워크 환경이 적극적으로 사용된다.

③ 개발 프로세스의 초기에 결함이 발견될 수 있다.

④ 새로운 코드나 표준 라이브러리의 컴포넌트를 사용하여 기능이 구현되는 신규 소프트웨어 개발에서 가장 가치가 있다.

정답 1번

출제 포인트

테스트 주도 개발(TDD)의 개발 배경과 실무 특징을 정확히 이해하는지를 묻는 문제이다. TDD는 애자일 방법론에서 나온 개발 기법으로, 기능(증분)을 구현하기 전에 그 기능을 검증할 테스트 코드를 먼저 작성하고, 이후 테스트를 통과시키는 최소한의 코드를 작성한 뒤 리팩터링하는 짧은 반복 주기를 특징으로 한다. 이 문제는 TDD에 대한 옳지 않은 설명을 찾는 것이므로, 각 선지가 TDD의 목적·도구·효과·적용 범위 중 어디에 해당하는 서술인지 구분해야 한다.

핵심 개념

TDD의 개발 목적=멀티 스레드 등 특정 실행 환경의 테스트가 아니라, 요구사항을 짧은 반복(증분)으로 나누어 코드와 테스트를 함께 작성함으로써 설계 품질과 결함 조기 발견을 높이려는 것이다. TDD의 실행 환경=JUnit, NUnit 등 자동화된 단위 테스트 프레임워크를 적극 활용해 테스트를 반복 실행하고 회귀를 즉시 확인한다. TDD의 효과=테스트를 먼저 작성하므로 결함이 개발 프로세스 초기 단계에서 드러난다. TDD의 적용 범위=완전히 새로운 코드나 표준 라이브러리의 컴포넌트를 조합해 기능을 구현하는 신규 개발에서 특히 가치가 크며, 반대로 기존 코드를 대규모로 다시 짜야 하는 상황이나 사용자 인터페이스 등 테스트 코드 작성이 어려운 영역에서는 상대적으로 적용이 어렵다.

NoteBase

선지별 해설

① 옳지 않다. TDD는 멀티 스레드 애플리케이션의 테스트만을 위해 개발된 기법이 아니다. TDD는 애자일 개발에서 증분 단위로 기능과 테스트 코드를 함께 작성해 나가는 일반적인 개발 방법으로, 특정 실행 환경(멀티 스레드)을 겨냥해 만들어진 것이 아니다.
② 옳다. TDD를 실천할 때 JUnit과 같은 자동화된 테스트 프레임워크는 테스트를 반복적으로 빠르게 실행하고 결과를 즉시 확인하기 위해 적극적으로 사용된다.
③ 옳다. 테스트를 먼저 작성하고 이를 통과하도록 코드를 작성하므로, 결함이 개발 프로세스의 초기 단계에서 발견될 수 있다.
④ 옳다. TDD는 새로운 코드나 표준 라이브러리의 컴포넌트를 사용해 기능을 구현하는 신규 소프트웨어 개발에서 가장 가치가 있다고 알려져 있다.

오답 포인트

①을 옳은 설명으로 착각하기 쉬운 이유는 '테스트'라는 단어에 이끌려 멀티 스레드 테스트 도구와 TDD를 혼동하기 때문이다. TDD는 실행 환경의 종류(단일 스레드/멀티 스레드)를 겨냥한 기법이 아니라, 개발 순서와 방식(테스트 먼저 작성)에 관한 방법론임을 구분해야 한다.

다시 나오면 이렇게 판단

TDD 문제에서는 먼저 '무엇을 묻는 선지인가(목적/도구/효과/적용 범위)'를 나눠서 본다. 특정 실행 환경이나 특정 기술(멀티 스레드, 분산 시스템 등)을 위해 TDD가 개발되었다는 서술이 나오면 옳지 않은 설명일 가능성이 높다고 판단한다.

암기 포인트

TDD=애자일에서 증분+테스트 코드를 함께 작성하는 방법론(특정 실행 환경 전용 기법 아님). JUnit 등 자동화 테스트 프레임워크 적극 활용. 결함을 개발 초기에 발견. 신규 코드·표준 라이브러리 컴포넌트로 기능을 구현하는 신규 개발에서 가장 가치가 크다.

NoteBase

문 9. CMMI의 5단계 소프트웨어 프로세스 성숙도 모델에서 Level 2에 해당하는 주요 프로세스 영역이 아닌 것은?

① 측정 및 분석(Measurement and analysis)

② 요구사항 개발(Requirements development)

③ 프로세스 및 제품 품질 보증(Process and product quality assurance)

④ 프로젝트 모니터링 및 통제(Project monitoring and control)

정답 2번

출제 포인트

CMMI(Capability Maturity Model Integration)의 성숙도 단계별 핵심 프로세스 영역(KPA, Key Process Area)을 정확히 구분할 수 있는지를 묻는다. 특히 Level 2(관리 단계, Managed)와 Level 3(정의 단계, Defined)에 속하는 프로세스 영역을 혼동하지 않아야 한다.

핵심 개념

CMMI 단계적 표현(Staged representation)은 성숙도를 5단계로 나눈다. Level 1은 초기(Initial) 단계로 별도의 프로세스 영역이 없다. Level 2 관리 단계의 핵심 프로세스 영역은 요구사항 관리, 프로젝트 계획, 프로젝트 모니터링 및 통제, 공급자 계약 관리, 측정 및 분석, 프로세스 및 제품 품질 보증, 형상 관리로 총 7개이다. Level 3 정의 단계에서는 조직 차원의 표준 프로세스를 갖추며, 요구사항 개발, 기술적 해법, 제품 통합, 검증, 확인, 조직 프로세스 정의, 조직 프로세스 개선, 조직 교육, 통합 프로젝트 관리, 위험 관리, 의사 결정 분석 및 해결 등이 포함된다. 여기서 요구사항 관리(Requirements Management)는 Level 2, 요구사항 개발(Requirements Development)은 Level 3라는 점이 자주 출제되는 구분 포인트이다.

선지별 해설

① 측정 및 분석(Measurement and analysis) — Level 2의 핵심 프로세스 영역에 해당한다. 프로젝트와 조직의 목표를 지원하기 위해 측정 지표를 수립하고 데이터를 수집·분석하는 활동이다.

② 요구사항 개발(Requirements development) — Level 2가 아니라 Level 3의 핵심 프로세스 영역이다. 고객 요구·제품 요구·제품 구성요소 요구를 이끌어내고 개발하는 활동으로, Level 2의 요구사항 관리(기존에 합의된 요구사항의 변경을 통제하는 활동)와는 구분된다. 발문이 묻는 'Level 2에 해당하는 것이 아닌 것'에 해당하므로 정답이다.

③ 프로세스 및 제품 품질 보증(Process and product quality assurance) — Level 2의 핵심 프로세스 영역에 해당한다. 수행된 프로세스와 산출물이 정해진 기준과 절차를 준수하는지 객관적으로 평가하는 활동이다.

④ 프로젝트 모니터링 및 통제(Project monitoring and control) — Level 2의 핵심 프로세스 영역에 해당한다. 프로젝트 계획 대비 실제 진행 상황을 추적하고 이탈이 발생하면 시정 조치를 취하는 활동이다.

NoteBase

오답 포인트

요구사항 관리와 요구사항 개발을 같은 것으로 착각하면 오답을 고르게 된다. 요구사항 관리는 이미 합의된 요구사항의 변경 이력과 추적성을 관리하는 활동(Level 2)이고, 요구사항 개발은 요구사항 자체를 도출·분석·문서화하는 활동(Level 3)으로 시점과 성격이 다르다.

다시 나오면 이렇게 판단

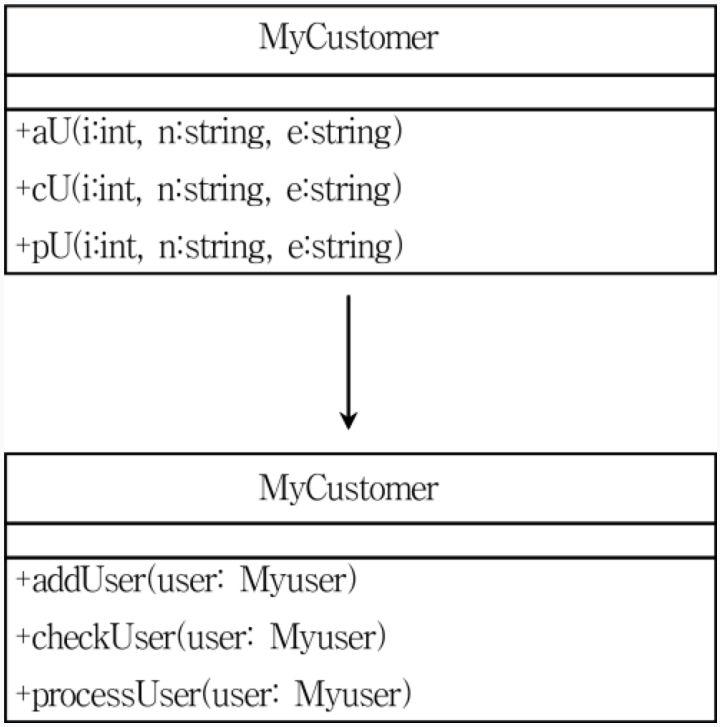
선지에 '관리(Management)'로 끝나는 프로세스 영역인지 '개발(Development)'로 끝나는 프로세스 영역인지부터 확인한다. 이름에 개발·통합·검증·확인·조직(Organizational) 등이 들어가면 대체로 Level 3 이상이고, 계획·모니터링·관리·보증·측정처럼 프로젝트 운영과 직결된 활동은 대체로 Level 2로 분류한다는 감각으로 접근한다.

암기 포인트

CMMI Level 2(관리) 7대 핵심 프로세스 영역: 요구사항 관리, 프로젝트 계획, 프로젝트 모니터링 및 통제, 공급자 계약 관리, 측정 및 분석, 프로세스 및 제품 품질 보증, 형상 관리. 'Level 2=관리·계획·통제·보증·측정', 'Level 3=개발·통합·검증·확인·조직 표준'으로 짝지어 외워 둔다.

NoteBase

문 10. 그림과 같이 MyCustomer 클래스를 대상으로 리팩토링을 실시하였다. 이와 관련 있는 리팩토링 기법만을 모두 고른 것은?



ㄱ. 메소드 추출(Extract Method)

ㄴ. 메소드 상향(Pull Up Method)

ㄷ. 메소드명 변경(Rename Method)

ㄹ. 매개변수 세트를 객체로 전환(Introduce Parameter Object)

ㅁ. 메소드를 매개변수로 전환(Parameterized Method)

① ㄱ, ㄴ

② ㄴ, ㄷ

③ ㄷ, ㄹ

④ ㄱ, ㄷ, ㅁ

정답 3번

출제 포인트

변경 전과 변경 후의 MyCustomer 클래스 다이어그램을 비교하여 실제로 적용된 리팩토링 기법을 찾아내는 문제이다. 메소드 시그니처의 이름과 매개변수 구성이 어떻게 바뀌었는지를 정확히 대조하는 것이 핵심이다.

NoteBase

핵심 개념

변경 전 메소드는 aU(i:int, n:string, e:string), cU(i:int, n:string, e:string), pU(i:int, n:string, e:string)로, 이름이 축약되어 있고 매개변수가 정수·문자열·문자열 세 개의 개별 항목으로 나열되어 있다. 변경 후에는 addUser(user:Myuser), checkUser(user:Myuser), processUser(user:Myuser)로 바뀌었다. 첫째, aU·cU·pU라는 축약된 메소드명이 addUser·checkUser·processUser라는 의미가 분명한 이름으로 바뀌었는데 이는 메소드명 변경(Rename Method)에 해당한다. 둘째, i:int, n:string, e:string이라는 개별 매개변수 세 개가 하나의 객체 매개변수 user:Myuser로 통합되었는데, 이는 서로 관련된 매개변수 묶음을 하나의 객체로 전환하는 매개변수 세트를 객체로 전환(Introduce Parameter Object)에 해당한다.

선지별 해설

① ㄱ, ㄴ(X): 메소드 추출(Extract Method)은 한 메소드 안의 일부 코드를 별도의 새 메소드로 분리하는 기법인데, 그림에서는 메소드의 개수가 3개에서 3개로 그대로이고 내부 코드 분리가 드러나지 않으므로 해당하지 않는다. 메소드 상향(Pull Up Method)은 여러 하위 클래스에 중복된 메소드를 상위 클래스로 옮기는 기법인데, 그림에는 클래스가 MyCustomer 하나뿐이고 상속 관계가 나타나지 않으므로 해당하지 않는다.
② ㄴ, ㄹ(X): 메소드 상향(Pull Up Method)은 위에서 설명한 대로 상속 구조가 있어야 적용되는 기법인데 그림에는 단일 클래스만 제시되어 있으므로 해당하지 않는다. 매개변수 세트를 객체로 전환(Introduce Parameter Object)은 i, n, e라는 개별 매개변수가 user 객체 하나로 합쳐진 부분과 정확히 일치하므로 옳다.
③ ㄷ, ㄹ(O): 메소드명 변경(Rename Method)은 aU→addUser, cU→checkUser, pU→processUser처럼 축약된 이름을 의미가 분명한 이름으로 바꾼 부분과 일치하고, 매개변수 세트를 객체로 전환(Introduce Parameter Object)은 i:int, n:string, e:string 세 매개변수가 user:Myuser 하나의 객체로 통합된 부분과 일치한다. 그림에 나타난 두 가지 변화를 모두 정확히 설명하므로 옳다.
④ ㄱ, ㄷ, ㅁ(X): 메소드 추출(Extract Method)은 ①에서 설명한 대로 해당하지 않는다. 메소드명 변경(Rename Method)은 옳게 포함되어 있지만, 메소드를 매개변수로 전환(Parameterized Method)은 여러 메소드가 유사한 코드를 갖고 있을 때 그 차이를 매개변수로 두어 하나의 메소드로 통합하는 기법인데, 그림에서는 여전히 addUser·checkUser·processUser 세 개의 별도 메소드가 존재하고 하나로 통합되지 않았으므로 해당하지 않는다.

오답 포인트

메소드 상향(Pull Up Method)과 메소드 추출(Extract Method)은 그림에 상속 구조나 메소드 내부 코드 분리가 전혀 나타나지 않는데도 이름이 그럴듯해 보여 고르기 쉬운 함정이다. 또한 메소드를 매개변수로 전환(Parameterized Method)은 여러 메소드를 하나로 합치는 기법인데, 그림은 반대로 메소드 개수가 그대로 유지된 채 이름과 매개변수 형태만 바뀐 것이므로 혼동하지 않아야 한다.

NoteBase

다시 나오면 이렇게 판단

리팩토링 전후 비교 문제는 먼저 클래스의 개수와 상속 관계가 바뀌었는지(상향/하향 계열 기법), 메소드의 개수가 늘거나 줄었는지(추출/통합 계열 기법), 메소드 이름이 바뀌었는지(이름 변경), 매개변수의 형태가 개별 값에서 객체로(또는 그 반대로) 바뀌었는지를 순서대로 확인한다. 이 문항처럼 이름만 바뀌고 매개변수가 객체로 묶였다면 메소드명 변경과 매개변수 세트를 객체로 전환의 조합임을 바로 알 수 있다.

암기 포인트

이름만 바뀌면 메소드명 변경(Rename Method), 여러 개별 매개변수가 하나의 객체로 묶이면 매개변수 세트를 객체로 전환(Introduce Parameter Object), 상속 관계에서 상위로 옮기면 메소드 상향(Pull Up Method), 코드 일부를 떼어내 새 메소드로 만들면 메소드 추출(Extract Method), 여러 메소드를 매개변수 차이로 하나로 합치면 메소드를 매개변수로 전환(Parameterized Method)으로 구분해 기억해 둔다.

NoteBase

문 11. 배열, 레코드, 구조체 등을 매개변수로 사용하는 모듈 사이의 결합도는?

① 데이터 결합도(Data Coupling)

② 스탬프 결합도(Stamp Coupling)

③ 제어 결합도(Control Coupling)

④ 내용 결합도(Content Coupling)

정답 2번

출제 포인트

모듈 간 결합도(Coupling)는 전달되는 인자의 형태와 성격에 따라 등급이 나뉜다. 이 문제는 배열, 레코드, 구조체와 같이 여러 데이터 항목을 묶은 자료구조 전체를 매개변수로 넘길 때 해당하는 결합도 종류를 묻는다.

핵심 개념

결합도는 약한 것부터 강한 것 순으로 자료 결합도(Data Coupling) - 스탬프 결합도(Stamp Coupling) - 제어 결합도(Control Coupling) - 외부 결합도(External Coupling) - 공통 결합도(Common Coupling) - 내용 결합도(Content Coupling)로 나뉜다. 자료 결합도는 단순 자료값(정수, 문자 등 하나의 값) 하나만 매개변수로 주고받는 경우이다. 스탬프 결합도는 배열, 레코드, 구조체처럼 여러 항목을 묶은 자료구조 전체를 그대로 매개변수로 전달하는 경우이며, 이때 호출받는 모듈은 그 구조체 안의 일부 항목만 필요해도 구조체 전체를 받게 된다는 특징이 있다. 제어 결합도는 처리 방식을 결정하는 제어 신호(플래그, 스위치 값)를 매개변수로 넘겨 상대 모듈의 흐름을 제어하는 경우이다. 내용 결합도는 한 모듈이 다른 모듈의 내부 자료나 코드를 직접 참조하거나 수정하는, 가장 강한(나쁜) 결합이다.

선지별 해설

① 데이터 결합도(Data Coupling) — X, 단순 자료(값 하나)만 주고받는 경우이며 배열·레코드·구조체 같은 자료구조 전체를 넘기는 경우가 아니므로 해당하지 않는다.
② 스탬프 결합도(Stamp Coupling) — O, 배열·레코드·구조체 등 여러 자료 항목을 하나로 묶은 자료구조 전체를 매개변수로 사용하는 경우에 해당하는 결합도이다.
③ 제어 결합도(Control Coupling) — X, 자료구조 자체가 아니라 처리 흐름을 지시하는 제어 신호(플래그 등)를 넘기는 경우에 해당한다.
④ 내용 결합도(Content Coupling) — X, 매개변수 전달 방식과 무관하게 한 모듈이 다른 모듈의 내부를 직접 참조·변경하는 경우이다.

오답 포인트

자료 결합도와 스탬프 결합도를 혼동하기 쉬운데, 구분 기준은 넘기는 것이 단순 값 하나인지 아니면 여러 항목을 묶은 자료구조 전체인지이다. 배열, 레코드, 구조체라는 단어가 나오면 바로 스탬프 결합도를 떠올려야 한다.

NoteBase

다시 나오면 이렇게 판단

매개변수의 형태를 먼저 확인한다. 단일 값이면 자료 결합도, 배열·레코드·구조체 등 묶음 자료구조 전체이면 스탬프 결합도, 처리 흐름을 바꾸는 제어 신호이면 제어 결합도, 모듈 내부를 직접 침범하면 내용 결합도로 판단한다.

암기 포인트

결합도 약함에서 강함 순서: 자료(단순 값) - 스탬프(배열·레코드·구조체 전체) - 제어(신호) - 외부 - 공통(전역변수) - 내용(직접 참조). '배열·레코드·구조체=스탬프'로 묶어 외워 둔다.

NoteBase

문 12. 요구분석 단계를 순서대로 바르게 나열한 것은?

ㄱ. 요구사항 검증
ㄴ. 요구사항 명세화
ㄷ. 타당성 조사
ㄹ. 요구사항 추출 및 분석

① ㄷ → ㄹ → ㄱ → ㄴ

② ㄷ → ㄹ → ㄴ → ㄱ

③ ㄹ → ㄱ → ㄷ → ㄴ

④ ㄹ → ㄷ → ㄴ → ㄱ

정답 2번

출제 포인트

요구분석(requirements analysis) 과정을 구성하는 하위 활동들의 표준적인 진행 순서를 알고 있는지 묻는 문제이다. 요구분석은 프로젝트의 실행 가능성을 먼저 따진 뒤 사용자의 요구를 모으고, 이를 정리하여 문서로 만들고, 마지막으로 그 문서가 맞는지 확인하는 흐름으로 진행된다.

올바른 순서

타당성 조사(ㄷ) → 요구사항 추출 및 분석(ㄹ) → 요구사항 명세화(ㄴ) → 요구사항 검증(ㄱ)
먼저 프로젝트를 진행할 가치와 실현 가능성이 있는지 타당성 조사로 확인한다. 타당성이 확보되면 이해관계자와 현장으로부터 요구사항을 수집하고 이를 분석하여 정리한다. 분석된 요구사항은 문서 형태로 명세화하여 개발자와 고객이 함께 볼 수 있게 만든다. 마지막으로 작성된 명세서가 실제 요구사항을 빠짐없이, 모순 없이 반영했는지 검증하는 단계를 거친다.

선지별 해설

① ㄷ → ㄹ → ㄱ → ㄴ: X, 명세화(ㄴ) 이전에 검증(ㄱ)을 두어 순서가 뒤바뀌었다. 검증은 명세서가 작성된 이후에 그 내용을 확인하는 활동이므로 명세화보다 앞설 수 없다.
② ㄷ → ㄹ → ㄴ → ㄱ: O, 타당성 조사로 시작해 요구사항을 추출·분석하고, 이를 명세화한 뒤 마지막에 검증하는 순서로 요구분석의 표준적인 흐름과 일치한다.
③ ㄹ → ㄱ → ㄷ → ㄴ: X, 타당성 조사(ㄷ)보다 요구사항 추출 및 분석(ㄹ)을 먼저 배치했고, 명세화(ㄴ) 전에 검증(ㄱ)을 두어 두 군데 모두 순서가 어긋난다.
④ ㄹ → ㄷ → ㄴ → ㄱ: X, 요구사항 추출 및 분석(ㄹ)을 타당성 조사(ㄷ)보다 앞세웠다. 프로젝트의 실행 가능성 검토가 요구사항 수집보다 먼저 이루어져야 한다.

NoteBase

오답 포인트

타당성 조사를 요구사항 추출·분석보다 뒤에 두거나, 검증을 명세화보다 앞에 두는 두 가지 실수가 오답 선지의 공통된 함정이다. '무엇을 할지 조사(타당성) → 무엇을 원하는지 모으고 정리(추출·분석) → 글로 남기기(명세화) → 맞는지 확인(검증)'이라는 자연스러운 업무 흐름을 떠올리면 순서를 헷갈리지 않는다.

다시 나오면 이렇게 판단

요구분석 단계를 묻는 문제는 각 활동이 '조사 → 수집·정리 → 문서화 → 확인'이라는 네 갈래 중 어디에 속하는지부터 분류한다. 타당성 조사는 항상 맨 앞, 검증은 항상 맨 뒤에 위치한다는 두 가지 고정점만 기억하면 중간의 추출·분석과 명세화의 순서만 판단하면 되므로 오답을 빠르게 걸러낼 수 있다.

암기 포인트

요구분석 4단계: 타당성 조사 → 요구사항 추출 및 분석 → 요구사항 명세화 → 요구사항 검증. 앞글자로 '타-추-명-검'으로 기억해 둔다.

NoteBase

문 13. 시스템을 구성하는 물리적인 노드와 통신 경로, 그리고 컴포넌트의 수행 환경을 표시하는 UML 다이어그램은?

① 배치(deployment) 다이어그램
② 통신(communication) 다이어그램
③ 타이밍(timing) 다이어그램
④ 컴포지트(composite) 다이어그램

정답 1번

출제 포인트

UML 다이어그램 중에서 시스템을 구성하는 물리적인 하드웨어 노드(서버, 장비 등)와 노드 간 통신 경로, 그리고 그 노드 위에서 컴포넌트가 어떻게 배치되어 실행되는지를 표현하는 것이 배치 다이어그램(deployment diagram)이라는 점을 아는지 묻는다. UML 다이어그램의 종류별 용도를 정확히 구분하는 것이 핵심이다.

핵심 개념

배치 다이어그램은 소프트웨어와 하드웨어의 물리적 구성을 나타내는 구조 다이어그램으로, 노드(node)라는 요소로 서버·단말기 등 물리적 자원을 표시하고 노드 사이를 연결선으로 이어 통신 경로를 표시하며, 각 노드 내부에 컴포넌트나 아티팩트를 배치해 실행 환경을 나타낸다. 통신 다이어그램은 객체들 사이의 메시지 교환을 순서와 함께 표현하는 행위 다이어그램이며, 타이밍 다이어그램은 시간 흐름에 따른 객체 상태 변화를 표현하는 행위 다이어그램이다. 컴포지트 구조 다이어그램은 클래스나 컴포넌트의 내부 구조와 그 내부 부분들 사이의 협력 관계를 표현하는 구조 다이어그램이다.

선지별 해설

① 배치(deployment) 다이어그램: O. 물리적인 노드, 노드 간 통신 경로, 그 위에서 수행되는 컴포넌트의 실행 환경을 표시하는 다이어그램으로 발문의 설명과 정확히 일치한다.
② 통신(communication) 다이어그램: X. 객체들 사이에 주고받는 메시지의 순서와 상호작용을 표현하는 행위 다이어그램으로, 물리적 노드나 실행 환경을 표시하지 않는다.
③ 타이밍(timing) 다이어그램: X. 시간의 흐름에 따라 객체나 역할의 상태가 어떻게 변화하는지를 표현하는 행위 다이어그램으로, 물리적 하드웨어 구성과는 관련이 없다.
④ 컴포지트(composite) 다이어그램: X. 클래스나 컴포넌트의 내부 구조와 부분들 사이의 연결 관계를 표현하는 구조 다이어그램으로, 노드나 통신 경로가 아니라 논리적인 내부 구성을 다룬다.

NoteBase

오답 포인트

통신·타이밍 다이어그램을 배치 다이어그램과 혼동하기 쉬운데, 이 둘은 모두 행위 다이어그램으로 객체 간 상호작용이나 상태 변화의 시간적 흐름을 다룰 뿐 물리적 하드웨어나 실행 환경과는 무관하다. 컴포지트 다이어그램도 구조 다이어그램이라는 점에서 배치 다이어그램과 혼동될 수 있지만, 컴포지트 다이어그램은 하나의 요소 내부 구성을 다루고 배치 다이어그램은 여러 물리적 노드 사이의 배치를 다룬다는 차이가 있다.

다시 나오면 이렇게 판단

발문에 물리적 노드, 통신 경로, 실행 환경이라는 키워드가 함께 나오면 배치 다이어그램을 먼저 떠올린다. 메시지 순서나 상호작용이 언급되면 순차·통신 다이어그램, 시간에 따른 상태 변화가 언급되면 타이밍 다이어그램, 내부 부분 간 구조적 연결이 언급되면 컴포지트 구조 다이어그램으로 구분한다.

암기 포인트

배치(deployment) 다이어그램=노드+통신경로+실행환경(물리적 구성), 통신 다이어그램=메시지 교환 순서, 타이밍 다이어그램=시간에 따른 상태 변화, 컴포지트 다이어그램=내부 구조와 부분 간 협력 관계로 구분해 암기한다.

NoteBase

문 14. 프로그램 인스펙션(inspection)을 통해 결함을 검출하고자 할 때, 결함 유형에 따른 검사항목을 옳게 짝지은 것은?

① 데이터 결함 - 모든 반복문은 확실히 종료되는가?

② 제어 결함 - 버퍼 오버플로우의 가능성이 있는가?

③ 인터페이스 결함 - 매개변수들의 순서가 올바른가?

④ 메모리 관리 결함 - 예상하지 않은 입력이 문제를 일으킬 수 있는가?

정답 3번

출제 포인트

프로그램 인스펙션(코드 검사)에서 결함 유형별로 실제 어떤 항목을 점검하는지를 정확히 짝지을 수 있는지를 묻는다. 결함 유형(데이터·제어·인터페이스·메모리 관리·입출력 등)과 그 유형에 해당하는 전형적인 점검 질문을 서로 바꿔치기해 혼동을 유도하는 문제이다.

핵심 개념

데이터 결함=변수의 선언·초기화·사용 값이 올바른지(예: 모든 변수가 사용 전에 초기화되었는가)를 점검한다. 제어 결함=조건문·반복문의 흐름이 올바른지(예: 모든 반복문이 확실히 종료되는가, 조건식이 맞는 변수를 검사하는가)를 점검한다. 인터페이스 결함=함수·메서드 호출 시 매개변수의 개수·형(type)·순서가 맞는지를 점검한다. 메모리 관리 결함=포인터·동적 할당·연결 구조가 올바르게 해제·재할당되는지(예: 버퍼 오버플로우 가능성, 메모리 누수)를 점검한다. 입출력 결함=입력 변수가 모두 사용되었는지, 예상하지 않은 입력이 문제를 일으키지 않는지를 점검한다.

선지별 해설

① X: '모든 반복문이 확실히 종료되는가'는 반복문의 흐름 제어를 확인하는 항목이므로 제어 결함의 점검 항목이다. 데이터 결함의 점검 항목이 아니다.
② X: '버퍼 오버플로우의 가능성이 있는가'는 메모리 공간의 할당·해제와 관련된 항목이므로 메모리 관리 결함의 점검 항목이다. 제어 결함의 점검 항목이 아니다.
③ O: '매개변수들의 순서가 올바른가'는 함수·메서드 호출 시 인터페이스(연결 부분)가 맞는지를 확인하는 전형적인 항목이므로 인터페이스 결함의 점검 항목으로 옳게 짝지어져 있다.
④ X: '예상하지 않은 입력이 문제를 일으킬 수 있는가'는 입력 값 처리와 관련된 항목이므로 입출력 결함의 점검 항목이다. 메모리 관리 결함의 점검 항목이 아니다.

NoteBase

오답 포인트

결함 유형의 이름만 보고 넘어가지 않고, 뒤에 붙은 점검 질문이 실제로 그 유형이 다루는 범위(변수 값=데이터, 흐름=제어, 호출 관계=인터페이스, 할당·해제=메모리 관리, 입력값=입출력)에 속하는지를 하나씩 대조해야 한다. 반복문 종료 여부와 버퍼 오버플로우처럼 서로 다른 결함 유형에 속하는 점검 항목을 뒤바꿔 놓는 함정에 주의한다.

다시 나오면 이렇게 판단

선지의 앞부분(결함 유형)을 가리고 뒷부분의 점검 질문만 읽어, 그 질문이 실제로 다루는 대상(변수 초기화=데이터, 조건·반복 흐름=제어, 매개변수 개수·순서·형=인터페이스, 메모리 할당·해제·오버플로우=메모리 관리, 입력값 처리=입출력)이 무엇인지 먼저 판단한 뒤, 앞부분의 유형명과 일치하는지 대조한다.

암기 포인트

인터페이스 결함은 함수·메서드 '호출' 관련(매개변수 개수·순서·형), 메모리 관리 결함은 '할당·해제·오버플로우' 관련, 제어 결함은 '조건·반복 흐름' 관련, 입출력 결함은 '입력값 처리' 관련으로 구분해 암기해 둔다.

NoteBase

문 15. Iterator 패턴에 대한 설명으로 옳지 않은 것은?

① 집합 객체의 요소들에 대해 순서대로 접근하는 방법을 제공한다.

② 배열(Array), 배열리스트(ArrayList), 해시 테이블과 같은 객체를 처리하는 데 사용하는 패턴이다.

③ 서로 다른 인터페이스를 사용하기 때문에 각각의 객체를 참조하기 위한 다형성 코드(polymorphic code)를 개발하는 것이 불가능하다.

④ 서로 다른 집합 객체 구조에 대해 동일한 방법으로 순회할 수 있다.

정답 3번

출제 포인트

Iterator(반복자) 패턴의 목적과 동작 방식을 정확히 알고 있는지 묻는 문제이다. Iterator 패턴은 집합 객체(컬렉션)의 내부 구조를 노출하지 않으면서 그 안의 요소들에 순차적으로 접근할 수 있는 방법을 제공하는 행위(behavioral) 디자인 패턴이다. 핵심은 배열, 리스트, 해시 테이블처럼 서로 다른 내부 구조를 가진 집합 객체라도 클라이언트가 동일한 순회 인터페이스(예: hasNext(), next())로 접근할 수 있게 한다는 점이다.

핵심 개념

Iterator 패턴은 집합 객체(Aggregate)와 반복자(Iterator)를 분리한다. 각 집합 객체는 자신에 맞는 구체적인 Iterator 구현체를 제공하지만, 모든 Iterator는 공통의 인터페이스를 따른다. 그 결과 클라이언트 코드는 순회 대상이 배열이든 배열리스트이든 해시 테이블이든 상관없이 동일한 방식으로 순회하는 다형적(polymorphic) 코드를 작성할 수 있다. 즉 서로 다른 내부 인터페이스를 가진 객체들이라도, 공통 Iterator 인터페이스 덕분에 다형성 코드 개발이 오히려 가능해지는 것이 이 패턴의 존재 이유이다.

선지별 해설

① O — Iterator 패턴은 집합 객체 내부의 요소들에 대해 순서대로(순차적으로) 접근하는 방법을 제공하는 패턴이 맞다.
② O — 배열, 배열리스트, 해시 테이블처럼 서로 다른 내부 구조를 가진 객체들을 처리(순회)하기 위해 사용하는 패턴이 맞다.
③ X — 틀린 표현 → 올바른 표현: "다형성 코드를 개발하는 것이 불가능하다" → "다형성 코드를 개발하는 것이 가능하다." 각 집합 객체가 서로 다른 내부 인터페이스를 사용하더라도, Iterator 패턴은 공통의 반복자 인터페이스를 제공하므로 클라이언트는 객체의 종류에 상관없이 동일한 방식으로 참조하는 다형성 코드를 작성할 수 있다. 이것이 바로 Iterator 패턴을 쓰는 이유이므로 지문의 서술은 패턴의 목적과 정반대이다.
④ O — 서로 다른 집합 객체 구조에 대해 동일한 방법(공통 인터페이스)으로 순회할 수 있다는 것이 Iterator 패턴의 핵심 효과가 맞다.

NoteBase

오답 포인트

③을 옳은 설명으로 착각하기 쉬운 이유는 "서로 다른 인터페이스를 사용하기 때문에"라는 전제 자체는 맞는 말이기 때문이다. 그러나 결론이 반대로 서술되어 있다. Iterator 패턴은 그 서로 다른 내부 구조에도 불구하고 공통 인터페이스를 통해 다형성 코드 작성을 가능하게 만드는 패턴이므로, "불가능하다"가 아니라 "가능하다"가 되어야 한다. ①②④는 모두 Iterator 패턴의 정의와 목적을 그대로 서술한 옳은 문장이라는 점도 함께 확인해 둔다.

다시 나오면 이렇게 판단

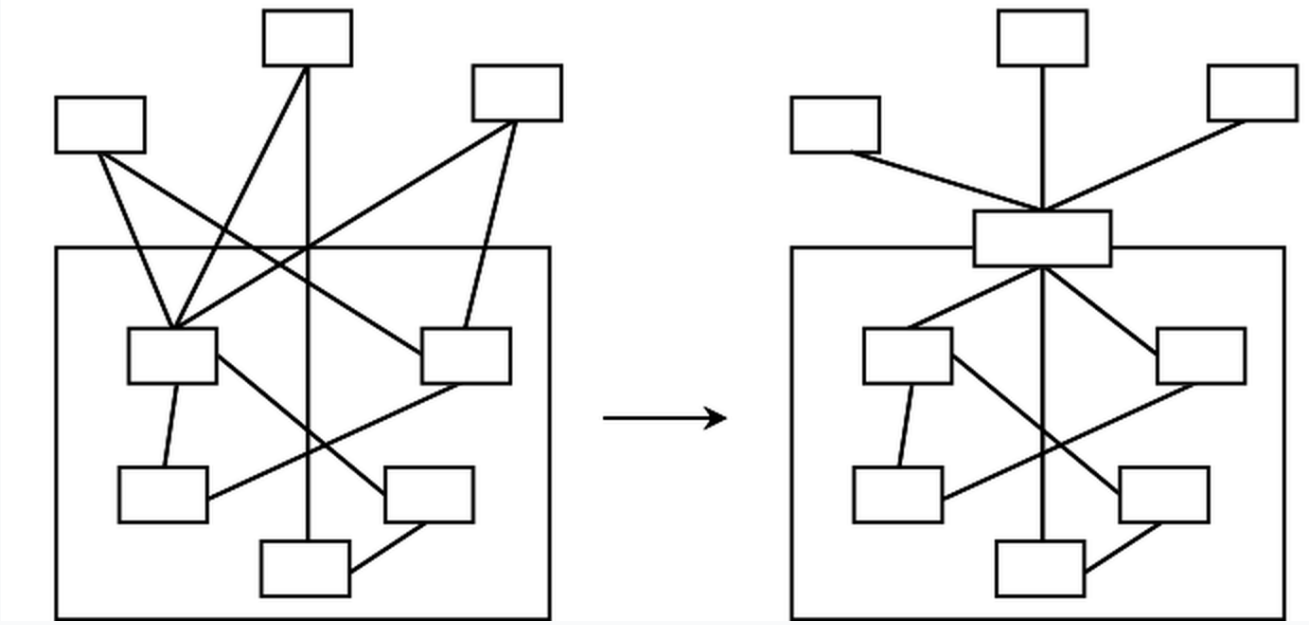
Iterator 패턴 관련 문제는 항상 "공통 인터페이스로 서로 다른 집합 구조를 동일하게 순회할 수 있는가"를 기준으로 판단한다. 지문에서 "~할 수 없다", "~불가능하다"처럼 패턴의 목적을 부정하는 서술이 나오면 우선 의심한다. Iterator 패턴의 존재 이유 자체가 서로 다른 구조를 통일된 방식으로 다루기 위함이므로, 이를 부정하는 문장은 대부분 오답 선지이다.

암기 포인트

Iterator 패턴 = 집합 객체의 내부 구조를 몰라도 순차 접근이 가능하게 하는 패턴, 서로 다른 집합 객체(배열·리스트·해시 테이블 등)를 공통 인터페이스(hasNext/next)로 동일하게 순회, 서로 다른 인터페이스를 가진 객체라도 다형성 코드 작성이 가능해지는 것이 핵심 효과라고 정리해 둔다.

NoteBase

문 16. 그림과 같이 서브시스템 사이의 의사소통 및 종속성을 최소화하기 위하여 단순화된 하나의 인터페이스를 제공하는 디자인 패턴은?



① Adapter 패턴

② Bridge 패턴

③ Decorator 패턴

④ Facade 패턴

정답 4번

출제 포인트

왼쪽 그림처럼 외부의 여러 객체가 서브시스템 내부의 각 모듈과 직접 여러 갈래로 얽혀 통신하면 의사소통 경로가 많아지고 서브시스템 내부 변경이 외부에 그대로 영향을 미친다. 오른쪽 그림은 이 복잡한 연결을 걷어내고 외부 객체들이 서브시스템 상단에 새로 놓인 하나의 창구 역할 객체만을 거쳐 접근하도록 바꾼 상태이다. 이렇게 여러 서브시스템(모듈)에 대한 접근을 단순화된 하나의 통합 인터페이스로 감싸 외부와 내부 사이의 의사소통 경로 수와 종속성을 줄여 주는 구조적 디자인 패턴이 무엇인지 묻고 있다.

NoteBase

선지별 해설

① Adapter 패턴: X. 서로 호환되지 않는 인터페이스를 가진 두 클래스 사이에 변환기를 두어 하나의 인터페이스를 다른 인터페이스로 바꿔 주는 패턴이다. 그림처럼 여러 서브시스템 모듈을 하나로 묶어 단순화하는 목적이 아니라 인터페이스 형식 불일치를 해소하는 목적이므로 그림과 맞지 않는다.
② Bridge 패턴: X. 추상화 부분과 구현 부분을 분리하여 각각을 독립적으로 확장할 수 있게 하는 패턴이다. 서브시스템 접근을 단일 창구로 단순화하는 그림의 의도와는 다르다.
③ Decorator 패턴: X. 객체에 새로운 책임(기능)을 동적으로 덧붙일 때 사용하는 패턴으로, 그림에서 보이는 서브시스템 간 연결 단순화와는 관련이 없다.
④ Facade 패턴: O. 여러 서브시스템(모듈)에 대한 개별 접근을 감추고 하나의 통합된 상위 인터페이스(창구)를 제공하여, 외부 객체가 서브시스템 내부 구조를 알 필요 없이 단순화된 창구 하나만 사용하게 하는 패턴이다. 이는 그림 오른쪽에서 상단에 새로 등장한 단일 박스가 모든 외부 연결을 받아 서브시스템 내부로 전달하는 모습과 정확히 일치한다.

오답 포인트

Adapter는 '인터페이스 형식 변환', Facade는 '여러 서브시스템 접근의 단일 창구화'로 목적이 다른데, 둘 다 '인터페이스를 다룬다'는 점만 보고 혼동하기 쉽다. 그림에 여러 개의 서브시스템 모듈과 여러 외부 객체가 등장하고, 변경 후 그 모두를 대표하는 박스가 하나만 추가된 구조라면 Adapter가 아니라 Facade로 판단해야 한다.

다시 나오면 이렇게 판단

그림에서 서브시스템 내부 모듈 개수가 여럿이고 외부와의 복잡한 다대다 연결이 변경 후 하나의 창구 박스로 모이는 형태라면 Facade 패턴이다. 반대로 두 클래스의 인터페이스 형식이 안 맞아 중간에 변환 객체 하나가 끼워지는 단순한 1대1 구조라면 Adapter 패턴으로 구분한다.

암기 포인트

Facade=여러 서브시스템을 하나의 단순화된 인터페이스로 감싸 의사소통·종속성을 최소화, Adapter=호환되지 않는 인터페이스 형식을 변환, Bridge=추상화와 구현의 분리, Decorator=객체에 기능을 동적으로 추가.

NoteBase

문 17. 통합 테스트에 대한 설명으로 옳지 않은 것은?

① 회귀 테스트는 복잡하고 시간이 중요한 프로젝트에 적용하면 효율적이다.

② 점증적인 통합은 빅뱅 방식에 비해 인터페이스를 완전하게 테스트할 가능성이 더 높다.

③ 객체지향 시스템의 드라이버는 사용자 인터페이스를 대체하여 사용함으로써 인터페이스의 구현 이전에 기능성 테스트를 수행할 수 있다.

④ 객체지향 시스템의 스텁은 하나 이상의 협동 클래스들이 아직 완전히 구현되지 않은 상황에서 사용할 수 있다.

정답 1번

출제 포인트

통합 테스트(integration test)의 기본 개념을 정확히 아는지를 묻는 문제다. 회귀 테스트의 실무적 한계, 점증적 통합과 빅뱅 통합의 차이, 객체지향 시스템에서 드라이버와 스텁의 역할을 각각 구분해서 알아야 한다.

핵심 개념

회귀 테스트(regression test)는 코드가 변경될 때마다 기존 테스트 케이스 전체 또는 일부를 재실행해 새로운 결함이 발생하지 않았는지 확인하는 활동이다. 규모가 크고 복잡하며 일정이 촉박한(시간이 중요한) 프로젝트일수록 테스트 케이스 전체를 매번 재실행하는 것은 시간과 자원이 많이 들어 비효율적이므로, 변경 영향을 받는 부분을 선별해 재테스트하는 전략이 필요하다. 즉 회귀 테스트는 복잡하고 시간이 중요한 프로젝트에서 오히려 적용하기 어렵고 비효율적인 활동이다.
점증적 통합(incremental integration)은 모듈을 하나씩 순차적으로 결합하면서 테스트하는 방식으로, 결합할 때마다 인터페이스 오류를 국지적으로 확인할 수 있어 빅뱅 방식(전체 모듈을 한꺼번에 결합해 테스트하는 방식)보다 인터페이스를 더 완전하게 검증할 가능성이 높다.
객체지향 시스템의 통합 테스트에서 드라이버(driver)는 아직 만들어지지 않았거나 통합되지 않은 상위 모듈(예: 사용자 인터페이스)을 대신하여 하위 모듈을 호출해 주는 테스트용 코드이며, 스텁(stub)은 아직 완전히 구현되지 않은 협동 클래스(하위/협력 모듈)를 대신하여 최소한의 응답만 돌려주는 테스트용 코드이다.

NoteBase

선지별 해설

① X - 회귀 테스트는 복잡하고 시간이 중요한 프로젝트에 적용하면 효율적이다 → 오히려 그러한 프로젝트에서는 테스트 케이스 전체를 매번 재실행하기 어려워 비효율적이다(선별적 재테스트 전략이 필요하다).
② O - 점증적 통합은 결합 단위마다 인터페이스를 순차적으로 검증하므로, 전체를 한꺼번에 결합하는 빅뱅 방식보다 인터페이스를 완전하게 테스트할 가능성이 더 높다.
③ O - 객체지향 시스템의 드라이버는 아직 구현되지 않은 사용자 인터페이스를 대신하는 역할을 하므로, 인터페이스 구현 이전에도 기능성 테스트를 수행할 수 있게 해 준다.
④ O - 객체지향 시스템의 스텁은 아직 완전히 구현되지 않은 협동 클래스가 있어도, 그 클래스를 대신하여 하위 기능을 테스트할 수 있게 해 준다.

오답 포인트

①을 옳다고 잘못 고르기 쉬운 이유는 '회귀 테스트가 효율적이다'라는 서술만 보고 회귀 테스트 자체의 필요성과 헷갈리기 때문이다. 회귀 테스트가 중요하다는 것과, 복잡하고 시간이 중요한 프로젝트에서 전체 재실행이 효율적이라는 것은 별개의 문제다. 후자는 명백히 비효율적이라는 점을 구분해야 한다. 또한 드라이버와 스텁의 역할(드라이버=상위/UI 대체, 스텁=하위/협동 클래스 대체)을 반대로 기억하면 ③, ④를 혼동할 수 있다.

다시 나오면 이렇게 판단

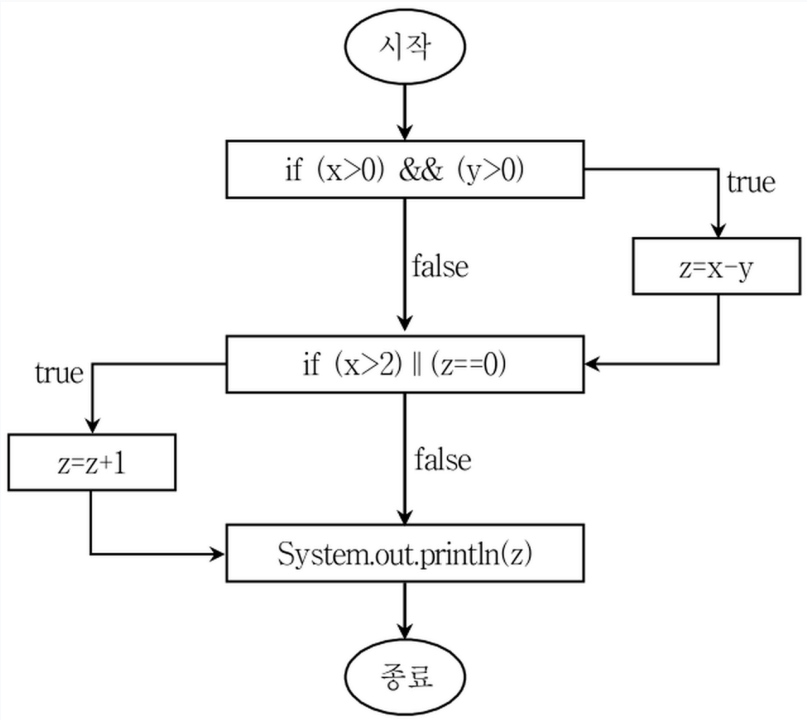
'회귀 테스트'와 관련한 선지가 나오면 항상 '전체 재실행의 비효율성'과 '선별적 재테스트의 필요성'을 먼저 떠올린다. '점증적 통합 vs 빅뱅' 비교 선지는 인터페이스 검증의 완전성 측면에서 점증적 방식이 우월하다는 방향으로 판단한다. '드라이버/스텁' 선지는 드라이버=상위 모듈(UI) 대체, 스텁=하위 협동 클래스 대체라는 역할 매핑을 기준으로 옳고 그름을 가린다.

암기 포인트

회귀 테스트: 복잡·시간 촉박한 프로젝트 → 전체 재실행은 비효율적(선별 재테스트 필요). 점증적 통합 > 빅뱅(인터페이스 검증 완전성). 드라이버 = 상위(UI) 대체, 스텁 = 하위(협동 클래스) 대체.

NoteBase

문 18. 다음 제어 흐름 그래프에 나타난 프로그램을 테스트할 때, 옳지 않은 것은?



① 분기 커버리지를 만족하는 최소의 테스트 케이스는 2개이다.

② 기본 경로의 개수는 3개이다.

③ 문장 커버리지를 만족하는 최소의 테스트 케이스는 1개이다.

④ {(x:1, y:2, z:0), (x:5, y:0, z:0)}은 분기 커버리지를 만족하지 못한다.

정답 4번

출제 포인트

제어 흐름 그래프(CFG)를 바탕으로 문장 커버리지·분기 커버리지 같은 화이트박스 테스트 기준과 기본 경로(사이클로매틱 복잡도) 개수를 계산하고, 주어진 테스트 케이스가 실제로 분기 커버리지를 만족하는지 직접 대입해 검증하는 능력을 확인하는 문제이다.

NoteBase

조건 정리

그래프의 흐름은 다음과 같다.
시작 → 조건1 if (x>0) && (y>0)
조건1이 참이면 z=x-y를 실행한 뒤 조건2로 진입한다.
조건1이 거짓이면 z=x-y를 거치지 않고 곧바로 조건2로 진입한다(이때 z는 테스트 케이스가 준 초기값을 그대로 유지한다).
조건2 if (x>2) || (z==0)
조건2가 참이면 z=z+1을 실행한 뒤 print(z)로 간다.
조건2가 거짓이면 곧바로 print(z)로 간다.
print(z) → 종료.
결정 노드는 조건1, 조건2 두 개이다.

사용할 공식

기본 경로(basis path)의 개수는 사이클로매틱 복잡도로 구하며, 노드 수를 N, 간선 수를 E라 하면 $V(G) = E - N + 2$ 이고, 이는 구조적 프로그램에서 결정 노드 수에 1을 더한 값과 같다.

계산 과정

노드는 시작·조건1·z=x-y·조건2·z=z+1·print·종료의 7개이고, 간선은 시작→조건1, 조건1참→z=x-y, 조건1거짓→조건2, z=x-y→조건2, 조건2참→z=z+1, 조건2거짓→print, z=z+1→print, print→종료의 8개이다. 따라서 $V(G) = 8 - 7 + 2 = 3$이며, 결정 노드가 조건1·조건2 2개이므로 $2 + 1 = 3$으로도 같은 값이 나온다. 기본 경로는 3개이므로 ②는 옳은 설명이다.
분기 커버리지는 조건1과 조건2 각각의 참·거짓을 최소 한 번씩만 지나면 되므로, 테스트 케이스 2개로도 두 조건의 참·거짓을 모두 채울 수 있다. 예를 들어 (x=5,y=3)은 조건1 참→z=2, 조건2도 참(x>2)이 되고, (x=0,y=0,z=1)은 조건1 거짓, 조건2도 거짓(x>2 거짓, z==0 거짓)이 되어 두 케이스만으로 조건1참·조건1거짓·조건2참·조건2거짓이 모두 나온다. 그러므로 최소 2개로 분기 커버리지를 만족할 수 있어 ①은 옳은 설명이다.
문장 커버리지는 그래프의 모든 문장(노드)을 한 번씩만 지나면 충분하므로, 조건1 참, 조건2 참 경로 하나(z=x-y와 z=z+1, print를 모두 지나는 경로)만으로 전체 문장이 한 번씩 실행된다. 따라서 최소 1개의 테스트 케이스로 문장 커버리지를 만족할 수 있어 ③은 옳은 설명이다.
④를 검증하면, 첫 번째 케이스 (x:1, y:2, z:0)은 조건1이 1>0 참, 2>0 참이므로 참이 되어 z=x-y=1-2=-1이 되고, 조건2는 x>2(1>2, 거짓), z==0(-1==0, 거짓)이므로 거짓이 되어 그대로 print(-1)한다. 즉 이 케이스는 조건1-참, 조건2-거짓을 지난다. 두 번째 케이스 (x:5, y:0, z:0)은 조건1이 5>0 참, 0>0 거짓이므로 전체가 거짓이 되어 z=x-y를 거치지 않고 z는 입력값 0을 그대로 유지한 채 조건2로 간다. 조건2는 x>2(5>2, 참)이므로 참이 되어 z=z+1=1이 되고 print(1)한다. 즉 이 케이스는 조건1-거짓, 조건2-참을 지난다. 두 케이스를 합치면 조건1의 참·거짓과 조건2의 참·거짓이 모두 한 번씩 나타나므로 이 테스트 케이스 집합은 분기 커버리지를 만족한다. 따라서 "분기 커버리지를 만족하지 못한다"는 ④의 설명은 틀렸다.

NoteBase

선지 판단

① 분기 커버리지를 만족하는 최소의 테스트 케이스는 2개이다 → O. 조건1·조건2 각각의 참·거짓을 두 개의 테스트 케이스 조합으로 모두 채울 수 있다.
② 기본 경로의 개수는 3개이다 → O. 노드 7개, 간선 8개인 그래프의 사이클로매틱 복잡도는 $8 - 7 + 2 = 3$ 이다.
③ 문장 커버리지를 만족하는 최소의 테스트 케이스는 1개이다 → O. 조건1-참, 조건2-참 경로 하나로 모든 문장이 한 번씩 실행된다.
④ {(x:1, y:2, z:0), (x:5, y:0, z:0)}은 분기 커버리지를 만족하지 못한다 → X. 실제로 대입해 보면 첫 번째 케이스는 조건1-참·조건2-거짓, 두 번째 케이스는 조건1-거짓·조건2-참을 지나 두 조건의 참·거짓이 모두 채워지므로 이 테스트 케이스 집합은 분기 커버리지를 만족한다.

오답 포인트

조건문에 &&나 ||로 여러 조건이 묶여 있어도 분기 커버리지는 그 조건식 전체의 참·거짓만 기준으로 삼는다는 점을 놓치면 잘못 판단하기 쉽다. 또한 조건1이 거짓일 때는 z=x-y가 실행되지 않고 z가 테스트 케이스에서 준 초기값을 그대로 유지한다는 점을 놓치면, 두 번째 케이스에서 조건2의 참·거짓 판정을 잘못 계산하게 된다.

다시 나오면 이렇게 판단

그래프를 노드와 간선으로 세어 $V(G) = E - N + 2$ 공식으로 기본 경로 수를 구하고, 분기 커버리지 문제는 주어진 테스트 케이스를 실제로 조건문에 대입해 각 조건의 참·거짓 결과표를 만든 뒤, 모든 결정 노드의 참·거짓이 한 번씩 나오는지 직접 확인한다.

암기 포인트

커버리지 강도는 문장 커버리지 < 분기 커버리지 < 조건 커버리지 순으로 강해진다는 점, 기본 경로(사이클로매틱 복잡도)는 $V(G) = E - N + 2$이며 구조적 프로그램에서는 결정 노드 수에 1을 더한 값과 같다는 점을 기억해 둔다.

NoteBase

문 19. 오픈 소스 도구와 기능을 연결한 것으로 옳지 않은 것은?

① Cucumber - 행위 주도 개발(Behavior Driven Development) 지원 도구

② Maven - 프로젝트 관리 도구

③ Git - 분산 버전 관리 도구

④ JaCoCo - 자바 소스 보안 취약점 분석 도구

정답 4번

출제 포인트

오픈소스 개발 도구의 명칭과 실제 기능을 정확히 매칭할 수 있는지 묻는 문제이다. 빌드·형상관리·테스트 관련 도구들의 용도를 구분하는 것이 핵심이다.

핵심 개념

Cucumber는 자연어에 가까운 시나리오 문장으로 테스트 케이스를 작성해 실행하는 행위 주도 개발(BDD) 지원 도구이다. Maven은 프로젝트의 빌드 절차와 라이브러리 의존성을 관리하는 프로젝트 관리 도구이다. Git은 분산 방식으로 소스 코드의 변경 이력을 관리하는 버전 관리 도구이다. JaCoCo(Java Code Coverage)는 테스트 코드가 실제 소스 코드의 어느 부분을 얼마나 실행했는지 측정하는 코드 커버리지 측정 도구이며, 소스의 보안 취약점을 분석하는 도구가 아니다.

선지별 해설

① Cucumber - 행위 주도 개발(BDD) 지원 도구: O. Cucumber는 요구사항을 자연어 시나리오로 표현하고 이를 바탕으로 자동화 테스트를 수행하는 BDD 지원 도구가 맞다.
② Maven - 프로젝트 관리 도구: O. Maven은 빌드 자동화와 의존성(라이브러리) 관리를 담당하는 프로젝트 관리 도구가 맞다.
③ Git - 분산 버전 관리 도구: O. Git은 중앙 서버 없이도 각 개발자가 저장소 전체 이력을 가지는 분산 버전 관리 도구가 맞다.
④ JaCoCo - 자바 소스 보안 취약점 분석 도구: X. JaCoCo는 보안 취약점 분석 도구가 아니라, 단위 테스트가 자바 소스 코드를 얼마나 실행했는지를 측정하는 코드 커버리지 측정 도구이다. → 올바른 표현: JaCoCo - 자바 소스 코드 커버리지 측정 도구.

오답 포인트

도구 이름의 앞글자나 어감만으로 기능을 추측하면 안 된다. JaCoCo를 'Ja(va) Co(de) Co(verage)'로 정확히 풀어 기억하지 않으면 이름만 보고 보안 관련 도구로 착각하기 쉽다. 오픈소스 도구 문제는 각 도구의 실제 산출물(무엇을 측정·생성·관리하는지)을 기준으로 판단해야 한다.

NoteBase

다시 나오면 이렇게 판단

도구 이름이 낯설게 느껴져도 카테고리(빌드 관리, 버전 관리, 테스트, 커버리지, 정적분석 등)로 먼저 분류한 뒤 각 선지의 설명이 그 카테고리와 맞는지 확인한다. 코드 커버리지 도구(JaCoCo, Cobertura 등)와 정적 분석·보안 취약점 분석 도구(SonarQube, FindBugs, Fortify 등)는 서로 다른 범주이므로 혼동하지 않는다.

암기 포인트

Cucumber=BDD 테스트 시나리오, Maven=빌드·의존성 관리, Git=분산 버전 관리, JaCoCo=자바 코드 커버리지 측정(보안 취약점 분석 아님)으로 각각 한 단어씩 짝지어 외워 둔다.

NoteBase

문 20. 국제표준과 관련된 내용으로 옳지 않은 것은?

① IEEE 830 - 소프트웨어 요구명세서 표준

② IEEE 1219 - 소프트웨어 유지보수 표준

③ IEC 62304 - 항공기 소프트웨어 안전성 표준

④ ISO 26262 - 자동차 기능 안전성 표준

정답 3번

출제 포인트

소프트웨어공학 분야에서 자주 등장하는 국제표준(IEEE, IEC, ISO 계열)의 표준번호와 그 표준이 다루는 대상을 정확히 짝지을 수 있는지를 묻는다. 표준번호만 보고 적용 분야를 혼동하기 쉬운 부분을 노린 문제이다.

핵심 정리

IEEE 830은 소프트웨어 요구사항명세서(SRS) 작성 지침을 다루는 표준이다. IEEE 1219는 소프트웨어 유지보수 프로세스를 규정하는 표준이다. IEC 62304는 의료기기 소프트웨어의 생명주기 프로세스, 즉 의료기기 소프트웨어 안전성을 다루는 표준이며 항공기 소프트웨어와는 무관하다. ISO 26262는 자동차 전기·전자 시스템의 기능 안전성을 다루는 표준이다. 항공기 소프트웨어 안전성 분야의 대표적 표준은 IEC 62304가 아니라 DO-178C(RTCA/EUROCAE 공동표준)이다.

선지별 해설

① IEEE 830 - 소프트웨어 요구명세서 표준: O, 소프트웨어 요구사항명세서(SRS)의 작성 형식과 내용을 규정하는 표준으로 맞는 설명이다.
② IEEE 1219 - 소프트웨어 유지보수 표준: O, 소프트웨어 유지보수 활동과 프로세스를 규정하는 표준으로 맞는 설명이다.
③ IEC 62304 - 항공기 소프트웨어 안전성 표준: X, IEC 62304는 항공기가 아니라 의료기기 소프트웨어의 안전성(생명주기 프로세스)을 다루는 표준이다. 항공기 소프트웨어 안전성 표준은 DO-178C이므로 이 선지가 틀린 설명이다.
④ ISO 26262 - 자동차 기능 안전성 표준: O, 자동차에 탑재되는 전기·전자 시스템의 기능 안전성을 다루는 표준으로 맞는 설명이다.

오답 포인트

표준번호(IEEE 830, IEEE 1219, IEC 62304, ISO 26262)는 익숙하지만 각 표준이 다루는 산업 분야(요구명세, 유지보수, 의료기기, 자동차)를 뒤섞어 기억하면 틀리기 쉽다. 특히 IEC 62304를 항공기나 일반 임베디드 소프트웨어 표준으로 착각하지 않도록 의료기기라는 적용 대상을 명확히 구분해서 기억해야 한다.

NoteBase

다시 나오면 이렇게 판단

표준번호가 나오면 먼저 적용 산업 분야(요구공학·유지보수 같은 소프트웨어공학 프로세스 표준인지, 의료기기·자동차·항공기 같은 특정 산업의 안전성 표준인지)부터 구분한다. IEC로 시작하면 의료기기 관련 여부를, ISO로 시작하면 자동차 관련 여부를 먼저 떠올려 대조하면 오답 선지를 빠르게 골라낼 수 있다.

암기 포인트

IEEE 830=요구명세서, IEEE 1219=유지보수, IEC 62304=의료기기 소프트웨어 안전성, ISO 26262=자동차 기능 안전성, DO-178C=항공기 소프트웨어 안전성으로 대상 산업을 짝지어 암기해 둔다.

NoteBase