**🧭 1. Shift-Left / Shift-Right Terminology**

* **Shift-Left Approach** →  
  📌 *“Testing early in the SDLC”* (e.g., unit testing, static analysis, ATDD).  
  Testing activities move closer to the *development phase*.
* **Static Analysis** →  
  📌 Analyzing code or documents **without executing** the program. Usually automated.
* **Dynamic Testing** →  
  📌 Testing by **running** the program (executing the code).
* **ATDD (Acceptance Test-Driven Development)** →  
  📌 *Write acceptance criteria first* → convert to tests → build software to pass those tests.  
  Collaboration between business, testers, and developers.
* **TDD (Test-Driven Development)** →  
  📌 *Write unit test first* → fail → write code → pass → refactor.
* **BDD (Behavior-Driven Development)** →  
  📌 *Write scenarios in Given–When–Then format* that describe **behavior**, not just acceptance.

**🧪 2. Types of Testing**

* **Unit / Component Testing** →  
  📌 Testing individual units of code. Usually done by developers.
* **Component Integration Testing** →  
  📌 Testing interactions between components (e.g., modules, functions).
* **System Testing** →  
  📌 Testing the complete system as a whole.
* **System Integration Testing** →  
  📌 Testing the system’s interaction with **other systems**.
* **Acceptance Testing** →  
  📌 Testing to check if the system meets **business/user needs**. Usually final check before release.
* **Alpha vs Beta Testing** →
  + *Alpha*: done by internal users in development environment.
  + *Beta*: done by external users in real or simulated environment.

**🧰 3. Test Types**

* **Functional Testing** →  
  📌 “What the system does.” Check features/functions against requirements.
* **Non-Functional Testing** →  
  📌 “How well the system performs.” Check performance, security, usability, etc.
* **Regression Testing** →  
  📌 Check that recent changes didn’t break existing functionality.
* **Confirmation Testing (Re-testing)** →  
  📌 Check if the specific defect fix works as expected.
* **White-Box Testing** *(aka structure-based, glass-box)* →  
  📌 Tests based on internal structure/code logic.
* **Black-Box Testing** *(aka specification-based)* →  
  📌 Tests based on requirements/specifications only (no code knowledge).

**🔗 4. Development Models**

* **Sequential Development Model** →  
  📌 Classic waterfall or V-model. One phase follows another.
* **Incremental Development Model** →  
  📌 Build system in *pieces (increments)*. Each increment adds functionality.
* **Iterative Development Model** →  
  📌 Repeating cycles of build–improve–build again.
* **Agile Model** →  
  📌 Short iterations (sprints), incremental delivery, early testing.
* **V-Model** →  
  📌 Sequential, but testing activities are mapped to development stages (e.g., system design ↔ system testing).
* **Big-Bang Integration** →  
  📌 Integrate everything at once and test → fast but hard to debug.

**📊 5. DevOps / CI/CD Terminology**

* **DevOps** →  
  📌 A culture/methodology that integrates development and operations for faster delivery.
* **CI/CD (Continuous Integration / Continuous Delivery)** →  
  📌 Automated build, test, and deployment pipeline.
* **Regression Risk** →  
  📌 The risk that changes break existing working features.
* **Automation Challenges** →  
  📌 High setup cost, maintenance, skills required, may miss UX issues.

**🧾 6. Test Artefacts & Work Products**

* **Test Basis** →  
  📌 Any document or source used to design tests. Examples:
  + Requirements, user stories, state diagrams → system/acceptance testing
  + Code, component specs → unit/component testing
* **Test Harness** →  
  📌 Support code/tools needed to test components in isolation.
* **Impact Analysis** →  
  📌 Assess what parts of the system might be affected by a change. Helps decide **how much regression testing** is needed.

**⚡ 7. Non-Functional Quality Characteristics (ISO 25010 Terms)**

* **Compatibility** → how well the system works with others.
* **Portability** → how easily the system moves to a new environment.
* **Functional completeness** → whether all required functions are present.
* **Functional appropriateness** → whether functions help achieve the goals effectively.
* **Performance efficiency** → how well resources/time are used.
* **Maintainability / Modifiability** → how easy it is to change the system.
* **Usability** → how easy it is for users to use.

**🧭 8. Commonly Confusing Keywords in Exam Phrasing**

| **Keyword / Phrase** | **Meaning in Exam Context** |
| --- | --- |
| **“Best describes” / “Most suitable”** | Pick the **most correct** or **complete** answer, not just a true one. |
| **“Typically”** | Think about **real-world practice**, not exceptions. |
| **“Prior to dynamic testing”** | Means **before executing code** (static analysis, reviews). |
| **“Regression risk”** | Risk that something else breaks after a change. |
| **“Test-first”** | Tests written before code (TDD, ATDD). |
| **“Retrospective”** | Meeting after a sprint/iteration to identify what went well and what to improve. |
| **“Trigger for maintenance testing”** | An event that causes retesting or regression testing (e.g., change, fix, enhancement). |
| **“Test basis for…”** | Documents that you use to design test cases. |
| **“Early testing”** | Reviews, static analysis, unit tests. |
| **“Hot fix”** | Urgent fix in production. Usually involves confirmation + regression testing. |
| **“Re-testing vs Regression testing”** | Re-testing = fixed defect; Regression = ensuring nothing else broke. |
| **“Quadrant Q4 tests”** (Agile Testing Quadrants) | Non-functional quality tests (e.g., performance, security). |

**🧠 9. Common Confusion Traps**

* **“All testing should be automated”** → ❌ Trick. Not true.
* **“White-box can only be done at component level”** → ❌ Not strictly. Mostly component, but not *only*.
* **“Retrospective is for testers only”** → ❌ Retrospective involves the *whole team*.
* **“Impact analysis = regression testing”** → ❌ Impact analysis is **before** regression testing to plan its scope.
* **“Functional vs Non-functional”** →
  + Functional = features,
  + Non-functional = performance, security, usability, etc.

**📝 Tip for CTFL Exam:**

✅ When you see a **long question**, focus on these:

* **Verb** (e.g., identify, describe, match)
* **Scope** (e.g., shift-left, DevOps, impact analysis)
* **Test level or test type mentioned**
* **Keyword traps** (e.g., “only,” “all,” “always,” “typically,” “best describes”)

### **Answer 1:**

### **i. Unit testing** ✅ → **Developer responsibility**

* Unit testing checks **individual units or functions** of code.
* It is almost always written and executed by **developers**.
* Purpose: detect defects **very early** in the development lifecycle.

### **ii. Component Integration testing** ✅ → **Developer responsibility**

* This involves testing the interaction between **integrated components** developed by different team members.
* Usually done by **developers** before the whole system is handed off to testers.
* Ensures components work together correctly.

### **iii. System integration testing** ❌ → **Tester responsibility**

* System integration testing validates that **the entire system** (multiple subsystems) works together as expected.
* It’s typically conducted by **testers**, not developers.
* Focuses on end-to-end scenarios and real system behavior.

### **iv. ATDD (Acceptance Test–Driven Development)** ❌ → **Shared, but not mainly developer**

* ATDD involves **collaboration between business analysts, testers, and developers** to define acceptance criteria before implementation.
* Developers contribute, but this is not solely their responsibility.
* It’s a **shared activity**, not “developer rather than tester”.

### **v. Test system performance** ❌ → **Tester responsibility**

* Performance testing (load, stress, scalability) is **typically done by testers** or performance specialists.
* Developers may optimize performance, but testing it is not usually their main task.

#### **Answer 2:**

#### **a) “...opportunity for team members to voice concerns and depart on a positive note”** ❌

* While retrospectives **do allow sharing concerns and positive feedback**,
* This alone does **not guarantee process improvement**.
* It focuses more on communication and morale rather than structured improvement.

#### **b) “Daily retrospectives...”** ❌

* **Retrospectives are not done daily** — that’s confusing it with **daily stand-ups**.
* Retrospectives typically occur at the **end of an iteration or sprint**, not every day.
* So this statement is factually incorrect.

#### **c) “Objective metrics and factual data presented during a retrospective prompt the team to consider areas for improvement”** ✅

* This **directly reflects the purpose of retrospectives**:
  + Analyze what went well and what didn’t.
  + Use **data (e.g., defect trends, test coverage, velocity)** to identify patterns.
  + **Agree on specific actions** to improve future processes.
* This data-driven approach leads to **real, measurable process improvements**.

#### **d) “...comprehensive review of all test cases...”** ❌

* Reviewing test cases may be **a part of a retrospective**, but **that’s not its primary purpose**.
* Test case reviews are typically done in **test review sessions**, not retrospectives specifically.

#### **Answer 7:**

#### **a) “...achieved only with test automation and tools”** ❌

* This is **incorrect** because **shift-left** is not limited to automation.
* It also involves **manual activities** such as static reviews, walkthroughs, and inspections early in the lifecycle.
* Automation helps but is not the only factor.

#### **b) “...integrates manual testing practices like static analysis alongside test automation and tools”** ❌

* This is **close**, but **static analysis** is not a manual practice — it’s an automated process that analyzes code without execution.
* Also, the statement doesn’t fully capture the **emphasis on early testing** (like reviews, TDD, unit tests, etc.).

#### **c) “...emphasizes early testing automation, primarily involving testers in TDD, unit testing, and component integration testing”** ❌

* TDD and unit testing are **developer** responsibilities, not mainly tester activities.
* Shift-left does not primarily involve **testers** in TDD.

#### **d) “...relies on both manual testing methods like reviews and automated testing practices such as TDD”** ✅

* Correct because:
  + **Manual methods** like reviews, walkthroughs, and inspections catch defects early.
  + **Automated practices** like TDD, static analysis, and CI testing ensure quality early in development.
  + This aligns with the **core principle of shift-left**: **move quality activities earlier** in the lifecycle

### **Answer 13:**

### **i. TDD (Test-Driven Development)** ❌

* **Responsibility:** Developers
* TDD is primarily done by developers before or during coding.
* It focuses on writing unit tests first, then developing the code to make those tests pass.  
  ➡️ **Not a tester’s main responsibility.**

### **ii. System testing** ✅

* **Responsibility:** Testers
* System testing involves validating the entire integrated system against requirements.
* Testers design and execute system tests, often after integration but earlier in shift-left strategies.  
  ➡️ **This is a key tester responsibility.**

### **iii. Test system performance** ✅

* **Responsibility:** Testers
* Performance testing (e.g., load, stress, endurance) is usually carried out by specialized testers.
* It ensures the system performs as expected under various conditions.  
  ➡️ **Tester’s responsibility.**

### **iv. ATDD (Acceptance Test–Driven Development)** ❌

* **Responsibility:** Shared, but primarily developers + business + testers collaborate.
* ATDD involves writing acceptance criteria before implementation.
* While testers can contribute, it’s not solely their responsibility.  
  ➡️ **Not mainly a tester task.**

### **v. Test component security** ❌

* **Responsibility:** Can be both, but often developers handle security checks at the component level during development.
* Testers usually focus on system-level or end-to-end security.  
  ➡️ **Not primarily testers for component security.**

#### **Answer 21:**

#### **a) “It assesses the system's maintainability when changes occur...”**

* **Maintainability** is a **quality attribute** (ease of making changes).
* Impact analysis does not measure maintainability; it identifies what needs to be retested.  
  🚫 Incorrect focus.

#### **b) “Its objective is to evaluate if the test team can complete regression testing before the release”**

* This relates to **test scheduling and planning**, not to **impact analysis** itself.
* Impact analysis **feeds into** planning, but its **primary purpose** isn’t time estimation.  
  🚫 Mixing test planning with analysis.

#### **d) “Its task is to assess the advantageous and disadvantageous effects of a test environment change”**

* Impact analysis is about **system or code modifications**, not **environmental changes**.
* Environment change impact is handled separately (e.g., environment validation).  
  🚫 Wrong scope.

#### **Answer 22:**

#### **i. Component testing** ❌

* This focuses on testing **individual units or modules** in isolation.
* Regression testing can occur here, but the statement says:

“… no adverse consequences **within the system** …”

* That implies a **broader scope** than just components.  
  🚫 Not the best match for this scenario.

#### **ii. Component integration testing** ❌

* Tests **interaction between components**.
* If the change is deeper in the code, regression could happen here.
* But the statement again focuses on the **system level** impact, not component level.  
  🚫 Less likely in this scenario.

#### **iii. System testing** ✅

* Tests the entire integrated system against requirements.
* Regression testing at this level ensures that the change **didn’t break existing system functionality**.  
  ✅ Matches the scenario perfectly.

#### **iv. System integration testing** ✅

* Tests interactions **between different systems** (e.g., logistics app ↔ payment gateway, warehouse system, or map service).
* A change in one system can cause **adverse effects** in interconnected systems.  
  ✅ Also relevant.

#### **v. Acceptance testing** ✅

* Conducted to confirm that the system **still meets business needs** after a change.
* This often includes **regression checks** to ensure end-to-end business flows are not broken.  
  ✅ Relevant too.

### **Answer 23:**

### **1. Key Test Categories**

1. **Functional Testing**
   * Focuses on **what the system does** (functions, features, business requirements).
   * Examples: unit tests, integration tests, system tests for features.
2. **Non-functional Testing**
   * Focuses on **how the system performs** rather than what it does.
   * Examples: performance, security, usability, reliability.
3. **Black-box Testing**
   * **Behavior-based testing** where the tester doesn’t know the internal code.
   * Tests based on specifications, requirements, or use cases.
4. **White-box Testing**
   * **Code-based testing**, knowing the internal structure and logic.
   * Examples: statement coverage, branch coverage, path testing.
5. **Gray-box Testing**
   * **Hybrid approach**: tester knows some internal structure but tests primarily through interfaces.
   * Useful for integration and system testing when partial knowledge is available.

### **2. Why other options are incomplete**

#### **a) Functional, Non-functional, Black-box, Specification-based** ❌

* **Specification-based testing** is essentially a **subset of black-box testing**, so it’s redundant.
* Does not cover **white-box or gray-box techniques**.

#### **b) Functional, Non-functional, White-box, Glass-box** ❌

* **Glass-box** is another name for white-box, so that’s redundant.
* Missing **black-box testing**, which is widely used for functional verification.

#### **d) Functional, Non-functional, Specification-based, Structural testing** ❌

* **Structural testing** = white-box testing.
* Again, misses **black-box** and **gray-box testing**.

#### **Answer 25:**

#### **a) Requirement specs (functional and non-functional), risk analysis reports, epics and user stories, models of system behavior, state diagrams** ❌

* These are more suitable for **system testing** or **functional testing**, because they describe system behavior, risk areas, and technical flows.
* While some items (user stories, functional specs) may inform acceptance testing, **they do not capture the full business context or contractual obligations**.

#### **b) Sequence diagrams, interface and communication protocol specs, external interface definitions** ❌

* These are **design-level artifacts**.
* Useful for **integration testing** or **system integration testing**, **not acceptance testing**, which focuses on business validation.

#### **c) Detailed design, code, data models, component specifications** ❌

* These are **low-level technical artifacts**.
* Useful for **unit testing, component testing, or white-box testing**, not for verifying business requirements.

#### **d) Business processes, user or business requirements, regulations, legal contracts, and standards** ✅

* These represent the **business context and obligations**.
* Acceptance testing verifies:
  + Does the system support the **business processes** correctly?
  + Does it comply with **regulations and standards**?
  + Does it meet **contractual obligations** with the client?
* This aligns perfectly with the **purpose of acceptance testing**: **validate the system from the customer’s/business perspective**.

### **Answer 28:**

### **Key points:**

1. **Testing inside the company:**
   * The users are **inside the development environment**, not in the actual production environment.
2. **Diverse group of potential users:**
   * Real or representative users are invited to **simulate real-world usage**.
3. **Developers observe the testing:**
   * Developers actively monitor user interactions to gather feedback and detect issues.
4. **Independent test team may also test:**
   * Quality assurance or dedicated testers can be involved for **structured observations**.

#### **a) Alpha Testing** ✅

* Conducted **inside the development organization**.
* Involves **real users or potential customers** interacting with the system under observation.
* Developers and QA teams observe and gather feedback.
* Matches the description **perfectly**.

#### **b) Beta Testing** ❌

* Conducted **outside the company** in the **real user environment** (customer’s site).
* Users test the software without direct developer supervision.
* Not applicable here because the description clearly mentions **testing within the company**.

#### **c) Development Acceptance Testing** ❌

* This is **internal testing to verify that the software meets specifications** before release.
* It **does not necessarily involve external or representative users**.
* The description emphasizes real users interacting, so this is less accurate.

#### **d) Production Acceptance Testing** ❌

* Conducted **after deployment in the production environment**.
* This is more formal and environment-specific, not what is described here.

### ✅ **Summary:**

* **Alpha Testing** = internal testing **with potential users**, developers observe, feedback is gathered.
* **Beta Testing** = external user testing **in real environment**.

#### **Answer 29:**

#### **a) Functional testing can’t be done with the help of the white-box test design techniques** ❌ **(False)**

* **White-box testing** involves looking **inside the code** to design test cases (e.g., statement coverage, branch coverage).
* While functional testing is **primarily black-box (specification-based)**, it **can still use white-box knowledge** to improve test coverage, for example:
  + Ensuring all paths that implement a function are tested.
  + Verifying that functional behavior corresponds to the code.
* Saying it **“can’t be done”** is incorrect.  
  ✅ This statement is **false**.

#### **b) Functional testing focuses on the functionality of a system…** ✅

* True. Functional testing verifies **what the system does**, not how it does it internally.

#### **c) Functional testing can be applied to the acceptance testing level** ✅

* True. Functional testing occurs at **unit, integration, system, and acceptance levels** to verify that functions meet requirements.

#### **d) The objective of functional testing is checking whether a function or feature is complete, correct and appropriate** ✅

* True. Functional testing ensures **correctness, completeness, and appropriateness** of software behavior.

#### **Answer 32:**

#### **a) Regression testing** ✅

* **Definition:** Regression testing is performed to confirm that **recent changes (patches, fixes, enhancements) have not adversely affected existing functionality**.
* In this case:
  + Applying a **security patch** is a change.
  + Testing posting, messaging, and newsfeed ensures that the **system continues to work correctly**.
* Matches perfectly with the scenario.

#### **b) Confirmation testing (Re-testing)** ❌

* **Definition:** Testing to verify that a specific defect has been **fixed correctly**.
* It only focuses on the **patch itself**, not on the **impact on existing functionality**.
* The scenario also requires checking **core functionalities**, so this alone is not sufficient.

#### **c) Maintenance testing** ❌

* **Definition:** Maintenance testing occurs when software is modified post-release (patches, enhancements, bug fixes).
* While a security patch **is part of maintenance**, maintenance testing is a **broader term**.
* The **specific activity described**—checking that existing features still work—is **regression testing**.

#### **d) Security testing** ❌

* **Definition:** Security testing focuses on verifying **security aspects** (confidentiality, integrity, access control, vulnerabilities).
* While part of the task is security-related (patch verification), the **major focus in this scenario is also verifying unaffected core functionality**, which is **regression**.

#### **Answer 33:**

#### **a) Regression approach** ❌

* Regression testing is **retesting after changes** to ensure nothing breaks.
* It is **not an integration strategy**.

#### **b) Hybrid approach** ❌

* Combines **top-down and bottom-up integration**, integrating incrementally.
* Not all components are integrated at once.

#### **c) Big-bang approach** ✅

* All modules/components are integrated **at once** at the end.
* Matches **exactly** the description in the question.
* Advantage: no stubs or drivers needed.
* Disadvantage: debugging is difficult.

#### **d) Collapsing approach** ❌

* Not a standard integration strategy. Likely refers to **incremental strategies**, but doesn’t match the description.

#### **Answer 37:**

#### **a) Testing based on an analysis of the specification of the test object** ❌

* “Test object” is vague and less precise.
* Black-box testing is defined **per component or system**, not generically “test object”.

#### **b) A test technique based on an analysis of the specification of a component or system** ✅

* Accurate and formal definition.
* Covers both **component level** and **system level** testing.
* Matches **ISTQB and CTFL definitions**.

#### **c) An approach to testing in which test conditions are based on component requirements** ❌

* Too narrow.
* Black-box testing can also be applied at **system level**, not only at the component level.

#### **d) Testing based on test conditions of the acceptance criteria of user stories** ❌

* This is **a specific application** of black-box testing (e.g., acceptance or specification-based testing).
* Not the **general definition** of black-box testing.

#### **Answer41:**

#### **a) It takes too long to execute compared to other testing methods** ❌

* Confirmation testing is usually **quick**, as it involves **a small set of tests**.

#### **b) It might miss regressions introduced by the fix in other areas** ✅

* Correct. The narrow focus on the failed tests means **other impacted areas could break unnoticed**.
* This is why confirmation testing is often **supplemented with regression testing**.

#### **c) It requires testers to have a deep understanding of the code changes** ❌

* Not necessarily. Testers can re-run the failed tests **without knowing the internal code**, especially in black-box testing.

#### **d) It doesn't provide sufficient information about the fix's effectiveness** ❌

* Confirmation testing **does verify that the specific defect is fixed**.
* The problem is **coverage of potential side effects**, not effectiveness of the fix itself.

#### **Answer: 43**

#### **a) Complexity of the development environment** ❌

* While environment complexity may influence testing effort, it does **not directly define the scope** of what needs to be tested.

#### **b) Size of the existing system** ✅

* Correct.
* Larger systems often have **more modules, interdependencies, and components**, so maintenance testing must cover more areas.
* The scope is **proportional to system size and affected components**.

#### **c) Time available for testing** ❌

* Time may limit how much testing can be performed, but it **doesn’t define the inherent scope**; it’s a constraint, not a determinant.

#### **d) Cost of fixing potential defects** ❌

* Cost influences **risk prioritization**, but it does not determine which parts of the system are affected or should be tested.

#### **Answer 48:**

#### **a) Regression testing** ❌

* Regression testing verifies that **changes or fixes do not break existing functionality**.
* It **does not measure ease of modification**.

#### **b) Non-functional testing** ❌

* Non-functional testing includes **performance, security, usability, reliability, maintainability, etc.**
* While modifiability is technically a **non-functional attribute**, “non-functional testing” is a **broad category**, not the specific testing being performed here.

#### **c) Adaptability testing** ❌

* Adaptability testing evaluates **how easily a system can be adapted to new environments** (e.g., OS, hardware, platforms).
* The focus here is **environment adaptation**, not maintainers’ ability to modify the system.

#### **d) Modifiability testing** ✅

* Correct.
* **Modifiability** (also called maintainability) testing checks **how easy it is for developers or maintainers to modify or extend the system**.
* Examples: changing functionality, updating code, adding features.