5 Design by Contract - exam questions

You should know the answers to these questions

What is the distinction between Testing and Design by Contract? Why are they complementary techniques?

- **Testing** diagnoses and cures defects after they occur. It verifies the software by checking if it meets the expected behavior through test cases.
- **Design by Contract (DbC)** prevents specific types of defects by explicitly defining preconditions, postconditions, and invariants as contracts between components.
- They are complementary because:
  - **Testing** detects a wide range of coding mistakes.
  - **DbC** prevents mistakes due to incorrect assumptions between provider and client.
  - Together, they enhance **traceability** by linking code and requirements and ensuring higher reliability.

What's the weakest possible condition in logic terms? And the strongest?

- **Weakest**: `{true}`—it imposes no constraints and is always true.
- **Strongest**: `{false}`—it imposes a condition that is impossible to satisfy.

If you have to implement an operation on a class, would you prefer weak or strong conditions for pre- and postcondition? And what about the class invariant?

- **Precondition**: Strong preconditions make implementation easier as they constrain the input. However, weak preconditions are preferred by clients because they reduce constraints on the input.
- **Postcondition**: Strong postconditions are ideal as they provide detailed guarantees of the output state.
- **Class Invariant**: Must remain strong to ensure the consistency of the class across all operations.

If a subclass overrides an operation, what is it allowed to do with the pre- and postcondition? And what about the class invariant?

- **Precondition**: The subclass can weaken or maintain the precondition but cannot strengthen it. This ensures compatibility with clients using the superclass.
- **Postcondition**: The subclass must maintain or strengthen the postcondition.
- **Class Invariant**: The subclass must maintain the class invariant.

Compare Testing and Design by contract using the criteria "Correctness" and "Traceability".

- **Correctness**:
  - **Testing** identifies deviations from expected behavior during execution.
  - **DbC** ensures correctness by enforcing contracts during development.
- **Traceability**:
  - **Testing** maps requirements to test cases and observed behavior.
  - **DbC** embeds requirements in the source code via contracts, enabling direct traceability.

What's the Liskov substitution principle? Why is it important in OO development?

- **Definition**: Subtypes must be substitutable for their supertypes without altering the correctness of the program.

- **Importance**: It ensures polymorphism and behavioral consistency, making systems more robust and reusable.

What is behavioral subtyping?

- Behavioral subtyping requires that a subclass satisfies all the behavioral contracts of its superclass. This includes respecting preconditions, postconditions, and invariants, ensuring that the subclass does not violate the expectations set by the superclass.

When is a pre-condition reasonable?

- A precondition is reasonable if:
  - It is justifiable based on the requirements specification.
  - It can be satisfied and checked by clients.
  - It uses only public operations of the class.

You should be able to complete the following tasks

What would be the pre- and post-conditions for the methods top and isEmpty in the Stack specification? How would I extend the contract if I added a method size to the Stack interface?

- `top`:
  - Precondition: The stack is not empty (`!isEmpty()`).
  - Postcondition: Returns the last element pushed onto the stack without removing it.
- `isEmpty`:
  - Precondition: None (always callable).
  - Postcondition: Returns `true` if the stack has no elements, otherwise `false`.
- **Extension for `size`:**
  - Precondition: None (always callable).
  - Postcondition: Returns the current number of elements in the stack.

Apply design by contract on a class Rectangle, with operations move() and resize().

- **Class Invariant**: `width > 0 && height > 0`.
- `move(dx, dy)`:
  - Precondition: None.
  - Postcondition: The rectangle's position is updated by `(dx, dy)`.
- `resize(dw, dh)`:
  - Precondition: `width + dw > 0 && height + dh > 0`.
  - Postcondition: The dimensions are updated to `width + dw` and `height + dh`.

Write consumer-driven contracts for a given REST-API .

- Define a contract for each endpoint describing:
  - **Consumer requests** (method, path, headers, body).
  - **Provider responses** (status codes, headers, body structure).
- Example for `GET /users/{id}`:
  - Consumer expectation: Returns user details for a valid ID.
  - Provider obligation: Responds with `200 OK` and JSON data or `404 Not Found`.

Can you answer the following questions?

Why are redundant checks not a good way to support Design by Contract?

- Redundant checks increase code complexity by introducing unnecessary verification logic that must also be validated and maintained.
- They incur a performance penalty, as verifying redundant conditions takes additional execution time.
- Responsibility is misaligned: Design by Contract assigns precondition checks to the client, making redundant checks within the supplier's code counterproductive.

- They blur the separation of concerns, complicating debugging and testing.

You're a project manager for a weather forecasting system, where performance is a real issue. Set-up some guidelines concerning assertion monitoring and argue your choice.

1. **Enable Assertions During Development**:
   - Assertions can catch violations of preconditions, postconditions, and invariants early.
   - Ensure that the system behaves correctly during development and testing phases.
2. **Disable Assertions in Production**:
   - To avoid performance overhead in critical real-time operations, turn off assertions in production builds.
3. **Profile the System**:
   - Use performance profiling to identify bottlenecks caused by assertion checks.
4. **Monitor Critical Preconditions**:
   - Maintain lightweight, essential precondition checks in performance-critical components, ensuring these checks are fast and minimal.
5. **Adopt Conditional Compilation**:
   - Use compile-time or runtime configurations to include or exclude assertions based on the environment.

Argument: These guidelines strike a balance between maintaining correctness during development and achieving high performance during production.

If you have to buy a class from an outsourcer in India, would you prefer a strong precondition over a weak one? And what about the postcondition?

- **Precondition**: A **strong precondition** is preferable, as it clearly defines the supplier's expectations and limits the scenarios they must handle. This clarity makes it easier to verify the correctness of the outsourced class.
- **Postcondition**: A **strong postcondition** is preferred because it guarantees a more predictable and detailed outcome from the class, reducing ambiguity and potential integration issues.

Do you feel that design by contract yields software systems that are defect free? If you do, argue why. If you don't, argue why it is still useful.

- **No**, Design by Contract does not guarantee defect-free systems because:
  - It addresses specific errors related to interface and contract violations but cannot catch all possible coding or logic errors.
  - It assumes the correctness of the implementation, which may still contain flaws.
- **However**, it is still highly useful because:
  - It reduces defects caused by miscommunication between components.
  - It embeds requirements into the source code, improving traceability and documentation.
  - It facilitates better testing by specifying clear conditions for input, output, and system states.

How can you ensure the quality of the pre- and postconditions?

How can you ensure the quality of the pre- and postconditions?

1. **Validate Against Requirements**:
   - Ensure preconditions and postconditions align with the system's requirements and use cases.
2. **Iterate Through Peer Reviews**:
   - Conduct code reviews focused on verifying the correctness of contracts.
3. **Use Static Analysis Tools**:
   - Employ tools that can analyze and validate contracts for logical consistency and completeness.
4. **Test Edge Cases**:
   - Design test cases that verify contracts hold in boundary and exceptional conditions.
5. **Document Contracts Clearly**:
   - Clearly document the rationale for each contract to ensure they are well-understood and consistently applied.

Why is (consumer-driven) contract testing so relevant in the context of micro-services?

- **Ensures Compatibility**: Validates that services adhere to agreed contracts, preventing integration failures.
- **Isolates Services**: Allows testing of microservices independently by mocking dependencies based on contracts.
- **Prevents Breaking Changes**: Changes in one service are tested against consumer contracts, ensuring backward compatibility.
- **Improves Traceability**: Makes explicit which parts of the contract are actively used by consumers, focusing testing efforts.
- **Supports Agile Development**: Enables rapid iterations by verifying only the necessary components without requiring full-system integration tests.

Assume you have an existing software system and you are a software quality engineer assigned to apply design by contract. How would you start? What would you do?

1. **Analyze Existing System**:
   - Identify key components and their interactions.
   - Review documentation to understand implicit contracts and assumptions.
2. **Define Contracts**:
   - Explicitly define preconditions, postconditions, and invariants for critical operations and classes.
3. **Refactor Code**:
   - Introduce assertions to enforce these contracts in the source code.
4. **Test Contracts**:
   - Design tests to verify the validity of preconditions, postconditions, and invariants.
5. **Automate Validation**:
   - Use tools to automate contract validation and regression testing.
6. **Educate Team Members**:
   - Train developers on the principles and practices of Design by Contract to ensure consistent adoption.
7. **Iterate Incrementally**:
   - Apply Design by Contract incrementally, starting with high-risk or high-impact components to minimize disruption.