Übungsaufgabe

Spezifikations- vs. Implementierungsbasierte Entwicklung: Design by Contract, Assertions und Validierung

Universität:	Technische Universität Berlin
Kurs/Modul:	Algorithmen und Datenstrukturen
Erstellungsdatum:	September 6, 2025

Aufgabe 1: Begriffe und Grundkonzepte

In dieser Aufgabe werden zentrale Begriffe der spekifications- vs. implementierungsbasierten Entwicklung eingeführt. Ziel ist es, eine klare, formale Sprache für DbC, Vor-/Nachbedingungen, Invarianten und Assertions zu entwickeln und deren Unterschiede herauszuarbeiten.

a) Begriffe und Grundkonzepte

- Definieren Sie Design by Contract (DbC), Precondition (Vorbedingung), Postcondition (Nachbedingung), Invariant (Invariante) und Assertion im Kontext der Software-Entwicklung.
- Erläutern Sie den Unterschied zwischen spefizationsbasierter Entwicklung und implementierungsbasierter Entwicklung. Diskutieren Sie je zwei Vor- bzw. Nachteile.

b) Abgrenzung und Beziehungen

- Beschreiben Sie, wie Pre-/Postbedingungen, Invarianten und Assertions zusammenwirken, um Korrektheit und Robustheit eines Softwaresystems sicherzustellen.
- Geben Sie ein kleines, sprachunabhängiges Beispiel, in dem eine Precondition verletzt wird, und erläutern Sie, welche Art von Fehlersignal daraus resultieren sollte.

c) Technische Notation

- Formulieren Sie die drei Konzepte Pre-/Postbedingungen und Invariante mit der üblichen mathematischen Semantik. Verwenden Sie klare, wiederholbare Aussagen (z. B. *Folie* oder *Relationen*).
- Diskutieren Sie, warum reine Spezifikationen oft nicht ausreichen, wenn Implementierungsdetails nicht transparent bleiben.

d) Kurze Reflexion Nennen Sie zwei typische Missverständnisse oder Fehlinterpretationen von DbC und schlagen Sie jeweils eine Gegenmaßnahme vor.

Aufgabe 2: Spezifikationen, Assertions und Validierung in einer kleinen API

In dieser Aufgabe arbeiten Sie an einem einfachen Beispiel aus der Praxis, bei dem Spezifikationen, Assertions und Validierung eine zentrale Rolle spielen. Es handelt sich um textliche Formulierungen und keine konkrete Programmiersprache.

a) Beispiel-Schnittstelle: Stack-API Stellen Sie sich einen einfachen Stack mit den Operationen push(x) und pop() vor. Der Stack enthält ausschließlich Werte vom Typ T (beliebig).

Formulieren Sie für die Operationen jeweils Precondition, Postcondition:

- push(x): Precondition, Postcondition
- pop(): Precondition, Postcondition

Beziehen Sie sich bei den Bedingungen auf die Konzepte aus Aufgabe 1 (Pre, Post, Invariant) und verwenden Sie klare Formulierungen.

b) Invariante des Stack-Objekts Formulieren Sie eine Invariante, die während der Lebenszeit eines Stack-Objekts gültig bleiben soll. Beschreiben Sie, wie diese Invariante durch Pre-/Postbedingungen in den Operationen unterstützt wird.

c) Assertions zur Validierung Skizzieren Sie, wie Assertions in einer Implementierung verwendet würden, um die obigen Bedingungen zu validieren. Geben Sie drei konkrete Beispiele (Pre-, Post- und Invariante) an, inklusive der Art der Prüfung (z. B. Größen- oder Gleichheitsprüfungen).

d) Validierungsszenarien und Tests Nennen Sie mindestens drei sinnvolle Testszenarien bzw. Assertions-Geschichten, mit denen man die Korrektheit der Stack-API gegen die Spezifikation validieren könnte (z. B. Leere- vs. volle Zustände, korrekte Reihenfolge der Werte). Diskutieren Sie kurz, wie Validierung in der Praxis als Teil eines Testpools integriert werden könnte.

Aufgabe 3: Fallstudie – Design by Contract in einer kleinen Algorithmus-Implementierung

Betrachten Sie eine Funktion, die eine Liste von Ganzzahlen sortiert zurückgibt. Angenommen, die Funktion heißt sortiere(list) und akzeptiert eine nicht-leere Liste von ganzzahligen Werten.

a) Spezifizieren Sie Pre- und Postbedingungen für sortiere(list). Berücksichtigen Sie dabei Eigenschaften wie Nicht-Veränderung der ursprünglichen Liste (falls gefordert), Stabilisierungs- und Konsistenz-Eigenschaften sowie die Reihenfolge der Ausgabewerte.

b) Assertions-Strategie Skizzieren Sie eine Assertions-Strategie, die Pre-, Post- und Invarianten abdeckt, insbesondere wie man sicherstellt, dass die Ausgabe sortiert ist und dass keine Elemente verloren gehen.

c) Validierung im Entwicklungsprozess Diskutieren Sie, wie Validierungsschritte (Inline-Assertions, Property-Based-Tests oder Beispiel-Inputs) in den Entwicklungsprozess integriert werden könnten, um Design by Contract konsequent anzuwenden. Welche Metriken oder Qualitätskriterien unterstützen diesen Prozess?

Lösungen

Lösung zu Aufgabe 1: Begriffe und Grundkonzepte

1. a) Begriffe und Grundkonzepte

- Design by Contract (DbC): Ein vertraglich definierter Aufbau zwischen Komponenten (z. B. Client und Implementierung), der Pre-/Postbedingungen, Invarianten und Assertions verwendet, um Korrektheit und Robustheit zu garantieren. Der Vertrag spezifiziert, was sowohl die Komponente zu liefern hat als auch unter welchen Voraussetzungen sie dies liefern kann.
- Precondition (Vorbedingung): Eine Bedingung P über Eingaben und ggf. Zustand, die vor Ausführung einer Operation wahr sein muss. Sie definiert, welche Annahmen der Aufrufer erfüllen muss.
- Postcondition (Nachbedingung): Eine Bedingung Q über Eingaben, Ausgangswert und Zustand nach Ausführung der Operation. Sie garantiert das erzielte Ergebnis bzw. den neuen Zustand.
- Invariant (Invariante): Eine Zustandsbedingung I, die während der Lebenszeit eines Objekts bzw. einer Datenstruktur (insb. zwischen ausführenden Operationen) unverändert gelten muss.
- Assertion (Assertionsbedingung): Eine zu einem bestimmten Punkt im Programm verwendete Bedingung, die während der Ausführung geprüft wird, um die Korrektheit von Zwischenzuständen sicherzustellen. Assertions dienen häufig als Laufzeitprüfungen und Debug-Hilfen.

Erläuterung der Unterscheidung zwischen spefizationsbasierter Entwicklung und implementierungsbasierter Entwicklung:

- Spezifikationsbasierte Entwicklung:
  - Vorteile:
  - Klare, überprüfbare Verträge ermöglichen bessere Wartbarkeit und klare Schnittstellen.
  - Ermöglicht frühzeitige Verifikation bzw. formale Beweisführung und bessere Kommunikationen zwischen Auftraggeber und Entwickler.
  - Nachteile:
  - Erhöhter Aufwand für das Formulieren präziser Spezifikationen; potenziell schwierige Abstraktionen erfordern zusätzliche Schulung.
  - Kann zu Divergenzen zwischen spefizierter Formulierung und tatsächlicher Implementierung führen, wenn Implementierungsdetails nicht transparent bleiben.
- Implementierungsbasierte Entwicklung:
  - Vorteile:
  - Konkrete, pragmatische Umsetzung erleichtert schnelleres Prototyping und unmittelbare Nutzbarkeit.
  - Geringerer anfänglicher Planungsaufwand, insbesondere in kleinen Projekten.
  - Nachteile:

  - Fehlende oder uneinheitliche Spezifikation erschwert Korrektheitssicherung und spätere Wartbarkeit.
  - Mehr Aufwand für Debugging, da Vertragsaspekte oft implizit bleiben.

1. b) Abgrenzung und Beziehungen Pre-/Postbedingungen, Invarianten und Assertions arbeiten zusammen, um Korrektheit und Robustheit sicherzustellen:

- Invariante I muss während der Lebenszeit eines Objekts in jedem gut definierten Zustand gelten, insbesondere vor und nach jeder öffentlichen Operation.
- Precondition P begründet die Bedingungen, unter denen eine Operation sicher aufgerufen werden kann. Wenn P verletzt wird, sollte das Verhalten außerhalb des Vertrags liegen (z.. Exception, Fehlersignal).
- Postcondition Q garantiert das Verhalten nach der Ausführung, sofern P erfüllt war. In der Regel muss Q auch das Invariante-Verhalten sicherstellen.
- Assertionen A dienen als interne Prüfungen an bestimmten Stellen des Codes und prüfen explizit, ob eine notwendige Bedingung zum Zeitpunkt der Ausführung gilt (z.. Nachbedingungen nach einem Schritt, Invariante nach einer Teiloperation).

Kleines, sprachunabhängiges Beispiel:

- Precondition: Für eine Operation `divide(a,b)` gilt $b \neq 0$.
- Postcondition: Das Ergebnis r erfüllt $a = r \cdot b$ (und ggf. zusätzlicher Kontext).
- Invariante: Für alle Zwischenzustände gilt, dass der Speicherbereich, der die Operanden speichert, denselben Typ und Größenbereich hat.
- Fehlersignal bei Verletzung der Precondition: Die Implementierung löst z.. eine Ausnahme aus (z.. `IllegalArgumentException`) oder erzeugt eine Fehlermetrik.

1. c) Technische Notation

- Pre-/Postbedingungen (mathematische Semantik):
  - Precondition $P(x, \sigma)$: Zustandsbedingung vor der Ausführung unter Eingabe x und Zustand σ.
  - Postcondition $Q(x, \sigma)$: Zustandsbedingung nach der Ausführung unter Eingabe x und neuem Zustand σ'.
  - Invariante $I(\sigma)$: Zustandbedingung, die für alle erreichbaren Zustände σ gelten muss.

Beispiel für eine Stack-Operation push(x):

- Pre: $P_{\text{push}}(x) \equiv \text{True}$ (Für alle $x \in T$ zulässig).
- Post: $Q_{\text{push}}(x) \equiv (\text{size}' = \text{size} + 1) \land (\text{top}' = x) \land \forall i < \text{size}.\ \text{elem}'[i] = \text{elem}[i]$.

Beispiel für eine Stack-Operation pop():

- Pre: $P_{\text{pop}} \equiv \text{size} > 0$.

- Post: Es gilt $\exists y : \text{pop}() = y$ mit $\text{size}' = \text{size} - 1$ und $\text{top} = y$ (bzw. Rückgabewert y); der Rest des Stack-Inhalts bleibt unverändert.

Warum Spezifikationen allein oft nicht ausreichen: Spezifikationen beschreiben, was richtig sein soll, aber nicht immer, wie die Implementierung die Details der Plattform, Optimierungen oder Nebenwirkungen handhabt. Gleichzeitig müssen Implementierungen oft konkrete Entscheidungen offenlegen (z.. Speicherung, Indizierung), damit der Vertrag durchsetzbar bleibt. Eine eng verzahnte Verbindung von Spezifikation, Invariante und Assertions erleichtert, Korrektheit auf unterschiedlichen Abstraktionsebenen systematisch zu prüfen.

1. d) Kurze Reflexion

- Typische Fehlinterpretationen:
  - Fehlinterpretation A: DbC bezieht sich lediglich auf Laufzeitprüfungen und Debugging. Gegenmaßnahme: DbC ist ein ganzer Vertrag, der auch formale Semantik, statische/verifikationstechnische Unterstützung und klare Schnittstellenbeschreibung umfasst; nutze Assertions an strategischen Stellen und verifiziere Verträge gegebenenfalls formal.
  - Fehlinterpretation B: Precondition ist ausschließlich eine Verpflichtung des Aufrufer und der Anbieter kann sich davon freisprechen, solange die Implementierung robust ist. Gegenmaßnahme: Klare Trennung von Zuständigkeiten—Aufrufer muss Precondition erfüllen; Implementierung muss Postcondition liefern, und invarianten Zustand sicherstellen; dokumentiere Klarstellungen und nutze ggf. Fehlermeldungen/Exceptions bei Verletzungen.
- Gegenmaßnahmen:
  - Verwende klare, maschinenlesbare Vertragsbeschreibungen (z.. formale Spezifikationen oder strukturierte Contract-Dokumente) zusätzlich zu verständlichen Textbeschreibungen.
  - Ergänze Inline-Assertions und Tests (Property-Based-Tests, Beispiel-Inputs), um Contract-Gültigkeit kontinuierlich zu validieren.

Lösung zu Aufgabe 2: Spezifikationen, Assertions und Validierung in einer kleinen API

2. a) Beispiel-Schnittstelle: Stack-API Stellen Sie sich einen einfachen Stack mit den Operationen push(x) und pop() vor. Der Stack enthält ausschließlich Werte vom Typ T (beliebig).

Formulieren Sie für die Operationen jeweils Precondition, Postcondition:

- push(x): Precondition, Postcondition
- pop(): Precondition, Postcondition

Beziehen Sie sich bei den Bedingungen auf die Konzepte aus Aufgabe 1 (Pre, Post, Invariant) und verwenden Sie klare Formulierungen.

- push(x):
  - Precondition: $P_{\text{push}}(x) \equiv \text{True}$ (für alle $x \in T$ zulässig).
  - Postcondition: $\text{size}' = \text{size} + 1 \wedge \text{top}' = x \wedge \text{elem}' = \text{elem} \frown [x]$
- pop():
  - Precondition: $P_{\text{pop}} \equiv \text{size} > 0$.
  - Postcondition: Es gilt $\text{size}' = \text{size} - 1 \wedge \text{ret} = \text{top}$ und $\text{elem}' = \text{elem}[0..\text{size} - 2]$ (die übrigen Elemente bleiben unverändert).

2. b) Invariante des Stack-Objekts Formulieren Sie eine Invariante, die während der Lebenszeit eines Stack-Objekts gültig bleiben soll. Beschreiben Sie, wie diese Invariante durch Pre-/Postbedingungen in den Operationen unterstützt wird.

- Invariante $I(\sigma)$:
  - $\text{size} \geq 0$.
  - Die interne Sequenz $\text{elem}[0..\text{size} - 1]$ entspricht exakt dem Inhalt des Stack in dieser Lebenszeit, d. h. für alle $i \in [0, \text{size} - 1]$ gilt $\text{elem}[i] = \text{StackContent}[i]$.
- Unterstützung durch Pre-/Postbedingungen:
  - Push erfüllt mit Postbedingung, dass size erhöht wird und das neue oberste Element x ist; dadurch bleibt die Relation $\text{elem}' = \text{elem} \frown [x]$ konsistent mit der Invariante.
  - Pop erfüllt mit Postbedingung, dass das oberste Element entfernt wird und size verringert wird; die verbleibenden Indizes bleiben konsistent mit der Invariante.

2. c) Assertions zur Validierung Skizzieren Sie, wie Assertions in einer Implementierung verwendet würden, um die obigen Bedingungen zu validieren. Geben Sie drei konkrete Beispiele (Pre-, Post- und Invariante) an, inklusive der Art der Prüfung (z. B. Größen- oder Gleichheitsprüfungen).

- Pre-Assertion (für pop):
  - Bedingung: $\text{size} > 0$.
  - Prüfungstyp: Größenprüfung.
  - Meldung: „Stack leer – pop-Anfrage ungültig“.
- Post-Assertion (nach push(x)):
  - Bedingung: $\text{size}' = \text{size} + 1 \wedge \text{top}' = x$.
  - Prüfungstyp: Gleichheits- und Größenprüfung.
  - Meldung: „Push erfolgreich: Größe und oberstes Element stimmen nicht überein“.

- Invariante-Assertion (nach jeder öffentlichen Operation):
  - Bedingung: size $\geq 0 \wedge$ elem[0..size $- 1$] entspricht Stack-Inhalt.
  - Prüfungstyp: Gleichheits-/Indexprüfung.
  - Meldung: „Stack-Invariante verletzt“.

2. d) Validierungsszenarien und Tests Nennen Sie mindestens drei sinnvolle Testszenarien bzw. Assertions-Geschichten, mit denen man die Korrektheit der Stack-API gegen die Spezifikation validieren könnte (z. B. Leere- vs. volle Zustände, korrekte Reihenfolge der Werte). Diskutieren Sie kurz, wie Validierung in der Praxis als Teil eines Testpools integriert werden könnte.

- Szenario 1: Leerer Stack, Pop soll fehlschlagen bzw. Ausnahme verursachen. Erwarteter Fehlerzustand bzw. Verhalten.
- Szenario 2: Folge von Push-Operationen (z.. push(1), push(2), push(3)) und anschließendes Pop-Verhalten (erst 3, dann 2, dann 1) sowie Größenprüfungen nach jedem Schritt.
- Szenario 3: Randfall-Größe: Push bis zur Kapazitätsgrenze (falls implementiert) und weiterer Push, sowie anschließendes Pop-Verfahren; Test, dass Größe nie negativ wird und keine Elemente verloren gehen.

Diskussion:

- Validierung im Praxis-Kontext oft als Teil eines kontinuierlichen Testpools (Unit-Tests + Property-Based-Tests) implementiert.
- Eigene Assertions-Strategien (Pre/Post/Invariante) werden direkt in den Code integriert und durch automatisierte Tests ergänzt; Property-Based-Tests prüfen invarianten Eigenschaften über eine große Menge von zufällig generierten Eingaben.

Lösung zu Aufgabe 3: Fallstudie – Design by Contract in einer kleinen Algorithmus-Implementierung

3. a) Spezifizieren Sie Pre- und Postbedingungen für sortiere(list) Betrachten Sie eine Funktion sortiere(list), die eine nicht-leere Liste von ganzzahligen Werten zurückgibt.

- Precondition:
  - $list \neq \emptyset$ (nicht-leere Liste).
  - Erwartung: Elemente des Inputs gehören der Menge $\mathbb{Z}$ an.
- Postcondition (Variante A – Ausgabe inkl. unveränderter Eingabe):
  - Die Rückgabe $sortierte\_Liste$ ist monotone nicht fallend: $\forall i : sortierte\_Liste[i] \leq sortierte\_Liste[i + 1]$.
  - Multiset(Gleichheit): Für jede Zahl v gilt $\#(v \text{ in } list) = \#(v \text{ in } sortierte\_liste)$.

  - Länge erhält sich: $|sortierte\_liste| = |list|$.
  - Eingabe bleibt unverändert: $list$ bleibt in der ursprünglichen Form erhalten.

Hinweis: Alternativ ist auch möglich, dass die Funktion eine sortierte Kopie der Eingabe zurückgibt, während die Originale unverändert bleibt (wie hier beschrieben). Falls stattdessen in-place sortiert wird, müsste Postbedingung entsprechend angepasst werden (z.. Input verändert sich, Output ist erfüllt durch denselben Speicherort).

3. b) Assertions-Strategie Skizzieren Sie eine Assertions-Strategie, die Pre-, Post- und Invarianten abdeckt, insbesondere wie man sicherstellt, dass die Ausgabe sortiert ist und dass keine Elemente verloren gehen.

- Pre-Assertion: Vor Funktionsaufruf $list \neq \emptyset$ und Typ der Einträge ist Integer.
- Post-Assertionen:
  - (Sortierung) Die Ausgabe ist sortiert: $\forall i : result[i] \leq result[i+1]$.
  - (Invariante/Erhaltung der Werte) Multipset der Ausgabe entspricht dem Eingabemultipset: Multiset($result$) = Multiset($list$).
  - (Längenäquivalenz) $|result| = |list|$.
- Invarianten (falls die Implementierung intern den Eingabewert verändert):
  - Falls in-place sortiert wird, die Referenz auf list bleibt gültig, und nach Beendigung gilt: list enthält eine sortierte Kopie der ursprünglichen Werte.

Begründung: Assertions sichern die Übereinstimmung von Output-Eigenschaften (Sortiertheit, Vollständigkeit, keine Duplikat-Fehler) mit dem Contract.

3. c) Validierung im Entwicklungsprozess Diskutieren Sie, wie Validierungsschritte (Inline-Assertions, Property-Based-Tests oder Beispiel-Inputs) in den Entwicklungsprozess integriert werden könnten, um Design by Contract konsequent anzuwenden. Welche Metriken oder Qualitätskriterien unterstützen diesen Prozess?

- Inline-Assertions:
  - Vorteile: Direktes Feedback am Quellcode, frühzeitige Erkennung von Verletzungen der Contract-Bedingungen.
  - Umsetzung: Assertions an relevanten Stellen (Pre-, Post-, Invariant-Punkte) mit aussagekräftigen Fehlermeldungen.
- Property-Based-Tests (z.. QuickCheck-ähnlich):
  - Vorteile: Breites Testen der Contract-Eigenschaften über viele zufällig erzeugte Eingaben.
  - Fokus: Ausprägungen der Precondition, generische Listen mit unterschiedlicher Länge und Wertebereichen.
- Beispiel-Inputs/Beispiele-Tests:

  - Explizite Beispiel-Eingaben (Randfälle, normale Fälle) ergänzen, um Vertrauen in die Contracts zu schaffen.

Welche Metriken oder Qualitätskriterien unterstützen diesen Prozess?

- Abdeckungsmetriken (Testabdeckung) für Verträge (Welche Teile des Contracts werden wirklich geprüft?).
- Metriken zur Contract-Fehlerdichte (Anzahl gefundener Contract-Verletzungen pro Testlauf).
- Metriken zur Stabilität der Contracts über Refactorings (Regression-Rate der Assertions).
- Mutationsanalyse (Mutation Score): Tests sollten auch mutierte Contracts entdecken.