Übungsaufgabe

Praktische Übungen und Projektdurchführung: Beispielprojekte, Tests mit ScalaTest, Debugging.

Universität:	Technische Universität Berlin
Kurs/Modul:	Programmieren II für Wirtschaftsinformatiker
Erstellungsdatum:	September 6, 2025

Aufgabe 1: Praktische Übungen und Projektdurchführung: Beispielprojekte, Tests mit ScalaTest, Debugging

In dieser Aufgabe arbeiten Sie an einem praktischen Beispielprojekt in Scala, erstellen Tests mit ScalaTest und debuggen Fehler im Build und Code. Befolgen Sie die Schritte und dokumentieren Sie Ihre Ergebnisse.

a) Projektsetup und Struktur

Erstellen Sie ein neues sbt-Projekt und legen Sie eine klare Struktur fest. Beschreiben Sie die wichtigsten Dateien und Verzeichnisse, z. B. `build.sbt`, src/main/scala, src/test/scala. Legen Sie fest, wie eine rein funktionale Programmierpraxis in das Projekt integriert wird.

b) Tests mit ScalaTest

Entwerfen Sie ScalaTest-Tests, die zentrale Eigenschaften der geplanten Funktionen prüfen. Verwenden Sie geeignete Styles (z. B. FunSuite oder FlatSpec) und berücksichtigen Sie Randfälle wie leere Listen oder negative Werte. Dokumentieren Sie, welche Arten von Tests Sie verwenden und warum.

c) Debugging

Stellen Sie typische Fehlersituationen in Ihrem Beispielprojekt bereit und dokumentieren Sie die Debugging-Schritte. Nutzen Sie sbt, die REPL, Logs und ggf. Debugging-Tools. Beschreiben Sie, wie Sie Fehler reproduzieren, isolieren und nachweisen, dass sie behoben sind.

d) Projektdokumentation

Erstellen Sie eine kurze Projektdokumentation, die Ziel, Technologien, Setup-Schritte und Lernziele festhält. Ergänzen Sie Hinweise zur Weiterentwicklung und zur Nachverfolgung von Testergebnissen.

Aufgabe 2: Test- und Debugging-Strategien im Team

In dieser Aufgabe setzen Sie sich mit Strategien auseinander, wie Tests in den Buildprozess integriert werden und wie Debugging- sowie Code-Review-Prozesse im Team ablaufen.

a) Build- und Test-Integration

Skizzieren Sie eine Build- und Test-Strategie für ScalaProjekte. Berücksichtigen Sie den Einsatz von ScalaTest, PropertyChecks und AsyncTests sowie eine sinnvolle Aufteilung in Unit-, Integrations- und End-to-End-Tests. Definieren Sie klare Kriterien, wann Tests als bestanden gelten.

b) Debugging-Strategien im Team

Beschreiben Sie, wie Debugging im Team organisiert wird: Logging-Standards, reproduzierbare Bug-Dokumentation, Infrastruktur zur Nachverfolgung von Fehlern sowie regelmäßige Code-Reviews, die Debugging-Fortschritte unterstützen.

c) Praktische Übung

Führen Sie eine kleine Übungsaufgabe durch, bei der ein fehlerhaftes Verhalten gezielt reproduziert, gefunden und behoben wird. Dokumentieren Sie die Schritte, die Sie unternommen haben, sowie die Ergebnisse der Tests vor und nach der Behebung.

d) Abschlussreflexion

Verfassen Sie eine kurze Reflexion über das Gelernte, die Rolle von Tests und Debugging in der Praxis sowie den Nutzen von Teamprozessen bei der Projektdurchführung.

Lösungen

Lösung zu Aufgabe 1

a) Projektsetup und Struktur

Für eine rein funktionale Herangehensweise in einem Scala-Projekt empfiehlt sich eine klare, unveränderliche Struktur sowie ein isoliertes Build-Setup. Beispielhafte Lösungsvorschläge:

- Aufbau eines sbt-Projekts

```
name := "FPScalaProj"
version := "0.1.0"
scalaVersion := "2.13.11"

libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "3.2.9" % Test
)
```

- Verzeichnisstruktur

- build.sbt
- project/
  - plugins.sbt
  - build.properties
- src/main/scala/
  - com/example/fpproblem/Expr.scala (Abstrakte Datentypen)
  - com/example/fpproblem/Utils.scala (rein funktionale Hilfsfunktionen)
- src/test/scala/
  - com/example/fpproblem/SumSuite.scala (ScalaTest-Tests)
  - com/example/fpproblem/EvaluatorSuite.scala

- Beispielinhalte (rein funktionale Praxis)

```
// src/main/scala/com/example/fpproblem/Utils.scala
package com.example.fpproblem

object FPUtils {
  // rein funktionale, reine Funktion
  def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)

  // Beispiel für unveränderliche Datenstrukturen
  def reverseSafe[A](xs: List[A]): List[A] = xs.reverse
}


// src/main/scala/com/example/fpproblem/Expr.scala
package com.example.fpproblem
```

```
sealed trait Expr
case class Num(n: Int) extends Expr
case class Add(a: Expr, b: Expr) extends Expr

object Evaluator {
  def eval(e: Expr): Int = e match {
    case Num(n)     => n
    case Add(a, b)  => eval(a) + eval(b)
  }
}
```

- Wie wird rein funktionale Programmierpraxis integriert? - Vermeidung von var- und Nebenwirkungen; Nutzung von val und rekursiven Funktionen. - Nutzung von immutable Collections; Übergabe von Datenstrukturen statt In-Place-Änderungen. - Fehlerbehandlung über Options, Try oder Either statt Exceptions, wo sinnvoll. - Klare Separation von Daten (Datenmodell) und Funktionen (Logik); Funktionen first-class machen. - Einsatz von Typkonstrukten wie ADTs (z. B. sealed trait Expr) zur sicheren Musterung.

b) Tests mit ScalaTest

- Auswahl des Styles: FunSuite oder FlatSpec (hier: FunSuite) mit Matches-/Should-Herstellern.
- Beispielhafte Tests:

```
// src/test/scala/com/example/fpproblem/SumSuite.scala
package com.example.fpproblem

import org.scalatest.funsuite.AnyFunSuite
import org.scalatest.matchers.should.Matchers

class SumSuite extends AnyFunSuite with Matchers {
  test("sum of empty list is 0") {
    FPUtils.sum(Nil) shouldBe 0
  }

  test("sum of positive numbers") {
    FPUtils.sum(List(1, 2, 3)) shouldBe 6
  }

  test("sum handles negatives") {
    FPUtils.sum(List(-1, -2, 3)) shouldBe 0
  }

  test("sum of a large list") {
    FPUtils.sum((1 to 1000).toList) shouldBe 500500
  }
}
```

- Begründung der Tests - Randfälle: leere Liste, negative Werte, große Listen. - Auswahl des

Styles (Lesbarkeit, einfache Erweiterbarkeit) und Einsatz von Matchers für klare Assertions. - Hinweise zur Testabdeckung (ggf. mit scoverage) hinzufügen.

c) Debugging

- Typische Fehlersituationen und Vorgehen: - Fehlerfall 1: Sum(FPUtils.sum) liefert falsches Ergebnis - Reproduzieren: sbt test; Stacktrace prüfen. - Isolieren: Prüfen, ob der Fehler in List-Verarbeitung oder im Folding liegt. - Nachweis der Behebung: neue/angepasste Tests laufen grün. - Fehlerfall 2: Evaluator.scala wirft Pattern-Match-Fehler bei unbekanntem Expr - Reproduzieren: Tests hinzufügen, die unbekannte Formen abdecken. - Lösung: Abdeckung aller Fälle (oder Default-Fall) sicherstellen. - Debugging-Werkzeuge und Praktiken - sbt-REPL für explorative Prüfung: sbt console. - Logs statt Println, z. B. SLF4J/Logback (Konfiguration in src/main/resources). - Reproduzierbare Bug-Dokumentation (Environment, Input, Schritt-für-Schritt-Reproduktion). - Regelmäßige Code-Reviews fokussieren sich auf das Debugging-Verhalten der Änderungen. - Vorgehen in der Praxis - Reproduzieren, isolieren, Hypothesen testen, fixen, Tests erneut ausführen, Review-Input einholen, dokumentieren.

d) Projektdokumentation

- Ziel des Projekts - Demonstration rein funktionaler Prinzipien in Scala; Gegenüberstellung zu OO-Paradigmen in einfachen Beispielen. - Technologien - Scala 2.13, sbt, ScalaTest, ggf. sbt-scoverage, Option/Try/Either, immutable Collections. - Setup-Schritte - Projekt aufsetzen, Build-Skript, Abhängigkeiten installieren, Tests ausführen. - Beispielbefehle:

```
$ sbt test
$ sbt console
$ sbt clean coverage test
```

- Lernziele - Verstehen von Funktionsprinzipien, Abstraktion durch Algebraische Datentypen, Umgang mit Nebenläufigkeit/Parallelität auf funktionale Weise (z. B. Future, map/flatMap). - Hinweise zur Weiterentwicklung - Einführung Property-based Tests (ScalaCheck), End-to-End-Tests, Integration in CI. - Erweiterung der Domäne (z. B. weitere FP-Konzepte, Monaden-Lernen, State-Threading). - Nachverfolgung von Testergebnissen - Nutzung von CI-Dashboards, Coverage-Berichte, Pull-Request-Checks, klare Metriken (Grüne Tests, Coverage z. B. 80–90

Lösung zu Aufgabe 2

a) Build- und Test-Integration

- Grundidee: Eine robuste Build- und Test-Strategie, die Unit-, Integrations- und End-to-End-Tests sinnvoll trennt und mittels ScalaTest, PropertyChecks und Async-Tests abbildet. - Aufbauvorschlag (SBT)

```
name := "FPScalaProj"
version := "0.1.0"
scalaVersion := "2.13.11"

lazy val core = (project in file("core"))
  .settings(
    name := "fp-core",
    libraryDependencies ++= Seq(
      "org.scalatest" %% "scalatest" % "3.2.9" % Test,
      "org.scalacheck" %% "scalacheck" % "1.15.4" % Test
    )
  )

lazy val integration = (project in file("integration"))
  .settings(
    name := "fp-integration",
    libraryDependencies ++= Seq(
      "org.scalatest" %% "scalatest" % "3.2.9" % Test
    )
  )

// Mehrere Module ggf. per aggregate verbinden
```

- Plugins und Coverage

```
in project/plugins.sbt
addSbtPlugin("org.scoverage" % "sbt-scoverage" % "1.8.3")

coverageEnabled := true
coverageMinimum := 80
coverageFailOnMinimum := true
```

- Testarten - Unit-Tests: reine Funktionen, keine Seiteneffekte. - Property-Checks: scala-check zur Erzeugung von Randfällen. - Async-Tests: Verwendung von AsyncFunSuite, um Future-basiertes Verhalten zu prüfen. - End-to-End-Tests: Separate Module oder Subprojekt mit vollständiger Konfiguration (z. B. REST-Client-Tests, UI-Akzeptanztests via Selenium bzw. Play-ähnliche Tests).

- Definition von Erfolgskriterien - Alle Unit-Tests grün. - Integrations- und End-to-End-Tests laufen grün. - Mindestens 80–90- Keine build-warnings; CI-Build status grün.

b) Debugging-Strategien im Team

- Logging-Standards - Konsistentes Logging über SLF4J/Logback; Level-Flags (DEBUG, INFO, WARN, ERROR) gezielt einsetzen. - Keine sensiblen Daten in Logs; strukturierte Logs (Kontextinformationen, IDs, Input-Parameter, Versionen). - Reproduzierbare Bug-Dokumentation - Issue-Ticket mit Environment, Reproduktionsschritten, verwendeten Eingaben, erwarteter vs. tatsächlicher Output. - Screenshots/Logs anhängen; Zeitstempel beachten. - Infrastruktur zur Nachverfolgung von Fehlern - Nutzung eines Issue-Trackers (GitHub Issues, JIRA o.ä.), Verknüpfung zu Commits/PRs. - Automatisierte Tests, die Bug-Reproduktionen abdecken, mit Verweis auf das Ticket. - Code-Reviews - Review-Fokus auf Debugging-Ansätze, Nachvollziehbarkeit der Fehlerursache, Reproduzierbarkeit der Schritte, Minimierung von Seiteneffekten. - Checks: Wurden Randfälle abgedeckt? Wurden Tests angepasst/ergänzt?

c) Praktische Übung

- Ziel der Übung: Ein gezielt reproduzierbares fehlerhaftes Verhalten wird gefunden, reproduziert, behoben und verifiziert. - Vorgehen (Beispiel mit einer sicheren Division) - Ausgangsproblem: Eine Funktion divide(a, b) wirft bei b = 0 keine sichere Fehlermeldung, sondern führt zu einer Ausnahme. - Reproduzieren:

```
def divide(a: Int, b: Int): Int = a / b
assert(divide(4, 0) throws ArithmeticException) // Beispiel-Reproduktion
```

- Isolieren: Tests hinzufügen, die Division durch Null abdecken. - Beheben: Umgehung durch sichere Typen:

```
def safeDivide(a: Int, b: Int): Either[String, Int] =
  if (b == 0) Left("Division durch Null")
  else Right(a / b)
```

- Verifizieren: Tests laufen grün; Änderungen in der Codebasis überprüfen. - Dokumentation der Ergebnisse - Reproduktionsschritte, beobachtetes vs. erwartetes Verhalten, Commit-Message. - Verweis auf aktualisierte Tests und ggf. neue Property-Tests.

d) Abschlussreflexion

Die Integration von Tests und Debugging im Team erhöht die Nachvollziehbarkeit von Fehlern, reduziert Regressionsrisiken und fördert eine klare Kommunikationskultur. Durch automatisierte Tests, konsistente Logging-Praktiken und strukturierte Code-Reviews lassen sich Bugs frühzeitig erkennen und beheben. Die Kombination aus Unit-, Integrations- und End-to-End-Tests sowie Property-Based Testing stärkt das Verständnis funktionaler Konzepte und deren Auswirkung in praxisnahen Anwendungsszenarien. Teamprozesse schaffen Transparenz, verringern Duplizierung von Debugging-Arbeiten und unterstützen eine kontinuierliche Lern- und Verbesserungsfähigkeit.