Software Engineering: Design & Construction

Department of Computer Science
Software Technology Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Practice Exam **December 20, 2016**

First Name	
Last Name	
Matriculation Number	
Course of Study	
Department	
Signature	

Permitted Aids

- Everything except electronic devices
- Use a pen with document-proof ink (No green or red color)

Announcements

- Make sure that your copy of the exam is complete (10 pages).
- Fill out all fields on the front page.
- Do not use abbreviations for your course of study or your department.
- Put your name and your matriculation number on all pages of this exam.
- The time for solving this exam is 45 minutes.
- All questions of the exam must be answered in English.
- You are not allowed to remove the stapling.
- All tasks can be processed independently and in any order.
- You can use the backsides of the pages if you need more space for your solution.
- Make sure that you have read and understood a task before you start answering it.
- It is not allowed to use your own paper.
- Sign your exam to confirm your details and acknowledge the above announcements.

Topic	1	2	3	Total
Points				
Max.	12	11	7	30

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴⌴

Topic 1: Introduction — 12P

a) Testing — 1P

Why do we consider “Testing” to be part of the design?

b) Design — 1P

From a software design point-of-view: Why do we develop new language features?

c) Design Patterns — 1P

Does the Strategy Pattern support the Single-Responsibility Principle? Justify your answer.

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴⌴

d) Variance 5P

Take a look at the following Scala code example:

```
class MySuperclass
class MyClass extends MySuperclass
class MySubclass extends MyClass
//As a reminder:
//Boolean <: AnyVal <: Any
//everything <: Any
//Nothing <: everything
//Null <: every AnyRef

val f1: (Any) => Boolean = ???
val f2: (Seq[AnyRef]) => Boolean = ???
val f3: (Any) => Any = ???
val f4: (Seq[MySubclass]) => Boolean = ???
val f5: (Seq[MySuperclass]) => Null = ???
val f6: (List[_]) => AnyVal = ???

val f: (Seq[MyClass]) => Boolean = // f1 ?, f2 ?, ..., f6 ?
```

The last line is just a short hand for assigning each variable f_1 ... f_6 to f.

Now state for each f_i if this assignment is valid. (3P)

Discuss for **one** case why it succeeds or why it is rejected by the compiler. Use the terminology covariance and contravariance. (2P)

e) Fragile Base Classes 4P

Listing 1 shows a small library for implementing shopping carts. Listing 2 shows an example of how the library can be used to implement a `CountingCart`, which keeps track of how many products are currently in the cart. At a later point in time the developer of the base library (Listing 1) decides to add a `clear` method as shown in Listing 3. Discuss two issues of the following design related to the **fragile base class problem** and give concrete examples for each of them.

Listing 1: Provided Library

```
import scala.collection.mutable

class Product(name: String, price: Double)

class Cart {
  private[this] val products = mutable.Set[Product]()

  def add(p: Product): Unit = products += p

  def addAll(p: Seq[Product]): Unit = {
    for (product <- p) {
      add(product)
    }
  }

  def remove(p: Product): Unit = products -= p
}
```

Listing 2: Example Usage

```
class CountingCart extends Cart {
  private[this] var numberOfItems = 0

  override def add(p: Product): Unit = {
    numberOfItems = numberOfItems + 1
    super.add(p)
  }

  override def remove(p: Product): Unit = {
    numberOfItems = numberOfItems - 1
    super.remove(p)
  }

  def size(): Int = numberOfItems
}
```

Listing 3: Extension of Provided Library

```
def clear(): Unit = products.clear()
```

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴⌴

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴⌴

Topic 2: S.O.L.I.D. 11P

a) Origins of the Interface Segregation Principle (ISP) 4P

Folklore says that the ISP was invented by Robert C. Martin while working for Xerox on a flexible printer system that would not only print but also perform a variety of printer-related tasks like stapling and faxing. There was one main *Job* class that was used by most other tasks. During the development of the system, making a modification to the Job class therefore affected many other classes and a compilation cycle would take a long time. This made it very time-consuming to modify the system. Since the big Job class had many methods specific to a variety of different clients, the solution was to introduce multiple smaller interfaces for each client. A client would then only operate on a specific, small interface to the Job class. Modifications to the Job class would then affect a smaller number of interfaces and therefore less classes had to be recompiled.

From this description, it sounds like there are other S.O.L.I.D. principles that could have been violated by Xerox's design. Which ones, why, and how would you redesign the system to adhere to them?

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴

b) Design Review 7P

In Figure 1, you find a basic class hierarchy for products of an online shop. Assume that there are no checked exceptions, but that the given exceptions are complete, i.e., a method will not throw any exception that is not given in its signature. For each S.O.L.I.D. principle, say how the design violates or honors it (as much as you can tell from the diagram). Suggest improvements and draw a UML diagram that incorporates your improvements.

You can use the back of the page if the given space does not suffice.

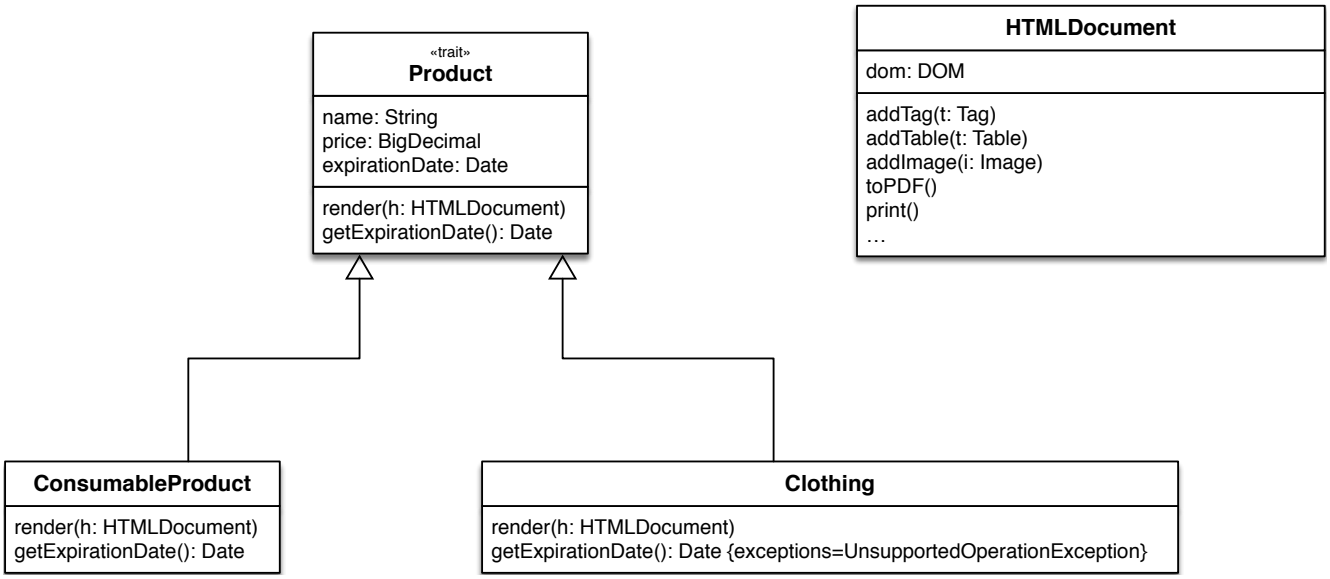

Figure 1: Class diagram for the product hierarchy

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴⌴

Topic 3: Scala 7P

a) Mixin Composition and Linearization 4P

Given the following Scala code example:

```
abstract class Pizza() {
  def toppings: List[String]
  def price: BigDecimal
}

trait Large extends Pizza {
  abstract override def price: BigDecimal = super.price * 2
}

trait Cheese extends Pizza {
  abstract override def toppings: List[String] = "Cheese" :: super.toppings
  abstract override def price: BigDecimal = super.price + 0.5
}

trait Salami extends Pizza {
  abstract override def toppings: List[String] = "Salami" :: super.toppings
  abstract override def price: BigDecimal = super.price + 1.0
}

class BasePizza extends Pizza {
  def toppings = List("Tomato Sauce")
  def price = 5.0
}

class MargheritaPizza extends BasePizza with Cheese
class LargeMargheritaPizza extends MargheritaPizza with Large
class SalamiPizza extends MargheritaPizza with Salami
class DoubleCheeseSalamiPizza extends MargheritaPizza with Salami with Cheese
```

1. State the **linearization** for the class `DoubleCheeseSalamiPizza` (2P)

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴⌴

2. Is there a difference between the following two ways of defining a large salami pizza? Justify your answer. (1P)

```
val salamiPizza1 = new LargeMargheritaPizza with Salami
val salamiPizza2 = new SalamiPizza with Large
```

3. How does the shown implementation compare to an implementation using the classical decorator design pattern? (1P)

Name: _____ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴⌴

b) Scala vs Java 1P

What is the difference between *super* calls in Scala compared to *super* calls in Java?

c) OCP and Pattern Matching 2P

Given the following pattern matching example. Does this implementation support the Open-Closed Principle? Justify your answer.

```
trait Book
class ScienceFictionNovel extends Book
class CrimeNovel extends Book

def getGenre(b: Book) = b match {
  case (_: ScienceFictionNovel) => "Science Fiction"
  case (_: CrimeNovel) => "Crime"
}
```