Course Description
Software Engineering 1

1 Course description

1.1 Introduction

This is the course descriptor for the Software Engineering course.

Software engineering 1 covers functors and monads, which are advanced patterns used to structure code in a composable way to solve a variety of problems in query-management, UI rendering, parallel processing, and more. The focus of the course will be on the patterns, and not on a pervasive philosophy of functional programming. These patterns are used in the practice of modern mainstream programming languages, from C# and Java to the JavaScript world. For this reason, we will build examples in TypeScript, a modern popular statically-typed variant of JavaScript with both a rich type system and broad appeal for professionals.

Thanks to this course, students will understand advanced patterns and the important principle of composition at a higher level of abstraction.

1.1.1 Learning goals

The course has the following learning goals: - (ICODE) students can *recognize and formally interpret* important properties of abstract patterns such as associativity, preservation of identity, or homomorphism; - (WCODE) students can *write small snippets of code* in order to complete the implementation of a known abstract pattern.

The course, and therefore also the learning goals, are limited to **referentially transparent**, **type-safe** programs written in TypeScript.

The corresponding competences connected to these learning goals are: - realisation.

1.1.2 Learning materials

The mandatory learning materials are: - reader of the lectures, which will be made available one week after the lecture - materials on the Github repository (github.com/hogeschool), which is already online

1.1.3 Exam

The exam is divided in two parts: #### Theory - (MC) a series of fifteen (15) *multiple-choice* questions on the formal properties of examples of code, ranging from associativity to functoriality and more.

Practice

- (BA) a series of five (5) *backward assignments*, that is exercises where students, given partial code implementing a monad or function, are requested to fill in the missing code that matches the given specification. All monads and functors seen in class may be asked here.

Scoring The theory questions are worth one point each. The practice questions are worth three points each. The maximum total score is therefore 30.

Both parts of the exam must score above 55%: if there are more than 6 errors in the theory, or more than 2 errors in the practice, then the points for the course will not be awarded and the insufficient part will need to be retaken.

The final grade is computed as follows:

grade = score * 30 / 10

This is justified: even though there are more MC's than BA's, the BA's require actively *writing code*, which is significantly more complex than providing a description of formal properties, seen that it combines technical understanding and creativity.

Exam matrix The exam covers all learning goals.

Exam part	ICODE	WCODE
MC	V	
BA		V

1.2 Lecture plan

The course is made up of eight lectures, usually planned as one lecture per lesson week (for details, see lesson schedule on HINT).

The lesson units covered by the course are the following: - Types, functions, and their composition - Functors, their implementation, and properties - Monoids in set, monoids in programming - Monads as monoids and (endo-) functors - Very simple monads: Identity - Sum monad(s): Plus, Exception, Option - State monad - Composition of monads: Parser - Composition of monads: Coroutine

Note that each lesson unit does not necessarily correspond to one lesson week (for example, one lesson unit could span during two lesson weeks or two lesson units could be handled during one lesson week).

1.3 Study points and contact time

The course awards students 4 ECTS, in correspondence with 112 hours of study.

The course consists of eight frontal lectures for the theory, and eight assisted practicums. The rest is self study.

2 Lecture plan

The course revolves around the building of abstractions which can be safely composed. We will begin with a short discussion on our basic toolbox, referentially transparent functions and types, and then start defining abstract patterns such as functors and monoids.

We will then move onto a combination of the concepts of functor and monoid, thereby obtaining monads. We will then show various monads which have been proven to be powerful in actual programming practice.

2.1 Basic toolbox

- Functions, types
- Composition of types and referential transparency

2.2 Functors

- Generic types as functions between types
- Generic types *induce* `map`, which is a function between functions
- Examples:
  - Identity
  - Pair (left, right, product)
  - Sum (left, right, product)
  - Option
  - List
- Properties of functors: preservation of identity and composition
  - Useful tip: performance optimization by merging maps
- Functors can be composed into new functors
  - List(Option)

2.3 Monoids

- Fundamental structure which arises everywhere
- Joining, unit, associativity
- Examples:
  - string, plus, empty
  - number, plus, 0

  - number, times, 1
  - list, concat, empty
  - ...
- Monoids over functors:
- Reformulation of monoids in functional terms: unit becomes a function (eta/return)

2.4 Monads introduction

- Monads arise everywhere we can augment a functor with joining, unit, and associativity
- The bind operator
- Examples:
  - Identity (bind is simply function composition!)
  - Pair (left and right)

We now move on to more and more advanced monads which are quite useful in practice.

2.4.1 Sum(s)

- Concept: safe representation of a happy-flow and an error flow

- Sum (left and right)
- Option
- Exception
- Examples:
  - Optional in Java
  - Maybe in Haskell
  - Optional in C++

2.4.2 State

- Concept: arbitrating state-management, own DSL
- Example:
- Mini-language

2.4.3 Combining monads: parser

- Concept: different monads can be composed together in order to form a larger monad
  - The larger monad is still a monad!
- Let us show that parser is the combination of state and option/exception
- Example:

  - Parser for the mini-language from the State
  - Mini-language with exceptions

2.4.4 Coroutines

- Concept: modelling domain-specific concurrency in a concurrency-unsafe host-language
- Examples:
  - Promises
  - TPL
  - Mini-language with exceptions and breakpoints