HOGESCHOOL ROTTERDAM / CMI

Development 4

INFDEV02-4
2015-2016

Number of study points:	4 ects
Course owners:	The DEV team

Module description

Module name:	Development 4
Module code:	INFDEV02-4
Study points and hours of effort:	This module gives 4 ects, in correspondence with 112 hours: • 2 X 3 x 6 hours of combined lecture and practical • the rest is self-study
Examination:	Written examination and practicums (with oral check)
Course structure:	Lectures, self-study, and practicums
Prerequisite knowledge:	INFDEV02-1, INFDEV02-2, and INFDEV02-3.
Learning materials:	• Book: Design patterns, elements of reusable object-oriented software; author Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. • Slides: found on N@tschool and on the GitHub repository github.com/hogeschool/INFDEV02-4 • Exercises and Assignments, to be done at home and during practical part of the lectures (pdf): found on N@tschool and on the GitHub repository github.com/hogeschool/INFDEV02-4
Connected to competences:	realiseren en ontwerpen
Learning objectives:	At the end of the course, the student: • **is able to use** and **create** interfaces and abstract classes. `(ABS)` • **has developed skills** to adopt a new programming language with little support. `(LEARN)` • **is able to apply** the concepts of data encapsulation, inheritance, and polymorphism to software. `(ENC)` • **can apply** the concepts of data types. `(TYPE)` • **understands** basic human factors. `(BHF)`
Course owners:	The DEV team
Date:	April 25, 2016

1 General description

Designing a object-oriented software is a complex and time-consuming task. A misuse of typical object-oriented language features, such as polymorphism or inheritance, cause errors and inflexible solutions. Design patterns provide the means to write code which is both maintainable and re-usable.

1.1 Relationship with other teaching units

Subsequent programming courses build upon the knowledge learned during this course.

The course analysis 4 covers software testing, which is an activity that benefits from the competent use of design patterns.
Knowledge acquired through the programming courses is also useful for the projects. A word of warning though: projects and development courses are largely independent, so some things that a student learns during the development courses are not used in the projects, some things that a student learns during the development courses are indeed used in the projects, but some things done in the projects are learned within the context of the project and not within the development courses.

2 Course program

The course is structured into six lectures. The six lectures take place during the six weeks of the course, but are not necessarily in a one-to-one correspondence with the course weeks. For example, lectures one and two are fairly short and can take place during a single week.

2.1 Chapter 1 - Intro to design patterns

Topics

- What are design patterns?
- Visiting polymorphic instances.
- Pure object-oriented Visitor pattern
- Functional approach to Visitor pattern

2.2 Chapter 2 - Iterating collections

Topics

- What is an iterator?
- Why using iterators?
- Iterating a generic collection.

2.3 Chapter 3 - Extending behaviours

Topics

- Decorator pattern.
- Decorating over iterator

2.4 Chapter 4 - Entity construction

Topics

- Creating object without specifying the class type.

2.5 Chapter 5 - Adapting behaviours

Topics

- Managing different objects with shared behaviours.
- Adapter pattern.

3 Assessment

The course is tested with two exams: A series of Assignments which have to be handed in, but won't be graded. There will be an Oral check, which is based on the Assignments and a written exam. The final grade is determined as follows:

`if Theoretical exam-grade` $>= 5.5$ `then return Oral check-grade else return 0`

Motivation for grade A professional software developer is required to be able to program code which is, at the very least, *correct*.
In order to produce correct code, we expect students to show: *i*) a foundation of knowledge about how a programming language actually works in connection with a simplified concrete model of a computer; *ii*) fluency when actually writing the code.
The quality of the programmer is ultimately determined by his actual code-writing skills, therefore the written exam will contain require you to write code, this ensures that each student is able to show that his work is his own and that he has adequate understanding of its mechanisms.

3.1 Theoretical examination INFDEV02-4

The general shape of a Theoretical exam for `INFDEV02-4` is made up of a short series of highly structured open questions. In each exam the content of the questions will change, but the structure of the questions will remain the same. For the structure (and an example) of the theoretical exam, see the appendix.

3.2 Practical examination INFDEV02-4

There are 2 Assignments which are mandatory, and formatively assessed for Feedback.

- All assignments are to be uploaded to N@tschool or Classroom in the required space (Inlevermap or assignment);
- Each assignment is designed to assess the students knowledge related to one or more Learning objectives. If the teacher is unable te assess the students' ability related to the appropriate Learning objective based on his work, then no points will be awarded for that part.
- *The teachers still reserve the right to check the practicums handed in by each student, and to use it for further evaluation.*
- The university rules on fraude and plagiarism (Hogeschoolgids art. 11.10 – 11.12) also apply to code;

Structure of exam INFDEV02-4

The general shape of a theoretical exam for `DEV 3` is made up of only two, highly structured open questions.

3.2.0.1 Question 1:

General shape of the question: *Given the following class definitions, and a piece of code that uses them, fill in the stack, heap, and PC with all steps taken by the program at runtime.*
Concrete example of question:

```
1  public interface Option<T>
2  {
3
4    U Visit<U>(Func<U> onNone, Func<T, U> onSome);
5  }
6  public class Some<T> : Option<T>
7  {
8    T value;
9    public Some(T value) { this.value = value; }
10   public U Visit<U>(Func<U> onNone, Func<T, U> onSome)
11   {
12     return onSome(value);
13   }
14 }
15 public class None<T> : Option<T>
16 {
17   public U Visit<U>(Func<U> onNone, Func<T, U> onSome)
18   {
19     return onNone();
20   }
21 }
22 ...
23 Option<int> number = new Some<int>(5);
24 int inc_number = number.Visit(() => { throw new Exception("Expecting a value...
       "); }, i => i + 1);
25 Console.WriteLine(inc_number);
```

Concrete example of answer: ...
Points: *3 (30% of total).*
Grading: *one point per correctly filled-in execution step.*
Associated learning objective: is able to use and **create** interfaces and abstract classes. `(ABS)`

3.2.0.2 Question 2:

General shape of question: *Given the following class definitions, and a piece of code that uses them, fill in the declarations, class definitions, and PC with all steps taken by the compiler while type checking.*
Concrete example of question:

```
1  public interface Option<T>
2  {
3    U Visit<U>(Func<U> onNone, Func<T, U> onSome);
4  }
5  public class Some<T> : Option<T>
6  {
7    T value;
8    public Some(T value) { this.value = value; }
9    public U Visit<U>(Func<U> onNone, Func<T, U> onSome)
10   {
11     return onSome(value);
```

```
12    }
13  }
14  public class None<T> : Option<T>
15  {
16    public U Visit<U>(Func<U> onNone, Func<T, U> onSome)
17    {
18      return onNone();
19    }
20  }
21  ...
22  Option<int> number = new Some<int>(5);
23  int inc_number = number.Visit(() => { throw new Exception("Expecting a value...
        "); }, i => i + 1);
24  Console.WriteLine(inc_number);
```

Concrete example of answer: ...
Points: *4 (30% of total).*
Grading: *Grading: one point per correctly filled-in type checking step.*
Associated learning objective: can apply the concepts of data types. (TYPE)

3.2.0.3 Question 3:

General shape of question: *Given the following implementation of a design pattern fill in the missing blocks of code.*
Concrete example of question:

```
1  public interface Option<T>
2  {
3   BLOCK 1:
4
5  }
6   BLOCK 2:
7
8  {
9    T value;
10  BLOCK 3:
11
12   public U Visit<U>(Func<U> onNone, Func<T, U> onSome)
13   {
14     return onSome(value);
15   }
16 }
17 public class None<T> : Option<T>
18 {
19  BLOCK 4:
20
21
22 }
23 ...
24 Option<int> number = new Some<int>(5);
25 int inc_number = number.Visit(() => { throw new Exception("
       Expecting a value..."); }, i => i + 1);
26 Console.WriteLine(inc_number);
```

Points: *4 (40% of total).*
Grading: *Grading: one point per correctly filled-in block of code.*
Associated learning objective: is able to use and **create** interfaces and abstract classes. (ABS)

Appendix 1: Assessment matrix

	Dublin descriptors
ABS	1, 2, 4
LEARN	1, 4, 5
ENC	1, 2, 4
TYPE	1, 2, 4
BHF	1, 2, 4

Dublin-descriptors:

1. Knowledge and understanding
2. Applying knowledge and understanding
3. Making judgments
4. Communication
5. Learning skills