HOGESCHOOL ROTTERDAM / CMI

Development 4

INFDEV02-4
2016-2017

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: • **can apply** the concept of type variables. `(TYPVAR)` • has **understood** the behavioural design patterns. `UNDBEH` • can **implement** the behavioural design patterns. `IMPBEH` • has **understood** the structural design patterns. `UNDSTR` • can **implement** the structural design patterns. `IMPSTR` • has **understood** the creational design patterns. `UNDCRE` • can **implement** the creational design patterns. `IMPCRE`
Course owners:	The DEV team
Date:	June 14, 2017

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.
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 course, but are not necessarily in a one-to-one correspondence with the course weeks.

2.1 Chapter 0 - reuse through generics

Topics

- Using generic parameters
- (**Advanced**) Using covariance and contravariance in the presence of generic parameters
- (**Advanced**) Designing interfaces and implementation in the presence of generic parameters

2.2 Chapter 1 - Intro to design patterns

Topics

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

2.3 Chapter 2 - Iterating collections

Topics

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

2.4 Chapter 3 - Extending behaviours

Topics

- Decorator pattern. `UNDSTR`
- Decorating over iterator. `IMPSTR`

2.5 Chapter 4 - Entity construction

Topics

- Creating object without specifying the class type. `UNDCRE`, `IMPCRE`

2.6 Chapter 5 - Adapting behaviours

Topics

- Managing different objects with shared behaviours. `UNDSTR`
- Adapter pattern. `IMPSTR`

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 Practicum check, which is based on the Assignments and a written exam. The final grade is determined as follows:

`if Theoretical exam-grade` $>= 75\%$ `then return Practicum 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 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 is an assignment for each pattern covered in the lessons that is mandatory. The assignment asks you to implement a GUI system in the fashion of Windows Form with an immediate drawing library, such as Monogame. In the assignment you have to show that you can use effectively the design patterns learnt during the course. Your GUI library should allow to create at least clickable buttons and text labels organized in a single panel. All assignments are to be presented at the end of the practical assessment to the teacher upon request
At the practicum check you will be asked to solve exercises on design patterns based on the assignment. The maximum score for this part is up to 10 points. 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 `INFDEV02-4` is made up of 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);
```

Points: *3 (30% of total).*
Grading: *one point per correctly filled-in execution step.*
Associated learning objective: can **implement** the behavioural design patterns. `IMPBEH`

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:

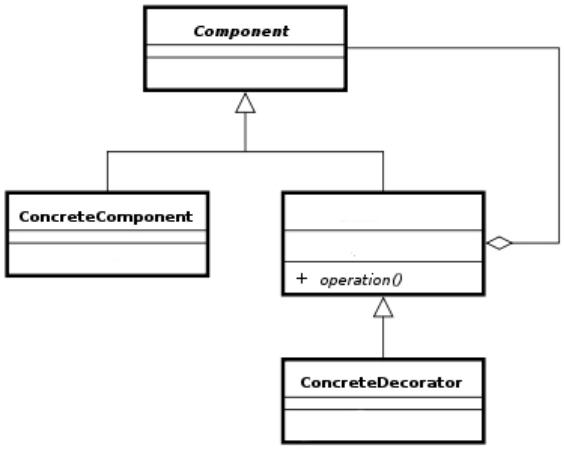
```
1  public interface Option<T>
2  {
3    U Visit<U>(Func<U> onNone, Func<T, U> onSome);
4  }
5  public class Some<T> : Option<T>
6  {
7    T value;
8    public Some(T value) { this.value = value; }
9    public U Visit<U>(Func<U> onNone, Func<T, U> onSome)
10   {
11     return onSome(value);
12   }
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);
```

Points: *3 (30% of total).*
Grading: *Grading: one point per correctly filled-in type checking step.*
Associated learning objective: has **understood** the behavioural design patterns. `UNDBEH`

3.2.0.3 Question 3: **General shape of question:** *Given the following UML diagram of a design pattern, fill in the missing parts.*

Concrete example of question:



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
UNDBEH	1, 4, 5
IMPBEH	2, 3, 5
UNDSTR	1, 4, 5
IMPSTR	2, 3, 5
UNDCRE	1, 4, 5
IMPCRE	2, 3, 5

Dublin-descriptors:

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