

Правительство Российской Федерации

**Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Национальный исследовательский университет
«Высшая школа экономики»**

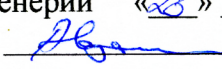
**Факультет компьютерных наук
Департамент программной инженерии**

Утверждаю
Академический руководитель
образовательной программы
по направлению 231000.62
«Программная инженерия»
К.Ю. Дегтярев
«01» 12 2014 г.


**Программа дисциплины
Конструирование программного обеспечения**

для направления 231000.62 «Программная инженерия»
подготовки бакалавра

Автор программы:
Tom Verhoef, dr.ir,
T.Verhoeff@tue.nl

Одобрена на заседании Департамента программной инженерии «28» 08 2014 г.
Руководитель Департамента  С.М. Авдошин

Рекомендована Академическим советом образовательной программы
«Программная инженерия» «28» 11 2014 г.

Менеджер Департамента программной инженерии  Т.В. Климова

Настоящая программа не может быть использована другими подразделениями университета и другими вузами без разрешения департамента-разработчика программы.

1. Scope and Regulations

The course "Software Construction" ("Software Engineering" BS curriculum, 2nd year) syllabus lays down minimum requirements for student's knowledge and skills; it also provides description of both contents and forms of training and assessment in use. The course is offered to students of the Bachelor Program "Software Engineering" Faculty of Computer Science of the National Research University "Higher School of Economics" (HSE). The course is a part of B.Sc. curriculum pool of required courses and it is a single-module course (semester A quartiles 1 and 2). The duration of the course amounts to 28 Lecture hours. Also, 24 academic hours are intended for students' Self-Studying (SS) activity.

The course focuses on systematic design of larger object-oriented programs. We will introduce the appropriate concepts, terminology, and notations to help communicate about programs and about programming. Design is an activity involving options and motivated (rational) design decisions. The programming language Java is used as vehicle. It is not a Java language course, but instead we teach how Java language features can be used systematically to construct larger programs.

The syllabus is prepared for teachers responsible for the course (or closely related disciplines), teaching assistants, students enrolled on the course as well as experts and statutory bodies carrying out assigned or regular accreditations in accordance with

- educational standards of the National Research University The Higher School of Economics (NRU HSE)

2. Course Objectives

After this course the student should be able to elicit a set of requirements for, design, implement and evaluate (empirically) highly usable user-interfaces for a wide variety of application areas.

Students will attend lectures (for motivation, explanation of theory and examples, feedback on previous work, questions and discussion, in preparation for assignments), read study material (to learn about interaction design and evaluation), and work on practical group assignments that focus on evaluation and on interface design and implementation.

3. Learning Outcomes

Upon mastering the discipline, the successful students will:

- Know what is required for large application developing; which programming technologies fit which types of applications.
- Be able to elicit requirements for various types of applications; design program structure and build prototypes.
- Acquire skills/experience in application developing.

4. The Course within the Program Framework

The course is within the block of basic disciplines.

The course is based on the students' knowledge of mathematics, basics of information science, algorithm theory, and OOAD, and on modeling skills for methodological search of the solution.

The course prerequisite is successful completion of the Software Construction course (because it is used to design and implement a user-interface), common sense, good knowledge of English (because there are often descriptions of situations, given in English).

5. Topic-Wise Course Contents

№	Topic Name	Course Hours, Total	Audience Hours, Lectures	Self-Study
Module 1				
1	Introduction. Imperative core of Java	3	2	1
2	Divide and conquer. Java methods, functional decomposition	4	2	2
3	Pre/post contracts formal notation	5	3	2
4	Unit testing	4	2	2
5	Robustness, Exceptions	4	2	2
6	Data abstraction	5	3	2
Module 1, totally:		25	14	11
Module 2				
7	Type hierarchy, Iteration abstraction	4	2	2
8	Nested classes for iterators; generic data types	5	3	2
9	Simple GUI; Observer pattern, callbacks	4	2	2
10	Composite design pattern	4	2	2
11	GUI graphics, global organization Command Pattern	4	2	2
12	Concurrency, threads for background work	4	3	1
Module 2, totally:		25	14	11
Total:		50	28	22

Notes:

1. Each № above should correspond to the separate theme, the theme may span over one or more lectures.

2. Module above is about a half of semester. Modules 1 and 2 belong to the first semester. For the first semester the Total must be 14 lectures (i.e. 28 hours for lectures) and 22 hours for self-study activity.

6. Course Assessments

Assessment Type	Assessment Form	Week	Description
Intermediate (week)	Peach ³ Assignment	1	Series 1: DiceGame; Candy
		2	Series 2: Divide & Conquer; DiceGame Decomposed; Candy 2

		3	Series 3: Peer Review of Secure Key Collection; Secure Key Collection; DiceGame Contracts
		4	Series 4: Powerize; Count Digits with Radix
		5	Series 5: Statistics; DiceGame Robust
		6	Series 6: Mutable versus Immutable Fractions; DiceGame OO Alternative; DiceGame OO
		7	Series 7: Iterable IntRelation; IntRelation via Map of Sets; Robust IntRelation; Peer Review of Powerize
		8	Series 8: Generic Relation; Extra Iterator for IntRelation; Peer Review of Iterable IntRelation
		10	Series 9: Simple Kakuro Helper
		11	Series 10: Simple Kakuro Helper 2; Testing of KakuroCombinationGenerator
		12	Series 11: Undo-Redo Facility
		13	Series 12: Simple Kakuro Helper 3
		14	Kakuro Puzzle Assistant: Kakuro Puzzle Assistant (Background Solver); Kakuro Puzzle Assistant (Undo-Redo, Solver)
Final	Final Test	16	

The overall course grade G (10-point scale) is calculated as a sum of:

$$G = 0,5 T + 0,5 E,$$

where T is the written final test grade, E is the practice activity and home works assignments.

It is rounded (up or down) to an integer number of points. Taking part in both elements of assessment is required to obtain a final grade.

6.1. Summary Table: Correspondence of ten-point to five-point system's marks

Ten-point scale [10]	Five-point scale [5]
1 – unsatisfactory 2 – very bad 3 – bad	Unsatisfactory – 2
4 – satisfactory 5 – quite satisfactory	Satisfactory – 3
6 – good 7 – very good	Good – 4
8 – nearly excellent 9 – excellent 10 – brilliant	Excellent – 5

6.2. Assessment Criteria

Intermediate Assessments: Modules 1 and 2 — Peach³-based home works assignments.
Final Assessments: Exam at the end of Module 2.

The writing test is to assess the core course content. The time limit is 120 mins. One (10-point scale) grade is given for the test.

7. Detailed Course Contents

7.1. Topic 1: Imperative Core of Java

At compile time, a Java program consists of a collection of classes, where each class can have typed member variables and parameterized methods. Syntactic elements of imperative Java program texts include comments, predefined types (including integers, booleans, floating point numbers, strings, and arrays), literal values, local typed variables, expressions, assignment statements, statement blocks, control statements, and method calls on standard libraries (including input/output facilities).

At run time, there is, in general, also a collection of objects, each object being instantiated from a class, where each variable in an object has a value from its type. During execution, statements are executed under control of variable values, thereby updating variable values and creating new objects. Objects that are no longer reachable will be destroyed by the automatic garbage collector.

Imperative non-procedural Java programs consist of one class with one parameterized static method, and without (global) variables and objects, but possibly with constant definitions.

See [4, §2.(1, 2, 5), §3.(1, 3–6), §7.(1, 2.(1, 2))].

7.2. Topic 2: Coding Standard

Program texts must be readable, for various reasons. This helps prevent mistakes when writing the text; it makes it easier to locate defects (both by the program author and by others); it makes it easier to evolve programs; it makes it easier to inspect and verify programs.

See note [2].

7.3. Topic 3: Divide & Conquer

Humans are bad at handling complex things as a whole. To manage such complexity, we apply Divide & Conquer: split a complex thing into smaller parts, handle the parts separately, and integrate the results. Such parts are referred to as modules or units. In view of the three steps, it would have been more appropriate to call this technique Divide, Conquer, and Rule.

Divide & Conquer offers many benefits, but at some cost. Therefore, it is important to balance the trade-offs.

7.4. Topic 4: Procedural Abstraction

An expression or statement block can be made available for on-demand invocation, by encapsulating it inside a named method definition. For a statement block, this is a **void** method, and for an expression a method returning a typed result. The caller of the method can abstract from the implementation details as embodied in the expression or statement block, and focus only on the resulting effect. Method parameters can be used to abstract from the particular problem instance; they make the method more generally applicable. The designer of the expression or statement block can abstract from the specific context in which the method will be called.

Procedural Java programs consist of classes having only static methods and static variables, without objects.

See [4, §2.3.1, §4.(1–4), §7.2.3]

7.5. Topic 5: Contracts

When applying Divide & Conquer, it is important to formulate the subproblems clearly. Each module interface is specified syntactically and semantically in a so-called two-party contract. Such a contract defines assumptions (preconditions), effects (postconditions), and possibly also invariant relations (invariants). The two parties to the contract are typically referred to as client and provider. Reasoning about client code is always done in terms of the contract, never in terms of the provided implementation. Similarly, reasoning about provided implementation code is always done in terms of the contract, and never in terms of specific invocations by client code. The contract guarantees a separation of concerns; otherwise, Divide & Conquer will not work well.

See notes [8, 9], and [4, §4.6].

7.6. Topic 6: Functional Decomposition

This is a design technique that carries out Divide & Conquer (solely) with procedural abstraction. Decomposition is driven by considering the required functionality of the program being designed. In requirements, functionality can often be discovered in the verbs.

Design guidelines: aim for single-purpose, general methods (SRP = Single Responsibility Principle), low coupling, high cohesion, low complexity (no deeply nested control structures), avoid code duplication (DRY = Don't Repeat Yourself). Parameters, return values, global versus local data, recursion.

7.7. Topic 7: Javadoc

Comments in Java programs serve various purposes. A comment can provide meta-information about the program, such as the author name(s), date of creation and latest modification, and license information. It can document usage information, in particular, contracts. It can help visualize global structure, and it can explain small program fragments.

Javadoc is a form of comment that can be recognized as such, and that can be extracted and transformed into a set of hyperlinked HTML pages. Such a comment can be further structured by tags. Javadoc is typically used for usage documentation and contracts.

See [4, §4.5.4].

7.8. Topic 8: Unit Testing

When applying Divide & Conquer, the resulting parts can and must be checked separately, before integrating them. These checks are best automated, to make them reliable and repeatable. Such tests are known as unit tests. JUnit is a framework to assist in the development of automated test cases for Java classes.

7.9. Topic 9: Test-Driven Development

Once you have decided that unit tests are an integral part of the software to be delivered, it is worthwhile to reconsider the order development steps. In Test-Driven Development, unit tests are implemented before the code being tested:

1. Gather and analyze requirements.
2. Specify a class and its operations informally: javadoc summary sentences for the class and its key methods.
3. Specify the class more formally: an abstract model with invariants, headers, and contracts.
4. Result: a class without implementation, that is, no data representation and empty method bodies.
5. Create a corresponding unit test class.
6. Implement rigorous test cases.
7. Choose a data representation and implement the class methods.
8. Test the implementation.

For larger classes, steps 5 through 7 are applied for (groups of) methods separately.

7.10. Topic 10: Exceptions and Errors

To make contracts robust, interface conditions need to be checked at run time. When such a condition is violated, the normal flow of execution must be interrupted. In Java, the exception mechanism and the assert mechanism can be used for this. These mechanisms provide a minimally-intrusive way to deviate from the normal flow of execution. In particular, they can break out of (deeply) nested method calls, without extra code in the intermediate methods. However, the exception mechanism incurs a runtime penalty, and should not be abused for normal behavior.

See [4, §3.7, §8.(3, 4.1)].

7.11. Topic 11: Data Types and Data Abstraction

A (data) type consists of a set of values and related operations on those values. Java offers built-in primitive data types, and a class mechanism for user-defined data types. The class mechanism can be used to provide record types (labeled product), enumerations types, and Abstract Data Types (ADT).

Data needs to be stored, accessed, and operated on. Doing so efficiently (both in time and in memory) involves (sometimes advanced and complex) data structures and algorithms. Data abstraction is a facility that allows the designer to abstract from the details of these data structures and algorithms. In particular, the data representation and the implementation of operations can be encapsulated in an Abstract Data Type. In that way, it will be easy to change the representation and implementation, without affecting the using occurrences of the ADT.

The implementation of an ADT involves an abstraction function, mapping the data representation to abstract values, and a representation invariant, indicating which data representations are valid and represent an abstract value. An ADT is either immutable or mutable, depending on whether values (objects) of that type can or cannot change state after creation.

7.12. Topic 12: Data Decomposition, Relationship between Types

It turns out that decomposition driven by functionality (functional decomposition) leads to an architecture that is not so stable over time, because small changes in functionality can affect the decomposition drastically. Letting data considerations drive the decomposition (data decomposition) leads to a more stable architecture. In requirements, data can often be discovered in the nouns.

Data types can be related in various ways. They can be used next to each other glued together by ad hoc code. Objects of two types can be related to each other, that is, there is a mathematical relationship between two types, in the sense of a subset of the cartesian product of their value sets (one-to-one, one-to-many, or many-to-many). Objects of one type can contain or be composed of objects of one or more other types (aggregation, composition).

In data abstraction, the user of a data type reasons in terms of abstract values and operations on those values, abstracting from implementation details (how values are stored and manipulated).

Design guidelines: aim for single-purpose, general classes (SRP = Single Responsibility Principle), low coupling, high cohesion, low complexity (no deeply nested class definitions), avoid code duplication (DRY = Don't Repeat Yourself); favor composition over inheritance.

7.13. Topic 13: Type Hierarchy

Inheritance is a way of deriving one class definition from another, where all member variables and method signatures and implementations are inherited. The inherited class is also known as a subclass, and the class from which it is derived is called its superclass. In the subclass, additional member variables and operations can be introduced. It is also possible to override implementations of inherited method in the subclass. Thus, inheritance is a versatile mechanism for reuse, but it also has some dangers.

Polymorphism: Let U be a subclass of class T. In any place where an object of class T may be used, an object of U may be used. Every U 'behaves' like a T. That is, the program will compile, but not necessarily behave as expected. A variable of (static, compile-time) type T can hold any value of

type T or subclass of T. Thus, the (dynamic, run-time) type of the value of a variable can differ from the variable's (static, compile-time) type.

Liskov Substitution Principle (LSP): If operations in the subclass U adhere to the contracts of T, that is, their contracts are inherited from the superclass T, then we call U a subtype of T. In that case, substituting an object of a subtype for a type will not break the program semantically. It not only compiles, but works as expected.

Interfaces in Java provide another typing layer for objects.

See [4, §5.(3, 5–6, 7.1), §10.(1.4, 2.1)].

7.14. Topic 14: Iteration Abstraction

To iterate over a collection means to visit each item of the collection exactly once. The process of iteration involves initialization of the iteration, deciding whether another item needs to be visited, and, if so, picking the next item to visit. How this is to be done depends on the implementation of the collection. Iteration abstraction encapsulates the details of iteration, so that a user of the collection can iterate over it, without knowing all the implementation details.

See [4, §10.1.5].

7.15. Topic 15: Nested Classes

Nested class definitions allow classes to be coupled more tightly, without opening them up completely for all other classes.

See [4, §5.7.(2, 3)].

7.16. Topic 16: Design Patterns

Some design problems recur frequently, but in varying disguises and contexts. A design pattern provides a systematic solution schema for such a recurring problem. The solution schema needs to be instantiated for each specific occurrence of the problem to provide an actual solution.

In particular, the following patterns will be covered: Strategy Pattern, Observer Pattern, Decorator Pattern, Factory Pattern, Command Pattern, Adapter Pattern, and Composite Pattern.

See [1] and note [10]. For more information see [5, 6, 12].

7.17. Topic 17: Event-Driven Programs

Typically, a program has the initiative on the user interface: it offers the user a choice of what functionality to activate next. In Java GUI applications, this polling loop is known as the main event loop, and it is hidden from the programmer. The programmer constructs (the rest of) the application as a collection of event listeners, where each event listener handles a specific event generated by the user via (a component in) the GUI.

See [4, §6.1.3].

7.18. Topic 18: Elementary GUI Design

GUI programs involve a number of user interface components that are fairly independent of the programming language and operating system. In particular, the following components will be covered: text labels, text fields, text areas, buttons, check boxes, radio buttons, panels, menus, menu items, graphical output on a canvas, mouse control. Components can also act as containers of other components, giving rise to a hierarchical structure (cf. the Composite design pattern). This course is not about the ergonomics of GUI design, but a first encounter of how a GUI is organized in software.

See [4, §6.(1, 3–4, 6–8), 13.4.1].

7.19. Topic 19: Concurrency

A GUI program needs to be responsive to user actions on the GUI, for instance, allowing the user to cancel a long-running computation. Thus, long-running computations should not be executed in the GUI thread, because that would lock up the GUI. A simple solution is to execute the longrunning computation in a separate thread. Still, there may be a need for the GUI thread and the

computation thread to interact. For instance, the GUI might need to show a progress bar or some intermediate results of the computation. The designer should be aware of race conditions.

In Java, a `SwingWorker` can be used for this purpose.

This topic is only a first encounter with concurrency, and does not address all design issues.

See [4, §12.(1, 2.(1–3))].

8. Learning Technologies

- <http://ext.peach3.nl/> (learning management system on which you can access the course for HSE. This site contains downloads, schedule information, group assignments, etc.).
- Slides, assignment descriptions, various handouts.

9. Courseware Readings and Reference Materials

9.1. Basic Readers

[DE] David J. Eck. Introduction to Programming Using Java. Version 6.0, June 2011.

[EB] Eddie Burris. Programming in the Large with Design Patterns. Pretty Print Press, 2012.

[FF] Eric Freeman & Elisabeth Freeman. Head First Design Patterns. O'Reilly Media, 2004.

[JB] Joshua Bloch. Effective Java (Second Edition). Addison-Wesley, 2008.

[LG] Barbara Liskov and John Guttag. Program Development in Java. Addison-Wesley, 2001.

[JLS] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification (3rd Edition). Addison-Wesley, 2005.

[JD] Sun/Oracle. Java SE 7 Documentation. <http://download.oracle.com/javase/7/docs/>

9.2. Additional Readings and References

[1] Eddie Burris, Programming in the Large with Design Patterns. Pretty Print Press, 2012. programminglarge.com

[2] Coding Standard for 2IP15. A separate note for Programming Methods (2IP15). Programming Methods (2IP15) Course Outline

[3] Checklist for Larger Object-Oriented Programs. A separate note for Programming Methods (2IP15).

[4] David J. Eck. Introduction to Programming Using Java, Version 6.0, Hobart and William Smith Colleges, June 2011. math.hws.edu/javanotes.

[5] Eric Freeman, Elisabeth Freeman. Head First Design Patterns. O'Reilly Media, 2004.

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. Also known as the Gang-of-Four Book, or GOF Book.

[7] B. Liskov, J. Guttag. Program Development in Java. Addison-Wesley, 2001.

[8] Notation. A separate note for Programming Methods (2IP15).

[9] Specification. A separate note for Programming Methods (2IP15).

[10] Tom Verhoeff. From Callbacks to Design Pattern. Dept. Math. & CS, Eindhoven University of Technology, Oct. 2012.

[11] David Watt and Deryck Brown. Java Collections: An Introduction to Abstract Data Types, Data Structures, and Algorithms. J. Wiley, 2001.

[12] Wikipedia, Software Design Pattern, en.wikipedia.org/wiki/Software_design_pattern (accessed 30 Oct. 2012).

10. Technical Support

OHP for lectures and classes.

