Course Outline

The following topics will be covered in this course (not necessarily in this order).

Introduction 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.

See [13, Appendix B] for a concise overview of Java.

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. Imperative procedural Java programs include the above, with the addition of method definitions (see below under Procedural Abstraction). See [4, §2.(1, 2, 5), §3.(1, 3–6, 8), §4.7, §7.(1, 2, 5)].

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].

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.

Divide & Conquer in design involves three steps: (1) split a big design problem into smaller subproblems (decomposition), (2) solve the subproblems independently (possibly by doing Divide & Conquer again), and (3) integrate the subproblem solutions into a total solution (composition, integration). This gives rise to a hierarchic structure. When a subproblem has the same form as the original problem, a recursive solution can be considered.

In Java, solutions for subproblems can be wrapped up in various ways: as an expression, as a statement block, as a (possibly parameterized) method, as a class, or as a package (collection of related classes).

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, 3, §4.(1–4), §7.2.3]

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. This also relates to the Strategy design pattern.

See notes [8, 9], and [4, §4.6].

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.

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].

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. JaCoCo is a tool to determine unit test quality, by measuring code coverage.

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.

   Result: a class without implementation, that is, no data representation and empty method bodies.
4. Create a corresponding unit test class.
5. Implement rigorous test cases.
6. Choose a data representation and implement the class methods.
7. Test the implementation.

For larger classes, steps 5 through 7 are applied for (groups of) methods separately. See [10].

We use Mercurial for configuration management with TDD see [11].

Robustness, 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.(1–4)].

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.

Aliasing and sharing.

See [4, §2.3, §5.(1–4), §7.3].

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-

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.

See [4, §5.(5, 6)].

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, §5.7].

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.

Java has a special form of the **`for`**-statement: **`for`** `(T v : C) S`. It can be used with any collection `C` over a type `T`, where `C` implements the interface `Iterable<T>`. That is, `C` provides a method `iterator()` that

returns an `Iterator<T>` object having methods **`boolean`** `hasNext()` and `T next()`.

See [4, §10.1.5].

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

See [4, §5.8].

Parameterized Types, Generic Classes To improve the reuse of classes, they can be parameterized through (generic) type parameters.

See [4, , §10.(1.(4–7), 2–5)].

Design Principles and 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.

Good object-oriented designs, including those based on design patterns, adhere to a number of design principles, sometimes referred to as SOLID (SRP, OCP, LSP, ISP, DIP).

See [1] and note [12]. For more information see [5, 6, 14].

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)]

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, §3.9, §6.(1–3, 5–7), 13.4.1]

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 long-running 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 intro]

Design and Code Review The techniques discussed above are also the basis for evaluation of software, be it your own work or that of others.

References

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

[2] *Coding Standard for 2IPC0*. A separate note for Programming Methods (2IPC0).

[3] *Checklist for Larger Object-Oriented Programs*. A separate note for Programming Methods (2IPC0).

[4] David J. Eck. *Introduction to Programming Using Java*, Version 7.0, Hobart and William Smith Colleges, August 2014. `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 (2IPC0).

[9] *Specification*. A separate note for Programming Methods (2IPC0).

[10] *Test-Driven Development (TDD)*. A separate note for Programming Methods (2IPC0).

[11] *Configuration Management*. A separate note for Programming Methods (2IPC0).

[12] Tom Verhoeff. *From Callbacks to Design Pattern*. Dept. Math. & CS, Eindhoven University of Technology, Mar. 2015.

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

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

Author: Tom Verhoeff, Department of Mathematics & Computer Science, Eindhoven University of Technology