Software Construction 2011–2012

Introduction

This document (pdf) covers all relevant information concerning the course **Software Construction 2011–2012** in the Master Software Engineering, University of Amsterdam.

(NB: be sure to refresh this page; it may be cached by your browser.)

Schedule

Lectures and workshops will be from 9:00–11:00 on Mondays. Lectures will be given by Paul Klint, Jurgen Vinju, Tijs van der Storm and guest lectureres. Practical course will be on Mondays from 11:00 to 17:00 and Tuesdays the whole day. For details about rooms see rooster.uva.nl.

Primary contact for this course is Tijs van der Storm.

Week 2 - 4

- 09–1: Introduction (slides)
- 16–1: Lecture Grammars and Parsing
- 23–1: Lecture Domain-Specific Languages
  - Grading part I lab assignment

Week 5 - 8

- 30–1: Lecture Code Quality
  - Test I: technique
- 06–2: Lecture Debugging
- 13–2: Guest Lecture: Jeroen van den Bos, "Derric: a DSL for Digital Forensics"
  - Test II: philosophy
- 20–2: Guest Lecture: Eelco Visser, Linguistic Abstraction for the Web"
  - Grading part II lab assignment

Week 9

- 27–2: Paper writing time: deadline Sunday, March 4th.

How to pass this course

Required skills:

- Create good low level designs
- Produce clean, readable code
- Reflect upon and argue for/against software construction techniques, patterns, guidelines etc.
- Assess the quality of code.
- Select and apply state of the art software construction tools and frameworks.

Required knowledge:

- Understand basic principles of language implementation (parsing, AST, evaluation, generation)
- Understand the basic aspects of code quality
- Understand encapsulation and modular design

You will be graded on the following course assignments:

- Part 1 of the practical course: this grade will be an indication that you can use to improve your code for Part 2.
- Part 2 of the practical course.
- Research paper.

These components are described in more detail below. The final grade is the average of the research paper grade and the final grade of the practical course (i.e. the grade for Part 2).

Preconditions for getting a grade:

- Be present at all lectures.
- Pass the two reading tests.

Literature Research Paper

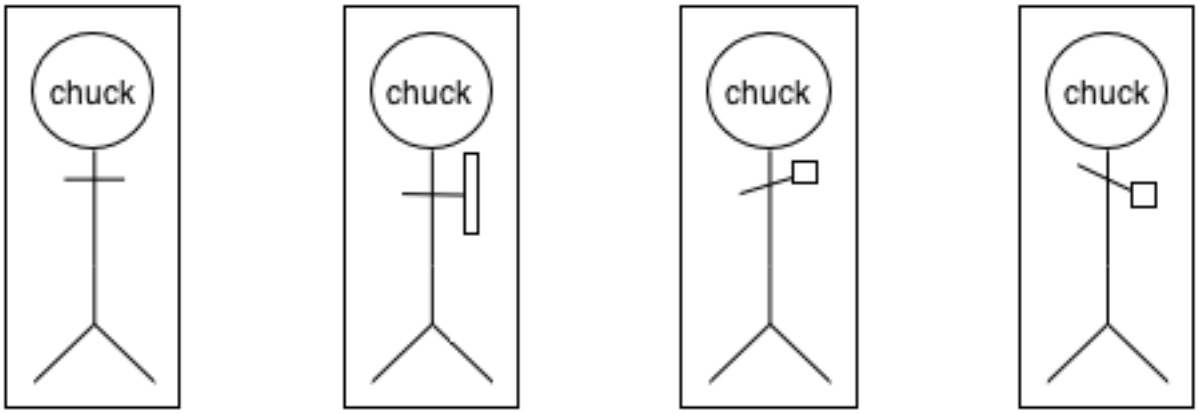
To pass this course you are required to write a paper. It can be seen as an exercise in argumentation. This paper should be between 3–5 pages in size. For the topic, select one of the topics in the appendix at the end of this document. Moreover, select a *position* on the topic, based on the papers listed for each topic. For instance, in the topic "Design by Contract", there are two papers advocating the use of design by contract. You could take that position, but you could also take a position against design by contract. Either way, you are required to find a minimum of *two* academic papers that support your position. In the paper you should defend your position using the literature of the appendix and the additional 2 papers.

Lab assignment: Super Awesome Fighters

The goal of the lab assignment is to implement a domain-specific language (DSL) for specifying Super Awesome Fighter (SAF) bots. An example of such a specification looks like this:

```
chicken
{
  kickReach  = 9
  punchReach = 1
  kickPower  = 2
  punchPower = 2
  far [run_towards kick_low]
  near [run_away kick_low]
  near [crouch punch_low]
}
```

A simplistic graphical rendering of such fighter bots could look like this:



The different figures represent different attacks (kicking, punching and blocking).

The required set of conditions, moves, attacks and strengths is as follows:

- Conditions: stronger, weaker, much_stronger, much_weaker, even, near, far, always.
- Moves: jump, crouch, stand, run_towards, run_away, walk_towards, walk_away.
- Attacks: punch_low, punch_high, kick_low, kick_high, block_low, block_high
- Strengths: punchReach, kickReach, kickPower, punchPower.

This is the minimum; you are free to extend this set if you want.

The specification includes strengths (like kickReach = 9) and (conditional) move/attack pairs. For instance, chicken should "run_towards" (his opponent) and "kick_low" if he is "far". Conditions, like far, can be composed with "and" and "or". An implementation of this DSL will simulate two bots fighting each other.

The SAF language has no formal reference. It is however used in the following paper:

> James R. Williams, Simon M. Poulding, Louis M. Rose, Richard F. Paige, Fiona A. C. Polack. Identifying Desirable Game Character Behaviours through the Application of Evolutionary Algorithms to Model-Driven Engineering Metamodels. In: Proceedings of the 3rd International Symposium on Search Based Software Engineering (SBSE'11), Springer LNCS 6956, 2011, pp. 112–126.

You can find some examples there.

James R. Williams kindly provided me with the following notes on the semantics:

- Fighter properties are from 1 to 10 inclusive.
- If a property is not specified, it defaults to 5.
- There are three properties that are derived from the configurable properties: weight, height and speed. The can be computed as follows:

```
weight = (punchPower + kickPower) / 2
height = (punchReach + kickReach) / 2
speed = | 0.5*(height-weight) |
```

- The height and weight properties are only used to calculate the speed, and do not affect anything else (such as the punch/kick/block height).
- Certain moves take longer than one time step to perform (as determined by the player's speed). A player cannot make a new move until the previous move has completed.
- The fight action and move action occur at the same time - allowing, for example, the player to both run and kick at the same time.
- The set of rules that are applicable at each time step is calculated by evaluating the condition of each rule.
- A rule is selected randomly from the set of rules that are applicable at that instant. To cover the case that there are no applicable rules, there must be an "always" rule.
- If the selected rule has a "choose" construct, then the option is, again, chosen at random.

You will have to fill in any aspect that is unspecified.

There are two ways to fulfill the assignment based on the technology you choose to use. The front-end part of both variants differs, the back-end is the (mostly) the same.

Rascal front-end

If you choose to use Rascal, a DSL for source code analysis and transformation, the deliverables of the first part of the assignment are as follows:

- A correct Rascal grammar for the syntax of SAF

- A meta-model/abstract syntax ADT for representing SAF models
- Basic IDE features for SAF files (including syntax highlighting and outlining)
- A well-formedness/consistency checker hooked up to the IDE. This means that, if there are errors, they are shown in SAF editors.
- A source-to-source transformation (or "model-transformation") that makes implicit defaults explicit.
- A generator that converts SAF models to an XML document. This functionality should be made available in the IDE.

Many of you have used Rascal during Software Evolution. Furthermore, since Rascal is a programming language specifically designed for implementing DSLs, the assignment is slightly more complex than the other variant. It also requires ample knowledge of the Rascal language and APIs.

Java front-end

If you choose Java as the primary implementation language, it is unrealistic to require the more advanced IDE features required in the Rascal variant. In this case, the assignment is as follows:

- Write a parser for the SAF language, using a carefully selected parser generator for use in Java. Candidates for this choice are: Jacc, JavaCup, Rats!, ANTLR, JavaCC, SableCC, but there are others. You're expected to be able to motivate your choice.
- The parser should produce an abstract syntax tree (AST); this is a tree of objects representing the source program. The AST is described by a *class hierarchy*. (NB: this class hierarchy is required! You could use this requirement in your selection of parser generator).
- A well-formedness/consistency checker implemented on top of your AST classes.

Although the SAF language is really simple, do not underestimate the complexity of using a parser generator effectively.

Back-end

For both the Rascal and Java variant, the back-end of the SAF implementation is more or less the same:

- A semantics/AI for all expressions (conditions, fights, and moves) in the SAF language. This component produces the behavior of a SAF fighter model. It encodes how two fighters interact.
- A graphical simulator to animate matches between two fighters.

NB: these are *two* things. Modularize accordingly.

In the Rascal variant, the SAF models are represented by the XML document, generated from Rascal. This entails an additional requirement in that case: the selection of a suitable XML API. Again you should be able to motivate your choice. Possible candidates include: JDOM, DOM, SAX, XOM, JAXB etc. In the Java case the SAF models are instances of the AST class hierarchy, so the AI can be implemented directly on top of those.

BONUS: in both variants, there's an opportunity to obtain a bonus point if you write a SAF *compiler*. This compiler should generate Java code that interfaces with the graphical simulation component that you already have developed. If you use Rascal, the code generator should be implemented in Rascal using string templates. In Java, you could use a third-party templating solution.

Further requirements

- Your parser should correctly parse sample fighters that we will provide in due course. Think about keyword reservation, longest-match for identifiers, priority and associativity of binary and/or unary operators.
- For static consistency checking, it is required to implement at least *three* checks.
- For the game AI, all constructs in SAF should have a user visible effect. (e.g., you can not give any two different fighters the behavior to run away from a fight).

You must be able to motivate your semantic rules (both in the case of static checking and game AI). Finally, you *must*, adhere to the syntax of the examples. This will allow everyone to share fighters for experimentation.

Honor's Track

The honor's track is meant for excellent students. The students on the honor's track will collaborate to develop a more advanced DSL in Rascal. This DSL is the current assignment of the Language Workbench Competition. This competition aims to compare the strengths and weaknesses of different workbenches.

In 2011, the Rascal team submitted a solution for a different assignment, the description of which can be found in the following technical report:

Tijs van der Storm, The Rascal Language Workbench, CWI Technical Report SEN–1111, 2011.

This year the assignment is a DSL for Piping and Instrumentation. The assignment can be found in the document LWC'12 Assignment.

The components of interest are:

- Syntax of piping models and a controller language
- Static checking of both languages
- Visualization of a piping model
- Simulation of piping models (i.e. animation)
- Code generation

The honor's team will work in close collaboration with CWI. If the project is successful it will be presented on the LWC meeting at the Code Generation conference in Cambridge, UK. Note that, the code for the assignment will be published and used to show off the strengths of Rascal. It is therefore very important that the code is typical and of the highest quality.

What we look for when grading your code

We take the principles laid down in *Code Complete* as guidelines when grading your solutions. More specifically, the following aspects of quality code will be our focus:

- Functionality (e.g., are the requirements implemented)
- Tests (e.g., presence of meaningful unit tests)
- Simplicity (absence of code bloat and complexity; YAGNI)
- Modularity (e.g., encapsulation, class dependencies, package structure)
- Layout and style (indentation, comments, naming of variables etc.)
- Sensible use of design patterns (e.g., Visitor)

More concretely, we ask you to take the following list of advice into consideration.

- Code quality is of the utmost importance in this course. You will write clean, consistently formatted, concise code. Your naming and indentation convention will be consistent.

- You show that you master the concepts of encapsulation, modularity and separation of concerns. This should be visible from the code. The structure of the code should show the design.
- Method and functions should realize a single piece of functionality. You adhere to the Don't Repeat Yourself (DRY) principle.
- You will select tools and libraries wisely. You can argue why you chose to use a particular artifact.
- You know your (standard) libraries and APIs. Do not reimplement (simple) functions that can be expected to be in a (standard) library. Especially, do not claim that your version is faster, because: it is irrelevant, and, you're probably wrong. Make the trade-off for reusing a library: do you really need a heavy dependency, for some simple functionality?
- Test your code using unit tests if this is meaningful. Do not write tests, because your are somehow supposed to. Do not write your own testing framework; use appropriate libraries and/or language features of the platform (e.g. JUnit on Java). Separate test code from main code.
- Use asserts in the correct way. Asserts are used to document and check assumptions. They are *not* used for input validation or error handling.
- Use exception handling wisely. Do not implement your Exception class in a situation where a standard library exception makes perfect sense. Handle exceptions sanely, if possible. Empty catch-blocks are unacceptable 99.9% of the time.
- Non-constant static variables should be avoided at all cost.
- If you are forced to need instanceof a lot, you probably have a flaw in your design.
- You are expected not to indulge in elaborate gold plating. For instance, fancy graphics/user interfaces are not important. YAGNI: You Ain't Gonna Need It. Focus on the simplest thing that could possibly work, first.
- Do not optimize your code unless you can argue there is a real problem (proven by profiling). Simplicity of the code has priority.

  We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. —Donald Knuth

- You are not supposed to show off how smart you are.

  Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. —Brian Kernighan

- You are expected to write comments, only if you need to explain a complicated algorithm or motivate a particular piece of code. Do not engage in obligatory comments. Javadoc (or similar) is ok, but think about the purpose of Javadoc first.
- It is unacceptable that there are remnants of dead code, commented out sections, or debugging print statements etc. in the code that you will present for grading.
- You will only present *working code* for grading. Note: working code implies your project compiles without errors. Additionally, you should use the IDE in the correct way, setup dependencies correctly, provide build-scripts if necessary.

Please take this advice to heart. It will influence your grade.

Note that, although Rascal is not an object-oriented programming language, this does not imply that you cannot modularize and encapsulate. On the other hand, your are forbidden to "write Java/C/PHP/C# etc." in Rascal. If you go with Rascal, be sure that you know how to use comprehensions, visit, pattern matching etc.

Administrativia

Each participant will get access to a Google Code Subversion repository, setup for this course. The URL of the Google Code project is:

> http://code.google.com/p/sea-of-saf/

Please sign up for a Google account if you haven't done that already, and notify Tijs van der Storm to get a subdirectory in the project repository.

IMPORTANT: You are required to complete the lab assignment *individually*. We will use clone detection tools to detect plagiarism.

IMPORTANT: You are **required** to use this Subversion repository to facilitate easy access. You should also commit **regularly**: NO huge final commit before the deadline.

Deadlines

- Part 1 (front-end): 23rd of January, 2012
- Part 2 (back-end): 20th of February, 2012

Theory Tests

There will be two tests during the course. They will consist of open and multiple-choice questions based on the lectures and the syllabus below. This is *required* reading.

Technique

- Bertrand Meyer, *Applying "Design by Contract"*, 1992, Meyer92.
- Karl J. Lieberherr, Ian M. Holland, *Assuring Good Style for Object-Oriented Programs*, 1989, LieberherrHolland89.
- Robert C. Martin, *The Open-Closed Principle*, 1996, Martin96.
- Ralph Johnson, Brian Foote, *Designing reusable classes*, 1988, Johnson-Foote88.
- Marjan Mernik et al. *When and How to Develop Domain Specific Languages*, 2005, MernikEtAl05.

Philosophy

- D. L. Parnas, *On the criteria to be used in decomposing systems into modules* , 1972, Parnas72
- William R. Cook, *On understanding data abstraction, revisited*, 2009, Cook09.
- Herbert Simon, *The Architecture of Complexity*, Simon62
- Carliss Y. Baldwin, Kim B. Clark, *Modularity in the Design of Complex Engineering Systems*, BaldwinClark06
- Horst Rittel, Melvin Webber, *Dilemmas in a General Theory of Planning*, RittelWebber84.

Appendix: Paper Topics

Structured programming with or without gotos?

- E.W. Dijkstra *Goto considered harmful*, 1968, Dijkstra68;
- D. Knuth, *Structured Programming with go to statements*, 1974, Knuth74.

State Transactional Memory

- Simon Peyton Jones, *Beautiful Concurrency*, 2007, PeytonJones07.
- Calin Cascaval et al. *Software Transactional Memory: Why is it Only a Research Toy?*, 2008, CascavalEtAl08.
- Bryan Cantrill and Jeff Bonwick, *Real-world Concurrency*, 2008, Cantrill-Bonwick08.

Internal vs external DSLs

- Marjan Mernik et al. *When and How to Develop Domain Specific Languages*, 2005, MernikEtAl05.
- Martin Fowler, *Implementing an Internal DSL*, 2007 Fowler07.

Aspect-Oriented Programming

- Gregor Kiczales et al. *Aspect-Oriented Programming*, 1997, KiczalesEtAl97.
- Robert E. Filman, Daniel P. Friedman, *Aspect-Oriented Programming is Quantification and Obliviousness*, 2000, FilmanFriedman00.

Literate Programming in the 21st Century

- Bentley, Knuth and McIllroy, *A Literate Program*, 1986, BentleyEtAl86.
- Knuth, *Literate Programming*, 1984, Knuth84.

Prototype-based vs class-based object-orientation

- James Noble, Brian Foote, *Attack of the Clones*, 2002, NobleFoote02.
- Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, 1986, Lieberman86.

Design by Contract

- Bertrand Meyer, *Applying "Design by Contract"*, 1992, Meyer92.
- Jean-Marc Jézéquel, Bertrand Meyer, *Design by Constract: The Lessons of Ariane*, 1997, JezequelMeyer97.

Fluent or Law-of-Demeter?

- Martin Fowler, *FluentInterface*, 2005, Fowler05.
- Karl J. Lieberherr, Ian M. Holland, *Assuring Good Style for Object-Oriented Programs*, 1989, LieberherrHolland89.

Maximizing reuse minimizes use

- T.J. Biggerstaff, *The Library Scaling Problem and the Limits of Concrete Component Reuse*, 1994, Biggerstaff94.
- James M. Neighbors, *Draco: A Method for Engineering Reusable Software Systems*, 1989, Neighbors89.

Design Patterns are Code Smells

- Jan Hannemann, Gregor Kiczales, *Design Pattern Implementation in Java and AspectJ*, 2002, HannemannKiczales02.
- Peter Norvig, *Design Patterns in Dynamic Languages*, 1996, Norvig96.