Course "Software Language Engineering"

This is it!
This is the course!
(for language engineers)

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team

What’s “software language engineering” anyway?
Software language engineering is the application of systematic, disciplined, and quantifiable approaches to the development (design, implementation, testing, deployment), use, and maintenance (evolution, recovery, and retirement) of these languages. Of special interest are (1) formal descriptions of languages that are used to design or generate language-based tools and (2) methods and tools for managing such descriptions, including modularization, refactoring, refinement, composition, versioning, co-evolution, recovery, and analysis.
Software practitioners are rapidly discovering the immense value of Domain-Specific Languages (DSLs) in solving problems within clearly definable problem domains. Developers are applying DSLs to improve productivity and quality in a wide range of areas, such as finance, combat simulation, macro scripting, image generation, and more. But until now, there have been few practical resources that explain how DSLs work and how to construct them for optimal use. Software Language Engineering fills that need. Written by expert DSL consultant Anneke Kleppe, this is the first comprehensive guide to successful DSL design. Kleppe systematically introduces and explains every ingredient of an effective language specification, including its description of concepts, how those concepts are denoted, and what those concepts mean in relation to the problem domain. Kleppe carefully illuminates good design strategy, showing how to maximize the flexibility of the languages you create. She also demonstrates powerful techniques for creating new DSLs that cooperate well with general-purpose languages and leverage their power.
Software language engineering from the point of view of Grammarware

"Grammarware hacking"

Grammar knowledge

Manual coding

Parser specification

Parser generation

Parser

Codebase as input

"Grammarware engineering"

Semi-automatic recovery

Semi-automatic customisation

Incremental improvement by transformations

Grammar

Parser specification

Parser generation

Parser

Test-data set as input

Test-data generation


What’s a “software language” anyway?
The term "software language" refers to artificial languages used in software development including general-purpose programming languages, domain-specific languages, modeling and meta-modeling languages, data models, and ontologies. Examples include general purpose modeling languages such as UML, but also domain-specific modeling languages for business process modeling, such as BPMN, or embedded systems, such as Simulink or Modelica, and specialized XML-based and OWL-based languages and vocabularies. The term "software language" also comprises APIs and collections of design patterns that define a language implicitly.
Java?

This is a software language in that it is used in software development for the implementation of software systems.
This is a software language in that it is used in software development for models of software systems to be gradually transformed into actual code of software systems.
This is a software language in that it is used in software development as modeling software systems to help understanding between system owner and developer.
UML on the whiteboard?

Is this a software language?
These are software languages because they support the development of web-based software systems as they cover some aspect of the systems’ user interfaces.
This is, “probably”, a software language because it is used to author content for a web-enabled knowledge representation et al. system.
This is not a software language because it is, in fact, a natural language that has existed and continues to exist independently of computer science.
Such a language may be a software language, e.g.:

http://www.omg.org/spec/SBVR/1.0/

A barred driver must not be a driver of a rental.

It is prohibited that a barred driver be a driver of a rental.

It is obligatory that no barred driver is a driver of a rental.

[http://www.w3.org/2004/12/rules(ws)/paper/67/]
Motivation

(Who needs SLE?)
SLE – what for?

- Compiler Construction ("classic")
- Domain-specific languages ("hype")
- Model-driven engineering (?)
- Program generation
- Software reverse engineering
- Software re-engineering
- Technical space travel
- ...

SLE – what for?

- Compiler Construction ("classic")
- Domain-specific languages ("hype")
- Model-driven engineering (?)
- Program generation
- Software reverse engineering
- Software re-engineering
- Technical space travel

...
Compiler construction

![Diagram of compiler construction process:

- Program source
- Scanner (tokenizer) → Tokens
- Parser → Syntactic Structure
- Semantic Routines
- IR: Intermediate Representation (1)
- Analysis/Transformations/optimizations
- IR: Intermediate Representation (2)
- Code Generator
- Assembly code]

Resource: https://engineering.purdue.edu/~milind/ece573/2011spring/ (Milind Kulkarni)
“Wishful” reading
SLE – what for?

- Compiler Construction ("classic")
- **Domain-specific languages** ("hype")
- Model-driven engineering (?)
- Program generation
- Software reverse engineering
- Software re-engineering
- Technical space travel

...
A DSL is a focussed, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are natural/suitable for the stakeholders who specify that particular concern.

[https://github.com/slecourse/slecourse/wiki/dslDesign]
A DSL for state machines

How to support such visual syntax?
How to textually represent state machines?
How to interpret textual/visual representation?
A DSL for state machines

**initial state** Locked {
  Ticket/Collect $\rightarrow$ Unlocked;
  Pass/Alarm $\rightarrow$ Locked;
}

**state** Unlocked {
  Ticket/Eject $\rightarrow$ Unlocked;
  Pass $\rightarrow$ Locked;
}

We need code generation.
The LINQ DSL for data queries

Program that uses query expression [C#]

```csharp
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        int[] array = { 1, 2, 3, 6, 7, 8 };
        // Query expression.
        var elements = from element in array
                        orderby element descending
                        where element > 2
                        select element;
        // Enumerate.
        foreach (var element in elements)
        {
            Console.Write(element);
            Console.Write(' ');
        }
        Console.WriteLine();
    }
}
```

Output

8 7 6 3

How to embed the DSL into the host language?
General Purpose

C

Components

State Machines

Sensor Access

Domain Specific

LEGO Robot Control

[https://github.com/slecourse/slecourse/wiki/dslDesign]
Big Language

with many first class concepts!

[https://github.com/slecours/slecours/wiki/dslDesign]

Small Language

with a few, orthogonal and powerful concepts
"Wishful" reading
SLE – what for?

- Compiler Construction (“classic”)
- Domain-specific languages (“hype”)
- Model-driven engineering (?)
- Program generation
- Software reverse engineering
- Software re-engineering
- Technical space travel
- ...

Model-driven engineering

Model-driven engineering (MDE) is a software development methodology which focuses on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular application domain), rather than on the computing (or algorithmic) concepts. The MDE approach is meant to increase productivity by maximizing compatibility between systems (via reuse of standardized models), simplifying the process of design (via models of recurring design patterns in the application domain), and promoting communication between individuals and teams working on the system (via a standardization of the terminology and the best practices used in the application domain).

MDE example

public class HolidayRes : ServicedComponent {
    public HolidayRes() {} 
    public Offer[] findOffers (Criteria criteria) {} 
    public Confirmation book (Offer offer) {} 
}
SLE – what for?

- Compiler Construction ("classic")
- Domain-specific languages ("hype")
- Model-driven engineering (?)
- **Program generation**
- Software reverse engineering
- Software re-engineering
- Technical space travel

...
Program generation for ANTLR-based parsers

```antlr
company : 'company' STRING '{' department* '}' EOF;

department : 'department' STRING '{'
  ('manager' employee)
  ('employee' employee)*
  department* '}'
;

employee : STRING '{'
  'address' STRING
  'salary' FLOAT
 '}';
```

```java
package org.softlang.parser;

import org.antlr.runtime.*;
import java.util.Stack;
import java.util.List;
import java.util.ArrayList;

public class CompanyParser extends Parser {
  public static final String[] tokenNames = new String[] {
    "<invalid>", "<EOF>", "<DOWN>", "<UP>", STRING, FLOAT, "WS", "'comp"
  };
  public static final int WS=6;
  public static final int T_12=12;
  public static final int T_11=11;
  public static final int T_14=14;
  public static final int T_13=13;
  public static final int T_10=10;
  public static final int T_9=9;
  public static final int T_8=8;
  public static final int T_7=7;
  public static final int STRING=4;

  // delegates
  // delegators

  public CompanyParser(TokenStream input) {
    this(input, new RecognizerSharedState());
  }
  public CompanyParser(TokenStream input, RecognizerSharedState state) {
    super(input, state);
  }

  public String[] getTokenNames() { return CompanyParser.tokenNames; }
  public String getGrammarFileName() { return "Company.g"; }
```
Program generation is an essential element of language implementation, but it also used in other contexts.
Program generation for remote-method invocation

Remote Object Stub: serves as Remote Reference

Same interface as remote object

Serialized method name, parameters and return value

The purple functionality requires program generation (or reflection).

SLE – what for?

- Compiler Construction ("classic")
- Domain-specific languages ("hype")
- Model-driven engineering (?)
- Program generation
- **Software reverse engineering**
- Software re-engineering
- Technical space travel
- ...

“Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction.”

Chikofsky, E. J.; Cross, J. H. (January 1990). "Reverse engineering and design recovery: A taxonomy". *IEEE Software* 7: 13–17. doi: [10.1109/52.43044](http://dx.doi.org/10.1109/52.43044).
Package view of a GMF project for a library
(Joined work with Davide Di Ruscio and Alfonso Pierantonio)

Gray means generated and used by custom code.
Light gray means generated code not used by custom code.
Red means generated code that was amended.
Green means custom code referring to generated code.
Yellow means custom code not referring to generated code.

Some additional languages:
- the scheme for extracted data
- the visual language for graphs
Dependencies in a GMF project for a library
(Joined work with Davide Di Ruscio and Alfonso Pierantonio)
SLE – what for?

- Compiler Construction ("classic")
- Domain-specific languages ("hype")
- Model-driven engineering (?)
- Program generation
- Software reverse engineering
- **Software re-engineering**
- Technical space travel

...
The reengineering of software was described by Chikofsky and Cross in their 1990 paper \cite{RL: as mentioned before}, as "The examination and alteration of a system to reconstitute it in a new form". Less formally, reengineering is the modification of a software system that takes place after it has been reverse engineered, generally to add new functionality, or to correct errors.
Method extraction in Java

```java
void printOwning(double amount) {
    printBanner();
    //print details
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}
```

rollers
```java
void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}
void printOwning(double amount) {
    printBanner();
    printDetails(amount);
}
```
## Other kinds of extraction

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Focus</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>OO programming</td>
<td>statements</td>
<td>method</td>
</tr>
<tr>
<td>OO programming</td>
<td>features</td>
<td>class</td>
</tr>
<tr>
<td>Functional programming</td>
<td>expression</td>
<td>function</td>
</tr>
<tr>
<td>Functional programming</td>
<td>type expression</td>
<td>datatype</td>
</tr>
<tr>
<td>Functional programming</td>
<td>functions</td>
<td>type class</td>
</tr>
<tr>
<td>Functional programming</td>
<td>literal</td>
<td>predicate</td>
</tr>
<tr>
<td>Logic programming</td>
<td>EBNF phrase</td>
<td>nonterminal</td>
</tr>
<tr>
<td>Syntax definition</td>
<td>code fragment</td>
<td>macro</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>content particle</td>
<td>element type</td>
</tr>
<tr>
<td>Document processing</td>
<td>sentences</td>
<td>paragraph</td>
</tr>
<tr>
<td>Cobol programming</td>
<td>sentences</td>
<td>subprogram</td>
</tr>
<tr>
<td>Cobol programming</td>
<td>data description entries</td>
<td>copy book</td>
</tr>
<tr>
<td>Cobol programming</td>
<td>data description entries</td>
<td>group field</td>
</tr>
</tbody>
</table>
Transformations (refactorings) other than extraction

- Inlining (inverse of extraction)
- Introduction / elimination
- Fold / unfold (similar extract / inline)
- Pull up / push down
- Add parameter
- Reorder
- ...

Extraction of a Haskell datatype

Input program with focus

\[
\textbf{data \ Prog} \quad = \quad \texttt{Prog ProgName } [\textbf{Dec}] [\textbf{Stat}]
\]
\[
\textbf{data \ Dec} \quad = \quad \texttt{VDec Id Type} \mid \ldots
\]
\[
\textbf{data \ Stat} \quad = \quad \texttt{Assign Id Expr} \mid \texttt{If Expr Stat Stat} \mid \ldots
\]

Output program after extraction and integration

\[
\textbf{data \ Prog} \quad = \quad \texttt{Prog ProgName Block}
\]
\[
\textbf{data \ Block} \quad = \quad \texttt{Block [Dec] [Stat]}
\]
\[
\textbf{data \ Dec} \quad = \quad \ldots
\]
\[
\textbf{data \ Stat} \quad = \quad \ldots \mid \texttt{BlockStat Block}
\]
\ldots
A generic view on extraction

List of abstractions

1. Introduction

2. Replacement

Focused fragment
Generic steps of extraction

1. Lookup focused fragment.
2. Determine free names in focused fragment.
3. Enforce language-dependent check on focus.
5. Find host for new abstraction.
6. Introduce abstraction.
7. Construct application.
8. Replace focus by application.
Language issues in refactoring (re-engineering)

Transformation requires certain language services.

Transformation principles are to be mapped to languages.

Sets of transformation operators may be viewed as DSL.

...
Example: coupled transformation

There is an abundance of scenarios in software development which involve software transformations that affect multiple artifacts essentially in a coupled manner. That is, if one of the artifacts of the scenario is changed, then one or more of the remaining artifacts have to be co-changed. Here are some scenarios that involve such changes and co-changes:

• XML data binding
• Object-relational mapping
• Updateable views in databases
• Data integration and data exchange
• Multi-view specification and consistency
• Instance mapping in reply to schema evolution
• Program transformation in reply to schema evolution
• Model transformation in reply to metamodel evolution

In the scope of coupled software transformations, arbitrary software artifacts can be considered: programs, specifications, data models, data, metamodels, object graphs, transformations (!), and others. Further, arbitrary relationships between the artifacts can be considered: conformance (e.g., between a model and a metamodel), consistency (e.g., between two models at the same level or different levels of abstraction), bidirectional transformations (BX; e.g., between primary database table and an updatable view), and others. Coupled software transformations define how changes of given artifacts are completed into co-changes of the remaining artifacts so that the intended relationships between all the artifacts are maintained.
Coupled transformation of XMLware

The primary transformation is defined at the XML schema level, while the transformations at the document-processing level (e.g., XSLT) and the XML-stream level are supposed to be implied.

Some similar work has been reported on XML grammars [Lämmel and Lohmann 2001; IBM Research 2002]. More generally, we view pairs of transformations on schema and data as an important instance of the notion of "coupled transformation" [Lämmel 2004a].

The derivation of a program transformation from a schema transformation is weakly understood both in the XML context and the database context. However, object-oriented program refactoring [Griswold and Notkin 1993; Opdyke 1992] instantiates this sort of coupling, where class structures can be refactored and all dependent method implementations are "automatically" updated. Clearly, evolutionary transformations can go beyond mere refactoring. In [Kort and Lämmel 2003a], we consider coupled transformations for types and functions in a functional program, while we even go beyond refactoring. Some forms of model transformations [Sendall and Kozaczynski 2003] (in the sense of the emerging field of model-driven development) might be applicable in the grammarware context.

Evolution comprises refactoring, enhancement, as well as clean-up. In the broader sense, evolution also comprises re-targeting grammarware from one technology to another. Basic grammar transformations for refactoring, enhancement, and clean-up were developed in [Lämmel 2001a]. Evolutionary transformations of software have generally not yet received much attention, except for the refactoring mode of evolution. The situation is not different for grammarware, but some initial ideas are summarised in [Lämmel 1999b; 2004b], where rule-based programs are transformed in a number of ways, including some grammar-biased modifications, some of them going beyond refactoring.

6.5 Principle: reverse-engineer legacy grammarware

We can not assume that suitable base-line grammars are readily available for all legacy grammarware. However, it is fair to assume that there is some encoded grammar knowledge available, from which base-line grammars can be recovered by means of reverse engineering. The grammar knowledge can reside in data, e.g., one can infer an XML schema.
SLE – what for?

- Compiler Construction ("classic")
- Domain-specific languages ("hype")
- Model-driven engineering (?)
- Program generation
- Software reverse engineering
- Software re-engineering
- Technical space travel
- ...

Technical space travel

Diagram:
- Objects
- Relations
- XML

Arrows connect Objects to Relations, and Relations to XML, indicating a flow or relationship between the concepts.
O/X mapping (XML data binding)

```xml
<element name="point">
  <complexType>
    <sequence>
      <element name="x" type="xs:int"/>
      <element name="y" type="xs:int"/>
    </sequence>
  </complexType>
</element>
```

Maps to

```java
public class Point {
  public int x;
  public int y;
}
```
XML data binding with JAXB

Resource: http://www.oracle.com/technetwork/articles/javase/index-140168.html (SUN, Oracle)
Generator functionality for XML data binding
O/R mapping with EntityFramework


A fragment of a BNF grammar that defines the concrete syntax of a simple language.

```plaintext
[axiom]  program  ::=  declarations statements
[decs]   declarations ::= declaration “;” declarations
[nodec]  declarations ::= ε
[dec]    declaration  ::=  id “;” type
[concat] statements  ::=  statement “;” statements
[skip]   statements  ::=  ε
[assign] statement  ::=  id “;=” expression
[var]   expression  ::=  id
...```

A fragment of a DTD for the XML representation of the organisational structure in a company.

```xml
<!DOCTYPE company [
<!ELEMENT company (dept*)>
<!ELEMENT dept (name, manager, unit*)>
<!ATTLIST dept dept_num ID #REQUIRED >
<!ELEMENT unit (employee | dept) >
<!ATTLIST employee busunit IDREF #IMPLIED >
<!ELEMENT employee (person, salary) >
<!ELEMENT person (name, address) > ... ]>
```

Some algebraic data types in Haskell notation for event traces of the execution of C programs.

```haskell
data ExecProg = ExecProg [Either ExecStmt EvalExpr]
data ExecStmt = ExecStmt [Either ExecStmt EvalExpr]
data EvalExpr = EvalCall FuncCall
  | EvalAssign Assign
  | EvalOthers [EvalExpr]
data FuncCall = Call [EvalExpr] [ExecStmt]
data Assign = Assign [EvalExpr] Desti
data Desti = ...
```
Modeling-centric view on technological spaces
Various spaces

Dragan Djuric, Dragan Gašević, and Vladan Devedžić:
The Tao of Modeling Spaces, JOT 2006.

Space travel

Educational objectives
(What to learn here?)
SLE skills

- How to parse textual syntax?
- (How to parse visual syntax?)
- How to abstract from concrete syntax?
- How to check context conditions?
- How to perform semantic analysis?
- How to generate code?
- How to leverage intermediate languages?
More SLE skills

- How to design languages?
- How to implement languages?
- How to embed languages?
- How to transform source code?
- How to generate source code?
- How to design transformation suites?
- How to map between abstraction levels?
Yet more SLE skills

- How to design languageware?
- How to test it?
- How to assess it?
- How to evolve it?

Let’s see how much ground we can cover. This is just one course. Also, SLE is an emerging subject.
Skills – overall?

Make the transition from a software engineer, overall, to a software language engineer, to do the right thing, whenever languages are involved. (They are involved all the time.)
Grammarware lifecycle (as opposed to the software lifecycle)

The discussed principles can be integrated in a grammarware life cycle; see Fig. 6. By having a proper grammarware life-cycle we can invigorate the normal software life-cycle. Most notably, the distinction of base-line grammars vs. grammar use cases allows us to apply evolutionary transformations to the former such that the adaptations of the latter are mostly implied. That is, grammar use cases are supposed to evolve with base-line grammars. There are clearly evolution scenarios that are inherently technology- and use-case-specific, in which case evolutionary transformations must be carried out on grammar use cases.

To align the grammarware life cycle with the normal software life cycle, we will briefly go through Fig. 6. We will focus on forward engineering — knowing what we will neglect some trips through the figure. There are the following phases:

- Provision of base-line grammars.
- Customisation to derive grammar use cases.
- Implementation to obtain actual grammar-dependent software.
- (Potentially grammar-based) testing of the grammar-dependent software.

Here is one scenario for forward engineering from Fig. 6: going from a base-line grammar to an object-oriented visitor framework through a customised class hierarchy. The derivation of the use case requires a class dictionary. (Hence, either the base-line grammar must be Grammarware lifecycle (as opposed to the software lifecycle)
Perceived view "Lines of code"  

Proposed view "Impact ratio"

<table>
<thead>
<tr>
<th>All software assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammars in compiler front-ends</td>
</tr>
<tr>
<td>Grammars in a broad sense</td>
</tr>
<tr>
<td>Ingrained grammar dependencies</td>
</tr>
</tbody>
</table>

Fig. 4. In need of a paradigm shift: On the left-hand side, we only care about obvious grammar for ms, namely the ratio of "all software" to "grammars in compiler front-ends". On the right-hand side, we admit two important facts: (i) there are many grammars other than those in compiler front-ends; (ii) ingrained grammar dependencies have a deep impact on most software.

5. PROMISES OF GRAMMARWARE ENGINEERING

At this point, the reader might face the following question: Somehow we managed to deal with all these kinds of grammarware decades. So what? That is, what are the potential benefits for IT?

The overall promise of grammarware engineering is that it leads to improved quality of grammarware and to increased productivity of grammarware development. These promises should provide a good incentive since grammars permeate software systems and software development. Of course, it is difficult to justify such general claims at this time. To provide some concrete data, we will report on two showcases (or even success stories). Afterwards, we will identify more detailed promises on the basis of these showcases, but we will also refer to further scattered experiences with engineering aspects of grammarware.

5.1 Showcase: grammar recovery

This showcase is discussed in detail in [Lämmel and Verhoef 2001b; Lämmel 2005]. Using elements of the emerging engineering discipline for grammarware, we were able to rapidly recover a relatively correct and complete syntax definition of VS Cobol II. The starting point for this recovery project was IBM's industrial standard for VS Cobol II [IBM Corporation 1993]. The syntax diagrams had to be extracted from the semi-formal document, and about 400 transformations were applied to the raw syntax in order to add missing constructs, to fix errors, and to ultimately obtain a grammar that could be used for parsing.
Some SLE resources
Language Implementation Patterns

Create Your Own Domain-Specific and General Programming Languages

Terence Parr
SLEish events
Welcome

The 6th International Conference on Software Language Engineering (SLE) is devoted to topics related to artificial languages in software engineering.

SLE's mission is to encourage and organize communication among communities that have traditionally looked at software languages from different and yet complementary perspectives.
4th Summer School on
Generative and Transformational Techniques in Software Engineering
3–9 July, 2011, Braga, Portugal
Co-located with: SLE 2011 | CSXW 2011 | ITSLE 2011
Welcome to the Homepage of the

**ACM/IEEE 15th International Conference**
on Model Driven Engineering Languages & Systems
**MODELS 2012**

Sept. 30th - Oct. 5th, 2012 - Innsbruck/AUSTRIA

Models have long been used in the engineering of software systems, but as computer based systems become larger, complexer and more critical to human society the importance of modeling in the software engineering lifecycle has rapidly grown. MODELS is the premier conference series for model-based software and systems engineering which since 1998 has been covering all aspects of modeling, from languages and methods to tools and applications.

In 2012 we will continue this tradition in the midst of Tyrol’s breathtaking scenery and stunning mountains. A university town, with a strong record of academic excellence and a vibrant student scene, Innsbruck combines tradition and splendour with modern facilities and attractions. We hope you will join us in this international conference and Olympic city to help shape the modeling methods and technologies of the future.
12th IEEE International Working Conference on Source Code Analysis and Manipulation

23–24 September 2012 – Riva del Garda, Trento, Italy

Sponsored by:

IEEE computer society  SD  GRAMMATech  CREST

The aim of this working conference is to bring together researchers and practitioners working on theory, techniques and applications which concern analysis and/or manipulation of the source code of computer systems. While much attention in the wider software engineering community is properly directed towards other aspects of systems development and evolution, such as specification, design and requirements engineering, it is the source code that contains the only precise description of the behaviour of the system. The analysis and manipulation of source code thus remains a pressing concern.
Welcome to WCRE 2013

The Working Conference on Reverse Engineering (WCORE) is the premier research conference on the theory and practice of recovering information from existing software and systems. WCRE explores innovative methods of extracting the many kinds of information that can be recovered from software, software engineering documents, and systems artifacts, and examines innovative ways of using this information in system renovation and program understanding.


This year WCRE will be held in University of Koblenz-Landau, Germany.

http://wcre.wikidot.com/2013
SLEish papers

Read more here:
Programming with Models? Modeling with Code. The Role of Models in Software Development
An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models
Guest Editors' Introduction to the Special Section on Software Language Engineering
The API Field of Dreams - Too Much Stuff! It's Time to Reduce and Simplify APIs!
Domain specific language implementation via compile-time meta-programming
The Impedance Imperative Tuples + Objects + Infosets = Too Much Stuff!
Extending grammars and metamodels for reuse: the Reuseware approach
Controllable Combinatorial Coverage in Grammar-Based Testing
Semantics First! - Rethinking the Language Design Process
SugarJ: library-based syntactic language extensibility
Toward an engineering discipline for grammarware
Using production grammars in software testing
Automated Co-evolution of GMF Editor Models
Embedding Languages without Breaking Tools
Engineering a DSL for Software Traceability
A Case Study in Grammar Engineering
Safe Composition of Transformations
Why Software Language Engineering?
SLEish technologies
Technologies with focus on language implementation or language services

Stratego/XT  Rascal  ASM
Xtext  ATL  GMF  Recoder
JastAdd  SugarJ  EMF  JDK
TXL  ATL  EMF  Recoder
JDK
This SLE course: content / structure
## Lecture (Historical data)

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Educational objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.10.2012</td>
<td>Introduction</td>
<td>Observe SLE in the computer science context</td>
</tr>
<tr>
<td>30.10.2012</td>
<td>Grammars and parsing</td>
<td>Implement parsers</td>
</tr>
<tr>
<td>06.11.2012</td>
<td>Language processing</td>
<td>Distinguish forms of processors</td>
</tr>
<tr>
<td>13.11.2012</td>
<td>Attribute grammars</td>
<td>Implement semantic analysis</td>
</tr>
<tr>
<td>20.11.2012</td>
<td>Rewriting &amp; strategies</td>
<td>Implement software transformations</td>
</tr>
<tr>
<td>27.11.2012</td>
<td>Automated refactoring</td>
<td>Discover challenges of refactoring</td>
</tr>
<tr>
<td>11.12.2012</td>
<td>Templates and friends</td>
<td>Discover challenges of generative metaprograms</td>
</tr>
<tr>
<td>18.12.2012</td>
<td>Grammar-based testing</td>
<td>Testing language processors</td>
</tr>
<tr>
<td>08.01.2013</td>
<td>Magnolia: design and implementation</td>
<td>Access to practical experiences</td>
</tr>
<tr>
<td>15.01.2013</td>
<td>Domain-specific languages</td>
<td>Implement DSLs by embedding</td>
</tr>
<tr>
<td>22.01.2013</td>
<td>Language design and evolution</td>
<td>Master a process for language design</td>
</tr>
<tr>
<td>29.01.2013</td>
<td>Linguistic architecture</td>
<td>Transpose software architecture to SLE</td>
</tr>
</tbody>
</table>
# Lab (Historical data)

<table>
<thead>
<tr>
<th>Date</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>08.11.2012</td>
<td>Implement parsers and interpreters with <a href="#">Rascal</a></td>
</tr>
<tr>
<td>15.11.2012</td>
<td>Implement semantic analyses with <a href="#">JastAdd</a></td>
</tr>
<tr>
<td>22.11.2012</td>
<td>Implement term-rewriting strategies with <a href="#">Rascal</a></td>
</tr>
<tr>
<td>29.11.2012</td>
<td>Transform code with <a href="#">Rascal</a></td>
</tr>
<tr>
<td>06.12.2012</td>
<td>Presentations Assignment No. 1</td>
</tr>
<tr>
<td>13.12.2012</td>
<td>Generate Java bytecode with <a href="#">ASM</a></td>
</tr>
<tr>
<td>10.01.2013</td>
<td>Presentations Assignment No. 2</td>
</tr>
<tr>
<td>17.01.2013</td>
<td>Implement embedded DSLs with <a href="#">SugarJ</a></td>
</tr>
<tr>
<td>24.01.2013</td>
<td>Test language processors with <a href="#">QuickCheck</a> or <a href="#">randoop</a></td>
</tr>
<tr>
<td>31.01.2013</td>
<td>Implement a coupled transformation with <a href="#">Rascal</a></td>
</tr>
<tr>
<td>07.02.2013</td>
<td>Presentations Assignment No. 3 and Exam preparation</td>
</tr>
</tbody>
</table>
## Assignments (Historical data)

<table>
<thead>
<tr>
<th>No.</th>
<th>Topic</th>
<th>Date of issue</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Implement a program analysis</td>
<td>15.11.2012</td>
<td>05.12.2012</td>
</tr>
<tr>
<td>2</td>
<td>Implement a program transformation</td>
<td>06.12.2012</td>
<td>09.01.2012</td>
</tr>
<tr>
<td>3</td>
<td>Implement a DSL through embedding</td>
<td>15.01.2013</td>
<td>07.02.2013</td>
</tr>
</tbody>
</table>
Exam rules (Historical data)

- Form of exam: oral
- Window for exam: 20 Feb - 30 March 2012
- Structure of exam (30min):
  - 7min: discussion of an abstract concept chosen by examiner for a lecture chosen by student
  - 7min: discussion of a language- or technology-specific concept chosen by examiner for another lecture chosen by student
  - 10min: Q&A for an assignment solution chosen by student without much overlap to the previous topics
  - 6min: discussion of an assignment improvement
  - See here for the abstract as well as language- and technology-specific concepts per lecture.
- Admission to exam
  - Regular attendance of lecture and lab.
  - Assemble teams of 1-3 members, subject to approval.
  - Each team submits reasonable attempts to all assignments.
  - Assignments must be presented in the lab.
Hidden agenda

- Most of you enjoy and finish the exam.
- Some of you fall in love with SLE.
- Few of you co-submit to SLE 20??.
- All of us have fun.

This is it!
This is the course!
(for language engineers)

http://softlang.wikidot.com/course:sle
Thanks!

Questions? Comments?