Software Evolution: Conclusion, Discussion, Future Work

Dr. Vadim Zaytsev aka @grammarware
UvA, MSc SE, 7 December 2015
<table>
<thead>
<tr>
<th>Week</th>
<th>Topic</th>
<th>Presenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>W44</td>
<td>Introduction</td>
<td>V. Zaytsev</td>
</tr>
<tr>
<td>W45</td>
<td>Metaprogramming</td>
<td>J. Vinju</td>
</tr>
<tr>
<td>W46</td>
<td>Reverse Engineering</td>
<td>V. Zaytsev</td>
</tr>
<tr>
<td>W47</td>
<td>Software Analytics</td>
<td>M. Bruntink</td>
</tr>
<tr>
<td>W48</td>
<td>Clone Management</td>
<td>M. Bruntink</td>
</tr>
<tr>
<td>W49</td>
<td>Source Code Manipulation</td>
<td>V. Zaytsev</td>
</tr>
<tr>
<td>W50</td>
<td>Conclusion</td>
<td>V. Zaytsev</td>
</tr>
<tr>
<td>W51</td>
<td>Legacy and Renovation</td>
<td>D. Blasband</td>
</tr>
</tbody>
</table>
Software Types

S

P

E

Laws of Software Evolution

http://www.computer.org/web/awards/mills-meir-lehman

Types of maintenance

EASY

System

extract

Representation

synthesise

Results

analyse

Rascal

http://rascal-mpl.org
Reverse Engineering

* Parsing, slicing, exploration...

* PR, KR, IR, MUD, clustering...

Parsing is...
3.1 Two classes of parsing methods

A parsing method constructs the syntax tree for a given sequence of tokens. Constructing the syntax tree means that a tree of nodes must be created and that these nodes must be labeled with grammar symbols, in such a way that:

- leaf nodes are labeled with terminals and inner nodes are labeled with non-terminals;
- the top node is labeled with the start symbol of the grammar;
- the children of an inner node labeled $N$ correspond to the members of an alternative of $N$, in the same order as they occur in that alternative;
- the terminals labeling the leaf nodes correspond to the sequence of tokens, in the same order as they occur in the input.

Left-to-right parsing starts with the first few tokens of the input and a syntax tree, which initially consists of the top node only. The top node is labeled with the start symbol.

The parsing methods can be distinguished by the order in which they construct the nodes in the syntax tree: the top-down method constructs them in pre-order, the bottom-up methods in post-order. A short introduction to the terms “pre-order” and “post-order” can be found below. The top-down method starts at the top and con-
deconstructing the problem into primitives. Section 7 presents a framework and algorithm for the evolution of modelling artefacts when languages evolve. Section 8 concludes the paper and describes future work.

2. Modelling languages

To allow for a precise discussion of language evolution, we briefly introduce fundamental modelling language concepts. This introduction which we elaborated in [10] is based on foundations laid by Harel and Rumpe [13] and Kühne [21]. The two main aspects of a model are its syntax (how it is represented) and its semantics (what it means).

Firstly, the syntax comprises concrete syntax and abstract syntax. The concrete syntax describes how the model is represented (e.g., in 2D vector graphics or in textual form), which can be used for model input as well as for model visualisation. The abstract syntax contains the “essence” of the model (e.g., as a typed Abstract Syntax Graph (ASG)—when models are represented as graphs).

A single abstract syntax may be represented by multiple concrete syntaxes. There exists a mapping between a concrete syntax and its abstract syntax, called the parsing function. There is also a mapping in the opposite direction, called the rendering function. These are the concrete mapping functions. Mappings are usually implemented, or can at least be represented, as model transformations. The abstract syntax and concrete syntax of a model are related by a surjective homomorphic function that translates a concrete syntax graph into an abstract syntax graph.

Secondly, the semantics of a model are defined by a complete, total and unique semantic mapping function which maps every abstract syntax model onto a single element in a semantic domain, such as Ordinary Differential Equations, Petri nets [39], or a set of behaviour traces. These are domains with well-known and precise semantics. For convenience, semantic mapping is usually performed on abstract syntax, rather than on concrete syntax directly. More explicitly, the abstract syntax can be used as a basis for semantic anchoring [4].

A meta-model is a finite model that explicitly describes the abstract syntax and static semantics, which are statically checkable, of a language. Dynamic semantics are not covered by the meta-model. The abstract syntax of a model can be represented as a graph, where the nodes are elements of the language and the edges are relations between these elements, and also elements of the language. Instance models of the language are said to conform to the meta-model of the language. In [21], Kühne refers to this relation as linguistic instance of. The description of the abstract syntax is typically specified in a modelling language such as UML Class Diagrams [34]. Static semantics can be described in a constraint language such as the Object Constraint Language (OCL) [36]). Often, but not necessarily, the concrete syntax mapping is directly attached to a meta-model, where every element of the concrete syntax can be explicitly traced back to its corresponding element of the abstract syntax.

Fig. 1 shows the different kinds of relations involving a model m. Relations are visualised by arrows, “conforms to”-
Program Models

V. Zaytsev, A. H. Bagge, Parsing in a Broad Sense, MoDELS 2014.
http://bibtex.github.io/MoDELS-2014-ZaytsevB.html
read(text);
read(n);
lines = 1;
chars = 1;
subtext = "";
c = getChar(text);
while (c != \'\eof\')
    if (c == \'\n\')
        then lines = lines + 1;
            chars = chars + 1;
        else chars = chars + 1;
        if (n != 0)
            then subtext = subtext ++ c;
                n = n - 1;
    c = getChar(text);
write(lines);
write(chars);
write(subtext);

STRATOS

Our goal in designing STRATOS (Figure 1) is to support planners in the release planning decision-making process through visualization. Our solution combines the flow visualization of Sankey diagrams with the multivariate illustration of Parallel Coordinates. Our data set is inherently hierarchical, with plans containing releases and releases containing features, thus we opted to use a forest or multiple tree layout [7].

STRATOS visualizes the important factors of release planning in a unified, single layout. This ensures that all of the factors are readily available to the planner. We also implemented interactive brushing, allowing components to interactively reveal relationships in the data. Our design process relied on gaining knowledge and feedback from a release planning domain expert.

Design Process and Guidelines

We used a method akin to Sedlmair et al.'s design study methodology [18]. We worked closely with a release planning domain expert who emphasized that a visualization could help planners in release planning. He helped identify important patterns and relationships, providing guidance for the development of STRATOS. We iterated on several visualization designs specifically to highlight these relationships which were not readily apparent with basic visualizations.

The underlying design guidelines for STRATOS are:

1. Consider as many as possible factors. Knowing the conditions of multiple factors and constraints of release planning is important for planners to be able to make good and well-informed decisions. The visualization design must take into account visualizing as many factors as possible.

2. Provide a holistic view. The visualization must also be able to show how the factors and constraints relate to each other. A holistic view allows planners to consider most of the factors with considerable ease rather than trying to do so while switching between views. Hybrid visualization brings together different aspects of existing visualization techniques to create something novel. We bring the advantages of several techniques together to make data comparisons more transparent.

3. Support comparison between plans. While plans will be shown as distinct, consistency across representations must be used to support comparison between plans.

4. Support different strategies for decision-making. Different planners often have different ways of deciding what is the best plan regarding their project. An interactive visualization should allow planners to explore the data however they prefer (e.g. allowing a planner to start their exploration of the data anywhere in the visualization).

5. Support both individual and collaborative exploration of data. Release planning can be performed either as an individual or as a team. This extends Guideline 4 in case of collaborative planning.

6. Support details-on-demand [19]. While visually conveying information allows planners to do simple comparisons at-a-glance, they must be able to access detailed information for fine-grained analysis.

7. Minimize required interactions. Minimizing interaction overhead by avoiding deeply nested menus and other complicated actions while still providing full visualization and data access could help simplify the planner's task.
Information Retrieval
To Measure is to Know

* Scales
  * nominal
  * ordinal
  * interval
  * ratio
  * absolute

* Examples
  * team size
  * code size
  * run time
  * SIG stars
  * colours
Goal—Question—Metric

* **Goal** – conceptual
  * purpose / issue / object / pov

* **Question** – operational
  * can be multiple per goal

* **Metric** – quantifiable
  * can be multiple per question

Clone Terminology

* Clone
* Clone pair
* Clone class
* Type I
* Type II
* Type III
* Type IV
Clone Types

* Type I: exact
  * copy-paste + indent/comment

* Type II: parametrised
  * copy-paste + convention/typing

* Type III: near-miss
  * copy-paste + hacking/maintenance

* Type IV: semantic
  * copy-paste + refactoring

Other Clone Types

* **Structural** clones
  * implementation patterns & notations

* **Artefact** clones
  * entire files, classes, functions...

* **Model** clones
  * not-quite-code

* **Contextual** clones
  * duplicate due to usage patterns

Hamid, Zaytsev, Detecting Refactorable Clones by Slicing Program Dependence Graphs, SATToSE 2014.
How Much Code is Cloned?

* 12.7% [Baxter et al. ICSM’98]
* 10–15% [Kapser & Godfrey JSME’06]
* 7–24% [Roy & Cordy JSME’10/WCRE’08]
* 50% [Ducasse et al. JSME’06]
* 7–23% [Baker WCRE’95]

Clone Management

Detection -> Documentation -> Tracking

Visualisation

Annotation

Recommendation

Prevention

# IDE-based Approach

<table>
<thead>
<tr>
<th></th>
<th>SimScan</th>
<th>SDD</th>
<th>Simian</th>
<th>CloneDigger</th>
<th>ChiP</th>
<th>CeR</th>
<th>CloneBoard</th>
<th>SHINOB</th>
<th>CloneDR</th>
<th>Zibran and Roy</th>
<th>JClone</th>
<th>CloneTracker</th>
<th>CeDAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrated Detection Engine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy-paste Detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-1 Clone Detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-2 Clone Detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-3 Clone Detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-the-fly/Focused Detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clone Visualization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Change History View</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clone Genealogy View</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clone Tracking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clone Refactoring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SCAM

* Partial evaluation
* Generative programming
* Staging and morphing
* Optimisation
* Folding and unfolding
* Superoptimisation
Language Conversion

Original program → Syntax swap → Target program

Restructuring

Up-compilation

* CSS to SASS
* ~70% less code
* ~5% less padding
* ~10% in mixins
* ~8% to children
* ~2 CSS decls per SASS var
Stay tuned:
guest lecture
next week