Live programming

Programmeren als boetseren



Tijs van der Storm (@tvdstorm / storm@cwi.nl)



Introduction

about me...

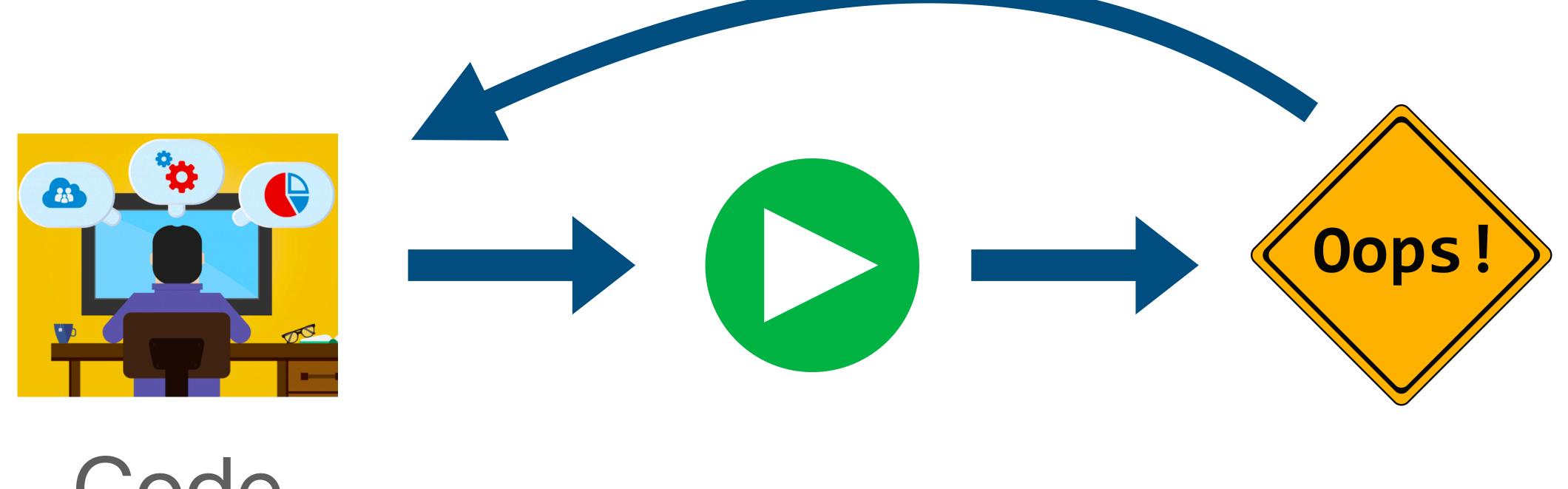
- Groepsleider Software Analysis & Transformation; Prof in Groningen
- Wij doen onder andere onderzoek naar:
 - hoe beter programmeertalen te maken (language engineering)
 - hoe betere programmeertalen te maken (language design)
- We gebruiken hiervoor Rascal, een meta-programmeertaal
 - https://www.rascal-mpl.org/
- Vandaag: live programmeren





Programmeren

back to the drawing board...



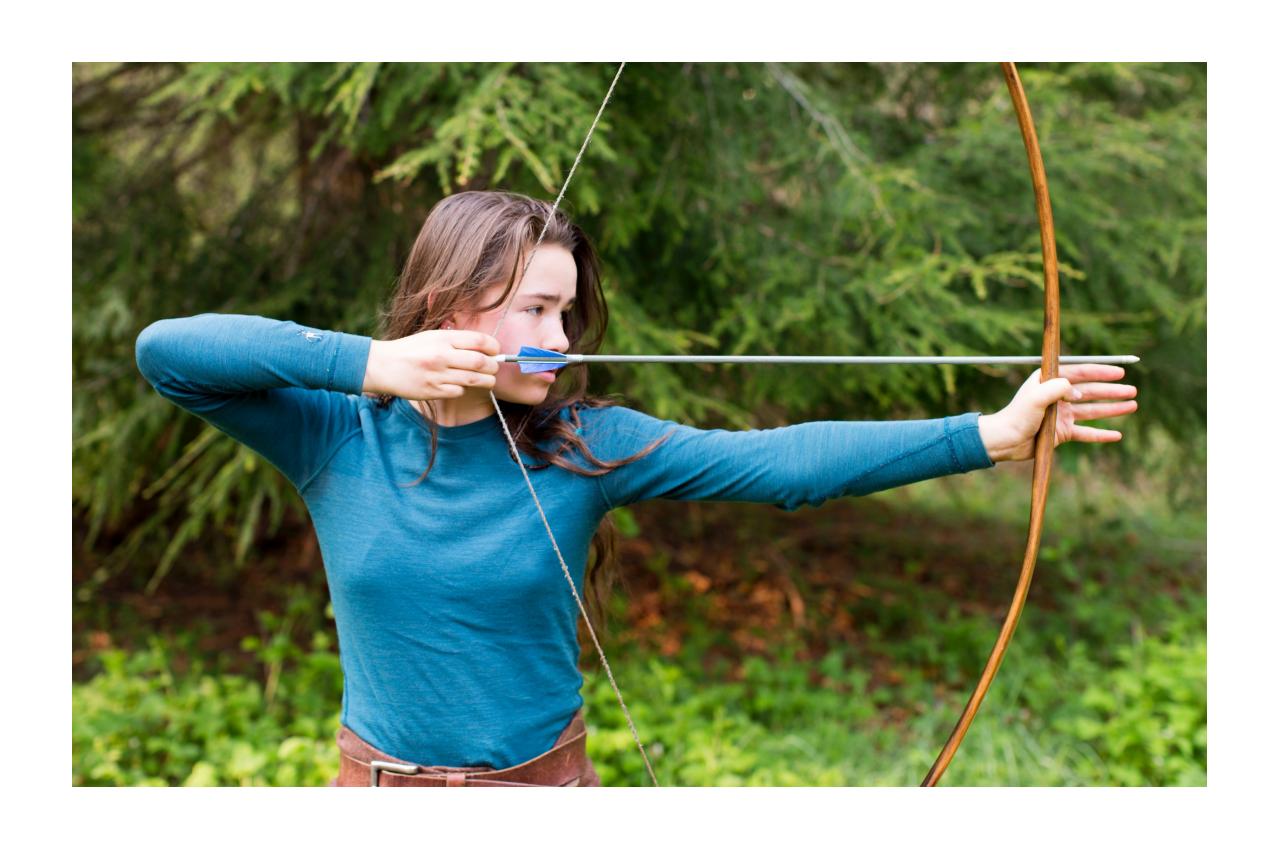
Code schrijven

RUN!

Fout (%)

Programmeren...

...is als boogschieten

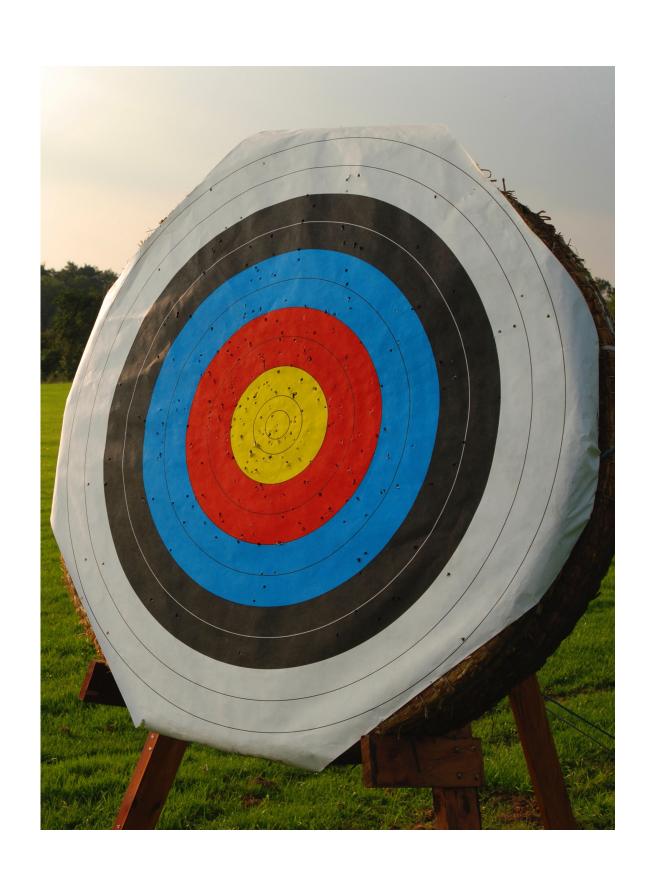




Richten, schieten, missen; opnieuw. Olympische sport...

Live programmeren





Vloeiende ervaring, continue feedback: "kinderspel"

Programmeren als boetseren

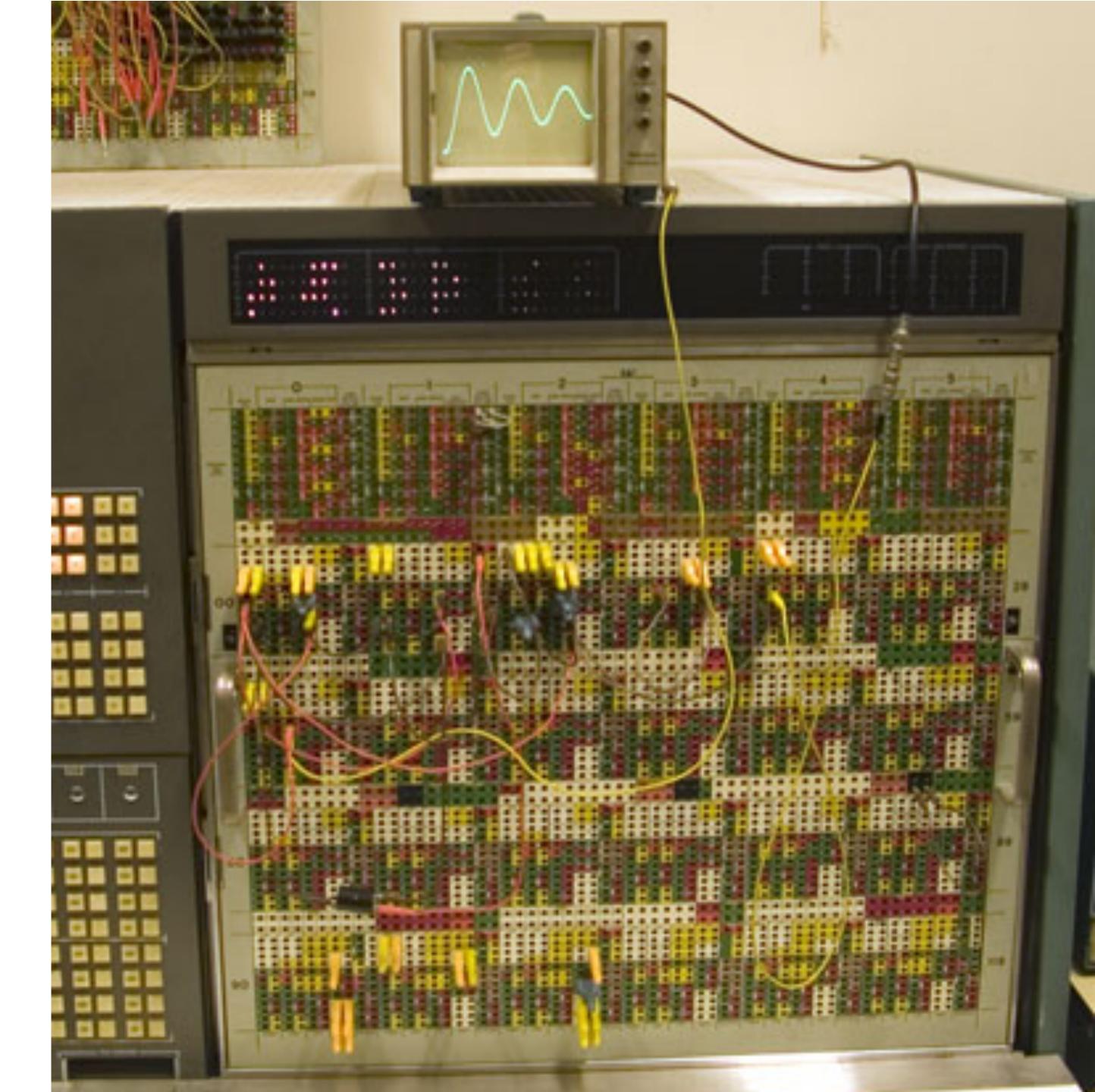
- Er zijn geen stappen (edit, compile, run; repeat)
- Voortdurende, continue feedback
- Effect van handeling meteen zichtbaar
- Maakproces en materiaal vloeien naadloos in elkaar over

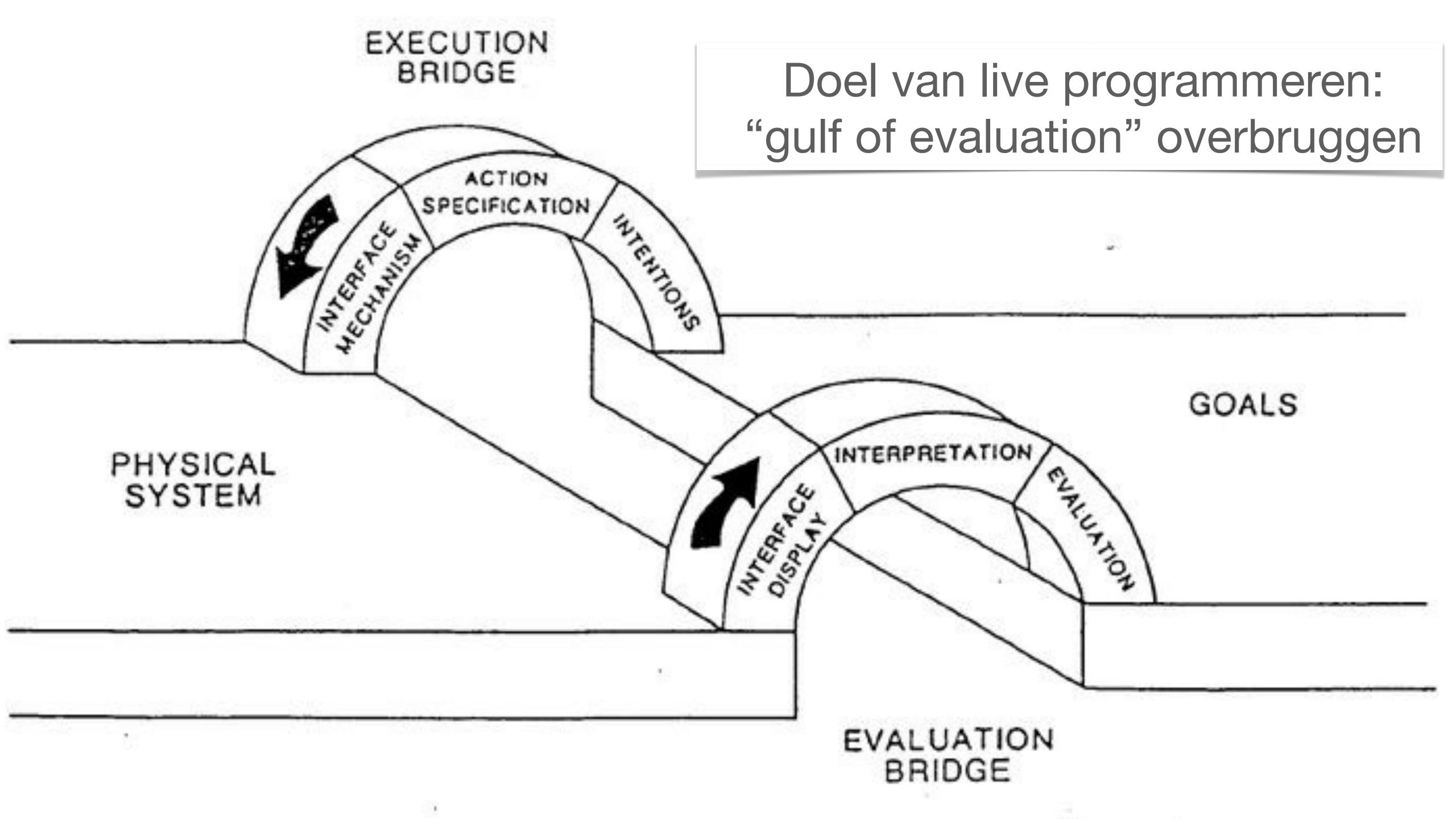
Wat als programmeren zo kon zijn?



Waar komt "live" programming vandaan?







Intermezzo: little languages domain-specific languages (DSLs)

- General purpose programmeertalen:
 - "groot", algemeen gebruik
 - Bv. Java, C, Swift, Javascript, etc.
- Domein-specifiek:
 - "klein", toegespitst
 - Bv. HTML, SQL, TeX, CSS, Graphviz, ...
- Kan groot effect op kwaliteit software hebben

general purpose:



domein-specifiek:



Live demo

State machines

```
1 events
2  open OPEN
3  close CLOSE
4 end
5
6 state closed
7  open => opened
8 end
9
10 state opened
11  close => closed
12 end
```

Interactieve enquêtes

```
1 form taxOfficeExample {
    "Did you sell a house in 2020?"
     hasSoldHouse: boolean
    "Did you buy a house in 2020?"
     hasBoughtHouse: boolean
    "Did you enter a loan?"
     hasMaintLoan: boolean
   if (hasSoldHouse) {
      "Private debts for the sold house:
        privateDebt: integer
      "What was the selling price?"
        sellingPrice: integer
      "Value residue:"
       valueResidue: integer =
        sellingPrice - privateDebt
```

```
Did you sell a house in 2020?

Did you buy a house in 2020?

Did you enter a loan?

Private debts for the sold house: 0

What was the selling price?

Value residue:
```

Het maken van live programmeertalen

Waarom is dit een uitdaging?

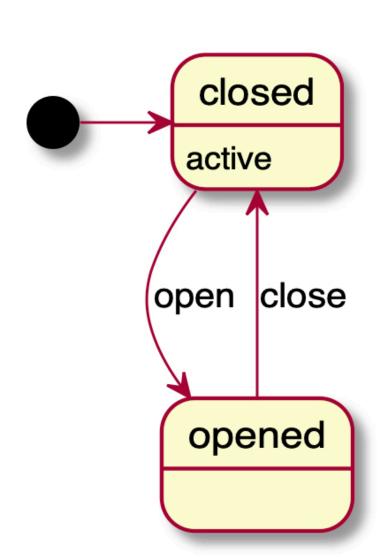
- Broncode is een soort sjabloon
- Het draaien van het programma is het invullen van het sjabloon (met data)
- Als je het sjabloon verandert, wat moet er dan met de ingevulde data gebeuren?
- Die data moeten gemigreerd worden, zonder het programma te stoppen.

Sjabloon...

state closed
 open => opened
end

state opened
 close => closed
end

...ingevuld



"alle deuren"

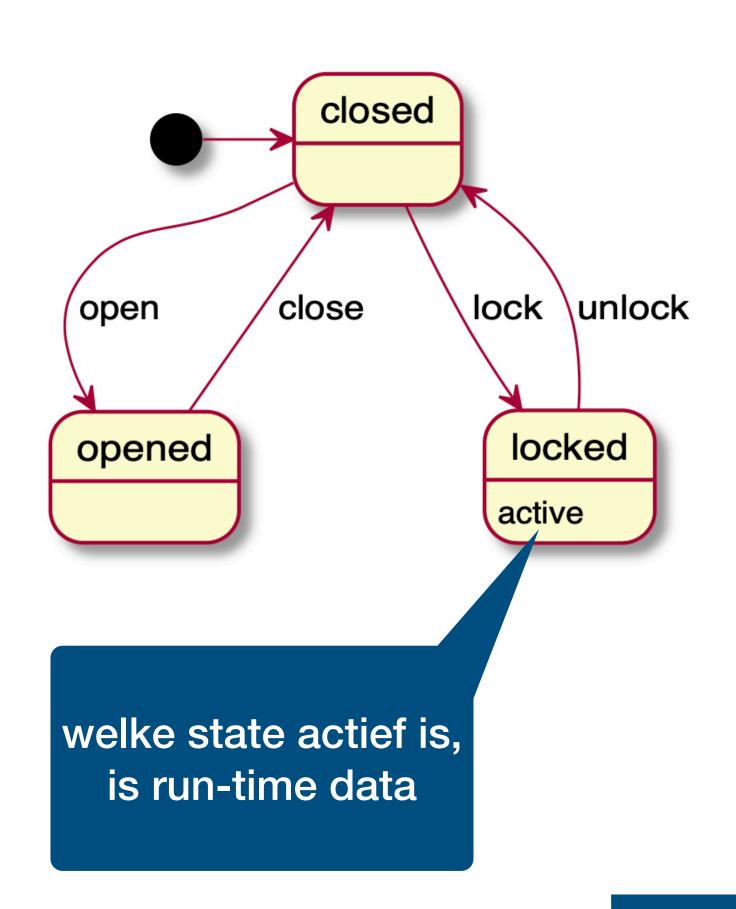
"een specifieke deur"

Voorbeeld van toestandsmigratie

```
state closed
  open => opened
  lock => locked
end
```

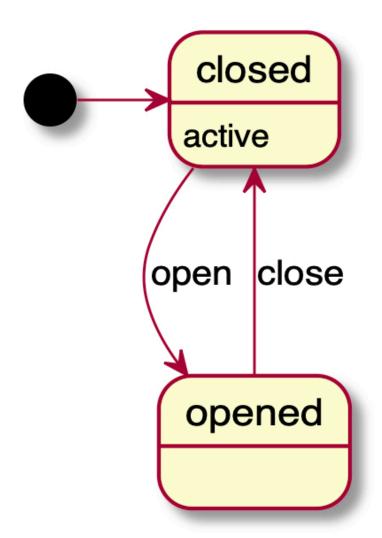
state opened
 close => closed
end

state locked
 unlock => closed
end



state closed
 open => opened
end

state opened
 close => closed
end



welke state moet nu actief zijn!?!

verwijder "locked"

Onderzoek naar live programming

fundamentele technieken om programmeertalen live te maken

CrossMark



Constraint-based Run-time State Migration for Live Modeling

Ulyana Tikhonova Centrum Wiskunde & Informatica (CWI) Amsterdam, The Netherlands ulyana.tikhonova@cwi.nl

Tijs van der Storm Centrum Wiskunde & Informatica (CWI) Amsterdam, The Netherlands University of Groninge Groningen, The Netherlands

Abstract

Live modeling enables modelers to incrementally update models as they are running and get immediate feedback about the impact of their changes. Changes introduced in a model may trigger inconsistencies between the model and its run-time state (e.g., deleting the current state in a statemachine); effectively requiring to migrate the run-time state to comply with the undated model. In this paper, we introduce an approach that enables to automatically migrate such runtime state based on declarative constraints defined by the language designer. We illustrate the approach using NEXTEP. a meta-modeling language for defining invariants and migration constraints on run-time state models. When a model changes, Nexter employs model finding techniques, backed by a solver, to automatically infer a new run-time model that satisfies the declared constraints. We apply Nexter to define migration strategies for two DSLs, and report on its

CCS Concepts • Software and its engineering → Domain specific languages; Software prototyping; • Theory **of computation** → *Programming logic*;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies or components of this work owned by others than the author(s) mus be honored. Abstracting with credit is permitted. To copy otherwise, or epublish, to post on servers or to redistribute to lists, requires prior specific

Jouke Stoel Centrum Wiskunde & Informatica (CWI) Amsterdam, The Netherlands Eindhoven University of Technology Eindhoven. The Netherlands

Thomas Degueule Centrum Wiskunde & Informatica (CWI) Amsterdam, The Netherlands thomas.degueule@cwi.nl

Keywords live modeling, run-time state migration, DSL,

Ulyana Tikhonova, Jouke Stoel, Tijs van der Storm, and Thomas Degueule, 2018, Constraint-based Run-time State Migration for Live Modeling. In Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5-6, 2018, Boston, MA, USA. ACM, New York, NY, USA, 13 pages.

Live modeling [27, 28] allows users of executable model ing languages to enjoy live, immediate feedback when edit ing their models, without having to restart execution. Live modeling has its roots in live programming, which allows plication simultaneously, without long edit-compile-restart cycles [18, 21]. When the programmer changes the program code, its execution state is updated accordingly on-the-fly This effectively bridges the "gulf of evaluation" [14] between

While the value of live modeling and live programming is widely recognized [13, 14, 26], engineering live software languages requires a lot of effort and a deep understanding of the host language and its particular domain of application [4]

A central problem for live modeling languages is how to reconcile changes to a model with the run-time state of its execution. Changes to a model or program might invalidate he current run-time state. That is, when the executing model is modified, its run-time state still corresponds to the version pefore the change. The problem is analogous to migrating

JET. OPEN

Best Software Engineering Technology

Constraint-based Run-time State Migration for Live Modeling Ulyana Tikhonova, Jouke Stoel,

Tijs van der Storm, Thomas Dequeule Published in ACM SIGPLAN International Conference on

Software Language Engineering (SLE) 2018

running program by changing its source code, receiving paper, we propose an approach to bridge the gap between models · Models at run time DSL code. By leveraging ordinary text differencing and ori- 1 Introduction gin tracking, TMDIFF produces deltas defined in terms of the the model deltas are applied at run time to update a running system, without having to restart it. Since the model deltas her about the effect of that action [23]. Live programming execution. We therefore propose a generic, dynamic patch and observing its behavior. In a live programming environcase study of a live programming environment for a simple see the behavioral effects of their actions and learn predict-Communicated by Prof. Alfonso Pierantonio, Jasmin Blanchette, Francis Bordeleau, Nikolai Kosmatov, Prof. Gabriele Taentzer, Prof. Manuel Amsterdam University of Applied Sciences, PO Box 1025, 1000 BA Amsterdam, The Netherlands Centrum Wiskunde & Informatica, PO Box 94079,

nttps://doi.org/10.1007/s10270-017-0608-7

SPECIAL SECTION PAPER

University of Groningen, Johann Bernoulli Institute, Nijenborgh 9, 9747 AG Groningen, The Netherlands

Toward live domain-specific languages From text differencing to adapting models at run time Riemer van Rozen¹ · Tijs van der Storm² Received: 27 June 2016 / Revised: 26 May 2017 / Accepted: 20 June 2017 / Published online: 14 August 2017 acterized by incremental change and immediate feedback. and executing state machines. Instead of long edit-compile cycles, developers modify a

mmediate feedback as it instantly adapts in response. In this running programs and textual domain-specific languages (DSLs). The first step of our approach consists of applying a novel model differencing algorithm, TMDIFF, to the textual etamodel of a language. In the second step of our approach,

 $languages \cdot Text \ differencing \cdot Model \ patching \cdot Adapting$

an action performed by a user and the feedback provided to aims to bridge the gulf of evaluation by shortening the feed are unaware of any run-time state maintained during model back loop between editing a program's textual source code architecture, RMPATCH, which can be customized to cater for ment, the running program is updated instantly after every domain-specific state migration. We illustrate RMPATCH in a change in the code [34]. As a result, developers immediately ing how the program adapts to targeted improvements to the code. In this paper, we are concerned with providing generic, reusable frameworks for developing "live DSLs", languages whose users enjoy the immediate feedback of live execution. We consider such techniques to be first steps toward providing automated support for live languages in language workbenches [8].

> In particular, we propose two reusable components, TMD-DSLs, based on a foundation of metamodeling and model interpretation. TMDIFF is used to obtain model-based deltas from textual source code of a DSL. These deltas are then applied at run time by RMPATCH to migrate the execution of the DSL program [38]. This enables the users of a DSL to modify the source and immediately see the effect.

Adapting Game Mechanics with Micro-Machinations

Amsterdam University of Applied Sciences
Duivendrechtsekade 36-38 1096 AH
Amsterdam, The Netherlands
r.a.van.rozen@hva.nl

ABSTRACT

In early game development phases game designers adjust game rules in a rapid, iterative and flexible way. In later phases, when software prototypes are available, play testing provides more detailed feedback about player experience More often than not, the realized and the intended game play emerging from game software differ. Unfortunately, ad-justing it is hard because designers lack a means for effiently defining, fine-tuning and balancing game mechanics The language Machinations provides a graphical notation for expressing the rules of game economies that fits with a designer's understanding and vocabulary, but is limited to design itself. Micro-Machinations (MM) formalizes the meaning of core language elements of Machinations enabling soning about alternative behaviors and assessing qualit making it also suitable for software development. We propose an approach for designing, embedding and adapting game mechanics iteratively in game software, and demonstratively in game trate how the game mechanics and the gameplay of a tower efense game can be easily changed and promptly play tested the approach shows that MM enables the adaptability neede to reduce design iteration times, consequently increasing opportunities for quality improvements and reuse.

In computer game development, developers face problems that arise from increased complexity. Challenges include increased development speed, changing technologies, and growing teams of experts with varying vocabularies. Teams nay consist of artists, software engineers, domain experts the large number of games produced today lacks a common vocabulary, methods for designing games and sharing This work is part of the EQuA Project. http://www.equaproject.nl

This work is part of the Automated Game Design Project.

earch is performed at the SWAT group of CWI.

Joris Dormans[†] Amsterdam University of Applied Sciences Duivendrechtsekade 36-38 1096 AH Amsterdam, The Netherlands j.dormans@hva.nl

knowledge and artifacts. The need for common design vocabularies [6,23], game design patterns [4], specialized game grammars [9,16], and computer assisted design tools [18,19] has been expressed for some time, but so far no tool, method, or framework has surfaced as an industry standard. As result, game design relies strongly on iterative prototyping, play-testing, and reprogramming parts to improve games.

tem defined by the game mechanics [10]. Therefore, game play can only be evaluated after the system has been built Because a designer's understanding about how game rules affect the player is constantly changing, they have to make adaptations quickly and often. However, as software de ity of the designer to design, play test, gain feedback and improve, which results in longer design iterations and missed opportunities for improving the quality. From a software engineering point-of-view gameplay adaptations represent a problematic stream of badly defined, poorly understood reuirements that result in wasted effort, ineffective attempt reuse, and repetitive and error-prone coding cycles, in-

back. The language Machinations [1] provides a graphical notation for expressing the rules of game economies that is gaining popularity with designers. Micro-Machinations (MM) [14] formalizes the meaning of core language features of Machinations and adds modularity, making it also suit

informed and well-documented design decisions. The approach entails modeling game mechanics as embeddable software artifacts with MM. Additionally, it provides a means for adapting game mechanics at run-time using a library We demonstrate that changes to the game economy of a tower defense game can be easily designed and embedded in software. Our approach shows that it is feasible to significantly reduce design iteration times, by improving flexibilit and adaptability, thereby increasing opportunities for qual

Semantic Deltas for Live DSL Environments

Tijs van der Storm Centrum Wiskunde & Informatica (CWI) Science Park 123, Amsterdam, The Netherlands

Abstract—Domain-specific languages (DSLs) require IDE supenvironments to bridge the "gulf of evaluation" between DSL code and the running application. Semantic deltas are distinguished from textual or structural deltas in two ways. First, they have meaning in the application domain captured by the DSL. Second, they can be interpreted at runtime so that the behavior of he running system adapts to the evolved DSL code. Semantic deltas have the potential to support back-in-time debugging, and exploring what-if scenarios. I present early experiences in building a live DSL environment and identify areas for future

The goal of live programming is to bridge the "gulf of evaluation" [11],—the cognitive impedance mismatch between the source code of a program and its dynamic manifestation at runtime. Concretely, live programming aims to provide continuous semantic feedback on the state of running programs. The program can be stopped, restarted, its state can be inspected. Changes to the program are immediately reflected in the running instance.

Programming can be seen as a complex process of turning problem space requirements into solution space artifacts. The original problem specification is often "lost in translation" As a result, evolving software systems is time-consuming and error-prone. An important goal of DSLs is to bridge this "gulf of encoding" by providing high-level notations, tailored to specific problem domains [17]. They can be seen as specification languages to directly express requirements from which the actual code is automatically generated. Problemoriented notations improve communication with stakeholders and domain experts. A live DSL environment may amplify this benefit: the DSL code is not only easier to understand and maintain, but can also be inspected, tested and explored

Turning DSL implementations into live DSL environments requires reasoning about changes to DSL code. This is where semantic deltas come in. The essential idea is to represent

changes to the source code of a DSL program as explicit,

- is that the evolution history and application state history are
- Semantic deltas as a foundation for live DSL environment have many potential advantages:
- Liveness Changes to the program are immediately applied to the running application. As a result, the user of such a system can enjoy immediate feedback on the effect of changes to the program.
- Persistence Recording the sequence of deltas and storing it on disk allows a session to be restarted at any later moment in time and facilitates post-mortem analyses.
- · Version control Storing source-code semantic deltas provides a basis for domain-specific version control.
- End-user programming Effacing the distinction between source changes and runtime state changes has the potential to enable end-users to modify their applications. Meta-level and object-level can be modified using the
- · Backtracking If deltas maintain enough information to be inverted, this enables going back in time. In essence, application-level undo/redo and reverting to a previous version become one and the same thing.
- known, states (or program versions) in the past can be updated, after which deltas that are still valid are replayed.
- Time branching To explore what-if scenarios, an application (program + running instance) can be forked, so that the resulting two instances follow different branches in
- forked applications can be merged again. As with all merging, this might require interactive conflict resolution.

In the rest of this paper I present a simple example of a live DSL environment and discuss related work, open research questions and opportunities to push forward the development

II. EXAMPLE: LIVEOL

remantically meaningful change packages. Such deltas are A. A Live DSL Environment

Live Literals

Tijs van der Storm, Felienne Hermans

Live programming environments improve programmer experience by providing views of program execution which are continuously, and instantaneously updated. In most existing work on liveness, these views are considered part of the IDE: separate windows, panels, or widgets allow programmers to inspect and interact with live data and program execution. In this paper we present "live literals" where the source code itself is used as vehicle for immediate feedback and direct manipulation. Live literals are like ordinary programming language literals, but they are automatically updated after changes to the code. We illustrate the concept of live literals in Javascript using three applications: embedded spreadsheets, live units tests, and probes.

Aarieke Huisman Eelco Visser

Live programming

"programmeren als boetseren"

- Live programming verkleint de kloof tussen programma en executie
- Geen "edit, compile, run; repeat"-cycle, maar een vloeiende overgang
- Programmeurs ervaren snellere feedback en directere omgang met programma's
- Actief onderzoek naar generieke principes om talen meer "live" te maken.

