DOI:10.1145/3596217

Describing a framework to support simpler development of languages best suited to express the problems and solutions of each particular domain.

BY WALTER CAZZOLA AND LUCA FAVALLI

Scrambled Features for Breakfast: Concepts of Agile Language Development

LANGUAGE IS A vehicle of thought. Language philosophers suggest that language has seminal importance in the transfer of thought between individuals, whether they embrace the strong linguistic determinism hypothesis or the more relaxed linguistic relativism. This sentiment was shared by L. Wittgenstein in 1922; his *tractatus* argued that language limits what we can think about and that philosophy often attempts to say the unsayable. Fifty years later, mathematician E.W. Dijkstra agreed with the same concepts in the context of programming languages: The FORTRAN programming language was no longer an adequate vehicle of thought for the modern age because "it wastes our brainpower, is too risky and therefore too expensive to use."[12] K.E. Iverson, creator of APL, devoted his 1979 Turing Award lecture to showing that mathematical notation can be effectively combined "with the advantages of executability and universality found in computer languages" to

form a "single coherent language,"[22] even though some may argue against

$$\{1 \geq \rho\omega : \omega \bullet e \leftarrow \omega\,[?\rho\omega] \bullet (\nabla(\omega < e)/e), ((\omega = e)/\omega), \nabla(\omega > e)/\omega\}$$

being an intelligible implementation of quicksort. Computer scientists and well-known essayists such as Paul Graham[18] are convinced that language philosophy applies to programming languages. As such, the way we think of programs and the language that we use to write them are deeply connected.

For users to be able to *speak their mind*, programming languages should be designed so that they do not get in the way of their thought. In other words, programming languages should be designed to properly express problems and solutions of a *domain*: *Domain-specific languages* (DSLs) are programming languages that employ terms and concepts from a problem domain. DSLs improve comprehensibility by limiting comprehensiveness, so domain experts can both *validate* specifications written by others and *express* new ones themselves. There has

» key insights

- **Domain-specific languages are a powerful tool to properly convey solutions to the problems of the several application domains of complex software systems.**
- **Language development carries unique challenges that must be handled through proper methodology and tooling.**
- **Agile development processes and language workbenches can play a key role in easing language development and asserting the language-oriented programming paradigm as a valuable resource for the development of complex systems.**

ILLUSTRATION BY JUSTIN METZ

been interest in the value provided by a style of software development that seeks to describe software systems using a collection of DSLs. According to this vision—called *language-oriented programming*[17]—each reasonably-sized component of a complex software system should be thought about, designed, and developed using a language specifically created for that purpose.

How feasible is this approach? How expensive would it be to write each DSL from scratch? Language development is not a cost-free activity. To be used effectively, language development involves creating an entire ecosystem—that is, a language interpreter or a language compiler but also any integrated development environment (IDE) services, such as syntax highlighting and debuggers.[27] Such a problem is inherently complex and ultimately deemed the language-oriented programming paradigm unsuccessful. In this work, we tried to address this problem by combining proper methodology and tooling.

From a methodological perspective, we chose an agile process that has proven itself to be both productive and cost-effective: Scrum.[a] In Scrum, the *product owner* splits the work for a complex problem into a *product backlog* made of several features to be implemented, each called a product backlog item. Before each iteration (*sprint*), the stakeholders meet in a sprint planning event to refine the product backlog as needed and to determine the *sprint goal*—that is, the collection of product backlog items that must be implemented in this sprint. Each sprint ends when the selected product backlog items meet the so-called *definition of done*. A product backlog item meets the definition of done when the system passes the validation testing with regards to user stories created for that item as well as other quality requirements, such as maintainability and cognitive complexity. The end of each sprint represents an *increment of value*—that is, a working, self-contained product whose quality does not decrease. The process always keeps all the actors involved at all times: Frequent releases of incremental versions keep the motivation of the developers high and give other stakeholders a chance to validate progresses.

> LUDWIG WITTGENSTEIN
> **The limits of my language mean the limits of my world.**

From a **tooling** perspective, we chose tools that enable mocking, prototyping, and modularization at different levels of granularity. Modularization is a key aspect of agile software development: Software systems must evolve according to evolving business requirements,[23] and modularity makes evolution easier because software components can be understood and changed independently of one another. Most traditional tools for language implementation, such as YACC[24] or antlr,[31] are not well suited to such development; they are designed to produce optimized parsers for languages whose grammar is known ahead of time. Therefore, they do not easily allow the developer to introduce new language constructs in an agile way.

In our approach, modularization is twofold: On top of traditional modules, we also modularize along the dimension of software features. Feature-oriented programming[32] is a development style in which objects are obtained as the composition of several individual services provided by features rather than from traditional object-oriented classes. Feature-oriented programming aims to generalize inheritance to achieve a more granular structure by encouraging the development of independent and reusable services, since the implementation of a module can be composed across several features without them affecting each other. Instead, complex hierarchies and sub-classes are discouraged because traditional modularization techniques may cause code pertaining to the same feature to be scattered across several modules. Feature location—that is, locating the scattered positions of code pertaining to the same feature—is a common activity and a daunting problem.[33] This problem worsens as the project evolves, since features must be located long after their implementation, in a large, often undocumented code base by a different, unfamiliar developer.[13]

Feature-oriented programming in the context of language development is supported by language workbenches,[17] such as MPS,[39] Rascal,[2] Racket,[14] Neverlang,[36] Melange,[11] and Spoofax.[40] To a different extent, each language

a https://www.scrum.org/

workbench makes it possible to componentize the language implementation across both its dimensions: the language-constructs dimension and the compilation-phases dimension. Language workbenches foster reuse of modular language components and can address problems that arise when development is faced in an iterative, incremental way, such as the feature-location problem[33] and the expression problem.[b]

Based on this premise, the marriage between language workbenches and agile software development seems like a promising approach in theory, since language workbenches can stem some of the shortcomings of the agile process. Yet, the application of such an approach in production environments is very limited, so we decided to practice agile language development ourselves. To apply Scrum to language development, we directly map the most important Scrum concepts to feature-oriented language-programming concepts. Each *product backlog item* is represented by a *language feature*[26] expressing the domain concepts implemented as part of the *product goal*. Therefore, the language features are developed as components that are loosely coupled, replaceable, and isolated, so that each product backlog item can easily be traced back to a concrete artifact. Similarly, each *sprint goal* is obtained by extending a DSL obtained as the result of a preceding sprint with all the newly developed language features. According to this approach, DSLs are developed iteratively. Each sprint produces a usable *increment* in which only a subset of the intended features is released—that is, each sprint releases a DSL that will be extended in the subsequent sprints.

Next, we outline the concept and theory of agile language development with Scrum and the Neverlang language workbench to consider this topic from the right perspective. Then, we share our experience to offer insights on problems and solutions regarding the agile development of programming languages with language workbenches in an industrial environment. Our experience shows that agile programming can be a valuable framework to express the problems of their development and to reason about their solutions.

Language Development and Language Workbenches

A programming language is like a breakfast: A good one is the first step toward a nice day. A good breakfast needs the right ingredients and nutrients. Language features that can be separately compiled and distributed are the ingredients of our feature-first design methodology. This recipe is the result of a sentiment that has been widely shared across several researchers in the last decades: There was active interest in the modularization of languages into building blocks and their composition into new interpreters. Language developers came up with many recipes made of several different ingredients.

A famed example is the creation of interpreters using *monads*.[28,35] Monads are algebraic datatypes with a unit that wraps a type into a monadic value and a bind function that can transform monadic values. Monads are commonly used in functional programming but can be generalized to express abstract syntax trees and to interpret them by performing transformations over monadic types. Alternative solutions involve using *mixins* and *traits* to enable a streamlined resolution of feature reuse problems by separating module definitions from their connections.[9,16] For instance, traits can be used to inject semantics over an abstract syntax tree deprived of any behavior so that semantics are completely decoupled from the abstract data representation. Therefore, the syntax and semantics of a language can be separated in different modules.

The general sentiment behind all these approaches to language development is the realization that language-feature reuse can be generalized to both the constructs (syntax) and the behavior (semantics) dimensions. Modularizing along the dimension of constructs makes it possible for different language designs to share parts of the syntax with other languages. For instance, some constructs (along with their semantics) can be excluded to implement a restricted DSL, or one may want to keep the implementation of the semantics of a given construct while replacing its syntax. On the other hand, a modularization along the dimension of compilation phases makes it possible to reuse and substitute parts of the *semantics* of a language implementation. For instance, a compiler and an interpreter for the same language might share most of their code, such as the parser, the development environment, type checkers, and optimizers.

In summary, different recipes can share ingredients with one another and languages can leverage this opportunity by performing modularization along *both* dimensions. To this goal, a language implementation should distinguish between static semantic phases (for example, type checking), dynamic semantic phases (for example, evaluation) and language constructs, in such a way that the syntax of a language can be freely extended and restricted, and the semantics can be varied at will with limited changes to the original recipe.

Arguably, this vision is fully realized by *language workbenches*, in which the dimensions of language implementation can be expressed through powerful abstractions. Racket is an extensible language in which programmers can use powerful macros to implement their own DSLs as libraries. MPS is a language workbench that uses a *projectional editor* to visualize abstract syntax trees in a user-friendly way through dedicated views; developers can also implement their own DSLs, either from scratch or from other previously implemented languages. Spoofax is a modular language workbench that focuses on separation of *linguistic concerns*—that is, aspects of the programming language. These language workbenches are just some of many examples in which this vision is fully realized; each language workbench brings forth its own recipe for language decomposition and reuse. Our contributions to this research field are the Neverlang[36] language workbench and a Neverlang-based recipe for language decomposition.[4] In this work, we chose the Neverlang language workbench due to its unique approach to modularization, which makes it particularly suited for use in an agile context. Other

b As communicated by P. Wadler in the Java genericity mailing list in November 1998.

workbenches may reach similar results but lack some key features, such as separate compilation, fully exogenous composition, and native support for language product lines. In fact, Neverlang differs from other language workbenches by explicitly embracing the feature-oriented programming paradigm[32] and language product lines.[26] In Neverlang, language features are promoted to first-class citizens. This abstraction fits an agile style of development nicely because each product backlog item can be mapped to exactly one Neverlang language feature.[c] In summary, the Scrum team can trivially measure the increment of value in terms of the number of Neverlang linguistic assets created during each sprint. Similarly, refactoring a product backlog item is easy because its code is not scattered and can be traced back to a specific Neverlang source file.

Figure 1. Applying the Scrum framework through an iterative language engineering process.

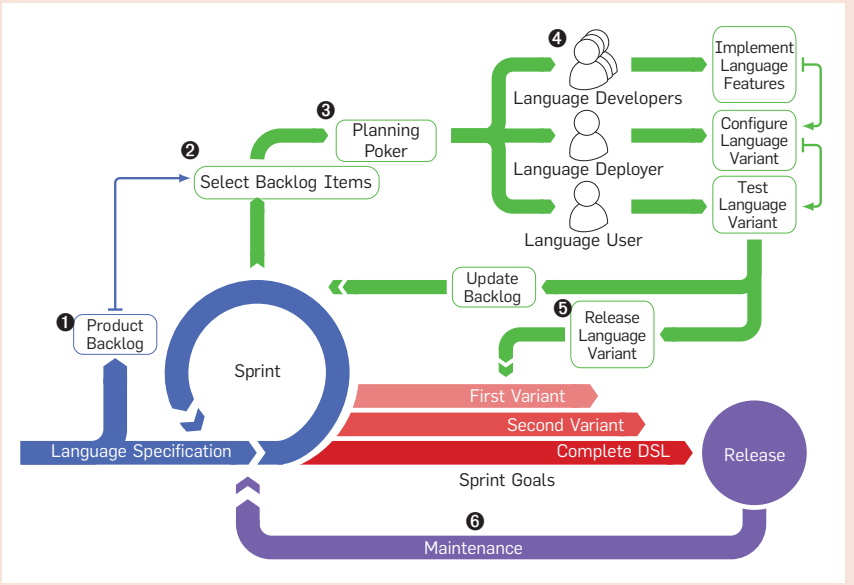
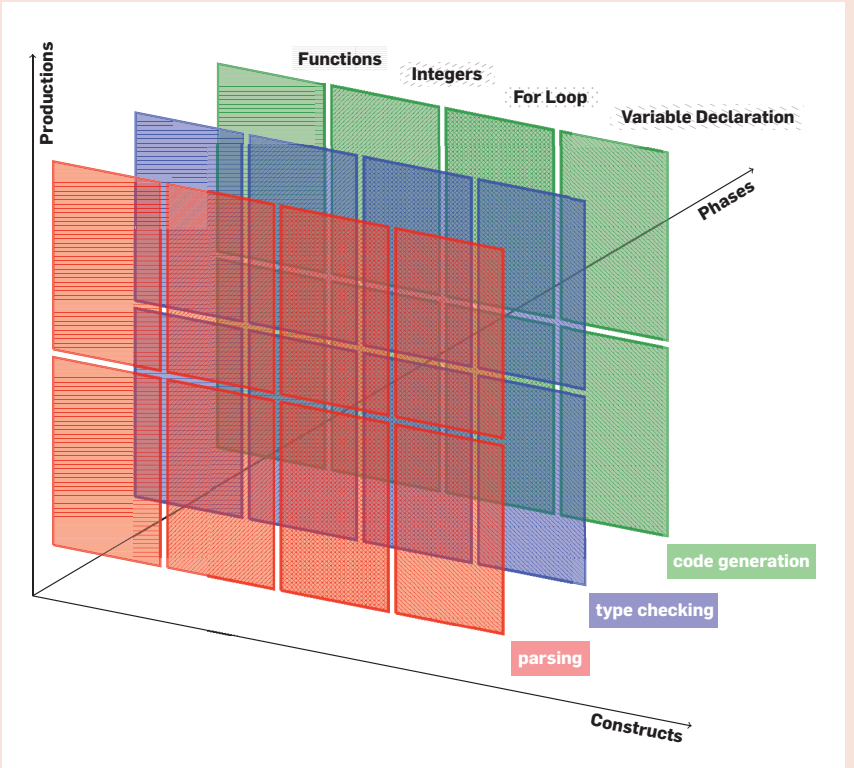


Figure 2. Dimensions of the language decomposition: syntactic constructs (differ by pattern) and compilation phases (differ by color).



Agile Language Development

It is impossible to foresee all the requirements of a project at the beginning. To favor adaptive iterations over long-term estimations and to promptly address changes in requirements, we outlined a development process for programming languages based on Scrum, dubbed *agile language development*. The overall agile language development process is shown in Figure 1.

Product goal. We discuss the agile language development process through the lens of our case study conducted on Tyl,[d] in which the product goal was a DSL for the development of ERP applications commissioned by the eponymous software company.[e] ERP applications are complex software suites designed to manage the many stages of a business, such as sales, inventory management, shipping, and payment. Our partnership with Tyl enabled us to put theory into practice in an industrial production environment. Tyl is a programming language designed to flatten the learning curve of unexperienced programmers and domain experts, so they can be involved in the development of ERP applications with limited training. Tyl was designed as a Java-like language that trades generality for a set of business-specific features, such as built-in datatypes for currencies and an embedded query language. The DSL revolves around user interaction with

c Please refer to Vacchi and Cazzola[36] for a complete Neverlang overview. The mathematical models behind the modularization used by Neverlang for the syntax and the semantics are discussed in Cazzola and Vacchi[8] and Cazzola et al.[5] respectively.

d Visit https://doi.org/10.5281/zenodo.7276536 to further inspect an excerpt of the project (which was simplified for demonstration purposes), including each of the DSL variants obtained as the result of a sprint goal.

e A small Italian software company dedicated to enterprise resource planning (ERP), software development, and integration.

forms: Programmers are given default implementations of standard operations (such as navigating or saving the contents of a form), called *events*. Tyl can then be used to customize the effect of these events. As shown in Figure 1-❶, the Tyl specification was used to set the product goal and then split into product backlog items to populate the initial product backlog. Each product backlog item represents a different language feature to be implemented in Neverlang. Next, the first sprint started.

Getting started. During each sprint, the Scrum team sets a language variant of the product goal as the current sprint goal and selects the corresponding items from the product backlog (Figure 1-❷). The sprint goal for the first sprint was a proof-of-concept language for the evaluation of arithmetic expressions. The Scrum team estimates the effort needed to implement the selected backlog items through a planning poker event[19] (Figure 1-❸). As a result of the planning poker event, the team is split into three roles (Figure 1-❹). One team member is chosen as the language user, one team member is chosen as the language deployer, and all remaining team members are selected as language developers. Depending on the size of the Scrum team and on the duration of each sprint, some team members may be assigned more than one role. Once the roles have been established and until the end of the sprint, each member behaves according to an engineering process we outlined in a previous work.[4] In this process, decisions are made iteratively based on what is observed during development to optimize separation of concerns, task parallelism, and adaptability of the requirements. More precisely, language developers are responsible for developing language features using Neverlang. The language deployer configures the available language features into a language variant according to the sprint goal. The language user translates the user stories reported in each product backlog item into test cases. Tests are used to support the creation of language features in a test-driven development fashion and to verify the validity of the language implementation before release.

Listing 1. Modular addition implementation in Neverlang.

```
module neverlang.commons.staging.expr.AddExpr {
  reference syntax {
    AddExpr <- Term;
    AddExpr <- AddExpr "+" Term;
  }
}
module neverlang.tyl.compiler.AddExpr {
  reference syntax from neverlang.commons.staging.expr.AddExpr
  role(type-checking) {
    0 .{
      $0.value = $1.value; $0.position = $1.position;
    }.
    2 .{
      Class c1 = $3.value; Class c2 = $4.value;
      TylTypeCompatibilityTable tct = $$TylTypeCompatibilityTable;
      try {
        $2.value = tct.get(c1.getName(),c2.getName(),"+");
      } catch (TylUnsupportedOperationException ex) {
        $$Errors.add(
          TylUnsupportedOperationException(
            #0.col,#0.row,ex.getMessage()));
      }
      $2.position = $3.position;
    }.
  }
  role(code-generation) {
    0 .{ $0.code = $1.code; }.
    2 .{ $2.code = "%s + %s".formatted($3.code, $4.code); }.
  }
}
slice neverlang.tyl.compiler.AddExprSlice {
  concrete syntax from neverlang.commons.staging.expr.AddExpr
  module neverlang.tyl.compiler.AddExpr with role type-check
  module neverlang.tyl.compiler.AddExpr with role code-generation
}
```

Completing product backlog items. Using Neverlang, language developers can implement language features selected as product backlog items for the current sprint using abstractions that explicitly model reuse of both syntactic and semantic dimensions. Compared to other language development frameworks, we found Neverlang's abstractions to be particularly suited to an agile development approach, since components can be prototyped and then incrementally replaced by their final implementation by composing modules into new language features. Figure 2 schematizes the modularization techniques used in Neverlang, whereas Listing 1 shows a product backlog item implemented by a language developer during the first sprint according to the modularization depicted in Figure 2: A language implemented in Neverlang is decomposed along the two dimensions of syntactic constructs and of compilation phases. Figure 2 shows that Tyl was decomposed into four constructs (x axis) and three phases (y axis). The z axis represents different grammar productions pertaining to the same syntactic construct. Each production is represented by a box, with a different color depending on the semantic dimension—that is, the compilation phases. For instance, Tyl is divided into a parsing phase (red), a type-checking phase (blue), and a code-generation phase (green).

The implementation of addition expressions in Tyl shown in Listing 1 mirrors this modularization technique. The code is split into modules, each containing a **`reference syntax`** and

zero or more **roles**. In this example, the **reference syntax** (red box) declares all the productions of the addition expression language construct in Backus-Naur Form. The **reference syntax** will be used by Neverlang to generate a parser. The `type-checking` role (blue) shows the implementation of a compilation phase that performs type checking over addition expressions written in Tyl, by adding any detected error to a global data structure called `$$Errors`. Roles are implemented in Java, with additional syntactic sugar for accessing nonterminals. Nonterminals are referenced with an absolute numbering that starts from 0 and grows left to right and top to bottom. For instance, the red arrows in Listing 1 show that 0 is used to refer to the `AddExpr` nonterminal in the first production, 1 to refer to the `Term` in the first production, 2 to refer to the `AddExpr` nonterminal in the second production, and so on. Accessing nonterminals enables the generation and retrieval of attributes stored in nodes of the parse tree (such as `$1.value` in Listing 1) according to the syntax directed translation technique.[1] Similarly, the `code-generation` role (green box) implements the semantics to translate Tyl code into Java according to the value of the code attributes on child nodes of the parse tree.

Finally, all these elements are composed together by the **slice** construct (black box in Listing 1), in which the syntactic dimension and all semantic dimensions of a language feature are listed; dimensions can be implemented in either the same or different modules. This modularization technique fits the requirements of agile development and adapts them to the scope of language development. In Neverlang, each product backlog item is implemented as a **slice**. Therefore, if the team must face a change in the requirements, it suffices to detect the backlog items that need to be changed, remove the corresponding slices from the language implementation, and replace them with the updated slices that conform to the new requirements. Moreover, the mechanism is flexible enough that parts of a slice that do not need to be changed (for instance, the **reference syntax**) can be reused in new slices without changes nor code duplication. Similarly, mocking and prototyping are cost effective, because mocked slices can be easily replaced by final ones once they are available.

> ALFRED V. AHO
>
> **Computer science is a science of abstraction—creating the right model for thinking about a problem and devising the appropriate mechanizable techniques to solve it.**

Completing a sprint. The development process of Neverlang slices is not linear and the three roles may interact several times before language features can be considered done. The definition of done is agreed upon by the Scrum team and stakeholders before beginning the first sprint and generally includes conformance to user stories and quality metrics.[4] When the language user declares all the language features that meet the definition of done, the sprint goal has been met and the sprint ends. In this example, the first sprint ended as soon as the language was able to parse well-formed expressions. As shown in Figure 1-❺, before beginning the next sprint, the language variant is released. This differs considerably from the traditional approach to language development. Whether the development process follows a waterfall or an iterative model, intermediate products are invalidated upon completion of the next version. Conversely, deploying and maintaining several fully functioning language variants at the same time provides continuous validation of the reusability of linguistic assets across several scenarios. Moreover, there might be scenarios in which having several language variants may prove useful, such as the teaching programming activity.[6] Finally, the product backlog is updated by removing any completed product backlog item and adapted to any changes to the requirements.

Iterating the process. On each subsequent sprint, different team members may (and should) play the roles of language deployer and language user. When a sprint goal coincides with the product goal, the process is complete, and the complete DSL can be released. However, the agile development process never really ends and can be resumed at any time to maintain and update the product, as well as for any post-release change of the requirements (Figure 1-❻).

The development of Tyl consisted of six sprints; the accompanying table summarizes the product backlog

items completed and/or refactored on each sprint, as well as each DSL variant released when the respective sprint goal was reached. As discussed so far, the first sprint resulted in a proof-of-concept release. Similarly, the sprint goal for the second sprint was a proof-of-concept language that extended the first release with variables and control flow structures. The releases from the third on were more feature-rich and began including ERP domain-specific concepts. The six sprints can be logically split into three groups: **proof of concept** (sprints #1 and #2), **code structure** (sprints #3 and #4), and **domain-specific features** (sprints #5 and #6). The modular implementation allows for iterative evolution through the gradual introduction of new and updated features via the implementation of new Neverlang slices. On each sprint, the domain-analysis phase was performed with the help of the domain experts from Tyl. The sprint goals for sprints #3 and #4 focused on code structure: New slices were added and some existing components were replaced to support functions and their namespaces. Sprint #3 (*Function Libraries*) introduces constructs for function definition and invocation; a new function table component tracks the scope so that the *function-check* compilation phase could validate function identifiers. This phase is considered orthogonal to all others and was implemented separately. Finally, we swapped the mocked *symbol table* slice used for the proof-of-concept with a scope-aware implementation to reach the new sprint goal. Sprint #4 was devoted to support *Forms and Events*. In Tyl, a prototype describes the factory behavior of a form through a set of *events*. An example for such an event written in Tyl is shown in Listing 2. The framework (developed in parallel by a separate Scrum team) routes and dispatches events in response to user interaction. Using events written in Java was a viable solution, although the domain experts considered Java too verbose and obscure for non-programmers. Using a DSL such as the one exemplified in Listing 2 to express events allowed for a smoother development experience by reducing the boilerplate code and replacing any API with a domain-specific syntax. Instead of directly interacting with the properties of a *context object*, developers can interact with these properties using simple assignments. For instance, the developer can write `this.fields.username.value` instead of `ctx.getField("fields").getField("username").getValue()` on line 5 of Listing 2, with the added benefit of performing a sanity check, whereas the corresponding Java code would fail at runtime. In fact, the language variant released for this sprint

Listing 2. Programmable events in Tyl.

```
1  form HelloForm : SimpleForm {
2    property message = "Hello, ";
3    event OnSave {
4      message (
5        this.message + this.fields.username.value
6      ); /* displays "Hello, <username>" */
7    }
9    event OnClsoe { /* ... */ }
10   /* This typo (OnClose -> OnClsoe) raises a compiler error *
11    * (it does not conform to the prototype)                 */
12   field username {
13     event OnChange {
14       log("username changed to " + this.value);
15     }
16   }
17 }
```

Sprints of the agile development process of Tyl.

#	Sprint Goal	Total Backlog Items	New Backlog Items	Updated Backlog Items	Key Features	Compilation Phases
1	Calculator	10	2 (mostly off-the-shelf components)		Arithmetic Expressions Basic Types	Translation Basic Type Checking
2	Control Flow	18	8	1	Variables Control Flow Symbol Table	Basic Type Checking Code Generation
3	Function Libraries	31	13	6	Function Definition Invocation and Library Definition Function Table Updated Symbol Table (with Local Scope)	Basic Type Checking Function Check Code Generation
4	Forms and Events	39	8	4	Form Event Field Property Definition	Extended Type Checking Function Check Code Generation
5	Business-Specific Types	44	5	7	Date/Time Currency Fixed-Point, and so on	Extended Type Checking v2 Function Check Code Generation
6	Query DSL	49	5 + support code (SQL-like DSL select, update, and so on.)	10 (all nullable operations)	Nullable types Null-safe expression	DB Schema Checking Extended Type Checking v3 Function Check Code Generation

introduced a new *extended type-checking* compilation phase and all Neverlang slices needed for its implementation. This version validates a form implementation against its prototype: The compiler would raise an error whenever a programmer tries to implement an undeclared event. The last two sprints focused on the introduction of the business-oriented features. Most notably, Sprint #5 introduced built-in types to represent date/time values, currencies, and in general, all of the types that are deemed useful in business-oriented applications. Arithmetic operations between the new datatypes are native. For instance, "2014.02.01 - 2014.01.01" should evaluate to 31 days. The compiler also performs type checking and automated conversions whenever types are compatible for the given operation. After Sprint #6, the product goal was met and Tyl was ready to be released. On each sprint, some Neverlang slices had to be replaced by new ones to keep up with the agile introduction of new language features, but the associated effort was reasonable because it was easy to detect the affected language components and replace them with updated ones without affecting the implementation of other components.

Listing 3. Handling nullable types in Tyl.

```
1  form PersonForm : SimpleForm {
2    event OnSave {
3      var currentYear: int = 2014;
4      select age from Person where id == 123 {
5        event Each(record) {
6          var i: int = record.age;
7          // causes an error
8          var i: int = record.age default -1;
9          // assigns -1 when record.age is null
10         var age: int? = record.age;
11         // null-safe expressions propagate null
12         var birthYear: int? = ?( currentYear-age );
13       }
14       event Error(code) {
15         /* recover from DB error */
16       }
17     }
18   }
19 }
```

Overview of Agile Support in Language Workbenches[a]

Spoofax[40] decouples syntax and semantics. The same semantic strategies can be reused across several language features by desugaring different concrete syntaxes into the same abstract syntax by means of rewrite rules.

MontiCore[25] languages can be iteratively extended via syntax and semantics redefinition. Sub-languages can be prototyped through *embeddings*.

LISA[20] can combine and iteratively extend languages by means of *multiple inheritance* and aspect-like constructs.

Silver[37] decouples syntax and semantics using attributes, so that the semantics are executed over an AST rather than a parse tree, thus removing any coupling between semantics and any irrelevant details such as terminals.

CBS[30] provides an extensive library of off-the-shelf *language-specification components* for fast prototyping of new languages.

Meta-Programming System (MPS)[39] supports concurrent views of the same AST so that programmers with different expertise can collaborate.

Melange[10] uses aspects to allow language extension both in the dimension of language constructs and of semantic phases.

Racket permits programmers to iteratively add languages to a codebase so that extra-linguistic mechanisms are turned into linguistic constructs.[14]

Rascal[2] modules can import, extend, and merge other modules with semantics that can be reused via a *pattern-based dispatch* mechanism.

a This is not exhaustive. For a full comparative survey on language workbenches, please see Iung et al.[21] and Vasudevan and Tratt.[38]

Facing challenges. The original specification of the Tyl DSL made some bold assumptions and design decisions. For instance, it was decided to remove any null references altogether to prevent unexpected crashes and vulnerabilities. During the sprint planning event for Sprint #6, the product backlog items selected for the sprint goal included a SQL-like DSL to be embedded in Tyl, inspired by LINQ, JOOQ and QueryDSL. This is when we realized a language that interacts with databases cannot simply ship without null values and language specification of the Tyl DSL had to be updated accordingly. A traditional waterfall model would not support such a sudden change of direction at such a late stage of development (the very last sprint before the final release). Conversely, an agile development process allowed for a progressive replacement of all the involved components while verifying that the rest of the implementation would not regress. The product backlog items needed for the introduction of null values were added to the product backlog and then selected as part of the following sprint goal. Inspired by modern programming languages such as Kotlin, the new user stories were written so that null values must be handled in a safe way. All type were kept non-nullable by default but were supported by the introduction of *nullable types*: should the programmers interact with data from an external source, they can explicitly declare it to be *null-unsafe* (see Listing 3). The newly introduced product backlog items included constructs

for the support of native `null`-aware arithmetic and the safe propagation of `null` values without throwing exceptions. This is where a language workbench that promotes language features to first-class citizens shines: Neverlang slices that did not conform to the new requirements could be easily detected based on the product backlog items completed during the preceding sprints. These slices could then be replaced by alternative implementations without affecting other slices due to their code not being tangled. Effectively, we could steer the language implementation in a different direction at a very late stage of the development.

Additional development challenges arose from the requirement of enforcing implicit conversions between business-critical, non-conventional datatypes such as timestamps, currencies, and units of measurement. Adding new datatypes in the context of agile development of programming languages involves resolving the well-known expression (or extensibility) problem, which is now a classic problem in programming languages. In fact, each sprint may introduce new datatypes that were not originally foreseen. Each new datatype represents a new data variant and requires the introduction of new operations.

Both new data variants and new operations may affect pre-existing ones. In our context, a new datatype must be supported by operations to perform type checking and type promotion against pre-existing datatypes. Solving the expression problem in our context means supporting the introduction of new datatypes and any additional type-checking and type-promotion semantics without affecting existing Neverlang slices. The Neverlang modularization technique[3,4] was capable of solving the expression problem because new type-checking semantics and type conversions can be implemented in the roles of a new module and compiled separately by leveraging a technique called *restraint semantic dispatch*.[7] Tyl developers smoothly introduced business-specific datatypes and the ability to define custom new types based on existing ones through a code-generation tool and JSON templates. Each new type implements any needed roles and can automatically be plugged-in, converted and promoted to other existing types without changing their code.

> TONY HOARE
> **I call it my billion-dollar mistake. It was the invention of the null reference in 1965.**

A Perspective on Agile Language Development

How easy is it to adopt an agile language development process? What are its benefits and drawbacks? How does it compare to traditional language development? In other words, what is the lesson we learned from this experience?

In our experience, agile language development is a process that is easy to pick up and start practicing. It provides smaller, incremental goals as well as constant feedback to the development team and stakeholders. However, the actual application of agile language development still has its fair share of challenges, including how to deal with the complex inter-relations between language features traditionally implemented as part of a monolithic compiler or interpreter. Tackling such problems requires adequate tools, giving developers the ability to realize the features that constitute a language variant in isolation. In a sense, agile language development is possible only if the language development framework supports a modularization technique that supports modules and features as distinct entities with low coupling, so that language features can be replaced with limited impact on the rest.

All language workbenches can empower such a development style, although different approaches to modularization may fit the agile development process better. So, agile language development is not only a matter of methodology and tooling, but also a matter of design. It requires developers to carefully design each language feature and for the language workbench to have the ability to compose language features in a strictly exogenous way,[29] so that all language features are unaware of the language they will be part of. Instead, any gap between the semantics of incompatible language features is filled by the language deployer at configuration time using only glue code that is external to the implementation itself. Such a development style is like the infamous monkey-patching technique often used by JavaScript browser plugins. However, Neverlang slices improve

on the limitation of monkey patches by not breaking modular boundaries: Slices can accommodate the composition between Neverlang modules, but they can never access their implementation. Moreover, Neverlang slices can never cause any problems associated to a dependency to a monkey-patched module: Neither slices nor modules can depend on another slice, therefore each patch is always local to a specific language feature. Were the same modules to be used in a different slice, the patch would not propagate.

To summarize, the repeatability of our experience and the applicability of agile language development are limited with regards to the capability of the language workbench to exogenously compose language features using only glue code. Moreover, developers must be given the ability to determine if their language-feature implementation is truly modular—during each sprint and in real time. Although finding an objective solution to this problem may be a complex task, the developer can be supported by defining the software metrics that best suit the language workbench and their respective target values, such as cohesion, coupling, complexity, and maintainability metrics.[4] Achieving a true modular structure is a key factor: Reuse of off-the-shelf components is a staple of agile development, either for prototyping, mocking, or for actual implementations and is supported in different ways by several language workbenches. For instance, Xtext provides a generic expression language (Xbase) that can be imported as a library by other languages, whereas MPS provides an extensible BaseLanguage. Racket is entirely designed around the concept of extending the language by creating extra-linguistic mechanisms and turning them into linguistic constructs[15] to support faster problem solving. As it is often the case in the context of software development, agile language development is not a silver bullet and cannot undo the inherent complexity of software systems,[34] but it can provide a framework to reason about complex systems in terms of their application domains and the tools to easily develop the languages that are best suited to express the problems and the solutions of each particular domain without reliance on long-term timelines and hard-to-maintain documents.

Acknowledgments

The authors thank D. Zonca, E. Vacchi, and the other staff of Tyl for their help and feedback in the various sprints. They also thank the anonymous reviewers for their valuable comments and suggestions, which greatly improved the presentation of our work. This work is partially supported by the Italian Ministry of University and Research (MUR) project "T-LADIES" (PRIN 2020TL3X8X). C

References

1. Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, USA, 2nd ed. (2006).
2. Basten, B. et al. Modular language implementation in Rascal—Experience report. *Science of Computer Programming 114* (Dec. 2015), 7–19.
3. Bertolotti, F., Cazzola, W., and Favalli, L. On the granularity of linguistic reuse. *J. Systems and Software* (April 2023).
4. Cazzola, W. and Favalli, L. Towards a recipe for language decomposition: Quality assessment of language product lines. *Empirical Software Engineering 27*, 4 (Apr. 2022).
5. Cazzola, W., Giannini, P., and Shaqiri, A. Formal attributes traceability in modular language development frameworks. *Electronic Notes in Theoretical Computer Science 322* (Apr. 2016), 119–134.
6. Cazzola, W. and Olivares, D.M. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing 4*, 3 (Sept. 2016), 404–415.
7. Cazzola, W. and Shaqiri, A. Modularity and optimization in synergy. In *Proceedings of the 15th Intern. Conf. on Modularity*, D. Bator (Ed.), ACM (March 2016), 70–81.
8. Cazzola, W. and Vacchi, E. On the incremental growth and shrinkage of LR Goto-Graphs. *Acta Informatica 51*, 7 (Oct. 2014), 419–447.
9. Cazzola, W. and Vacchi, E. Language components for modular DSLs using traits. *Computer Languages, Systems & Structures 45* (Apr. 2016), 16–34.
10. Combemale, B., Barais, O., and Wortmann, A. Language engineering with the GEMOC Studio. In *Proceedings of the Intern. Conf. on Software Architecture Workshop*, IEEE (Apr. 2017), 189–191.
11. Degueule, T. et al.Melange: A meta-language for modular and reusable development of DSLs. In *Proceedings of the 8th Intern. Conf. on Software Language Engineering*, ACM (Oct. 2015), 25–36.
12. Dijkstra, E.W. The humble programmer. *Communications of the ACM 15*, 10 (Oct. 1972), 859–866.
13. Dit, B. et al. Feature location in source code: A taxonomy and survey. *J. of Software: Evolution and Process 25*, 1 (Jan. 2013), 53–95.
14. Felleisen, M. et al. A programmable programming language. *Communications of the ACM 61*, 3 (March 2018), 62–71.
15. Felleisen, M. et al. The Racket Manifesto. In *Proceedings of the 1st Summit on Advances in Programming Languages and Informatics 32*, S. Krishnamurthi and G. Morrisett, (Eds.), May 2015, 113–128.
16. Findler, R.B. and Flatt, M. Modular object-oriented programming with units and mixins. In *Proceedings of the 3rd Intern. Conf. on Functional Programming*, M. Felleisen, P. Hudak, and C. Queinnec, (Eds.), ACM (Sept. 1998), 94–104.
17. Fowler., M. Language workbenches: The killer-app for domain specific languages? Martin.Fowler.com (May 2005).
18. Graham, P. Beating the averages. http://www.paulgraham.com/avg.html, (Apr. 2003).
19. Grenning, J. Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting 3*, (Aug. 2002), 22–23.
20. Henriques, P.R. et al. Automatic generation of language-based tools using the LISA system. *IEEE Proceedings—Software 152*, 2 (Apr. 2005), 54–69.
21. Iung, A. et al. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering 25*, 5 (Sept. 2020), 4205–4249.
22. Iverson, K.E. Notation as a tool of thought. *Communications of the ACM 23*, 8 (Aug. 1980), 444–465.
23. Jayatilleke, S. and Lai, R. A systematic review of requirements change management. *Information and Sofware Technology 93*, (Jan. 2018), 163–185.
24. Johnson, S.C. YACC: Yet another compiler-compiler. Technical Report, (CS-TR-32), Bell Laboratories, Hill, NJ, USA (July 1975).
25. Krahn, H., Rumpe, B., and Völkel, S. MontiCore: A framework for compositional development of domain specific languages. *Intern. J. on Software Tools for Technology Transfer 12*, 5 (Sept. 2010), 353–372.
26. Kühn, T. and Cazzola, W. Apples and oranges: Comparing top-down and bottom-up language product lines. In *Proceedings of the 20th Intern. Software Product Line Conf.*, R. Rabiser and B. Xie (Eds.), ACM (Sept. 2016), 50–59.
27. Kühn, T., Cazzola, W., Giampietro, N.P., and Poggi, M. Piggyback IDE support for language product lines. In *Proceedings of the 23rd Intern. Software Product Line Conf.*, ACM (Sept. 2019), 131–142.
28. Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM Symp. on Principles of Programming Languages*, R.K. Cytron and P. Lee (Eds.), ACM (Jan. 1995), 333–343.
29. Méndez-Acuña, D. et al. Leveraging software product lines engineering in the development of external DSLs: A systematic literature review. *Computer Languages, Systems & Structures 46* (Nov. 2016), 206–235.
30. Mosses, P.D. Software meta-language engineering and CBS. *J. of Computer Languages 50* (Feb. 2019), 39–48.
31. Parr, T.J. and Quong, R.W. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience 25*, 7 (July 1995), 789–810.
32. Prehofer, C. Feature-oriented programming: A fresh look at objects. In *Proceedings of the 11th European Conf. on Object-Oriented Programming*, M. Akşit and S. Matsuoka (Eds.), Springer (June 1997), 419–443.
33. Rubin, J. and Chechik, M. A survey of feature location techniques. *Domain Engineering: Product Lines, Languages and Conceptual Models*, I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, (Eds.), Springer (2013), 29–58.
34. Saffer, D. *Designing for Interaction: Creating Innovative Applications and Devices*. New Riders (2010).
35. Steele, G.L. Building interpreters by composing monads. In *Proceedings of the 21st Symp. on Principles of Programming Languages*, H-J. Boehm, B. Lang, and D. Yellin (Eds.), ACM (Jan. 1994), 472–492.
36. Vacchi E. and Cazzola, W. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures 43*, 3 (Oct. 2015), 1–40.
37. Van Wyk, E. et al. Silver: An extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science 203*, 2 (Apr. 2008), 103–116.
38. Vasudevan, N. and Tratt, L. Comparative study of DSL tools. *Electronic Notes in Theoretical Computer Science 264*, 5 (July 2011), 103–121.
39. Völter, M. and Pech, V. Language modularity with the MPS Language Workbench. In *Proceedings of the 34th Intern. Conf. on Software Engineering*, IEEE (June 2012), 1449–1450.
40. Wachsmuth, G.H., Konat, G.D.P, and Visser, E. Language design with the Spoofax Language Workbench. *IEEE Software 31*, 5 (2014), 35–43.

Walter Cazzola (cazzola@di.unimi.it) is a professor at the Università degli Studi di Milano, Computer Science Department, Milan, Italy.

Luca Favalli is a post-doctoral fellow Università degli Studi di Milano, Computer Science, Milan, Italy.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/scrambled-features-breakfast