Building White-Box Abstractions by Program Refinement

Mehrdad Afshari Zhendong Su

Department of Computer Science, University of California, Davis, USA

{mafshari, su}@ucdavis.edu

Abstract

Abstractions make building complex systems possible. Many facilities provided by a modern programming language are directly designed to build a certain style of abstraction. Abstractions also aim to enhance code reusability, thus enhancing programmer productivity and effectiveness.

Real-world software systems can grow to have a complicated hierarchy of abstractions. Often, the hierarchy grows unnecessarily deep, because the programmers have envisioned the most generic use cases for a piece of code to make it reusable. Sometimes, the abstractions used in the program are not the appropriate ones, and it would be simpler for the higher level client to circumvent such abstractions. Another problem is the impedance mismatch between different pieces of code or libraries coming from different projects that are not designed to work together. Interoperability between such libraries are often hindered by abstractions, by design, in the name of hiding implementation details and encapsulation. These problems necessitate forms of abstraction that are easy to manipulate if needed.

In this paper, we describe a powerful mechanism to create *white-box abstractions*, that encourage flatter hierarchies of abstraction and ease of manipulation and customization when necessary: program refinement.

In so doing, we rely on the basic principle that writing directly in the host programming language is as least restrictive as one can get in terms of expressiveness, and allow the programmer to reuse and customize existing code snippets to address their specific needs.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Miscellaneous; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords white-box abstractions, program refinement, syntactic manipulation, reusability

Onward!'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4076-2/16/11...$15.00
http://dx.doi.org/10.1145/2986012.2986025

1. Introduction

Programming is about expressing ideas. Ideas expressed in programs vary widely in complexity. Programs can exhibit small ideas or very complex ones. Complex ideas are built from simple ones. There are three ways to build complex ideas from simple ones: by combining them into a compound one, by comparing them with each other without unifying them, and via abstraction, *i.e.* distancing them from other ideas that accompany them in their concrete existence [14].

Complex programs, like ideas, are generally composed of smaller pieces. In order to make building complex systems tractable, and to be able to reuse these smaller pieces in different contexts, we need to rely on abstraction. Programming languages provide various ways to build abstractions, like procedures [3], abstract data types [13], classes [6], objects [7], and actors [8]. Different programming languages provide different means of abstraction.

In practice, software systems often consist of complex abstractions composed of other complex abstractions, forming a deep hierarchy of abstractions, created by different people at different times to achieve different goals. While some of the nodes in this hierarchy are essential to the program, a deep hierarchy of abstractions has some obvious downsides.

One issue is imperfections of the abstractions themselves, *i.e.* they do not fully abstract away the related ideas that accompany the underlying concrete instantiations of the abstraction and the subtleties *leak* to the higher level observer, making it responsible for specifically working around such leaks, thereby hindering the reusability of the abstraction in arbitrary contexts. A library abstracting a TCP-based network connection as if it were a local in-memory stream might limit the caller who may also need to access

the network-specific state and manually tweak flushing the stream and traffic flow parameters that may not necessarily be exposed via an explicit API, but nevertheless affect the caller's expectations, making it distinguishable from a local stream a disconnection event, for instance.

Another concern is understandability of programs relying heavily on complex abstractions: by design, many of the language features and techniques to build abstractions, *e.g.* procedural abstraction and object orientation, aim to build *black-box* abstractions. Black-box abstractions are double-edged swords. The advantage of hiding the internals is that the component can be isolated and reasoned about as a separate unit with clear interfaces and boundaries. The disadvantage is that the interfaces can be arbitrary and lacking documentation, or worse, having incorrect documentation that does not perfectly reflect the subtleties of the implementation, causing confusion for the programmer. Anecdotally, sometimes reading the source code for the component, if available, can be the best path for understanding the subtleties of the implementation. Deep abstraction hierarchies can make this more difficult.

1.1 Abstraction by Refinement

In this paper, we introduce a new mechanism for building abstractions: *program refinement*. In particular, we capture the programmer's intent by tracking modifications to pieces of a program. A modified piece of code is implicitly considered a specialized verion of the original, inducing a relationship between the two, not too dissimilar to inheritance in object-oriented programming, but in a static syntactic fashion.

The core idea is treating a certain procedure (β) as the *base* template and letting the programmer modify it as they wish. The new, specialized, procedure (Γ) can be formally described of as a pair consisting of the original base procedure and its differences (Δ).

$$\Gamma = (\beta, \Delta)$$

Δ is the refinement applied by the programmer and captures the intent of the specialization on the base template. The way Δ is interpreted is implementation-dependent. In the simplest implementations, it can be a syntactic difference provided by a version control tool.

For it to be a proper abstraction, we need to be able to liberate Δ from being meaningful only in the context of that particular base, and be able to apply it to other base templates as well, computing a new specialization with the same delta over a new base template. This is formalized by a *merge* operation:

$$\Gamma_2 = merge(\Gamma, \beta_2) = merge((\beta, \Delta), \beta_2)$$

The actual behavior of a merge operation is also implementation dependent. In the simplest case, it is a version control-like syntactic automerge, but it can be made smarter and more semantic-oriented. The smartness of the merge operation is, in a way, representative of how capable the programming system is in capturing the programmer's intent.

A single base can serve as the template for many specializations. There can, in principle, be a nested tree of specializations. When the root base changes, the changes would propagate by applying successive merge operations. The merge operation can be unsuccessful. We will leave the discussion on how we resolve this issue to section 3.

We have used this simple formalization to describe the idea as an analogy to another known idea, and the differences between the two.

At this point, the description might sound similar to a macro system, or a template metaprogramming feature in a language like C++. Subtle, but key, differences, however, exist, as we describe.

Refinement vs. Macros Macros are powerful abstraction tools. Similar to refinement, they offer specialization from a symbolic template. Macros, in languages that embrace them, like flavors of Lisp, operate at the abstract syntax tree level, and therefore make the full power for the underlying language accessible to the programmer. However, there are two key differences:

1. Macro expansions are evaluated in the scope of the use site. In this manner, they are similar to copy-pasted code. A key feature in refinement abstractions is evaluation within the environment of the original base template.
2. Macros need to be predefined. A programmer generally needs to think beforehand about what macros to write, and provide appropriate "holes" in them for the external arguments. The programmer would often overgeneralize the macro before the complexity is actually needed in the program. The opposite can also occur, where the macro definition does not support parameterization of certain parts of itself. Clearly, arbitrary procedures do not become macros automatically, but you can apply refinement to any procedure in the program, without any special consideration when the procedure is being authored.

Refinement vs. Metaprogramming Templates Metaprogramming template systems vary in design and function. To address the differences, we take C++ template system as a popular, concrete instantiation. In contrast with the C++ templates:

1. Refinements, being syntactic, are constrained by the expressiveness of the host language only. Templates, however, can only be parameterized in certain areas. An arbitrary statement cannot be fed into a C++ template as an argument. The parameterization potential is usually limited to types, values, and function references.
2. Refinements can be applied to any base procedure, whereas templates, like macros, need to be predefined as such.
3. Depending on the way the template system is implemented, its expansions can exhibit the second limitation described for macros, *i.e.* redefinition of the environment in which the template is expanded.

1.2 Main Contributions

This paper makes the following contributions:

- We introduce a new general paradigm for building abstractions by allowing the programmers to refine existing code.
- We present and discuss the design choices in GoClr, our development environment for Go featuring abstraction by refinement.
- We illustrate the usefulness of this abstraction toolkit through a collection of case studies.
- We discuss open issues, such as usability, challenges in merging, interactions with external editors, and possible approaches for resolving them.

1.3 Paper Outline

The rest of this paper is organized as follows. First, in section 2, we use two examples to motivate building abstractions by refinement and illustrate its use. In section 3, we describe our design and realization of a system with support for building abstractions via program refinement for a real-world language, Go. We then use a few examples in section 4 to highlight the utility of the paradigm. In section 5, we discuss a few open issues. Finally, section 6 surveys related work, and section 7 concludes.

2. Illustrating Examples

To motivate and illustrate the utility of white-box abstractions created with program refinement, we highlight the following examples.

Depth-First Search Imagine using a package that implements some graph operations, among other things. The package contains a public DFS function that does a depth-first traversal of a graph passed via a root node as an argument

```
1  func DFS(root *Node) {
2    q := Stack{}
3    q.Push(root)
4    for !q.Empty() {
5      node := q.Pop()
6      if !visited(node) {
7        markVisited(node)
8        for adj := adjacentNodes(node) {
9          if !visited(adj) {
10           q.Push(adj)
11         }
12       }
13     }
14   }
15 }
```

Figure 1. The original depth-first search function provided by the library.

```
1  func DFS(root *Node, look func(*Node)) {
2    q := Stack{}
3    q.Push(root)
4    for !q.Empty() {
5      node := q.Pop()
6      if !visited(node) {
7        markVisited(node)
8        look(node)
9        for adj := adjacentNodes(node) {
10         if !visited(adj) {
11           q.Push(adj)
12         }
13       }
14     }
15   }
16 }
```

Figure 2. Depth-first search procedure extended to support a custom processing via a function reference.

and marks them as visited. The end result is that all reachable nodes are marked as visible in some state variables internal to the package for future use, for instance to check graph connectivity.

A programmer using this package is interested in the depth-first search functionality (Figure 1), but needs to perform a custom task when a new node is visited, like printing its satellite data.

Had the original author of the `DFS` function had the foresight that it would be used this way, they would have pro-

vided a generic way to pass in, say, a function pointer to the `DFS` function (Figure 2). The caller would have then supplied a function that takes a node and processes it as an argument to the `DFS` function. Note that providing this functionality is only possible if the underlying language has the required bells and whistles, like the ability to pass functions as arguments. Furthermore, this approach limits the degree of freedom of the client to intervene at the specific point after `visit` is called in the function. Any other functionality would still be unsupported. Realistically, the caller might want to use the DFS code to find back-edges in the graph, which requires more changes to `DFS` than just being able to pass the callback that would be run on every visit.

Nevertheless, the original author has not provided us with this functionality. We are stuck with a decision to copy and paste the `DFS` source code or modify it in-place.

There are a number of problems with explicit copying and pasting. First, the programmer needs to figure out where to paste the copied code. If the code is pasted in the caller context, it will not compile, because its identifiers refer to dependencies that are meaningless in the caller's scope. Therefore, the programmer needs to manually resolve the references, if possible. It is not always possible due to private identifiers within packages.

Pasting the code in the callee context is essentially forking the function into two. The first downside is doing that means you are essentially forking the dependency library and updates to the dependency will not be as straightforward to use from then on. Second, even with a good version control system, the updates and bug fixes to the original `DFS` would not propagate to the cloned implementation.

Modifying the code in-place has the obvious downside of potentially breaking the existing clients who are relying on the subtleties of the existing behavior of the function.

With our system, this problem is easily fixable by right-clicking on the `DFS` identifier in the programming environment. The system will let you specialize `DFS` function for that specific call site. That way, you can modify the body at will and add appropriate statements wherever needed. The language does not even need to support function pointers.

If the original library changes upstream, the system will automatically try to merge the specialized versions of the functions with the updates to them fetched from the upstream package source. Should the merge succeed automatically, the update would be seamlessly applied to all of the specialized versions of `DFS`.

Importantly, the visual footprint of the specialized `DFS` is confined to that particular caller only. It would not be visible when browsing the source code of the dependency package.

Syntax Tree Visitor Programming language toolchains often provide procedures that parse the program text into a tree data structure representing the program. Scanning the abstract syntax tree has many use cases from pretty-printing to enabling editor refactorings and compiler optimization and code generation. Since this is a common operation, the libraries defining the AST structure usually provide an explicit API to help *visit* the nodes in the tree. Nonetheless, due to many node types in the AST, implementation of a visitor is tedious and one can end up with a long `switch-case` construct.

Indeed, the structure of various visit procedures are usually the same, reflecting the structure of the AST, as opposed to the nature of the specific process, making this problem a prime candidate to leverage program refinement.

One could start by calling the pretty-printing procedure often shipped with the language library, passing in an AST representation. To get to the more specialized use case in mind, the programmer makes the programming environment–likely by right clicking on the function invocation and selecting "Specialize" in a visual environment–aware of their intent to customize the body of the callee for their own purpose. The programming environment then provides them with the existing callee source code as the basis for modification. When the changes are saved, the programming environment asks for a summary of changes, analogous to a commit message in a revision control system.

The message is made visible at the specialized call site to make the user aware that the callee is in fact modified and would potentially exhibit different behavior at run time.

When a modification is more generally usable, the programmer may want to give it a unique name, so it can be used from multiple call sites. This is possible by simply renaming the function when editing the initial template. The new function act as if it were a separate function declarared adjacent to the original template in terms of access to the variables in its scope, without requiring source-level changes to the original library. The existence of this function would be dependent upon continued existence of the original template and changes to the original are propagated to the specialized versions as well.

3. Design and Realization

To experiment with building abstractions by program refinements, we designed a prototype system, GoClr (pronounced "go clear"). In this section, we describe some of the design challenges we faced and how we tackled them and the rationales behind our choices.

3.1 Programming Environment

GoClr has its own custom programming environment that is based on the Go programming language [2] and Git version control system [1] internally. Go was chosen as the programming language for the following technical and conventional reasons:

- Go is designed to understand the need for external packages and tools for package management.
- Go packages are conventionally distributed as source code and dependencies are often compiled in a static binary at build time.
- Go is a simple language and lacks many of the conventional mechanisms to build abstractions, like generics, making it a particularly suitable testbed for building abstractions with program refinement, due to a more acute need for alternate abstraction mechanisms, and minimal potential complexity arising from interference with existing language features.
- The Go community seems to strongly prefer lightweight abstractions and has a tendency to more strongly resist overengineering relative to the communities of more common languages. Specifically, this helps in being able to statically resolve the callees in invocations, as refining dynamically dispatched procedures is confusing and unsupported.

3.2 Projects and Build Strategy

For simplicity, the initial version of GoClr is not designed to actively interoperate with other development environments. GoClr normally stores program pieces as separate Git objects and generates textual Go source files to be fed to the actual compiler toolchain only when the project is going to be built. For practical purposes, and for seamless working with dependencies, an import mechanism is provided. Importing existing Go source packages will parse and transform them to the internal data format, while preserving the connection between the original location of tokens and the abstract syntax tree. This connection is necessary to identify and propagate changes to the program when a dependency is updated, for instance.

3.3 Merging

A necessary feature for realization of our vision is support for propagation of changes when a piece of code that serves as the base for one or more refined specializations changes, either at the source code level in an external code repository serving the dependency, or within the current project. Proper merging is critical to providing a great user experience when relying on program refinement-induced abstractions.

Mechanics of a Merge For a refinement-oriented programming system to provide effective abstractions at scale, a mechanism to propagate updates to the upstream procedures that serve as templates is essential. Merging is performed by differencing an specialized code snippet from its original base and storing the information. The system also needs to keep an index of code snippets that are derived from the original template.

When a change in an upstream repository is detected, the index is looked-upon to see if there are any derivatives of that upstream change in the codebase. If such derivatives exist, the system tries to automatically apply the stored difference for that particular derivation to the new version of the function. Theoretically, this process can be done manually or automatically. In its primitive form, the programmer is asked to observe the differences and either keep the existing derivation, refusing further propagation of upstream changes, or to rewrite the derivation manually based on the new upstream function. In the latter case, the system would compute the new delta and update the internal propagation link to the new upstream version, to accomodate for future upstream changes.

It is, however, conceivable that systems that relies solely on manual changes would be tedious to operate at scale. Therefore, at least some level of automatic merge seems necessary.

Automerging A difference between an upstream procedure and its refined specializations is an expression of programmer intent. A programming system, by observing the difference, can capture what the programmer intends to express in the form of “like X, but with Y”.

Distillation and store the difference information between a code snippet and its refinement can be done at various levels: at the low level, a purely string based approach without any understanding of the programming language can be used. At a higher level, programming language syntax can be taken into account, and changes would be stored as something along the lines of “surrounded the function body with an `if` statement”. This level of understanding of the program meaning is helpful in avoiding uncompilable code generation by the merge process.

More sophisticated analysis, perhaps with the help of statistical techniques and machine, can be used to capture the meaning of more sophisticated changes, like fixing an out-of-bound-access bug that may require reordering of control flow within the function that can be impractical with a syntax-level differencing engine.

Of course, it is possible for automatic merge tools to fail. When that happens, the user has the option to manually do the merge—by effectively rewriting the specialization on a new base—, or disentangle the specialization from the base piece of code, thus creating a new copy of the code. Obviously, future changes to the base piece will not be propagated to the distinct specialized copy anymore and the new piece takes on its own independent life.

***Merging in* GoClr** The prototype implementation of GoClr only supports simple automatic syntactic merges by piggy-backing on Git itself. While accurate and sensible automerging is a distinct problem from the general idea of abstraction with program refinement, a smart merge subsystem is critical for a good programmer experience, especially as the project and as a result, the quantity of specializations grow. To help solve this problem, we envision providing API hooks for smart mergers that can understand semantics of the differences and the programmer intent behind changesets. Such smart merge tools can then automatically reapply the modifications inferred on the new base version.

3.4 User Interface

The user interface is a critical piece of the solution. Specialized versions of the code should be hidden from the programmer except when they are explicitly looking for them or they are interested in a particular specialization relevant to a specific call site. Otherwise, the clutter caused by the visibility of many variations of a single procedure will make the system unbearable: imagine C++ programmers having to see template expansions for each specialized type.

It is also of utmost importance to properly highlight and indicate that a callee is specialized for a particular call site. GoClr will let you provide a short comment when specializing a base template that will be visible to the reader under a specialized function name in the call site.

4. Applications

Building abstractions by program refinement is helpful in various ways to the programmer. In this section, we discuss a select few of its potential applications.

Debugging Aid Often in debugging scenarios, the programmer might be interested in temporarily customizing the functionality of a procedure in a specific invocation, without having it behave differently for the program at large. GoClr lets the programmer do exactly the customization they need per individual call site. The programmer can choose to customize a specific procedure when called from a specific location and add diagnostics and print statements to aid debugging, for instance.

The changes will affect only calls originated from a specific call site and the rest of the program executes as it normally would.

This debugging technique can be applied even if the programmer does not expect to specialize the function permanently. To get back to the original functionality, the programmer can just revert the specialization within the programming environment.

Customizing Generated Code Automatic program synthesis tools and code generation tools expect the generated code file to not be edited manually, because the edits would be lost if the code generation tool is run again. Therefore, generated code is usually kept in a separate file and maintains a clean, minimal, interface to the other pieces of the program that are non-generated. This style may make sense for tools like parser generators, that have a very clear, isolated, functionality, but they effectively discourage the use of a class of automatic programming tools that require more customization on the output produced and are more entangled with the host program.

Naturally, we can simply consider the generated code a separate dependency and propagate changes in the output of the code generator to specialized versions of functions that are based on pieces of the generated output.

Exposing Hidden State in Dependencies There are times when excessive focus on encapsulation cause problems. For instance, a concrete problem with the TLS package in older versions of the Go runtime library was the lack of an exposed connection identifier. In order to perform meaningful authentication over an established but unauthenticated TLS channel, while preventing man-in-the-middle attacks, you need a way to bind the underlying TLS channel to the higher level authentication sequence. There used to be no easy way to extract a connection identifier from the Go runtime library's TLS package. Forking that piece of the library and manually adding a method that exposes the internal state variable is a way to accomplish it, and it is a huge burden. Luckily, with a simple program refinement technique, a method can be effectively added to the package that reads the connection ID from the internal state variables and returns the value for use by the caller, effectively circumventing the overly strict encapsulation policies of the package, for good reason.

Lightweight Forking Considering the vast variety of freely available source code on the web, sometimes all the programmer wants is to write a program whose functionality can leverage a subset of another program, with minor additions and differences. For instance, a static analysis tool might be based on a compiler toolchain that was not in-

tended to be used as a library. GoCLR can be used to help the programmer extend and manage the fork without severing the ties to the original program, *i.e.* future changes and bug fixes in the original program can still propagate through the derivative.

5. Open Issues

Refining programs is fundamentally a new way to define abstractions. The GoCLR is in prototype stage and work should to be done to ensure seamless cohabitation of this concept with other language features present in more featureful languages. Furthermore, we must assess its utility and usability, work to identify the applications for which it is most useful and effective and how to ensure we maintain a delightful user experience.

Effectiveness The utility of programming languages, techniques, paradigms, and tools are often subjective and difficult to evaluate. This is especially true for new paradigms and ideas that have not been widely applied. Established paradigms like Object-Oriented Programming and Aspect-Oriented Programming [11] faced a similar issue. In section 4, we follow their footsteps [4, 11] and illustrate the utility of using program refinements to infer abstractions with a few case studies.

Empirical studies are needed to quantify the impact of availability of different ways to build abstractions on programmer productivity. Unfortunately, a meaningful empirical study is hard to do before widespread adoption of a paradigm. While we believe there are compelling use cases for building abstractions on top of program refinements, there are concerns about syntactic modifications leading to the prolification of divergent specialized versions of a procedure that hardly resemble their original base and it may prove to be hard to reason about them as a general, unified, thing, which might lead to adverse effects on programmer productivity. Quantifying such effects is an open issue.

Interaction with External Editors In order to capture refinements, propagate changes, and present the appropriate code specializations in their right context, the programming environment needs to store some metadata. In our implementation of GoCLR, this metadata and the associated code is not meant to be modified outside the environment, therefore the developer is mostly confined to the GoCLR editor. In order to resolve this problem, a standard format for persisting the specializations and the appropriate links and metadata should be developed. Even with such standard format that could be supported in alternative editors, some programmers strongly prefer sticking to their favorite plain text editors without additional functionality. It is conceivable that a useful implementation of the concepts presented in this paper would require a smart editor to be effect. Not supporting plain text editors can hinder its adoption among some programming circles, therefore research into adapting the techniques to a text editor and command line tool-based environment remains an open issue.

Merge Conflicts While syntax oriented merges can work well when the changes are spread away, as they become more granular, too many conflicts start to emerge. The burden of resolving conflicts is enough that if they are frequent, it will discourage people from using the system.

Programming language-aware merge tools can help alleviate this problem because they can take a more semantic oriented view at the language and do a better job at merging. That said, the merge problem is definitely one that has a lot of room for improvements.

6. Related Work

Kiczales [10] identifies the issue of leaky abstractions and the necessity for being able to reach into them at times. He observes that in practice, the implementation cannot always be hidden, citing performance characteristics show through in significant ways as an example of how abstractions can leak. This work discusses the deficiencies in mainstream abstraction frameworks and suggest application of a *metaobject protocol* technology to resolve the problems. The metaobject protocol is a reflection mechanism that lets the client reach into an abstraction and alter its behavior. In dynamic environments like Ruby and JavaScript, the "monkey patching" technique is commonly used to swap a value of an object property or a method body at run time to achieve the desired results. Reflection is often limited in its power in more static environments and commonly these all the prior techniques operate at the granularity of a method at best. In comparison, program refinement can work by syntactic manipulation of statements within method bodies. Since it is a syntactic tool, its power is effectively only limited by the expressivity of the host language itself.

Domain specific languages [16] provide an alternative path to managing complexity without the need to build a deep hierarchy of abstractions. Domain specific languages that are implemented as code generators synergize well with program refinements.

Embedded domain specific languages [5] are basically language extensions for which the parsing is handled by some of the objects used in the program, depending on the context. These languages increase expressiveness and concision of programs, but still require careful upfront thinking by the library author. Of course, program refinement is not con-

fined to any particular language, so in principle, the domain specific parts of a program can also be refined and specialized.

There is a body of work related to detecting cloned code [9, 12], automatically propagating patches through them [15], and. One distinction of these systems from our work is that we do not increase the footprint of the codebase, whereas copying-and-pasting excessively increases the code size and visually cluttering to the programmer.

7. Conclusion

In this work, we have introduced a new, generic, way to build abstractions by program refinements. Program refinement is a powerful tool for building many forms of abstractions because it is limited only by the expressiveness of the host language.

The key insight about basing abstractions on modifications at the syntactic level is interpreting such changes in the context of the original definition, as opposed to the caller's scope, while the effect would be limited to a specific call site. This gives the programmer implementing the caller an easy way to reach into the implementation and customize the concrete code behind an abstraction to achieve the desired effect that is executed when necessary.

We believe that the ability of building custom abstractions via arbitrary syntactic manipulation of code is a powerful tool that can alleviate the need for narrower, more specific, abstraction tools that exist in some programming languages, and liberates the programmer from fighting with abstractions that confuse the programmer down the line and hinder program understanding and programmer agility.

References

[1] Git. `https://git-scm.com/`.

[2] Go programming language. `https://golang.org/`.

[3] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[4] M. Afshari, E. T. Barr, and Z. Su. Liberating the programmer with prorogued programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 11–26, 2012.

[5] M. Bravenboer and E. Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 365–383, 2004.

[6] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Some features of the SIMULA 67 language. In *Proceedings of the Second Conference on Applications of Simulations*, pages 29–31, 1968.

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. 1983.

[8] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, 1973.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, 2007.

[10] G. Kiczales. Towards a new model of abstraction in the engineering of software, 1992.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97*, pages 220–242, 1997.

[12] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, Sept. 2005.

[13] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, 1974.

[14] J. Locke. *An essay concerning human understanding*. 1689.

[15] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180, Sept 2004.

[16] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6): 26–36, June 2000.