Replacing TODO-Comments by Applying Hole-Driven Development to C#: A Proof of Concept

BERNHARD MAYR, University of Innsbruck, Austria

Tightening the feedback loop while being able to quickly test new ideas allows programmers to stay focused. However, popular general-purpose programming languages do not provide short feedback cycles while implementing new features or fixing bugs. In this proof of concept, psychological foundations of complex problem-solving will be laid out, before the history and status quo of note-taking in programming is presented. After that, the concept of hole-driven development will be introduced. Combining these ideas will lead to a concept that enables programs that are executable, despite being incomplete. This allows programmers to gradually refine their mental models while being able to test their assumptions by already running potentially incomplete programs.

Additional Key Words and Phrases: code comments, executable comments, hole-driven development, developer experience

1 PROBLEM, BACKGROUND AND MOTIVATION

Despite a lot of companies' marketing efforts, developing software is not easy and no silver bullet was yet found. A common effort in finding this silver bullet over the last decades is tightening the *feedback loop*. This term describes the time and work that needs to be done between writing a line of code and actually being able to see or test its effect [1]. Aguiar et al. [1] created the visual analogy of adapting one's aim while shooting with a bow and arrow vs. using a hose to target a goal. A tighter feedback loop offers an easier (not an easy) development model [3].

In a very simplistic view, programming can be seen as the act of converting requirements to source code. With this task being carried out by people, one could look at psychology's research field of complex problem-solving and apply it to programming. There is not a lot of active research being conducted in this area, but Weinberg [13] and Naur [9] did some fundamental research combining psychology's complex problem solving and the act of programming. In his essay *Programming as Theory Building* Naur [9] hypothesized that programs are more than just their textual artifacts. Instead, individuals, as well as teams create a shared understanding of the problem they have to solve. Programming is about creating this understanding, hence improving programming is about improving the creation of this understanding, thus improving communication.

Communication does not only happen in teams. Someone, developing something that can not be accomplished in a few minutes, needs to communicate with oneself as well as the computer. Decomposing requirements, keeping track of them, discovering bugs, and transforming code changes are just some of the tasks that might need some form of communication. Quite a lot of this communication happens via code comments [14]. Comments are not only used for documentation purposes, but as well to make notes of things that have to be changed or considered. In contrast to the parts of source code that are actually run, comments are informal and not bound to any syntax rules, making them hard to parse and interpret for machines. This leads to comments being forgotten quite frequently [10]. Preventing their forgetting and actually making use of them for the act of programming is the main motivation for this research.

Veröffentlicht durch die Gesellschaft für Informatik e.V.
in L. Hirsch & M. Kurz & N. Wagener (Hrsg.):
Mensch und Computer 2023 – Workshopband, 03.-06. September 2023, Rapperswil (SG)
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.18420/muc2023-mci-src-414

Another motivational factor for making comments executable is the author's involvement in the StateML[1] project, which aims at providing a shared visual language for modeling event-driven behavior based on Statecharts [6]. This multi-modal (visual, textual, . . .) language enables the cooperative definition and development of behavior by providing live-visualizations for non-developers. StateML artifacts can be simulated, but the language itself is not executable, it must be embedded in a host programming language which executes all side effects. To fully utilize the potential of Statecharts and improve stakeholder collaboration in agile and incremental software development processes, it is necessary to partially execute those models of behavior.

This work represents a proof of concept for applying the concept of hole-driven development to general-purpose programming languages, in this case C#.

2 RELATED WORK

The term *hole-driven development* is not yet scientifically defined, but based on other programming concepts like *type-driven development* or *test-driven development*. By applying type-driven development, developers lift requirements into the space of static types. This allows type checkers to enforce those rules and thus reduce the mental burden for developers [2]. In test-driven development, requirements are transformed to test cases before they get implemented, hence the tests aid the implementation of the system under development [8]. The concept of holes can be found in programming languages like Agda (called goals), Coq, Haskell and Idris [2, 5, 12]. Hole-driven development is described by Brady [2] as "[h]oles allow you to develop programs incrementally, writing the parts you know and asking the machine to help you [...]". An example for a hole in Idris is depicted in Listing 1.

```
main : IO ()
main = putStrLn ?greeting
```

Listing 1. A hole named greeting in an Idris program

As shown in Listing 2, executing `greeting` against the program of Listing 1 in the Idris REPL, the compiler signals that *greeting* is a hole (depicted by the question mark in front of the label) and that the hole is of type String. This information is also shown during compilation and aids developers in finding the "missing puzzle pieces" for their holes. Types can help in finding these pieces, e.g. by applying heuristics [2] or acting as input for type-based search engines for functions and packages, like Hoogle[2] for Haskell. There are also examples of emulating the concept of holes in languages that do not natively support it. The creator of Scala, Martin Odersky, added the ??? operator to Scala's standard library to mark the right-hand-side of a method as not implemented [11]. He also mentions its applicability to teaching by just telling his students to replace all ???'s.

```
*Hello> greeting
?greeting : String
```

Listing 2. The Idris typechecker providing information for a hole

Regarding the common usage of comments, Nie et al. [10] and Ying et al. [14] provide a sophisticated overview of their different variants. In contrast to comments that provide documentation, they define the term *action comments*, which describes comments that mark code that needs to be adapted. Micah Elliott proposed a standard called *PEP 350 – Codetags* to Python [4]. It should unify the representation and tooling of action comments, but it was dismissed[3] because of "[...] no desire to make the standard library conform to this standard." Tools like

[1]https://www.stateml.org/
[2]https://hoogle.haskell.org/
[3]https://peps.python.org/pep-0350/#rejection-notice

Catana[4] or imdone[5] allow the extraction and organization of TODO-comments based on plain source code search. They promise to manage developer annotations by bridging source code comments to project management software. The tool TrigIt enables executable comments in Java[10]. By working directly on the abstract syntax tree of the Java code, action comments written in the TrigIt-DSL can automatically be triggered if source code changes or time passes by, not letting teams and developers forget about their action comments.

3 APPROACH AND RESULTS

The goal of this research is a proof of concept of adding hole-driven comments to general-purpose programming languages like C#, TypeScript or Java. This proof of concept targets the programming language C#, Visual Studio 2022 Community Edition was used for testing the hole reports. However, they are implemented using Roslyn Analyzers[6], which do not depend on a specific integrated development environment. Surveying existing solutions for incomplete program definitions and experimenting with them led to the following requirements, their usage is shown in Listing 3:

Holes DSL A C#-internal domain specific language to construct holes, which is loosely based on PEP 350 – Codetags [4], TrigIt [10] and the comment types identified in [14].

Executability The main promise of hole-driven comments is their executability. One should be able to already run an under-specified program and choose some of the unspecified behavior at runtime.

Hole Reporting To provide value to the programmer, the holes need to be reported. Roslyn Analyzers provide capabilities to analyze the source code and report errors independent of the used editor. These can be used to detect and report the holes.

Build Differentiation Enabling simulation capabilities can be implemented by supporting different configurations in different build variants. As an example, users might want to simulate holes in a debug build while breaking the release build due to under-specified behavior.

Support Mocking Libraries Providing data at run-time might not always be the best solution (e.g. timing issues, no user interface, . . .), so users should be able to integrate external mocking libraries that provide the data.

Refactoring Integrations Based on Roslyn's analyzers and code fixes, certain refactoring operations could be offered to developers. This should work like TrigIt's code manipulations [10] or Idris' heuristics [2].

Comment Transformation Again, Roslyn should make it possible to analyze an existing project and transform existing TODO-comments into holes.

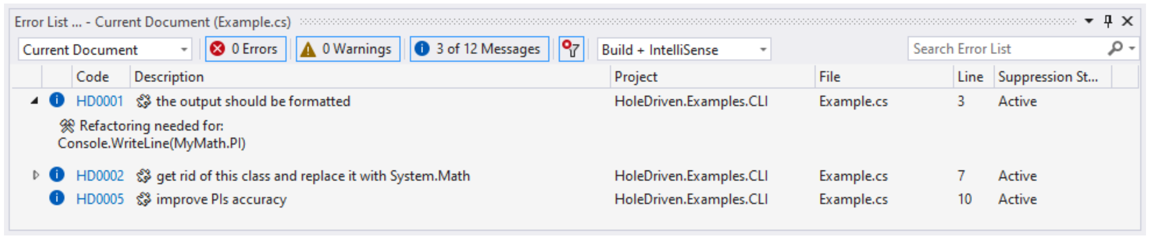
```
Hole.Refactor(
    "the output should be formatted",
    () => Console.WriteLine(MyMath.PI));

[Hole.Idea("get rid of this class and replace it with System.Math")]
public class MyMath
{
    public static double PI => Hole.Provide(
        "improve PIs accuracy",
        value: 3.1415);
}
```

Listing 3. Exemplary API design for hole-driven comments in C#

[4]https://catana.dev/
[5]https://imdone.io/
[6]https://learn.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview

Figure 1 shows Visual Studio's error list for the source code of Listing 3. It shows that the holes get recognized and reported to the IDE's way of showing information regarding source code. If the build configuration is set to Debug, the holes get reported with type information, if it is set to Release, they are reported as errors, thus preventing a successful build.



Code	Description	Project	File	Line	Suppression St...
HD0001	the output should be formatted Refactoring needed for: Console.WriteLine(MyMath.PI)	HoleDriven.Examples.CLI	Example.cs	3	Active
HD0002	get rid of this class and replace it with System.Math	HoleDriven.Examples.CLI	Example.cs	7	Active
HD0005	improve PIs accuracy	HoleDriven.Examples.CLI	Example.cs	10	Active

Fig. 1. Visual Studio's error list reports the detected holes of Listing 3

4 NOVELTY, DISCUSSION AND CONTRIBUTIONS TO HCI

The results in section 3 show that it is technically possible to integrate the holes into general-purpose languages like C#. Using the source code itself as a medium to note ideas and partial thoughts provides the foundation for a lab assistant-like system as Brady [2] imagines. Future research should validate assumptions like that the feedback loop is actually shortening, programmers benefit (in terms of working memory relief) of using holes and the performance and stability of the application under development is not negatively affected. Another aspect of research should be the applicability to other general-purpose programming languages. Further research could also be conducted regarding developer satisfaction of using hole-driven systems as well as potential applications for improving stakeholder communication by co-creation of interactive prototypes. Regarding the contributions to the area of HCI, attaching hole-driven development capabilities to existing development environments, might enable new ways of approaching and solving problems without having to rebuild established systems. Holes might find applications in prototyping, teaching or motivating people to get started with programming, as originally intended by the creators of Smalltalk [7].

REFERENCES

[1] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. 2019. Live Software Development: Tightening the Feedback Loops. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming (Programming '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3328433.3328456
[2] Edwin Brady. 2017. *Type-Driven Development with Idris*. Manning Publications Co, Shelter Island, NY.
[3] Luke Church, Chris Nash, and A. Blackwell. 2010. Liveness in Notation Use: From Music to Programming. In *Annual Workshop of the Psychology of Programming Interest Group*. Psychology of Programming Interest Group, Madrid, Spain, 2–11.
[4] Micah Elliott. 2005. PEP 350 – Codetags. https://peps.python.org/pep-0350/.
[5] Ben Gamari. 2019. Haskell: Typed Holes.
[6] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (June 1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9
[7] Alan C. Kay. 1993. The Early History of Smalltalk. *ACM SIGPLAN Notices* 28, 3 (March 1993), 69–95. https://doi.org/10.1145/155360.155364
[8] D. D. McCracken. 1957. *Digital Computer Programming*. John Wiley & Sons, New York.
[9] Peter Naur. 1985. Programming as Theory Building. *Microprocessing and Microprogramming* 15, 5 (May 1985), 253–261. https://doi.org/10.1016/0165-6074(85)90032-8
[10] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. 2019. A Framework for Writing Trigger-Action Todo Comments in Executable Format. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering. ACM, Tallinn Estonia, 385–396. https://doi.org/10.1145/3338906.3338965

[11] Martin Odersky. 2011. Adding ??? To Predef? https://www.scala-lang.org/old/node/11113.html.

[12] Patrick Redmond, Gan Shen, and Lindsey Kuper. 2021. Toward Hole-Driven Development with Liquid Haskell. *CoRR* abs/2110.04461 (2021), 7. https://doi.org/10.48550/ARXIV.2110.04461

[13] Gerald M. Weinberg. 1971. *The Psychology of Computer Programming*. Van Nostrand Reinhold, New York.

[14] Annie T. T. Ying, James L. Wright, and Steven Abrams. 2005. Source Code That Talks: An Exploration of Eclipse Task Comments and Their Implication to Repository Mining. In *Proceedings of the 2005 International Workshop on Mining Software Repositories - MSR '05*. ACM Press, St. Louis, Missouri, 1–5. https://doi.org/10.1145/1083142.1083152