WASP Project Course 2021

Composable Software Tools for the Working Programmer

Background

Modern software technology gives us tools that can systematically analyse and transform software and thereby dramatically improve developer productivity. However, typical software tools operate as parts of idiosyncratic frameworks within IDEs or other specialised tools. In industrial practice, this makes it hard to integrate software tools with other automated processes, to deploy them organisation-wide, and to enable non-expert software engineers to prototype and experiment.

In this project, we will explore typical software technology problems from the perspective of the UNIX principle that each tool should *do one thing and do it well*.[1] To the best of our knowledge, this principle has never been systematically applied to software tools.

As participants in this project, you will explore design options for decomposing software tools into minimal but re-composable components and select and implement a subset of the proposed tools to prototype and showcase their design (Figure 1 shows an example decomposition purely for refactoring). Your goal is to find a decomposition that is both *general* and *practical*, to give non-experts the ability to analyse and transform source code on the UNIX command shell, and to combine your tools with other shell tools.

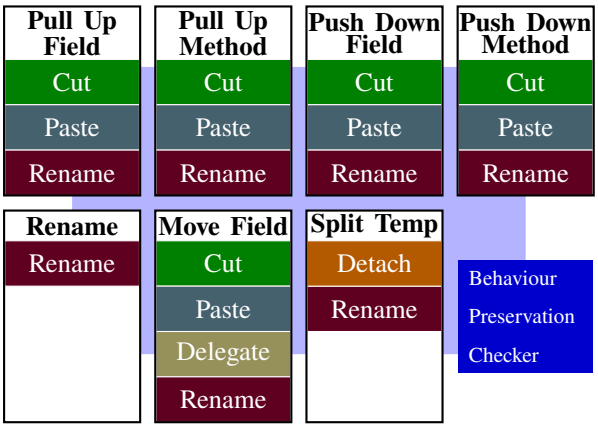

Figure 1: Splitting 7 refactorings into 5 reusable primitives in Program Metamorphosis [14].

Constraints: Students should have a background in software technology and/or compiler construction, or complementary expertise (e.g., user studies / software engineering methodology) and sufficient technical skill to contribute to all phases of the project.

Participants

Industrial partner: Ericsson

Industrial supervisor: Patrik Åberg (`patrik.aberg@ericsson.com`)

Academic supervisor: Christoph Reichenbach (`christoph.reichenbach@cs.lth.se`)

Coordinating WARA representative: Christoph Reichenbach (WARA-SW)

[1] First formulated by McIlroy as "make each program do one thing well" [12].

Suggested WASP PhD students: Momina Rizwan, Idriss Riouak

Challenges to investigate

As project participants, you will explore the literature and existing tools (e.g. Clang [3], SpotBugs [2, 8], Error Prone [1]), previous (dis)integration efforts for IDEs (Monto [18] and LSP [17]) and refactoring: (JunGL [20], Program Metamorphosis [14], and Schäfer et al.'s work [15]) as well as program analysis DSLs and frameworks (e.g., Soot [19]/Phasar [16], Doop [4], Klee [6], PQL [11], Flix [10], MetaDL [7], Infer [5]) and extensible tools aimed at end-users (Spoon [13], Coccinelle [9]), as well as others that you can find.[2] First, you will develop a *set of use cases* in collaboration with the industrial supervisor, develop a *list of design goals* together with both supervisors, and design a set of tools that satisfies the design requirements listed below to satisfy the design requirements while optimising for your (possibly conflicting) design goals:

Design requirements

Your toolset design must satisfy all of the following:

- Run in a UNIX shell environment (Linux, OS X).
- Not depend on interactive use or availability of a GUI.
- Not tie communication between tools and developers/other tools to any specific libraries or framework, beyond what the specification of your target language (plus other applicable standards) guarantee.

Your prototype *implementation* may diverge from these requirements, within reason, e.g. by offering extensions that are specific to some given compiler framework.

Design goals

Your design goals should aim to maximise utility for the intended end users, i.e., software engineers who are familiar with your target language but not with compiler technology or complex program analyses.

Implementation

You will demonstrate your design with the implementation of a set of prototype tools for either **C** (which is likely to simplify the evaluation) or **Java** (which is likely to simplify the implementation). Your implementation may be incomplete, as long as it allows you to answer relevant research questions.

[2] Two examples of other related tools that attempt to follow the UNIX design principle to some degree are DMCE (https://github.com/PatrikAAberg/dmce) and `clang-rename` (https://releases.llvm.org/3.9.0/tools/clang/tools/extra/docs/clang-rename.html)

Research Questions

Below is a list of possible research questions, but you are free to propose others:

- How effective is your design at helping non-expert software engineers accomplish the tasks that you have gathered in your set of use cases?
- What effective criteria are there for comparing UNIX-style toolset designs, given the motivation in the Background section?
- What effective strategies are there for interfacing with tools that need "project-wide" information (e.g., finding all call sites)?
- What effective strategies are there for experimenting with program transformation (i.e., to observe the result of a transformation without committing to it)?
- What are the strengths and limitations of streaming UNIX pipelines, the communication primitive that allows us to connect different tools, in this context?
- What effective strategies are there for reducing or eliminating the need to re-analyse input programs when processing the same input program more than once, e.g. in a loop or a pipeline?

Resources

- **Tools**: You may use and adapt any Open Source systems / libraries for this project, including your own research tools, as long as they will be Open Source at the end of the project.
- **Evaluation**: Developers from Ericsson will help evaluate the students' design for industrial fit and explore options for continuing this work within the WARA-SW beyond this course.
- **Storage**: You can use any public git repository or private git resources e.g. at LTH.
- **CI**: You can use WARA-Common resources for Continuous Integration.
- **Web and wiki**: You can use WARA-SW resources for private wikis or a web presence, as needed.

Deliverables

The industrial and academic supervisors for this project ask for the following deliverables:

- A project report in SIGPLAN conference format that collects your design and insights. Aim for a conference paper.
- A set of Open Source prototype tools that demonstrate your core insights.

The WASP project course manager may add more requirements (e.g., demonstrator videos).

References

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building useful program analysis tools using an extensible java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23, 2012.

[2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and Johan Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[3] Bence Babati, Gábor Horváth, Viktor Májer, and Norbert Pataki. Static analysis toolset with clang. In *Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary)*, pages 23–29, 2017.

[4] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of OOPSLA '09*, pages 243–262, New York, NY, USA, 2009. ACM.

[5] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[6] Ricardo Corin and Felipe Andrés Manzano. Taint analysis of security code in the klee symbolic execution engine. In *International Conference on Information and Communications Security*, pages 264–275. Springer, 2012.

[7] Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. Metadl: Analysing datalog in datalog. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 38–43. ACM, 2019.

[8] David Hovemeyer and William Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[9] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding Error Handling Bugs in OpenSSL Using Coccinelle. In *European Dependable Computing Conference*, pages 191–196, Valencia, Spain, April 2010.

[10] Magnus Madsen and Ondřej Lhoták. Fixpoints for the masses: Programming with first-class datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[11] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using pql: a program query language. *Acm Sigplan Notices*, 40(10):365–383, 2005.

[12] M. D. McIlroy, E. N. Pinson, and B. A. Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[13] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

[14] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program Metamorphosis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 394–418, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In *European Conference on Object-Oriented Programming*, pages 369–393. Springer, 2009.

[16] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410. Springer, 2019.

[17] Fredrik Siemund and Daniel Tovesson. Language server protocol for extendj. 2018.

[18] Anthony M Sloane, Matthew Roberts, Scott Buckley, and Shaun Muscat. Monto: A disintegrated development environment. In *International Conference on Software Language Engineering*, pages 211–220. Springer, 2014.

[19] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, Riverton, NJ, USA, 2010. IBM Corp.

[20] Mathieu Verbaere, Ran Ettinger, and Oege De Moor. Jungl: a scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, pages 172–181, 2006.

Keywords

program analysis, program transformation, command-line interface