Course on *Introduction to Functional Programming*

Lecture on *Basic software engineering*

SS 2023

SoftLang Team, University of Koblenz, 2023

Programmers are obsolete.

We need to be software engineers more than ever!

Ralf: Wouldn't it also make sense to have a lecture on software engineering principles and practices within that course on the introduction of functional programming with Haskell?

ChatGPT: Yes, it could be helpful to include a lecture on software engineering principles and practices in a course on functional programming with Haskell. Here are some possible topics that could be covered in such a lecture:

1. Modular programming: breaking programs into smaller, more manageable pieces.
2. Code organization: structuring projects and modules to improve readability and maintainability.
3. Documentation: writing clear and concise documentation for functions, modules, and projects.
4. Version control: using a version control system (such as Git) to manage changes to code over time.
5. Testing: writing tests to ensure that code behaves correctly and that changes do not introduce new bugs.
6. Continuous integration and deployment: automating the testing and deployment of code changes.
7. Debugging: techniques for finding and fixing bugs in code.
8. Performance optimization: profiling code and optimizing it for speed and memory usage.

These topics could be discussed in the context of Haskell specifically, but many of them are relevant to software engineering in general. The lecture could emphasize the importance of good software engineering practices for writing high-quality, maintainable code, and provide examples of how these practices can be applied in the context of functional programming with Haskell.

Ralf: Ok, I think I will pick a few of them: modular programming, code organization, documentation, version control, and testing. I'll add packaging. I will skip CI, debugging and performance optimization, as I consider them to be advanced for the course at hand. What technologies from the Haskell ecosystem should I leverage?

ChatGPT: Great, those topics sound like a good set to cover in the lecture on software engineering principles and practices.
In terms of technologies from the Haskell ecosystem to leverage, here are some suggestions:

1. Cabal: Cabal is a build system and package manager for Haskell. It can be used to manage dependencies, build and install packages, and create distributable packages.
2. Stack: Stack is another build system and package manager for Haskell. It is similar to Cabal, but provides additional features such as reproducible builds and support for multiple GHC versions.
3. Haddock: Haddock is a documentation generator for Haskell. It can generate documentation from source code annotations, and provides a web-based interface for browsing the documentation.
4. Git: Git is a widely-used version control system, and is well-supported in the Haskell ecosystem. Ther are several libraries and tools available for working with Git from Haskell.
5. QuickCheck: QuickCheck is a library for property-based testing in Haskell. It allows you to define properties that your code should satisfy, and generates test cases automatically to check them.
6. Hspec: Hspec is a testing framework for Haskell that provides a more traditional unit testing interface allows you to write test cases that assert specific behavior of your code.
7. Cabal package format: The Cabal package format is a standardized format for distributing Haskell packages. Understanding how to create and distribute packages in this format is important for sharin your code with others in the Haskell community.

By leveraging these technologies, you can provide students with a comprehensive view of how to organize and package Haskell projects, document their code, manage dependencies and versions, and test their code effectively.

Ralf: Sounds good -- except that I also had HUnit in mind. Would you recommend against using HUnit for (unit) testing?

ChatGPT: No, I wouldn't recommend against using HUnit for unit testing in Haskell. HUnit is another widely-used testing framework in the Haskell ecosystem, and is well-suited for testing individual functions and modules.

HSpec and QuickCheck are often used together for testing in Haskell. HSpec provides a behavior-driven testing syntax, while QuickCheck provides property-based testing capabilities. HUnit, on the other hand, provides a more traditional unit testing syntax.

In summary, using HUnit in addition to HSpec and QuickCheck can provide students with a more complete understanding of testing in Haskell, and allow them to choose the appropriate testing framework for their specific needs.