APPENDIX B

Software Engineering Practices

There are many practices for the various phases and types of software development beyond writing the code. Here we will cover some common software engineering practices, including tools and techniques that you can apply to your software project.

Planning and Management Practices: Agile

Agile practices, practices that attempt to follow the guidance of the Agile Manifesto (`https://agilemanifesto.org/`), have become a dominant approach to software development. Agile practices start with the concept of a minimal viable product or MVP. The MVP is a version of software that provides the minimum capabilities for users to use and developers to learn from.

Scrum

Scrum is one of the most common agile practices for organizing work. In a nutshell, the scrum practice is focused on the short daily scrum meeting like a rugby scrum, or American football huddle. In the scrum or huddle,

© Paul D. Crutcher, Neeraj Kumar Singh, and Peter Tiegs 2021
P. D. Crutcher et al., *Essential Computer Science*,
https://doi.org/10.1007/978-1-4842-7107-0

the team coordinates on the work for the day and then breaks out and does the work.

The work that the team needs to do is divided into small completable segments, called stories. Those stories are organized into an ordered list called a backlog. The ordered list of stories is then divided into groups that a scrum team can complete in a fixed time segment. This time segment can be any length, but at the end of each time segment, there should be a viable/usable software. The time segments are called sprints or iterations. They are called sprints to remind the team that they are short and will finish. They are called iterations because after one is complete, small adjustments can be made and then you start again. These iterations commonly range from one week to one month, with two weeks being the most common.

Scum recommends three ceremonies in addition to the scrum or huddle. First, before the beginning of each iteration, the team confirms what stories the team will be completing in that iteration. The team will use the ordered backlog and, if available, feedback from previous iterations to determine what should be done to have a viable/usable software product at the end of the sprint. This is called the planning ceremony. The second ceremony scrum recommends is the review. This happens at the end of the sprint and is where the team reviews the software that they created with their stakeholders and users of the software, if that is possible. Any feedback from the review should be added to the backlog and taken into consideration for future iterations. A retrospective, also conducted at the end of a sprint, is the third ceremony. In the retrospective, the team examines how they are working and looks for areas to improve. Any feedback the team has will again be used to plan future iterations.

Scrum also recommends three roles to coordinate the work. The most important role is the team. The team is all the developers in the scrum. Next is the product owner or PO. The product owner's main responsibility is to represent the stakeholders for the team and manage the backlog of work. Finally, there is the coach; this person's responsibility is to enable the team to work effectively. The coach will organize the ceremonies and help the team implement any of the feedback from the retrospectives. The coach was previously referred to as the scrum master, but that term has fallen out of favor.

While scrum can be effective and is often a developer's first introduction to agile practices, there are some constraints to the practice. First, scrum does not specifically outline how the work is broken into stories. It does not cover requirements, analysis, or design; in some ways, it assumes that those are already complete. Second, scrum does not cover how the work will be implemented, validated, or deployed. There are other agile practices that cover these areas such as test-driven development, paired programming, and continuous integration and deployment (CI/CD). Third, scrum works a specific team size of from five to nine people. Fewer people than that, and all of the ceremonies are not really required; more people than that, and the ceremonies are not sufficient.

Kanban

Kanban is another agile practice for organizing work. The Kanban process was originally developed by Toyota in Japan for automotive manufacturing. Kanban means a signboard in Japanese. The original Kanban practice used a card or sign that traveled with the work until the work was completed; then the card was returned to the board. If no cards were available on the board, then work was stalled somewhere in the process, and the team could focus on that area until work was completed and a card returned. Cards were intentionally limited to reduce the amount of uncompleted work in progress. For agile software, this principle of WIP (work in progress) limits can be applied to the development of software. An individual or team can have a WIP limit and start work up to that point and then focus on completing that work.

Kanban is like scrum in that work is organized into small compliable segments and then organized into a backlog. Also, like scrum, Kanban does not cover how work will be implemented, validated, or deployed.

Kanban differs from scrum in that instead of fixed time segments like iterations, it has a fixed amount of work in progress. Kanban works well when the work items are of similar scale. Kanban can also scale down to a smaller team size or even an individual. It can scale down because Kanban does not have the concepts of the various scrum ceremonies or dedicated roles.

Analysis and Design

As noted in the preceding, neither scrum nor Kanban specifically covers the analysis and design phases of a project. This has led to the incorrect belief that analysis, architecture, and design are not needed for agile projects. Instead, those practices assume that analysis, architecture, and design have happened to create the backlog. If that is not the case, then you might consider inserting this work as work items or stories into the backlog and have that work as part of the regular work the team does. Another approach is having dedicated time periods such as every third sprint to analyze and design the upcoming work to load into the backlog. Another approach that could work with a larger team is to have one team responsible for doing the analysis and design and feeding the backlog through their own agile processes.

Scaling Agile Practices

Both the practices of scrum and Kanban work well for small teams; however, those practices become problematic as the number of people on the team and the number of teams working on the project scale up. There are a few recommendations about how to approach scaling up agile practices such as Scrum of Scrums or the Scaled Agile Framework (SAFe). The key to scaling agile practices up is to constantly keep agility in mind – being able to quickly change direction and get back up to speed.

Documentation

Documenting your software project is an important way to communicate to the future. In the future, there may be different developers or maintainers of the software. Ideally there will be future users of the software. Questions such as why does it work this way or how do I do this should be found in the documentation, without direct contact to you or the development team.

Requirements, Design, and Architecture

Documenting the requirements, the design, and the architecture is a way to record and communicate what you learned during the design and analysis phases. This is to inform the developer(s) on what to develop.

The formality of writing requirements and design will vary by the type and scope of projects. This formality could be as informal as writing a user story in the form “A user desires some outcome, because of some reason.” A fully specified safety-centric software system where every known possibility is documented will require more formality in its requirements.

We find that for most projects, using the Cockburn use case template is a highly effective way of capturing and communicating the requirements. The template helps to guide the requirement creation, and it helps avoid specifying design and implementation details into the requirement.

Implementation details, like how to interface with a system and what components make up a system, can be documented in the design and architecture. Design and architecture will typically have illustrations in addition to text. These documents should inform the developers of the project how the software should work within itself and with the world.

Over time the requirements will change, the design will grow, and the architecture will morph. It is important to remember that these documents should also be able to change via controlled practices.

Comments and Code

The code itself is a document about what is implemented. Comments in the code should be limited to adding context and not a retelling of what is in the code. This context will be helpful to maintainers of the code.

Well-written code is code that acts as its own documentation. Meaningful variable and function names can help code be its own documentation. However, source code is limited in expressiveness compared to natural languages. When this additional expressiveness is needed, it is a good time to write additional comments around that code.

User

User documentation can take multiple forms: web pages, online help, or even console output. This documentation should provide a road map to your software and guide the users to accomplish what they desire.

Testing

Testing your software is done to both validate and verify your software. Verification is proving that your software behaves as expected, and validation is proving that your software does not behave in unexpected ways.

Phases and Categories of Testing and Goals

Testing your software can be done with various goals and at different phases of the software lifecycle.

Algorithm Testing, Unit Testing, Integration Testing, and the Like

Algorithm testing is typically done early in the lifecycle. Algorithm testing is used to test a selected algorithm with a sample data set that your software will be using. This is used to profile and understand whether the algorithm will be the best match for the data.

Unit testing is done throughout the development of the software. It is often tied into the continuous integration system. Continuous integration is the practice of building and testing your software on every commit to an SCM (source control management) system, which we discuss in more detail in this chapter. Unit testing is when you test the software at the smallest unit possible. This could be a single function, or class in object-oriented programming. The goal of unit testing is to validate the units of software work with a variety of inputs. Having unit tests with sufficient coverage is helpful during the maintenance phase, because it allows a unit of software to be improved while demonstrating that the inputs and outputs are not negatively changed.

If unit testing is focused on individual software units, then integration testing is focused on testing how those units work together. Integration testing has a primary goal of verifying that the software does what it is expected to do when all of the pieces come together. It also has a validation role in that it will help identify any adverse interactions between various units of the software.

There are other types of testing to be aware of, such as exploratory testing, performance testing, and user acceptance testing. Exploratory testing is where a user specifically “explores” to find issues that have not been found through the other types of testing that are done regularly. Performance testing is looking to record the performance in time or memory of your software. User acceptance testing is testing whether a user will accept the software deliverable.

Test-Driven Development

Test-driven development (TDD) is the discipline of developing your tests first, before you write any of the production code, and then writing the production code to make the tests pass. This is a particularly useful practice, especially for unit tests. It can keep the test coverage high for your software. It can also help enforce a good modular design, by making it difficult to have cross-dependencies given the goal of always having to pass tests. Despite all these benefits, it is not practiced as much as it could be. TDD requires a fairly complete knowledge of what the software should do, which is not always possible. It also is sometimes difficult to get over the hurdle of writing the tests first when the value to the users comes from the production software, trading the immediate satisfaction of writing the production code first to the delayed satisfaction of writing tests first.

Developing for Debug

Debugging is typically the exploration of the software to find the root cause of a defect or bug in the software. A debugger is software that will allow a developer to step through the code, one line at a time. This brings the computer speed down to the speed of the developer, so they can observe the effects of each line being executed. For source line debugging, it is best to have the source code available when you are debugging the software. If you do not have the source code, debug symbols are the next best thing. Debug symbols provide source-level information to a debugger without providing the full source code. There are situations where developers will need to debug without the benefit of source code or symbols. When you are developing software, there are activities you can do to support debug for the future engineers needing to debug your software.

Asserts and Exceptions

Asserts and exceptions are program language constructs that can be used to support debugging. An assertion will act as a checkpoint on some fact in the source code, like the value of a variable. An assertion is typically implemented with an assert keyword, which will typically stop the execution of the software, if the assertion is false. Adding assertions to your code will help prove that the data you expect is available. Assertions are typically automatically removed when the code is compiled in an optimization. And assertions that evaluate false actually halt the program, so assertions should be used with caution.

Exceptions are like assertions. Exceptions will check for an event that is not expected to occur. When an exception occurs, an exception handler in your code can catch the exception. Once an exception is caught, it can be raised up the stack for another exception handler to deal with, or it can be handled immediately. A raised exception will provide data about where a defect originates from. For debugging, unhandled exceptions are defects that need to be addressed. Adding code to raise exceptions is a good technique for making your code more debugger-friendly.

Logging and Tracing

Two other practices that help make your code more debugger-friendly are logging and tracing. Logging is recording events that occur in the software to an external file, for instance, so a human or machine can go back and follow the events of software execution. Tracing is using logs or live data to observe the behavior of the software while it is running.

Most modern languages have built-in support for logging. It is a good practice to use these logging frameworks whenever possible. Using a logging framework will help distinguish between messages intended for the logs and messages intended for active users. When adding logging to your software, you need to strike a balance between how precise or frequent you want your log messages to be and the number of messages in the log. Remember that logging takes compute time and that if there is too much information in the log, it may hide meaningful events.

Source Control Management

Source control management (SCM) is the practice of managing the source code of your software. This practice includes managing the directory structure of the source code, maintaining a history of revisions of the code, and versioning the code.

Purpose and Mechanism

Source code management gives the developer or development team confidence to proceed with development knowing that they can go back to a previous revision of the source code, should they need too. A fundamental purpose of SCM is to preserve the progression of the source code development.

SCM systems allow for branches of the software to exist simultaneously, so different revisions of source code can be compared or merged. This allows a team of developers to operate safely, in their own environment, without impacting each other with moment-to-moment changes. When your code branch is ready, you use SCM to integrate the branch to a trunk or mainline of the source code.

SCM systems typically have the same common concepts (Table B-1), although different tools may call these concepts by different terms.

Table B-1. *Common SCM Terms*

Term	Definition
Workspace	The directory structure on a development machine for the source code of software.
Revision	A single incremental change of the source code.
Branch	A line of revisions that are derived from a single point in the past.
Mainline	The branch of the code that is where the integration of various branches occurs. Sometimes called trunk.
Version	A specific revision that has meaning or value.

Imagine using SCM for a small team. For example, a developer will check out a workspace. The workspace will define the directory structure of the source code on the developer's system. As the developer makes changes to the source code, they will commit this code to the SCM system creating a revision. The developer may be creating multiple revisions on a branch. They will then want to share their revisions with the rest of the team by merging their revisions into the mainline. On the mainline, the development team will define the next version by linearly selecting the head revision on the mainline.

For another example, a bug is discovered in the recent version and needs to be fixed. In this case, a developer will check out a workspace based on that previous version. Then they will create a branch to fix the code. As they fix the code, the developer will create revisions by progressively committing their code to the SCM system. They can compare their revisions to the revisions on another branch to identify changes or even to help discover the root cause of the bug. Once they have fixed the bug, they can again merge into the mainline and create another version.

Both examples are somewhat simplified and mix concepts from multiple SCM tools. Each SCM tool will have its own process and workflow. SCM tools can generally be split into two categories: centralized and distributed. A centralized SCM system maintains in a single location a definitive list of revisions and versions. This has an advantage of maintaining linearity of the software and explicit control of a version. A distributed SCM system does not require a central system to maintain the distributions but allows multiple systems to maintain individual history and then add history of revisions from another node in the SCM system. This has the advantage of allowing the full capabilities of an SCM system while being disconnected from the team, but the linearity of the revisions is not guaranteed.

Tools

There are many source code management tools. Each tool has its own unique differences. In the following, we will review two of the most common tools that demonstrate the centralized and distributed SCM systems.

Perforce Helix

Perforce Helix is a good example of a centralized version control system for SCM. It allows developers to define their workspace from the various branches in the overall source code tree. By being a centralized system, it can enforce that revisions are committed in a linear order and can maintain that order. One area where Perforce Helix stands out is how it handles source assets that are not text, such as large binary files like game assets.

Git

Git has become the industry de facto SCM. It is an example of a distributed revision control system. Git maintains a repository history of revisions locally within the workspace. To interact with another instance of the Git repository, a developer can push changes to the other instance or pull and merge changes from that other instance. Because Git does not have a centralized location, other solutions like GitHub have been put in place to act as a central instance of the repository. Other processes have emerged around Git to help define definitive versions such as having merge or pull request as a gate to a mainline branch and using tags to capture the linear progression for versions.

Build Optimizations and Tools

Build tools coordinate the compilation and linking of the source code into usable software.

Purpose and Mechanism

Originally source code had to be first compiled into object files one at a time, and then all those object files had to be linked together into an executable or library. As software got larger and larger, a tool to coordinate the effort of compiling and linking many files together became necessary. This is the basis of what a build tool does.

Adding to the complexity of compiling source code into object files, some of those object files depended on other object files to exist before they could be linked together. And in this case, some of those upstream object files were needed for more than one downstream object file. Managing this collection of object file dependencies is another piece of what the build tool does. Build tools will typically enable a declaration of dependencies and will make sure that the dependencies are satisfied before attempting to compile and link a file. Most build tools will optimize the satisfaction of dependencies by first checking if they exist and then creating them only once, if it does not exist.

Scripted or interpreted languages like Python, Ruby, and JavaScript don't need to compile the source code into object files. Scripted languages can still benefit from build tools that manage the dependencies and create packages and other collateral.

Another thing a build tool does is manage configuration parameters for multiple configurations to inform the compiler and linker how to behave. This allows the object files and software to have multiple configurations, such as debug instances or even support for multiple operating system instances.

This ability to coordinate multiple tools like a compiler and a linker led build tools to be used to coordinate additional tools that are expected in a modern software project like unit test runners, security checkers, and document generators.

Build tools will typically have their own source file to define the configurations and parameters. The configuration file will usually list targets that will be the output of some action and the dependencies that need to be satisfied before the output can be created. Typically build tools also allow a developer to define the tools and parameters to call to create the output. Make and most modern build tools also have default rules for doing the basics of compiling and linking object files.

Tools

There are a lot of build tools available. Some are specific to a language, and many modern languages such as Go and Rust have a build tool distributed with the language. Some build tools are fully declarative, meaning that all the possible options and dependencies are defined in the configuration files. Most build tools are primarily declarative with limited scripting ability for loops and conditional statements. Another category of build tools are generators, like Cmake and GNU Autotools, which use data to configure and generate a build script. Then this build script can be called by another build tool.

Make

Make is one of the older build tools. There are multiple implementations of make that have mostly the same feature set; the most common make is GNU make. The make configuration file is called the Makefile. Make provides a declarative syntax for defining targets and dependencies. Each target line starts with the target followed by a space-separated list of dependencies on a single line. The commands to create the target are the subsequent lines, tab indented, under the target line. Typically, these lines are shell commands that make use of the underlying command shell. By default, the targets are expected to be files that are created on the file system; however, a target can have a `.phony` decorator added to it so that make knows the target can be satisfied even if no output file is created. This allows for an easy name like ALL or drivers to be applied to a list of dependencies instead of the direct output, such as `my_cool_program.exe`.

Gradle

Gradle is a more modern build tool that is built on top of the Groovy language and its Java Virtual Machine (JVM). Gradle configurations are written in a domain-specific language designed for builds. Like make, Gradle can define targets and dependencies. Unlike make, these targets do not have to be files that are created. Gradle remembers what targets have been satisfied in a build cache. Gradle can even share this build cache between multiple systems on a network making it easier to split the build work to improve build time. The commands to satisfy the targets do not have to be shell commands; they can be methods in Groovy.

Cmake and Ninja

Cmake takes a different approach than Gradle or make. Instead of defining the build targets and commands directly in the CMakefile, Cmake defines a script for generating the targets and commands for another build tool. This provides the ability to consistently model the targets and dependencies for your software project and then generate equivalent logic for multiple systems, such as different integrated development environments or different implementations of make.

Ninja is a modern build tool like make. It is intended to be highly performant and minimal compared to build systems like Gradle. Cmake generates Ninja build files, a common practice, with the rich syntax being handled by Cmake and the performant build done by Ninja.

Continuous Integration and Continuous Delivery

Continuous integration (CI) is the practice of building and testing your software on every commit to an SCM system. Continuous delivery builds on the concept of continuous integration to deliver the software to users automatically, typically when the software is merged to the mainline in the SCM system. The term continuous integration was coined by Martin Fowler in 2000.

Purpose and Mechanism

Prior to the practice of continuous integration, when a new version of the software needed to be built and tested, all the various branches and different developers' work would come together for integration in a so-called “big-bang.” A build would be attempted, and if not successful, engineers would have to find the reasons. This could be caused by code conflict or even incompatible code between engineers. Once the initial work to resolve the conflicts and any side effects would be resolved and the build be complete, then testing could begin. All of this is very painful and time consuming, hence the moniker "big-bang." If this integration testing found issues, then the code needed to be changed and any side effects again resolved, and the process would start again. Historically this process could take days or even weeks. So we want to avoid big-bangs.

Continuous integration addresses this "big-bang" integration problem by shrinking integrations into a continuous stream of micro-integration events. In the practice of continuous integration, developers push their changes regularly, ideally daily, to be integrated to a mainline in the SCM system, using build tools that automatically build and validate (through unit testing, for instance) the new integrated version. If this build does not work, the developer can see that within hours and make corrections in the small amount of code that they worked on, instead of digging through everybody's code in the "big-bang" integration style. If the build is successful, then automated unit tests and integration tests can be run. Again, if the tests fail, there is only a small amount of code that could have introduced the failure, so the developer can easily find and fix their code.

Continuous integration systems wait for source code to be pushed to the SCM system and activate when there is a change. The CI system will either monitor the SCM system or be triggered by an event on the SCM system. At that point, the CI system will check out the code and invoke the build tool automatically, and then the CI system will run the tests. Typically, the CI system will report on the status of the build so the developer and the team can review the results.

Continuous deployment utilizes the same CI systems for deployment or delivery activities. After the source code is integrated, built, and tested, the CI system can be triggered to automatically deploy the software. The deployment may require additional steps or stages such as more testing, checking security scans, packaging the software for install, and copying it to a location to either run online or download to install on a local system.

Tools

Like build tools and SCM systems, there are a lot of options for CI/CD systems. They define the stages and steps to integrate and deploy the software. CI/CD systems also define how the tools will interact with the SCM systems.

Jenkins

Jenkins is one of the oldest CI/CD systems. It is still the most popular CI/CD system. Jenkins provides a lot of flexibility in how it can be configured and deployed. Originally Jenkins enabled a wide variety of plugins to expand the configuration interface for defining the rules for your software's CI and CD. Jenkins also provides a scripted, domain-specific language and a declarative syntax, both based on Groovy, to define the CI/CD pipeline. Jenkins is typically installed on-premises, but there are online and commercial offerings. When Jenkins is installed on your premises, you need to provide your own compute capacity for build and testing.

CircleCI

CircleCI is a popular Software as a Service (SaaS) CI/CD system. It provides an online tool to create a CI/CD pipeline and the compute resources for compilation and testing. CircleCI provides a simple UI for defining the connection to the SCM system and a YAML-based declarative syntax for defining the pipeline.

GitLab CI/CD

GitLab CI/CD is an example of a CI/CD system that is built into the SCM system. The GitLab CI/CD system is available wherever the GitLab SCM system is installed. Because GitLab CI/CD is integrated with the SCM system, it requires minimal configuration to connect to the source code. For configuring the CI/CD pipeline, GitLab uses a YAML-based declarative syntax. Using the GitLab SaaS solution provides both the CI/CD system interface and the compute capacity for build and test. Using GitLab with your own environment requires you to provide your own compute capacity. Despite GitLab CI/CD being associated with the GitLab SCM solution, GitLab CI/CD can work with a variety of Git solutions including GitHub.