Just Enough Software Engineering for Domain Scientists in Research Software Development

Melody L. Hammel
Department of Computer Science
Ball State University
Muncie, IN 47306, USA
mlhammel@bsu.edu

Lan Lin
Department of Computer Science
Ball State University
Muncie, IN 47306, USA
llin4@bsu.edu

2024 4th Interdisciplinary Conference on Electrics and Computer (INTCEC) | 979-8-3503-4945-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/INTCEC61833.2024.10603285

***Abstract*—For domain science researchers, software frequently acts as a tool to accelerate and enhance their research, opening up possibilities that were previously unimaginable. Often cases, it is the domain scientists who know about programming that take the task of software development. Lacking adequate training in software engineering, software development can become a daunting challenge. Thus, we present in this paper some widely-adopted and most effective software engineering practices a domain scientist who finds themselves developing software may need, all in one place. From basic *Git* usage to software testing and code specification, the information presented here should help any domain scientist to develop the best software they can, while spending as little time, effort, and money along the way as possible. We hope this information will prove beneficial to domain researchers involved in software development, empowering them to improve the reliability and maintainability of well-crafted research software.**

***Index Terms*—software engineering, software testing, continuous integration, code specifications, domain software quality**

I. Introduction

Software development projects come in all different shapes and sizes. Often, software development is done by professionals trained in software engineering to make the product the best they can. However, contractual software developers can be pricey. Domain scientists often find themselves in need of software to serve a domain-specific or research purpose without adequate funding to pay software professionals to do the job. Thus, they must make the software themselves, without any necessary software engineering training, and ensure the project goes smoothly and quickly and the software turns out good in the long run. To solve this dilemma, based on our experience helping domain science researchers develop a non-trivial piece of domain software [1]–[3], we present here what we believe to be *just enough* knowledge and expertise from the software engineering realm to help domain scientists with their software development. We surveyed and include in this paper topics on version control (Section II); software engineering best practices (Section III); and testing (Section IV). We also discuss the importance of developer-oriented code specifications in Section V. What we present here is by no means an exhaustive list of considerations in domain software development, but rather it serves as an initial stride toward addressing the myriad of aspects that warrant attention in developing high-quality, cost-effective, robust, reliable, and maintainable domain software.

II. Version Control

Long-touted by developers as a central pillar of software development, version control is seen by many as an essential part of any software project. Typical version control systems provide many useful benefits to any project in which they are implemented, including recovering working versions of projects if something breaks, tracking contributions by different developers collaborative project, easily managing changes made, and more. However, these systems can seem intimidating. When working with a version control system such as *Git* [4], familiarity with the command-line will help immensely.

A. Git *Basics*

As of now, *Git* [4] is the most popular *distributed* version control system out there. Other distributed (such as *Mercurial* [5]) and centralized (such as *Subversion* [6]) version control systems do exist, but are rarely used in modern, mainstream software development outside of specialized scenarios and in maintaining legacy systems. Due to its popularity, there are many different clients, methods, strategies and practices for using *Git*. Many *Git* clients with GUIs and graphical integration in most popular IDEs exist to make *Git* easier to work with, but most more-experienced developers prefer to use it in the command-line, for the most precise control over the system and direct knowledge over every command that is being executed, which can sometimes be hidden in the graphical solutions.

1) Necessary Vocabulary: There is a set of basic vocabulary used to talk about concepts in *Git* that must be understood by a learner.

- **Local repository**
  This is all the data that *Git* knows about. It holds the project's history, all branches contained within the project, commit messages, and information about every version of the code that currently exists and has ever existed. One can think of this as the "brain" of *Git*, contained within the `.git` folder under the root directory of your *Git* repository.
- **Working tree**
  This represents your local files. The code that you write, every part of the project that you commit, don't commit, ignore, stage, et cetera is included under the "working tree." Generally, *Git* commands utilize information stored in the local repository to update your working tree accordingly.
- **Remote repository**
  Your collaboration tool, where your files and changes are stored on the Internet. It is like a local repository, but on the cloud, and also including the current version of your working tree. To collaborate with a team, one needs to *push* commits and changes onto that repository and *pull* changes teammates have made down onto their local machine.
- **Upstream**
  Onto or from the remote repository. For example, used as an adverb, "push changes upstream"; used as an adjective, "set an upstream branch to track a local one"; or, used as a noun, "pull changes from upstream."

2) Usage: *Git*'s usage is based around a system of *commits*. When working in a *Git* repository, you can edit and save files like usual (typically code, but no matter what is contained with them), but the only way to truly save these changes into both history and your local repository is to *commit* them. Committing saves that change – the files' current state at the time of the commit – into the local repository as a snapshot in time, attributing the committer's name to the *diff* (the changes made to the file since the last commit) and saving a checkpoint of the file if anyone ever needs to restore it to an older state in the future. After that, the local repository will have a commit added into it, and you can *push* the new state of the local repository and working tree upstream to the remote repository if working collaboratively, e.g., on *GitHub*, *Bitbucket*, or *GitLab*.

3) Creating a Git *Repository:* For all our examples, *GitHub* [4] will be used, being the most popular remote repository hosting service. To create a repository on *GitHub*, you first must create an account – once registered, you must initialize a *Git* repository in a directory, make the initial commit, add a remote repository (which you must create on *GitHub*, shown in Figure 1) and push your changes upstream, setting the upstream branch to track your local branch. This sequence is illustrated in Table I.

4) Making Changes: After creating a *Git* repository in a folder, changes can be made and written into files as usual, but the only way to save these changes into your local repository is to commit them. In order to commit files, they must first be

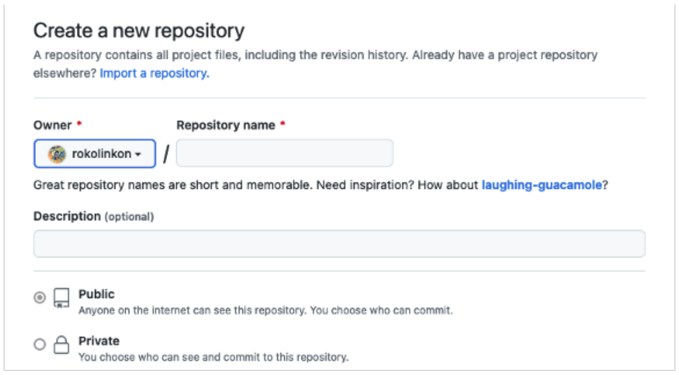

Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.
Owner * rokolinkon / Repository name *
Great repository names are short and memorable. Need inspiration? How about laughing-guacamole?
Description (optional)
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.

Fig. 1. Creating a repository on *GitHub*

staged. *Staging* means that operations in *Git* will be applied to them. Once staged, files can be committed. When committing, the committer should specify a commit *message* to explain what was changed in the commit. Once a commit has been added, if a remote repository exists, the local repository will now be one commit ahead of the remote. The change must then be pushed onto the remote repository. *Git* keeps track of the status of the local repository, which can be viewed at any time. The commands for these tasks are shown in Table I.

5) Staying Up-to-Date: Not only is it important to make sure your own changes are recorded and saved on the remote repository, but for collaboration to truly take place, you must also be able to view changes your colleagues on the same repository have made. To do this, you must *pull* changes from upstream to ensure you have the latest version of the code to work on. Using the `git pull` command retrieves information about changes in the remote repository on the current branch and integrates them into your own local repository, updating your working tree accordingly. To only update your local repository and *not* your working tree, you can use `git fetch`. Fetching is useful in scenarios where you have local changes and don't necessarily want to unconditionally overwrite them with a pull; fetching the changes first allows you to view the changes that were made on the remote repository so you can take care of your local changes before merging the changes into your working tree.

If not pulled frequently, code on your local machine can end up without changes that have already been made on the remote repository. Then, the next time you attempt to push your changes upstream, a *merge commit* must be made in order to merge your commit history with the upstream commit history, which have now diverged.

6) Merge Conflicts: If you and another developer are working on the same file at the same time (or you forgot to pull changes before beginning working), you may encounter a *merge conflict*. This means that the remote and local repositories have two different versions of the same file, and *Git* is unable to update the working tree without overwriting changes made by at least one commit. These can be difficult to overcome, given *Git* edits the file to place lines telling the individual to resolve the conflict where it occurred. In simple

TABLE I.
COMMANDS TO CREATE A *Git* REPOSITORY, MAKE CHANGES, AND STAY UP-TO-DATE

Command	Description
`git init`	Initialize a local repository
`git add <file>`	Stage the files for the initial commit
`git commit -m <message>`	Commit staged file(s) with the specified message
`git remote add remote <url>`	Add a remote repository named "remote"
`git push --set-upstream remote master`	Set up the remote "master" branch to track the local master branch
`git push`	Update the state of the remote repository to the local repository's state
`git status`	Display the current status of the local repository
`git pull`	Download commits from the remote repository and merge them into your working tree
`git fetch`	Retrieve information about the remote repository and how many changes are upstream

scenarios, one can just remove those lines to keep both sets of changes. However, it may end up not being so simple (and much progress can be lost if using GUI clients that provide the option to "accept local").

7) Ignoring Files: Sometimes, specific files should not be pushed to the remote repository, since they may be user- or machine-specific, e.g., build files, environment files, virtual environments, et cetera. These files that should not be pushed to the remote repository can be specified with a `.gitignore`. This is a file either at the root of a repository or in a user's home directory that specifies a list of files, directories, and patterns for *Git* operations to ignore. For example, when `git status` is run, files specified in the `.gitignore` will not be listed as changed, even if they have, and will remain unstaged when a whole directory is staged. Thus, these files will not be pushed to the remote repository, which removes the chance that a user pulling changes from upstream will have their own local configuration files replaced by another developer who pushed theirs.

The `.gitignore` itself must be committed to take effect, but an individual user can create a *global* `.gitignore` that lives in their home directory, specifying files and patterns to be ignored across all repositories on the user's machine. The benefit of this is that it does not need to be committed to the repository, in case an individual user wants to ignore files that they won't need to add to all future `.gitignore` files. The syntax of `.gitignore` is shown in Table II.

TABLE II.
`gitignore` EXAMPLES

Syntax	Description
`env.ts`	Exact file name: ignores that specific file
`build/`	Directory: ignores directory named "build"
`*.pyc`	Wildcard: ignores all files ending in `.pyc`
`# Ex`	Comment: for organizational purposes

B. Git *Advanced*

The basic features that *Git* provides are a good bare minimum. We cover here a few more we find to be most useful in most developers' day-to-day lives.

1) Branching: Branching is one of the main reasons *Git* became more popular than its competitors. Other version control systems may have branching systems, but *Git*'s is considered among the easiest to use and most intuitive. Branches can be used for many different things: protecting the master branch to ensure it is always production-ready, avoiding frequent local-remote merge commits when multiple people are working at the same time, organizing work and commits, and tracking progress on specific features according to a branching strategy (to be discussed in Section III-B). Once a branch is created, development can continue as normal, committing, pushing, and pulling. When you no longer have need of a branch, it can be merged back into the master branch, and all changes made on that branch will be merged with the history of the master branch. If you still want to keep the base branch around, the changes can instead be *rebased* onto the current branch to ensure all branches are up-to-date with the latest changes. Rebasing, as opposed to merging, edits the branch's history to include the commits from a different branch, whereas merging creates a new commit to add the changes in. Switching between branches updates your working tree accordingly – if changes were made on a branch that were not made on another, switching to the latter will see those changes missing from your files. A list of possible branch operations is shown in Table III.

2) Undoing Mistakes: In most university settings, this topic seems to be frequently left out. However, when changes are spread across files and subtle enough that they aren't easy to find, *Git*'s ability to turn back time can save a project on its last legs.

Git allows three different operations to undo changes: reverting, resetting, and restoring. *Reverting* refers to creating a new commit in the history that performs the inverse of a selected commit. For example, if a recent commit adds four lines and removes two, the commit that reverts that commit will add the two removed and remove the four added. However, this can cause a merge conflict if there have been commits made after the commit being reverted. *Resetting* refers to deleting commits from history, which allows for a mass deletion of work all at once, and thus can be rather dangerous. Importantly, providing a commit hash to reset removes all commits *up to but not including* the specified commit. By default, this does not update the working tree unless the option `--hard` is provided. *Restoring* is an oper-

TABLE III.
BRANCH OPERATIONS

Command	Description
`git branch`	List existing branches and highlight currently-selected branch
`git checkout -b <branch-name>`	Create a new branch called `<branch-name>` with the same history as the selected branch
`git switch <branch-name>`	Switch to the branch called `<branch-name>`
`git merge <branch-name>`	Merge `<branch-name>` into the currently-selected branch
`git branch -d <branch-name>`	Delete `<branch-name>`
`git push <remote> :<branch-name>`	Push deletion of `<branch-name>` to remote
`git rebase <branch-name>`	Pull commits from `<branch-name>` into selected branch

ation affecting only the working tree; it *restores* the working tree back to the state of the most recent commit. This is useful for getting rid of changes made since the last commit that have yet to be committed, for example to prepare for a pull that fixes the issues you were already working on. The commands to perform these operations are shown in Table IV. Anywhere a commit hash is provided can be replaced by the `HEAD~n` syntax, which represents the commit *n* commits ago in the log, zero indexed. So, in a commit history with hashes `f8d3118`, `22fd140`, `c1a7480`, and `1006444`, with `f8d3118` being most recent and `1006444` being the oldest, `HEAD~2` represents commit `c1a7480`.

3) Stashing: Rarely does anyone speak of stashing; it is akin to a hidden art among *Git* enthusiasts. *Stashing* a change or set of changes refers to "setting changes aside" without committing them so you can re-apply them to your working tree another time. It undoes all the uncommitted changes to your working tree and saves them in something called a *stash* in your local repository. This is useful if you have uncommitted changes that aren't ready to be committed yet but you need to work on something else for the time being, or pull from the remote repository, or try something else but save your current work. However, this can quickly grow complicated, as you can have multiple stashed changes at one time from multiple different branches and periods of time, stash specific files, chunks of files, et cetera.

By default, creating a stash with no extra arguments stashes all uncommitted *changes* to files. This means it does *not* stash untracked files (newly created files that have yet to be committed into the local repository) by default. The stash will be named with the latest commit at the time of its creation and the branch it was created on. However, it is much more helpful to give stashes descriptions so you know what you were working on in each specific stash. Stashes can then be either *popped* onto the working tree (which applies the changes from the stash back to your working tree and simultaneously deletes the stash) or *applied* to the working tree (which applies the changes from the stash and keeps it around, in case you need to apply those changes to multiple branches, for example). All stash commands, when not provided any argument, default to `stash@0`, or the first stash in your list (pronounced "stash at index zero"). Useful stash operations are shown in Table V.

4) Other Useful Commands: Git, of course, has far too many features to list all here. We have included a list of some more commands we find to be useful, also shown in Table VI.

- **Cherry-pick** *Cherry-picking* refers to taking individual commits from a branch and rebasing them onto a different branch. This can easily lead to merge conflicts, but this can be useful for things like bug hotfixes; if a developer fixes a bug on a branch that isn't master but the bug still exists on the master branch, that fix can be cherry-picked onto the master branch to ensure users aren't affected by it without rebasing *all* of the changes from that branch onto master before they may be ready.
- **Blame** Though the word has a rather strong connotation, *blame* refers to the *Git* feature that allows the user to check who wrote every line of code in a file and with which commit they did so.
- **Show** Sometimes, the blame of a file can become crowded. Use this command to view what was changed in a specific commit. Providing no argument defaults to the most recent commit.
- **Fetch** *Fetching* changes as opposed to *pulling* changes refers to updating your local repository without updating your working tree. In essence, this pulls information about what commits are upstream that are missing from your local repository without merging those changes into your local branch yet, allowing you to view what changes have been made (if any) and clean up your working tree accordingly (for example, stashing current changes made to pull new ones, if they exist).
- **Diff** Viewing the *diff* between two commits means viewing the difference between them – changes that have been made from one to the next. This can also be used with no arguments to see what changes have been made to your working tree since the last time you committed – for example, if you come back to your repository after a moment and forget what changes you made that you forgot to commit.
- **Log** The *log* is a surprisingly powerful tool. Being highly customizable to one's needs, it can provide exactly what information you need and nothing that you don't if you use it correctly. It presents precisely your needed information, down to formatting by placing fields manually using string interpolation or showing a graph of all the commits and how they arrived in the current branch using ASCII art.

TABLE IV.
COMMANDS TO UNDO MISTAKES

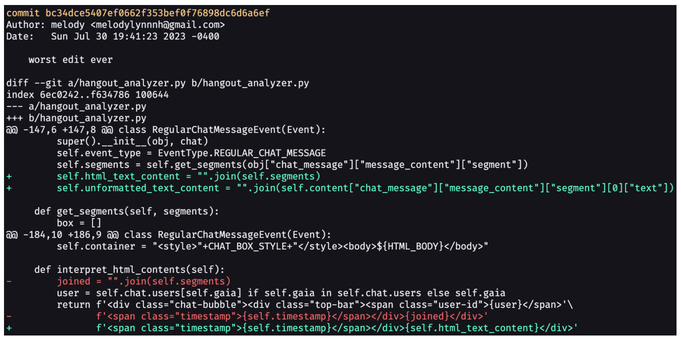
Command	Description
`git revert <hash>`	Undo the commit with hash `<hash>`
`git reset <hash>`	Remove all commits up to commit with hash `<hash>`
`git reset <hash> --hard`	Remove all commits up to `<hash>` and update the working tree
`git restore <file>`	Restore `<file>` back to its state at the most recent commit
`git push <remote> --force`	Push a recent reset, updating the remote commit history

TABLE V.
USEFUL `stash` COMMANDS

Command	Description
`git stash`	Save a new stash of all changes to files with a default name
`git stash save <description>`	Save a new stash with description `<description>`
`git stash list`	List existing stashes
`git stash pop`	Apply changes from stash to working tree and delete stash from list
`git stash apply`	Apply changes from stash to working tree and keep stash in list
`git stash show [stash]`	View `[stash]`'s diff (the changes included in the stash)
`git stash drop [stash]`	Remove `[stash]` from the list
`git stash clear`	Drop all stashes
`git stash branch <branch-name> [stash]`	Create branch `<branch-name>` using `[stash]`'s changes

TABLE VI.
MISCELLANOUS USEFUL COMMANDS

Command	Description
`git cherry-pick <commit>`	Rebase `<commit>` onto the current branch
`git blame <file>`	Display file `<file>` annotated with names and commit hashes in which each line was last edited
`git show [commit]`	Display changes made in `[commit]`
`git fetch`	Update the local repository from upstream without updating the working tree
`git diff`	View changes made to the working tree since the most recent commit
`git log`	View the log of commits
`git log --pretty=format:"(%h) %an - %ar: \"%s\""`	Print the `git log` with format (hash) name - time: "commit message"
`git log --oneline`	Display each commit in the log on one line
`git log -p`	Show changes made in every commit in the log
`git log --graph`	Show an ASCII-art graph of where each commit came from



```
commit bc34dce5407ef0662f353bef0f76898dc6d6a6ef
Author: melody <melodylynnnh@gmail.com>
Date:   Sun Jul 30 19:41:23 2023 -0400

    worst edit ever

diff --git a/hangout_analyzer.py b/hangout_analyzer.py
index 6ec0242..f634786 100644
--- a/hangout_analyzer.py
+++ b/hangout_analyzer.py
@@ -147,6 +147,8 @@ class RegularChatMessageEvent(Event):
         super().__init__(obj, chat)
         self.event_type = EventType.REGULAR_CHAT_MESSAGE
         self.segments = self.get_segments(obj["chat_message"]["message_content"]["segment"])
+        self.html_text_content = "".join(self.segments)
+        self.unformatted_text_content = "".join(self.content["chat_message"]["message_content"]["segment"][0]["text"])

     def get_segments(self, segments):
         box = []
@@ -184,10 +186,9 @@ class RegularChatMessageEvent(Event):
         self.container = "<style>"+CHAT_BOX_STYLE+"</style><body>${HTML_BODY}</body>"

     def interpret_html_contents(self):
-        joined = "".join(self.segments)
         user = self.chat.users[self.gaia] if self.gaia in self.chat.users else self.gaia
         return f'<div class="chat-bubble"><div class="top-bar"><span class="user-id">{user}</span>'\
-                f'<span class="timestamp">{self.timestamp}</span></div>{joined}</div>'
+                f'<span class="timestamp">{self.timestamp}</span></div>{self.html_text_content}</div>'
```

Fig. 2. A commit diff where some content was added and some was removed

III. BEST PRACTICES

Many problems in domain software development come from poor or no application of software engineering best practices. Though it may take time and effort to ensure these practices are put into place, the cost to solve the problems later on would be considerably higher as opposed to if these best practices were followed from the start.

A. Commit Strategy

Because collaborating on code projects is based around committing – and because one of the main problems that can be run into is merge conflicts – it is best practice to commit frequently and in smaller chunks. Committing changes to multiple files at once is generally frowned upon; if you have made changes in multiple places in the code and have yet to commit, it is better to use `git add -p` to stage files in parts for smaller individual commits. If you commit many changes at once, you are likely to miss changes upstream, ending up with a merge conflict. Additionally, commits should have descriptive commit messages so readers know exactly what was changed with each commit. This makes it easier to tell when a change was made without reading exact diffs. Used in tandem with other software engineering practices, this makes code changes easier to keep track of and makes collaboration easier as a result.

B. Branching Strategy

The main purpose of branching is *Master Branch Protection*. In general, the default branch created by *Git* is either the *master* or *main* branch. This is the branch that should always be kept production-ready; code should not end up in this branch unless it has been thoroughly tested, linted and reviewed, and assured to be of good quality.

To actually go about protecting the master branch in a code repository, there must be other branches that one can write code onto intsead of the master branch. There must be rules for how branches should be created and used so that branches have reasons to exist. Such we call a *branching strategy* – plenty different ones exist in software engineering to serve various purposes, but we propose that the most applicable to most software development scenarios is *feature branching*.

With feature branching, a new branch is created for every new *feature* that is being implemented into the software. For example, if a software exists that pulls data from an external server and a new caching system is to be implemented, a branch called "cache" could be created for this feature. After it is implemented and integrated into the software properly, the feature branch is merged into an intermediary development branch, where it undergoes tests and continuous integration to ensure it is of good quality. After a few features have been merged into the development branch and they are all ensured to be good (likely both via automated checks and manual code review), the development branch can be merged back into the master branch, where continuous deployment tasks are run to deploy the software to end-users.

But what if a bug is discovered in the master branch? In this scenario, feature branching extends with a *hotfix* branch. It is created off of the master branch when necessary to fix a bug and merged straight back into the master and development branches as quickly as possible. Then, if a feature branch needs this hotfix, the master or development branch can be rebased onto it. A possible visual demonstration of this branching strategy is shown in Figure 3.

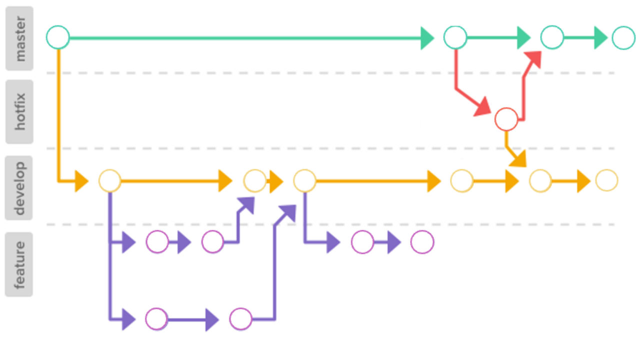


Fig. 3. An example of feature branching

C. Pull Requests

Since the master branch is to be protected from less-than-quality code and features are being developed on their own branches, we need to ensure that only code put through lots of review is able to make it onto the master branch. For this, we use *pull requests*. These function as a system of requesting code to be merged into a different branch. Their primary application for our branching strategy is to merge branches containing finished features into the development branch and for merging the development branch into the master branch. They serve as a *request* to *pull* code from the base branch into the destination branch. Pull requests, unlike merging with no review, allow developers to review the changes made in a branch and *approve* them or request changes before they are merged into the destination branch. A pull request also serves as a good place to run automatied checks and continuous integration, putting the code through a unit test suite and a linter to make sure there are no obvious bugs or problems. Pull requests can be set up in *GitHub* (and most other online *Git* repositories) to require a certain number of approvals before they can be merged and, in that sense, are somewhat akin to voting on a bill for code to be merged into a different branch.

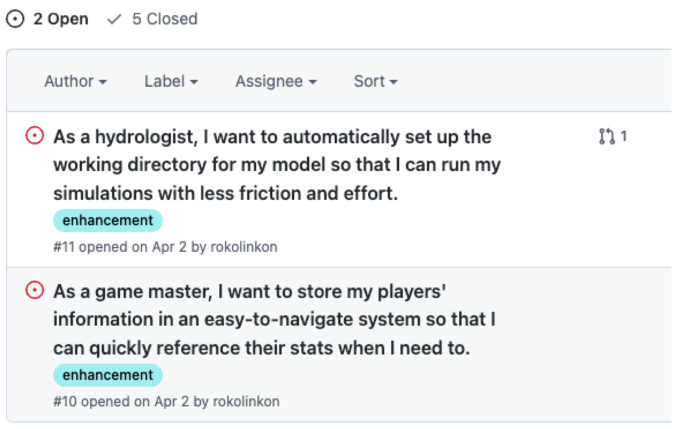
D. Issue Tracking

Issue tracking, also known as *card tracking* on occasion, is how bugs and feature assignments are tracked in large software projects. It involves a system of creating *issues* or *cards* and assigning them to different developers, using them to track, eponymously, issues, or tasks that are still to be done on the project. Having an issue tracking system in place means that everyone always knows what work still needs to be done and what work is already done on the project and, if used diligently, allows developers to avoid the situation where two people work on the same feature at the same time, or a feature is not taken care of by anyone on the team. With issue tracking, the first developer to begin work on a feature would claim the card, so other developers check the cards and will know not to take it. *GitHub*'s built-in *Issues* is an easy issue tracking solution to use, but dedicated issue tracking services exist, such as *Jira* [7] and *Trello* [8].

E. User Stories

A *user story* describes a use case for a software; it is a way of describing the features of a software from the perspective of a hypothetical user. For many, this can be an easy way to tell when a feature is done, as once the user story can be fulfilled in the software, that means the feature is complete. This has the potential to be dangerous, if checks aren't in place to test and lint the code (among other quality control tasks). Assuming those practices are properly followed, user stories can be an easy way to track feature progress, and can mesh well with issue tracking systems. Larger overarching issues can be created for user stories as features. Many smaller issues describe individual pieces that must be implemented as part of those features. Example user stories typically follow the format of "as a [profession], I want to [feature in the app] so that I can [accomplish something]." Some example user stories include

- As a hydrologist, I want to automatically set up the working directory for my model so that I can run my simulations with less friction and effort.
- As a game master, I want to store my players' information in an easy-to-navigate system so that I can quickly reference their stats when I need to.
- As a parent, I want to view cheap healthy recipes in an online catalog so that I can have more options to cook for my children.



As a hydrologist, I want to automatically set up the working directory for my model so that I can run my simulations with less friction and effort.
enhancement
#11 opened on Apr 2 by rokolinkon

As a game master, I want to store my players' information in an easy-to-navigate system so that I can quickly reference their stats when I need to.
enhancement
#10 opened on Apr 2 by rokolinkon

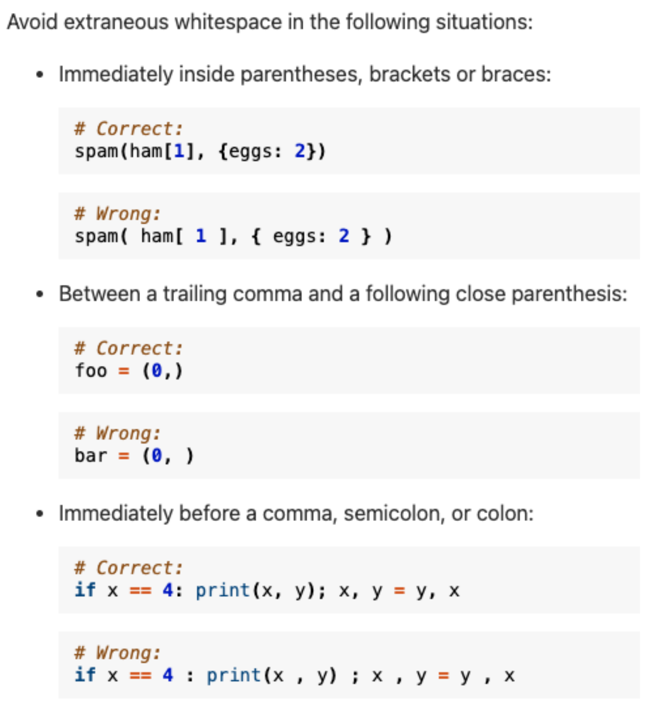
Fig. 4. User stories stored using *GitHubIssues*

F. Clean Code

In general, *clean code* is a set of standards to be followed when writing code to make it more readable, maintainable, and easy to write and understand. However, "more readable, maintainable, and easy" are subjective terms – what goes into these things can vary depending on whom you ask. For example, in Google's standards for TypeScript, they ban Automatic Semicolon Insertion and require all statements to be explicitly terminated with semicolons. However, in the TypeScript standards at the Ball State University Digital Corps, semicolons are removed by the linter and are considered bad practice. There are generally well-accepted standards for most languages that are considered best practice and which one to choose will depend on the project's needs and its members' preferences.

No matter what standard is being followed, there are some basic concepts that are always applied that are language-agonistic. They go toward making your code more readable in any language. Good naming conventions, the Don't Repeat Yourself (DRY) principle, object-orientation, and more can be applied to most modern programming languages. An example excerpt from PEP 8 [9] is shown in Figure 5.

1) Naming Conventions: Names of objects, functions, methods, classes, variables, types and anything else in code that can be named should be descriptive of what the thing is (but not too descriptive so as to make the name too long). Generally, the rule is that things should be explicitly defined in your code such that comments are unnecessary, and



Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

```
# Correct:
spam(ham[1], {eggs: 2})
```

```
# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
```

- Between a trailing comma and a following close parenthesis:

```
# Correct:
foo = (0,)
```

```
# Wrong:
bar = (0, )
```

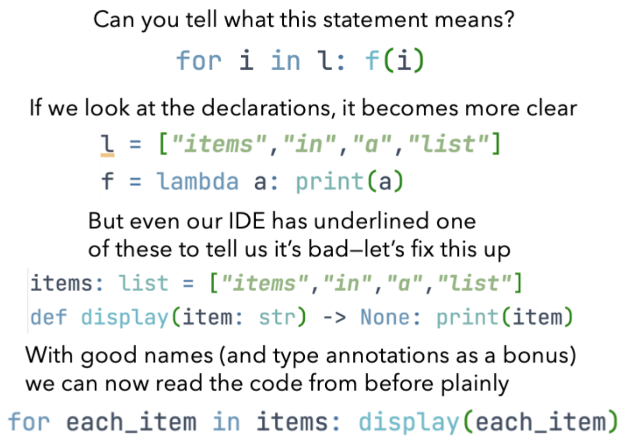
- Immediately before a comma, semicolon, or colon:

```
# Correct:
if x == 4: print(x, y); x, y = y, x
```

```
# Wrong:
if x == 4 : print(x , y) ; x , y = y , x
```

Fig. 5. An example clean code standard

individuals reading your code shouldn't have to read method implementations to know what they do. An example is shown in Figure 6.



Can you tell what this statement means?

```
for i in l: f(i)
```

If we look at the declarations, it becomes more clear

```
l = ["items","in","a","list"]
f = lambda a: print(a)
```

But even our IDE has underlined one
of these to tell us it's bad–let's fix this up

```
items: list = ["items","in","a","list"]
def display(item: str) -> None: print(item)
```

With good names (and type annotations as a bonus)
we can now read the code from before plainly

```
for each_item in items: display(each_item)
```

Fig. 6. An example of good naming conventions

2) DRY: Don't Repeat Yourself: When someone says "write DRY code," they're referring to the acronym of "Don't Repeat Yourself" [10]. Repeated code is replaceable by functions, variables, and classes, and refactoring repeated code helps make code reusable without cluttering the codebase. One defines functions to allow code to be reused, and classes to give structure to common constructs in code. There are many attributes of modern programming languages specifically designed for avoiding repetition in code, so writing DRY code is easier than ever.

G. Object-Orientation

Object-orientation is a paradigm of programming and writing code that is "object-oriented" – that is, code that is written to represent real-world objects. Admittedly, not everything in code is translatable to real-world objects, so objects can be difficult to decipher. Object-oriented code, when all its practices are followed and found in its purest form, is designed to be reusable. The code is written in such a way that common functions and attributes are grouped together into structures called *objects*, and common attributes are only known to the objects they inhabit. There are a few main principles of Object-Oriented Programming (OOP) that are considered best practice to follow and can improve the structure of code they are used in. First, however, some basic vocabulary should be understood:

- **Class**: The abstraction of an object defining its structure, attributes, and methods
- **Object**: The instantiation of a class with its (non-static) attributes defined and unique to that instance
- **Function**: A unit of code that takes parameters and returns a value that is not attached to any class instance
- **Method**: A function that is attached to an instance of a class and typically operates on attributes of that class
- **Static**: A variable or method of a class that is the same between all instances of the class and doesn't depend on the object being instantiated to exist

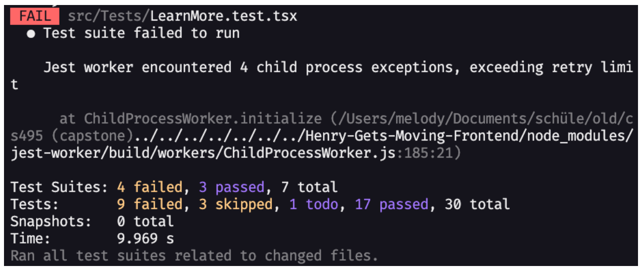
There are four key concepts of OOP that are considered its main principles (or "pillars"). *Abstraction* provides one of the fundamental ways to deal with complexity. An abstraction focuses on the outside view of an object and separates an object's behavior from its implementation. Turning code from low-level boilerplate into more readable and comprehensible code not only helps readers, but also writers of code to avoid repetitive work. This can be done by *abstracting* more complex lower-level tasks into methods and creating variables, classes, and structures to help group information together and give it meaningful names. *Encapsulation* provides a mechanism to hide internal implementation details from outside a module. This means that code outside the class should not be able to access and modify internal properties of the class. Encapsulation can be achieved using *private* and *protected* attributes and methods, common in object-oriented languages like Java and C++. *Public* attributes can be accessed and modified by any code. *Protected* attributes can only be accessed and modified within the class itself and within subclasses of that class. *Private* attributes can only be accessed within the class itself. In Java, by default, attributes without access modifiers can only be accessed within the same package. *Inheritance* provides a method of reuse by *subclassing*. A *child* class (or *subclass*) is created that *inherits* all the *public* and *protected* properties of its *parent* class (or *superclass*), while being able to be extended further. Finally, *polymorphism* supports many implementations behind a common single interface. A function or method operating on an object should also be able to operate on subclass instances without knowing it – just that the object it is operating on has the expected type.

H. Coupling and Cohesion

Coupling refers to the degree of interdependence between modules in an object-oriented system. It represents how much different modules are communicating with each other and using each other's methods and attributes. More communication and usage between modules means greater coupling, and more difficulty of editing one without affecting the other. Typically, in software design, high coupling is associated with low *cohesion*. Cohesion refers to the degree to which code in the same module belongs together. Low cohesion in a software system means that pieces of the same module aren't properly interacting with each other or have no real reason to be placed together (called *coincidental cohesion*). High cohesion is desired and can be achieved in a number of ways, suggesting that the question "why is this code grouped together?" can be given a sensible answer. The highest degree of cohesion, also called *atomic*, is rarely achievable and often cumbersome to get to – *functional cohesion* is a good substitute and the most common type of cohesion to strive for. *Functional cohesion* refers to grouping code together that shares similar functionality or all works together to serve a common purpose.

IV. Testing

Testing remains one of the most effective means to ensure the quality and reliability of a piece of software. It is much more than just the use of the software as an end-user to verify functionality. It involves the concept of writing code to test other code, typically in an automated fashion to make sure that an entire suite of software works as expected, according to a specification of correct behavior. Test automation allows tests to be run repeatedly without any user input. With continuous integration, developers can run tests with every change they make, constantly assuring that the changes don't break anything in the software. When tests are already in place for a feature before its implementation is begun, developers can use those tests as a benchmark for feature progress, considering the feature "done" when all the tests for the feature pass. An example test output with automated React [11] testing is shown in Figure 7.



```
FAIL  src/Tests/LearnMore.test.tsx
  ● Test suite failed to run

    Jest worker encountered 4 child process exceptions, exceeding retry limi
t

      at ChildProcessWorker.initialize (/Users/melody/Documents/schüle/old/c
s495 (capstone)../../../../../../../Henry-Gets-Moving-Frontend/node_modules/
jest-worker/build/workers/ChildProcessWorker.js:185:21)

Test Suites: 4 failed, 3 passed, 7 total
Tests:       9 failed, 3 skipped, 1 todo, 17 passed, 30 total
Snapshots:   0 total
Time:        9.969 s
Ran all test suites related to changed files.
```

Fig. 7. An example test output with automated React testing

A. Unit Testing

The lowest level of testing one starts with is *unit testing*. Here, software is broken down into the smallest unites you can code in, where one piece of production code corresponds to a few tests. It is effective in testing every facet of code, from success to failure to edge cases. Every small feature should be unit tested. Desiderata of good unit tests include but are not limited to:

- **No human intervention**
  Tests should be able to be exectued with only one command, without requiring the user to provide any inputs during the test run. This works in conjunction with good model-view separation – if the model and view are not kept separate, unit tests cannot be properly run without invoking some methods that will prompt user input.
- **No human interpretation**
  Test output should be easily interpretable so the reader knows what test failed and what needs to be changed in order to pass the test without having to read deep into stack traces. Generally, a well-established testing framework will suit this principle fine, as they typically exist with consistent output on passed and failed tests. Assuming your test names are good enough and tests are split up to test one thing at a time in isolation, you should be able to understand what is wrong just from reading the name of the test that failed and the error that occurred.
- **Test one thing**
  Unit tests should test only one *unit* of the software, which allows tests to be easily interpreted, since reading the name will give you a good hint as to exactly what failed. This also suggests that tests should not rely on each other to run properly – most test runners run tests out of order or in parallel, so relying on the order they run in to set up parameters for future tests can lead to problems and should be avoided.
- **Test for failure**
  As well as testing the code works with valid inputs and in edge cases, one should also make sure that the code in question fails correctly when given invalid inputs or run in abnormal scenarios. Code should be designed to fail in a specific way, as documented in the specification of correct behavior. This is also intended behavior that requires testing and verification. Most test runners support failure testing and allow testers to expect exceptions to be raised. It is important to ensure that the method is properly raising exceptions instead of continuing to run in an incorrect state.

B. *Test Design*

Before unit tests are written, a test design considers systematically how the behavior of an individual unit under test changes depending on the various inputs it is given. Inputs can range from arguments to a function to the files in a directory on the system the tests run on. Anything the unit uses to vary its output is considered an input. If the tester is the same person who wrote the implementation or specification, testing is easy, as they know exactly how the behavior should change based on the inputs. With proper abstraction, any specifically noted piece of functionality or designated case for the unit deserves a few test cases.

As a trivial example, we will use a hypothetical calculator class. The specification for this calculator states that it should not operate with numbers above 999 or negative numbers. Of course, this is not a very useful calculator, but it works well to serve our purpose here as an example for test design and unit testing. We can thus write tests based on the specification that we have of the calculator: it has methods `add` and `subtract`, which take two numbers as parameters and return an number. They function as one would expect based on the names. Thus, we can devise the following tests: addition with both operands and return value in the range [0, 999], addition with either operand or return value being negative, addition with either operand or return value above 999, subtraction with both operands and return value in the range [0, 999], subtraction with either operand or return value being negative, and subtraction with either operand or return value above 999. We also need to test that correct exceptions are raised when calling either function with a parameter that is not a number – this ensures that we have comprehensible error messages that will be useful to end-users to help figure out what they did wrong.

C. *Integration Testing*

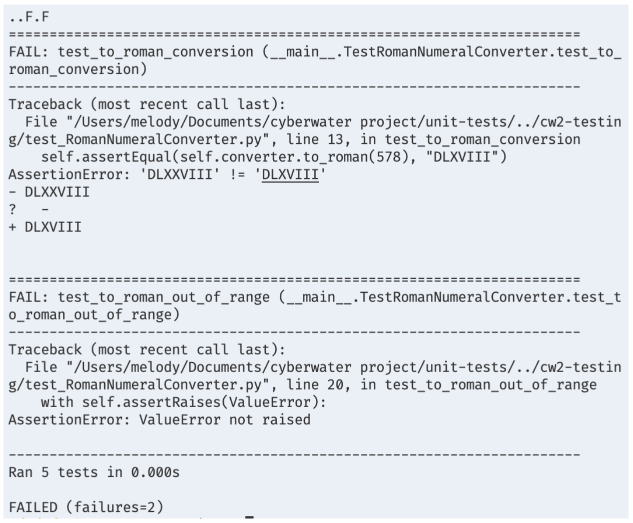
Integration testing is a higher level of testing than unit testing that involves putting units of the software together as they would typically be used to test the integration of those units. This helps ensure that not only do these parts work on their own in isolation, but they can properly communicate with one another in order to perform a greater overarching task that the software is designed to complete. For some software, this can involve *sanity testing*, where the inverse of a function is called on the function itself, ensuring that the initial input is the same as the result received back. For example, in the calculator mentioned earlier, a sanity test would call `add(subtract(x, 1), 1)` to test that these two inverse functions, when called in a sequence, can return the same number back.

D. *Mocks*

Mocks are used in unit testing when code does something it should not do in test execution, for example connecting to an external server. If done every single time tests are run, this could incur usage costs, make tests run significantly slower, or affect live data on the server that we don't want to interfere with. Many testing frameworks support mocks by default, but for languages that don't, there are typically workarounds. Python, for example, supports *object-method replacement* [1], in which methods and attributes of objects can be replaced within testing code to versions of them that don't call to external servers.

E. *Test-Driven Development*

Test-Driven Development (TDD) is a software engineering practice that uses tests as a basis for developing code. It can take many different forms, depending on how pervasively it is implemented in a software project, from “red, green, refactor” [12] to test coverage. TDD stipulates writing tests for a function or feature to be developed before any code is written. It interweaves coding, unit testing, and refactoring (design), and allows unit tests to serve as a functional specification for how the feature should work. Once the tests pass, the feature can be considered “done.” This form of a developer-oriented specification can help developers figure out what they are doing along the way and exactly what is going wrong if their code isn’t perfect. An example unit test output is shown in Figure 8. For example, with the calculator we have mentioned, we could write the tests designed in Section IV-B before implementing any of the code for the calculator. Then, as we implement the code, we cover both valid and invalid scenarios, until all designed tests pass, and the feature is complete.



```
..F.F
======================================================================
FAIL: test_to_roman_conversion (__main__.TestRomanNumeralConverter.test_to_
roman_conversion)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/melody/Documents/cyberwater project/unit-tests/../cw2-testin
g/test_RomanNumeralConverter.py", line 13, in test_to_roman_conversion
    self.assertEqual(self.converter.to_roman(578), "DLXVIII")
AssertionError: 'DLXXVIII' != 'DLXVIII'
- DLXXVIII
?   -
+ DLXVIII


======================================================================
FAIL: test_to_roman_out_of_range (__main__.TestRomanNumeralConverter.test_t
o_roman_out_of_range)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/melody/Documents/cyberwater project/unit-tests/../cw2-testin
g/test_RomanNumeralConverter.py", line 20, in test_to_roman_out_of_range
    with self.assertRaises(ValueError):
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 5 tests in 0.000s

FAILED (failures=2)
```

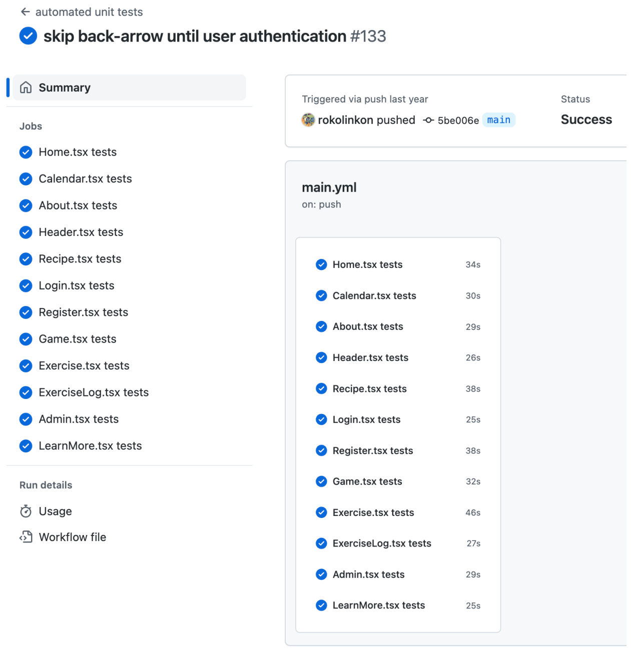
Fig. 8. The tests beginning to pass as we implement functionality, suggesting what we still need to do

F. Test Coverage

Test coverage allows unit tests to be run with some basic statistics provided, conveying information about not only passed and failed tests, but also how much code is *covered* by the tests. If a line of code is *covered*, that means the test suite has executed that line of source code, and if the tests covering the line pass, we can gain confidence in that it is roughly in working order. Beware that high test (code) coverage doesn’t necessarily mean that a test suite is good – it may still not cover enough edge cases or failure cases to ensure that everything works as expected. Just because a line of code is covered at least once does not mean it has been adequately tested, but it is still a useful metric to have. For larger projects, it is generally accepted that 80% is a good number to strive for. Besides code coverage, test coverage can also mean requirements coverage, model coverage, et cetera.

G. Test Automation and Continuous Integration

Continuous integration plays a crucial role in the automated quality assurance of code. The integration process is automated. Each integration triggers an automated build and a set of tests to run. For code that is assured to be in working order, it can be automatically merged into the master branch, allowing multiple developers to easily collaborate on a project. Typically it involves automatic test and linter running upon a certain action being taken, e.g., a push to a remote repository or a pull request being created. There are many ways to implement continuous integration. One way is to use *GitHub Actions*, which allows developers to write code in YAML [13] that defines when *jobs* should be run to take certain actions on the code in the code base. Then, *GitHub* displays metrics about passed and failed tests on commits where the code was pushed or on the pull request screen, as shown in Figure 9.



← automated unit tests

✓ skip back-arrow until user authentication #133

Summary

Jobs
- ✓ Home.tsx tests
- ✓ Calendar.tsx tests
- ✓ About.tsx tests
- ✓ Header.tsx tests
- ✓ Recipe.tsx tests
- ✓ Login.tsx tests
- ✓ Register.tsx tests
- ✓ Game.tsx tests
- ✓ Exercise.tsx tests
- ✓ ExerciseLog.tsx tests
- ✓ Admin.tsx tests
- ✓ LearnMore.tsx tests

Run details
- Usage
- Workflow file

Triggered via push last year
rokolinkon pushed 5be006e main

Status
Success

main.yml
on: push

✓ Home.tsx tests	34s
✓ Calendar.tsx tests	30s
✓ About.tsx tests	29s
✓ Header.tsx tests	26s
✓ Recipe.tsx tests	38s
✓ Login.tsx tests	25s
✓ Register.tsx tests	38s
✓ Game.tsx tests	32s
✓ Exercise.tsx tests	46s
✓ ExerciseLog.tsx tests	27s
✓ Admin.tsx tests	29s
✓ LearnMore.tsx tests	25s

Fig. 9. Continuous Integration in *GitHub*

Continuous integration works well in development projects that have branching strategies that enable it to be run before changes are merged into the master branch – it serves as a reasonable roadblock for less-than-quality code to make it into the repository or be reviewed by a human developer, since the committer will know right away whether their code needs to be fixed or not.

V. Code Specifications

Code specifications are developer-oriented documentation describing the functionality of a piece of code and providing information necessary for a developer who may use this code in writing new code. This includes testers, who will need to know what the intended input and output is of the code they are testing in order to test it properly. A specification should be developed before the code is even written, so it can be set in stone and functionality doesn't change much during development. The specification should contain all the information necessary to write thorough unit tests for a piece of software, or to implement the software. If the functionality changes over time, the specification should be updated as well. Specification is not a one-time document; it should evolve with the software and will continue to be a useful artifact of the development process. It serves as a reference for how the functionality of a module has evolved over time, and can help develop the code that the specification is for in the first place. If the specification is good, code implementation is just a matter of following the specification, with all the intended behavior already carefully thought-out.

Code specification should include, at the bare minimum, methods, parameters and their types, return values and their types, instance variables, static variables and methods, and side effects. Your code will be impossible to use if no one knows how to call your methods without receiving an error, so usage examples are key to include. Similarly, a user also needs to know how *not* to call your methods. Including exceptions that the method raises in its specification is *vital* for users to know what they are doing wrong and why. For example, using the calculator example from Section IV-B, we could write a simple specification as follows:

- `add(augend, addend)`
  Params
  `augend: int | float`: the number to be added to
  `addend: int | float`: the number to add to `augend`
  Exceptions
  `ValueError` if either input parameter or the result is greater than 999
  `ValueError` if either input parameter or the result is less than 0
  `TypeError` if either input parameter is not an `int` or `float`
  Returns `int | float` – the sum of the two input arguments, `augend + addend`
  Examples
  `add(1, 2)`: returns `int(3)`
  `add(1.0, 2.2)`: returns `float(3.2)`
  `add(998, 2)`: raises `ValueError`
  `add(-1, 2)`: raises `ValueError`
  `add("1", "2")`: raises `TypeError`
- `subtract(minuend, subtrahend)`
  Params
  `minuend: int | float`: the number to be subtracted from
  `subtrahend: int | float`: the number to subtract from `minuend`
  Exceptions
  `ValueError` if either input parameter or the result is greater than 999
  `ValueError` if either input parameter or the result is less than 0
  `TypeError` if either input parameter is not an `int` or `float`
  Returns `int | float` – the difference of `subtrahend` subtracted from `minuend`, `minuend - subtrahend`
  Examples
  `subtract(2, 1)`: returns `int(1)`
  `subtract(2.2, 1.0)`: returns `float(1.2)`
  `subtract(1000, 2)`: raises `ValueError`
  `subtract(2, 3)`: raises `ValueError`
  `subtract("23", "6")`: raises `TypeError`

This specification includes all the methods of the hypothetical class, their parameters and types, as well as exceptions that can be raised by each method and how to avoid them. It also includes return values, their types, and descriptions about how the methods reach those return values, plus usage examples as to how the methods would respond to various inputs. A hypothetical developer now has everything they need to start using these methods.

VI. Conclusion

We present in this paper some best software engineering practices that we find to be necessary and useful for the smooth development of domain software of acceptable quality. The practices we outlined will help ensure that reworking and fixing problems that crop up later in development be kept minimal. Such practices are widely adopted in the software industry for a reason – employing them significantly enhances the overall quality of the software and, at the same time, has substantial savings in terms of budget, time, effort, and other valuable resources. Although what we have endeavored to put together is in no sense a complete list, it provides a starting point to prepare domain science researchers engaged in software development towards the goal of producing high quality research software cost-effectively.

VII. Acknowledgments

This work was generously funded by the National Science Foundation (NSF) under Grants 1835602 and 2209834.

References

[1] L. Connelly, M. Hammel, B. Eger, and L. Lin, "Automated unit testing of hydrologic modeling software with CI/CD and Jenkins," in *Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering*, 2022, pp. 225–230.

[2] M. Hammel and L. Lin, "Assuring domain software quality through workflow testing and specification," in *Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering*, 2023, pp. 37–44.

[3] L. Connelly, M. Hammel, and L. Lin, "Leveraging best industry practices to developing software for academic research," in *Proceedings of the 7th International Conference on Management Engineering, Software Engineering and Service Sciences*, 2023, pp. 7–13.

[4] E. Don, *Git Prodigy: Mastering Version Control with Git and GitHub*. Independently published, 2023.

[5] B. O'Sullivan, *Mercurial: The Definitive Guide, Illustrated edition*. O'reilly & Associates Inc, 2009.

[6] M. Mason, *Pragmatic Guide to Subversion (Pragmatic Programmers)*. Pragmatic Bookshelf, 2011.

[7] G. Cantrell, *Automate Everyday Tasks in Jira: A practical, no-code approach for Jira admins and power users to automate everyday processes*. Packt Publishing, 2021.

[8] B. Joiner, *Supercharging Productivity with Trello: Harness Trello's powerful features to boost productivity and team collaboration*. Packt Publishing, 2023.

[9] "PEP 8: The Python Style Guide," https://pep8.org/.

[10] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[11] M. Schwarzmüller, *React Key Concepts: Consolidate your knowledge of React's core features*. Packt Publishing, 2022.

[12] A. Mellor, *Test-Driven Development with Java: Create higher-quality software by writing tests first with SOLID and hexagonal architecture*. Packt Publishing, 2023.

[13] M. Soni, *Hands-on Pipeline as YAML with Jenkins: A Beginner's Guide to Implement CI/CD Pipelines for Mobile, Hybrid, and Web Applications Using Jenkins (English Edition)*. BPB Publications, 2021.