Leveraging Best Industry Practices to Developing Software for Academic Research

Levi T. Connelly, Melody L. Hammel, Lan Lin
Department of Computer Science
Ball State University
Muncie, IN 47306, USA
{ltconnelly, mlhammel, llin4}@bsu.edu

***Abstract*—*Best industry practices in software development are vital to the success of a project. When these practices are not well-applied, the development process can be severely hindered, and the final product can be of poor quality as a result. Implementing techniques for managing source code: version control, issue tracking, a branching strategy, a pull request strategy, a coding standard, unit testing, CI/CD, and automated testing, are not just widely-used industry practices for no reason. Thus, we took to applying these practices to a development project for research designed to reduce user time and effort in hydrologic modeling studies, CyberWater. The software package is built on legacy software and the development team is made up of a wide variety of people from various backgrounds, not all computer science. Applying these best industry practices to their development project has made their lives easier and the final product better. We report our experience in this paper and hope it provides some useful suggestions to domain scientists in an academic setting regarding how to develop high-quality research software.*

Keywords-source code management; version control; issue tracking; branching strategies; pull requests; coding standards; clean code; automated testing; unit testing; CI/CD; specifications; software quality

I. INTRODUCTION

Many industry conflicts are a byproduct of poor industry practices. Some of these conflicts, such as programming errors or mistakes incurred through continuous development on a repository, can be avoided by having a distinct guide for software developers to follow such that focus can remain on development and research. With a template already in place for the practices the developers should be using for the programming and maintenance process, more time is allotted for other parts of the development process that take more time and can help to make the product better. This also significantly increases efficiency - with less time spent wrestling with the problems brought about by poor industry practices, more time can be spent actually developing the product.

Domain scientists outside of the fields of computer science and software engineering are typically given funding to create a software project, but typically not *enough* funding is given to hire experts in software development, so they end up having to do it themselves. With little knowledge on industry practices in software engineering and little experience in developing software, conflicts can arise, slowing down the development process and resulting in a possibly worse product overall. Domain scientists may know how to write domain software, but they typically are not systematically educated on software engineering practices and tools, leading to much more effort needing to be put into the project than is necessary should best industry practices in software development be applied.

II. THE SOFTWARE ENGINEERING PROBLEM PRESENTED WITH *CYBERWATER*

The *CyberWater* project [1] was created with the goal of creating a new cyberinfrastructure with open data, open modeling framework software; as a result, the project is expected to reduce the user time and effort required for hydrologic modeling studies, allowing related discoveries to be made sooner. The project team includes hydrologists, climate experts, meteorologists, computer scientists and CI experts, from multiple universities and CUAHSI, who collaborate closely to ensure *CyberWater* will engage the broad communities for domain scientists' benefits.

One software engineering problem presented with *CyberWater* was that there is a lot of moving parts that comprise *CyberWater*; therefore, it was expected that without some grounding in positive industry practices such as automated testing and version control workflows that errors could slowly make themselves known in the project. To mitigate this, Ball State was put in charge of managing how development should be made on the project and what tools should be used to design and implement automated tests for *CyberWater*.

III. OUR EXPERIENCES IN APPLYING SOME BEST INDUSTRY PRACTICES

A. Improving Workflow and Source Code

Taking care of source code can be very cumbersome. There are a lot of methods that industries use to ensure that functionality is communicated from the developer to the end-user such that the intermediary steps require the code to be refined and evaluated before reaching its user. For *CyberWater*, this meant creating an environment to give the developers assurance that the code they were writing was considerably less likely to fault once it had reached its end-user. By implementing a steady and explicit workflow, enforcing version control and issue tracking, prompting code to be held for review, holding that code to a professionally proposed standard, and implementing automated pipelines to test the code before releasing it, we could prevent a considerable number of bugs from being introduced into the final releases.

B. *Enforcing Version Control and Issue Tracking*

Version control and issue tracking were a necessary set of industry practices that needed to be introduced to the project. It is not feasible to work on a smaller project in size without a fundamental understanding of version control, let alone this project. Version control is a must if multiple developers are going to work on a single repository at the same time. Having version control ensures that features and tasks can be split up into sections and merged into a development branch such that work neither lost nor stunted.

Issue tracking is also vital, as it introduces a medium through which bugs and errors can be monitored and settled. As per best industry practices, it is expected that bugs and errors do not go ignored, and that there exists a system through which these issues can be mitigated. The issue tracking system allows a user to communicate directly with the developers such that progress on the repository can be made in a timely fashion. There are many platforms that can be used for issue tracking, such as *Jira* [2], which we initially looked into but dropped due to limited budget. We ended up using *Bitbucket* [3] for version control, and its provided issue tracking feature called *issues*. Working in tandem with the issue tracking system, just as with version control, to have an organized way to manage in-progress fixes for bugs and to merge them back into the production-ready product, should be a branching strategy.

C. *Defining a Branching Strategy*

One of the hassles of version control is asserting that the means by which branches are created, merged, and removed implies that no work will be lost while developers work on separate features at the same time. Setting up a branching strategy allows a team of developers to be certain that their work is not only consistently tracked and implemented, but also that the versions they release are always in a production-ready state.

The key to a good branching strategy is setting up particular but arbitrary feature branches that then get merged into a development branch. Through this development branch, where integration occurs, we can move passing code into a feature-branch or master-branch such that all code in the master branch is in a production-ready state. The main idea of having multiple branches is so that no non-functional or non-production-ready code makes it into the master branch. Thus, it can always be assured that the master branch is free from known issues. If an issue arises or is brought to attention with code that is already in the master branch, a hotfix branch can be forked from the master branch. The use cases are typically if an easily-exploitable bug was found in the code of the master branch or if the app is unresponsive or breaking. Post-hotfix, the branch is merged into both the master and development branch - this ensures that no one working on the development branch is attempting to work around a bug that has already been fixed, and that the master branch stays production-ready and issue-free. Of these branches, the only two that remain permanent are the master and development branches - feature, hotfix, and release branches can be safely deleted after merging with no harm to the repository. Figure 1 illustrates our proposed branching strategy.

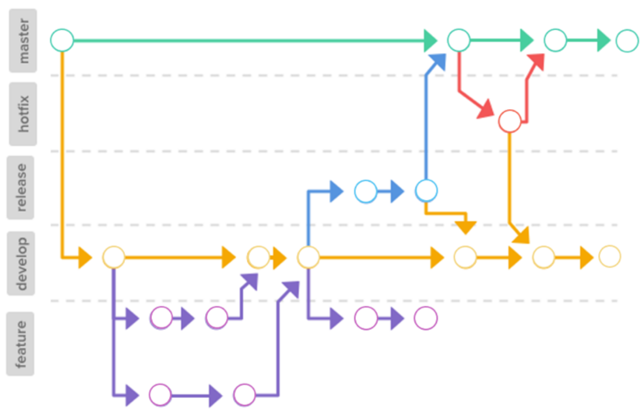


Figure 1: Our proposed example branching strategy

D. *Defining a Pull Request Strategy*

Pull requests are an essential part of software development in industry, as it creates an environment for the code to be reviewed in an efficient and professional manner. Pull requests were designed with merge conflicts in mind such that an administrator, or group of administrators, of the repository could mitigate conflicts by reviewing a small subset of a developer’s code and decide whether or not to merge that change into the preexisting codebase. They provide a simple, web-based way for developers to submit their work, and a similarly simple system for administrators to review and possibly implement changes made by developers. This also allows for less backtracking on old code, since code is implemented and merged into the correct branches in small intervals, making it easier to review and catch mistakes.

Pull requests are much like voting on a bill that, when passed, will change the existing logic of the system dependent on what adaptations you make to it before sending it on its way upstream. This fundamental part of industry practices ensures that unkept code does not make its way into release by placing responsibility on the administrator and accountability on the developer. This leaves more incentive for the developer to abide by coding standards when they make small adaptations to the code.

Pull requests are typically used when a developer has made changes that will affect the release. Thus, it raises their code for review so that others working on the project can make sure it is in good quality. Pull requests should be used once a developer has finished cleaning and optimizing their code, and are relatively sure that it is in a state where the project can "pull" changes from their fork. Commonly, developers make the mistake of "lazy merging," where their code is not thoroughly reviewed before making a pull request. Developers should ensure that they have tested every feature and bug fix in the branch they are attempting to merge, staying thorough to avoid breaking the master branch. They can also be used when making changes to a project of which a developer is not a part - for example, patching an open-source project on GitHub [4]. Relative to the branching strategy mentioned in the prior section, they should be used when merging from development to release, release to master, and hotfix to master. Pull requests should not be made from the master branch - this is where we want the code to be merged.

Although pull requests sound rather similar to peer reviews, they differ in some notable ways. First, peer reviews involve reviews by multiple users simultaneously. With pull requests, administrators and developers are given time to review the code individually, on their own time. Peer reviews, as well, are a direct form of feedback - suggestions will be given directly to the code author. With pull requests, the feedback is more indirect - they can be rejected with reasons given as to why, which the developer must read, interpret, and fix on their own time. Peer reviews are appropriate for only large releases, typically, whereas pull requests are appropriate for any release, since they don't require simultaneous attention of multiple project members. Finally, peer reviews must be enforced by the project team, whereas pull requests are built into most online repository services, like GitHub and Bitbucket.

E. Defining a Coding Standard

It is appropriate to have all of the developers on board with the same coding standards. One of Ball State's focuses during the *CyberWater* project was asserting a coding standard for the developers of the project. Given that the language for this project was primarily Python, the focus aimed towards closely aligning the habits of the developers with the PEP 8 style guide [5]; however, it is expected that for any given language, there exists a community that finds the most appropriate standards for a given language and implements these standards into a linter that can be used by each developer on a given project to keep their styles consistent across the repository.

General clean code practices are also given regarding coding standards. Abiding by the concept of single-indentation, or maintaining abstraction and complexity in a given method, or even simply making your variable names self-explanatory are essential principles for best industry practices. Robert C. Martin goes over this in great detail in his *Clean Code* book [6] released in 2008. The focus of clean code is to ensure that maintainable code is delivered during development that will be legible once time has passed such that a lack of documentation would not heavily inhibit the workflow of the project were adaptations needed to be made to that code later. If a bug is later on discovered in a piece of code, and that code follows no clean coding standards, the code will be much harder to read in order to discover where the bug lies, causing extensive time to be lost simply trying to understand what each line of code means, what variable names mean and are referring to, what the side effects of a function are, and various other issues.

Beyond the PEP 8 standards of naming conventions and how many lines to have between methods, the much more important clean code standards to follow are ones involving abstraction and descriptive (but not overly detailed) naming schemes. If a method is named poorly, a user might have to spend time looking over the method's code to see what it actually does and why it is used in a specific *other* method – this leads to the possibility of a developer having to backtrack through miles of code just to figure out what the purpose of one single method call is. Similarly, this could get even more complex if the variable names are inadequate. Variable names should be descriptive of their purpose, rather than difficult-to-understand acronyms or entirely nameless, like *x* or *a*. The purpose of variables is to give a higher-level name to a value in programming – if a developer doesn't give their variables useful names, then there's little point in using a variable at all, when everything could essentially just be anonymous.

For example, take this method (shown in Figure 2) designed to sum up two instance variables of a particular but arbitrary class, then yield an original and a result as a generator.

```python
def add_stuff(self):
    self.stuff1 += self.stuff2
    yield stuff2, stuff1
```

Figure 2: A method designed to sum up two variables

This method takes no parameters – how is a given user to know what is actually happening here? What are these variables being added? What is the purpose of this method? Some of this could be communicated if both the method itself and the variables were changed to be more descriptive, as shown in Figure 3.

```python
def calculate_total_balance(self):
    self.current_balance += self.deposit
    yield deposit, current_balance
```

Figure 3: The new-and-improved method and variable names

Now the purpose of the method is clear, and users reading the code can infer that these variables of the class should already have been set in some way before this method was called.

IV. Our Experiences in Testing

Having an explicit and reliable workflow is nice, but if there doesn't exist a system to require the environment to filter bugs before releasing it, then the workflow is better defined as an unnecessary set of extra steps for the developer. Continuous Integration / Continuous Deployment (CI/CD) pipelines ensure that when code is pushed to a given branch, preferably the integration branch, it can automatically be migrated to a higher-level branch where it can then be pushed into production. By enforcing that automated testing of the development code be in charge of what code was released for production, we could assert that production code always passed our given tests. We were able to assert this using an open-source tool called *Jenkins* [7].

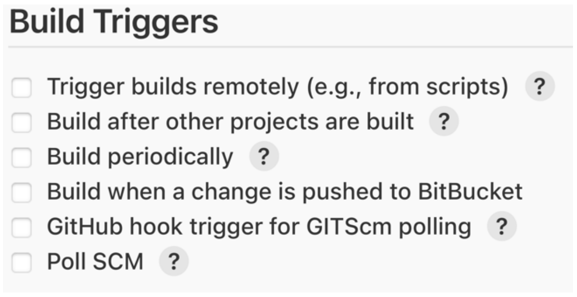
A. *Using* Jenkins *for CI/CD*

Jenkins was one of the most useful tools for the *CyberWater* project. Although alternatives were available for automated testing, like *GitLab* [8] CI/CD tools and *Atlassian Bamboo* [9], the extensive work that has been done on *Jenkins* and the fact that it is open source made it a viable candidate for what we needed to use it for. Many of the extensions made available through *Jenkins* simplified the process through which automated testing could be performed on the *CyberWater* project.

Some of the extensions available for *Jenkins* that simplified our experience were tools like the Environment Variable Injection extension which wrapped logic for modifying the `Path` variable on Windows machine so we could make our `Path` variables relative to the machine the project was being run on. This was vital given that our project required we access the Python distro and packages contained within the project that we downloaded for *VisTrails* [10] (an open-source scientific workflow and provenance management system used by *CyberWater*) and *CyberWater*.

Setting up *Jenkins* is simple. By downloading the *jar* or *war* files necessary to get the server started, you can execute those files with Java and start a server locally on the machine it is being executed from. Next, go through the account set-up and configure the repository you want to target using *Jobs*. This was how our team was able to set up *Jenkins* with our *Bitbucket* repository after configuring the credentials for an administrative account monitoring the repository.

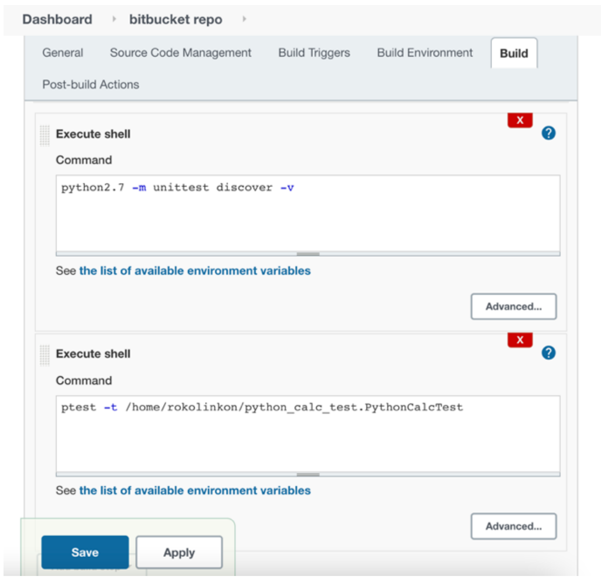
Jenkins jobs can also be run automatically, by setting up *Build Triggers* to determine when tests are run. The notable option we utilized was 'Build when a change is pushed to BitBucket,' shown in Figure 4. Figure 5 shows how we set up build steps in a *Jenkins* job.



Build Triggers

- ☐ Trigger builds remotely (e.g., from scripts) ?
- ☐ Build after other projects are built ?
- ☐ Build periodically ?
- ☐ Build when a change is pushed to BitBucket
- ☐ GitHub hook trigger for GITScm polling ?
- ☐ Poll SCM ?

Figure 4: Build Triggers in *Jenkins*



Execute shell

Command

```
python2.7 -m unittest discover -v
```

See the list of available environment variables

Advanced...

Execute shell

Command

```
ptest -t /home/rokolinkon/python_calc_test.PythonCalcTest
```

See the list of available environment variables

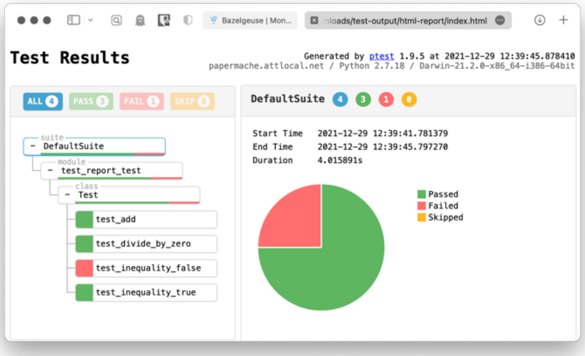
Advanced...

Figure 5: Setting up build steps in a *Jenkins* job

B. *Automated Unit Testing Using Python* Unittest *or* Ptest

One of the final necessary steps for implementing proper automated tests into the project was finding a suitable unit test library. *Ptest* [11] proved to be one of the best options, despite the fact that the project was locked in Python 2.7, because much of its development was done early on before the deprecation of Python 2. In addition, the Python library simplified the means by which test could be written by utilizing decorators to denote tags, groupings, setup-teardown practice, and whether to run the tests concurrently or in parallel.

Ptest was also an exceptional wrapper for basic unit tests because it allowed for a clean visual output of passing or failing tests by graphing these rates and outputting them to a stylized HTML file (see Figure 6). By linking the location of the outputs and using *Jenkins* automated scripts to give them a unique location each time a test was run, we could retrieve a unique graphical output for all passing tests through *Jenkins* via *Ptest*.



Test Results

```
Generated by ptest 1.9.5 at 2021-12-29 12:39:45.878410
papermache.attlocal.net / Python 2.7.18 / Darwin-21.2.0-x86_64-i386-64bit
```

ALL 4 PASS 3 FAIL 1 SKIP 0

```
suite
- DefaultSuite
  module
  - test_report_test
    class
    - Test
      test_add
      test_divide_by_zero
      test_inequality_false
      test_inequality_true
```

DefaultSuite 4 3 1 0

```
Start Time   2021-12-29 12:39:41.781379
End Time     2021-12-29 12:39:45.797270
Duration     4.015891s
```

Figure 6: *Ptest* HTML test report

Unittest [12], however, proved to be ultimately more useful in the end, as it integrated well with CI/CD and test automation, especially within *Jenkins*. *Ptest* relies primarily on the developer reading the output and is more focused toward an individual developer testing their code manually. *Unittest* has one key functionality that makes it viable for automated testing on *Jenkins*: `AssertionError`. Since assertions in *Unittest* are essentially just functions that compare the output of the first argument to the other, throwing an exception if the comparison is false, a failing test in *Unittest* can be detected by *Jenkins* as a failed build. Failed tests in *Ptest*, on the other hand, are marked only in the terminal output of the tests and in the GUI test report - since no exceptions are thrown, *Jenkins* sees nothing out of the ordinary and will assume, failing tests aside, the build to be successful. Therefore, since *Unittest* utilizes exceptions to communicate test results, allowing them to be recognized through automated test running systems, we decided to continue using *Unittest* moving forward. *Ptest* could technically still be used, if a system was designed so that an exception is raised when a test fails, but if no one is ever using the graphical test report, the extra hassle isn't necessarily worth it.

Though considering all of this, none of the backend unit testing would have been feasible had there not been a separation between backend and UI in *VisTrails*. Luckily, since the project is built in Python, object-method replacement was an option by using the dictionary structure of instantiated Python objects and swapping them out with method-structured functions. Using this method, the project could successfully sever the connection from the user interface and focus on the backend code alone for testing. We refer the readers to [13] for details of how we used this novel technique for backend black-box unit testing.

Similarly, when applied to the *CyberWater* project, the testing is still external. In the prior state of the project, code was written and no or few test cases were developed – the principle of Test-Driven Development was entirely ignored. Ideally, test cases should be developed *in tandem* with the code if not *before* it. With Test-Driven Development, projects of a similar nature or similar scale are less likely to allow bugs into the final version, and are likely to have a smoother development process, with the project team able to focus more closely on the domain-specific aspects of the project, rather than wrestling with bugs due to the lack of a good testing workflow.

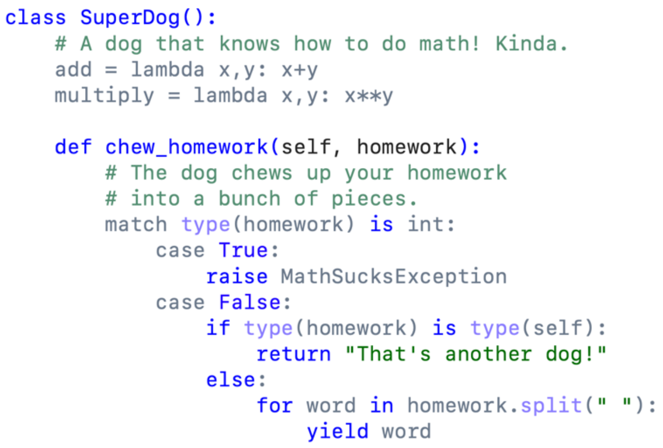
C. Defining a CI/CD Pipeline

Continuous Integration / Continuous Development is an involved process with many steps and is an industry standard when working on a software development project. In our experiences with the *CyberWater* project, we made many suggestions to aid in the development process, making the final product higher quality and the development process easier. Notably, we focused rather closely on testing. We developed a system of testing modules based on their input ports and output ports, since this is an integral functionality of *VisTrails* that the developers work with for their *CyberWater* extension. We applied this, most recently, to a workflow designed to pull data from the USGS website. We wrote tests based on specific inputs and the expected outputs they were to generate, whether that was a direct output or a changed state as a result of a function call. However, our progress with testing was quickly slowed by the lack of detailed specifications.

D. The Importance of Good Specifications

Specifications are an integral part of designing and writing code to be tested by someone else, especially in a black-box scenario. If a developer in test cannot write tests for whatever reason, whether it be limitations because they don't know the expected type of an output, they don't know what the intended output is, or they aren't aware of all the valid inputs, then they often have to resort to reading the code and guessing what the intended behavior is. This can become even worse if the developers are not utilizing clean code standards. The testers will often have to ask multiple questions, using up a lot of the developers' time. The problem could be solved before it is even brought about with the existence of good specifications. We asserted that specifications should be different from end-user documentation. Documentation should be high-level, describing the overall functionality of a module in a domain-specific way. Specifications, on the other hand, should be useful for the tester - often, testers don't know or don't care about the high-level functionality of a module, but rather what outputs it should produce based on specific inputs. They need an expected outcome based on regular inputs, so they don't need to read the code just to get started. Having a test fail just because the tester wasn't aware of what format the output would be in wastes time for the developer having to explain it, and the tester having to try to understand it. If detailed specifications were given in the first place, the process would have moved forward much more efficiently.

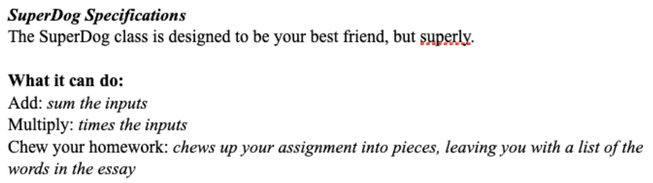
In our experience of applying automated unit testing to *CyberWater*, we found ourselves with lots of end-user documentation, but with very little helpful specifications in the way of what would be useful to us. There was a significant amount of time and effort involved in testing the modules we were given when the development team was unsure of what we needed for unit testing. To illustrate this concept, consider a hypothetical Python class that a tester should test. Let's call this class *SuperDog*. It has methods `add`, `multiply`, and `chew_homework`, shown in Figure 7, and we want these all to be tested to ensure they work properly. Bad specifications for the testers would be high-level and contain very little information about how this code is actually structured and what one would actually need to know should they call these methods directly.



```python
class SuperDog():
    # A dog that knows how to do math! Kinda.
    add = lambda x,y: x+y
    multiply = lambda x,y: x**y

    def chew_homework(self, homework):
        # The dog chews up your homework
        # into a bunch of pieces.
        match type(homework) is int:
            case True:
                raise MathSucksException
            case False:
                if type(homework) is type(self):
                    return "That's another dog!"
                else:
                    for word in homework.split(" "):
                        yield word
```

Figure 7: The code for the *SuperDog* class to be tested



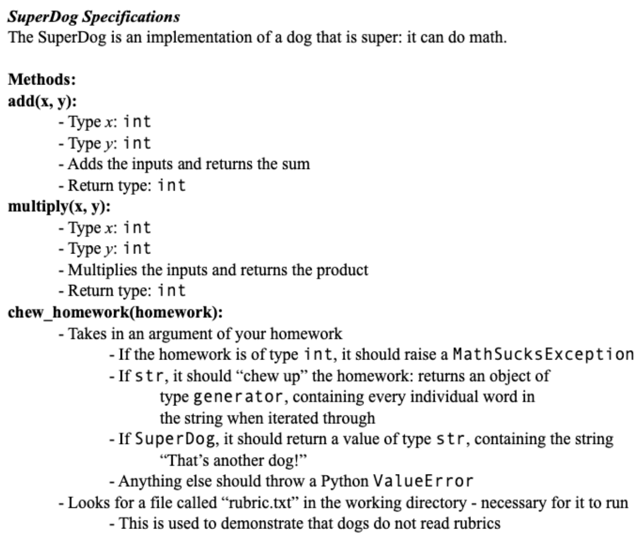
SuperDog Specifications
The SuperDog class is designed to be your best friend, but superly.

What it can do:
Add: *sum the inputs*
Multiply: *times the inputs*
Chew your homework: *chews up your assignment into pieces, leaving you with a list of the words in the essay*

Figure 8: Less than optimal specifications for *SuperDog*

As shown in Figure 8, we are given a high-level description, but we know very little about how to test it. The verbiage is inconsistent: are we *adding* or *summing* the inputs? Are we *multiplying* or *timesing*? Is it *homework*, an *assignment*, or an *essay*? We are left with many more questions: how do we test this? What types are the inputs? What are the return types? Do these methods have any side effects? What does it mean to "chew up homework"? What is the homework? Is the input only of one possible type? These are all questions a tester would likely have to ask the developers about, taking up more time and effort for everyone involved, assuming they don't give up and read the code. We don't even know if these are all necessarily methods – they are just listed as "what it can do."

An example of good tester specifications is shown in Figure 9. Everything, inputs and outputs, has its type listed so the testers are not stuck guessing when writing their tests. The verbiage is consistent: we know that the `add` and `multiply` methods take in two arguments and perform mathematical operations on them and return the result. We have a description as to what the "homework" argument is – it can be of various types, and we know the intended output for each type it should be. We also now know this method has a side effect: if a file `rubric.txt` is not in the working directory, it will fail. The testers now have specific exception types to test for and know helpful things about the function and output of the functions, notably `chew_homework`: the output will likely be `iterable`, and the `iterable` should have specific things in it. Thus, the code is much easier to test, even without ever having seen the code itself. It is unnecessary for the testers to know



SuperDog Specifications
The SuperDog is an implementation of a dog that is super: it can do math.

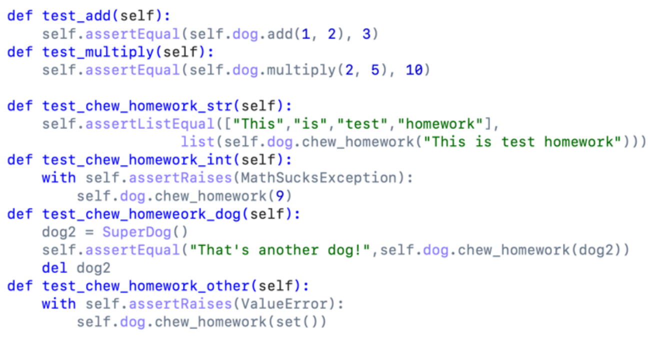
Methods:
add(x, y):
- Type *x*: `int`
- Type *y*: `int`
- Adds the inputs and returns the sum
- Return type: `int`

multiply(x, y):
- Type *x*: `int`
- Type *y*: `int`
- Multiplies the inputs and returns the product
- Return type: `int`

chew_homework(homework):
- Takes in an argument of your homework
  - If the homework is of type `int`, it should raise a `MathSucksException`
  - If `str`, it should "chew up" the homework: returns an object of type `generator`, containing every individual word in the string when iterated through
  - If `SuperDog`, it should return a value of type `str`, containing the string "That's another dog!"
  - Anything else should throw a Python `ValueError`
- Looks for a file called "rubric.txt" in the working directory - necessary for it to run
  - This is used to demonstrate that dogs do not read rubrics

Figure 9: Better specifications for *SuperDog*

how something is done, as long as they know whether it's being done right. Given those specifications, a tester could write some tests like the ones shown in Figure 10.



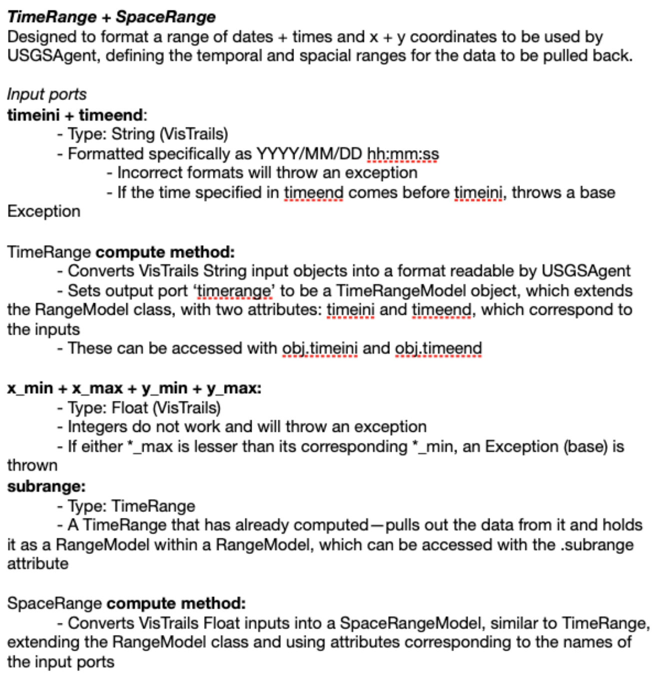
```python
def test_add(self):
    self.assertEqual(self.dog.add(1, 2), 3)
def test_multiply(self):
    self.assertEqual(self.dog.multiply(2, 5), 10)

def test_chew_homework_str(self):
    self.assertListEqual(["This","is","test","homework"],
                        list(self.dog.chew_homework("This is test homework")))
def test_chew_homework_int(self):
    with self.assertRaises(MathSucksException):
        self.dog.chew_homework(9)
def test_chew_homeweork_dog(self):
    dog2 = SuperDog()
    self.assertEqual("That's another dog!",self.dog.chew_homework(dog2))
    del dog2
def test_chew_homework_other(self):
    with self.assertRaises(ValueError):
        self.dog.chew_homework(set())
```

Figure 10: *SuperDog* tests

From these tests, we learn about some errors in the *SuperDog* code: the `add` and `multiply` methods are missing the `self` parameter, the `multiply` method is actually using exponentiation, and the `chew_homework` method has no error handling for an input of an unexpected type. With these specifications, if a given tester is proficient enough in Python to know what a *generator* is, then they will easily be able to test the code we have written.

This specification philosophy was then applied to two of the modules we were given to test for the *CyberWater* project: *TimeRange* and *SpaceRange*. These are related modules to be used in larger workflows, so their specifications are written in tandem. Figure 11 shows the new specifications we wrote for *TimeRange* and *SpaceRange,* for the developers' use as good examples for documenting future modules, while working with them closely on testing these modules.



TimeRange + SpaceRange
Designed to format a range of dates + times and x + y coordinates to be used by USGSAgent, defining the temporal and spacial ranges for the data to be pulled back.

Input ports
timeini + timeend:
- Type: String (VisTrails)
- Formatted specifically as YYYY/MM/DD hh:mm:ss
    - Incorrect formats will throw an exception
    - If the time specified in timeend comes before timeini, throws a base Exception

TimeRange **compute method:**
- Converts VisTrails String input objects into a format readable by USGSAgent
- Sets output port 'timerange' to be a TimeRangeModel object, which extends the RangeModel class, with two attributes: timeini and timeend, which correspond to the inputs
- These can be accessed with obj.timeini and obj.timeend

x_min + x_max + y_min + y_max:
- Type: Float (VisTrails)
- Integers do not work and will throw an exception
- If either *_max is lesser than its corresponding *_min, an Exception (base) is thrown

subrange:
- Type: TimeRange
- A TimeRange that has already computed—pulls out the data from it and holds it as a RangeModel within a RangeModel, which can be accessed with the .subrange attribute

SpaceRange **compute method:**
- Converts VisTrails Float inputs into a SpaceRangeModel, similar to TimeRange, extending the RangeModel class and using attributes corresponding to the names of the input ports

Figure 11: *TimeRange* and *SpaceRange*'s new specifications

For these specifications, there are clear descriptions as to the inputs and outputs and what methods we need to call to adequately test these modules. We know what inputs are valid, and what results invalid inputs should produce. We are also given some use cases, which allows us to determine which tests could be necessary depending on how specific attributes of the modules will be used. We also do not have any information that does not matter to us – we do not need to know *how* the values are converted into the correct format or what they are typically used for in a larger, higher-level sense.

In essence, when writing specifications for a tester, it requires thinking much differently from writing documentation for an end-user. Things written for the end-user should be high-level and focus on (possibly domain specific) functions, but things written for testers should be low-level and focus on what can be accessed by a tester who cannot see the code but must use the code to test in the backend. If a project is going to be tested by individuals or teams external to the project development team, it is essential that they know what they need to be looking for and how to write the tests without too much friction in creating said tests, allowing the project to run smoothly for everyone involved.

V. Conclusion

Industry practices in software development did not become industry practices for no reason. Utilizing these practices well in a project is vital to the ultimate success and efficiency of the project, and we expect that applying these practices to the *CyberWater* project will improve its development process significantly, making it both more efficient and causing the code produced to be of higher quality. Establishing processes and workflows for managing source code with version control, issue tracking, pull requests, branching strategies, clean code, and CI/CD are essential to working on a project with multiple people, and help to improve the final product while making the development process easier and less issue-prone, giving developers a blueprint to follow and improving quality of work for everyone involved, both for the developers and the final product. Our experiences reported here can be tailored to typical research projects in an academic setting, in which domain scientists need to write code while assuring their developed software is of high quality.

Acknowledgment

This work was generously funded by the National Science Foundation (NSF) under Grant 1835602. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] CyberWater, https://www.cuahsi.org/cyberwater.

[2] Jira Software, https://www.atlassian.com/software/jira.

[3] Bitbucket, https://bitbucket.org/product/.

[4] GitHub, https://github.com.

[5] PEP 8 – Style Guide for Python Code, https://peps.python.org/pep-0008/.

[6] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 1st ed., Pearson, 2008.

[7] Jenkins, https://www.jenkins.io.

[8] The One DevOps Platform | GitLab, https://about.gitlab.com.

[9] Bamboo Continuous Integration and Deployment Build Server, https://www.atlassian.com/software/bamboo.

[10] VisTrails, https://www.vistrails.org//index.php/Main_Page.

[11] Ptest, https://pypi.org/project/ptest/.

[12] Unittest – Unit Testing Framework, https://docs.python.org/3/library/unittest.html.

[13] L. T. Connelly, M. L. Hammel, B. T. Eger, and L. Lin, "Automated Unit Testing of Hydrologic Modeling Software with CI/CD and Jenkins," Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering (SEKE 22), 2022, pp. 225-230.