Automated Unit Testing of Hydrologic Modeling Software with CI/CD and Jenkins

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

Abstract

Composed of developers with diverse backgrounds in multiple disciplines, the NSF CyberWater *project team needed to research and implement effective software testing methods to improve the team's workflow efficiency and software quality. In this paper we present a practical and effective strategy for automated black-box testing of* CyberWater *modules using a Continuous Integration / Continuous Deployment (CI/CD) pipeline and the* Jenkins *automation server, Python* unittest *and* ptest*, and a novel technique we call* object-method replacement*, which isolates the backend from the front-end logic. Our experience can be adapted and extended to other research projects to mitigate the risk of programming errors and mistakes incurred through continuous development on a code repository.*

1 Introduction and Related Work

An automated software testing workflow plays a crucial role in preventing issues from creeping into the software, but the implementation of these workflows varies from project to project due to the complicated nature of software and testing. The NSF *CyberWater* project team faced some unique challenges with automated testing. In particular, their software was derived from legacy systems that integrated with *VisTrails* [7], a third-party software used to build scientific workflows as well as to support data analysis and visualization needed for hydrologic modeling and simulation. The design of *CyberWater* modules (as extended *VisTrails* modules) tightly couples the frontend and backend logic. We came up with a new technique called *object-method replacement* that allows us to address the testing problem created by the tightly coupled frontend and backend code without the need for proprietary tools for Windows-based GUI testing. Using this technique, we developed black-box unit tests using both Python *unittest* [6] and *ptest* [5] frameworks for two example modules: the `PythonCalc` module that comes with the *VisTrails* package and *CyberWater*'s `MainGenerator` module. Additionally, we researched and developed a CI/CD pipeline with a minimal set of tools supported by *Jenkins* [4] so that software developers from the domain would be able to use the pipeline on daily basis after a few training sessions efficiently and effectively.

Light-weight, agile methods and processes have drastically impacted the practice of software testing and quality control, putting testing first and in parallel with development to decrease development cost while increasing product quality [8]. State-of-the-art testing practices such as unit testing, continuous integration (CI), test-driven development, a test pyramid, test coverage analysis, etc. are considered a mandatory and indispensable part of modern software development [11]. A crucial step to leverage the benefits brought by the best industry practices, shared with successful practitioners, relies on a CI implementation [12]. Although the ideas are appealing to embrace, the choices for implementing unit testing, CI and test automation are usually heavily influenced by the particular software task and development environment, and unavoidably intimidating to software developers whose main expertise is in the domain sciences [9, 10]. The risk of software faults, programming errors and mistakes can be mitigated by continuous integration, regression testing and test automation, implemented by a testing workflow proposed here.

2 The NSF *CyberWater* Project and Challenges of Automated Unit/Module Testing

Funded by NSF, the *CyberWater* project aspires to build a new cyber infrastructure with an open data, open modeling framework and software to reduce the user time and effort required for hydrologic modeling studies, allowing related discoveries to be made sooner [2].

One of the key challenges of testing *CyberWater* modules is that the frontend and the backend are tightly coupled. Because the backend code depends on the frontend code for

DOI reference number: 10.18293/SEKE2022-074

input, the backend cannot be tested apart from the frontend without modifying the code for the backend.

In the *VisTrails* GUI (see Figure 1), each module is displayed with a set of input and output ports. Users can specify values for certain input ports for modules, while other input ports must receive data from the output of another module. Users can specify the flow of data from one module to the next by dragging a connector from the first module's output port to the next module's input port. A collection of connected modules that produces an output is called a *workflow*. When executed, the *workflow* usually generates a graph or model that visualizes the input datasets.

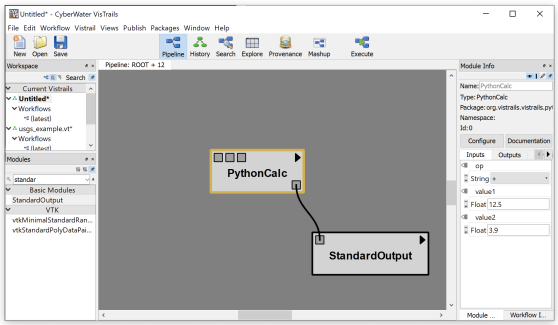

Figure 1. The *VisTrails* GUI

To retrieve the input data needed for the module's computation, the backend code of each module calls a method called `get_input`, which retrieves the input data from the module's input ports from the frontend. This dependency makes it so the backend code for *VisTrails* modules cannot be run or tested without input from the GUI without modifying the code.

3 Our Solution to Automated Unit Testing with CI/CD

3.1 A Novel Technique for Model-View Separation

We developed a technique called *object-method replacement* to achieve model-view separation. To isolate and test the backend logic we exchange a method from a specific instance of a class with a newly defined function. We define the replacement function to be nearly identical to the instance's original method, but we remove any dependencies to the frontend logic. With the backend logic isolated, we unit test the *VisTrails* module without having to modify the source code. By replacing methods in an instance of a class rather than modifying the class itself, changes to methods only affect the instance that is being altered, not the class in its entirety. Therefore, other instances of the class will not be affected by the changes.

Object-method replacement leverages the `__dict__` attribute of Python objects. This attribute is a dictionary that maps the names of each of the object's local attributes (as a string) to its corresponding value. The `__dict__` attribute is mutable, so developers can alter the values of the attributes stored in the `__dict__`, as shown below:

```
<instance>.__dict__['<attribute name>'] = <new value>
```

Because *VisTrails* modules correspond to Python classes, we can instantiate a class corresponding to a *VisTrails* module and leverage the instance's `__dict__` attribute to replace the instance's methods. This technique allows us to replace methods such as `get_input` that introduce dependencies to the GUI. We can replace the `get_input` method with a new version that simulates the GUI input, as shown below:

```
<instance>.__dict__['get_input'] = <replacement function>
```

Since we only modify the `__dict__` of a single instance of a class, this modification only changes the method for the specific instance of the class we are modifying.

Object-method replacement can be generalized with the function shown below. We call this function `mutate_method`.

```
import types

def mutate_method(obj, method_name, new_method):
    obj.__dict__[method_name] = types.MethodType(new_method, obj)
```

The function has three parameters: the target object, the name (string) of the method in that object we want to replace, and the new function. This replacement function can be structured like a normal function but with self parameters and calls to self inside the function if needed. The function we used to replace the `get_input` method is shown below.

```
from vistrails.core.modules.basic_modules import ModuleError

def get_input(self, port_name, allow_default=True):
  if port_name in self.inputPorts:
    if allow_default:
      defaultValue = self.inputPorts[port_name][0].get_output("value")
      if defaultValue is not None:
        return defaultValue
    raise ModuleError(self, "Missing value from port %s" % port_name)
```

Though this revised version of the `get_input` function is not defined in a class, the first parameter is `self`. If Python attempted to execute this revised `get_input` method without first binding it to a class, the program would throw an error. However, after using the `mutate_method` to inject the replacement function into an instance of the class we are testing, the replacement method functions the same as any other method of the class.

Object-method replacement allows developers to test the functionality of methods with dependencies to the GUI

without using specialized software to automate GUI interactions. Simply write a replacement version of the method that performs the same functionality without interacting with the GUI, instantiate a new instance of the class one is testing, and inject the new method in place of the old one.

3.2 Python Unit Testing Frameworks

There are two frameworks we have been using in tandem throughout this project: *ptest* by Karl Gong [5], and Python's built-in *unittest* library [6]. *Ptest* provides multiple advantages: testing using decorators allows for better in-code documentation with tags for grouping and a description given to every test, as well as the assertion of exceptions thrown within the decorator itself using the `expected_exceptions` parameter. These decorators include `BeforeMethod` and `AfterMethod`, which define code to be run before and after every individual test is run. They also include `BeforeClass` and `AfterClass`, which define methods to run before all of the tests run and to run after every test has finished, respectively. The `TestClass` decorator defines a class similar to creating a superclass of `unittest.TestCase` in Python's builtin, but has a parameter that allows for running tests in parallel with multiple threads. The most notable feature, however, is the test report that is generated. *Ptest*, upon finishing its tests, generates a graphic test report using HTML, CSS, and JavaScript as a visual representation of the tests that ran. It includes information about which tests failed, as well as reports of what was happening using the library's `preporter` module. It displays stack traces and descriptions of the tests that failed that were specified by the tester. This is shown in Figure 2. Python's `unittest`, alternatively, also offers distinct advantages. One problem we found with *Ptest* is that *GitHub* [3] *Actions*, *Bitbucket* [1] *Flows*, and *Jenkins* [4] *Pipelines* would all still recognize a build as passing even when *ptest*-based tests would fail. Using Python's `unittest`, however, these systems would recognize builds to be failing if the tests failed. It also provides setup and teardown methods, akin to *ptest*, and terminal output. *Ptest* also has a terminal output, shown in Figure 3.

3.3 *Jenkins* for CI/CD

One challenge that our team faced was developing a CI/CD pipeline to automatically test and merge changes made to *CyberWater* modules. We found that an efficient solution was to setup a Jenkins [4] server and link it to our *Bitbucket* repository. With *Jenkins*, developers can define jobs, which automatically run a set of tests to determine whether to automatically merge the changes from the *dev* branch into the *master* branch.

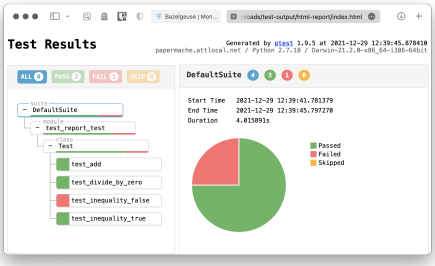

Figure 2. *Ptest*'s generated test report

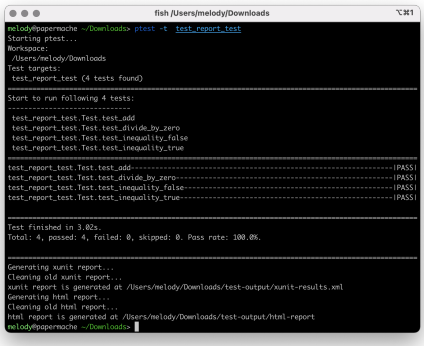

Figure 3. *Ptest*'s terminal output

After installing *Jenkins* on our Linux server, our team defined a **Freestyle project** from the dashboard. We then linked the **Freestyle project** to our *Bitbucket* repository in the **Source Code Management** section. This section allows developers to insert the URL and credentials for *Jenkins* to access the *Bitbucket* repository.

We then configured *Jenkins* to build the *dev* branch. Under **Build Triggers**, we selected **Build when a change is pushed to *Bitbucket***. This trigger configures *Jenkins* to automatically run the *job* every time someone pushes changes to the repository. Finally, we added a build step to execute a Unix shell command to run our *ptest* and *unittest* scripts.

Developers also have the option to add **Post-build Actions** which offer helpful features such as sending an email with the build results or publishing an HTML report of the build.

4 *CyberWater* Unit Testing Case Studies

4.1 `PythonCalc` Module Testing

The `PythonCalc` module is a simple module designed to incite familiarity with how to create *VisTrails* modules and the functionality of the *VisTrails* interface. It functions as a simple calculator with four possible operations:

addition, subtraction, multiplication, and division. Its input ports consist of two `Floats` or `Integers` and a `String`. All of these are wrapped as *VisTrails* modules, not Python literals. The `Float` / `Integer` input ports consist of two numbers to be operated on. The `String` input port is where the module expects an operator – this can be either '+', '-', '*', or '/'. Anything else entered will raise an exception. The module then reads the values of the number input ports, reads the `String` input port, and performs the operation on the numbers. For example, if the user entered 3, 5, and '+', the module would add 3 and 5, and return 8. The output port of the module is the result of the operation. It is typically sent to the `StandardOutput` module to print it to the persistent console that accompanies *VisTrails*. Shown in Figure 4 is an example of a possible workflow with the module, with the output in the console.

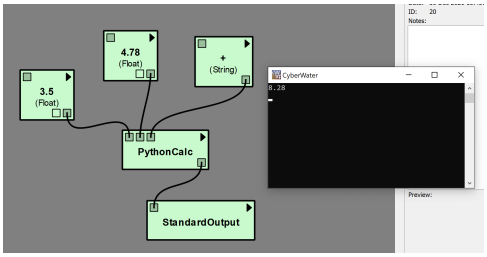

Figure 4. Usage of the `PythonCalc` module within *VisTrails*

Beginning the tests is where object-method replacement must be used. After injecting our own `get_input` method into the instance of the class we created as shown in Figure 5, we first test to ensure that all of the operators work properly, using two arbitrary numbers and an operator, asserting that the result is correct (four tests). We then test using an invalid operator to assert that an exception is raised, and we attempt to divide by zero to assert that a `ZeroDivisionException` is raised (two tests; see Figure 6). All the unit tests (using either Python *unittest* or *ptest*) pass when configured to run within *Jenkins*.

4.2 `MainGenerator` Module Testing

The `MainGenerator` module is designed as a component of the *CyberWater* framework to set up the directory for running a user's model where the simulation will take place and data will be stored. Shown in Figure 7, its input ports consist of `01_Path` and `02_GPF`, of types `Directory` and `File`, both wrapped as *VisTrails* modules. It then has 15 more input ports, named `Dataset_01`, `Dataset_02`, etc. `01_Path` is the aforementioned directory of the simulation, and `02_GPF` is a "global parameters file," designed to hold specific information needed in the simulation. The

```
@BeforeMethod(enabled=True, description="init pythoncalc & mutate_method")
def before(self):
    preporter.info("initializing pythoncalc")
    self.python_calc = PythonCalc()

    preporter.info("mutate get_input method")
    mutate_method(self.python_calc, "get_input", get_input)

    preporter.info("init constants and assign input ports")
    self.f1, self.f2, self.s1 = Float(), Float(), String()
    self.f1.setValue(3.0)
    self.f2.setValue(2.0)

def assign_parameters(self):
    self.python_calc.set_input_port("value1", self.f1)
    self.python_calc.set_input_port("value2", self.f2)
    self.python_calc.set_input_port("op", self.s1)
```

Figure 5. Setup methods for the `PythonCalc` tests using *ptest*, including object-method replacement of `get_input`

```
@Test(tags=["arithmetic", "error"],expected_exceptions=Exception)
def test_invalid_operator(self):
    self.s1.setValue('m')
    self.assign_parameters()
    self.python_calc.compute()

@Test(tags=["arithmetic", "error"],expected_exceptions=ZeroDivisionError)
def test_division_by_zero(self):
    self.s1.setValue('/')
    self.f2.setValue(0.0)
    self.assign_parameters()
    self.python_calc.compute()
```

Figure 6. Testing for an invalid operator and the divide-by-zero fault using *ptest*

`Dataset_NNs` are of type (*VisTrails*) `String`, indicating all the received datasets (with a maximum of 15) to be imported for running the user's model.

Based on the module specification provided by the development team, it begins by checking whether the directory the user has entered in `01_Path` exists. If it does, it deletes the folder and re-creates it (to ensure there are no files inside it already). If it doesn't, it creates the folder. It then copies the file specified in `02_GPF` into the new directory that was created. Then, it outputs the directory from `01_Path` in its first output port, and all the `String` objects of the `Dataset_NN` input ports are compiled into a Python `Dict<str, str>`, with the keys being the input ports, `Dataset_NNs` and the values being the `Strings` that were given to those input ports, but converted back into Python `strs`.

It is necessary to use object-method replacement to replace the `get_input` method of this module, just as with all other modules with input ports (see Figure 8). Figure 9 shows a diagram of our designed unit tests based on the `MainGenerator` module specification.

We begin by running a simple test of the `compute` method with no inputs to the module, asserting that it raises an exception. Then we proceed with three tests for the first input port `01_Path`, testing the module's behavior on a di-

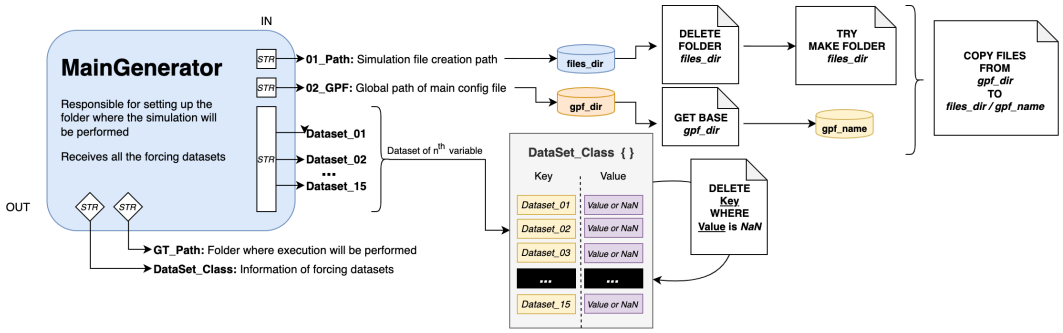

Figure 7. A diagram of the inputs and outputs of the `MainGenerator` Module

```
@BeforeMethod(enabled=True)
def __init__(self):
    preporter.info("initializing maingenerator module")
    self.maingen = MainGenerator.MainGenerator()
    mutate_method(self.maingen, "get_input", get_input)
```

Figure 8. Object-method replacement of the `get_input` method for `MainGenerator` testing using *ptest*

rectory that already exists, a directory that does not exist, and a directory that exists but for which the module doesn't have permission to access. Similarly we design three tests for the second input port `02_GPF`, assuming the file exists, or it doesn't already exist, or it exists but with no read permission. In each case, we make an assertion or assert a raised exception.

The module has two output ports `GT_Path` and `DataSet_Class` as shown in Figure 7. We test that `GT_Path` outputs the working directory we gave the module (with one test). For `DataSet_Class` we design four tests. We first test that specifically the first data set is present in the output when run, due to the special way it is handled (its input port is always shown on the GUI). We then test, giving the module two random datasets (say, chosen DataSet_01 and DataSet_04) that every dataset that was not given any input does not appear in the output. Next we give the module any dataset that isn't the first one, asserting that it exists in the output and the first one doesn't. Finally, we test that, when given no datasets, the output simply consists of an empty Python dictionary.

In all, there are twelve tests we designed, and all pass using both *ptest* and Python *unittest*. The unit tests ran automatically using *Jenkins* jobs. Figure 10 shows the HTML test report of *ptest*. Figure 11 and Figure 12 show our unit tests that test the second input port `02_GPF` using *ptest* and *unittest*, respectively.

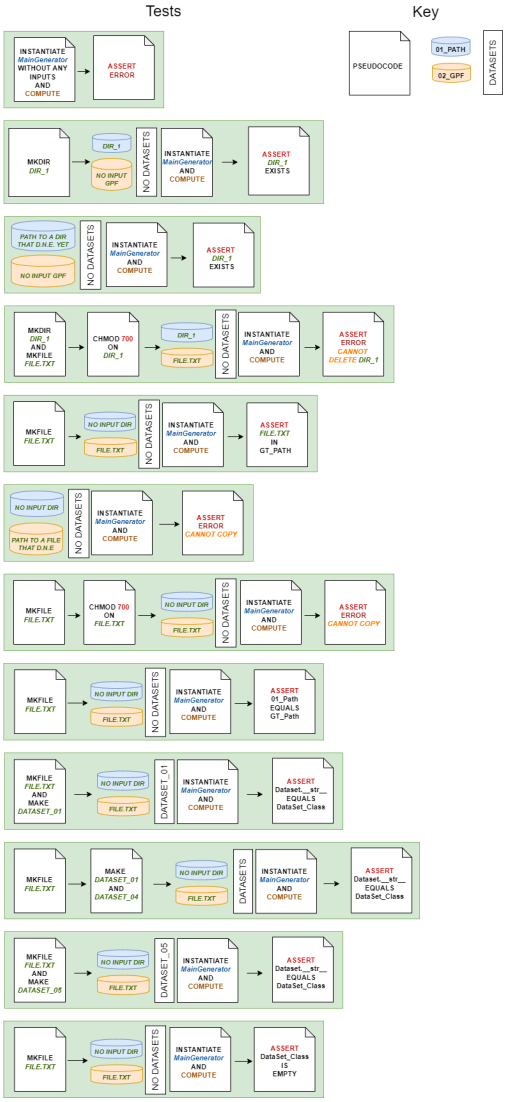

Figure 9. Design for the `MainGenerator` unit tests

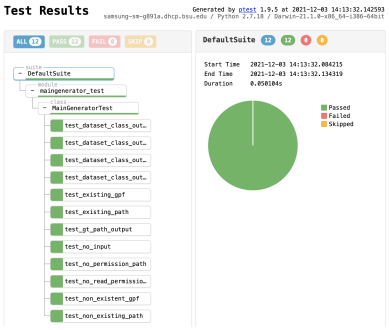

Figure 10. The *ptest* HTML output of the twelve `MainGenerator` tests

```
@Test(tags=['02_GPF'],
description='testing valid,existing gpf')
def test_existing_gpf(self):
    set_up(self.maingen, gpf=True)
    self.maingen.compute()
    assert_true(exists("test_directory/test_gpf"))

    rmtree("test_directory")
    remove("test_gpf")

@Test(tags=['02_GPF', 'error'],
description='testing non-existent file',
 expected_exceptions=IOError)
def test_non_existent_gpf(self):
    set_up(self.maingen)
    gpf = File() ; gpf.setValue("non_existent_gpf")
    self.maingen.set_input_port("02_GPF", gpf)
    self.maingen.compute()
    rmtree("test_directory")

@Test(tags=["02_GPF", "error"],
description="testing gpf with no read permission",
 expected_exceptions=Exception)
def test_no_read_permission_gpf(self):
    set_up(self.maingen)
    gpf = File()
    open("no_perms", "w").close()
    set_mode("no_perms", 700)
    gpf.setValue("no_perms")
    self.maingen.set_input_port("02_GPF", gpf)
    self.maingen.compute()
    rmdir("test_directory") ; remove("no_perms")
```

Figure 11. *Ptest* tests for `02_GPF` for `MainGenerator`

5 Conclusion and Future Work

This paper reflects on and reports our experience in applying black-box unit testing and test automation, in a CI/CD pipeline supported by *Jenkins*, to the *CyberWater* software developed for hydrologic modeling studies. We propose a novel technique called object-method replacement that provides a solution to the problem of testing software with tightly-coupled frontend and backend without the need to revise the underlying code. This is not a replacement for model-view separation, but rather a solution for software testers who want to automate testing of a legacy program with model-view separation violations. It could also be applied in scenarios where a method of an object needs to have its functionality temporarily altered. We demonstrate a testing workflow using the *Jenkins* automation server for CI/CD, and Python *unittest* and *ptest* frameworks for test automation. Future work along the line includes automated testing of more complicated *CyberWater* modules and integrated workflows. The preliminary results are promising.

```
def test_existing_gpf(self):
    set_up(self.maingen, gpf=True)
    self.maingen.compute()
    self.assertTrue( exists("test_directory/test_gpf") )
    rmtree(td);remove(tg)

def test_non_existent_gpf(self):
    set_up(self.maingen)
    gpf = File() ; gpf.setValue("non_existent_gpf")
    self.maingen.set_input_port("02_GPF",gpf)
    with self.assertRaises(IOError):
        self.maingen.compute()
    rmtree(td)

def test_no_read_permission_gpf(self):
    set_up(self.maingen, gpf=True)
    set_mode(tg, 700)
    with self.assertRaises(IOError):
        self.maingen.compute()
    remove(tg)
```

Figure 12. Python *unittest* tests for `02_GPF` for `MainGenerator`

Acknowledgments

This work was generously funded by the National Science Foundation (NSF) under Grant 1835602. It was also supported in part by an Undergraduate Honors Fellowship, funded by the Honors College, Ball State University.

References

[1] ATLASSIAN Bitbucket. `https://bitbucket.org/product/`.
[2] CyberWater. `https://www.cuahsi.org/projects/cyberwater/`.
[3] GitHub. `https://github.com`.
[4] Jenkins - Build great things at any scale. `https://www.jenkins.io`.
[5] Ptest 2.0.3 - Light test framework for Python. `https://pypi.org/project/ptest/`.
[6] Unittest - Unit testing framework. `https://docs.python.org/3/library/unittest.html`.
[7] VisTrails. `https://vistrails.org`.
[8] P. Ammann and J. Offutt. *Introduction to Software Testing, 2nd Edition*. Cambridge University Press, 2016.
[9] L. D. Couto, P. W. V. Tran-Jørgensen, R. Nilsson, and P. G. Larsen. Enabling continuous integration in a formal methods setting. *International Journal on Software Tools for Technology Transfer*, 22(6):667–683, 2020.
[10] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437, 2016.
[11] D. Spinellis. State-of-the-art software testing. *IEEE Software*, 34(5):4–6, 2017.
[12] S. Stolberg. Enabling agile testing through continuous integration. In *2009 Agile Conference*, pages 369–374, 2009.