EvaP

Jennifer Stamm, Stefan Neubert

Winter Term 2015/16

Contents

1 Introduction

1.1 EvaP – An Evaluation System

The online platform EvaP is used for evaluation of courses at the Hasso Plattner Institue (HPI). Its development started in 2011 when the student representatives decided to redevelop the former system EvaJ. Now, it is an Open Source project hosted on GitHub, even though the main developers are still part of the student representative team.

EvaP's time to shine is at the end of each semester. Each university course is evaluated with specific questionaires chosen by the lecturer. Students are allowed to give anonymous feedback to different aspects of the lecture, the lecturer itself and additional tutors through a grading system and comments. EvaP encourages evaluation by rewarding participation with points. These reward points can be redeemed for currency to use at HPI events. EvaP's latest feature is the distribution of course grades through the platform.

1.2 Motivation – Why should EvaP be Analyzed, Verified and Tested?

EvaP is an important source for feedback at HPI. The platform offers students a way to express their critique anonymously. It documents the feedback for lecturers. Thus, they do not need to collect and save the feedback themselves. Additionally, they can take their time to evaluate the student's feedback. Furthermore, the evalutation of all courses is saved centrally. This allows to gain an overview over the quality of HPI courses and compare feedback over time as well as with other courses. Therefore, EvaP is an important tool at the HPI.

Since EvaP is developed by students, the responsible persons and main developers change regularly as older students graduate and new ones enroll. The change of responsible persons shifts the view of which features, programming paradigms and quality assurance are most important. Consequently, the requirements change on a regular basis as well. A change of main developers comes with an inevitable loss of knowledge about the existing code. Besides, the Open Source aspect allows developers without inside knowledge to contribute code as well. Even though all code is reviewed and checked by the main developers, they may not grasp it as well as self-written code. Events like Hackdays or Hacking Hours are used to promote EvaP's development. While these practices ensure the advancement of EvaP, they may endanger its quality.

Because of EvaP's importance at the HPI, its quality should be ensured. Therefore, we will analyze, verify and test the software within the scope of this lecture.

2 Milestone 1

As suggested, the duration of milestone 1 is from the beginning of the project until the end of December. Milestone 1 includes the set up of the software project which led to

the discovery of the first bug. It includes the first steps taken to gather information, get comfortable with the project and planning of the project. Lastly, it includes the phase of testing the project regarding graph coverage.

2.1 Set Up

Instead of running natively on the developer's machine, EvaP is wrapped in a Vagrant execution environment[1]. This allows developers to develop features with their preferred tools on their favorite operating system out of Mac OS X, Windows, Debian and Centos, without having to go through the complicated process of collecting dependencies for their specific platform. The complete project set up normally consists of installing git, a virtual machine provider for Vagrant and Vagrant itself, cloning the repository and running the shell command `vagrant up`.

As all core developers of EvaP use unix-based platforms, a bug in an external sub module used by EvaP remained unnoticed and undealt with: When we tried to set up the project on windows machines that use a line feed and a carriage return (`LF CR`) to end lines in text files — as opposed to unix systems that only use a single line feed (`LF`) — the set up failed. The bug was fixed quickly: firstly only for EvaP, subsequently in the external module the line ending policy had to enforce `LF` only.

Now, the set up therefore works on all major operating systems, allowing an easy start for all developers who want to contribute to EvaP, and more insights into the system for us.

2.2 First Steps

Incidentally, this is the first semester for the student representatives to host *EvaP Hacking Hours* biweekly. This is an event to give students a space to develop and work on EvaP. We will use it as a possibility to stay in contact with the main developers. As testers of EvaP it is a valuable resource to be able to talk directly with developers. This way we can gather current information easily. We are able to verify the content of old artifacts with them as well as our future findings.

2.2.1 Application Survey

The current main developer of new features is Johannes Wolf. He is supported by Johannes Linke who is mainly responsible for a good coding style, including code review and refactoring. We interview them about information regarding the first steps of our project.

Development Paradigm and Development Languages There are no development paradigms explicitly determined. But as stated there is a main developer responsible for code review. As it is, all newly developed code is reviewed before it is accepted.

[1] `https://www.vagrantup.com/`

Everyone is able to review code and discuss it with the author and other reviewers. This practice shall ensure readable code and distribute knowledge about code changes. Additionally, it is implicitely assumed that paradigms of the used development languages are followed. The development languages are:

- Python 3 through the Django framework
- HTML
- Javascript
- CSS

As an example for paradigms given by the languages we checked the document known as *PEP 0008*[2]. This is the style guide for Python code written by Guido van Rossum, the author of Python, and followed by most open source projects in Python.

Requirements / Specification / Documentation / Artefacts Requirements were elaborated at the start of EvaP's predecessor EvaJ several years ago. No original artifacts were stored even though most requirements still hold. Examples are:

- Evaluation should be anonymous
- Written feedback is only readable by a small, strongly specified selection of people
- Written feedback is reviewed before it is available to the lecturer

Symptomatic for an open source project managed by students there is no formal specification of EvaP. Though there are different information gathered in the project wiki hosted on Github[3]. The responsibility to specify new features is on the main developer, Johannes Wolf. He collaborates directly with the users of the feature.

Current testing status / Bug repositories The project is hosted open source on GitHub. Different tools allow to include badges on the overview page to display the status of the latest build (Figure 1). The following tools are already used:

- *Travis CI*: A continuous integration service to build and test projects hosted on GitHub
- *Gemnasium*: An automated service for monitoring project dependencies for possible updates
- *Landscape*: A service checking the code for errors, code smells and deviations from stylistic conventions
- *Coveralls*: A service relying on Travis CI that tracks the code coverage

[2] `https://www.python.org/dev/peps/pep-0008/`
[3] `https://github.com/fsr-itse/EvaP/wiki`

- Another feature of Github is used to track bugs: the label *[T] Bug* for *issues*.

Since this is already an extensive list, we will not to spend much time on researching further tools.

Figure 1: Badges of testing tools on the GitHub overview page (01.12.2015)

Personal involvement Our first contact with EvaP was as users. As HPI bachelor students we evaluated courses of the first semester. As a tutor for a course Stefan used EvaP to read feedback. As a member of the student representatives Jennifer reviewed comments before publishing the evaluation. During the *Evap Hackday 2014* and *Evap Hackday 2015* Jennifer joined the team developers. She familiarized herself with the development practice to fork, code and create pull requests for review and solved several small issues.

2.2.2 Initial Test Plan

We developed an initial plan based on our findings in the application survey. As suggested we followed the questions discussed during the lecture. The test plan includes evaluating and enhancing the existing artifacts, cross-checking the results found by the tools used and eventually improving the testing status.

Five V&V Questions We started with evaluating EvaP regarding the five basic verification and validation questions:

1. When do verification and validation start? When are they complete?

   Verification and validation started along the start of the project and will be an important part during the duration of the project.

2. What particular techniques should be applied during development?

   As EvaP is a software too big to be wholly tested by us within the scope of the seminar project, we will try to apply as many techniques as possible on a small subpart of the project. We will evaluate the techniques and make recommendations for further testing to the main developers.

3. How can we assess the readiness of a product?

   Since EvaP is already in use it is certainly ready. Our testing will not influence the readiness.

4. How can we control the quality of successive releases?

   Additionally to continous integrationa and automated tests, a code review before merge is already implemented in the development process to control the quality of successive releases. We hope to enable developers to gain more knowledge about the system by enhancing the available artifacts and documentation.

5. How can the development process itself be improved?

   The main weakness of the development process is the sparse distribution of knowledge and the change of developers over time. Since only students are interested in developing EvaP we do not see a possibility to improve this part of the process.

Test Classifications and Approaches Considering the following test approaches we came to the conlusion described below:

1. Validation vs. Defect Testing

   Since there is no requirements document or even a list of features, it is hard to implement validation testing. As we will not draw up a corresponding document, we will not apply validation testing. However, if possible, we will try to detect defects in the software by finding inputs leading to incorrect behavior. Thus, we will apply defect testing.

2. Development, Release, User Testing

   Because of the applied continuous integration process, development testing is already in use. Release and acceptance testing is not possible; there are no specified releases. User testing is applied in a way as most users are familiar with the developing process and report encountered bugs directly.

3. Unit/Component, Integration, System Testing

   EvaP is a web application, so there are no hardware components directly involved. While there are software dependencies, this aspect is already covered by the tool Gemnasium. For a web application it would be important to work in the most popular web browsers (desktop and mobile). Since there are no complaints known about EvaP not working in a certain web browser and no plans exist about developments that would use web browser specific features, we will not focus on integration testing. Similar to validation testing, without a specified requirements we can not execute system testing. Instead we will focus on unit and component testing.

In conclusion, we will use tests to find unnoticed defects and we will apply different testing techniques on units or components of EvaP. These approaches align with our goals to help EvaP's developers and to learn about different testing techniques.

Available Artifacts The most thorough artifact, the GitHub wiki, offers two artifacts with potential regarding coverage-based testing:

- A finite state machine describing the states of courses in the evaluation process (Figure 2)[4]
- Description of a few use cases with UML use case diagrams[5]

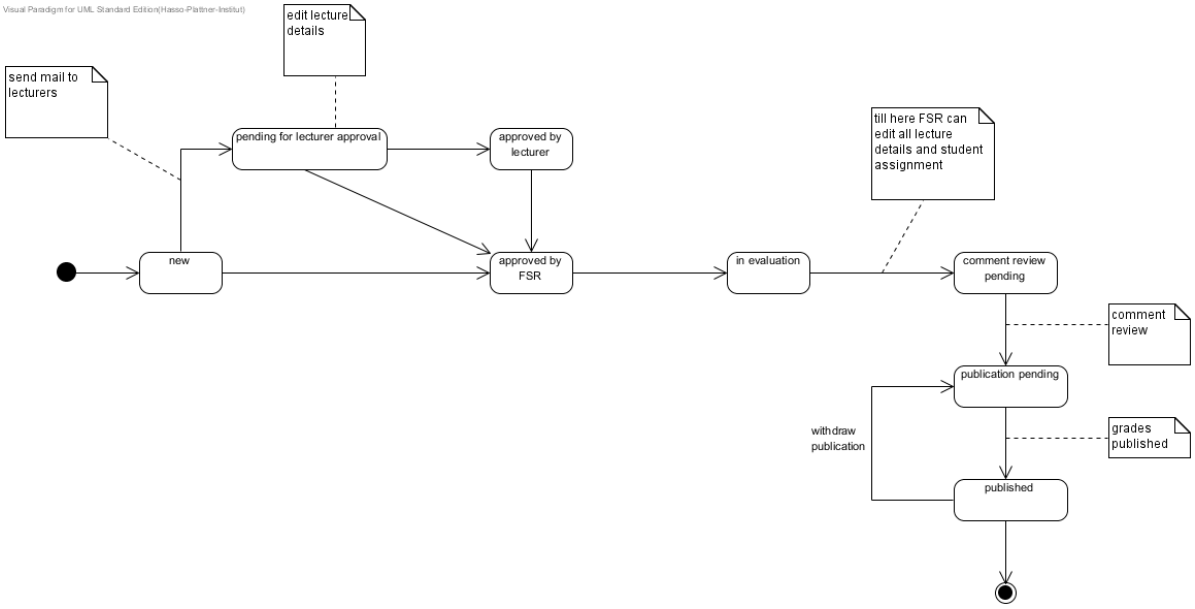

Figure 2: Original FSM: Possible states of a course

Initial Test Plan Based on our findings, the initial test plan is as follows:

- Create a control flow graph of a function, apply coverage criteria to define test sets and implement tests
- Check and if necessary update the FSM of evaluation states
- Use or create a UML use case diagram including its elaboration to develop an activity diagram
- Investigate coverage found by the tool *Coveralls*

[4] `https://github.com/fsr-itse/EvaP/wiki/Evaluation-States`
[5] `https://github.com/fsr-itse/EvaP/wiki/Use-Cases`

2.2.3 Test Automation

The project is already covered by several tools. Our research showed the used tools are popular in the Python community if the project is hosted on GitHub. Because of this and since the developers are comfortable with their choices, we will observe the functionality of the tools during the project. We will focus this section on describing our experience with the set up of running tests locally.

Django Tests The web framework Django used by EvaP comes with an integrated test runner embedded in Django's `manage.py` utility script. This test runner executes python test cases inside an isolated execution environment provided by Django.

To be able to run the specified tests, one therefor has to set up EvaP as described in subsection 2.1, access the server via a remote console via `ssh` and call `python3 manage.py test <evap module test> [<evap module test> [...]]`. The provided console output then informs about successful and failed tests. Exemplary usage of this utility is provided by the configuration of *Travis* located in the `.travis.yml` file in the project's root. Travis is a web service that automatically tests the code of each pull request submitted on GitHub.

By these means it is enforced, that all existing tests are run at least once before code changes are merged into the production source code.

PyCharm integration For local development we decided to code and test with PyCharm, a professional IDE for python, that also claims to support Django and Vagrant out of the box. Since the recently published release of PyCharm version 5, this support almost covers all configuration issues of remote development, only the auto-configuration of remote access via ssh still is buggy and only sometimes runs out of the box. Whilst previous versions did not configure the path mapping to the virtual execution environment correctly, the developer now only has to run the automatic project configuration of PyCharm for Django Vagrant projects to be able to execute tests. The test integration then provides an intuitive graphical overview of the status of all test cases, and can even be configured to rerun automatically on code changes. However, as the execution of the whole test set currently takes a few minutes, it is not feasible to constantly run the whole test set during development.

Code coverage measurement Code coverage of python projects can be measured with the coverage package[6]. This package records statement coverage and can be configured to measure branch coverage as well. According to our research, there currently does not seem to be a tool that implements further coverage criteria such as logic coverage. To run the tests with coverage, one has to call the utility script with `coverage run`. From the collected coverage data one can generate a HTML report or convert the data in a xml-based format that can e.g. be read by PyCharm.

[6] `https://pypi.python.org/pypi/coverage`

PyCharm is also able to run tests with coverage itself and afterwards enriches its editor by displaying the current line coverage along the source code. Due to an IDE bug, this integration is not as reliable as one would need for an efficient testing workflow, as most times the IDE shows a measured coverage of 0% for all lines, even though the tests were run successfully. To be able to anyway view the coverage results, one has to manually import the generated xml coverage reports. EvaP uses an integration of the web service *Coveralls* into the Travis build process that roughly provides the same view as PyCharm. Neither Coveralls, nor PyCharm display the collected branch coverage data at the moment.

In summary the support of automated testing through the selected tools is very good for line- and branch coverage. During the work on milestone 1 we have witnessed major improvements in the IDE PyCharm concerning test execution and coverage data analysis. It is to expect, that even more improvements will follow within near future.

2.3 Graph Coverage

According to our test plan we chose parts of EvaP to test based on graph coverage. We will test the function `send_publish_notifications` by creating a control flow graph and test cases based on path coverage criteria. We will investigate the documented finite state machine (FSM). We will create an activity diagram based on the documented use-case Finally, we will investigate if our added test cases changed the line coverage, since this is the given measurement by the used tool *Coveralls*.

2.3.1 Selected Control Flow Graph

We chose to test the function `send_publish_notifications` from `evaluation/tools.py`. As most of EvaP's functionality deals with data management, this is one of the few functions with a complex control flow graph. Its purpose is to send an email notification to participants and contributers of all course for which new evaluation results have been published. To achieve this, the function has to traverse all newly published resources, collect the involved users and merge the notifications per user in order to send each user at most one mail.

During our research we found two promising tools for automatic control flow graph creation. The first tool is open source and hosted on GitHub (`https://github.com/danielrandall/python-control-flow-graph`). Unfortunately, it is not documented and its execution lead to errors. After a few tries, we estimated that fixing the tool would take more time than creating the control flow graph by hand. The second tool found was even more promising as it was a report of a master graduate from the university of Texas titled: "Control flow graph visualisation and its application to coverage and fault localization in Python" by Jackson Lee Salling. His tool even visualized edge-pair and prime path coverage in control flow graphs. But unfortunately he did not respond to our request if we could use the tool for our project or if he had tips for published tools. Therefore, we created the control flow graph by hand.

We experienced some difficulties with Python's code style, for example with the assignment of default values to missing parameters:

```
1 def send_publish_notifications(grade_document_courses=None,
      evaluation_results_courses=None):
2     grade_document_courses = grade_document_courses or []
```

This assignment of either the `grade_document_courses` or an empty list is dependent on the evaluation of `grade_document_courses`. Here are several statements hidden in one line, disguised by a lazy evaluation of a boolean disjunction.

```
def send_publish_notifications(grade_document_courses=None,
    evaluation_results_courses=None):
    grade_document_courses = grade_document_courses or []
    evaluation_results_courses = evaluation_results_courses or []

    publish_notifications = defaultdict(lambda: CourseLists(set(), set()))

    for course in evaluation_results_courses:
        # for published courses all contributors and participants get a
            notification
        if course.can_publish_grades:
            for participant in course.participants.all():
                publish_notifications[participant].evaluation_results_courses.add
                    (course)
            for contribution in course.contributions.all():
                if contribution.contributor:
                    publish_notifications[contribution.contributor].
                        evaluation_results_courses.add(course)
        # if a course was not published notifications are only sent for
            contributors who can see comments
        elif len(course.textanswer_set) > 0:
            for textanswer in course.textanswer_set:
                if textanswer.contribution.contributor:
                    publish_notifications[textanswer.contribution.contributor].
                        evaluation_results_courses.add(course)
            publish_notifications[course.responsible_contributor].
                evaluation_results_courses.add(course)
    for course in grade_document_courses:
        # all participants who can download grades get a notification
        for participant in course.participants.all():
            if participant.can_download_grades:
                publish_notifications[participant].grade_document_courses.add(
                    course)

    for user, course_lists in publish_notifications.items():
        EmailTemplate.send_publish_notifications_to_user(
            user,
            grade_document_courses=list(course_lists.grade_document_courses),
            evaluation_results_courses=list(course_lists.
                evaluation_results_courses)
        )
```

Listing 1: The control flow graph is based on this function.

As the chosen function takes two lists of courses with published results as input values, one can almost achieve Node and Edge Coverage with only one test case, that includes one course for each if/else-branch. Only the assignment of empty lists as default arguments would require a second test case. This clearly shows, that neither Node nor Edge Coverage alone is an appropriate coverage criteria to decide whether the function has been fully tested, as both test cases are highly unrealistic. Additionally, the complexity of the first test case would be comparable to the complexity of the tested function itself, which makes it just as likely that the test case contains faults as that failing tests flag a fault in the tested code.

Edge Pair Coverage seems to be a much more useful criteria in this case, as it is able to describe whether loops have been executed or bypassed. For example, one would have to implement both a test case which enters the loop in node 7_{for} via the edge pair $(5, 7_{for}), (7_{for}, 9_{if})$ and to bypass it using the edges $(5, 7_{for}), (7_{for}, 21_{for})$. Edge Pair Coverage would also require a test set that executes the edge pair $(21_{for}, 23_{for}), (23_{for}, 21_{for})$ — a path that would only be executed if the function processes a course without participants. This however is an impossible input, and even though the test case could still simulate this kind of input, it does not make much sense to test a setting that can not occur, as a course with results always needs to have at least one participant.

Requesting the test set to achieve Complete Path Coverage is also not possible, as the function under test contains multiple loops. Therefore, Prime Path Coverage would be a possible criteria to test the function with each loop being bypassed, run once and run twice. However, the graph coverage tool provided by Ammann and Offutt[7] determines a total of 101 necessary test paths to cover all prime paths. Even if we would remove all infeasible test paths from that list, the number of test paths would still be over-the-top.

Hence, we decided to go for Specified Path Coverage to cover typical usages of the function and also some special cases that could lead to unexpected behavior. This lead to a test set of six test cases, that achieve both Node and Edge Coverage.

2.3.2 Finite State Machine of Course States in the Evaluation Process

2.3.3 Use Case, Elaboration and Activity Diagram of ...

2.3.4 Line Coverage with COVERALLS

3 Milestone 2

3.1 Logic Coverage

3.2 Input Coverage

3.3 Analysis

Tools were run on the function `send_publish_notifications(..)` and on the file `tools.py`.

[7] `https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage`

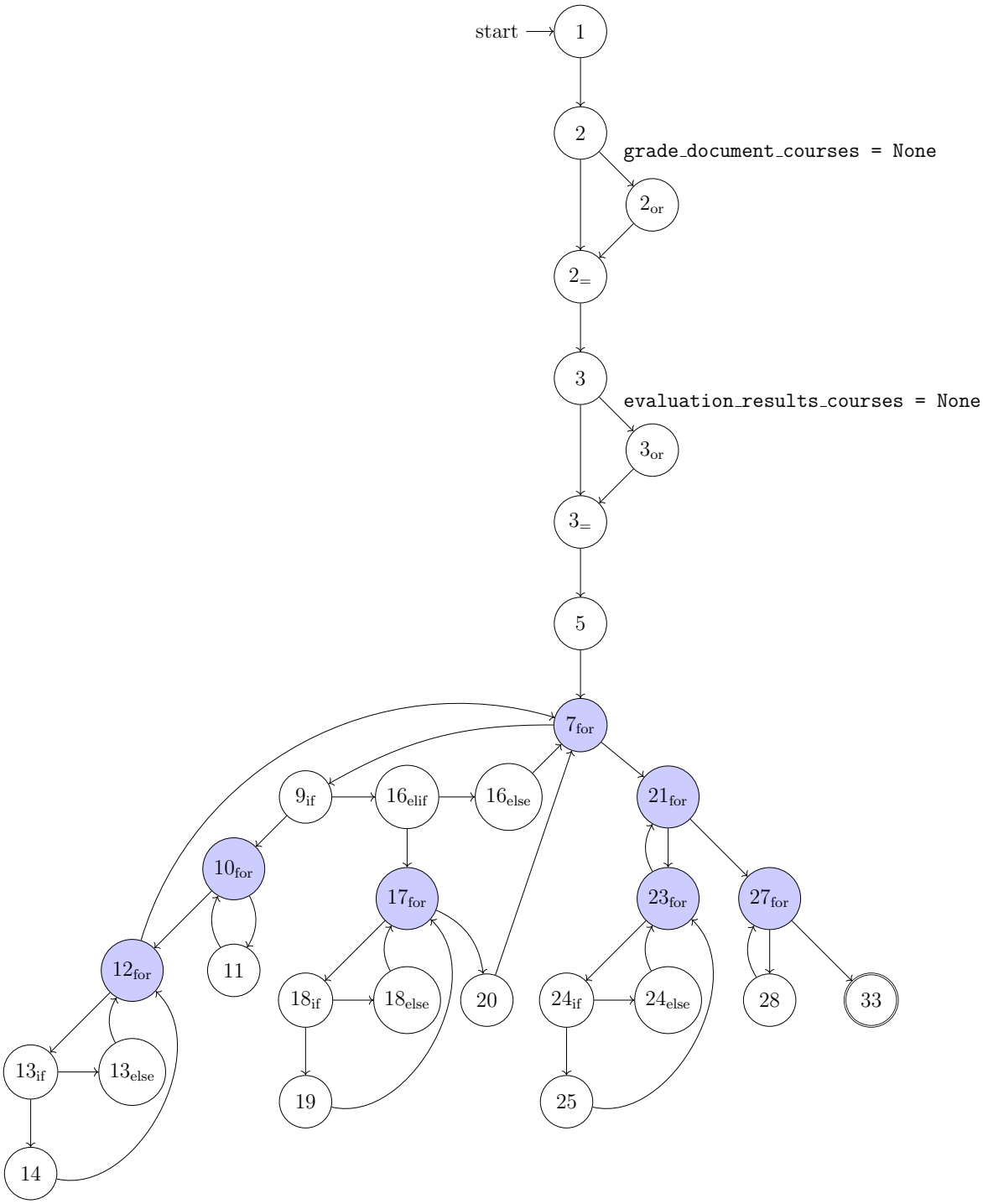

Figure 3: Control Flow Graph of the Function `send_publish_notifications` in `evap/evaluation/tools.py`

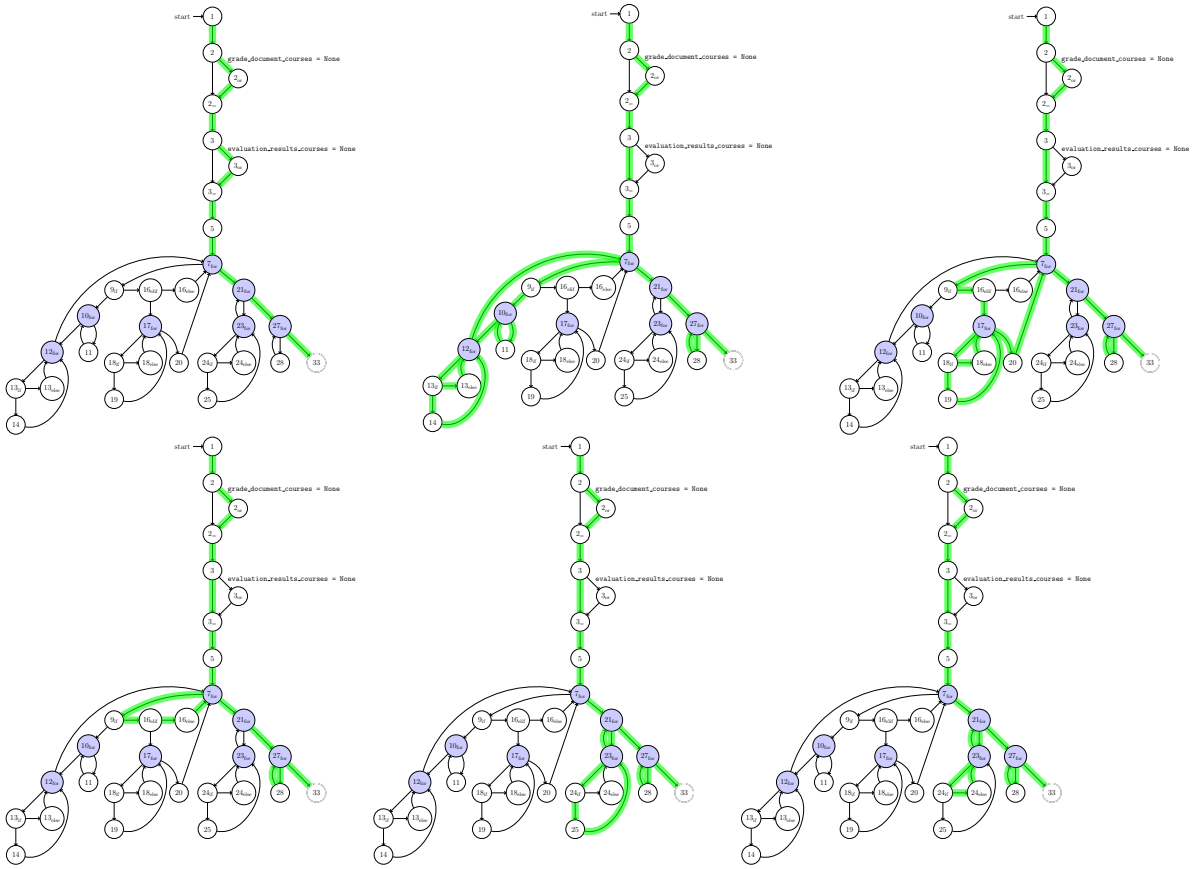

Figure 4: Visualization of the six test cases to cover `send_publish_notifications`.

Pylint The Python quality checker *Pylint* is a tool to help with coding standard as recommended by PEP 8, error detection and refactoring. Pylint message categories:

- (C) convention: programming standard violation
- (R) refactor: code smell
- (W) warning: python specific problem
- (E) error: likely bug
- (F) fatal: an error occurred preventing pylint from continuing the analysis

```
C:\Users\Jennifer\Documents\Studium\EvaP\evap\evaluation>pylint tools.py
No config file found, using default configuration
************* Module evap.evaluation.tools
C: 21, 0: Line too long (106/100) (line-too-long)
C: 22, 0: Line too long (105/100) (line-too-long)
C: 26, 0: Line too long (117/100) (line-too-long)
C: 27, 0: Line too long (104/100) (line-too-long)
C:  1, 0: Missing module docstring (missing-docstring)
C:  8, 0: Missing function docstring (missing-docstring)
C:  3, 0: standard import "from collections import defaultdict" comes before "fr
om evap.evaluation.models import EmailTemplate" (wrong-import-order)
C:  4, 0: standard import "from collections import namedtuple" comes before "fro
m evap.evaluation.models import EmailTemplate" (wrong-import-order)
```

Figure 5: Pylint messages for `send_publish_notifications(..)`

As expected after our testing Pylint found mostly violated coding conventions in `send_publish_notific` (5). The investigation of the file `tools.py` leads to a few warnings and refactoring hints that we will discuss with the developers on the next occasion.

PyCharm

Pychecker Even though you still find some outdated recommendations this tool's peak seems to be over. The last update was in 2013. It is written for Python 2.x and has never been ported to Python 3.x. Since the installation process requires Python 2.x while we work with Python 3.x in EvaP we will drop our investigation with this tool.

pep8 The Python style guide checker *pep8* checks code against some of the style conventions in PEP 8. It differentiates between errors and warnings. pep8 throws a few more errors about lines being too long, because the line length was set to 100 instead of 80 characters in Pylint.

Pyflakes Unlike Pylint and pep8 *Pyflakes* does not check for violations of coding style but instead focuses on checking for errors. The tool is less intuitively to use as there is no report if no errors are found. It did not report any errors for either `send_publish_notifications(..)` or `tools.py`

```
C:\Users\Jennifer\Documents\Studium\EvaP\evap\evaluation>pep8 send_publish_notif
ications.py
send_publish_notifications.py:6:80: E501 line too long (97 > 79 characters)
send_publish_notifications.py:8:1: E302 expected 2 blank lines, found 1
send_publish_notifications.py:8:80: E501 line too long (93 > 79 characters)
send_publish_notifications.py:15:80: E501 line too long (84 > 79 characters)
send_publish_notifications.py:18:80: E501 line too long (89 > 79 characters)
send_publish_notifications.py:21:80: E501 line too long (106 > 79 characters)
send_publish_notifications.py:22:80: E501 line too long (105 > 79 characters)
send_publish_notifications.py:26:80: E501 line too long (117 > 79 characters)
send_publish_notifications.py:27:80: E501 line too long (104 > 79 characters)
send_publish_notifications.py:32:80: E501 line too long (85 > 79 characters)
send_publish_notifications.py:38:80: E501 line too long (84 > 79 characters)
```

Figure 6: pep8 messages for `send_publish_notifications(..)`

Landscape