EvaP

Jennifer Stamm, Stefan Neubert

Winter Term 2015/16

Contents

0. Introduction

0.1. 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.

0.2. 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.

1. Milestone 1: Testing

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.

1.1. 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.

1.2. 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.

1.2.1. 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 implicitly 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*.

These tools are already the most common used ones for open source python development for good reasons, as they provide a good feature set at no costs. Therefore we wont research further tools for bug tracking, source code versioning or automated testing.

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.

1.2.2. 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]
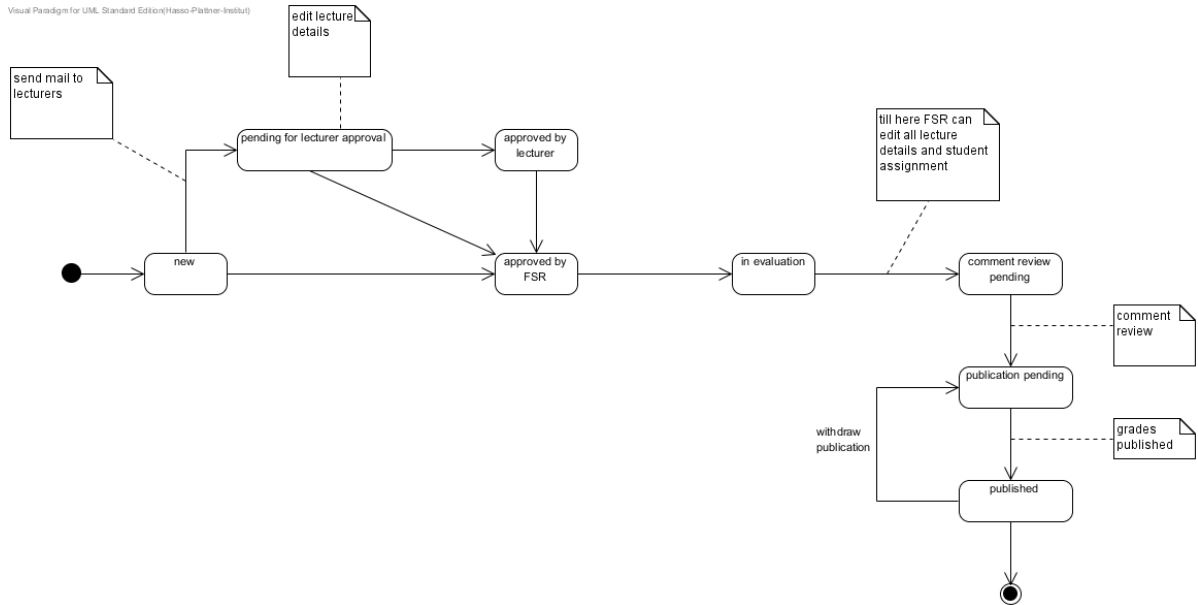
Figure 2: Original FSM: Possible states of a course

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

8

1.2.3. 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 1.1, 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[6], 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[7]. 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://www.jetbrains.com/pycharm/`
[7]`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.

1.3. 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 about reviewing comments before publishing them. Finally, we will investigate if our added test cases changed the line coverage, since this is the given measurement by the used tool *Coveralls*.

1.3.1. 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.

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[8] 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. (Figure 4) The first test case runs the function without arguments, hence testing the usage of the default arguments. Test cases two and three test, whether a course with results notifies its participants and contributers, whereas the fourth test case asserts, that a course without results won't do so. In the fifth and sixth test cases the functions behavior with uploaded grade documents is checked.

Even though the graph seemingly is covered by these test cases, in reality there still

---

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

```python
1   def send_publish_notifications(grade_document_courses=None,
        evaluation_results_courses=None):
2       grade_document_courses = grade_document_courses or []
3       evaluation_results_courses = evaluation_results_courses or []
4
5       publish_notifications = defaultdict(lambda: CourseLists(set(), set()))
6
7       for course in evaluation_results_courses:
8           # for published courses all contributors and participants get a
                notification
9           if course.can_publish_grades:
10              for participant in course.participants.all():
11                  publish_notifications[participant].evaluation_results_courses.add
                        (course)
12              for contribution in course.contributions.all():
13                  if contribution.contributor:
14                      publish_notifications[contribution.contributor].
                            evaluation_results_courses.add(course)
15          # if a course was not published notifications are only sent for
                contributors who can see comments
16          elif len(course.textanswer_set) > 0:
17              for textanswer in course.textanswer_set:
18                  if textanswer.contribution.contributor:
19                      publish_notifications[textanswer.contribution.contributor].
                            evaluation_results_courses.add(course)
20              publish_notifications[course.responsible_contributor].
                    evaluation_results_courses.add(course)
21      for course in grade_document_courses:
22          # all participants who can download grades get a notification
23          for participant in course.participants.all():
24              if participant.can_download_grades:
25                  publish_notifications[participant].grade_document_courses.add(
                        course)
26
27      for user, course_lists in publish_notifications.items():
28          EmailTemplate.send_publish_notifications_to_user(
29              user,
30              grade_document_courses=list(course_lists.grade_document_courses),
31              evaluation_results_courses=list(course_lists.
                    evaluation_results_courses)
32          )
```

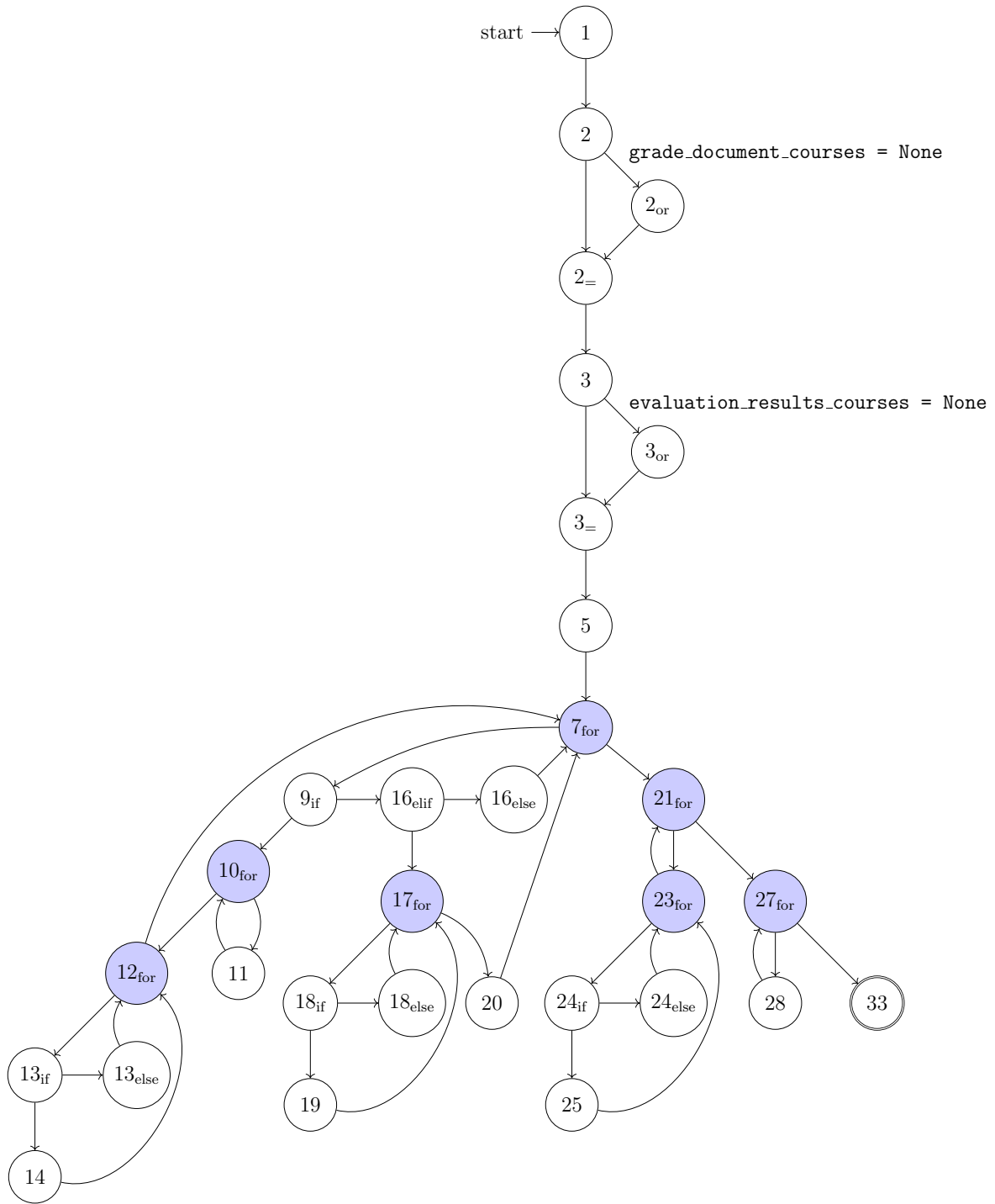Listing 1: The control flow graph is based on this function.

12

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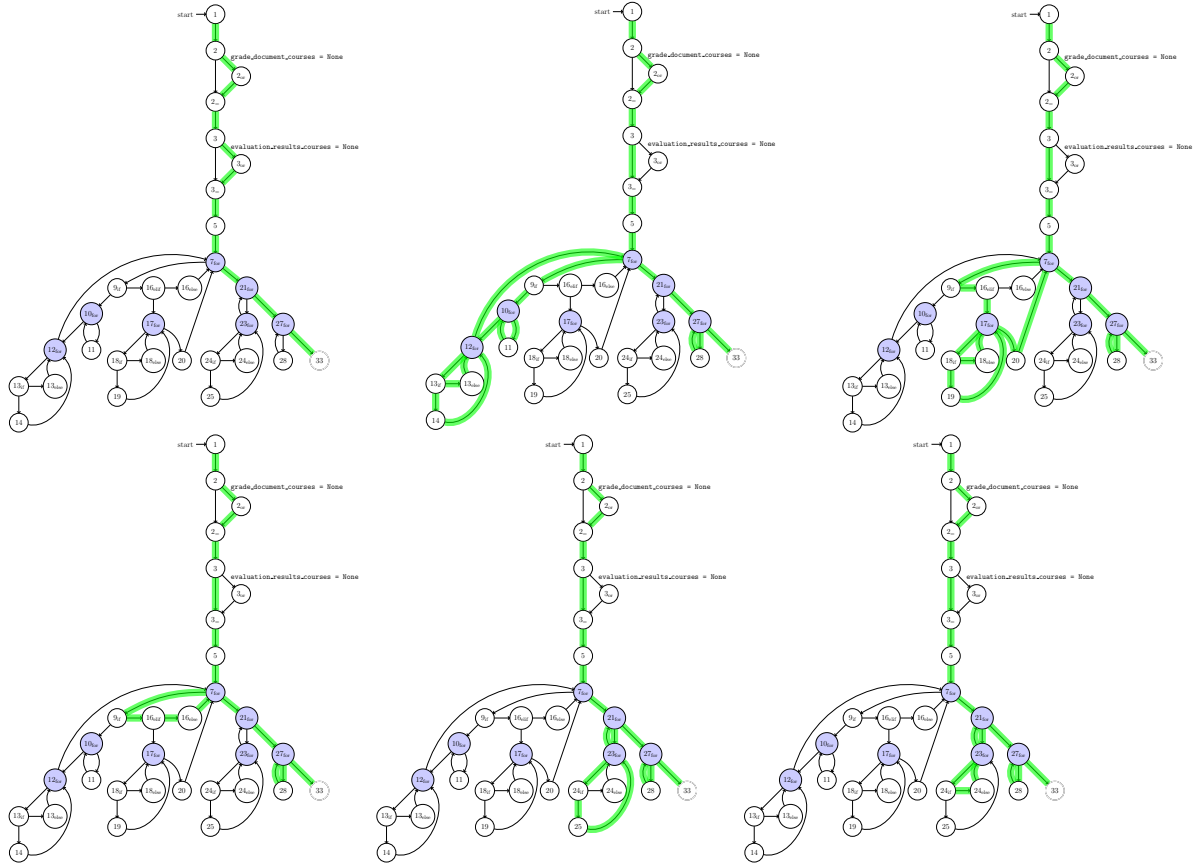evaluation/tools.py

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

would be common usages of the function that remain uncovered by the test set. For example, if a student is participant in two published courses, he should not receive two notifications, but be notified about both changes in one mail. This shows, why Specified Path Coverage can easily lead to a ill-founded feeling of good testing coverage. We will handle some of the remaining special cases in subsection 1.4 where we discuss Input and Mutant Coverage.

1.3.2. Finite State Machine of Course States in the Evaluation Process

We used two methods to investigate the available artifact the FSM describing the states of courses in the evaluation process.

Firstly we executed the system on our machines. We used test data to observe and test:

- if every state of a course exists in the system

- if every transition exists as well as how and by whom the transition is initiated

- if states exist in the system that are not represented in the FSM

14

FSM name	system name
new	new
pending for lecturer approval	prepared
approved by lecturer	lecturer approved
approved by fsr	approved
in evaluation	in evaluation
comment review pending	evaluated
publication pending	reviewed
published	published

Table 1: Comparing state names in FSM with state names in system

- if transitions are possible that are not represented in the FSM

We found that some names of states differ between the Final State Machine and the system (Table 1.3.2).

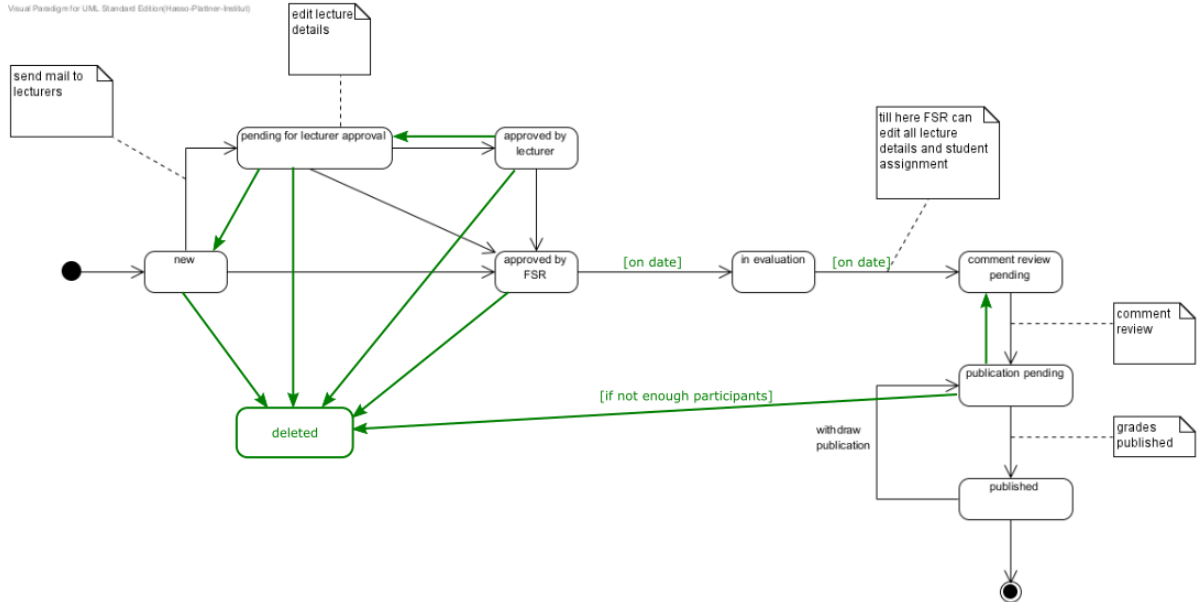Furthermore we found missing states and annotations (Figure 5).

Figure 5: Updated FSM: Possible states of a course

Secondly, we were able to compare the FSM directly with the implementation found in `evap/evaluation/models.py` (Excerpt shown in Listing 2). A direct comparison is easy as there is a fsm package for the used webframework Django that was used to implement the FSM.

Since the FSM was created several years ago, we suspect that either it was not complete from the start or it became outdated after updates to the code base. It is a pleasant suprise that there is a module that supports the direct implementation of FSMs since we learned that deriving FSMs is the hardest task if working with them.

```
1  @transition(field=state, source=['new', 'editorApproved'], target='prepared')
2      def ready_for_editors(self):
3          pass
4
5  @transition(field=state, source='prepared', target='editorApproved')
6  def editor_approve(self):
7      pass
8
9  @transition(field=state, source=['new', 'prepared', 'editorApproved'], target='
       approved', conditions=[has_enough_questionnaires])
10 def staff_approve(self):
11     pass
```

Listing 2: Excerpt of the implementation of course states.

1.3.3. Use Case, Elaboration and Activity Diagram of Reviewing Comments

One of the most important features of EvaP is the ability to review comments before publishing the results. This allows the responsibles to prevent hurtful and deconstructive comments from being visible to the lecturers. Even more, the responsibles can change the comment in a way that the lecturer still receives the criticism. This is the reason for the course state *comment review pending*. The described use case is documented in German on the Github wiki mentioned above (`https://github.com/fsr-itse/EvaP/wiki/Use-Case%3A-Review-Comments`).

Following is the use case elaboration translated, updated and adapted to the template presented in the lecture (Table 1.3.3). The activity diagram (Graphic 6) used to explore scenario testing and to create test cases is based on this elaboration.
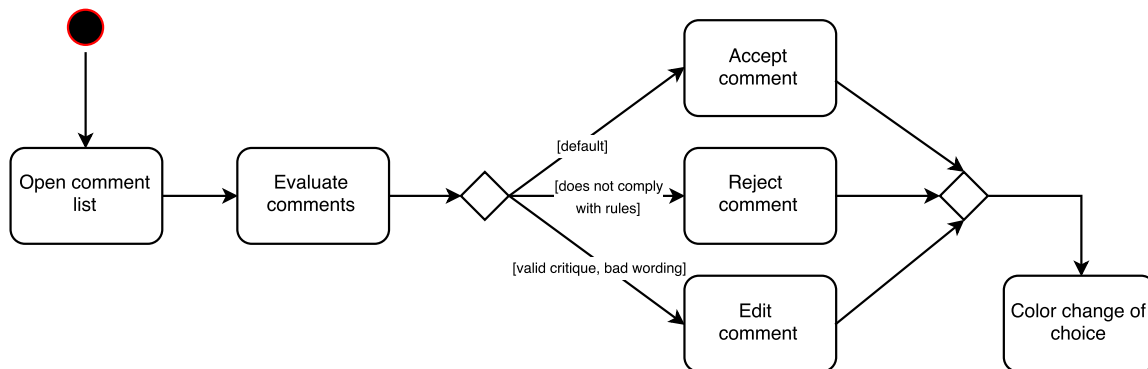

Figure 6: Activity diagram of the use case *Review Comments*

Scenario Testing — Application: Learn the Product   As a student representative you are responsible for supervising the evaluation process. This includes reviewing all

Use Case Name	Review Comments
Summary	Student representatives review comments for violation of comment rules before publishing course results
Actor	Student representatives (FSR)
Precondition	A course has been evaluated and entered the state *comment review pending*
Description	1. FSR opens list with comments of a course 2. FSR evaluates comments regarding the comment rules 3. FSR accepts all comments and marks them as *publishing: yes* 4. FSR receives a small notices as the mark turns from white to green and fades back to a gray
Alternatives	3. FSR rejects one or more comments and marks them as *publishing: no* 3. FSR rejects one or more comments and edits the comments before marking them as *publishing: yes* 3. FSR marks one or more comments and marks them as *publishing: private* if they should only be visible to the specific tutor not all lecturers
Postcondition	All comments marked as *publishing: yes* or *publishing: private* comply with the comment rules

Table 2: Use case elaboration of reviewing comments

comments and validate their compliance with the comment rules. After a course in evaluation enters the state *comment review pending* open the list with comments. They are arranged in groups according to the question answered. For each comment read it carefully and evaluate its compliance with the comment rules. Is the comment constructive critique? Could its content hurt the lecturers? Is it about a specific lecturer and could it be harmful if another lecturer read it? Depending on your evaluation mark the comment as either *publishing: yes*, *publishing: no* or *publishing: private*. Observe the color change of your choice from white to green and its fading to grey. This will help to ensure that you did not accidently mark the wrong choice.

Scenario Testing — Experience Review While writing the testing scenario above we noticed that its application, in our case to learn the product, limits its use. This testing scenario is useful for testers and eventually new users. But, in comparison to coded tests, there is no automatic run of the scenario testing possible, respectively no automatic evaluation of a test run is possible. Therefore, there is no automatic feedback to the developer if the scenario is still operable after changes to the code. Furthermore, there is no direct evaluation available to the manager who normally can view a green light if all automatic tests run through.

Additionally, we noticed a few risks involved with relying on scenario testing.

A scenario involves several features that are even based on each other. If a tester is not able to work through a scenario, the reason might not be obvious. A tester might not be able to know which of the involved features failed. A scenario testing might not always help with locating a bug.

It is thinkable that a testing scenario exposes design errors rather than code errors. In our example the tester might wonder why there are not all three options *yes, no, private* available for a comment. To find the reason you need the knowledge that the comment answered a generic question instead of a question tailored to a specific lecturer. That the knowledge is not obvious on the webpage but needed as background knowledge, could be seen as a design error. The code however does not contain this as an error.

As a software project has only limited resources available it is a risk to rely on scenario testing. Since a tester runs the scenario testing, repetition of testing the same scenario is expensive. Furthermore, repetition of a scenario testing could be unnecessary. Communication between developers and testers is essential to avoid repeating a scenario testing after no changes influencing the features were made. And after several repetitions a variation of the testing scenario might be preferable as it could detect bugs not covered by the previous run.

While scenario testing definitely has a very specific use and purpose it was pretty fun to write a hypothetical story to help a tester work through the use case. It is an obviously creative way to generate tests for a software. This allows it to ease the daily routine of a developer/tester. Additionally, following a story allows a better insight into the system than following the manual. It is in man's nature to love stories.

Test Cases   There is no unit test for the actual code that offers the three marking options to the student representatives while reviewing comments. The source code lies in `evap/staff/views.py` in the function `course_comments_update_publish()`. But following the use case we added a test, that checked if a marking is saved correctly in the model in `evap/staff/tests.py`. There are already tests that check whether a comment marked as private is visible to an unauthorized result viewer in `evap/results/tests.py`.

Even though it would be pleasant to have a test case covering the offering of the three options and the marking, the more important part of this use case, the decision, can not be tested. As the activity diagram shows a critical part is the decision, if a comment is to be marked as *publishing: yes*, *publishing: no* or *publishing: private*. At the moment this decision has to be made by the student representatives. Even the student representative do not have documented guidelines for their decision. Currently, it is based loosely on the aspect of constructive critique and empathy.

We can imagine that in the future an assistant system could help with the decision. The default decision could be to publish comments as most comments are published at the current time. Based on wording it could suggest to not publish comments containing swearwords. Such a system could not only be used to help with the decision but also with checking decisions made. It could highlight comments marked different from the suggestions. Such a system would allow some tests regarding the presented use case.

1.4. Input and Mutant Coverage

Whilst testing the function `send_publish_notifications` we experienced a great difference between formal graph coverage criteria and actual sensible test cases. A more proper way to evaluate the readiness of this function concerning the test coverage would therefore be to assess its ability to deal with common input data correctly.

One kind of input we had not yet tested in subsection 1.3, was if the system behaves reasonable if a participant or contributer was part of two courses which are published. In this case, the system nonetheless should only send one notification message per participant and contributer, and not one message per participation and contribution. For this specific case, we were able to simply extend one of our six test cases to publish two courses with common participants and contributers whilst keeping the expected number of sent messages the same.

As this still is a very manual way of determining which kind of input is produced, the EvaP developers integrated a way to use anonymized real-world data for testing. The anonymization process in `evap/evaluation/management/commands/anonymize.py` involves complete exchange of names and minor shuffling of course data. This data could then be used to perform input coverage testing, as it quite precisely models the possible inputs. An obvious problem occurs, if — despite this data containing years of real-life interactions with the system — a unprecedented input data constellation forms. Even though this is unlikely, this test input data would still not guarantee correctness of the function under test. As the software is in use at the HPI, one can however assume, that as soon as such constellation occurs and an failure is produced, this incident will be reported by the involved students.

Currently there is no implementation of a test that actually uses this input data, as this involves a great amount of additional manual preparation and specification of expected outputs.

Another approach to increase the conclusiveness of our test set would be to run the same test set on modified code and make sure, all mutants are killed.

Unfortunately, currently there seems to be no tool that supports automatic mutation testing on a Django project. The most promising application 'mutpy'[9] brings this testing technique to python 3, but crashes when run on projects depending on the Django library.

Therefore we performed manual mutation testing and applied changes to our function under test. These changes included alterations of boolean expressions in conditions and changes to the foreach-loops, as well as commenting out code. All mutations that did not produce syntax- and therefore runtime errors were successfully killed by our test set which suggests, that it is able to distinguish most false behavior from expected behavior. As there was no documentation on the expected behavior and the function's code was the template for the written tests, this is not a big surprise, but merely indicates, that our test code is correct. However, the test set can be valuable once the code tested changes, as it then can help to identify changing behavior as well.

2. Milestone 2: Analysis

2.1. Tools for Automatic Static Analysis

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

2.1.1. 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

As expected after our testing Pylint found mostly violated coding conventions in `send_publish_notific` (7). 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.

---

[9]https://bitbucket.org/khalas/mutpy

```
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 7: Pylint messages for `send_publish_notifications(..)`

2.1.2. 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.

2.1.3. 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

```
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 8: pep8 messages for `send_publish_notifications(..)`

about lines being too long, because the line length was set to 100 instead of 80 characters in Pylint.

2.1.4. 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 neither `send_publish_notifications(..)` nor `tools.py`

2.1.5. Landscape

As described above the service Landscape is currently deployed in the development process. The service advertises to find errors, possible problems and security issues. Every time the master branch of the repository is updated Landscape runs several code quality tools including pylint on the whole codebase. It accumulates the findings, creates an easy access to them. A rating based on the findings helps to get an overall idea about the current status.
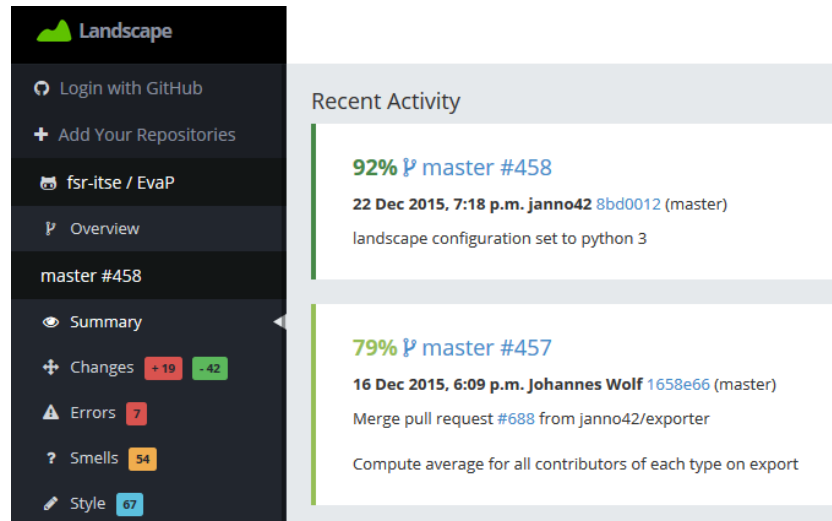

Figure 9: Landscape's rating of the code after changing the configuration from Python 2.x to Python 3.x

During our first investigation of the tool, we noticed that something was off about its configuration. Landscape was still configured to run its checks against Python 2.x while EvaP already made the switch to Python 3.x. Thankfully a fix of the configuration gave a significant boost to the rating (Figure 9).
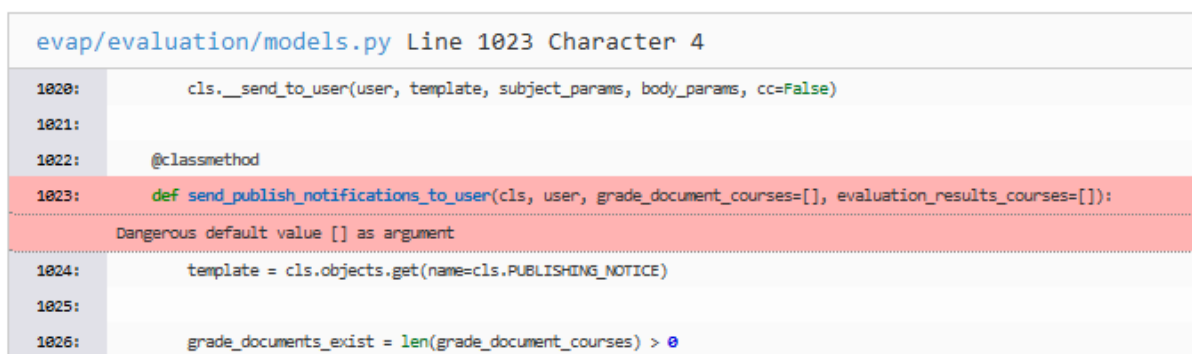

Figure 10: Landscape's display of an error found with static analysis

As we checked out the errors found section, we noticed that one of the found errors ac-

tually might have an impact on our test example function `send_publish_notifications`. Landscape notified that the method `send_publish_notifications_to_user` has a dangerous default value as an argument. (Figure 10) In production, this method is called by our function, during testing we mocked it.

In the problematic code, a developer decided to make the function's arguments optional. As both arguments are expected to be lists, he wanted the default argument to be empty lists. The expected behavior was, that when the function is called without the respective arguments, the missing arguments would be set to empty lists. Python however parses the definition of the function when initially parsing the file. This is also when the default arguments are evaluated, therefore instead of creating new empty lists as default parameters on each execution of the function, those lists are only created once.

The found code problem is obviously a fault, but it does never become an error or failure in the current system, as the function neither adds values to passed lists, nor is it called more than once during one script execution. But if future usages of the function — within EvaP or another project — would change that, the fault could potentially become an error and failure.

A possible fix for the fault is to set the default arguments to `None`, and to create empty lists inside the function's body, as can be seen in `send_publish_notifications`.

2.1.6. PyCharm

The static analyses that can be conducted with PyCharm far outreach the possibilities of the other tested tools. Not only allows the IDE to search for possible faults in several types of source code (i.e. Python, JavaScript, HTML, CSS, Django templates), but it also suggests automatic refactoring to solve these issues.

All found code issues are classified by their severity into the categories 'Server problem', 'Typo', 'Info', 'Weak Warning', 'Warning' and 'Error'. One can both customize the type of issues and the severity of those that should be found by PyCharm. This is an important feature as — whilst providing good inspections for a series of possible faults — PyCharm tends to be pessimistic inaccurate. As PyCharm offers to find issues of around 400 types (a number that even can be increased with additional IDE plugins), the result list of the inspection is rather long and verbose when it is run with default settings. Especially concerning the English language checks, PyCharm heavily tries to match all occurring words with spelling rules. When it comes to variable names, this is often misleading and tends to produce loads of false positives. (Figure 11)

Included in the list of issue types are both style guide or code convention violations and possible bugs due to actual faults. For example, PyCharm checks for complete source code documentation, makes assumptions on the developer's intentions in contrast to possible typing errors and tries to apply data flow analysis to find uninitialized or unused variables.

With the help of the PyCharm code inspection, 15 faults could easily be fixed by either automatic refactoring or simple manual changes. These fixes included false missing HTML tags and wrong tag nesting, dangerous Python variable initialization and comparison as well as minor code convention violations. About a hundred additional code
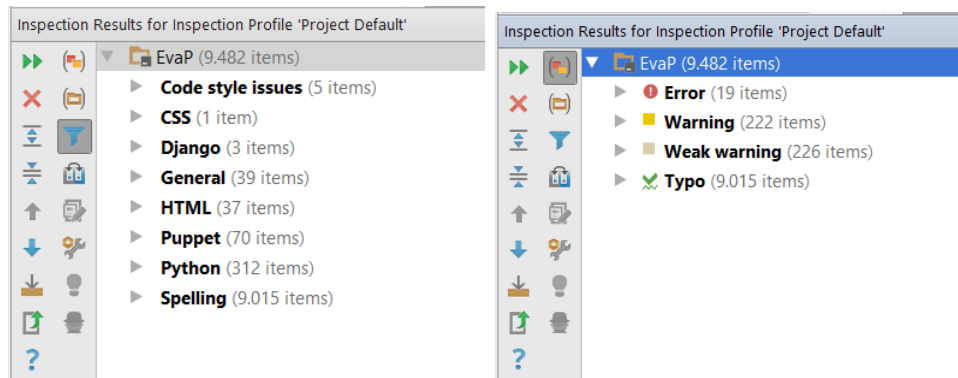
Figure 11: With the default configuration, PyCharm finds almost 9.5 thousand issues in the EvaP project. Most of them are false positives.

smells can be fixed with small effort.

In addition to code inspections, PyCharm analyses the module dependencies and suggests to update them if the used version is outdated. This feature is roughly equivalent to the data provided by the online service *Gemnasium*, that monitors the project's GitHub repository.

2.2. Experiences summarized

Out of the tested tools three were particularly useful.

- Pylint is pretty verbose, but very helpful if you are working on adjusting one file to the style guide. Its report is especially nice because it delivers statistics about the previous and the current run to compare the improvements due to the last code changes.

- PyCharm is especially useful for developers, because its inspection leads to the source code itself. Additionally PyCharm suggests fixes for its findings which enhances and accelerates the process quite a lot.

- Landscape's advantage is its setup as a service. It allows everyone to get an overview over the code quality without installing a tool on their machine. Furthermore, its automatic run after every code change on the master allows comparison between two states.

It is unfortunate that a tool as PyChecker that has not been updated for a while and does not work for Python 3.x is still listed and even highly recommended. While the tools pep8 and Pyflakes were not outstandingly useful for us they seem to serve their purpose.

The tools' usefulness regarding testing differs as well:

1. PyCharm is the most useful, because it is simultaneously the IDE

2. Landscape is helpful as it does not only locate possible erros, but suggest why a tester should be rethink the code or write a test

3. pylint and pep8 are great, because a uniform code style helps new developers and testers to easily find into the project.

Additional screenshots of the tools' results can be found in Appendix A.

3. Milestone 3: Verification

Verification of a software artifact is difficult if not impossible for non-trivial properties. However, provided with certain limitations, a good tool can still be able to for example find infinite loops, detect dead locks or dangerous concurrent data modification.

3.1. Verification on Python

With Python being a dynamically typed programming language, the verification process gets even more problematic, as hardly any information on a variable used in the algorithm is static: Types and values can change, and due to *duck typing*, an object can be interpreted as different types at the same time. As long as one tries to verify properties of side-effect free, short code scopes, some certainty about correctness can still be gained.

This, for example includes the availability of needed imports and the question whether a variable has been initialized or not. Such properties can be found using the already presented tools for code analysis. As these tools consequently stick to pessimistic inaccuracy for some properties, they can guarantee to find some faults if present. If no errors are reported, one can be sure that concerning this error type the code is bug-free. Most times, however, one has to manually work through a list of false positives beforehand.

3.1.1. Verification on EvaP

For our project specifically we were not able to extend the verification process to more than the already discussed analysis. This is due to three reasons: Firstly, EvaP consists mainly of code for request routing, data management and HTML output. All three fields are heavily supported by the Django framework. Therefore, they are not within our scope to verify. Secondly, even small methods usually rely on huge data models. Whilst this might be a hint that the current software architecture could be improved, as dependencies of small modules are too broad, it still leaves us with the problem of teaching a verification tool the very data structure. This is aggravated by the fact, that most verification tools only work on primitive data types. Thirdly, EvaP suffers from the problems of Python as discussed above.

```
1  def mix(a, b, alpha):
2    if a is None and b is None:
3      return None
4    if a is None:
5      return b
6    if b is None:
7      return a
8
9    return alpha * a + (1 - alpha) * b
```

Listing 3: The mix function from `evaluation/tools.py` that interpolates between two numeric values.

```
1  def mix2(a, b):
2      if a == 0 and b == 0:
3          return 1
4      if a == 2:
5          return 2
6      if b == 3:
7          return 3
8
9      return 4
10
11 def expected_result_set():
12     return [1, 2, 3, 4]
```

Listing 4: The modified function for verification with the same branch structure as the mix function.

3.1.2. Python Explorer with Z3

Even though Python is an unaccommodating ground for proofs, verification of dynamic programming language is a current and important topic. For Python specifically we have found PyExZ3[10] — a verification tool that tries to perform symbolic execution on Python code with the help of the Z3 theorem prover[11]. As result of a symbolic execution, the tool aims to supply a tester with a set of input data, that covers all possible execution paths of the function under test.
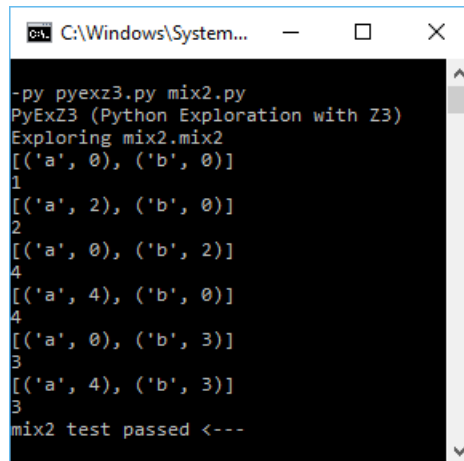
We tried to apply the Python Explorer on a small, side-effect free function of EvaP, that is used to interpolate numeric values. (Listing 3)

Unfortunately, PyExZ3 cannot detect `None`-values, neither can it run symbolic execution with floating point values. This shows, that even a tiny code fragment can be to complex to be properly verified with current tools.

PyExZ3 was able to find all paths through a modified version of the script, that

---

[10]https://github.com/thomasjball/PyExZ3
[11]http://z3.codeplex.com/

Figure 12: PyExZ3 produces possible input data for test cases.

shares the same branching structure, but only uses integrals. (Listing 4) The additional function `expected_result_set` helps PyExZ3 to determine, whether it found all possible execution paths. (Figure 12)

4. Conclusion

A. Results from ASA

```
Raw metrics
-----------

+-----------+--------+-------+----------+------------+
|type       |number  |%      |previous  |difference  |
+===========+========+=======+==========+============+
|code       |239     |70.29  |30        |+209.00     |
+-----------+--------+-------+----------+------------+
|docstring  |11      |3.24   |0         |+11.00      |
+-----------+--------+-------+----------+------------+
|comment    |17      |5.00   |3         |+14.00      |
+-----------+--------+-------+----------+------------+
|empty      |73      |21.47  |7         |+66.00      |
+-----------+--------+-------+----------+------------+


Duplication
-----------

+-------------------------+------+----------+------------+
|                         |now   |previous  |difference  |
+=========================+======+==========+============+
|nb duplicated lines      |0     |0         |=           |
+-------------------------+------+----------+------------+
|percent duplicated lines |0.000 |0.000     |=           |
+-------------------------+------+----------+------------+


Messages by category
--------------------

+-----------+--------+----------+------------+
|type       |number  |previous  |difference  |
+===========+========+==========+============+
|convention |59      |7         |+52.00      |
+-----------+--------+----------+------------+
|refactor   |2       |0         |+2.00       |
+-----------+--------+----------+------------+
|warning    |4       |0         |+4.00       |
+-----------+--------+----------+------------+
|error      |0       |1         |-1.00       |
+-----------+--------+----------+------------+


Messages
--------

+-------------------+-------------+
|message id         |occurrences  |
+===================+=============+
|line-too-long      |32           |
+-------------------+-------------+
|missing-docstring  |15           |
+-------------------+-------------+
|invalid-name       |6            |
+-------------------+-------------+
|unused-variable    |4            |
+-------------------+-------------+
|wrong-import-order |3            |
+-------------------+-------------+
|bad-whitespace     |3            |
+-------------------+-------------+
|too-many-locals    |2            |
+-------------------+-------------+
```

Figure 13: Pylint messages for `evaluation/tools.py`

```
Global evaluation
-----------------
Your code has been rated at 6.52/10 (previous run: 5.00/10, +1.52)
```

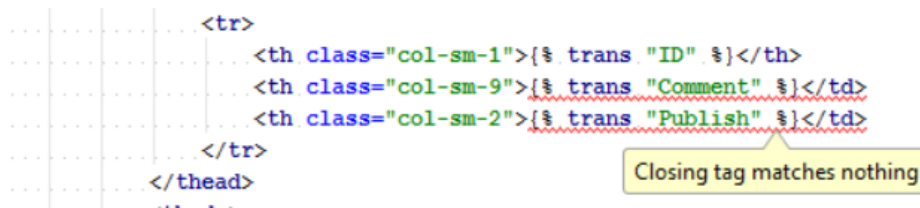Figure 14: Pylint rating for `evaluation/tools.py`

Figure 15: PyCharm Inspection result: the closing tag must be a `</th>` as well.
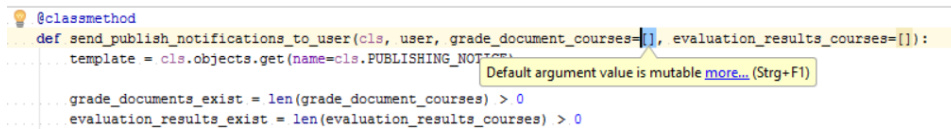

Figure 16: PyCharm's Inspection finds the same dangerous default argument as Landscape.
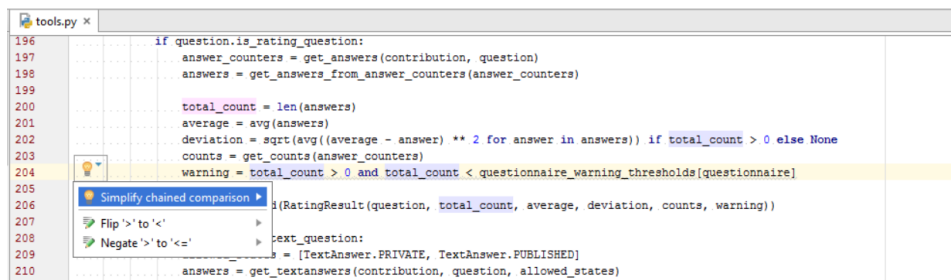

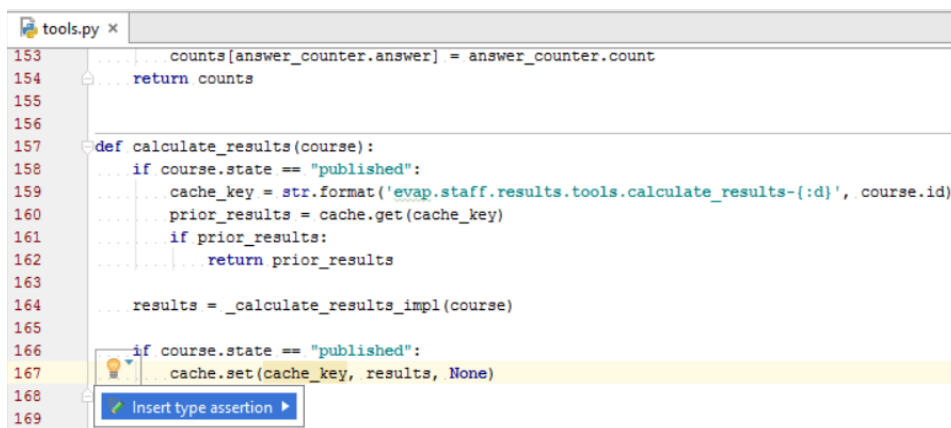Figure 17: PyCharm suggests code changes for better code style.

Figure 18: PyCharm cannot know that the course state does not change within the method and therefore believes, that the highlighted variable could be uninitialized.