Conversion of User Story Scenarios to Python-Based Selenium Source Code for Automated Testing

Indra Kharisma Raharjana [1], Muhammad Faris Arifin [1], Ahmad Iklil Nur [1], Nadlir Mubarak [1]

[1] *Information Systems, Faculty of Science and Technology, Universitas Airlangga, Indonesia*

Abstract **–A user story is an artifact in software development processes that can be used for automated test cases. This study proposes a terminal-based program for converting user stories into source codes for automated testing. We also propose a modified user story format that can be directly converted into functional Python-based Selenium source codes. We then implement the program by creating user stories for several local Django projects and using the converted codes. The implementation results show that the code from the program has a reliability rate of 72% for successfully executing it as the users' intentions.**

Keywords **– automated testing, user story, process innovation, source code conversion, selenium**

1. Introduction

The user story is an artifact used in the agile software development process [1], [2]. The user story describes a feature from the perspective of the person who needs the software [3]. It consists of a written text, the conversation about it, and acceptance criteria [4]. Around 90% of agile practitioners implement user stories as their method to elicit requirements for their software [5], [6], [7]. User stories can also be used as the basis for test cases in software testing processes [8]–[10]. Acceptance criteria assigned to user stories are used as test cases and merged into acceptance tests, which will then be used to check the system behavior compliance with the user stories [11].

Software testing can be categorized into two types, which are manual and automated testing [12], [13]. Manual software testing is a process in which human experts execute the tests and ensure that the software behavior is according to expectations [13]. However, manual testing can be time-consuming and costly [14]. Hence automation of the testing process is a possible solution to alleviate the problem. Automated testing is done using a software tool to test or check the software execution [15]. There are several benefits of implementing automated testing, such as faster execution than manual testing, more time for the testers, and the ability for the developers to create automated tests that often use the same programming language as the software product [16]. This also means that using generator tools for automated test cases could accelerate software development [17], [18], [19].

This research proposes a software application that generates Python-based Selenium source codes from user stories defined by the users. The generator takes user stories with a modified format, which converts them into functional source codes. The source codes are then implemented into projects using the Django framework and executed with the built-in testing function from Django.

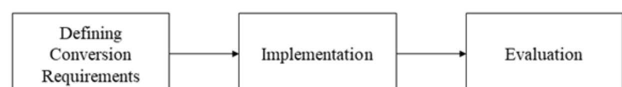
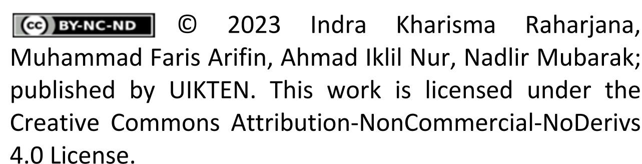


Figure 1 Research procedure

DOI: 10.18421/TEM121-39
https://doi.org/10.18421/TEM121-39

Corresponding author: Indra Kharisma Raharjana,
Information Systems, Faculty of Science and Technology, Universitas Airlangga, Indonesia
Email: indra.kharisma@fst.unair.ac.id

Received: 16 October 2022.
Revised: 05 February 2023.
Accepted: 09 February 2023.
Published: 24 February 2023.


© 2023 Indra Kharisma Raharjana, Muhammad Faris Arifin, Ahmad Iklil Nur, Nadlir Mubarak; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at https://www.temjournal.com/

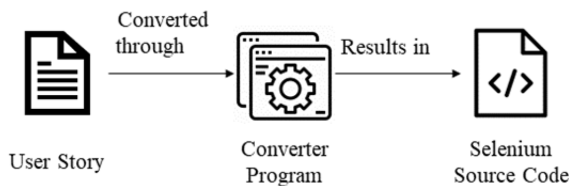


Figure 2 Procedure for converting user stories to functional Selenium source codes

2. Methods

The procedure of this research is shown in Figure 1. We start by defining the requirements for the conversion process, which consists of the user story and its format as the system's input and the source code as the conversion result.

After that, we implement the converted results into several Django projects by first analyzing the projects' HTML elements and their navigation and then creating the user stories based on the defined format. We then run the tests using Django's built-in testing function and the codes as its test cases. After compiling the results of the tests, we evaluate them to measure the reliability of the converter program.

2.1. Defining conversion requirements

The converted program is a terminal-based Python program. To execute it, the user must run it using a Python IDE, such as IDLE or PyCharm. There are two primary components needed to be taken into consideration for the functions of the program. The first one is the input requirements, while the second one is the output requirements.

Feature: Descriptive text of what is desired in order to realize a named business value as an explicit system actor (as an explicit system actor, I want to)

Scenario: Some determinable business situation
Given: some precondition
And: some other precondition
When: some action by the actor
And: some other action
And: yet another action
Then: some testable outcome is achieved
And: something else we can check happens too

Scenario: A different situation
Given: some precondition
When: some action by the actor
And: some other action
Then: some testable outcome is achieved
And: something else we can check happens too

Figure 3 User Story Scenario Format

```
Webdriver: C:/Users/user/ChromeDriver/chromedriver.exe

Feature: Login User

Scenario: Login Success
Given: I'm on #http://127.0.0.1:8000/login
When: I @fill "user3" on {id_username}
and: I @input "headeyes" on {id_password}
and: I @click on {btn_login}
Then: I should be on #http://127.0.0.1:8000/tutorial
and: I should @see "Tutorial"

Scenario: Wrong username, then check the URL
Given: I'm on #http://127.0.0.1:8000/login
When: I @fill "notuser3" on {id_username}
and: I @fill "headeyes" on {id_password}
and: I @click on {btn_login}
Then: I should still be on #http://127.0.0.1:8000/login
```

Figure 4 Example of a user story using the modified format

The former revolves around the contents of user stories that could be taken and used by the program, while the latter is the desired Python-based Selenium source code as its outcome. Figure 2 shows the conversion process and the components involved in it.

We uses Behat Cheat Sheet user story format for the input requirements as a reference. Figure 3 shows an excerpt from it which defines the contents of a user story. While Figure 4 shows an example of a user story using the modified format. We then analyze the format and implement the rules to convert it into a functional Python-based Selenium source code. The rules for the modified user story format are as follows:

1. The file format of the user story must be a .txt file.
2. The path of the ChromeDriver executable file must be stated first. ChromeDriver is another version of WebDriver which enables automated testing through the Google Chrome browser. This is necessary for executing the Selenium program.
3. Actions by the actors do not involve managing external files, such as uploading and downloading a file. This is because it is the limitation of the converter program.
4. Rules for the scenario parts:
   a. Given (only 1), with the format:
      i. I'm on #URL
   b. When (1 or more), with the format:
      i. I @input/fill “text” on {html_element_id}
      ii. I @click on {html_element_id}
      iii. I @click “text” on {html_element_id}
      iv. I @go to #URL
   c. Then (1 or more), with the format:
      i. I should @see {html_element_id}
      ii. I should @see "text"
      iii. I should be @on #URL
      iv. It should @contain "text"
      v. It should @contain {html_element_id}

The symbols in the scenario parts are used to mark specific elements for the program to find certain parts more effectively. The functions of the symbols are:

- \# = mark an URL, for example: #http://…
- @ = mark an action, for example: When: @input/@fill/@click; Then: @see/@on/@contain
- “" = mark a text, for example: "Sign In Success", "Tutorial Player"
- {} = mark a HTML element ID, for example: {id_username}, {id_button}, {id_carousel}

Table 1 Page layout description

Scenario Part	Condition	Code Result
Given	If the URL is defined.	*selenium.get({URL})*
When	If the action is either input or fill and element ID is defined.	*variableName = selenium.find_element_by_id({id_element})* *variableName.send_keys({input})*
	If the action is click and element ID is defined.	*variableName = selenium.find_element_by_id({id_element})* *variableName.click()*
	If the action is click, a text and element ID is defined.	*variableName = selenium.find_element_by_id({id_element})* *for option in variableName.find_elements_by_tag_name("option"):* *if option.text == "text":* *option.click()*
	If the action is go and URL is defined.	*selenium.get({URL})*
	If the action is either see or contain and text is defined.	*assert “{text}” in selenium.page_source*
	If the action is either see or contain and the element ID is defined.	*if selenium.find_element_by_id(“{id_element}"):* *assert True* *else:* *assert False* *assertEqual(selenium.find_element_by_id({id_element})*
	If the action is on and the URL is defined.	*assertEqual(selenium.current_url, {URL})*

After defining the rules and the user story format, we define the lines of code that are expected as its outcome. The content of the source code are as follows:

1. Necessary Django and Selenium testing modules. These modules are imported automatically before going through the user story. The lines of codes are:
   - fromdjango.testimport LiveServerTestCase
   - from selenium import webdriver
   - fromselenium.webdriver.common.keys import Keys
   - from Selenium.common.exceptions import NoSuchElementException
2. Name of the class is taken from the feature name defined in the user story.
3. Name of the functions, which are taken from the name of each scenario. This means that each scenario represents one test function.
4. In each test functions, two lines of codes are defined automatically, which are:
   - driver = webdriver.Chrome("{path to the webdriver file}")
   - selenium = driver
5. After that, in each test functions, the scenario parts are defined. Table 1 shows the scenario parts, the conditions, and the resulting code

Table 2 Results of testing the converted code using Django testing function

Project Name	Feature Name	Scenario	Actual Result	Expected Result
	Login User	Login success	Passed	Passed
		Wrong username, check URL	Failed	Passed
		Wrong password, see if element exists	Passed	Passed
		Empty username and password, check URL and see if element exists	Failed	Passed
		Click on Sign Up	Passed	Passed
	Register User	Register success	Passed	Passed
		Register with existing user	Passed	Passed
		One of the field is not filled	Passed	Passed
		Two password fields are incorrect	Failed	Passed
	Manage Project	Add new project	Passed	Passed
		Search existing project	Failed	Passed
		Check project detail	Passed	Passed
		Edit project	Error	Passed
		Delete project	Passed	Passed
		Add feature with 1 scenario	Passed	Passed
		Add feature with 2 scenarios	Passed	Passed
		Cancel creating a feature	Error	Passed
		Search valid feature	Passed	Passed
		Generate valid sequence diagram	Error	Passed
		Generate invalid sequence diagram	Failed	Passed
		Edit feature name	Passed	Passed
		Cancel editing a feature	Passed	Passed
		Delete feature	Error	Passed

UserStoryScenToUseCaseSpec	Login	Login success	Passed	Passed
		Wrong username	Passed	Passed
		Wrong password	Passed	Passed
	Register	Register success	Passed	Passed
		Password similar to email address	Passed	Passed
		Password is too short	Passed	Passed
		Password checker doesn't match with password	Passed	Passed
	Manage Project	Manage project	Passed	Passed
		Detail project	Error	Passed
		Delete project	Error	Passed
	Manage Detail	Edit project	Passed	Passed
		Add scenario	Passed	Passed
	Login	Login success	Passed	Passed
		Wrong username	Passed	Passed
		Wrong password	Passed	Passed
		Empty username and password	Passed	Passed
	Register	Register Success	Passed	Passed
		Register with existing user	Failed	Passed
		Two password fields did not match	Passed	Passed
		Register with at least one empty field	Passed	Passed
		Register with short password	Passed	Passed
	Manage Project	Create New Project	Passed	Passed
		Edit Project	Failed	Passed
		Delete Project	Passed	Passed
	Manage Use Case	Create New Use Case	Passed	Passed
		View Use Case	Passed	Passed
		Delete Project	Error	Passed

2.2. *Implementation*

Implementation is done by choosing and analyzing Django projects first. This is needed to identify the behavior, navigation, and HTML elements that are used in the projects. For the projects, we chose three local Django projects taken from AgileRE-2021 project from GitHub, which are UserScenario2Seq, UserStoryScenToUseCaseSpec, and UseTivy. These projects generate UML artifacts, with their primary functions involving user input through forms. It should be noted that the chosen projects do not have features related to managing files, such as uploading and downloading a file.

After identifying the necessary parts of the projects, we create user stories based on them.

The user stories are then converted using the program. After converting it, the codes are inserted into the projects and executed with the built-in Django testing function. Using the testing function yields three possible results, in which the possible outcomes for each scenario are: Passed, Failed, and Error.

2.3. Evaluation

The codes' results are then evaluated to measure the program's reliability. Measurement is done by comparing the actual results during implementation with the expected results. After that, the results that match the expected ones are summed, then divided by the total of the test cases done.

3. Result and discussion

We analyzed the projects and made a total of 12 features and 50 scenarios as the test cases. The results of the tests can be seen in Table 2.

We found that over 36 test cases have passed, seven have failed, and seven have resulted in errors. The same 36 test cases also match the expected results. This means that the tests had been successfully executed as intended with a rate of 72%.

While the results did indicate that the converter program proved to be acceptable, there were, however, issues faced throughout the implementation process. The issues that were faced are as follows:

1. Testing fails if the specified project features are locked behind the project's authentication system (for instance, a feature requires a user to be logged in first to access it). To solve this issue, we used a workaround by stating the authentication URL and submitting the required data first before accessing the project feature in the user story. This workaround, however, made the user story deviate from its convention.
2. Scenario tests are not executed chronologically due to how Selenium works. This may lead to certain test cases not being executed properly. For instance, the user may want the test cases to be executed in chronological order, such as creating a project, viewing it, and then deleting it. However, Selenium might execute it into creating a project, deleting it, and then viewing the project. This makes viewing the project return an error because the project had already been deleted first.
3. Dynamic elements are not interactable, such as elements created after the user interacts with an element responsible for creating it. This causes Selenium to return an error as a result.

4. Conclusion

The main contribution of this study was developing an automated system that used Python-based Selenium source codes to transform the user story scenarios into executable test cases. User stories can be used as the basis for creating automated test cases. We have proposed a terminal-based program to successfully convert user stories with modified format into functional Python-based Selenium source codes. Implementing the source codes into several Django projects for testing indicates the program has acceptable reliability. However, the program has several issues, but they can be remedied by solving the limitation from the input requirements and defining more options for the resulting source code. Besides those, the program could be improved even further by modifying it into a GUI-based program for a better user experience.

Acknowledgments

This work was supported by Universitas Airlangga through Penelitian Unggulan Fakultas under Grant 212/UN3/2021.

References

[1]. Raharjana, I. K., Siahaan, D., & Fatichah, C. (2021). User Stories and Natural Language Processing: A Systematic Literature Review. *IEEE Access*, *9*, 53811–53826. https://doi.org/10.1109/ACCESS.2021.3070606

[2]. Nasiri, S., Rhazali, Y., Lahmer, M., & Adadi, A. (2021). From User Stories to UML Diagrams Driven by Ontological and Production Model. *International Journal of Advanced Computer Science and Applications*, *12*(6), 333–340. https://doi.org/10.14569/IJACSA.2021.0120637

[3]. Dwitam, F., & Rusli, A. (2020). User stories collection via interactive chatbot to support requirements gathering. *Telkomnika (Telecommunication Computing Electronics and Control)*, *18*(2), 890–898. https://doi.org/10.12928/TELKOMNIKA.V18I2.14866

[4]. Schön, E. M., Thomaschewski, J., & Escalona, M. J. (2017). Agile Requirements Engineering: A systematic literature review. *Computer Standards & Interfaces*, *49*, 79–91. https://doi.org/10.1016/j.csi.2016.08.011

[5]. Dalpiaz, F., & Brinkkemper, S. (2018, August). Agile requirements engineering with user stories. In *2018 IEEE 26th International Requirements Engineering Conference (RE)* (pp. 506-507). IEEE.

[6]. Vallon, R., José, B., Prikladnicki, R., & Grechenig, T. (2018). Systematic literature review on agile practices in global software development. *Information and Software Technology*, *96*, 161–180. https://doi.org/10.1016/j.infsof.2017.12.004

[7]. Hermawan, A., & Manik, L. P. (2021). The Effect of DevOps Implementation on Teamwork Quality in Software Development. *Journal of Information Systems Engineering and Business Intelligence*, *7*(1), 84. https://doi.org/10.20473/jisebi.7.1.84-90

[8]. Raharjana, I. K., Harris, F., & Justitia, A. (2020). Tool for Generating Behavior-Driven Development Test-Cases. *Journal of Information Systems Engineering and Business Intelligence*, *6*(1), 27. https://doi.org/10.20473/jisebi.6.1.27-36

[9]. Rane, P. P. (2017). *Automatic generation of test cases for agile using natural language processing* [Doctoral dissertation, Virginia Tech].

[10]. Fischbach, J., Vogelsang, A., Spies, D., Wehrle, A., Junker, M., Freudenstein, D., & Story, A. U. (2020). SPECMATE: Automated Creation of Test Cases from Acceptance Criteria. *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation,* 321–331. https://doi.org/10.1109/ICST46399.2020.00040

[11]. Fischbach, J., Femmer, H., Mendez, D., Fucci, D., & Vogelsang, A. (2020, October). What Makes Agile Test Artifacts Useful? An Activity-Based Quality Model from a Practitioners' Perspective. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-10).

[12]. Sneha, K., & Malle, G. M. (2017). Research on software testing techniques and software automation testing tools. *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, 77–81. https://doi.org/10.1109/ICECDS.2017.8389562

[13]. Rafiq, M., Ashraf, R., & Abid, H. (2020). Automated VS. Manual Testing: A Scenario Based Approach Towards Application Development. *Gyancity Journal of Electronics and Computer Science*, *5*(1), 47–55. https://doi.org/10.21058/gjecs.2020.51006

[14]. Garousi, V., Bauer, S., & Felderer, M. (2020). NLP-assisted software testing: A systematic mapping of the literature. *Information and Software Technology*, *126*, 106321. https://doi.org/10.1016/j.infsof.2020.106321

[15]. Malik, M. I., Sindhu, M. A., Khattak, A. S., Abbasi, R. A., & Saleem, K. (2021). Automating test oracles from restricted natural language agile requirements. *Expert Systems*, *38*(1), 1–22. https://doi.org/10.1111/exsy.12608

[16]. Oliinyk, B., & Oleksiuk, V. (2019, November). Automation in software testing, can we automate anything we want. In *Proceedings of the 2nd Student Workshop on Computer Science & Software Engineering (CS&SE@ SW 2019), Kryvyi Rih, Ukraine* (pp. 224-234).

[17]. Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2018). Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, *68*(1), 45-66.

[18]. Shamshiri, S., Rojas, J. M., Galeotti, J. P., Walkinshaw, N., & Fraser, G. (2018, April). How do automatically generated unit tests influence software maintenance?. In *2018 IEEE 11th international conference on software testing, verification and validation (ICST)* (pp. 250-261). IEEE.

[19]. Raharjana, I. K., Aprillya, V., Zaman, B., Justitia, A., & Fauzi, S. S. M. (2021). Enhancing software feature extraction results using sentiment analysis to aid requirements reuse. *Computers*, *10*(3), 36. https://doi.org/10.3390/computers10030036