

4DT903 Project Proposal

Automated Test Case Generation from User Stories

Samuel Berg (sb224sc)

October 2025

Contents

1	Objective	2
1.1	Domain and Problem	2
2	Models	3
2.1	Metamodels and Domains	3
2.2	Metamodel Relations	4
2.3	Tool Integration	4
3	Transformations	5
3.1	Transformation Pipeline	5
3.2	Combining Transformations	6
	References	7
A	Figures	A

1 Objective

1.1 Domain and Problem

This project addresses the challenge of bridging requirements engineering and software testing by automating the generation of test cases from natural language user stories [1].

In agile software development, user stories serve as the primary format for capturing functional requirements [2]. However, translating these stories into comprehensive test cases remains a manual, time consuming, complex and error prone process. Teams often struggle with:

- Inconsistent test coverage across different user stories.
- Manual effort required to maintain traceability between requirements and tests.
- Delayed test creation that pushes testing to later development phases.
- Difficulty ensuring all acceptance criteria are properly tested.

The project will use Model-Driven Engineering (MDE) to automatically transform structured user stories into executable test case specifications [1, 3] that can be exported to popular testing frameworks (such as JUnit for Java, pytest for Python or testing for Go), thereby improving consistency, traceability, and development velocity.

2 Models

2.1 Metamodels and Domains

The project involves three distinct domains, each requiring its own metamodel:

1. User Story Metamodel (Source Domain):

This metamodel captures the structure of user stories following the standard format [2, 4]:

- UserStory: Container element with ID, title, priority, and status.
- Actor: The role performing the action (“As a [role]”).
- Goal: The desired functionality (“I want to [action]”).
- Benefit: The business value (“So that [outcome]”).
- AcceptanceCriteria: Testable conditions using “Given When Then” format:
 - Precondition (Given): Initial state.
 - Action (When): Trigger event.
 - ExpectedResult (Then): Expected outcome.

2. Test Specification Metamodel (Intermediate Domain):

This platform-independent metamodel represents test cases abstractly [1, 3]:

- TestSuite: Groups related test cases.
- TestCase: Individual test with name, description, and priority.
- TestStep: Atomic test actions (Setup, Execute, Assert, Teardown).
- TestData: Input values and expected outputs.
- Assertion: Verification statements with comparison operators.
- Traceability: Links back to source user story elements.

3. Testing Framework Metamodel (Target Domain):

This metamodel represents the structure of popular testing frameworks:

- TestClass: Container for test methods.
- TestMethod: Individual test function with annotations/decorators.
- SetupMethod / TeardownMethod: Fixture management.
- AssertStatement: Framework-specific assertion syntax.
- TestAnnotation: Metadata (e.g. @Test, @DisplayName, @Tag).

2.2 Metamodel Relations

The metamodels are connected through the transformation pipeline [5]:

- User Story \rightarrow Test Specification: M2M transformation mapping acceptance criteria to abstract test cases.
- Test Specification \rightarrow Testing Framework: M2M transformation adapting abstract tests to framework-specific structure.
- Testing Framework \rightarrow Code: M2T transformation generating executable test code.

2.3 Tool Integration

Model Creation:

- User stories will be created using a custom Eclipse-based editor built with EMF, providing a structured form interface for entering user story components [5].
- Models will be stored in XMI format for compatibility with EMF-based tools.

Model Consumption:

- Generated test code files (.java, .py, .go) will be consumed by JUnit 5 (for Java), pytest (for Python) or testing (for Go).
- Test frameworks will execute the generated tests within standard development environments (Eclipse, IntelliJ IDEA, VS Code).

Model Updates:

- When user stories are modified, the transformation pipeline can be re-executed to regenerate affected test cases.
- A traceability model will track which test cases are derived from which user stories, enabling selective regeneration [1].

3 Transformations

3.1 Transformation Pipeline

1. M2M Transformation: User Story to Test Specification (QVTo) [6, 7]

This transformation will:

- Create one TestSuite per UserStory (or group related stories).
- Generate one TestCase per AcceptanceCriteria.
- Map “Given When Then” structure to TestStep sequences:
 - Given → Setup steps.
 - When → Execute steps.
 - Then → Assert steps.
- Extract test data from acceptance criteria descriptions using pattern matching [4].
- Establish traceability links between test elements and story elements.
- Assign test priorities based on user story priority.

2. M2M Transformation: Test Specification to Testing Framework (QVTo) [6, 7]

This transformation adapts the platform-independent test model to a specific framework:

- Map TestSuite to TestClass with appropriate naming conventions.
- Transform TestCase to TestMethod with framework annotations.
- Convert generic Assertions to framework-specific assertion methods (e.g., assertEquals, assertThat).
- Generate Setup/Tearardown methods for shared test fixtures.
- Add framework-specific metadata (tags, display names, test order).
- Handle multiple target frameworks using transformation parameters.

3. M2T Transformation: Testing Framework to Code (Acceleo) [8]

This transformation generates executable code:

- Produce Java files with JUnit 5, Python files with pytest or Go files with testing syntax.
- Generate properly formatted, readable code with comments linking to source user stories.
- Include necessary imports and class/module structure.
- Apply coding conventions (naming, indentation, documentation)

3.2 Combining Transformations

The transformations will be orchestrated using an Eclipse workflow script [5] that:

1. Loads the user story model (XMI file).
2. Executes the first M2M transformation (Story \rightarrow Test Spec)
3. Saves the intermediate test specification model
4. Executes the second M2M transformation (Test Spec \rightarrow Framework)
5. Executes the M2T transformation to generate code files
6. Outputs test files to a designated directory

Parameterization: The workflow will accept parameters for:

- Target testing framework (JUnit/pytest/testing).
- Output directory.
- Naming conventions.
- Test organization preferences.

Batch Processing: Multiple user story models can be processed in sequence, with all generated tests consolidated into a single test suite structure.

Validation: Each transformation will include validation steps to ensure model compliance with metamodels and report any inconsistencies in the source user stories (e.g. missing acceptance criteria, ambiguous conditions) [6].

This MDE approach ensures consistency, maintainability, and traceability while significantly reducing the manual effort required to create comprehensive test suites from requirements [1,3]. Here is an overview of the flow of metamodels, models and transformations (see Figure 1).

References

- [1] S. C. Allala, “Transforming user requirements to test cases using model-driven software engineering and natural language processing,” master’s thesis, Florida International University, 2023. FIU Electronic Theses and Dissertations.
- [2] T. Rahman and Y. Zhu, “Automated user story generation with test case specification using large language model,” *arXiv preprint arXiv:2404.01558*, April 2024.
- [3] J. Gutiérrez, M. Escalona, and M. Mejías, “A model-driven approach for functional test case generation,” *Journal of Systems and Software*, vol. 109, pp. 214–228, 2015.
- [4] A. from Springer chapter], “User story based automated test case generation using nlp,” in *Springer Conference Proceedings*, Springer, 2024.
- [5] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2nd ed., 2017.
- [6] A. Serebrennikova, S. Shershakov, and A. Kalenkova, “Assessing and improving quality of qvto model transformations,” *Software Quality Journal*, 2015.
- [7] Eclipse Foundation, “Qvt operational (qvto) documentation,” 2024.
- [8] Eclipse Foundation, “Acceleo user guide,” 2024.

A Figures

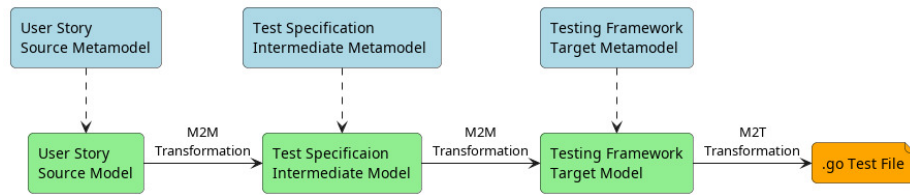


Figure 1: Overview of the Transformations to be implemented in this project