

Collaborative Software Development: from Goals to Coding

Pedro Miguel Filipe Santa Rita Monteiro

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. António Rito Silva

November 2019

Acknowledgments

First of all, I would like to express my sincere gratitude towards my supervisor Professor António Rito Silva for accepting and guiding me through this whole process, and proving his support and incredible insight, which has allowed me to complete a Thesis that I can proudly say I am happy with.

I would like to thank my parents for their love and patience, and for providing and taking care of me over all these years. I would also like to thank my grandparents for supporting my education and all their love and encouragement throughout all these years.

Last but not least, to the Pedros, Nunos and Alices that accompanied me through this journey, and certainly made it a much more pleasant one.

Without any of you, this would not have been possible. – Thank you.

Abstract

Developers work on, and switch between different tasks throughout the day. There are many tools that support developers with the creation of tasks. The plugin for Eclipse IDE, Mylyn, and its successors, go a step further and keep track of each task's context. We want to take the next step and introduce the concept of activity context.

In this thesis we developed a task-centric collaborative software development tool that supports developers by providing an activity context for their projects. The project's software architecture and development practice is used to automatically break down and connect the work as a set of tasks, grouped in an activity. The workflow provided by our tool fosters compliance with the software architecture and development practice, and in that way, increases productivity.

With our approach, during user testing we achieved 100% compliance with the software architecture, 70% compliance with the development practice, lower task execution times, and a low standard deviation in performance, supplementing the benefits of our tool.

Keywords

Social Software Engineering; Task-Centric; Task Context; Activity Context.

Resumo

Desenvolvedores de software trabalham em, e trocam entre diferentes tarefas durante o seu dia. Existem várias ferramentas que suportam os desenvolvedores na criação de tarefas. O plugin para o Eclipse IDE, Mylyn, e os seus sucessores, dão um passo à frente e mantêm o contexto de cada tarefa. Nós queremos dar o próximo passo e introduzir o conceito de contexto de atividade.

Nesta tese desenvolvemos uma ferramenta colaborativa para desenvolvimento de software com base em tarefas que suporta os desenvolvedores criando um contexto de atividade para os seus projetos. A arquitetura de software e o processo de desenvolvimento do projeto são usados para automaticamente dividir e ligar o trabalho a realizar num conjunto de tarefas, arrumados numa atividade, e desse modo aumentar a produtividade.

Com a nossa abordagem, nos testes com utilizadores obtemos 100% de conformidade com a arquitetura de software, 70% de conformidade com o processo de desenvolvimento, tempos de execução de tarefas mais baixos e um pequeno desvio padrão no desempenho, suplementando os benefícios da nossa ferramenta.

Palavras Chave

Engenharia de Software Social; Centrado nas Tarefas; Contexto de Tarefa; Contexto de Atividade.

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Overview	4
1.3	Thesis Outline	5
2	Related Work	7
2.1	Task Context	9
2.2	Task Patterns	10
2.3	Work breakdown	12
2.3.1	Dependencies	12
2.3.2	Change Propagation	14
2.3.3	Microtasks	15
2.4	Automatic Task Suggestion	16
3	Design	19
3.1	Overview	21
3.2	Solution	22
3.3	Model	23
3.4	Workflow	26
3.5	Goals and Collaboration	29
4	Implementation	31
4.1	Platform	33
4.2	Activity View	34
4.3	Design View	38
4.3.1	New Not Yet Supported Projects	38
4.3.2	Existing Not Yet Supported Projects	41
4.4	Goals and Collaboration	43
4.5	System	46
4.5.1	Task Suggester	47

5	Evaluation	49
5.1	Testing Method	51
5.2	Hypothesis	54
5.3	Results - Overall	54
5.3.1	Compliance	55
5.3.2	Performance	56
5.3.3	Unsuccessful User Tests	58
5.4	Results - Students vs Professionals	58
5.4.1	Compliance	59
5.4.2	Performance	60
5.5	Identified Problems and Solution	61
5.6	Questionnaire	63
5.6.1	Students vs Professionals	65
5.7	Conclusion	65
6	Conclusion	67
6.1	Future Work	70
6.1.1	Scope Extension - Delving into code level	71
6.1.2	Scope Shift - Information Flow, Collaboration, Gamification	71
A	Code of Project	75
B	User Test Guide	79
C	Questionnaire Results	87

List of Figures

2.1	Create/Edit Task Patterns. (1) Edit Task Pattern Name and Description (2) Edit Abstraction Services, Decisions and Problems (3) Edit Tasks Attached to Task Pattern (4) Details to selected Task (5) Hide Details to selected Task [1]	11
2.2	. Task Compare View. (1) Task Pattern Specification (2) Task Pattern Details Tree Table (3) Abstraction Services without Object (4) Object without Abstraction Service (5) Document Abstraction Service Subtask (6) Abstraction Service with Instance [1]	12
2.3	Relationships instances from repositories using Bugzilla(*) or Jira(y) [2]	13
2.4	Codes developed through open coding [2]	13
2.5	The microtasks in CrowdCode [3]	16
2.6	The CrowdCode environment and the Write Function microtask [3]	18
3.1	Tool Model	23
3.2	Task Status	26
3.3	Workflow A Example	27
4.1	Eclipse Workspace	34
4.2	Activity View	34
4.3	Plugin's Main Menu	35
4.4	Artifacts in an Activity	36
4.5	Task View	37
4.6	Artifact Type View	39
4.7	Dependency Type View	39
4.8	Task Type View	40
4.9	Rules View	41
4.10	Package View	42
4.11	Artifact View	42
4.12	Dependency View	43

4.13 Collaboration - Activity/Artifact View	44
4.14 Collaboration - GitHub View	44
4.15 Goal View	45
4.16 Plugin's Share Class and Status Class	46
5.1 Test Success Rate	55
5.2 Task Success Rate	56
5.3 Average Time per Task	56
5.4 Box Plot - Task 2 Performance	57
5.5 Task Success Rate (split)	59
5.6 Average Time Task (split)	60
5.7 Box Plot - Task 2 Performance (split)	61

Listings

A.1 Method suggestNew() in the TaskSuggester class.	75
---	----

Acronyms

API	Application Program Interface
DOI	Degree of Interest
FDA	Frequency Duration Age
GUI	Graphical User Interface
IDE	Integrated Development Environment
IST	Instituto Superior Técnico
IT	Information Technology
JSON	JavaScript Object Notation
MSc	Master of Science
NASA-TLX	NASA Task Load Index
NPC	Non-Player Character
OS	Operating System
QA	Quality Assurance
SUS	System Usability Scale
SWT	Standard Widget Toolkit
UI	User Interface
VM	Virtual Machine
IST	Instituto Superior Tecnico

1

Introduction

Contents

1.1	Motivation	3
1.2	Overview	4
1.3	Thesis Outline	5

1.1 Motivation

During the development of software, developers work on multiple tasks throughout the day. A task can refer to the creation of new artifacts and/or the modification of artifacts that have already been created. Switching from one task to another inevitably results in a change of context, which means that developers must reorganize their workspace each time they change tasks. This is a problem because in addition to the time it takes to complete a task, there is also an additional time cost to switching tasks, and developers do this switch several times a day. This situation can be helped by saving and restoring the context of each task, which comprises the artifacts involved in it. By keeping track of the artifacts involved in a task the developer can switch from one task to another without having to reorganizing the information in the workspace. A plugin for Eclipse IDE, MyLyn presents this solution, using a degree-of-interest algorithm for establishing the task context and increasing productivity.

The task switching that developers do can refer to completely different contexts, from sending emails to working on the most recent project being developed, to bug fixing an older project. With the complexity that comes with software development, the cost of task switching includes not only the reorganization of the information in the workspace but also the developers mindspace and their focus on the new task at hand. As developers work more productively when iterating on the same project instead of switching back and forth, a day's work would ideally be kept to the same project and the task switches would be accompanied by small context changes.

However, MyLyn's Task Context deals with task switching in the scope of the workspace and strictly focuses on reorganizing the present information according to the current task. There is:

- No organization of the tasks, which can be even of different projects.
- No structure to the tasks themselves, other than the context data.
- No relationship between these tasks, other than the subtask relationship.

Presenting the tasks with no relation of context similarity and random differences in context leaves not only the task switching cost open to reduction, but even the task execution time itself.

Previous articles on Task Context by several authors [1, 4–9] leading up to this years article [10] on the direct relationship between the flow of information and productivity, have inspired us to further expand upon this work.

1.2 Overview

We believe that providing developers with structured activities that are made of interrelated tasks would increase productivity. Incentivizing work on a sequential context would make it easier to choose what to do next and the process could be automated by providing a structure to the activity.

An activity can be seen as an instance of a project's workflow following a practice, such as the creation of a single story to the deployment of its respective code in a Bottom-Up practice. In order to provide a structure to the activity we must look at the data itself, the artifacts in the workspace and how they relate to each other.

As we investigate the artifacts that are used and produced during the execution of tasks, we can see that some of them serve the same purpose and can be abstracted into a type of artifact. An artifact type is for example, a Story, a Domain Class or a Domain Test. We can also notice that these types of artifacts are also connected with each other. The software architecture of the project is then directly related to the execution of an activity and a necessity to provide its structure.

Another aspect that is necessary for the structure of an activity is the development practice. This means that while the artifacts produced to achieve a development goal are the same independently of the practice followed, their sequence of creation and manipulation can change drastically, depending on the followed development practice.

Joining together the software architecture of the project and the development practice in use it is then possible to establish rules that can provide activities with a structure by looking at the current state of the artifacts in the workspace.

This leads us to the main research question:

- In the frame of a software architecture, can the association of working context with a development activity foster the compliance with predefined development practices, and improve the developer's productivity?

To answer this question we defined the development context associated with an activity as the set of tasks performed to accomplish the activity, as well as the set of artifacts created/modified during the execution of tasks.

In this context we've defined the compliance with development practices as the particular sequences by which tasks are performed in the context of an activity.

In this same context we've defined productivity as the number of steps required to (1) implement a new activity, (2) perform a software change in the context of an activity.

In this thesis we developed a task-centric software development tool that supports developers by providing a structured view of the working context, making it easier to choose and start working on a task as well as incentivizing prolonged work in the context of an activity.

The tool allows the developers to describe both the development practice and the project's software architecture as a set of rules. The architecture is represented as packages and types of artifacts, and the practice as types of dependencies between the artifacts.

The activity's context is then the set of tasks performed to accomplish the activity, as well as the set of artifacts created/modified during the execution of its tasks. A project can have activities representing different practices.

Based on the rules describing the architecture and practice, as well as the artifacts that have to be present in order to accomplish a new task and the connection between these artifacts, tasks representing the work to be done are generated and suggested to the developers. Upon activation the tasks automatically create their own context.

By allowing the developers to create the rules themselves, the tool becomes very flexible.

1.3 Thesis Outline

The structure of this thesis is as follows:

- Related Work (chapter 2) - Study of task context related systems and other task-management approaches
- Design (chapter 3) - Introduction of the base concepts, model, workflow and reasoning.
- Implementation (chapter 4) - Description of the implemented tool's system and features.
- Evaluation (chapter 5) - Description of the testing method and result analysis.
- Conclusion (chapter 6) - Review of our project, its contributions and future.

2

Related Work

Contents

2.1 Task Context	9
2.2 Task Patterns	10
2.3 Work breakdown	12
2.4 Automatic Task Suggestion	16

In Chapter 2 we analyse several papers that provide us insight into how to design our tool. We begin by studying the two systems that inspired us, MyLyn in the section **Task Context**, and another task-centric tool that iterates over task context in **Task Patterns**.

Then we move on to tasks themselves, what they are, how they relate to each other, and how they could be recommended. In **Work Breakdown** we first go over how tasks relate to each other in the subsection **Dependencies** and then how change is propagated through tasks in **Change Propagation**. In this section we also look into crowdsourcing and **Microtasks** in a subsection with that same name. Last but not least, go over several ways to recommend tasks in **Automatic Task Suggestion**.

At the end of each section/subsection we compare the respective approach with our approach.

2.1 Task Context

In order to support the programmer's tasks, which are proven as an effective and efficient way to organize the programming activities, Integrated development environments (IDE), such as Eclipse, present a view of the system as an aggregation of artifacts.

Programmers typically work on multiple tasks each day, and each of them requires a different setup of the workspace, which is aggravated by the fact that most modifications to a system affect multiple modules and each of these modules might require understanding from other modules.

In Eclipse close to all the functionality is provided by plugins, on top of a small run-time kernel. A task-centric software development tool, MyLyn for Eclipse IDE, uses Task Context in order to improve productivity [5, 7, 8].

Task Context uses a degree-of-interest (DOI) algorithm to show only the work context relevant to the task, representing the program's produced and accessed elements and their respective relationships relevant to completing this particular task.

Each interaction a user has with the systems artifacts are stored as events (selection, edit, command, propagation and prediction). This interaction history is used to calculate the DOI of each element, resulting in a graph that represents the task's context. This calculation considers the frequency, recency and weight of the interaction to provide an accurate current interest on each element.

Every target element of an event is represented in the graph as a node, for example, if a programmer navigates from a method call to its declaration, these two selection events will result in an edge representing the Java reference relation between the two corresponding element nodes.

The task context is built as the programmer works and can be recreated from stored interaction history. It represents the program elements and relationships relevant to completing a particular task. It is important that multiple artifact types are supported since there is a mixed use of documents and web-pages. Task Context is layered over Eclipse as the extension MyLyn, providing a plethora of DOI-based ele-

ments, such as task list (shared from repository, email or local), a view/edit pane for the task resources (including web browser and MS Office documents) and a filterable tree view with all available elements. Activating a task automatically focuses the tree view and closes all non-relevant documents except those in external windows.

MyLyn's task context deals with task switching in the scope of the workspace and strictly focuses on re-organizing the present information according to the current task. However there is no organization of the tasks which can be even of different projects. There is no structure to the tasks themselves other than the context data. There is also no relationship between these tasks other than the subtask relationship. Presenting the tasks with no relation of context similarity and random differences in context leaves not only the task switching cost open to reduction, but even the task execution itself. While Mylyn focuses on highlighting the present relevant artifacts for the current task, we provide a pattern for the current activity including the creation and focus on all the artifacts involved in the current task and facilitating the flow into the next task.

2.2 Task Patterns

Social software engineering concerns the development of tools to improve the collaborative development of software.

Task Patterns presented in [4] [1], extend the scope of task-centric software development tools, such as MyLyn for Eclipse IDE (Tasktop Pro extension) and the NEPOMUK social semantic desktop (KASIMIR sidebar for personal task handling), by collecting the data from a user's task and organizing it in a Task Pattern.

While Task Context focuses on an individual's productivity, Task Patterns additionally focus on collaboration.

The artifacts resulting from a user's task, such as code, e-mails, documents and web-resources are collected by Abstraction Services and make up a task pattern that can be shared to a repository and obtained by other users, resulting in an enhancement life-cycle. They support both individual task execution by providing knowledge on how to execute a task based on the code and artifacts previously used, and experience sharing as the task pattern is enriched by the knowledge of the individual himself, making it a re-usable knowledge structure.

A pattern can be enhanced not only by contribution of a user (explicit enhancement), also by simply instantiating a task pattern within a task context, which causes new information to be attached to the pattern by rules.

Patterns have different ways to transfer information. Abstraction Services *"allow a guided decomposition of a task"*. After selecting a pattern, the Information Abstraction Services help identify resources,

collaborators and subtasks to be used in the task. Potential artifacts can be described with different abstraction levels: recommendation, example and abstract description retrieval. Similar tasks can go in different directions, *"Decisions as filter"* allows showing only the relevant artifacts by filtering the abstraction services. Problem awareness and solution is also supported by Abstraction Services. They are given as short textual descriptions and include the suitable artifacts. KASIMIR supports task patterns with a create/edit view (Figure 2.1) and a compare view (Figure 2.2). A resulting shortcoming is a complex user interface.

While task patterns guide by reuse, we guide by definition. The structuring of the task is driven by explicit rules which are based on the architectural structure. This way we support individual task execution with suggested tasks. Collaboration is also promoted as the developers follow the same architecture, and therefore the same tasks, whose status is shared to GitHub.

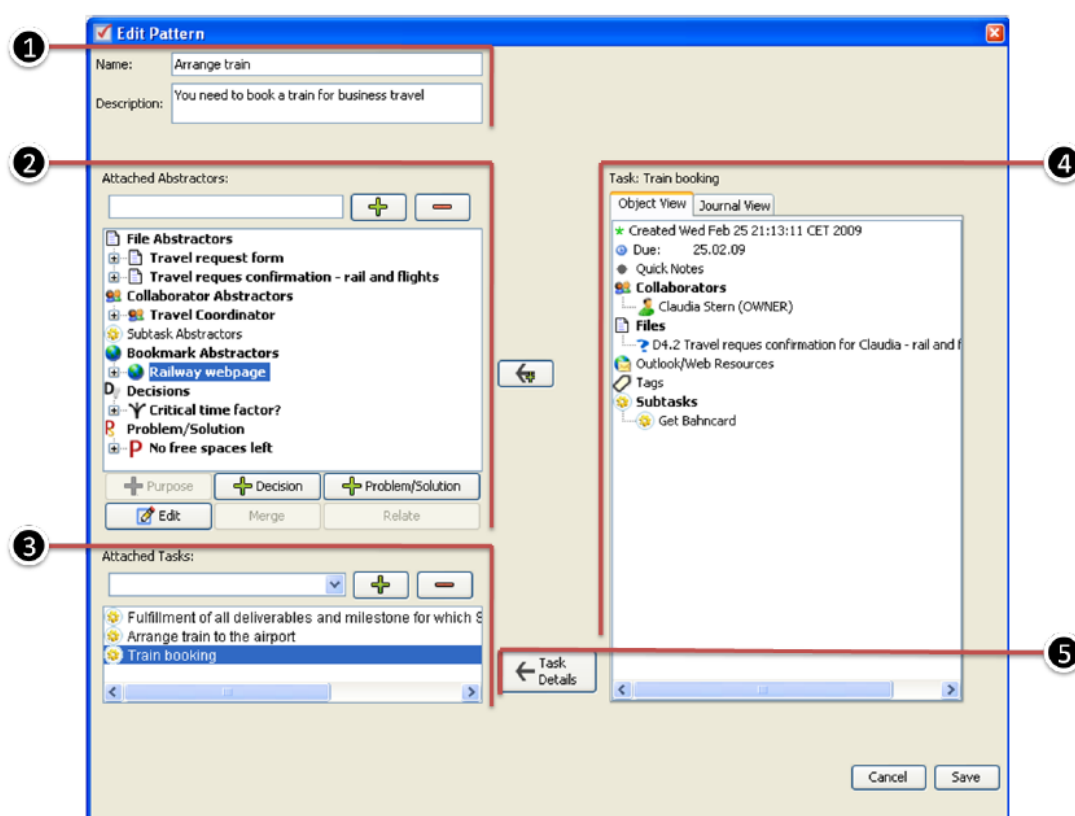


Figure 2.1: Create/Edit Task Patterns. (1) Edit Task Pattern Name and Description (2) Edit Abstraction Services, Decisions and Problems (3) Edit Tasks Attached to Task Pattern (4) Details to selected Task (5) Hide Details to selected Task [1]

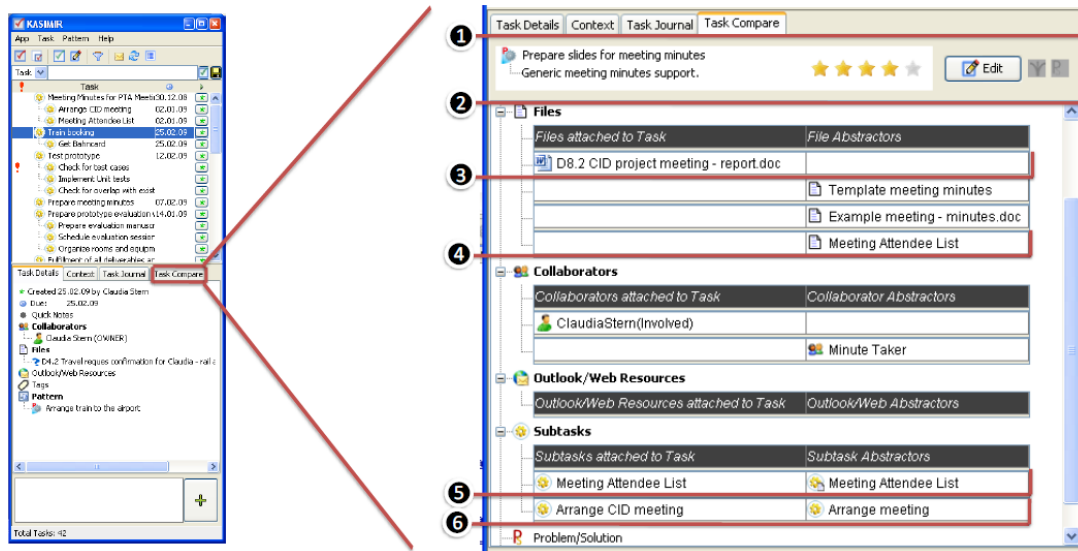


Figure 2.2: . Task Compare View. (1) Task Pattern Specification (2) Task Pattern Details Tree Table (3) Abstraction Services without Object (4) Object without Abstraction Service (5) Document Abstraction Service Subtask (6) Abstraction Service with Instance [1]

2.3 Work breakdown

2.3.1 Dependencies

A software developer's day-to-day involves breaking down and working on several tasks, which come from the analysis of problems and user stories. The overall development efficiency and quality is affected by the breakdown of tasks into *"well defined fine-grained sub-tasks"* which help working uninterruptedly on a single task.

Software developers often use issue repositories to define the workload of the project. The relationship between these activities, bugs and tests is not only mostly manually set, it is also inconsistent across repositories, as seen in the different names used in each of the repositories in Figure 2.3. The most frequently used relationships describe work breakdowns. The authors of [2], coded how these relationships are used. The analysis of titles of issues selected from the repositories above resulted in six codes describing the dependencies as a work breakdown relationship in Figure 2.4.

Specification refinement *"applies when a child issue describes one step of the work breakdown for the parent issue"*, example Parent: Improve issue editor, Child: Show reporter on new issue tooltips. The three following codes are cases of specification refinement. Instance of parent *"applies when each child issue is a particular case in which the work described by the parent issue should be performed"*, example Parent: Create new VMs, Child: four Windows machines. Check validity *"applies when a child issue describes a verification activity for a parent issue"*, example Parent: Improve issue editor, Child: Add unit test. Expectation *"applies when the child issue describes constraints or suggestions on how a parent*

issue can be fulfilled", example Parent: Improve issue editor, Child: Depends on field in issue editor should fill available horizontal space. Problem *"applies when the child issue describes a problem that occurs in a parent issue"*, example Parent: Transaction Logging, Child: non-unique messageid causes transaction not to be logged in transaction repo. Reverse specification is the reverse of specification refinement, example Parent: Show reporter on new issue tooltips, Child: Improve issue editor. Unknown applies when either none or more than one of the other codes apply, example Parent: Create static screens for Direct configuration in Administrative GUI, Child: Trust Bundles.

The understanding and recognition of use relationships provides the ability to improve and create new tools. As tasks are broken down into sub-tasks, in our tool, activities are broken down into tasks, which are structured with artifacts. The relationships between these artifacts is essential for automatically suggesting tasks. The relationships between artifacts allow adaptation to different development processes. Our approach also has the benefit of relationships being automatically set, avoiding user errors.

	Mylyn *	Connect †	HBase †
Blocked			299
Breaks			47
Clone			32
Contains			12
Depends-on	2174	205	286
Duplicates	1520	43	159
Incorporates			208
Part-of-epic		559	
Requires			189
Relates-to			1901
Subtask		1643	1368
Supercedes			31
Supported-by		1361	
Total	3694	3811	4532

Figure 2.3: Relationships instances from repositories using Bugzilla(*) or Jira(y) [2]

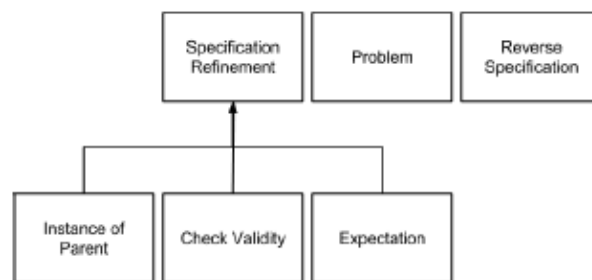


Figure 2.4: Codes developed through open coding [2]

2.3.2 Change Propagation

Change propagation is the practice of developers accessing artifacts in dependent tasks while working on their own.

Paper [11] found that 64% of linked task pairs exhibit change propagation (true positives). The remaining 36% (false negatives) can be explained by the seven identified situations. The following are task dependencies that don't propagate change. Task decomposition, one parent task with multiple child tasks refining the work described in the previous, which consequently involves little coding effort. Separation of concerns, tasks pairs that address different aspects of the same concept resulting in few shared artifacts. Synchronization, coordination of work on artifacts shared by task pairs so as to not result in a merge conflict. The following are artifact centric task dependencies. Work Continuation, continuing development of artifacts on one task in another task. Artifact Reuse, similar to the previous but focused on using the output of the linked task instead of continuing work. Emerging Task Decomposition, similar to Task Decomposition but the central task has significant work done and then the child tasks are created as needed. Small Bug Fixes, corrections to artifacts in both tasks.

It was found that 93% of all non-linked task pairs do not exhibit change propagation (true negatives). Six motifs that allow the classification of the remaining 7% false positives as either true positives, which are pairs that should have been linked, or true negatives, which represent irrelevant change propagation, were also found. Task Cluster Membership, task pairs that address the same implementation topic and typically have links to tasks that are directly linked. Support Cluster Membership, similar to the previous, tasks implement/test a particular feature. Mutual Access of Utility Artifacts, task pairs that mainly read/update common utility artifacts. Orthogonal Concerns, task pairs with overlapping topics end up accessing the same artifacts. Core Artifact Access, task pairs unrelated to each other that access small parts of a couple common artifacts at the core of the system. Grand Tasks, tasks with many artifacts and unrelated tasks.

Developers *"use links to manage control flow, data flow, task composition, and simultaneity dependencies"*, which is relevant for the design of support tools. Change propagation metrics allow us to reason about the implicit dependencies of task pairs, reclassify task pairs links a-posteriori (facilitating maintenance efforts), and identify the situations where coordination support, such as developer change notification, is needed.

Our tool addresses the complexity and possibility of conflict resulting from linked activities by explicitly defining the relation between the activities tasks. The tasks inside an activity can be said to be linked by Artifact Reuse as a task's resulting artifacts are often another task's used artifacts. The structure of a task is customizable, so any undesirable change propagation can be easily fixed. To make sure that there is no undesired change propagation between activities, tasks from different activities are not linked.

2.3.3 Microtasks

Open source software development allows everyone to contribute, but there are several barriers to this contribution, such as learning about the project code structure and tools and figuring out a way to contribute, and the fact that tasks can be large and have a granularity of hours or days. These aspects discourage contribution and reduce the pool of participants.

Microtask crowdsourcing decomposes work into short self-contained microtasks. While overhead is introduced, the pool of participants is broadened since, not only the barrier to contribution is lower, the work can be done at a finer scale with more parallelization.

The authors of [3] have implemented their approach in CrowdCode, a cloud IDE for crowd development, limited to crowdsourcing small functional libraries.

Work begins with a client request specifying the API of a library to be implemented by the crowd. The API is described through a set of data types, functions and their signatures and descriptions. This represents a significant burden on the requesting developer because the descriptions of functions and all data structures must be clearly specified.

The project has a global queue of the artifact's microtasks. Each artifact can have multiple microtasks associated to it, but only one in the global queue so that there is at most one person working on an artifact to reduce conflicts.

When logging in users get a microtask from the queue and can choose to skip it or do it. If they accept it means that they are the only people working on that artifact. The microtasks themselves work mostly at a function level (Figure 2.5), they can be writing pseudocode for a function, searching for an existing similar function, writing the description of a function, replacing pseudocode with actual code, listing test cases for a function, implementing a test case, executing all tests, and fixing a bug. As only a single person is working on an artifact it is recommended to submit the changes at most after eight minutes even if the task is not complete. This way, iterative workflows are supported and microtasks for the same work are generated until the work is complete.

Completing/submitting a microtask causes the corresponding artifact to update its state (ex: Write Function Description microtask causes function to change state from "non described" to "described"), generate more microtasks for the same work (ex: after writing the function description a microtask for writing the pseudocode is generated), and send events to other artifacts that depend on it (ex: adding a parameter to a function's signature sends an event to all artifacts that call it, functions and tests, so that the microtasks needed are generated).

Completing microtasks award points increased by the times they have been skipped. There is also a leaderboard with users and their points, and a global chat. With help of a small user study composed of 12 university students, it was concluded that gamification has a surprising motivational power.

At the core CrowdCode is a *"system for tracking work as a graph of artifacts, dynamically generat-*

ing microtasks in response to state changes in artifacts and propagating events across dependencies”. CrowdCode has several limitations as it does not support design tasks, such as new data types, requires the correctness of work to be evaluated solely through tests, and contributions are decontextualized and may need correction.

Our tool supports every kind of task, data type and evaluation. It does so while providing a structure to each activity to ease contribution. However, we do not go into class level code generation and do not focus on the definition of tasks for newbies.

Microtask type	Description	Possible subsequent microtasks
<i>Write Function</i>	Sketch or implement a function using code, pseudocode, and pseudocalls.	<i>Write Function, Reuse Search, Machine Unit Test</i>
<i>Reuse Search</i>	Given a pseudocall and the surrounding code, determine if an existing function provides the functionality or that no function does.	<i>Write Call, Write Function Description</i>
<i>Write Function Description</i>	Given a pseudocall and the surrounding code, write a description and signature for a new function for this behavior.	<i>Write Call</i>
<i>Write Call</i>	Replace the specified pseudocall with a call to the specified function or edit the function to implement the behavior in an alternative way.	<i>Write Function, Reuse Search, Machine Unit Test</i>
<i>Write Test Cases</i>	Given a description of a function, list test cases.	<i>Write Test</i>
<i>Write Test</i>	Given a test case and the description of a function, implement the test case or report an issue in the test case.	<i>Machine Unit Test, Write Test Cases</i>
<i>Machine Unit Test</i>	Executes all implemented tests, notifying functions if they fail a test	<i>Debug</i>
<i>Debug</i>	Edit code to fix bug, report an issue in a test, or create stubs describing issues in function call	<i>Machine Unit Test, Write Function, Reuse Search</i>

Figure 2.5: The microtasks in CrowdCode [3]

2.4 Automatic Task Suggestion

Automatically identifying task relationships are useful to developers in various scenarios. The paper [9] investigates how context data, extracted from developer interactions with development artifacts, can be used to predict relationships between tasks. They evaluate several context similarity models. The Jaccard model, summarized to the summed relevance of artifacts. The DOI Jaccard as a degree-of-interest [5] Jaccard model. The FDA Jaccard as a Frequency-Duration-Age relevance model proposed by the authors.

The context similarity models are evaluated on two types of task relationships useful for work coordination, "dependsOn" and "blocks", and two types useful for personal work management "isNextTo" and "isSimilarTo".

It was concluded to be possible to predict task relationships with these context similarity models, which all displayed similar accuracy, superior to both random prediction and mining of textual task descriptions. The key factor for the enhanced prediction is the task relationships, the more related a task, better the

prediction is.

Task recommenders based on context similarity can be useful since on a single work day and even when focused on a single component, developers switch between tasks with different contexts. However it is also noted that the accuracy of prediction is still *"far from perfect"* when applied in practice.

From a students survey, dimensions that might make task contexts more accurate were identified such as the technology, architecture and tools used.

They also think that a task recommender will need to consider other information such as the text description, discussions and annotations tasks, as well properties such as the task severity, complexity or simply the level of concentration and remaining work time (Allen, 2001).

A major limitation to context similarity is that the context must be available for the comparison at the prediction time, which means that new tasks cannot be handled, they need to have been started for the relationships to be predicted.

Previous research has shown that *"about half of development work is rather informal and not documented as tasks with textual descriptions"*, making text similarity prediction not possible. Typically, on development projects some work is defined as tasks with descriptions, other is only informal. In practice, they think that a hybrid approach of context and content-based prediction of task relationships might be the most appropriate.

Paper [6] presents the thesis that *"the efficiency and quality of software development can be improved through the automatic generation and recommendation of task breakdowns to developers based on patterns learned from development tasks specified in projects"*.

By analysing the natural language of tasks in open source issue repositories and extracting stats such as distinct and non-distinct verb-noun instances (ex: task contains replace metro) and rules such as verb-noun rules (ex: if parent task contains *"replace metro"* then child task contains *"use CXF"*), patterns can be learned and categorized. Given that patterns can be learned, the analysis of historical issue data and the use of task context to generate task breakdowns, putting together verb-noun instances that depend on the task category and fit in the project lexicon, should be possible and result in a tool to recommend task breakdowns.

Our approach of using the architecture of the project as well as the development practice to suggest tasks based on the artifacts and their relationships is very versatile. Not only can it deal with new tasks, it also supports run-time changes to the architecture. The cost is the one time description of the architecture.

CrowdCode

10 lines of code

0 functions written

1 microtasks completed

Alice

score

Your score ★

10 points

leaderboard

Leaders

10 Alice

Ask the Crowd

group chat

project statistics

current user

microtask

Edit a function

10 pts

Can you implement the function below?

instructions

If you're not sure how to do something, you can indicate a line or portion of a line as **pseudocode** by beginning it with `///#`. If you'd like to call a function, describe what you'd like it to do with a pseudocall - a line or portion of a line beginning with `///!`. Update the description and header to reflect the function's actual behavior - the crowd will refactor callers and tests to match the new behavior. (Except if you are editing a function that was specified and directly requested by the client - denoted by a function that starts with CR - in which case you can't change this function's name or parameters, but you can change its description).

Note that all function calls are pass by value (i.e., if you pass an object to a function and the function changes the object you will not see the change).

IMPORTANT: If you think the function may require more than a few minutes to write, please use pseudocode and pseudocalls to break up the function into smaller pieces that others can work on. If you've gotten two or more reminders to submit, **YOU SHOULD SUBMIT NOW!**

data structures

Types

Type names may be String, Boolean, Number, any type below (bold text), and arrays of any type (e.g., String[], Number[]).

Board

properties- "rows": String[]

Boards are an array of 8 character strings, where each row is a string and each character represents an element of the board. Elements must be either "-" (unoccupiable space), "o" (empty space that can be occupied), "r" (normal red), "R" (red King), "b" (normal black), and "B" (black King). Black players start at the top and move downwards; red players start at the bottom and move upwards. Kings can move upwards and downwards.

Example:

{ "rows": ["-b-b-b-b-",
"b-b-b-b-b-",
"o-o-o-o-o-o",
"o-o-o-o-o-o",
"o-o-o-o-o-o",
"o-o-o-o-o-o",
"o-o-o-o-o-o",
"o-o-o-o-o-o"] }

code editor

function description

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

Client REQUEST

Given a board and a list of moves (that have already been checked for validity), executes the moves. Moves can be either an array containing a single move or (iff multiple jumps are taken) an array of valid jump moves for a single piece.

See <http://simple.wikipedia.org/wiki/Checkers> for background on English draughts rules. Note that the rules used should be for the American variant of checkers called "English draughts" (e.g., a player who has the opportunity to jump may instead choose a different move).

@param Board board - the initial board prior to the move

@param Move[] moves - the move(s) to execute

@RETURN Board - NEW BOARD

*/

function CRdoMoves(board, moves)

{

var newBoard = //! copy existing board

for (var i = 0; i < moves.length; i++)

{

if (//! move is a jump

{

//! remove piece from the board

}

//! create new board with piece moved

//! check if move created a King

//! Do we need to do something with checking for victory?

}

return newBoard;

}

Recent Activity

You earned 10 points for writing test cases!

activity feed

Give us feedback on CrowdCode! What do you like? What don't you like?

Send feedback

Submit

Skip

Help, I don't know Javascript!

Figure 2.6: The CrowdCode environment and the Write Function microtask [3]

18

3

Design

Contents

3.1 Overview	21
3.2 Solution	22
3.3 Model	23
3.4 Workflow	26
3.5 Goals and Collaboration	29

In Chapter 3 we begin with our research question and its influence on the design and evaluation of our tool in **Overview**. In **Solution** we go over the main problems we identified by studying related system in Chapter 2 and how we built our tool to solve them. In **Model** we define the base concepts and elements that make up our system. In **Workflow** we describe and exemplify the several workflows that our tool allows. And at last in **Goals and Collaboration** we talk about our approach to sharing information between the project's members.

3.1 Overview

We believe that providing developers with structured activities that are made of interrelated tasks would increase productivity. Incentivizing work on a sequential context would make it easier to choose what to do next and the process could be automated by providing a structure to the activity.

An activity can be seen as an instance of a project's workflow following a practice, such as the creation of a single story to the deployment of its respective code in a Bottom-Up practice. In order to provide a structure to the activity we must look at the data itself, the artifacts in the workspace and how they relate to each other.

As we investigate the artifacts that are used and produced during the execution of tasks, we can see that some of them serve the same purpose and can be abstracted into a type of artifact. An artifact type is for example, a Story, a Domain Class or a Domain Test. We can also notice that these types of artifacts are also connected with each other. The software architecture of the project is then directly related to the execution of an activity and a necessity to provide its structure. In our concept the architecture is represented by the artifacts. Another aspect that is necessary for the structure of an activity is the development practice. This means that while the artifacts produced to achieve a development goal are the same independently of the practice followed, their sequence of creation and manipulation can change drastically, depending on the followed development practice. Each development practice has a set structure.

Joining together the software architecture of the project and the development practice in use it is then possible to establish rules that can provide activities with a structure by looking at the current state of the artifacts in the workspace.

This leads us to the main research question:

- In the frame of a software architecture, can the association of working context with a development activity foster the compliance with predefined development practices, and improve the developer's productivity?

To answer this question we defined the development context associated with an activity as the set of tasks performed to accomplish the activity, as well as the set of artifacts created/modified during the

execution of tasks.

In this context we've defined the compliance with development practices as the particular sequences by which tasks are performed in the context of an activity.

In this same context we've defined productivity as the execution time and success of the steps required to (1) implement a new activity, (2) perform a software change in the context of an activity.

In this thesis we developed a task-centric software development tool that supports developers by providing a structured view of the working context, making it easier to choose and start working on a task as well as incentivizing prolonged work in the context of an activity.

The tool allows the developers to describe both the development practice and the project's software architecture as a set of rules. The architecture is represented as packages and types of artifacts, and the practice as types of dependencies between the artifacts.

The activity's context is then the set of tasks performed to accomplish the activity, as well as the set of artifacts created/modified during the execution of its tasks. A project can have activities representing different practices.

Based on the rules describing the architecture and practice, as well as the artifacts that have to be present in order to accomplish a new task and the connection between these artifacts, tasks representing the work to be done are generated and suggested to the developers. Upon activation the tasks automatically create their own context. By allowing the developers to create the rules themselves, the tool becomes very flexible.

3.2 Solution

We propose a new approach that allows the definition of types of tasks such that it handles problem A, it distinguishes between task and activity, where an activity is a set of tasks, to address problem B, it is based on the dependencies of the artifacts, to handle problem C, it associates the task structure to the system architecture, to solve problem D:

- A - Problem: No support for new tasks.
- A - Solution: Our tool is able to support new tasks by giving a structure to types of tasks, which we call task types.
- B - Problem: No support for the developer to stay focused on tasks with similar context.
- B - Solution: We provide a pattern for the current activity including the creation and focus on all the artifacts involved in the current task and facilitating the flow into the next task.
- C - Problem: No automation, unable to suggest tasks and relying on the developers to do everything, leading to errors.

- C - Solution: In our tool, activities are broken down into tasks, which are structured with artifacts. The relationships between these artifacts allows for automatically suggesting tasks. With our approach, the whole activity and its tasks is managed automatically, avoiding user errors.
- D - Problem: Not data-centered, and so, unable to foster compliance with the architecture and development practice.
- D - Solution: Our tool uses a data-centric workflow inside the activity where task start and task completion are based on, respectively, the available artifacts and the produced artifacts, and our rules representing the architecture and development practice.

An approach of using the architecture of the project as well as the development practice to suggest tasks based on the artifacts and their relationships is very versatile. Not only can it deal with new tasks, it also supports run-time changes to the architecture. The cost of task suggestion is a one-time description of the architecture with our tool's rules.

The main difference from traditional task-centric software tools is a data-centric workflow where task start and task completion are based on, respectively, the available artifacts and the produced artifacts. On the other hand, whenever a produced artifact is changed, all the tasks that use it as precondition may be reactivated.

3.3 Model

In order to define our tool, we first defined a model. The main concepts that make up our model can be seen in Figure 3.1.

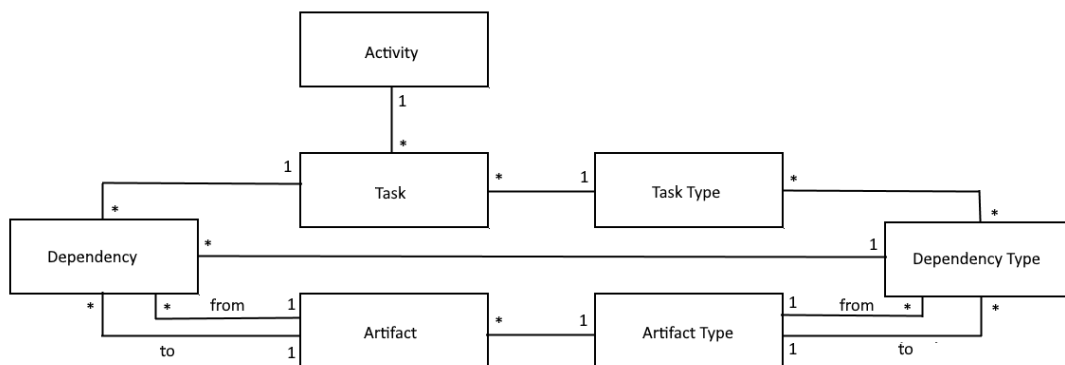


Figure 3.1: Tool Model

These concepts can be used to define different development practices. A development practice is defined by a group of Task Types, Dependency Types and Artifact Types, such that the dependency types constraint the creation and change of artifacts according to their types and the relation between

them. For instance, a dependency type defines a causal relation in the creation of the artifacts of the given type. A task type aggregates several of these causal relationships to be followed in a single task by a developer.

A development project comprises several activities and each activity follows a single development practice, that can differ from those of other activities. This way, even keeping the same artifacts, by having different dependencies and task types, developers can easily switch between development practices, such as bottom-up and top-down.

An activity is defined by a set of tasks that, together, allow the achievement of a development goal. This can be seen as an instance of a project's workflow following a practice, such as the creation of a single story to the deployment of its respective code in a Bottom-Up practice.

The activities context is then the set of interrelated tasks performed to accomplish the activity, as well as the set of artifacts created/modified during the execution of its tasks.

Activities have the benefit of providing a structured sequential context that is data-centered, while increasing performance with some mechanisms of automatic support, from task suggestion to artifact creation, and by maintaining the sequential context of changes that occur during the activity implementation.

On the other hand, the activities context is mostly self-contained, meaning that tasks in another activity won't be modified by changes to the artifacts in the current activity if the produced artifacts are shared. In case the modified artifacts have tasks in other activities that produce artifacts not present in the current activity these are set to be redone.

A task is essentially a transformation of artifacts. We can structure the data of a task as the artifacts that are needed for the transformation to take place, the artifacts that result from the transformation, and the relationships between the artifacts that were used and the artifacts that were produced. The existence of the initial artifacts and the relationships they have with the final artifacts are therefore the precondition for the creation of a task. Of course, during task execution, other artifacts can be associated with it.

We abstract tasks into task types. Task types can be seen as templates of dependency types. Just as we previously saw the importance of the relationships between tasks, this relevance is extended to the relationships between the elements inside the tasks, the artifacts. Establishing dependency types between our artifact types is therefore essential for defining the workflow.

Just like artifact types, dependency types are also custom. A type of artifact can then have as many types of dependencies as needed with any other type of artifact including itself. The artifacts of type "Class" can have a dependency with those of type "Test". The meaning of this dependency is that a "Class" results in a "Test". A dependency type is then a relationship between two artifact types, where the first is needed for the second.

Task suggestion is then based on the current artifacts available whose dependencies are not yet realized

in a task. This means that if all the used artifact types in a task type are present in the workspace as artifacts of the matching types that have been produced during previous tasks, and are therefore in a "Complete" status, and there is no current task of that type, a task is created realizing the dependencies stated in the task type. If the produced artifact does not exist yet, it is created.

Task reactivation is also dependency based. If an artifact with a "Complete" status is modified, then the tasks that contain dependencies where it results in another artifact are reactivated. This creates a chain reactivation. As a reactivated task is complete, tasks that contain dependencies with the resulting artifact as the used artifact are reactivated, and so on. This reactivation chain is disabled by default as there is no certainty that changes are required, so the user should decide if the reactivation chain should be activated or not.

As the base elements of the model we have the artifacts, the base elements of a system, representing the data. The artifacts are the files in the workspace, and they can be abstracted into several types, for example: Story, Class and Test. This means that when using our tool each artifact is assigned a type. As packages/folders are a group of artifacts, they can also be attributed a type, usually the same as the contained artifacts. Assigning a type to packages gives knowledge on where artifacts of a certain type should be, which is important as our tool automatically creates artifacts.

Given dependencies and task types, the system generates new tasks for artifacts whenever their dependencies are enabled.

An artifact's life cycle begins when it is identified by the system. At this time its status is set to "Incomplete". If it has dependencies enabled with other artifacts and there is a task type matching these relationships, a task is created. Upon task completion both the task and the artifact change their state. The artifact's state is set to "Complete". It is possible that the task can still be revisited. If an artifact is changed after a task where it was used is completed, the task will be reactivated as it is possible that the dependent artifacts need changes.

The main statuses a task can have are "Available", "Active", "Complete", "To Redo", "In Progress" and can be seen in Figure 3.2.

The status "Available" means that the task was suggested, and it has not been activated yet. The status "Active" means that the task is the one being currently worked on. The status "Complete" means that the task was active and then completed. The status "To Redo" means that the task was on "Complete" status but an artifact belonging to its used artifacts was modified, triggering the task's status change into "To Redo". The status "In Progress" means that the task was completed but is still not finished because it needs other tasks to be completed first.

Special rules can also be attributed to artifact types. The purpose of these rules is for particular use cases that cannot be covered by artifact types, dependency types or task types. For example, we might not want to have a task where the artifact of type "Class" is created to be completed, until the artifact of

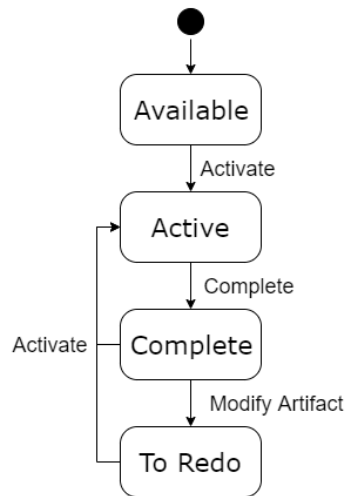


Figure 3.2: Task Status

type "SystemTest" is complete. So, in this case the rule would force the respective tasks to remain in progress while allowing the activity to continue. In Chapter 4 we explain the implementation and give a concrete example.

3.4 Workflow

There are three different ways to use our tool:

- Option A - The developer works on an already supported project.
- Option B - The developer creates his own rules for a not yet supported new project
- Option C - The developer creates his own rules for an already existing not yet supported project

A supported project is a project where we already created the rules that state the software architecture and the development practice.

There are also two different workflows when working on an activity. Either we are implementing a new feature by completing an activity, or we are make modifications by altering the artifacts in an already existing activity. We will now provide an example of a workflow for an already supported project (A) where we are completing a new activity, following Figure 3.3.

A user starts a new activity (Activity 1) on his project. There is a task type (Task Type A) that does not require any present artifacts to be instanced, so the task suggestion system presents a new task to create a Story. To note that, although we have referred to a Story as the first artifact type in our examples, the architecture is custom, it can take any desired form within the rules. The developer starts the task and the system creates an artifact of type Story ("StoryA.txt") automatically and opens it. The user

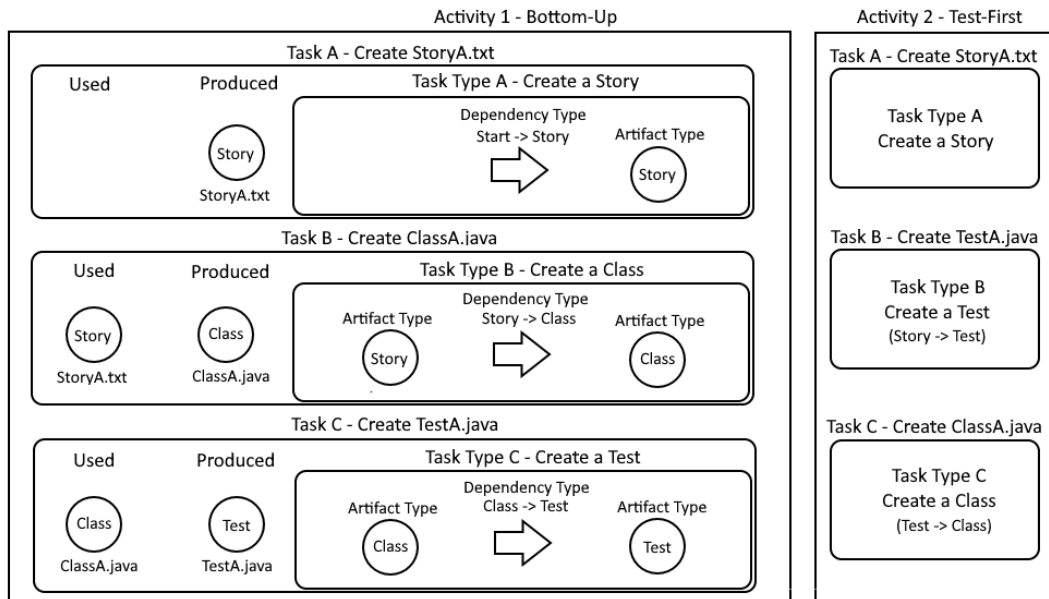


Figure 3.3: Workflow A Example

writes the story and upon task completion the task suggerter identifies a task type (Task Type B) for the now available artifact of type Story and presents a new task (Task B). The user starts the task and the artifact of type Class ("ClassA.java") is created and the context is opened, meaning both the Story and Class artifacts are now in the editor. The user writes the Class artifact and completes the task. A task (Task C) for creating a Test artifact is suggested. The user starts the task and the artifact of type Test ("TestA.java") is created and the context is opened, meaning both the Class and Test artifacts are now in the editor. The user writes the Test artifact and completes the task. All the tasks in the activity are complete and the system does not suggest any new tasks. This means that the activity is complete.

We are now moving on to the workflow where the user wants to modify an already existing activity. The user modifies the Story artifact ("StoryA.txt") and the task for creating the Class artifact is automatically set to the status "To Redo" (Task B). The user starts the task and the context is opened, meaning both the Story and Class artifacts are now in the editor. The user makes changes to the Class artifact and completes the task. The task for creating the Test artifact (Task C) is set to "To Redo". The user starts the task and the context is opened, meaning both the Class and Test artifacts are now in the editor. The user makes changes to the Test artifact and completes the task. All the tasks in the activity are complete and the system does not suggest any new tasks. This means that the activity is complete.

There is a cost for providing a structure to the activity, as well as the automation capabilities and that is the establishment of the rules defining the architecture and the development process. In the case of an already supported workflow the cost for the developers is none, but in the case of a new workflow a

single developer needs to previously create the rules that define the project (Option B).

In this case the developer would first create the artifact types. Still following Figure 3.3, these are Story, Class, and Test. Then he would create the dependency types. These would be a dependency type connecting the artifact type "Start" (already existing artifact type with the sole purpose of making the first connection) to the artifact type Story, another connecting Story to Class, and another connecting Class to Test. The developer would then create the task types. The task type "A - Create a Story" is made with the dependency going from "Start" to Story, the task type "B - Create a Class" is made with the dependency going from Story to Class, and the task type "C - Create a Test" is made with the dependency going from Class to Test.

As the rules that define the project are now created the developer can then move on to the workflow above from Option A.

The case of an already existing not yet supported project (Option C) means that it has rules that the tool does not understand, and so the developer needs to explain these rules to the tool. Concretely this means attributing artifact types to the existing artifacts and establishing the dependencies between them. Of course, the developer only has to do this if he intends to use these artifacts in activities, particularly to track modifications.

If there were three preexisting artifacts, one called "StoryOne.txt", another called "ClassOne.java", and another called "TestOne.java", the developer would assign the Story artifact type to "StoryOne.txt", the Class artifact type to "ClassOne.java" and the Test artifact type to "TestOne.java".

Now the system would know what the artifacts are but it still would not know how they relate to each other. The developer would create a dependency between the artifact "Start" and "StoryOne.txt" with the dependency type connecting "Start" and Story, a dependency between the artifact "StoryOne.txt" and "ClassOne.txt" with the dependency type connecting Story and Class, and a dependency between the artifact "ClassOne.txt" and "TestOne.txt" with the dependency type connecting Class and Test.

The tool now has full understanding of the artifacts and their relations, so the developer can then move on to the workflow above from Option A.

3.5 Goals and Collaboration

During the development of software, teams have to coordinate in the division of goals into a set of work tasks. This is currently achieved through a mix of personal meetings and the use of issue management tools.

Developers break down their work into tasks in order to accomplish the projects goals. This work breakdown is represented by a set of connected tasks that need to be assigned, managed and tracked.

Our tool supports developers by integrating the development goals with the project's software architecture to break down and connect the work as a set of tasks.

In the context of the project of the Software Engineering course at IST, the student groups use the GitHub project functionalities to manage the issues associated to each sprint which information is used by the teachers to assess the students contribution.

As GitHub and its functionalities are widely used, not only by software engineering students, but also by any group of people that requires coordination in their work, fact that is emphasized in case of no co-location of members, we chose it as our platform for collaboration.

In our tool the developers align their goals with the project's software architecture by joining them with the activities. Each goal can span several activities. By associating a goal with an activity, we can also associate it with its containing tasks and artifacts.

This way we can give the user information on if the goal is complete, that is, all its activities are complete. And more importantly, what tasks and what artifacts are currently being worked on by other members of the project.

If a task has the status "Active" it is displayed on GitHub as an open issue. If a task has the status "Complete" it is displayed on GitHub as a closed issue. This management is done automatically by the tool, as long as the user is logged in. Given that a task is "Active" its artifacts are marked as being worked on.

Goals are displayed on GitHub as milestones and connected to issues. The contribution of each user can also be seen on GitHub.

4

Implementation

Contents

4.1 Platform	33
4.2 Activity View	34
4.3 Design View	38
4.4 Goals and Collaboration	43
4.5 System	46

In Chapter 4 we begin with where we chose to implement our tool in **Platform**. In **Activity View** we describe one of our main views where the developers execute their usual workflow. In **Design View** we describe our view where the developers create the rules for their project, whether it is a new project, in the subsection **New Not Yet Supported Projects**, or an existing project in **Existing Not Yet Supported Projects**. In **Goals and Collaboration** we describe our approach to sharing information between the project's members. At last, in **System** we give a general idea of the Plugin's code structure and analyse the code of its main system in the subsection **Task Suggester**.

4.1 Platform

When deciding on a platform to implement our tool there were several options. We could go for a web based online implementation that many classic issue-based systems go for, we could opt for a windows application, or an IDE plugin like Mylyn.

An online implementation has the benefit of being easily accessible and prone for collaboration, however it is the option that is furthest away from the artifacts and the workspace.

An IDE plugin is the closest to the artifacts and the workspace and has the benefits of using the IDE's API to provide several important features like managing the editor. Being a plugin itself also means that it can be used in any system that the IDE supports, where an OS-based application would need additional support.

The ideal choice of implementation would be a hybrid application, providing an online view for ease of access and collaboration, managing the flow of information through every tool in the pipeline, and using these tool's APIs for closer access to the workspace and the artifacts. However, the scope is simply too large in this case.

We decided to implement our tool as an IDE plugin, specifically for Eclipse, as it is a very popular development environment for Java, the language currently used in the Software Engineering course at IST. We benefit from close access to the workspace and the artifacts, can provide a tool closely integrated with the working environment, and can still offer online functionalities through the GitHub API. A figure representing the whole environment can be seen on Figure 4.1.

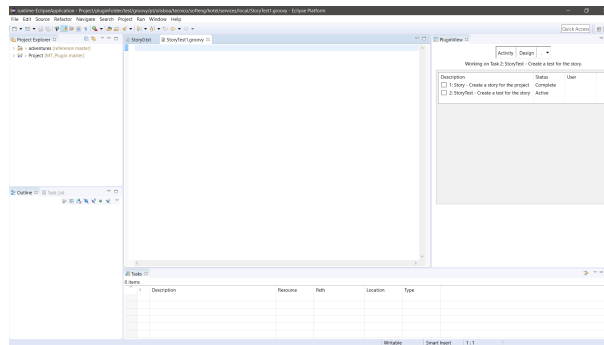


Figure 4.1: Eclipse Workspace

4.2 Activity View

Our Plugin is split into two main views, the Activity View and the Design View. These two views are accessible through the top bar seen on Figure 4.2 and are accompanied by the main menu seen on Figure 4.3.

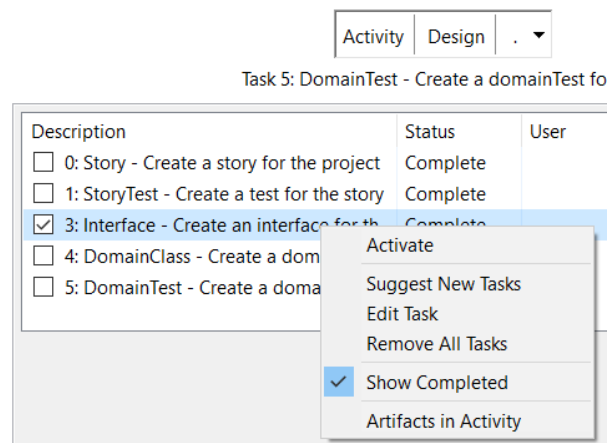


Figure 4.2: Activity View

This separation is important as there are three different ways to use our Plugin:

- Option A - The developer works on an already supported project.
- Option B - The developer creates his own rules for a not yet supported new project
- Option C - The developer creates his own rules for an already existing not yet supported project

Of course, it is also possible to modify the rules on an already supported project.

A supported project is a project where we already created the rules that state the software architecture and the development practice. All that is left to do is work on the tasks that are suggested to us as the

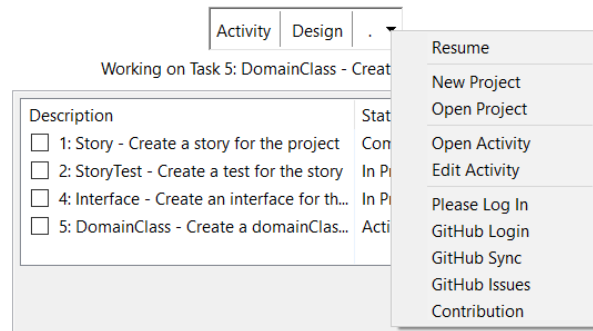


Figure 4.3: Plugin's Main Menu

system structures the project.

If a new project is not supported yet the first thing we do is create our rules. Then the system will take care of structuring the project. If we have a project that we already worked on, that means we produced artifacts and gave it a structure. In this case, in addition to creating our rules, we have to define in the system what the artifacts that we already created are, and how they relate to each other.

The usual workflow for a developer working on a supported project will only make use of the Activity View, the Task View, their menus and the respective options in the main menu. These options in the main menu are the "Resume", "Open Project", "Open Activity" and "Edit Activity" options. The "Resume" option restores the Plugin status to the last saved status, this is, the project, the activity and containing tasks last being worked on before closing Eclipse. The "Open Project" option to switch the project that we want to work on. The "Open Activity" option to switch the activity we want to work on. To note, that there is always a new activity available in addition to activities completed or in progress. And the "Edit Activity" that allows changing the development type that the activity follows and renaming activities. The default development type that a new activity follows is the default development type assigned to the project when it was created.

The Activity View represents a single activity on the project. In this view we see all the tasks contained in the activity, ordered by creation date. Being ordered by their creation date makes it possible to trace back the sequence they were worked on. The main information available to the developer regarding a task in this view is its description and the status. The description is automatically created based on the type of the task and the number of tasks in the activity. If the first task suggested is a task type with the description "Create a story for the project" whose main produced artifact is an artifact of type "Story" then its description will be "0: Story – Create a story for the project". This structure makes it easy for the developer to identify the task's purpose, the main product of the task and the sequence in the activity. The other main information is the task status. The main statuses a task can have are "Available", "Active", "Complete", "To Redo", and "In Progress" and can be seen in Figure 3.2. The tasks statuses and the transitions between them are described in the previous chapter "Design", at the end of

the "Model" section.

The Activity View, just like all the other views, is accompanied by a menu accessible through right click. The Activity View seen on Figure 4.2 has the options "Activate", "Suggest New Tasks", "Edit Task", "Remove All Tasks", "Show Completed", "Artifacts in Activity". The option "Activate" sets the task status to "Active", opens the Plugin's Task View, creates the task's context and opens it in the IDE's editor. The "Suggest New Tasks" option re-reads all the artifacts data and suggests new tasks if available. While working on a supported project this option is usually unnecessary. The "Edit Task" option allows the developer to change the task description and change the task status to any of the previously stated statuses including "Active" and "Complete". When using this option with several selected tasks it will change all their statuses. The "Remove All Tasks" option removes every task from the activity for a fresh start. The artifacts created during this activity are not removed. It is also not part of the usual workflow. The "Show Completed" option allows the developer to hide the completed tasks in order to focus on tasks that still need to be worked on. The "Artifacts in Activity" option can be seen on Figure 4.4.

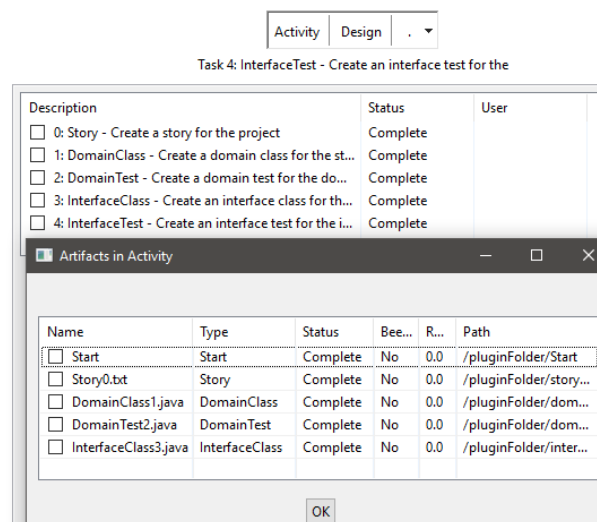


Figure 4.4: Artifacts in an Activity

This option shows the context of the whole activity. That is all the artifacts modified during its execution as well as their data such as the artifact type and artifact status. For ease of use, clicking on an artifact will open it in the IDE's editor.

The Task view seen on Figure 4.5 is accessible by activating a task in the Activity View. There is at most only one active task. There are two reasons for this choice. The first reason is that we want the developer to stay focused on a task at a time. The second reason is to track the artifacts modified during the execution of the task. The Task View is composed by the task description at the top, the used artifacts list, the produced artifacts list, and the menu. The used artifacts are the artifacts that already existed and triggered the creation of the task. The produced artifacts list includes the produced

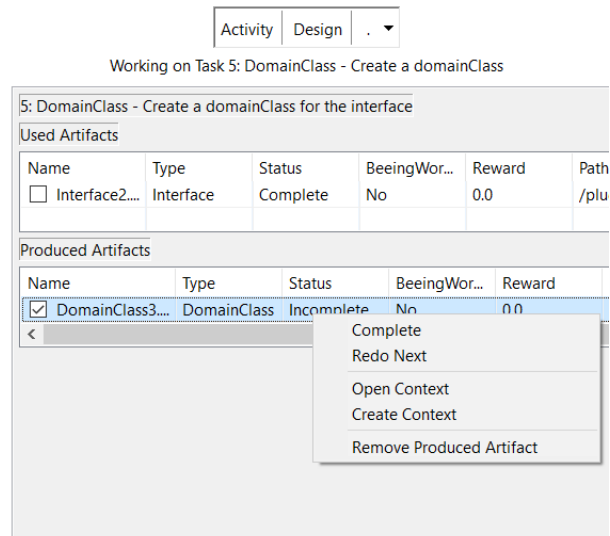


Figure 4.5: Task View

artifacts of the task, and other artifacts that were modified during the execution of the task. Each artifact is described by its file name, artifact type, status, if its being worked on, its reward and its path. The Task View's menu has the options "Complete", "Redo Next", "Open Context", "Create Context" and "Remove Produced Artifact". The "Complete" option tries to change the tasks status to "Complete". However, in order to be completed it needs to satisfy two conditions. The first one is that all its dependencies are realized. Meaning that if the task's corresponding task type has a dependency type from an artifact type "Story" to two of artifact type "Class", two artifacts of the type "Class" need to be present for the condition to be satisfied. The second condition is that there is no rule stating that tasks with the corresponding main produced artifact type stay in "In Progress" status until stated otherwise. After changing the active task's status, the Activity View is reopened, and new tasks are suggested if available. The "Redo Next" option causes the tasks depending on the main produced artifact to have their status changed to "To Redo" if they were in "Complete" status. This option is usually for cases where the active task was automatically reactivated, and we want to keep following the change propagation. This option is off by default as it may not be desired. The "Open Context" option closes the current IDE's editor elements and opens all the artifacts in the current task. It is called by default when a task is activated and can be called again if the workspace needs to be refocused. The "Create Context" option creates the produced artifacts if they do not exist. This option is also called by default when a task is activated and can be called again if there was an error on creation. The "Remove Produced Artifact" option removes the selected artifacts from the produced artifact list. This option is relevant because every modified artifact during the execution of the task is added to this list and it is possible that some of them are not desired as resulting from the task.

As the reader might have noticed, each task type can have several produced artifact types, and one of

them is the main produced artifact type. This means that a task has a main produced artifact. There are several reasons for this choice, beginning with ease of implementation of a few functionalities. As artifacts are edited during the actualization of a task they are added as produced artifacts of that task. It is always important to keep in sight what the main purpose of the task is. It is also important to keep the size of a task contained. We are not working with microtasks but we believe that focusing the work of a task on few artifacts (small context) is optimal for maintaining compliance with the architecture and the development practice and so, productivity.

This finishes up the part of the interface directly related to the main workflow (Option A). The most clicked options will very likely be the "Activate" option on an available task in the Activity View, and the "Complete" option in the Task View, which leads to another "Activate" option on a new available task.

4.3 Design View

4.3.1 New Not Yet Supported Projects

Before we go into the Design View, we need to create the project. In order to create the project we access the main menu in the rightmost icon of the top bar of the Plugin. The option we are looking for is "New Project". The "New Project" option opens a shell where we can describe the new project. The descriptors are the name of the project and the type of the project. The type of the project is hard coded and comprises a data template and a metadata template. The assigned project type also sets the default metadata template of all created activities, which can be changed in the option of the same menu "Edit Activity". The data and metadata templates refer to the data in the respective sections of the Design View that we will go over now.

In the Design View a developer can describe the software architecture and the development process of the project. The Design View has two sections. A Data section containing a Package list, an Artifact list, a Dependency list, and a Goals list. And a Metadata section containing an Artifact Type list, a Dependency Type list, a Task Type list and a Rules list.

In order to create the rules that define our new project we will begin in the Metadata section of the Design View. We start by creating the artifact types that we will be dealing with.

The Artifact Type View can be seen on Figure 4.6. An artifact type is described by a name, a file extension, and a reward. Knowing the file extension of the artifact type is necessary for the Plugin to automatically create the files. The Artifact Type View menu contains the options "New Artifact Type", "Edit Artifact Type", and "Remove Artifact Type". The "New Artifact Type" options opens a shell where we can describe a new artifact type. The descriptors are the same as the ones present on the view, the name, the extension and the reward. The "Edit Artifact Type" option allows us to edit one or more selected artifact types. The "Remove Artifact Type" option permanently deletes one or more artifact

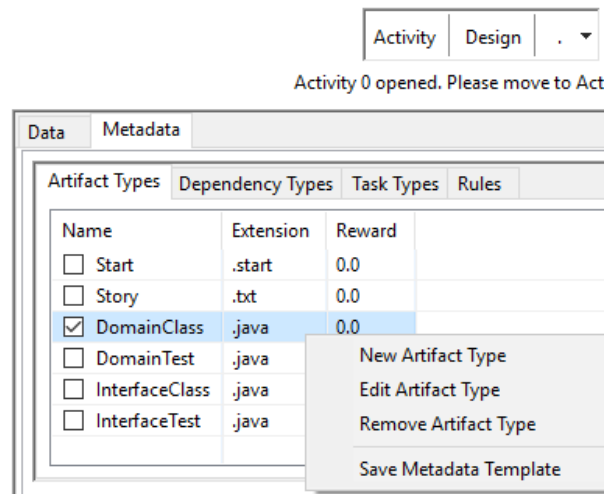


Figure 4.6: Artifact Type View

types.

As seen on Figure 4.6 we have an artifact type named "Start". This is a preexisting artifact type that comes with every project and has the sole purpose of making the first connection. That is, in this current development practice, the precondition for the existence of an artifact of type "Story".

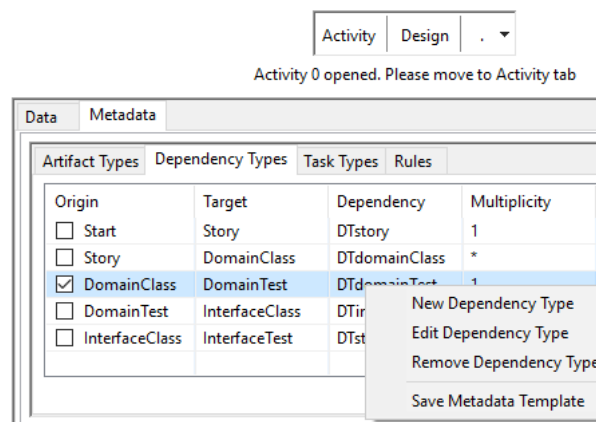


Figure 4.7: Dependency Type View

Having the artifact types, we can now describe the relationships between them. The Dependency Type View can be seen on Figure 4.7. A dependency type is described by the origin artifact type, the target artifact type, a name and a multiplicity. The multiplicity can be a natural number or "one or more". The standard multiplicity is one. It is important to state different multiplicities because a product artifact might be dependent on more than one artifact of the same type. The Dependency Type View menu contains the options "New Dependency Type", "Edit Dependency Type", and "Remove Dependency Type". The "New Dependency Type" option opens a shell where we can describe the new dependency type. The descriptors are the same as the ones present on the view, the origin artifact type, the target

artifact type, the dependency name and the multiplicity. The "Edit Dependency Type" option allows us to edit one or more selected dependency types. The "Remove Dependency Type" option permanently deletes one or more dependency types.

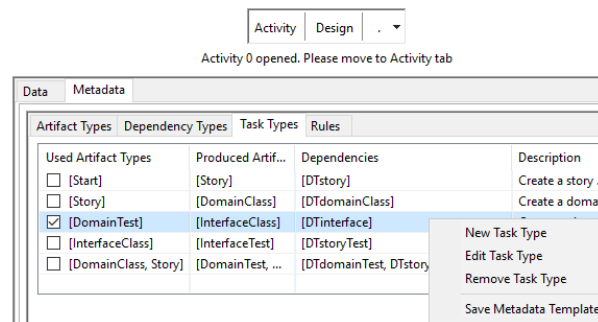


Figure 4.8: Task Type View

Having dependency types, we can now group them in a task type. The Task Type View can be seen on Figure 4.8. A task type is described by the containing dependency types and their used artifact types and produced artifact types, and a description. A task type contains one or more dependency types which are identified in the dependency types column by their name. The group of artifacts types set as origin in these dependency types is show in the used artifact types column. The group of artifact types set as target in these dependency types is shown in the produced artifact types column. The Task Type View menu contains the options "New Task Type", "Edit Task Type", and "Remove Task Type". The "New Task Type" option opens a shell where we can describe the new task type. The descriptors are just the dependency types and the description. The dependency types contain all the artifact types involved in the task type so there is no need to describe them. It is also to note that the first dependency type added is taken as the main dependency type of the task. This means that the main product of the task is an artifact of type corresponding to its target artifact type. The description should be generalized so that it can be used to generate new tasks. The "Edit Task Type" option allows us to edit one or more selected task types. The "Remove Task Type" option permanently deletes one or more task types.

The last view in the Metadata section is the Rules View. The Rules View can be seen on Figure 4.9. The purpose of this view is for particular use cases that cannot be covered by the other views and will rarely be used at this point of the project structuring. Unlike the previous metadata where everything is custom, here we can only select from existing rules and assign them to artifact types. A rule is then described by an artifact type, a name and a description. The Rules View menu contains the options "New Rule", and "Remove Rule". The "New Rule" option opens a shell where we can describe the new rule. The descriptors are the same as the ones present on the view, the artifact type, the name and the description. The "Remove Rule" option permanently deletes one or more rules. The only currently available rule is "AddStatusInProgress" and what it does is force tasks to remain in the "In Progress"

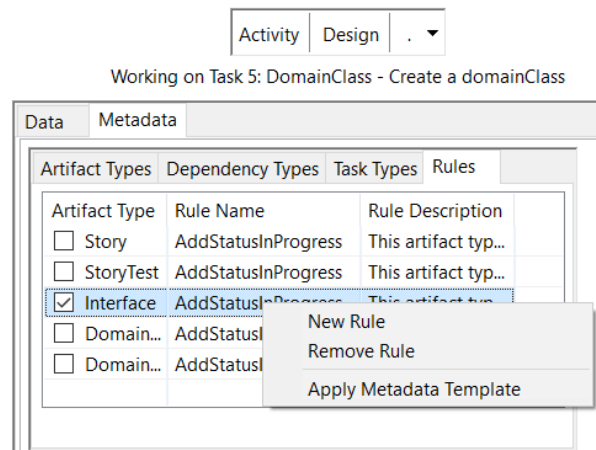


Figure 4.9: Rules View

status after completion, allowing new tasks to be suggested but stating that the task was not finished yet because the artifacts from the new tasks are required. This is especially useful for test-first development where regular tests will not pass at creation. Having the artifact types, the dependency types, the task types and maybe rules, the developer has successfully created the metadata template for the new project. We can save the metadata template with the "Save Metadata Template" option in every menu of the Metadata View. This opens a shell where we can give it a name and save it. All there is left to do is switch to the Activity View.

4.3.2 Existing Not Yet Supported Projects

Working over an already existing project that is not yet supported is a very different situation and requires quite a different workflow. In a supported project the developer workflow consists of using the Activity View and the Task View. In a new not yet supported project the developer creates his rules in the metadata section of the design tab, and then, the workflow switches to that of a supported project. An already existing not yet supported project means that it has rules that the Plugin does not understand, and so the developer needs to explain these rules to the Plugin. Concretely this means additionally using the data section of the Data View.

The Package View can be seen on Figure 4.10. This view includes all the folders in the project. A package is described by its name, system file path, type and if it is the default package for that type. A package can be attributed a type and set as default. The consequence of this is that new artifacts with that type will be created in that package. While this is not necessary, as otherwise, new artifacts will simply be created in a new folder with their artifact type as name, it is very useful for structuring the project and making sure the Plugin automatically creates the artifacts in their predestined location. The name column is used for a simple way to identify the package, but very often there will be packages with

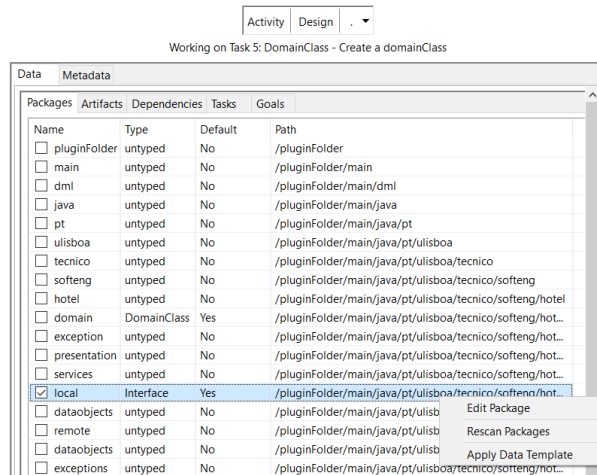


Figure 4.10: Package View

the same name, so it is necessary to differentiate them by the file system path. The Package View menu contains the options "Edit Package", and "Rescan Packages". The "Edit Package" option opens a shell where we can set the packages type and default setting. The "Rescan Packages" option causes the Plugin to rescan the project folder and refresh this view's table according to the new and the removed packages. This is often unnecessary as the Plugin calls this often, but it is useful in immediate situations.

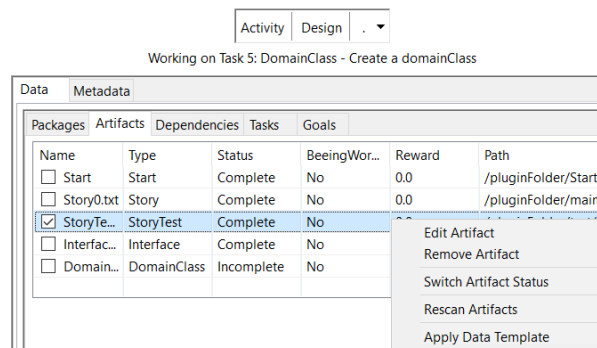


Figure 4.11: Artifact View

Setting a type on a package, also sets the type of its containing artifacts to that of the package. For the cases where we want an artifact to have a different type, and for the cases where the artifact is still untyped we can use the Artifact View seen on Figure 4.11 to set the type of each artifact. This view includes all the files in the project. An artifact is described by its name, artifact type, status, path, reward and if it is being worked on. Completing a task where the artifact is a produced artifact sets the artifact to the "Complete" status. Otherwise, it is in the "Incomplete" status. The reward is the number of points the developer wins once the status of the artifact is "Complete". The Artifact View menu contains the options "Edit Artifact", "Remove Artifact", "Switch Artifact Status", and "Rescan Artifacts". The "Edit

Artifact” option opens a shell where we can edit one or more artifacts. The descriptors are the artifact type, the status and the reward. The ”Remove Artifact” option deletes the artifact information from the Plugin. The ”Switch Artifact Status” switches the artifacts status to the opposite status. The ”Rescan Artifacts” option causes the Plugin to rescan the project folder and refresh this view’s table according to the new, modified and removed artifacts. This is often unnecessary as the Plugin calls this often, but it is useful in immediate situations.

Origin	Target	Dependency
<input type="checkbox"/> Start	Story0.txt	DTstory
<input type="checkbox"/> Story0.txt	DomainClass1.java	DTdomainClass
<input checked="" type="checkbox"/> DomainClass1.java	DomainTest2.java	DTdomainTest
<input type="checkbox"/> DomainTest2.java	InterfaceClass3.java	DTinterface
<input type="checkbox"/> InterfaceClass3.java	InterfaceTest4.java	DTstoryTest

Figure 4.12: Dependency View

As the existing artifacts in the project were not created by a task and so they are not structured by a task type, although the Plugin is now aware of their types there still is no knowledge of the relationships between them. In the Dependency View seen on Figure 4.12 we can establish the relationships between our artifacts. A dependency is described by the origin artifact, the target artifact, and the dependency type. The Dependency View menu contains the options ”New Dependency”, ”Edit Dependency”, and ”Remove Dependency”. The ”New Dependency” option opens a shell where we can describe the new dependency. The descriptors are the same as the ones present on the view, the origin artifact, the target artifact, and the dependency type. The ”Edit Dependency” option allows us to edit one or more selected dependencies. The ”Remove Dependency” option permanently deletes one or more dependencies. With artifact types, the dependency types, the task types and maybe rules, the developer created in the metadata section, and the newly attributed types to packages and artifacts, as well as the establishment of dependencies between these artifacts, the developer has finished describing the projects structure and all that is left to do is switch to the Activity View.

4.4 Goals and Collaboration

As we discussed in Chapter 3, GitHub and its functionalities are widely used, not only by software engineering students, but also by any group of people that requires coordination in their work. In our Plugin

the developers align their goals with the project's software architecture by joining them with the activities. Each goal can span several activities. By associating a goal with an activity, we can also associate it with its containing tasks and artifacts.

In order to do the connection with GitHub we used the GitHub V3 API library "GitHub Java API" (org.eclipse.egit.github.core). On GitHub our goals are represented as milestones, and our tasks are represented as issues. Unfortunately, the chosen library milestone API functionality does not appear to work, so we can't display it here.

In the last section of the Plugin's main menu seen on Figure 4.3 we have the GitHub related options. This section starts with a "Please Log In" sign that changes into "Logged In as *username*" once the developer logs into GitHub. The following options are "GitHub Login", "GitHub Sync", "GitHub Issues", and "Contribution". The "GitHub Login" option opens a shell where we can write our GitHub username in a normal text field, and our password is a password text field. Once we click on "OK" we should be logged in and the text above the option changes. The password is not stored. Unfortunately, special authentication methods such as OAuth are not yet supported. The "GitHub Sync" option synchronizes our tasks and goals with their respective counterparts issues and milestones on GitHub. This is often unnecessary as the Plugin calls this often, but it is useful in immediate situations. The "GitHub Issues" option opens the GitHub issues web page of the current project. The "Contribution" option opens the GitHub pulse web page of the current project where each developer's contribution is tracked.

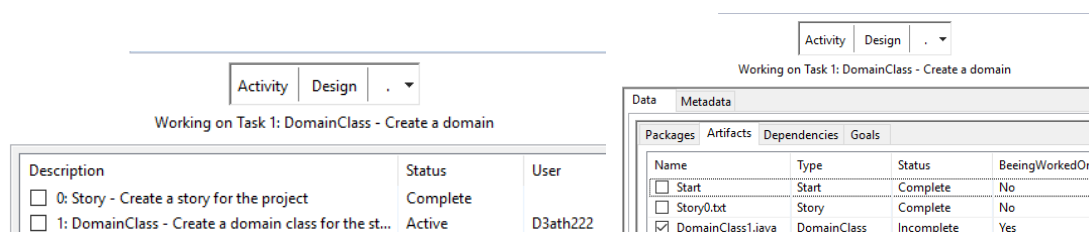


Figure 4.13: Collaboration - Activity/Artifact View

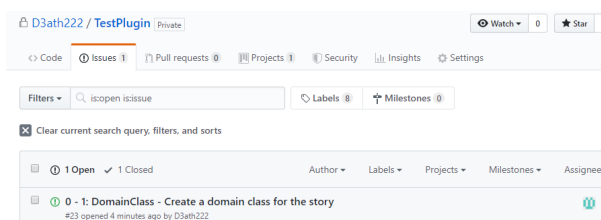


Figure 4.14: Collaboration - GitHub View

When we are logged into GitHub we gain the ability to know what tasks and what artifacts are currently being worked on by other members of the project. On the left part of Figure 4.13 we can

see that we have the task for the creation of a "Story" as complete, and the task for the creation of a "DomainClass" as active. As we are logged in, we can see our username in the user column. On the right part of the same figure, we can see that the artifact of type "DomainClass" is marked as currently being worked on. On Figure 4.14 we can see that in the GitHub issues web page our active task is Open, and there is a Closed issue, which is our complete task. This management is done automatically by the Plugin, as long as the developer is logged in.

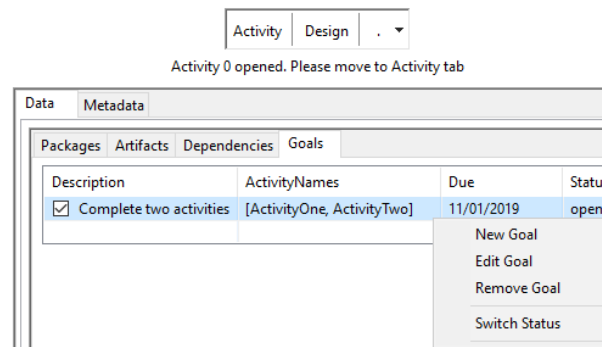


Figure 4.15: Goal View

In the Goal View seen on Figure 4.15 we can create goals and share them as milestones to GitHub. A goal is described by its description, the names of the activities that it is associated with, the date it is due, and its completion status. The Goal View menu contains the options "New Goal", "Edit Goal", "Remove Goal", and "Switch Status". The "New Goal" option opens a shell where we can describe the new goal. The descriptors are the similar to the ones present on the view, the goal description, the names of the activities that it is associated with, and the date it is due. The "Edit Goal" option allows us to edit one or more selected goals in this view and on GitHub. The "Remove Goal" option permanently deletes one or more goals from this view and from GitHub. The "Switch Status" manually changes the goal status from "open" to "closed" or the reverse, both on the view and on GitHub. That said, the switch is done automatically by the Plugin once all the goal's activities are complete.

When creating a goal we associate activity names to it instead of activities themselves. This is done because they may not exist yet, as goals represent work to be done. Once a goal is created, a milestone representing it is also created on GitHub. Once an activity with the described name is created, its tasks are associated to the goal on GitHub. That is, the goal becomes a milestone on the issue representing the task.

This concludes our section on Goals and Collaboration, where we described how we currently share goals, tasks, artifacts and all their statuses in real time, to make it easier for members of the same project to know what tasks to work on, by automatically sharing the status of the project. In Chapter 6 we discuss the next step to take regarding collaboration.

4.5 System

In Eclipse close to all the functionality is provided by plugins, on top of a small run-time kernel. In order to implement our tool as an Eclipse Plugin, we used Eclipse 4, Standard Widget Toolkit (SWT), and WindowBuilder 1.9.1. Our Plugin is then a fragment that extends `org.eclipse.e4.workbench.model`. As libraries we also used Gson 2.8.5 and GitHub Java API 2.1.5.

There is a main view that contains our own composite, that extends SWT Composite. This main composite represents all the area of the Plugin which is split in a top bar and space for other composites below it. The main composite manages who should be in the space below the top bar, whether it is the activity composite or the design composite. These are the activity and the design views that we already know. In the design composite we have a SWT TabFolder with the data view and the metadata view, each of them also having their own TabFolder for each of their containing views. In each view we have a JFace TableViewer for the table, and a SWT Menu for the menu. The windows that pop up from the menu options are SWT Shells.

```
public class Share {  
  
    public static Status status;  
    public static Project project;  
    public static PluginView pluginView;  
    public static MetadataView metadataView;  
    public static DataView dataView;  
    public static ArtifactView artifactView;  
    public static TaskView taskView;  
    public static ActivityView activityView;  
    public static MainActivityView mainActivityView;  
    public static MainTaskView mainTaskView;  
    public static String pluginFolderName = "pluginFolder";  
    public static Label message;  
  
    public class Status {  
        private List<Artifact> artifacts;  
        private List<ArtifactType> artifactTypes;  
        private List<Task> newTasks;  
        private List<Activity> taskContext;  
        private String projectType;  
        private List<TaskType> taskTypes;  
        private List<Dependency> dependencies;  
        private List<DependencyType> dependencyTypes;  
        private List<Component> components;  
        private List<Goal> goals;  
        private List<Rule> rules;  
        private List<RuleType> ruleTypes;  
        private List<CustomMetadata> customMetadata;  
        private Activity activeActivity;  
        private Task activeTask;  
        private String projectLocation;  
        private int newArtifactNumber;  
        private transient IProject project;  
        private transient GithubConnector github;
```

Figure 4.16: Plugin's Share Class and Status Class

On the left side of Figure 4.16 we can see the Share class that is used to communicate information between the views and the class that maintains the status of the current project on the right. The status of a project is written to its respective file on the project's folder by our JSON translator that uses Gson. The projects are created by the ProjectCreator class that creates the base java project structure, the new project's Status JSON file, and then applies the chosen data template representing the architecture, and the metadata template representing the development practice.

In order to track changes to the artifacts, update them and reactivate tasks, we have a class that implements `IResourceChangeListener`, which allows us to receive and deal with events related to resources in Eclipse.

The GitHub connection and functionality is managed by the `GithubConnector` class that uses the GitHub Java API.

4.5.1 Task Suggester

In the TaskSuggester class we create new tasks for the current activity, and create new activities if needed. The creation of new tasks is done by two methods. The first method creates tasks for existing artifacts. This usually happens in Option C situations where the developer has artifacts that were not created by the Plugin, that are connected with each other matching a not yet instantiated task type. The second method is responsible for creating tasks that create new artifacts. This method is almost always the one outputting all the tasks in Option A situations.

The second method, suggestNew() is then responsible for creating the new tasks for the current activity based on the current state of the project. We will now go over its code, which is available in Appendix A.

- 1 - We begin by checking if the current activity has tasks.
- 2 - If that is not the case then we create the tasks for the initializing task types (the task types with a dependency that has the artifact type "Start" as origin). We move to step 6.
- 3 - If the current activity already has tasks then we iterate over all the completed artifacts in the activity.
- 4 - In case a matching task type is found, it has not been initialized as a task using the current artifact, and all the required used artifacts are available and complete, then the task of that task type is created. We move to the iteration in step 3 until there are no artifacts left.
- 6 - The tasks created in the method are returned and added to the current activity.
- 7 - If a new activity is needed it is created. This cycle of task suggestion is now over.

5

Evaluation

Contents

5.1	Testing Method	51
5.2	Hypothesis	54
5.3	Results - Overall	54
5.4	Results - Students vs Professionals	58
5.5	Identified Problems and Solution	61
5.6	Questionnaire	63
5.7	Conclusion	65

In Chapter 5 we begin with the reasoning and description of the validation of our tool in **Testing Method**. In **Hypothesis** we present our tentative answer to the research question. In **Results - Overall** we analyse compliance, performance and unsuccessful user tests, while comparing our baselines. In **Results - Students vs Professionals** we analyse compliance and performance per user type. In **Identified Problems and Solution** we go over the issues that user testing raised and how we intend to solve them. In **Questionnaire** we analyse the results of the questionnaire, including per user type. At last, in **Conclusion** we match our hypothesis with the results and conclude our evaluation.

5.1 Testing Method

We decided to focus the evaluation on answering our main research question:

- In the frame of a software architecture, can the association of working context with a development activity foster the compliance with predefined development practices, and improve the developer's productivity?

To answer this question, we defined the development context associated with an activity as the set of tasks performed to accomplish the activity, as well as the set of artifacts created/modified during the execution of tasks.

In this context we've defined the compliance with development practices as the particular sequences by which tasks are performed in the context of an activity.

In this same context we've defined productivity as the execution time and success of the steps required to (1) implement a new activity, (2) perform a software change in the context of an activity.

In the previous chapters we went over how we developed a task-centric software development tool that supports developers by providing a structured view of the working context, and now we will present how we intend to evaluate it.

In order to test compliance, we need to do user testing. As per our definition of productivity in the context of our research question, we require the implementation of a new activity, and a software change in the context of an activity that was previously finished. Regarding the compliance with the development practice we should then observe the particular sequences by which tasks are performed in each activity. We also decided to focus more user tests on our Plugin, instead of splitting the tests into creating balanced performance baselines for the Plugin and without using the Plugin. The main reason for this choice is the fact that having two balanced baselines would require that either half the user tests be done without the Plugin or the user test time doubled. As we want as many users as possible to test the Plugin and are limited by a short test time when dealing with students and professionals, we decided to do more tests with the plugin, that we can still split between the two user types, and still do enough tests to create a performance baseline without the Plugin.

These requirements lead to our User Test Guide, which can be split in four main parts. In order to familiarize the user with our tool, its environment and the test itself, we begin with a learning part where we guide the user through the realization of a new activity. Then we perform a modification to the metadata template in the Design View in order to give the user insight into the functionalities of our tool and set up the next activity. In the third part the user realizes a new activity on his own. This activity follows the structure we just created in the Design View. In the last part we return to the first activity, where the user performs, on his own, a software change that spans across every task in the activity. The User Test Guide B can be summarized as:

- Activity 1: Learning (Guided).
- Design Change: Create structure for Activity 2 (Guided)
- Activity 2: Complete activity alone (using structure from Design Change)
- Activity 3: Perform change in Activity 1 and re-complete activity alone

Activity 1 is used to teach the user how to use the Plugin in a basic scenario. We go through a series of tasks, starting with the creation of a Story, until no more tasks are suggested. We also track execution time for each task. In Activity 1 and Activity 3 we have five tasks, and in Activity 2 we have three tasks. These are:

- Task 1 - Creation of a Story (A1, A2, A3).
- Task 2 - Creation of a DomainClass (A1, A2, A3).
- Task 3 - Creation of a DomainTest (A1, A3).
- Task 4 - Creation of an InterfaceClass (A1, A2, A3).
- Task 5 - Creation of an InterfaceTest (A1, A3).

For each task we do the following steps:

- Activate the available task (A1, A2, A3).
- Drag and drop the created artifact from the Plugin's folder into the source folder (A1, A2).
- The code is provided to the user, which is briefly explained (A1).
- Rename the artifact to its name in the story (A1, A2).
- Complete the active task (A1, A2, A3).

Note: In this chapter we may refer to "Activity x" as "Ax" and "Task x" as "Tx".

In Activity 1 we want the user to comply with the architecture, which is made of a "Story", a "DomainClass", a "DomainTest", an "InterfaceClass" and an "InterfaceTest". In this simple activity the compliance with the architecture is achieved just by initiating a task a moving the created artifacts to the source package. We also want the user to comply with the development practice, which is achieved by activating the single available task, completing the created artifact, and marking the task as completed. The tasks, for the creation of the artifacts, are suggested in the order above.

In "Design Change" we change the metadata template and apply it to our next activity. This part is used to give the user a bit of insight into how our Plugin works and to set up our next activity. We also track the time the user takes to make this simple change in the Design View.

In Activity 2 we no longer guide the user. We provide him a new "Story" similar to the previous one. In this activity we expect the user to go through the same steps as in Activity 1, with the difference that the code is not provided. In this activity we track both success rate and task execution time. The success criteria for a task is completing all of the steps, that is, working on and completing the artifact produced in the task between the task's activation and completion. The success rate for the activity is the number of tasks successfully completed.

In Activity 2 we also want the user to comply with the architecture, which is made of a "Story", a "DomainClass", and an "InterfaceClass", as we removed the tests. In this simple activity the compliance with the architecture is achieved just by initiating a task and moving the created artifacts to the source package. We also want the user to comply with the development practice, which is achieved by activating the single available task, completing the created artifact, and marking the task as completed. The tasks, for the creation of the artifacts, are suggested in the order above.

In Activity 3 we also do not guide the user. We ask him to return to Activity 1 and re-complete the activity after replacing this activity's story artifact with the one from Activity 2. Once again, we track the activity's success rate and the task execution time. When the "Story" artifact is changed, the task that uses that artifact is set to the status "To Redo". We expect the user to activate and complete that task. New tasks with the status "To Redo" will then appear. Once every task has the status "Completed" the activity is finished.

In Activity 3 we already have compliance with the architecture, as we created all the artifacts in Activity 1. As we are changing artifacts, the system suggests tasks to be redone. The compliance with the development practice can either be done in the usual order of "Story", "DomainClass", "DomainTest", "InterfaceClass", "InterfaceTest", since every artifact requires change, or by choosing a task with "To Redo" status. Once Activity 3 is complete, the user move on to answer the questionnaire.

5.2 Hypothesis

Prior to performing the user tests we established the following hypothesis. It is based on the two main factors at play, the compliance with the development practice and software architecture, and the performance.

"By working in an activity context we can achieve an equal or higher percentage of compliance with the software architecture, a higher percentage of compliance with the development practice, reduced task execution time, and a low variation in user performance resulting from the fostered process. We can also obtain positive results on the questionnaire."

- The compliance with the software architecture is measured by the activation of every task in the test. Without the Plugin it is represented by the creation of all the artifacts in the test.
- The compliance with the development practice is measured by the particular sequences by which tasks are performed in each activity of the test. Meaning that our compliance with the development practice is our success rate, where for a user test to be successful every task in every activity of that user test is done successfully and in the expected sequence. Without the Plugin it is measured by the particular sequences by which artifacts are completed in each activity of the test.
- For performance compare the average execution time of coding tasks. We also look into the average standard deviation of Task 2 as it is one of the two tasks present throughout all three activities, and of the two it is the one with the highest coding effort. Our expectation is that by fostering compliance the users are guided such that they do not deviate from the intended behavior, and so, we can achieve a low standard deviation during the execution of tasks, and in that way further increase performance.
- With positive results in the questionnaire we mean that the average of the answers to the questions "Working in an activity fostered my compliance with the predefined development practice", "Using the Plugin made the process faster", and "The Plugin restricted my freedom of development" on the questionnaire answered at the end of each test, is greater than or equal to 4 (Agree) for the first two questions, and less than or equal to 2 (Disagree) for the last question.

5.3 Results - Overall

We performed 20 user tests with the Plugin, where all the tests were performed following the User Test Guide. We also decided to split our tests in two categories. We tested our tool with 9 students, and we tested our tool with 11 professionals. Since we have two testing categories, we will go over the test results in a general manner first, and then compare the two categories.

Each user test took an average of 20 minutes working in tasks, plus the time of reading the guide and answering the questionnaire. The overall average age is 28 years old. There was only one female user. We performed 6 user tests without the Plugin, identical to the procedure in the User Test Guide, but following the Story in each activity instead (Task 1), meaning that there are 6 non-guided tasks, instead of 8. Each user test took an average of 13 minutes. The overall average age is 22 years old. Every user had previous experience with Java, JUnit and Eclipse, although none regularly worked with all three. Every user has a computer science background and related current occupation.

5.3.1 Compliance

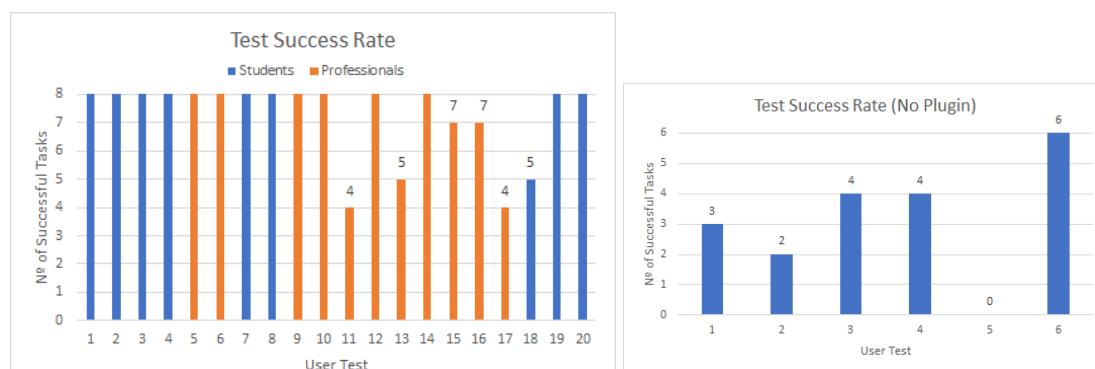


Figure 5.1: Test Success Rate

Overall, we had 100% compliance with the architecture in every test with the Plugin, as every task in every activity was activated at least once. This is all that is required since each task automatically creates its artifacts, and the artifact location can also be dealt with by the Plugin.

Without the Plugin we also had 100% compliance with the architecture as every artifact was created.

Overall, we had 70% user test success rate with the Plugin, as the number of user tests where every task in every activity of that user test was done successfully and in the expected sequence, was 14 out of 20. On Figure 5.1 we can see the overall task success rate per user test. The 8 tasks for which we collected data are the non-guided tasks from Activities 2 and 3. In 2 of the 20 tests (10%, tests 11 and 17) the user completed half of the test (4 out of 8) successfully. In 2 of the 20 tests (10%, tests 13 and 18) the user completed 5 out of 8 tasks successfully. In 2 of the 20 tests (10%, tests 15 and 16) the user completed 7 out of 8 tasks successfully.

Without the Plugin we had 17% user test success rate, as only 1 test completed every artifact in the expected sequence.

Overall, we had 90% task success rate with the Plugin, as the number of tasks that were completed successfully at the right sequence was 144 out of 160, with 16 failed tasks. On Figure 5.2 we can see the success rate per task. In Activity 2, the last task failed in 2 of the 20 tests (10%, tests 15 and 16). In

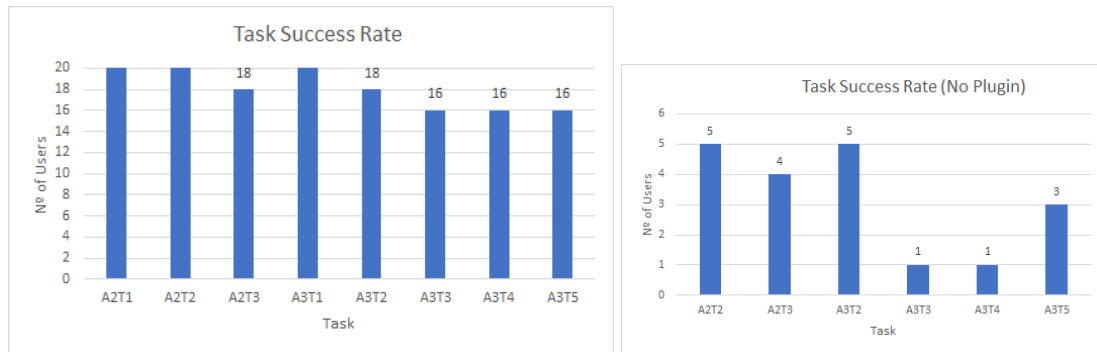


Figure 5.2: Task Success Rate

2 of the 20 tests (10%, tests 11 and 17) the second task of Activity 3 failed. In 4 of the 20 tests (20%, tests 11, 13, 17 and 18) the last three tasks of Activity 3 failed in a chain reaction.

Without the Plugin we had 53% task success rate with 19 out of 36 tasks completed successfully.

5.3.2 Performance

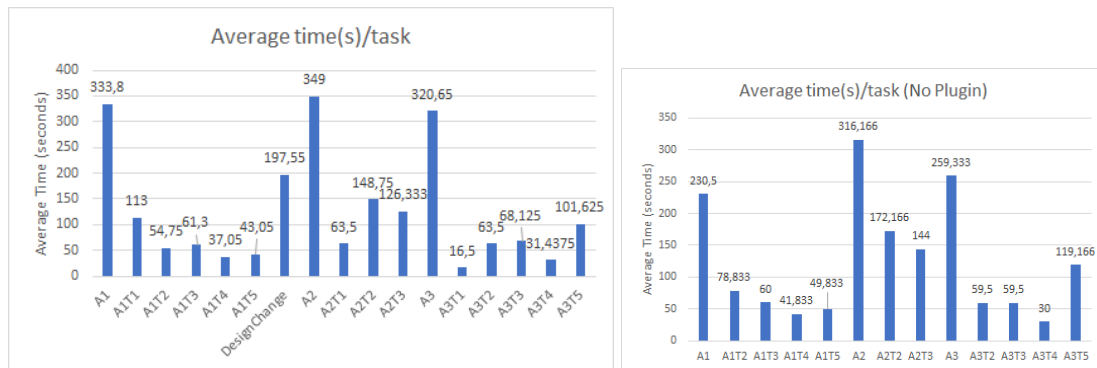


Figure 5.3: Average Time per Task

On Figure 5.3 we have the average time taken to complete a task, including bars for each activity. The average task time for tasks that do not involve coding is 54 seconds (A1, A2T1, A3T1, A3T2, A3T4). The average task time in the learning activity for tasks that do not involve coding is 62 seconds (A1). The average task time after the learning activity for tasks that do not involve coding is 44 seconds (A2T1, A3T1, A3T2, A3T4), which means an 18 second improvement just from finishing a learning activity. We believe this number could be further reduced. The average task time for tasks that involve coding is 111 seconds (A2T2, A2T3, A3T3, A3T5). Without using the Plugin, tasks involving coding took an average of 50 more seconds.

By order of average time of task involving a coding effort, we expected the following sequence for descending average task time: A2T2 > A2T3 > A3T5 > A3T3. The expected coding effort matches the

resulting coding effort. If the coding effort is being reflected in the task times as it seems to be, that means that the cost of using the Plugin, at least in the user test situation, is close to constant.

By order of just coding effort we expected the following sequence for descending average coding effort time: $A2T2 > A3T5 > A2T3 > A3T3$. As the tasks A2T2 and A2T3 involve the cost of using the Plugin and the cost of the code effort, their averages when subtracting the average cost of using the Plugin (44 seconds) become 105 seconds and 82 seconds respectively. As the tasks A3T3 and A3T5 involve an almost negligible cost of using the Plugin as they are done in a task reactivation workflow their average is left as is, at 68 seconds and 102 seconds respectively. This leaves us at $A2T2 (105s) > A3T5 (102s) > A2T3 (82s) > A3T3 (68s)$, matching the expected order, and confirming that users take a similar amount of time to use the Plugin.



Figure 5.4: Box Plot - Task 2 Performance

Figure 5.4 shows a box plot graph with the times taken to complete Task 2 in each of the activities. As we can see the results are highly concentrated, specially in Activity 3. The standard deviation in Activity 1 is 20 seconds (36% of average A1T2 task time), the standard deviation in Activity 2 is 55 seconds (37% of average A2T2 task time), and the standard deviation in Activity 3 is 18 seconds (35% of average A3T2 task time), which gives us an average of 31 seconds (36% of average Task 2 task time). The standard deviation is higher in Activity 2 due to Task 2 being the task with the highest coding effort. Even still there are only two outliers, with every other result being between the box's whiskers. Although Task 2 in Activity 1 takes an average of 55 seconds to complete, and Task 2 in Activity 3 takes an average of 64 seconds to complete, the standard deviation is 2 second shorter and the results are all between the box's whiskers. This is likely because the users already have some experience using the Plugin and knowledge of the development practice being enforced. Our low standard deviation in each of the activities and on average, implies that the Plugin is working as expected and the compliance with the architecture and development practice is correctly being fostered, resulting in an increase in

productivity.

5.3.3 Unsuccessful User Tests

We will now go over each of the user tests with the Plugin that did not have 100% success rate.

- In tests 11 and 17, the users did all of Activity 3 in its second task. As they finished the second task, they noticed that other tasks had errors, and so followed the errors in the package explorer and completed every task while the second task was still active.
- In test 13, the user made a small mistake during the second task of Activity 3 that stopped task 3 from being completed, and so returned to task 2 with task 3 still active. After fixing the mistake he followed the errors in the package explorer and completed the remaining tasks while the third task was still active.
- In test 15, the user did the last task of Activity 2 without activating the task.
- In test 16. the user did part of the last task of Activity 2 with the task active and then completed it. After noticing the feature was not implemented yet he implemented it without reactivating the respective task.
- In test 18, the mistakes occurred during Activity 3. The user did task 4 during task 3 instead. As he had already done task 4, he activated it and completed it immediately. In task 5 he did both task 5 and task 2.

We also go over user tests with 100% success rate where the users made a mistake and corrected it immediately.

- In test 5, during task 2 of Activity 2 the user first created a new class instead of using the class created by the plugin. After realizing his mistake, he deleted the class and proceeded as usual.
- In test 8, during task 2 of Activity 2 the user forgot to generate getters and setters and was in doubt if he should reactivate the task were the respective class was created or not.
- In test 20, during task 1 of Activity 3 the user started doing task 2 but immediately realized the task was not activated and restored the previous state by "Alt Z'ing", and then proceeded as usual.

5.4 Results - Students vs Professionals

We performed 20 user tests, 9 of them with students, and the other 11 with professionals. Judging our tool in an academic environment and in a professional environment allows obtaining feedback regarding

two different contexts where our tool could be used.

Each student and professional user test took an average of 20 minutes working in tasks. Every student user is currently enrolled in the MSc in Information Systems and Computer Engineering at Instituto Superior Tecnico. Every professional had a computer science course background and had a related occupation. These were Software Architect, Process Architect, Technical Lead, QA Manager, IT Consultant and Developer. The average age for students is 23 years old, and the average age for professionals is 32 years old. Every user, except a single student, were male.

5.4.1 Compliance

Overall, we had 89% student user test success rate, as the number of student user tests where every task in every activity of that user test was done successfully and in the expected sequence, was 8 out of 9. On Figure 5.1 we can see the overall task success rate per student user test. The students have the color blue, and the professionals have the color orange. In 1 of the 9 tests (11%, test 18) the user completed 5 out of 8 tasks successfully.

Overall, we had 55% professional user test success rate, as the number of professional user tests where every task in every activity of that user test was done successfully and in the expected sequence, was 6 out of 11. On Figure 5.1 we can see the overall task success rate per professional user test. In 2 of the 11 tests (18%, tests 11 and 17) the user completed half of the test (4 out of 8) successfully. In 1 of the 11 tests (9%, test 13) the user completed 5 out of 8 tasks successfully. In 2 of the 11 tests (18%, tests 15 and 16) the user completed 7 out of 8 tasks successfully.

There is a 34% difference in overall user test success rate between the student users and the professional users, favoring the students.

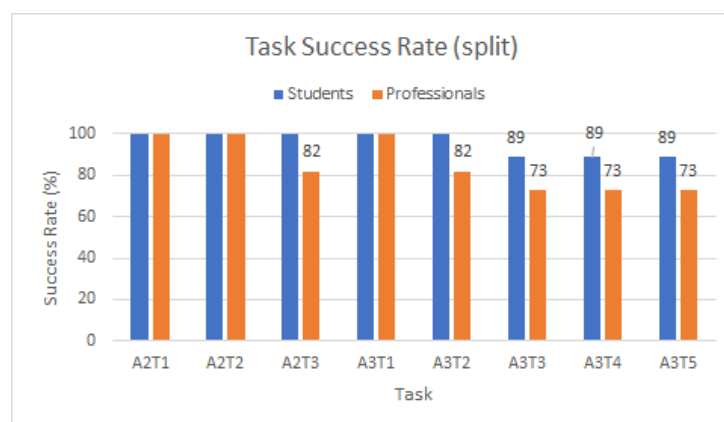


Figure 5.5: Task Success Rate (split)

Overall, we had 96% task success rate by the student users, as the number of tasks that were completed successfully at the right sequence was 69 out of 72, with 3 failed tasks. On Figure 5.5 we can see the success rate per task. These 3 failed tasks belong to the same test, where the last three tasks of Activity 3 failed in a chain reaction.

Overall, we had 85% task success rate by the professional users, as the number of tasks that were completed successfully at the right sequence was 59 out of 88, with 13 failed tasks. On Figure 5.5 we can see the success rate per task. In Activity 2, the last task failed in 2 of the 11 tests (18%, tests 15 and 16). In 2 of the 11 tests (18%, tests 11 and 17) the second task of Activity 3 failed. In 3 of the 11 tests (27%, tests 11, 13 and 17) the last three tasks of Activity 3 failed in a chain reaction.

There is a 11% difference in overall task success rate between the student users and the professional users, favoring the students.

5.4.2 Performance

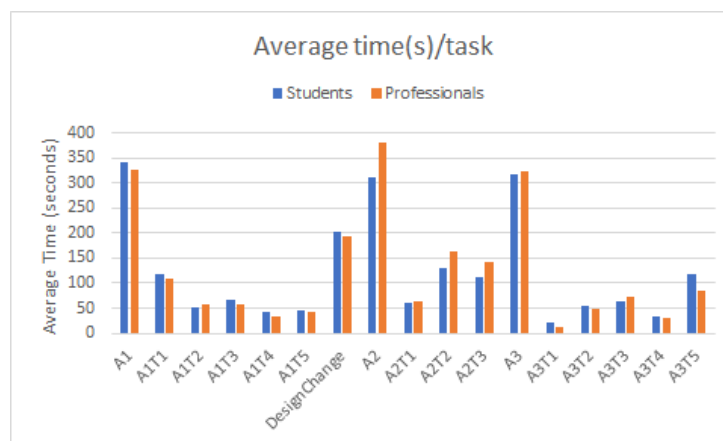


Figure 5.6: Average Time Task (split)

On Figure 5.6 we can see the average time per task, with columns for students and professionals. There is close to no difference between students and professionals when it comes to task time, with professionals being overall 15 seconds faster, but 30 seconds slower in Activity 2's tasks 2 and 3 which are the ones that require the most coding effort.

Figure 5.7 shows a box plot graph with the times taken to complete Task 2 in each of the activities for the students and for the professionals. As we can see the results are highly concentrated, especially in Activity 3. There is an exception, which is also the only visible difference between students and professionals, in Activity 2. Here there are three outliers between 250 and 300 seconds, where the following highest time is at 172 seconds. For the students the highest result is 13 seconds shorter, at

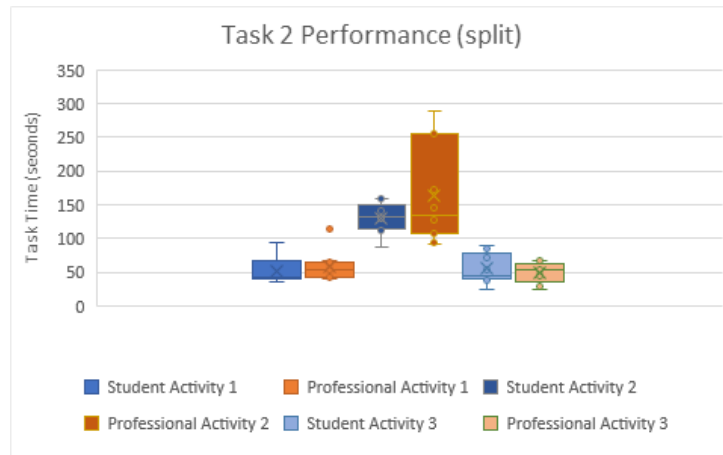


Figure 5.7: Box Plot - Task 2 Performance (split)

159 seconds. The standard deviation for the students in Activity 2 is 21 seconds (37% of average A2T2 student task time), and the standard deviation for the professionals in Activity 2 is 68 seconds (41% of average A2T2 professional task time). The average standard deviation for the students is 21 seconds (37% of average Task 2 student task time) and the average standard deviation for the professionals is 34 seconds (38% of average Task 2 professional task time).

5.5 Identified Problems and Solution

The problems revealed by the user testing can be split in three categories. Problems related to the User Test Guide followed by the users in each test, usability problems related to the Plugin itself, and problems related to the testing context and the users working experience.

Overall the main cause of task failure was untimely or lack of one of the three: task activation, task completion, task reactivation. The issues we will now go over often contribute to this problem.

The **User Test Guide** had three main issues:

- The **first issue**, and most obvious one, is that although users acknowledge that a learning phase is important, they don't seem to enjoy following a guide and much prefer going through the test by talking instead of reading through the guide. This is a problem, because every test should aim to be realized under the same conditions, and so we strived to do so. A user suggested to have a video guide instead.
- The **second issue** is that in Activity 1 we did not introduce a feature in the learning phase that would have proven useful in every test, and possibly prevented mistakes. This feature is the "Open Context" feature which rearranges the editor to display the artifacts related to the current task. This would have been helpful for two reasons. Because there was often more than one artifact with the

same name in the editor, and because if the user was not sure what to do, he could simply reopen the current context. Although it was not introduced, the user in test 20 did make use of it.

- The **third issue** is that in Activity 3 we did not explain the task reactivation functionality well enough to be understood by everyone at first glance. When an artifact is changed, if there is a task in the current activity that makes use of it to create another artifact, that task is set to the status "To Redo" which indicates that it is highly likely that that task must be redone. Activity 3 was all about making a change and redoing the activity according to the change. Explaining how this functionality works beforehand and asking for doubts could have increased the success rate in Activity 3.

The **Plugin** has three main issues:

- The **first issue** is how artifacts are added to the produced artifacts of a task. When working on a task, every artifact edited is added to the produced artifacts of a task. This happens because it makes sense that an artifact changed during a task is a relevant product of that task. For the cases where it isn't relevant, we also have the option to remove it. That said, during Activity 3, when the artifact produced during task 2 is modified, other two artifacts that use this class are indirectly modified as well and so added to the produced artifacts of the task. These artifacts are however, artifacts produced by the next tasks. This revealed a problem, where we only want artifacts directly modified by the user to be added to the produced artifacts of the task. It was counter-intuitive to add these artifacts to the task and then expect the users not to work on them, especially as the errors are evident on the package explorer. We believe that this problem had a significant impact on the success rate of tests 11, 13, 17 and 18. The user of test 20 also voiced his concern on this matter.
- The **second issue** regards the way the user interacts with the plugin, particularly the menu in each of the views. The way the Plugin displays information and allows the user access to functionality is standardized. Every view displays information in a table, such as the tasks in the Activity view. And every view allows access to functionality through a menu by right clicking. The main problem identified here is that sometimes it is not obvious for the user what he can do, even if it is just a right click away. It is also not obvious what the functionality is being called on. For example, in the Task view, the "Complete" functionality for completing a task is called by right clicking on one of the tables with artifacts. The artifacts also have a status which can be "Incomplete" or "Complete" and this confused a few users that were not sure if they were completing the artifact or the task. We believe that, for the Activity view, the "Activate" functionality for initiating the task should also have a clearly visible button, and it should be renamed to "Activate Task". And in the Task view, there should be a clearly visible "Complete" button for completing the task, and it should be renamed to "Complete Task".

- The **third issue** is an annoying bug that caused the mouse's right click button to also trigger a left click, which sometimes lead to difficulty in selecting the right element in the current table, particularly in the Activity view.

These three issues were resolved in the final version of the Plugin. There were two issues related to the **testing context** and the users working experience:

- The **first issue** is that some users claimed to be used to, when seeing an error, going to fix it immediately. This does not bode well for how the activity workflow is currently implemented if completing a task and activating the correct task for the artifact that has an error is easily forgotten upon seeing an error. A solution to this problem may be to implement a "Error Fixing" button or a "Auto-Activate Task" toggle that causes the correct task to be activated when an artifact is edited instead of adding it to the produced artifacts. This could also prove to be a favored way to work by some users.
- The **second issue** is that the we aimed the test time to be not too large, at around thirty minutes length. A limited learning time, paired with the users not having any prior knowledge on how the plugin works, leads to mistakes that could easily be avoided.

5.6 Questionnaire

An 18-question questionnaire was answered at the end of each user test. The questions that make up the questionnaire are four custom questions related to the Plugin (questions 1, 6, 7 and 8), the ten questions in System Usability Scale (SUS) for measuring usability (questions 9 through 18), and four questions adapted from NASA Task Load Index (NASA-TLX) to assess performance (questions 2 through 5). As we do not compare our system with other systems tested with either of these assessment tools, we did not include every question.

The full results are available in Appendix C. Here follow the results from each question in the format "Result. (Average — Mode)".

- Every user felt comfortable with Java and JUnit. (4,6 — Strongly Agree).
- Only one user thought he didn't successfully complete every task. This was on test 18 where the user did the tasks out of order. To note that every user completed every task, although in 6 of the tests they didn't do so while following the activity as expected. (4,35 — Strongly agree)
- Three users, tests 2, 7 and 14, thought the test to be mentally demanding. All of them had 100% success of the test. (2,6 — Disagree)

- Only one user, test 14, felt rushed to complete the test. (1,9 — Disagree)
- No users felt frustrated during the test. (1,7 — Strongly Disagree)
- Only one user, test 10, thought that using the Plugin did not make the process faster. The user explained saying that for bigger projects he thought the Plugin would make the process faster, but for this test he felt he would have been faster without using the plugin. (3,8 — Agree)
- Every user agreed that working in an activity fostered their compliance with the predefined development practice. (4,05 — Agree)
- Seven users thought that the plugin restricted their freedom of development. Ten other users disagreed. (2,75 — Disagree)
- Only one user, test 14, would not want to use the Plugin in other projects. (3,65 — Agree)
- Three users, tests 3, 9 and 10, found the Plugin unnecessarily complex. (2,3 — Disagree)
- Only one user, test 18, thought the Plugin was not easy to use. (3,55 — Agree)
- Three users, tests 11, 12 and 18, thought that they would need the support of a technical person to be able to use the Plugin. (2,35 — Disagree)
- Only one user, test 3, did not find the various functions in the Plugin well integrated. (3,85 — Agree)
- No user thought that there was too much inconsistency in the Plugin. A user mentioned that for use in at an enterprise level we could not allow users to have the degree of customization available in the Design view, and that it should be disabled for the common user. (1,8 — Disagree)
- Only one user, test 4, thought that most people would not learn to use the Plugin very quickly. (4— Agree)
- Two users, tests 3 and 10, found the Plugin cumbersome to use. (2,2— Strongly Disagree)
- Two users, tests 4 and 14, did not feel very confident using the Plugin.(3,4— Agree)
- No user thought they would needed to learn a lot of things before they could get going with this Plugin. (1,95— Disagree)

Overall the results of the questionnaire are positive. There was no question where the average or mode went against the ideal answer. The question that was closest to a negative result was question 8: "The Plugin restricted my freedom of development". Out of the seven users that disagreed with this question, none of them disagreed with question 7: "Working in an activity fostered my compliance

with the predefined development practice”, and only two of them chose the option “Neither Agree nor Disagree”. This leads us to believe that at least five users felt like the compliance with the development practice was fostered at the cost of freedom of development. We are not satisfied with this result as we don’t believe this to be the case. As there was no feedback from the seven users related to this matter, we leave the question open. In the real world, adherence to development practices is not total, where as our Plugin encourages full compliance. Another related aspect is that we have a single active task, and tasks follow a finish to start relationship. Perhaps, encouraging a more varied approach would be beneficial, even at the cost of compliance.

5.6.1 Students vs Professionals

The answers to the questionnaire were quite similar between students and professionals. There are only three questions with a variance greater than half a value (0,5), questions 8, 15 and 16, and only question 8 has a difference greater than one user.

- Two students thought that the plugin restricted their freedom of development. Six other students disagreed. (2,3 — Disagree)
- Five professionals thought that the plugin restricted their freedom of development. Four other professionals disagreed. (3,1 — Agree)

This result gives a bit more insight into our still open question. As the professionals answer is almost split in half, work experience such as other tools used or current workflow might have something to do with the difference in opinions. Still, this question is left open.

5.7 Conclusion

”By working in an activity context we can achieve an equal or higher percentage of compliance with the software architecture, a higher percentage of compliance with the development practice, reduced task execution time, and a low variation in user performance resulting from the fostered process. We can also obtain positive results on the questionnaire.”

- We achieved 100% compliance with the software architecture.
- We achieved 70% compliance with the development practice, 53% more than without the Plugin. We also achieved a 90% task success rate, 37% more than without the Plugin. There was a 34% difference in the first, and a 11% difference in the second, between the students and professionals, in favor of the students.

- We achieved a higher performance during the execution of coding tasks with the Plugin, and low deviation in user performance, concluding that by fostering compliance we can achieve a low standard deviation during the execution of tasks, and in that way further increase performance.
- The average and mode of the answers to the questions "Working in an activity fostered my compliance with the predefined development practice", "Using the Plugin made the process faster", and "The Plugin restricted my freedom of development" on the questionnaire was (4,05 — Agree), (3,8 — Agree), and (2,75 — Disagree) respectively. We are not satisfied with the results to the third question, so we leave it open. This last question also had a 0,8 difference between the students and professionals in favor of the students.

6

Conclusion

Contents

6.1 Future Work	70
---------------------------	----

Developers work on, and switch between different tasks throughout the day. By analysing the approach that current task-centric tools take to increase a developer's productivity, we identified several problems, the most relevant one being a lack of structure to, not only the task, but the workflow itself. We then took what we believe to be the next step, and created the concept of activity context, which led to our research question:

- In the frame of a software architecture, can the association of working context with a development activity foster the compliance with predefined development practices, and improve the developer's productivity?

To answer this question we defined the development context associated with an activity as the set of tasks performed to accomplish the activity, as well as the set of artifacts created/modified during the execution of tasks. We also defined the compliance with development practices as the particular sequences by which tasks are performed in the context of an activity.

In order to define our tool we created its base concepts. A development practice is then represented by our metadata template, defined by a group of Task Types, Dependency Types and Artifact Types. And the software architecture is represented by our data template, defined by a group of Goals, Packages, Dependencies and Artifacts.

A development project comprises several activities and each activity follows a single development practice, that can differ from those of other activities. This way, even keeping the same artifacts, by having different dependencies and task types, developers can easily switch between development practices, such as bottom-up and top-down.

Activities have the benefit of providing a structured sequential context that is data-centered, while increasing performance with some mechanisms of automatic support, from task suggestion to artifact creation, and by maintaining the sequential context of changes that occur during the activity implementation.

We then implemented our tool as an Eclipse Plugin where we benefit from close access to the workspace and the artifacts. The Plugin is split into two main views, the Activity View for the execution of activities, and the Design View for the creation of templates. Three different workflows are then possible:

- Option A - The developer works on an already supported project.
- Option B - The developer creates his own rules for a not yet supported new project
- Option C - The developer creates his own rules for an already existing not yet supported project

A supported project is a project where we already created the rules that state the software architecture and the development practice. The usual workflow for a developer working on a supported project will only make use of the Activity View. There are also two different workflows when working on an activity.

Either we are implementing a new feature by completing an activity, or we are make modifications by altering the artifacts in an already existing activity.

If a new project is not supported yet then we first have to create our metadata template. If we have a project that we already worked on, that means we produced artifacts and gave it a structure. In this case we have to define in the system what the artifacts that we already created are, and how they relate to each other in the data template.

In order to evaluate our tool we created an hypothesis: "By working in an activity context we can achieve an equal or higher percentage of compliance with the software architecture, a higher percentage of compliance with the development practice, reduced task execution time, and a low variation in user performance resulting from the fostered process. We can also obtain positive results on the questionnaire.". We than performed 20 user tests with 9 students and 11 professionals, and 6 user tests without the Plugin. We achieved 100% compliance with the software architecture, 70% compliance with the development practice, 53% more than without the Plugin. We also achieved a 90% task success rate, 37% more than without the Plugin. We achieved a higher performance during the execution of coding tasks with the Plugin, and concluded that by fostering compliance we can achieve a low standard deviation during the execution of tasks, and in that way further increase performance.

On average all the answers to the questionnaire were positive, but we were not satisfied with the results to "The Plugin restricted my freedom of development" so we leave the question open. Regarding the user types, the students user tests were on average more successful, with a 34% difference in compliance to the development practice, and a 11% difference in task success rate.

Our approach of using the architecture of the project as well as the development practice to suggest tasks based on the artifacts and their relationships turned out to be very versatile. Not only can it deal with new tasks, it also supports run-time changes to the architecture. The cost of task suggestion is a one-time description of the architecture with our tool's rules.

Our Plugin is most promising for projects that value promoting compliance, such as student project with a predefined structure, and enterprise projects following a predefined practice.

Our main purpose of fostering the compliance with the architecture and the development practice was successfully achieved. In the next section we look into how we could further iterate over the activity context, and further increase compliance.

6.1 Future Work

In this section we discuss features that are worth expanding upon.

6.1.1 Scope Extension - Delving into code level

Currently, upon activating a task, its produced artifacts are automatically created in their default package. The creation of an artifact results in an empty artifact with a custom name, a custom extension, and a custom location. Being able to also customize what the artifact's initial code should be would further foster the compliance with the architecture. There are multiple ways this feature could be implemented, with either metaprogramming or simple file edition as the output, and either a Plugin view, another file, or the creation of a domain language as input.

Creating the content of an artifact is just the first step. Then we can gain knowledge of the actual state of an artifact. These two features paired with an improved listener to artifact changes would close to completely automate the functions of the Plugin on the Activity View, and skyrocket compliance with both the architecture and the development practice.

Having knowledge of an artifact at the code level, we can give more value to the data than the dependencies when choosing if a task should be reactivated or not. While we still cannot be sure that an artifact with no errors has to be modified, having knowledge of its content such as fields and method calls can help guide reactivation.

During the user tests several professional users suggested integrating test automation. We also think that this feature is worth looking into.

6.1.2 Scope Shift - Information Flow, Collaboration, Gamification

We chose to implement our tool as an Eclipse Plugin and are satisfied with the choice. Today's software development pipeline can be very large and drastically different for each product. An expansion of the Plugin would only make sense if it could have an overview of the pipeline and manage the flow of information. We are talking about a completely different scale, as it would require either integration with several other tools or modifications on both sides in order to standardize inputs and outputs. We would more than foster compliance with the architecture and development practice, foster compliance with the whole production process.

As we focused our main research question on the activity context and fostering compliance, collaboration became more of a bonus instead of a main feature. While we do allow collaboration through GitHub with the sharing of goal, task and artifact status, we feel that there is a lot that can be done in this context. The main features to look into on this front would be automatic task attribution, context sharing and a custom User View on the Plugin with statistics and member information.

Once again, we also have a lot of interest in gamification that stopped being a priority as the scope was refined. As the reader might have noticed we still have a reward value for completing artifacts. We also feel that there is a lot that can be done in this context, starting from UI modifications for reward

information on task activation and reward acquisition on task completion, to eventually gamifying the activity workflow itself where the user is the player, the task is the user's quest, and the activity is a NPC with the metadata template as a personality. We feel that by applying a game metaphor to the Plugin, it would also increase motivation, and with it, performance.

Bibliography

- [1] B. Schmidt and U. V. Riss, “Task patterns as means to experience sharing,” in *International Conference on Web-Based Learning*. Springer, 2009, pp. 353–362.
- [2] C. A. Thompson, G. C. Murphy, M. Palyart, and M. Gašparic, “How software developers use work breakdown relationships in issue repositories,” in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 281–285.
- [3] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. Van Der Hoek, “Microtask programming: Building software with a crowd,” in *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 2014, pp. 43–54.
- [4] B. Schmidt and W. Reinhardt, “Task patterns to support task-centric social software engineering,” in *Proceedings of the rd Workshop Social Information Retrieval for Technology-Enhanced Learning (SIRTEL) at the th International Conference on Web-based Learning (ICWL)*, 2009.
- [5] M. Kersten and G. C. Murphy, “Using task context to improve programmer productivity,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 1–11.
- [6] C. A. Thompson, “Towards generation of software development tasks,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 915–918.
- [7] M. Kersten and G. C. Murphy, “Reducing friction for knowledge workers with task context,” *AI Magazine*, vol. 36, no. 2, pp. 33–41, 2015.
- [8] —, “Task context for knowledge workers,” in *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.
- [9] W. Maalej, M. Ellmann, and R. Robbes, “Using contexts similarity to predict relationships between tasks,” *Journal of Systems and Software*, vol. 128, pp. 267–284, 2017.

- [10] G. C. Murphy, M. Kersten, R. Elves, and N. Bryan, “Enabling productive software development by improving information flow,” in *Rethinking Productivity in Software Engineering*. Springer, 2019, pp. 281–292.
- [11] C. Mayr-Dorn and A. Egyed, “Does the propagation of artifact changes across tasks reflect work dependencies?” in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 397–407.



Code of Project

Here is the code for the method in the TaskSuggester class responsible for creating the new tasks for the current activity based on the current state of the project. The description of this method is made at the end of Chapter 4. The full code of the project is accessible at <https://github.com/socialsoftware/archlyn>.

Listing A.1: Method suggestNew() in the TaskSuggester class.

```
1 public List<Task> suggestNew() {
2     ArrayList<Task> tasks = new ArrayList<Task>();
3     ArrayList<Dependency> dependencies = new ArrayList<Dependency>();
4     ArrayList<Artifact> used = new ArrayList<Artifact>();
5     ArrayList<Artifact> produced = new ArrayList<Artifact>();
6     if(!Share.status.getActiveActivity().getTasks().isEmpty()) {
7         for (Artifact a: this.getArtifactsInTaskContext(
8             Share.status.getActiveActivity().getTasks())) {
9             System.out.println("Origin " + a);
10            Boolean foundTask = false;
```

```

10     for (TaskType tt: Share.status.getTaskTypes()) {
11         System.out.println("TT " + tt);
12         foundTask = false;
13         for (ArtifactType atu: tt.getUsed()) {
14             if (atu.getType().equals(a.getType().getType())) {
15                 for (Task t: Share.status.getActiveActivity().getTasks())
16                 {
17                     System.out.println("T" + t);
18                     if (t.getTaskType().getDescription().equals(
19                         tt.getDescription())) {
20                         for (Artifact au: t.getUsed()) {
21                             if (au.getPath().equals(a.getPath())) {
22                                 System.out.println("found T");
23                                 foundTask = true;
24                             }
25                         }
26                     }
27                 }
28             }
29         }
30         if (!foundTask) {
31             System.out.println("not found T");
32             for (ArtifactType at: tt.getUsed()) {
33                 if (at.getType().equals(a.getType().getType())) {
34                     used.add(a);
35                 }
36                 else {
37                     for (Artifact ua:
38                         this.getArtifactsInTaskContext(
39                             Share.status.getActiveActivity().getTasks
40                             ())) {
41                         System.out.println(at + " " + ua.getType())
42                             ;
43                         if (ua.getType().getType().equals(
44                             at.getType()) && ua.getComplete()) {
45                             System.out.println("adding " + ua);
46                             used.add(ua);
47                             break;
48                         }
49                     }
50                 }
51             }
52         }
53     }

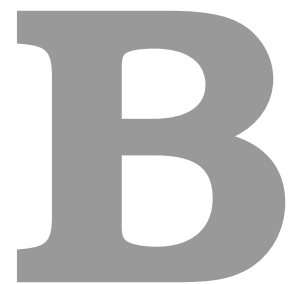
```

```

41         }
42     }
43     System.out.println("check size " + Arrays.toString(
        used.toArray()));
44     if (used.size() == tt.getDependencies().size()) {
45         System.out.println(Arrays.toString(used.toArray()
        ));
46         Task toAdd = new Task(new ArrayList<Artifact>(
            used), new ArrayList<Artifact>(produced), tt,
            descriptionNumber+": "+tt.getProduced().get
                (0).getType()+" - " + tt.getDescription());
47         descriptionNumber++;
48         toAdd.setStatus("Available");
49         tasks.add(toAdd);
50         used.clear();
51         produced.clear();
52         dependencies.clear();
53         break;
54     }
55 }
56 used.clear();
57 produced.clear();
58 dependencies.clear();
59 }
60 }
61 }
62 }
63 }
64 else {
65     for(TaskType tt: Share.status.getTaskTypes()) {
66         if(tt.getUsed().get(0).getType().equals("Start")) {
67             used.add(Share.status.getArtifacts().get(0));
68             Task toAdd = new Task(new ArrayList<Artifact>(used), new ArrayList
                <Artifact>(produced), tt, descriptionNumber+": "+tt.getProduced
                    ().get(0).getType()+" - " + tt.getDescription());
69             descriptionNumber++;
70             toAdd.setStatus("Available");
71             tasks.add(toAdd);

```

```
72         break;
73     }
74 }
75     used.clear();
76     produced.clear();
77     dependencies.clear();
78 }
79     return tasks;
80 }
```



User Test Guide

Here is the User Test Guide used throughout every user test. The User Test Guide can be summarized as follows:

- Activity 1: Learning (Guided) - Steps 1.1 through 5.6
- Create structure for activity 2 (Guided) - Steps 6.1 through 6.7
- Activity 2: Complete activity alone (using structure from section 2) - Steps 7.1 through 7.2
- Activity 3: Perform change in Activity 1 and re-complete activity alone - Steps 8.1 through 8.2

Test Guide

In the context of my thesis I developed a plugin for Eclipse IDE, with the main objective of using structured activities (groups of tasks) in order to foster compliance with the software architecture and development practice of a project, and in that way improve the developer's productivity. The plugin is split in two views. In the Activity view (Fig. 1) we work on the tasks suggested to us. In the Design view (Fig. 7) we define the rules that represent the architecture and development practice, and in that way, structure the activity and allow task suggestion.

Today we will work on a project that already has its rules set in the Design view. The first task in our activity is the creation of a story. I will now give you the story.

1.1 – Please read the story and express any doubts.

For each task we will go through a series of steps. We activate the task, the artifacts are automatically created, we move and rename them, we write the code, we complete the task. Then we activate the next available task and repeat. The process will continue in this fashion until every task is complete.

1.2- In the Activity view, double click on the available task ("0: Story – Create a story for the project") in order to activate it (Fig. 1).

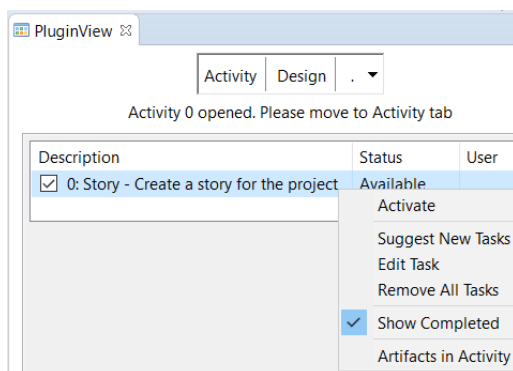


Figure 1 - Activity View

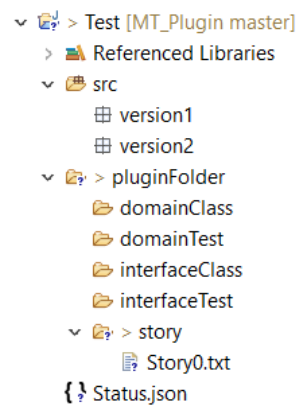


Figure 2 - Package Explorer

Activating the task brings up the Task view (Fig. 3). In the task view we have a table with the artifacts that we use and a table with the artifacts that we produce. The artifacts are automatically created in the "pluginFolder" on the Package Explorer (Fig. 2).

1.3 – Drag and drop the created artifact ("Story0.txt" in "pluginFolder\story") into "src\version1" (Fig. 2).

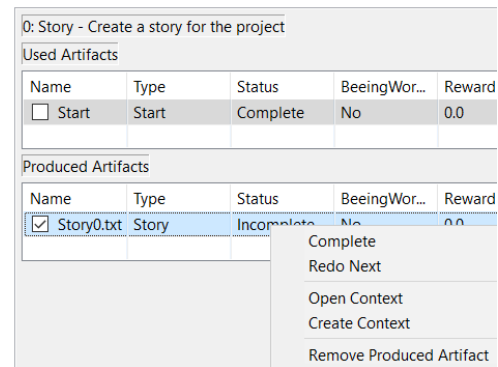
I will provide the code which I ask you to once again read and express any doubts.

1.4 – Rename the created artifact ("Story0.txt") to its name in the story ("StoryHotel"). You can do this by right clicking the artifact in the package explorer and selecting "Refactor\Rename".

We are now finished working on this task's artifact. We will now set our task as complete. In the activity view we will then see a newly suggested task and activate it

1.5 – In the Task view right click anywhere on the table area to bring up the menu. Select the “Complete” option in the menu in order to complete the task (Fig. 3).

We will now repeat steps 1 through 5 for each of the new tasks until there are no more available tasks in the Activity view. Test artifacts have an additional step.



0: Story - Create a story for the project				
Used Artifacts				
Name	Type	Status	BeeingWor...	Reward
<input type="checkbox"/> Start	Start	Complete	No	0.0
Produced Artifacts				
Name	Type	Status	BeeingWor...	Reward
<input checked="" type="checkbox"/> Story0.txt	Story	Incomplete	No	0.0

Figure 3 - Task View

2.1 – Please read the “**DomainClass**” part of the story if needed and express any doubts.

2.2 - In the Activity view, double click on the available task (“1: DomainClass – Create a domain class for the story”) in order to activate it.

2.3 – Drag and drop the created artifact (“DomainClass1.java” in “pluginFolder\domainClass”) into “src\version1”.

I will provide the code which I ask you to once again read and express any doubts.

2.4 – Rename the created artifact (“DomainClass1.java”) to its name in the story (“Hotel”). You can do this by right clicking the artifact in the package explorer and selecting “Refactor\Rename”.

2.5 – In the Task view right click anywhere on the table area to bring up the menu. Select the “Complete” option in the menu in order to complete the task.

3.1 – Please read the “**DomainTest**” part of the story if needed and express any doubts.

3.2 - In the Activity view, double click on the available task (“3: DomainTest – Create a domain test for the domain class”) in order to activate it.

3.3 – Drag and drop the created artifact (“DomainTest2.java” in “pluginFolder\domainTest”) into “src\version1”.

I will provide the code which I ask you to once again read and express any doubts.

3.4 – Rename the created artifact (“DomainTest2.java”) to its name in the story (“HotelTest”). You can do this by right clicking the artifact in the package explorer and selecting “Refactor\Rename”.

3.5 – **Additional step:** In HotelTest.java, hover “@Test” and click on “Add JUnit 5 library to the build path” (Fig. 4). Then right click and select “Run As\JUnit Test” to make sure the test passes.

3.6 – In the Task view right click anywhere on the table area to bring up the menu. Select the “Complete” option in the menu in order to complete the task.

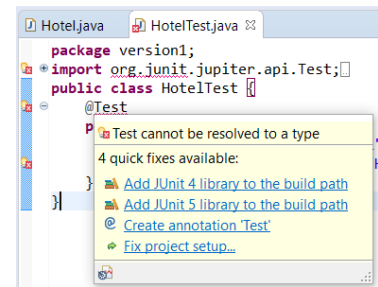


Figure 4 - Add JUnit to build path

4.1 – Please read the “**InterfaceClass**” part of the story if needed and express any doubts.

4.2 - In the Activity view, double click on the available task (“4: InterfaceClass – Create an interface class for the domain test”) in order to activate it.

4.3 – Drag and drop the created artifact (“InterfaceClass3.java” in “pluginFolder\interfaceClass”) into “src\version1”.

I will provide the code which I ask you to once again read and express any doubts.

4.4 – Rename the created artifact (“InterfaceClass3.java”) to its name in the story (“HotelInterface”). You can do this by right clicking the artifact in the package explorer and selecting “Refactor\Rename”.

4.5 – In the Task view right click anywhere on the table area to bring up the menu. Select the “Complete” option in the menu in order to complete the task.

5.1 – Please read the “**InterfaceTest**” part of the story if needed and express any doubts.

5.2 - In the Activity view, double click on the available task (“5: InterfaceTest – Create an interface test for the interface class”) in order to activate it.

5.3 – Drag and drop the created artifact (“InterfaceTest4.java” in “pluginFolder\interfaceTest”) into “src\version1”.

I will provide the code which I ask you to once again read and express any doubts.

5.4 – Rename the created artifact (“InterfaceTest4.java”) to its name in the story (“HotelInterfaceTest”). You can do this by right clicking the artifact in the package explorer and selecting “Refactor\Rename”.

5.5 – **Additional step:** In HotelInterfaceTest.java, right click and select “Run As\JUnit Test” to make sure the test passes.

5.6 – In the Task view right click anywhere on the table area to bring up the menu. Select the “Complete” option in the menu in order to complete the task.

Our activity is now complete as all our tasks are complete and there are no new tasks suggested. We will now go do another activity, but this time we want it to be different. We don't want tests in this activity so we will go into the Design view and change the rules accordingly.

6.1 - On the plugin's top bar left click on "Design" and then on the tabs "Metadata\DependencyTypes".

6.2 - Right click the forth dependency type, the one with "DTInterface" as Dependency, and select "Edit Dependency Type" (Fig. 5). Select "DomainClass" as the "Origin Artifact Type". Select "InterfaceClass" as the "Target Artifact Type". Press "OK" (Fig.6).

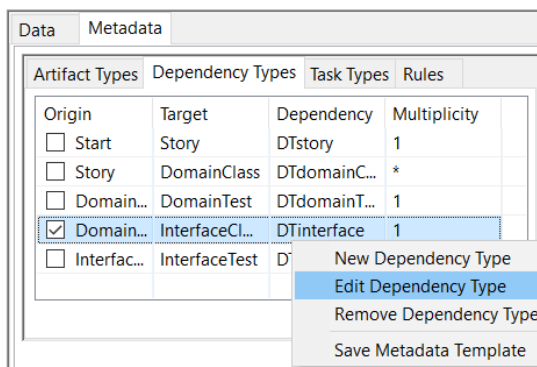


Figure 5 - Design view (Dependency Types)

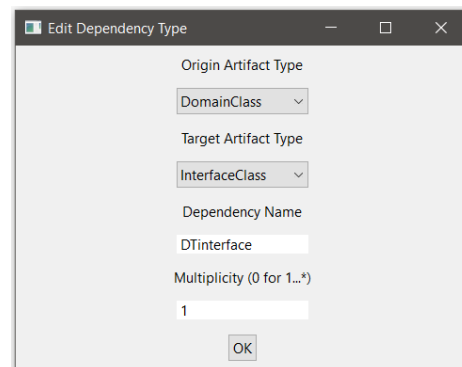


Figure 6 - Edit Dependency Type

In this step we made it so the creation of a "DomainClass" artifact leads to the creation of a "InterfaceClass" artifact, skipping the "DomainTest". We will now remove the task types corresponding to the creation of "DomainTest" and "InterfaceTest" in the Task Type tab.

6.3 - Left click on the "Task Types" tab. Select the three bottom task types, right click and select "Remove Task Type" (Fig. 7).

6.4 - Right click anywhere on the table and select "New Task Type" (Fig. 8). Select "DTInterface" in the first combo box and click on "Add Dependency Type". Write "Create an interface class for the domain class" as the Description. Press "OK".

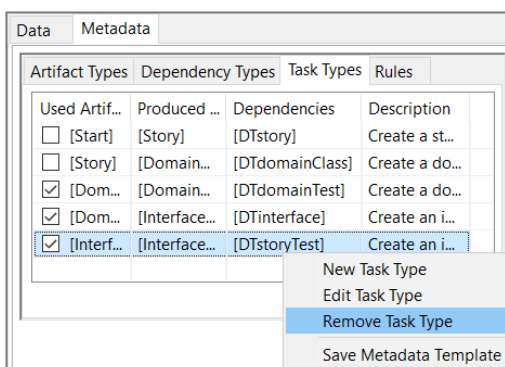


Figure 7 - Design View (Task Types)

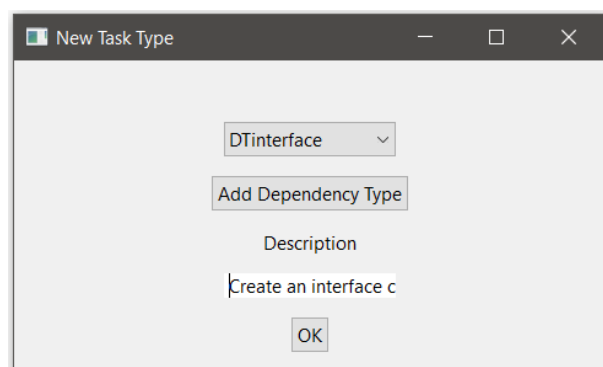


Figure 8 - New Task Type

Now that we have our new rules we are going to save our metadata template and attribute it to our next activity. We will now edit our activity names for them to be clearer.

6.5 – Right click anywhere on the table and select “Save Metadata Template”. Name it as “HotelNoTests”. Press “OK”.

6.6 - Click on the main menu in the rightmost button on the top bar (Fig. 9). Select “Edit Activity”. Choose the current activity (0). Rename it to “ActivityHotel”. Set the Type to “Validation” (Fig. 10). Press “OK”.

6.7 - Click on the main menu in the rightmost button on the top bar. Select “Edit Activity”. Choose the next activity (1). Rename it to “ActivityHotelRoom”. Set the Type to “HotelNoTests”. Press “OK”.

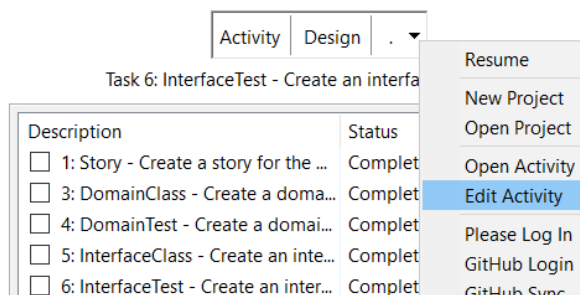


Figure 7 - Main Menu

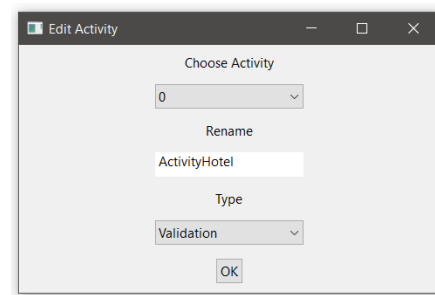


Figure 8 - Edit Activity

We are now ready to begin our new activity. Now I will give you a new story. The difference from the previous story is that the Hotel now has a number of rooms. This change is reflected throughout every class. Please follow the story as we did before and complete the activity on your own. Move the artifacts into “src\version2”.

7.1 – Click on the main menu in the rightmost button on the top bar. Select “Open Activity”. Select “ActivityHotelRoom”. Press “OK”.

7.2 – Follow the new story and complete the activity. The ending result is on Fig. 11 and Fig. 12.

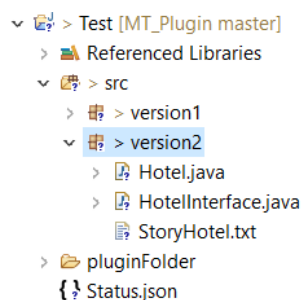


Figure 9 - Activity 2 Artifact Result

Description	Status
<input type="checkbox"/> 2: Story - Create a story for the project	Complete
<input type="checkbox"/> 7: DomainClass - Create a domain clas...	Complete
<input type="checkbox"/> 9: InterfaceClass - Create an interface ...	Complete

Figure 12 - Activity 2 Task Result

Now that our new activity is complete we will revisit the first activity. There you will, once again on your own, make the modifications necessary in order to change the original story into our new story. In this activity we have tests so those will require change too.

8.1 - Click on the main menu in the rightmost button on the top bar. Select "Open Activity". Select "ActivityHotel". Press "OK".

8.2 – Change to old story into the new one (Fig. 13). Complete the activity. The activity is complete once every task has the "Complete" status.

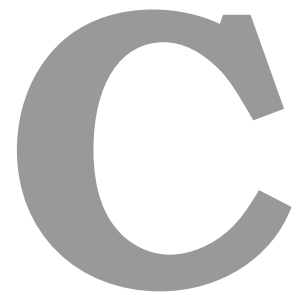
Hint – Changing an artifact in the activity sets tasks that use that artifact to the status "To Redo" (Fig. 13). You might want to activate the first task with "To Redo" status, change it's produced artifact, complete it and activate the new task with "To Redo" status if there is one.

Description	Status
<input type="checkbox"/> 16: Story - Create a story for the project	Active
<input type="checkbox"/> 17: DomainClass - Create a domain cl...	To Redo
<input type="checkbox"/> 19: DomainTest - Create a domain test...	Complete
<input type="checkbox"/> 20: InterfaceClass - Create an interface...	Complete
<input type="checkbox"/> 21: InterfaceTest - Create an interface ...	Complete

Figure 10 - Activity View (Task in "To Redo" status)

As both our activities are finished, the testing is now complete.

Thank you very much for your participation.



Questionnaire Results

Here are the results obtained from the questionnaire filled by each user after his user test. Each graph represents the results from each of the 18 questions. Each answer has a column for students in blue, and a column for professionals in orange.

