Collaborative software development: from goals to coding

Pedro Monteiro, *IST*

Abstract—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 process are used to automatically break down and connect the work as a set of tasks and increase productivity.

Index Terms—Social Software Engineering Task-Centric Task Context Activity Context.

1 Introduction

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 sucessors, 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.
Our research question is:

- 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 developers 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.
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. 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.
During user testing, we achieved 100% compliance with the software architecture and 70% compliance with the development practice. The low standard deviation in performance shown in the user tests supplements the benefits of our tool.
The structure of this paper is as follows:

- Related Work - Study of Task Context related systems and other task-management approaches
- Design - Introduction of the base concepts
- Implementation - Description of the implemented tool's features
- Evaluation - Testing the tool through several development practices
- Conclusion - Review of our project, its contributions and future.

2 Related Work

2.1 Task Context

A task-centric software development tool, MyLyn for Eclipse IDE, uses Task Context in order to improve productivity [1], [2], [3]. 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.

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

Task Patterns presented in [7] [8], extend the scope of task-centric software development tools, such as MyLyn for Eclipse IDE (Tasktop Pro extension) by collecting the data from a user's task and organizing it in a Task Pattern. 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.

2.3 Work breakdown

2.3.1 Dependencies

The authors of [10], identified how issue relationships are used. The analysis of titles of issues from open source repositories resulted in several codes describing the dependencies as a work breakdown relationship. 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.

2.3.2 Change Propagation

Paper [6] studies change propagation, which is the practice of developers accessing artifacts in dependent tasks while working on their own. 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

The authors of [4] have implemented a microtask workflow in CrowdCode, a cloud IDE for crowd development, limited to crowdsourcing small functional libraries. 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.

2.4 Automatic Task Suggestion

Paper [5] investigates how context data, extracted from developer interactions with development artifacts, can be used to predict relationships between tasks. Paper [9] also supports the use of task suggestion to increase the quality and productivity of software development. 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.

3 Design

By analysing other task-centric tools we identified several problems. 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.

In order to define our tool, we first defined a model. The main concepts that make up our model can be seen in Figure 1.

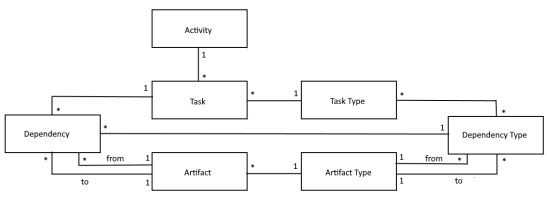

Figure 1: Tool Model

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. In our concept the architecture is then represented as-is.

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, top-down.

An activity is defined by a set of tasks. An activity can be seen as an instance of the project's software architecture following a practice, such as the creation of a single story to the deployment of its respective code in a Bottom-Up practice. The activity's 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 self-contained, meaning that tasks in another activity won't be modified by changes to the artifacts in the current activity. There are two main reasons on why it is important that activities are self-contained. The first one is that once an activity is finished it is unlikely that the user wants its tasks to be automatically reactivated by outside changes. As many tasks will find themselves sharing artifacts it would eventually become unmanageable for the user to keep track of the relevance of the reactivations in different activities. The second one is that we are focused on keeping the context contained and the work in sequential contexts. That said, if the user feels like a change that reactivated a task in the current activity should reactivate a task in another activity he can manually reactivate it.

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.

We abstract tasks into task types. Task types can be seen as templates of dependency types. Just as we previously mentioned 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 partially 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 statuses a task can have are "Available", "Active", "Complete" and "To Redo". 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 tasks status change into "To Redo".

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 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.

3.1 Workflow

There are three different ways to use our tool:

- Workflow A - The developer works on an already supported project.
- Workflow B - The developer creates his own rules for a not yet supported new project
- Workflow 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 2.

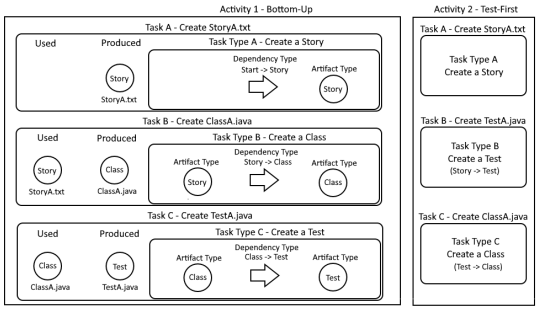

Figure 2: Workflow A Example

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 writes the story and upon task completion the task suggester 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 (B).
In this case the developer would first create the artifact types. Still following Figure 2, 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 Workflow A.
The case of an already existing not yet supported project (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

Workflow A.

4 Implementation

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.
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. These two views are accessible through the top bar seen on Figure 3 and are accompanied by the main menu. Each view contains a table with the elements related to that view, and a menu accessible through right click to perform actions on the selected elements, or other functions.

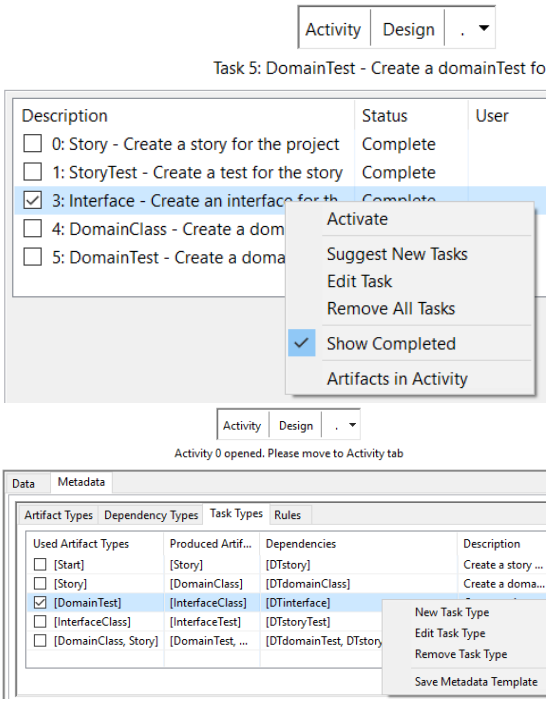

Figure 3: Activity/Design View

4.1 Workflow A - Task Execution

The usual workflow for a developer working on a supported project will be contained to the Activity View, and the Task View accessible through it. In the main menu we can open the project and select the activity that we want to work on, which is applied to the Activity View. In this menu we can also edit activities to change their name and their metadata template. 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 tasks purpose, the main product of the task and the sequence in the activity.
The most important function of the Activity View is to activate tasks, which sets the task status to "Active", opens the Task View, creates the tasks context and opens it in the IDEs editor. Another relevant function is showing the context of the whole activity in a new shell, 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 editor. Other functions include editing the task's name and status, and filtering the table.

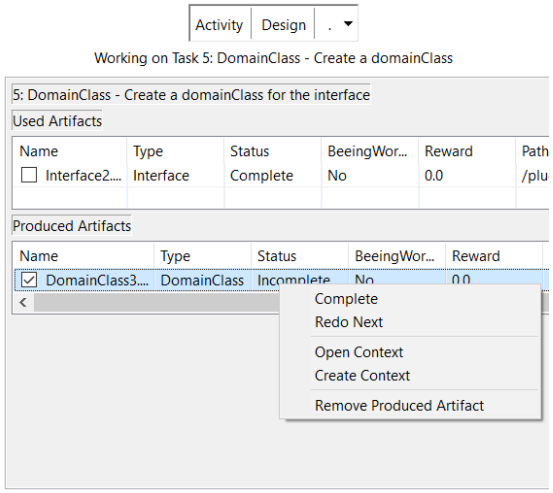

Figure 4: Task View

The Task view on Figure 4 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 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 and its path. The Task Views menu most relevant options are "Complete", "Open Context" and "Remove Produced Artifact". The "Complete" option changes the task status to "Complete" if all its dependencies are realized, meaning that if the tasks 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. After changing the active tasks status, the Activity View is reopened, and new tasks are suggested if available. 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 “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 execution 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.

4.2 Workflow B - Metadata Template

Before we go into the Design View, we need to create the project with the “New Project” option of the main menu. A project is created given a name and a project type. The type refers to a hard coded template that comprises a data template and a metadata template.
We start by creating the artifact types that we will be dealing with in the Artifact Type View in the metadata section of the Design View. A new artifact type is created given a name and a file extension for the automatic creation of the files. We also have the options to edit and remove artifact types.
Having the artifact types, we can now describe the relationships between them in the Dependency Type View. A new dependency type is created given the origin artifact type, the target artifact type, a name and a multiplicity. It is important to state different multiplicities because a product artifact might be dependent on more than one artifact of the same type. We also have the options to edit and remove dependency types.
Having dependency types, we can now group them in a task type in the Task Type View. A new task type is created given a description and the containing dependency types, which result in the used artifact types and produced artifact types. We also have the options to edit and remove task types.
The last view in the metadata section is the Rules View. The only currently available rule is “AddStatusInProgress” which forces tasks to remain in the “In Progress” status after completion. This is 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 view in the metadata section, by giving it a name. All there is left to do is switch to the Activity View.

4.3 Workflow C - Data Template

An already existing not yet supported project means that it has rules that the Plugin does not understand, and so the developer needs to define these rules in the Plugin.
The Package 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. 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.
Setting a type on a package, also sets the type of its containing artifacts to that of the package. We can also use the Artifact View to individually 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 and path, 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. We can edit an artifact’s type and status, as well as removing the artifact information from the Plugin. Often the Plugin will refresh the packages and artifacts by scanning the project folder for changes or receiving resource change events. The refresh can also be done manually for immediate situations.
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 we can establish the relationships between our artifacts. A new dependency is created given the origin artifact, the target artifact, and the dependency type. We also have the options to edit and remove dependencies.
With 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.

5 Evaluation

We decided to focus the evaluation on answering our main research question. 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. In every activity we expect the user to comply with the architecture and the development practice. We also track the execution time of each task. The User Test Guide 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

Note: In this section we may refer to "Activity x" as "Ax" and "Task x" as "Tx".
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. Our tasks 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 Plugins 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).

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.
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. The architecture is now made of a "Story", a "DomainClass", and an "InterfaceClass", as we removed the tasks for the creation of tests (3 and 5)
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 activitys story artifact with the one from Activity 2. 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.

5.1 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."

- 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 codding 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.

5.2 Results

We performed 20 user tests, where all the tests were performed following the User Test Guide. We tested our tool with 9 students and 11 professionals. Each user test took an average of 20 minutes working in tasks, plus the time of reading the guide and answering the questionnaire. 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. The overall average age is 28 years old. There was only one female user. We also performed 6 user tests without the Plugin. Each user test took an average of 13 minutes.

5.3 Compliance

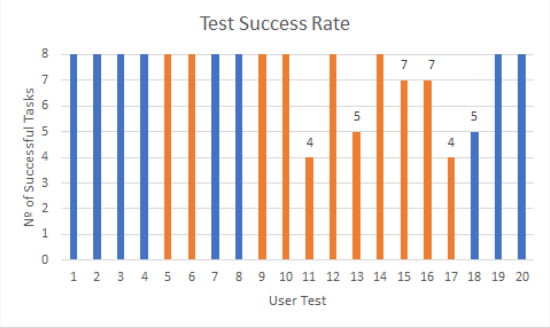

Figure 5: Test Success Rate

Overall, we had 100% compliance with the architecture in every test, 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.
Overall, we had 70% user test success rate, 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 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. There is a 34% difference in overall user test success rate between the student users and the professional users, favoring the students. 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, as the number of tasks that were completed successfully at the right sequence was 144 out of 160, with 16 failed tasks. On Figure **??** we can see the success rate per task. 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. There is a 11% difference in overall task success rate between the student users and the professional users, favoring the students. Without the Plugin we had 53% task success rate with 19 out of 36 tasks completed successfully.

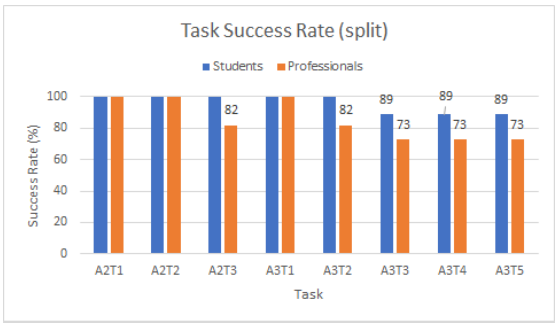

Figure 6: Task Success Rate (split)

Figure 7: Task 2 Performance

5.4 Performance

By analysing the average time taken to complete tasks we concluded that all users took a similar amount of time to use the Plugin. Without using the Plugin, tasks involving coding took an average of 50 more seconds. Figure 7 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 codding effort. Even still there are only two outliers, with every other result being between the boxs 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 boxs 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.

5.5 Identified Problems and Solution

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:

- Users much prefer going through the test by talking instead of reading the guide. Every test should aim to be realized under the same conditions, and so we strived to do so.
- We did not introduce the "Open Context" feature. It would've been helpful because there was often more than one artifact with the same name in the editor, and if the user was not sure what to do, he could simply reopen the current context.
- We did not explain the task reactivation functionality properly. A clear explanation could have increased the success rate in Activity 3.

The **Plugin** has three main issues:

- With an active task every edited artifact is added to the produced artifacts. An artifact where Eclipse marked an error counts as an edited artifact. It was counter-intuitive to add 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.
- Due to the way tables and menus are utilized, it was often not obvious for the user what the functionality was being called on. That is, for example, if the "Complete" option in the Task View refers to the task itself, or the artifact, because the menu is called over the artifact.
- 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.

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:

- Some users claimed to be used to, when seeing an error, going to fix it immediately. This does not bode well for our current implementation. 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.
- We aimed the test time to 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

A questionnaire was answered at the end of the user test with questions related to the Plugin, for measuring usability, and assessing performance. Overall the results of the questionnaire were positive. There was no question were the average or mode went against the ideal answer. The question that was closest to a negative result, and the only one with a relevant difference between the user types, was question 8: "The Plugin restricted my freedom of development". At least five users felt like the compliance with the development practice was fostered at the cost of freedom of development. The professionals answer was almost split in half, so work experience such as other tools used or current workflow might result in the difference in opinions. We are not satisfied with this result as we don't believe this to be the case, so 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.

6 Conclusion

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 developers 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, 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 them implemented our tool as an Eclipse Plugin were 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:

- Workflow A - The developer works on an already supported project.
- Workflow B - The developer creates his own rules for a not yet supported new project
- Workflow 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 achieve 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. 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. On average all the answers to the questionaire 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 tools 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.

7 Future Work

- A - Delve into code level: Customize an artifact's code.
- B - Maintain an artifact's actual state (possible because of A)
- C - Improve artifact change listener
- D - Test automation
- E - Expand into managing the flow of information of the production pipeline.
- F - Context sharing and automatic task attribution
- G - Gamify the Activity workflow

Functionalities A, B, C and D together would completely automate the functions of the Plugin on the Activity View, and skyrocket compliance with both the architecture and the development practice. In E 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 and gamification became more of a bonus instead of main features. 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. We feel that by applying a game metaphor to the Plugin, it would also increase motivation, and with it, performance.

References

[1] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11. ACM, 2006.

[2] Mik Kersten and Gail C Murphy. Task context for knowledge workers. In *Proc. AAAI 2012 Activity Context Representation workshop*, 2012.

[3] Mik Kersten and Gail C Murphy. Reducing friction for knowledge workers with task context. *AI Magazine*, 36(2):33–41, 2015.

[4] Thomas D LaToza, W Ben Towne, Christian M Adriano, and Andr Van Der Hoek. Microtask programming: Building software with a crowd.

[5] Walid Maalej, Mathias Ellmann, and Romain Robbes. Using contexts similarity to predict relationships between tasks. *Journal of Systems and Software*, 128:267–284, 2017.

[6] Christoph Mayr-Dorn and Alexander Egyed. Does the propagation of artifact changes across tasks reflect work dependencies? In *Proceedings of the 40th International Conference on Software Engineering*, pages 397–407. ACM, 2018.

[7] Benedikt Schmidt and Wolfgang 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.

[8] Benedikt Schmidt and Uwe V Riss. Task patterns as means to experience sharing. In *International Conference on Web-Based Learning*, pages 353–362. Springer, 2009.

[9] C Albert Thompson. Towards generation of software development tasks. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 915–918. IEEE Press, 2015.

[10] C Albert Thompson, Gail C Murphy, Marc Palyart, and Marko Gaparic. How software developers use work breakdown relationships in issue repositories. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 281–285. ACM, 2016.