Activity Diagram as an Orientation Catalyst within Source Code

Ján Lang, Dávid Spišák

Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 2, 842 16 Bratislava Slovakia
{jan.lang, qspisakd}@stuba.sk

Abstract: There is a premise that the activity diagrams can communicate their knowledge to the source code. This article analyzes the opportunity of the activity diagrams to improve the comprehensibility, orientation, reading, and modularization of the source code. It proposes an Activity Diagram Driven Approach (ADDA) and verifies application suitability of the approach in comparison to Use Case and Package-based approach. It highlights the strengths and weaknesses of such behavior description and discusses the identified limits and benefits of the proposed approach. It proposes an extension of the source code modularization at metamodel level based on source code parts associated with certain elements of the activity diagram. The proposed solution is evaluated over several test cases from different aspects using implemented plug-in and the results show appropriate use of the proposed approach.

Keywords: UML; activity diagram; source code; orientation; comprehensibility; modularization

1 Introduction

Initial analysis of business processes often leads to a set of activity diagrams (AD) that describe how the work is carried out within the organization. These diagrams may contain so-called swimlines which visually distinguish contribution to the implementation of business processes and responsibility for implementation of sub-activities throughout the process. Implicitly, within these diagrams, there is documentation of explicit links between the actions and roles, players, i.e., the users. Such a depicted business process can gain representation in the form of software—to a human somehow readable form of a source code. Unlike behavior diagrams, source code is rather difficult to read and highly technical in nature with almost no business information. It is difficult to mentally grasp even related source code parts of the project with respect to its complexity and scope.

This problem met a resolution in the decomposition of a complex solution into smaller units [1] or packages [10] etc. Such a decomposition or in other words, modularization may carry into the software a considerable set of implications. One of them is the issue of preserving the intent in the source code [7], low cohesion and high coupling of the source code [8], [29] its readability [3] and comprehensibility or simply orientation at all. Preserving or grabbing source code intent is not only a problem of reading an unknown author's source code but even of its own ones, especially after some time. Actually, even the author himself has to make a lot of effort to get in his own source code after a while. To comprehend a certain business processes, a more or less complex ones already available in the form of source code, would not be so simple. One could feel the need to get rid of unnecessary details, to abstract or to see things from a higher perspective.

It is known that the system maintenance consumes approximately 70 percent of the total cost of the software product [13]. The use and benefits of the information stored in the behavioral diagrams and their mapping to the source code is confirmed by developers themselves when up to 17 of 19 developers would welcome a tool to help them navigate in the source code, especially if they do not know the source code [21]. The same developers have suggested that the error rate could be significantly reduced because the diagrams provide a better overview of what the developer may currently work on. This paper presents the design and implementation of the Activity Diagram Driven Approach (ADDA) and source code mapping, i.e., organizing source code into a structure based on activity diagrams in order to achieve better modularization, readability, comprehensibility, and orientation in the source code as such. The work is divided into several sections. Section 2 refers to the core principle of the activity diagram-driven source code modularization. Section 3 provides consideration in the context of the activity diagram's metamodel and its extension. Section 4 includes an extensive evaluation and reflects on related work.

2 Related Work

The proposed approach ADDA can be confronted across all phases of software development. One can come across activity diagrams within a workflow modeling during the requirements elicitation phase [14]. At this stage, we usually do not have any source code except for existing projects and for example specifications in the form of Changelog. Further analysis and design may only refine the identified model of the upcoming system. Connection with the implementation phase of software development is found in the study of the coverage degree of the source code by the activity diagram [19]. They proved that it is possible to generate source code not only from structural class diagrams but also from behavioral diagrams namely UML AD. To achieve automatic generation L. Jim and P. Klint had to define relatively complex stereotypes and limitations.

According to L. S. Jim and P. Klint [19], K. Hyungchoul [20] and Heinecke et al. [17], user-acceptance tests (UAT) can also be generated from UML AD.

A similar connection between the software development phases also provides another UML artifact. M. Bystrický in his work [5], [6] and [7] presented the idea of the source code modulation from the point of view of use cases. In this approach, it represents a use case by a single file - a markdown file. This file, in addition, contains business information and source code implementing the behavior of the use case. Thus, the user finds the relevant source code in one place. This is the most fundamental difference between M. Bystrický's approach and the proposed ADDA approach. Using UML AD even UML UC in both cases, is just a way of looking at the source code - simply a certain perspective of looking. Using different views, respectively projections, in relation to the source code has also been investigated by J. Porubän and M. Nosáľ [27] in "Leveraging Program Comprehension with Concern-Oriented Source Code Projections". The authors recognized the possible need to look at the source code from different viewing angles according to the actual needs of the programmer. ADDA approach can also be considered as one of the source code projections. However, J. Porubän and M. Nosáľ did not come with UML AD in their solution, so they chose a different approach for source code projections based on annotations. However, a number of authors have dealt with the UML activity diagrams [17], [19], [21], the use of a new view at the source code e.g. from the perspective of use cases [7] even from the perspective of interrelated pieces of (multidimensional) software knowledge [34] or readability [4] comprehensibility [26], [24], [29], reusability and manageability [12], [11]. There are several types of contributors that participate in the development process. They prefer different types of perspective according to Alistair Cockburn's [9] UC levels of abstraction. Software comprehension supported by structural diagrams is also provided [18] based on measuring the time and correctness of responses. In another experiment demonstrating the need for diagramming participants desired a wide range of information contained in diagrams. They also declare the need for flexible, adaptive, and responsive diagramming tool support. Just for that reason, the prototype ADACSCO (Activity Diagram As a Catalyst of a Source Code Overview) has been deployed as an extension of the existing IDE Eclipse.

3 Activity Diagram Driven Source Code Modularization

In principle, activity diagrams are used to communicate reality or ideas about business processes [17], [25], [42]. They offer comprehensive support for the control-flow of the majority of them [30] and they are at least one level of abstraction above compared to use cases presented in Cockburn's Five Level Use

Case Model [9]. However, activity diagrams besides their textual description, in addition, visualize the process control flow. It is possible to consider the case when one business process consists of more than one use case [15], [35]. In this case, one executable node may also represent just one use case that embodies lower-level steps in the overall activity according to UML Specification. All of this only confirms a higher or equal perspective of the activity diagram's view of a model in comparison to the perspective of use cases. In some literature, this is referred to as a difference in the granularity of the view of the model [2]. It also destroys misunderstanding of the activity diagrams as just a mean for a single-use case expression [22].

As mentioned above, activity diagrams are describing execution flows called – the flow of work, workflow, working process or business process represented by actions - nodes called ActivityNodes interconnected by edges. And so, the description of the workflow consists of ActivityNodes (ControlNodes, ObjectNodes, and ExecutableNodes) and flow-of-control constructs (synchronization, decision, and concurrency) principally analogous to Petri Nets. The activity diagramming is specific for its possibility to expose other artifacts such as real object instances [31], the naming of roles that cooperate in the business process, visualization of parallel sub-processes [32], and specification of event-driven behavior [13]. The considered approach of activity diagram-driven source code modularization attempts to use explicitly created links to map the activity diagram with the source code as close as possible. Additionally, classes would not be logically grouped in a suitable way by belonging to elements or activity diagrams. Element implementation information would be scattered throughout the code. However, the benefit of the alternative with comments and tags could be the assignment at the level of methods and attributes. The second alternative is to link classes or source files directly with an activity diagram's elements so that their relationship appears in the AD action tree rather than in the source code. However, mapping classes to AD elements represents a higher granularity of association abstraction in comparison to mapping methods or attributes to AD elements and of course, it brings in certain redundancy in the form of element-irrelevant parts of the source code. But, such a form of decomposition is not in contradiction with a possible more detailed implementation.

The modularization units that would account for such a structure are the activity diagrams and the elements of the activity diagram - predominantly actions. Each element of the activity diagram respectively the activity diagram itself may have its behavior implemented by a specific part of source code. In our experiments, those specific parts will be classes. The activity diagram element to source code relevance can be visualized for example by an explicit link between the element and the class. For a simpler and more transparent view, it is possible to document these associations in the form of a tree structure, where the root of the tree is the activity diagram, its nodes are the elements, and the tree leaves represent the classes themselves. Creating these explicit links, respectively bindings allows defining the structure of the source code based on activity diagrams. Such a structure preserves a degree of business information that can help the user to orientate between the source code classes.

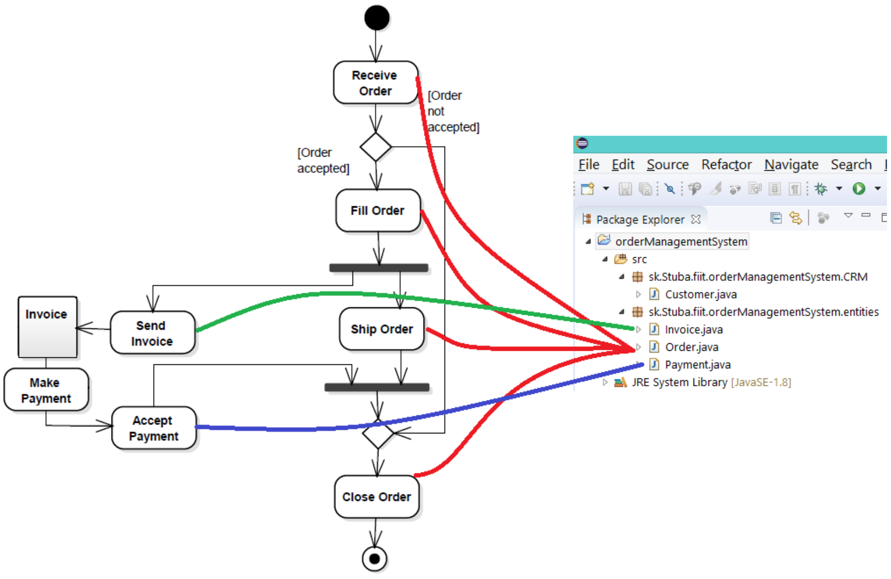

Figure 1

Associations between UML AD elements and classes in Package Explorer (PE)

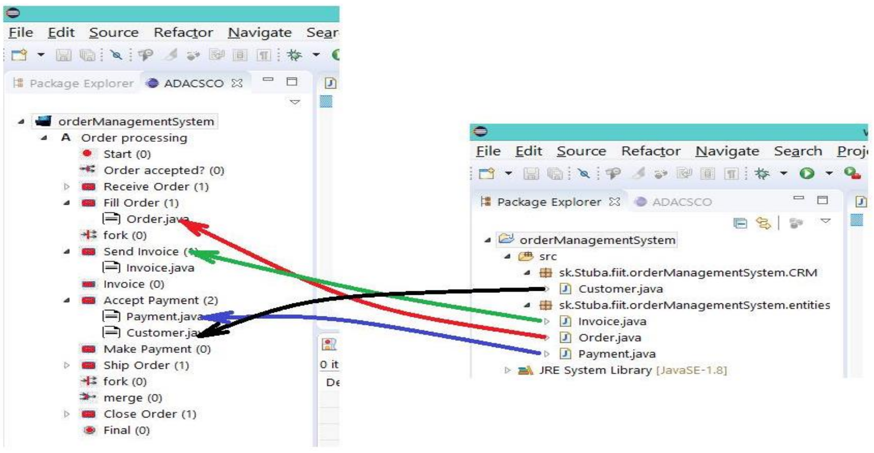

Figure 2

Tree structure organization of UML AD elements in ADACSCO plug-in

Colored lines in Figure 1 connect executable nodes of the activity diagram with classes of the Order Management System project listed in PE of the Integrated

Development Environment (IDE) Eclipse tool. The extension of the above visualization is illustrated by the Tree Structure Organization of UML AD in Figure 2 supplemented by the association orientation. The number in common brackets next to the name of each executable node expresses the number of classes participating in the implementation of the behavior behind the action. Implementation of the plug-in for IDE Eclipse seems to offer a new perspective for organizing the project - in terms of business processes. However, the principle is not only bound to the above mentioned IDE either language of the source code. Binding classes to actions assume the source code in the object-oriented paradigm. An interesting aspect is the direction of the binding. An arrow of the association goes from class to action. However, several different scenarios of modularization supported by an association AD with specific parts of the project - the source code can be considered. For the purpose of evaluation, we have created all diagrams in Enterprise Architect (EA). Interestingly, in this context, it seems to be an estimation of the burden resulting from the proposed approach. For the purpose of evaluation, prepared AD was neither difficult nor complex. It can be assumed that there is no potentially greater burden put on the developers neither in the case of enterprise projects nor projects with a higher number of AD. Hypothetically, this results from the fact that a discrete backlog task does not matter how processed (iteratively, agile. . .) will always be or should be just a subset of a particular AD. And for this purpose, each developer performs the synchronization of changes made to it for all affected repositories including the update in associations between AD elements and classes. The nature of the approach itself, as it is apparent from the description, suggests that it is a non-invasive method from the source code point of view. The above scenarios represent a way to modularize the source code driven by AD. Regarding the source code modularization, there are several ways of implementation e.g. based on packages, from the architectural point of view according to the MVC pattern or driven by use cases [7]. However, none of these directly support organizations according to the studied business processes in a complex view. All the things mentioned above especially modularization related are heading to improve orientation itself. Straight and tidy units, respectively parts of a project, create a premise for better orientation.

4 Activity Diagram Metamodel Extension

The proposed approach does not implement all of the elements for simplicity. This is also because they are not expected to be used frequently and could uselessly complicate the proposed solution and make the proposed structure less apparent. The yellow-colored element of the metamodel has been included in the alpha version of the implemented prototype. This approach proposes UML metamodel extension in the form of an association between the activity diagram element - ActivityNode and the SourceCode element by composition see Figure 3. One

ActivityNode can aggregate multiple parts of the source code. Deleting a particular activity from the activity diagram also results in an adequate response on the source code side. This synchronization is fully supported by the implemented prototype.

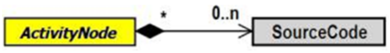

Figure 3

UML AD metamodel – Control node part. Based on OMG® Unified Modeling Language

In order to confirm or disprove all the above-mentioned assumptions even expectations including the extension of the metamodel within the ADDA approach, testing and evaluation were performed. For this purpose as the prototype ADACSCO plug-in has been implemented according to the above-mentioned specification in Java as an Open Source IDE Eclipse project.

5 Evaluation of the Proposed Approach

Orientation in UML AD is relatively simple following the instructions in the UML specification [25]. This good orientation in UML AD ensures the control of even object flow [33]. The swim lines may contribute as well. They group the elements of the activity diagram, e.g. according to the actor who performs these actions [25]. We expect these features can improve orientation in the source code. Experimental verification of these expectations is provided within the Evaluation by selected user types, Performing tasks by participants using ADDA and Multiple Users' Collaboration support.

Activity diagram driven approach proposed in this paper is expected to play the role of an imaginary catalyst that supports orientation within a source code. Let us assume that, AD is considered to be a catalyst (CA) according to a certain analogy with formal expression in chemistry, then

$$\mathbf{AD} \ \Leftrightarrow \mathbf{CA}\#(\mathbf{1})$$

AD is intended to be used in conjunction with the source code (SC) just as any other artefact, therefore

$$\mathbf{AD} + \mathbf{SC} \rightarrow \mathbf{ADSC}\ \#(\mathbf{2})$$

Of course, catalyst supported source code (ADSC) can be compiled by Compiler (C). Only the code is really compiled. In general, the catalyst does not affect the reaction balance in chemistry even in this analogous case for the source code compilation. So AD does not affect the way the code is compiled. AD remains unchanged by compilation. AD is just an artifact like many others,

$$\mathrm{C(ADSC)} \rightarrow \mathrm{CSC} + \mathrm{AD}\#(3)$$

As a result of compiling source code in the IDE with links to various artifacts not excluding AD is the Compiled Source Code (CSC). AD as a catalyst and at the same time one of the artifacts is not consumed and still remains. Concerning orientation (O) in the existing source code, we can assume a situation where at the beginning (in time denoted as t1) of the interaction with the source code, the degree of a user orientation is denoted as Ot1. After some time of interaction (in time denoted as t2), study or acquaintance with the source code, the degree of the user orientation will be referred to as Ot2. So, the duration of getting to know the source code Δt without the use of AD as a catalyst can be expressed as:

$$\mathrm{Ot1} \rightarrow \mathrm{Ot2}\colon \mathrm{t2} > \mathrm{t1}\colon \Delta\mathrm{t} = \mathrm{t2} - \mathrm{t1}\#(4)$$

Another situation occurs if we use AD as a catalyst. Catalyst - AD influences only the duration of the user's interaction with the source code ΔtAD in order to get the required level of the orientation. While Ot3 is the user orientation in the beginning and Ot4 is the user orientation after some time of interaction with the source code.

$$Ot3 \overset{AD}{\rightarrow} Ot4\colon t4 > t3\colon \Delta tAD = t4 - t3\#(5)$$

So, we can express our hypothesis as:

$$\Delta tAD < \Delta t\#(6)$$

To prove this hypothesis, a series of tests were carried out under the same conditions with respect to the source code sample and the group of participants. The experimental group of participants - for evaluation purposes, a group of 12 participants was created. All participants who participated in the evaluation activities were all experienced in programming (each worked for at least 3 years as a developer) and recognized the Java language in which the source code sample was implemented. Four of the participants previously worked as testers and two of them have had practical experience with system analysis. The age of participants varied when the youngest was 28 and the oldest 49 years old. Source code's working set - a source code's working set is a set of source code classes to which the developer needs access throughout a particular assignment. For evaluation purposes, we have used a source code fragment whose characteristics are listed in Tables 1, 2, and 3.

Evaluation was performed by the following ten testing scenarios.

A.) Activity diagram coverage by ADACSCO. Implementation of ADACSCO, a data model has been created over which the plug-in is working. To confirm that, this data model covers all the important AD UML 2.5 elements (specifically ActivityNodes, flow-of-control constructs - decision and concurrency) we have compared it with the UML AD metamodel. Due to the limited space in the article, we do not provide a visualization of this mapping. Finally, all metamodel entities respectively the metamodel elements that were identified in the Activity diagram driven source code modularization section as necessary are included and implemented in the proposed data model.

B.) UML AD creating in ADACSCO. Source code and related eight sample ADs were prepared in Enterprise Architect and available in the form of .png images as well as .xml files. Respondents did not recognize the source code before. Each of the participants had to get in these ADs into ADACSCO in two ways - manually and automatically.

Manually creating a tree structure - in this case, participants had to redraw the ADs based on prepared images using ADACSCO editor functionality. The sequence of actions that participants had to perform was as follows: create a project in Eclipse IDE then redraw activity diagrams in ADACSCO including activity diagram elements for each and finally assign parts of the source code to AD elements. Respondents did not consider creating a project and activity diagram as problematic just as the creation of elements in these diagrams. They even marked them as actions that would be rarely performed. The complications were seen just in the assignment of the classes to the actions. Assigning classes to actions may occur in an ongoing project. Therefore, in such a case, it can be perceived as counterproductive. Of course, it can be argued that each developer would use ADACSCO only on the part of the AD that is the subject of his work. Therefore, he should assign classes only to selected and a significantly smaller number of elements. There could be a lot of duplicate work when two developers work on the same AD. In order to avoid similar situations, ADACSCO has been extended to support collaboration between developers by export and import functionality (described in Multiple Users' Collaboration support).

Automatically creating a tree structure - in this case, participants had to create ADs using a .xml file import ADACSCO functionality. This option unburdens the developer not only from the tree structure creation but also in certain cases from the manual assignment of classes to selected elements of the activity diagram. A prerequisite for using this functionality is a properly designed activity diagram (in our case in the Enterprise Architect) exported in the form of .xml file. ADACSCO can read the file, convert it properly into the tree structure and visualize it.

During the import phase, the AD and its elements are automatically created. Created links between actions are also supported by the plug-in. If the activity diagram has a relationship between the activity diagram element and a class, ADACSCO can create these links as well. So if the associated class already exists in the project, ADACSCO will offer link with the element based on name matching. If the class has not been created yet ADACSCO will offer an option for its creation. No negative comments were recorded by the participants in the assessment of the implemented functionality.

C.) Activity diagram elements data customizing. The ADACSCO plug-in allows creating of tagged values for each project, AD, or AD's elements. The tagged value consists of the name and the value that belongs to it. In this way, it is possible to record the required set of data in the tree structure of ADACSCO for each element of the AD. In assessing this functionality, each participant had to find as many ways of applying the tagged values as possible, respectively through tagged values should store as much data as one thinks to be useful. The most commonly created tagged values were: priority - the importance of why a snippet of the code was attached to the AD element, complexity - the degree of implementation difficulty, dates - each participant gave at least one date, requirements - participants would welcome to add non-functional requirements related. There were also: information useful in the agile development process - participants saw the potential of tagged values also in recording information in agile development method when it could be recorded for example a sprint number, start and end date or product owner and attributes for implementing user acceptance tests (UAT) - tagged values can also be used in activity diagrams to generate UAT tests.

D. Created structure modification support. Evaluating the available options for modifying the structure of AD, participants were asked to edit the AD in EA. Then, transfer these changes into ADACSCO or they could skip editing the diagram in EA if they did not consider it important. As part of this evaluation, participants did not make any serious comments and did not notice any shortcomings of the plug-in as well.

E. Multiple Users' Collaboration support. Multiple Users' Collaboration has been evaluated at the source code level as well as ADACSCO Tree Structure. Source code level collaboration point of view - ADACSCO does not prevent or restrict users from using Git, SVN for collaboration. Because ADACSCO does not directly work with source files but only opens them with IDE Eclipse Editor (similar to PE), there are no temporary or permanent copies of the source code files. This behavior ensures that source code files management does not require any additional configuration or tracking of a different set of source code files. In other words, if one developer uses ADACSCO and the other does not, it has no impact on working with Git. This behavior was tested in the following scenario. Each of the two participants had their own computer while the first one updated the source code using ADACSCO, and the other used the standard PE. They used GitHub service through terminal commands. Description of the test scenario: Respondent 1 (PC 1): Updating the NaturalPerson.java class (ADACSCO); Inserting changes to the shared Git repository (push). Respondent 2 (PC 2): Downloading changes from shared Git repository (pull), Updating the NaturalPerson.java and LegalEntity.java (Package Explorer), Inserting changes to the shared Git repository (push). Respondent 1 (PC 1): Downloading changes from shared Git repository (pull). The result of evaluating the use of ADACSCO with Git was consistent with the assumption. This means that after the testing scenario, participant 2 had available changes made by participant 1, and participant 1, in turn, saw changes made by participant 2.

Collaboration at the Tree Structure level of ADACSCO plug-in - tree-level collaboration means sharing a previously created activity diagram, its elements including elements with the associated source files. The goal of enabling

ADACSCO activity diagrams to be shared is to avoid unnecessary duplicate activity with developers. If someone has already created the structure, it is unnecessary to create it again. Tree-level collaboration at the ADACSCO platform is made possible by AD exporting and importing. This means that the developer exports the already created activity diagram in ADACSCO. The diagram is stored in a .xml file that can be delivered to a different developer, e.g., by email. The tree-level collaboration was evaluated by all 12 participants. Eleven of them exported their AD and put the export on a common USB. The last twelfth participant imported these AD into ADACSCO. As a result of this evaluation, the twelfth participant had access to the tree structures created by his colleagues in his ADACSCO. This evaluation also successfully attempted to show a possible way of using the plug-in in an already running project, even when a large system is distributed among multiple developers.

F.) Modularization support. Modularization of the software is the distribution of monolithic code to modules having a certain modular structure [28]. Three ways of organizing the source code, namely Package-based approach, the Use Case-based approach, and the ADDA approach were confronted during the evaluation process. Package-based approach - traditional approach uses packages to organize classes. These packages used to be based on domains (as was the case in the evaluated sample code). The basic modularization unit is a package. Use Case-based approach - this approach uses an organization based on use cases. One use case is included in a file (a markdown file) that contains the source code needed to implement the flow of a given instance of the use case. The basic modularization unit is a single step of the use case. ADDA approach - a source code modularization approach that uses the structure based on the UML AD, where the main modularization unit is an action element of the activity diagram. This approach was described in more detail in section Activity diagram driven source code modularization. Each approach from those mentioned above provides a different source code modularization base. For a closer comparison of all three approaches, selected values were identified. The results are shown in Tables 1, 2, and 3. When comparing the results of the basic modularization units (package, activity diagram element and single use case step), we can see that the lowest average number of source code files per modularization unit is reported by the use case approach, see Table 3. The reason that the markdown file associated with the use case has the best results is that it contains only the source code for that use case. This means that it does not provide the user any unnecessary methods, attributes, comments, and so on.

However, when looking at the results, the activity diagram - the ADDA approach (see Table 2) shows nearly 8,308 source code rows per AD whereas the package has approximately 3,420 source code rows per packages, see Table 1.

Table 1
Modularization: Package based approach project perspective

Number of packages	12
Number of classes	157
Number of methods	689
Number of source code rows	41033
Average number of source code files per packages	13.08
Average number of methods per packages	57.42
Average number of source code rows per packages	3419.42

Table 2
Modularization: ADDA approach project perspective

Number of activity diagrams	8
Average number of elements per activity diagram	8.25
Average number of source code files per activity diagram	32.86
Average number of source code files per activity diagram element	4.59
Average number of methods per activity diagram	124.028
Average number of methods per activity diagram element	17.75
Average number of source code rows per activity diagram	8307.75
Average number of source code rows per activity diagram element	1164.57

However, this result is correct and expected because one class can participate in the implementation of multiple elements of the AD. This is why the ADDA approach shows a higher number of source code rows. This view, however, does not rule out the fact that repetitive occurrences of redundant parts of the code can be eliminated by more detailed methods to action binding. If we compare the ADDA and Package based approach depending on the working sets, we get 41,033 to 8,308 in favor of the ADDA approach. The ADDA approach shows a better result also in decomposition AD into its elements too. The average number of source code files per activity diagram element is only 4.59 to 13.08 in favor of the ADDA approach. We believe that a smaller number of files per modularization unit reduces the user's mental load. At the end of the modularization evaluation, a significant majority of all participants involved in the study opted for modularization based on ADs and use cases. Only one of them voted for modularization based on packages.

G.) Readability and comprehensibility evaluation. Readability and comprehensibility are the properties of the source code which can be evaluated not only on the basis of subjective evaluation of participants but also from the perspective of different metrics. As was already mentioned ADDA approach does not modify the source code in any way, but only provides a new look at its structure. Based on this argument it could be assumed that the metrics for Package-based approach and the ADDA approach will be identical. However, this assumption is not entirely correct. ADDA allows you to assign the same file of a source code (e.g, class) to multiple elements. As a result of this behavior, ADACSCO creates duplicates in the tree structure. It is then possible to define two assumptions. When calculating metrics for both projects of both approaches, the ADDA approach will have worse results due to the duplicates mentioned above. When calculating metrics for the source code's working set (the set of source code files the developer will need when implementing the specified assignment), the ADDA approach will have better results because the Package working set contains many more source code files. In order to confirm both assumptions, but not exclusively (Number of Code Rows (NCR), Average Number of Classes (ANC), Average Method Complexity (AMC) - Average Cycling Complexity [23], Average Number of Direct Descendants of a class (ANDD), Average Number of Inherited Methods (ANIM) and the Average Coupling Between Object (ACBO) [16]) metrics have been calculated to identify ADDA's impact on readability and comprehensibility. All 6 metrics for readability and comprehensibility were calculated using the Eclipse IDE Metrics plug-in. Cyclomatic complexity for AMC was also verified by the CyVis tool. Table 4 lists the results of the experiment confirming the assumptions described above. Better results for the project have the Package-based approach and on the contrary, the results measured for the source code working set are in favor of the ADDA approach.

Based on the results shown in Table 4 it can be concluded that the ADDA approach improves the readability and comprehensibility of the source code the developer is working on, but only within the scope of one activity diagram. One backlog task usually does not exceed one use case which is comparable to the above AD range. The principle of problem decomposition always leads to simpler tasks. This may be a recommendation to create a backlog, consisting of tasks not exceeding one AD. Table 4 also shows that the ADDA approach reduces the working set of classes the developer needs at the time of implementation. It is worth mentioning another finding based on the participants' testimonies. Classes are not just class clusters, as it is in the case of packages because of the structure based on AD. This only confirms the conclusion obtained on the basis of the metrics calculation. So, readability and comprehensibility are better when working with classes in an ADDA manner more than working with them in package organization.

H.) Source code orientation catalyzation. This section does not reflect issues such as proper source code offsetting or use of appropriate names or decision and cyclic structures but it deals with simple quick and accurate identification of the file associated with actions. The key metric for the purpose of the evaluation was the time to find related artifacts. This should be supported by the tree structure itself based on activity diagrams and orientation-supporting elements - decorations in the form of graphical differentiation of individual elements of the activity diagram, graphical resolution based on integrity, information about node number of descendants, and prefix for structured activities.

Table 3

Modularization: Use case based approach project perspective

Number of use cases	11
Average number of source code files per use case's step	1
Average number of methods per use case	7.2
Average number of source code rows per use case	428.79

Graphical differentiation between activity diagram's elements in the plug-in is done using icons based on UML 2.5. Each element of the activity diagram stores information about implementation completeness and entails information about the number of source code files that are related to it. Improvement in the source code orientation was tested by participants in fulfilling selected tasks. Respondents' statements revealed a positive assessment of the above-mentioned artifacts to support orientation in the related source code.

Table 4

Metric results

Tool/Metrics	Package Explorer		ADACSCO	
	Project	Package	Project	AD
NCR	41033.0	3730.3	76 861.8	9 607.7
ANC	157.0	14.3	303.0	37.9
AMC	27.1	2.5	34.8	4.4
ANDD	0.714	0.1	0.9	0.1
ANIM	3.39	0.3	3.8	0.5
ACBO	2.1	0.2	2.7	0.3

I.) Performing tasks by participants using ADDA. Respondent's role was to perform several tasks (Modify VAT calculation from 20% to 15%, Change the notification email text after successful client creation, etc.) on a selected source code sample using three different approaches. All 12 participants attended the evaluation. Respondents were divided into three groups of four. The first group used a traditional approach based on packages the second used ADACSCO and ADDA approaches. The last third group used the Use Case-based approach implemented by the ADACSCO plug-in too. The time needed to identify the class was measured for each task, also the number of clicks and the number of open classes for each participant. None of the participants previously met the source code sample and did not have any more information about it. The evaluation results confirmed the expectations. The traditional Package-based organization has much worse (approximately four times worse) results than ADDA and Use Case-based approach see Figure 4. ADDA approach was not the best in terms of time but its results did not lag behind the results of UC-based approach. So, our assumption that the time required to obtain orientation in the source code in ADDA approach will be shorter than in the standard package-oriented approach has been confirmed. Performing tasks using Package Explorer; participants - who did not know the source code - had to browse the classes that could have the required functionality based on the similarity in names. This observation was confirmed by the participants themselves.

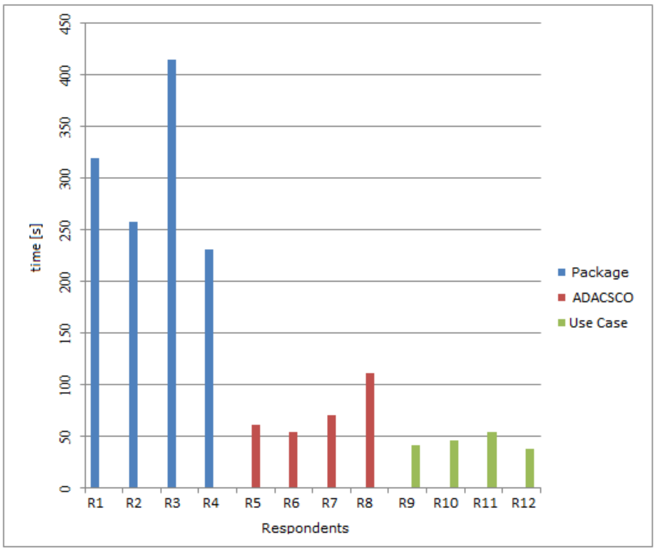

Figure 4
Task completion - time comparison between approaches

Respondents working with ADDA; have been much more successful in terms of time see Figure 4. They said it was useful that they were able to reduce the number of classes from about 150 to about 8 on the basis of functionalities. They found it great in finding the right class and of course very useful if someone did not know the code. Respondents' statements in further discussion only confirmed the results where model-based approaches that provide users with functionality information have a significant positive effect on the orientation between source code files. Previous testing has revealed that ADDA and use Case-based approaches have a significant impact on improving orientation. In order to make a subjective comparison between the two successful approaches participants have tried both approaches in the reverse order. Subsequently, participants' subjective opinion was investigated. It was not possible to distinguish which approach was better from participants' subjective statements because each response was based on personal preferences. As a result, 5 participants would use a UC-based approach and three of them approach based on activity diagrams.

J.) Evaluation by selected user types. ADDA's main asset is a source code extension by the information that may be beneficial not only for developers but also for testers, managers or analysts, see Activity diagram driven source code modularization. In order to verify this assumption, an interview was conducted with an analyst, project manager, and two testers (none of whom belonged to the original group of 12 participants from the previous evaluation activities). Each of these four participants should subsequently confirm or refute the usefulness of the proposed approach in their normal workload. The participant, who works as an IT analyst, proposed an extension as it is possible to associate other types of files with AD – for example by GUI designs, data samples, etc. Improving the overview of the project, the project manager (PM) must have was appreciated by the respondent working as a project manager. He emphasized that it would be interesting to see how many processes are already implemented and how many still remain. Further use of the approach was mentioned according to a price estimation of future projects. It could be based on the overall business process duration or just a single action duration. Respondents - testers identified usage of the approach in unit testing. They claimed that as well as being able to associate the source code according to AD elements, it is also possible to associate the test scripts. Another option that they pointed out was directed to the description of errors directly related to the process or its particular part. They have appreciated the ability to add the time of the latest tests and versions. Test time and test version information could be recorded as tagged values. From the approach assessment by specialists, it can be assumed that the approach can also be beneficial to non-developers. This conclusion can only be put forward as just a premise. The actual benefit could only be tested and found within a real project in which ADDA and ADACSCO were not used.

Conclusions

This paper presented the idea of organizing source code from the activity diagram perspective denoted here as the ADDA approach to improve modularization, readability, comprehensibility, and mainly orientation in the source code. In order to achieve this goal, an analysis of the possible benefits of UML AD was performed. The result of this analysis was the identification of important features of the activity diagram which could make a significant contribution in binding with the source code. Based on the results of the analytical activities, ADDA has been suggested which mainly uses explicit associations between the source code parts and the activity diagram elements. These bindings allowed us to define a new modularization structure that can be depicted as a tree where the root of the tree is an activity diagram its nodes are the elements and the leaves are the classes themselves. The ADACSCO plug-in to Eclipse IDE has been implemented for evaluation purposes to create and work with a tree structure defined by the ADDA approach. ADDA approach evaluation using ADACSCO consisted of 6 parts: Evaluation of ADACSCO activity diagram coverage, Workflow evaluation, Modularization assessment, Readability and Comprehensibility assessment, Evaluation of orientation, and evaluation by selected user types. Twelve participants took part in the evaluation. Within the selected evaluation parts, a comparison has also been made with another two existing approaches – Package-based approach and Use Case-based approach. The ADACSCO AD coverage assessment was performed to demonstrate that each element of the activity diagram identified in the ADDA proposed approach has its own representation in ADACSCO. The result was achieved by mapping the selected metamodel of the activity diagram to the ADACSCO data model. The orientation was evaluated in the form of subjective evaluation of 12 participants and was divided into two parts. The first one consisted of testing and commenting elements supporting the orientation in the source code. The second part of the evaluation was to perform certain tasks by participants using the Package-based approach, ADDA approach and Use Case-based approach. When performing these tasks, the values were measured: time, the number of clicks, and a number of open classes. The ADDA approach had the second-best results only slightly behind the Use Case-based approach. The Package-based approach, on the other hand, had worse results than the other two approaches. The reason for these differences between the results was that the Package-based approach did not contain any business information. Evaluation by selected user types has demonstrated the possibility of using ADDA and ADACSCO also for project team members such as analyst, project manager, and tester. Finally, it can be generalized that creating explicit links between the available artifacts in the development of the software supports the orientation in the source code itself. Defining the source code structure based on activity diagrams maintains a degree of business information that can help the user navigate between the source code classes.

Acknowledgment

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under grant No. VG 1/0759/19, the Slovak Research and Development Agency under contract No. APVV-16-0213, and by the Operational Programme Integrated Infrastructure for the project: Electronic methods for detecting unusual business operations in the course of trade (ITMS code: 313022W057), co-funded by the European Regional Development Fund (ERDF).

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. (2006) Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Formal Methods for Components and Objects. Springer Berlin Heidelberg, 364-387, https://doi.org/10.1007/11804192_17

[2] Sonia Berman and Thembinkosi Daniel Semwayo. (2007) A Conceptual Modeling Methodology Based on Niches and Granularity. In Conceptual Modeling - ER 2007, Springer Berlin Heidelberg, 338-358, https://doi.org/10.1007/978-3-540-75563-0_24

[3] Jürgen Börstler, Michael E. Caspersen, and Marie Nordström. (2015) Beauty and the Beast: on the readability of object-oriented example programs. Software Quality Journal 24, 2 (feb 2015), 231-246, https://doi.org/10.1007/s11219-015-9267-5

[4] Raymond P. L. Buse and Westley R. Weimer. (2010) Learning a Metric for Code Readability. IEEE Trans. Softw. Eng. 36, 4 (July 2010), 546-558, https://doi.org/10.1109/TSE.2009.70

[5] Michal Bystrický and Valentino Vranić. (2016) Development Environment for Literal Inter-language Use Case Driven Modularization. In Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016) ACM, New York, NY, USA, 12-15, https://doi.org/10.1145/2892664.2893465

[6] Michal Bystrický and Valentino Vranić. (2016) Literal Inter-language Use Case Driven Modularization. In Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016) ACM, New York, NY, USA, 99-103, https://doi.org/10.1145/2892664.2892680

[7] Michal Bystrický and Valentino Vranic. (2017) Preserving use case flows in source code: Approach, context, and challenges. Computer Science and Information Systems 14, 2 (2017) 423-445, https://doi.org/10.2298/csis151101005b

[8] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. (2016) Using Cohesion and Coupling for Software Remodularization: Is It Enough? ACM Trans. Softw. Eng. Methodol. 25, 3, Article 24 (June 2016), 28 pages, https: //doi.org/10.1145/2928268

[9] A. Cockburn. (2001) Writing Effective Use Cases. Addison-Wesley

[10] Ferruccio Damiani, Arnd Poetzsch-Heffter, and Yannick Welsch. (2012) A type system for checking specialization of packages in object-oriented programming. In Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC. ACM Press. https://doi.org/10.1145/2245276.2232058

[11] Anil Saroliya Deepa Dhabhai, A. K. Dua. (2015) Review paper: A Study on Metric For Code Readability. International Journal of Advanced Research in Computer Science and Software Engineering 5, 6 (June 2015), 463-466

[12] James L. Elshoff and Michael Marcotty. (1982) Improving Computer Program Readability to Aid Modification. Commun. ACM 25, 8 (Aug. 1982), 512-521, https://doi.org/10.1145/358589.358596

[13] Rik Eshuis. (2006) Symbolic model checking of UML activity diagrams. ACM Transactions on Software Engineering and Methodology 15, 1 (jan 2006), 1-38, https://doi.org/10.1145/1125808.1125809

[14] Rik Eshuis and Roel Wieringa. (2001) A Real-Time Execution Semantics for UML Activity Diagrams. In Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 76-90, https://doi.org/10.1007/3-540-45314-8_7

[15] M. Fowler and Safari Tech Books Online. (2004) UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley. https://books.google.sk/books?id=nHZslSr1gJAC

[16] R. Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. In Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering. IEEE Comput. Soc. https://doi.org/ 10.1109/step.1997.615494

[17] Andreas Heinecke, Tobias Brückmann, Tobias Griebe, and Volker Gruhn. (2010) Generating Test Plans for Acceptance Tests from UML Activity Diagrams. In 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems. IEEE. https://doi.org/ 10.1109/ecbs.2010.14

[18] D. Hendrix, J. H. Cross, and S. Maghsoodloo. (2002) Corrections to "the effectiveness of control structure diagrams in source code comprehension activities". IEEE Transactions on Software Engineering 28, 6 (jun 2002), 624-624, https://doi.org/10.1109/tse.2002.1010064

[19] S. L. Jim and P. Klint. (2006) From UML diagrams to behavioural source code. (2006). unpublished thesis

[20] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and Inyoung Ko. (2007) Test Cases Generation from UML Activity Diagrams. In Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007) IEEE. https://doi.org/10.1109/snpd.2007.189

[21] Seonah Lee, Gail C. Murphy, Thomas Fritz, and Meghan Allen. (2008) How can diagramming tools help support programming activities?. In 2008 IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE. https: //doi.org/10.1109/vlhcc.2008.4639095

[22] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. (2014) Automatic Early Defects Detection in Use Case Documents. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14). ACM, New York, NY, USA, 785-790, https://doi.org/10.1145/2642937.2642969

[23] T. J. McCabe. (1976) A Complexity Measure. IEEE Transactions on Software Engineering SE-2, 4 (dec 1976), 308-320, https://doi.org/ 10.1109/tse.1976.233837

[24] Mohd Nazir, Raees A. Khan, and Khurram Mustafa. (2010) A Metrics Based Model for Understandability Quantification. CoRR abs/1004.4463 (2010) arXiv:1004.4463

[25] OMG. Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011

[26] Mishra P. (2014) Measuring The Understandability And Maintainability Of C# Inheritance And Interface Source Codes. Universe of Emerging Technologies and Science 1, 1 (June 2014)

[27] Jaroslav Porubän and Milan Nosál. (2014) Leveraging Program Comprehension with Concern-oriented Source Code Projections (2014) https://doi.org/10.4230/oasics.slate.2014.35

[28] Girish Maskeri Rama and Naineet Patel. (2010) Software modularization operators. In 2010 IEEE International Conference on Software Maintenance. IEEE. https://doi.org/10.1109/icsm.2010.5609546

[29] Luz Rello, Horacio Saggion, Ricardo Baeza-Yates, and Eduardo Graells. (2012) Graphical Schemes May Improve Readability but Not Understandability for People with Dyslexia. In Proceedings of the First Workshop on Predicting and Improving Text Readability for Target Reader Populations (PITR '12) Association for Computational Linguistics, Stroudsburg, PA, USA, 25-32

[30] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. (2006) On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modelling - Volume 53 (APCCM '06) Australian Computer Society, Inc., Darlinghurst, Australia, Australia

[31] Ksenia Ryndina, Jochen M. Küster, and Harald Gall. Consistency of Business Process Models and Object Life Cycles. In Models in Software Engineering. Springer Berlin Heidelberg, 80-90, https://doi.org/ 10.1007/978-3-540-69489-2_11

[32] Mahesh Shirole and Rajeev Kumar. (2012) Testing for concurrency in UML diagrams. ACM SIGSOFT Software Engineering Notes 37, 5 (sep 2012) 1 https://doi.org/10.1145/2347696.2347712

[33] Martin Siebenhaller and Michael Kaufmann. (2006) Drawing Activity Diagrams. In Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis '06) ACM, New York, NY, USA, 159-160 https://doi.org/10.1145/1148493. 1148523

[34] Valentino Vranić and Adam Neupauer. (2019) Abstract Layers and Generic Elements as a Basis for Expressing Multidimensional Software Knowledge. In Modelling is going to become Programming, a workshop at 23rd European Conference on Advances in Databases and Information Systems. to be published by Springer, https://doi.org/10.1109/ icpc.2013.6613836

[35] Zhiqun Wang. (2013) The application of business activity diagram to capture use case. In 2013 IEEE Third International Conference on Information Science and Technology (ICIST) IEEE https://doi.org/10.1109/ icist.2013.6747568