

Systematic Story Driven Modeling

Ira Diethelm¹, Leif Geiger², Albert Zündorf²

Technical Report, University of Kassel

¹Gaußschule ²SE, Universität Kassel
Löwenwall 18a Wilhelmshöher Allee 73
38100 Braunschweig 34121 Kassel

(ira.diethelm | leif.geiger | albert.zuendorf)@uni-kassel.de

<http://www.se.e-technik.uni-kassel.de/se>

Abstract: Story Driven Modeling (SDM) is a technical software development process employing UML based modeling in all project phases, including implementation and test. SDM proposes object games for refining textual usecase scenarios into so called story boards, i.e. sequences of UML interaction diagrams. From these story boards the modeler derives class diagrams and UML based method behavior specifications and UML based JUnit tests. The code generators of the Fujaba CASE tool turn this automatically in a Java implementation and run the JUnit tests checking whether the method behavior conforms to the usecase scenarios. This paper reports about significant improvements in our systematic approach for turning story boards into method behavior specifications. We have used this process in quite a number of educational and research projects and we are exporting our ideas to the first industrial projects.

1 Introduction

This paper reports about significant improvements of our *Story Driven Modeling* (SDM) approach, cf. [KNNZ00, Zü01], towards a very systematic process for the development of object oriented software. Based on initial ideas from [DGMZ02] and [DGZ02], we have run a large number of educational and research projects and some first trials in industrial projects. With the experiences we gained within these projects, we were able to develop systematic guidelines for the most crucial phase in SDM, the derivation of method behavior from example scenarios. These guidelines exploit that in SDM the analysis phase as well as the design and implementation phase use very similar UML based notations allowing to model at a very high object oriented level of abstraction. This facilitates to generalize example behavior into general behavior specifications.

In the following sections we describe all phases of SDM with an emphasis on the derivation of method behavior specifications. Then we give conclusions and future work.

2 Requirements elicitation

Our approach starts with the usual requirements elicitation resulting in usecase diagrams and structured usecase descriptions. In this paper we use a library system example. The overall example project covers the usage of a database for the management of books, customers and lending processes, a full GUI, multiple user support, etc. For the sake of simplicity, here we focus on the return of books and especially of the adding of returned book index cards into the book catalogue. This means, we just use a sorting problem to illustrate how scenarios are modeled and behavior specifications are systematically derived.

For our approach, it is important that the usecase scenarios have a certain format and that they cover all alternatives. For the sorting problem that means, we systematically consider the sorting of a stack with only one card, the sorting of a stack with two cards and so on.

Each usecase scenario consists of a description of the start situation, an invocation, an elicitation of all executed steps and a description of the resulting situation, cf. below.

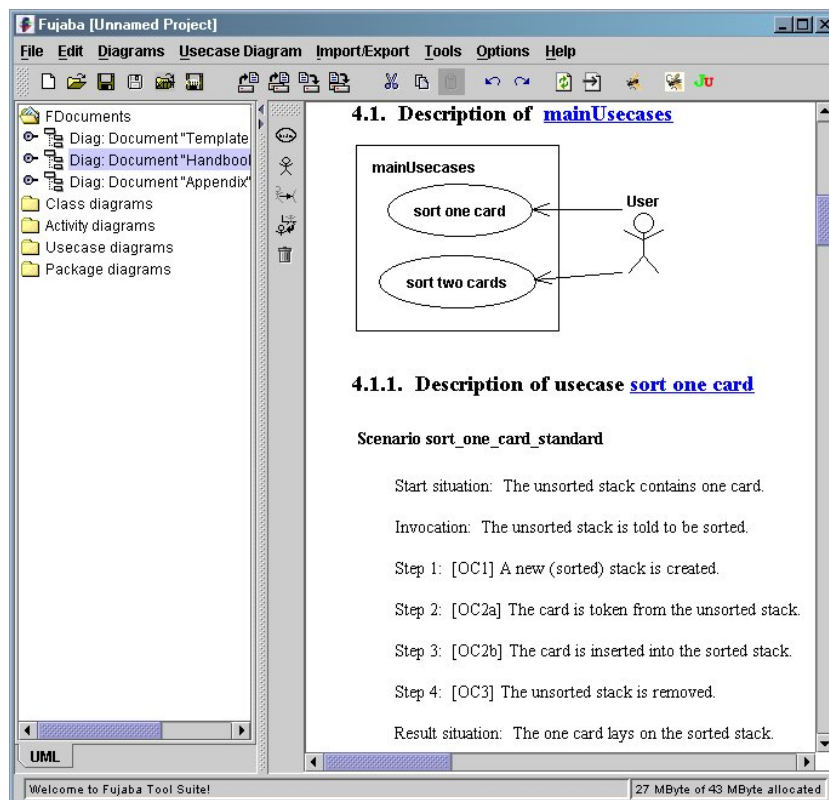


Abbildung 1: Usecase diagram with textual scenario

3 Story Boarding - refining usecases

To elicit the usecase scenarios, the developers may already perform a so-called object game. In the object game we use cards or physical items representing objects from our problem domain. In this example these are simply book index cards. Using these items, the developers go through certain usecases. During this problem all steps are protocolled as usecase descriptions.

Note, our object game is comparable with CRC card approaches, see e.g. [Bo91]. However, the outcome of a CRC card session is some kind of class diagram. The outcome of our object games are scenario descriptions. We turn these scenarios description into UML interaction diagrams, first, and we derive the class structure, second. Due to our experiences in many educational and research projects, deriving scenario descriptions first is much more appropriate and feasible then targeting on class structures. Actually, CRC card approaches go through scenarios, too. However, they do not protocol the sequence of executed steps but only the employed classes, relations, and methods. Thus in CRC card approaches, the most valuable information get lost. Our object game protocols exactly these valuable scenario information.

The developers may either protocol their object games in textual form and turn this into UML interaction diagrams in a second step or they may use UML interaction diagrams, directly. In either case, the outcome of the next step should be a UML based scenario description.

Figure 2 shows such a diagram, a so called story board. A story board is a sequence of UML collaboration diagrams that show the changes of the object structure in this scenario comic strip alike. Note, that we concentrate on one scenario at a time what means that we don't have to deal with the general case but we deal with the current example run, only. In figure 2 the trivial case, that there is only one index card which has to be sorted, is protocolled using a story board. As mentioned before, our scenarios always have to match the given structure: start situation, invocation, several steps, result situation.

So, the first activity of figure 2 contains an object diagram describing the start situation. The start situation has been modeled by the developer using an object *unsorted* representing a stack which has a *first* link to a card object called *c1*. Note, if a textual usecase description is available, our tool automatically adds the textual description of a step to the corresponding activity in the story board.

The next activity always models the invocation of the scenario. In our approach this is usually a collaboration message / method call. Here, the method *sort* is called on the *unsorted* object. Note, through this convention, in SDM each usecase is mapped to a dedicated method that implements the corresponding functional requirement.

The following steps model the changes which are protocolled during the object game. In the first of these steps (marked with *OC1* in the story board comment) a new stack is created which will then contain the index cards inorder. Note, that we use `<<create>>` and `<<destroy>>` markers to model creation and removal of objects and links. In the second step (*OC2a*) a *removeFromStack* message is send to object *c1*. This results in the removal of the *first* link from the *c1* object to the *unsorted* object. In the next step (*OC2b*) a link to

the *sorted* object is created as consequence of an *insertIntoStack(sorted)* message. Then the *unsorted* object is removed (OC3).

The last activity now models the result situation at the end of the scenario. In our case, this is a stack object containing the card (inorder).

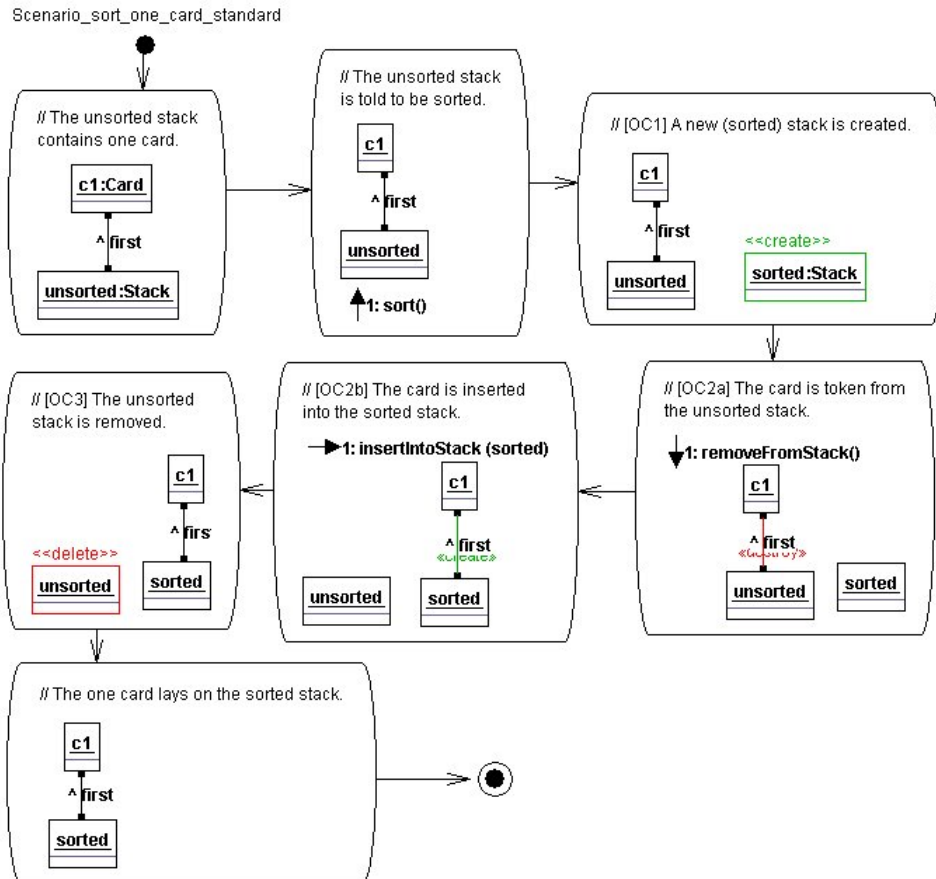


Abbildung 2: Story board for sorting of one index card

Figure 3 shows the scenario for sorting two cards. In the start situation, these two cards are unsorted, so that sorting is really needed in this scenario. In the start situation of the story board this is modeled using a *value* attribute for the card objects. The invocation and the first step equals the scenario for one card. In the second step (TC2a), the *removeFromStack* message is sent as well, but the effects differ. As in the first scenario, the *first* linked is removed but also the *next* link from object *c1* to object *c2*. The card *c2* then becomes the first object of the unsorted stack. OC2b shows the insertion of this card into the sorted stack as described above. OC3a is again removal from the unsorted stack and OC3b insertion into the sorted stack. Note that here, the card *c2* has to be inserted at the right place. In the

last step the unsorted stack is removed and the result situation consists of a stack containing two cards in the correct order.

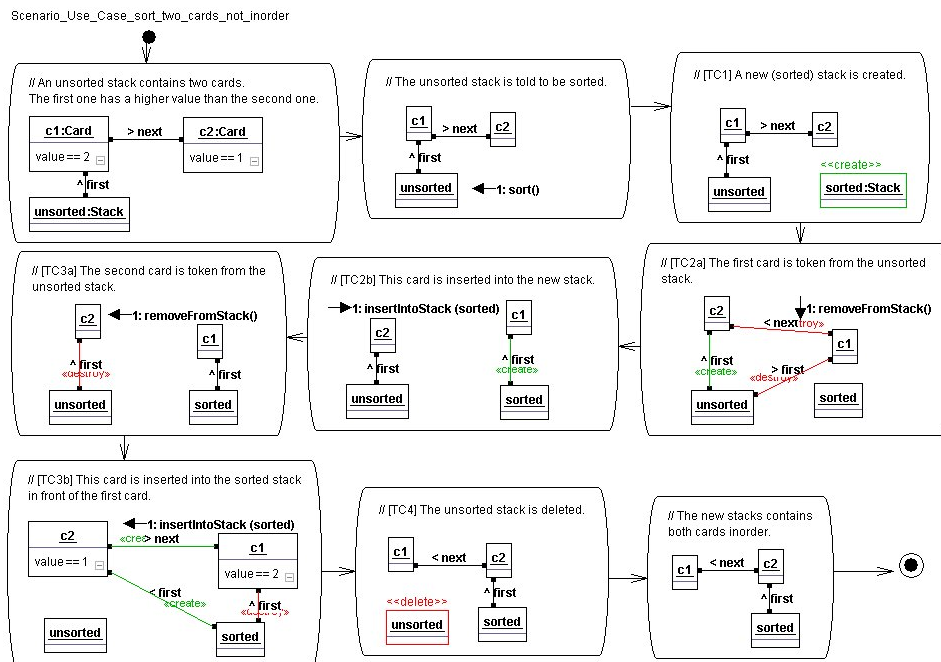


Abbildung 3: Story board for sorting of two index cards

Due to our experience, the refining of usecases described above may easily be done by multiple team members in parallel. The object game is done with the whole team. A flip-chart may be used to protocol important steps or to develop some coarse grained story boards. Every requirement engineer gets a set of scenarios to work on and some copy of the flipchart content. Then every requirements engineer elaborates detailed story boards for his or her scenarios on his own. During this process, a common class diagram ensures a consistent use of object kinds, attributes, links and methods. The derivation of a class diagram is described in the next chapter.

As mentioned above, we use story boards to describe one example scenario at a time. Though, as story boards a based on activity diagrams, they could also be used to model branches or loops. So, more experienced developers may model “more general” scenarios, directly. However, we made the experience that concentrating on one example run is much easier and is very helpful especially for beginners. In addition, a set of alternative scenarios is much easier to read and to understand then a single complex activity diagram. This is an important property e.g. for customers and newcomers to the project. So, our approach suggests to use a set of simple alternative scenarios instead of a small number of complex activity diagrams. Additionally, our CASE tool automatically generates test cases out of each story board (cf. [GZ03]). Thus a large set of alterantive scenarios results in a large set

of test cases for the later implementation.

In our experience, the step from the textual usecase descriptions to the UML based scenarios is a very important modeling activity. We observed, that development teams very easily agree on rough textual usecase descriptions. However, these textual descriptions omit many very important details. The story boarding phase frequently reveals severe misunderstandings in differences in interpretation. During story boarding, the team decides on how to model the different states of the system and how the different execution steps are employed in this model. This may result in heated discussions. However, it resolves many interpretation conflicts in a very early phase and provides an ideal basis for the subsequent development steps. After story boarding, all team members have a detailed common understanding of the design and implementation concepts of the desired system. As a participant of an industrial tutorial said: “Story boarding is where the decisions are made”.

4 Derivation of the design specification

After having understood the problem during the object games and having protocolled it using story boards, the next phase is the systematic derivation of class diagrams and of behavior specifications.

4.1 Derivation of the class diagram

The class diagrams are easily derived from the story boards. They may be derived during story boarding or afterwards. This derivation uses the following approach: For every object in the story board, the developer has to decide from which class this object might be. He or she adds the corresponding class to the class diagram if it doesn't already exist. Objects having similar links and attributes are normally mapped to the same class. Next, for every attribute usage and every method call on an object, the attribute or method is added to the corresponding class. For every link between two objects in the story board an association between the corresponding classes has to be added to the class diagram. If the two objects are from the same class, a self-association is added to that class as e.g. the *next* link between two cards in our example.

After having created such a basic class diagram, the developer has to think about the cardinalities. In SDM, this is usually easy to decide. If a story boards contains an object with multiple outgoing links of the kind, the corresponding cardinality is obviously to-many. In this way associations may also become m-to-n. If the developer feels unsure about cardinalities this may be discussed by looking at the story boards and considering missing situations (e.g. can it happen that a card has more than one direct neighbor?).

Using this approach the following class diagram can be derived out of the two example scenarios:

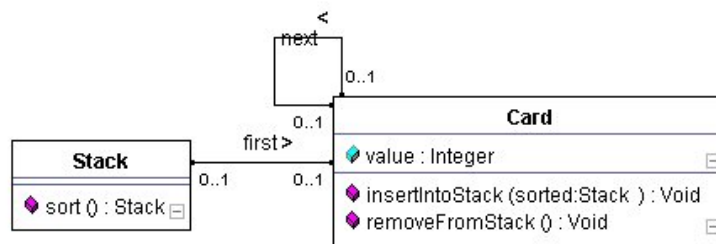


Abbildung 4: The derived class diagram

Due to our experiences, usually most of the important design decisions are already made during story boarding. Thus, the derivation of a first class diagram is usually very easy. In addition, our approach avoids many mental problems that many beginners have with self associations when they try to develop a class diagram without looking at example scenarios. From our teaching experiences, we observed that the quality of the class diagrams has been increased dramatically, since we have introduced story boarding in the development process.

4.2 Derivation of the behavior specifications

Now, “only” the behavior specifications are missing. This means that the method bodies are still empty. SDM uses the following approach for the systematic derivation of method bodies from story boards:

1. identify all usages of a method in the story boards
2. for each usage:
 - (a) identify all effects of this method call. These are changes to the object structure or subordinate method invocations. These effects may be shown in the current activity and in following activities.
 - (b) identify the minimal context required for this method call to be able to execute the identified effects.
 - (c) copy the minimal contexts to the activity diagram modeling the body of the considered method.
 - (d) identify “similar” activities within the method body and try to merge them.
 - (e) Resolve conflicts in the resulting control flow by adding appropriate branching conditions.
3. add loops and branches to cover the general case.

The result of this approach will be a UML interaction diagram, a so called story diagram, that specifies the behavior of this method. The Fujaba CASE tool then generates executable Java code out of the class diagram and out of these method body specifications, cf. [Fu02, FNT98].

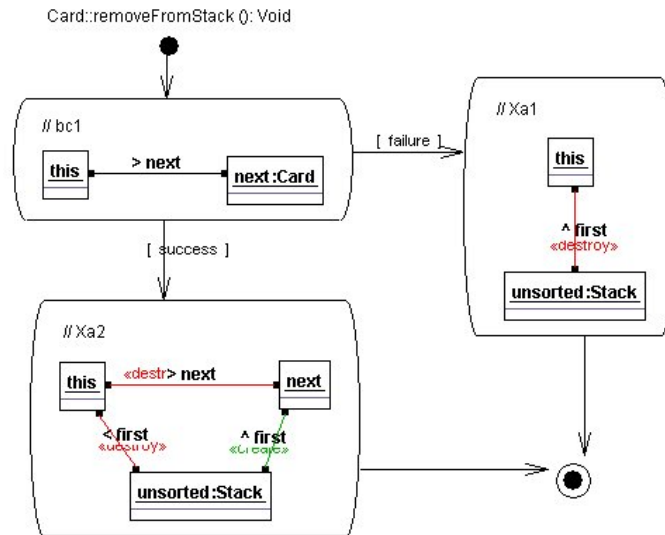


Abbildung 5: Derived story diagram for method `removeFromStack`

Using our approach for method `removeFromStack`, the developer will first find this method call in the activities marked with `OC2a`, `TC2a` and `TC3a`. The developer examines one usage after the other. For each invocation, first its effects are identified. In activity `OC2a`, the `removeFromStack` operation is called on the `c1` object. Activity `OC2a` contains no other method call and thus, all object modifications within `OC2a` are most likely caused by the `removeFromStack` operation. In `OC2a`, this effect is the removal of the `first` link between the `c1` object and the `unsorted` object. Thus, the `first` link and the `unsorted` object belong to the minimal context of the `removeFromStack` operation. The `sorted` object of activity `OC2a` is not affected by the `removeFromStack` operation, nor it is necessary in order to identify any affected object. Thus, the `sorted` seems not to belong to the minimal context of the `removeFromStack` operation. In the subsequent activity `OC2b` method `insertIntoStack` is invoked. We decide that this method invocation does not belong to the effects of the `removeFromStack` operation and stop here. Thus, the developer just copies objects `c1` and `unsorted` and the connecting `first` link to a new activity `Xa1` in a new story diagram for method `removeFromStack`, cf. the upper right activity of Figure 5. Note, rectangles within story board activities represent example objects. Contrarily, rectangles within story diagram activities represent local variables of the corresponding method. Thus, during copying, we rename the target of the method invocation, in this case the `c1` object, into the `this` variable.

Now we consider activity *TC2a* of Figure 3. Again the *removeFromStack* operation is invoked on a *c1* object. However, this time a *first* link and a *next* link are destroyed and another *first* link is created. The subsequent activity seems not to belong to the *removeFromStack* operation. Thus, we copy objects *c1*, *c2* and *unsorted* and their connecting links into a new activity *Xa2* in the story diagram for method *removeFromStack*, cf. the lower left activity of Figure 5. Again, we rename object *c1* into variable *this*. For clarification we also rename object *c2* into variable *next*.

Activities *Xa1* and *Xa2* of Figure 5 show significant structural differences. Thus, we do not merge them into a single activity.

Note, the effects of a method invocation shown in a story board occur directly after the method invocation. Thus, when we copy a minimal context from a story board into a method body, this new activity is firstly connected to the start activity. In our example, after considering the story board activities *OC2a* and *TC2a*, the story diagram for method *removeFromStack* would consist of activities *Xa1* and *Xa2* both directly reached from the start activity. This is a conflict in the control flow structure since one activity must not have multiple outgoing transitions without any branch conditions attached to them. Such conflicts need to be resolved either by merging the corresponding activities or by adding appropriate branch conditions.

In this example, the difference between the method body activities *Xa1* and *Xa2* is the additional *next* object within *Xa2*. Thus, we create a new activity *b1* that consists of the part of the object structure that makes the difference, i.e. the *this* object and the *next* object. This new activity *b1* is then used as a branch condition at the beginning of method *removeFromStack*. If *b1* matches to the current object structure, we follow the *success* link to activity *Xa2*. Otherwise, we follow the *failure* link to activity *Xa1*.

The usage of method *removeFromStack* in activity *TC3a* effects object *c2* and *unsorted* and the connecting link. We copy these elements into the method body specification. The resulting activity is equivalent to the upper right activity of Figure 5 and thus, these two activities are merged. This merge creates no conflict.

Let us look at the resulting story diagram for method *removeFromStack*. In the first activity *b1* the object structure is analyzed starting with the object matched by variable *this*. Then it is checked whether or not a *next* link to an object of class *Card* exists. If yes, this object is matched by the *next* variable and the activity is left using the *success* transition. Otherwise the *failure* transition is used. Then the effects as described in the scenarios are executed and the method is left afterwards. Note, if we leave out the class name for a variable then this variable has already been matched to some object of the runtime object structure and we refer to this already existing content. This is especially true for the *this* variable.

The method body of method *insertIntoStack* can easily be derived using the same approach. Since we would need a scenario for three cards to cover all possible cases (insertion into an empty list, at the beginning of a non empty list, at the end and somewhere between two cards) we leave this out due to paper space restrictions.

As an example for a more complex control flow, let us consider the story diagram for method *sort* of class *Stack*. Method *sort* is called in the invocation step in Figure 2 and 3. In Figure 2, the invocation step has no effects, thus nothing needs to be copied into the story

diagram for method *sort*. Activity *OC1* creates a *sorted* object. This requires no additional context and we copy just the *sorted* object into a new activity *mcOC1*, see Figure 6.

From the point of view of the *sort* method, in activity *OC2a* it just calls method *removeFromStack*. The other effects are caused within the called method. In order to call *removeFromStack*, we only need the *c1* object. However, in order to reach the *c1* object, the current stack object and the connecting link are also required as minimal context. Thus we copy object *unsorted* and *c1* and the connecting *first* link into a new activity *mcOC2a*. Object *unsorted* is renamed into variable *this*. In the story diagram for method *sort*, the new activity *mcOC2a* is appended to activity *mcOC1*, since the operations of *mcOC2a* occur after the operations of activity *mcOC1*.

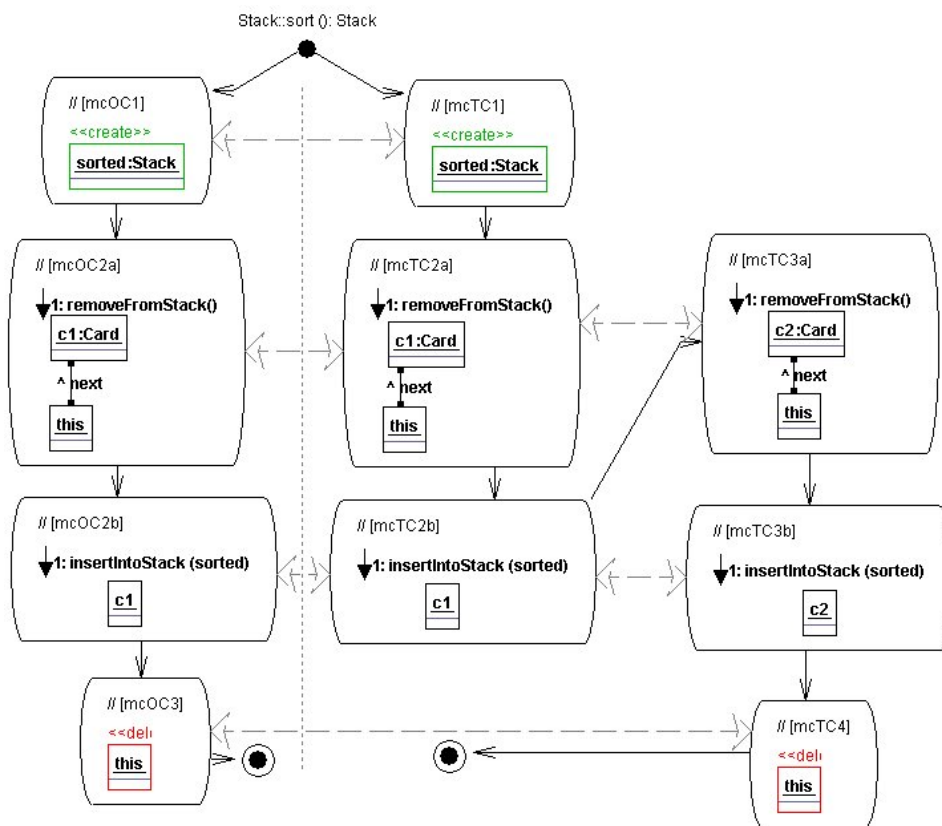


Abbildung 6: Control flow of method *sort*

In activity *OC2b* of Figure 2, method *sort* just does a method call on method *insertIntoStack*. For this operation only object *c1* is needed. This time we need not to include the current stack object since object *c1* is already known from the previous activity. Thus we copy only *c1* and the method call into a new activity *mcOC2b* in Figure 6. The new activity is appended to *mcOC2a*.

Finally, we consider activity *OC3* which just deletes the already known *unsorted* object. This leads to activity *mcOC3* in Figure 6. Note, a *destroy* marker on the *this* object may look suspicious. However, in Fujaba the *destroy* marker is translated into a *removeYou* operation that isolates the corresponding object such that it may be garbage collected. This does not corrupt the *sort* operation that is still running on the *this* object.

The story board for two cards in Figure 3 is handled just in the same way. This creates activities *mcTC1* through *mcTC4* on the right of Figure 6. The resulting story diagram in Figure 6 is invalid since the start activity has two leaving transitions. However, activities *mcOC1* and *mcTC1* are obviously equivalent and thus they are merged into one activity *F* in Figure 7. Merging two activities attaches all incoming and outgoing transitions to the result activity. Thus, after the merge of activities *mcOC1* and *mcTC1* the new activity *F* has two outgoing transitions to activities *mcOC2a* and *mcTC2a*. Luckily, this inconsistency is again easily resolved by merging these two activities into a new activity *Xa* in Figure 7. Similarly, *mcOC2b* and *mcTC2b* are merged into *Xb*. At this point in time, activity *Xb* has two successors, namely *mcOC3* and *mcTC3a*, which are not structurally equivalent. Thus *mcOC3* and *mcTC3a* can not be merged. Before we consider this conflict, we recognize that activity *mcTC3a* is equivalent to activity *Xa* and may be merged into the latter. This results in the backward transition from *Xb* to *Xa* in Figure 7. At this point in time, activity *Xa* has two successors, namely *Xb* and *mcTC3b* which are equivalent and thus may be merged. Now *Xb* has successors *mcOC3* and *mcTC4* which are equivalent and therefore merged into activity *L*. Thus, Figure 7 results from Figure 6 by systematically merging equivalent activities.

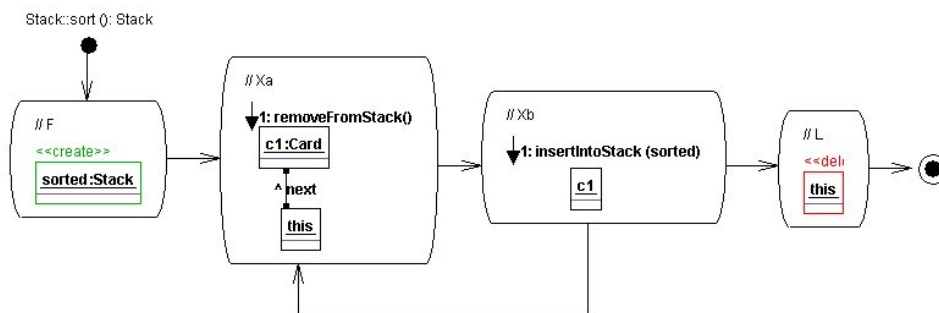


Abbildung 7: Control flow of method *sort*

The story diagram of Figure 7 still contains a conflict since activity *Xb* has two outgoing transitions. This conflict is resolved by adding an appropriate branch condition. If we compare activities *Xa* and *L* and think about it, we find that the difference is whether still a “*first*” object exists in the stack or not. Thus, we add an activity *bc2* consisting just of the *this* variable and a *c1* variable connected by a *first* link, see Figure 8. Systematically, the new branch activity *bc2* is a successor of activity *Xb*. If *bc2* finds a match, we want to proceed with activity *Xa* and thus there should be success transition from *bc2* to *Xa*. For the other case, a failure transition should lead to *L*. Thus, our systematic approach would lead to some kind of *do-while* loop where the loop body is executed at least once. When

we applied our systematic approach to this example, at this point we immediately spotted a weakness. A general sort operation should be able to deal with an empty stack of cards, too. However, we did not provide a story board for the case of sorting an empty stack. Thus, the systematic story diagram derivation did not cover this case. We could now add a story board for the missing case or, as we did, we may just fix the problem manually by moving the branch activity *bc2* to the head of the loop. This results in the story diagram shown in Figure 8. Note, Fujaba translates the story diagram in Figure 8 automatically into a Java implementation of our *sort* method. Thus, after the systematic derivation of story diagrams the implementation of the desired system is already done.

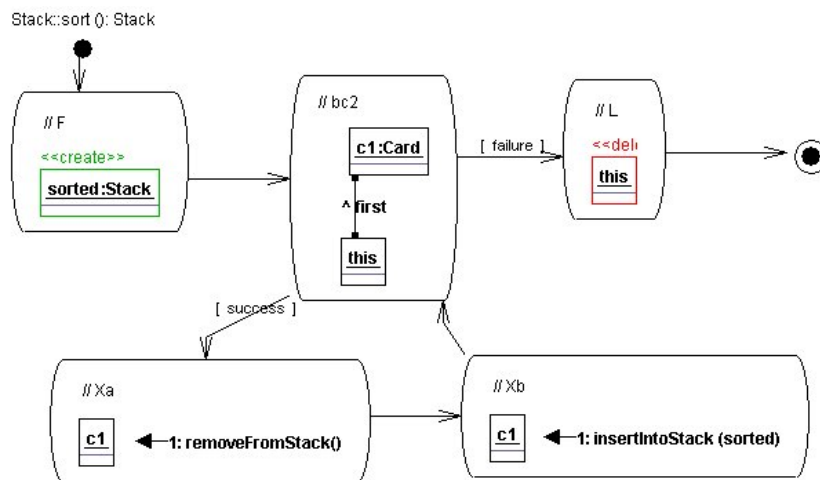


Abbildung 8: Derived method body of method *sort*

5 Summary

This paper introduces Story Driven Modeling as systematic software development approach. Instead of CRC cards, SDM employs a so-called object games. The steps of the object games are protocolled as text scenarios and or as so-called story boards. From these story boards, the developers systematically derive the class diagram. Then behavioral specifications are derived by analysis and comparison of all story boards. Finally, our CASE tool generates executable code out of the derived class diagram and the behavioral specifications. In addition, story boards may be turned into JUnit test specifications, cf. [GZ03]. These JUnit tests ensure that the derived behavior specifications actually realize the provided usecase scenarios.

In contrast to usual software engineering processes as e.g. the Rational Unified Process [JBR99], SDM provides systematic support for the actual software development work.

In [DGMZ02], we used a comparable yet not so elaborated approach for the derivation of statecharts from story boards. In [DGMZ02] method invocations were mapped to tran-

sitions within one target statechart for a whole class. Then the effects of the method call were mapped to *do*-actions of states within that statechart. In [DGMZ02], we also tried to merge equivalent states, however, since the target statechart combines all effects of all method calls to all objects of a certain class, very large statecharts were created and the identification of equivalent states became very complicated. In addition, [DGMZ02] did not yet include the insight, that certain branch conflicts need manual intervention. The automatic approach for branch conflict resolution in [DGMZ02] frequently generated a lot of redundancy within the statechart or even non-connected statechart parts. This paper improves the situation significantly through the idea of mapping effects to different methods. This adds the concept of separation of concern to our approach. If one considers the example of the *removeFromStack* method, this basic method covers two very different cases in the activities *TC2a* and *TC3a* of Figure 3. After method *removeFromStack* has taken the responsibility for these different cases the derivation of method *sort* became much easier, since from the point of view of the *sort* method, now activities *TC2a* and *TC3a* have become equivalent, just a call to method *removeFromStack*. Therefore, the construction of the *sort* method has been facilitated, considerably.

Compared to our previous work in [DGZ02], the merging of equivalent activities within the behavior specifications has become much more systematic. In [DGZ02] we basically used the method of “having a sharp look” to identify similar and recurring activities within story boards and to merge such similar activities to control flow graphs. Therefore, [DGZ02] also proposed to provide mnemonic names for each activity. Actually, this paper just developed this idea by turning these mnemonic names into method calls within that activity. This move then allowed to separate out semantically connected steps into corresponding methods e.g. the different *removeFromStack* activities. In addition, our new approach narrows the comparison of different activities with respect to merging. Merging focuses now on conflicts within the control flow structure of the story diagram of a single method. One usually just looks for activities with multiple successors and considers to merge these successors.

With respect to practical experiences, this paper provides many new guidance for software developers. We frequently use this approach in teaching at highschools and at universities. For novices, we recommend to develop very detailed story boards. This may be done in an iterative approach starting with coarse grain story boards where many things happen within a single step. Then we recommend to split large activities into small ones with very limited effects, e.g. only one link or object created or modified. Then we encourage our students to add mnemonic names for these simple activities and next to turn these mnemonic names into method invocations. These iterations provide the basis for the systematic derivation of behavior specifications using our approach. More experienced developers usually skip this detailed elaboration of story boards. They employ pretty complex activities within story boards that frequently include many method invocations within one activity. Experienced developers are then still able to map different effects within one activity to the different method invocations. They also derive method bodies already from a small number of story boards that do not cover all different cases. More experienced developers prefer to consider missing cases within the method bodies and to modify the control flow manually as we did when we turned the *do-while* loop of our *sort* method into a usual *while* loop to cover

the case of an empty stack without explicitly providing the corresponding story board. However, also more experienced developers get a lot of leverage from using story boards for the conceptual design and behavior analysis. To summarize this point, our approach provide many detailed help for novices. Developers that have climbed the learning curve may easily skip many of these details while still exploiting the general idea. This enables experienced developers to use our ideas for larger and more complex applications.

As next steps, we try to provide more sophisticated tool support for our approach within the Fujaba CASE tool. Fujaba supports our story board and story diagram notation and code generation for the latter. We plan to extend this by an analysis component that looks for method calls within story boards that have not yet contributed to the derivation of method bodies and that finds not yet included invocations of a currently considered method within different story board(activitie)s. In addition the copying of minimal contexts should be supported as well as the identification and merging of equivalent states. In the reverse direction, one may start with a story diagram or one may modify a story diagram and Fujaba should either support the derivation of meaningful story boards or support the adaption of story diagram changes within existing story boards.

Literaturverzeichnis

- [Bo91] G. Booch: Object Oriented Design with Applications, Benjamin/Cummings Publishing Company, Inc, 1991.
- [DGMZ02] I. Diethelm, L. Geiger, T. Maier, A. Zündorf: Turning Collaboration Diagram Strips into Storycharts; Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE 2002, Orlando, Florida, USA, 2002.
- [DGZ02] I. Diethelm, L. Geiger, A. Zündorf: UML im Unterricht: Systematische objektorientierte Problemlösung mit Hilfe von Szenarien am Beispiel der Türme von Hanoi; in Forschungsbeiträge zur "Didaktik der Informatik" - Theorie, Praxis und Evaluation; GI-Lecture Notes, pp. 33-42 (2002)
- [FNT98] T. Fischer, J. Niere, L. Torunski: Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Stroy-Driven-Modeling, Diplomarbeit bei A. Zündorf, Universität-Gesamthochschule Paderborn, 1998.
- [Fu02] Fujaba Homepage, Universität Paderborn, <http://www.fujaba.de/>.
- [GZ03] L. Geiger, A. Zündorf: Transforming Graph Based Scenarios into Graph Transformation Based JUnit Tests, Applications of Graph Transformations with Industrial Relevance (AGTIVE), Charlottesville, Virginia, USA, 2003.
- [JBR99] I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process; Addison-Wesley, ISBN 0-201-57169-2, 1999.
- [KNNZ00] H. Köhler, U. Nickel, J. Niere, A. Zündorf: Integrating UML Diagrams for Production Control Systems; in Proc. of ICSE 2000 - The 22nd International Conference on Software Engineering, June 4-11th, Limerick, Ireland, acm press, pp. 241-251 (2000)
- [SN02] C. Schulte, J. Niere: Thinking in Object Structures: Teaching Modelling in Secondary Schools; in Sixth Workshop on Pedagogies and Tools for Learning Object Oriented Concepts, ECOOP, Malaga, Spanien, 2002.

[Zü01] A. Zündorf: Rigorous Object Oriented Software Development, Habilitation Thesis, University of Paderborn, 2001.