Simplifying Use Case Models using CRUD Patterns

Prashant, Sarika Gupta

Abstract- In this paper, we have presented CRUD, a use-case patterns that is proven useful for developing maintainable and reusable use-case models. These patterns focus on designs and techniques used in high-quality models, and not on how to model specific usages. In CRUD we merge short, simple use cases such as Creating, Reading, Updating, and Deleting pieces of information into a single use case forming a conceptual unit.

krywords— Create data, delete data, information handling, merge use cases, read data, short flow, short use case, simple operation, update data.

I. INTRODUCTION

The `CRUD` pattern consists of one use case, called `CRUD Information` or `Manage Information[1,7]`, modeling all the different operations that can be performed on a piece of information of a certain kind, such as creating, reading, updating, and deleting it. This pattern should be used when all flows contribute to the same business value and are all short and simple. Quite often systems handle information that, from the system's viewpoint, is very easily created in the system. After a simple syntax or type check, and some trivial calculation or business-rule check "Business Rules", the information is simply stored in the system. No advanced calculations, verifications, or information retrieval will have to be performed. The description of the flow is only a few sentences long, and there are probably not more than one or two minor alternative paths in the flow. Reading, updating, or deleting the information are equally simple operations. Each of them can be described in a few sentences.

Such operations can be modeled as use cases and are to be included in the use-case model[10]. If these use cases are not included in the model, some stakeholder will probably miss them; otherwise, the functionality should not have been included in the system in the first place. This does not necessarily mean that this kind of functionality should be expressed as separate use cases. Instead, according to the `CRUD: Complete` pattern, we group them together in so-called `CRUD` use cases, including all four types of operations on some kind of information creation, reading, updating, and deletion of any such information.

This procedure has a few obvious advantages[2]. First, the size of the model will be reduced, which will make it easier to grasp because the number of use cases will be reduced. Second, nobody will be interested in a system containing only a subset of these use cases, for example, read and delete, but not create and update. Grouping these flows together in a single use case called something like `CRUD X` ensures that all four are included in the model, and makes it clear to every reader of the model that this is the use case where all this functionality is captured. Third, the value of each of the separate use cases is very small (if any) for the stakeholders; it is the whole collection of them that gives a value to the stakeholders. Together these use cases form one conceptual unit.

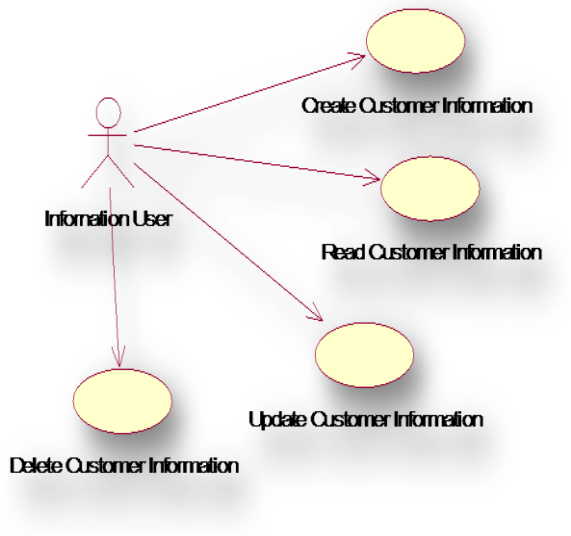

Fig. 1. The four simple operations should not be modeled as separate use cases. Instead, they should be merged into one use case including all four operations as separate flows.

An instance of a `CRUD` use case will perform either a creation, a reading, an updating, or a deletion, and after that it will cease to exist. This instance will not continue to live and wait for the next operation to be performed. That operation will be performed by another instance of the same use case.

A `CRUD` use case may of course include other (basic) flows than the four common ones, such as searching for an item, or performing some simple calculation based on an item. It is important not to merge advanced or complex operations into one use case[3,5]. They should remain separate use cases instead, because they will probably be developed, reviewed, designed, and implemented separately. As a general rule, when not sure whether to merge the different usages into one use case or to keep them as separate use cases, they should no doubt be kept apart. This decision will not affect the functionality of the system, only the model structure and hence its maintenance.

Manuscript received on April 14, 2012.

Prashant*, Assit. Prof.(IT), GCE, Gurgaon, Haryana, India. (prashantvats12345@gmail.com).

Sarika Gupta#, Assit. Prof.(IT), DCE, Gr. Noida, U.P., India. (sarika.mittal0108@gmail.com)

II. AN EXAMPLE IMPLEMENTING A CRUD USECASE

This section provides an example of the `CRUD: Complete` pattern[4,8]. It models the registration of a new task to be performed sometime in the future, the modification of a registered but not performed task, the cancellation of such a task, and the presentation of tasks that either failed during their execution or have not yet been performed Fig.1. As we can see, the four different alternatives are quite simple and short, and they are expressed as four basic flows, because none of them can be said to be superior to the others. Therefore, this is an application of the `CRUD: Complete` pattern even if some of the four basic flows in this case are different from the standard ones.

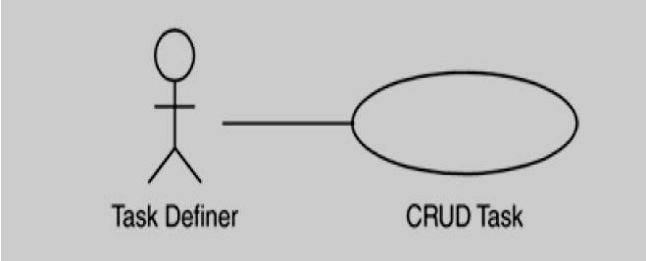

Fig. 2. The CRUD Task use case models the creation, modification, presentation, and cancellation of a task.

Error handling and exceptional flows[6,7] are expressed as alternative flows of the use case. The Item Look-Up and the Future Task blueprints are also useful in this example.

Use Case: CRUD Task

Brief Description

The use case registers, modifies, or cancels the information about a task to be performed as stated in information received from the Task Definer.

Basic Flow

The use case has four different basic flows:

- Register Task
- Modify Existing Task
- Cancel Task
- View Tasks That Failed

REGISTER TASK

The use case starts when the Task Definer chooses to register a new task. The use case presents a list of possible kinds of tasks to the Task Definer, and asks what kind of task is to be registered, what name it is to be assigned, and when it is to be performed.

The Task Definer enters the required information. The use case checks whether the specified time is in the future and whether the name of the task is unique.

The use case registers a new task in the system and marks the task as enabled.

The use case ends.

MODIFY EXISTING TASK

The use case starts when the Task Definer chooses to modify an already registered task. The use case retrieves the names of all the tasks not marked as active and presents them to the Task Definer.

The Task Definer selects one of the tasks. The use case retrieves the information about the task and presents it to the Task Definer.

The Task Definer modifies any of the presented information except the name of the task.

The Task Definer accepts the information. The use case checks whether the specified time is in the future and, if so, stores the modified information.

The use case ends.

CANCEL TASK

The use case starts when the Task Definer chooses to cancel a task. The use case retrieves all the tasks not marked as active.

The Task Definer selects one of the tasks. The use case retrieves the information about the task and presents it to the Task Definer.

If the Task Definer confirms the cancellation, the use case removes the task; otherwise, no modifications are made.

The use case ends.

VIEW TASKS THAT FAILED

The use case starts when the Task Definer chooses to view a list of all the tasks that have failed. The use case collects all the tasks with the status failed and presents their names to the Task Definer.

The use case ends.

Alternative Flows

CANCEL OPERATION

The Task Definer may choose to cancel the operation at any time during the use case, in which case any gathered information is discarded, and the use case ends.

INCORRECT NAME OR TIME

If the name of the task is performed not unique or the time is not in the future, the Task Definer is notified that the information is incorrect and is requested to re-enter the incorrect information.

The Task Definer re-enters the information. The flow resumes where the check of the information is performed.

III. AN ANALYSIS MODEL OF A CRUD USECASE

The analysis model[4,8,10] of a CRUD use case is based on all the flows of the use case. It must include the realization of all the basic flows as well as the realization of the alternative flows.

However, because the flows are quite simple in this case, the model usually contains one boundary class[9] for presentation and modification of the information, one control class to handle any checks, and one entity class or a few to store the information. The flow starts in the System Form when the Information User requests to create, read, update, or delete the information. An instance of the Information Handler is created, which opens an instance of the Information Form to the Information User.

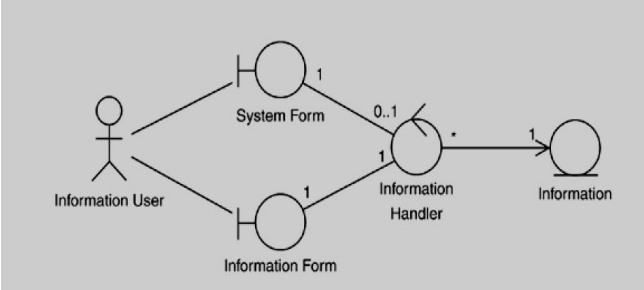

Fig. 3. An analysis model of a CRUD use case.

Depending on the chosen operation, the actor enters information about a new piece of information, or the Information Handler retrieves existing information and asks the Information User via the Information Form to select one item. The Information Form sends the new information or the identity of the selected item, respectively, to the Information Handler which performs the chosen operation. Finally, the Information Form and the Information Handler are removed and the use case ends.

IV. CONCLUSION

Here in this paper, we have studied and explained the CRUDE a use-case patterns that is proven useful when developing maintainable and reusable use-case models. These patterns focus on designs and techniques used in high-quality models. Further how we can prepare analysis model of a CRUD Usecase has also been discussed.

REFERENCES

1. Adolph, S., and P. Bramble . 2002. Patterns for effective use cases.Addison-Wesley.
2. Alexander, C., S. Ishikawa, and M. Silverstein . 1977. A pattern language: towns, buildings, construction. Oxford University Press.
3. Bass, L., P. Clements, and R. Kazman . 2003. Software architecture in practice. Addison-Wesley.
4. Bittner, K., and I. Spence . 2002. Use case modeling. Addison-Wesley.
5. Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal . 1996. Pattern-oriented software architecture, volume 1: a system of patterns. John Wiley and Sons.
6. Jacobson, I. Concepts for modeling large real time systems. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden.
7. Jacobson, I."Object-oriented development in an industrial environment." Proceedings of OOPSLA'87. Sigplan Notices 22(12) :183191.
8. Jacobson, I. 2003 (March). "Use cases yesterday, today, and tomorrow." The Rational Edge.
9. Jacobson, I., G. Booch, and J. Rumbaugh . 1999. The unified software development process. Addison-Wesley.
10. Jacobson, I., M. Christerson, P. Jonsson, and G. Övergaard . 1993. Object-oriented software engineering: a use- case driven approach. Addison-Wesley.