Relatório Técnico

Núcleo de Computação Eletrônica

Robustness Diagram: A Bridge Between Business Modelling and System Design

Pais, A. P. V.
Oliveira, C. E. T.
Leite, P. H. P. M.

NCE - 07/01

Universidade Federal do Rio de Janeiro

Robustness Diagram: A Bridge Between Business Modelling and System Design

Pais, A.P.V., Oliveira, C.E.T., Leite, P.H.P.M.
NCE - UFRJ - Universidade Federal do Rio de Janeiro
Brazil

Abstract

Use case driven development has proven being a good approach for capturing problem semantics in an orderly, structured description. However, it specifies an abstraction beyond practicity to guide system design process. Robustness diagrams are a simple solution for drafting a more formal description for business modelling. This very simplicity may, however, detract its value, falling short of capturing the rich business semantics. Stereotyping is the essence behind its robustness diagram mechanics. Symbols convey the abstraction necessary to catch the model semantics. Increasing the number of stereotypes we can achieve a closer match from model to design. Rules carefully stated for robustness diagram can help to translate high-level information into well-behaved and predictable symbolic descriptions. This enhancement to robustness diagram helps to patch the gap between abstract model and project into a paved continuum. It has been used to train programmers to extract working and consistent systems out of use case specifications.

1 Introduction

Use case driven development [1] is a good approach for capturing problem semantics in an orderly, structured description. However, use case scripts, despite matching closely just problem and requirements formulation, are still in a level too high to guide system implementation. Lack of formality and profusion of formats and templates used to write those scripts sums up to increase the gap between problem and solution.

Use case analysis is simple and intuitive and people can feel comfortable in using it to capture user interaction design. Paradoxically, it is hard to train people into it, since concepts are mostly informal and divergent opinions are formed around them. Concepts like "uses" and "extends" have subject interpretation, and due to conflicting interpretations, it have been even suggested to substitute them for other ideas[1]. Use case scripts have the same problem of many recommended forms. In fact, this constitutes the very power of UML [2], being flexible enough to accommodate to the prevailing culture in each development team. On the bad side, interpretation of these notations can be confusing and frustrating. The solution for this problem is a validating framework that can correlate use case free style with a more formal description. Use case could assume any form, but there should be a means to associate each statement and convention to a corresponding formal analysis element.

In the quest to fill this void, robustness diagrams [1] comes as a first candidate to do the job. Its simplicity is helpful for drafting a more formal description for business modelling. This very simplicity may, however, detract its value, falling short of capturing the rich semantics of scripts and the same time being too laconic about implementation issues. Fortunately, the robustness diagram carries in itself the cure for these deficiencies. Stereotyping [3] is the essence behind its mechanics. Symbols convey the abstraction that makes the necessary link to the model semantics at the same time that classifying objects in niches contributes towards the formalisation of ideas. Assuming a linear approach, increasing the number of stereotypes we can construct a rich vocabulary capable of capturing use case semantics in greater resolution. Drawing a collection of syntactic rules to bind these symbols together can also increase formality. Rules carefully stated for robustness diagram can help to translate high-level informal abstraction of use cases into well-behaved and predictable symbolic descriptions. On the other way around, constraints imposed by these rules help to revise informal text descriptions into more accurate and detailed specifications.

This proposal depicts a metaphoric architecture on the top of which new stereotypes are derived from the traditional boundary, control, and entity trinity [1]. A set of rules defines their interrelation and also hints their

correlation to semantic constructs of natural language description from use cases. Each new stereotype has an associated counterpart in the underlying architecture, conforming to a computer assembly metaphor. This enhancement to robustness diagram is helping to patch the gap between analysis and implementation into a paved continuum. Is has been used to train inexperienced programmers to extract working and consistent systems out of use case specifications. A couple of weeks training is enough to obtain consistent results that are being coded into production information systems.

2 Description

2.1 Starting use case driven analysis

Use cases start with requirement analysis. For this purpose, a tool, named e-Doc [4,5], was developed with the intent to capture requirements and associate them to use cases design. This tool is a web-based application capable of assemble together requirements and use cases issued by both business analysts and software developers. The teams can cooperate to develop the specifications, store them in a database and feed the results into a UML meta-model.

E-Doc captures meta-model data from Rational Rose [6] and displays in HTML format. It shows the interaction between requirements, actors and use cases. Use case information may be scattered among many diagrams, but the tool has the responsibility to show the information in an integrated format. Use case scripts appear together with requirements and diagrams in a web page. Information is immediately available to all participants, featuring a cooperative development environment web wide.

After use case specifications completion, the methodology adopted indicates the use of robustness diagrams to represent graphically the information described in use case scripts. This task has the purpose of guiding the analyst towards the design phase. The goal is to capture a detailed picture of user interaction steps into this symbolic diagram.

On the other way around, robustness analysis [1] enforces the validation of use case scripts. Feedback from robustness analysis can help to correct ambiguous statements, disconnected phrases or incomplete information. Cross checking between script and robustness diagram turns out in a refinement of use case specification and early detection of entity classes candidates for static diagram construction.

2.2 Robustness stereotypes for Objectory process:

Boundary Objects: Objects with which the actors will be interacting, like: windows, screens, dialogs, menus, buttons, etc.

Entity Objects: Represent the domain objects of your system. Many of your entity objects will come from your domain model. Eventually, new entity classes can arise out of robustness analysis.

Control Objects: Represent the business rules and the logic associated with the use case texts.

Figure 1 – Robustness Stereotypes

The rules that govern the associations between objects in the robustness diagram, according to the Objectory process [7] proposed by Jacobson, are:

- Actors can only be associated to boundary objects.
- Boundary objects can only be associated to controllers and actors.
- Entity objects can only be associated to controllers.
- Controllers can interact with both boundary objects and others controllers, but never with actors.

2.3 Proposal for new robustness stereotypes

Informal natural language descriptions are loaded with meaning that may escape from the three traditional stereotypes. Some more symbols could help to catch those elusive intentions behind use case scripts. The

target is not to build a whole ideographical system but fill in some essential gaps that hold some association with design level implementation. The basement for this proposal is a generic architecture devised to cover the average needs of information systems. Elements of this architecture that could be associated with use case statements were chosen to form the new vocabulary.

Rack Object: Represents the use case controller. Represents explicitly the use case itself and controls the execution flow throughout script scenarios. Invokes methods on boundary and entity objects to carry interaction and persistency/query tasks. Methods get invoked on it to perform multi-object validation of business rules.

Slot Object: Represents a stated assumed in a use case sequence. Normally can be identified from the script by a sentence that invokes the appearance of a new window or frame in a frameset. Any possible state or boundary appearance must be tracked down to a slot object, considering the main course, alternative, and exceptions. Each GUI invocation that changes the use case state should produce a corresponding slot object.

Board Object: Performs the role of a remote proxy, providing a local representative for an object in a different address space. The board acts as a flat and trimmed down representation of the real complex entity. It is easier to identify from the use case script, since it looks like the user sees the object, with fewer attributes, no aggregations or one to many relationships. The real entities can appear further on in the robustness- script interaction, since the factored parts of the composite may get involved in the use case. The board stands locally for validation into its data and becomes the target of exceptional course scenarios. Robustness validation rules forbid real entities to relate with slots, so that boards must be created for user interaction. Main course scenario will eventually establish the appearance of the entity behind the board since the validation of post condition will require the board to save itself into the entity.

Boundary Action Object: These are interface objects that capture and propagate events in the system. They represent a direct intervention of an external actor, invoking methods from objects that might result in a state change.

Guard Object: Essentially are decision control objects, determining a change in execution flow. The resulting action may cause the alternation to a different use case scenario. They will be converted to methods associated with objects they attach to.

Figure 2 - New Robustness Stereotypes

2.3.1 Auxiliary stereotypes

Some extra stereotypes were added to aid design automation. These stereotypes are markers used by automation scripts to detect especial conditions during diagram or code generation.

Exit Object: This is a marker stereotype that explicitates the termination of a use case scenario. It is used by sequence or state diagram generators to detect the end point for a scanning algorithm. It adds to robustness analysis by signalling visually the point where a post condition must hold or a scenario has aborted.

Exception Object: The exception object is a stereotype representing an exception condition established by the developer resulting from unfulfillment of business rules contract. It does not stand for system error exceptions, which are not accounted at this level. Exceptions are markers for fault scenarios explicitation, leading to a better use case understanding. They represent a validation feedback that alternate or exceptional scenarios have been accounted for. All scenarios can be tracked and can be matched to script notation for it, independently of templates or scripting style adopted. Automated code generation can be devised to produce a standard treatment of scenario failure based on this stereotype.

2.4 Association of the script with the symbols

The robustness diagram is build based on the corresponding use case script. Like the sequence diagram, associations are created among objects added in the diagram, conveying the semantics of the use case.

The diagram creation starts with a group of associations among ideas and symbols. Group of words that compose the text of the use case are them related to a corresponding stereotype that will be part of the diagram.
For example, the name of the use case presented in the diagram is associated to a rack object of same name.
Words as " window ", " dialogue box", or any other that represents a visual element, will be presented as a boundary object.
Words as "button", "menu item", or any other visual element conveying user's interaction will be shown as a boundary action associated with a boundary element.
Verbs as " it presents ", " show ", " exhibits ", preceding words that represent boundary objects, certainly relate to slots, representing a state of the use case.
Words that are part of the system domain will be presented as board objects, connected to the corresponding entities and slots.
Verbs that indicate action on other objects, or execution of routines, will be represented by control objects.
Sentences that represent conditional forks of the type " if ... then ... else..." will be shown as guard objects that divide the execution flow in two different paths.
The exceptions contained in the use case script will be presented in the diagram as exception objects, normally linked to one guard object responsible for some type of validation.
As example we can present a simple piece of use case script and the respective objects associates:
"USE CASE 1: Changing user password (rack)"
"1. The system presents (slot) a screen (boundary) filled with the user (board and entity) name, one edit box for the input of the old password, another one for the input of the new password, and another one for the confirmation of the new password.
2. The user fills the data and presses the OK button (boundary action) to modify the password (E1).
3. The password of the user is altered (control) and the use case finishes (exit).
4. When pressing the CANCEL button (boundary action) the use case finishes (exit).
E1. If the user fills the new password and confirmation fields with different information (guard), one dialog box is shown (exception) showing the error, and requesting the new data."

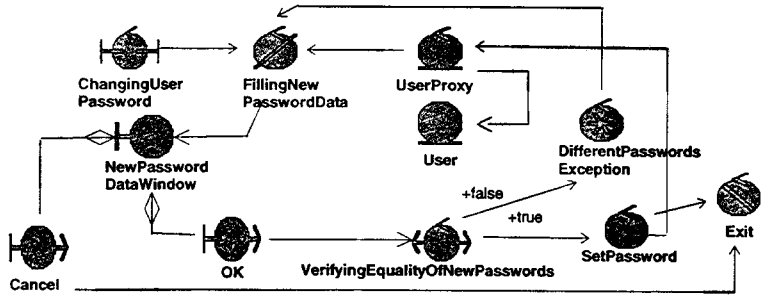

Figure 3 - Sample diagram extracted from use case script

2.5 Refinement of Use Case Scripts

The creation process of the robustness diagrams also helps to consolidate use case scripts. Vague names used in the script are changed by other, more specific in the context of the use case. Incomplete expressions are changed by a detailed description of the system operation.
For example, in the use case mentioned above, we would have to rewrite the script in the following form:
"USE CASE 1: Altering password of the user (rack)"
"1. The system presents a window for the entry of the new password data, filled with the name of the user, one edit box for the input of the old password, another one for the input of the new password, and another one for the confirmation of the new password.
2. The user fills the data and presses the OK button to modify the password (E1).
3. The system verifies the equality between the new password and its confirmation, and in case that they are equal, it executes a method to modify the password of the user for the new value.
4. When pressing the CANCEL button (boundary action) the use case finishes.

E1. If the system verifies that exists a difference between the new password and its confirmation of the same, the use case goes for an exception state, where one dialog box is presented showing the error, and requesting the new fulfilling. "

2.6 Association rules between the stereotypes:

Association rules involving stereotypes in a robustness diagram are devised to enforce formality. They also serve as a guide in the transformation of an informal abstraction into a precise description. Below, we describe a set of association rules that govern the proposed architecture:

1. The rack object represents the use case itself.
2. The rack object must be unique in a diagram and it should be connected to the slot object of the first window to be presented in the use case.
3. Every slot object must be connected to a boundary window.
4. A board object can only be connected to one entity object. This board object represents the proxy for this entity.
5. Board object is only necessary when entity attributes need to be presented in the interface.
6. Every board object is located between a slot object and an entity object.
7. A board object can be connected to more than one slot objects, if the entity data appears in more than a window.
8. A boundary action object must be connected to the boundary container object in which it appears.
9. A boundary action can only invoke a slot, a control or a guard.
10. An exception object can only be located after a guard object, in one of its execution paths.
11. The use cases called in the diagram are represented as use cases, not as rack objects.

3 Assessment of the architecture

Robustness diagrams, as any other diagram, are the expression of ideas in a language formed by a vocabulary of interconnected symbols governed by a set of rules. A richer vocabulary can enhance the expressivity of the language. Increasing the resolution of robustness vocabulary we expect a better tracking of information detail, resulting in a more accurate transcription of use case semantics. This higher fidelity enables inferences on this diagram to raise other UML diagrams with greater consistency.

The robustness validation rules were mainly derived from concepts extracted from the underlying MVC architecture [8]. Rule extensions emanate from the architecture proposed, where a more detailed framework envisages early guidance of analysis towards an easily identifiable set of architectural patterns.

3.1 Robustness rules validation

Robustness rules are the tools for achieving consistency of use case descriptions. A new set of rules is attached to the new vocabulary of stereotypes. Although have being used successfully, formal validation is still pending. Rules validation can be approached from two distinct contexts. In a context, these rules must prove to produce a useful robustness analysis. Rules themselves must be tested and checked against each situation to assess the applicability and usefulness of imposed constraints. Architecture must be confronted against user interface models seeking for adjustments or creation of new rules. This can be considered as long-term task, since prospection of possible situations is beyond any formal evaluation. The natural way is to submit the paradigm to a series of real challenges and get the adjustments along the way.

From another point of view, existing rules can be validated during usage on a modelling session. Rules impose conditions and restrictions that may get violated as the designer constructs the diagram. Existing diagrams can be scanned for faulty relationships between elements and tools that forbid rules violation can assist construction of new diagrams. A tool can assist designers in making the right choice disabling connection from current stereotypes to incorrect destinations. A textual hint can be displayed stating the reasons disallowing the operation. In an example, boundary actions cannot be connected directly to a boundary without the interposition of a slot in between. An explanatory hint would state that a new screen cannot be instantiated without the acknowledgement of a control element. This feedback would avoid a bad design and introduce refinements into the use case script.

3.2 Covering alternative paradigms

Robustness analysis is built on top of model-view-control paradigm. Rule enforcement guides the designer towards conceptual MVC architecture design. This can be a limitation, where some particular solutions may not fit into this paradigm. Simple systems may deviate from MVC requirements, opting for constructs that violate robustness rules.

RAD and web systems are cases where model view relationships prevail over traditional MVC. These systems rely on direct connections between view and model, where control is already embedded into one or another and does not play any role in use case description. Web systems mostly rely on sessionless connections where state is not preserved across interactions. The user simply navigates to the desired page, filling a form and submits it to storage in a database.

The intention of a control in the robustness diagram is to represent the invocation of a method. This only makes sense if there is a correspondence to use case script. Although actually a method must be called to perform the operation, there is no meaningful correspondence in the script. To support these alternative architectures, rules can be relaxed to accommodate the situation.

3.3 Source code generation - successes and failures

The main idea behind this work is to enhance robustness diagram expressivity to a point close enough to the design level. The desired consequence is that it now can be automatically translated to other languages. The enhanced diagram can be scanned to generate other UML. This process can be extended up to the generation of a complete system.

UML model diagrams have a well-defined structure, so that is possible to identify relationships between elements from distinct diagram representations. These correlations can be used to refine information down to the point of automatic code generation. Exploiting the enhanced robustness diagram together with other diagrams that can be extracted and individually refined we can achieve a high degree of automation. One concrete example is the extraction of a state diagram from the robustness analysis that generates a working implementation of state machine, mapping consistently use case mechanics.

3.4 State diagrams

State diagram extraction is possible through the interpretation of robustness element relationship. The architecture behind the proposed stereotypes includes for this purpose a state machine. State and transitions are identifiable by matching elements and associations in the robustness diagram. State diagram construction starts with the identification of states among the stereotypes. The state in a robustness diagram comes from use case perspective. Use case is about user interaction and from user perspective, state change is perceived by screen changes. As stated by robustness rules, each boundary must be attached to a slot element. The slot stands for the state in the architecture state machine, so identification of states is trivial.

A boundary action by definition is responsible for events initiated by the user. Boundary actions activate events expected to provoke changes in system state. Transitions can be tracked by interpreting robustness diagrams as an oriented graph. The elements are the node and the associations are the arcs. A path between two slots containing a connection with a boundary action can identify a transition. Navigability stated in associations determines the transition orientation. Guard stereotypes encountered alongside the path are account for guard conditions.

At present, automatic generation produces overstated or incomplete state diagrams, but consistent with use case description. Incompleteness is due to actions associated with transitions that may escape from trivial heuristics. Overstated transitions are due to the lack of sequence in robustness associations. Tracking algorithms may loop through paths belonging to different scenarios forking from the same element. The solution can be built in assisting tools that label associations according to the scenario being transcribed into the robustness diagram.

The resulting state diagram although consistent with use case semantics still needs to be refined by the designer. Approximation is good and minimal intervention is required. We expect that improvements in heuristics and robustness notation and construction process can produce fully automated designs.

3.5 Business rules

Business rules specification is still a weak point in robustness analysis, mainly because use case abstraction looks over it most of the time. Business rules are abstracted in an effort to focus on the interaction and achieve a better understanding of the overall picture. Robustness diagrams omit them to avoid cluttering or even because there is no representation available for the details. Even when business rules are depicted, they hold a superficial representation.
Robustness diagram represents business rules as validations to be performed over a dataset. These validations are represented by control stereotypes with a terse statement. Moreover, there is a lack of support for detailed business rules specification, only an indication that they must be verified. Detailed descriptions are left to sequence diagrams. In a sequence diagram, business rules can be represented by a message invoked on a controller or entity class followed by a sequence of interactions with collaborating classes.
Robustness diagrams emanated directly from use cases are evasive about events that are not initiated by the user. Such event specification are restricted to requirements and omitted from use case scripts. The fact that a business rule can only be modelled further down into the design level, skipping robustness analysis, constitutes a chasm between analysis and implementation. This can be a major drawback for utilisation of robustness diagram as a starting point to automatic creation of remaining diagrams and subsequent system implementation.
Robustness diagram limitations on non-interactive events also derive from the underlying MVC architecture. According to MVC paradigm, controllers are in charge at the architecture's core, but only take action under stimulus coming from user interacting with view elements. Autonomous events must have a representation in order to drive the controllers into action. A thread stereotype may introduce this concept, which may cater for three different needs of uncharted business rules.

4 Conclusion

High demand on quality responsive software, capable of fulfiling accurately user demands is driving the quest for tuning between business modelling and system design. Faithful accomplishment of client requirements is a measure of a software market value. Use case driven development shows its importance as an accurate vehicle to transit system specifications between the client and the provider. However, the danger of misunderstanding of the use case message down into the model refinement may put all the effort to waste.
Robustness diagrams are a provisional solution, with insufficient resources to solve the problem. To improve safety in this transition we propose a wider set of stereotypes sealing the holes where semantics was leaking. These symbols were conceived on top of solid architectural constructs, pervasive on most seasoned designs. Development was carried towards correctness by construction, obtained by design automation.
Filling the initial gap is important, but the research is directed to guide the whole process through closed loop control. Automated steps are interposed by human assessment of forward progression and feedback check of use case completeness. This process has being put to work in the reengineering of a large legacy academic management system producing initial positive results. The enhanced robustness analysis has been implemented in a professional modelling tool and scripts produce all logical diagrams automatically from the robustness diagram.

References

1. Rosemberg, D., Scott, K.; Use Case Driven Object Modeling With UML: A Practical Approach; Addison-Wesley; 1999
2. Page-Jones, M.: Fundamentals of Object-Oriented Design in UML(2000) - Dorset House Publishing
3. Fowler M., Scoott K.: UML Distilled - A Brief Guide to the Standard Object Modeling Language
4. Oliveira, R. F., Lobo, S. B., Teles, V. M.: e-Doc: Sistema de Apoio a Documentação de Modelos de Projeto de Software Orientado a Objetos que Utilizam a UML como Linguagem Padrão de Modelagem (Integrado a Ferramenta CASE Rational Rose) – B.Sc dissertation UFRJ 2000
5. http://tedmos.nce.ufrj.br/uml/aulas/aulas2/eDoc/index.htm
6. Kruchten, P. : Rational Unified Process: An Introduction, 2 edition (2000) - Dorset House Publishing
7. Jacobson, I.: The Unified Software Development Process(1999) - Dorset House Publishing
8. "Developer's Guide - Borland - Jbuilder 2", Borland, 1998.