Use Case-Based Software Development

Peter Haumer
IBM Rational Software
peter@haumer.net
http://haumer.net

Biography & Photograph:

Dr. Peter Haumer is a Content Developer for the IBM Rational Unified Process product platform. Currently, he is working as the content architect for the future generations of IBM's integrated process architecture. Before joining the RUP team, he worked as a Senior Professional Services Consultant for IBM's Rational Software Brand. He assisted and coached customers to be successful with the Rational Unified Process platform and Rational tools performing on-site consulting and providing training courses. His areas of work include requirements management, object-oriented analysis and design for enterprise application architectures, as well as Software process implementation. He is also member of Rational's steering committees for Model-Driven Development, Software Process Adoption, as well as Business Modelling, Requirements Management, and Rational XDE education. Before joining Rational he worked in basic research in the areas of requirements engineering and flexible CASE tool architectures.

Keywords:

Use Cases, Requirements, Object-Orientation, Analysis, Design, UML, Model-Driven Development, Rational Unified Process.

Summary:

Use case authors have the difficult task of writing their specifications for many different audiences. They have to balance language, content, and style of their use cases for the needs of non-technical as well as technical stakeholders. While most chapters in this book focus on the non-technical stakeholders, such as customers and users, this chapter discusses the developer as another important audience of use cases. To be able to successfully write use cases that are also usable for this audience the System Analyst has to understand what the developers are actually going to do with the use cases. Knowing what information is required will improve the quality of the use cases for the development team. (The difficulty will still be, of course, to keep the balance for the other audiences to not to negate the main benefit of use cases: to facilitate interdisciplinary communication of requirements.)

I will outline in this chapter the role Use Cases play for Object-Oriented Analysis and Design (OOAD) as it is being described in the Rational Unified Process (RUP 2003). I will begin by discussing the particular properties use cases possess that make them such an ideal tool for requirements management as well as guiding software design. I will define additional concepts that are also required to complement use cases for these tasks. Next, I will walk you through a core set of analysis and design activities, examining the role use cases play for these activities, using examples from a web based e-commerce application.

Applicability:

- Projects with interdisciplinary team communication.
- Projects utilising object technology and modern design frameworks (e.g. J2EE, .NET).
- Wide range of software and systems-engineering projects in terms of scale from small web-based e-business applications via business process engineering projects to large-scale systems engineering projects (the latter two applying a use case flow-down between systems and sub-systems not discussed in this chapter; see (Cantor 2003) and (RUP SE 2003) for more details).

Key Features:

- Industry leading software development process.
- Iterative, risk, and use case-driven analysis and design methodology.
- Full traceability from requirements to analysis to design to code.
- Augments requirements with user experience modelling to improve communication on requirements and designs.
- Facilitates predictable and repeatable iterative development through analysis and design mechanism and trace-based impact assessment.
- Extensive tool support available (not discussed here).

Strengths:

- Use cases present requirements in context improving communication within interdisciplinary teams by focusing on delivering actor value and not technical detail.
- Use cases relate to stakeholder goals which facilitate prioritisation and planning of iterative development (implementing high priority and risky requirements first).
- RUP OOAD defines well-understood and scalable process of transforming requirements into software solutions.
- RUP OOAD systematically established traceability as part of the primary activities to support iterative development as well as change impact-assessment and management.

Weaknesses:

- Reading and understanding use cases should be easy for any stakeholder. However, modelling and writing use cases that they are easily understood as well as successfully used for analysis and design requires a well-trained system analyst and designer.
- As a result, experience is required to avoid typical pitfalls when adopting use cases and OOAD such as functional decomposition of use cases, over-analysis of use cases and analysis models, avoiding the right amount of details in use cases, mixing requirements and design decisions, not using the right amount of analysis mechanisms to abstract from design decisions in the right places, not enforcing design mechanisms in the teams to systematically incorporate design decisions that lead to repeatable and maintainable designs, etc.

- RUP OOAD also requires an architect with solid methodological as well as technical understanding leading the design with experience, vision, and creativity.

Technique and Worked Example:

The Use Case-Centred Requirements Framework

I will start this chapter by listing and defining the key elements a designer needs to be provided with to do OOAD, i.e. the inputs to this discipline of software engineering.

Goals and Requirements

First off, let's talk about requirements and goals in general (just to avoid any misunderstandings with other sources and definitions of these terms):

> A *requirement* specifies a solution in terms of behaviour (e.g. input-output interactions), structure (e.g. exchanged and stored information, business rules), and quality (e.g. usability, performance) from a black box perspective.

A solution addresses goals stakeholders[1] have in respect to their environment, e.g. the goal to achieve or to improve something.

> A *goal* describes a solution-independent measurable desired state for a stakeholder's environment.

The goal allows stakeholders to express their needs for their world or application domain (e.g. their business, their automation environment, etc.) for which a solution or an improvement to an existing solution can be defined with requirements. Requirements address or realise the solution-independent goals with concrete solution specifications. Example for a goal: "Provide automated order handling." Example for a requirement addressing this goal (amongst many another requirements): "The sales system displays the types of available order transactions. The customer selects a transaction type. The sales web displays the list of books that the customer currently has pending." You see that one can find many different alternative solutions for a goal, e.g. the goal above could have also been addressed with a phone line utilizing touch tone codes or speech recognition requiring quite different set of input-output interactions. The mapping of goals to high-level requirements that have been scoped to be part of a project, which is establishing the solution, is documented in what the RUP calls a *Vision Document* (RUP 2003).

System Boundaries

Based on these definitions, it is not possible to write requirements without exactly knowing what the system boundary is going to be. Certainly, in early phases of a project when the boundary of the system is not clearly defined, yet, all kinds of information will be gathered from stakeholders about what needs to be achieved and done by a potential solution. However, until the boundary is defined, we will call these *stakeholder requests*. Once, the boundary is defined one can then map stakeholder requests to goals, requirements, or statements about the design.

The notion of system boundary plays an important role for organising our specifications especially in the case of projects in which a system is going to be established

[1] An individual who is affected by the outcome of the project.

that is actually decomposed into more than one sub-system. The (RUP SE 2003) defines a system as follows:

> A *system*, which may contain hardware (computational and non-computational), software and human workers, delivers some needed operational capability to its users. A system is composed of sub-systems, which collaborate to yield the behaviour and characteristics of the system.

Therefore, to be able to scale development of systems you need to produce a separate requirements specification for every sub-system, because the realisation of such a subsystem could be outsourced to another development team, might involve teams with completely different skill sets, as well as testing has to be done on each sub-system level. If you regard every system as a box for which you produce a black box requirements specification, you could depict a decomposed system as in the example of Figure 1 below.

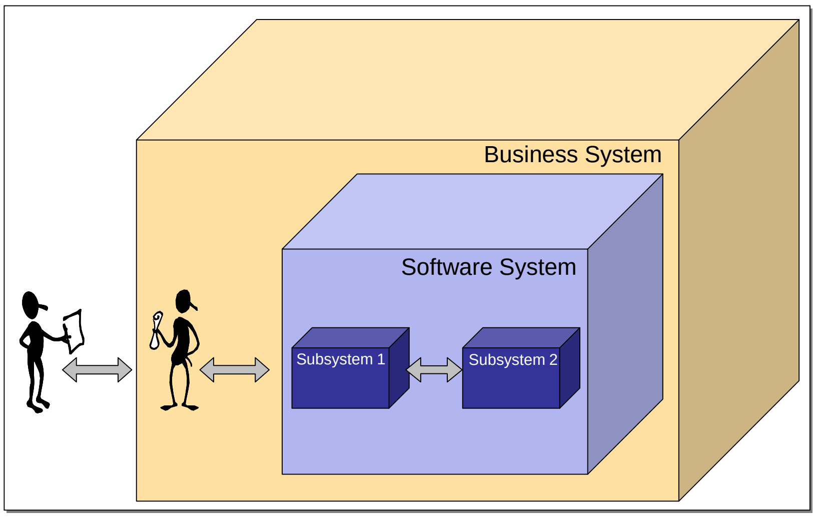


Figure 1: A Decomposed system.

This simple example shows three levels of decomposed systems of three different kinds:

- A *business system*, i.e. a business environment for business workers, for which the black box perspective comprises of the services the business has to offer to its customers (also often referred to as *Business Processes*).
- A *software system* that is installed and operated by business workers as well as other systems as part of the business realisation.
- A composition of the software system in *software sub-systems* (which realisations are also often referred to as *Software Components*).

Again, for all of these levels you specify requirements in terms of the functionality provided by the system's interfaces as well as qualities (also often referred to as *Non-Functional Requirements*) such as usability, reliability, performance, etc. Needless to say that a real system (think of a whole financial institution) might involve many more levels including hardware with embedded software, business units or departments as sub-systems of the business.

Use Cases

Now, we are finally ready to talk about how the interfaces between the boxes and their users (called *Actors*) of Figure 1 should be specified, which leads us to the definition of, in respect to this book, the most import concept:

> A *use case* defines a sequence of actions a system performs that yields an observable result of value to a particular actor.

In other words, a use case describes primarily functional but also non-functional requirements from the perspective of an actor achieving particular goals. In contrast to more classical styles of specifying requirements, use cases define clusters of requirements based on goals, i.e. a use case groups together all requirements for a solution to achieve one or more particular stakeholder goals.

Because there is always a many-to-many relationship between goals and requirements you as System Analysts have to decide which goals of the overall goal set defined for a project you are going to use to derive the use cases from. Your general guideline would be to check if your use cases communicate the most important actor goals that will be addressed by the system, i.e. the goals you can use to "sell" your system to stakeholders. The trick here, that makes use cases such a successful interdisciplinary requirements elicitation and validation technique, is the idea to present to stakeholders what they are primarily interested in: their customers' or their own goals, rather than grouping and presenting requirements based on some technical criteria or the solution's components or structure. As expressed in (Beyer et al. 1997) in contrast to technology experts which know and "like" technology, stakeholders tend to merely have the objective to get their jobs done and want computers to be as invisible as a pen's ballpoint so they can focus on their tasks.

As such, use cases cannot be decomposed into sub-units. They represent one unit of addressing needs or goals. Functional decomposition is a matter of Analysis that I describe in its own section on Page 10 using completely different concepts. Always remember, working with use cases is synthesis, not analysis. You do not break a use case apart like problems that are decomposed into smaller problems following a divide and conquer strategy. A use case describes a solution, synthesizing all the interactions necessary to achieve the goal expressed in its name. (Bittner et al. 2003) convey this by saying that a use case provides you with a complete experience that result in real value for at least one actor. If you need to link use cases together in order to provide value, your use cases degrade into hard to validate context-free functions and those high-level use cases you use to link them together degrade into empty shells.

Use Case Diagrams

Use case specifications are made up of graphical and textual parts. Let's have a look at an example of a graphical UML use case representation.

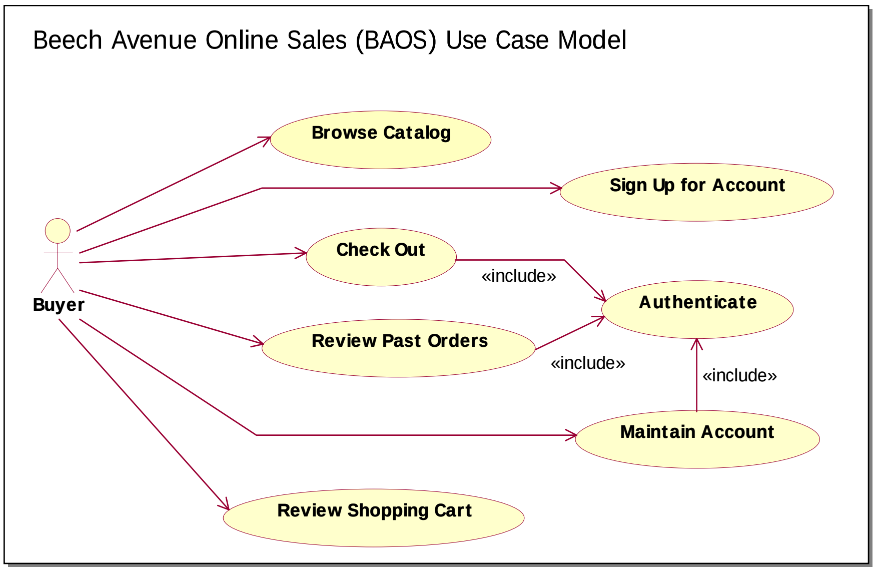


Figure 2: A Use Case diagram.

Figure 2 depicts use cases, in a UML use case diagram, for an online sales software system from the actor Buyer's perspective. Typical goals of buyers for online shopping are to browse a catalogue, maintain their shopping cart, check out their orders, review past orders, etc. As you can see in Figure 2 use case names are chosen to exactly express these goals. You can also see that use cases can depend on each other as the Check Out use cases depends on the Authenticate use case. Modelling these dependencies "smells" of functional decomposition and indeed use case relationships should be used in only a few circumstances. Using them in an extensive manner is a certain way to fail in applying use cases successfully and will result in confusing specifications. In Figure 2, the <<include>>-dependency addresses two issues: (a) you want to communicate the authentication goal to stakeholders saying that certain activities are only allowed to be performed after[2] the actor has been properly authenticated and (b) you want to reduce complexity in the textual part of the use case specifications by avoiding that authentication activities have to be described three times, i.e. in the specification of "Check Out", "View Last Orders", and "Maintain Account".

Let's take one more paragraph to clarify the difference between goals and use cases. A use case is solution/system boundary specific. Its name is derived from a related goal. The use case's textual representation shows the concrete requirements necessary to achieve the goal the use case name relates to. A use case can be related to many other goals identified for a particular development project. Figure 3 shows an example of use case "Check Out" addressing or supporting different other goals[3] especially goals of a more non-functional or quality character that are sometimes also called *softgoals* (Yu 1997).

---

[2] When Authentication exactly happens in the flow of events is not expressed in the diagram, but the textual use case specification. However, using the includes-dependency in the diagram sensitises the audience for this fact and is used for communication about it.

[3] This UML representation of a goal as a stereotyped class with the target icon as well as respective relationship stereotypes has been recently introduced in (RUP 2003).

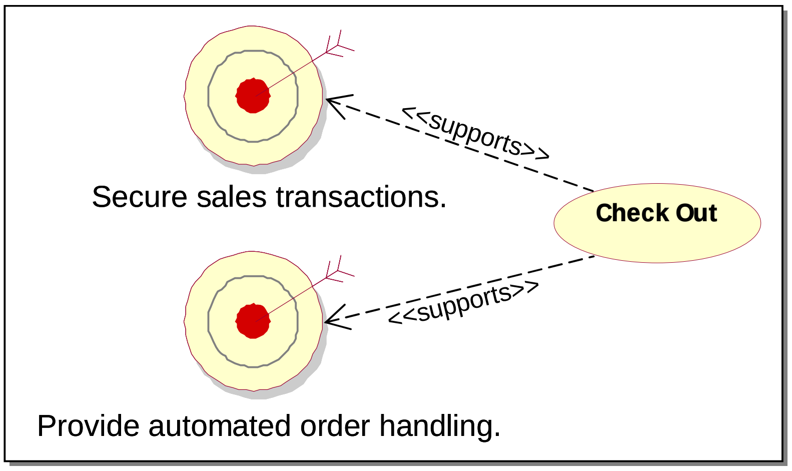


Figure 3: Goals addressed by "Check Out" use case.

Use Case Specifications

The graphical use case representation only presents an overview to your requirements. You always have to combine it with the textual representation of use cases that describes input-output scenarios with all their possible variants and exceptions, as well possible structural requirements (i.e. data), business rules, and non-functional requirements that directly apply to the use case. In cases where the non-functional requirements, rules, and data are of a more general nature and/or apply to more than one use case, these requirements are then typically factored into separate *supplementary specifications* (RUP 2003) and are not part of a use case specification.

Browse Catalog

Brief Description

The <Buyer> browses through the complete list of offered BAOS publications, can search and display a sub-set of this list by keywords, and display product details of each publication. The user can select publications to be placed in a "shopping basket" for later order.

Flow of Events

Basic Flow

The use case starts when the <Buyer> selects the BAOS catalog.

1. The system displays all *publications* available. It displays for each *publication* a *unique identifier*, its *title*, and *price*.
2. The <Buyer> enters and submits a *search text* to filter the displayed list of publications.
3. The system displays a filtered list of *publications* that only show entries that have the *search keyword* as a substring in the *title field*.
4. The <Buyer> selects one *publication* to review its details.
5. The system displays the details of the *publication*. It displays the *unique identifier*, the first *author*, the *title*, the *publisher*, and the *type* of the publication, and the *price*.

The use case ends.

Alternative Flows

<Buyer> adds publication to shopping basket

After Step 5, the <Buyer> selects the displayed publication to be placed into the shopping basket. The system adds the publication to the shopping basket. The use case resumes in Step 3.

<Buyer> sets a new filter

After Step 3, the <Buyer> can set a new filter. The system displays the newly filtered list. The use case ends or resumes in Step 4.

<Buyer> requests to see the complete *publications* list again

After Step 5, the <Buyer> requests to display the whole paged list of publications again. The use case resumes in Step 1.

Special Requirements

Performance

The system response in Step 3 must not take longer than 5 seconds.

Business Rules

The shopping basket cannot contain more than 10 items or items with a total value of more than 500 Euro.

Preconditions

None: The use case can be performed any time.

Postconditions

Publications list displayed
Publications details displayed

Figure 4: Use Case Specification for "Browse Catalog".

Figure 4 is an example use case specification for "Browse Catalog" from Figure 2. It had to be simplified for space reasons, but it contains the most salient aspects of a use case spec. Surely, there will be many more alternatives and special requirements you can think of, which have to be added. Also the alternatives should be structured using numbers for each step.

However, this small example already shows key features of use cases. First off, the use case presents requirements in context. Instead of listing isolated statements of system requirements, the use cases package requirements in stories so-called Flows which make up the Flow of Events.

> A *Flow* is a description of a partial path through the use case description. The *Flow of Events* describes the entire set of use case flows.

The Basic Flow describes the simplest and most straight forward way possible to achieve the use case's observable value. An Alternative Flow, as the name suggests, relates an alternative flow to the basic flow. They should always be written in a style that indicates

a) to what step in the basic or any other alternative flow they relate to,
b) under what condition or based on what event this alternative would be chosen,
c) enumerating the steps that are different from the basic or other alternative flow they relate to, and
d) where in the basic or other alternative flow the events would resume afterwards.

The flow of events makes up the complete picture of all possible behaviour the system supports. Particular flows can then be combined into scenarios that represent exactly one path through the flows, helping to explore one concrete case covered by the use case at a time. Scenarios are not explicitly represented in a use case spec, but can be derived from it.

> A *Scenario* is a specific instance or occurrence of a use case flow of events.

Applying Use Cases

In conclusion, you can say that use cases group together all possible stories that put system requirements in logical sequences describing how to achieve (and even not achieve) a goal. As you will partly see in subsequent sections, this is not only important for requirements elicitations and validation - to set focus and to establish a context of requirements enabling stakeholders to relate much better to the requirements - but also for many other disciplines of software engineering (also confer (Kruchten 2000) and (Kroll et al., 2003)):

- *Project planning and management*: Use cases help you to scope your project, i.e. deciding on priorities of requirements and change requests, by discussing values and goals addressed by the use cases. You also use the priorities to plan and assess iterative development activities of the flows of your use cases (cf. (Royce 1998)). Realizing use case flows as milestones of iterations ensures that at any time all milestones result into a system that delivers value to stakeholders for their review.

- *Analysis and design*: Will be discussed in the next sections.

- *Testing*: Use cases allow you to systematically derive test cases from the requirements by means of analysing use case scenarios and finding representative sets of input values for the conditions of alternative flows (see (Heumann 2001) for details).

- *Documentation*: If you look at how today's manuals and online help are organised, you see that they are structured based on what a user wants to achieve with the software. If I look, for instance, at the online help of the text processor I used to write this chapter, I see sections such as "Creating a document", "Moving around in documents", "Automatically correct text as you type", etc. I am not saying that you should copy-and-paste your use cases to get to your online documentation, but that use cases serve as an excellent starting point for the organisational structure of your documentation. Moreover, if you see other forms of documentation delivered for software these days: a tutorial is a collection of use case scenarios demonstrating the key values of the software to the user.

A more complete introduction to use cases for requirements management and especially use case writing would be out of the scope of this chapter. However, I will get back to the use case of Figure 4 discussing its structure and contents in the context of performing its analysis and design in subsequent sections. Please, refer to (Bittner et al. 2003) for an excellent text book on use case modelling and writing.

Object-Oriented Analysis with Use Cases

In this section, I describe the activities to systematically create software system analysis and design models from a use case specification. As you read above, we are taking prioritised flows of use cases as separate units of analysis, design, and implementation in iterative development to ensure that the system we construct iteratively provides the important goals first – especially when they are associated to technical risks – before we deal with the less important ones (Kruchten 2000)(Kroll et al. 2003). You will see that certain use case characteristics and styles directly support analysis and design. The focus for the following sections is to look at the role use cases play for analysis and design. Clearly, it cannot be the goal of this chapter to give you a detailed introduction into object-oriented analysis and design. Please, refer to these sources instead: (Jacobson et al. 1998), (Conallen 2002), (Eeles et al. 2002), (RUP 2003).

The goal of object-oriented use case analysis and design is to transform the "black box" use case specifications into a "white box" use case realisations.

> A *use-case realisation* describes how a particular use case is realised within the analysis and/or design model, in terms of classes and collaborations between these classes that exactly support the behaviour specified in the use cases.

Whereas an analysis model represents essential and technology independent abstractions of the solution focusing on concepts of the business domain and the behaviour specified in the use cases, the design model includes concrete technology and design decisions representing an abstraction of the implementation. Its models comprise of abstraction of concrete technology such as a J2EE session bean components or .NET Active Server pages. Whereas working on the analysis model has an emphasis on functional requirements ("getting the behaviour right"), the design model also systematically deals with non-functional requirements ("getting the quality right"[4]) on top of the functional requirements.

UML Representation of Use Case Interactions

So, how would you get started transforming our use case from Figure 4 into a use case realisation? A common starting point is to systematically explore the information

[4] Which is in most cases dependent on the technology you apply: For example, you tweak performance by choosing for specific component types and configuration settings for your middle tier server, or reliability with replication and redundancy features supported by certain server technologies, usability by applying certain styles and UI paradigms of a specific UI technology (standardised design principles and guidelines for Windows forms are quite different than for web forms), etc.

contained in the use case and to project this information onto conceptual representations.

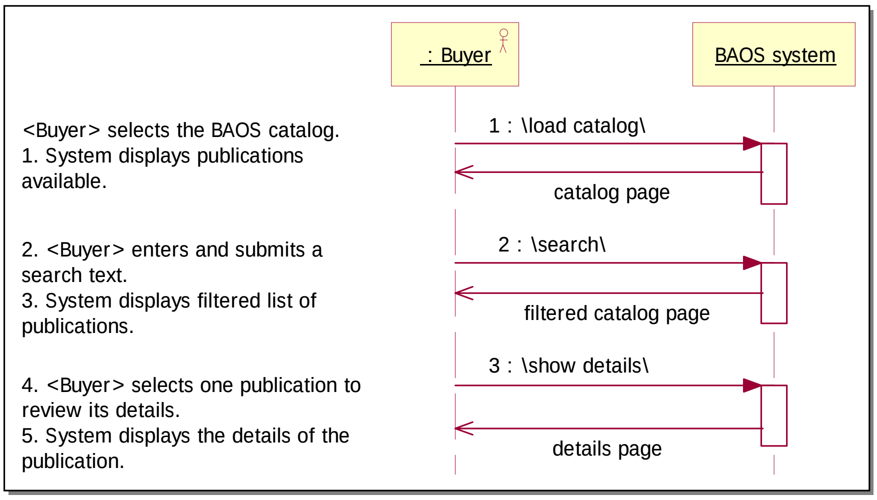

: Buyer
BAOS system
<Buyer> selects the BAOS catalog.
1. System displays publications available.
1 : \load catalog\
catalog page
2. <Buyer> enters and submits a search text.
3. System displays filtered list of publications.
2 : \search\
filtered catalog page
4. <Buyer> selects one publication to review its details.
5. System displays the details of the publication.
3 : \show details\
details page

Figure 5: Scripted Sequence Diagram showing black box specification of "Browse Catalog" use case.

You can see an example of such a representation in Figure 5. It shows a UML sequence diagram for our uses case's black box interactions between actor and system. Each pair of messages represents one round-trip of events in which the actor provides a stimulus or input and gets a response or output back from the system. The use case text has been used to annotate these round trips, which makes the diagram a so-called scripted sequence diagram. Because these round-trips of events denoted in the diagram play an important role for many design decisions for the system later on, many use case authors write their flow of events with such a diagram in mind. You see from the diagram's script that every numbered step in my use case also maps to such a round-trip in Figure 5.

User Experience Modelling

Just to provide you with one example, of how this structure supports design decisions, let's have a look at the user experience model in Figure 6 that is also directly derived from the use case's flow of events and enjoys particular popularity for web development projects.

> The *User-Experience (UX) Model* describes the user-experience elements of the system (the screens and input forms), the dynamic content that appears on the elements, and how the user navigates through the elements to execute the system functionality.

For a detailed introduction to UX Modelling see (Conallen 2002) or (RUP 2003).

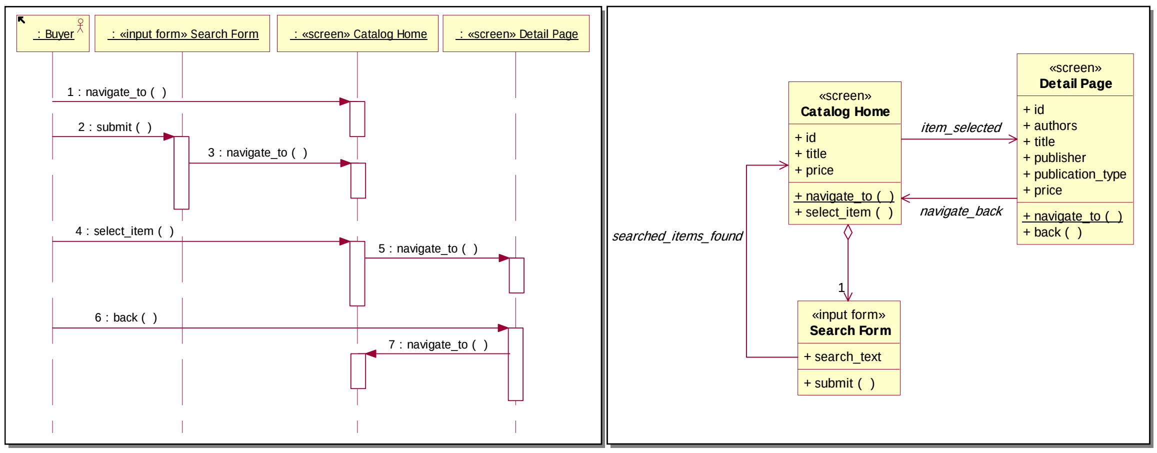


Figure 6: A user experience model.
(Left: Use Case Storyboard, Right: Resulting Navigation Map)

Many development teams decide that their use cases on the one hand have to be kept general enough that they could be realised with different UI design paradigms, but on the other hand concrete enough that they provide all essential requirements (e.g. inputs, outputs, business rules, and quality) for the application design. Instead, teams often create and maintain a separate user experience model in conjunction with use cases as well as screen mock-ups. Such a model is (still) a black box representation of the system helping to promote (a) improvements on the understanding stakeholders have about requirements making the system specification more tangible with a concrete flow of screens and (b) the conceptual design of consistent screen and navigation flows for the application's user interface (UI). Thus, the idea here is to separate UI design artefacts from requirements artefacts, but to also use them in synergy for improvements on requirements and UI design as well as consistency.

The UX model represents the concrete UI design that is sometimes necessary to communicate how the system is really going to be looking like. It is owned by a special group of creative designers that are generally responsible for creating the look-and-feel of the application as well as the navigation routes and contents of the pages. The UX model is used as a conceptual, UML-based plan to design concrete forms and screens, which in turn are being used in terms of use case storyboards to validate the requirements with stakeholders.

Figure 6 shows the UML representation that is then normally related to physical screen designs or mock-ups. As in most UML models the UX model also consists of a dynamic (Figure 6, left) as well as a static model (Figure 6, right). The diagram on the left depicts a use case storyboard that again could have been scripted with the use case flows. You could say that it details the system object of Figure 5 with the flow of concrete screens and forms to be used to realise the use case. The class diagram on the right represents the so-called navigation map which is produced for each individual use case, but also continuously integrated with the global navigation maps representing the combination of all use cases. Classes represent Screens and Forms in a navigation map. Screens present system outputs and Forms are reusable parts of this output, which in turn accept input from users. Associations represent paths to navigate from on screen to the next. You see that for the "Browse Catalog" use case I decided to use two screens: one showing the unfiltered and filtered catalog information and the other for the details of a particular publication. The form capturing search input from the user has been aggregated to the catalog screen (to be designed as a search-text field and submit action perhaps somewhere on the side or top of the catalog list). A stakeholder review of this model and use case could now result into the decision that searching should also be possible from the detail screen. This decision would lead to an additional alternative flow in the use case as well as an additional aggregation association between "Detail Screen" and "Search Form".

You can see from this last example that systematic walkthroughs of use cases with UX models cannot only be used to validate, but also to refine use case specifications by eliciting new requirements such as defining and exploring alternative flows. An interdisciplinary team can quickly relate to the story-bound application presented and identifying missing steps in the use case specification as well as validate inputs and outputs for completeness and consistency with the desired workflow. Conversely, the team can utilise the use case to verify if the screens and forms of the UX model cover all flows by checking input and outputs (class attributes) and interactions (class operations and associations).

Use Case Analysis Overview

Moving on in the use case analysis, utilizing the use case specification, scripted sequence diagrams, and the UX model (or just a subset of these, of course) you, as the designer, can now systematically map the use case to a set of standard abstractions representing system internal white box objects and distribute the behaviour described in the use case's flow of events to collaborations between these objects.

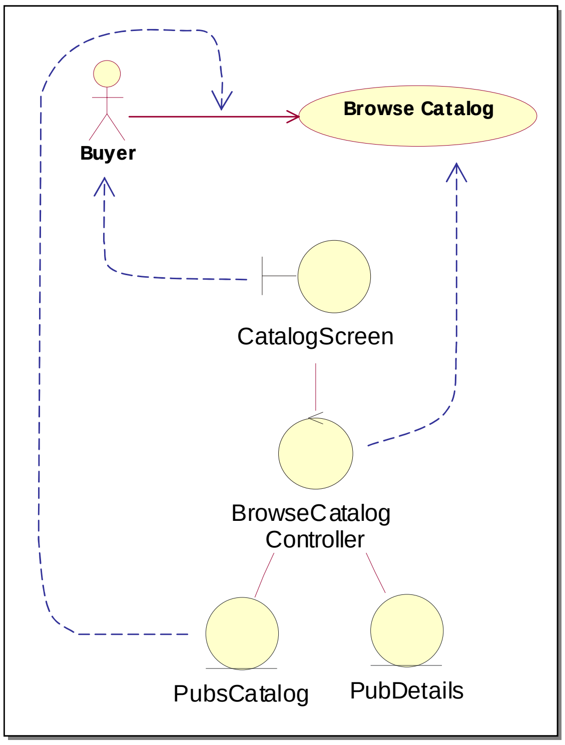


Figure 7: Use Case Analysis - Mapping use case elements to standard abstractions.

The RUP defines three commonly used[5] class stereotypes to use for this mapping always resulting to a variation of the so-called "Duck on Skis"-diagram for use cases as depicted in Figure 7:

[5] For example, see the UML specification 1.5 (OMG 2003) which defines these stereotypes in the example UML profile for software development.

- *<<Boundary>> classes* representing the interfaces to the outside world (either to a human actor's interfaces derived from the screens of the UX model or interfaces/APIs to actors representing external systems). Figure 7 shows the "CatalogScreen" as an example for a boundary analysis class.
- *<<Controller>> class*: representing the use case's actual flow of events, i.e. the functional behaviour expressed by a set of operations mapping the use case's input/output roundtrips. It is a recommended practice of use case-based analysis and design to identify one main controller per use case. Figure 7 shows "BrowseCatalogController" as an example for a controller analysis class.
- *<<Entity>> classes*: representing the inputs and outputs of the use case, i.e. the data flowing into the system and out of the system. Figure 7 depicts "PubsCatalog" and "PubDetails" as examples for entity analysis classes.

As indicated above, controllers represent use case specific behaviour aiming to provide an overview of the system responsibilities or services that need to be realised or made available for the use case. Boundaries and especially entities are modelled with the perspective to use them in more then one use case realisations. For example, it is likely that the data abstraction "PubDetails" will also be used in the realisation for the "Checkout" and "Maintain Shopping Cart" use cases.

Use Case Analysis Dynamic View: Distribute Use Case Behaviour

Thus, after this first step of identifying these static abstractions, your second step of the analysis is walking through the use case specification roundtrips and distributing the use case behaviour to the classes by assigning responsibilities. Behaviour is best represented with the UML's interaction diagrams such as sequence or collaboration diagrams. As you saw in Figure 5 there are three roundtrips in the use case's basic flow: "load the catalog", "search", and "show details". The procedure for distributing responsibilities for these three roundtrips is determined by the stereotypes: Boundaries take commands and input from actors and forward them to the controller for processing. The controller uses entities to persist and/or retrieve data to deliver output to a boundary for display to an actor.

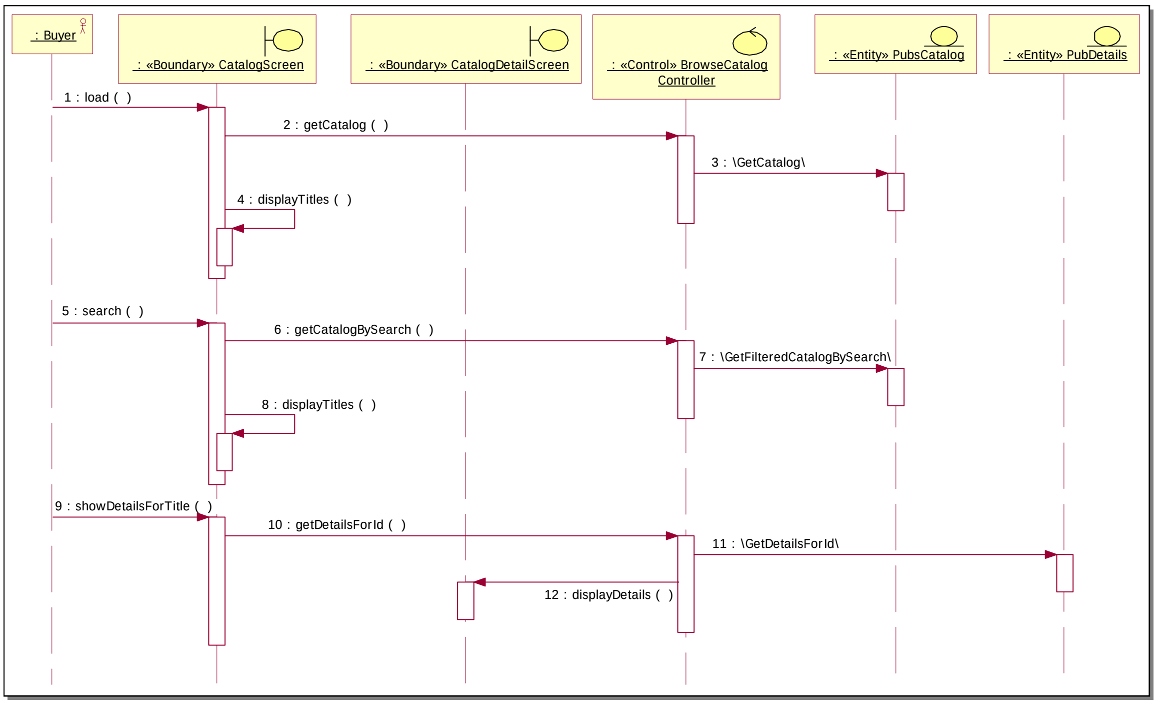


Figure 8: Allocating use case behaviour to analysis class responsibilities in sequence diagrams.

Figure 8 shows the resulting sequence diagram after distributing the three roundtrips for the “Browse Catalog” basic flow to the abstractions of Figure 7 (plus another boundary added for the publication details transforming our duck into a duck with two heads). For example, the use case’s first roundtrip describes how a catalog page is being retrieved labelled as a “load()” command the actor issues through the user interface (probably by typing the URL of our sales webpage) that is delegated by the boundary “CatalogScreen” representing this page to the controller via the “getCatalog()” responsibility, which in turn retrieves the requested information from the entity “PubsCatalog”. The result is then displayed in the same boundary from which the initial request came from via the “displayTitles()” responsibility. As you can see all three roundtrips of our case study use case describe simple, technology independent data retrieval operations and thus, all look very similar in their realisation.

Use Case Analysis Static View: Describe Relationships, Responsibilities, Attributes

Your third step in your use case analysis is to update your classes as depicted in Figure 9 with the responsibilities and relationships identified in the sequence diagram.

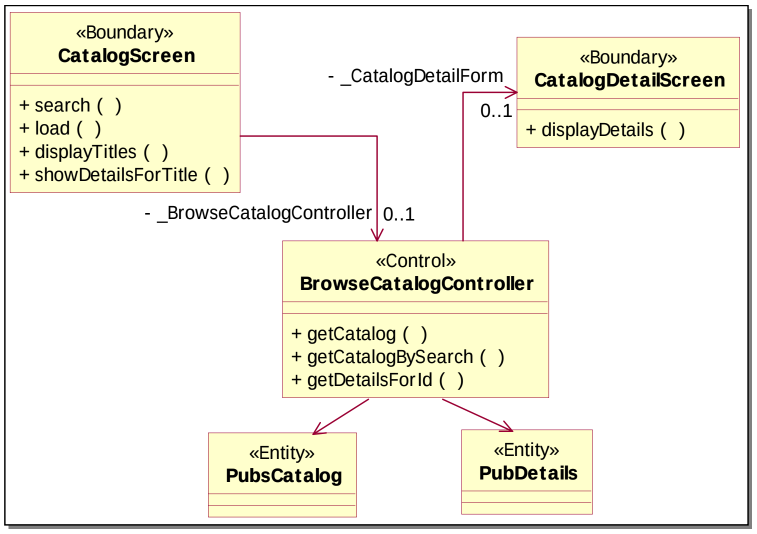


Figure 9: Analysis class diagram derived from sequence diagram in Figure 8.

As a result, responsibilities are added as operations. When one class calls another class's responsibility, a relationship between the two classes is added. Between classes with different stereotypes it does not really matter at this point if you use associations (expressing a structural relationship) or dependencies (expressing local-variable or parameter-usage relationships). This will change during design anyway depending on the technology and design mechanisms used to realise these abstractions. Nevertheless, it is important to capture that these classes need to know each other to design their relationship appropriately. On the other hand, relationships between classes of the same stereotype, especially for the entity classes, represent important analysis results (as you will see in Figure 10). Figure 9 shows the extended "duck-on-skis" resulting after this step, which actually many UML design tools generate automatically or semi-automatically from the sequence or collaboration diagrams.

Looking at this diagram and comparing it against the use case text from Figure 4, you can see that you so far mapped the dynamics, but not all the static requirements expressed in the use case to your classes. For example, for "Step 1", we realised the "*The system displays the first page of all publications available.*" part with the association between the "CatalogScreen" and the "BrowseCatalogController" and a dependency to "PubsCatalog" representing the data retrieved and presented, as well as the operations "CatalogScreen::load()", BrowseCatalogController::getCatalogPage()", and "CatalogScreen::displayTitle()".

The second sentence of "Step 1":"*It displays for each publication a unique identifier, its title, and price.*" is not yet mapped into the model. First you might want to argue if this information is actually to be found in the use case specification or not. Clearly, different use case writing styles might factor these requirements into a separate section (data requirements) or completely different supplementary specification documents; or keep this information just as part of the User Experience model[6]. Neverthe-

[6] Another style I discovered in the field with more classical Structured Analysis educated system analysts is to only capture data requirements in the analysis model itself. However, my experience shows that it is advantageous to capture data in the requirements specification or at least the user experience model, because these representations are much easier to understand for stakeholders who have to review, validate, and decide on these requirements.

less, although it is not important from which source we get this info from; it remains an important fact that detailed data specifications constitute important requirements that need to be available to do analysis. Therefore, you work with these requirements in the fourth step of analysis by refining the model to include an actual structure (in terms of associations and attributes) of how publications need to be represented in the system. This is now the real creative part of the analysis. Whereas before, we performed a standard mapping from the use case to classes, we now need to make decisions on how to represent the data, i.e. organise our entities. Surely, you cannot make good decisions just on the basis of one flow of one use case, because you want to structure the data that it fits the requirements of all use cases that make use of it in the end. However, you need to start somewhere and other steps of the analysis deal with the integration of different use case results. Further, during design you will make more changes on these structures to support, for example, non-functional requirements or technology specific constraints in the best possible manner. Thus, Figure 10 shows my initial decisions applying perhaps also a bit of long sight on what else has to be realised with these entities.

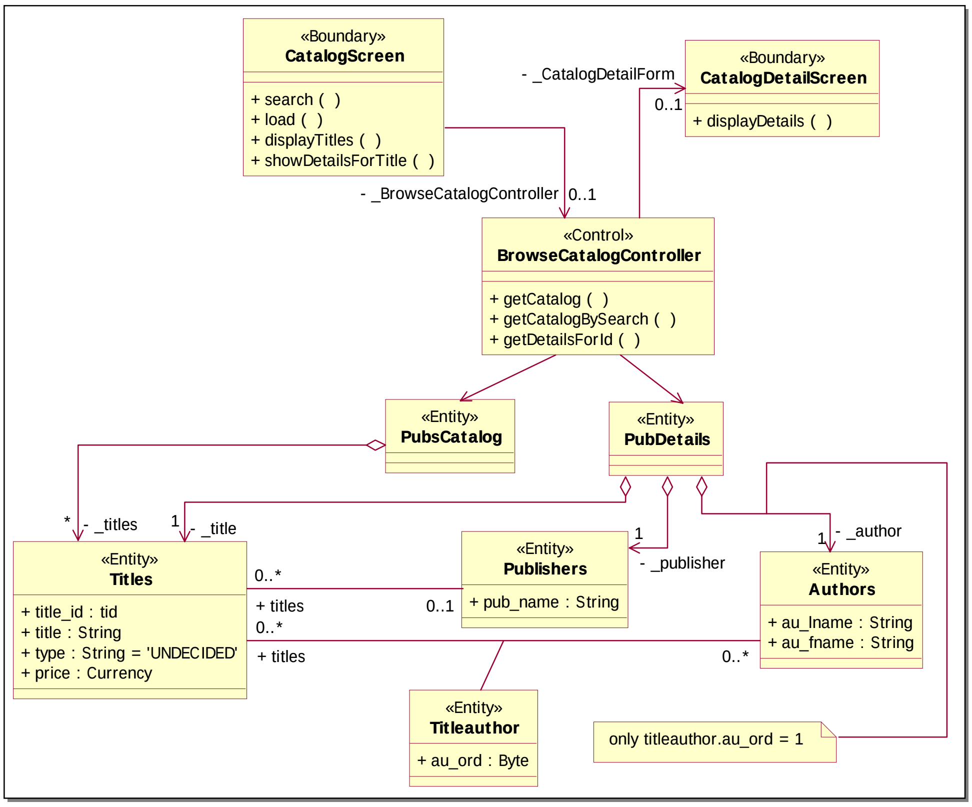


Figure 10: The completed Participants class diagram after analysing "Browse Catalog"'s basic flow.

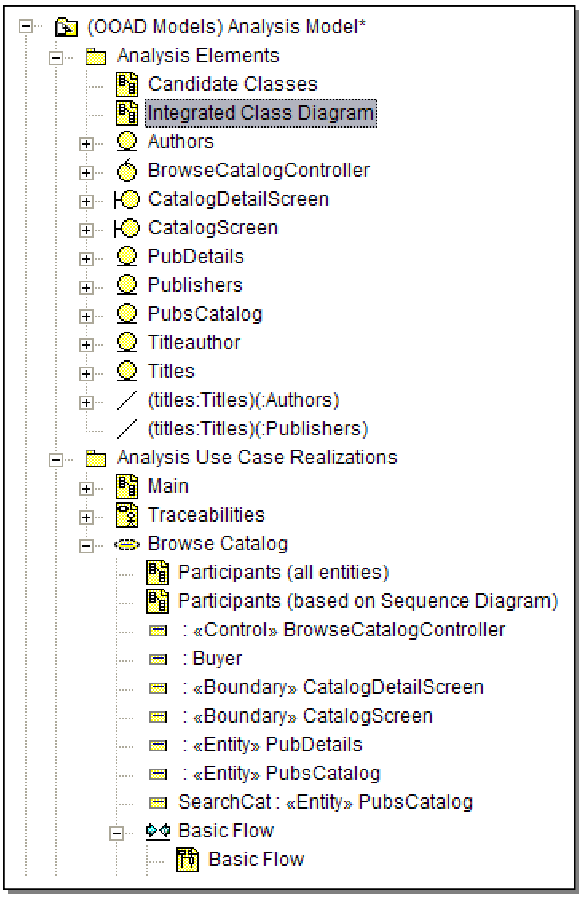
This lower part of the model in Figure 10 comprising of new entities will evolve, in comparison to more classical data modelling techniques, into the so-called logical data model being the basis for data related modelling activities in design. Your ultimate verification step after the extension of the entities (and especially after changes based on integration with other use case analysis results are performed) is now to perform use case walkthroughs to verify that the classes indeed support the specified behaviour.

As a result, what you see in Figure 10 is the UML class diagram representing the static structures needed to support the use case “Browse Catalog”. Such a use case specific diagram is also called the *Participants* diagram.

Your fifth step in the analysis would be, as already indicated, to integrate this participants diagram with all other participants diagrams in so-called Analysis Elements diagrams. However, because we only have space to work on one use case in this chapter, please refer to the RUP (RUP 2003) or (Eeles et al. 2002) for examples on this step.

Managing and Tracing Analysis Results

As you can see from this small example, you will generate quite a few different diagrams showing your analysis results from different views (such as dynamic and static). You will also do this for every use case you analyse. To deal with the management complexity and to not lose traceability to your use cases it is important to set-up a canonical model structure for your diagrams and model elements. The UML supports this with the concepts of packages and collaborations as well as design tools normally support hierarchical displays to set these up as indicted in Figure 11.



```
(OOAD Models) Analysis Model*
    Analysis Elements
        Candidate Classes
        Integrated Class Diagram
        Authors
        BrowseCatalogController
        CatalogDetailScreen
        CatalogScreen
        PubDetails
        Publishers
        PubsCatalog
        Titleauthor
        Titles
        (titles:Titles)(:Authors)
        (titles:Titles)(:Publishers)
    Analysis Use Case Realizations
        Main
        Traceabilities
        Browse Catalog
            Participants (all entities)
            Participants (based on Sequence Diagram)
            : «Control» BrowseCatalogController
            : Buyer
            : «Boundary» CatalogDetailScreen
            : «Boundary» CatalogScreen
            : «Entity» PubDetails
            : «Entity» PubsCatalog
            SearchCat : «Entity» PubsCatalog
            Basic Flow
                Basic Flow
```

Figure 11: Analysis Model Structure.

As you see in Figure 11 the analysis model is structured into two main packages:

- *Analysis Elements* Package: Representing the integrated static structure view on our analysis results. Although Figure 11 only shows the results after analysis of “Browse Catalog”, this package will always contain the continuously integrated results after merging all use cases analysis results available to the respectively current point of time. This way of organising classes into an integrated view ensures that analysis of the next use case will reuse and update already existing classes and not reinvent abstractions already present.

- *Analysis Use Case Realisations* Package: Representing a view on each use case including the use case specific static participants diagrams as well as the dynamic interaction diagrams. These diagrams refer to model elements maintained in the Analysis Elements package, which has the advantage that when classes are updated based on the analysis of other use cases, the updates are also reflected in classes participating in already analysed use cases. The results of every use case analysis are organised as part of a use case realisation that has the same name as the use case for traceability reasons. As you can see in Figure 11, the "Browse Catalog" use case realisation (modelled as an UML collaboration instance) consists of its participant diagrams (shown in Figure 9 and Figure 10) as well as an Interaction Instance for the use case's basic flow and a respective sequence diagram as well as the objects being presented in the sequence diagram.

It is important that a project-wide model structure guideline exists for a development team providing a well-defined way of organising the 'electronical filing cabinet' of UML diagrams. This may sound like a trivial statement, but I have seen many projects in which the lack of such a uniform guideline has not only led to the loss of traceability, but also the loss of analysis or design results, because they were hidden somewhere deep down in an undocumented package hierarchy.

Object-Oriented Design with Use Cases

Let's move on one step further in software development and discuss how to design our use case "Browse Catalog" based on the analysis results presented in the last section. The objective is to map the analysis results to a concrete technology specific realisation, which also adds to the coverage of functional requirements the coverage of non-functional requirements and design constraints. Because of space restraints however, I will only describe the mapping of the functional requirements into technology. I have chosen to use the .NET framework and ASP .NET as the solution technology for our sales web application.

Software Architecture and Use Cases

To cover the mapping of non-functional requirements and to truly understand the decision making activities you are performing during design, we would actually need to cover the notion of Software Architecture first.

> *Software Architecture* encompasses the set of significant decisions about the organisation of a software system.

Unfortunately, such a discussion as well as description of significant activities performed in architectural analysis, synthesis, and refinement would go beyond the scope of this chapter. Confer here again (RUP 2003), (Eeles et al. 2002), or (Conallen 2002).

However, the good thing about architecture is that because it is generally perceived as not a trivial but still such a very important task, one can find many *OOAD stakeholders* out there that provide you with quite successful ways of making architecture reusable within domains and technologies. These reuse strategies are normally described as architectural frameworks as well as design patterns. Architectural frameworks and design patterns are often delivered by the technology providers themselves to make the technology more successful[7]. Examples for these frameworks and patterns can be found for example for the J2EE technology in (Alur et al. 2003) and .NET technology at (Microsoft 2003) or (Fowler 2002) covering both.

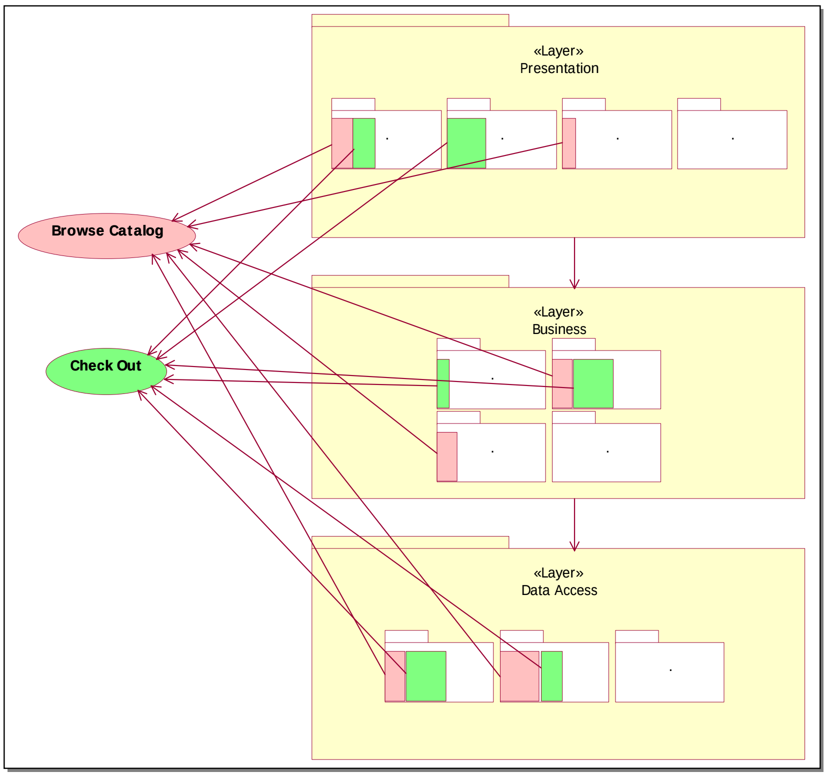


Figure 12: Relationship of Use Cases to Layered Architecture.

Typical enterprise application architectures are very often organised in layers of concern, for example, separating presentation from business logic from data storage and access. The question of interest in this section is: How do we map use cases into such a solution structure? In Figure 12 we see that all parts of the use case can have an influence on all parts of the architecture. This also means that a change in a use case (for example, a new field to be shown in the details screen) will trigger changes in all these levels in all possible packages. Therefore, we need to find a way to organise our Design Use Case Realisations to keep trace of what part exactly is being designed for what use case requirement. Of course, we are going to do this via the Analysis Use Case Realisations: Tracing to the use case from the analysis model from the design model.

Use Case Design Overview

In use case design, we systematically transform and refine the analysis classes captured in our analysis use case realisation to technology specific design classes. Looking at our analysis model results of Figure 10 and the layers of Figure 12 you might think the mapping is obvious: let’s put the boundaries into the presentation tier, controller into the business layer, and entities in the data access layer. Unfortunately, it is not that straightforward. As with the use cases can all parts of the analysis model

[7] Successful in respect to many concerns such as resilience of the application, cost of change and extensions, facilitation of reuse, realisation of a large set of *standard* non-functional requirements, and many, many more.

have an influence on all parts of the architecture. Consider for example entities: They have an influence on the design of all layers and the interactions between them. Publication catalog data (Entity "PubCatalog") is needed in the presentation tier, because it has to be displayed on the screens. It is processed in the business layer (e.g. if you would extend your use case to calculate rebates on the prices for a particular category of customers). It is accessed in the database and defines the structure of the database schema (not represented in Figure 12). Also, to be able to send the data from the database over the internet to the COM+ MTS application server executing the business logic and from there to the IIS web server using .NET Remoting, another representation of the "PubsCatalog" entity is needed to support this.

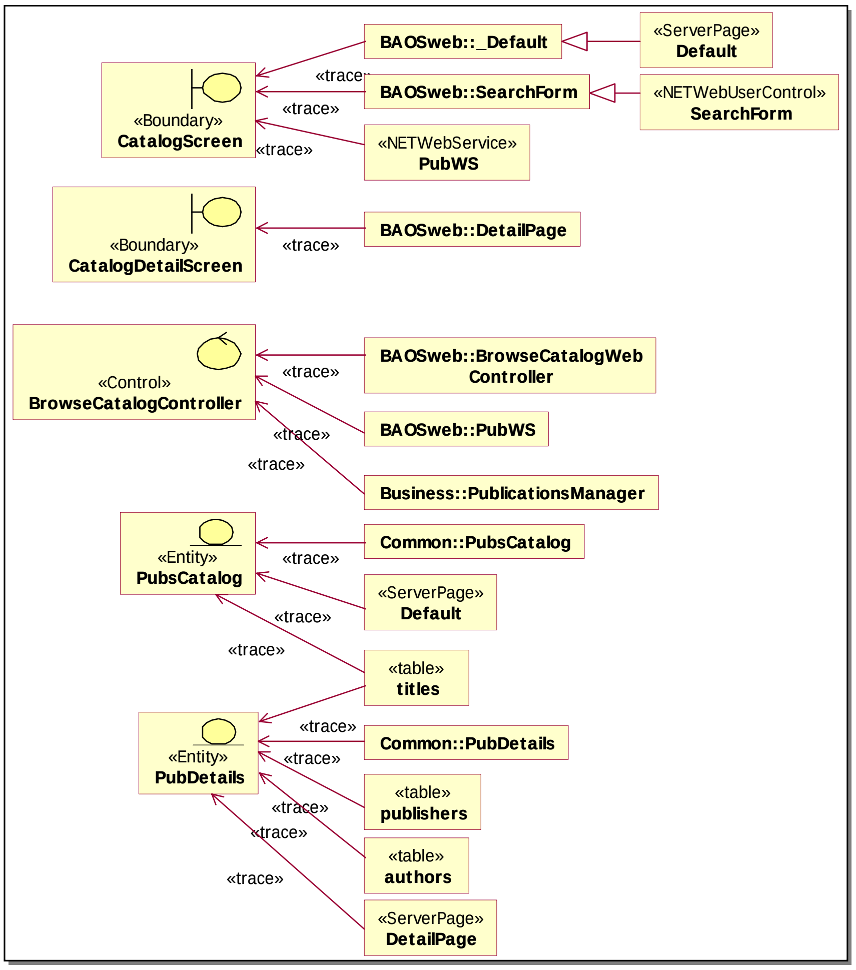


Figure 13: Mapping of Analysis to Design Classes.

To clarify this point I already present in Figure 13 some final design results. This traceability diagram shows how your analysis classes have been mapped to respective design classes. Here you see the mapping of "PubsCatalog" to database tables, an ASP .NET Server Page (containing controls to present the data; not shown), as well as a class Common::PubsCatalog that represents an ADO .NET dataset for transfer between servers utilising XML. The "BrowseCatalogController" also has been *split* into three classes covering two different concerns: presentation specific logic and business specific logic. The "BrowseCatalogWebContoller" represents the logic realizing the navigation flow in the web application, e.g. which screen to use for which presentation in which particular order, which business component to contact to get business logic processed, etc. A windows forms user interface would do things quite different. Thus, you would like to separate presentation from business logic to increase reusability of the business part. This type of presentation logic is also completely different for an application interface providing web services, which also has been model in Figure 13 with the “PubWS” class. Web services are intended not to provide a user interface, but an application interface for catalog functionality over an http protocol[8]. The other concern of representing the pure business logic for browsing the catalog independent of the presentation form is realised with the “PublicationsManager” component that we design in the next section. Finally, if you remember that boundaries do not only represent UI responsibilities, but are also used to model APIs to access external systems (not the case for “Browse Catalog”), you can also see that these sometimes need to be mapped to design on different levels of the architecture than the presentation tier.

To design a use case realisation, you start by identifying design elements as we just discussed and presented in Figure 13 and then continue to systematically transform analysis elements of your analysis use case realisation with these design elements. You will do this now for “Browse Catalog” bottom-up, i.e. specify and refine a design element representing a business component[9] in the next section and then in the following section replace elements of our analysis model with this component and other design elements. Surely, this could also be done the other way round: top-down. This is merely a matter of preference (as well as availability of already designed model elements) and normally performed in a more intertwined way.

Component Design

Components represent the realisation of subsystems for software applying the principles of encapsulation to minimise dependencies between different parts of the application to achieve more resilience; for example in respect to change impact. Therefore, software components constitute black boxes providing realisations for functions and non-functions. As for all systems, we produce specification and realisation documentation for components. Hence, we could actually draw a use case diagram for our component “PublicationsManager” and describe in a use case specification roundtrips of an actor (in this case a class from the ASP .NET user interface) calling the component. A more popular, but less goal-oriented way as use cases, for representing component specifications, is depicted in Figure 14.

[8] You see to classes called PubWS one representing the web service controller (would be realised with a .NET asmx.cs file), the other the application interface mapped to the boundary (would be realised with an .asmx file).

[9] Because of limited space, we start in the middle layer and omit the data access layer.

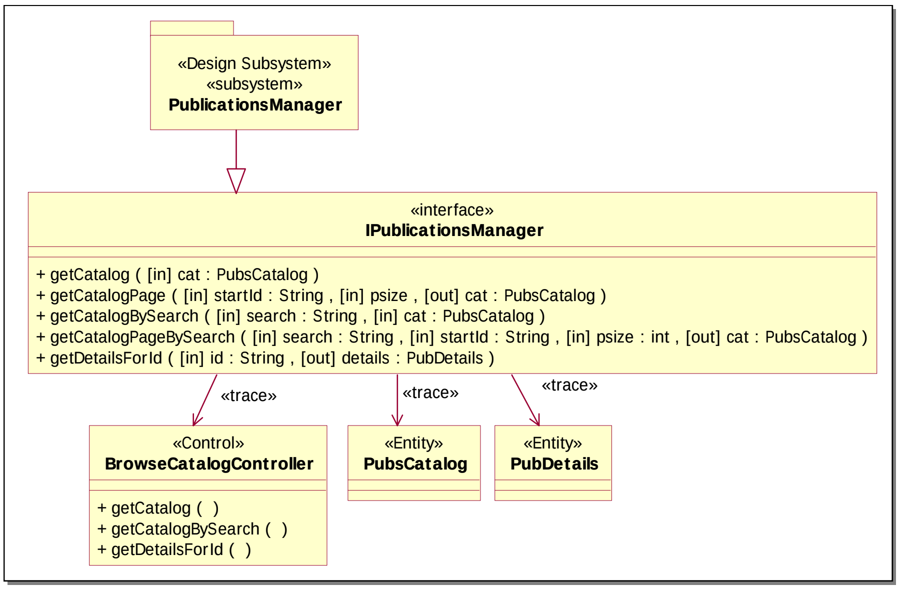

«Design Subsystem»
«subsystem»
PublicationsManager
«interface»
IPublicationsManager
+ getCatalog ([in] cat : PubsCatalog)
+ getCatalogPage ([in] startId : String , [in] psize , [out] cat : PubsCatalog)
+ getCatalogBySearch ([in] search : String , [in] cat : PubsCatalog)
+ getCatalogPageBySearch ([in] search : String , [in] startId : String , [in] psize : int , [out] cat : PubsCatalog)
+ getDetailsForId ([in] id : String , [out] details : PubDetails)
«trace»
«trace»
«trace»
«Control»
BrowseCatalogController
+ getCatalog ( )
+ getCatalogBySearch ( )
+ getDetailsForId ( )
«Entity»
PubsCatalog
«Entity»
PubDetails

Figure 14: A Component Specification.

Figure 14 presents the black box specification of our component defining its interface "IPublicationsManager", which as for every system interface describes input-output transformations, is this case using UML operation declarations. The "PublicationsManager" subsystem is represented as a package realizing the interface that contains the white box model elements doing the actual work of providing catalog data in a paged and searchable manner. These elements are developed with analysis and design activities in a very similar way as we discussed throughout this and the last section for the overall "Browse Catalog" use case and omitted for space reasons. See (Cheesman et al. 2001) for an excellent discussion on software components, their specification, and realisation with UML.

UI Design and completing the Design Use Case Realisation

In this final section, you will now put a user interface on top of the "PublicationsManager" component. We do this by taking the analysis model elements and transforming them to the design elements of Figure 13 resulting in updated interaction and participants diagrams. The way the design elements in Figure 13 had been actually identified was by applying several design patterns from (Microsoft 2003) ["Model-View-Controller with ASP .NET" and "Data Transfer Object"] and (Fowler 2002) ["Application Controller"]. An architect's responsibility is to select the appropriate set of patterns to be used for use case design in a repeatable and predictable fashion and documenting them as so-called *Design Mechanisms* (RUP 2003). Using the same mechanisms for similar design tasks in every use cases ensures maintainability of the design as well as centralises architectural decision making ensuring that not every designer invents his own solutions.

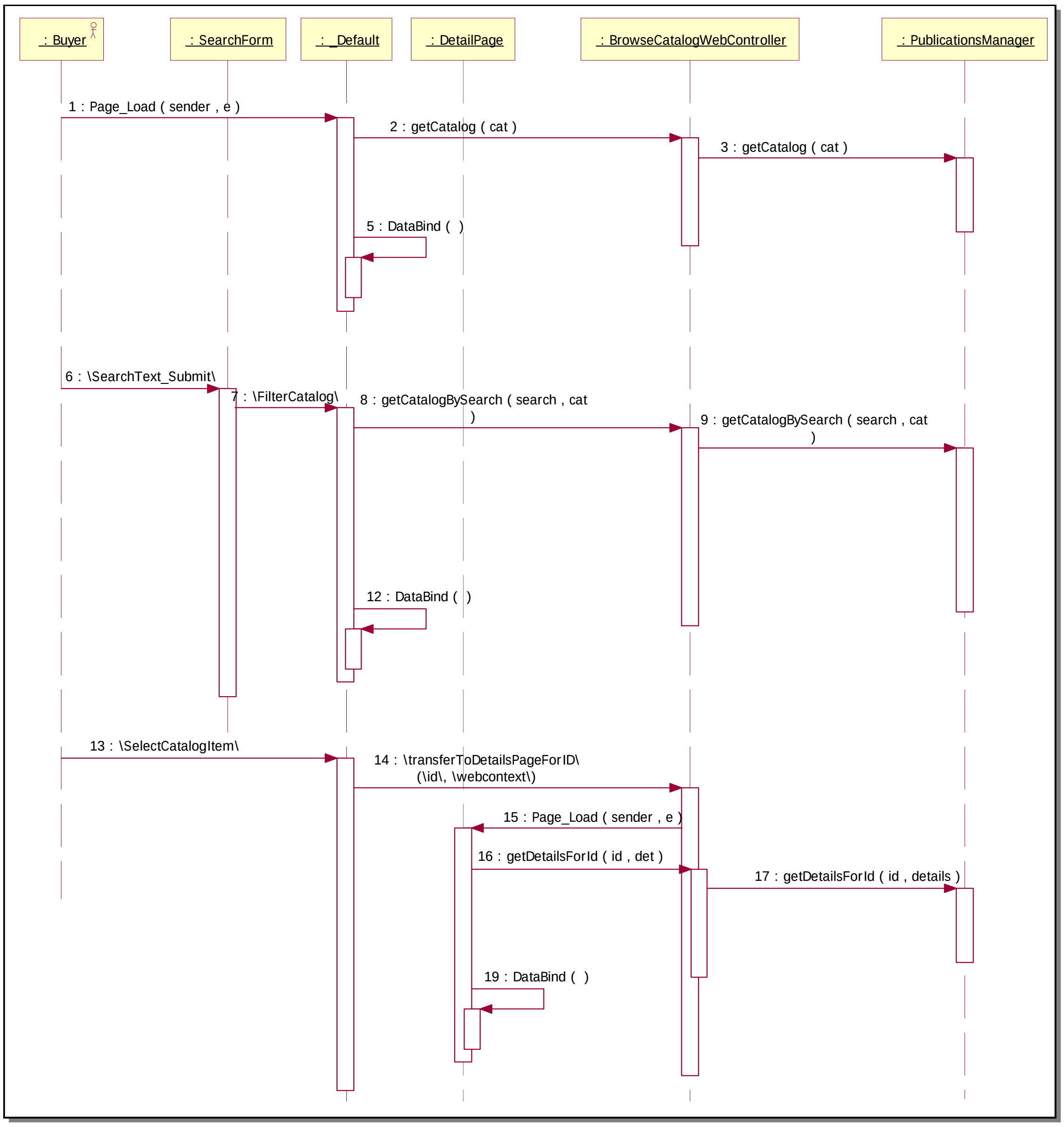

: Buyer
: SearchForm
: _Default
: DetailPage
: BrowseCatalogWebController
: PublicationsManager
1 : Page_Load (sender , e)
2 : getCatalog (cat)
3 : getCatalog (cat)
5 : DataBind ( )
6 : \SearchText_Submit\
7 : \FilterCatalog\
8 : getCatalogBySearch (search , cat)
9 : getCatalogBySearch (search , cat)
12 : DataBind ( )
13 : \SelectCatalogItem\
14 : \transferToDetailsPageForID\ (\id\, \webcontext\)
15 : Page_Load (sender , e)
16 : getDetailsForId (id , det)
17 : getDetailsForId (id , details)
19 : DataBind ( )

Figure 15: Design Use Case Realisation - Sequence diagram.

Figure 15 depicts the updated sequence diagram after incorporating the design elements and patterns. Our actor Buyer interacts with classes representing ASP .NET code-behind pages classes[10], which in turn interact with the "BrowseCatalogWebController" class. The web controller class uses the "PublicationsManager" to interface with the business component discussed in the last section.

Figure 16 shows the updated participants diagram presenting the classes from Figure 13 as well as resulting relationships from the sequence diagram. For example, we see that relationships had to be added in our design such as associations of the ASP .NET pages' code-behind classes to the dataset classes "PubsCatalog" and "PubsDetails". These classes contain the data the "PublicationsManager" component retrieved from

[10] ASP .NET defines for a server page two classes: one class for the HTML presentation content as well as a second class, called code-behind class the first one derives from through specialization, for event handling and dynamic presentation generation.

the database that has to be displayed in our pages and therefore have to be accessible in the code-behind classes for presentation generation and event-handling operations.

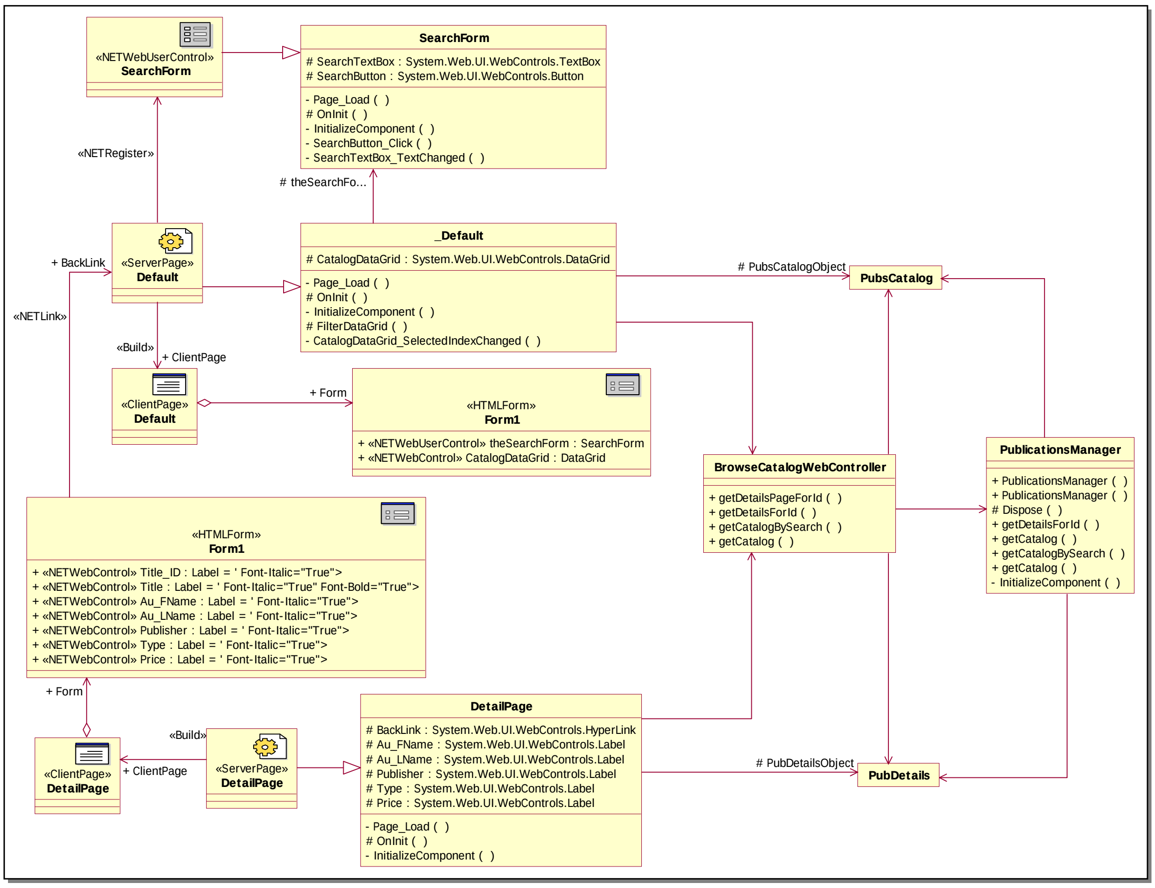

«NETWebUserControl»
SearchForm
SearchForm
# SearchTextBox : System.Web.UI.WebControls.TextBox
# SearchButton : System.Web.UI.WebControls.Button
- Page_Load ( )
# OnInit ( )
- InitializeComponent ( )
- SearchButton_Click ( )
- SearchTextBox_TextChanged ( )
«NETRegister»
# theSearchFo...
+ BackLink
«ServerPage»
Default
_Default
# CatalogDataGrid : System.Web.UI.WebControls.DataGrid
- Page_Load ( )
# OnInit ( )
- InitializeComponent ( )
# FilterDataGrid ( )
- CatalogDataGrid_SelectedIndexChanged ( )
# PubsCatalogObject
PubsCatalog
«NETLink»
«Build»
+ ClientPage
«ClientPage»
Default
+ Form
«HTMLForm»
Form1
+ «NETWebUserControl» theSearchForm : SearchForm
+ «NETWebControl» CatalogDataGrid : DataGrid
BrowseCatalogWebController
+ getDetailsPageForId ( )
+ getDetailsForId ( )
+ getCatalogBySearch ( )
+ getCatalog ( )
PublicationsManager
+ PublicationsManager ( )
+ PublicationsManager ( )
# Dispose ( )
+ getDetailsForId ( )
+ getCatalog ( )
+ getCatalogBySearch ( )
+ getCatalog ( )
- InitializeComponent ( )
«HTMLForm»
Form1
+ «NETWebControl» Title_ID : Label = ' Font-Italic="True">
+ «NETWebControl» Title : Label = ' Font-Italic="True" Font-Bold="True">
+ «NETWebControl» Au_FName : Label = ' Font-Italic="True">
+ «NETWebControl» Au_LName : Label = ' Font-Italic="True">
+ «NETWebControl» Publisher : Label = ' Font-Italic="True">
+ «NETWebControl» Type : Label = ' Font-Italic="True">
+ «NETWebControl» Price : Label = ' Font-Italic="True">
+ Form
«Build»
«ClientPage»
DetailPage
+ ClientPage
«ServerPage»
DetailPage
DetailPage
# BackLink : System.Web.UI.WebControls.HyperLink
# Au_FName : System.Web.UI.WebControls.Label
# Au_LName : System.Web.UI.WebControls.Label
# Publisher : System.Web.UI.WebControls.Label
# Type : System.Web.UI.WebControls.Label
# Price : System.Web.UI.WebControls.Label
- Page_Load ( )
# OnInit ( )
- InitializeComponent ( )
# PubDetailsObject
PubDetails

Figure 16: Design Use Case Realisation - Participants diagram.

To cut this technical discussion short, we see in these two diagrams that the essential analysis model has been transformed and extended for the concrete technical platform they are supporting. It is now imperative to verify that the “Browse Catalog” use case is still fully supported as specified after all these changes by reviewing sequence and class diagrams against the use case specification with a walkthrough. Finally, the implementation derived from this design will be tested against scenarios of the use case as well.

Summary and Comparisons:

This concludes this chapter in which I walked you through analysis and design with use cases. My intention was to provide system analysts with a basic understanding of what designers actually do with their use case artefacts to improve their ability to include them as an intended audience for their use case writing. In contrast to many of the scenario and story telling techniques presented in other chapters of this book, use cases represent a compromise between informality and natural language expressiveness to facilitate interdisciplinary communication on the one hand and enough structure to systematically drive software development activities such as user-experience modelling, analysis, and design as described in this chapter on the other hand. In addition, use cases focus, in contrast to scenarios and stories, on actor values and therefore are not decomposed in smaller parts that would loose that focus and would become hard to understand, the rationale hard to remember, as well as hard to verify/validate. Consequently, users or customers do not write the use cases themselves, but still must be able to relate to them and perhaps claim intellectual ownership on them. As a result of such an understanding between the different members of the interdisciplinary development team, a use case writing style guide should be established formalising the way use cases steps are written, flows are structured, glossary terms are referenced, etc.

Several of the elicitation techniques described in other chapters (e.g. use case workshops, contextual inquiry, scenario walkthroughs) can and should be applied to get to the actual use cases and to improve them.

References:

Alur, Deepak, John Crupi, and Dan Malks, Core J2EE Patterns, *Second Edition, Sun Microsystems Press, Prentice Hall, 2003.*

Beck, Kent, Extreme Programming Explained*, Addison-Wesley, 2000.*

Beyer, Hugh and Karen Holtzblatt, Contextual Design: Defining Customer-Centered Systems. *Morgan Kaufmann Publishers, 1997.*

Bittner, Kurt and Ian Spence, Use Case Modeling*, Addison-Wesley, 2003.*

Cantor, Murray, Rational Unified Process for Systems Engineering*, The Rational Edge, www.therationaledge.com, August, September, and October, 2003.*

Cheesman, John and John Daniels, UML Components: A Simple Process for Specifying Component-Based Software. *Addison-Wesley Longman, 2001.*

Conallen, Jim, Building Web Applications with UML*, Second Edition, Addison-Wesley, 2002.*

Eeles, Peter, Kelli Houston, and Wojtek Kozaczynski, Building J2EE Applications with the Rational Unified Process*, Addison Wesley, 2002.*

Fowler, Martin, Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.*

Heumann, Jim, Generating Test Cases From Use Cases*, The Rational Edge, www.therationaledge.com, June, 2001.*

Jacobson , Ivar, Grady Booch, and James Rumbaugh, The Unified Software Development Process*, Addison Wesley Longman, 1998.*

Jacobson, Ivar, Use Cases -- Yesterday, Today, and Tomorrow*, The Rational Edge, www.therationaledge.com, March 2003.*

Jacobson, Ivar, Use Cases and Aspects - Working Seamlessly Together*, Journal of Object Technology, Vol.2, No.2, http://www.jot.fm/issues/issue_2003_07/column1, 2003.*

Kruchten, Philippe, The Rational Unified Process: An Introduction*, Addison Wesley, 2000.*

Kroll, Per and Philippe Kruchten, The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process*, Addison Wesley, 2003.*

Microsoft, Enterprise Solution Patterns Using Microsoft .NET*, http://msdn.microsoft.com/practices/type/Patterns/Enterprise/, 2003.*

OMG and IBM Rational, Unified Modeling Language Specification, Version 1.5*, http://www.rational.com/uml, 2003.*

Royce, Walker, Software Project Management: A Unified Framework. *Addison Wesley Longman, 1998.*

RUP 2003.06, Rational Unified Process, *IBM Rational Software, 2003.*

RUP SE v2.0 2003.06, Rational Unified Process for Systems Engineering Plug-In, *Rational Developer Network (www.rational.net), IBM Rational Software, 2003.*

Yu, Eric, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97) Jan. 6-8, 1997, Washington D.C., USA. pp. 226-235.*

Recommended Reading:

(*Self-Reference*) All diagrams presented in this chapter been created with IBM Rational XDE Developer for Visual Studio .NET 2003.06. These models and the executable code in C# as well as VB .NET for this example can be downloaded from http://haumer.net/rational/BAOS/

(Bittner 2003) A comprehensive coverage of use case techniques and practices for requirements management, including useful examples showing how use-case specifications evolve over time.

(Eeles at al. 2002) An example-based introduction to RUP and in particular OOAD for designing J2EE applications.

(Conallen 2002) An excellent introduction to the basics of web application development in the context of the RUP. This book also shows how to use the UML and RUP's OOAD workflow to model web applications, introduces the Web Application Extension to the UML used in the design diagrams of this chapter, as well as user experience modelling.

(Jacobson 2003) describes how use cases naturally support Aspect-Oriented Programming (AOP). He explains how use cases represent the "crosscutting concerns" of aspect-orientation, because they will be realised throughout the solution. Thus, a use case realisation represents one "aspect" of the architecture, i.e. a modular unit of crosscutting implementation. Download this paper to learn how the things you learned here fit with this new trend in software engineering.