# Precise Use Cases™

David Gelperin
LiveSpecs Software,
dave@livespecs.com

## Abstract

*This paper describes a precise form of use case that promotes the specification of actor options and alternative course conditions. Precise cases use a precise activity description language with a Structured English grammar. The design goals for the notation were to maximize understandability by application experts, not familiar with formal object-oriented concepts, and to supply sufficient information and structure to support both manual and automated test design. In addition to a detailed example, a development process, and guidelines for modeling and test coverage are provided.* ***The reader is assumed to have a basic understanding of use cases.***

## 1. Introduction

Use Cases can be described by diagrams [16] and text [6]. Diagrams emphasize relationships between actors and cases and between cases. Text describes actor/system interaction at different levels of detail ranging from short summaries to longer precise specifications. Diagrams and summaries are useful during early elicitation, while more precise text is useful during detailed analysis and functional specification.

While use cases, in general, and their current detailed formulations, in particular, provide useful information to manual test design, no current use case formulation is adequate for automated test designing that includes result checking. This paper describes a test-adequate text-based formulation, Precise Use Cases. Because this formulation makes more details explicit, it lowers the risk of misunderstandings and provides a firmer foundation for system development and manual testing as well.

Constantine [4, 5] has proposed approaches to some of the same issues. Although our work focuses on testing rather than usage-centered design, Precise Use Cases conform to Constantine's definition of essential cases and therefore are likely to be effective at supporting user interface design.

In addition, using precise cases should provide a firm foundation for usage-based reading [25], for building operational profiles [23], for generating user documentation [17], and for increasing the accuracy of test and development effort estimation [24].

The remainder of the paper is organized as follows: Sections 2 and 3 describe the information needs of testing and an approach to satisfying those needs. Section 4 describes the architecture of a Precise Use Case, Section 5 provides a substantial example, Section 6 comments on aspects of the example, and Section 7 specifies the Structured English constructs used to precisely describe courses of action. Section 8 discusses alternative formats for course descriptions, Section 9 provides a few modeling guidelines, and Section 10 describes a development process for suites of cases. Finally, Section 11 provides test coverage criteria and Section 12 discusses related work.

## 2. A testing need

Effective testing must include usage scenarios. Use cases can be a valuable source of usage information and usage testing ideas.

The following information associated with a single usage scenario (i.e., complete path through a use case) is necessary (but not sufficient) to design a usage-based test:

- Sequence of user actions and system actions in the scenario
- Scenario pre, invariant, and post conditions
- Alternative initial states of the domain objects associated with the scenario
- Alternative scenario trigger events
- Alternative inputs to the scenario
- Alternative Boolean values of the predicates determining the flow of the scenario

If use cases are specified informally or semi-formally [3, 6, 16, 19, 23], then high quality, usage-based test design remains a manual process and some of the information identified above will always be missing,

unclear, or incorrect. Formality and precision are prerequisites for automating test design and for automatically verifying use cases as well.

Cockburn [7] reports that people do not like to write formal cases. He argues that, formal or not, use cases can not automatically produce system designs, UI designs, or feature lists. Formal cases seem to be pain, without gain. However, formal cases can support automated design of (manual or automated) test scripts. In addition, attempts at precision can bring to light critical issues hidden in informal descriptions. Sometimes formality is warranted due to the nature and extent of product risk.

Note that software development always get to formality and precision if code is produced. So the question is not if formality and precision, but when. Since ambiguity often masks the seeds of failure, introducing formality and precision earlier in the development process can reduce the risk of critical information being missed or guessed at by the developers.

## 3. Satisfying the testing need

Precise Use Cases lie at the heart of a larger-scale description of application usage called a Precise Usage Model. In addition to use cases, these models contain:

- Application name & abstract
- Population description - including population diagram and actor profiles
- Application domain description - including essential class descriptions, states, transition rules, class relationships, domain invariants, and function limits
- Definitions of derived attributes and conditions - optional
- Dependencies between conditions – optional
- Non-functional requirements – optional
- Work breakdown structure - optional
- Action contracts – optional [12]

This additional information provides context for the precise cases including background on the actors involved as well as descriptions of the application objects, their relationships, their state changes, and their constraints. This additional information provides semantics for the conditions and actions in the use cases. Precise Usage Models are consistent with the specifications described by Larman [21]. A sample Precise Usage Model for a Library Management System can be found in [9].

The twin goals that drove the design of Precise Use Cases were to maximize the understandability of the cases by application experts, not familiar with formal object-oriented concepts, and to supply sufficient information and structure to support both manual and automated test design.

## 4. Architecture of a Precise Use Case (PUC)

The formulation of Precise Use Cases was greatly influenced by Cockburn's book on Use Cases [6], but violates several of his guidelines. It also draws on DeMarco's Structured English [8].

A ***Precise Use Case*** contains the following core information:

1. Case Name and optional Abstract/Context Diagram
2. Risk Factors
   - Frequency of occurrence:
     - [frequency range per time interval or event]
   - Impact of failure, likely & worst case
     - [high, medium, low]
3. Case Conditions
   - Invariants and Pre-conditions
4. Interactive Courses of Action
   - Success course
   - Performed course(s)
   - Alternative course(s)
     - Single-site course(s)
       - Exception handler (EH)
       - Replacement alternative (RA)
       - Single site option (SO)
     - Site Range course(s)
       - Non-extension alternatives
         - Multi-site options (MO)
         - Utility Exit (UE)
         - Application Exit (AE)
       - Extensions
         - Utility Options (UO)
         - Application Options (AO)

A Precise Use Case contains one ***success course*** and zero or more ***performed*** and/or ***alternative courses***.

Additional information such as version, modeler, sources, duration, trigger events (e.g., specific times), undoing cases, or likely successors may also be included in the case. Additional candidates for inclusion can be found in [4] and [6].

A success course description contains the following core information: (1) name, (2) success conditions i.e., invariant, pre, and post conditions of success, (3) actors, and (4) interaction steps. Every usage scenario (i.e., complete path through a use case) begins at the first step of the success course and must satisfy the case invariants and pre-conditions. Success scenarios, including those taking alternative courses, successfully exit the last step of the success course and thus satisfy all of the success conditions.

A performed course is invoked with a "does <performed course name>" action and contains the following core information: (1) name, (2) course conditions, (3) actors, and (4) interaction steps. Performed

courses are used to simplify the success course description as well as to encapsulate common course segments.

There are at least eight categories of alternative courses. All alternative course descriptions contain the following core information: (1) name, (2) insertion site location or range, (3) alternative conditions, (4) actors, and (5) interaction steps. The categories of alternative courses include:

1. ***Exception handlers*** deal with abnormal conditions that block the successful completion of a system process. An exception condition is checked and if TRUE, the corresponding handler is invoked. The handler may compensate for the abnormal condition or take other steps to deal with the situation.
2. ***Replacement alternatives*** are elements of a set of one or more mutually exclusive alternative courses, one of which must be followed e.g., payment alternatives.
3. ***Single-site options*** are guarded by a set of guard conditions i.e., invariants and pre-conditions. If the guards conditions are TRUE, then the option is included in the flow e.g., dealing with discount coupons during checkout.
4. ***Multiple site options*** are single options that can interrupt an actor's process flow at multiple sites e.g. withdrawing an item during checkout.
5. ***Utility exits*** are alternative courses in an actor's process flow, which end the interaction. They are supplied by the software platform and continually provide a human actor the alternative of ending an interaction at any point.
6. ***Application exits*** are alternative courses, supplied by the application, which end the interaction. These exits can be permanent or temporary. Permanent terminations entail explicit terminations, where the actor notifies the system that they are terminating the application, and implicit terminations such as shopping cart abandonment or customer disappearance. Temporary terminations e.g., suspended sales, entail suspended interaction and mean that for a specified time period, status information is stored awaiting the return of an actor. If the actor does not return, the status information is deleted and the termination becomes permanent.
7. ***Utility options*** are extension use cases supplied by the software platform. Since they are extensions, they contain the same core information as any use case.
8. ***Application options*** are extension use cases from the same application or a different application. Since they are extensions, they contain the same core information as any use case.

A more comprehensive description of these eight categories can be found in [11].

## 5. A sample PUC

The following example is derived from a specification appearing in [22]. Additional examples can be found in [10]

**Case Name**: Get [new] Seat on Reserved Flight

**Risk Factors**: Frequency of occurrence: 0 to 2 times per reservation
Impact of failure: *likely case* – **low**, open seating is a workaround
*worst case* – **medium**, open seating in a large plane with many expensive seats is likely to anger important passengers

**Case Conditions**:

| | |
|---|---|
| Invariants: | None |
| Preconditions: | For reservation system, status is active<br>For passenger, system access status is signed on |

**Interactions**

*Success Course*:

| **Passenger** | **Web-based Airline Reservation System** |
|---|---|
| 1. requests seat assignment | 2. requests a reservation locator |
| 3. provides a (corrected) reservation locator alternative | 4. searches for reservation |
| Until (reservation located or all reservation locator strategies tried), actors repeat 3 to 4 | |

| | |
|---|---|
| | 5. offers seating alternatives, **unless**<br>a. reservation not located or<br>b. seat previously assigned or<br>c. no seats are available or<br>d. no seats are assignable |
| 6. selects a seating alternative | 7. assigns selected seat **unless**<br>a. no seating alternative selected |
| | 8. If (For reservation, seat previously assigned)<br>returns previous seat to inventory<br>*Post-conditions* **--** For flight, previously assigned seat is available<br>Endif |
| | 9. confirms assignment<br>SUCCESS EXIT |

Success Course Conditions

Invariants:

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located

Pre-conditions:

For flight, some seats are assignable
Passenger wants to get or change seat assignment

Post-conditions:

For flight, seating alternative was selected
For reservation, selected seat is assigned

*Alternative Courses:*

Exception Handlers (EH)*:*

**EH 1** - 5a (reservation not located)

EH1 Invariants:

For reservation system, status is active
For passenger, system access status is signed on
For passenger, reservation not located

| Passenger | Web-based Airline Reservation System |
|---|---|
| | 1. Offers help making reservation<br>FAILURE EXIT |

**EH 2** - 5b (seat previously assigned)

EH2 Invariants:

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
For reservation, seat previously assigned

| Passenger | Web-based Airline Reservation System |
|---|---|
| | 1. offers to change seat assignment |
| 2. wants<br>a. to change seat assignment<br>b. no change | 3. If (Passenger wants to change seat assignment), CONTINUE<br>Else, provides help and SUCCESS EXIT |

**EH3** – 5c or 5d (no seats are available or assignable)

**EH3 Invariants**:

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
For flight, no seats are assignable

| Passenger | Web-based Airline Reservation System |
|---|---|
| | 1. If (check in),<br>places passenger on standby<br>*Post-cond* **--** For passenger, name on standby list<br>Else,<br>advises when assignment will be possible<br>Endif<br>FAILURE EXIT |

**EH4** - 7a (no seating alternative selected)

**EH4 Invariants**:

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
For flight, some seats are assignable
For reservation, no seating alternative selected

| Passenger | Web-based Airline Reservation System |
|---|---|
| | 1. If (For reservation, seat previously assigned)<br>SUCCESS EXIT<br>Endif |
| | 2. If (day of flight),<br>advises to get assignment at check in<br>Else,<br>advises to try later<br>Endif<br>FAILURE EXIT |

Figure 1. Example of the reserved seat functionality in a web-based airline reservation system

## 6. Comments on the example

The use of explicit exception conditions assures that each exception handler has a visible ***point of origin***. A point of origin is marked by an individual exception condition preceded by the keyword ***unless***. For example in step 7 of the success course, some alternative must be selected before the system can assign a seat. An exception handler must deal with the situation when no seat is selected. A point of origin may be in any course of action. In the example above, steps 5 and 7 in the success course contain points of origin, while the exception handlers do not.

Precise use cases encourage the explicit identification of **valid input alternatives**. When there are only a few simple alternatives, they may be provided directly within the step as they are in the second step of exception handler 2. Otherwise, the alternatives should be specified in an associated glossary of input alternatives. The alternatives referenced in step 3 of the success course would be found in such a glossary.

## 7. Structured English in PUC courses

Structured English was originally developed to describe the actions of individual analysis entities [8]. The version used in PUCs is intended to describe interactions between multiple entities.

### 7.1. Courses of action

The types of courses that can appear in a PUC are listed below:

- **Success Course** -- each PUC has exactly one of these and every scenario begins at its first step.
- **Performed Course** -- refers to action sequences used to specify common or simplifying course segments.
- **Single-Site Course** -- refers to exception handlers, replacement alternatives, or single-site options.
- **Exception Handling Macros** -- analogous to macros in programming and classes in object-oriented analysis. Macros are used to group exception handlers that have predictable differences. They use a common set of steps and handler specific information to generate a specific exception handler. Macros make the common structure of multiple handlers explicit.
- **Non-extension Course** -- refers to multi-site options, utility exits, or application exits i.e., non-extension alternatives.
- **Extension Course** -- refers to a utility or application option i.e., extension alternatives.

### 7.2. Actions

<action> [, **unless** <exception conditions [N%]>]

An action has the structure <verb> <direct object [modifiers]> [<indirect object>]. See the many examples in this paper.

**does** <performed course name>
[**using** <parameter1>, <parameter2>, …]
[, **unless** <exception conditions [N%]>]

**selects/wants**
a. <valid input alternative [N%]>
b. <valid input alternative [N%]>
…

In this paper, specific action names signal inputs, outputs, and state changes.

Inputs are signaled by:
requests, provides, enters, selects, and wants

Outputs are signaled by:
advises, approves, confirms, displays, prints, offers, requests, and provides

State changes are signaled by:
accepts, deposits, places, records, returns, stores, and updates

The N% annotation of input alternatives and exception conditions is the (integer) probability that the alternative will be selected or the condition will be TRUE. The sum of probabilities of all alternatives must be 100%. The sum of probabilities of all exception condition must be less than 50%.

Examples:

- places passenger on standby
- approves drop request, **unless** course not added 1%
- **does** "adds a book" **using** "Writing Effective Use Cases",
- **unless** book is already present
- **wants** aisle seat
- **selects**
  a. credit or debt card 38%
  b. sufficient cash or equivalent 62%

### 7.3. Handling intermediate failure

We try, we fail, we compensate or try something different, and we succeed. Intermediate failure is a normal component of successful goal-directed behavior. There are three basic approaches to handling such failures: (1) report, (2) compensate, and (3) retry using a different approach.

<action>, **unless** <exception conditions [N%]>

Exception handlers can effectively model reporting and compensating behavior. Exception conditions in an **unless** clause are Booleans that all must evaluate to FALSE in order for an action or performed course to be performed. Each exception condition has a corresponding handler that is invoked if its condition is the first to evaluate to TRUE. The exception handler may attempt to compensate for the condition and may succeed or fail in this attempt. If compensation is not attempted or the attempt fails, then following the termination style [15], the handler will record/report the condition, possibly do other things, but will always cause a FAILURE EXIT from the course containing its invoking action-unless statement i.e., the course raising the exception. If correction succeeds, then (following the resumption style [15],) control returns to the point it would have reached initially had the exception condition been FALSE. This permits the evaluation of conditions to continue.

**does alternative strategies**
<strategy1 course>
[**using** <parameter1>, <parameter2>, …]
<strategy2 course>
[**using** <parameter1>, <parameter2>, …]
[<failure preparation course>
[**using** <parameter1>, …]]
**FAILURE EXIT**
**End**

A different construct (i.e., does alternative strategies) is needed to model the use of heuristic alternatives that are expected to fail occasionally. This construct contains an ordered sequence of performed courses for different strategies that are performed until one strategy succeeds or the final FAILURE EXIT clause is executed. Each strategy course, except failure preparation, must contain a SUCCESS EXIT and may contain one or more FAILURE EXITs. The failure preparation course must not contain a SUCCESS EXIT.

## 7.4. Selection

**If** (<selection conditions [N%]>)
<action> or does < ..>
[**Else**
<action> or does <…>]
[**Endif**]

Examples:

- If ((customer offers credit or debit card)
  and (payment amount is approved) 33%)
  accepts card payment
  Endif
- If (customer rejects all card payments)
  does "handles cash payment",
  **unless** cash or equivalent amount is insufficient
  Endif
- If (check in time and no assignable seat 2%),
  places passenger on standby
  Else
  advises when assignment will be possible
  Endif

Avoid directly embedding *if* statements within *if* statements. The embedded *if* should be placed in a performed course.

## 7.5. Repetition

**Until** or **While** <conditions> or **For each** <item name>,
**actors repeat** <step range>
or **actors repeat** <performed course name>

The step range must be immediately above or below the repetition statement.

Repetition must always encompass both actor and system actions.

| | |
|---|---|
| 3. provides a (corrected) reservation locator alternative | 4. searches for reservation |
| **Until** (reservation located or all reservation locator strategies tried), **actors repeat** 3 to 4 | |

Figure 2. Example of using a repetition construct

## 7.6. Inclusion of common actions

Actions and course of action that are common among a set of use cases can be included in a use case using
Include <common actions name>

Inclusion can be used for common: performed courses, exception handlers, single-site options (e.g. logon), utility exits, or application exits.

## 7.7. Scenario flow

Each usage scenario starts with the ENTER in the success course and ends with an EXIT. The EXITs may be SUCCESS or FAILURE.

CONTINUE means a return to the invoking course. If all conditions in the set of conditions (e.g., exception conditions) that triggered the CONTINUEing course have not been evaluated, then evaluate the unevaluated conditions, otherwise execute the next step.

ITERATE only appears in steps that are being performed within a repeat loop. It means that the flow should continue at the repeat statement and proceed normally depending on the evaluation of the repetition conditions.

### 7.8. Ambiguity of order in a step

For the following action:
actor provides
a. name
b. address
c. phone number
the order in which inputs are provided is **not** specified.

For the step actions: "requests checkout and provides items", the order of actions is **not** specified.

If order is important, write multiple steps.

For the following action:
system approves add request, **unless**
a. course is full
b. prerequisites are not satisfied
c. conflict in schedule
d. course load is unacceptable
the order in which exception conditions are evaluated is **not** specified. Therefore, if multiple conditions are TRUE, any of the corresponding exception handlers might be invoked.

## 8. Alternative formats for courses

The activity within courses can be formatted in one-column, two-column, or three-column tables. In a **one-column table**, the actions of all actors and the system, appear in a single stream. Therefore, to avoid confusion, each action description must explicitly name its associated actor or system.

The example above shows a **two-column format**. This is useful for describing dialogs between one or more actors and a fully automated system. System actions appear in the second column and the actions of all actors appear in the first. All actors are named in the heading of the first column. If there are multiple actors, each action in the first column should explicitly name its actor.

A **three-column format** is useful for describing trialogs i.e., when actors interact with a combination of technology and people. For example, at an airport or supermarket, we find ticket agents and cashiers as well as technology. The people and the technology are inside the system from the perspective of the actor. As with the two-column format, if there are multiple actors or multiple people or both, they should all be named in the headings and explicitly named with the actions. This format should not be used for multiple actors and a fully automated system.

## 9. PUC modeling guidelines

a. A use case may require access to specific resources (e.g., document files or applications) to be successful and an actor may be able to enable this access (e.g., by opening a file or signing onto an application). In this situation, define a single-site option to be inserted at the start of the use case that specifies resource enabling if the resources are not accessible e.g. actor log on.

Using this approach removes most dependencies between use cases, except for multiple cases that reference the same application object. For example, if one use case creates a document named "bob" and the next one tries to create a document with the same name, the actor should get a different response than if the first use case had not occurred.

b. Sometimes, there may be confusion in choosing between an "<action>, **unless**" and an "if (<selection condition>)". The guidelines are:
(1) when an action is always to be done, unless some exception condition is TRUE, use an "action, **unless**" . An exception condition is one that is **intended** to be exceptional. Sometimes it may actually be the rule, e.g., the printer may usually be inoperative.
(2) when an action is to be done sometimes, but not other times, use an "if (<selection condition>)" e.g., for adding some courses and deleting others use an if.

c. Model the flow of information, not user interface design nor navigation details. Use input actions like - enters, provides, or requests rather than fills in a form, clicks a button, or links to a specific web page. See the discussion of essential use cases in [4].

Additional recommendations can be found in [11].

## 10. A development process for suites of PUCs

PUCs are too detailed for effective use during early interactive elicitation. Their development can be characterized as the descriptive (as opposed to prescriptive) programming of interacting processes. Therefore, development of PUCs should start after some requirements have already been captured using a less formal style of use case [26]. Initial forms can be as informal as the user stories used in XP development [1] or any of the styles described in [6] or [21]. PUC development and informal elicitation can co-occur.

PUCs would be developed "off line" by an analyst or tester, but they require stakeholder support to answer the inevitable questions and to review the results. The issues uncovered during PUC development, if uncovered before system design, could be PUC's most important contribution to quality.

The following sequential development process for a suite of PUCs was derived from [6].

1. Specify the system scope and boundaries – use text and a context diagram
2. Brainstorm and exhaustively list the primary and supporting stakeholders [21] plus their (optional) profiles
3. Cluster primary stakeholders supporting the same organizational functions
4. Brainstorm and exhaustively list the goals of each stakeholder
5. Brainstorm and exhaustively list the conceptual classes [21] associated with these goals
6. Specify attributes and value options for each conceptual class
7. Review and modify goals and conceptual classes as needed
8. Specify definitions and dependencies as needed
9. Specify non-functional requirements as needed
10. For each goal, repeat steps 11 to 12
11. Select a goal and identify alternative success courses
12. For each success course
    a. Specify all stakeholders
    b. Specify risk factors
    c. List the steps
    d. Specify invariants and pre-conditions of success as well as post-conditions for the success course
    e. Specify invariant and pre conditions for the use case
    f. Identify alternative courses that need to be added and specify their steps
    g. Use performed courses to encapsulate repeated step sequences or to simplify course descriptions
    h. For each step, brainstorm and exhaustively list exception conditions
    i. For each exception condition, specify an exception handler using exception handling macros to group similar patterns
    j. Review and modify the invariant, pre, and post conditions as needed and add intermediate conditions where useful
13. Review suite of PUCs with stakeholders and subject matter experts and modify as needed

## 11. Test coverage criteria for PUCs

To describe test coverage, we need a few definitions. ***Single-goal use cases*** contain no application options i.e. extensions. **Multiple-goal** cases contain one or more application options in addition to the steps that accomplish the goal. **Transitional use cases** cause object state transitions e.g., returns a book, while **non-transitional cases** do not e.g. display copies borrowed. An ordered pair of transitional cases referencing the same object may be valid or invalid. For example, a customer record can only be updated, after it is created.

For a set of PUCs, a test suite should cover:

1. every (valid) ordered pair of single-goal cases where one or both are non-transitional
2. every (valid & invalid) ordered pair of transitional cases that reference the same object
3. every multiple-goal case

If this is too expensive, then either cover every ordered pair of "high or medium risk" use cases or just cover every ordered pair of "high risk" use cases. In either case, cover every multiple-goal case.

Within a single-goal case, a test suite should cover every step in the success course, performed courses, and alternative courses.

A use case will normally contain multiple scenarios i.e., complete execution paths, because it contains repetition cycles and alternative courses. A scenario may have multiple interpretations, because it contains input alternatives and derived conditions for inputs (e.g., invalid) or states (e.g., unavailable). There are usually many distinct ways to interpret a derived condition e.g., many distinct ways to be invalid or state.

For a use case, the test suite should cover every:

1) repetition cycle 0 (if possible), 1, and an even number of times
2) unique cause + 1 interpretation [13] of selection, repetition, alternative course, and derived conditions
3) input alternative

In addition, the test suite should cover:

1) output boundaries
2) valid and invalid input boundary values [18].

If one or more of these criteria causes the test suite to be "too big" (i.e., unaffordable because criteria are too strong), risk information must be used to weaken or delete criteria.

Consider adding the "every pair of input partition values" [28] criterion, if you have an adequate budget surplus and its incremental cost-effectiveness has been shown.

## 12. Related work

Others have proposed adding rigor to usage descriptions. Proposals have entailed Message Sequence Charts and Process Algebra [2], Abstract State Machine Language [14], Activity Diagrams [20], and Modular Petri Nets [29]. These alternatives, however, do not present the application choices, essential to effective test design, as clearly and completely as PUCs. More importantly, these other approaches have not evolved from informal forms of the same technique. Effective informal forms make transition to formality much easier.

The Object Constraint Language [27] has been proposed for specifying conditions, but does not satisfy the understandability requirement.

## 13. Acknowledgements

This work owes an enormous debt to Alistair Cockburn's book [6]. I am also grateful to Ian Alexander, James Bach, Sofia and Stanislov Passova and Natan Aronshtam for helpful comments.

## 14. References

[1] Extreme Programming info available at www.extremeprogramming.org/rules/userstories.html

[2] Andersson, Michael and Bergstrand, Johan **"Formalizing Use Cases with Message Sequence Charts"** MS Thesis, Lund Institute of Technology May 1995 Available at www.efd.lth.se/~d87man/EXJOBB/Formal_Approach_to_UC.html

[3] Armour, Frank and Miller, Granville **Advanced Use Case Modeling** Addison Wesley 2001

[4] Constantine, Larry and Lockwood, Lucy **"Structure and Style in Use Cases for User Interface Design"** Available at www.foruse.com/Resources.htm#style

[5] Constantine, Larry and Lockwood, Lucy **Software for Use** ACM Press Addison Wesley 1999

[6] Cockburn, Alistair **Writing Effective Use Cases** Addison-Wesley 2001

[7] Cockburn, Alistair **"Use Cases, Ten Years Later"** STQE, SQE, Vol. 4, No. 2, March/April 2002

[8] DeMarco, Tom **Structured Analysis and System Specification** Prentice-Hall 1978

[9] Gelperin, David **"Precise Usage Model for Library Management System"** Available at www.LiveSpecs.com

[10] Gelperin, David **"Precise Use Case Examples"** Available at www.LiveSpecs.com

[11] Gelperin, David **"Modeling Alternative Courses in Detailed Use Cases"** Available at www.LiveSpecs.com

[12] Gelperin, David **"Specifying Consequences with Action Contracts"** Available at www.LiveSpecs.com

[13] Gelperin, David **"Testing Complex Logic"** Available at www.StickyMinds.com

[14] Grieskamp, Wolfgang, Lepper, Markus, Schullte, Wolfram, and Tillmann, Nikolai "**Testable Use Cases in the Abstract State Machine Language**" in *Proceedings of Asia-Pacific Conference on Quality Software (APAQS'01)*, December 2001.

[15] Garcia, Alessandro F., Rubira, Cecilia M. F., Romanovsky, Alexander, Xu, Jie. "**A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software**". Journal of Systems and Software, Elsevier, Vol. 59, Issue 2, November 2001, pp. 197-222.

[16] Ivar Jacobson et al *Object-Oriented Software Engineering* Addison-Wesley 1992

[17] Jungmayr, Stefan and Stumpe, Jens "**Another Motivation for Usage Modeling: Generation of User Documentation**" Proceedings of CONQUEST ' 98, Nuernberg, Germany, September 28-29, 1998

[18] Kaner, Cem, Falk, Jack, Nguyen, Hung Quoc **Testing Computer Software** John Wiley 1999

[19] Kulak, Daryl and Guiney, Eamonn **Use Cases: Requirements in Context** ACM Press Addison-Wesley 2000

[20] Kosters, Georg, Six, Hans-Werner, and Winter, Mario "**Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications**" Requirements Engineering Journal, Vol. 6, No. 1 2001 pp 3-17

[21] Larman, Craig **Applying UML and Patterns** Prentice-Hall PTR 2002

[22] Rumbaugh, James "**Getting Started : Using Use Cases to Capture Requirements"** Journal of OO Programming, SIGS Publications, Vol. 7, No. 5, Sept 1994 pp 8-10, 12, & 23

[23] Runeson, Per and Regnell, Bjom "**Derivation of an integrated opertational profile and use case model"** 9th Symposium on SRE IEEE press Nov. '98 pp. 70-79

[24] Schneider, Geri and Winters, Jason P. **Applying Use Cases** Addison Wesley 1998

[25] Thelin, Thomas, Runeson, Per, and Regnell, Bjorn "**Usage-based reading – an experiment to guide reviewers with use cases**" Information and Software Tech. 43 (2001) pp. 925-938

[26] Wiegers, Karl E. "**Listening to the Customer's Voice"** Software Development, March 1997

[27] Warmer, Jos and Kleppe, Anneke. **The Object Constraint Language** Addison Wesley 1999

[28] Williams, Clay and Paradkar, Amit "**Efficient Regression Testing of Multi-Panel Systems**" IEEE International Symposium on Software Reliability Engineering, Nov. 1999

[29] Woo, Jin Lee, Sung, Doek Cha, and Yong, Rae Kwon "**Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering**" IEEE Transactions on Software Engineering, Vol. 24, No. 12, December 1998