Tutorial #9:
The Art of Writing Use Cases

Presented by
Rebecca J. Wirfs-Brock and
John A. Schwartz

Tutorial Presented at OOPSLA 2002:
The ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications

Copyright is held by the author(s).
OOPSLA'02, November 4-8, 2002,
Seattle, Washington, USA.
2002 ACM 02/0011.

The Art of Writing Use Cases

Rebecca Wirfs-Brock
rebecca@wirfs-brock.com
John Schwartz
Schwajoh@ohsu.edu

www.wirfs-brock.com

Copyright 2002, Wirfs-Brock Associates, Inc. 1

Goals

The goal of this tutorial is to enable you to

- Effectively create elements of a use case model: actors, use cases, glossaries and use case diagrams
- Choose the form of use case description you need
- Develop or customize a use case template for your specific project needs
- Effectively write and critique use cases
- Add necessary detail and precision, and relate use cases to other requirements descriptions

Copyright 2002, Wirfs-Brock Associates, Inc. 2

Wirfs-Brock Associates Copyright 2002. All rights reserved

Agenda

A Use Case Model: Use Cases, Actors, Glossaries, Diagrams

Finding and Naming Use Cases and Actors

Dissecting a Use Case Template

Writing Tips and Guidelines

Alternative Paths

Emphasis

Copyright 2002, Wirfs-Brock Associates, Inc.

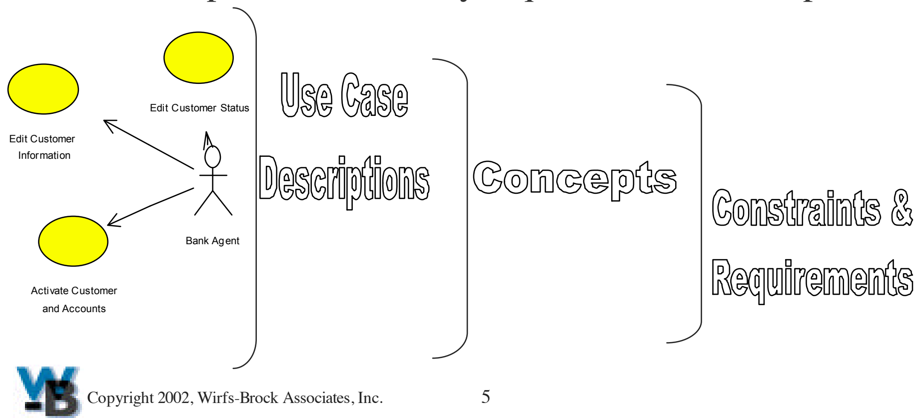
A Use Case Model: Use Cases, Actors and Glossaries

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

Use Case Model

A Use Case Model includes structured use case descriptions that are grounded in well-defined concepts constrained by requirements and scope



Copyright 2002, Wirfs-Brock Associates, Inc. 5

Use Cases

Use cases describe a system from an external usage viewpoint

A collection of task-related activities...

- Making a Payment
- Transferring Funds Between Accounts

... describing a discrete "chunk" of the system

They do not describe:

- user interfaces
- performance goals
- application architecture
- non-functional requirements

Copyright 2002, Wirfs-Brock Associates, Inc. 6

Wirfs-Brock Associates Copyright 2002. All rights reserved

Use Case Function and Forms

The Writing Task	The Best Form To Use
Present Overview	**Narrative**
Describe simple sequence of events	**Scenario**
Emphasis actor-system interaction	**Conversation or Essential Use Case**

Copyright 2002, Wirfs-Brock Associates, Inc. 7

Narrative Form

Free-form text in paragraph format

Describes the intent of the user in performing the use case

Describes high-level actions of the user during the use case

Refers to key concepts from the problem domain that are involved in the use case

Copyright 2002, Wirfs-Brock Associates, Inc. 8

Wirfs-Brock Associates Copyright 2002. All rights reserved

The Narrative Form

Make a Payment

The user can make online payments to vendors and companies known to the bank. Users can apply payments to specific vendor accounts they have. There are two typical ways to make payments: the user can specify a one-time payment for a specific amount, or establish regular payments to made on a specific interval such as monthly, bi-weekly, or semi-annually.

Copyright 2002, Wirfs-Brock Associates, Inc. 9

Scenario Form

One particular path through a use case written from the actor's point of view

Describes a sequence of events or list of steps to accomplish. A step may be optional.

Each step is a simple statement

May describe:

- Actors' intentions (what they accomplish, not the minutiae of their gestures or individual keystrokes)
- System responsibilities and actions

Copyright 2002, Wirfs-Brock Associates, Inc. 10

Wirfs-Brock Associates Copyright 2002. All rights reserved

The Scenario Form

Register Customer With Automatic Activation

1 User enters registration information:

Required information: user name, email address, desired login ID and password, and confirmation password

One of: account number and challenge data, or ATM # and PIN

Optional: language choice and company

2 System checks that password matches confirmation password.

3 System validates required fields and verifies uniqueness of login ID

4 System verifies customer activation information.

5 System creates and activates customer online account.

6 System displays registration notification.

Copyright 2002, Wirfs-Brock Associates, Inc. 11

Conversation Form

A dialog between the actor and the system that emphasizes interactions and shows cause and effect

Can show optional and repeated actions

Each action can be described by one or more substeps

May describe:

- Actor intentions and actions
- System responsibilities and actions

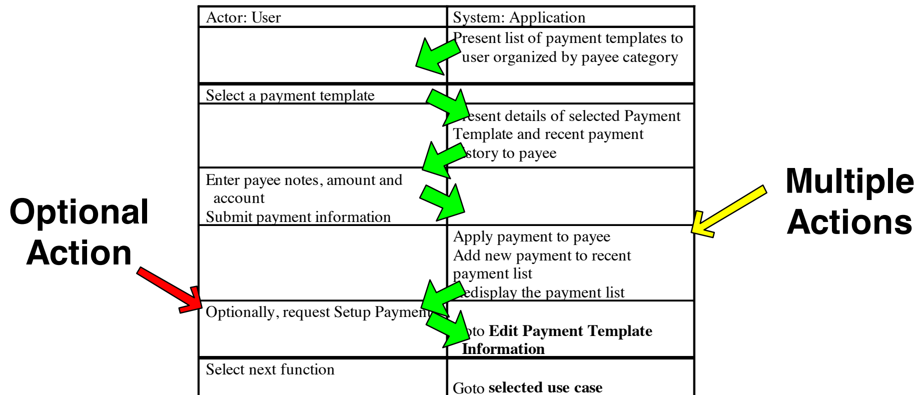
Copyright 2002, Wirfs-Brock Associates, Inc. 12

Wirfs-Brock Associates Copyright 2002. All rights reserved

The Conversation Form

Make A Payment

General Flow



Actor: User	System: Application
	Present list of payment templates to user organized by payee category
Select a payment template	
	esent details of selected Payment Template and recent payment story to payee
Enter payee notes, amount and account Submit payment information	
	Apply payment to payee Add new payment to recent payment list edisplay the payment list
Optionally, request Setup Paymen	to **Edit Payment Template Information**
Select next function	Goto **selected use case**

Invoking Another Use Case

Copyright 2002, Wirfs-Brock Associates, Inc. 13

Comparing the Three Forms

Form	Strengths	Weaknesses
Narrative	• Good for high-level summaries • Can be written to be implementation independent	• Easy to write at too high or too low a level • Not suitable for complex descriptions • Can be ambiguous about who does what
Scenario	• Good for step-by-step sequences	• Hard to show parallelism or arbitrary ordering • Can be monotonous
Conversation	• Good for seeing actor-system interactions • Can show parallel and optional actions	• Easy to write at too detailed level: pseudo pseudo-code • Only two columns: What about multiple actors?
All Forms	• Informal	• Informal

Copyright 2002, Wirfs-Brock Associates, Inc. 14

Wirfs-Brock Associates Copyright 2002. All rights reserved

Use Cases Vary by Abstraction Level

Steve Registers for English 101, or

Student Registers for Course, or

User Uses System, or

Student Registers for Variable Credit Course, or

Student Registers for Music Course

Copyright 2002, Wirfs-Brock Associates, Inc. 15

Use Cases Vary in Scope

How broadly do we focus our descriptions?

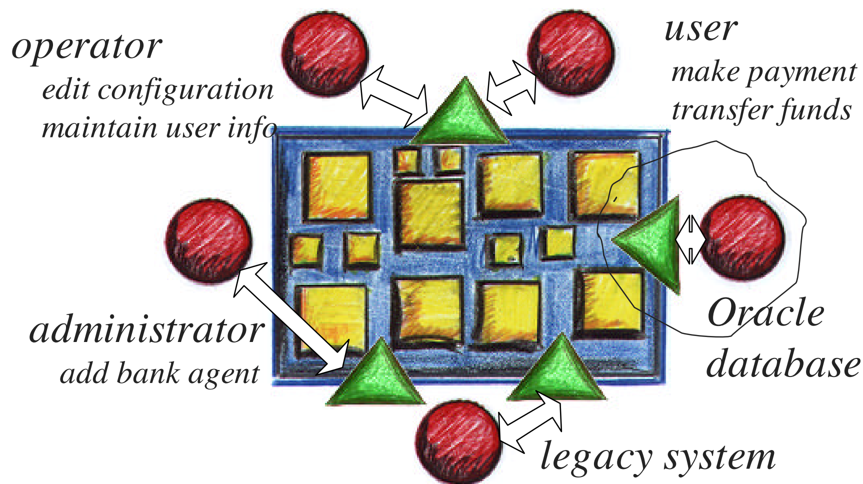
- component: describing the interactions with a web applet
- application: describing how the user interacts with online banking services
- organization: describing how the user interacts with online banking functions that connect with banking applications…

We typically start by describing application level scope

Copyright 2002, Wirfs-Brock Associates, Inc. 16

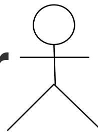
Wirfs-Brock Associates Copyright 2002. All rights reserved

Different Perspectives



Copyright 2002, Wirfs-Brock Associates, Inc. 17

Actor



Any one or thing that interacts with the system causing it to respond to business events

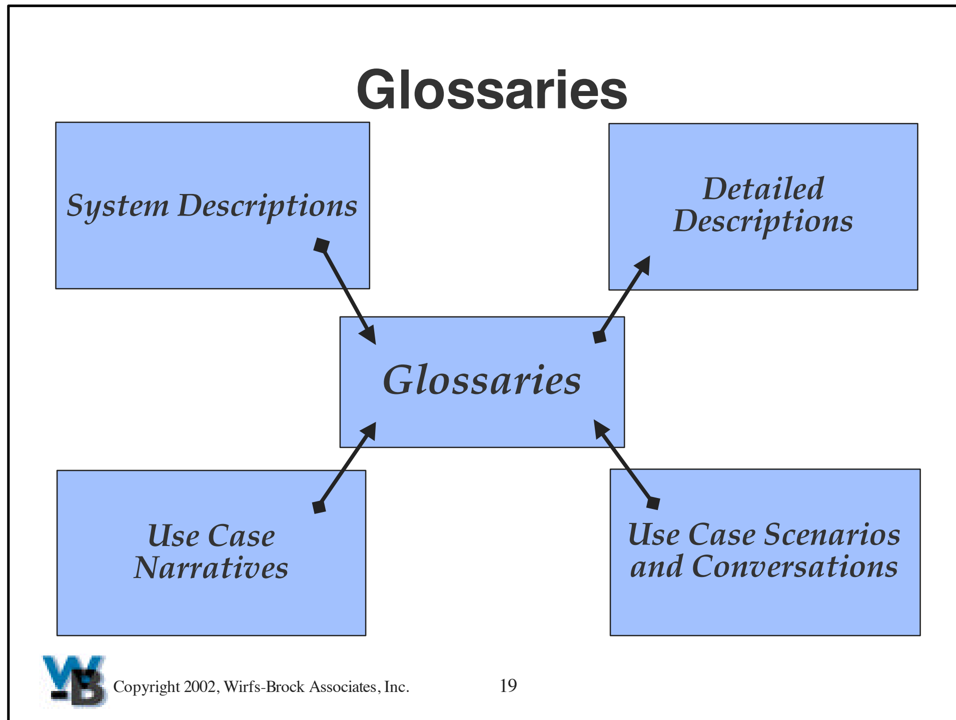
Something that

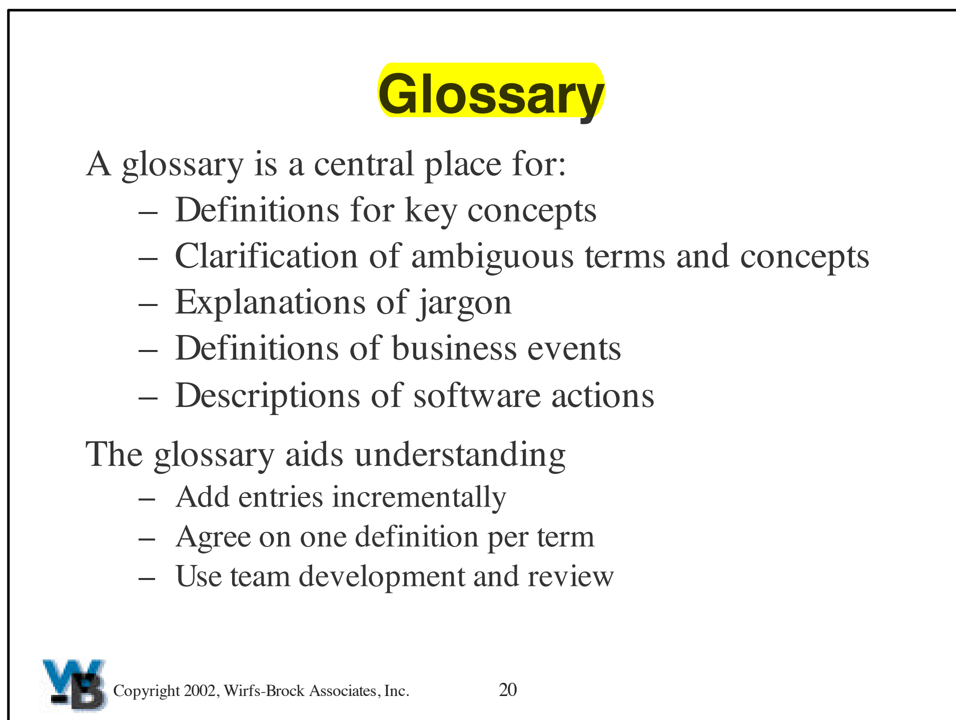
- stimulates the system to react (primary actor), or
- responds to the system's requests (secondary actor)

Something we don't have control over

Copyright 2002, Wirfs-Brock Associates, Inc. 18

Wirfs-Brock Associates Copyright 2002. All rights reserved





Wirfs-Brock Associates Copyright 2002. All rights reserved

Writing Glossary Entries

Identify a concept and its distinguishing characteristics

More than a synonym for a word

- Explain why a concept is important
- Mention typical sizes or values
- Clarify likely misunderstandings
- Show an example
- Explain graphical symbols
- Relate entries …

Identifies a way of mentally dividing reality for purpose of talking or thinking

Copyright 2002, Wirfs-Brock Associates, Inc. 21

A Good Form for Definitions

Name of Concept related to a **Broader Concept + Characteristics**

Contrast: A compiler is a program that translates source code into machine language

With a definition that leaves out context: A compiler translates source code into machine language

What performs this translation? A computer? A person?

Copyright 2002, Wirfs-Brock Associates, Inc. 22

Wirfs-Brock Associates Copyright 2002. All rights reserved

Improving Glossary Definitions

Contrast the original:

Account In the online banking system there are accounts within the bank which customer-users can access in order to transfer funds, view account balances and transaction historical data, or make payments. A customer has one or more accounts which, once approved by the bank can be accessed. The application supports the ability for customers to inform the system of new accounts, and for the customer to edit information maintained about the accounts (such as name and address information).

With a definition that says what an account is and how it is used:

Account An account is ***a record of money*** **deposited at the bank for checking, savings or other uses.** A customer may have several bank accounts. Once a customer's account is activated for online access, account information can be reviewed and transactions can be performed via the internet.

Copyright 2002, Wirfs-Brock Associates, Inc. 23

Another Glossary Entry

Automatic activation. Automatic activation is ***an optional function of the online banking software*** **that enables immediate access to bank accounts.** *To automatically activate an account, a customer provides information that associates him with an account, called challenge data, such as mother's maiden name. Online access is granted once the challenge data is validated against bank records. Alternatively, the customer can supply a valid ATM bankcard number and PIN. All accounts associated with that ATM card would be activated.*

Characteristics:

- Optional feature
- Details of how the automatic activation function works

Copyright 2002, Wirfs-Brock Associates, Inc. 24

Wirfs-Brock Associates Copyright 2002. All rights reserved

Use Pictures to Relate Concepts

wire center— the geographical area served by a central office

central office— a building where local call switching takes place

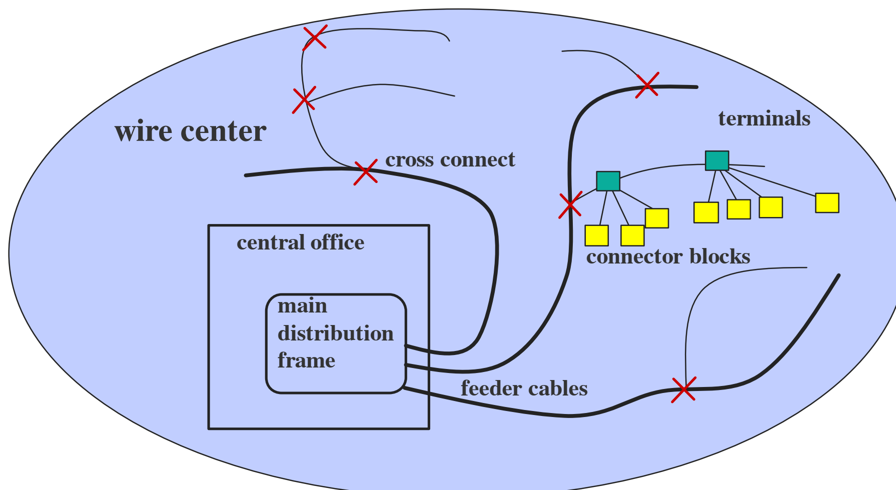
main distribution frame— a large connector at a central office, which connects switching equipment to feeder cables

feeder cable— a large cable that connects to the main distribution frame at a central office and feeds into distribution cables

distribution cable— a cable that connects between a feeder cable and one or more terminals

Copyright 2002, Wirfs-Brock Associates, Inc.

A Picture Relating Hierarchical Concepts



Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

Define Acronyms
and
Their Concepts

Example:

OSS— Operations Support System: As defined by the FCC, a computer system and/or database used at a telephone company for pre-ordering, ordering, provisioning, maintenance and repair, or billing

Copyright 2002, Wirfs-Brock Associates, Inc.

Use Case Levels

Use cases can be written at differing levels of abstraction and scope. Each serves a purpose:

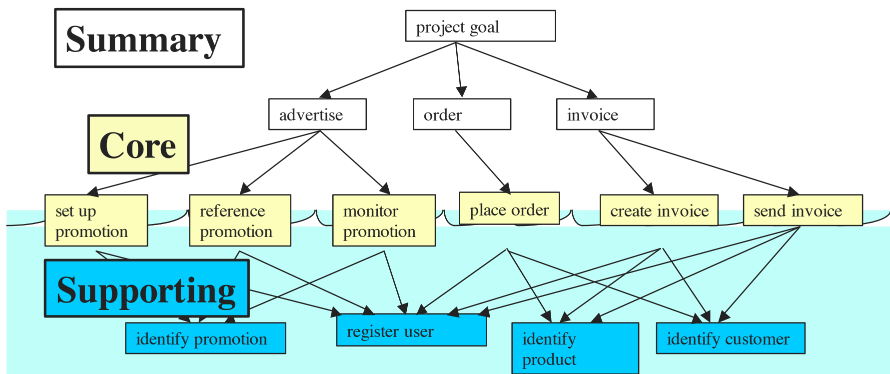
- **Summary**— General descriptions and sweeping overviews of system functionality or business processes
- **Core**— Task-related descriptions of users and how they interact with the system; descriptions of a specific business process
- **Supporting**— Descriptions of lower-level activities used to complete subparts of a core use case
- **Internal**— Descriptions of the behaviors of and interactions between internal system components

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

Use Case Models Vary in Shape

Sailboat – balanced use cases
Classical business functions

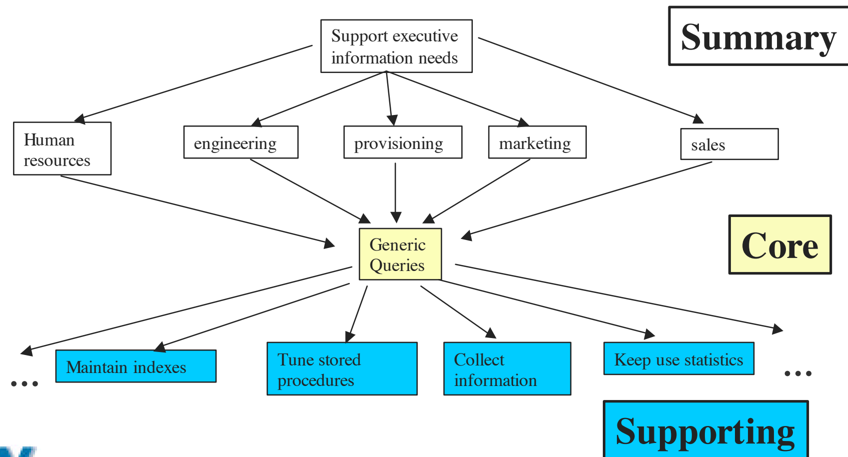


Alistair Cockburn, *Humans and Technology*

Copyright 2002, Wirfs-Brock Associates, Inc. 29

Use Case Models Vary in Shape

Hourglass—small core
Ad hoc information query/data warehousing



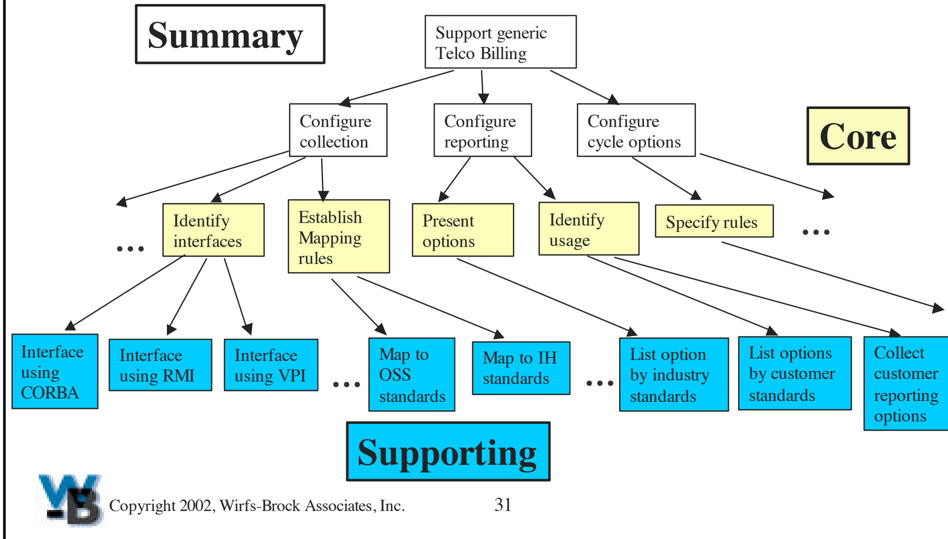
Copyright 2002, Wirfs-Brock Associates, Inc. 30

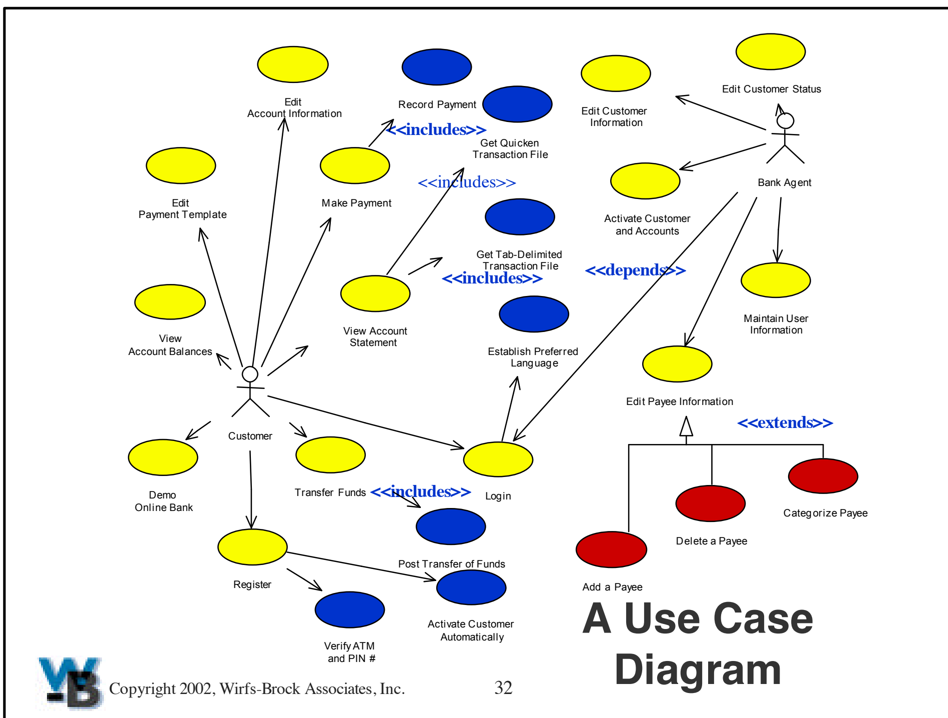
Wirfs-Brock Associates Copyright 2002. All rights reserved

Use Case Models Vary in Shape

Pyramid—supporting use case rich

Software application development environment





Wirfs-Brock Associates Copyright 2002. All rights reserved

Use Cases Can Be Related

UML defines these relationships between use cases:

- **Dependency—** The behavior of one use case is affected by another
  - Being logged into the system is a pre-condition to performing online transactions. **Make a Payment** depends on **Log In**
- **Includes—** One use case incorporates the behavior of another at a specific point
  - **Make a Payment** includes **Validate Funds Availability**
- **Extends—** One use case extends the behavior of another at a specified point
  - **Make a Recurring Payment** and **Make a Fixed Payment** both extend the **Make a Payment** use case
- **Generalize—** One use case inherits the behavior of another; it can be used interchangeably with its "parent" use case
  - **Check Password** and **Retinal Scan** generalize **Validate User**

Copyright 2002, Wirfs-Brock Associates, Inc. 33

Finding and Naming Use Cases and Actors

Copyright 2002, Wirfs-Brock Associates, Inc. 34

Wirfs-Brock Associates Copyright 2002. All rights reserved

Finding Use Cases

Describe the functions that the user will want from the system in terms of goals the system helps them accomplish. “I want to …

- transfer money between accounts”
- get money from my account”
- make payments”
- set up vendors for automatic payments”

Describe the operations that create, read, update, and delete information

Describe how actors are notified of changes to the internal state of the system and how they communicate information about external events to the software

Copyright 2002, Wirfs-Brock Associates, Inc.

Naming Use Cases

Name a use case with a verb-noun phrase that states the actor’s goal

Use concrete, “strong” verbs instead of generalized, weaker ones. Weak verbs may indicate uncertainty

- Strong Verbs: create, merge, calculate, migrate, receive, archive, register, activate
- Weaker Verbs: make, report, use, copy, organize, record, find, process, maintain, list

Be explicit. Use specific terms. They are stronger

- Strong Nouns: property, payment, transcript, account
- Weaker Nouns: data, paper, report, system, form

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

Naming Actors

Group individuals according to their common use of the system. Identify the roles they take on when they use the system

Each role is a potential actor

Name each role and define its distinguishing characteristics. Add these definitions to your glossary

Don't equate job title with role name. Roles cut across job titles

Use the common name for an existing system; don't invent a new name to match its role

Don't waste time debating actor names

Copyright 2002, Wirfs-Brock Associates, Inc. 37

Places to Look for Actors

Who uses the system?

Who gets information from this system?

Who provides information to the system?

What other systems use this system?

Who installs, starts up, or maintains the system?

Copyright 2002, Wirfs-Brock Associates, Inc. 38

Wirfs-Brock Associates Copyright 2002. All rights reserved

Dissecting A Use Case Template

Copyright 2002, Wirfs-Brock Associates, Inc.

A Use Case Template

Use case name

Preamble

Use case body (narrative, scenario or conversation)

Supplementary details and constraints

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

The Preamble: Setting the Stage

Level— summary, core, supporting, or internal use case?

Actor(s)— role names of people or external systems initiating this use case

Context— the current state of the system and actor

Preconditions— what must be true before a use case can begin

Screens— references to windows or web pages displayed in this use case

Copyright 2002, Wirfs-Brock Associates, Inc. 41

Supplementary Details: Completing The Picture

Variations— different ways to accomplish use case actions

Exceptions— something expected that goes wrong during the execution of a use case

Policies— specific rules that must be enforced by the use case

Issues— questions about the use case

Design notes— hints to implementers

Post-conditions— what must be true about the system after a use case completes

Other requirements—constraints this use case must conform to

Priority— how important is this use case?

Frequency— how often is this performed?

Copyright 2002, Wirfs-Brock Associates, Inc. 42

Wirfs-Brock Associates Copyright 2002. All rights reserved

WritingTips and Guidelines

Copyright 2002, Wirfs-Brock Associates, Inc. 43

The Writing Process

Full Team	Small Teams or Individuals	The Products
Align on scope, level of abstraction, actors, goals, point-of-view		Actors, Candidate Summary Use Case Names
	Write summary descriptions	Narratives
Collect and clinic, brainstorm key use cases		Candidate Core Use Case Names
	Write detailed descriptions	Scenarios OR conversations
Collect and clinic, identify gaps and inconsistencies		Potential new Use Cases
	Revise and add precision	Revised Use Cases with Supplementary Details

Copyright 2002, Wirfs-Brock Associates, Inc. 44

Wirfs-Brock Associates Copyright 2002. All rights reserved

Options for Organizing Use Cases

Choose an organization for your use cases
- by actor
- arranged in a workflow
- in a hierarchy

Include a section for common use cases initiated by different actors

Copyright 2002, Wirfs-Brock Associates, Inc. 45

Writing Scenarios and Conversations

Start by writing the success story, the "happy path"

Capture the actor's intentions and actions, and system responsibilities, from beginning to end goal
- Be clear on where to start
- Describe how the goal is achieved
- End there

Define what information passes between the system and actor
- All steps should be visible to or easily surmised by the actor
- Resist the temptation to get too detailed
- Convey how the system will work—but stop short of explaining algorithms

Copyright 2002, Wirfs-Brock Associates, Inc. 46

Wirfs-Brock Associates Copyright 2002. All rights reserved

Writing a Scenario

Use a list

Record actor and system actions, identifying each

Scenario 1 Name
1. System does this first
2. Actor first does this
3. Actor next does this

Scenario 2 Name
1. System does this first
Actor:
2. First does this
3. And then does this

this is one way to break up the monotony

Copyright 2002, Wirfs-Brock Associates, Inc. 47

Writing a Conversation

Use a table

Separate actor actions from system responses

Record rounds between the actor and system

	Actor Actions	System Responses
Batch round	And I respond by ..	I do this
Interactive Round	I tell you this… …and this, too…	I am responding to what you are telling me and giving you feedback while you are talking

Copyright 2002, Wirfs-Brock Associates, Inc. 48

Wirfs-Brock Associates Copyright 2002. All rights reserved

Write General *and* Specific Cases

Choose this option when your audience needs both general and specific usage descriptions

High-level use case names state a general goal. Write one narrative for each general goal:

Narrative: Make a payment

Describe what online payment means and typical ways of making them

Write scenarios or conversations that describe more specific goals:

Scenario 1: Make a **recurring** payment

All the steps in paying my monthly phone bill …

Scenario 2: Make a **non-recurring** payment

All the steps in paying a fixed amount …

Scenario 3: Make a **regular** payment

All the steps in paying a monthly loan …

Copyright 2002, Wirfs-Brock Associates, Inc. 49

Write Two "Versions" of the Same Use Case

Choose this option when some want a quick idea, while others want to see the details

First, write a narrative

Then, choose an appropriate form. Rewrite the use case body at this lower-level of detail

Add an "overview" section to your template if you have diverse readers for your use case descriptions

Leave the narrative as an overview

Copyright 2002, Wirfs-Brock Associates, Inc. 50

Wirfs-Brock Associates Copyright 2002. All rights reserved

Include Actor Actions

Be explicit about what the actor does. Don't disguise them as "system collects" or "system captures" actions

Actor actions disguised as system activities:

Scenario: Withdraw Fixed Cash Amount (Fast Cash)

1. Present transaction screen
2. Capture fast cash withdrawal request
3. Post transaction to bank and receive confirmation
4. Dispense money, card and transaction receipt

Fixed:

Scenario: Withdraw Fixed Cash Amount (Fast Cash)

1. ATM presents transaction screen
2. Customer selects "Fast Cash" option
3. ATM posts fast cash amount withdrawal transaction to bank and receives confirmation
4. ATM dispenses money, card and transaction receipt

Copyright 2002, Wirfs-Brock Associates, Inc. 51

Include System Actions

Be explicit about what the system does

No system behavior described:

Scenario: Withdraw Fixed Cash Amount (Fast Cash)

1. Customer selects "Fast Cash" option
2. Customer takes cash, card and receipt

Fixed:

Scenario: Withdraw Fixed Cash Amount (Fast Cash)

1. ATM presents transaction screen
2. Customer selects "Fast Cash" option
3. ATM posts fast cash amount withdrawal transaction to bank and receives confirmation
4. ATM dispenses money, card and transaction receipt

Copyright 2002, Wirfs-Brock Associates, Inc. 52

Wirfs-Brock Associates Copyright 2002. All rights reserved

Describing Actions

Show actor intent, not precise movements

Intention: User enters name and address

Movements:

System asks for name

User enters name

System prompts for address

User enters address

Use simple grammar

Subject…verb…direct object…prepositional phrase

The system…deducts…the amount…from the account balance

Write actions that move the process forward

"Validate that…," don't "Check whether"

Copyright 2002, Wirfs-Brock Associates, Inc. 53

Condense Information Entry and/or Validation Actions

List of Seemingly Unrelated Items:

Enter name

Optionally, enter address

Optionally, enter telephone number

Fixed:

Enter personal information (required: name; optional: address and phone number)

Copyright 2002, Wirfs-Brock Associates, Inc. 54

Wirfs-Brock Associates Copyright 2002. All rights reserved

State System Responses At a Reasonably High Level

Includes Too Many Low Level Details and Substeps:

System opens connection to the bank
System requests authorization of bankcard number and PIN #
Bank confirms bankcard and PIN are valid
System requests active accounts for bankcard
Bank returns account list
System creates active online account entries for each account

Fixed:

System validates bankcard and PIN #s
System activates accounts associated with bankcard

Make sure what is going on, and why is it is being done is obvious to the typical reader. Know your audience

Copyright 2002, Wirfs-Brock Associates, Inc. 55

Showing Optional Actions

Make clear when a step is optional

Optionally, select an available course section

In any order, do one or more of the following:

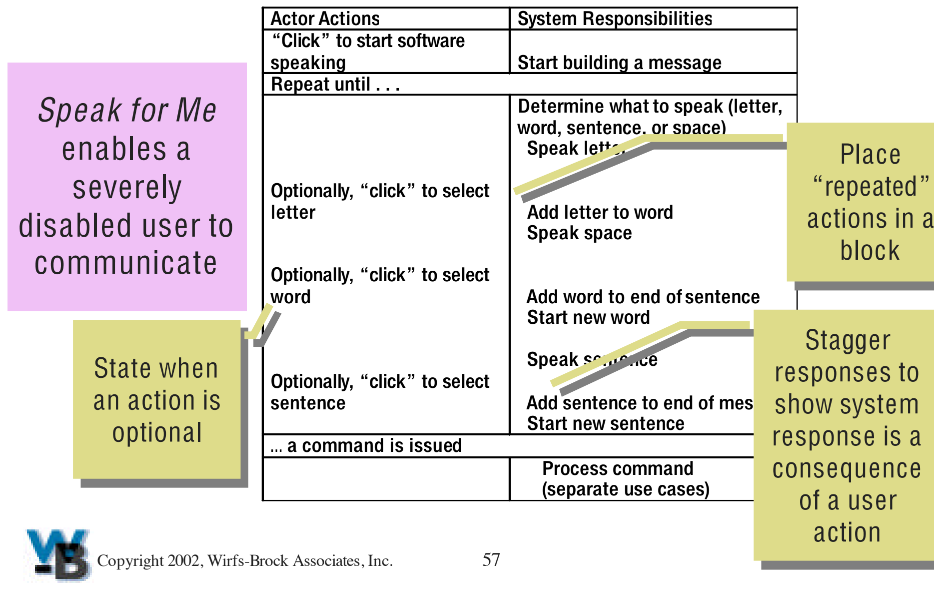
eat
drink
be merry

Indent to show a "block" of related actions

Copyright 2002, Wirfs-Brock Associates, Inc. 56

Wirfs-Brock Associates Copyright 2002. All rights reserved

Showing Repeated and Optional Actions in the "Build A Message" Conversation



Copyright 2002, Wirfs-Brock Associates, Inc. 57

Maintain a Consistent Level of Detail

Mixed level of detail:

1 | Check for required fields

2 | Capture user ID and password
Ask security component for validation
Issue SQL statements to security database for logon authorization…

3 | Open connection to bank server
Read account summaries…

Fixed:

1 | Check for required fields

2 | Login user to domain

3 | Display account summaries and bulletin

Copyright 2002, Wirfs-Brock Associates, Inc. 58

Wirfs-Brock Associates Copyright 2002. All rights reserved

Don't Embed Checks

Conversation: Registration with Automatic-Activation

	10. If bank supports automatic activation with ATM and PIN then... If ATM and PIN #s are valid then....

Fixed: Write exception details below the body

Conversation: Registration with Automatic-Activation

	10. Validate ATM and PIN #

Exception – Step 10: ATM and PIN #s are invalid— Report error to user

Copyright 2002, Wirfs-Brock Associates, Inc. 59

Leave Out Information Formats and Validation Rules

User Name: First name, last name (24 characters max, space delimited)

email address with embedded @ sign signifying break between user identification and domain name which includes domain and sub-domain names delimited by periods and ending in one of: gov, com, edu...

Fixed:

Required: user name, email address, desired login ID and password
One of: account number and challenge data, or ATM # and PIN
Optional: Company Name

Document information model details in a separate place!

Copyright 2002, Wirfs-Brock Associates, Inc. 60

Wirfs-Brock Associates Copyright 2002. All rights reserved

Don’t Mention Objects in System Actions

Objects mentioned:

Create customer and account objects

Fixed:

Record customer account information

Remember who the readers are!

Copyright 2002, Wirfs-Brock Associates, Inc. 61

Remove Clutter

Metatext— text that describes text that follows

The purpose of this use case is to describe how customers make payments.

Vague Generalities— well known principles

Each input screen shall fit entirely within the window and use as little scrolling space as possible.

Piling On— extra meaningless empty words, paragraphs, charts, sections, overbearing templates

Before piling on	**After**
Use Case	Business Use Case
Requirements	Requirements Specification Document

Copyright 2002, Wirfs-Brock Associates, Inc. 62

Wirfs-Brock Associates Copyright 2002. All rights reserved

Leave Out Presentation Details

Widget details described:

Display note in a read/write text field

From account in a drop-down list box

Amount in a currency field

Fixed:

Display payment template editable fields (note, from account, amount)

Reference screens used by a conversation

Screens: See Login Page

Copyright 2002, Wirfs-Brock Associates, Inc. 63

Keep Rules in a "Policies" Section

If policies are recorded in other requirements documentation, reference them in your use case template. If you don't have other documentation, add applicable rules to the policies section of the appropriate use case.

Use Case: Register Customer

A new user must request access and gain approval in order to perform online banking functions. Registration can be done instantly, if the bank supports automatic activation, or the user can enter a request which will be approved by a bank agent.

Policies:

Customer challenge data must be validated against customer account records before activating on-line access.

Copyright 2002, Wirfs-Brock Associates, Inc. 64

Wirfs-Brock Associates Copyright 2002. All rights reserved

Use a Table for Complex Rules

Shipping Method	Shipping Time	Total price: add both columns	
		Per Shipment	Per Item
Standard Shipping	3 to 7 business days	$3.00 per shipment **plus**	$0.95 per book
2nd Day Air ***Note:*** *Not available to P.O. Boxes, the U.S. Virgin Islands, Guam, or APO/FPO addresses.*	2 business days	$6.00 per shipment **plus** add an additional $10 for AK, HI, PR, or American Samoa	$1.95 per book
Next Day Air ***Note:*** *Not available to P.O. Boxes, the U.S. Virgin Islands, Guam, or APO/FPO addresses.*	1 business day	$8.00 per shipment **plus** add an additional $15 for AK, HI, PR, or American Samoa	$2.95 per book

Copyright 2002, Wirfs-Brock Associates, Inc.

Document Global Requirements in a Central Place

Distinguish between system-wide requirements and those than span several use cases

Global requirement: System must run 7 by 24

Specific requirement: Account information should be encrypted and transmitted over a secure connection

Reference those requirements that are satisfied by the use case below the use case body

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

Document Hints and Ideas

Design Notes

Errors and warnings about registration information contents should be collected and returned to the user in a detailed message rather than stopping at the first detectable error.

Payments should be shown in time order, with the current date first.

The user should not see payments that he should have visibility of. Prevent a user from seeing a payments from secret accounts that he should be unaware of.

Add design notes as they occur to you

Copyright 2002, Wirfs-Brock Associates, Inc. 67

Make Clear What You Don't Know

Write questions about unsolved issues

Assign responsibility to someone for resolving each issue

List them with the appropriate use case

Issues:

- Should the credit check be performed after the order is submitted or before?
- What happens if credit is denied?

If you are unclear about a detail, don't write fiction; it could become fixed

Copyright 2002, Wirfs-Brock Associates, Inc. 68

Wirfs-Brock Associates Copyright 2002. All rights reserved

Alternative Paths

Copyright 2002, Wirfs-Brock Associates, Inc.

Alternative Paths

For each significant action:

Is there another significant way to accomplish it that could be taken at this point? **(Variation)**

Is there something that could go wrong? **(Exception)**

Is there something that could go really, really wrong **(Error)**

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

Choices for Describing Variations

Add textual descriptions of variations in the variations section of the use case template (if simple), or reference an additional use case (if variation is complex or deserves special emphasis)

or

Include a description of a variation in the use case body, when you want to emphasize different choices or options

or

Draw an activity diagram that shows decision points, alternate paths, and parallel activities

Copyright 2002, Wirfs-Brock Associates, Inc. 71

Variations in *Register with Auto-Activation*

1. User enters registration information
2. System checks passwords match
3. System verifies login ID uniqueness

Variations :

1a. User enters ATM card # and PIN – see **Validate ATM card and PIN**

1b. User enters challenge data and account – see **Validate Challenge Data**

Copyright 2002, Wirfs-Brock Associates, Inc. 72

Wirfs-Brock Associates Copyright 2002. All rights reserved

Exceptions

Exceptions are deviations from the typical case that happen during the normal course of events

- They should be handled, not ignored
- How to resolve them can be open to debate
  - What if a user mistypes her password?
  - What if an order can't be fulfilled?
  - What if a connection to a web browser is dropped?

Copyright 2002, Wirfs-Brock Associates, Inc.

Errors

Errors are when things unexpectedly go wrong. They can result from malformed data, bad programs or logic errors, or broken hardware

- Little can be done easily to "fix things up and proceed"
- Recovery requires drastic measures
  - What if the disk is full?
  - What if equipment cannot be provisioned?
  - What if the OS crashes?

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

When Do We Think About Failure?

People typically think about failure only after they've describe "normal" conditions:

"[Describing exceptions] is often tricky, tiring, and surprising work. It is surprising because quite often a question about an obscure business rule will surface during this writing, or the failure handling will suddenly reveal a new actor or new goal that needs to be supported. Most projects are short on time and energy. Managing the precision level to which you work should therefore be a project priority..."

— Alistair Cockburn, *Agile Software Development*

Copyright 2002, Wirfs-Brock Associates, Inc.

Reasons To Think About Them Early, Often, Sooner *And* Later

Usability may be affected

- Consider software that enables a severely disabled user to construct messages and communicate with others. Shouting "stack overflow!" or "network unavailable!" isn't acceptable

The degree to which a user can or should be involved in exception handling has profound design implications

Solutions may not be obvious or "easy". Experimentation may be required

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

Recoverable Vs. Unrecoverable Exceptions

In use cases, exceptions may have been identified, as well as how they should be addressed

- Invalid password entered—After three incorrect attempts, inform the user that access is denied to the online banking system until she contacts a bank agent and is assigned a new password.

Recoverable exceptions can be handled deftly. The user will be able to continue his or her task

In other cases, the user won't be able to continue. These are *unrecoverable exceptions*

Copyright 2002, Wirfs-Brock Associates, Inc.

A More Realistic Story

Goals may be compromised because of exception conditions. From the user's perspective, recoverable exceptions often represent a series of compromises

- Part of a book order is out of stock. The user can choose to: split the order and back order the out of item stock items, ship what's available now and ship things when they are in stock, cancel the order, or modify the order
- This situation forces her to unexpectedly make decisions and "steer" the darn software

Sometimes the user can and should be actively engaged in "steering" the system

Sometimes this is absolutely inappropriate

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

The Mismatch Between Use Case And Program Execution

A single use case step can result in thousands of requests between collaborating objects, any number of which could cause numerous object exceptions

There isn't a direct correspondence between use case and program exceptions

Don't expect use cases to identify all or even most potential exceptional cases

Use cases exceptions—at whatever level of detail they are described—should guide, not prescribe design solutions

Copyright 2002, Wirfs-Brock Associates, Inc. 79

Avoid Use Case Writing Paralysis

Keep use case descriptions simple

Periodically update them to reflect design reality, but don't become obsessed with adding exception details

Don't tack on to use case descriptions exceptions that are design, architecture, or implementation-level concerns

Copyright 2002, Wirfs-Brock Associates, Inc. 80

Wirfs-Brock Associates Copyright 2002. All rights reserved

A Strategy For Handling Exceptions For A Key Collaboration

Brainstorm exceptions and errors cases it could address

Decide on reasonable handling and recovery strategies

Design your software to detect and react accordingly
- Create exception classes
- Assign exception handling responsibilities to objects

Explore alternatives. Test for usability and feasibility. Iterate

Copyright 2002, Wirfs-Brock Associates, Inc. 81

List What Might Go Wrong

Users enter misinformation or fail to respond

Invalid information

Unauthorized requests

Untimely requests

Dropped communications

Failures due to broken or jammed equipment

Errors in data, corrupt log files, bad or inconsistent data, missing files

Failure to accomplish some action within a prescribed time limit

Copyright 2002, Wirfs-Brock Associates, Inc. 82

Wirfs-Brock Associates Copyright 2002. All rights reserved

Choices for Describing Exceptions

Add textual descriptions of in the exception section of the use case template, which may reference an additional use case

or

Describe exceptions for a number of use cases (and their effects) in a separate section of your use case model

Copyright 2002, Wirfs-Brock Associates, Inc. 83

Describing Exceptions Makes Requirements More Complete

Possibilities in *Place An Order*

Ideal situation

- Good credit, items in stock → accept order

Recoverable exceptions:

- Bad credit and preferred customer → accept order
- Low stock, and OK to reduce quantity → accept reduced quantity order

Unrecoverable exceptions:

- Bad credit and not a preferred customer → decline order
- Out of stock → decline order

Copyright 2002, Wirfs-Brock Associates, Inc. 84

Wirfs-Brock Associates Copyright 2002. All rights reserved

Exceptions Added to *Place An Order*

Scenario: Place An Order

1. Identify the customer
2. Identify each order item and quantity
3. System accepts and queues the order

Exceptions:

1a. Bad credit and Preferred Customer—Accept order
1b. Bad credit and not Preferred Customer—Decline order
2a. Low on stock—Ask customer to accept reduced quantity

Copyright 2002, Wirfs-Brock Associates, Inc. 85

When to Create a New Use Case to Describe An Alternative

Write another...

- when an alternative appears complex
- when an alternative is important and you want to emphasize it

Document simpler alternatives in the supplementary part

Document more complex ones as separate use cases

Rewrite and reorganize for clarity!

Give new use cases specific names that identify specific conditions

Copyright 2002, Wirfs-Brock Associates, Inc. 86

Wirfs-Brock Associates Copyright 2002. All rights reserved

Documenting Exceptions For a Specific Use Case

Name the exception below the use case body

Tell what step it relates to

Document what happens:

- Briefly describe what happens, or
- Refer to another use case describing the exception handling

Optionally tag an exception as being recoverable or not (you decide whether this adds value to your descriptions)

Copyright 2002, Wirfs-Brock Associates, Inc. 87

Develop and Document General Policies

Describe and explain general policies that affect the use case model. Collect them in a central place.

- The online banking application is designed to cover communications failures encountered during a financial transaction. A full set of single-point failures was considered. Some double-point failures were explicitly not considered
- The general strategy is to ensure that transaction status is accurately reflected to the user. Failures in validating information will cause the transaction to fail, whereas intermittent communications to the external database or to the backend banking system during the transaction will not cause a transaction to fail

Copyright 2002, Wirfs-Brock Associates, Inc. 88

Wirfs-Brock Associates Copyright 2002. All rights reserved

Explain And Document Policies

Exception or Error	Recovery Action	Affect on User
Connection is dropped between UI and Domain Server after transaction request is issued.	Transaction continues to completion. Instead of notifying user of status, transaction is just logged. User will be notified of recent (unviewed) transaction results on next login.	User session is terminated… user could've caused this by closing his or her browser, or the system could have failed. User will be notified of transaction status the next time they access the system.
Connection dropped between domain server and backend bank access layer after request is issued.	Attempt to re-establish connection. If this fails, transaction results are logged as "pending" and the user is informed that the system is momentarily unavailable.	User will be logged off with a notice that system is temporarily unavailable and will learn of transaction status on next login.

Use descriptions approachable to users, developers and other stakeholders

Copyright 2002, Wirfs-Brock Associates, Inc. 89

Specify Pre- and Post-Conditions Only When You Need to Be Formal

Pre-conditions should make clear when a use case can execute
- An account must be in good standing and the daily withdrawal limit not exceeded in order to withdraw cash

Post-conditions may be relevant to other systems
- Being overdrawn may trigger transaction fees

Pre-conditions may be set by other systems
- An account can be overdrawn through direct payments

Once you add pre- and post-conditions to one use case, you will need to add them to dependent ones!

Copyright 2002, Wirfs-Brock Associates, Inc. 90

Wirfs-Brock Associates Copyright 2002. All rights reserved

Document Pre and Post-Conditions That Affect Future System Behavior

Define post-conditions in terms of things that will impact future system behavior

Post-condition that restates the user's goal:

- Post-condition: Customer has withdrawn cash
- *So what? The customer receives cash but what does this say about the next time he/she wants to withdraw case, or any other use case?*

Fixed:

- Post-condition 1: Account balance is positive
- Post-condition 2: Account is overdrawn
- *If the account is overdrawn, the user may not be able to withdraw more case… or an overcharge fee may be assessed*

Copyright 2002, Wirfs-Brock Associates, Inc. 91

Add Precision To Post-Conditions

Instead of documenting just the "happy path" post-condition, define post-conditions for each path through the use case and resulting system state

- At least one post-condition for each successful goal
  - Customer receives cash → Account is overdrawn or Account balance is positive
- One for each exception
  - Account daily limit would be exceeded - Customer withdraws lesser amount → Account is in good standing and Account daily withdrawal limit reached
  - Amount would exceed overdraw limit - We refuse to disburse cash → account is in good standing
- One or more for each variation
  - Fast cash → Account is overdrawn or Account balance is positive

Copyright 2002, Wirfs-Brock Associates, Inc. 92

Wirfs-Brock Associates Copyright 2002. All rights reserved

Emphasis

Copyright 2002, Wirfs-Brock Associates, Inc.

Emphasize What's Important Within a Use Case

Things gain prominence by their position and appearance. To increase an item's emphasis:

Put it first

Highlight it

Surround it by white space

- Put it in a bulleted list

Mention it in multiple places

Give it more room

Repeat or restate it in different forms

Say it another way

Mention it in multiple places

Copyright 2002, Wirfs-Brock Associates, Inc.

Wirfs-Brock Associates Copyright 2002. All rights reserved

What’s Emphasized?

Template 1	Template 2
Use Case: Make a Payment Author: Rebecca Last Revision Date: 10/04/02 Change History:… Version: 0.4 Status: Preliminary Review Level: Core	Use Case: **Make a Payment** Actor: Bank Customer Pre-condition: User has an active account and is authorized to transfer funds

Don’t choose a template that inadvertently emphasizes the wrong things

Copyright 2002, Wirfs-Brock Associates, Inc. 95

What’s Emphasized?

Choose course by optionally, in any sequence:

- **Browse Course Catalog**
- **Choose Next Course from Degree Plan**
- Enter course section

One way to make a supporting use case stand out is to use **BOLD**

Copyright 2002, Wirfs-Brock Associates, Inc. 96

Wirfs-Brock Associates Copyright 2002. All rights reserved

Emphasize What's Important Within a Use Case Model

Place first those use cases you wish to emphasize

Choose the form of use case descriptions according to what you want to emphasize:
- A conversation emphasizes system/actor dialog
- A narrative emphasizes the high points of a story, not the details

Repeat and restate to make points stand out

Choose a template that doesn't inadvertently emphasize the wrong things

Copyright 2002, Wirfs-Brock Associates, Inc. 97

Wirfs-Brock Associates Copyright 2002. All rights reserved

THE ART OF WRITING USE CASES TUTORIAL NOTES

I. Description and Objectives

This is an introduction to use cases, a technique for structuring system usage descriptions, and the principles of a user-oriented development process. You will be able to apply the principles and techniques to your projects, writing appropriate usage descriptions.

The topics include:
III. The context for use cases
IV. Use case modeling constructs
V. System glossary
VI. A Use Case Template
VII. Narratives
VIII. Scenarios and Conversations
IX. Other Descriptions, Exceptions, and Variations
X. A Use Case Model Checklist
XI. The Writing Process
XII. More Tips and Techniques

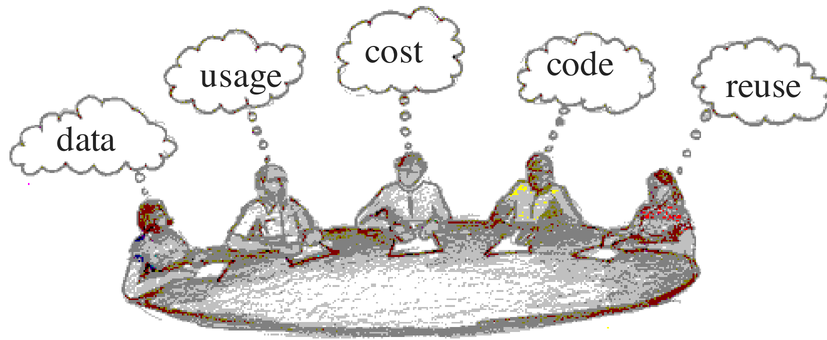
II. Further Resources

There are several good books about use cases. We recommend these three:
Writing Effective Use Case , Alistair Cockburn, Addison-Wesley, 2001, ISBN 0-201-70225-8
Use Cases Requirements in Context, Daryl Kulak and Eamonn Guiney, Addison-Wesley, 2000, ISBN 0-201-657678-8
Software for Use A Practical Guide to the Models and Methods of Usage-Centered Design, Larry Constantine and Lucy Lockwood, ACM Press,1999,ISBN 0-201-92478

Andy Pol’ s website, The Use Case Zone has many pointers to online articles, templates and use case discussions: http://www.pols.co.uk/usecasezone/

III. The Context for Use Cases: Team Development

Development is never done in a vacuum; there is always a context. Many of the stakeholders in our development efforts do not speak in our native object-oriented tongue. In our role of analyst we face two challenges: correctly interpreting stakeholders' knowledge of the problem, their concerns and requirements in our models, and presenting our design work in terms they can understand.

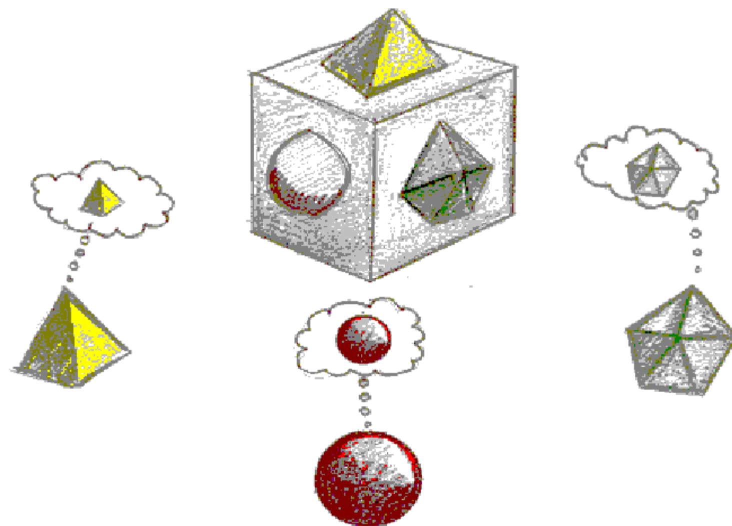


A good system never dies, it is adapted and improved upon.

A system takes form through a series of textual and graphical descriptions. At each time-slice of the project, the description should be less ambiguous, but each form should be describing the same thing. Each description, when viewed by a practitioner with experience in the corresponding natural, graphical, or programming language, can be evaluated according to a number of well-known criteria. Typically, the system came to be through a structured process known as design, often preceded by a form of requirements gathering and specification called analysis. During analysis, one of your tasks is to describe our system' s usage with use cases.

Each participant in the life of a software system has a unique set of criteria for evaluating its quality during its development. The target values that are used during such evaluation varies according to their point-of-view. To begin simply, let's imagine sets of criteria that are important from three points-of-view:

- user
- analyst/designer
- programmer



To bring together all of these perspectives, you need a systematic way to consider the problem. Once you can agree on the nature and requirements of the problem, you can make informed decisions about and document which parts of the problem you intend to automate with the computer. Finally, you will have solid ground for determining whether or not the program that you build for our machine has, in fact, accomplished your goals.

Copyright 2002 Wirfs-Brock Associates

The User

The user is particularly concerned that the system be easy-to-use. Of course, this requires that the application controls and processing be transparent, consistent, correct and natural to the user. The system must also do the job, i.e., it must be complete, and it should be configurable to an individual user's specific needs.

The Analyst/Designer

From the analyst/designer' s point-of-view, the requirements, the specification and design must be simple and easy to understand. It must be modular and traceable to the requirements. Due to an ever-evolving specification, it must be flexible and extensible. Specified portions must be reusable. Further, the system under development is constrained by business and user requirements. The functional characteristics of the design should be concise without losing the details of its execution behavior.

The Programmer

Programmers have all the issues of the designer. But when entering the implementation domain, they must be certain that the application is possible. Beyond that, they must live with the constraints imposed by the hardware that application performs on.

Building Consensus

System development has three areas of activity: understanding and documenting the problem and its requirements, specifying how the various users will be able to use the system to satisfy the requirements and how the system will fulfill all of the remaining non-usage requirements, and implementing the specification as software executing on appropriate hardware.

IV. Use Case Modeling Concepts

A specification is a statement of what the system is to do in the context of your problem. It describes how the requirements that you have elicited by asking the right probing questions will be fulfilled. Requirements that can be satisfied by interactions between a user and the program can be described by use cases.Use cases present a model of how your system is used and viewed by its users. This use case model is just one view developers need to understand as they proceed with design and implementation. It is also a view that other stakeholders in the specification of a product can readily understand and comment on. A usage model, expressed as use cases forms the basis for a behavioral description of a system.

Let's introduce the core concepts of use case models:

Use Case

A *use case* is a description of system functionality from a particular point-of-view. Many use cases describe task-related activities. For example, in the Online Bank application, which we draw upon to illustrate concepts in this course, we wrote use cases to describe these activities, among others

- making a payment
- transferring funds between accounts
- reviewing account balances

Each use case describes a discrete "chunk" of the online banking system. These use cases were described from the users' viewpoint.

Use cases don' t dive into implementation details, but describe how something behaves from an external perspective. A use case may include more or less detail, depending on its intended audience and what level it of the system it describes.

Three Use Case Forms

We recommend you consider three forms of use case descriptions. Each different form has its strengths and weaknesses. Depending on what you need to describe, and at what level of detail, you should pick the appropriate form to

write a use case description. You might choose to first write high-level overviews, then add detail and describe the sequences of actions and interactions between the user and the program. The form you choose depends on what you are trying to express.

You may write one or more forms for each use case, depending on the needs of your audience. Write narratives to present a high-level overview. Then, if appropriate, write one or more scenarios or conversations that elaborate this high-level description.

The Writing Task	The Best Form To Use
Present Overview	**Narrative**
Describe simple sequence of events	**Scenario**
Emphasis actor-system interaction	**Conversation or Essential Use Case**

Here are examples of each of the three forms.

First, a use case narrative taken from an on-line banking project:

Make a Payment
Narrative

The user can make online payments to vendors and companies known to the bank. Users can apply payments to specific vendor accounts they have. There are two typical ways to make payments: the user can specify a one-time payment for a specific amount, or establish regular payments to made on a specific interval such as monthly, bi-weekly, or semi-annually.

It offers a high-level view of how the requirements of "Make a Payment" are satisfied.

A use case narrative has a very simple format. It begins with the name of the use case, and is followed by a brief, textual description that explains at a high level how an actor interacts with our system in order to accomplish a task or goal. Here is another narrative:

Register Customer
Narrative

To use the system, a customer must be registered. There are two ways to register. If the bank supports "automatic activation", all the customer must do is supply identification information. After the system verifies the customer has an account and the information is corrector, the customer may use the system. If the bank does not support automatica activation, the customer submits a request to be activated, along with identification information. After a bank employee has check the information and activated the customer, the customer may use the sys This may take a few days.

There are two scenarios outlined in the narrative: one for automatic activation, another with manual activation. We write a sequence of actions to describe each. Here is an example of the scenario for registering with automatic activation.

Register Customer with Automatic Activation Scenario

1 User enters registration information:

Required information: user name, email address, desired login ID and password, and confirmation password

One of: account number and challenge data, or ATM # and PIN

Optional: language choice and company

2 System checks that password matches confirmation password.

3 System validates required fields and verifies uniqueness of login ID

4 System verifies customer activation information.

5 System creates and activates customer online account.

6 System displays registration notification.

Notice that, along with the sequence of actions, we include some notion of the types of information that are used.

Finally, the more detailed *conversation* form allows us to clearly show the system's responses to the actions of the user. Here we have many opportunities to demonstrate decision-making, iteration, and dependency among the parts of the problem.

Actor: User	System: Application
	Present list of payment templates to user organized by payee category
Select a payment template	
	Present details of selected Payment Template and Recent payment history to Payee
Enter payee notes, amount and account Submit Payment Information	
	Apply payment to payee Add new payment to recent payment list Redisplay the payment list
Optionally, request Setup Payments	
	Goto **Edit Payment Template Information**
Select next function	
	Goto **selected use case**

Each form has its strengths and weaknesses. Conversations show more detail, scenarios show step-by-step sequences, narratives are free-form text. The form you choose depends on what you want to convey to your reader, and how much detail you want to show.

Copyright 2002 Wirfs-Brock Associates

Form	Strengths	Weaknesses
Narrative	• Good for high-level summaries • Can be written to be implementation independent	• Easy to write at too high or too low a level • Not suitable for complex descriptions • Can be ambiguous about who does what
Scenario	• Good for step-by-step sequences	• Hard to show parallelism or arbitrary ordering • Can be monotonous
Conversation	• Good for seeing actor-system interactions • Can show parallel and optional actions	• Easy to write at too detailed level: pseudo pseudo-code • Only two columns: What about multiple actors?
All Forms	• Informal	• Informal

Abstraction, Scope, and Detail

Use cases can be written very concretely, or they can generalize specific actions to cover broader situations. For example, we could write use cases that describe:

Steve registers for English 101, or
Student registers for Course, or
User uses System, or
Student registers for Variable Credit Course, or
Student Registers for Music Course

In order to choose the right level of abstraction to write a use case, you need to understand how the behaviors of both the actor and the system might be expressed to cover the widest range of situations without losing any important details. Clearly, "User uses System" is too high-level, and "Steve registers for English 101" is too concrete. However, it may be important to write use case descriptions for "Student registers for Course" and, if the system's or user's actions are sufficiently different, to also describe "Student registers for Variable Credit Course." In fact, if registering for a music course means signing up for practice sessions in practice rooms in addition to classroom instructions, it too may need additional description. You can also express variations within a single use case description.

Use cases vary in scope and detail. You can use them to describe all or part of our "system". Which system boundary do we mean: At a particular component (describing the web applet)? across the application (on-line banking)? or across multiple applications within the organization (the bank)?

We typically start by describing application level scope. The amount of detail that we choose to put into use cases varies. We could describe general actions: *Enter deposit amount*. Or specific detail: *Press number keys followed by enter key*

Write at the level that seems appropriate to your readers. This typically means describing actor actions and system responses that match the goal for the use case. So, to follow that guideline, if the use case were named "Make Deposit," we'd describe the user general action of "enter deposit amount," not his or her gestures: "Press number keys followed by enter key."

Recipe: Finding the Use Cases

1. Describe the functions that the users will want from the system.
2. Describe the operations that create, read, update, or delete information that the system requires. Describe these operations.
3. Describe how actors are notified of changes to the internal state of the system.

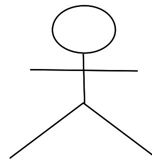
Identify actors that inform the system about events that the system must know about. Describe how the users will communicate the information about these events.

Actors

An *actor* is some one or some thing that interacts with our system. We divide actors in to two groups:

- those that stimulate the system to react (primary actors), and
- those that respond to the system' s requests (secondary actor)

We model actors so we can understand what behaviors they' ll need from our system, if they are primary actors. We model secondary actors so we understand how our system uses external resources. In the Unified Modeling Language, the stick figure icon is how we show an actor on a use case diagram. This is the standard notation for an actor, although you may choose another icon that is more meaningful.



Actors are the external agents that use (or are used by) our system. Those that initiate activity are worth considering as a group. These *primary* actors stimulate the system to take action and form the basis of most of our usage descriptions. The other, *secondary* actors, interact with the system only because the system poses questions to them or issues a command. They are usually external programs or devices, although sometimes the system will direct a human to perform a task.

Most often, systems engage with an actor called the user. In fact, we often unconsciously equate an actor with this user. But such a narrow vision will often make us overlook significant areas of the system' s requirements. For example, many systems require support for administrators and technicians that periodically maintain and configure the system. These activities are quite different from the user's tasks. Systematic consideration of the various actors that are involved with our system will ensure a more complete understanding of what it must do.

Guidelines for Finding and Naming Actors

GUIDELINE: Focus on primary actors.

In the on-line banking system, we have a number of human actors. The one we initially focused on was the customer-user who accesses financial services including bill payment, account balance and statement inquiries, and funds transfer. An agent of the bank (or bank agent) can perform several tasks: customer maintenance, console operation, and bank administrative functions which include bank agent maintenance, and system configuration.

GUIDELINE: Group individuals according to their common use of the system. Identify the roles they take on when they use or are used by the system

Each role is a potential actor

Name each role and define its distinguishing characteristics. Add these definitions to your glossary

Copyright 2002 Wirfs-Brock Associates

GUIDELINE: Focus initially on human actors. Ask:

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- Who gets information from the system?
- Who provides information to the system?

GUIDELINE: Name human actors by their role.

Specific people may play several roles; several actors may represent them. We could divide our bank agent actor into several, more distinct roles: console operator, bank agent administrator, customer administrator, and system configuration manager. These finer distinctions, while easily made, didn't really help us gain any new insights about system requirements for bank employee usage. While important, bank agent usage wasn't a high priority. The customer-user facilities were of primary interest to the project manager and sponsors. So we backed off and did not enumerate these kinds of bank agent actors.

GUIDELINE: Don't equate a job title with an actor name.

This wasn't a problem on the online banking application. Since we didn't directly interact with bank employees we didn't know their job titles. We were arm's length from end users, so it was easy for us to create a single bank agent category. However, we have seen several projects where jobs and titles get in the way of understanding of how users need to use the system. Supervisory job titles don't always equate with more features; usage often cuts across job function.

GUIDELINE: Don't waste time debating actor names.

Actor names should be nouns or noun phrases. Don't be too low level when naming actors. Don't be too abstract in describing or naming an actor. We didn't have the benefit (or bias) of knowing the name of any existing legacy applications at banks. The physical name of the transaction service, e.g. CICS, seemed too physical and not very descriptive; our next line of thought was that this actor represented our connection to existing legacy applications. So, we settled on calling this external actor a legacy connection and left it at that!

GUIDELINE: Be consistent in showing actors. Your choices are:

- Show *all actors that interact with the system*, even remote systems,
- Show only those *initiators of the contact*,
- Show only those actors that *need the use case*,
- Show only *human* actors, not the system

We recommend you use the first strategy, and distinguish actors that initiate contact as *primary actors*, and actors that the system touches as *secondary actors*.

GUIDELINE: An actor name for an existing system should refer to its common name.

GUIDELINE: Names of non-human actors are more recognizable if they simply remain the name of the system.

Don't invent clever, more abstract names if it causes confusion. In the online banking application it was fairly easy to find our non-human actors. We recorded information about On-line Banking System customers and their transactions in an Oracle Database, and accessed legacy systems (either CICS, IMS/DC) to perform financial transactions and pull current account information. This led us to two external actors: legacy connection and database. These actors mainly were of interest to the development team who needed to model objects that represented interfaces to these external actors; the project sponsor only cared about the kind of legacy connections that would be supported, and that Oracle was the database we had selected.

Copyright 2002 Wirfs-Brock Associates

GUIDELINE: If you are building a system whose behaviors are based on privileges and rights of individuals rather than on their roles, record these variations in a manner that lets you track their impact on the design - don't try to solve it with actors alone.

Sometimes we need to know more about individual users than their actor roles. You may need to describe individuals' rights and capabilities, and note what privileges are required to exercise certain system functionality. Simply defining actors doesn't buy us enough information. This issue came up in our system design when we started considering version two On-line Banking System features. In release one, a customer-user could register and use all banking functions; in version two, a major requirement was that multiple users could be associated with a single customer. Each user might use a different set of the customer's accounts. A user could grant account visibility if he/she had appropriate privileges (the ability to do account and user maintenance). Initially, we debated splitting customer-user into primary- user and customer-user, but talked ourselves out of creating a new actor to solve our conceptual problem. It wasn't clear that 'primary user' was the right distinction. One clue was that our domain expert didn't like this idea at all. He felt that since all customer-users had the potential to do account and user maintenance, they shouldn't be arbitrarily divided into different actors. We also realized that our second attempt at factoring bank agent into roles hid the requirement that our system needed to let banks configure the capabilities of individual bank agents. Those that were trained in customer administration weren't likely to also perform console operator functions, but it was up to the bank to decide who could do what; it was up to our system to enforce and grant these capabilities.

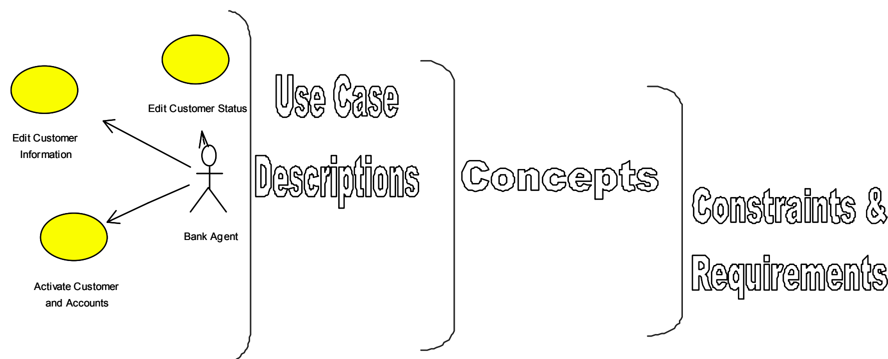
These activities are quite different from the user's tasks. Systematic consideration of the various actors that are involved with our software will ensure a more complete understanding of what the software must do.

Recipe: Finding Actors

1. Focus initially on human and other primary actors.
2. Group individuals according to their common tasks and use of the system.
3. Name and define their common role.
4. Identify systems that initiate interaction with the system.
5. Identify other systems used to accomplish the system's tasks (these are secondary actors).
6. Use common names for these other "system" actors.

Use Case Models

A single use case describes a discrete *chunk* of the system's functionality. A *use case model* is a collection of related descriptions of our system's behavior. These descriptions are backed up by clearly understood concepts, and should satisfy system requirements.Use case descriptions are typically written from an external perspective; that of a user performing task-related activities. These descriptions form the basis of our view of how the various *actors* in our problem will interact with the program and flesh out one of our perspectives of the specification.



While you initially focus on use cases initiated by human actors, there are a number of other system initiated use cases that can be documented, such as:

- initializing the system on startup
- broadcasting change information to other active components
- backing up the database

Use Cases Can Be Related

Use case diagrams can show a big picture of the application by demonstrating what actors participate in what use cases, and by showing the relationships among the various use cases. Relations like "uses", "depends", and "extends" are added when this additional level of detail provides useful information.

GUIDELINE: Don't show everything!

GUIDELINE: You can have more than one system view. Don't try to put all of your useful information into one diagram.

The Unified Modeling Language defines these relationships between use cases:

Dependency—The behavior of one use case is affected by another

Being logged into the system is a pre-condition to performing online transactions. Make a Payment depends on Log In

Includes—One use case incorporates the behavior of another at a specific point

Make a Payment includes Validate Funds Availability

Extends—One use case extends the behavior of another at a specified point

Make a Recurring Payment and Make a Fixed Payment both extend the Make a Payment use case

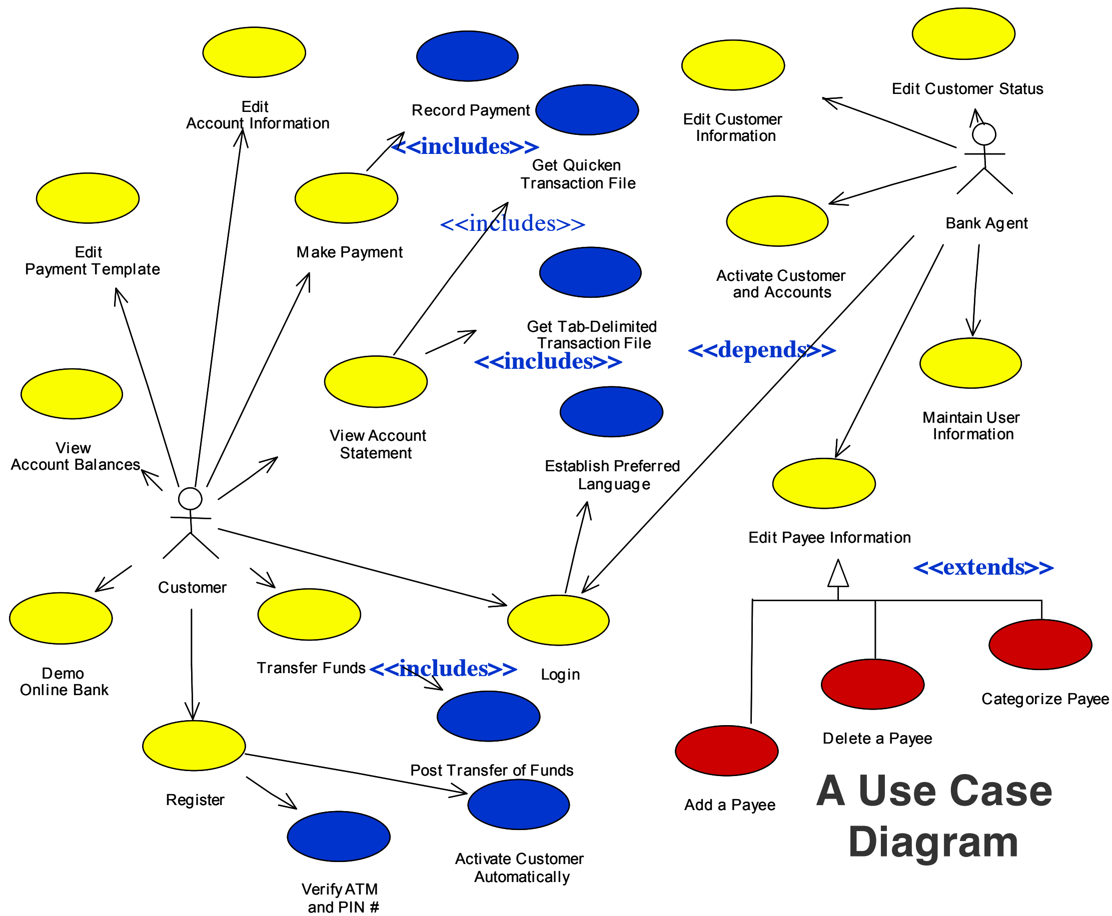
Generalize—One use case inherits the behavior of another; it can be used interchangeably with its "parent" use case

Check Password and Retinal Scan generalize Validate User

Use Case Diagrams

A use case diagram shows a high-level picture of the users and the use cases they participate in. In a complex system, several use case diagrams can be drawn to show different views of how the system is used. The Unified Modeling Language includes a graphical notation for representing use cases as ellipses and actors as stick figures. The lines drawn between actors and use cases indicate that the actor is initiating the use case. Use cases can call upon other use cases, indicated by the <<includes>> relationship, or vary the behavior of a use case, indicated by the <<extends>> relationship. The dependency relationship is shown by a dashed line. Generalization (not shown in the diagram

below) is shown by an open arrow pointing to the use case being generalized. This is the same as the inheritance relationship between classes.



Guidelines for Drawing Use Case Diagrams

GUIDELINE: Identify the "shape" of your use case model, then draw one or more use case diagrams that present meaningful snapshots of your system's behavior.

GUIDELINE: Don't include every use case in a single Use Case Diagram.

You can draw more than one use case diagram. A use case can be shown on more than one diagram, too. The purpose of a use case diagram is to convey a particular organization of use cases.

Some possible diagrams: A diagram showing core use cases and their initiating actors; a diagram that emphasizes the interactions and dependencies between two actors; a high-level diagram that identifies summary use cases; a detailed diagram that shows how certain core use cases are fulfilled by "including" supporting use cases; a diagram that identifies key variations with use cases that "extend" other use cases

Use Case Levels

Use cases can be written at various levels of abstraction. They can describe sweeping overviews of system functionality. These are termed "summary" use cases. Use cases can describe task related activities of users as they interact with the system. These are "core" or task level use cases. You can describe how your software behaves in support of core use cases. We term these "supporting" use cases. You can dig even deeper and describe how components in our

software behave and interact. These "internal "use cases are of value to those designing how the responsibilities of the system are distributed between components.

The most useful level to consider from the external actor's understanding is the *core level*. This will be the focus of our writing in class. However, sometimes other stakeholders need to see the big picture and will need to read summary use case description. Developers need the extra precision found in supporting and internal use cases. Core level use cases are linked to lower level *supporting use cases*, and are part of higher level strategies.

Use Case Model Shapes

Depending on the nature of the system you are trying to describe, your use case model may assume one of several shapes. Alistair Cockburn, in *Writing Effective Use Cases,* identifies the sailboat shape. It is a use case model that includes a well structured set of core use cases that are defined to meet strategic behaviors outlined in a few summary level use cases. In this sailboat image, most of the use cases are core- those found at the waterline where the sailboat sits in the water. At the core level, you identify specific tasks of various actors using the system. Below this waterline are supporting use cases that are used to fulfill one or more core use case functions.

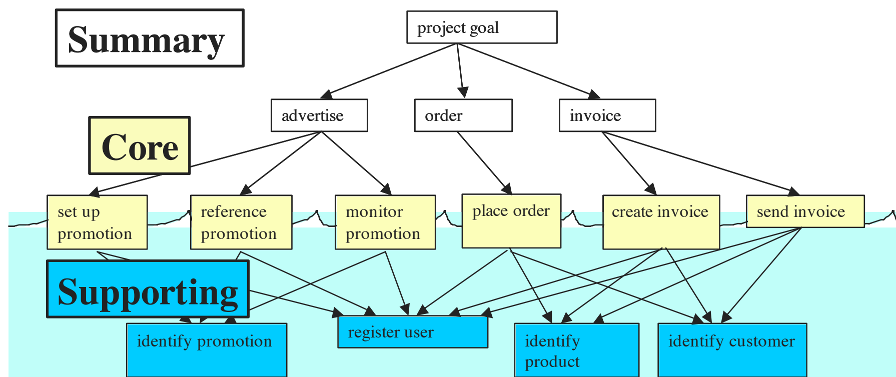


FIGURE 1. A sailboat shaped Use Case Model. A balanced number of core or task-level use cases.

A second characteristic use case model shape is the "hourglass". This use case model is characterized by a small (could even be one) number of core or task-level use cases that call on a wide-range of supporting use cases. The core use cases could support several strategic goals. In this use case model shape, variations and complexities are typically hidden to the software user performing a core use case.

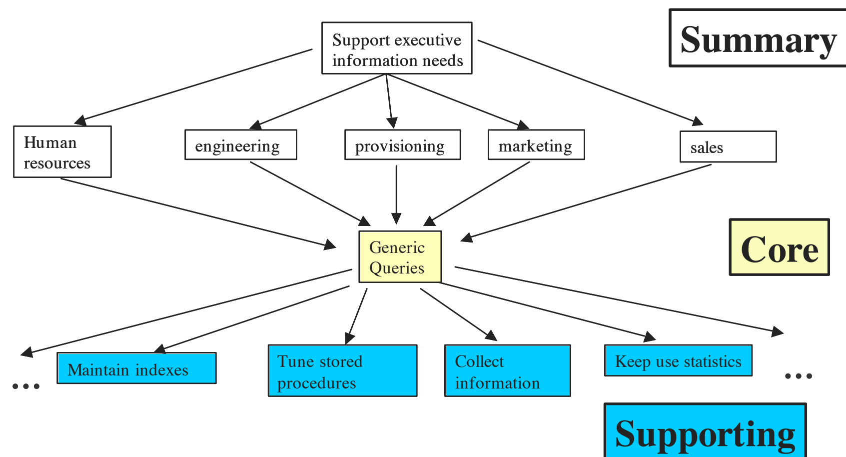


FIGURE 2. An hourglass shaped Use Case Model. Much of the complexity of the software is not evident to the user.

A third shape is a “pyramid”. In this Use Case model, there are many supporting use cases, each defining functionality that can be called on by a few core use cases. This is typical of a software application development environment or an operating system. Sometimes, there may be little or no distinction between core and supporting use cases: all may be exposed and usable by the same actors.

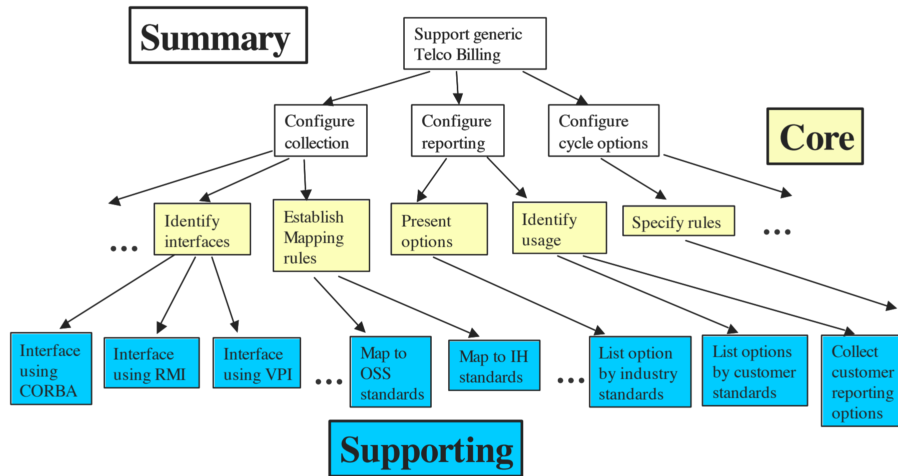


FIGURE 3. A pyramid-shaped Use Case Model. Core use cases resting on numerous supporting use cases.

Example: Defining Usage Requirements

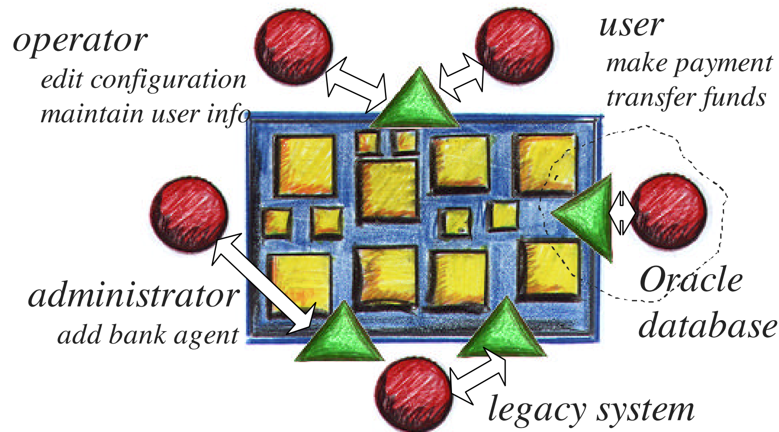
The On-line Banking System requirements consists of support for all of the tasks that the users need to perform with the system. They initiate the activities of the system and their agendas are reflected in the use cases:

- Login
- Register Customer
- View Account Balances
- View Account Statement
- Transfer Funds
- View Session Activities
- Select Setup Choice
- Edit User Information
- Edit Account Information
- Delete Account
- Edit Payment Template Information
- Make Payment

When you define what your system does for its users, you are also determining the boundaries of our system: what’s inside?, what’s out of scope?, what does your system do for its users?, how do they interact with it?, and how does it interact with other systems?

In the on-line bank, although the end users using the web were of primary concern, there were other people and systems that interacted with the software. These actors also initiated and participated in a number of use cases. External systems, and how they were used were important, too.The interactions and usage of legacy software were important to specify so that it could be isolated and viewed in a uniform way by other parts of the system. The use of the database was of concern to developers and the sponsors. Although the database was a secondary actor; the details of what was stored in the database, and the requirements for storing transaction details (not internal transactions) made it important to describe it in a manner that was understood by both parties.

Drawing the System Boundary Actors and Use Cases



Use cases are only part of any system specification. Use cases are often accompanied by supporting information, pictures, more formal descriptions of algorithms, etc., that are meaningful to people who will build or use the system.

The sources of funding of the on-line bank were a consortium of South American banks, and a major computer manufacturer. They specified schedule, cost, deliverables, variability from one bank to another, support for legacy connectivity, user languages, development tools and languages, hardware platforms, and distribution requirements.

The user requirements came from representatives of the banks: the tasks to be performed on-line, the user interface, and the roles of the people that will use the application.The technical architect imposed a set of non-functional requirements on the system: reusability, performance characteristics, robustness, configurability, support for technology standards, error-handling, and fault tolerance. The patterns of usage were not nearly as difficult as the "internal", structural and behavioral requirements imposed by the system architect and the sources of funding.

It is for this reason that use cases are only part of any system specification. They are accompanied by supporting information, pictures, more formal descriptions of algorithms, hardware componentry, etc., that are meaningful to people who will build or use the system.

V. Glossaries

The purpose of a glossary is to clarify terms so that team members can know what they are agreeing or disagreeing on. A common set of terms that are defined and understood forms the basis for all our descriptions. A glossary should be developed to accompany a use case model as well as other requirements documentation.

A glossary is a central place for:

- Definitions for key concepts
- Clarification of ambiguous terms and concepts
- Explanations of jargon
- Definitions of business events
- Descriptions of software actions

The glossary is built incrementally. Terms in the glossary form a working description of the concepts and events that exist in the various domains of the problem, and clarify the terms that we use to describe requirements and write use cases. A good glossary entry follows this form:

"Name of a concept" related to a "broader concept" + any distinguishing characteristics

For example:

A **compiler** is a **program** that translates source code into machine language.

Here is an example from the on-line bank contrasting an original version with an improved version, reworked for clarity:

Improving Glossary Definitions

Contrast the original:

Account In the online banking system there are accounts within the bank which customer-users can access in order to transfer funds, view account balances and transaction historical data, or make payments. A customer has one or more accounts which, once approved by the bank can be accessed. The application supports the ability for customers to inform the system of new accounts, and for the customer to edit information maintained about the accounts (such as name and address information).

With a definition that says what an account is and briefly describes how it is used:

Account An account is ***a record of money*** **deposited at the bank for checking, savings or other uses.** A customer may have several bank accounts. Once a customer's account is activated for online access, account information can be reviewed and transactions can be performed via the internet.

Experience has shown the value of developing a common set of terms for the development team. Seasoned developers, because of their wide experience in the domain, will have encountered multiple, varying definitions for many of the core concepts. A concept glossary levels the playing field and unifies these diverse points-of-view. For team members new to the domain, a concept glossary offers a jumpstart to understanding the domain, and is vital to understanding the requirements.

GUIDELINE: Write definitions for key concepts.

GUIDELINE: Build incrementally when writing requirements.

GUIDELINE: Add supplementary information.

Why is this concept important? What are typical sizes or values? Clarify likely misunderstandings. Explain graphical symbols

GUIDELINE: Determine an appropriate name for each concept.

GUIDELINE: Normalize names.

Identify behaviors that are the same but have different names. Identify behaviors that are different but have the same name.

GUIDELINE: Define acronyms and their concepts.

Example: OSS - Operations Support System: As defined by the FCC, a computer system and/or database used at a telephone company for pre-ordering, ordering, provisioning, maintenance and repair, or billing

GUIDELINE: Use pictures to relate concepts.

Example: We recommend defining terms and relating them with a picture as the best way to get across complex relationships. Here are some related concepts:

wire center- the geographical area served by a central office

central office- a building where local call switching takes place

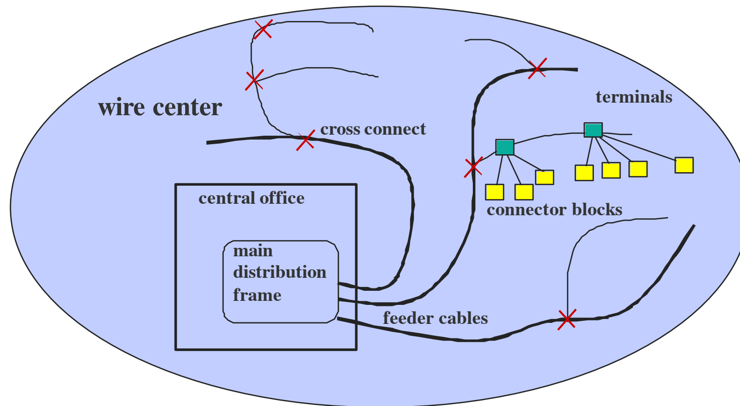
main distribution frame- a large connector at a central office, which connects switching equipment to feeder cables

Copyright 2002 Wirfs-Brock Associates

feeder cable- a large cable that connects to the main distribution frame at a central office and feeds into distribution cables

distribution cable- a cable that connects between a feeder cable and one or more terminals

and a picture showing how they are related:



GUIDELINE: Avoid vague words.

GUIDELINE: Avoid Using *is when* or *is where*.

Good: An overplot is an overlap between two or more graphic entities drawn at the same place on a page

Bad: An overplot is when two things overlap

GUIDELINE: Define a particular status as a list of possible states.

Example: A proposal's approval status is its current stage in the process for granting or denying it: *awaiting department approval, awaiting chair approval, awaiting board approval, or denied*.

GUIDELINE: Use team development and review to build consensus for definitions.

VI. A Template For Writing Use Case Descriptions

Here is a template for filling in additional information that can accompany the description of the interaction between the actor and the system. Several authors have proposed their versions of a Use Case Template. They are similar but have slight differences. This discussion presents an overview of elements that can be part of a use case template.

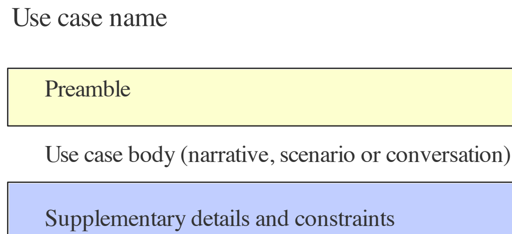


FIGURE 4. The parts of a Use Case Template

We recommend you start by adopting a template that is fairly lightweight (we include more information in this template than you may need to get started). Depending on where you are in a project, you may start by only filling in part of the information in a template...and then add more details in a second iteration. For example, you may start by only writing a narrative and identifying the actor for the use case. Later, you may describe exceptions and add conversations or scenarios that expand on the basic narrative.

It is useful to divide the template into three parts:

- the preamble- which defines the context of the use case
- the body - which describes the actor' s interactions with the system, and
- supplementary information - which adds details and constraints on the use case' execution

The Preamble

The preamble contains information that "sets the stage" for the behavior described in the body of the use case In the preamble, you may find the following information:

- Level - summary, core, supporting or internal use case?
- Actor(s) - role names of people or external systems initiating this use case
- Context - the current state of the system and actor
- Preconditions - what must be true before a use case can begin
- Screens - references to windows or web pages displayed in this use case (if a UI is part of the system)

The Body

Description of the use case' s behavior. This description can be:

- A narrative - a free form paragraph or two of text.
- A scenario - a step-by-step description of one specific path through a use case
- A conversation - a two-column (or more columns if showing dialogs between multiple actors and/or system components) description of the dialog between the actor and the system.

In a single use case, you may write a narrative, and, once you've worked out how the actor will interact with the system, then write either a scenario or conversation. Nothing says that the body has to be restricted to one form. But most of the time we see writers start by writing a very brief narrative (of just a couple of sentences), then write either a scenario or a conversation that goes into more depth. They leave both forms around—the narrative as an overview (which only certain stakeholders read) and the other as an in depth presentation of actor/system interaction.

Supplementary Details

- Variations - different ways to accomplish use case steps
- Exceptions - errors that occur during the execution of a step
- Policies - specific rules that must be enforced by the use case
- Issues - questions about the use case
- Design notes - hints to implementers
- Post-conditions - what must be true about the system after a use case completes
- Other requirements- what constraints must this use case conform to
- Priority- how important is this use case?
- Frequency - how often is this performed?

GUIDELINE: When you start writing use cases, describe the key points. Typically, this means giving the use case a name, identifying the actor and writing the use case body (one of the three forms).

GUIDELINE: Fill in template fields as information becomes available.

As you write a narrative, you may think of an issue or a note to the designer. Jot these down when you think of them. Don't wait for the perfect time. The right time to add a detail is when it occurs to you. You can always note a fact, then fill in more complete details later.

GUIDELINE: Make clear how complete a use case is.

Daryl Kulak and Eamonn Gray in their book *Use Cases Requirements in Context* identify four phases of a use case description: facade, filled, focused and finished.

Whether you pick these four "degrees" of completeness or some other measure of completeness, it is a good idea to note whether a use case is a first draft, whether it has been reviewed, when it has been revised or approved by various stakeholders, and "signed off" as finished.

GUIDELINE: For more formal projects, information about the current state and history of a use case can be added to the template..

Add this information as supplementary details. This lets readers of the use case see the main points first. If you include this information are part of the preamble, it adds clutter that has to be scanned over before the reader finds the main facts about the use case.

A use case description can start out simply, then get quite complex as template details are filled in. Start simply, writing down what you know and issues that need to be addressed. Through several revisions and refinements get to a "finished" use case.

VII. The Narrative Form

Narratives are free-form text in paragraph format. A narrative describes the intent of the user in performing the use case, high-level actions of the user during the use case, and refers to key concepts from the problem domain that are involved in the use case.

Below is an example narrative from the On-line Banking System Specification Documents. We have briefly described the purpose of *Log In* and what happens as a result of the user successfully completing the *Log In*. We've also included a set policies that relate to logging in, and have listed some exceptions that may arise during *Log In*.

Use Case: *Log In*

Log In is the primary entry point into the On-line Banking System. *Log In* verifies that the user is previously registered with the On-line Banking System, and that s/he has correctly entered user id and password information. After a successful login, a registered user can use the system's main functions. All others, regardless of whether they have registered or not, have access to the On-line Banking System Demo and Registration Page.

Recipe: Writing Use Case Narratives

1. Give the use case a descriptive name.

   GUIDELINE: Begin the use case name with an active verb.

2. Identify the actor that uses the use case.
3. Identify the intended audience of the use case.
4. Specify the actor's goals for the use case.

   GUIDELINE: Use active verbs to describe the actor' s goal.

5. Write a description consistent with the name and the user's goal; one that elaborates the use case.

   GUIDELINE: Maintain a single point-of-view: the actor' s.

   GUIDELINE: Describe intent, not action.

   GUIDELINE: Capture the simple, normal use cases first. You will describe the variations as *secondary* use cases later.

   GUIDELINE: If the use case changes the state of some information, describe the possible states.

   GUIDELINE: Write the use case description at a level appropriate for the intended audience.

   GUIDELINE: Leave out details of user interface, performance, and application architecture. Put these details in a central document, and reference these requirements in the use case.

6. Describe any business rules or policies that affect the use case in a separate place: either in a policies section below the use case body; or in a global policies section. Reference globally applicable rules or policies in the use case policy section of the template.

VIII. Scenarios and Conversations: Writing More Detailed Usage Descriptions

One key to developing a usage model is knowing how much to describe. A closely related question is, "What' s the best way to present detailed information?" Use case narratives are general descriptions about how a system supports an actor' s goal. There may be numerous ways to achieve any goal. Sometimes it helps to clarify things by concretely describing actions and information for a specific situation.

Scenarios and conversations are forms that are useful to show in more detail how an actor achieve' s a specific goal.

How many use cases should be written? A glib answer: "As many as it takes to get the main ideas across." The number is highly dependent on how close your intended audience is to the problem, and how many details they need spelled out.

Here' s one general word of advice: Write to be read. If it clarifies and brings understanding to your system' s behavior, write narratives to describe the general situation, then augment those narratives with specific descriptions. If your readership only looks at details, narratives likely won't be of value.

GUIDELINE: High-level use case names state a general goal. Write one narrative use case for each general goal, and as many scenarios or conversations as it takes to get the main ideas across.

For example:

Narrative: Make a payment

Describe what online payment means and typical ways of making them

Write scenarios or conversations that describe more specific goals:

Scenario 1: Make a recurring payment

All the steps in paying my monthly phone bill …

Scenario 2: Make a non-recurring payment

All the steps in paying a fixed amount …

Scenario 3: Make a regular payment

All the steps in paying a monthly loan …

Sometimes, your use cases are read by diverse audiences. Some want to only see details. Others only want "big picture" overviews. In the interests of keeping everything together, and not creating a maintenance problem, we suggest you bundle both a general and a specific description together in a single use case.

GUIDELINE: Write two "versions" of the same use case: one version a narrative, the other version a more detailed form.

Example:

First, write a narrative

The "View Recent Account Activity" narrative describes generally how users view the current or previous account period's transactions

Then, choose an appropriate form. Rewrite the use case body at this lower-level of detail

The View Recent Account Activity conversation includes the details of optional actions, such as downloading a file containing recent transactions in several different formats

Leave the narrative as an overview. Consider adding an "overview" section to your template if you have always have diverse readers for your use cases.

What is a Scenario?

Scenarios are one means to describe a specific path through a use case. A scenario list specific steps toward that goal. It describes a sequence of events or list of steps to accomplish.Each step is a simple declarative statement with no branching. A scenario may describe:

- Actors and their intentions
- System responsibilities and actions

All steps should be visible to or easily surmised by the actor. We typical state a statement by naming who is performing the step. Our goal is to convey how the system and actor will work together to achieve a goal. Even though a scenario can show more detail, resist putting in too much detail. Much of that detail can be placed in the preamble or supplementary parts of the use case template.It should be clear where a scenario starts. Describe the steps in achieving the actor's goal. End there.

For example, we might write a scenario toward the user's goal of "Register a Customer." This specific scenario explains a variation of this task called "Register Customer with Auto-Activation."

Example Scenario: Register Customer with Auto-Activation

1. User enters registration information:
   - *Required information*: user name, email address, desired login ID and password, and confirmation password
   - *One of*: account number and challenge data, or ATM # and PIN
   - *Optional*: language choice and company
2. System checks that password matches confirmation password.
3. System validates required fields and verifies uniqueness of login ID
4. System verifies customer activation information.
5. System creates and activates customer on-line account.
6. System displays registration notification.

Recipe: Writing Scenarios

The purpose of a scenario is to describe the *flow of events* in the use case. These events can be initiated by the user or performed by the system, but should express the steps of the process as the user understands it.

1. For each use case, determine the "happy path" to the actor's goal.

GUIDELINE: Ignore other possible paths through the use case at first. Write these "secondary" scenarios later.

GUIDELINE: Refer to the specific use case that the scenario elaborates, if the use case has been written.

2. Write a scenario as a sequence of steps, ordered by time.

GUIDELINE: Every step in a scenario should be visible to or easily surmised by the user.

GUIDELINE: Write each step as a simple, explanatory statement.

GUIDELINE: Keep information and actions concrete.

GUIDELINE: Focus on ordering and definition of steps.

GUIDELINE: Factor lower-level details into new descriptions.

GUIDELINE: Keep steps ar roughly the same level of abstraction.

In following example, several steps have been compressed to keep actions at the same level.

Mixed level of detail:

1 | Check for required fields
2 | Capture user ID and password
  | Ask security component for validation
  | Issue SQL statements to security database for logon authorization…
3 | Open connection to bank server
  | Read account summaries…

Fixed:

1 | Check for required fields
2 | Login user to domain
3 | Display account summaries and bulletin

3. Number the steps.

GUIDELINE: Don't get carried away. Keep the numbering one level deep. Remember, the goal is clarity.

4. Look for steps that might repeat within the scenario.

GUIDELINE: To show repetition, use *repeat* or *while* statements.

GUIDELINE: Avoid the tendency to write pseudocode unless your audience are programmers who only understand code.

5. Look for steps that depend on a condition.

GUIDELINE: To show that a step depends on a condition, use an *if* statement.

GUIDELINE: When the logic for expressing a conditional statement becomes too complex, write another, *alternative* scenario.

GUIDELINE: Distinguish between *variations* and *exceptions*. Describe recovery from exceptions in a supplementary note or another scenario if the recovery is complex. Document variations in either a supplementary note, or another scenario if the actions are interesting.

6. Look for sequences of steps that repeat *across* scenarios.

GUIDELINE: Don't do this early in your project. Later, factor out portions of a scenario that repeat in other supporting scenarios, give them a name, and refer to them within the core use case with a reference to the supporting use case' name.

7. Look for optional steps.

GUIDELINE: Preface optional steps or actions with "Optionally,..". Indent optional steps for clarity.

8. Show the range of values of data that is used in the scenario.

GUIDELINE: If the user changes the information, specify the possible states that the information might go through.

What is a Conversation?

A conversation describes a significant sequence of interactions between an actor and the system, or between one part of our system and another. It is a detailed description of a Use Case that clearly defines the responsibilities of each participant.

There are two central parts to a conversation, a description of requests or inputs, and a corresponding description of the high level actions taken in response. Together, these "side-by-side" descriptions capture a sequential ordering of communications.

Like a scenario, it can show optional and repeated actions. Each action can be described by one or more substeps

The focus of a conversation is to detail the types of interactions, the flows of information, and the first-level system logic of the system, all from the user's point-of-view. If desired, it can also be used to drill down to the deeper levels of system logic, as seen by a developer.

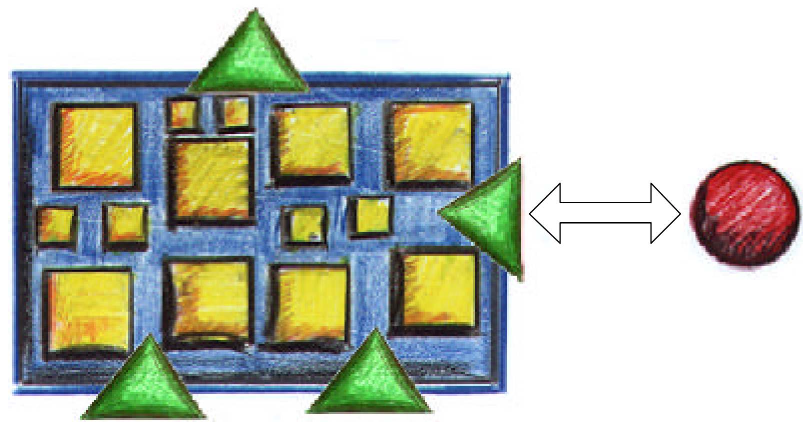


FIGURE 5. A conversation shows a dialog

Because of the various ways in which a user task can be performed successfully, there may be one or more conversations for a single use case narrative.

Often, it is too big a gap to move directly from informal Use Case written in narrative form to design. Also, use cases written at this higher level are full of ambiguities and extraneous details that have little to do with what our system must do for the user. As you restructure use cases into conversations, you add more detail by:

- showing branching and looping,
- describing constraints on what our software should do,
- describing the context in which the conversation occurs,
- identifying the actors that initiate the activity,
- defining the “standard” course of action and alternatives to it,
- raising unanswered questions, and
- adding design notes.

TIP: This supporting information is often as important as is defining the order to the user's actions.

Notation

Use a table format to record a conversation several stylistic shorthand conventions. Other use cases invoked during a conversation are marked in bold text. These use cases could be “used” (UML's “includes” relationship), or transfer of control could passed via a “goto”. These control flow conventions proved extremely relevant to the UI designer and application server implementation, but are unimportant to a high-level view of a use case.

Optional actions, for example (Indicate Setup Payees), are labelled. Show looping or repetitive steps by merging adjacent cells in a row to bracket the beginning and end of a block of repeated or optional actions:

Repeat	
actions go here	
Until proposed schedule is built	

FIGURE 6. Showing Repetition, or an optional block of actions

Placing dialog in adjacent cells of the same row shows an interactive round (an actor action that invokes a nearly simultaneous system response). Placing the system response in the row immediately after the provoking action denotes a batch round.

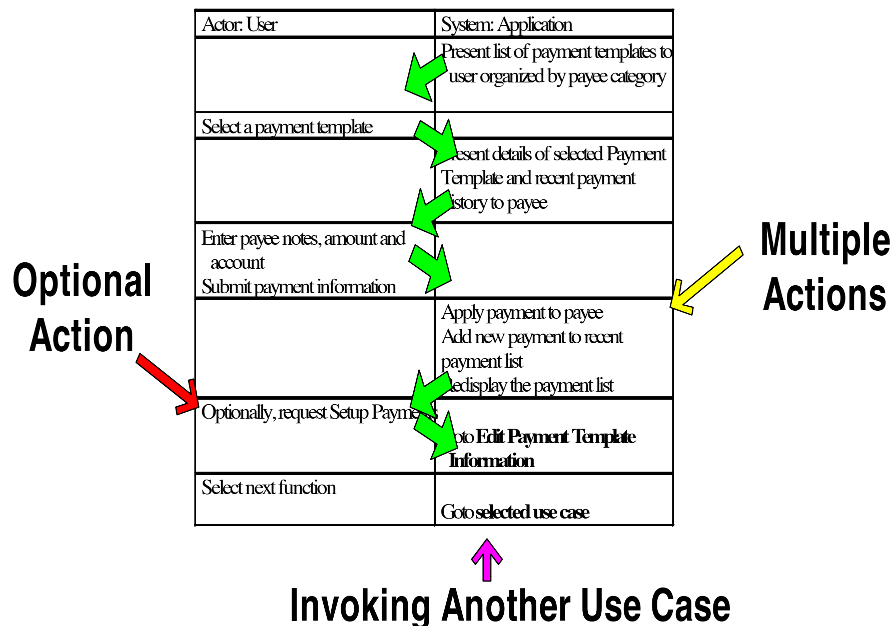


FIGURE 7. Conversation notation

This is an example of a dialog between the customer-user and the system. We used a table format to record this dialog and several stylistic shorthand conventions. Other use cases invoked during payment were marked in bold text. These use cases could be "used" (UML's uses relationship), or transfer of control could passed via a "goto." These control flow conventions proved extremely relevant to the UI designer and application server implementation, but are unimportant to a high-level view of a use case.

In the on-line bank, our web-based interface design did not allow for simultaneous interactions; instead information would be batched and passed along with an action tied to a button. A more traditional window application or a Java applet has the potential for many more overlapping activities.

Writing Conversations

Knowledgeable experts from diverse backgrounds can readily construct conversations. Conversations can either be developed by a team or drafted by an individual then reviewed, explained, and revised by a small group. It is important that teams who develop conversations blend the talents of developers, users, and other specialists. Each contributor has a unique and valuable perspective. No perspective should dominate, yet a certain interest may take center stage during a working session. It is important that side concerns be recorded, and worked through, perhaps as an outside activity. Respect and appreciation for the concerns of others is important; teamwork and a spirit of joint development is crucial. For example, in one working session, we dived into technical design details for several minutes, backed up to re-examine whether the flow of the conversation we had proposed was still workable, then summarized what issues were solved and what new ones were raised by a single decision. Technical, user interface and business issues were all discussed in a single session while holding everyone's attention.

One key to building a good conversation is to preserve its dual purpose of

1. recording of the important events and information that are conveyed between the user and the system; and
2. guiding developers who will be creating the object design model.

To meet these objectives, conversations must be written at a fairly high level. It often is the case that sequencing of model responses (for example the details of recording a payment transaction) are not accurate reflections of the tasks that the system must do. Yet, they need not be early on. What is important, is that an interdisciplinary team is sketching out how they expect their software system to work.

Conversations capture the flow of communication between actor and system. If the nature or the amount of information changes significantly, the demands on your object model also change. So we suggest that you include sufficient detail, and reflect changing interaction and interface design if conversations are to actually guide object modeling.

What is a dialog?

The basic form of conversation is the dialog. A dialog is a conversation where both sides participate in a structured sequence of rounds of interaction. Each round is a pairing of an action taken by the user, and the software system's response to this action. A round is one of two types. It is either an interactive round or a batch round. An interactive round features interplay of user actions and system responses. For example, the validation of a single key press among many is typical of an interactive round. In contrast, filling out several entry fields and then submitting them all at once is a more typical of a batch round.

This sequence of rounds establishes a necessary ordering of the interactions, and details the individual activities of the user and the system's response in each round. Many of our conversations between human actors and our system are of this form

	Actor Actions	System Responses
Batch round	And I respondby ..	I do this
Interactive Round	I tell you this…	I am respond to what you are telling me and giving you feedback while you are talking

FIGURE 8. General form of a conversation

Choosing Between Conversations and Scenarios

The detailed form you choose to use depends on two primary factors:

- whether or not your system has meaningful dialogs between its users; and
- personal preference

Use a scenario when:

- a simple list of actions is sufficient
- actor-system interactions aren't interesting

Chose a conversation when:

- there are many interactions and you want to describe them
- you want to show more details in your system responses
- you want to separate the roles of actor and system and clearly identify at each point the system does for the actor

We have written conversations for systems where there isn't a lot of interaction between actor and system. This becomes readily apparent from looking at the staggered pattern of filled in cells.

Most projects write high-level use case narratives, then standardize on one of the two more detailed forms to describe all use cases. Whether you want to write conversations or scenarios may not be obvious until you understand the nature of your system' s interactions with its users.

Recipe: Writing Conversations

GUIDELINE: Write a conversation if it is important to show the patterns of interaction between the actor and the system.

GUIDELINE: Write a conversation if you want to show the first cut at system-actor actions in greater detail.

GUIDELINE: If you have written a scenario and find that it does not offer you enough detail, rewrite it as a conversation.

1. List the actor actions in the left column and the system actions in the right column.

GUIDELINE: Leave out presentation details.

GUIDELINE: Maintain a consistent level of detail.

GUIDELINE: Don't embed alternatives in your action descriptions.

It can get complicated for your readers to deciphers nested "if then... else, if.." statements if they are liberally sprinkled through your action statements. You can keep the statements simple if you write the "happy" path description in the body of the conversation. Call out exceptions and variations below.

Pseudo-code:

Conversation: Registration with Automatic Activation

	10. If bank supports automatic activation with ATM and PIN then... If ATM and PIN #s are valid then....

Fixed:

Conversation: Registration with Automatic Activation

	10. Validate ATM and PIN #

Exception

Step 10: ATM and PIN #s are invalid- Report error to user

GUIDELINE: Don't mention "objects" in system responses.

Remember that your readers what to know what is happening from an external perspective, not what the system is doing behind the scenes. For example, rather than stating "create customer and account objects" you can rewrite the system's response to more clearly explain what the system has done to benefit the actor: "record customer account information".

GUIDELINE: Write conversations with a small group (maximum of 3).

When we first started defining the On-line Banking System, we wanted every developer to understand all facets of the system. This quickly proved impractical and slowed everyone down. So, two of us focused on use cases and conversations, interacting primarily with the system architect and domain expert. Eventually one person took over maintaining use cases; everyone else used them as reference material. For example, the developer who designed and implemented the application server only raised questions when conversations were unclear or inconsistent, and was quite content to not always work from the latest documentation until things settled down. The project manager and project sponsor didn't read these documents at all (unlike other projects we've worked on where management enjoys reading and commenting on them in detail).

2. When the system has an *immediate* response to an actor action (such as validating a key stroke), list them in the same row.

Copyright 2002 Wirfs-Brock Associates

GUIDELINE: Leave out information formats and validation rules.

These are best kept in a separate place that can be maintained and updated as business procedures and policies may change. Only summarize what information is presented to or collected from the actor in the conversation.

Rules and information model embedded:

User Name: First name, last name (24 characters maximum, space delimited)

email address with embedded @ sign signifying break between us identification and domain name which includes domain and s domain names delimited by periods and ending in one of gov, edu...

Fixed:

Required: user name, email address, desired login ID and passwor

One of: account number and challenge data, or ATM # and PIN

3. When the response is delayed until an entire actor action is complete, list it in the row immediately below the row with the actor action.
4. Write any assertions about the software's states during the conversation.
5. As you consider the actions in the conversation, document any ideas about "how" in a *Design Notes* section.
6. Test the conversation with a walkthrough.

GUIDELINE: Use specific examples to walk through use cases and conversations.

GUIDELINE: Trying to abstract or write more general conversations too early tends to create problems. It is better to deal with specific situations first, then review and combine things as appropriate, after you have the big picture. This strategy led us to write different conversations to record different typical uses. For example, we discovered two common ways customers could make payments, one for paying same amount to the same vendor and one where the amount paid varies. This led us to write two separate conversations, Make Similar Payment and Make Payment. In version two, when we would support automatic payments, Make Recurring Payment would be added to our Payment Use Case conversations.

7. Check conversations for completeness.
8. Relate the conversations through their preconditions and postconditions.

IX. Other Descriptions, Exceptions and Variations

"Other" requirements are those that are not captured within the body of a use case, or within or within other parts of the use case template. They can either be kept in a central place or can document the use case where they seem to apply. In fact, if you are following a rigorous requirements specification process, you may gather and record many requirements that, while they *may* impact your system's usage and design, belong elsewhere.

Keep Common Requirements in a Central Place

GUIDELINE: Document requirements spanning use cases in a central place.

For example: "Financial transactions must be secure.", and "System must run 7x24"

GUIDELINE: Refer to specific "central" requirements by name in the use cases that they impact if this impact is not obvious to the reader and it's important to know.

Note Specific Requirements in Use Cases they Affect

GUIDELINE: Document specific requirements in the use case that they pertain to.

For example: "Registration response time must be less than one minute."

GUIDELINE: Look for requirements that are invisible to the actor.

For example: "System must not lose any requests", or "Application servers will be widely distributed"

GUIDELINE: Look for performance requirements that affect system behavior.

Design Notes

Design notes, if part of your use case template can "round out" your usage descriptions with ideas that occur to you that might be useful during design. Since a use case isn't a descriptions of a solution, don't write these details there. But if you think of a good design idea, you may want to jot it down and keep it with your use cases.

GUIDELINE: Add design notes as they occur to you when writing scenarios and conversations.

For example: "Errors and warnings about registration information contents should be collected and returned to the user in a detailed message rather than stopping at the first detectable error", and "Payments should be shown in time order, with the current date first."

GUIDELINE: Write design notes as hints or suggestions, not as instructions to the designer. Don't be too detailed.

Alternatives

Distinguishing which courses of action are the "main" paths is often difficult. We have two options for documenting alternative courses of action (variations) and points of potential error (exceptions) in a use case. If the alternative can be stated simply, we embed *if-statements* in the description. For a slightly more complex alternative, we note these alternatives in supplemental text below the basic path. Or, when the alternative flow of events is complex, we can write completely new use cases for these alternatives. In the latter two situations, we reference the point in the original use case where the alternative takes place.

Variations

A *variation* can be a different action on the part of either the actor or the system. When you see this possibility, be opportunistic! Don't let the insight go by. Capture the conditional choice in an *if* statement, describe the difference in supplemental text below the use case body, or write another use case. One that incorporates the alternate action. Note the name of the new use case so that you can write it later.

Exceptions

On the other hand, actions that have the potential for errors, again, on the part of either the actor or the system. Treat these errors similarly to variations, but note them under a separate heading in the supplementary part of the use case template. They are the source of many of the error-handling requirements of the system.

GUIDELINE: Describe the exception and its resolution. Identify whether it is recoverable (e.g. the actor can continue on with his/her task in some fashion) or unrecoverable.

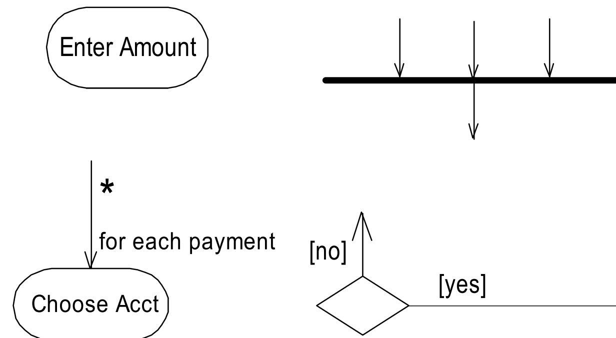
GUIDELINE: For each recoverable exception describe how the actor/system needs to respond to make forward progress.

GUIDELINE: For each unrecoverable exception, make clear what state the system returns to after detecting this condition, and how the actor is notified of this condition.

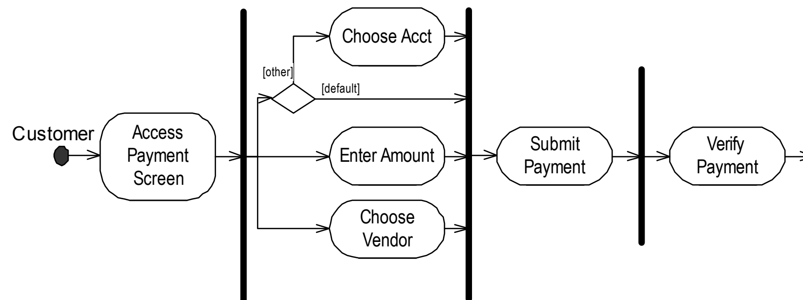
Activity Diagrams

Activity Diagrams are a UML standard way of describing sequences of actions, the dependencies among them, and the parallelism and synchronization characteristics. Use them as a way of visualizing activities at several levels: the process level that demonstrates how different use cases interact, the task level that shows the activities of a user when performing a use case, and the subfunction level that shows the internal workings of a single step, whether it be performed by a user or a computer program.

These elements show an activity (the oval), synchronization of activities (the synchronization bar), decision-making (the diamond), pre and post conditions (the *guards*, text inside the square brackets annotating the arrows), and iteration (the asterisk on annotating an arrow).



This activity diagram demonstrates the actions that take place when "Making a Payment". It is at the task level and describes a single use case.



GUIDELINE: Use an activity diagram to describe a single use case. The goal is to understand what actions that take place and the dependencies between them

GUIDELINE: Use an activity diagram to understand workflow across use cases. Activity diagrams are great for showing connections and dependencies between the use cases of an application.

GUIDELINE: Use an activity diagram to show parallel activities. Activity diagrams are particularly good at showing parallelism, synchronization, and pre and post conditions.

Recipe: Writing Exceptions

The typical paths through the use case is specified in *primary* descriptions. Alternatives to these paths can be written as *secondary* use cases, or named in *variations* and *exceptions* sections.

1. Look for potential exceptions in each primary use case.

GUIDELINE: When looking for exceptions, ask:

- Is there something that could go wrong at this point? (exception)
- Is there some exceptional behavior that could happen at any time?

GUIDELINE: Be opportunistic! Document the exceptions whenever they occur to you.

The first step is identifying the exception. The next step is resolving how it will be handled.

GUIDELINE: Keep the exceptions at the same level of abstraction as the use case description.

2. Determine which exceptions should be written as separate use cases.

GUIDELINE: Defer writing these secondary use cases until you feel satisfied with your primary ones.

GUIDELINE: Write secondary use cases according to the recipe and guidelines for primary ones.

3. Document the exceptions.

GUIDELINE: Describe the exception condition. Note whether it can be recovered from or not. Describe the actions the actor or system take to recover; or to end the use case in an unrecoverable situation.

GUIDELINE: Choose the clearest way to describe how the exception is handled

Options include:

- Briefly describe what happens, or
- Refer to another use case that describes the exception handling

4. Refer to the place in the original use case where the exception takes place.

GUIDELINE: Insert footnote numbers or tags into the main scenario, and tag the alternatives with the same number.

Recipe: Writing Variations

The typical path through the use case is specified in the *body* of the use case. Alternatives to these paths can be written as *secondary* use cases, or named in *variations* sections.

1. Look for potential alternatives in each use case body.

GUIDELINE: When looking for variations, ask:

- Is there some other action that can be taken at this point?

GUIDELINE: Be opportunistic! Document the variations whenever they occur to you.

2. Determine which variations should be written as new use cases.

GUIDELINE: Defer writing these secondary use cases until you feel satisfied with your primary ones.

GUIDELINE: Write secondary use cases according to the recipe and guidelines for primary ones.

3. Document the variations.

GUIDELINE: If the variation is easily added to the use case body, put it there. Show that variations are optional by indicating that one of several choices can be made for a particular step.

GUIDELINE: If the description of variations clutters up a use case description, write about it in the supplementary part of the use case template.

Refer to the place in the original use case where the variation takes place.

GUIDELINE: Insert footnote numbers or tags into the main body, and tag the variation with the same number.

Assertions

Assertions about our system's behavior are useful for:

- Generating the flow of system events
- Determining use case dependency relationships
- Understanding the states of the application

We make three kinds of assertions: preconditions, postconditions, and constants.

Pre-conditions

Pre-conditions are what must be true of the state of the application for the use case, scenario or conversation to be *applicable*. They can also imply the possibility of some order of the use cases, as we will see in the next section on post-conditions.

Post-conditions

Post-conditions are what must be true of the state of the application as a *result* of completion of the use case, scenario, or conversation.

For example, in the On-line Banking *Make a Payment* use case, debiting an account leaves the system in one of two states:

- InGoodStanding
- OverDrawn

These post-conditions of *Make a Payment* lead to two different system states. In the first case: "OverDrawn" leads to *not permitting another make payment* use case to execute (until the Account is InGoodStanding) because InGoodStanding is a precondition for Make a Payment. In the second case, InGoodStanding enables another Make a Payment to be executed.

GUIDELINE: Document pre and post-conditions where the system responds differently as a result.

We have seen many people struggle with the question, "what's a good post condition?" A bad post condition adds clutter and doesn't add any information. Restatements of the actor's goal don't add information. If the goal is to Make a Payment, then saying that a payment has been made doesn't add any information. A good test of a post-condition is that it states something about the system that may or may not be obvious from completing a use case. And, ideally, a post-condition may enable another use case to be executed.

Example of a poorly stated post-condition that restates the use case goal:

Post-condition: Customer has withdrawn cash

So what? The customer receives cash but what does this say about the next time he/she wants to withdraw case, or any other use case?

Fixed:

Post-condition 1: Account balance is positive

Post-condition 2: Account is overdrawn

Note that the user may have achieved his/her goal, to withdraw cash, but depending on the amount withdrawn and the account's balance, his/her account may be in one of two possible states after successfully withdrawing cash. Now that's interesting!

GUIDELINE: Specify pre- and post- conditions only when you need to be formal

Once you add pre- and post-conditions to one use case, you will need to add them to dependent ones! A use case model that only has pre and post-conditions on a few use cases begs the question, is this complete or are there gaps in this specification?

GUIDELINE: Check for completeness of use case dependencies by asking how each use case is enabled, and the conditions it sets that enable others.

Example: Pre-conditions should make clear when a use case can execute

An account must be in good standing and the daily withdrawal limit not exceeded in order to withdraw cash

Post-conditions may be relevant to other systems

Being overdrawn may trigger transaction fees

Pre-conditions may be set by other systems

An account can be overdrawn through direct payments

GUIDELINE: Complete the specification of pre and post-conditions by documenting the possible states of the system after each exceptional condition, and each variation of a step.

Example:

Often, there are multiple post-conditions for one scenario or conversation

At least one for each successful goal...

Customer receives cash? Account is overdrawn or Account balance is positive

One for each exception...

Account daily limit would be exceeded - Customer withdraws lesser amount? Account is in good standing and Account daily withdrawal limit reached

Amount would exceed overdraw limit - We refuse to disburse cash? account is in good standing

One or more for each variation...

Fast cash? Account is overdrawn or Account balance is positive

Constants

Constants, sometimes called *invariants* are what must be true of the state of the application *during* the entire progress of the use case, scenario, or conversation. They are often contextual and must not be changed at any moment during the use case.

GUIDELINE: Be careful about getting too formal. Assertions tend to make requirements look incomplete if they vary in their formality.

GUIDELINE: Use pre-conditions to make it clear *when* a conversation might execute.

GUIDELINE: Write post-conditions as if you were going to use them as a basis for writing a test plan. You are.

GUIDELINE: Write constants to describe conditions that should not change during the conversation.

X. Use Case Model Checklist

At the end of the day, the goal of a usage model is to convey how a system behaves and responds to its users. A good usage model conveys how a system behaves, and how behaviors are related.

You can look over a use case model to:

- Check for internal consistency between use cases
- Identify “central” use cases
- Identify unmet or externally satisfied preconditions
- Review the actor’s view for completeness
- Review the handling of exceptions
- See that use case dependencies, extensions and includes relationships have been documented

Organizing Your Use Cases

Organizing use cases is important. A pile of usage descriptions, arranged alphabetically, doesn't orient readers to the usage terrain. We suggest that you choose an organization that helps orient your typical reader.

Some possible organizations:

- by level (summary first, core next, supporting, then internal ones last)
- by actor
- by type of task
  arranged in a workflow

Be consistent. Keep various forms of a single use case together.

XI. A Use Case Writing Process

The task of writing can be shared, but the best way to develop a common language is for teams to work on developing a rhythm to their work. Sometimes it is best to get group consensus, othertimes it is best to work alone (or in a small group) to create use cases that others can review. Writing, like programming, can be done solo, then reviewed as a group. Once you pick a template and learn the common ideas, you can try writing solo, then critiquing as a group. Group review can lead to a common style and format for usage descriptions. We suggest this process as one way to work collectively and individually to develop a use case model:

Full Team	Small Teams or Individuals	The Products
Align on scope, level of abstraction, actors, goals, point-of-view		Actors, Candidate Summary Use Case Names
	Write summary descriptions	Narratives
Collect and clinic, brainstorm key use cases		Candidate Core Use Case Names
	Write detailed descriptions	Scenarios OR conversations
Collect and clinic, identify gaps and inconsistencies		Potential new Use Cases
	Revise and add precision	Revised Use Cases with Supplementary Details

FIGURE 9. A Process for Developing a Use Case Model that includes both team and individual work.

Note: Although not everyone is a skilled writer, most developers can write good use cases. It is a matter of writing and reading good use cases (and then adopting a common style). This involves practice and critical review.

XII. Tips and Techniques

We have pulled many commonsense writing guidelines from Ben Kovitz' s wonderful book *Practical Software Requirements*. They were either paraphrased or taken verbatim from his chapter on writing. Other guidelines on what extra efforts can have big payoffs come from our experience. If you apply these principles to your writing of use cases and other technical writing, your readers will be the beneficiary of your efforts.

Broad Principles

GUIDELINE: Read other people' s writing. If your own documents are hard to understand, you don't notice because you already know what it' s supposed to say.

Writing is a craft. If writing is a large part of your job, people will judge you not on the basis of your thinking, but on the basis of your writing.

GUIDELINE: Write for human beings.

- Is there a way to express this that would be easier to understand?
- Am I overloading the reader with too much information at once? Should I provide some sort of roadmap, or break it up into smaller sections or smaller sentences?
- Which details are more important to my readers and which are less important? How can I make clear which details are which?
- Is this statement too abstract for my readers to understand without illustration? Are these details too narrow and disconnected for my readers to understand without explaining the underlying principle common to them all?
- What reasonable misinterpretations could my readers make when reading this passage?
- Will my readers see any benefit from reading this section? How does it relate to my specific reader' s job? Does anyone have a reason to care about this? Will people see this as a waste of time?
- What is the feel of the writing?
- Is the document boring? Would anyone want to read it?

GUIDELINE: Choose the best alternative for expressing your thought, despite the rules.

GUIDELINE: When you have information that can be presented in a list, it is usually the best way. People like lists

GUIDELINE: Choosing the way to say something should derive from the content.

GUIDELINE: use a consciously designed organization for your document. Then there is "a place for every detail, every detail in its place."

GUIDELINE: Reinforcement makes a document understandable. Illustrations, overviews, section headings. Repetition, on the other hand, is decoy text.

Decoy Text

GUIDELINE: Avoid metatext. Text that describes the text that follows.

GUIDELINE: Avoid generalities.All information in a requirements document should be specific to the software to be built.

GUIDELINE: Avoid piling on words or explanations.

Remove clutter at all levels. You can clutter sentences, words, paragraphs, or sections of documentation with extra meaningless words. Overbearing templates also contribute to clutter.

An example:

Piling on: Business Use Case

Clutter Removed: Use Case

Another example:

Piling on: Requirements Specification Document

Clutter Removed: Requirements

GUIDELINE: Keep extraneous documents out of your requirements document. Schedules, acceptance criteria, traceability matrices, feedback forms, etc.

Avoiding Common Mistakes

GUIDELINE: Put related material together. Avoid making your document a jigsaw puzzle.

GUIDELINE: Don't mix requirements with specification. The what with the how. Don' t confuse means with ends.

GUIDELINE: Choose the most appropriate vocabulary for expressing a requirement. Don't force fit your descriptions into inappropriate diagrams, charts, and tables just because they are "usual".

GUIDELINE: Avoid "Duckspeak" (from *1984*). Meaningless sentences expressing conformation to standards.

For example, "The order data validation function shall validate the order data."

GUIDELINE: Know the vocabulary of your readers and use it. Don't invent unnecessary terminology.

GUIDELINE: Be aware of what content you are putting in your document. Don't mix levels.

Jumping back and forth between program design, requirements, and specification will only confuse the reader.

GUIDELINE: Don't start with a table of contents taken from another document.

This is equivalent to forcing the content of one document into the table of contents of another

GUIDELINE: Use consistent terminology.

GUIDELINE: Don't write for the hostile reader. Assume the reader will try to understand.

GUIDELINE: Make the requirements document readable. If it is not, the development staff won't read it.

Poor Uses of Documentation

GUIDELINE: Avoid documentation for the sake of documentation. Don't try to make your documentation an end in itself.

GUIDELINE: Requirements documents are not written to impress the customer with double-talk.

GUIDELINE: Don't write a CYA document. In these cases, most information must be communicated to the development staff by oral tradition.

GUIDELINE: Write questions about unsolved issues.

Put them with the appropriate use case description (or with the document you are working on) to show you' re not done.

Example: Should the credit check be performed after the Order is submitted or before? What happens if credit is denied?

GUIDELINE: If you are unclear about a detail, don' t write fiction; it could become fixed.

Guidelines for each element of a Use Case Template

In addition to the above general guidelines for writing, we offer these specific guidelines for writing use cases drawn from our direct experience.

Use Case Name:

A name of some actor task to be accomplished with the system. Name it from the actor's point of view

Good Example: ***Place an Order****, or* ***Cancel an Order****, or* ***Make a Payment***
Bad Example: ***Process Order Record***
This is named from the system's point-of-view
Bad Example: ***Placing an Order***
This is not stated with an active verb

Narrative Description:

A high-level narrative paragraph describing activities of a task

Actors:

Role names of Person or External System initiating this use case

Good Example: ***bank customer***

Bad Example: ***novice user***

This is a skill level, not a role. If novices do things differently, than skilled users, then perhaps their different forms of interaction might be described… but the role is user (not novice or skilled user)

Context:

A description about the current state of the system and the actor
Good Example: ***The bank customer is a primary user***
Bad example: ***The customer wants cash***
So what? Expressing desires clutter our descriptions. Always assume actors want to accomplish some goal, and that the system is ready to respond. Don't state the obvious.
Bank customer: ***The bank customer is logged on***
This is obvious. Don't state the obvious. It adds clutter.

Level:

Is it Summary, Core, Supporting or Internal?
Example:
Place Order (summary)
Order Long Distance Phone Service (core)
Enter Customer Address (supporting)
Obtain Secure Connection (internal)

Preconditions:

Anything significant about the system that must be true. Usually stated in terms of key concepts and their states.

Good Example: ***A bank customer's account is in good standing***
This must be true before he can make a withdrawal
*Bad Example**: The bank customer is logged in***
This is context, not something true about the state of the system

Post conditions:

Anything that has changed in the system that will affect future system responses as a result of successfully completing the use case. Usually stated in terms of key concepts and their states.

Good Example: ***The bank customer's account is overdrawn***
This means that the customer cannot make another withdrawal until the account balance is positive
Bad Example: ***The bank customer received cash***
This says nothing about how the system will respond in the future

Business Policies:

Business specific rules that are always true that must be enforced by the system.
Test for whether a policy is application specific or a business policy: Who established this policy? Was it the application designer, or was it the way we do business?

Good Example: ***Shipping dates must not fall on Sunday or holidays***
*Bad Example****: The system must determine the shipping date***
This is a statement of something the system must do, a system responsibility, not a rule that the system will enforce.

Application Policies:

Limits on the way than an application can behave.
Here's a simple test for whether a policy is application specific or a business policy: Who established this policy? Was it the application designer, or is this the way we do business?

Good Example: ***A user cannot incorrectly enter a password more than three times during a login attempt***
Bad Example: ***The password is encrypted then matched with the stored encrypted password***
This states how the system is going to validate the password, a system responsibility

Alternatives:

Deviations from a step that occur due to exceptions or decisions made by the system or actor. An alternative can either be a variation or an exception.

Variations *Optional actions for a step that are normal variations (not errors)*
Exceptions *Errors that occur during the execution of a step*
An alternative form can be written as either

- Step number. Variation or Exception Name – Brief statement of how this alternative will be handled,

Example:

Scenario: Identify Customer
1. Operator enters name
2. System finds and displays near matches

Variations:
1a. Operator enters billing address
1b. Operator enters phone number
1c. Operator enters customer address

Exceptions:
2a. No near match found—Notify operator to retry search
2b. Too many near matches found—Notify operator how many matches were found, and give option to narrow search or display matches

or, if handling the alternative warrants it:

- Step number. Reference to Use Case that describes the interactions with the system to handle the alternative

Good example:

Scenario: Identify Customer
1. Operator enters name
2. System finds and displays near matches

Exceptions:
2a. Too many near matches found—use ***Narrow Search Request***

Copyright 2002 Wirfs-Brock Associates

Issues:

Questions that need to be resolved about this use case, scenario or conversation.
Issues should be stated simply. If you know who should resolve this issue, identify them.

Good Example:

Should a credit check be performed for new customer before placing orders? Should credit checking be performed if an order exceeds a certain amount? To be resolved by: John

Bad Example:

What about credit checking?

(What is meant by this question? Is it unclear exactly what the issue with credit checking is.)

Design Notes:

Design decisions that occur to you as you describe the usage

Good Examples:

If the bank does not permit automatic activation, the fields for ATM and PIN number should not be displayed*. (Hints to the application designer)*

User Beware! If the user enters an incorrect ATM PIN number, it is possible that he could be suspended from use of his/her ATM. We must be sure to let the user know about that error.
(Important notes about how the errors should be presented to the user—from the analyst's perspective)

Bad Example:

All errors should be reported to the user.

(Too vague. What's a designer to do with this note?)

Screens:

References to windows or web pages that are displayed during the execution of this use case

Good Examples:

Include a reference to a hand drawn "sketch" of a UI or a mock-up (this is good in early prototyping). Include a prototype screen "captured" off the display. Label important important elements where information is gather and/or presented, and important user actions occur.

Bad Example:

Include detailed screens after they are implemented
(Too specific. What's the point of showing this level during requirements?)

Priority:

How important is this?

Frequency:

How often this is performed?

Good Example: ***200 times a month***
Bad Example: ***200 times*** (What' s the unit of time?)

What It Really Takes to Handle Exceptional Conditions

Rebecca Wirfs-Brock
Wirfs-Brock Associates
www.wirfs -brock.com
rebecca@wirfs-brock.com

This material is taken from *Object Design: Roles, Responsibilities and Collaborations* by Rebecca J Wirfs-Brock and Alan McKean, to be published by Addison Wesley in November 2002. Copyright Addison-Wesley 2003. Used with permission of the publisher.

Henry Petroski, structural engineer and historian, talks of the need to understand the consequences of failure: "The consequences of structural failure in nuclear plants are so great that extraordinary redundancies and large safety margins are incorporated into the designs. At the other extreme, the frailty of such disposable structures as shoelaces and light bulbs, whose failure is of little consequence, is accepted as a reasonable trade-off for an inexpensive product. For most in-between parts or structures, the choices are not so obvious. No designers want their structures to fail, and no structure is deliberately under designed when safety is an issue. Yet designer, client, and user must inevitably confront the unpleasant questions of 'How much redundancy is enough?' and 'What cost is too great?' " As software designers, we too need to make our software machinery hold up under its anticipated use.

Software need not be impervious to failure. But it shouldn't easily break. A large part of software design involves building our software to accommodate situations that, although unlikely, still have to be dealt with. What if the user mistypes information? How should the software react? What if items a customer wants aren't available? Even if the consequences of not delivering exactly what the customer wants are not catastrophic, this situation must be dealt with reasonably—in ways acceptable to the customer and the business.

When information is mistyped, why not notify the user and let them re-enter it. Not enough stock on hand? Again, ask the user to cancel or modify their order. Software should detect problems and then engage the user in fixing them!

But what if a user is unable to guide the software? Shouting "stack overflow!" or "network unavailable!" won't help a disabled person who communicates by using software that interprets her eye blinks and constructs messages. "Punch in the gut" error messages are unacceptable in that design. It should handle many exceptional conditions and keep running without involving the user.

There is an enormous difference between making software more reliable and "user attentive," and designing it to recover from severe failures. Fault tolerant design incorporates extraordinary measures to ensure that the system works despite failure. For example, telephone-switching equipment is extremely complex, yet has to be very reliable. Redundancies are built into the hardware and the software. Complicated mechanisms are designed to log and recover from many different faults and error conditions. If a hardware component breaks, a redundant piece of equipment is provisioned to take its place. The software keeps the system running under anticipated failure conditions without losing a beat.

The more serious the consequences of failure, the more effort you need to take to design in reliability. Alistair Cockburn, in *Agile Software Development*, recommends that the time you

spend designing for reliability fit with your project's size and criticality. He suggests four levels of criticality:

- **Loss of comfort.** When the software breaks there is little impact. Most shareware falls into this category.
- **Loss of discretionary monies.** When the software breaks it costs. Usually there are workarounds, but failures still impact people, their quality of work and businesses effectiveness. Many IT applications fall into this category.Applications that affect a business' customers do so as well. If a customer gets overcharged because of a billing miscalculation, this doesn't cause the business severe harm. Usually the problem gets fixed, one way or the other, when the customer calls up and complains!
- **Loss of essential monies.** On the other hand, some systems are critical. At this level of criticality, it is no longer possible to correct the mistake with simple workarounds. The cost of fixing a fault is prohibitive and would severely tax the business.
- **Loss of life.** If the software fails people could get injured or harmed. People who design air traffic control systems, space shuttle control software, pacemakers, or anti-locking brake control software spend a lot of time analyzing how to keep the system working under extreme operating conditions.

The greater the software's criticality, the more justification there is for spending time to design it to work reliably. Even if not a matter of life and death, other factors may drive you to design for reliability:

- Software that runs unattended for long periods may operate under fluctuating conditions. Exceptional conditions in its "normal" operating environment shouldn't cause it to break.
- Software that "glues" larger systems together often needs to check for errors in inputs and work in spite of communications glitches.
- Components designed to "plug in" and work without human intervention need to detect problems in their operating environment and run under many different conditions. Otherwise, "plug and play" wouldn't work.
- Consumer products need to work, period. Their success in the marketplace depends on high reliability.

A Strategy For Increasing your System's Reliability

Reliability concerns crop up throughout development. But once you've decided on the basic architecture of your system, assigned responsibilities to objects, and designed collaborations, you can take a closer look at making specific collaborations more reliable—by designing objects to detect and recover from exceptional conditions.

We suggest you start by characterizing the different types of collaborations in your existing design. This will give you a sense of where you need to focus efforts on improving objects and designing them to be more resilient. Then, identify key collaborations that you want to make more reliable. Once you've characterized you system's patterns of collaborations and prioritized you work, you need to get very specific:

- List the exceptions and errors cases you want your design to accommodate.
- Decide on reasonable exception handling and error recovery strategies to employ
- Try out several design alternatives and see how responsibilities shift among collaborators. Settle on a solution that represents a best compromise.

- Define additional responsibilities for detecting exceptions and obligations of other objects for resolving them if that is part of your solution.
- Look at your design for holes, unnecessary complexity, and consistency

A system is only as reliable as its weakest link. So it makes little sense to design one very reliable object surrounded by brittle collaborators. Or to make one peripheral task very reliable while leaving several central ones poorly designed. The system as a whole needs to be designed for reliability, piece by piece.

Determine where collaborations can be trusted

One way to get a handle on how collaborations can be improved is to carve your software into regions where "trusted communications" occur. Generally, objects located within the same trust region can communicate collegially, although they may still encounter exceptions and errors as they perform their duties. Within a system there are several different cases to consider:

- collaborations between objects that interface to the user and the rest of the system;
- collaborations between objects within the system and objects that interface with external systems;
- collaborations between objects outside a neighborhood and objects inside a neighborhood;
- collaborations between objects in different layers;
- collaborations between objects at different abstraction levels,
- collaborations between objects of your design and objects designed by someone else;
- collaborations between your design and objects that come from a vendor-provided library

Who an object receives a request from is a good indicator of how likely is it to accept a request at face value. Who an object calls upon determines how confident it can be that the collaborator will field the request to the best of its ability. It's a matter of trust.

Trusted vs. Untrusted Collaborations

When should collaborators be trusted? Two definitions for collaboration are worth re-examining:

> *Collaborate: 1. To work together, especially in a joint intellectual effort. 2. To cooperate treasonably, as with an enemy occupation force. —the American Heritage Dictionary.*

The first definition is collegial: objects working together towards a common goal. When objects are within the same trust region, their collaborations can be conscientiously designed to be more collegial. Both client and service provider can be designed to assume that if any conditions or values are to be validated; the designated responsible party need only do them once.

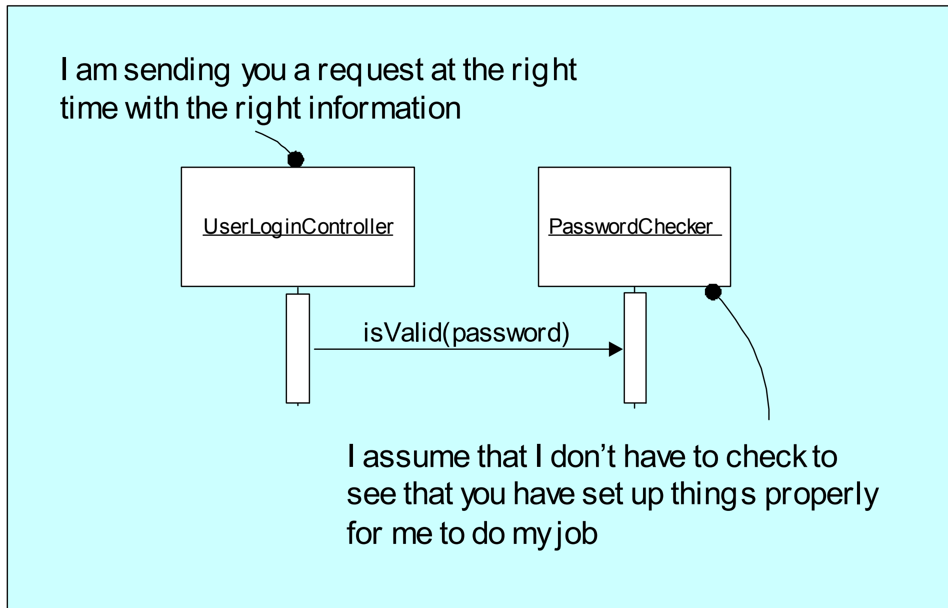


Figure 1. Trust Assumptions

In general when objects are in the same architectural layer or subsystem, they can be more trusting of their collaborators. And they can assume that objects that use their services call upon them appropriately.

The second definition requires you to think critically. When collaborators are designed by someone else, or when they are in a different layer, or a library, your basic assumptions about the appropriate design for that collaboration need to be carefully examined. If a collaborator can't be trusted—it doesn't mean it is inherently more unreliable. But a more defensive collaborative stance may appropriate. A client may need to add extra safeguards—potentially both before and after calling an untrusted service provider.

If a request is from an untrusted or unknown source, extra checks may be made before a request is honored. There are several situations to consider:

- When an object sends a request to a trustworthy colleague
- When an object receives a request from a trusted colleague
- When an object uses an untrusted collaborator
- When an object receives a request from an unknown source
- When an objects receives a request from a known untrustworthy source.

Collaborations between trusted colleagues. A client that provides a well-formed request expects its service provider to carry out that request to the best of its ability. When an object receives a request from a trusted colleague, it typically assumes that the request is correctly formed, that it is sent at an appropriate time, and that data passed along with the request is well formed (unless there is an explicit design decision that the receiver takes responsibility for validating this information).

During a sequence of collaborations between objects within the same trust region there is little need to check on the state of things before and after each request. If an object cannot fulfill its responsibilities and is not designed to recover from exceptional conditions, it could raise an exception or return an error condition enabling its client (or someone else in the collaboration chain) to responsibly handle the problem. But the object may be legitimately to not check. And it

won't even notice when things fail. In a trusted collaboration there is no need to check for invalid collaborations. So if trust is ever violated, things can go terribly wrong.

When using an untrusted collaborator. When collaborators are untrusted, extra precautions may need to be taken. Especially if the client is designed to be responsible for making collaborations more reliable. You may pass along a copy of data instead of sharing it with an untrusted collaborator. Or to check onconditions after the request completes.

When receiving requests from an unknown source. Designers of objects that are used under many different situations—such as those included in a class library or framework— have to balance their objects' expected use (or misuse) with overall reliability goals. There aren't any universal design rules to follow. Library designers must make a lot of hard choices. You can design your object to check and raise exceptions if data and requests are invalid (that's certainly a responsible thing to do, but not always necessary) or not (that's the simplest thing, but not always adequate). Your goal should be to design your framework or library to be consistent and predictable, and to provide enough information so that clients canattempt to react and recover when you raise exceptions.

When receiving requests from an untrusted client. Requests from untrusted sources often are checked for timeliness and relevance. Especially if your goal is to design an object that works reliably in spite of untrustworthy clients. Of course there are degrees of trust and degrees of paranoia! Designing defensive collaborations can be expensive and difficult. In fact, designing every object to collaborate defensively leads to poor performance and potentially introduces errors.

Implications of Trust

Determining "trust regions" for a system is straightforward. And once you determine them, it is easier to decide where to place extra responsibilities for making collaborations more reliable:

> *In the application that enables a disabled user to communicate, all objects within the "core" of the application were designed to work together and are considered to be within the same trust region. Objects in the application control and domain layers all assume trusted communications. Objects at the "edges" of the system —within the user interface and in the technical services layer —are designed to take precautions to make sure outgoing requests are honored and incoming requests are valid. For example, the Selector debounces user eye blinks and only presents single "click" requests. And the MessageBuilder quite reasonably assumes that it receives "trusted" requests from the objects at the edges: the Selector and the Timer. Objects controlled by the MessageBuilder assume they are getting reasonable requests, too. So requests to add themselves to a message, or to offer the next guess are done without questioning the validity of input data or the request. Trusted collaborations within the "core" of the system greatly simplified the implementation of the MessageBuilder, the Dictionaries, the Guesser, the Message, and Letter, Word and Sentence objects' responsibilities.*
>
> *Objects at the "edges" of the system have additional responsibilities for detecting exceptions and trying to recover if they can, or if not, to report them to a higher authority*

(someone at the nurse's station). When a message cannot be reliably delivered, extra effort is made to send an alarm to the nurse's station and raise an audio signal.

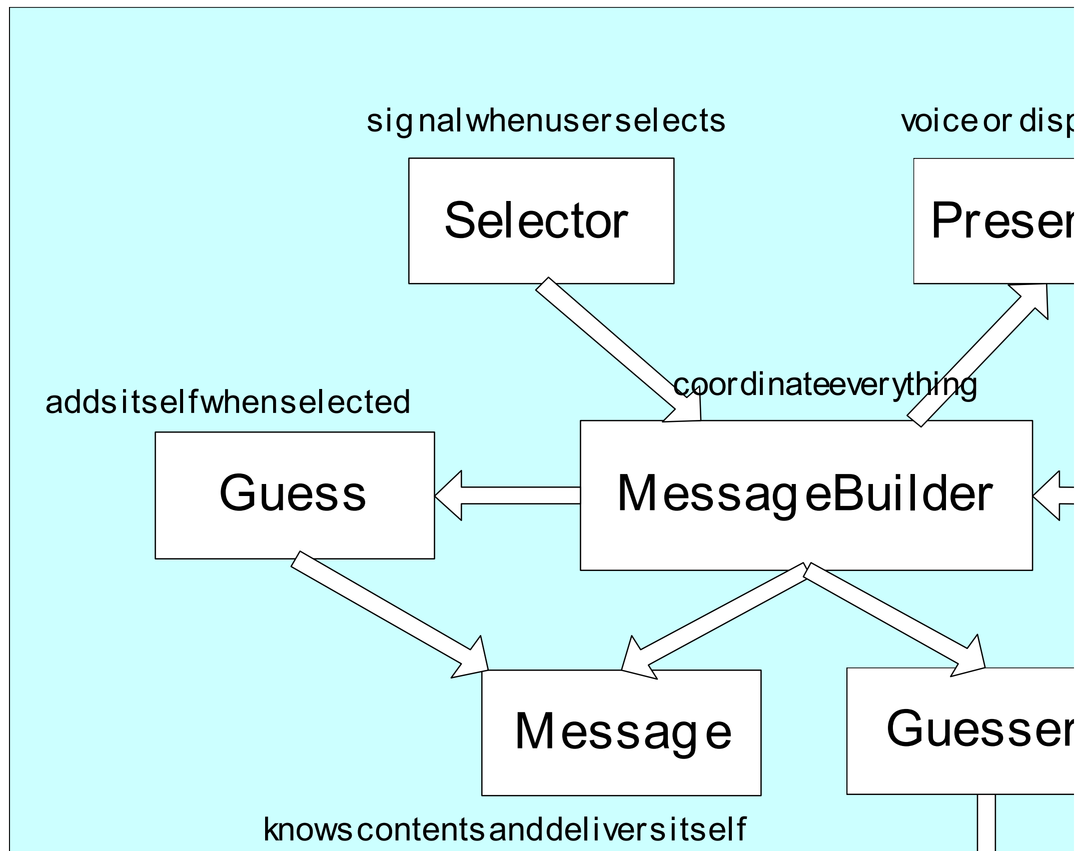


Figure 2. The Selector and the Timer are designed to deliver trusted requests to the Message Builder, allowing it to focus on coordinating the construction of the user's message

In a large system, it is useful to distinguish whether collaborations between components can be trusted, and furthermore, to identify guarantees, obligations and responsibilities of each component. Once these constraints are agreed upon, each component can be designed to do its part to ensure the system as a whole works more reliably.

A telco integration framework receives service order requests and schedules the work to provision services and set up billing. The architecture of the system consists of a number of "adapter" components that interfaced to external applications. Collaborations between an adapter and its "adapted" application were generally assumed to be untrusted, while collaborations between any adapter and core of the system were trusted. The order taking adapter component received requests to create, modify or cancel an order from an external Order Taking application. These requests were converted into an internal format used by the scheduler that was part of the framework integration services. The order taking adapter did not trust the Order Taking application to give it well-formed requests: it assumed that any number of things could be wrong (and they often were). It took extraordinary efforts to guarantee that requests were correctly converted to internal format before it passed them to the scheduler.

Even so, it was still possible to receive requests that were inconsistent with the actual state of an order: for example a request to cancel an order could be received after the work had already been complete. It was business policy not to "cancel" work that had already been completed. So while collaborations between the order-taking adapter and the scheduler were trusted, well-formed requests could still fail.

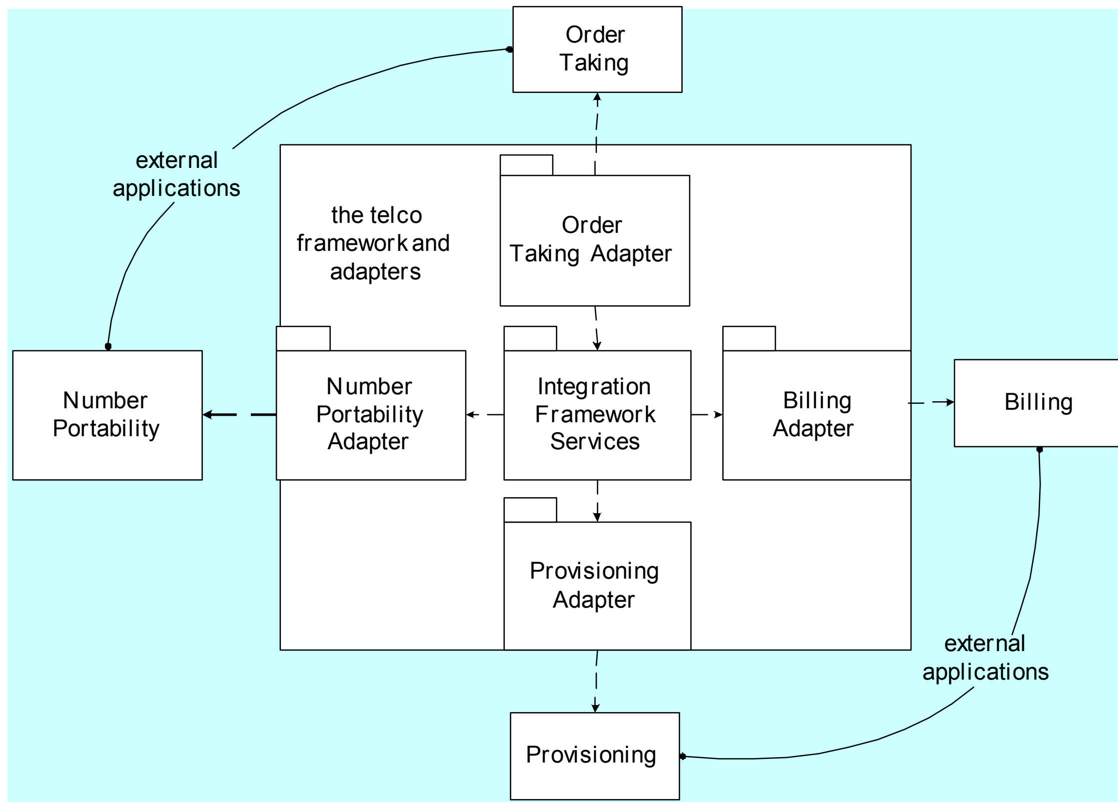


Figure 3. The telco integration framework architecture

Identify Collaborations to Make Reliable

At first, you may not know just exactly what measures to take to increase your system's reliability. First, identify several areas where you want to ensure reliable collaborations. Revisit your initial design and take a stab at improving it. You might consider:

- Collaborations in support of a specific use case or task
- How an object neighborhood responds to a specific request
- How an interfacer handles errors and exceptions encountered in an external system
- How a control center responds to exceptional conditions and errors raised by objects under its control

Once you've identified a particular collaboration to work on, consider what needs to be done. Maybe no additional measures need to be taken—objects are doing exactly what they should be

doing. More likely, you will want to add specific responsibilities to some objects for detecting exceptional conditions and responsibilities to others for reacting and recovering from them. The first step to making any collaboration more reliable is to understand what might go wrong.

Once you've gauged how reliable your software needs to be, consider key collaborations and look for ways to make them more reliable. As you dig deep into design and implementation you will uncover many ways your software might break. But let's get real! While it is up to us designers to decide what appropriate measures to take, to propose solutions, and to work out reasoned compromises, extraordinary measures aren't always necessary.

Will Use Cases Tell Us What Can Go Wrong?

> *"The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair."* —Douglas Adams, *Mostly Harmless (Hitchhiker's Guide Series #5)*

Ideally, some requirements document or use case should spell out the right thing to do when things go wrong. But use cases generally describe software in terms of actors' actions and system responsibilities, not what can go wrong and how to remedy it. At best, use case writers will identify a few problems and briefly describe how some of them should be handled. But even then, use case writers may have been going astray. What someone considers a big problem might not be. Just because someone describes a possible exception and how it should be resolved doesn't mean it will actually happen. Your design may have successfully sidestepped around the potential problem.

But that doesn't relieve you from the responsibility of identifying real problems and resolving them. As you dig into design, you are likely to identify many exception conditions and devise ways of handling them. When your solutions are costly or represent compromises, review them with all who have a stake in your software's overall reliability. They should weigh in on your proposed solutions.

It is easy to waste a lot of time considering things that might go wrong, but won't, or pondering the merit of partial solutions when there is no easy fix. To not get bogged down, distinguish between errors and exceptions. Errors are when things are wrong. Errors can result from malformed data, bad programs or logic errors, or broken hardware. In the face of errors, there is little than can be done to "fix things up" and proceed. Unless your software is required to take extraordinary measures, you shouldn't spend a lot of time designing your software to recover from them.

On the other hand, exceptions aren't normal, but they happen and you should design your software to handle them. This is where the bulk of your energy should go—solving exceptional conditions. If exceptional conditions have been identified for a use case, how they should be accommodated may have been as well:

> *Invalid password entered —After three incorrect attempts, inform the user that access is denied to the online banking system until he contacts a bank agent and is assigned a new password.*

To translate this into an appropriate design solution you'll need to assign some object the responsibility for validating the password; several more are likely to be involved in recovering

from this problem. This is pretty easy—there is nothing difficult or challenging in designing an object to validate a password or report an error condition to the user.

But wait. Is this an error or an exception? Mistyped passwords are a regular if infrequent occurrence. We want our software to react to this condition by giving the user a way to recover, so we view it as an exception, not an error. In fact, most use cases describe exceptions that cause the software to veer off its "normal" path. Some will be handled deftly and the user will be able to continue with their original task. These are *recoverable exceptions*. With others, the user won't be able to complete their original task. The use case will end abnormally, but the application will keep running. From the user's perspective these are *unrecoverable exceptions*. Rarely will use cases mention errors, unless their authors are experienced at describing fault tolerant software
.

Object Exceptions are Different than Use Case Exceptions

Let's get one thing clear. Exceptions described in use cases are fundamentally different than exceptions uncovered in a design. Use case exceptions reflect the inability of an actor or the system to continue on the same course. Object exceptions reflect the inability of an object to perform a requested operation. During execution of a single step in a use case scenario, potentially several use case-level exceptions could happen. However, the execution of a single use case step could result in thousands of requests between collaborators, any number of which could cause numerous different object exceptions. There isn't a one-to-one correspondence between exception conditions described in use cases and object exceptions. Regardless, we need to make our application behave responsibly. We also need to make it reasonably handle the many more exceptional conditions that arise during execution.

Object Exception Basics

An exception condition detected during application execution invariably leads some object or component to veer off its "normal" path and fail to complete an operation. Depending on your design, some object may raise an exception, while another object may handle it. By handling an exception, the system recovers and puts itself into a predictable state. It keeps running reliably even as it veers off the "normal" path—to an expected but "exceptional" one. Left unhandled, however, exceptions can lead to system failure, just as unhandled errors do.

It is up to you to decide what to do when an exception condition is encountered. Many object-oriented programming languages define mechanisms for programmers to declare exceptions and error conditions, signal their occurrence, and to write and associate exception-handling code that executes when signaled. Alternatively, you could design an object to detect an exception condition, and instead of raising an exception, it could return a result indicating that an exception occurred. Partly it's a matter of style and largely a matter of implementation language that determines whether you design your objects to raise exceptions or report exception conditions. Either design described below would "handle the exception condition" of an invalid password.

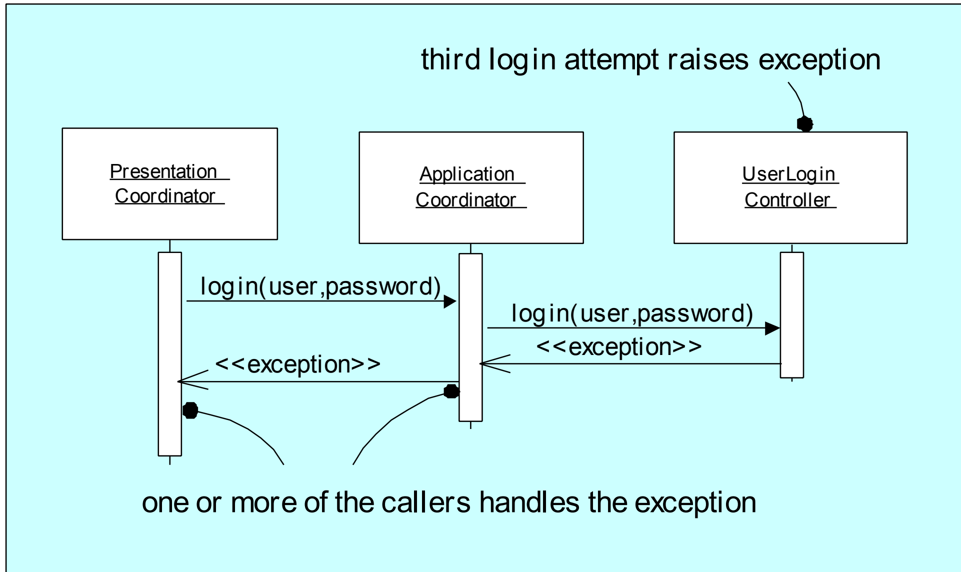


Figure 4. Execution transfers directly to callers' exception handling code

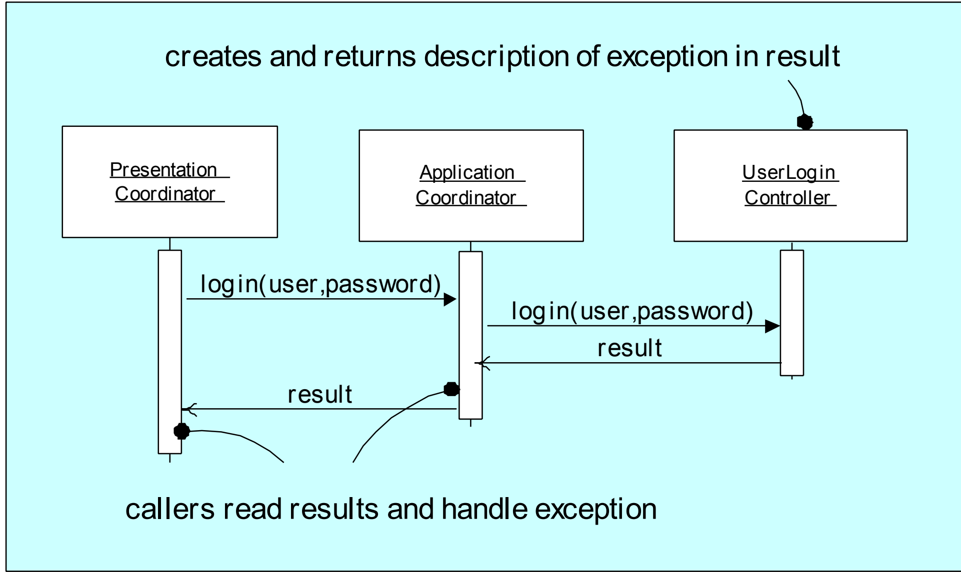


Figure 5. Caller checking a result for exceptions during the call

The first uses exception facilities in the programming language; the second returns values that signify an exceptional condition. Both techniques convey the exceptional condition to the client. Yet another design alternative would to make a service provider smart. It might remember that an exception condition has occurred and provide an interface for querying this fact.

Let's look further at what it means to define and use exception facilities in an object-oriented programming language. When an object detects an exception and signals this condition to its client, it is said to raise an exception. In the Java programming language, the term is "throw an exception." In order to throw a specific exception, a programmer would declare that a particular type of Throwable object (which contains contextual information) to be sent along with the exception signal. An object throws an exception by executing a statement:

```
if (loginAttempts > MAX_ATTEMPTS) {
     throw new LoginAttemptsException();
}
```

The handler of an exception signal has several options. It could fix things up and then transfer control to statements immediately following the code that raised the exception (resumption). Or, it might re-signal the same or a new exception, leaving the responsibility for handling it to a possibly more knowledgeable object (propagation). In most cases, instead of grinding to a halt, it is desirable to make progress. This involves a cooperative effort on behalf of both the object raising the exception, the client sending the exception-causing request, and one or more objects in the collaboration chain if the requestor chooses not to handle the exception then and there.

There must be enough information available to an object that takes responsibility for handling the exception to take a meaningful action. Be aware that when you design an exception object you can declare information that it will hold. The object that detects the exception condition when it creates an exception object populates it with this information.

We offer these general guidelines for declaring and handling exceptions:

Avoid declaring lots of different exception classes . The more classes of exceptions you define, the more cases an exception handler must consider (unless it groups categories of exceptions together). To keep exception handling code simple, define fewer classes of exceptions and, design clients to take different actions based on answers supplied by the exception object.

Deep and wide exception class hierarchies are seldom a good idea. They significantly increase the complexity of a system yet the individual classes are seldom actually used. Compare the complexity of an IOError class hierarchy with twenty subclasses (probably arranged insome sub-hierarchy structure) with one I/O error class that knows an error code with twenty possible values. Most programmers can remember and distinguish 5-7 clearly different exception classes, but if you give them 20-30 exception classes with similar names and subtle distinctions they will never be able to remember them all and will have to continually refer back to the system documentation.

Identify exception classes the same way you identify any other classes— via responsibilities and collaborations. Unless two exceptions will have really distinct responsibilities or participate in different types of collaborations they shouldn't need different classes. Outside the world of exceptions you wouldn't normally create two distinct classes simply to represent two different state values, so why create multiple exception classes simply to represent different values of an error code?

A case where it makes sense to have different exception classes would be for FileIOError and EndOfFile exceptions. Some people might try to treat EndOfFile as a FileIOError but this wouldn’t be a good design choice. FileIOError represents a truly exceptional and unexpected occurrence. Its collaborators are likely to have to take drastic actions. EndOfFile is usually an expected occurrence and its collaborators are likely to respond to it by continuing the normal operations of the program. Seldom, if ever, do you want to respond in the same way to both of these exceptions. But you are quite likely to want to respond in an identical manner to all FileIOErrors.

Name an exception after what went wrong, not who raised it. This makes it easy to associate the situation with the appropriate action to take. The alternative makes it less clear why the handler is a performing specific actions. An exception handler may also need to know who originally raised it (especially if it was delegated upward from a lower-level collaborator), but this can easily be defined to be included as part of the exception object. In this coding example, TooManyLoginAttemptsException explains what happened not who threw it:

```
try {
     loginController.login(userName, password);
}
catch (TooManyLoginAttemptsException(e)) {
     // handle too many login attempts
}
```

Recast lower-level exceptions to higher-level ones whenever you raise your abstraction level. When very low-level exceptions percolate up to a high-level handler, there is little context for the handler to make informed decisions. Recast an exception whenever you cross from one level of abstraction to another. This enables exception handlers that are way up a collaboration chain to make more informed decisions and reports. Not taking this advice can lead your users to believe that your software is broken, instead of just dealing with unrecoverable errors:

> *A compiler can run out of disk space during compilation. There isn't much the compiler can do in this case except report this condition to the user. But it is far better for the compiler to report "insufficient disk space to continue compilation" than to report "I/O error #xxx". With the latter message, the user may be led to believe there is a bug in the compiler, rather than insufficient resources which could be corrected by the user. If this low-level exception were to percolate up to objects that don't know to interpret this I/O error exception, it will be hard to present a meaningful error message. To prevent this, the compiler designers recast low-level exceptions to higher-level ones whenever subsystem boundaries were crossed.*

Provide context along with an exception. What are most important to the exception handler are what the exception is and any information that aids it in making a more informed response. This leads to designing exception objects that are rich information holders. Specific information can be passed along including: values of parameters that caused the exception to be raised, detailed descriptions, error text, and information that could be used to take corrective action. Some designers, when recasting exceptions, embed lower level exceptions as well, providing a complete trace of what went wrong.

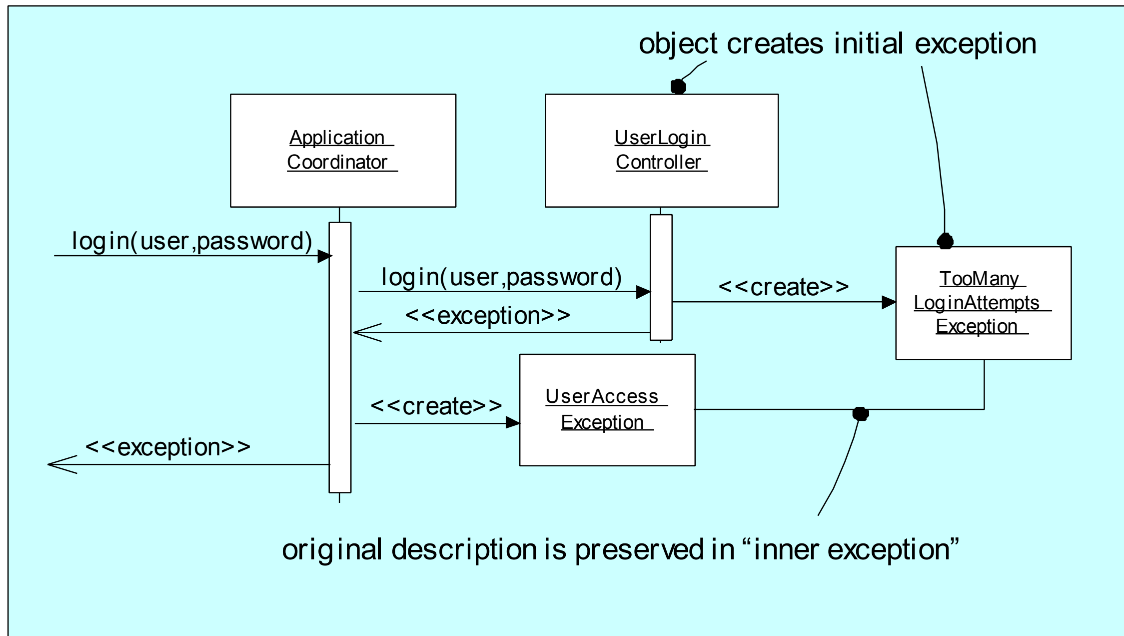


Figure 6. Preserving information in "inner exceptions"

Preserve information in "inner exceptions" Assign exception-handling responsibilities to objects that can make decisions. There are many different ways to "handle" an exception: it could be logged and rethrown (possibly more than once), until someone takes corrective action. Who naturally might handle exceptions? As a first line of defense, consider the initial requestor as the first line of defense. If it knows enough to perform corrective action, then the exception can be taken care of right away and not be propagated. As a fallback position, it is always appropriate to pass the buck to some object that takes responsibility for making decisions and controlling the action. Controllers and objects located within a control center are naturals for handling exceptions.

Handle exceptions as close to the problem as you can. One object raises an exception, and somewhere up the collaboration chain another handles it. Sure this works, but it makes your design harder to understand. It can make it difficult to follow the action if you carry this to extremes.

Objects that interface to other systems and components often take responsibility for handling faulty conditions in other systems they interface to, relieving their clients of having to know about lower-level details and recovery strategies. Objects that play a role of providing a service often take on added responsibility to handle an exception and retry an alternative means of accomplishing the request.

Consider returning results instead of raising exceptions. Instead of raising exceptions, you always can design your exception-taking object to return a result or status that is directly checked by the requestor. This makes it more obvious who's got to take at least some responsibility—the requestor.

Exception and Error Handling Strategies

In the case of errors as well as exceptions, it is a matter of how much effort and energy you want to expend handling them. Highly-fault tolerant systems are designed to respond to take extraordinary measures. A highly fault tolerant system might recover from programming errors by running an alternate algorithm, or from a disk suddenly becoming inaccessible by printing data on an alternate logging device. Most ordinary software would break (gracefully or not, depending again, on the design and the specific condition).

There are numerous ways to deal with a request that an object can't handle. Doug Lea, in *Concurrent Programming in Java*, poses the question "What would you do if you were asked to write down a phone number and you didn't have a pencil?" to explore several options. One possibility, is what Lea calls unconditional action. In this simple scheme, you'd go throughthe motions of writing as if you had a pencil, whether you did or not. Besides looking silly, this is only acceptable if nobody cares that you fail to complete your task.

Employing this strategy often leads to unpredictable results. In real life, you likely wouldn't be so irresponsibility, and your software objects shouldn't behave this way either. If an object or component or system that receives a request isn't in the proper state to handle it, nothing can be guaranteed. An unconditional act could cause the software to trip up immediately, or worse yet, to fail later in unpredictable ways. Ouch! There are more acceptable alternatives:

- Inaction—Ignore the request after determining it cannot be correctly performed.
- Balk—Admit failure and return an indicationto the requestor (by either raising an exception or reporting an error condition).
- Guarded suspension—Suspend execution until conditions for correct execution are established, then try to perform the request.
- Provisional action—Pretend to perform the request, but do not commit to it until success is guaranteed.
- Recovery—Perform an acceptable alternative.
- Appeal to a higher authority—Ask a human to apply judgment and steer the software to an acceptable resolution.
- Rollback—Try to proceed, but on failure, undo the effects of a failed action.
- Retry—Repeatedly attempt a failed action after recovering from failed attempts.

These strategies impact the designs of both clients as well as objects fulfilling requests, and, possibly, other participants in recovery activities. No one strategy is appropriate in every situation. Inaction is simple but leaves the client uninformed. When an object balks, at least the requestor knows about the failure and could try an alternative strategy. With guarded suspension, the object would patiently wait until some other object gave it a pencil (the means by which someone knows what is needed and supplies it is unspecified).

Provisional action isn't meaningful in this example, but it makes sense when a request takes time and can be partially fulfilled in anticipation of it later completing it. Recovery could be as simple as using an alternate resource—a pen instead of a pencil. Appealing to a higher authority might mean asking some human who always keeps pencils handy and sharp to write down the number instead. Rollback doesn't make much sense in this example, since nothing has been partially done—unless the pencil breaks in the middle of writing down the number. In this case the object would throw away the partially written number. Rollback is a common strategy where either all or nothing is desired and partial results are unacceptable. Retrying makes sense only when there is a chance of success in the future.

To sum up, there will always be consequences to consider when choosing any recovery strategy:

> *"The designer or his client has to choose to what degree and where there shall be failure. Thus the shape of all designed things is the product of arbitrary choice. If you vary the terms of your compromise...then you vary the shape of the thing designed. It is quite impossible for any design to be 'the logical outcome of the requirements' simply because the requirements being in conflict, their logical outcome is an impossibility."* —David Pye

Mixing or combining strategies often leads to more satisfactory results. For example, one object could attempt to write down the phone number but broadcast a request for a pencil if it fails to locate one. It might then wait for a certain amount of time. But if no one provided it with one, ultimately it might ignore the request. Meanwhile, the requestor might wait awhile for confirmation, and then locate another to write the phone number after waiting a predetermined period of time. The best strategy isn't always obvious or satisfying. Compromises don't always feel like reasonable solutions—even if they are the best you can do under the circumstances.

Design a solution

So far, we've considered strategies for handling failures for a single request. Making larger responsibilities more reliable can get much more complex. Once you've identified a particular part of your design that you want to make more reliable, think through all the cases that might cause objects to veer off course. Start simply, then work up to more challenging problems. Given the nature of design, not all acceptable solutions may seem reasonable at first. You may need time for a solution to "soak in" before it seems right.

Brainstorm Exception Conditions

Complex software can fail in many, many ways. Even simple software can have many places where things could go wrong. Thinking through all the ways software might fail is difficult work. Make a list. Enumerate all the exceptional conditions you can think of for a specific chunk of system behavior. Whether you are working on your design in support of a use case, or designing some collaboration deep inside your system, list everything that you reasonably expect could go wrong. Consider:

- Users behaving incorrectly—entering misinformation or failing to respond within a particular time
- Invalid information
- Unauthorized requests
- Invalid requests
- Untimely requests
- Time out waiting for a response
- Dropped communications
- Failures due to broken or jammed equipment, such as a printer being unavailable

- Errors in data your software uses including corruptlog files, bad or inconsistent data, missing files
- Critical performance failures or failure to accomplish some action within a prescribed time limit

This list is intended to jog your thinking. But be reasonable. If some condition seems highly improbable...leave it off your list. Put it on another list (the list of exceptions you didn't design for). If you know that certain exceptions are common, say so. If you don't know whether an exception might occur, put a question mark by it. You may not know what are reasonable and expected conditions if you are building something for the first time. People and software and physical resources can cause exceptions. And the deeper you get into design and implementation, the more exceptions you'll find.

Limit Your Scope: Pick a Likely Exception and Resolve It. Take exception design in bite-sized increments. If you've already designed your objects to collaborate under normal conditions, start modestly to make it more reliable. Pick a single exception that everyone agrees is common enough and you think you know how it should be handled. If you are designing collaborations for a specific use case, tackle one "unhappy path" situation. What actions should occur when there are insufficient funds when making an online payment?What if the user blinks her eyes too rapidly and makes a false selection? What if the file is locked by another application?

After you've decided on what seems a reasonable way to handle that situation, design a solution using the object-oriented design techniques we've described. Minimize or purposefully ignore certain parts of your design in order to concentrate on those objects who will take exception, and those who will resolve it. You needn't reach all the way from the user interface to the lowest technical service objects. Here is what we consider to be both in and out of scope for the exceptional case of insufficient funds:

Make A Payment- Insufficient Funds
Assume a well-formed request (no data entry errors)
Ignore backend system bottlenecks
Ignore momentary loss of connections or communication failures (they will be handled by connection objects in the technical service layer)
Offer the user an opportunity to enter an alternate amount

Determine who should detect an exception and how it should be resolved. Assume that everything goes according to plan up to the point of where the particular exception you are considering is detected.

We know the existing backend banking system returns an error code indicating insufficient funds to our external interface component. Now what?
The backend banking component reports the exception via a Result object to the FundsTransfer object that is responsible for coordinating the transaction. The FundsTransfer interprets this as an "unrecoverable exception" which ca uses it to halt and return a Result (indicating failure) to the User Session.

Describe additional responsibilities of collaborators. Objects that are service providers, controllers and coordinators are often charged with exception handling responsibilities.

In the online banking application, the FundsTransferTransaction—a service-provider/coordinator—coordinates the work of performing a financial transaction. It makes relatively few decisions, only altering its course when the result is in error. It is responsible for validating funds transfer information, forwarding the request to the backend banking interface component, logging successful transaction, and reporting results.

Objects within the application server component are within the same trust region.They receive untrusted requests from the UI component and collaborate with the backend banking component (each of those collaborations span another trust boundary). The backend-banking component interfaces to the backend banking system, a trusted external system that either handles the request or reports an error. Occasionally, communications between the backend bank system fail, and then our software must take extraordinary measures.

Objects at the edges of a trust region can either take responsibility forguaranteeing that incoming requests are well formed, or they can delegate all or part of that responsibility.In the online banking application, any incoming request from the user component is validated. The UserSession object receives and validates requests from the UI component, then creates and delegates the request to specific service providers. When a request to transfer funds is received from the UI component, a FundsTransferTransaction is created. It has responsibility for validating the funds transfer information and reacting to errors reported from the backend system.

As you work through exception handling scenarios assigning additional responsibilities to collaborators, make sure you consider:

- Who validates information received from untrusted collaborators
- Who detects exceptions
- How exceptions are communicated between collaborators (via raised exceptions or error results)
- Who recovers from them
- How recovery is accomplished
- Who recovers from failed attempts at recovery
- Who recasts exceptions, or translates them to higher levels of abstraction

Record Exception Handling Policies

Once you've decided how to solve one exceptional condition, tackle another. Often you can leverage earlier work. If you decide that "these type of exceptions" are very similar to "those" ones, you'll likely want to handle them consistently. Write down general strategies you will attempt to follow. Deciding on exception handling policies can save a lot of work:

> ***System Exception Policies***
> ***Recoverable software exceptions.*** *These are caught exceptions that do not necessarily mean an unstable state in the software (corrupt message, time outs, etc.). The strategy to be followed in these cases is to first log the exception and then try to handle it (if retrying is likely to succeed). If not, raise the exception so it can be handled (if the caller is within the same process); or to return an error (if the caller is not within the same process).*
> ***Unrecoverable software exceptions.*** *These are caught exceptions that presumably can lead to an unstable state, like running out of memory or a task being unresponsive. The*

response in these cases is to log the cause of the exception and to restart the application unless the severity there is a "hold&do not restart" indication for that specific condition.

Document Your Exception Handling Designs

You will likely want to beef up existing design documentation with exception handling details. But don't pile on details. You can easily make a collaboration story incomprehensible or a diagram illegible obscuring the main storyline. Instead, draw new diagrams to show how specific exceptions are handled. Leave existing diagrams alone. Any new diagram will look nearly identical to the "normal" case, but will include additional details about how an exception is detected, communicated and dealt with.

Your stakeholders and fellow designers will get a much better sense of your exception design if you explain it. Describe what exceptions you considered, how each is resolved, and what you consider to be out of scope:

> *The online banking application is designed to cover communications failures encountered during a financial transaction. A full set of single -point failures was considered. Some double-point failures were explicitly not considered, as they are both unlikely and covering them adds undue complexity to the processing of transactions. In each case, the general strategy is to ensure that transaction status is accurately reflected to the user. Failures in validating information will cause the transaction to fail, whereas intermittent communications to the external database or to the backend banking system during the transaction will not cause a transaction to fail.*

In our opinion a picture isn't worth a thousand words and a thousand words doesn't always cut it either. If you can find a way to explain concepts and design strategies using a combination of visual and textual information, you'll be a more effective communicator. Here is an example showing key components and objects involved in performing a "prototypical" online banking transaction. A table that explains what exceptions can occur and their impacts on the user, accompanies it. Once this multi-media explanation was created, how the software was designed to react to exceptional conditions was easily communicated.

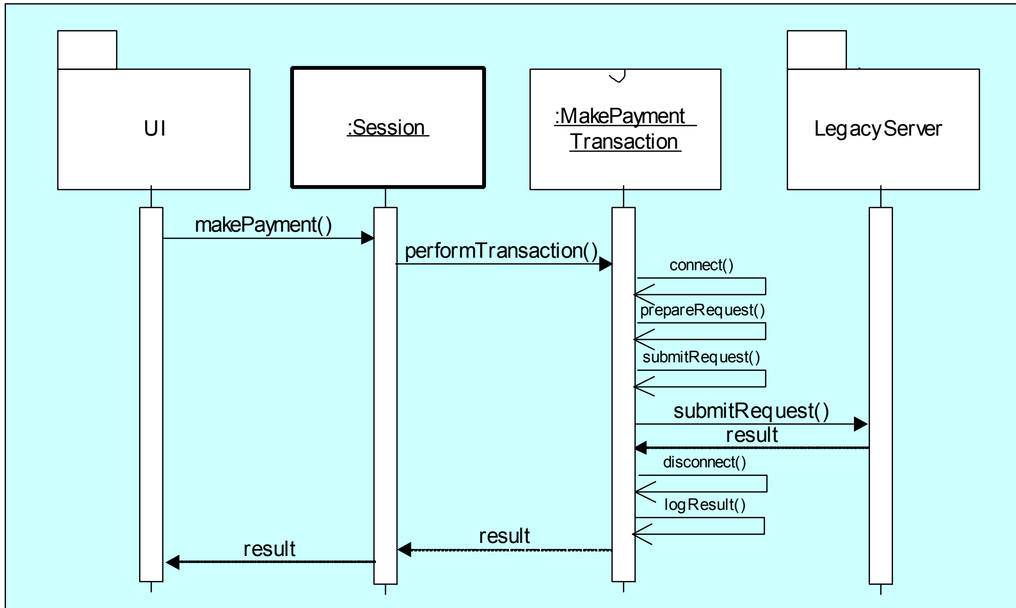


Figure 8. A "high-level" sequence diagram showing a typical banking transaction

Exception or Error	Recovery Action	Affect on User
Connection is dropped between UI and Domain Server after transaction request is issued	Transaction continues to completion. Instead of notifying user of status, transaction is just logged. User will be notified of recent (unviewed) transaction results on next login.	User session is terminated… user could've caused this by closing his or her browser, or the system could have failed. User will be notified of transaction status the next time they access the system
Failure to write results of successful transaction to domain server log	Administrator is alerted via console and email alerts. Transaction information is temporarily logged to alternative source. If connections cannot be re-established, the system restricts users to "read only" and account maintenance requests until transaction logging is re-established	User can see an unlogged transaction in transaction history constructed from backend banking query… but won't have it embellished with any notes he or she may have entered
Connection dropped between domain server and backend bank access layer after request is issued	Attempt to re-establish connection. If this fails after a configurable number of retries, transaction results are logged as "pending" and the user is informed that the system is momentarily unavailable…check in later. When connections are re-established, status is acquired and logged. Further logins are prevented until backend access is re-established	User will be logged off with a notice that system is temporarily unavailable and will learn of transaction status on next login
Backend banking request fails	Error condition reported to user. Transaction fails. Failed transaction is logged	User receives error notification but can continue using online services

Table 1. A table that explains online banking transaction exceptions and their impacts on the system and its users

Review Your Design for Holes

Even with best intentions, you just can't spot all the flaws in your work. Have you ever had that "Aha! moment," explaining something to someone else. Simply talking about your design with someone else helps you see things clearly. A fresh perspective will help spot gaps in your design. The most common bugs in exception handling design, according to Charles Howell and Gary Veccellio, who analyzed several highly reliable systems, crop up when:

- failing to consider additional exceptions that might arise when writing exception handling logic. Don't let your guard down! Any action performed when handling an exception could cause other exceptions. Often the appropriate solution to this situation is to raise new exceptions from within the exception handling code.
- mapping error codes to exceptions. At different locations in your design, various objects may have the responsibility to translate between specific return code values to specific exceptions. The most common source of error is to incompletely consider the range of error codes—mapping some, and not all cases. Mapping is often required when different parts of a system are implemented in different programming languages.
- propagating exceptions to unprepared clients. Unhandled exceptions will continue to propagate up the collaboration chain until either they are handled by some catchall object, or left to the run time environment. Designers usually want some graceful exception reporting or recovery. What they'll get instead, will be program termination, if clients aren't designed to handle an unexpected exception.
- thinking an exception has been handled when it has merely been logged. Exception code should do something meaningful to get the software back on track. As a first cut, you may implement a common mechanism to log or report an exception. But this doesn't mean it has been handled. You've done nothing but report the problem—which is only slightly more useful than taking no action at all.

In addition to these potential sources of error, look for places where complexity may have sneaked in:

- redundant validation responsibilities. When you aren't certain who should take responsibility, sometimes you put it in several places. There may be different levels of validation performed by different objects in a collaboration—first checking that the information is in the right format, next checking that it is consistent with other information. It is OK to spread these responsibilities between collaborators. But avoid two different objects performing identical semantic checks.
- unnecessary checks. If you aren't sure whether some condition should be checked, why not check anyway? Because it can decrease system performance and give you a false sense of security. This is an easy trap to fall into. By doing this, you've done absolutely nothing to increase your software's reliability and are likely to confuse those who will maintain your design.
- embellished recovery actions. Extra measures at first seem like a good idea... but wait. Is it really necessary to retry a failed operation,log it, and send email to the system administrator? Look for where extra measures detract from system performance, make your system more complex... and on a really bad day could clog up someone's inbox.

At the end of a review, you should be convinced that your exception handling actions are reasonable, cost effective and are likely make a difference in your system's reliability.

Summary

As a first step in increasing reliability, you need to understand the consequences of system failure. The more critical the consequences, the more effort and energy is justified designing for reliability. To clarify your thinking, distinguish between exceptions—unlikely conditions that your software must handle—and errors. Errors are when things are wrong—bad data, programming errors, logic errors, faulty hardware, broken devices. Most software doesn't need to be designed to recover from errors, but can be made more reliable by gracefully handling common exceptional conditions.

Approaches for improving reliability are rarely cut and dried. The best alternative isn't always clear. To decide what appropriate reactions should be taken involves sound engineering as well as consideration of costs and impacts on the system's users.

Objects do not work in isolation. To improve system reliability you must improve how objects work in collaboration. Collaborations can be analyzed for the degree of trust between collaborators. Within the same trust boundary, objects can assume that exceptions will be detected and reported, and that responsibilities for checking on conditions and information will be carried out by the appropriately designated responsible party. In some programming languages, exceptions can be declared. When an exception is raised, some other object in the collaboration chain will take responsibility for handling it. An alternative implementation technique is to return values from calls that can encode exceptional conditions.

When collaborations span trust boundaries, more precautions may need to be taken. Defensive collaborations—designing objects to take precautions before and after calling on a collaborator—are expensive and error prone. Every object shouldn't be tasked with these responsibilities. When you need to be very precise, define contracts between collaborators. Bertrand Meyer uses contracts to specify the obligations and benefits of the client and provider of a service. Spelling out these terms makes it absolutely clear what each object's responsibilities are in a given collaboration.

Further reading

Doug Lea has written a very handy book called *Concurrent Programming in Java: Design Principles and Patterns*. This book is invaluable, to even non Java programmers. It is packed with in depth discussions and examples and good design principles. Even if you aren't building highly concurrent applications, this book is worth careful study.

Advances in Exception Handling Techniques grew out of a workshop on exception handling for the 21st century. It is a collection of chapters written by programming language researchers, database designers, distributed system designers and developers of complex applications and mission critical systems who share their vision of the current state of the art of exception handling and design. You will find very readable papers that discuss exceptions from multiple perspectives.

Bertrand Meyer's book *Object-Oriented Software Construction (Second Edition)* is the definitive work on software engineering using the principle of Design by Contract. It is a weighty book. But the two chapters, Design by contract: building reliable software, and When

the contract is Broken: exception handling, are a good exposure to the thinking in terms of preconditions, postconditions, invariants and collaboration contracts.

Henry Petroski talks about the role of failure analysis in successful design in *To Engineer is Human*. Software designers clearly don't understand the laws that govern software failures as well as structural engineers understand physics and materials. But you can learn many lessons from this book.

References

Douglas Adams, *Mostly Harmless (Hitchhiker's Guide Series #5)*, Random House, 1993

Henry Petroski, *To Engineer is Human*, Vintage Books, 1992

David Pye, *The Nature and Aesthetics of Design*, Van Nostrand Reinhold Company, 1978

Alistair Cockburn, *Agile Software Development*, Addison-Wesley, 2002

Alexander Romanovsky, Christophe Dony, Jorgen Lindskov Knudsen, Anand Tripathi, Eds., *Advances in Exception Handling Techniques*, Springer-Verlag, 2001

Doug Lea, *Concurrent Programming in Java (tm) Second Edition: Design Principles and Patterns*, Addison-Wesley, 2000

Bertrand Meyer, *Object-Oriented Software Construction,* Prentice Hall, 1997

Charles Howell and Gary Veccellio, "Experiences with Error Handling in Critical Systems" in *Advances in Exception Handling Techniques*, Alexander Romanovsky, Christophe Dony, Jorgen Lindskov Knudsen, Anand Tripathi, Eds., Springer-Verlag, 2001