

PROCEEDINGS forUSE 2002

**First International Conference
on Usage-Centered, Task-Centered,
and Performance-Centered Design**

**25-28 August 2002
Portsmouth, New Hampshire**

Larry L. Constantine, Editor

forUSE2002



Ampersand Press

An imprint of Constantine & Lockwood, Ltd.
Rowley, Massachusetts

PROCEEDINGS

forUSE 2002, First International Conference on

Usage-Centered, Task-Centered, and Performance-Centered Design

Edited by Larry Constantine

Conference Organizers: Constantine & Lockwood, Ltd., <http://www.foruse.com>

Conference Chairpersons: Larry Constantine and Lucy Lockwood,
Constantine & Lockwood, Ltd.

Program Chairperson: James Noble, Victoria University of Wellington, New Zealand

Industry Liaison: Helmut Windl, Siemens AG, Germany

Professional Affiliate: Usability Professionals' Association, <http://www.upassoc.org>

Silver Sponsor: Classic Systems Solutions, Inc., <http://www.classicsys.com>

Bronze Sponsor: Ariel Performance Centered Systems, Inc., <http://www.arielpcs.com>



Published by Ampersand Press
Constantine & Lockwood, Ltd.
58 Kathleen Circle
Rowley, Massachusetts 01969
<http://www.foruse.com>

Copyright © 2002 by Constantine & Lockwood, Ltd.

All world rights reserved. No portion of this publication may be reproduced, stored for retrieval, or transmitted, in any form or by any means, electronic, mechanical, photographic, or otherwise, without the prior written consent of the publisher.

Printed in the United States of America by
The Yankee Printer
Hampton Falls, New Hampshire 03844
<http://www.yankeeprinter.com>

First printing, August 2002

[T33]

What It Really Takes to Handle Exceptional Conditions

Rebecca Wirfs-Brock

Henry Petroski, structural engineer and historian, writes 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?'" (Petroski, 1992)

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 should not 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 the items a customer wants are not 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!” will not 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, which should handle many exceptional conditions and keep running without involving the user at all.

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* (Cockburn, 2002), 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 does not 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 should not 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" would not work.
- Consumer products need to work, period. Their success in the marketplace depends on high reliability.

A Strategy for Increasing System Reliability

Reliability concerns crop up throughout development. But once you have 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.

I 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 have characterized your system's patterns of collaborations and prioritized your work, you need to get very specific:

- List the exceptions and error 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.

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

Trusted versus 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.

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 cannot be trusted—it does not 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.

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.

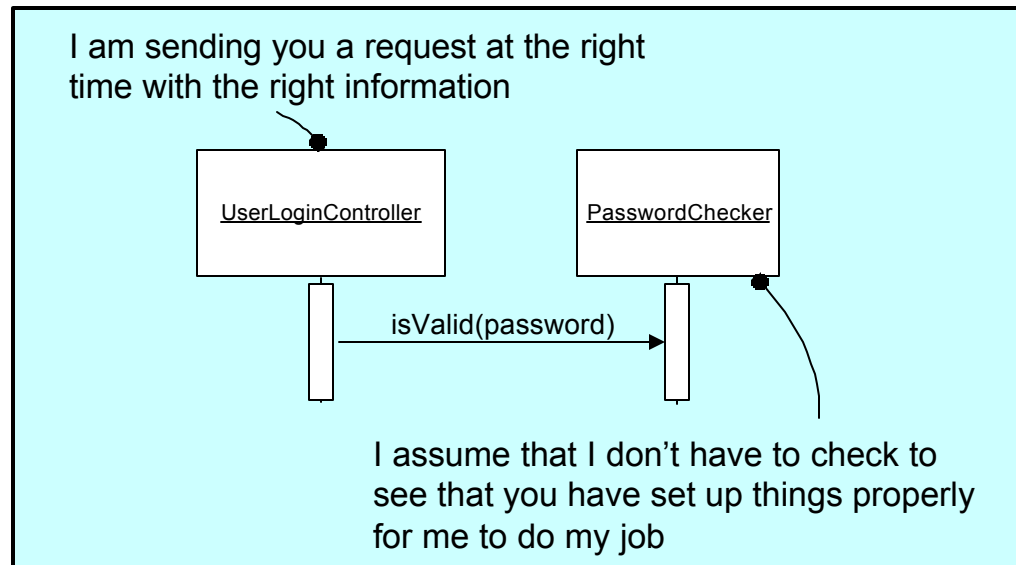


Figure 1 - Trust assumptions.

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 cannot be trusted—it does not mean it is inherently more unreliable. But a more defensive collaborative stance may be 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. Several situations need to be considered:

- 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 object 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. However, the object may legitimately not check and will not 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 on conditions 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 are not 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 is certainly a responsible thing to do, but not always necessary) or not (that is 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 can attempt 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.

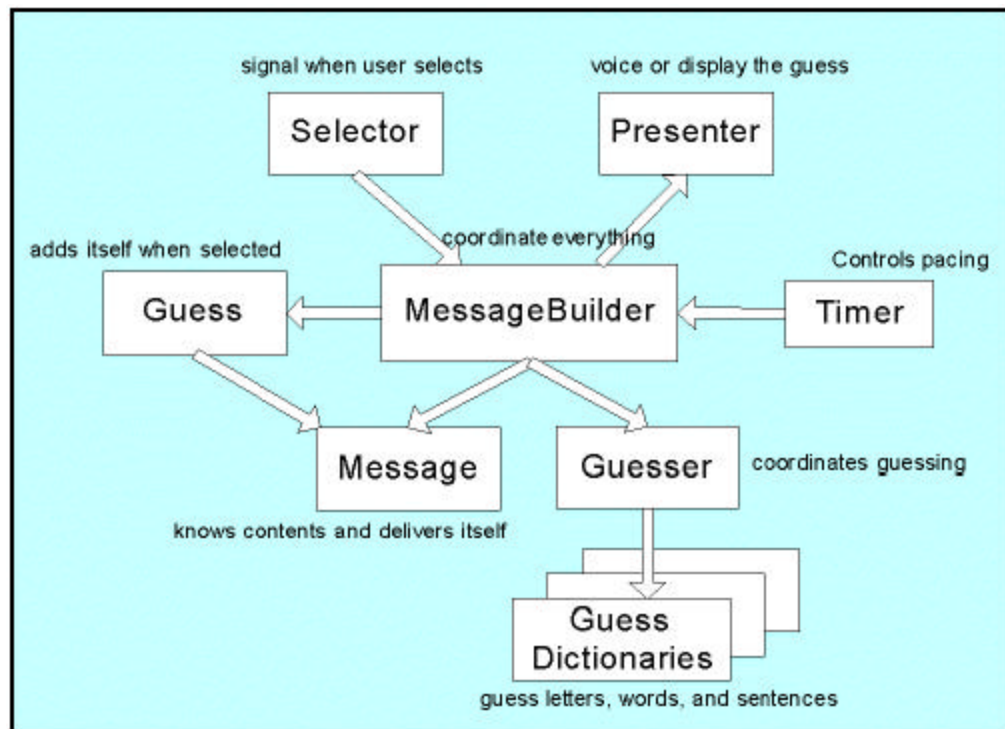


Figure 2 - The Selector and the Timer are designed to deliver trusted requests to the MessageBuilder, allowing it to focus on coordinating the construction of the user's message.

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.

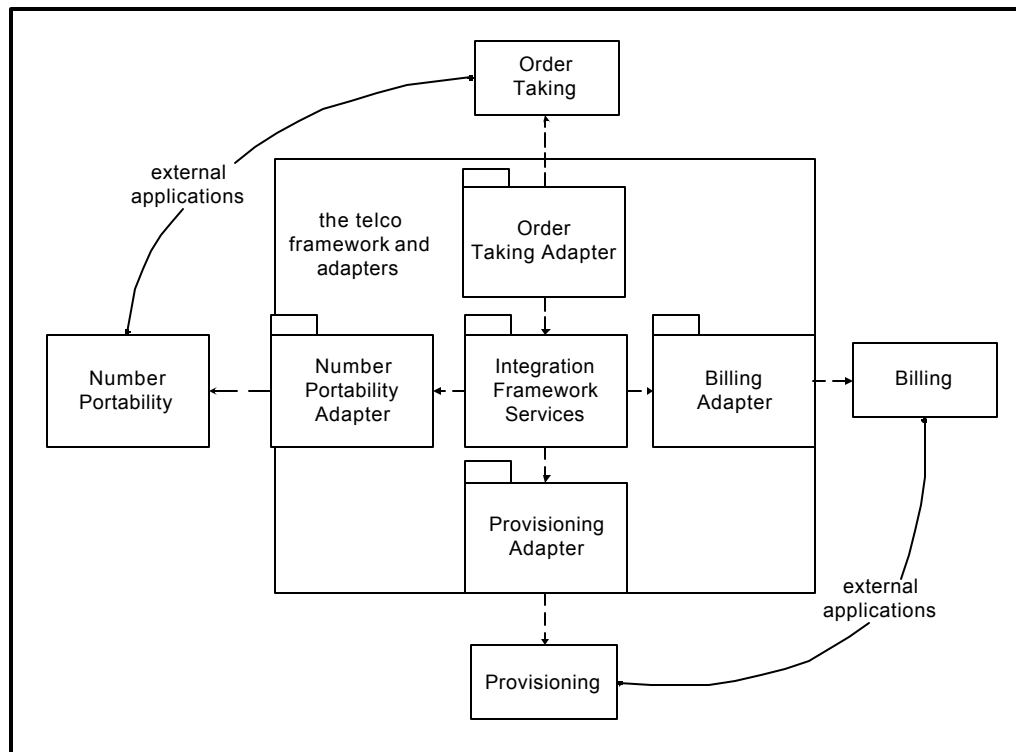


Figure 3 - The telco integration framework architecture.

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 have 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 have 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. However, while it is up to us as designers to decide what appropriate measures to take, to propose solutions, and to work out reasoned compromises, extraordinary measures are not 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* (1993).

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.

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 does not mean it will actually happen. Your design may have successfully sidestepped around the potential problem.

But that does not 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 will not or pondering the merit of partial solutions when there is no easy fix. To avoid getting bogged down, you need to 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 should not spend a lot of time designing your software to recover from them.

On the other hand, exceptions are not 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 will 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 will not be able to complete the 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

One thing must be made 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 is not 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 is 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 shown in Figures 4 and 5 would “handle the exception condition” of an invalid password.

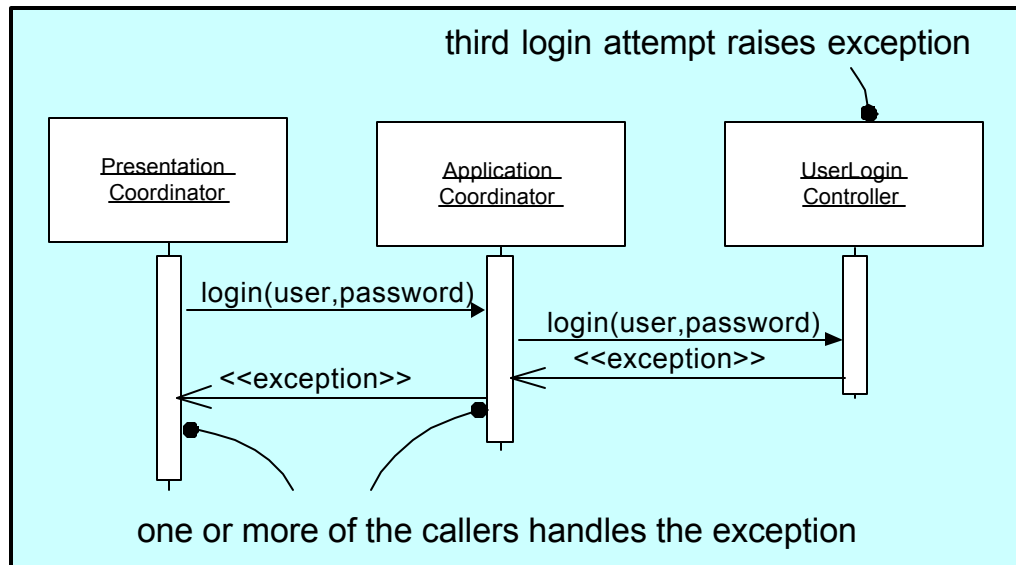


Figure 4 - Execution transfers directly to callers' exception handling code.

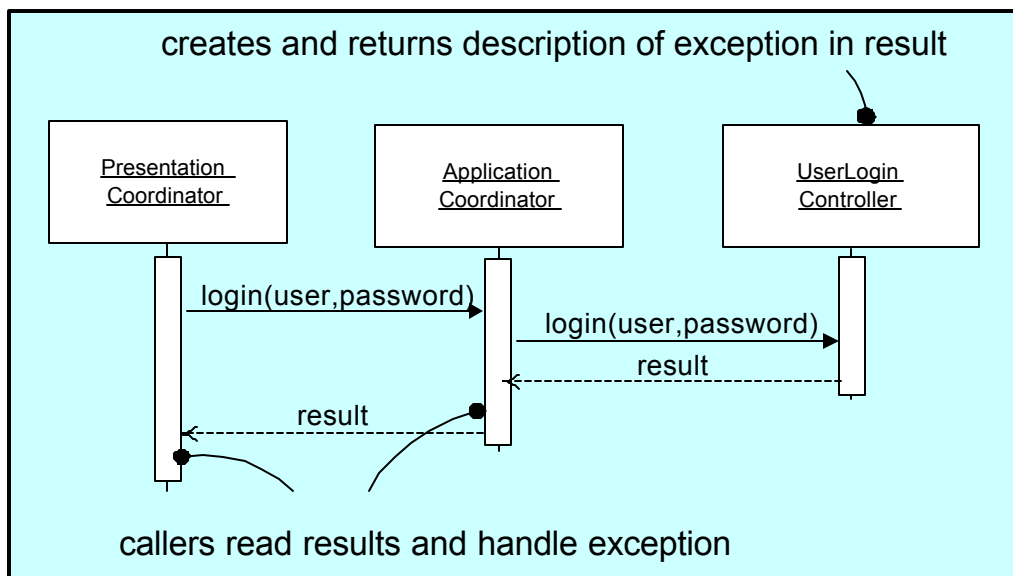



Figure 5 - Caller checking a result for exceptions during the call.



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 be 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 in some 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 should not need different classes. Outside the world of exceptions you would not 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 would not 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 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 is not 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 do not 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.

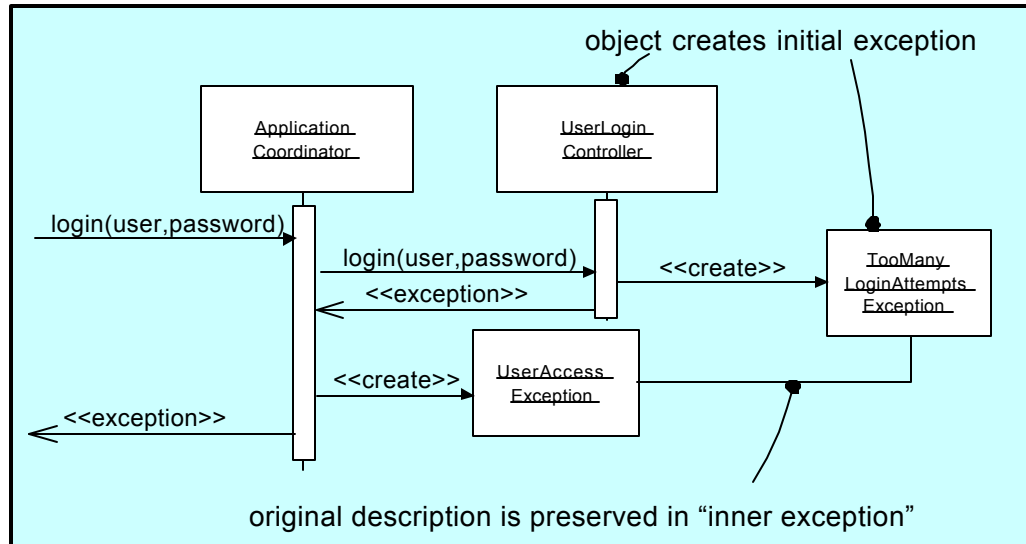


Figure 6 - Preserving information in "inner exceptions"


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 must 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 cannot handle. To explore several options, Doug Lea (2000) poses the question, “What would you do if you were asked to write down a phone number and you didn’t have a pencil?” One possibility is what Lea calls unconditional action. In this simple scheme, you would go through the 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 would not be so irresponsible, and your software objects should not behave this way either. If an object or component or system that receives a request is not 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 indication to 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 is not 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 does not 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 (1978)



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 is not always obvious or satisfying. Compromises do not always feel like reasonable solutions—even if they are the best you can do under the circumstances.

Design a Solution

So far, we have considered strategies for handling failures for a single request. Making larger responsibilities more reliable can get much more complex. Once you have 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 corrupt log 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 did not design for). If you know that certain exceptions are common, say so. If you do not 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 will find.

Limit your scope: pick a likely exception and resolve it.

Take exception design in bite-sized increments. If you have 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 have decided on what seems a reasonable way to handle that situation, design a solution using the object-oriented design techniques already described. Minimize or purposefully ignore certain parts of your design in order to concentrate on those objects that will take exception, and those that will resolve it. You need not 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 back-end 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 back-end 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 causes 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 for guaranteeing 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 have 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 will 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 do not 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 is not necessarily worth a thousand words and a thousand words does not 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 will 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.

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

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 have 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 will not 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.
Bac-kend banking request fails.	Error condition reported to user. Transaction fails. Failed transaction is logged.	User receives error notification but can continue using online services.

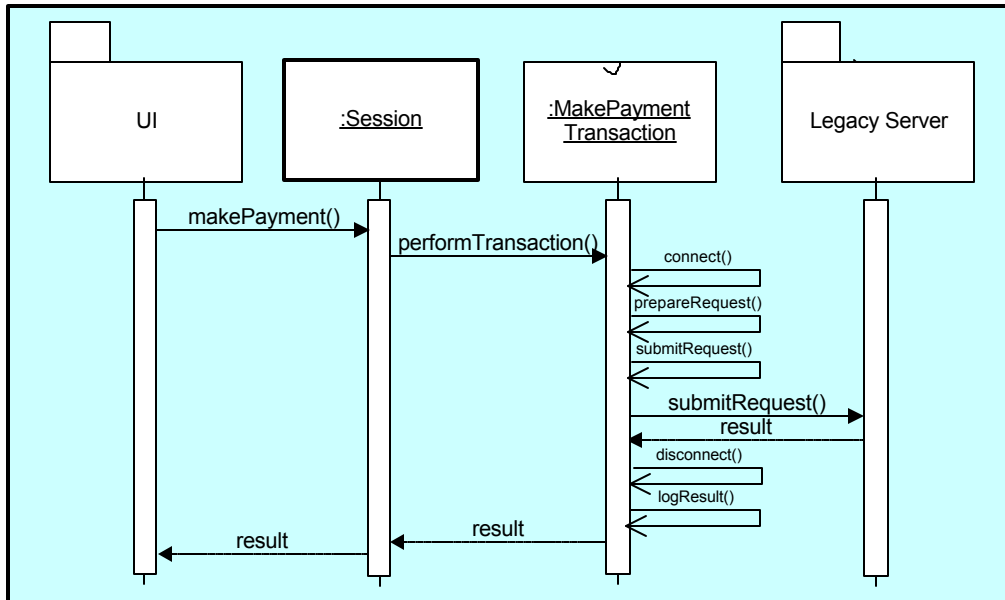


Figure 8 - A “high-level” sequence diagram showing a typical banking transaction.

Review Your Design for Holes

Even with best intentions, you cannot spot all the flaws in your work. Have you ever had that “Aha!” about your own mistake while 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. Do not 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. If clients are not designed to handle an unexpected exception, they will get program termination instead.
- 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. However, this does not mean it has been handled. You have 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 are not 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 are not 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 have 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 does not 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 is not 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. Not every object should be tasked with these responsibilities. When you need to be very precise, define contracts between collaborators. Bertrand Meyer (1997) 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.

Notes

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

References

- Adams, D. (1993) *Mostly Harmless (Hitchhiker's Guide Series #5)*. Random House.
- Petroski, H. (1992) *To Engineer is Human*, Vintage Books.
- Pye, D. (1978) *The Nature and Aesthetics of Design*. Van Nostrand Reinhold Company.
- Cockburn, A. (2002) *Agile Software Development*. Boston: Addison-Wesley.
- Romanovsky, A., Dony, C., Lindskov Knudsen, J., and Tripathi, A., eds. (2001) *Advances in Exception Handling Techniques*. Springer-Verlag.
- Lea, D. (2000) *Concurrent Programming in Java™ Second Edition: Design Principles and Patterns*. Boston: Addison-Wesley.
- Meyer, B. (1997) *Object-Oriented Software Construction*. Prentice Hall.
- Howell, C., and Veccellio, G. (2001) Experiences with error handling in critical systems. In Romanovsky, A., Dony, C., Lindskov Knudsen, J., and Tripathi, A., eds. (2001) *Advances in Exception Handling Techniques*. Springer-Verlag.