Designing Reliable Collaborations

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

Agenda

- How reliable *does* your software need to be?
- Collaboration
- Use-case and program-level exceptions and errors
- Exception guidelines
- Recovery strategies
- Making collaborations more reliable

Reliability Design Challenges

"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."

–Henry Petroski, *To Engineer is Human*

Software Design Considerations

- Most software need not be impervious to failures or misuse
- But it shouldn't easily break, either
- A large part of design involves accommodating situations that cause your software to veer from the normal path
- Designing certain collaborations to be more reliable increases your software's ability to handle anticipated problems

How Much Should We Think About Failure?

- The more serious the consequences of failure, the more effort you need to expend. Alistair Cockburn suggests four levels of criticality:
  - Loss of comfort
  - Loss of discretionary money
  - Loss of essential money
  - Loss of life

But Consider...

- Software that runs unattended for long periods of time under fluctuating operating conditions
- Software that “glues” larger systems together in spite of communications glitches and data errors
- Components that “plug in” and run in different environments
- Consumer products

When Should We Think About Failure?

- What the inventor of modern exception handling concepts says:

  "I have long (but quietly) advocated dealing with exception handling issues early in the design of a system. Unfortunately, there is a natural tendency to focus on the main functional flow of a system, ignoring the impact of exceptional situations until later."
  —John Goodenough, *Advances in Exception Handling Techniques*

When Do We Think About Failure?

- People typically think about failure only after they've described "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*

Why Do Architects Want Us To Think About Exceptions?

"At an architectural level, the basic patterns, policies, and collaborations for exception handling need to be established early, because it is awkward to insert exception handling as an after thought."
—Craig Larman, *Applying UML and Patterns*

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

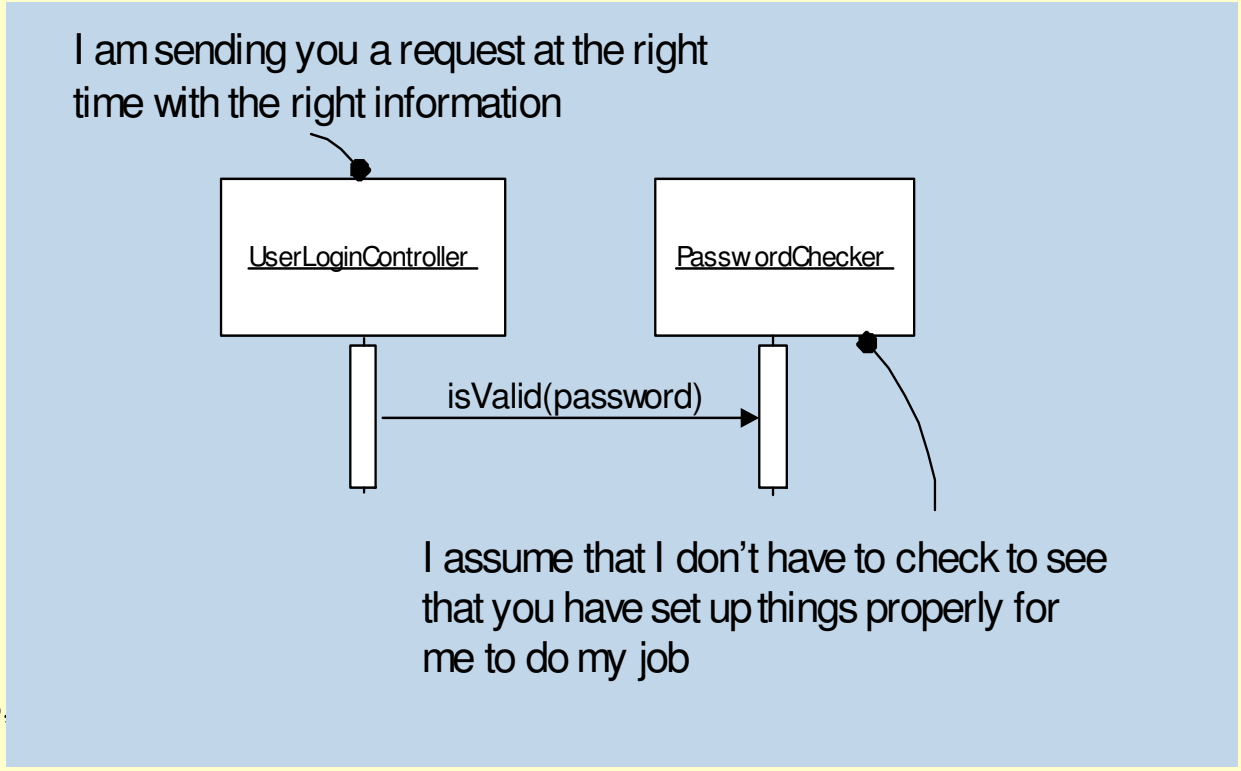
Strategies For Increasing Reliability

- Focus on where to make your design resilient. Identify key collaborations
- Work incrementally on developing consistent collaboration styles. As you work on representative collaborations, come up with policies and repeatable patterns
- Consider exception design early, when you can impact system architecture
  - Characterize different trust regions
  - Develop control centers, reporting, and recovery strategies
- Simplify your programming. Reduce exception handling clutter

Collaboration

Definition: Collaborate

- 1. To work together, especially in a joint intellectual effort
  - This definition is collegial: Objects working together toward a common goal. Both client and service provider can be designed to assume that if any conditions or values are to be validated, they need be done only once



Definition: Collaborate

- 2. To cooperate treasonably, as with an enemy occupation force*
  - If a collaborator can't be trusted, it doesn't mean it is inherently more unreliable. But it may require extra precautions
    - Pass along a copy of data instead of sharing it
    - Check on conditions after the request completes
    - Employ alternate strategies when a request fails

*From *The American Heritage Dictionary*

Fulfilling Requests From Unknown Or Untrusted Sources

- When you are designing an object that handles requests from unknown sources, you may need to take care, too
  - When receiving requests untrusted sources, you *are likely* check for timeliness, relevance, and correctly formed data
- But don't design every object to collaborate defensively
  - It leads to poor performance
  - Redundant checks are hard to keep consistent and lead to brittle code

Design Responsible Communities

- Determine where collaborations can be trusted
  - Carve your software into regions where “trusted communications” occur. Objects in the same trust region communicate collegially, although they still encounter exceptions and errors
  - Some objects still have to check for valid requests... but not all do
    - Give objects at the “edges” and borders responsibilities for verifying correctly formed requests
    - Assign objects that already have control and coordination responsibilities added responsibility for recovering from exceptions
    - Make objects that are reliable “service providers” take on more responsibility

"Build A Message" Use Case

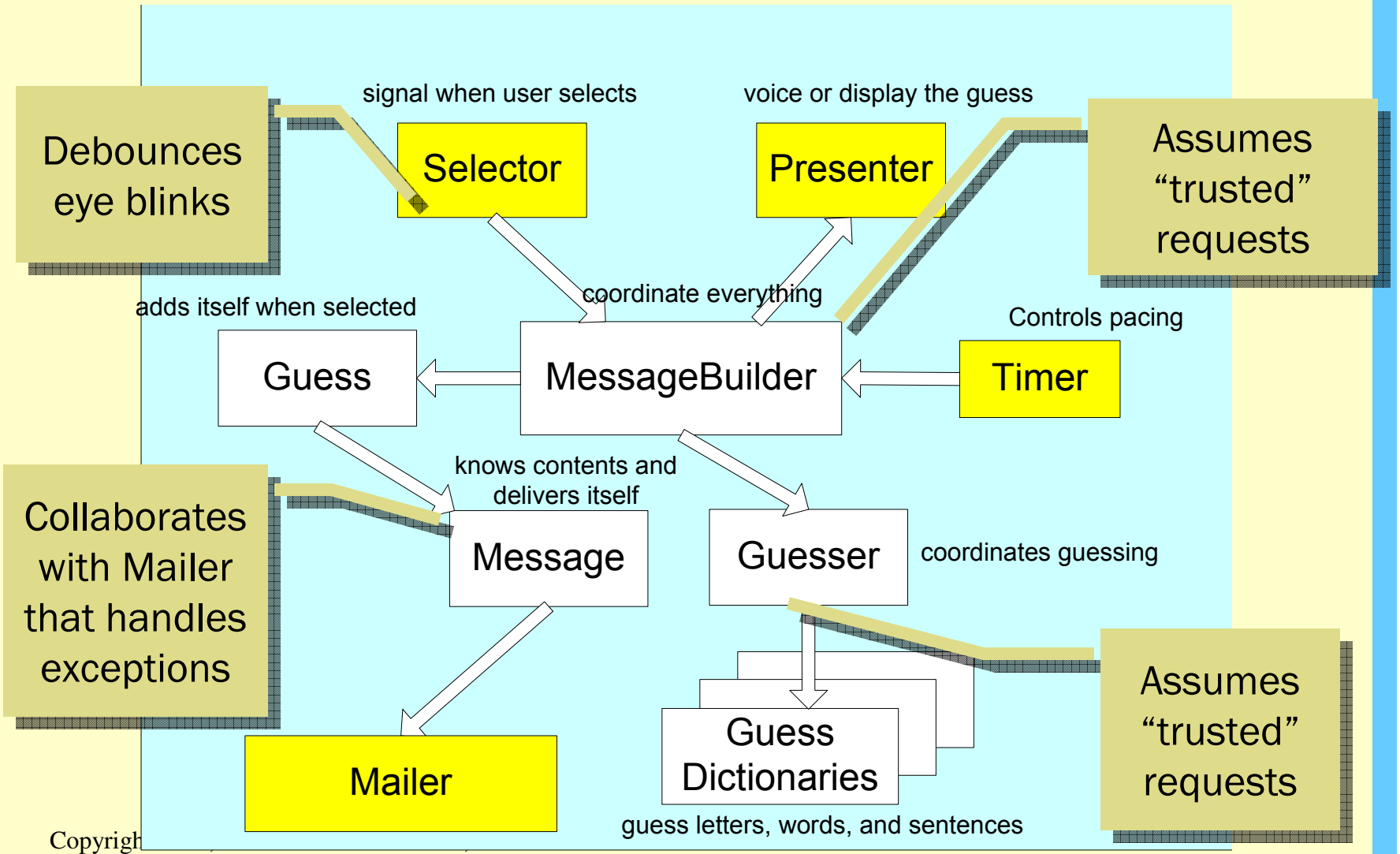
Speak for Me enables a severely disabled user to communicate

Actor Actions	System Responsibilities
"Click" to start software speaking	Start building a message
Repeat until . . .	
Optionally, "click" to select letter	Determine what to speak (letter, word, sentence, or space) Speak letter Add letter to word
Optionally, "click" to select word	Speak space Add word to end of sentence Start new word
Optionally, "click" to select sentence	Speak sentence Add sentence to end of message Start new sentence
… a command is issued	
	Process command (separate use cases)

Implications of Trust

- In Speak for Me, all objects in the application “core” are within the same trust region
- Objects in the application control and domain layers assume trusted communications between each other
- Objects at the “edges”—within the user interface and in the technical services layers—make sure outgoing requests are honored and incoming requests are valid

Objects At The "Edges" Take On Added Responsibilities



Collaboration Cases To Consider

- Collaborations between objects...
  - that interface to the user and the rest of the system
  - inside your system and objects that interface to external systems
  - in different layers or subsystems
  - you design and objects designed by someone else

Use Case-Level and Program-Level Exceptions and Errors

Definition: Exceptions

- Exceptions are deviations from the normal course that we anticipate
- They should be handled by getting the software into a predictable state and continuing on
- How to resolve exceptions can be open to debate
  - What if a sensor reports a fault?
  - What if an order can't be fulfilled?
  - What if a connection is dropped?

Definition: 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 from errors requires drastic measures
  - What if the disk is full?
  - What if equipment cannot be provisioned?
  - What if the OS crashes?

Recoverable Vs. Unrecoverable Use Case Exceptions

- *Recoverable exceptions* can be handled deftly enabling the user to continue his or her task
- In other cases, the user won't be able to continue as planned but the software won't break. These are *unrecoverable exceptions*

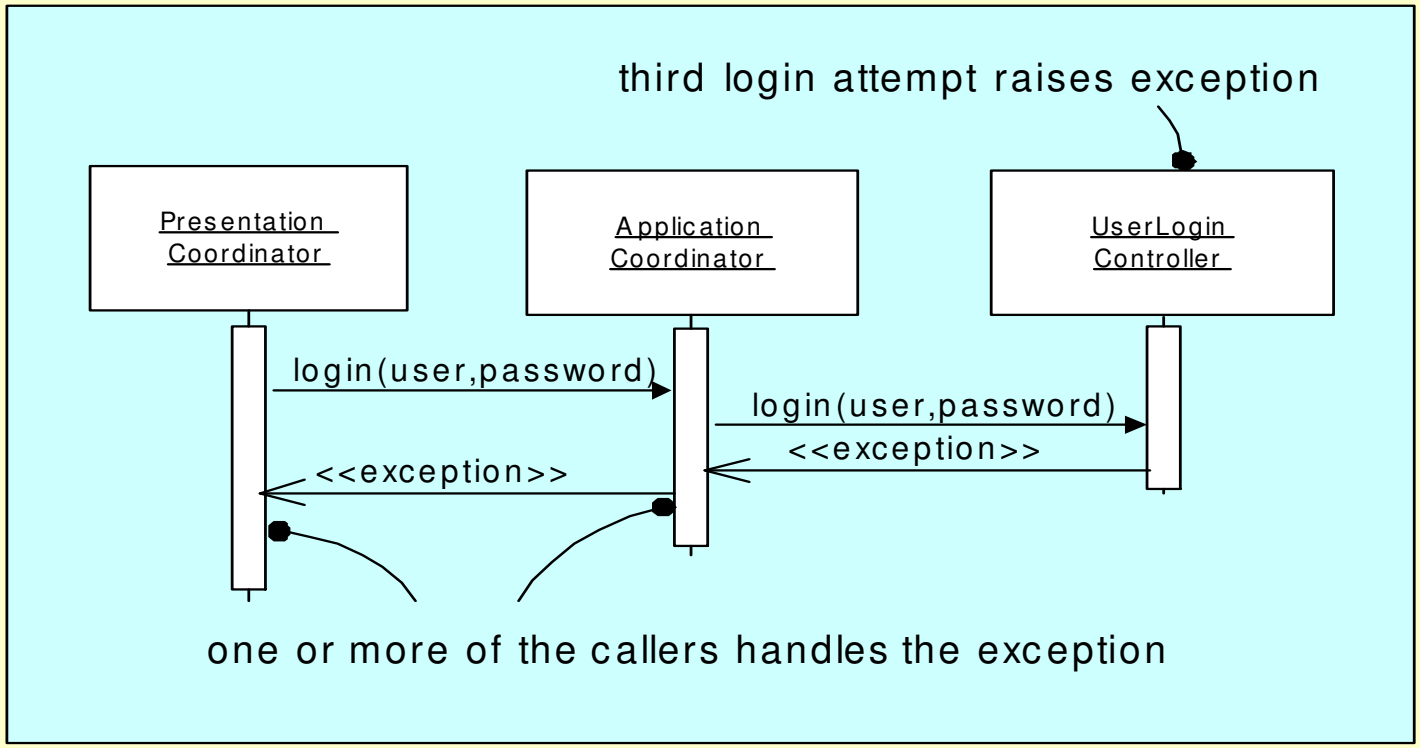
An Example: Is Invalid Password An Exception Or Error?

- Mistyped passwords are a regular, if infrequent occurrence
- We want to react to this condition by giving the user a way to recover. So we view it as an exception
- Most use case level descriptions define some exceptions and 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.
- ...but rarely do they mention errors or programming level exceptions

Programming Exception Basics

- An exception detected during program execution leads some object or component to veer off its “normal” path and fail to complete an operation
- Depending on your design, some object *raises* an exception, while another may *handle* it
- Handling an exception, means recovering by putting your software into a predictable state
- Left unhandled, exceptions lead to system failure, just as errors do

Exceptions Are Signaled And Handled



Programming languages define mechanisms for programmers to *declare* exception conditions (as Classes), *signal* their occurrence, and to write exception handling code

The Anatomy Of Java Exception Handling

- In Java, when you want to handle exceptions, you make calls to exception-raising code within a try-catch block

```
public void doSomething() throws ExceptionA, ExceptionC{
  try{
    // this may throw ExceptionA or B
  }
  catch(ExceptionB eb){
    // this may throw ExceptionC
  }
  finally{
    // clean up
  }
  // do something else
}
```

Recovery code

Unhandled, checked exceptions must be declared

This always executes

The Mismatch Between Use Case And Program Exceptions

- Exceptions described in use cases are fundamentally different from programming exceptions
  - Use case exceptions reflect the inability of an actor (a user) to accomplish their desired task using the software
  - Object exceptions reflect the inability of an object to perform a requested operation
- There isn't a direct correspondence between the two

The Mismatch Between Use Case And Program Exceptions

- A single use case step can result in thousands of requests between collaborating objects, any number of which could cause numerous object exceptions
- Just because someone describes an exception condition in a use case doesn't mean it will happen
  - Your implementation may successfully side-step around the potential exceptional case

Even More To Consider

- 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 forces the user to unexpectedly make decisions and "steer" the darn software
- Sometimes users should be actively engaged in "steering" the system. Sometimes this is inappropriate

What To Do About The Mismatch

- Stick to describing user-level exceptions and how they should be resolved in use case
  - Keep use case descriptions simple… don't tack on to use case descriptions detailed design, architecture, or implementation-level concerns
  - Keep the user's needs in mind
  - View use case exceptions—at whatever level of detail they are described—as guides, not prescriptions for object design solutions
- Document program level exceptions in code
- Create additional design-level documentation or diagrams to illustrate general exception handling policies and tricky solutions

Program Exception Declaration and Handling Guidelines

Exception Design And Reliable Collaborations

- Principles of exception handling design
  - Use exceptions to represent emergencies
  - Handle exceptions close to the problem as you can
  - Separate concerns: untangling exception handling from normal execution
  - Remove redundancies
  - Provide enough information so handlers can make informed decisions
  - Consistency

Use Exceptions Sparingly

- "Use exceptions to report emergency conditions."—Johannes Siedersleben, ECOOP 2003 Workshop on Exception Handling
- Use exceptions to report abnormal and rare events, not for normal control flow
  - A find operation may find zero, one, or many objects. These aren't exceptions, just expected results
  - On the other hand, a dropped database connection is an emergency that the caller needs to be notified about

Why Use Exceptions Only For Emergencies?

- Exceptions don't cost anything in terms of performance until they are raised or thrown
- Programs with lots of exceptions run very slowly

```
public static boolean testForInteger1(Object x){
   try {Integer I = (Integer) x; return true;}
   catch (Exception e) {return false;}}

public static boolean testForInteger2(Object x){
   return x instanceof Integer;}
```

750 times slower!

- If the normal path of execution is tangled with lots of exception handling code, it is hard to read

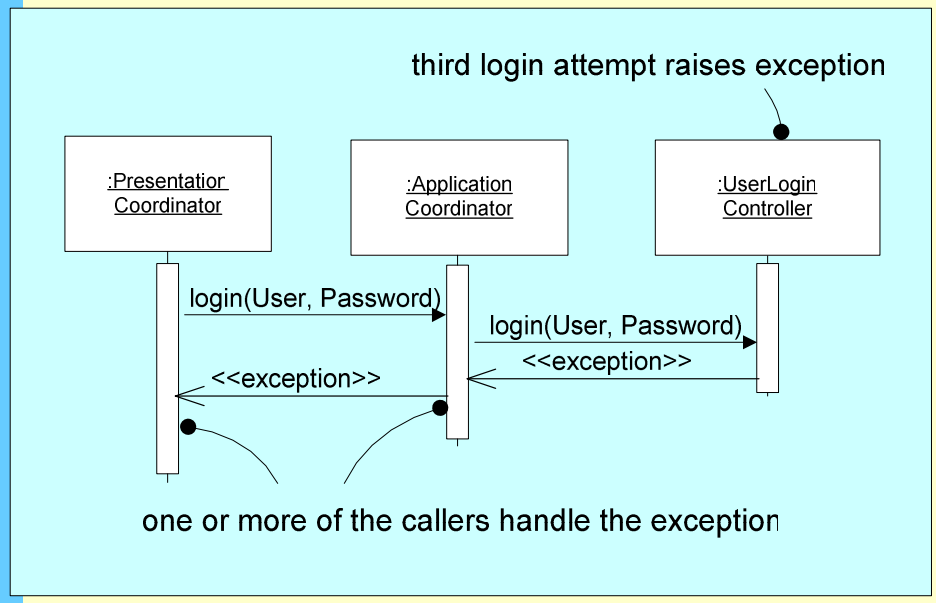
Don't Use Exceptions To Indicate Typical Conditions

- Avoid using exceptions to indicate conditions that can reasonably be expected as part of the typical execution sequence
  - In Java, FileInputStream.read() returns -1 at end of file
    - EOF is viewed as a normal condition encountered when reading a file
  - In Java StringTokenizer, you ask hasMoreTokens() before calling nextToken(). Calling nextToken() after hasMoreTokens() returns false, results in an unchecked NoSuchElementException
    - "End of tokens" is viewed as an abnormal condition because you are supposed to check first

Return Results Instead

- Declare distinguished objects
  - Instead of throwing NoSuchElement exception, the StringTokenizer designer could have returned an “emptyToken”
  - In SpeakForMe! the Guesser returns a NoChoice object if there aren’t any more words, letters or sentences to present to the user

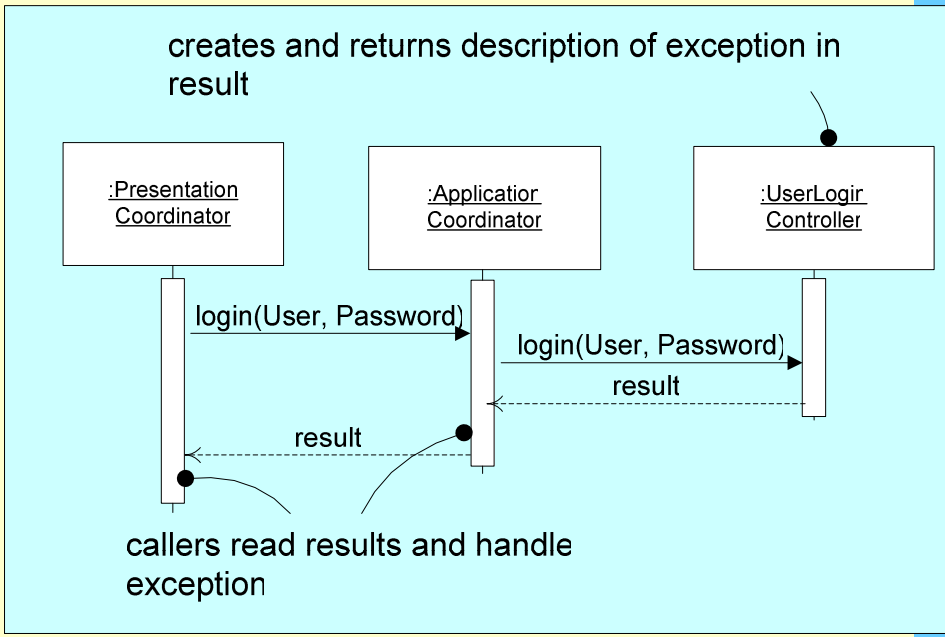
Results Can Be Distinguished From Exceptions



In UML, an exception is shown as a signal.. This looks nearly identical to a dashed line return

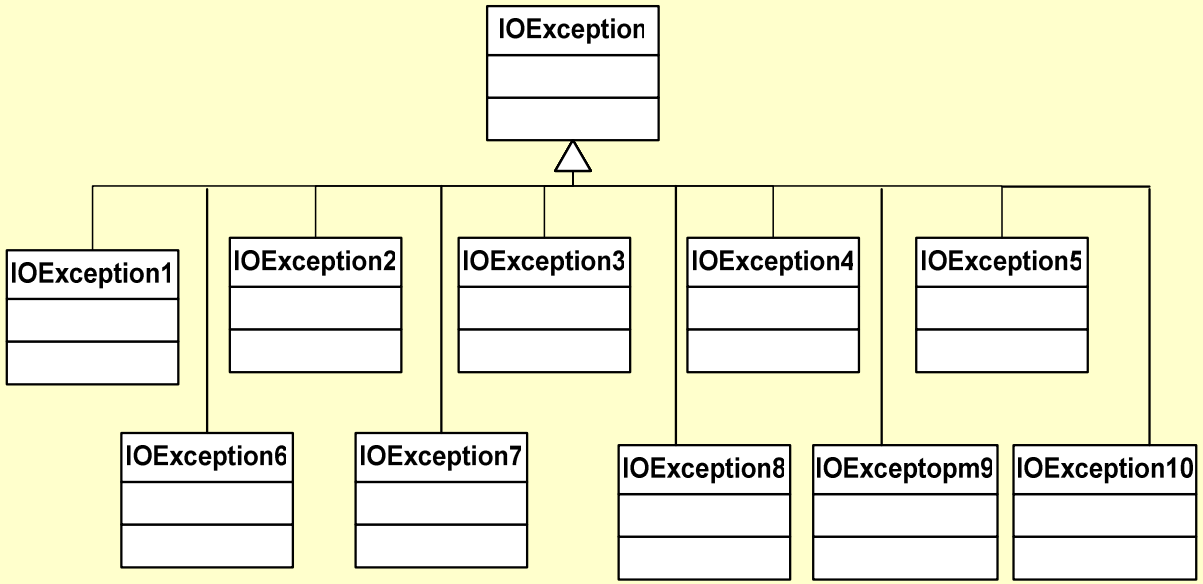
But the code is very different

```
If (loginAttempts > MAX_ATTEMPTS)
   {throw
        new TooManyLoginAttemptsException()}

if (loginAttempts > MAX_ATTEMPTS)
   (results.failed(); ...}

..return results;
```

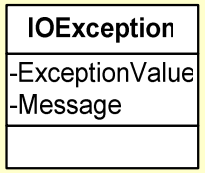



Guidelines For Defining Exception Classes

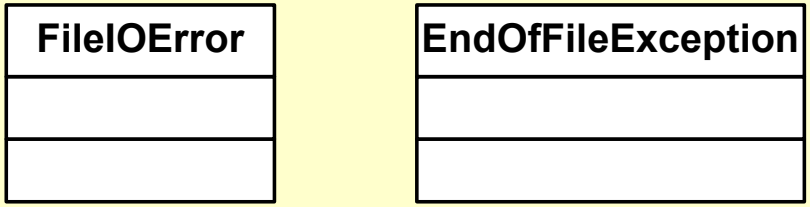
- Avoid lots of exception classes



OR



- Define new classes of exceptions when a handler's behavior is expected to vary

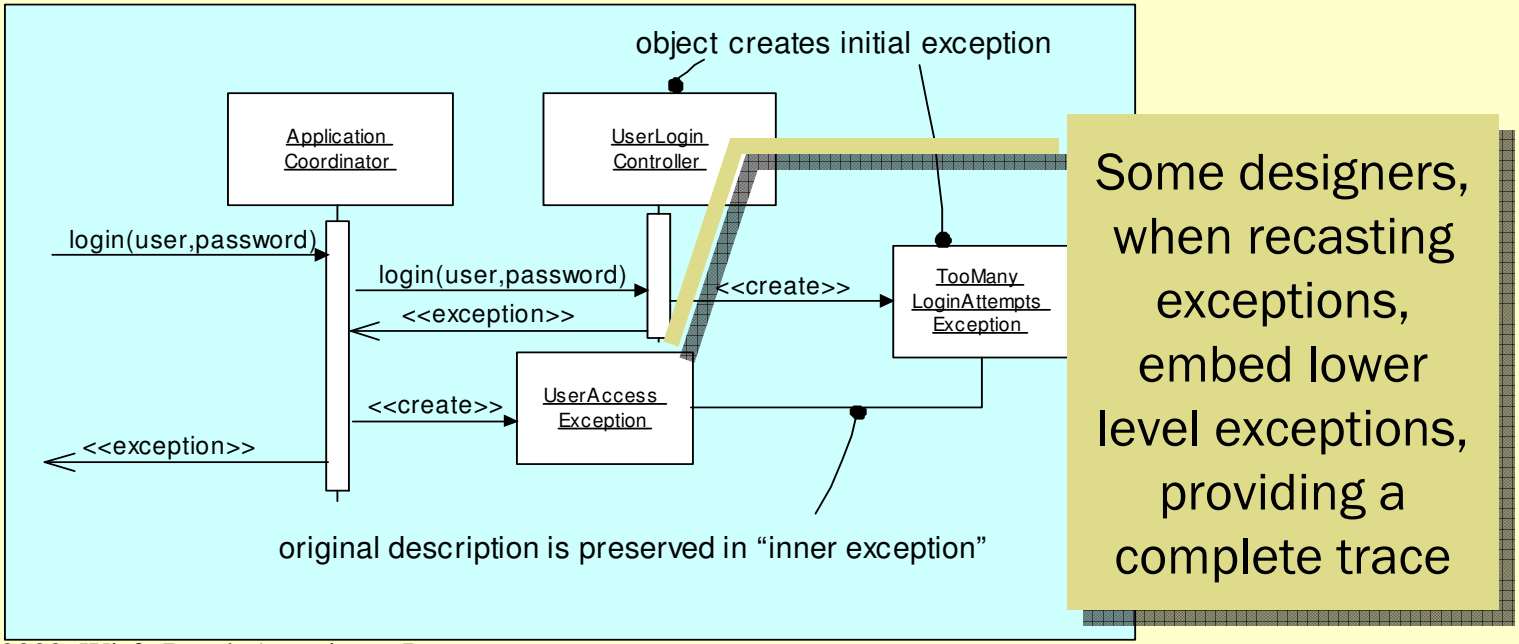


Guidelines For Defining Exceptions (II)

- Name an exception after what went wrong, not who raised it
- Recast lower-level exceptions to higher-level ones whenever you raise your abstraction level—when subsystem boundaries are crossed or when the caller won't know what to do based on the problem you describe
  - It is better for a compiler to report "insufficient disk space to continue compilation" than "I/O error #xxx." If a low-level exception percolates up to objects who don't know to interpret it, it will be hard for them to construct meaningful error messages leading users to suspect a bug in the compiler

Guidelines For Handling Exceptions

- Provide context for the handler in the exception
  - Specific information can be passed along such as values of parameters that caused the exception, detailed descriptions, error text, and information useful in taking corrective action



Some designers, when recasting exceptions, embed lower level exceptions, providing a complete trace

Use Built-In Nested Exception Support

- In C# a read only InnerException property holds the nested exception. Optionally, you can set it in the constructor.

```
public class MyException:ApplicationException {
   public MyException(String message):base(message){}

   public MyException(String message,Exception inner):base(message,inner){}
}

public class ExceptExample {
   public void ThrowInner(){
       throw new MyException("ExceptExample inner exception");}

   public void CatchInner(){
        try {this.ThrowInner();}
        catch(Exception e){
            throw new MyException("Error caused by trying ThrowInner.",e);}
   }
}
```

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemexceptionclassinnerexceptiontopic.asp

Use Built-In Nested Exception Support (II)

- In Java, Throwable is the superclass of Error and Exception classes. It contains the execution stack of its thread, and a message string. It can contain another throwable, a cause (new in release 1.4)

```
A cause can be associated with a throwable either:
1. via a constructor that takes the cause as an argument
try{lowLevelOp();}
catch(LowLevelException le) {
   throw new HigherLevelException(le);  // Chaining-aware constructor
}

2. via the initCause(Throwable) method. This allows a cause to be associated
with "legacy" code which predates the exception nesting mechanism:

try {lowLevelOp();}
catch (LowLevelException le) {// Legacy constructor
   throw (HighLevelException) new HighLevelException().initCause(le);
}
```

http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Throwable.html

Handle Exceptions As Close To The Problem As You Can

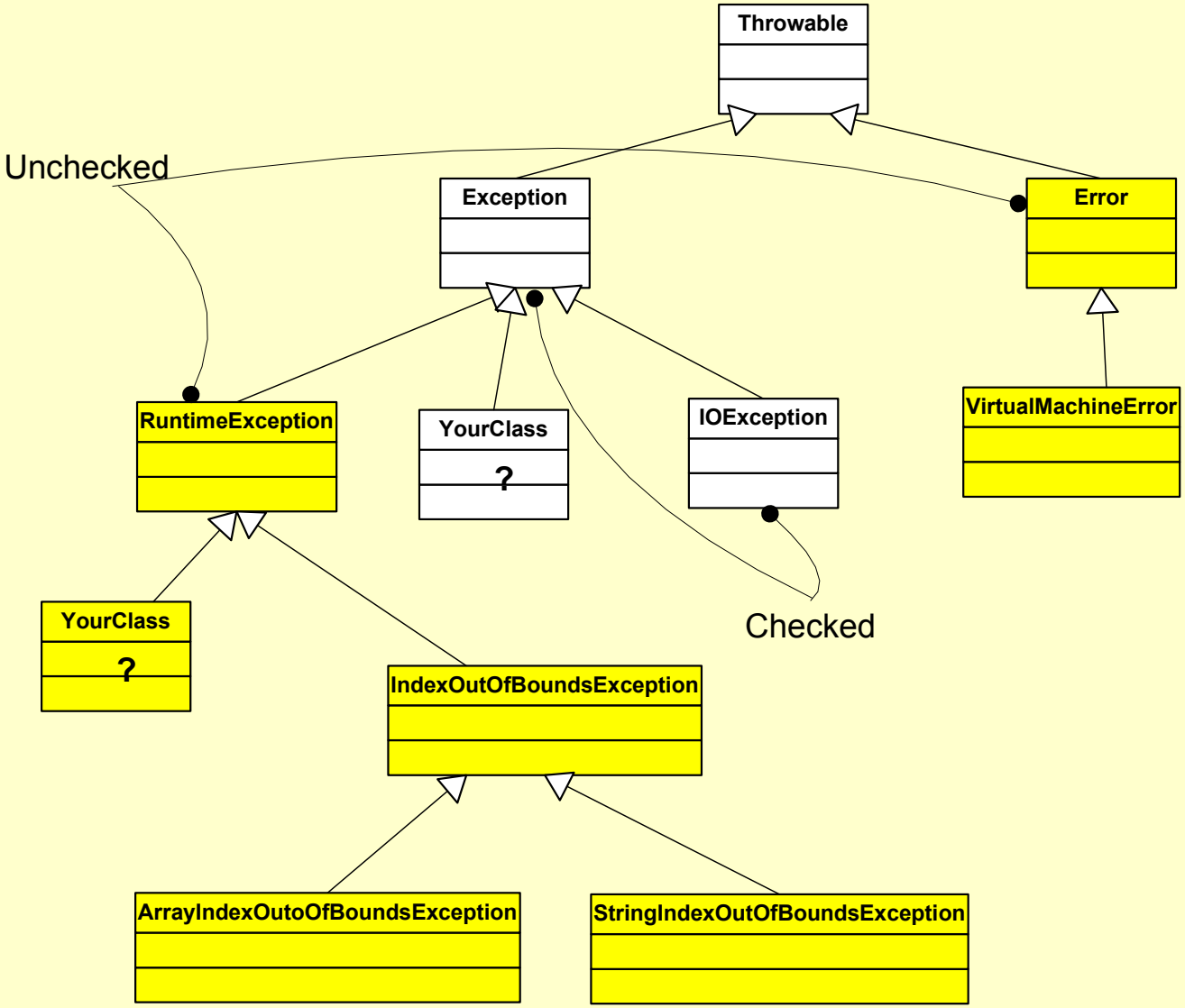
- Who might naturally handle exceptions?
  - As a first line of defense, consider the initial caller. If it knows enough to take corrective action, the exception can be taken care of right away
    - If an object attempts to open a data connection and the open fails, it should take action
- Unhandled exceptions propagate to the nearest matching catch block in the call chain. But don't let exceptions "fly" anywhere
  - Instead, give objects that are "control centers" responsibility for fielding problems and taking corrective action
    - In Speak for Me! the Mailer handles problems with delivery... and the user doesn't even know about behind-the-scenes recovery

Don't Use Checked Exceptions

- Java distinguishes between checked and unchecked exceptions. Placing a checked exception in a throws clause forces you to be explicit
  - When thrown, a checked exception (or a superclass) must be declared in the method signature
  - Clients must deal with the exception, either by
    - catching it, or
    - declaring it in their own throws clause (passing along the problem to someone else)
- Instead, use unchecked exceptions
  - They need not be declared in your method signature nor immediately dealt with by the caller
    - Makes it easier to spot the real handler
    - Avoids defensive exception swallowing code—e.g. code with empty catch blocks or simple logging

Don't Use Checked Exceptions

The downside to checked exceptions: increased code bulk, programming pain, and signature rigidity



Programming Language And Library Impacts

- The programming language and libraries you use influence the way you view program exceptions
- Exceptions can represent a structure for control flow, a structure for handling abnormal or unpredictable situations, or something in between
  - Exceptions in Java, Ada, C++, C#, were designed to mainly be used as control structures
  - Eiffel's exceptions are designed to represent abnormal, unpredictable erroneous situations

Eiffel's View of Exceptions

- A routine (method) must either succeed or fail: either it fulfills its contract or it doesn't. If it doesn't fulfill its contract, an exception is *always* raised
  - Exceptions are not part of the signature
  - Assertion violations and programming errors cause exceptions to be raised
  - Assertion checking and contract specification is an integral part of the Eiffel programming language
- The difference in style and substance is real:
  - Java: 1 unchecked exception thrown once every ~140 lines; checked more often; results in much more exception handling code
  - Eiffel: 1 every 4,600 lines, much less exception handling code

Recovery Strategies

"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

Choices and Conseqences

"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, *The Nature and Aesthetics of Design*

Recovery Strategies (I)

- Inaction—Ignore the request
- Balk—Admit failure
- Guarded suspension—Suspend execution until conditions for correct execution are established
- Provisional action—Pretend to perform the request, but do not commit to it until success is guaranteed
- *Recovery—Perform an acceptable alternative*

Recovery Strategies (II)

- 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
- Appeal to a higher authority—Ask someone to apply judgment and steer the software to an acceptable resolution
- Resign—Minimize damage, write log information, then signal definite and safe failure. Definite means no need to try hard to repair; safe means damage reducing efforts have been taken

Workable Solutions Aren't Always Simple

- Mixing or combining recovery strategies often leads to more satisfactory results
- This increases design complexity
- Solutions don't always feel reasonable—even if they are the best solution given the circumstances

Determining Who Should Take Action

- Objects do not work in isolation—they collaborate to fulfill larger responsibilities—a key question is which objects should take on added responsibilities for guaranteeing success in spite of individuals' failures?
  - You can place burden for success on the requestor, shifting some onto the object providing the service, split extra responsibilities between them, or designate others get involved
  - ...each choice has consequences

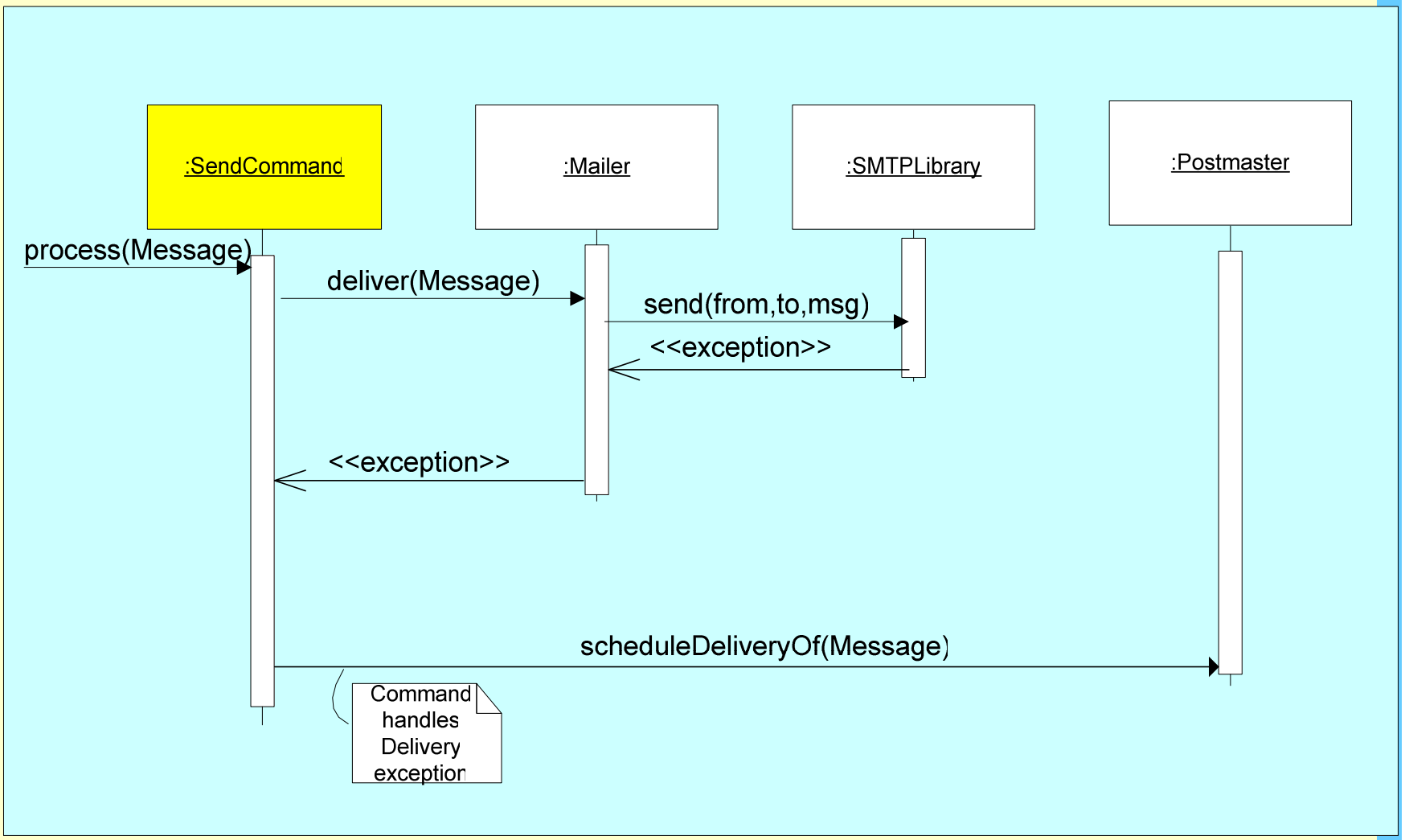
Considerations When Asking The Client To Check Before Making a Request

- Can clients easily check that success will be guaranteed?
  - If not, you may have to expand the service provider's interface
- What guarantees are there that after an object has been checked for readiness, it stays that way?
- Is the cost of checking prohibitive?
- Does checking produce undesirable side-effects?

Considerations When Giving The Client Some Responsibility For Recovery

- How much responsibility should it take?
- Is it reasonable for each client to employ individual recovery strategies, or should you design some common recovery facilities for clients to use?
- Or, should some object better equipped to handle the situation be told of the failure?

Controllers Often Handle Problems



Considerations When Giving The Service Provider Recovery Responsibility

- Is this over engineering?
- How easy is it for the service provider to detect that it has failed?
- Is it acceptable to introduce pauses or delays for the service provider to fix up things and carry on?
- What is the probability of the service provider getting what it needs to be able to continue?

Making Collaborations More Reliable

A Strategy For Handling Exceptions For A Key Collaboration

- Brainstorm exception and error cases that should be addressed
- 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. Fail fast and iterate

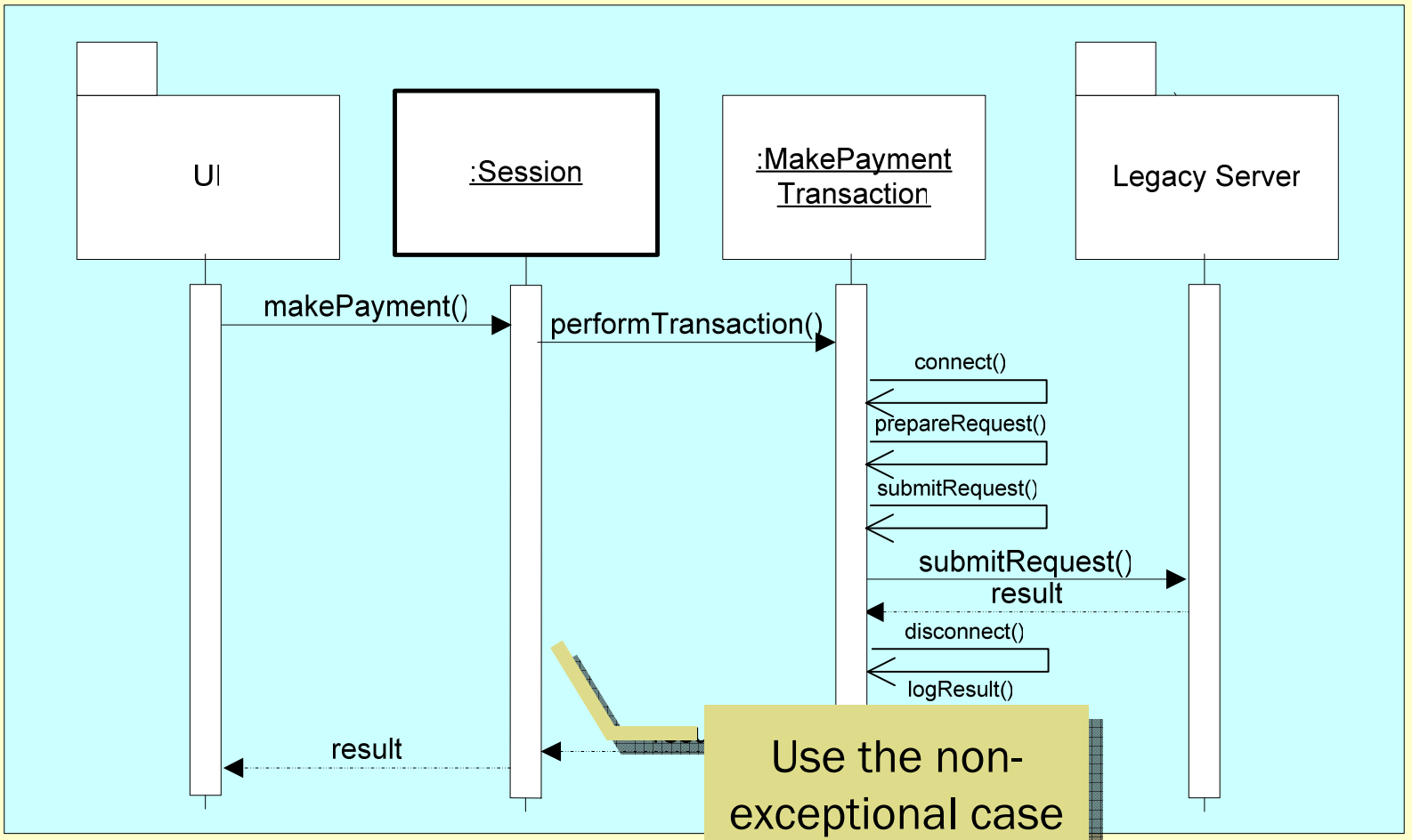
List What Might Go Wrong

- Enumerate all the exceptional conditions you can think of for a specific chunk of collaborative behavior. Consider:
  - Users entering misinformation or failing 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
  - Critical performance errors

Next...

- Pick off a single exception that everyone agrees is a relatively common occurrence
- Determine which object should detect the exception and how it should be resolved
- Describe additional responsibilities of collaborators
  - Expect to introduce new objects along with any complex recovery strategies—don't burden objects with too many exception handling responsibilities
  - Develop and document exception-handling policies as you go

Illustrate The Non-Exceptional Case



Use the non-exceptional case to guide your consideration of exceptional cases.

Describing Your Exception-Handling Design

- Add to existing collaboration stories without piling on too many details
  - Leave an unexceptional diagram alone
  - Draw a new diagram that illustrates how a key exception is handled…but don't draw many similar diagrams
  - Instead, write a simple document that explains exceptions considered, their resolution, and what is considered out of scope

Explain And Document Specific 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.

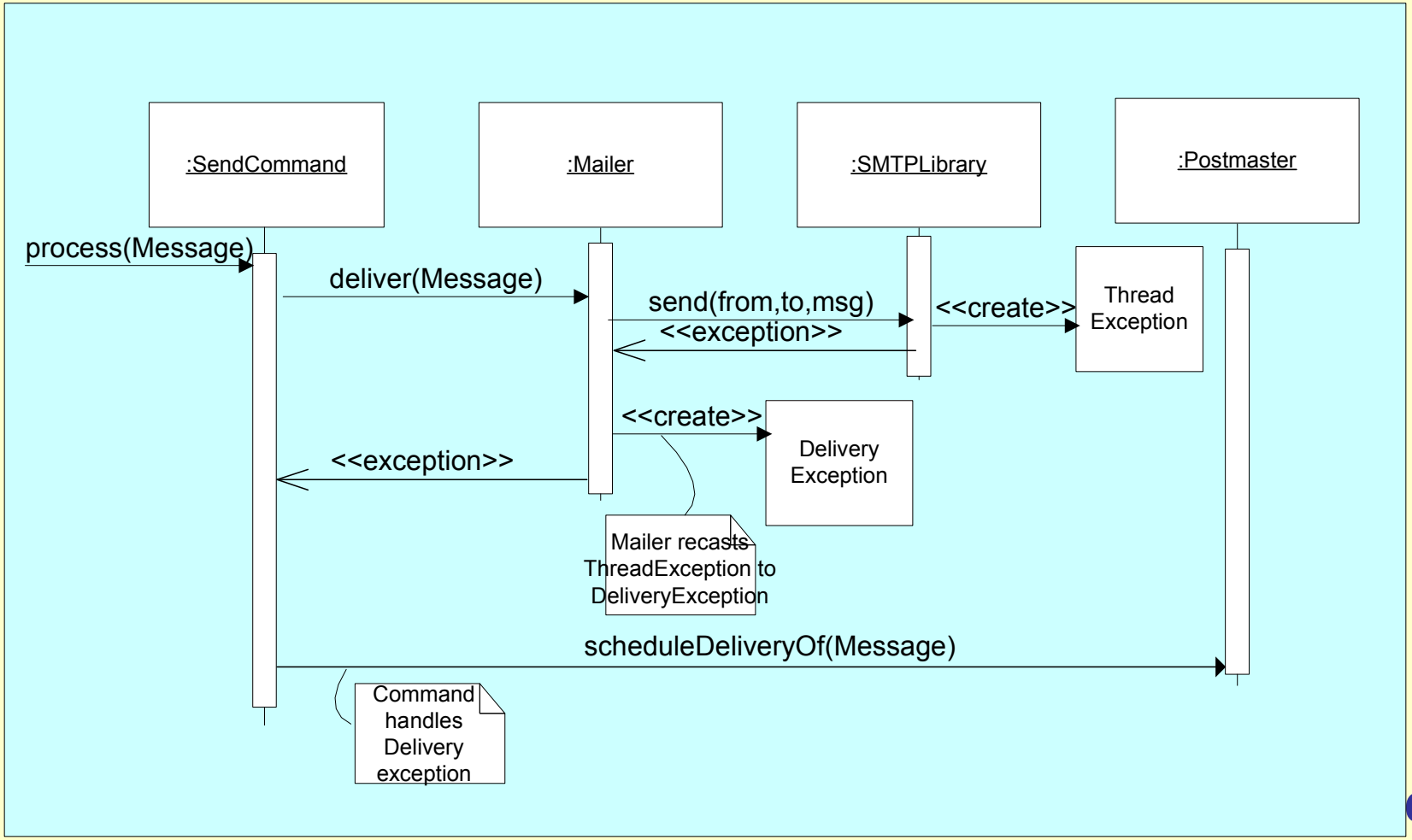
Use descriptions approachable to users, developers and other stakeholders

Develop General Policies

- Describe your solution and explain general policies
  - 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

The Limits of What Can Be Explained In A Diagram

- When you show an exception being raised, you won't necessarily know which object handles it unless you explicitly add that detail



Specify Contracts Between Collaborators

- Contracts, invented by Bertrand Meyer, spell out obligations and benefits for both clients and service providers
- A contract is a job description
  - A lazy service provider would place high demands on its clients and guarantee vary little
  - ...but a demanding client wants guarantees!
- Contracts are agreements for collaborators to work together in specific ways
  - They are contrary to *defensive* collaborations where nothing is trusted and everything is checked

Example: A Contract For A Request That Spans A Trust Boundary

Request: Funds Transfer	Obligations	Benefits
Client: Online banking app	(precondition) User has two accounts	Funds are transferred; balances adjusted
Service provider: backend banking system	(preconditions) Sufficient funds in the first account Honor requests only if both accounts active (postcondition) Both balances are adjusted	Only needs to check for sufficient funds and active accounts, need not check that user is authorized to access accounts

Who's Responsible For Guaranteeing Obligations?

- Decisions on who should take responsibility is partly style and partly a matter of trust between objects:
  - In untrusted collaborations, a client might take special preparations before making a request and make extra checks afterwards to verify the service was performed correctly
  - When handling requests from unknown sources, a service provider may take nothing may be taken for granted—everything is checked beforehand

When To Use Contracts

- Use them as a point of discussion when you are assigning responsibilities among collaborators
- But writing meaningful contracts is a lot of work, if you have no language support! Use them when you want to be formal and precise
- Contracts are especially useful for defining collaborations between your software and external systems
  - The hardest part is ensuring that guarantees are met, especially when a service provider collaborates with many others to get its job done

Frame Your Collaborations

"When you turn on a light, you probably think of your movement of the control button and the illumination of the light as a single event. In fact, of course, something more complex is going on."

— Michael Jackson

- Software systems can be thought of a set of related and interconnected sub-problems—and as a consequence may be comprised of several different "problem frames". Each different class of problem has different concerns and design issues

5 Problem Frames

- **Control Problems** - controlling state changes of external devices or machinery
- **Connection Problems** - receiving or transmitting information indirectly through a connection
- **Information Display Problems** - presenting information in response to queries about things and events known by your software
- **Workpiece Problems** - a tool that allows users to create and manipulate computer-processable objects or “workpieces”. Just like a lathe is a tool for woodworking, software is a tool that helps users create documents, compile programs, compose music, perform calculations, manipulate visual images, generate reports...
- **Transformation Problems** - converting some input to one or more output formats

Problem Frames And Collaboration Design

- Each different class of problem has different concerns and design issues.
  - ✓ Control frames—Do you need to determine whether attempts at changing external conditions had the desired effect? If so, you will design ways to probe whether things are as you expect. And if they aren't, well... is the problem your software or an external device, and how should you recover?
  - ✓ Connection problems–Connections break down, information gets lost or gets garbled. You may need to re-establish connections or try alternate paths, or...

Problem Frames and Design Focus

- ✓ Information Display Problems–Does your design have to accommodate imprecise questions or partial answers?
- Workpiece Problems—What's the real nature of the workpiece and how usable is your tool?
- Transformation Problems–What constitutes an acceptable loss of information or the reversibility of a transformation can be an issue. What constraints are there on speed or memory utilization?

When Problem Frames Affect Collaboration Design

- The ideal: Jackson advocates fully understanding the nature of the problems your software is trying to solve before you start design
- The agile reality: In a world full of imperfect knowledge and time constraints, be prepared. Characterize what problem frames your design must tackle... but expect more problems to crop up during design. When they do, use problem frames to focus your design strategies

How a Connection Problem Affects Your Design

"In many problems you'll find that you can't connect the [software] machine to the relevant parts of the real world in quite the way you would like. You would prefer a direct connection...instead you have to put up with an indirect connection that introduces various kinds of delays and distortion."
–Michael Jackson

- Basic strategies for dealing with connection issues:
  - Consider that your software is really interacting with "something in the middle" that is connected to "something out there" that doesn't always work.
  - Design your software to react in the face of potential time-delays, conflicting states between "connected" system as well as faulty connections

Review Your Collaboration Design

- Look for places where complexity may sneak in
  - *embellished recovery actions*—is it really necessary to retry a failed operation, log it, *and* send email?
  - *unnecessary checks*—if you aren't sure whether some condition should be checked, why not check anyway?
  - *redundant validation responsibilities*–when you are uncertain what objects should take responsibility, why not do it several places?

Review Your Collaboration Design

- The most common bugs
  - failing to address additional exceptions that arise in exception handling code
  - propagating exceptions to unprepared software
  - thinking an exception has been handled when it has merely been logged
  - range errors when mapping error codes to exceptions

— Charles Howell and Gary Veccellio, Advances in Exception Handling Techniques

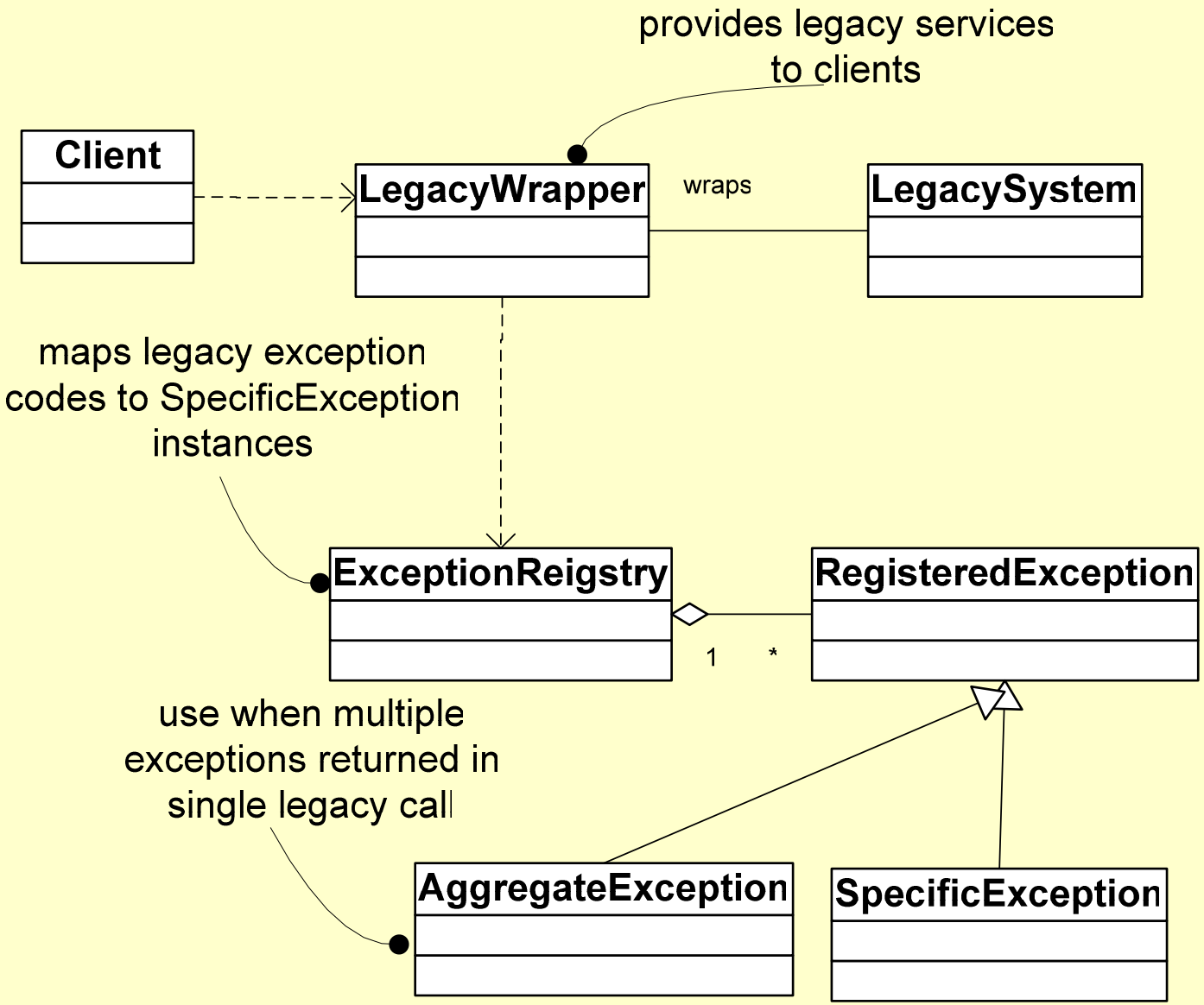
The Exception Registry Pattern* (i)

- Legacy exception conditions are often returned as coded values that have evolved in an ad-hoc fashion. The encoding of exception values may be arbitrary, with little or no planning
- To avoid having clients using legacy services deal with this complexity, design a set of exception classes and translate the exception codes into systematically thrown exceptions

*Pattern Author: Nik Boyd

http://home.labridge.com/~nikboyd/papers/patterns/except/index.html

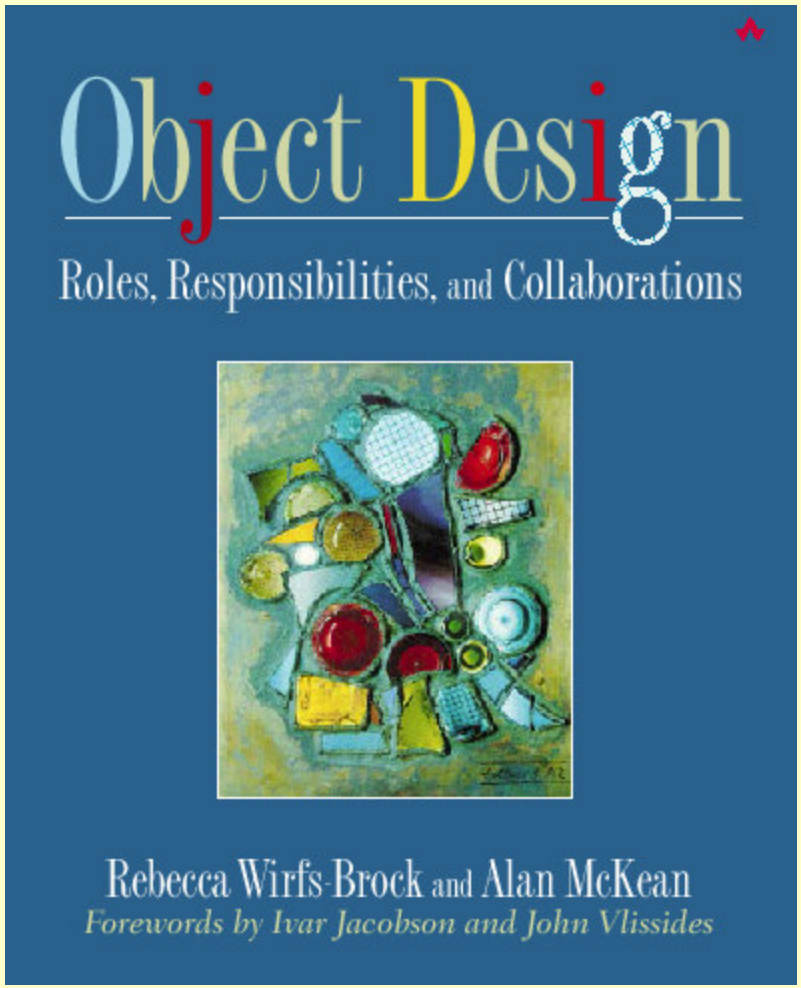
Exception Registry Pattern Classes



In Conclusion

- Designing reliable collaborations demands conscious action, awareness, and reflection
- Complexity naturally increases when software takes extraordinary measures to recover from exceptions and errors
- So…design wisely
  - Understand how reliable your software must be before taking extraordinary measures
  - Expect application-level exceptions to be found in use cases, and many more programming and design exceptions later
  - Look for simple solutions; just don't expect them
  - Develop consistent patterns and policies as you go
  - Use problem frames and trust regions to reason about your design, and contracts when you need to be formal

Resources



- Read more about exceptions, reliable collaborations, wicked problems, and object design strategies in our new book

  Object Design: Roles, Responsibilities and Collaborations, Rebecca Wirfs-Brock and Alan McKean, Addison-Wesley, 2003
- www.wirfs-brock.com for articles & resources