Errors and Exceptions
Rights and Responsibilities

Johannes Siedersleben
ECOOP 2003
Darmstadt

July 21, 2003

sd&m Research GmbH
Thomas-Dehler-Straße 27
81737 München
Telefon (089) 63 81 29-00
Telefax (089) 63 81 29-11

www.sdm-research.de

Introduction

- Background:

  Many business information systems from 1 to more than 100 man years built on different platforms (COBOL, C, C++, Java, C#, no Ada).

- Observation:

  Exception handling is a weak point of ALL software systems I have built or seen so far.

- My story:

  a) complaint
  b) some suggestions, but no panacea

- Questions addressed:

  How many and which exception classes are useful?
  When should an exception be thrown?
  Who is responsible for catching exceptions?
  How far may exceptions be thrown?

Problems Observed

- There is a mess of exceptions flying around. It is not clear when exceptions should be thrown, nor how they are caught.

- The code gets messy because of nested try-catch blocks.

- Many (sometimes all) catch blocks are either empty, contain nonsense code (output to the console, silly transformations of one exception class into another) or – at best – some logging, but no real exception handling.

- There is a huge number of exception classes creating undesired dependencies between the caller and the callee (information no longer hidden)

- Exceptions are misused in order to return ordinary values.

Exceptions and Programming Languages

- Exceptions = transfer of control (similar to goto) +
  additional information channel +
  stack unwinding

- Exceptions used for different purposes, e. g.:
  signaling failure,
  classifying a result (e.g. overflow)
  monitoring (that many records have been processed)
  Java: synchronizing threads
  ... (many more)

- Use of exception encouraged by cumbersome return of results (only one return value; constructors; overloaded operators; out-parameter undesired)

- Exceptions and object orientation are hard to integrate.

- Exceptions are slow, if the event occurs (in Java; other languages??).

Exceptions in Java

```
 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;
}
```

testForInteger1 is about 750 times slower than **testForInteger2** if called with non-integers

Checked Exceptions in Java

```
void foo() throws RemoteException { ..  }
```

- What should the caller do?
- Naive use of checked exceptions makes them ubiquitous.
- They are either ignored or just passed on.
- Checked exceptions considered harmful.

Application Errors

- Application Errors are part of the normal business. They must be handled by the immediate caller.

- Many operations cannot fail at all from the application point of view (e.g. getter-methods)

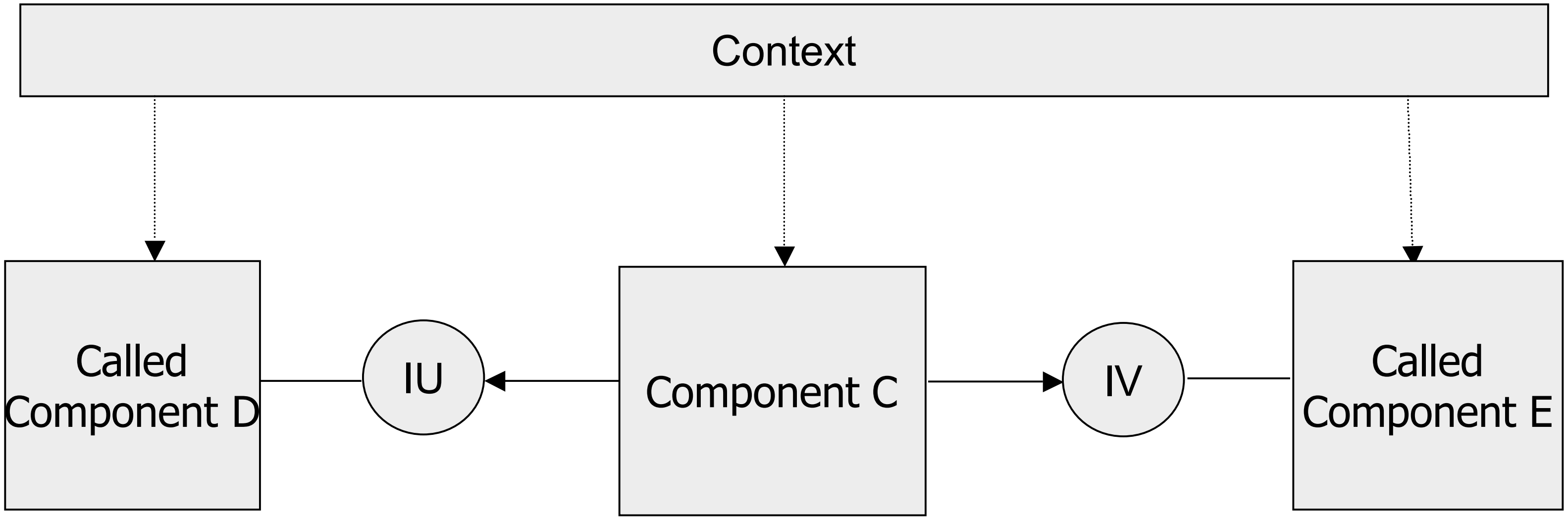
- Reporting application errors by exceptions is not encouraged:
  you would use only one service out of three
  (transfer of control and stack unwinding happen anyway)

- Most operations have very few different application errors; in many cases, *ok* and *nok* is sufficient. Application errors are always completely enumerated.

- Separate clearly
  control flow
  error message
  diagnostic information

Components

- export one or more interfaces;
- import zero, one or more interfaces (not components!);
- run in a given context: binding and (e.g.):
  transaction control, emergency handling.



Emergencies

- Any software can fail, and that is completely different from application errors.

- We suggest to call these failures *emergencies* (Parnas: *undesired event*)

- There is always a huge number of possible emergencies; they cannot be enumerated. Emergencies are really exceptional.

- The caller should not be bothered by emergencies; instead, there should be dedicated emergency handlers. The caller should know and handle only the outcomes it can deal with.

- It is a design decision what you consider to be an emergency.

- And this decision is local for the component:

  a find-operation can find zero, one ore many matching objects.
  the caller may consider a zero-result as an emergency.

Reporting Emergencies

should be very easy:

```
public void foo() {
    String result = ..                    // must never be null
    Emergency.ifNull(result, ...);
}
```

- Behind the scenes, an exception would be thrown, conveying all information needed
    - for the emergency handling
    - for debugging

- Assertions: emergencies could be stated as assertions; however, assertions are often turned off after testing.

- Emergency handling must never been turned off.

Emergency Handling and Safety Facades (1)

- The main options of emergency handling:
  ignore, *retry*, *call an alternative*, *resign*
  Note: *Call an alternative* means reconfiguration (i. e. use an alternative implementation of the same interface.

- Two possible outcomes of emergency handling:
  success xor *definite and safe failure*

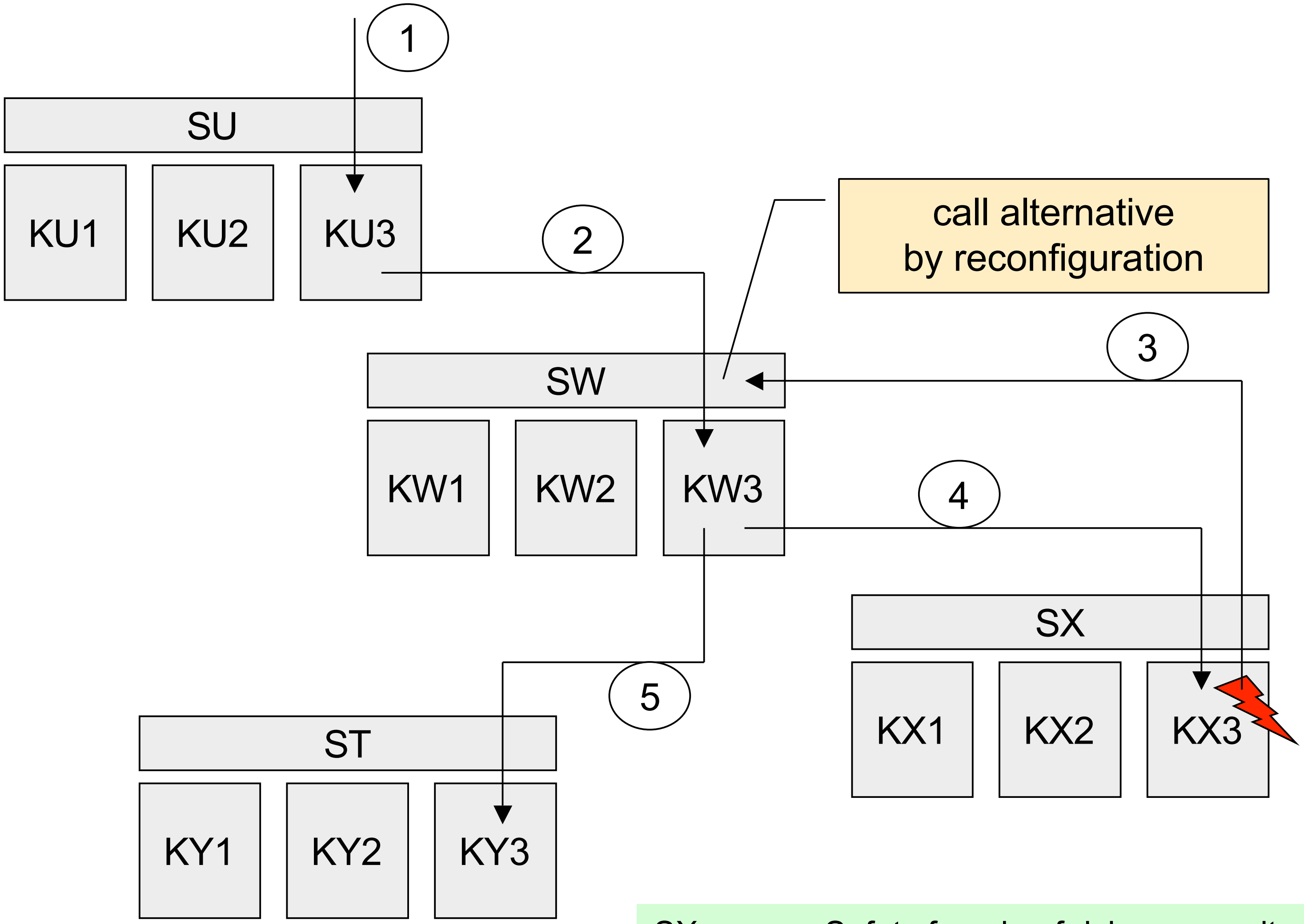
- Emergency Handling and normal business should be separated.

- Components run in different contexts (e.g. batch, local gui, web)

- One emergency handler per component??? Observation: Only few options for emergency handling, thus:

- Distinguish safe and unsafe calls (that is with/without emergency handling in between); components are grouped by risk communities.

Emergency Handling and Safety Facades (2)



SX	Safety facade of risk community X
KX1	Component 1 of risk community X

Preconditions

- Preconditions protect the called component against illegal calls.
- Stated in terms of input parameters and/or the components state.
  Notorious problem in Java: null parameters.
- Violated preconditions are the caller's problem, not the callee's. The callee would reject an illegal call:

```
 public void foo(String s) {
    Reject.ifNull(s, ...);     // s must not be null
    ...
}
```

- Strong vs. weak preconditions:
  square root: reject negative input
  matrix inversion: accept all non null square matrices

Postconditions

- Postconditions protect the called component against buggy implementations.

- It is of little use to check postconditions within the implementation:

```
int add(int a, int b) {

    int result = a + b;
    assert result == a + b;
    return result;
}
```

- Violated postconditions mean that the called components has failed. It is up to the next safety facade to handle the emergency.

Ten Rules

1. Have a clear distinction between emergencies and application errors.

2. Detect emergencies as early as possible.

3. Reject calls if there is a violated precondition.

4. Assume all input parameters to be non null by default.

5. Design risk communities accessed by safety facades.

6. Concentrate emergency handling in safety facades.

7. Let safety facades catch all exceptions but the *ViolatedPreconditionException*.

8. Report application errors using special return values (e.g. null) if possible.
   Use checked exceptions otherwise.

9. Handle application errors immediately.

10. Don't use self implemented exception classes unless they are necessary
    for the control flow.