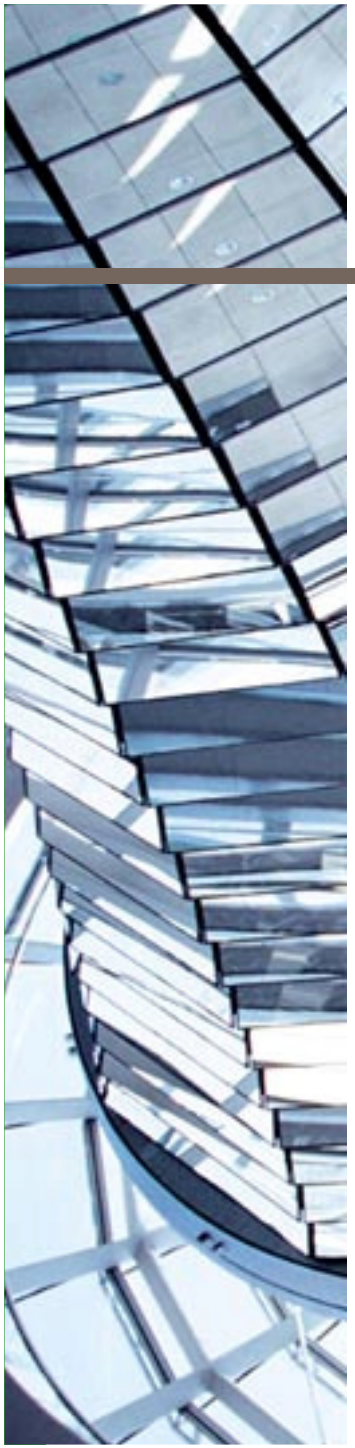


7. Exception Handling
Reference Architectures and Patterns

Winter Semester 2008 / 2009
Prof. Dr. Bernhard Humm
Darmstadt University of Applied Sciences
Department of Computer Science

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

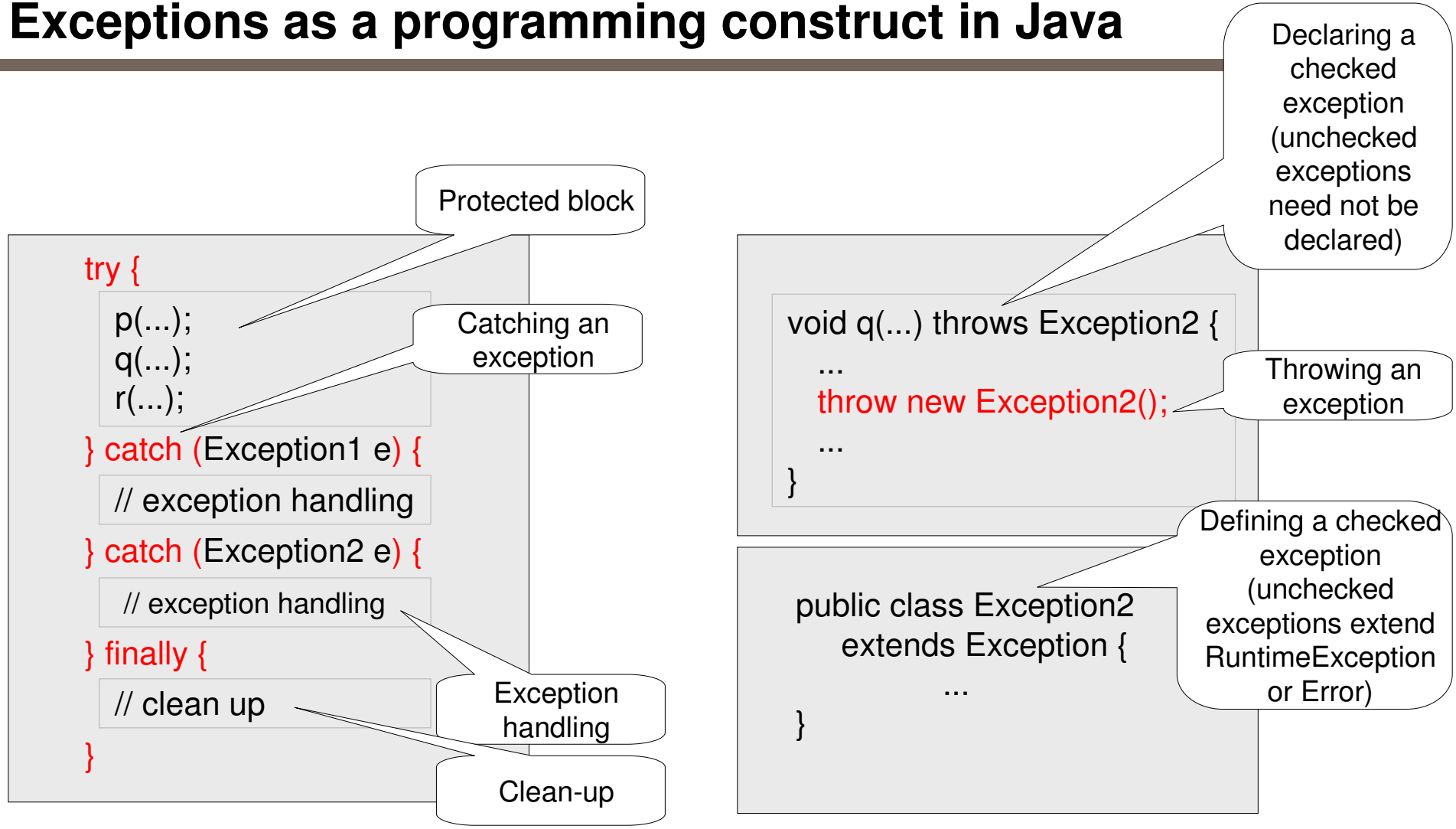
The lecture in the context of the entire course

1. Introduction
2. A reference architecture for business information systems
3. Application kernel
4. Persistence and transaction
5. Authorization
6. Client architecture
7. Exception handling
8. Other reference architectures: BI, systems integration, SOA, business domains

No computer program without exceptions

- Exceptions are undesired behaviour during the execution of a computer program
  - … due to programming errors, e.g., division by zero
  - … due to technical problems, e.g., network failures
  - … due to faulty operation by the user, e.g., *from* date is greater than *to* date
  - … due to exceptional business behaviour, e.g., credit limit is overdrawn
- Exceptions do occur:
  - Programming errors can be reduced by proper design an testing but cannot be avoided completely in complex systems
  - Technical problems can be reduced by redundancy etc. but cannot be avoided completely
  - Faulty operation can be reduced by ergonomic user interfaces but cannot be avoided completely
  - Exceptional business behaviour is part of the business and is not affected by computer programs
- → Exceptions cannot be avoided. So, we have to handle them!

Repetition:
Exceptions as a programming construct in Java

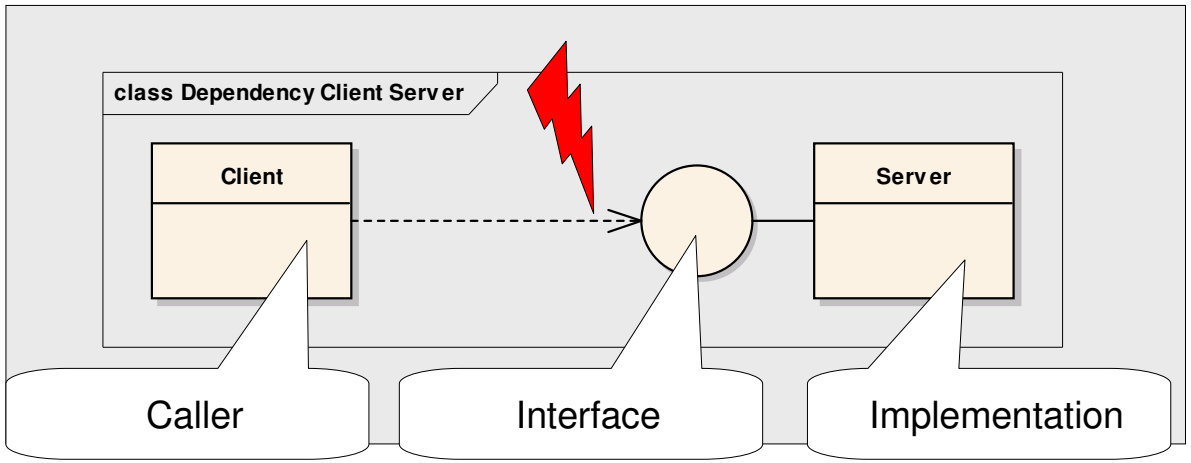


Aren't Java Exceptions enough?

- Exceptions as a feature of the programming language do not automatically lead to a proper design of exception handling. Sometimes they are even inappropriate!
- Business code tends to be littered with exception handling
- Exceptions may lead to spaghetti code: exceptions as the modern form of the go to statements (E. Dijkstra: "Go to considered harmful")
- There is no commonly accepted convention for using exceptions, e.g.:
  - When to use checked exceptions, when unchecked exceptions, when not to use exceptions at all?
  - Where to catch and handle exceptions?
  - How do handle exceptions?
- Textbooks on design and programming rarely cover exception handling
  (positive example: J. Siedersleben – *Moderne Softwarearchitektur*)

Example:
Exception handling may violate the principle of information hiding

```
…
try {
    result = PaymentManager.transfer(100, account1, account2);
} catch (DatabaseNotAvailableException e) {
    // what‘s next??
}
…
```



- The caller of an operation knows the interface only, not the implementation
- The caught exception reveals implementation details. The implementation urges the caller to handle the exception (checked exception)
- → in this example, exception handling violates the principle of information hiding. The caller does not know what to do ("I didn't even know that there was a database involved. I have no knowledge on databases but on payment transactions only!")

Solution: Classify Exceptions in business application exceptions (*A Exceptions*) and technical exceptions (*T Exceptions*)

- **Business application exception (A Exception):** exception that may occur in the business application context of the *interface*
  Examples:
  - Operation *withdraw money* in a banking application: credit limit overdrawn
  - Operation *open file* in an operating system: file not found
- **Technical exception (T Exception)**: Exception that results from the *implementation* and its technology
  Examples:
  - Operation *withdraw money* in a banking application: database not available
  - Operation *open file* in an operating system: crashed file server
  - Violated pre / post condition (result from programming errors and, hence, are no business exceptions!)
- **The *provider* of an operation classifies the exceptions**

Business application exceptions (A Exceptions)

- Must be enumerated completely (this is always possible)!
- ***Are integral part of the interface***
- 2 Design alternatives for A exceptions:
  - Return value
  - Checked Exception
- Examples:

```
int transfer (int amount, Account from, Account to)
/**
  post: result = 0 if transfer successful
  post: result = -1 if credit limit overdrawn
  …
*/
```

```
int transfer (int amount, Account from, Account to)
    throws CreditLimitException
/**
  error: CreditLimitException if credit limit overdrawn
  …
*/
```

- The caller of the operation must handle the business application exception (and is able to do so), e.g., inform the user!

Technical exceptions (T Exceptions)

- Depend on the implementation and its technology
- In general, cannot be enumerated completely
  (Murphy's Law: „if anything can go wrong it will")
- Design Recommendations:
  - Not part of the interface!
  - Always use *unchecked Exceptions* (RuntimeException oder Error) for T Exceptions
  - If you are using libraries or neighbouring components that throw T Exceptions as checked exceptions: catch the exception, wrap it into a new RuntimeException and rethrow ("exception chaining")
- Example:

```
try {
    result = PaymentManager.transfer(100, account1, account2);
} catch (DatabaseNotAvailableException e) {
 throw new RuntimeException(e);
}
```

- The caller is not able to handle T Exceptions and, hence, does not attempt to do so!

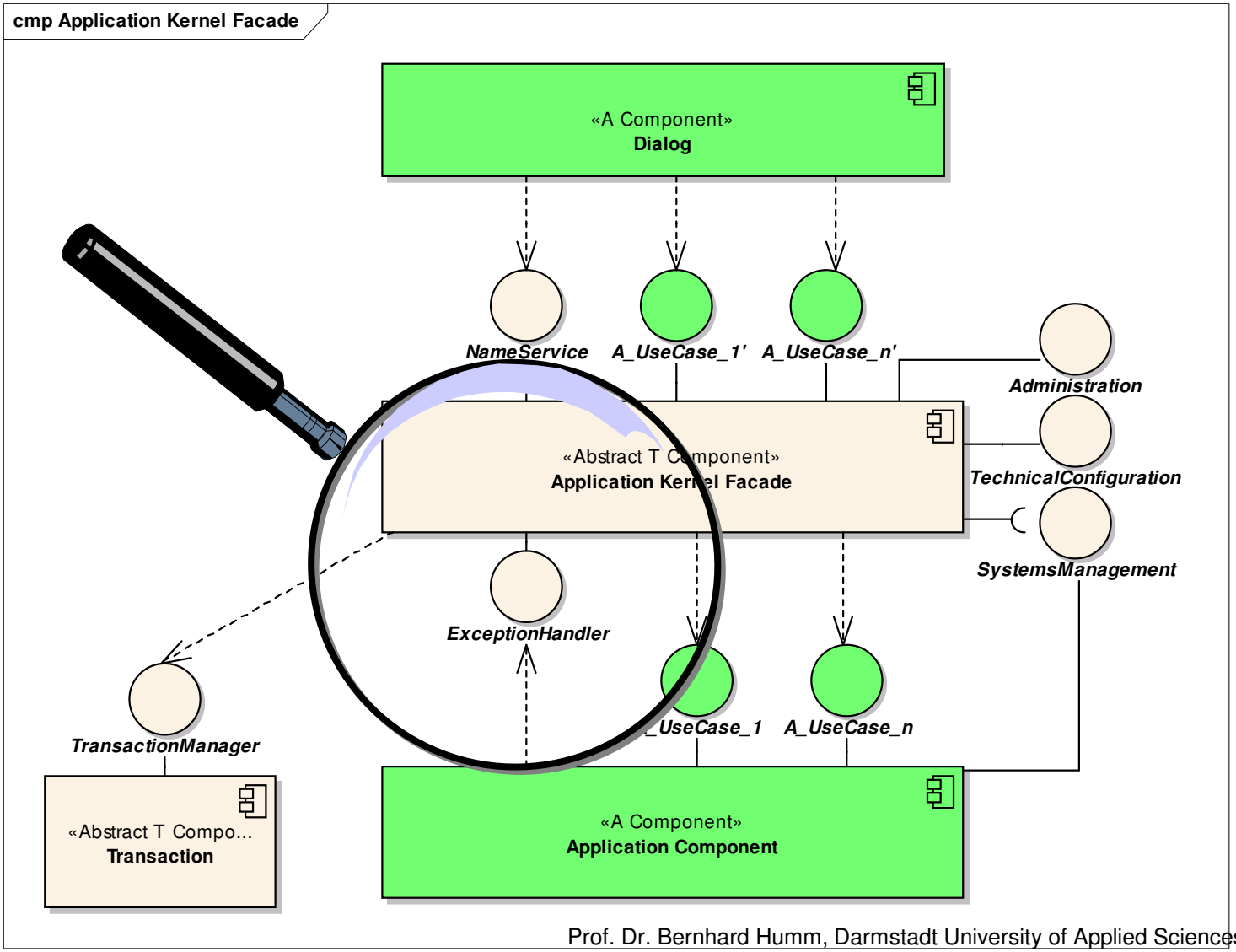
Who then handles technical exceptions? The ExceptionHandler

- Design recommendation:
  - Catch all technical exceptions (unchecked RuntimeExceptions, Errors) at one central point within the system and pass them on to the ExceptionHandler
  - The ExceptionHandler bundles all logic for handling technical exceptions

```
try {
   // main system loop
} catch (Throwable e) {
   exceptionHandler.handleException(e)
}
```

```
class ExceptionHandler {
    …
    void handleException(Throwable e) {
        // classify situation
        // perform corresponding action
    }
    ...
}
```

Reminder: The ExceptionHandler is one service of the Application Kernel Facade



The ExceptionHandler classifies technical exceptions according to their severity and triggers corresponding actions

- **Catastrophe** (global severe problem) → the entire system must be shut down
- **Local severe problem** → the user session must be terminated.
  In both cases, the user (if existing) must be informed. Clean-up work must be performed
- **Repairable problem** → the problem can be repaired, e.g., by *retry* or *compensating actions*
- **Uncritical problem** → The problem will be logged for maintenance purposes (applies for all 4 cases) and system operation can continue

Reminder: sequence for exception handling