Programmieren 3

Kapitel 6: Fehler und Ausnahmen

- Probleme und Fragestellungen
- Normal vs. Abnorm
- Sicherheitsfassade
- Optionen der Ausnahmebehandlung
- Protokollierung

Motivation

- Software hat Fehler
- Selbst fehlerfreie Software läuft in einem fehleranfälligen Umfeld (Betriebssystem, Hardware, Benutzer, ...)
- Die Syntax der Fehler und Ausnahmebehandlung ist einfach
- Ein klares Konzept zur Behandlung von Fehlern und Ausnahmen in einem Programmsystem ist nicht trivial und es gibt viele Alternativen
- In der Literatur werden Fehler und Ausnahmen meist stiefmütterlich behandelt

Crashkurs Ausnahmebehandlung Java

- In Java werden Ausnahmen durch
  `throw new MyException("Fehlermeldung")` geworfen.

- Sie werden vom Client in einer `try` – `catch` – `finally`-Klausel gefangen.

  try: alle Ausnahmen, die in diesem Block **geworfen** werden, können **gefangen** werden.

  catch: Die geworfenen Ausnahmen werden mit den Ausnahmeklassen in der catch- Klausel **verglichen**. Der Rumpf der ersten passenden catch-Klausel wird ausgeführt.

  finally: Vor der Rückkehr aus dieser Methode wird abschließend immer der Rumpf nach finally ausgeführt, **unabhängig** davon, ob die Ausnahme auftrat oder nicht.

- Geprüfte Ausnahmen werden gefangen oder weiter geworfen, ungeprüfte nicht.

Was ist eine Ausnahme?

Beispiel aus dem Pragmatischen Programmierer:

```
public class FileUtil {
        public static boolean open(String filename)
                throws FileNotFoundException {

                File f = new File(filename);
                if (!f.exists()) {
                        return false;
                }
                FileInputStream fis = new FileInputStream(f);
                return true;
        }
        ..
}
```

Der boolean-Wert sagt, ob die Datei existiert oder nicht.
Die Ausnahme sagt: Das Dateisystem ist kaputt (**this is exceptional**).

Was leisten Ausnahmen in Programmiersprachen?

- Sie leiten den Kontrollfluss an den nächsten passenden Catch-Block weiter.
- Sie wickeln den Aufruf-Stack ab.
- Sie sind ein zusätzlicher Informationskanal vom Gerufenen zum Aufrufer.
- Java-Ausnahmen protokollieren den Aufruf-Stack (printStackTrace)

Ausnahmen sind also nützlich, aber sind wir damit glücklich?

Probleme mit Ausnahmen in Programmiersprachen

- Zahllose Ausnahmen fliegen quer durch das System.
- Viele, manchmal alle catch-Blöcke sind entweder leer oder enthalten unsinnigen Code.
- Eine Unzahl von Ausnahmeklassen machen das System unübersichtlich und schaffen überflüssige Abhängigkeiten.
- Ausnahmen werden für die Rückgabe normaler Ergebnisse missbraucht.
- throw-catch verursacht oft undurchsichtige Strukturen (veredeltes goto); geschachtelte throw-catch-Klauseln machen den Code oft völlig unverständlich.

⇨ Die Behandlung der Ausnahmen muss kontrolliert und immer gleich erfolgen.

Fragen

- Wann verwendet man Ausnahmen, wann Returncodes?
- Wer trägt die Verantwortung für die Behandlung von Ausnahmen?
- Wie werden Fehler/Ausnahmen aus den unteren Schichten hochgereicht?
- Wann verwendet man in Java geprüfte Ausnahmen, wann ungeprüfte?
- Wann soll man eigene Ausnahmeklassen definieren?
- Wie geht man mit Bibliotheken um, die Ausnahmen werfen?

Ausnahmen und Softwarearchitektur

Wann verwendet man Ausnahmen, wann Returncodes?

- Schnittstellen besitzen Methoden
- Jede Methode hat **normale** Ergebnisse und **abnorme** Ergebnisse
- Normale Ergebnisse (und Fehler)
  - passieren dauernd, und
  - sie passen zur Abstraktionsebene der Schnittstelle.
- Abnorme Ergebnisse (Ausnahmen)
  - passieren selten, und
  - sie passen **nicht** zur Abstraktionsebene der Schnittstelle:
    Programmierfehler, technische Fehler, Berechtigungsfehler (?)

⇨ Frage nach Verwendung von Ausnahmen oder Returncodes wird entschieden durch Einteilung in **normale** und **abnorme** Ergebnisse.

Normal vs. abnorm

- Die Entscheidung **normal** vs. **abnorm**
  - hängt ab von der Schnittstelle: Ein und dasselbe Ergebnis ist in der einen Schnittstelle normal, in der anderen abnorm.
  - ist binär: entweder-oder.
  - ist meistens, aber nicht immer klar (z.B. Berechtigungsfehler).
- Erste Kriterien
  - Was sich zur Laufzeit nicht behandeln lässt, ist immer abnorm.
  - Normale Ergebnisse sind Bestandteil der Schnittstelle und gelten für jede Implementierung.
  - Abnorme Ergebnisse sind nicht Bestandteil der Schnittstelle; jede Implementierung hat ihre eigenen abnormen Ergebnisse.
  - Es gibt selten mehr als zwei oder drei normale Ergebnisse; die Anzahl der abnormen Ergebnisse ist unermesslich.
  - Abnorme Ergebnisse erfordern (fast) immer eine Sonderbehandlung.

Optionen der Ausnahmebehandlung

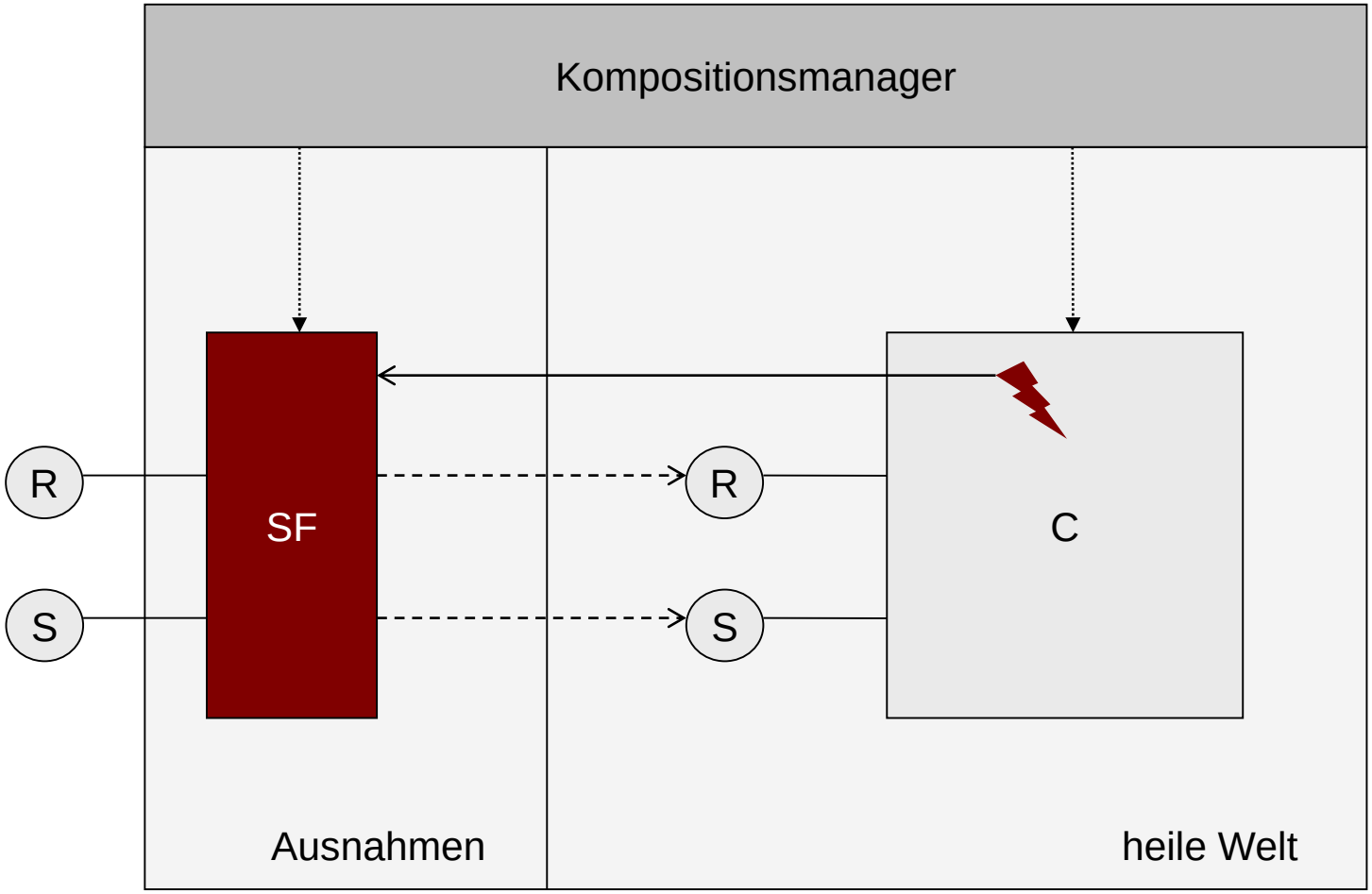
- Protokollieren und Weitermachen (selten).
- Protokollieren und Schaden begrenzen.
- Abwarten und Wiederholen.
- Rekonfiguration.

Danach gibt es nur noch zwei Ausgänge:

(1) Normales Ergebnis (obwohl es vielleicht etwas gedauert hat)

(2) Endgültiges und sicheres Scheitern

Architektur der Ausnahmebehandlung

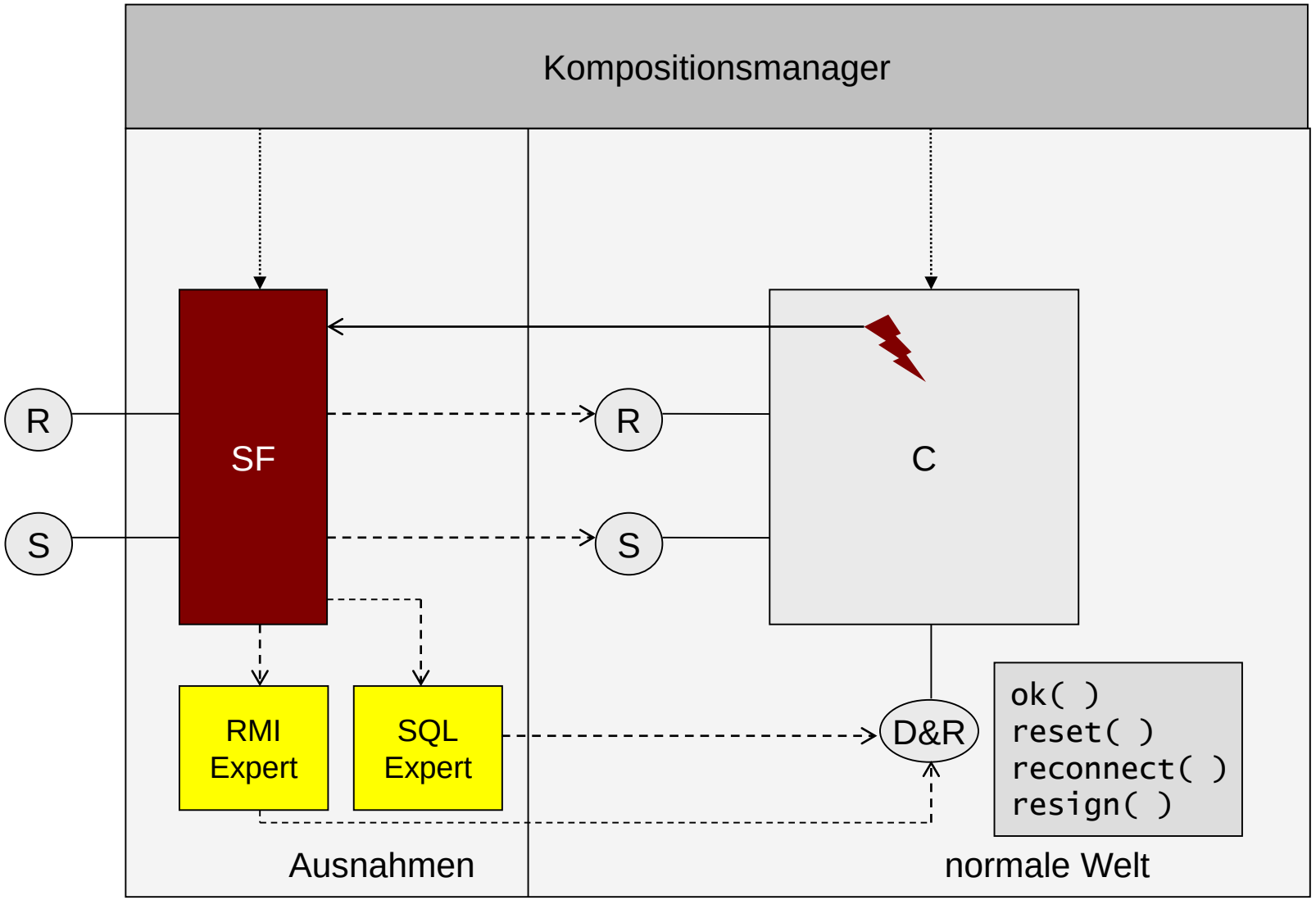


SF = Sicherheitsfassade

Sicherheitsfassade

- SF behandelt alle Ausnahmen die innerhalb einer Komponente auftreten können
- SF ist symmetrisch (exportiert / importiert dieselben Schnittstellen)
- Jedes abnorme Ergebnis wird direkt an die SF weitergeleitet
- Bei kleinen und mittleren Systemen evtl. eine SF für gesamten Anwendungskern ausreichend
- Vorteile
  - Geheimnisprinzip bleibt gewahrt
  - Komponente C besser wieder verwendbar, einfacher, robuster
  - Entwicklungsprozess einfacher (Trennung Komponente – Ausnahmenbehandlung)
- Nachteil
  - Nach Ausnahme kann man nicht an Aufrufstelle fortsetzen

Experten für Diagnose und Reparatur (D&R)

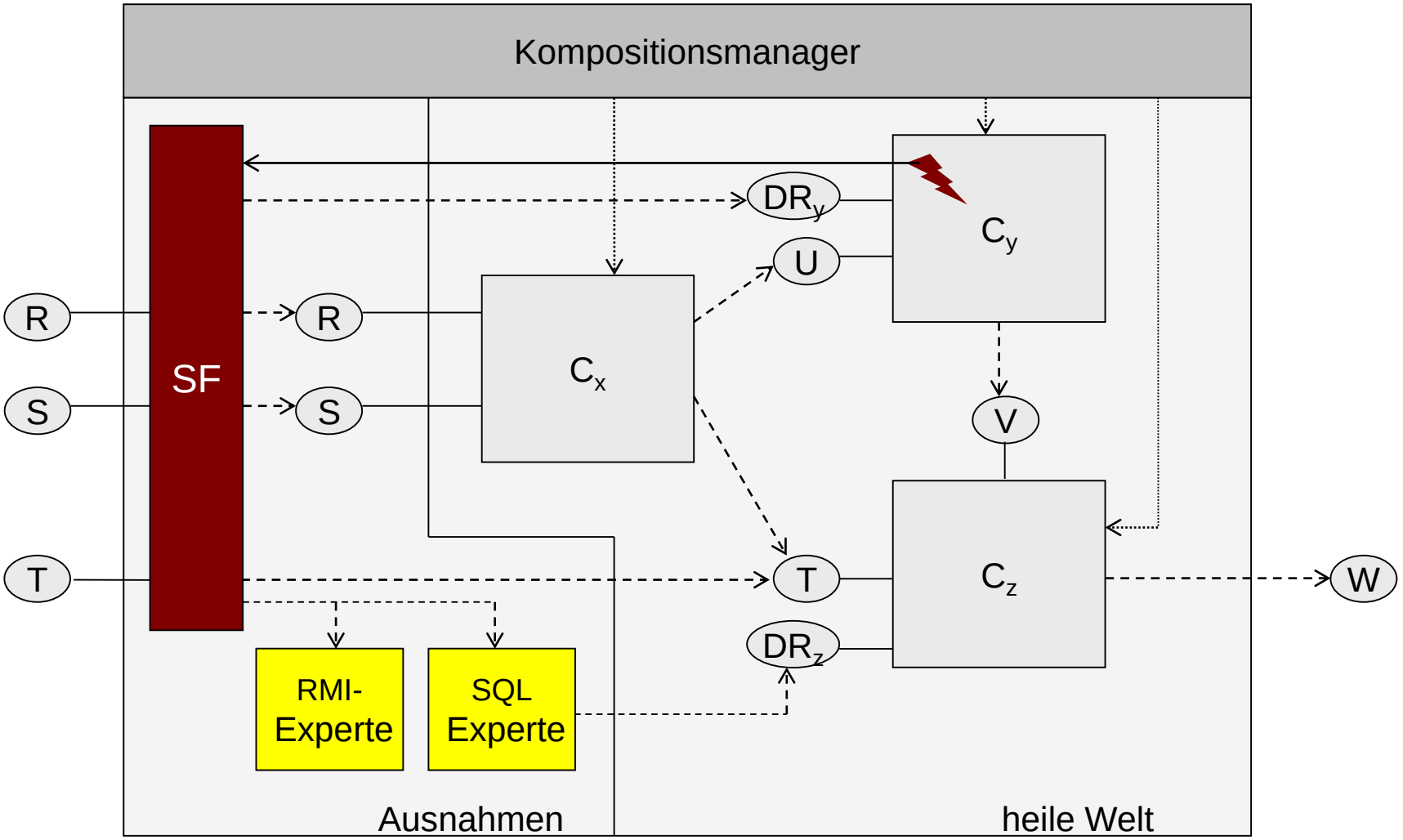


Diagnose und Reparatur

- Jede Komponente kann eigene **D&R-Schnittstelle** anbieten
- D&R-Schnittstelle ist nur dem Kompositionsmanager und SF bekannt
- Zu jeder D&R-Schnittstelle ein oder mehrere **D&R-Experten**
- D&R-Schnittstelle vor allem bei technischen Komponenten hilfreich (z.B. Zugriffschicht)
- Beispiele für D&R-Methoden: ok( ), reset( ), reconnect( ), resign( )
- D&R-Schnittstelle auch für Systemmanagement geeignet (analog zu JMX Java Management Extension)

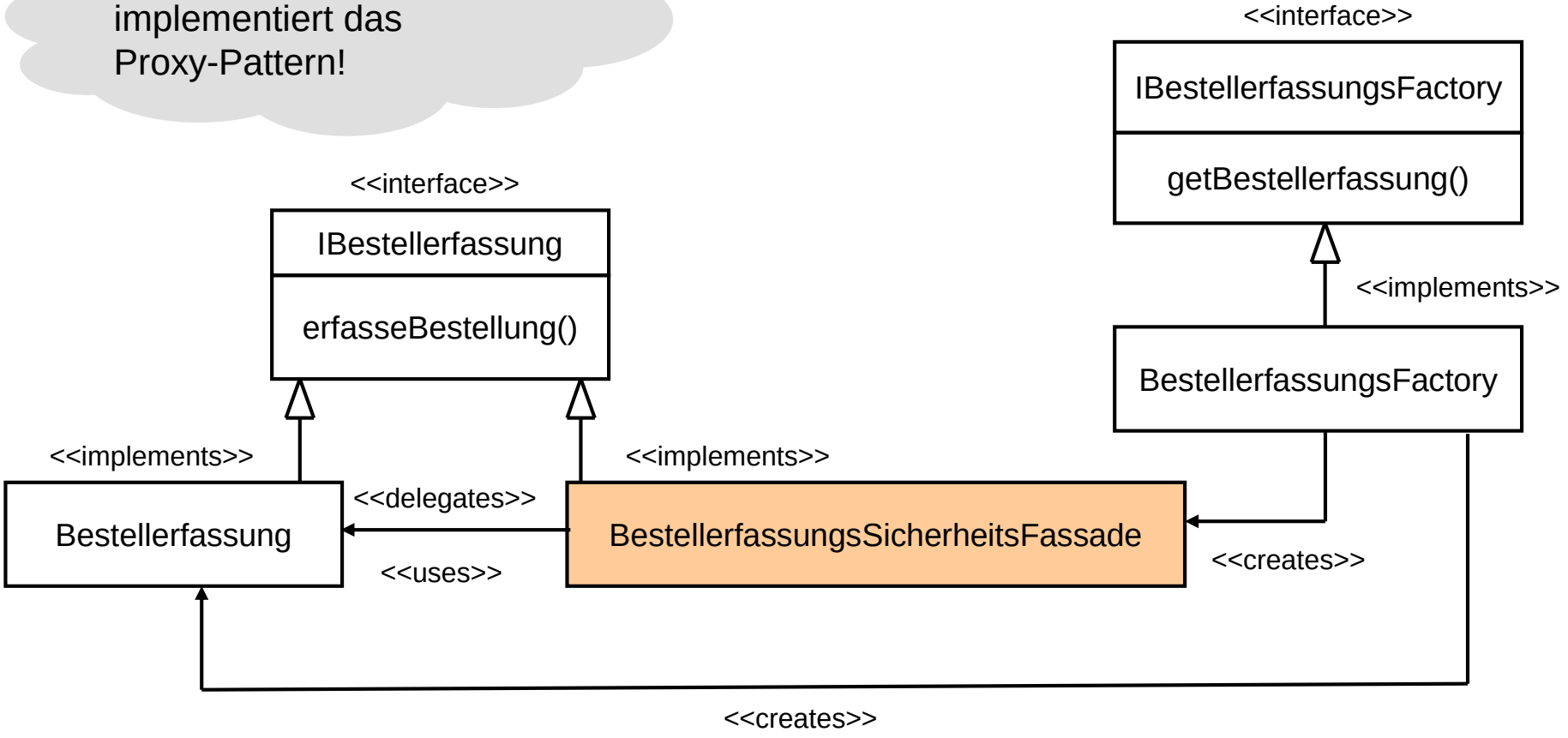
Komposition als Risikogemeinschaft

- Ausnahmen werden zentral behandelt



Beispiel: Sicherheitsfassade

Die Sicherheitsfassade implementiert das Proxy-Pattern!



Sicherheitsfassade: Code Beispiel

```
public class BestellerfassungSicherheitsfassade implements IBestellerfassung{
    private IBestellerfassung  bestellerfassung;

    public BestellerfassungSicherheitsfassade(IBestellerfassung bErfassung) {
        bestellerfassung = bErfassung;
    }
    public void erfasseBestellung(Bestellung b){
        try {
            bestellerfassung.erfasseBestellung(b);
        } catch (Exception e) {
             // Behandle die Ausnahme  TODO
        }
    }
}
```

```
public class BestellerfassungFactory {
    private IBestellerfassung bestellerfassung;

    public BestellerfassungFactory(IBestellerfassung bestellungerfassung) {
        this.bestellerfassung = bestellerfassung;
    }
    public IBestellerfassung getBestellerfassung() {
         return new BestellerfassungSicherheitsFassade(
                        new Bestellerfassung(bestellerfassung));
    }
}
```

Checked vs. Unchecked

Es gibt verschiedene Strategien, welche Art von Exceptions einzusetzen sind. Hier einige Vorschläge:

- **Checked Exceptions** dürfen nur in einem lokalen Kontext eingesetzt werden und müssen an der Grenze der Komponente
  - entweder endgültig behandelt werden oder
  - in eine unchecked Exception (== Runtime Exception) verwandelt werden.
- Grundsätzlich werden von der Anwendung nur **Runtime Exceptions** geworfen, da diese nicht deklariert werden müssen.
  Horrorszenario: 2/3 aller Methoden einer Anwendung tragen die Signatur …
  `throws MyProjectException { … }`
- Anmerkung : C++ und C# haben nur RuntimeExceptions

Checked Exceptions (geprüfte Ausnahmen)

- Checked Exceptions treten in Java in verschiedenen Formen auf, z.B:
- **Syntax:**
  - Ein Konstruktor kann keinen Wert zurückgeben:
    `public Integer(String s) throws NumberFormatException`
  - Viele Methoden geben bereits einen Wert zurück, für einen Fehlercode ist kein Platz:
    `int Integer.parse(String s) throws NumberFormatException`
- **Threads**: (sleep, wait, interrupt)
  - Hier wurden Exceptions zur Thread-Kommunikation missbraucht. Hat mit Ausnahmen im Sinn der Anwendung nichts zu tun.
  - `void sleep(int msec) throws InterruptedException`
- **Schlimme Sachen:**
  - Filesystem kaputt: `boolean createNewFile() throws IOException`
- **Checked Exceptions sollten vom Aufrufer sofort behandelt werden**

Behandlung von normal vs. abnorm in Java

- Normale Ergebnisse meldet man mit
  - Speziellen Rückgabewerten (z.B. null oder -1)
  - Returncodes oder
  - vorgefertigten, geprüften Ausnahmen.
  - Der unmittelbare Aufrufer kümmert sich um die Behandlung.

- Abnorme Ergebnisse meldet man mit
  - ungeprüften Ausnahmen
  - Emergency
  - Die nächste Sicherheitsfassade kümmert sich um die Behandlung.

Normale Ergebnisse mit Returncodes melden

```
public class Returncode {
    public abstract boolean ok( ) ;
    public boolean nok( ) {
         return !ok( );
    }
    ...
}


public class Ok extends Returncode { ... }

public class Nok extends Returncode { ... }



public class MyClass {
    public Returncode foo( ) {
       ...
       String result = bar( );
       if (null == result)
          return new Nok("bar returned null");
       ...
    }
}
```

Normale Ergebnisse mit vorgefertigten Ausnahmen melden

```
public class MyClass {

   public static class ExceptionA extends Exception {}
   public static class ExceptionB extends Exception {}

   private static final ExceptionA exceptionA = new ExceptionA();
   private static final ExceptionB exceptionB = new ExceptionB();

   public void foo( ) throws ExceptionA, ExceptionB {

       if( .. )                  // Fehler A
          throw exceptionA;      // normales Ergebnis

       if( .. )                  // Fehler B
          throw exceptionB;      // normales Ergebnis
   }
}
```

Emergency: Die dokumentierte Katastrophe

```
readFile(String filename) {
    try {
      open the file;
      determine its size;
      allocate that much memory;
      read the file into memory;
      close the file;
    } catch (Exception e) {
        Emergency.now(e);
    }
```

Anwendungscode

Ausnahme-behandlung

```
Emergency.ifTrue(<verbotene Bedingung>, <Info>)
und viele Varianten (ifNull, ifNotNull, ifFalse, ifNok, now )
```

```
public void tuwas() {
    String result = ..

    Emergency.ifNull(result);
```

Abnorme Ergebnisse mit Emergency melden

```
public class MyClass {
    public void foo() {
        ...
        String result = bar();
        Emergency.ifNull(result, "must never happen");
        ...
    }

}
```

```
public class Emergency {
    public static void ifTrue(boolean condition, String message) {
        if (condition)
           throw new EmergencyExcpetion(message);
    }
    public static void ifNull(Object object, String message) {
        if (null == object)
            throw new EmergencyExcpetion(message);
    }
    public static void now(Throwable t) {
        throw new EmergencyExcpetion(t.toString());
    }
        ...

}
```

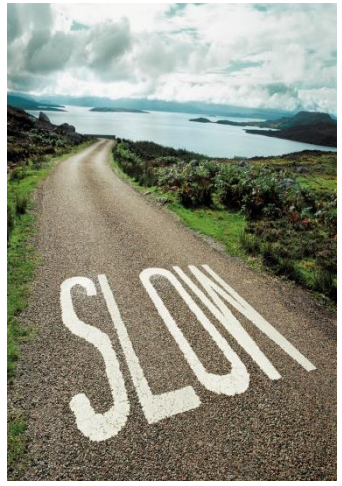
```
public class EmergencyException extends RuntimeException {
    ...

}
```

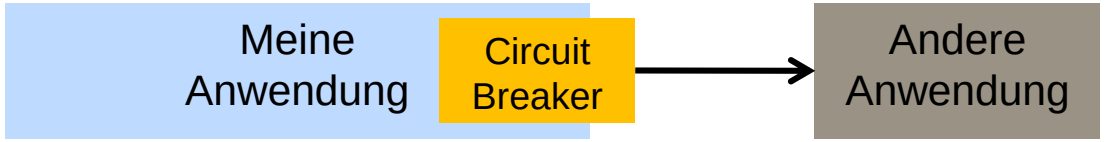
Strategien: Eskalation oder Deeskalation

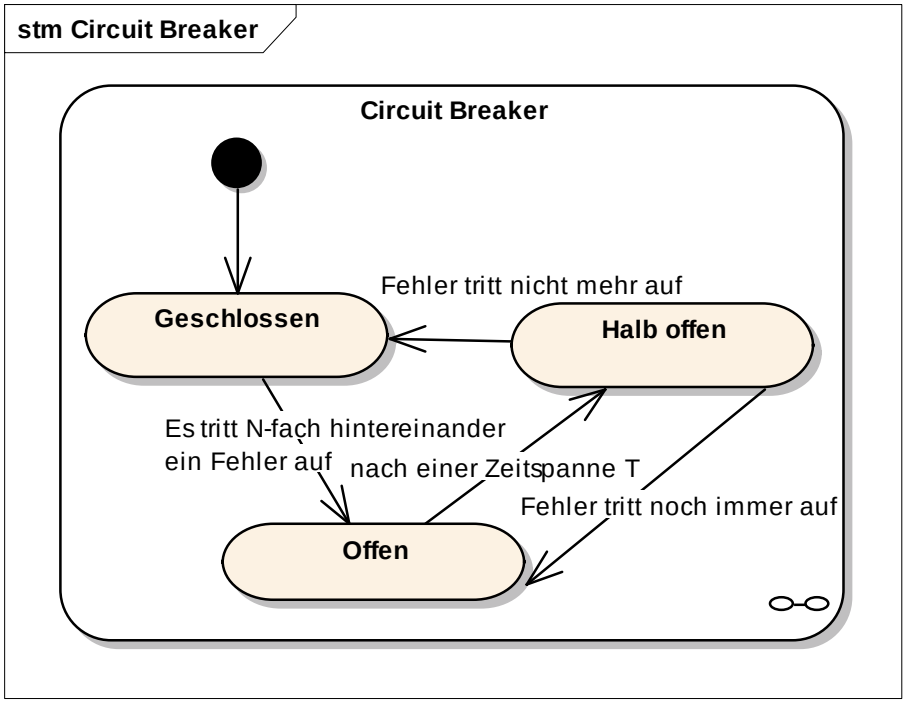
```
try {
   x.tuwas();
}
catch(MyException e) {      // Eskalation
  Emergency.now(e, "tuwas ist schiefgegangen");
}
```


```
try {
   int i = Integer.parseInt( s );
   return true;
}
catch(NumberFormatException e) {  // Deeskalation
   return false;
}
```



Verhindern, dass andere Systeme mein System instabil machen: Circuit-Breaker Pattern





➔ Einheitliche Behandlung von Ausfällen und Verbindungsproblemen.
➔ Zentrale Protokollierung der Zuverlässigkeit der integrierten Systeme.

Quelle: Richardson, Gwaltney; Ship It!

Ausnahmen und Protokollierung

- Welche Informationen sind für die Fehleranalyse relevant?
  - Da nicht immer sicher ist, dass nachvollzogen werden kann, wo der Fehler auftrat, müssen alle relevanten Informationen beim Protokollieren aufgeführt werden:
    Zeitpunkt, Klasse, Fehler-Id, User, Session
  - **kein Stacktrace** in permanenter Protokollierung
  - Stacktrace ist aber notwendig bei der Fehlersuche zur Analyse von Fehlern um Verursacher und Wirkung zusammenführen zu können

- Wann ist man sich sicher, dass eine Ausnahme vorliegt?
  - Rhetorische Frage, man ist sich erst beim endgültigen Scheitern sicher.
    D.h. solange eine Exception fliegt, lässt man sie fliegen, ohne den Stacktrace zu loggen.
  - Der Handler ist für das protokollieren (und nur der) zuständig.
  - Doppelt und mehrfach geloggte Ausnahmen verfälschen später die Statistik bei einer Analyse der Logfiles.

Hinweise für eine gute Protokollierung

- Verwenden Sie immer dasselbe Format für die Log-Einträge, um eine automatische Analyse zu erleichtern.
- Vergeben Sie eindeutige Fehler-IDs, die dem Betrieb (und ihnen) die Zuordnung erleichtern. Mit Hilfe dieser IDs kann auch eine Internationalisierung von Fehlern beim Client erleichtert werden.
- Sorgen Sie dafür, dass alle Informationen in eine Zeile passen.
- Pro Schicht sollte geloggt werden.
  D.h. pro Session-Bean auf dem Applicationserver, auf dem Rich Client auch noch mal.
- Verwenden Sie ein Standard-Logging-Framework. Wir empfehlen:
  - JDK 1.4 mit integriertem Logging
  - Apache log4j (mehr Handler, andere Levels)

Praktische Aspekte: Loglevels

Loglevel gibt an, ob eine Ausnahme geloggt werden soll

Log4J Loglevel:

- Trace: Alles wird geloggt, vor allem Entry und Exit jeder Methode (nie benötigt)
- Debug: Debuginformationen, die während der Entwicklung hilfreich sein können.
- Info: Statements, die fachliche Abläufe nachvollziehen helfen
- Warn: Hinweis auf (behandelte) Fehlersituationen
- Error: Hinweis auf (nicht behandelte) Fehlersituation, bei der Sicherheitsfassade mit Stacktrace und Fehler-Id, sonst am besten gar nicht

Ausnahmen und Tests

- Erwartete Ausnahmen werden in JUnit abgefangen und ausgewertet
- Alle anderen werden nicht gefangen
- Generell sollte man Methoden die Ausnahmen erzeugen können auf alle realistischen Fälle testen
- Dabei muss das System alle erfolgten Ausnahmen angemessen behandeln können
- Man unterscheidet zwischen Methoden die die Ausnahme erzeugen (throw / throws) und Methoden die die Ausnahme abhandeln (try / catch)
  - Bei Ausnahme erzeugenden Methoden muss der try/catch-Block im JUnit Test geschrieben werden
  - Bei Ausnahme behandelnden Methoden sollte der Test prüfen ob die Ausnahme angemessen abgefangen wurde

Ausnahmen in Junit (1)

- Der Konstruktor „Rational(String value)“ wirft eine Exception wenn „value“ einen nicht validen Wert enthält.

```
public void testConstructors() {
     …
    Rational r1 = new Rational ("4/6");
      …
      try {
         Rational r5 = new Rational("4:6");
         // In diesem Konstruktor ist lediglich ein / als
         // Sonderzeichen erlaubt
         fail();  // dieser Code darf nie erreicht werden,
                  // da der Konstruktor bereits vorher eine
                  // Exception werfen muss!
      } catch (Exception e){
         logger.info(„IllegalArgumentException n Konstruktor für r5
                      erkannt");
      }
      …
   }
```

Ausnahmen in Junit (2)

- „saveDocument“ speichert in eine Datei, diese Methode wird wie folgt geprüft

```
public String saveDocument(String content, String document){
   try {
      FileOutputStream fs = new FileOutputStream(document);
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(content);
      os.close();
      return "Datei erfolgreich gespeichert!";
   } catch (IOException e) {
         return "Fehler beim Speichern!";
   }
}
```

```
public void testSaveDocument() {
   …
   // date.ser ist eine verfügbare Datei
   String result = saveDocument("Text", "date.ser");
   assertEquals(result, "Datei erfolgreich gespeichert!");
   // readonly.ser ist nicht verfügbar (read only)
   result = saveDocument("Text", "readonly.ser");
   assertEquals(result, "Fehler beim Speichern!");
   …
}
```

Regeln

(1) Unterscheide normale und abnorme Ergebnisse.

(2) Entdecke Ausnahmen so früh wie möglich.

(3) Weise verletzte Vorbedingungen mit einer Ausnahme zurück.

(4) Setze alle Parameter als nicht null voraus.

(5) Bilde sinnvolle Risikogemeinschaften.

(6) Ausnahmen werden zentral behandelt (z.B. durch Sicherheitsfassaden) – und sonst von niemand.

(7) Melde Fehler (= normale Ergebnisse) über spezielle Werte, Returncodes oder vorgefertigte Ausnahmen.

(8) Behandle Fehler sofort beim unmittelbaren Aufrufer.

(9) Selbstdefinierte Ausnahmeklassen sind nur gerechtfertigt, wenn sie in eigenen Catch-Klauseln behandelt werden.

Zusammenfassung Fehler und Ausnahmen

- Die Behandlung von Ausnahmen muss in einem Projekt geregelt und einheitlich sein
- Wir unterscheiden zwischen normalen Ergebnissen (z.B. Fehlern) und abnormen (Ausnahmen)
- Es gibt mehrere Optionen bei der Ausnahmebehandlung (Protokollieren, Wiederholen, Rekonfiguration, Schaden begrenzen)
- Ausnahmen einer Komponente werden an einer zentralen Stelle behandelt (Sicherheitsfassade)
- Ausnahmen müssen korrekt protokolliert werden (Logging)