Fehlerbehandlung in mehrschichtigen Systemen

Schichten und Exceptions

Christian Rehn aka R2C2

Delphi-Tage 2012

Vorstellung [Folie 2]

- Christian Rehn aka R2C2
- Informatikstudent an der TU Kaiserslautern
- Moderator und Redakteur bei Delphi-Treff
- http://www.christian-rehn.de/

[1]

Organisatorisches [Folie 3]

- Ein Grundverständnis der OOP wird vorausgesetzt
- Folien enthalten nur wenig Text
  - Besser für Präsentation
  - Ausführliche Vortragsnotizen online: http://www.christian-rehn.de/
- Feedback erwünscht (persönlich, per Mail, im Forum, im Blog, ...)

[1] Im Gegensatz zum Rest stehen die Logos selbstverständlich nicht unter Creative Commons.

Überblick [Folie 4]

Inhaltsverzeichnis

1. Motivation

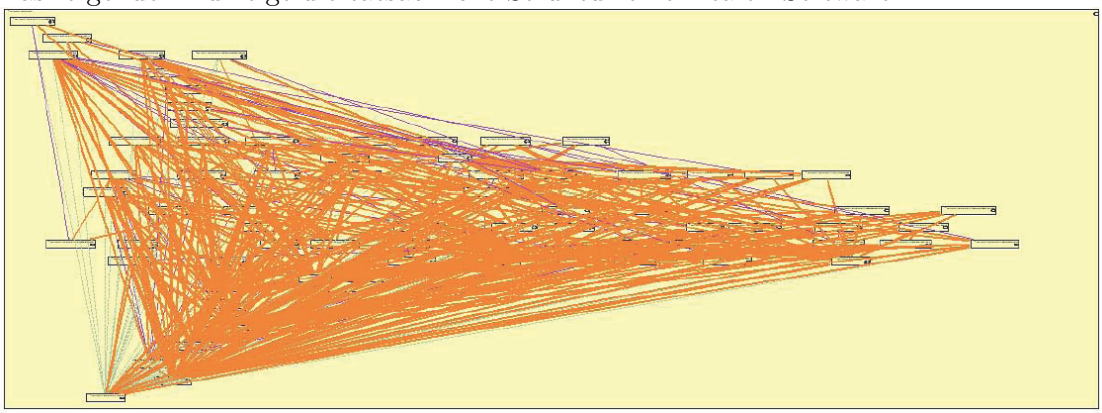
1.1. Warum Schichten?

Warum überhaupt nachdenken? (1/2) [Folie 6]

Warum sollten wir uns überhaupt Gedanken über so etwas machen?

Warum überhaupt nachdenken? (2/2) [Folie 7]

Das folgende Bild zeigt die tatsächliche Struktur einer realen Software.



[2]

Und das ist nur ein Subsystem von zwanzig. Wenn Software so aussieht, führt das fast unweigerlich zu unnötigen Bugs.

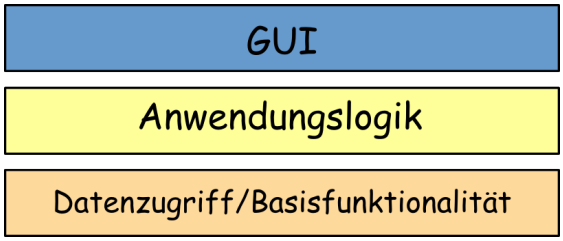
Das Bild wurde von einer Software des Fraunhofer IESE [8] [4] aus dem Code generiert, zeigt also die **tatsächlichen Abhängigkeiten**. Es ist anzunehmen, dass die Entwickler *keine* Idioten waren. Wahrscheinlich haben sie sich sogar Gedanken über die Architektur gemacht. Das zeigt, dass es schwer ist, die Architektur nicht aus den Augen zu verlieren – insbesondere, wenn man im Team und unter Projektbedingungen (Zeitdruck, etc.) arbeitet.

Wenn man also schon mit Nachdenken im Chaos landen kann, ist es ohne Nachdenken wohl noch viel schlimmer. Oft merkt man das nicht direkt, wenn man ein Problem auf Architekturebene hat. Alles funktioniert erstmal irgendwie trotzdem. – Und ein halbes Jahr später kriegt man ernsthafte Schwierigkeiten, die mit der Zeit immer schlimmer werden.

[2] Danke an das Fraunhofer IESE für das Bild

Deshalb ist es sinnvoll, sich immer vorzustellen, dass in einem halben Jahr irgend ein **„Halbidiot“** die Aufgabe erhält, die Software anzupassen. Wenn selbst dieser Halbidiot nicht mehr Fehler einbaut, als er behebt, ist die Software wartbar. Und wenn nicht, hat man ein potenzielles Problem. Oft genug sind wir nämlich selbst dieser Halbidiot. Wir erinnern uns nicht mehr so genau an alles, vergessen gewissen Einschränkungen, machen inkonsistente Änderungen, etc.

Schichten [Folie 8]



Schichten sind eine und vielleicht sogar die wichtigste Art der Strukturierung von Software.

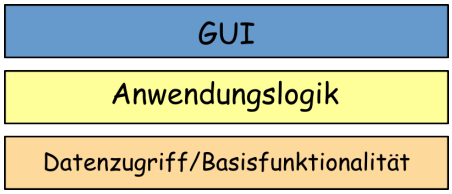
1.2. Warum Exceptions?

Warum Exceptions? [Folie 9]

```
try
  while not EndOfTalk do
  begin
    Present(slide);
    GoToNextSlide;
  end;
except
  on e: EFireAlarm do
  begin
    Panic;
    Shout(e.Message);
  end;
end;
```

1.3. Zusammen

Und wie passt das zusammen? [Folie 10]




```
on e: EFireAlarm do
begin
  Panic;
  Shout(e.Message);
end;
```

2. Abhängigkeiten und Schichten

2.1. Abhängigkeiten

Ripple Effects [Folie 13]

[3]

Ripple Effects

Änderungen an einer Stelle machen Änderungen an weiteren Stellen nötig, die wiederum weitere Änderungen nötig machen usw. Der Code wird dadurch fragil. Es schleichen sich leicht Fehler ein.

- Analogie: Ein Wassertropfen erzeugt sich allmählich ausbreitende kleine Wellen (engl. *ripples*).
- Ripple Effects entstehen durch zu viele und zu starke Abhängigkeiten

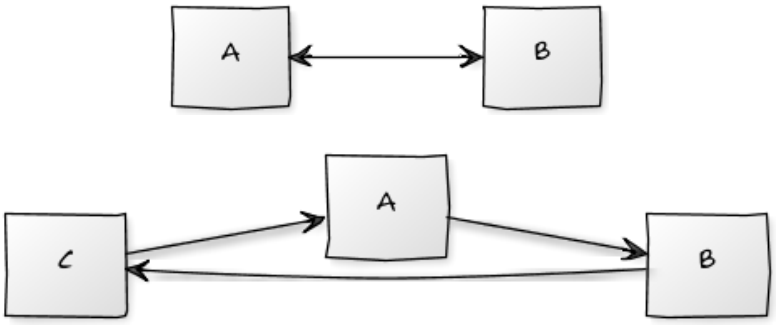
Abhängigkeiten [Folie 14]

```
myFancyOpenDialog.ShellTreeView.Path := pathToMyDocuments;
myFancyOpenDialog.FileNameEdit.Text := 'newFile.txt';
if  myFancyOpenDialog.ShowModal = mrOK then
begin
  pathToSaveFile := myFancyOpenDialog.ShellTreeView.Path +
      myFancyOpenDialog.FileNameEdit.Text;
  SomeMemo.Lines.LoadFromFile(pathToSaveFile);
end;
```

[3] CC-BY-SA Rainer Zenz http://commons.wikimedia.org/wiki/File:2006-01-14_Surface_waves-2.jpg

Das obige Beispiel zeigt eine schlechte Implementierung eines eigenen OpenDialogs. Der Code enthält sehr viele und vor allem unnötige Abhängigkeiten und erzeugt so Ripple Effects. Möchte man die optische Erscheinung des Dialogs ändern (z. B. durch Verwendung anderer Komponenten), so müsste man *jeden* Aufruf des Dialogs anpassen. Besser: Man versteckt die Details hinter passenden Properties, wie das `TOpenDialog` auch tut.

Zyklische Abhängigkeiten [Folie 15]



- Zyklische Abhängigkeiten erzeugen starke Kopplungen
- Ein Modul des Zyklus hängt automatisch von allen anderen ab
- Bessere Lösung:
  - Abhängigkeitsbeziehungen sollten azyklisch sein (Abhängigkeitsgraph ist ein DAG[4])
  - Besser noch: hierarchisch (Abhängigkeitsgraph ist ein Baum)

Zirkuläre Unit-Referenz [Folie 16]

```
[DCC Fataler Fehler] UnitXY.pas(7): F2047 Zirkuläre Unit-Referenz auf 'UnitXY'
```

Ein solcher Fehler ist also oft ein Hinweis auf ein Design-Problem.

[4] http://en.wikipedia.org/wiki/Directed_acyclic_graph

2.2. Grobstruktur

Um den genannten Problemen aus dem Weg zu gehen, ist es nötig, sich über die Architektur der Software Gedanken zu machen.

Architektur [Folie 17]

Architektur: Definition des SEI
The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. [2]

Architektur: Meine Definition
Die Softwarearchitektur beschreibt die groben Strukturen der Software und definiert wie man als Entwickler von der Software *denkt*.

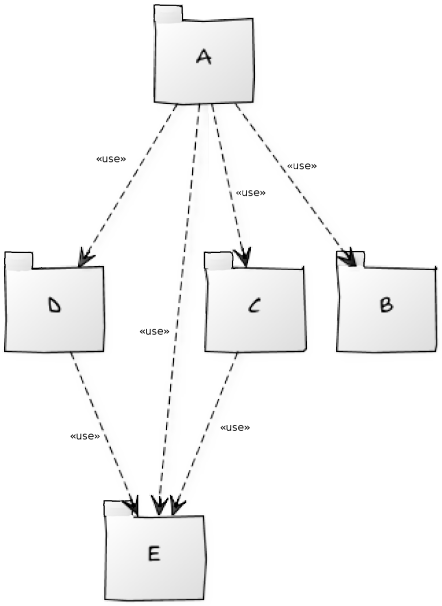
Definitionen für den Begriff „Softwarearchitektur“ gibt es wie Sand am Meer. Die des SEI ist vermutlich die bekannteste. Genau genommen gefällt mir diese Definition nicht sonderlich, weil ich sie für akademisch nichts-sagend halte. Deshalb habe ich meine eigene.

Grobstruktur [Folie 18]

Grobstruktur als Teil der Architektur

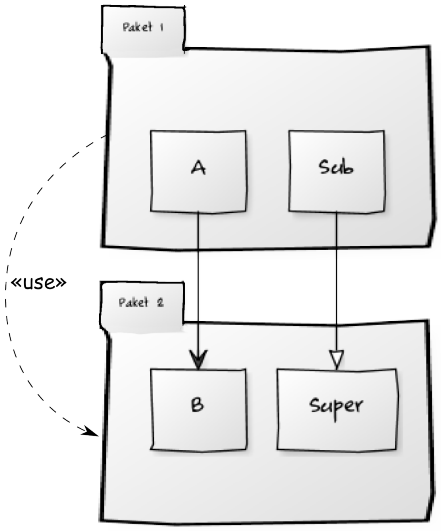
Architektur ist mehr, als nur eine grobe Struktur der Software. Insbesondere hat Software nicht nur eine Struktur, sondern viele Strukturen. So wie ein Haus nicht nur einen Grundriss hat, sondern auch eine Innenarchitektur, eine Struktur der Wasser- und Abwasserleitungen, eine Struktur der Stromleitungen, etc. Hier betrachten wir nur eine dieser Strukturen: Eine grobe statische Modulstruktur.

Kommunikation beschränken (1/2) [Folie 19]



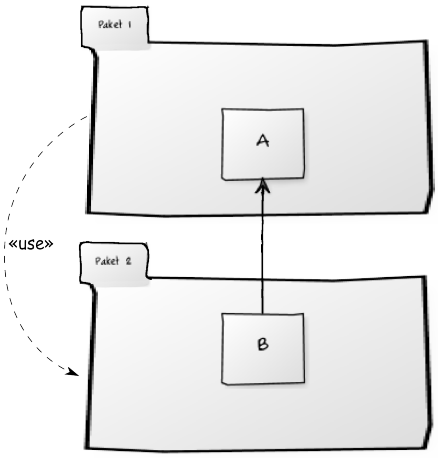
- Eine gute Architektur schränkt die Kommunikation zwischen den Modulen ein und verbietet gewisse Abhängigkeiten.
- Das obige Paketdiagramm zeigt eine Grobstruktur für eine Software. Klassen werden zu logischen Einheiten (in UML: „Paketen“; im Beispieldiagramm genannt A, B, C, D und E) gruppiert. Solche Pakete können „GUI“, „Datenbankzugriff“, „Anwendungslogik“, etc. sein.
- Die Architektur definiert Regeln, die für alle Module gelten. Beispiele:
  - „Module aus Paket A dürfen auf Module aus Paket B zugreifen (aber nicht umgekehrt).“
  - „Nur Module im GUI-Paket dürfen direkt mit dem Benutzer interagieren.“
  - „Die Kommunikation zwischen Client und Server geschieht ausschließlich über Webservices.“
  - „Ein Modul soll nicht gleichzeitig Datenquelle und Datensenke sein.“
  - ...
- Dadurch, dass die Abhängigkeiten gewissen Regeln unterliegen, wird die Struktur klarer und Abhängigkeiten (und damit auch RippleEffects) werden reduziert.

Kommunikation beschränken (2/2) [Folie 20]



- In UML-Klassendiagrammen zeigen alle Pfeile in Richtung der Abhängigkeit.
  - Deshalb zeigen Vererbungspfeile („Generalisierungspfeile„) in Richtung der Basisklasse, obwohl anders herum vererbt wird.
- Paketdiagramme zeigen größere (aber ebenfalls statische) Strukturen.
- Beziehungen, die zwischen Paketen gelten, müssen von den darin enthaltenen Klassen erfüllt werden.
  - Um das zu verdeutlichen, kann man Klassen in Pakete „schachteln“.

Dependency Inversion: Events [Folie 21]

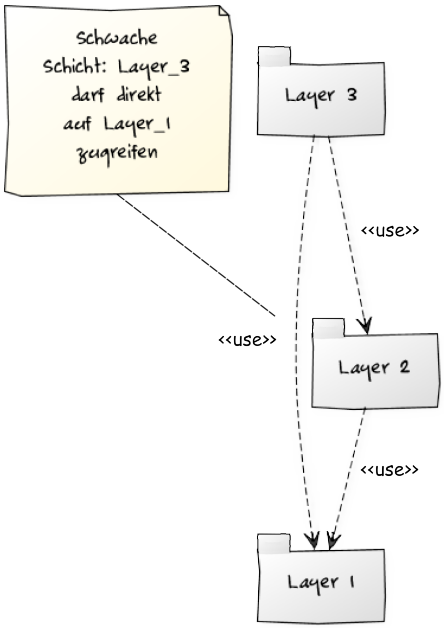



```
property OnSomeEvent: TNotifyEvent
    read FOnSomeEvent write
    FOnSomeEvent;
```

- Manchmal möchte man Aufrufe von niedrigeren in höhere Schichten realisieren, obwohl man das eigentlich nicht „darf“. So etwas wie in obiger Grafik ist also nicht möglich.
- Lösungen:
  - Callbacks, Events
  - In anderen Sprachen: Closures, Delegates, ...
  - (objektorientierte) Dependency Inversion (sprachunabhängig)
- In Delphi verwendet man i. d. R. Events. Im Anhang folgt noch eine kurze Erklärung zur objektorientierten Dependency Inversion.

2.3. Schichten

Schichten [Folie 22]



- Häufig benutzt man Schichtstrukturen um die Kommunikation zu begrenzen

  - Obere Schichten dürfen auf darunter liegende zugreifen; aber *niemals* umgekehrt
  - Strengere Variante: Nur auf die direkt darunter liegende Schicht darf zugegriffen werden
- Schichten sind Abstraktionen
  - Obere Schichten (z. B. die GUI) sind konkret und spezifisch für eine Anwendung
  - Untere Schichten werden immer allgemeiner
- Schichten können selbst wieder in Schichten aufgeteilt werden.

Virtuelle Maschinen [Folie 23]

Schichten sind logisch gesehen virtuelle Maschinen/Plattformen

- Das Betriebssystem (WinAPI) ist eine Schicht, die von der Hardware abstrahiert. So kann man als Entwickler die Schnittstelle des Betriebssystems als Plattform nutzen.
- Die RTL/VCL (neuerdings auch Firemonkey) ist eine Schicht, die vom Betriebssystem abstrahiert. So kann ein Delphi-Entwickler diese API als Plattform nutzen.

- Auch ein Domänenmodell (natürliche Klassen, die die Anwendungsdomäne modellieren; so sind z. B. die Klassen `Library`, `Book`, `Reader`, etc. Teil eines Domänenmodells für Bibliotheken.) ist eine Schicht. Darüberliegende Schichten können dieses als Plattform nutzen, können also so tun, als würde die darunter liegende Maschine schon die Anwendungsdomäne kennen. Im Beispiel: Wir können uns vorstellen, wir hätten einen Computer, der bereits weiß, was Bücher sind.

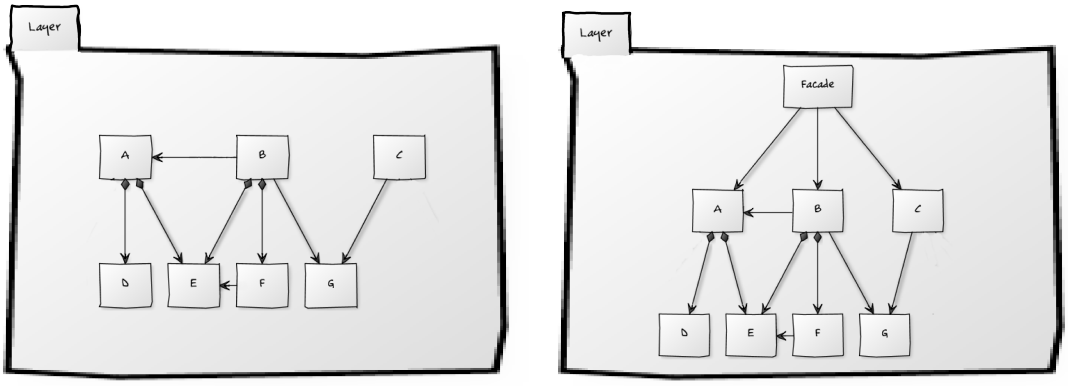
Law of Leaky Abstractions [Folie 24]

Law of Leaky Abstractions
All non-trivial abstractions, to some degree, are leaky. [11]

Abstraktionen (und damit Schichten) sind quasi nie perfekt. Bestimmte Aspekte der unteren Schichten haben indirekt Auswirkungen auf obere. Das ist nicht gut, weil man dann die Realisierung der unteren Schichten bedenken muss, obwohl man ja eigentlich eine Abstraktion hat, die einen davon befreien sollte. Vollständig verhindern lässt sich das aber fast nie. Ein typisches Beispiel für einen solchen Aspekt ist Performance. Iteriert man beispielsweise über ein zweidimensionales Array (oder eine Array-Property), so kann es performanceteschnich einen Unterschied machen, ob man das Array vertikal oder horizontal durchläuft. Die Abstraktion ist nicht perfekt, da man solche Aspekte nicht vollständig wegabstrahieren kann.

Die Konsequenz daraus sollte natürlich nicht sein, aufzugeben und gar keine Abstraktionen mehr zu verwenden. Vielmehr sollte man Abstraktionen möglichst gut machen, sich dabei aber der Grenzen bewusst sein.

Schichten und Fassaden [Folie 25]



- Schichten sind erstmal nur eine konzeptionelle Gruppierung und sind im Code i. d. R. nicht direkt ersichtlich
  - Nichtsdestotrotz kann es aber sinnvoll sein, alle Units einer Schicht in ein separates Verzeichnis zu legen
- Möchte man erreichen, dass Schichten austauschbar sind, so müssen sie definierte Schnittstellen besitzen
- Dazu eignet sich das Facade Pattern [5] (zu Patterns allgemein siehe [10])
  - Anstatt alle Klassen einer Schicht für höhere Schichten zugreifbar zu machen, wird eine meist zusätzliche, künstliche Fassadenklasse erstellt. Alle Zugriffe auf die Schicht laufen über diese Fassadenklasse, d. h. nur sie ist nach außen hin sichtbar.
  - Ändert sich etwas schichtenintern, hat das i. d. R. keinerlei Auswirkungen auf andere Schichten, wenn die Schnittstelle der Fassadenklasse erhalten bleibt. RippleEffects werden wirksam vermieden.
  - Die Schicht kann auch durch eine vollkommen andere ersetzt werden, wenn diese die gleiche Schnittstelle als Fassade anbietet.
  - Um Austauschbarkeit noch weiter zu verbessern, ist es sinnvoll, auch die Fassadenklasse nicht direkt zugreifbar zu machen. Stattdessen wird von außen nur eine abstrakte Basisklasse oder ein Interface deklariert.

Überall Schichten [Folie 26]

- Netzwerk:
  - Physical Layer, Link Layer, Network Layer, Transport Layer, Application Layer
- APIs und Frameworks:
  - x86, WinAPI, RTL, VCL/FM
  - x86, WinAPI, CLR, .NET-Framework, SWF/WPF
- Typische Informationssysteme
  - GUI, Anwendungslogik, Datenzugriff
- ...

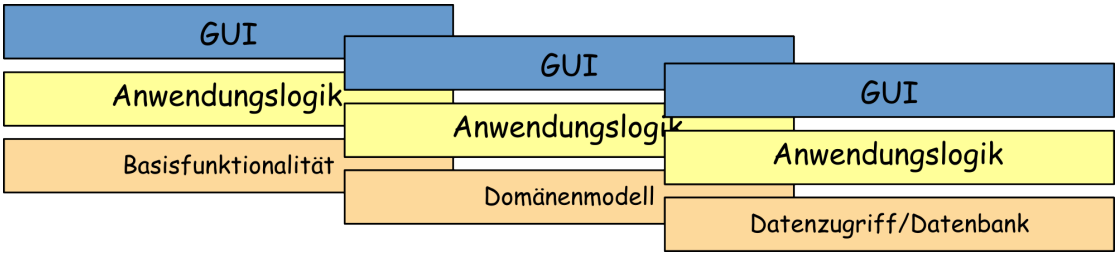
Schichten gibt es quasi überall. Wir verwenden sie ständig und ohne darüber nachzudenken. Letztendlich sieht man hier die Anwendung des Architekturmusters „Layers“ [3].

1, 2, 3, . . . , n Schichten [Folie 27]

- Eine Schicht: Alles in den Formularklassen/alle Klassen dürfen überall hin zugreifen ⇒ Chaos
- Zwei Schichten: z. B. Formular + Datenmodul
- Drei Schichten: Formular + Anwendungslogik + Basisschicht (oder ähnlich)
- . . . weitere Unterteilungen möglich . . .

Achtung: Zu viele Schichten sind kontraproduktiv. Insbesondere machen sie das Design komplexer, das System langsamer und blähen den Code auf. Zu wenige Schichten sind Chaos, zu viele Schichten auch.

Die typische Dreischichtenarchitektur [Folie 28]



Die typischen Delphi-Programme sind wohl Informationssysteme (Warenwirtschaftssystem und ähnliches; eine Datenbank, eine GUI und diverse Daten, die hin- und hergeschoben werden). Für Informationssysteme hat sich die typische Dreischichtenarchitektur etabliert. Die Abgrenzung der drei Schichten variiert dabei ein wenig.

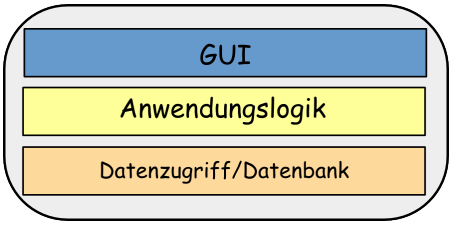
Multi-Tier? [Folie 29]

Layers vs. Tiers [Folie 30]

- Layer: logische Schicht
- Tier: physische Schicht

„Layer“ und „Tier“ lässt sich beides mit „Schicht“ übersetzen. In der Regel versteht man aber unter einem Layer eine logische, d. h. konzeptionelle Schicht, während ein Tier eine physische Schicht ist, d. h. sich zumindest theoretisch auf unterschiedliche Rechner deployen lässt.

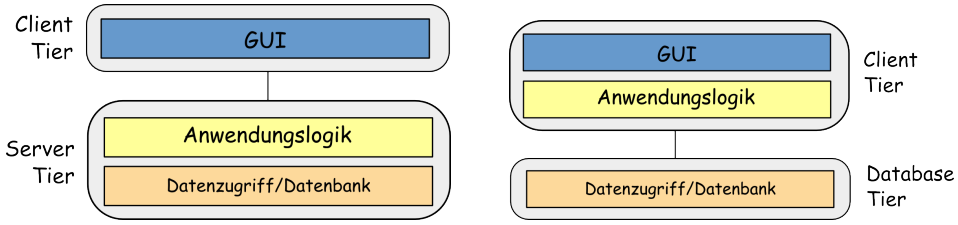
1-Tier [Folie 31]



- 1-Tier: Alles auf einem Rechner
- Beispiele:
  - Verwendung eingebetteter Datenbanken (SQLite & Co.), flat files, o. ä.

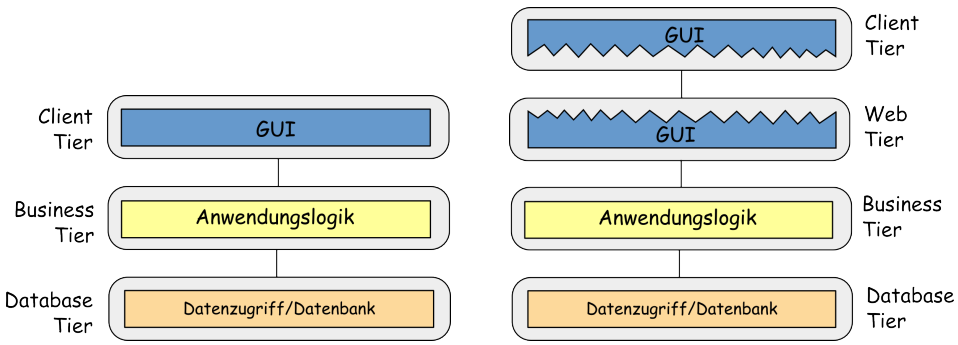
- Einfaches Programm ohne Datenbank
- Mainframes

2-Tier [Folie 32]



- 2-Tier: Die Software ist in Client und Server aufgeteilt
- Je nachdem, ob die Anwendungslogik eher auf Client- oder auf Serverseite liegt, ist es ein thin client oder ein fat client
  - 2-Tier (thin client): Client + Anwendungsserver
  - 2-Tier (fat client): Anwendungsprogramm + Datenbankserver
- Viele Delphi-Programme folgen dem Fat-Client-Modell: Delphi-Programm + Datenbankserver (MySQL, Firebird, o. ä.)
- Selbstverständlich kann es mehr als einen Client geben. Aber es kann auch mehrere Server geben (Replizierte DBs zur Lastverteilung). „Tier“ ist also nicht gleichzusetzen mit „Node“, also Rechner.

3-Tier, 4-Tier, n-Tier [Folie 33]



- 3-Tier: (Thin-)Client + Anwendungsserver + Datenbankserver
- 4-Tier: (Thin-)Client im Browser + Webanwendung + Anwendungsserver + Datenbankserver

3. Exceptions

3.1. Ausnahmesituationen

Ausnahmesituationen [Folie 35]

Wenn doch alles nur normal wäre...

Eine Ausnahmesituation ist eine, die sich vom „Normalfall“ unterscheidet. Demnach ist eine Ausnahmesituation also ein i. d. R. unerwünschter Zustand.

Ausnahmesituationen: Arten von Ausnahmen [Folie 36]

Verhinderbar oder nicht verhinderbar; das ist hier die Frage.

Grob lassen sich verhinderbare und nicht-verhinderbare Ausnahmen unterscheiden. Das Auftreten einer verhinderbaren Ausnahmesituation ist meist ein Bug.

Beispiele:

- Ein nicht zu verhindernder unerwünschter Zustand, den die Software selbst erzeugt (z. B. zwei User wollen sich gleichzeitig unter dem selben Namen registrieren)
- Ein nicht zu verhindernder unerwünschter Zustand der Umgebung (z. B. keine Leseberechtigung auf Datei oder ein Abbruch der Netzwerkverbindung)
- Eigener Bug (z. B. Zugriff auf nicht erzeugtes Objekt erzeugt `EAccessViolation`)
- Bugs des Aufrufers (z. B. jemand anderes ruft unsere Funktion mit falschen Parametern auf)
- Bugs des Aufgerufenen (z. B. eine Fremdkomponente, die nicht richtig funktioniert)

Möglichkeiten der Ausnahmebehandlung [Folie 37]

- Boolean-Rückgabewerte
- Fehlercodes

- Fehlerzustände
- Fehlerbehandlungsroutinen
- Assertions
- Exceptions

Boolean-Rückgabewerte [Folie 38]

```
if OpenDialog.Execute then
begin
  ...
end;
```

Vorteil:

- einfach

Nachteile:

- Wenig aussagekräftig
- Stört den Lesefluss („if execute“? Was soll das heißen?)
- Rückgabewert kann nicht anderweitig genutzt werden
- Tiefe Schachtelung bei mehreren Aufrufen hintereinander
- Fehlerbehandlung kann leicht vergessen werden

Fehlercodes [Folie 39]

```
const
  SHELLEXECUTE_MAX_ERROR = 32;

...

err := ShellExecute(...);
if err <= SHELLEXECUTE_MAX_ERROR then // something bad happened
```

```
begin
  case err of
    ERROR_FILE_NOT_FOUND: ...
    ERROR_PATH_NOT_FOUND: ...
    ERROR_BAD_FORMAT: ...
  else
    ...
  end;
end;
```

Vorteil:

- relativ einfach

Nachteile:

- Stört den Lesefluss
- Es gibt immer noch Leute, die keine Konstanten verwenden :-(
- Überschneidungen mit anderen Fehlercodes
  - Was ist an folgendem Code falsch? `if ShellExecute(...) = ERROR_SUCCESS then`
- Wird der Rückgabewert zusätzlich noch anders benutzt (also als Datum), entsteht eine Hybridkopplung[5]
- Tiefe Schachtelung bei mehreren Aufrufen hintereinander
- Fehlerbehandlung kann leicht vergessen werden

Fehlerzustände [Folie 40]

```
DoSomething(...);
if GetLastError <> NO_ERROR then
begin
  ...
end;
```

[5] Siehe hierzu mein Vortrag von Den Delphi-Tagen 2011: http://www.christian-rehn.de/2011/09/11/enbugging-wie-wir-fehler-machen-und-wie-wir-sie-vermeiden/

Fehlercode nicht in Rückgabewert, sondern per Funktion/Property o. ä. abfragbar.
Vorteile:

- relativ einfach
- Rückgabewert kann anderweitig benutzt werden

Nachteile:

- Es gibt immer noch Leute, die keine Konstanten verwenden :-(
- Überschneidungen mit anderen Fehlercodes
- Tiefe Schachtelung bei mehreren Aufrufen hintereinander
- Fehlerbehandlung kann leicht vergessen werden

Fehlerbehandlungsroutinen [Folie 41]

Event: OnError

Vorteile:

- Relativ einfach
- Eignet sich gut für Komponenten
- Trennt Fehlerfall vom Normalfall

Nachteile:

- Kann unübersichtlich werden, weil Fehlerursache und Fehlerbehandlung räumlich relativ stark getrennt werden
- Komplexe Kontrollflüsse nur sehr unschön zu realisieren
- Fehlerbehandlung kann leicht vergessen werden

Assertions [Folie 42]

```
procedure TSomeClass.DoSomething(param: TSomeObject);
begin
  Assert(param <> nil);
  ...
end;
```

Vorteile:

- Relativ einfach und leichtgewichtig
- Gut geeignet für Parameterprüfungen in privaten Methoden

Nachteile:

- Begrenztes Einsatzgebiet
  - Nicht gedacht für Ausnahmen, die keine Bugs sind

Exceptions [Folie 43]

```
procedure TMyList.Add(item: TMyItem);
begin
  if item = nil then
    raise EArgumentNil.Create('Cannot add nil to list.');
  ...
end;
```

Vorteile:

- Trennung von Normalfall und Ausnahmefällen
- Können beliebige Daten zum Fehler aufnehmen
- Erlauben das einfache Delegieren der Fehlerbehandlung

Nachteile:

- Vergleichsweise komplex
- Erlauben unübersichtliche Kontrollflüsse

Und wann nehm' ich jetzt was? [Folie 44]

- Boolean-Rückgabewerte, Fehlercodes: wenn die „Ausnahme“ regelmäßiger Teil des Kontrollflusses ist (wie bei `OpenDialog.Execute`)
- Fehlerzustände: Wenn man einen Fall wie oben hat, den Rückgabewert aber anderweitig nutzen möchte
- Fehlerbehandlungsroutinen: für Spezialfälle
- Assertions: für Ausnahmen aufgrund von möglichen Bugs; nicht an Schichtgrenzen
- Exceptions: für alles andere

3.2. Exceptions: Grundlagen

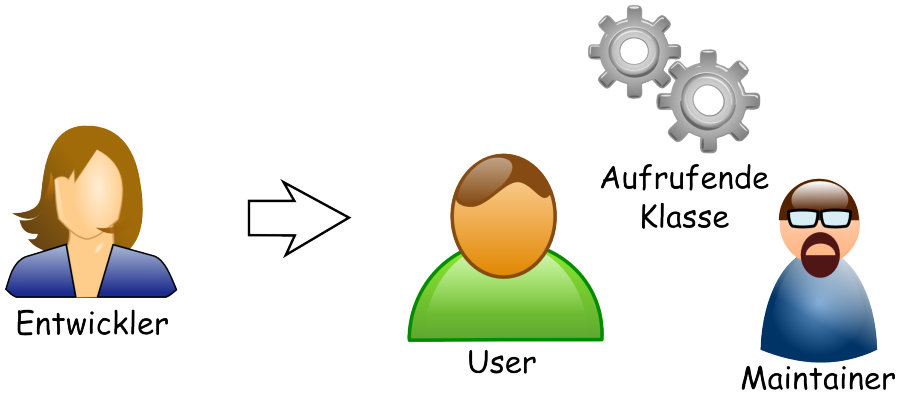
Wie man Exceptions wirft [Folie 45]


```
if coffeePot.isEmpty then
  raise EOutOfCoffee.Create('You drank too much coffee. Now there''s nothing
      left.');
```

Drei Teile:

- Bedingung für die Exception
- Klasse der Exception ⇒ wichtig für Entwickler
- Meldungstext ⇒ wichtig für User

Stakeholder: Die drei Empfänger [Folie 46]



Stakeholder einer Exception sind die Entwickler und die drei Empfänger der Informationen:

- Die aufrufende Klasse (sie muss ggf. die Exception behandeln)
- Der User (er bekommt ggf. eine Meldung, die ihn in die Lage versetzen soll, das Problem einzuordnen)
- Der Entwickler, der irgendwann ggf. die Informationen nutzt um einen Bug zu finden (Maintainer)

Die Klasse selbst, die den Fehler wirft, ist i. d. R. *kein* Empfänger der Exception. Wenn man Exceptions wirft, sollte man das vom Standpunkt der anderen Klassen tun. Man sollte sich also fragen „Was erwartet der Aufrufer an Exceptions?“ und nicht „Welche Exceptions treten hier auf?“

Wie man Exceptions fängt [Folie 47]


```
var
  foo: TFoo;
begin
  foo := TFoo.Create;
  try
```

```
    try
      foo.Bar;
    except
      // Exception behandeln
    end
  finally
    foo.Free;
  end;
end;
```

Der obige Code zeigt die Standardvorgehensweise beim Fangen von Exceptions:

- try..except in try..finally schachteln
- Create vor das erste try
- Pro Methode ein try..except..finally; nicht mehrere hintereinander
- Achtung: Hier wird angenommen, dass Konstruktor und Destruktor keine (nicht-verhinderbaren) Exceptions werfen
- Selbstverständlich gibt es Ausnahmen, aber das ist die generelle Vorgehensweise

Wie man Exceptions behandelt (1/2) [Folie 48]

- Verhinderbare Exceptions
  - Gar nicht
  - Ereignis in Logfile schreiben
  - Bugreport erzeugen
  - Programm geordnet abbrechen

- Gar nicht (Default-Verhalten oft ausreichend) ⇒ Aktion abbrechen und Nutzer benachrichtigen
- Ereignis in Logfile schreiben (hier hilfreich: `Application.OnException`)

- Bugreport erzeugen (hier hilfreich: madExcept [9]/EurekaLog [1])
- In besonders kritischen Fällen: Programm geordnet abbrechen

---

Wie man Exceptions behandelt (2/2) [Folie 49]

- Nicht-verhinderbare Exceptions: sehr stark vom konkreten Fall abhängig
  - User benachrichtigen
  - Rollback der Transaktion
  - Aktion nochmal versuchen
  - TCP-Verbindung wiederherstellen
  - Client aus Liste entfernen
  - ...

3.3. Exceptions im Detail

Trennung von Normalfall und Ausnahmefall (1/2) [Folie 50]


```
if Bla(42) then
begin
  FillChar(param, SizeOf(param), 0);
  param.value := 21;
  o := Blubb(parem);
  if GetLastError = NoError then
  begin
    if o.DoSomething('not very interesting') <> SUCCESS then
      HandleDoSomethingFailling;
```

```
  end
  else
  begin
    LogError('Failure! ' + GetLastError);
    ShowMessage('something bad happened');
  end;
end
else
begin
  LogError('Failure in Bla!');
  ShowMessage('something bad happened');
end;
```

Ohne Exceptions entstehen verschachtelte if-Konstruktionen und man ist gezwungen immer Normalfall und Ausnahmefall gleichzeitig zu betrachten. Das macht den Code tendenziell schwerer lesbar und fehleranfällig. Außerdem ist man dazu geneigt, die Fehlerbehandlung schnell *irgendwie* zu machen, damit es funktioniert, anstatt sich intensiv damit auseinander zu setzen. Das ist auch erstmal nur natürlich, da man zuerst den Programm beibringen will, den Normalfall richtig zu behandeln. Eigentlich möchte man sich mit den Ausnahmefällen erst hinterher beschäftigen. Die ständigen if-Konstruktionen zwingen einen aber fast dazu, immer direkt diese else-Blöcke zu schreiben. Siehe hierzu auch [7].

Trennung von Normalfall und Ausnahmefall (2/2) [Folie 51]


```
try
  Bla(42);
  FillChar(param, SizeOf(param), 0);
  param.value := 21;
  o := Blubb(parem);
  o.DoSomething('not very interesting');
except
  on e: EDoSomethingFailed do
  begin
    HandleDoSomethingFailling;
  end;
  on e: Exception do
```

```
  begin
    LogError('Fehler!' + e.Message);
    ShowMessage('something bad happened');
  end;
end;
```

Exceptions trennen den Normalfall von den Ausnahmefällen. Zuerst kann man so tun, als würden überhaupt keine Fehler auftreten. Hinterher kümmert man sich dann um das, was alles schiefgehen kann. Der Code wird übersichtlicher und, wenn man den except-Block schreibt, hat man den Kopf frei, weil der Normalfall ja schon implementiert (und vielleicht auch schon getestet) ist.

Ausnahme oder Fehler? [Folie 52]


```
try
  ...
except
  on e: EAccessViolation do
  begin
    ...
  end;
end;
```

Exception kann man grob in zwei Gruppen einteilen. Die einen zeigen Ausnahmesituationen an, die sich prinzipbedingt nicht verhindern lassen (abbrechende Netzwerkverbindung, Nebenläufigkeitseffekte, etc.). Die anderen zeigen Fehler an, die man eigentlich vorher vermeiden kann, sodass die Exception eigentlich nie auftreten sollte. Letztere fängt man üblicherweise nicht ab, sondern verhindert sie. Wenn sie auftreten, deuten sie auf einen Bug hin. So ist es nicht sonderlich sinnvoll eine EAccessViolation abzufangen.

In Java gibt es für diese Unterscheidung sogar Sprachunterstützung: Dort gibt es „checked Exceptions“, die man abfangen *muss* (der Compiler zwingt einen dazu) und „RuntimeExceptions“, die man normalerweise nicht abfängt, sondern vermeidet. Leider ist das Konzept in der Java API nur schlecht umgesetzt, sodass es teilweise mehr schadet als hilft.

Der Fall mit dem FileExists [Folie 53]


```
if FileExists(someFile) then
begin
  LoadFile(someFile);
end;
```

Vermeidbare Exceptions sollte man vermeiden. Tut das der obige Code? Auf den ersten Blick vielleicht ja, aber genau genommen: nein. Zum einen bedeutet das Vorhandensein einer Datei noch nicht, dass sie lesbar ist. Zum anderen beinhaltet der Code eine Race Condition[6]: Verschwindet die Datei genau zwischen der Prüfung mit `FileExists` und dem Zugriff in `LoadFile`, gibt es dennoch eine Exception. In den meisten Fällen ist das zwar vernachlässigbar unwahrscheinlich, aber es gibt Ausnahmen (beispielsweise Server, die regelmäßig Logfiles schreiben und ggf. löschen). Also auch, wenn das hier immer als Standardbeispiel für das Vermeiden von Exceptions verwendet wird, ist es trotzdem kein eindeutiger Fall.

Wächter [Folie 54]

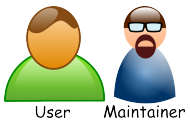

```
procedure AddItem(AItem: TMyItem);
begin
  if AItem = nil then
    raise EArgumentNil.Create('Cannot add nil.');

  ...
end;
```

[6] http://de.wikipedia.org/wiki/Time-of-Check-to-Time-of-Use-Problem

Auch in der Methode, in der die Exception geworfen wird, ist es teilweise möglich, den Normalfall von den Ausnahmefällen zu trennen. Ein Beispiel hierfür sind Wächer (Guards) am Anfang einer Methode. Sie prüfen auf einen Ausnahmezustand und werfen dann direkt eine Exception. So entsteht keine unnötig tiefe Verschachtelung von if-Blöcken.

Exception-Meldung [Folie 55]




```
raise EOutOfCoffee.Create('Du Hornochse bist zu durstig!');
```

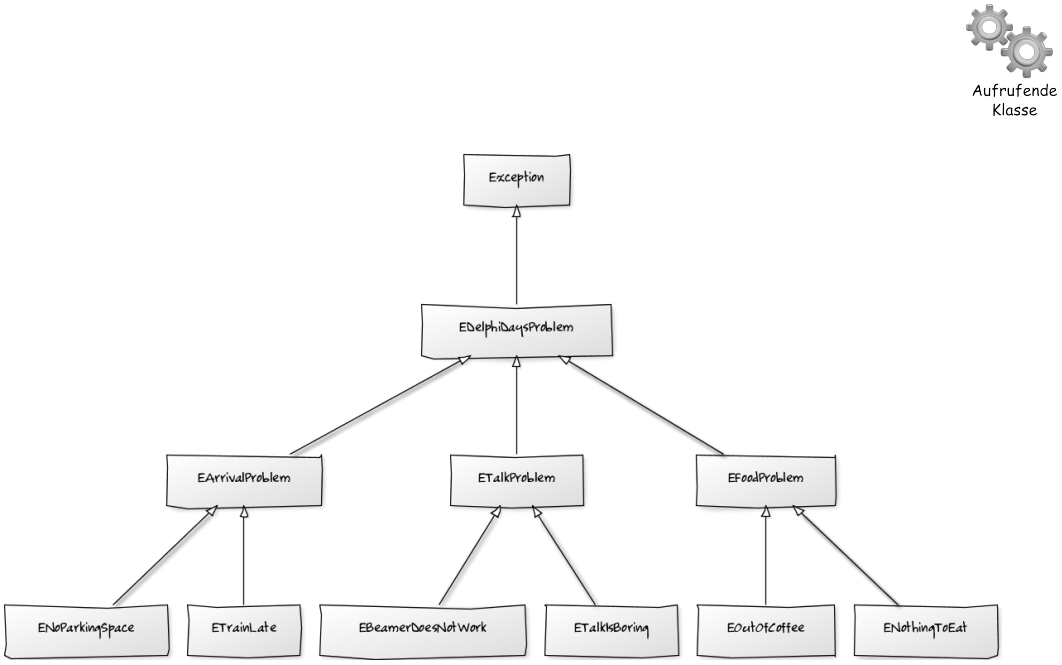
- Die Meldung der Exception sollte im Idealfall für den User einigermaßen verständlich sein.
- Auf jeden Fall aber muss sie für den Maintainer verständlich sein.
- In einer lokalisierten Software sollte man Exception-Meldungen deshalb genauso wie alle anderen Strings ebenfalls lokalisieren (wobei es hier aus Ausnahmen gibt: ein deutscher Maintainer wird mit einer chinesischen Fehlermeldung so einfach nichts anzufangen wissen).

Exception-Klasse [Folie 56]


```
type
  EOutOfCoffee = class(Exception)
  end;
```

- Die Exception-Klasse sollte das Problem bzw. den Ausnahmefall schon vollständig benennen. Zwei unterschiedliche Ausnahmen sollten auch immer zwei getrennte Exception-Klassen haben. Der Meldungstext ist für den User gedacht und sollte im Programm selbst nicht ausgewertet werden müssen.
- Es ist außer zu Testzwecken i. d. R. *nicht* sinnvoll direkt `Exception` zu werfen. Man sollte immer eine Subklasse nehmen.

Eine Hierarchie von Exceptions [Folie 57]



- Exception-Klassen sollte man hierarchisch strukturieren. Spezielle Exception-Klassen erben von allgemeinen.
- So kann man allgemeine Exception-Klassen fangen und so alle Subklassen gleich behandeln. Das ist robuster und spart Schreibarbeit.

on-Statements [Folie 58]

```
try
  ...
except
  on e: ETalkIsBoring do
  begin
    FallAsleep;
  end;
  on e: ETalkProblem do
  begin
    ShakeHead;
  end;
  on e: EDelphiDaysProblem do
  begin
    Complain(e.Message);
  end;
end;
```

- Beim Fangen von Exceptions nutzt man on-Statements um zu unterscheiden, um welche Exception-Klasse es sich handelt.
- Über den Parameter `e` kann man auch die Exception-Properties zugreifen.
- Die Reihenfolge der on-Statements ist entscheidend: Oben stehen die speziellen, unten die allgemeinen Exceptions. Beachtet man das nicht, werden die speziellen Exceptions oben schon in den allgemeinen on-Statements behandelt und der Code weiter unten wird nie ausgeführt.

Bestehende Exceptions [Folie 59]

- `EAbort`
- `EArgumentException`

  - `EArgumentNilException`
  - `EArgumentOutOfRangeException`
- `EInvalidOpException`
- `ENoConstructException`
- `ENotImplemented`
- `ENotSupportedException`
- `EProgrammerNotFound`

- Lange Zeit waren fast keine sinnvoll wiederverwendbaren Exceptions in der RTL definiert. Mittlerweile gibt es ein paar.
- Definiert sind diese Exceptions in `SysUtils`
- Achtung: Die folgenden Klassen sind *nicht* zur Wiederverwendung gedacht
  - `System.SysUtils.EInvalidOp`
  - `System.Classes.EInvalidOperation`
  - `System.Math.EInvalidArgument`
- `EAbort` ist eine stille Exception, d. h. sie wird standardmäßig dem Benutzer *nicht* angezeigt, sondern beendet den Prozess, wenn sie nicht abgefangen wird. Man kann sie auch über die Prozedur `Abort` auslösen.
- `EProgrammerNotFound` gibt es wirklich
  - Zitat aus der Delphi-Hilfe: „Nicht standardmäßige Art und Weise, Software-Fehler zu melden.“

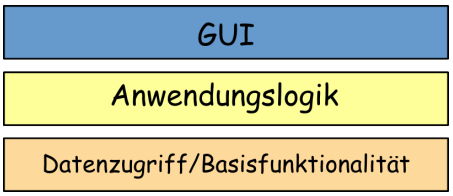
Exceptions sind Objekte [Folie 60]

```
EOutOfCoffee = class(Exception)
private
  ...
public
  property NumberOfEmptyPots: Integer ...;
end;
```

- Exceptions sind Klassen. Demnach gibt es die Möglichkeit, einer Exception Properties mit zusätzlichen Informationen zu spendieren.
- Diese Infos kann die aufrufende Klasse zur Behandlung der Ausnahme nutzen.
- Oder der Maintainer kann daraus nützliche Infos herauslesen um den Fehler zu finden.
- Insbesondere enthält eine Exception auch den Stack Trace, also die Folge von Prozeduraufrufen, die zur Ausnahme geführt hat.
- Hier sei auch nochmal auf madExcept [9] und EurekaLog [1] verwiesen.

4. Und zusammen...

Exceptions in Schichten [Folie 62]



Eine Schichtarchitektur bewirkt, dass Aufrufe in oberen Schichten meist Aufrufe auf der direkt darunter liegenden Schicht zur Folge haben usw. Es entstehen also geschachtelte Methodenaufrufe. Und irgendwo ganz unten kann eine Exception fliegen...

Grundregel [Folie 63]

Grundregel Exceptions in Schichten
Exceptions dort fangen, wo man weiß, wie sie zu behandeln sind. Nicht früher und nicht später.

Wenn es eine Möglichkeit gibt, eine Exception auf einer unteren Ebene zu fangen und vollständig zu behandeln, sodass höhere ebenen sich nicht damit herumschlagen müssen, prima. Wenn nicht, gilt folgende Tendenzregel:

Unten werfen, oben fangen [Folie 64]

Tendenzregel
Unten werfen, oben fangen

Meistens wirft man in unteren Schichten Exceptions und fängt sie in der GUI. Somit haben die Formularklassen in fast allen Methoden try..except-Blöcke. Und untere Schichten werden mit der ganzen Ausnahmebehandlung nicht weiter belastet. Ganz so einfach ist es dann aber doch nicht:

Problem [Folie 65]

```
procedure TSettingsDialog.OKButtonClick(Sender: TObject);
begin
  try
    ...
    settingsObject.StoreSettings;
  except
    on EFileStreamError do // ???
    begin
      ...
    end;
  end;
end;
```

- Offensichtlich wird ein FileStream zum Speichern verwendet; die obere Schicht weiß zu viel über die Interna der unteren
- Wenn irgendwann mal stattdessen die Einstellungen beispielsweise in ner Datenbank gespeichert werden sollen, muss der Code geändert werden (Ripple Effect)

Exception Chaining [Folie 66]

```
procedure TSettings.StoreSettings;
begin
  try
    ...
  except
    on e: EFileStreamError do
    begin
      raise ESettingsWriteError.Create('Could not write settings.', e);
    end;
  end;
end;
```

- Die Exception wird „umverpackt“. Eine neue Exception, die zur gegebenen Abstraktionsstufe (= Schicht) passt, wird geworfen. Damit weiß die obere Schicht nichts mehr über die Implementierungsdetails der unteren und die genannten Ripple Effects können nicht mehr auftreten.
- Damit der Maintainer trotzdem noch alle nötigen Informationen hat, um einen etwaigen Fehler zu finden, wird die alte Exception der neuen als Parameter mitgegeben. Die neue Exception speichert dann die alte Exception in der Peroperty `InnerException`.
- So packt jede Schicht ihre eigene Exception um die gefangene und eine Kette von Exceptions entsteht (Exception Chaining).
- Neuere Delphi-Versionen bieten hierfür auch eine Klassenmethode: `RaiseOuterException`.
- Das Problem ist im übrigen nicht auf Exceptions beschränkt. Hätte man stattdessen Fehlercodes verwendet, müsste man die Fehlercodes aus unteren Schichten auf neue Fehlercodes mappen. Das Problem dabei ist, dass das kaum ohne Informationsverlust geht. Mit Exception Chaining ist es kein Problem, alle Infos zu erhalten.
- Siehe auch [6] und [7].

Fazit [Folie 67]

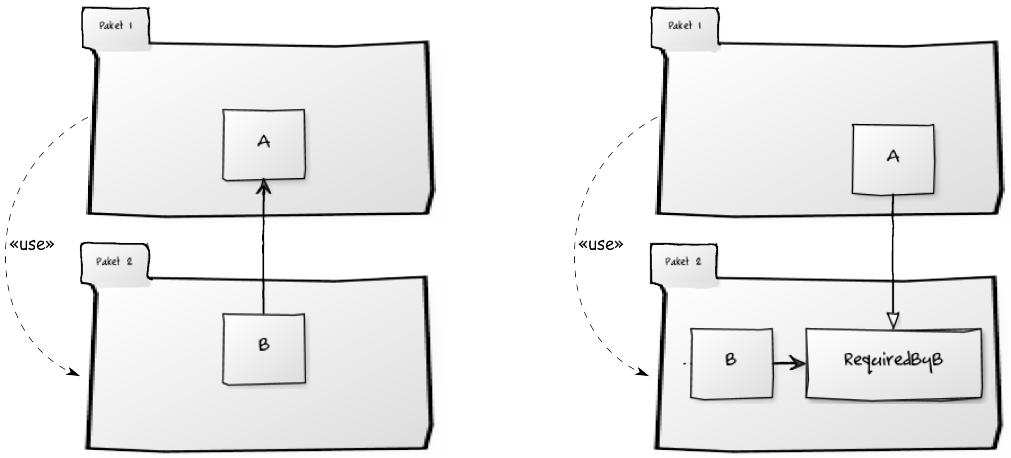
- Schichten
  - Gute Architekturen beschränken die Kommunikation der Klassen untereinander
  - Schichten bilden eine typische Grobstruktur, die das zum Ziel hat
  - Obere Schichten dürfen auf untere zugreifen, aber nicht umgekehrt
- Exceptions
  - Trennung von Normalfall und Ausnahmefall
  - Verhinderbare vs. nicht-verhinderbare Ausnahmesituationen
  - Die drei Empfänger einer Exception
- Zusammen
  - Exception Chaining

Vielen Dank! [Folie 68]

Fragen?

A. Anhang

Dependency Inversion [Folie 70]



- Manchmal möchte man Aufrufe von niedrigeren in höhere Schichten realisieren, obwohl man das eigentlich nicht „darf“.
- Lösungen:
  - Callbacks, Events
  - In anderen Sprachen: Closures, Delegates, ...
  - Dependency Inversion (sprachunabhängig)
    - Generelles Prinzip mit verschiedenen Ausprägungen
    - Eine der Ausprägungen ist das Observer-Pattern (siehe [10])
- Dependency Inversion: Abhängigkeiten lassen sich umkehren, wenn man eine zusätzliche Abstraktion (Basisklasse bzw. Interface) einführt.
- `B` greift nicht direkt auf `A` zu, sondern auf eine abstrakte Klasse oder ein Interface. `A` implementiert nun dieses Interface bzw. erbt von der abstrakten Klasse. jetzt kann `B` Methoden aus `A` aufrufen, ohne `A` direkt zu kennen. Vielmehr ist `A` nun abhängig von der abstrakten Klasse `RequiredByB`. Die Abhängigkeit hat sich herumgedreht.

Das nil-before-try-Idiom (1/2) [Folie 71]

```
var
  foo: TFoo;
  bar: TBar;
begin
  foo := TFoo.Create;
  try
    bar := TBar.Create;
    try
      try
        ...
      except
        ...
      end;
    finally
      bar.Free;
    end;
  finally
    foo.Free;
  end;
end;
```

Wenn man zwei oder mehr Objekte in einer Methode instanziiert, muss man try-Blöcke noch weiter schachteln, was den Code unübersichtlich macht.

Das nil-before-try-Idiom (2/2) [Folie 72]

```
var
  foo: TFoo;
  bar: TBar;
begin
  foo := nil;
  bar := nil;
  try
    foo := TFoo.Create;
    bar := TBar.Create;
    try
      ...
```

```
    except
      ...
    end;
  finally
    bar.Free;
    foo.Free;
  end;
end;
```

Wenn man nun stattdessen die Objektvariablen vor dem `try`-Block `nil` setzt und das Erzeugen im try-Block erledigt, kann man sich die Schachtelung sparen. Eine Auftretende Exception im ersten Konstruktoraufruf wird keine Speicherlöcher erzeugen, da das Free immer ausgeführt wird. Aber auch das Free wird immer funktionieren, da, falls der zweite Konstruktoraufruf nicht ausgeführt wurde, die Objektvariable noch `nil` ist.

Literatur [Folie 73]

Literatur

[1] *EurekaLog*. http://www.eurekalog.com/.

[2] Bass, Len, Paul Clemens und Rick Kazman: *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 2 Auflage, 2003.

[3] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern-Oriented Software Architecture*, Band 1: A System of Patterns. John Wiley & Sons, 1996.

[4] Duszynski, Slawomir, Jens Knodel und Mikael Lindvall: *SAVE: Software Architecture Visualization and Evaluation*. In: *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, Seiten 323–324, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Goetz, Brian: *Exceptional practices*. http://www.javaworld.com/javaworld/jw-08-2001/jw-0803-exceptions.html, Aug 2001.

[7] Köhntopp, Kristian: *Was bedeutet eigentlich „Never check for an error condition you dont́ know how to handle“*. http://blog.koehntopp.de/archives/2611-Was-bedeutet-eigentlich-Never-check-for-an-error-condition-you-dont-know-how-t
html.

[8] Lindvall, Mikael und Jens Knodel: *Software Architecture Visualization and Evaluation*. http://www.fc-md.umd.edu/save/.

[9] Rauen, Mathias: *madExcept*. http://www.madshi.net/madExceptDescription.htm.

[10] Rehn, Christian: *Patterns*. http://www.christian-rehn.de/2011/05/21/patterns/, Mai 2011.

[11] Spolsky, Joel: *The Law of Leaky Abstractions*. http://www.joelonsoftware.com/articles/LeakyAbstractions.html, Nov 2002.

Lizenz [Folie 74]