Fehlerbehandlung in mehrschichtigen Systemen

Schichten und Exceptions

Christian Rehn aka R2C2

Delphi-Treff

Delphi-Tage 2012

Vorstellung

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

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

Organisatorisches

- 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, ...)

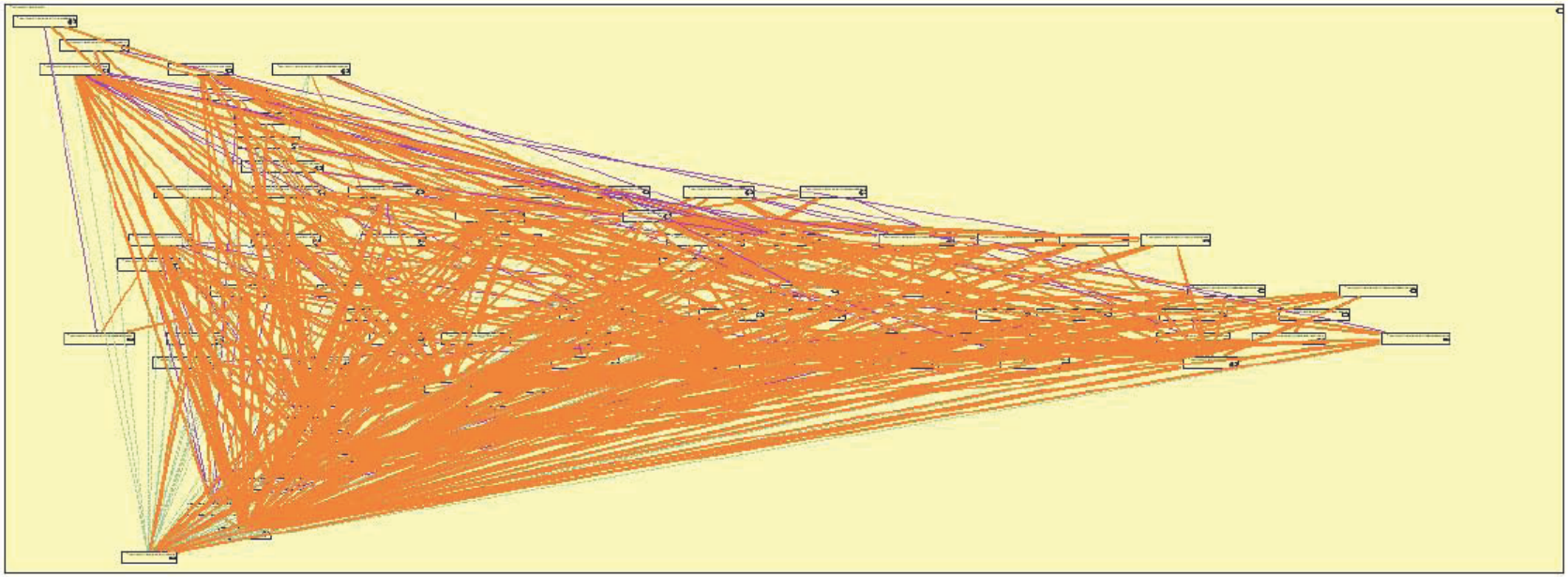
Überblick

Motivation

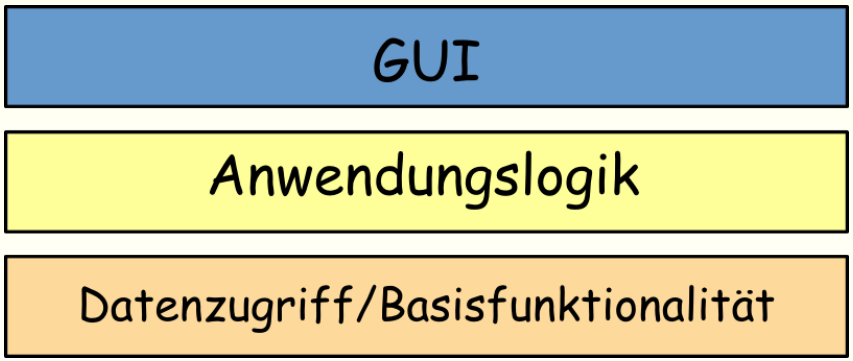
Warum überhaupt nachdenken? (1/2)

Warum sollten wir uns überhaupt Gedanken über so etwas machen?

Warum überhaupt nachdenken? (2/2)



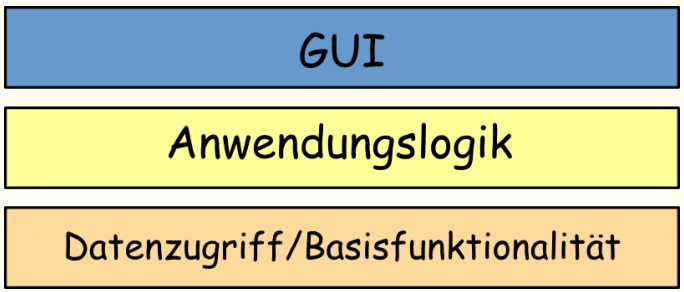
Schichten



Warum Exceptions?

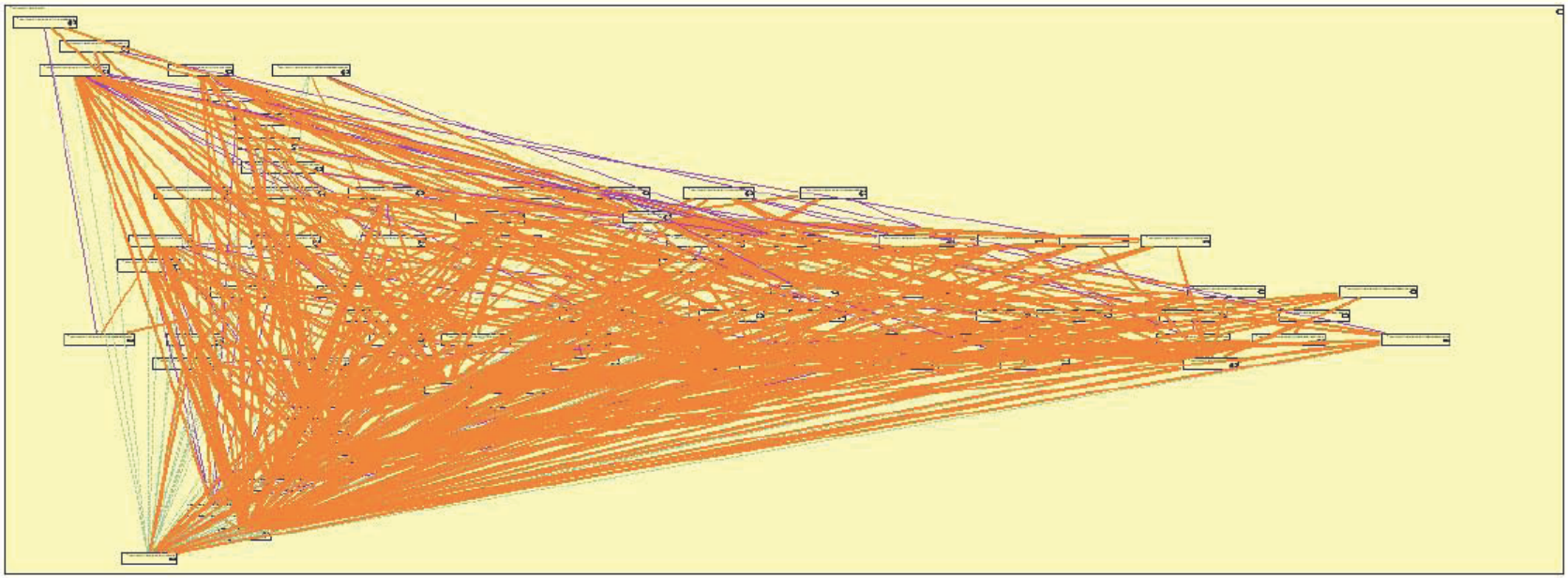
```
try
  while not EndOfTalk do
  begin
    Present( slide );
    GoToNextSlide;
  end;
except
  on e: EFireAlarm do
  begin
    Panic;
    Shout(e.Message);
  end;
end;
```

Und wie passt das zusammen?




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

Abhängigkeiten und Schichten



Ripple Effects

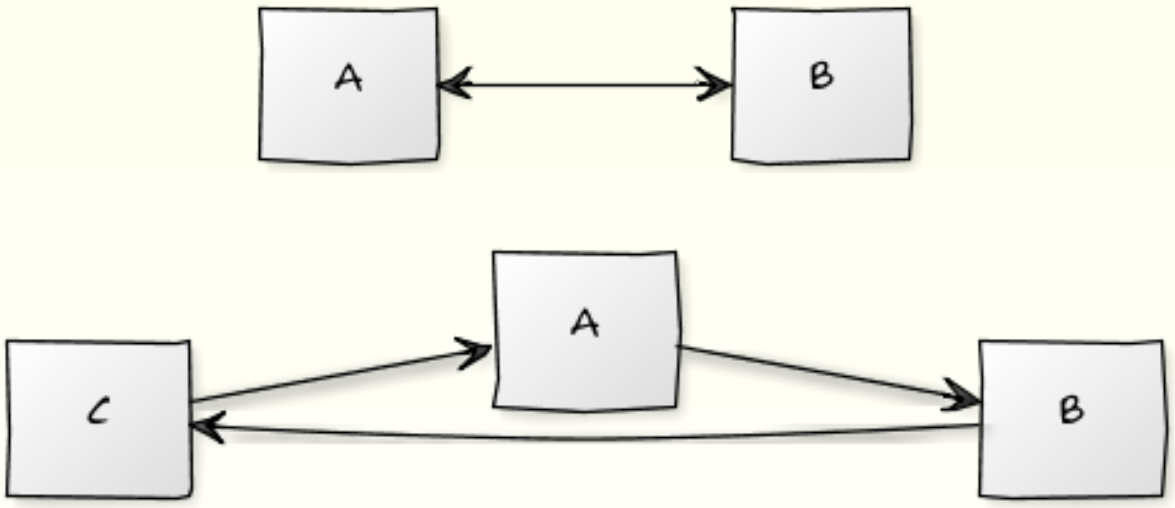
2

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

Abhängigkeiten

```
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;
```

Zyklische Abhängigkeiten



Zirkuläre Unit-Referenz

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

Architektur

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. [BCK03]

Architektur: Meine Definition

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

Architektur

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. [BCK03]

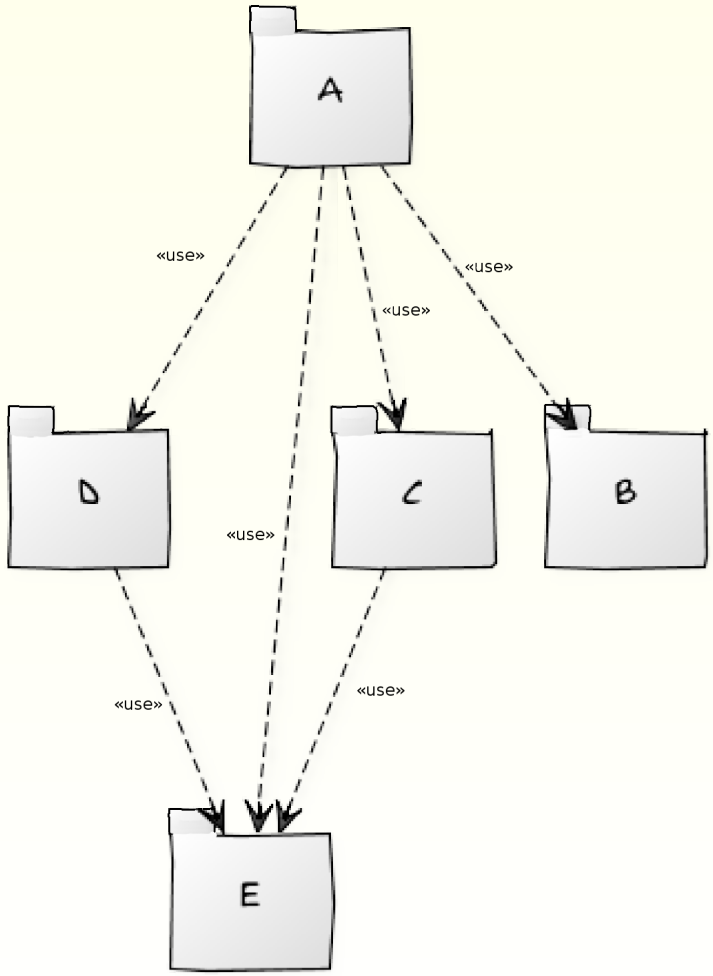
Architektur: Meine Definition

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

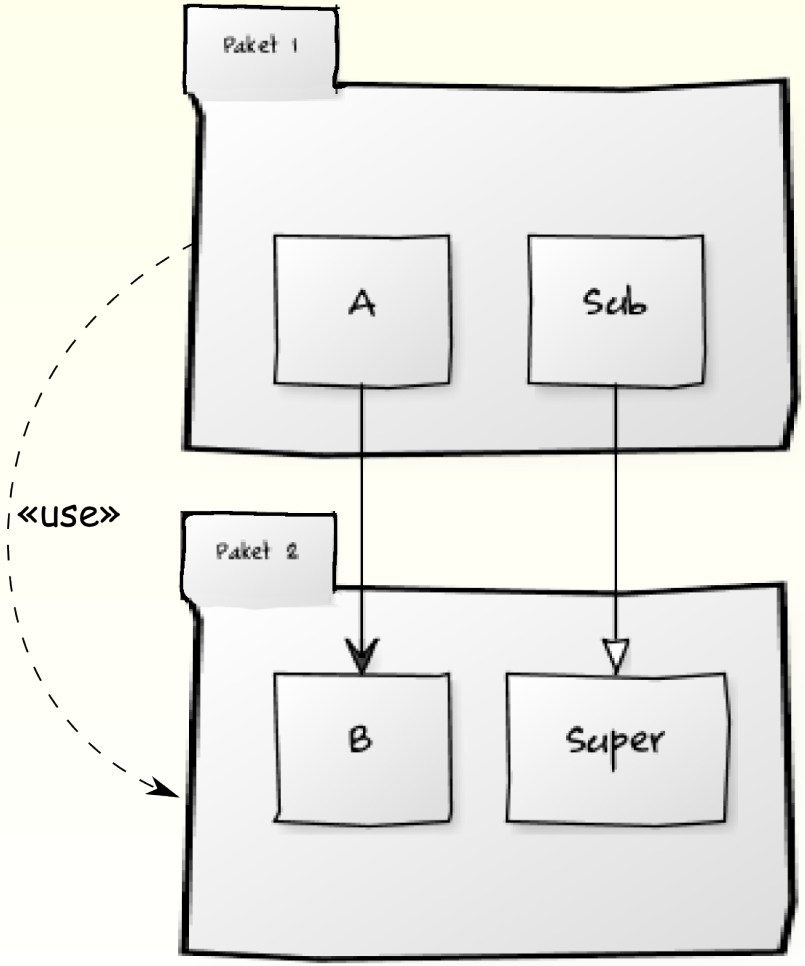
Grobstruktur

Grobstruktur als Teil der Architektur

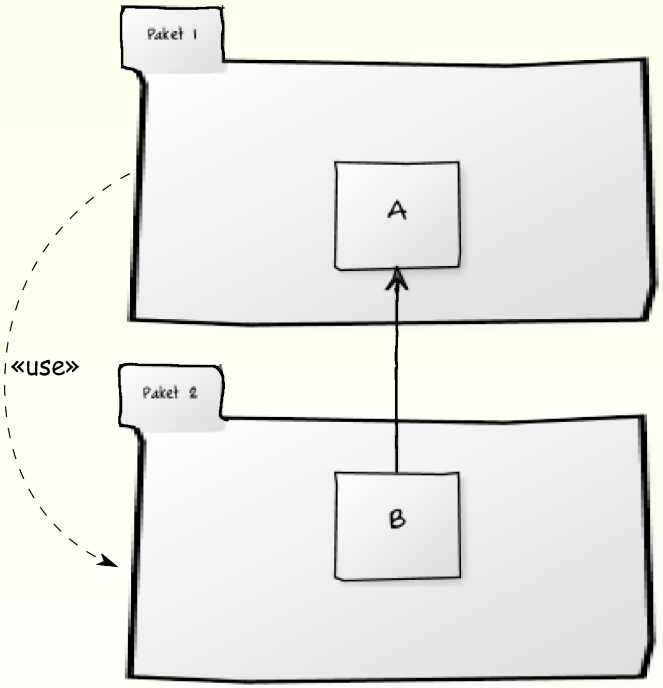
Kommunikation beschränken (1/2)



Kommunikation beschränken (2/2)

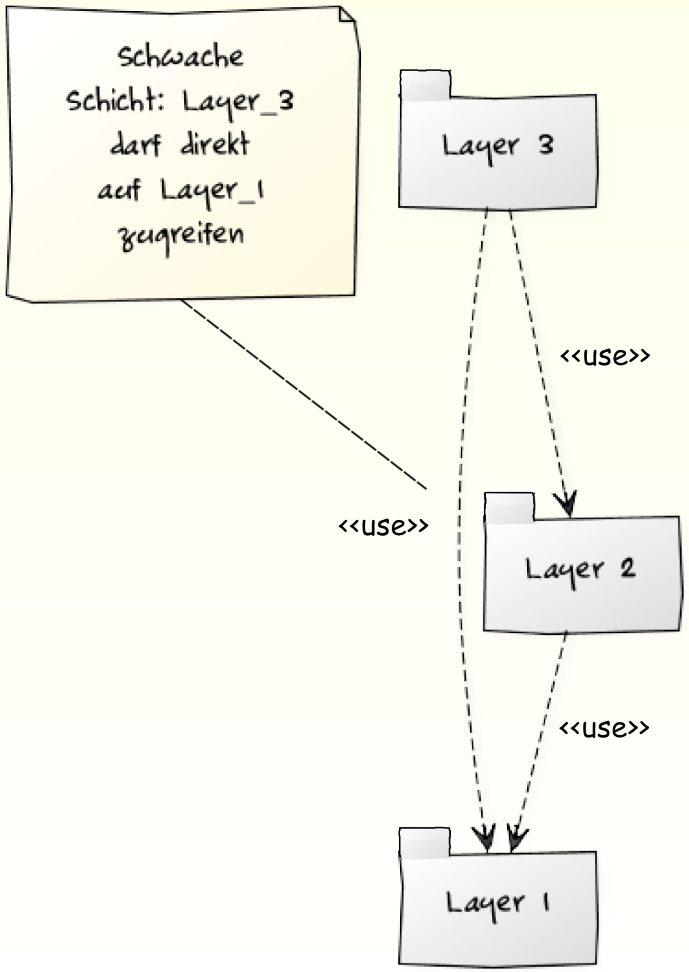


Dependency Inversion: Events




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

Schichten


Schwache
Schicht: Layer_3
darf direkt
auf Layer_1
zugreifen
Layer 3
<<use>>
<<use>>
Layer 2
<<use>>
Layer 1

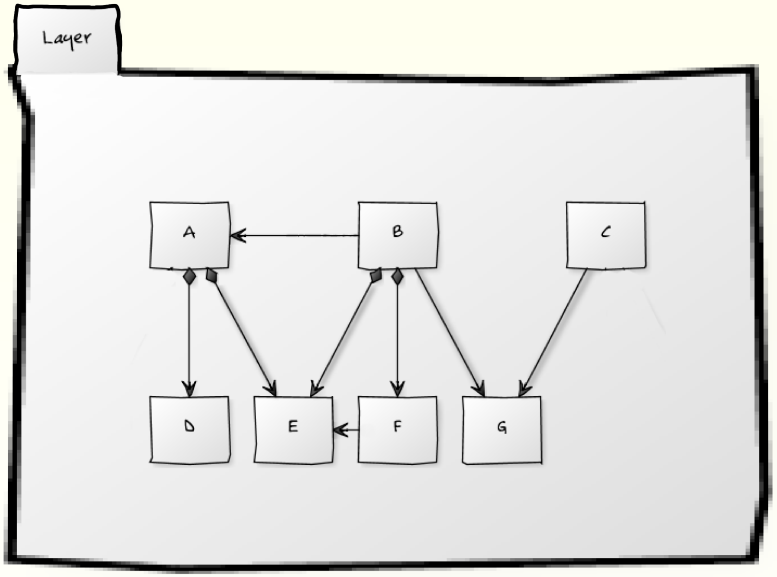
Virtuelle Maschinen

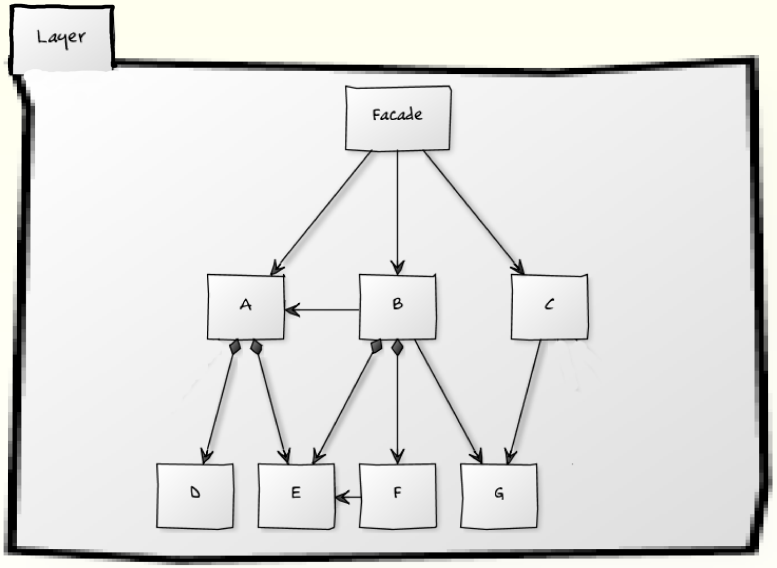
Law of Leaky Abstractions

Law of Leaky Abstractions

All non-trivial abstractions, to some degree, are leaky. [Spo02]

Schichten und Fassaden





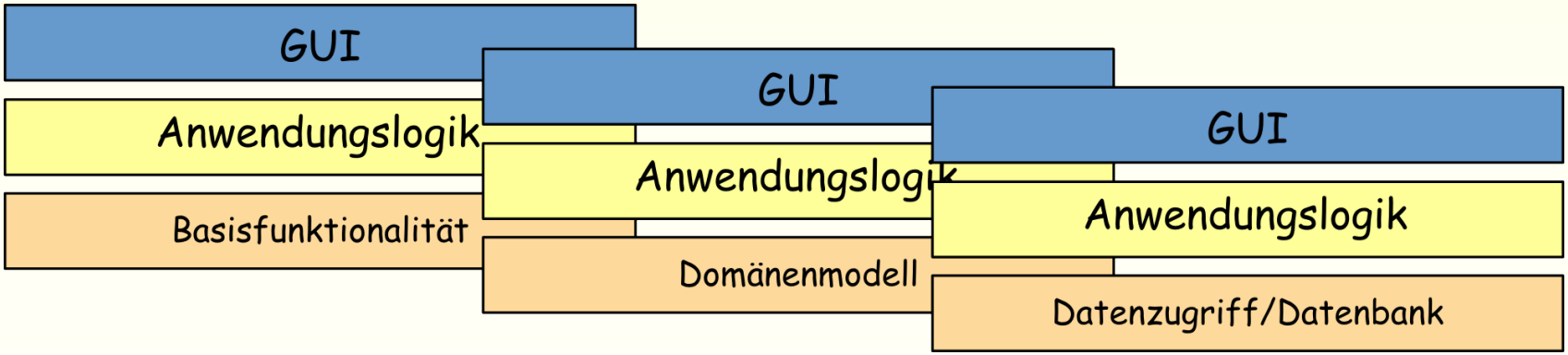
Überall Schichten

- 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
- ...

1, 2, 3, . . . , n Schichten

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

Die typische Dreischichtenarchitektur

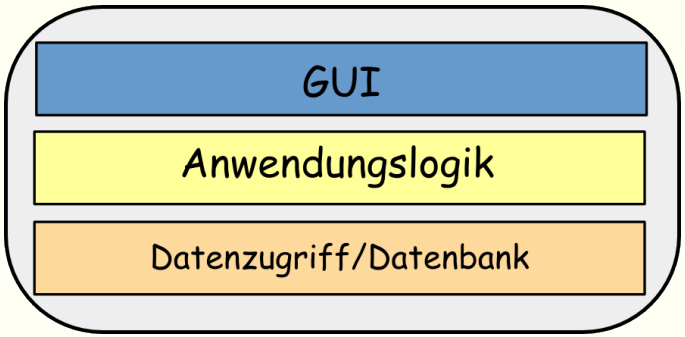


Multi-Tier?

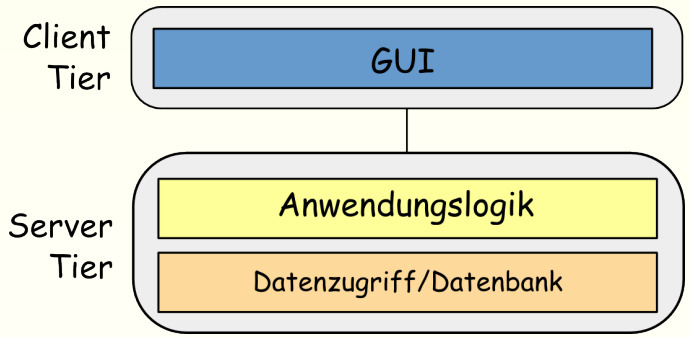
Layers vs. Tiers

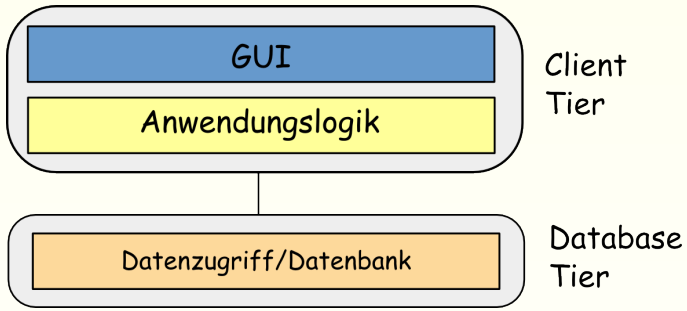
- Layer: logische Schicht
- Tier: physische Schicht

1-Tier

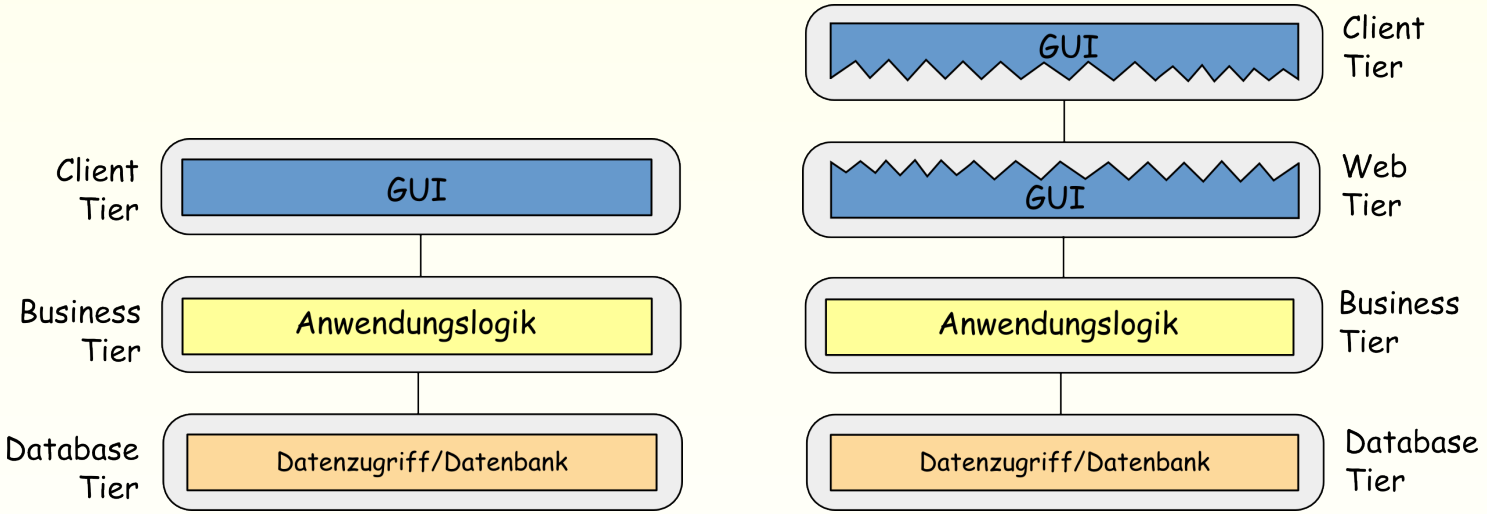


2-Tier


Client
Tier
GUI
Server
Tier
Anwendungslogik
Datenzugriff/Datenbank


GUI
Anwendungslogik
Client
Tier
Datenzugriff/Datenbank
Database
Tier

3-Tier, 4-Tier, n-Tier



Exceptions

Ausnahmesituationen

Wenn doch alles nur normal wäre...

Ausnahmesituationen: Arten von Ausnahmen

Verhinderbar oder nicht verhinderbar; das ist hier die Frage.

Möglichkeiten der Ausnahmebehandlung

- Boolean-Rückgabewerte
- Fehlercodes
- Fehlerzustände
- Fehlerbehandlungsroutinen
- Assertions
- Exceptions

Boolean-Rückgabewerte

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

Fehlercodes

```
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;
```

Fehlerzustände

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

Fehlerbehandlungsroutinen

Event: OnError

Assertions

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

Exceptions

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

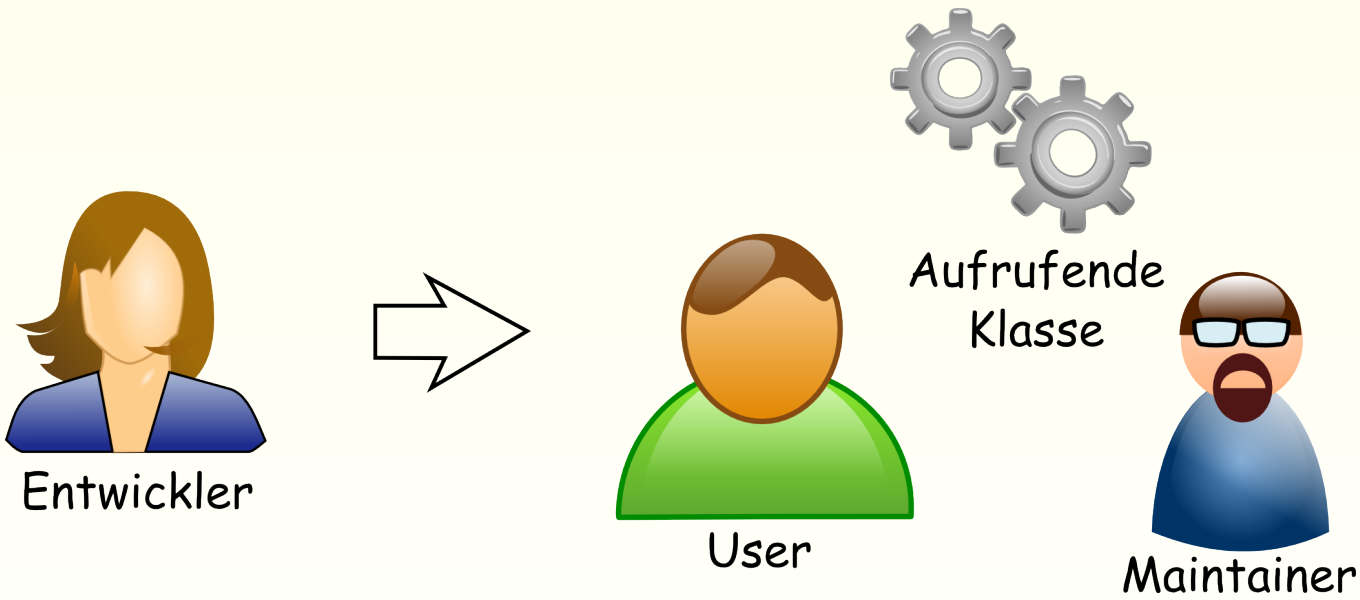
Und wann nehm' ich jetzt was?

- 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

Wie man Exceptions wirft


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

Stakeholder: Die drei Empfänger



Wie man Exceptions fängt


```
var
  foo: TFoo;
begin
  foo := TFoo.Create;
  try
    try
      foo.Bar;
    except
      // Exception behandeln
    end
  finally
    foo.Free;
  end;
end;
```

Wie man Exceptions behandelt (1/2)

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

Wie man Exceptions behandelt (2/2)

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

Trennung von Normalfall und Ausnahmefall (1/2)


```
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;
```

Trennung von Normalfall und Ausnahmefall (2/2)


```
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;
```

Ausnahme oder Fehler?


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

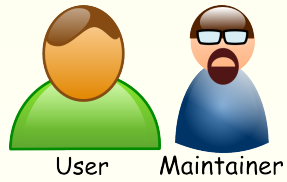
Der Fall mit dem FileExists


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

Wächter

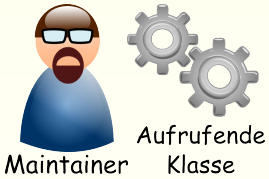

```
procedure AddItem(AItem: TMyItem);
begin
  if AItem = nil then
    raise EArgumentNil.Create('Cannot add nil.');

  ...
end;
```

Exception-Meldung




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

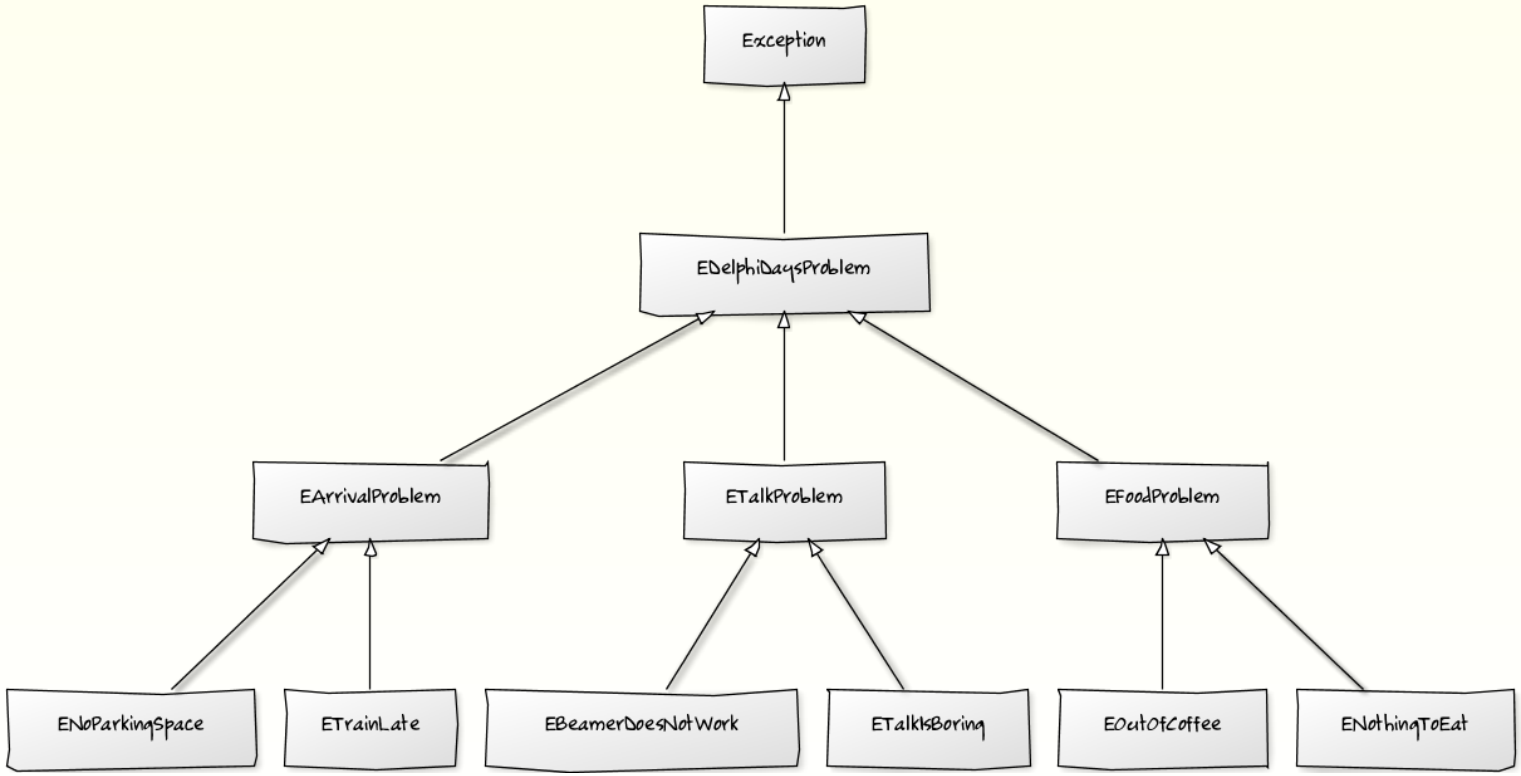
Exception-Klasse




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

Eine Hierarchie von Exceptions

Aufrufende Klasse



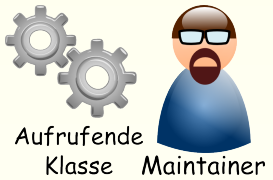
on-Statements

```
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;
```

Bestehende Exceptions

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

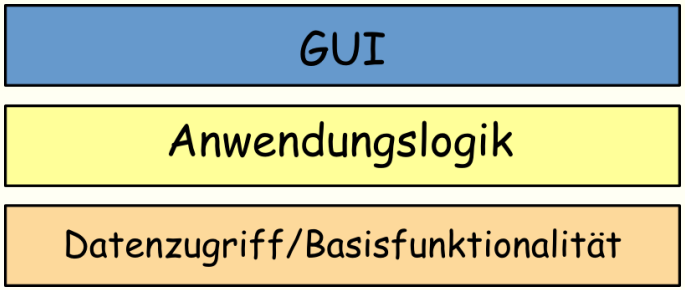
Exceptions sind Objekte




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

Und zusammen. . .

Exceptions in Schichten



Grundregel

Grundregel Exceptions in Schichten

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

Unten werfen, oben fangen

Tendenzregel

Unten werfen, oben fangen

Problem

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

Exception Chaining

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

Fazit

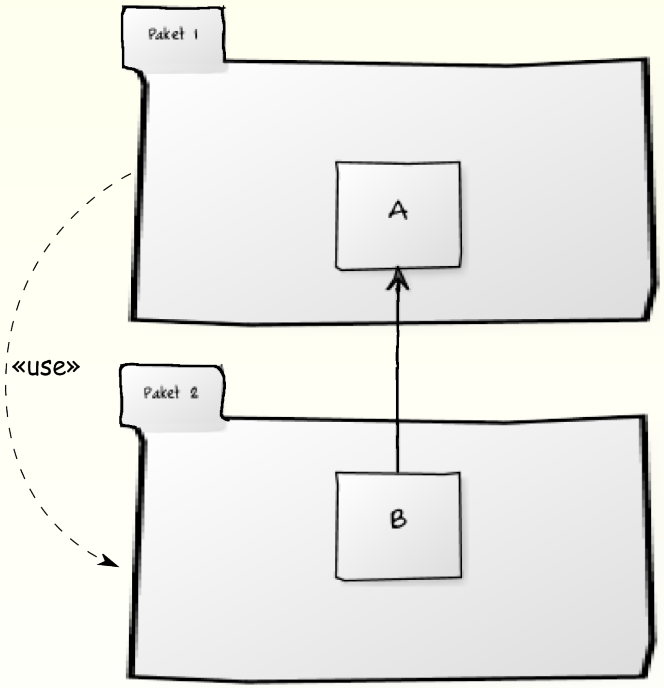
- Schichten
  - Gute Architekturen beschränken die Kommunikation der Klassen untereinander
  - Schichten bilden eine typische Grobstruktur, die das zum Ziel hat
  - Obere Schichten dürfen auf untere zugreifen, aber nicht umgekehrt
- Exceptions
  - Trennung von Normalfall und Ausnahmefall
  - Verhinderbare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

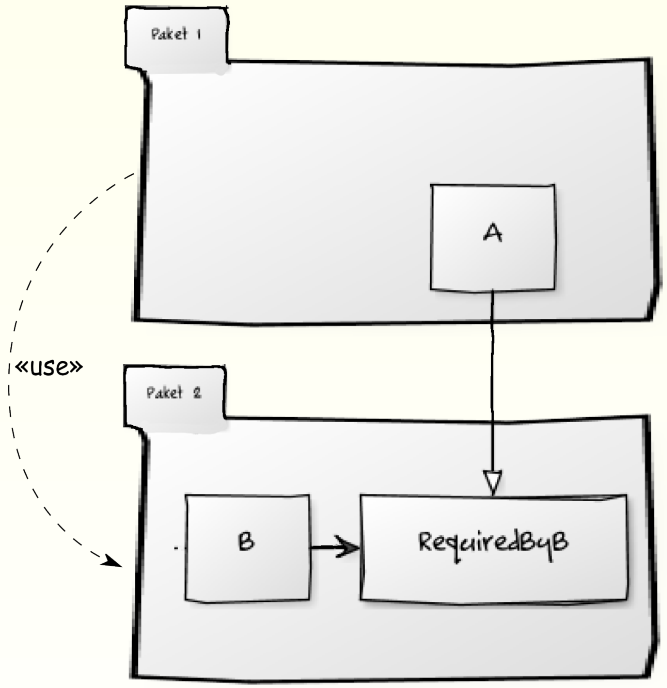
Vielen Dank!

Fragen?

Anhang

Dependency Inversion





Das nil-before-try-Idiom (1/2)

```
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;
```

Das nil-before-try-Idiom (2/2)

```
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;
```

Referenzen für Zitate

 Len Bass, Paul Clemens, and Rick Kazman.
Software Architecture in Practice.
SEI Series in Software Engineering. Addison-Wesley, 2 edition, 2003.

 Joel Spolsky.
The law of leaky abstractions.
`http://www.joelonsoftware.com/articles/LeakyAbstractions.html`, Nov 2002.

(eigentliche Bibliographie in den Vortragsnotizen)

Lizenz