Exception Handling in Multi-Layered Systems

Layers and Exceptions

Christian Rehn

ADUG Sydney Meeting
21st November 2012

Who Am I? [Slide 2]

- Christian Rehn
- CS Student at the University of Kaiserslautern
- Moderator and editor for Delphi-Treff (some German Delphi website)
- http://www.christian-rehn.de/

[1]

Organisational Stuff [Slide 3]

- A basic understanding of OOP is needed
- Few text on the slides
  - Better for presentation
  - There are more detailed talk notes online: http://www.christian-rehn.de/

[1] In contrast to the rest, the logos, of course, aren't CC licensed.

  - German version is even more detailed but I haven't had the time to translate everything

- Please give feedback (what can I do better?)

Overview [Slide 4]

Contents

---

1. Motivation

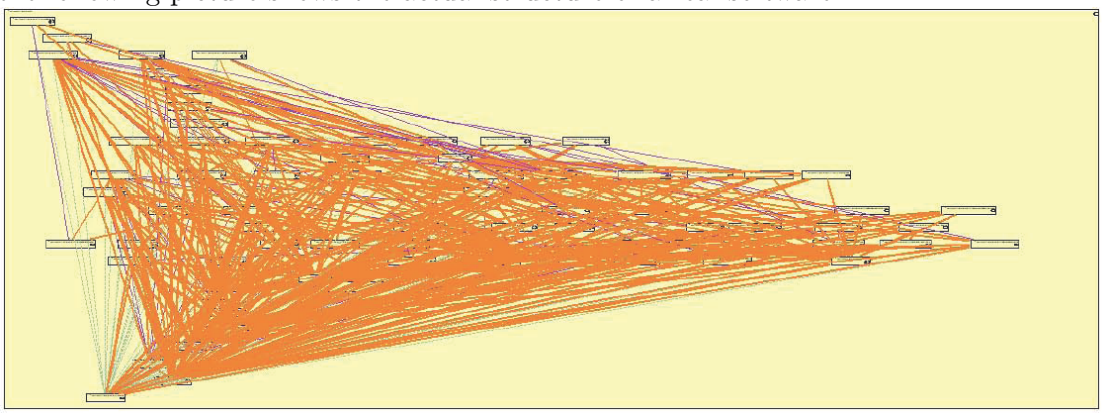
1.1. Why Layers?

Why Think? (1/2) [Slide 6]

Why should we think about all that?

Why Think? (2/2) [Slide 7]

the following picture shows the actual structure of a real software.



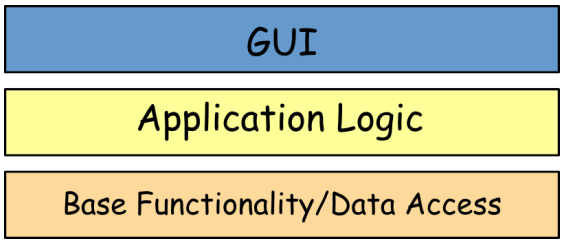
[2]

And that's just one subsystem of about 20. When software looks like this, maintenance becomes a nightmare.

This graphic has been produced by a software written at the Fraunhofer Institure for Experimental Software Engineering (IESE). It shows the actual dependencies between the classes. We can assume that the developers weren't idiots, so we see that it' s really difficult to retain a good architecture.

[2] Tanks to the Fraunhofer IESE for the picture

Layers [Slide 8]



Layers are one of the most important ways to structure software.

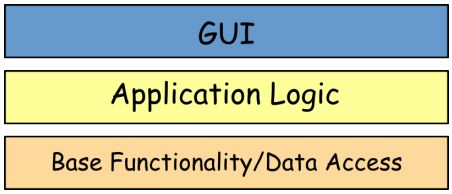
1.2. Why Exceptions?

Why Exceptions? [Slide 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. Putting Everything Together

And what's the link between these two topics? [Slide 10]




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

2. Dependencies and Layers

2.1. Dependencies

Ripple Effects [Slide 13]

[3]

Ripple Effects

Changes in one part of the system impose further changes in other parts which again impose other changes and so on. Changes ripple though the code, and the code gets fragile.

- Ripple Effects result from too many and too strong dependencies

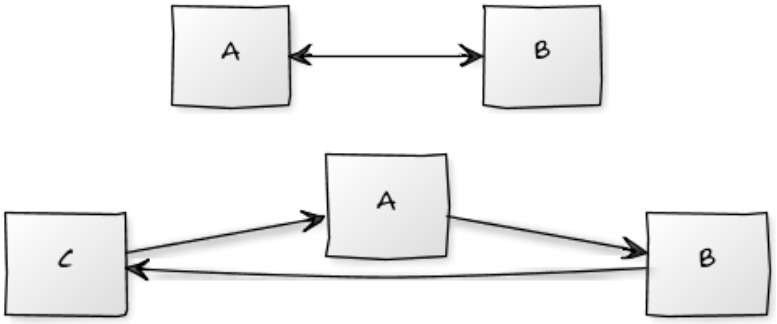
Dependencies [Slide 14]

```
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

The above code shows a bad implementation of an OpenDialog. Think of what you would have to do in case of an internal change and look at TOpenDialog for how to do it better.

Cyclic Dependencies [Slide 15]



- Cyclic Dependencies create strong couplings
- One module in the cycle depends on all others
- Better:
  - Dependency relations should be acyclic (dependency graph is a DAG[4])
  - Even better: hierarchical (dependency graph is a tree)

Circular Unit References [Slide 16]

```
[DCC Fatal Error] UnitXY.pas(7):  F2047 Circular unit reference to 'UnitXY'
```

Such an error often signals a design problem.

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

2.2. Coarse Structure

To avoid the aforementioned problems it is necessary to think about architecture.

Architecture [Slide 17]

Architecture: Definition by 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. [1]

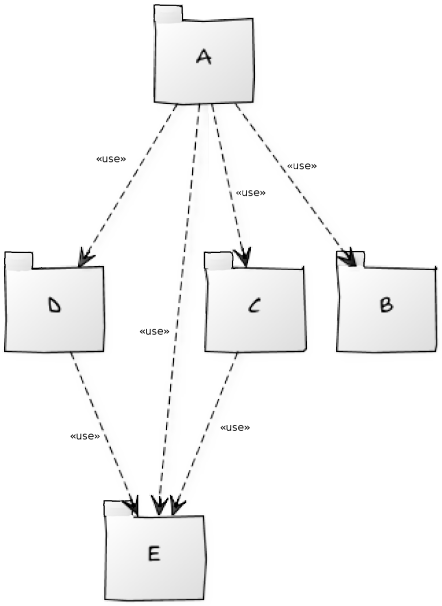
Architecture: My Definition
The software architecture describes the coarse structures of the software and defined how to *think* about it as a developer.

Coarse Structure [Slide 18]

A coarse decomposition structure as a part of architecture

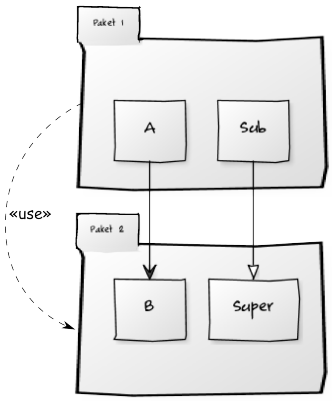
Architecture is more than just a coarse structure. In particular software doesn't have just one structure but many structures. Just like a house not only has a floor plan. There is interior design, plumbing, electricity, etc. Here we only discuss a coarse decomposition structure.

Constrain Communication (1/2) [Slide 19]



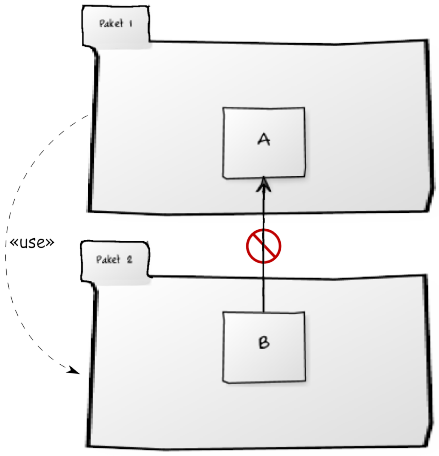
- A good architecture constrains the comminucation among the modules and disallows certain dependencies.
- The above package diagram shows a coarse structure for a piece of software. Classes are grouped into logical units (in UML: “packages”; in the diagram A, B, C, D and E). Such packages could be “GUI”, “DB Access”, “Application Logic”, etc.
- The architecture defines rules for all modules.

Constrain Communication (2/2) [Slide 20]



- In UML class diagrams all arrows point into the direction of the dependency.
- Package diagrams show bigger but also static structures.
- Relationships among packages must be retained by the classes contained in the packages.

Dependency Inversion: Events [Slide 21]

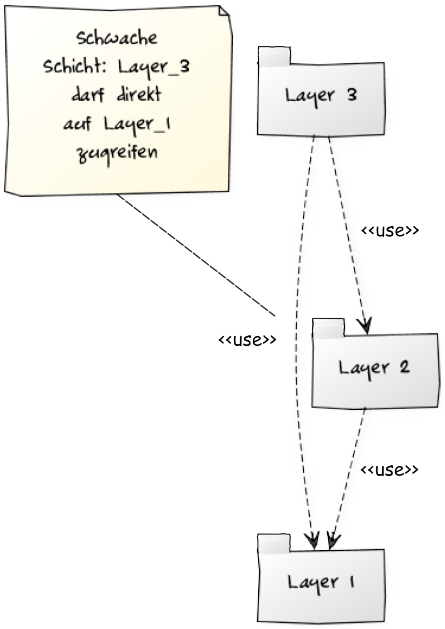



```
property OnSomeEvent: TNotifyEvent
    read FOnSomeEvent write
    FOnSomeEvent;
```

- Sometimes you want to make a method call in the "wrong direction".
- There are several possibilities how to solve that problem: Callbacks, Closures, Delegates, OO dependency inversion, ...
- In Delphi typically events are used

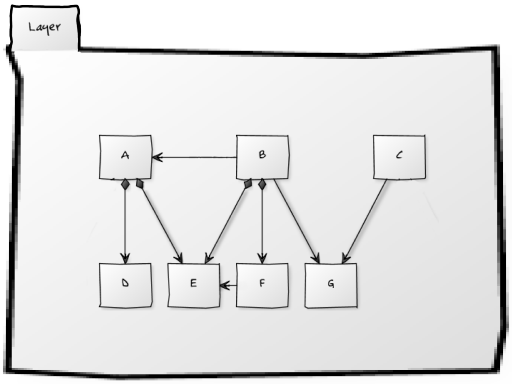
2.3. Layers and Tiers

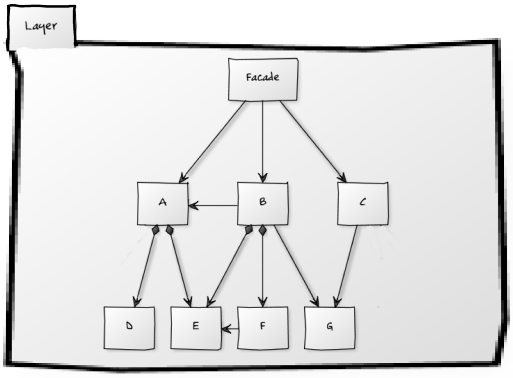
Layers [Slide 22]



- Often layers are used to constrain communication
  - Higher layers may access lower ones (but *never* the other way around)
  - A stricter variation: Only the next lower layer may be accessed
- Layers are abstractions
  - Higher layers (e. g. GUI) are more concrete and specific for an application
  - Lower layers comprise more and more generally usable functionality
- Layers can again be decomposed into sub-layers

Layers and the Facade Pattern [Slide 23]





- Layers are conceptual groupings in the code
- If you want to make layers replaceable, you need defined interfaces
- The facade pattern helps [3]

Layers Everywhere [Slide 24]

- Networking:
  - Physical layer, link layer, network layer, transport layer, application layer
- APIs and Frameworks:
  - x86, WinAPI, RTL, VCL/FM
  - x86, WinAPI, CLR, .NET-Framework, SWF/WPF
- Typical Information Systems
  - GUI, application logic, data access
- ...

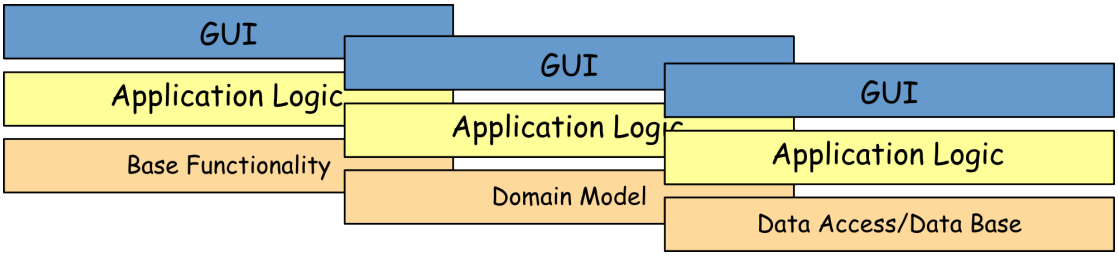
Layers are everywhere. It's an architectural pattern [2].

1, 2, 3, . . . , n Layers [Slide 25]

- One layers: Everything is done in the form/every class may access every other ⇒ chaos
- Two layers: e. g. form + data module
- Three layers: form + application logic + base layer (or similar)
- . . . Other breakdowns possible . . .

Be careful: Too many layers is bad, too. Everything gets more complex.

The Typical Three Layer Architecture [Slide 26]



Typical information systems have three layers. Where to make the cuts may vary.

Tears? – Tiers! [Slide 27]

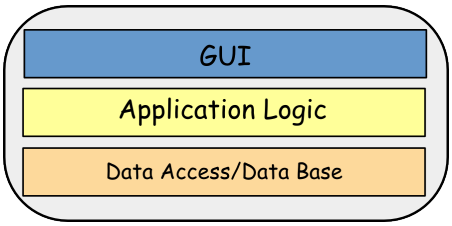
 [5]

Layers vs. Tiers [Slide 28]

- Layer: logical separation
- Tier: physical separation

Tiers may be deployed to separate computers.

1-Tier [Slide 29]

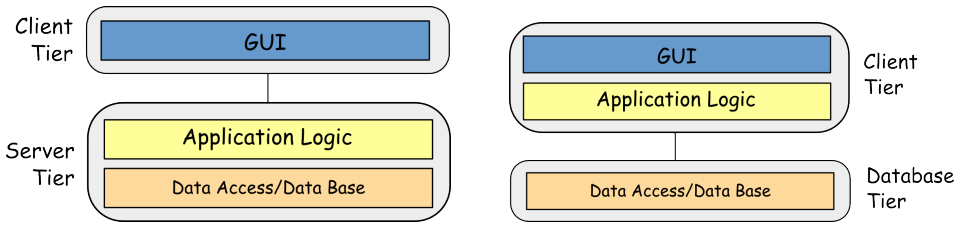


- 1-Tier: Everything on one machine

[5] CC-BY-SA 2.0 by Crimfants http://commons.wikimedia.org/wiki/File:Crying-girl.jpg

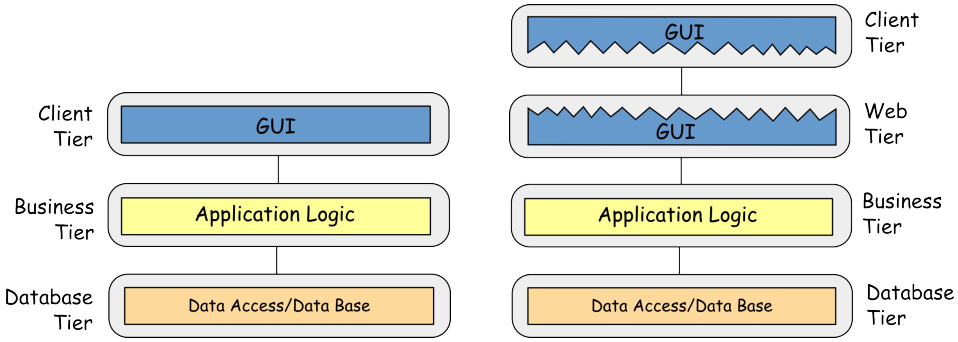
- Examples:
  - Embedded databases (SQLite & Co.), flat files, ...
  - Simple programs without databases
  - Mainframes

2-Tier [Slide 30]



- 2-Tier: theres a client and a server
- Depending on where the application logic is deployed to it's a thin or a fat client
  - 2-Tier (thin client): client + application server
  - 2-Tier (fat client): application client + db server

3-Tier, 4-Tier, n-Tier [Slide 31]



- 3-Tier: (thin) client + application server + db server
- 4-Tier: (thin) client in browser + web app + application server + db server

3. Exceptions

3.1. Exceptional Cases

Exceptional Cases [Slide 33]

If only everything would be normal...

An exceptional case is one which differs from the normal situation. Usually we don't like such cases but we have to think about them. And in fact they are subject to a large portion of the bugs. In typical software the normal cases work pretty well. And all those lovely bugs show up when there is some tiny condition that is not normal.

Note: There are exceptional cases or exceptional situations and there is the Exception language feature. I will write Exception with a capital 'E' when I mean the language feature and use the normal lower case version in the other cases. Beware that Headings as well as the first word in a sentence are capitalised either way.

Types of Exceptional Cases [Slide 34]

Avoidable, or not avoidable, — that is the question

We can make a coarse distinction between avoidable and unavoidable exceptions. If an avoidable exception occurs, this can be considered a bug. You cannot prevent me from taking scissors and cut the network cable. But you can prevent a division by zero or an access violation.

Possibilities for Exception Handling [Slide 35]

- Boolean return values
- Error codes
- Error states
- Error handlers
- Assertions

- Exceptions

Boolean Return Values [Slide 36]

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

Easy but limited.

Error Codes [Slide 37]

```
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;
```

More powerful but fragile.

What's wrong here? [Slide 38]

```
if ShellExecute(...) = ERROR_SUCCESS then
  ...
```

Error States [Slide 39]

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

Just like error codes but you have to use a separate function or property to get the information.

Error Handlers [Slide 40]

Event: OnError

Good for components and some special cases. But may get confusing if used too often.

Assertions [Slide 41]

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

Good for parameter checking in private methods. Should only be used for exceptional cases that are bugs.

Exceptions [Slide 42]

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

Very powerful but somewhat complex.

When to Use What? [Slide 43]

- Boolean return values, error codes: if the exception is a regular part of the control flow (like `OpenDialog.Execute`)
- Error states: If the return value shall be used for other purposes
- Error handlers: for special cases
- Assertions: for uncovering bugs (i. e. avoidable exceptions)
- Exceptions: for everything else

3.2. Exceptions: Basics

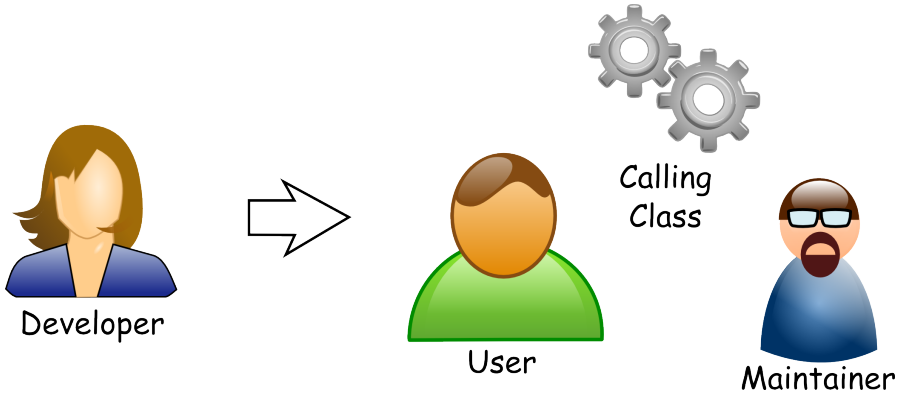
How to Raise Exceptions [Slide 44]


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

Three parts:

- Exception condition
- Exception class ⇒ important for developer
- Message text ⇒ important for user

Stakeholders: The Three Recipients [Slide 45]



How to Catch Exceptions [Slide 46]


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

That's the standard way to catch Exceptions.

How to Handle Exception (1/2) [Slide 47]

- Avoidable Exceptions
  - Ignore them
  - Log them
  - Create a bug report
  - Exit program

How to Handle Exception (2/2) [Slide 48]

- Unavoidable Exceptions: depends strongly on the concrete situation
  - Inform user
  - Rollback transaction
  - Retry
  - Reconnect
  - Remove client from the list
  - ...

3.3. Exceptions in Detail

Separating Normal Case and Exceptional Case (1/2) [Slide 49]


```
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;
```

Without Exceptions you have to nest several if-constructs and always consider the normal case and the exceptional cases at the same time. This has a negative effect on readability and may impose defects. Furthermore one is tempted to do the exception handling quick and dirty in order to get the programming running.

Separating Normal Case and Exceptional Case (2/2) [Slide 50]


```
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 separates the normal case from the exceptional cases. First you can pretend that there are absolutely no failures, no exceptional cases whatsoever. You can

concentrate on writing the normal case. And after the normal case is finished all the exceptions can be considered separately.

Avoidable or Unavoidable? [Slide 51]


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

Avoidable Exceptions like `EAccessViolation` should not be handled. They should be avoided instead. Normally the occurrence of an avoidable Exception can be regarded a bug. On the other hand unavoidable Exceptions (broken network links, concurrency effects, etc.) must be handled because there is no chance to avoid them.

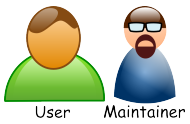
FileExists [Slide 52]


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

Avoidable Exceptions should be avoided. But sometimes it is hard to determine whether an Exception is really avoidable. The code above includes a race condition.

It appears only in very specific contexts, so one can regard the above code to be OK, but nevertheless if you are very precise, this is a bug.

Exception Message [Slide 53]




```
raise EOutOfCoffee.Create('You are too thirsty, you idiot!');
```

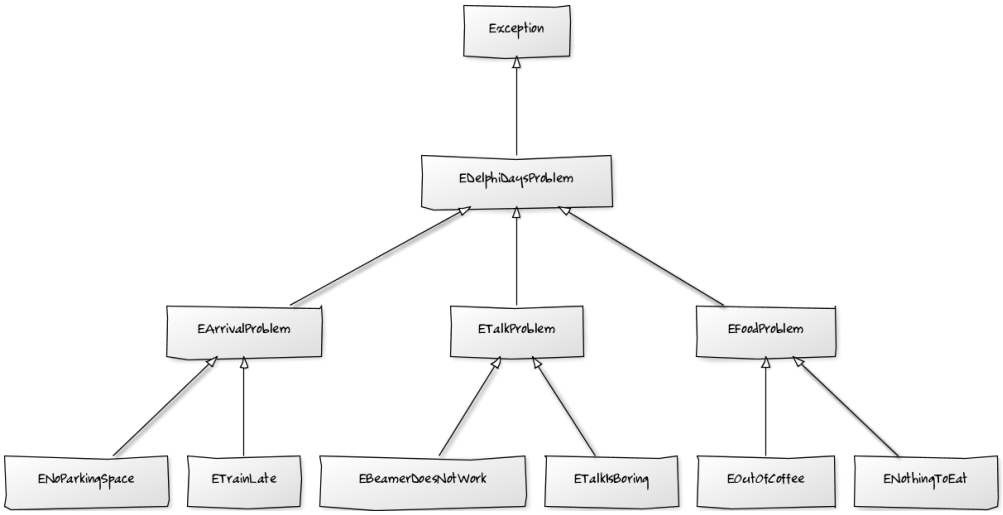
In the ideal case the Exception message is directly understandable by the user. In every case it *must be* understandable for the maintainer, though.

Exception Class [Slide 54]


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

The Exception class should name the problem completely. Two different exceptional cases should always raise two different Exception classes. So don't raise `Exception` directly but use subclassing.

A Hierarchy of Exceptions [Slide 55]



More general Exception classes can be caught if the handling is equal for all subclasses.

on-Statements [Slide 56]

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

```
  end;
end;
```

Remember that the order of the on-statements is significant.

Existing Exception Classes [Slide 57]

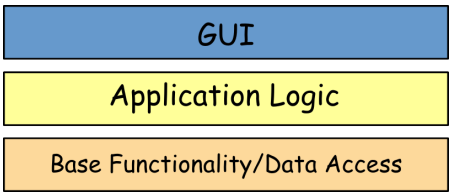
- `EAbort`
- `EArgumentException`
  - `EArgumentNilException`
  - `EArgumentOutOfRangeException`
- `EInvalidOpException`
- `ENoConstructException`
- `ENotImplemented`
- `ENotSupportedException`
- `EProgrammerNotFound`

The more recent Delphi versions have some common Exception classes in `SysUtils`.

4. Putting Everything Together

4.1. Exceptions and Layers

Exceptions in Layers [Slide 59]



A layered architecture results in cascading method invocations. And down there at the bottom an Exception can occur...

General Rule [Slide 60]

General Rule about Exceptions in Layers
Catch Exceptions at the point where you know how to handle them. Not earlier and not later.

Rule of Thumb [Slide 61]

Rule of Thumb
If in doubt, raise at the bottom and catch at the top.

Problem [Slide 62]

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

- Obviously a `FileStream` is used; the higher layer knows too much about the internal workings of the lower layers
- Suppose you change the lower layer so a database is used instead of a `FileStream`. In that case this code has to be changed (ripple effect)

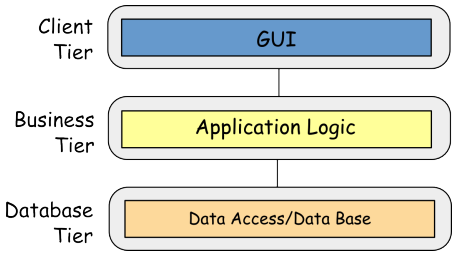
Exception Chaining [Slide 63]

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

- The Exceptions are nested. So the upper layer does not know the internal workings of the lower one and ripple effects are avoided.
- In order to not loose valuable information for the maintainer the old Exception is remembered in a property `InnerException`.
- Every layer wraps its own Exceptions around the caught ones, so a chain of Exceptions is created.
- Recent Delphi versions also provide an additional class method for that very purpose: `RaiseOuterException`.
- BTW: The same problem also arises when error codes or some similar mechanism is used. But there it's much harder not to loose information when mapping lower level errors to higher level ones.
- See also [4].

4.2. Exceptions and Tiers

Exceptions and Tiers [Slide 64]



- Exceptions are bound to one machine
- You have to explicitly code a mechanism to signal Exceptions over the network
  - Serialise Exception information (use error codes or string serialisations)
  - Transfer that over the network
  - Deserialise
  - Raise an Exception again
- Use a wrapper class for this purpose, so the calling classes don't have to know how (and that) this happens.

- Middleware might already produce such wrappers so you can transparently throw Exceptions over the network. Webservices are able to do this but I don't know how good the code generators for Delphi are[6]. I haven't used any middleware for Delphi, yet. Neither webservices nor DCOM, nor CORBA, nor DataSnap, nor anything else. So unfortunately I cannot tell which one has which capabilities.

A wrapper can look roughly like this:

Wrapper Classes [Slide 65]

```
procedure TFooWrapper.DoSomething;
begin
  ret := NetworkCallToMethodDoSomethingInSomeFooObjectOnSomeOtherMachine;
  case ret of
    FOO_SUCCESS: // do nothing;
    FOO_WRITE_ERROR: raise EFooWriteError.Create('could not write...');
    FOO_READ_ERROR: raise EFooReadError.Create('could not read...');
  else
    raise EFooException.Create('Unknown Problem with Foo'); // base class or the
        Exceptions above
  end;
end;
```

The same technique can be applied if you have DLLs or other libraries which cannot or do not raise Exceptions and use error codes or some similar mechanism instead.

[6] I one used webservices in Java and there the generators are bad as the don't produce real wrapper classes although they could.

Conclusion [Slide 66]

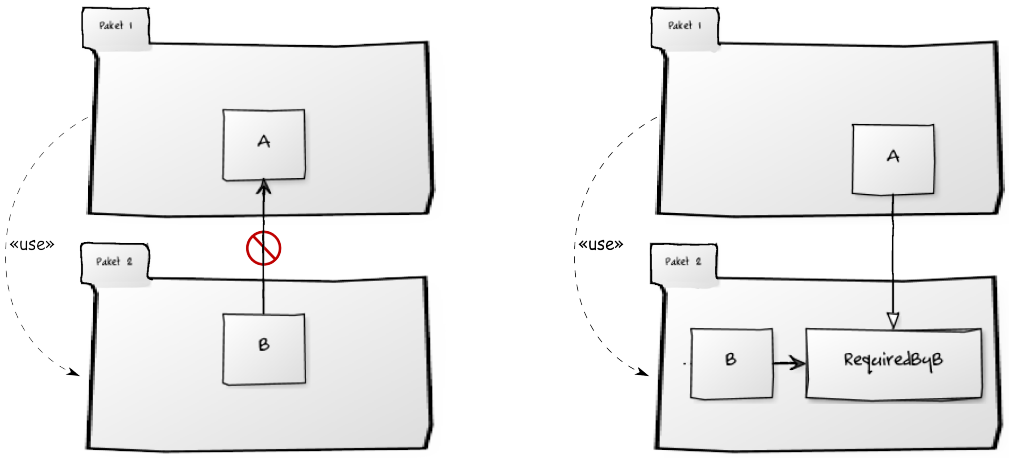
- Layers
  - Good architecture constrains the communication among the classes
  - Layers are a typical structure which supports this
  - Higher layers may access lower ones but *not* vice versa
- Exceptions
  - Separation of normal case and exceptional case
  - Avoidable and unavoidable exceptions
  - The three recipients of an exception
- Putting it all together
  - Exception chaining
  - Wrapper classes

Thank You! [Slide 67]

Questions?

A. Appendix

Dependency Inversion [Slide 69]



- Sometimes you want to call higher layers from lower ones. But that is not allowed.
- Solutions:
  - Callbacks, Events
  - In other languages: closures, delegates, ...
  - Dependency inversion (language independent)
    - General principle with different variants
    - One variant is the observer pattern [3]
- Dependency inversion: Dependencies can be inverted by adding an additional abstraction (base class or interface).
- `B` does not access `A` directly. Instead it used the abstract class/interface. `A` implements this interface or inherits from this abstract class, respectively. Now `B` can call methods from `A` without knowling it directly. Rather `A` is now dependent on the abstract class `RequiredByB`. The dependency has been inverted.

Virtual Machines [Slide 70]

A way of thinking: Layers are virtual machines.

Law of Leaky Abstractions [Slide 71]

Law of Leaky Abstractions
All non-trivial abstractions, to some degree, are leaky. [5]

Abstractions are never perfect. Sometimes you still have to think about aspects that should normally be abstracted away from you. Don't surrender but keep that in mind.

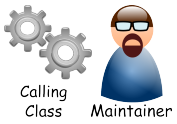
Guards [Slide 72]


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

 ...
end;
```

By the use of so-called guards, also the code which raises the Exception can be separated in the normal and the exceptional cases. This is not always possible but very typical for parameter checking. Nested if-constructs are avoided by that.

Exceptions are Objects [Slide 73]




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

Exceptions can also have additional properties as they are objects.

Literature [Slide 74]

References

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

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

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

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

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

Licence [Slide 75]