Refactoring mit Delphi 2007 in Rosenheim

Als Refactoring betrachtet man allgemein ein Vereinfachen, Verbessern und Stabilisieren einer bestehenden Codestruktur, welche keine Änderung auf das „beobachtbare“ Verhalten der Applikation und dessen Ergonomie bewirken soll.

“Refactoring is improving the design of code after it has been written”

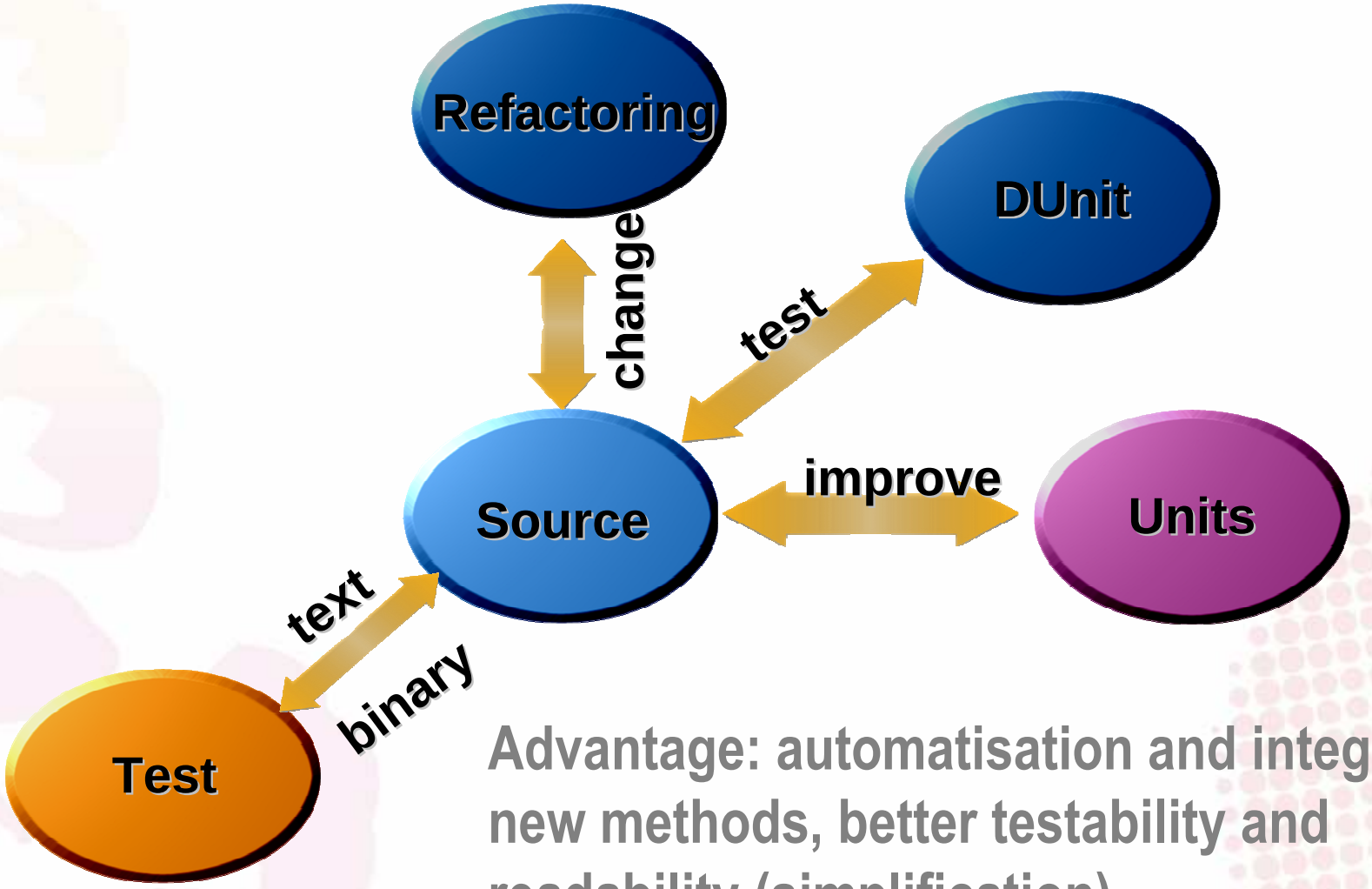
Kleiner

Kommunikation.....

Agenda 10.10.2008

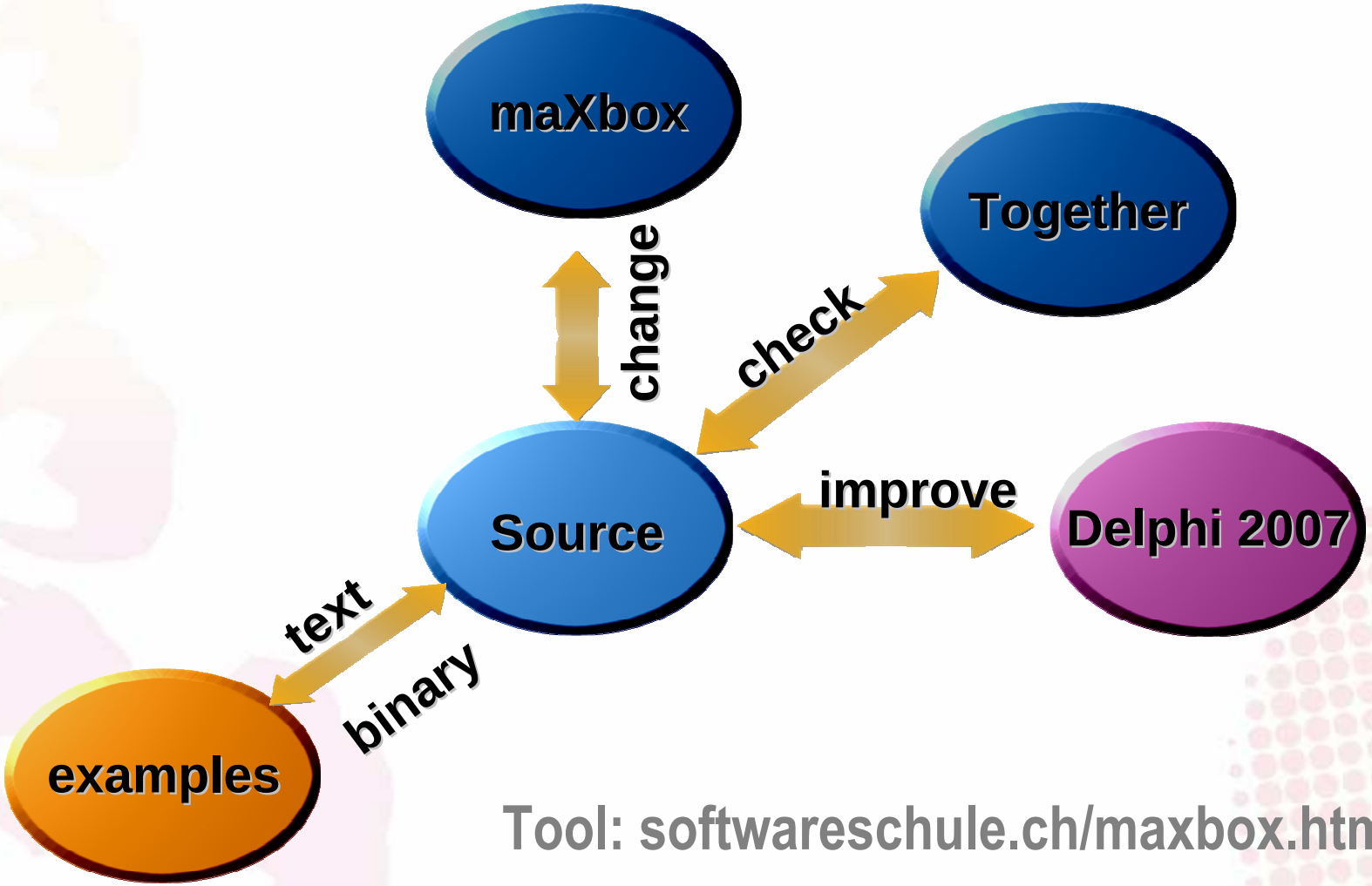
- What‘s Refactoring ?
- How to recognize Bad Code and what‘s the goal ?
- The Law of Demeter
- Process of Refactoring
- Techniques in Delphi 2007
- Introduction to DUnit
- Experiences, Code Analysis and Optimisations
- Refactoring and other Tools

Refactoring Context



Advantage: automatisation and integration of new methods, better testability and readability (simplification)

Course Preparation



Tool: softwareschule.ch/maxbox.htm

Refactoring: What‘s all about ?

Never touch a running system ?!:

```
//System.Get just reads the contents of memory locations:
Get(a, v);          or  v:= M[a];
o.IntToStr;         or  v:= IntToStr(o);
GetMem(p, Count * SizeOf(Pointer)); or GetMem(FDataPtr, MemSize);
```

If you can‘t see how to easily expand, test or maintain your code, it‘s probably time to refactor!

- How to recognize bad smells ?
- Refactor to change code, reduce to the Max!
- Build Test Class and run test to confirm it‘s okay

Refactoring: Definition

- Changes made to a program that only change its organization (structure), not its function.
- Behaviour preserving program transformations.
- Simplification: Your classes are small and your methods too; you've said everything and you've removed the last piece of unnecessary code.

When and why Refactoring ?

After a Code Review

By changes of a release

Redesign with UML (Patterns or Profiles)

Law of Demeter not passed

Bad Testability (FAT or SAT)

- Work on little steps at a time
- Test after each step
- Each method should do one thing
- Modify not only structure but also code format

When to refactor ?

- If a new feature seems hard to implement, refactor.
- If a new or changed feature created some ugly code, refactor.
- When you can’t stand (understand) to look at your code, refactor.

UEB: 8_pas_verwechselt.txt

Why is Refactoring important?

- Only defense against software decay.
- Often needed to fix reusability bugs.
- Lets you add patterns or templates after you have written a program;
- Lets you transform program into framework.
- Lets you worry about generality tomorrow;
- Estimation of the value (capital) of code!
- Necessary for beautiful software.

When Refactoring I ?

Bad Coordination

- Single Instance with Singleton
- Inconsistence with Threads
- Access on objects with Null Pointer
- Bad Open/Close of Input/Output Streams or I/O Connections
- Check return values or idempotence
- Check break /exit in loops UEB: 14_pas_primetest.txt
- Modification of static or const code
- Access of constructor on non initialized vars

When Refactoring II ?

Bad Functionality (Organisation)

- General Code Smoke Test (compilation) failed
- Comments, Coding Conventions (D. by Contract)
  - Type safety (pointers?), Declare, RTTI, Thread Safeness
- Exception Handling, Resource Leaks (memory)
- Import Statements (uses at runtime)
- Functions and Bindings (design time)
- No Bug Fixes & Release Planning
- No Versioning & System Handbook

UEB: 15_pas_designbycontract.txt

When Refactoring III ?

Bad Structure

- General Code Size (in module)
- Coupling (between classes or units)
  - Cyclic Dependency, Declare+Definition, ACD-Metric
- Cohesion (in classes)
- Layers, Tiers and Control Structures (runtime)
- Interfaces or Packages (design & runtime)
- Static, Public or Private (inheritance or delegate)
- Too big classes or units?

UEB: 10_pas_oodesign_solution.txt

Top Ten of Bad Smells

1. Duplicated or Dead Code
2. Long Method
3. Large Class (Too many responsibilities)
4. Long Parameter List (Object is missing)
5. Case Statement (Missing polymorphism)
6. Divergent Change (Same class changes differently depending on addition)
7. Shotgun Surgery (Little changes distributed over too many objects or procedures → patterns missed)
8. Data Clumps (Data always use together (x,y -> point))
9. Middle Man (Class with too many delegating methods)
10. Message Chains (Coupled classes, internal representation dependencies)

Refctor/Review Checklist

1. **Standards - are the Pascal software standards for name conventions being followed?**
2. **Bugs - Are the changes generally correct?**
3. **Are the Requirements Well Understood (Multilang)?**
4. **Are all program headers completed?**
5. **Are changes commented appropriately?**
6. **Does documentation use Correct Grammar?**
7. **Are release notes Clear? Complete?**
8. **Installation Issues, Licenses, Certs. Are there any?**
9. **Version Control, Are output products clear?**
10. **Test Instructions - Are they any? Complete?**

Precondition & Goals…

- Dokus, Testklassen zu Algorithmen
- Vorabversionen und Updates
- Installation, Support, Hotline der Firma
- Volle Verfügbarkeit über den Sourcecode
- Compilationsfähigkeit erstellt ($D, $L, tdb32info)
- Benchmarks als Performancekriterium
- Know-how Transfer durch Entwickler
- Tools und Review Kriterien vorhanden
- Sprache und Form bei Dokumenten festlegen
- Referenzarchitektur und Patterns bekannt

Modularisierung

- Patterns Suche, Test mit einem Dependency Analyzer
  Suchen nach einem Schichtenmodell (MVC etc.)
- Ist das Programmdesign optimal gewählt?
  - Zu viele globale Variablen?
  - Schlecht aufgeteilte Methoden bzw. Klassen?
  - Redundante Codesequenzen?

  Modularisierung (zur design- oder runtime ?)

```
TWebModule1 = class(TWebModule)
    HTTPSoapDispatcher1: THTTPSoapDispatcher;
    HTTPSoapPascalInvoker1: THTTPSoapPascalInvoker;
    WSDLHTMLPublish1: TWSDLHTMLPublish;
    DataSetTableProducer1: TDataSetTableProducer;
```

Modules of Classes

Large classes

- More than seven or eight variables
- More than fifty methods

- You probably need to break up the class

Components (Strategy, Composite, Decorator)

Inheritance

```
TWebModule1 = class(TWebModule)
    HTTPSoapDispatcher1: THTTPSoapDispatcher;
    HTTPSoapPascalInvoker1: THTTPSoapPascalInvoker;
    WSDLHTMLPublish1: TWSDLHTMLPublish;
    DataSetTableProducer1: TDataSetTableProducer;
```

Extendibility

Are Interfaces, Packages available?

```
type
  { Invokable interfaces must derive from IInvokable }
  IVCLScanner = interface(IInvokable)
  ['{8FFBAA56-B4C2-4A32-924D-B3D3DE2C4EFF}']
    function PostData(const UserData : WideString; const
          CheckSum : DWORD) : Boolean; stdcall;
    procedure PostUser(const Email, FirstName,
          LastName : WideString); stdcall;
```

Extendibility II

Are Business Objects available (Extensions)?

In a simple business object (without fields in the class), you do have at least 4 tasks to fulfil:

1. The Business-Class inherits from a Data-Provider
2. The query is part of the class
3. A calculation or business-rule has to be done
4. The object is independent from the GUI, GUI calls the object

"Business objects are sometimes referred to as conceptual objects, because they provide services which meet business requirements, regardless of technology".

Modules and Extendibility

Are Factory Patterns available ?

Shows the use of a simple factory method to generate various reports or forms:

1. Register a list with objects
2. Generate objects at runtime
3. A report or business-rule has to be done
4. The objectlist is independent from GUI, GUI calls the list and the list is expandable

“Use collections or lists, not case of or if statements.”

UEB: 55_pas_factorylist.txt

Wartbarkeit

- Namenskonvention, Namensräume, Kommentare
- Bridge zwischen Interface und Implementation (siehe auch Modularisierung)
- Installation und Configuration Guide durchsehen
- Abhängigkeiten der Module klären:

```
for i:= 0 to SourceTable.FieldCount - 1 do
    DestTable.Fields[i].Assign(SourceTable.Fields[i]);
DestTable.Post;
```

Wartbarkeit II

- die Dokumentation, insbesondere die exakte Spezifikation von Schnittstellen (Interfaces)
- die lokale Verständlichkeit von Anweisungen resp. von Kommentar im Code
- das Vermeiden von Duplicated, Dead Code
- das Vermeiden globaler Variablen
- die Parametrisierbarkeit von Methoden
- in das Programm eingebaute Prüfungen der Annahmen, die man über Zustände hat (Assertions)
- ein möglichst großer Umfang von automatisch ausführbaren Tests für das System

Wartbarkeit III

- die Dokumentation, insbesondere die Verständlichkeit von Modulen (Komponenten)

```
Program MailingLabels  //pseudo code referenzieren!
Begin
   ask the user for the name of the list
   read that list into memory
   ask the user how the list should be sorted: name or zip code
   sort the list
   ask the user about where to send the output: screen or printer
   print the labels
End.  { Program MailingLabels }
```

Testbarkeit (White Box)

- Es lassen sich beim Ändern einer Subroutine schnell mal die davon abhängigen Prozeduren aufzeigen (z.B. Bindings Report).

Die Änderung einer Funktion hat Einfluß auf die davon abhängigen Funktionen, die im Bindings Report aufgelistet werden, demzufolge ein erneuter Test der Funktion der Qualität nur zugute kommt.

- Viele haben ja ein eigenes, „automatisiertes“ Verfahren entwickelt, um Code auf Fehler hin zu durchsuchen (Testprotokolle).
- Wurde das Error Handling, Logger getestet ?
- Sind Testklassen wie DUnit im Einsatz?

Testability – We should avoid:

Bad Naming (no naming convention)

Duplicated Code (side effects)

Long Methods (to much code)

Temporary Fields (confusion)

Long Parameter List (Object is missing)

Data Classes (no methods)

- Large Class
- Class with too many delegating methods
- Coupled classes

UEB: 33_pas_cipher_file_1.txt

Software Safeness I

- Avoid pointers as you can
- Ex. of the win32API:

```
pVinfo = ^TVinfo;
function TForm1.getvolInfo(const aDrive: pchar; info:
    pVinfo): boolean;
// refactoring from pointer to reference
function TReviews.getvolInfo(const aDrive: pchar; var info:
    TVinfo): boolean;
```

> "Each pointer or reference should be checked to see if it is null. An error or an exception should occur if a parameter is invalid."

Software Safeness II

- Check Exception Handling

```
function IsInteger(TestThis: String): Boolean;
begin
  try
    StrToInt(TestThis);
  except
    on EConvertError do
      result:= False;
  else
    result:= True;
  end;
end;
```

Fehleranalyse und Refactor Relevanz

FehlerTyp-Nr / Bezeichnung / Verteilung / Relevant

10 Dokumentation - Kommentare, 0.5, Ja

20 Syntax - Syntax-Fehler (vor dem Kompilieren), 49.3, Nein

30 Build – package library, version control, 15.4, Ja

40 Assignment - Dekl., Doppelte Bezeichnungen, 8.8, Ja

50 Interface - Prozedur-Aufrufe und Referenzen, I/O, 1.5, Ja

60 Logging - Fehler-Nachrichten, Checks, 2.2, Ja

70 Daten - Struktur, Inhalt, Format, 1.5, Ja

80 Funktion - Schleifen, Rekursionen, Logik, 16.6, Ja/Nein

90 System – Konfiguration 1.5, Ja

100 Umgebung - Absturz, Compiler, Support, 3.7, Ja

Law of Demeter

You should avoid:

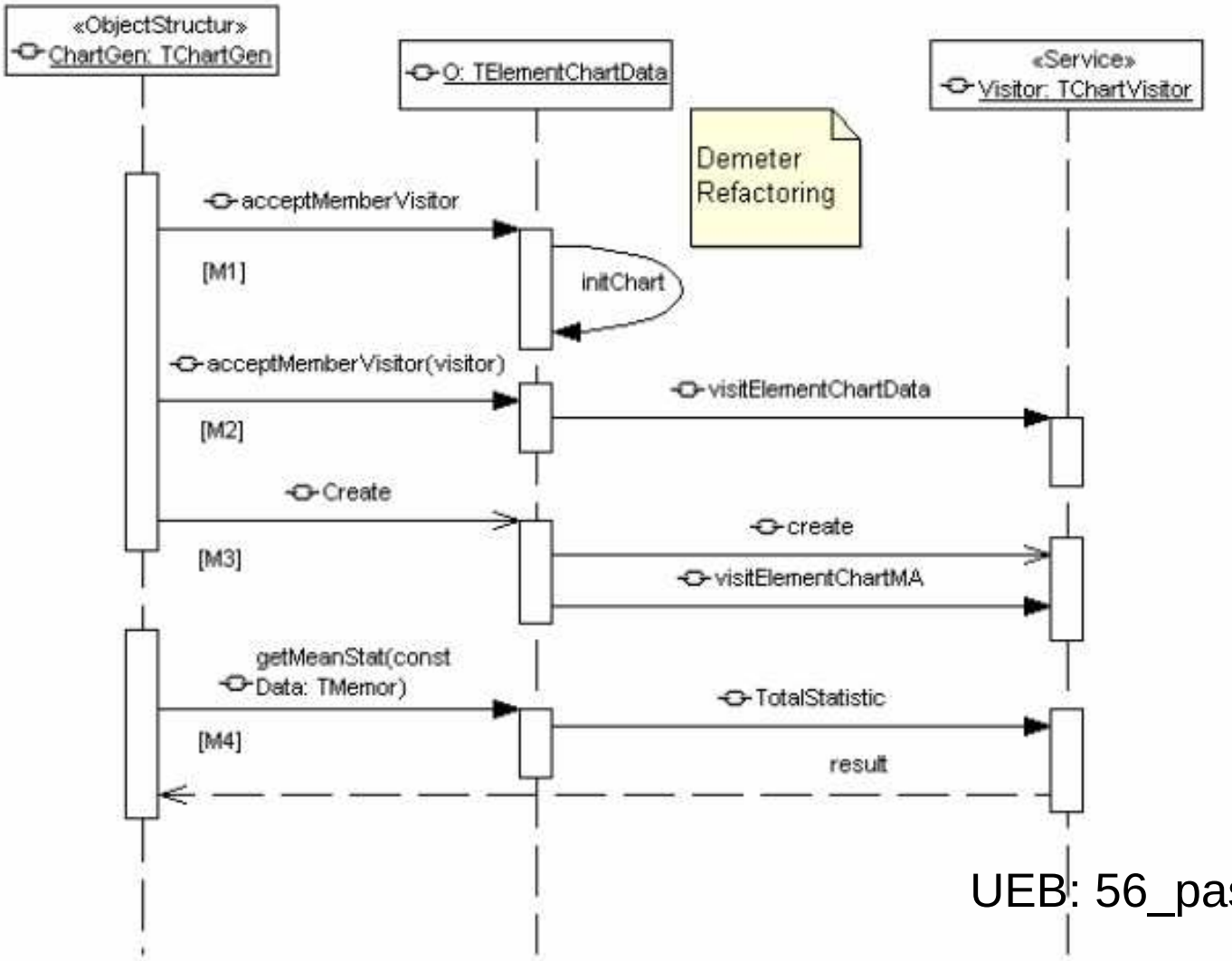
- Large classes with strange members
- More than seven or eight variables
- More than fifty methods
- You probably need to break up the class
  Components in (Strategy, Composite, Decorator)

Das Gesetz von Demeter (don‘t talk to strangers) besagt, dass ein Objekt O als Reaktion auf eine Nachricht m, weitere Nachrichten nur an die folgenden Objekte senden sollte:

Demeter konkret

1. [**M1**] an Objekt O selbst
   Bsp.: self.initChart(vdata);
2. [**M2**] an Objekte, die als Parameter in der Nachricht m vorkommen
   Bsp.: O.acceptmemberVisitor(visitor)
   visitor.visitElementChartData;
3. [**M3**] an Objekte, die O als Reaktion auf m erstellt
   Bsp.: visitor:= TChartVisitor.create(cData, madata);
4. [**M4**] an Objekte, auf die O direkt mit einem Member zugreifen kann
   Bsp.: O.Ctnr:= visitor.TotalStatistic

Demeter Test as SEQ



UEB: 56_pas_demeter.txt

Refactoring Process

The act of serialize the process:

- Build unit test
- Refactor and test the code (iterative!)
- Check with Pascal Analyzer or another tool
- Building the code
- Running all unit tests
- Generating the documentation
- Deploying to a target machine
- Performing a “smoke test” (just compile)

Let‘s practice

- 1
- 11
- 21
- 1211
- 111221
- 312211
- ???

Try to find the next pattern, look for a rule or logic behind !

Before R.

```
function runString(Vshow: string): string;
var i: byte;
Rword, tmpStr: string;
cntr, nCount: integer;
begin
cntr:=1; nCount:=0;
Rword:=''; //initialize
tmpStr:=Vshow; // input last result
for i:= 1 to length(tmpStr) do begin
  if i= length(tmpstr) then begin
     if (tmpStr[i-1]=tmpStr[i]) then cntr:= cntr +1;
     if cntr = 1 then  nCount:= cntr
     Rword:= Rword + intToStr(ncount) + tmpStr[i]
  end else
   if (tmpStr[i]=tmpStr[i+1]) then begin
      cntr:= cntr +1;
      nCount:= cntr;
   end else begin
     if cntr = 1 then cntr:=1 else cntr:=1; //reinit counter!
     Rword:= Rword + intToStr(ncount) + tmpStr[i] //+ last char(tmpStr)
end;
end; // end for loop
result:=Rword;
end;
```

UEB: 9_pas_umlrunner.txt

After R.

```
function charCounter(instr: string): string;
var i, cntr: integer;
      Rword: string;
begin
cntr:= 1;
Rword:=' ';
  for i:= 1 to length(instr) do begin
    //last number in line
    if i= length(instr) then
      concatChars()
    else
    if (instr[i]=instr[i+1]) then cntr:= cntr +1
    else begin
      concatChars()
      //reinit counter!
      cntr:= 1;
    end;
  end; //for
 result:= Rword;
end;
```

UEB: 12_pas_umlrunner_solution.txt

Testfunction

```
function teststring(vstring: string): integer;
var a, b, i: integer;
begin
 for i:= 1 to length(vstring) do begin
   a:= (ord(vstring[i]))
   b:= b + a;
 end
 result:= b
end;
```

Check with a reference value:

```
    if teststring(charCounter(RUN2)) = 578 then …
   //CheckEquals(578, teststring(charCounter(RUN2)))
```

Refactoring Techniken

Einheit	Refactoring Funktion	Beschreibung
Package	Rename Package	Umbenennen eines Packages
Package	Move Package	Verschieben eines Packages
Class	Extract Superclass	Aus Methoden, Eigenschaften eine Oberklasse erzeugen und verwenden
Class	Introduce Parameter	Ersetzen eines Ausdrucks durch einen Methodenparameter
Class	Extract Method	Heraustrennen einer Codepassage
Interface	Extract Interface	Aus Methoden ein Interface erzeugen
Interface	Use Interface	Erzeuge Referenzen auf Klasse mit Referenz auf implementierte Schnittstelle
Component	Replace Inheritance with Delegation	Ersetze vererbte Methoden durch Delegation in innere Klasse
Class	Encapsulate Fields	Getter- und Setter einbauen
Modell	Safe Delete	Löschen einer Klasse mit Referenzen

Consider the Options

The tool should provide options. Whenever you rename certain entities, a tool should automatically replace all relevant identifiers in code in order to keep code consistent

Each refactoring operation has its own set of constraints!

- How to treat extracted code in original method – you can choose between
- Leave – leaves the original text unchanged,activates the new method.
- Select – leaves the original text unchanged and selects it (does not activate the new method).
- Comment – puts comment braces around the original text and activates new method.
- Remove – removes the original code and activates new method.
- copy the vars to the new method,
- add the new class to the same unit as the source method's class (if any).

Constraints in Delphi

- If an error results from a refactoring, the engine cannot apply the change. For example, you cannot rename an identifier to a name that already exists in the same declaration scope. If you still want to rename your identifier, you need to rename the identifier that already has the target name first, then refresh the refactoring.
- You can also redo the refactoring and select a new name. The refactoring engine traverses parent scopes, searching for an identifier with the same name. If the engine finds an identifier with the same name, it issues a warning.

Refactoring with D9

If you only select by install win32 refactor won't work!!

- Delphi 2005 provides following refactoring operations:
- Symbol Rename (Delphi, C#)
- Extract Method (Delphi)
- Declare Variable and Field (Delphi)
- Sync Edit Mode (Delphi, C#)
- Find References (Delphi, C#)
- Extract Resource string (Delphi)
- Find unit /import Namespace(Delphi)
- Undo (Delphi, C#)
- Examples…

Refactoring in D9 SP3
Corrections done

- When extracting methods involving DWords, they are getting redeclared as Integers.
- When the parameter has the same name as the type, 'Find local Reference' and 'rename parameter' fail
- Rename refactoring fails on overloaded procedures with at least one parameter of the same name.
- After using ExtractMethod, dragging a code snippet from the Tools palette causes a stack overflow.
- Rename for components with event handlers doesn't work for VCL and VCL.Net apps after you save the project

Refactoring with D11

Delphi 2007 (since2006) provides following new refactoring operations:

- **New!** Introduce Variable refactoring d D #
  New! Introduce Field refactoring d D #
- **New!** Inline Variable refactoring d D #
- **New!** Change Parameters refactoring d D #
- **New!** Safe Delete refactoring d D #
- **New!** Push Members Up / Down refactoring d D #
  New! Pull Members Up refactoring d D #
- **New!** Extract Superclass refactoring d D #
  New! Extract Interface refactoring d D #
- **New!** Move Members refactoring d D #
- D2007 Import Namespace and better Undoing
- Examples…

UEB: demos/threads/thrddemo.exe

Refactoring und DUnit

Das Testen mit DUnit kann an Refactoring koppeln und setzt auf drei Elementen auf:

- Das eigentliche Testobjekt, fixture genannt
- Der Testfall zum Objekt, action genannt
- Das Resultat nach dem Testfall, check genannt

Als einfaches Beispiel sei hier das fortlaufende Zählen von Zeichen erwähnt (siehe practice). Das Resultat kann Positiv, alphanumerisch oder ein unerwarteter Fehler sein. Der Check testet, ob das Resultat dem Referenzwert entspricht.

DEMO: review*.exe

DUnit Conditions

We should keep 3 things in mind:

- Test methods are parameter-less procedures.
- Test methods are declared published.
- Test methods generally map methods from an application

But it's possible to set private methods which can be used independent of an application, like the function WaveFileCheck()!

Finding Testcases in DUnit

But finding meaningful testcases isn't that easy. As we grow with experiences I will show few testcases to inspire and deliver some ideas. Note that DUnit builds a separate instance of the class for each method that it finds, so test methods cannot share instance data. We dig into 4 methods:

1. **Test Roboter for Forms**
2. **Test an Exception (like testing a disaster recovery)**
3. **Test the Fileformat to prevent Viruses, Hoaks and so on**
4. **Test a Reference Value**

It's important to be a little familiar with DUnit. First we build the testclass which will test our units: (Testing procedures should be published so RTTI can find their method address with the MethodAddress method.) Add an instance variable for every fixture (i.e. starting situation like myForm, FFull...) you wish to use.

Testcase 1

1. The following calls check() to see if everything went OK. It's up to you to enlarge the check with more calls or methods like a test roboter does.

```
procedure TTestCaseMethods.testFormRobot;
begin
  myForm.lblserial.Caption:= 'another text test';
  myform.ShowModal;
  check(myForm is TForm, 'this isn’t right type');
end;
```

Testcase 2

2. The following example extends exception handling to do a couple of tests which can prove exception handling works. This case shows that it is possible to test the exceptions of the test subject:

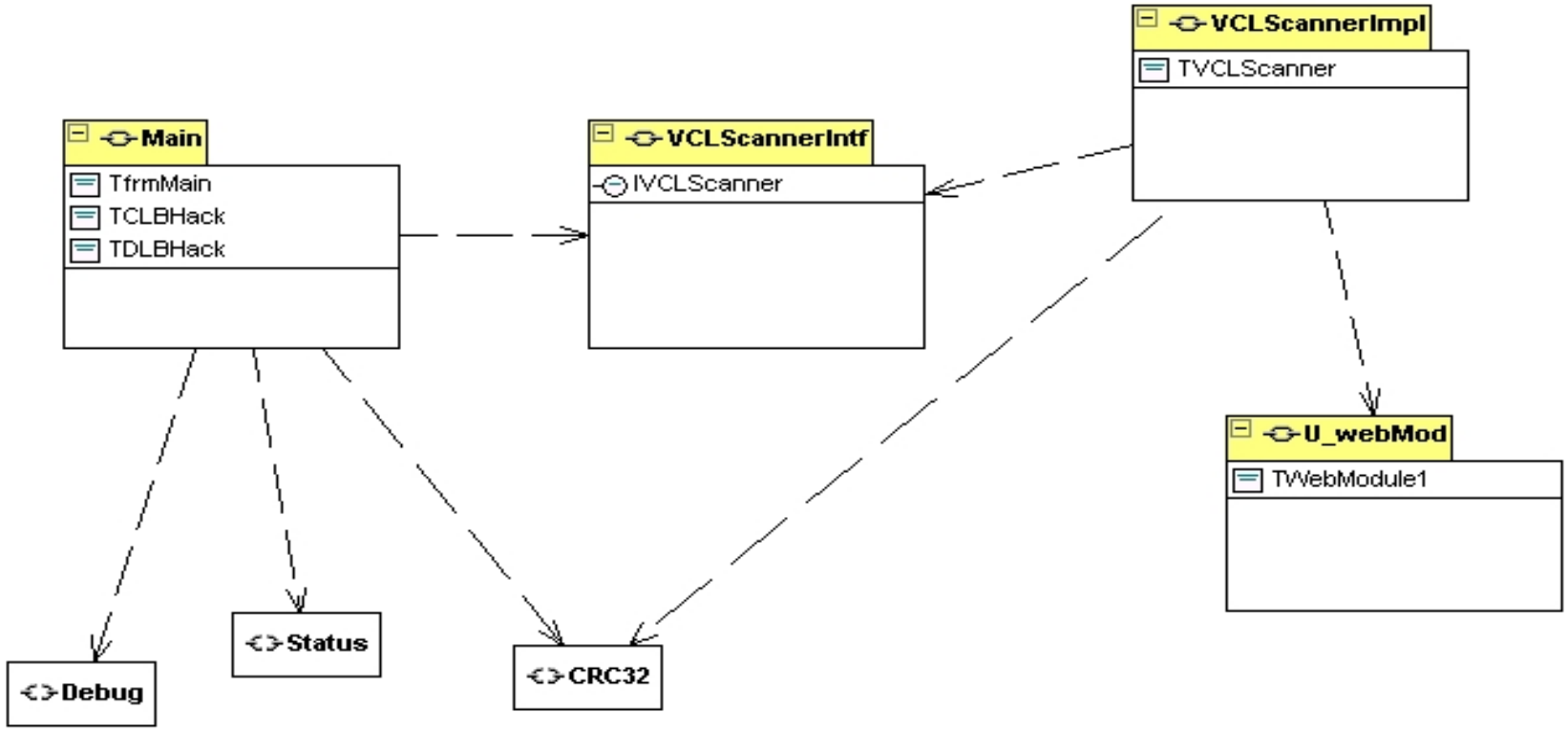
```
procedure TTestCaseMethods.testExceptionIndexTooHigh;
begin
  try
    fFull.Items[2];
    //Check(false, 'should have been an EListError.');
  except on E: Exception do
    Check(not(E is EListError));
  end;
end;
```

Testcase 3

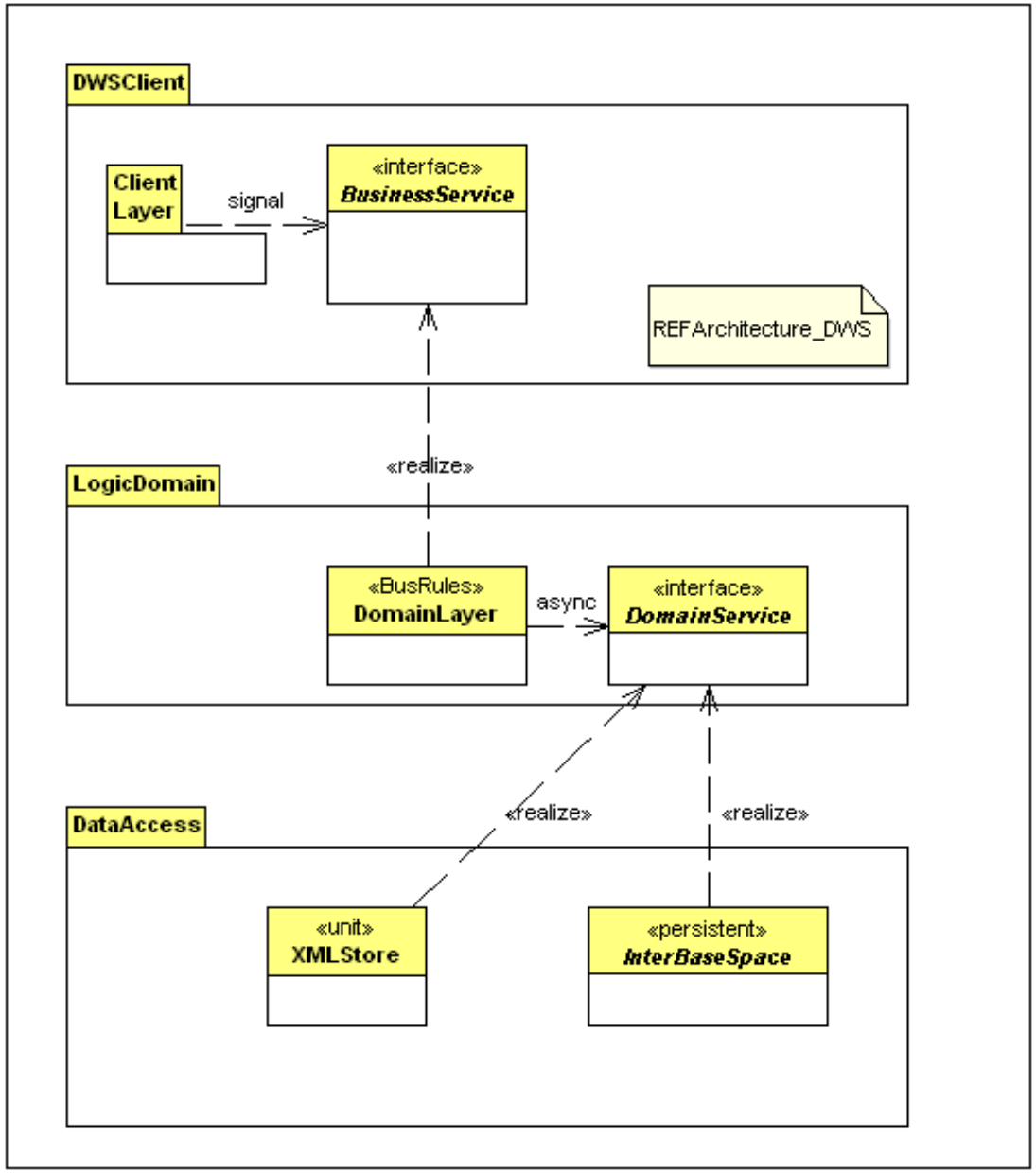
3. At last a reference test in this case a function which returns a value, so we can refactor as much we want as long the reference value is still the same. This test checks with checkEquals() if the rate of an income results the right value 1040. // When checkEquals fails, it will end up in the TestResult as a failure (not reference value, expected: <1040> but was: <1042>).

```
procedure TTestCaseMethods.testReferenceValue;
begin
  incomeRef:= createIncome2;
  incomeref.SetRate(4,1);
  checkEquals(1040, incomeRef.GetIncome(1000), 'no
                                          reference value');
  incomeRef.FreeObject;
end;
```

Test Units in practice



Experiences armasuisse components



DEMO: magazin/dws

Patterns & Refactoring

with Factory Example

Communi-cation	Design Pattern	Platform Independend	Loose Coupling
sync	Abstract Factory, Prototype	middle	yes (WSDL)
sync	Bridge, Observer, MVC	middle high	middle
sync/async	Chain of Responsibility	middle	yes (local or WAN)
sync	Command	very high	middle
sync/async	Facade	very high	very high
sync/async	Mediator	very high	very high

Anti Flexibility ?

with TSQLConnection Bsp.

```
Connection:= TSQLConnection.Create(NIL);
 with Connection do begin
   ConnectionName:= 'VCLScanner';
   DriverName:= 'INTERBASE';
   LibraryName:= 'dbexpint.dll';
   VendorLib:= 'GDS32.DLL';
   GetDriverFunc:= 'getSQLDriverINTERBASE';
   Params.Add(,user_name=SYSDBA');
   Params.Add('Password=masterkey');
     with TWebModule1.create(NIL) do begin
      getFile_DataBasePath;
      Params.Add(dbPath);
```

Testability Code

- Exception handling

```
{$IFDEF DEBUG}
  Application.OnException:= AppOnException;
{$ENDIF}
```

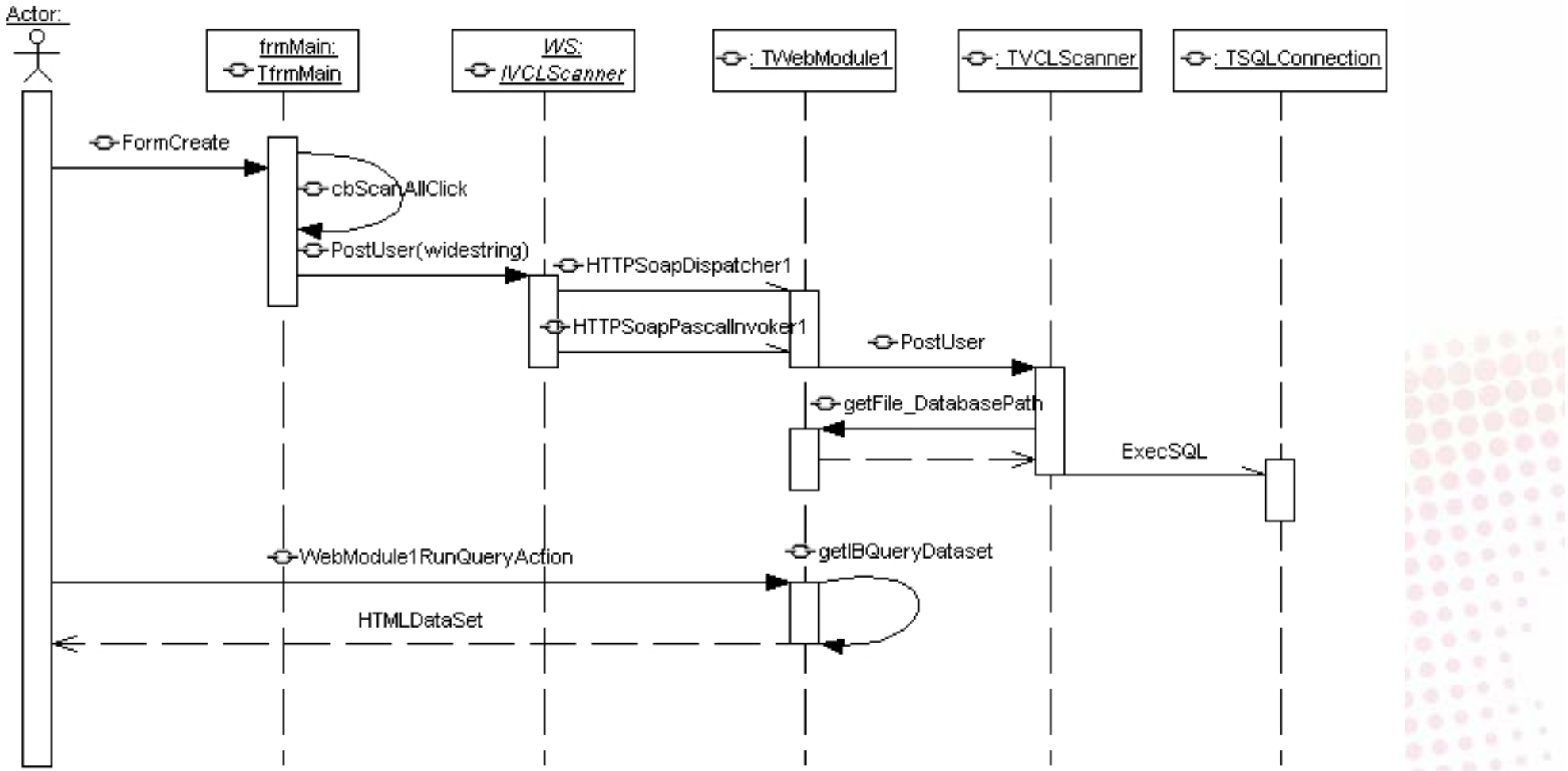
- Assert function

```
accObj:= TAccount.createAccount(FCustNo,
                                std_account);
assert(aTrans.checkOBJ(accObj), 'bad OBJ cond.');
```

- Logger

```
LogEvent('OnDataChange', Sender as TComponent);
LogEvent('BeforeOpen', DataSet);
```

Test Based on SEQ



Optimizations

Some tips and hints to optimize your code:

1. one statement and short var
2. assert call, IfThen()
3. use snippets
4. naming convention
5. $IF-directive
6. Const instead of var parameters
7. extract code from dfm
8. comment your code
9. uses list

Optimization – one and var

- one statement and with speed

An easy one states that only one statement should exists for every source line, like next and put a var lifetime as short as possible

```
begin ..
  I := 5; CallProc(I); //bad
//better
begin ..
I:= 5;
CallProc(I);

var Rec: TMyRec;
begin ..
   with Rec do begin
   .. title:= ´Hello Mars';
   end;
end;
```

Optimization – assert and IfThen()

assert call

Use assert calls as much, but don't forget the $C - setting is active or not. This means that identifiers used in the Assert procedure call, will be registered in your tests, but when compiled by Delphi, this code line will be (should) stripped out if $C- is defined.

```
procedure MyProc(p: pointer);
begin
   Assert(p <> NIL, ‘’);

nMax:= IfThen(nA > nB, nA, nB) // no if/then/else
```

Optimization – use code snippets

```
Procedure CopyRecord(const SourceTable, DestTable : TTable);
var i: Word;
        begin
          DestTable.Append;
          For i:= 0 to SourceTable.FieldCount - 1 do
             DestTable.Fields[i].Assign(SourceTable.Fields[i]);
          DestTable.Post;
        end;
```

Optimization – naming convention

TBitBtn;bitn
TButton;btn
TCheckBox;chk
TComboBox;cbo
TRadioButton;rb
TRadioGroup;rg .....

Ordinary types start with "T"
Exception types start with "E"
Pointer types tart with "P"
Interface types start with "I"
Class fields by properties (read/write) start with "F"
Method pointers start with "On/Before/After"

Optimization - **$IF-directive and const**

In Delphi 6, the new $IF-directive was introduced. The $IF-directive is followed by an expression, that evaluates to TRUE or FALSE. Differentiate your code!

Avoid var parameters that are used, but never set in the subprogram they belong to. You may omit the var keyword, or change it to a const parameter. Declare with the const directive, resulting in better performance since the compiler can assume that the parameter will not be changed.

```
procedure MyHexMaxProc(const i: integer); //not var
  begin ..
    if i = 5 then // !! i is not set
      begin .. end;
  end;
```

by the way: use not more than 3 parameters in a method, so the compiler can build it with fastcall directives, means registers are used instead of stack!

Optimization - extract code from dfm

```
Connection:= TSQLConnection.Create(NIL);
 with Connection do begin
   ConnectionName:= 'VCLScanner';
   DriverName:= 'INTERBASE';
   LibraryName:= 'dbexpint.dll';
   VendorLib:= 'GDS32.DLL';
   GetDriverFunc:= 'getSQLDriverINTERBASE';
   Params.Add('User_Name=SYSDBA');
   Params.Add('Password=masterkey');
     with TWebModule1.create(NIL) do begin
      getFile_DataBasePath;
      Params.Add(dbPath);
```

Optimization - comment your code

```
1.  Connection := TSQLConnection.Create(nil);
2.   with Connection do begin ...
3.  //after connection, create dataset to it
4.  DataSet := TSQLDataSet.Create(nil);
5.   with DataSet do begin ...
6.     SQLConnection := Connection;
7.  //build the command string
8.  CommandText :=
9.     Format('insert into KINGS values("%d","%s", "%s",
       "%s","%s")',[10, Email,FName,LName, logdate]);
```

Optimization – uses list

We know the linker is a smart one, but nevertheless init and finals are executed. Removing unused uses references has multiple benefits:

- less code to maintain, no need to bother about code that's not used
- code from initialization and finalization sections in unused units is not linked in and run, reducing the size of the EXE
- compilation runs smoother and quicker

When you drop a VCL component on a Delphi form the Delphi IDE automatically adds the unit or units required by the component to the interface section uses statement. This is done to ensure that the form file (DFM/XFM) can locate the code needed to stream the form and components. Even if you later remove the component, the units are not deleted from the uses statement.

(UEB: which unit is not necessary in ThSort ?)

Expand your Refactoring

Problem: You must refactor to make it compatible with Unit Testing or you must create a Unit Test Class before refactor !!

Refactoring in UML Diagrams (e. g. state -> superstate)

[Code Patterns]

[Syncedit]

[Macro recorder], [Code Completion], [Version Control]

Tools in practice:

„As far as I'm concerned, Castalia or Pascal Analyzer is a must have for any Delphi developer - I know I'd be lost without it now."

Refactoring Links:

- Delphi 9…\BDS\3.0\Help.pdf (1656 p.) ab S. 85
- Delphi 11 Tools: http://www.modelmakertools.com/
- Report Pascal Analyzer: http://www.softwareschule.ch/download/pascal_analyzer.pdf
- *Refactoring* Martin Fowler (1999, Addison-Wesley)
- Changing the structure of code without changing the functionality
- http://www.refactoring.com
- Discussion site on code smells
- http://c2.com/cgi/wiki?CodeSmell
- Castalia3 - http://www.delphi-expert.com/castalia2/
- http://www.informatics.sussex.ac.uk/courses/sde/Course-Outline/Lecture_slides/15/slides4.pdf

Q&A

max@kleiner.com
www.softwareschule.ch