

better  
office

## Verbesserung von Softwarequalität in Delphi

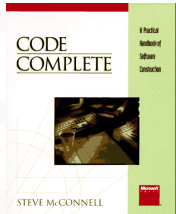
Jeroen Pluimers  
Edwin van der Kraan  
[jpluimers@better-office.com](mailto:jpluimers@better-office.com)  
[evdkraan@better-office.com](mailto:evdkraan@better-office.com)  
better office benelux






better  
office

## Welcome

- Target of this course
- better office benelux





better  
office

## Why Software Quality ?



- Software is getting more and more important in all aspects of (human) life,  
so failures have a big impact
- All serious applications need modifications in due time, software life-cycles are not predictable
- Quality reduces stress

better  
office

## How can you study Software Quality ?



- Books and articles have been written
  - Increases your insight
  - Stimulates good practices
  - Enables you to argue about a decision
- .. But that is not the emotional side
- Building quality software depends on your passion for
  - developing the right way and
  - do it right the first time.

better  
office

## Software Quality: the human factor

- <images>
  - Group of people with communication
  - Draw right and left handed
  - Emotion versus Technique






better  
office

## RAD

‘Developing software faster  
than you do now’

Steve McConnell

## What is RAD and what does it have to do with Quality ?

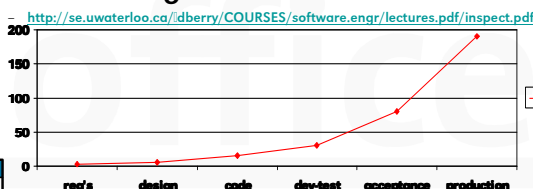
- RAD = Rapid Application Development
  - Not just for the first phase
- Doing it right the first time helps a lot
- Low-effort, high-payoff techniques
- Decisions like
  - Make or buy
  - Reuse
- But:
  - there is no silver bullet

## Low-effort / High-payoff techniques

- Aim to find
  - low-effort techniques,
  - principles and guidelines that have a high payoff
- For instance:
  - improving the quality of methods and comments can greatly ease the task of creating documentation ('the asterisk problem')

## Upstream / downstream

- Where did you introduce that bug ?
  - Create processes that help you to find bugs as soon as possible
  - Fix the bug when it's found



## Frameworks

- Delphi is a fabulous general-purpose development tool.
- But it gives the average developer too much freedom and too many choices that have to be made thoughtfully.
- The consequences are:
  - stress
  - wrong decisions
  - repetitive and boring work

## Frameworks

- Set of
  - Components, objects, routines
  - Guidelines
  - Documentation templates
  - Tools
- that lays on top of the development-tool with the aim to standardize and speed-up software construction.

## What do you need for quality ?

- You need a framework for rapidly developing quality Delphi applications
- The design of your framework should be based on principles that are key to quality

## Key Principles

...in a platform/language  
agnostic way



## Break up your sourcecode !

- Gains:

- Focus attention and energy
- Decrease complexity for the developer
- Increase ( conceptual ) overview
- Information Hiding
- Localise changes



## Cohesion

- “The act or condition of sticking together; a tendency to cohere; the force with which molecules cohere”
- Strong cohesion is good; weak cohesion is bad
- It should be basically impossible to separate the parts within a ‘unit’



## Coupling

- “A link connecting railway carriages etc.; a connection between two systems”
- Loose coupling is good; tight coupling is bad
- A communication-path should be as less specific as possible ( no knowledge of internals of the other unit )



## Interfaces

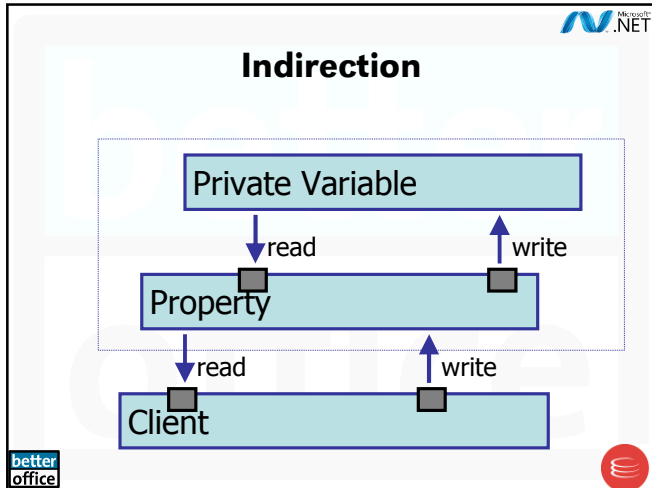
- “A surface forming a common boundary between two regions; a point where interaction occurs between systems”
- Interfaces should be defined and documented formally



## Indirection

- “Not going straight to the point”
- Indirection is very good; up to a certain point
- Indirection gives you more control over communication in a centralized location





Microsoft .NET

## Simplify

- “Make simple; make easy or easier to do or understand”
- Simplification is good
- Make your application and especially the structure of the application as simple as possible
- Low-tech is more maintainable than high-tech

better office

Microsoft .NET

## Create Documentation

- “The most important part of design that gets started when the application is almost finished and often is done by handwritten notes in a scrapbook”
- Documentation should be done right the first time
- Create guidelines and templates for Word and source-code

better office

Microsoft .NET

## Create Documentation

- Source code is often the only accurate description of the software. So make sure it is readable and of the highest quality.

better office

Microsoft .NET

## Use OOP wisely

- Abstraction
- Encapsulation
- Inheritance
  - Multiple Class Inheritance is not such a good idea
- Polymorphism
  - (Multiple) interface implementation is a good idea
- When used properly, OOP can increase quality, but if not it will make things very bad

better office

better office

## Delphi

some key principles in more detail...

Microsoft .NET

## Self-documenting code

- The only part of the software that is guaranteed to be done is the code!
- So create self-documenting code
  - Variable and method names
  - Proper layout



## Comments

- A comment can be
  - Repeat of the code
  - Explanation of the code
  - Marker in the code
  - Summary of the code
  - Description of the code's intent



## Comments

- Make writing comments as smooth as possible
  - Layout
    - No special formatting (VCL unit headers are a dork!)
    - No endline `//` comments, except for declarations
  - PDL-to-code (Program Design Language)
    - Start with some comments on the 'what'
    - Then replace them with the 'how'
  - Comment as you go
  - Avoid excessive commenting



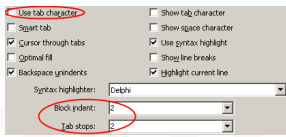
## Comments

- Make every comment count
  - Do not comment for silly purposes
  - Document surprises and workarounds
  - Avoid abbrevtns

'Don't document bad code – rewrite it'  
Kernighan and Plauger (1978)



## Constructing readable sourcecode

- Sourcecode Layout
  - Consistency is important
  - Editor options:
 
  - Quality layout matches the logical structure of the code



## Constructing readable sourcecode

- Sourcecode should be formatted for the human reader, not the computer
  - Accurately represent the logical structure
  - Consistently represent the logical structure
  - Improve Readability
  - Withstand modifications



## Constructing readable sourcecode

- Use whitespace
  - Grouping
  - Blank lines
  - Make logical expressions readable
- Alignment
  - Groups of related assignments
  - Data declarations
- Indentation ( 2 - 4 spaces is best )
- Use more parentheses



## Constructing readable sourcecode

- Indentation styles

<http://courses.knox.edu/cs322/322PDFlectures/L16Coding.pdf>

- Pure blocks are not an option in pascal
- Endline layout is horrible

```
if (i > 1000) then begin
    x := 21;
    y := 36;
end
else z := 15;
```

```
if (i > 10) then
    statement1;
statement2;
end;
```

- Emulated pure blocks
- Begin-end block boundaries



## Constructing readable sourcecode

- Do not use lines longer than the screen width (80/100/... characters)
- Layout comments with their corresponding code
- Use paragraphs
  - White space is important!



## Use assertions

- Document pre- and post-conditions using comments, especially if they are not obvious
- Check for pre- and post-conditions using the Assert keyword

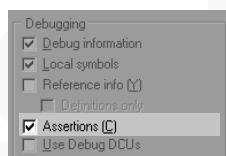
```
Assert(Count > 10,
      'not enough records');
```



## Use Assertions

- **procedure** Assert
 

```
(
    expr : Boolean
    [; const msg: string]
);
```
- Raises
  - EAssertionFailed
- Project Options | Compiler



## Exception handling

- Try...Finally
- Try...Except
- Global Exception Handler
  - in message loop of TApplication
- TApplicationHandler



## Implementation, Interface

- Declarations in interface can be seen outside that unit
- The interface section cannot contain statements (executable code)
- Variables declared in the interface section are globals: don't use them



## Classes and access-modifiers

- private
  - available only in that class and that unit
- protected
  - private + available in subclasses
- public
  - available to anyone who USES that unit
- published
  - Runtime Type Information (RTTI) is generated. The Object Inspector needs RTTI to show properties and events.
- strict private / strict protected
  - Since Delphi 2005: Like private / protected, but excludes the unit



## Interfaces and Classes

- Classes implementing interfaces is another form of Polymorphism
  - Some call it the “true polymorphism”...
- Be aware of reference counting
  - <http://wiert.wordpress.com/2009/08/20/delphi-tinterfaceddatamodule-revisited-inherited-in-your-dfm-files-when-your-datamodules-look-like-forms-in-the-designer/>
- Some notes
  - The .NET framework relies heavily on Interfaces
  - The VCL framework not, because of backward compatibility
  - But you are free to do your own!



## Properties

- Learn to create properties, also in forms and datamodules
- Use Set- and Get-methods where appropriate
- Ctrl-Shift-C can be a great help
- Using ModelMaker Code Explorer helps even more
  - [www.modelmakertools.com/code-explorer](http://www.modelmakertools.com/code-explorer)



## Events

- An event is a property of datatype ‘method-pointer’ type
  - TNotifyEvent = **procedure** (Sender: TObject) of object;
    - It contains two pointers
      - To an object instance
      - To a method
- You can assign events at runtime
  - MyForm.Button1.Click := MyForm.Button3Click;
- Events are a terrific way to **decouple** units (two-way communication with a one-way USES clause)
- Delphi unfortunately has a single-cast event handling system



## Datamodules

- Datamodules is not about data
  - They are about “non-visual”
- Use datamodules for all situations, where you want to
  - easily use Delphi components,
  - adapt properties in design-time,
  - create events in design-time
  - do not need user-interface



## TActionList / TActionManager

- Use TActionLists to separate application code from the User Interface
- TActionLists allow you to bring events from the form to the datamodule
- And they centralize some UI-properties (is this good?)
- They have some quirks, so make sure you try before you use them

## TFrame and Visual Form Inheritance

- Visual Form Inheritance and TFrames and are not always good for maintenance:
  - you can (too) easily change 'descendants'
- It is better to create your own "SuperComponents", for instance based on TFrame

## Build your own components

- "Component Templates" are good for copy-paste, don't use them as a replacement for components

## Build your own components

- Creating your own components can
  - Dramatically speed up development
  - Make your applications behave and feel much more consistent
  - Reduce maintenance, because of one central code-location

## Code generation

- Makes maintenance more difficult most of the time, because of code-duplication
- This can be made better when the generator
  - supports round-trip engineering (two-way)
  - generates classes in a hierarchy
    - TBaseXxxx (generated each time)
    - TXxx (generated once)

## Good programming techniques

that work for any language or platform



## Think

- Study at SEL 1987 found that extensive computer use (edit, compile, link, test) is correlated with low productivity.
- Of course our tools have changed a lot.
- But still:  
use your brain, not just your compiler !



## Think

Write your code for the reader,  
not for yourself.



## Code defensively

- Always assume that you or somebody else is going to make mistakes
  - Keep the code simple
  - Anticipate problems
  - Use assertions liberally
  - Document difficult code



## Coding: methods



## 1. Design the method

- Define
  - Purpose
  - Name
  - How to test it
  - Research re-use
- Write PDL (Program Design Language)
- Think about the data
- Check the PDL
- Iterate



## 2. Code the method

- Code
  - Construct the declaration (that's the interface of the routine)
  - Turn PDL into comments
  - Fill in the code below each comment
  - Check the code informally
  - Clean up the leftovers
  - Iterate



Microsoft .NET

### 3. Check the method

- Check
  - Mentally check the code for errors
  - Compile the code
  - Use the debugger to step through the code
  - Remove errors
  - Iterate

better office

better office

### Coding: Data

Microsoft .NET

Microsoft .NET

### Creating data

- Use
  - User defined types
  - Classes
  - TClientDatasets

better office

Microsoft .NET

### Creating data

- Use a template to declare variables

```
var  
    Index: Integer; // meaning of Index: better rename Index
```

- Document the meaning of variables
- Use naming conventions
  - F\* for private variables
- Remove unused declarations

better office

Microsoft .NET

### Creating data

- Initialize each variable close to where it is used (not at the top of the method)

Keep related actions together  
**Principle of proximity**

better office

Microsoft .NET

### Creating data

- Use the initialization-section to initialize variables with a global scope and lifetime.

better office

## Data names

- A Variable = Its Name
- Use good names
  - Readable
  - Memorable
  - Appropriate

A variable should fully and accurately describe the entity that the variable represents.



## Data names

- Use obvious words
  - 'LinesPerPage' is much better than 'LPP'
- Express the what, not the how
  - 'PrinterIsReady' instead of 'BitFlag'



## Data names

- Length
  - Optimum: 10-16 characters
  - Almost as good: 8-20 characters
- Use i, j or x only in a limited, local scope
- But as soon as loops are nested, use longer names



## Data names

- Watch out with 'temporary' variables
  - Temporary may indicate that you aren't sure of their real purposes
- Same with names like:
  - Util/Utils
  - Tool/Tools
  - Misc
  - Other



## Data names

- Typical boolean names
  - Done
  - Error
  - Found
  - Success
- Boolean variables names should
  - imply True / False (not: 'Status') and
  - they should be positive (not: 'NotFound').



## Data names

- Enumerated Types and Sets
  - Use prefixes (or suffixes) to group values
- Often better than booleans especially when used as parameters

```
procedure Paint(Value: string; Background: Boolean = True)
```

```
Paint('Programming is fun!', False);
```

– Versus

```
type
  TPaintBehaviour = (bhForegroundOnly, bhBackgroundAndForeground)
```

```
procedure Paint(Value: string;
  Behaviour: TPaintBehaviour = bhBackgroundAndForeground)
```

```
Paint('Programming is fun!', bhForegroundOnly);
```



## Data names

- Constants
  - Variables Don't; Constants Aren't
- Don't use capitals and underscores
- Name them like data
- [http://en.wikipedia.org/wiki/Constant\\_\(programming\)](http://en.wikipedia.org/wiki/Constant_(programming))



## Data names: abbreviations

- In general:
  - Don't
- If you have to:
  - Do not remove just one character from a word
  - Abbreviate consistently
  - Create names that you can pronounce
  - Avoid combinations that result in mispronunciation
  - Use a thesaurus
  - Document short names with translation tables



## Data names: abbreviations

- Avoid
  - misleading names or abbreviations
  - names with similar meanings
  - variables with different meanings but similar names
  - Names that sound similar or can be pronounced obscene
    - Uranus
  - Numerals in names
    - ForWeddingsAndAFuneral



## Data names: abbreviations

- Avoid
  - Misspelled words in names
  - Words that are commonly misspelled
  - Names of standard routines and variables
  - Names that are totally unrelated to what they represent
  - Names that contain hard-to-read characters ('confusion' vs. 'c0nfusion')



## Naming Conventions

- Global decisions
- Transfer knowledge
- Reduce name proliferation
- <http://stackoverflow.com/questions/262892/what-delphi-coding-standards-documents-do-you-follow>
  - Embarcadero/CodeGear/Borland standard
  - JCL/JVCL standard
  - Econos standard



## Naming Conventions

- Identify type definitions ('T', 'E')
- Identify fields ('F')
- Format names to enhance readability (not "GYMNASTICSPOINTTOTAL")
- Optionally:
  - Identify global and module variables ('g\_' and 'm\_')
- Better:
  - Do not use globals at all:
  - Use class variables in stead.



## Naming Conventions

- Some use Hungarian
  - Base types  
(wn [Window], ch [character]...)
  - Prefixes  
(a [array], c [count]...)
  - Standard Qualifiers  
(Min, First, Last...)

## Variables: general issues

- Scope
  - Minimize scope
  - Keep references to a variable together
- Binding
  - Use late binding for the contents of variables
    - Code-time      A := 47;
    - Compile-time    A := NumberOfBars;
    - Run-time        A := GetNumOfBarsFromIniFile ();

## Variables: general issues

- Use
  - Use each variable for exactly one purpose
    - One purpose only
    - Avoid hidden meanings ( 1..100, -1 )
    - Use all declared variables

## Variables: general issues

- Problems with globals
  - Inadvertent changes to global data
  - Aliasing problems with global data
  - Thread conflicts
  - Code reuse hindered by global data
  - Modularity and intellectual manageability damaged by global data

## Variables: general issues

- Instead of globals
  - Create access routines
  - Create a global variable that contains an object with properties  
( Application : TApplication in Forms.pas )
  - Use class variables

## Fundamental Data Types

- Avoid 'magic numbers' and 'magic strings', use constants instead
- The use of 0 (zero) and 1 (one) is allowed
- Make type-conversions obvious
  - Use a type-cast or
  - Use a conversion
- Avoid mixed-type comparisons and floating point comparisons  
if (Pi = 3) then // bad for 2 reasons
- Heed your compiler's warnings
  - The compiler is almost always right

## Coding: Control statements



## Using conditionals

```
if ( x ) then
begin
  //
end
else
begin
  //
end;
```



## Using conditionals

- Let the nominal path through the code not be obscured by the exceptions
- Put the normal case after the if



## Using conditionals

- Follow the if-clause with a meaningful statement
- Use >= etc. correctly: think through the endpoints

```
if not (a >= 10)
if (a < 10)
```
- Consider the else clause and maybe even create one to document that the else clause was considered:

```
if (FailureCondition) then
  HandleFailure()
else
  ; // no need to handle success
```
- Check for the flip-flop error



## Controlling loops

- Select the right kind of loop
  - FOR...NEXT
  - FOR...IN
  - WHILE...DO      test at begin of loop
  - REPEAT...UNTIL      test at end of loop;  
                         runs at least once



## Controlling loops

- Initialize variables directly before the loop
- Do variable housekeeping for the loop at the beginning or end of the loop
- Make each loop perform only one function
- Assure yourself that the loop ends
- Make termination conditions obvious
- Don't monkey with the loop index (i)
- The loop index is undefined after exiting the loop!



## Controlling loops

- Avoid
  - CONTINUE
  - BREAK
  - EXIT
- Use if-statements instead



## Controlling loops

- Make your loops short enough to view all code at once
- Limit nesting to three levels
- Make long loops especially clear



## Unusual Control structures

- Goto
- Recursion
  - Make sure it stops (use safety counter during debug/test)
  - Limit to one routine (not:  $A > B > C$ )
  - Keep an eye on the stack



## General Control structures

- Booleans
  - Make complicated tests simple: break them in parts or use functions
  - Form boolean expressions positively
    - if not Failure
    - if not NoSuccess
    - if Success
  - Use parentheses



## General Control structures

- Compound statements (begin/end blocks)
  - Use them instead of single statements
  - Do not nest too deeply
    - Retest some of the conditions
    - Use if...then...else
    - Use a CASE statement
    - Construct new methods
    - Redesign deeply nested code

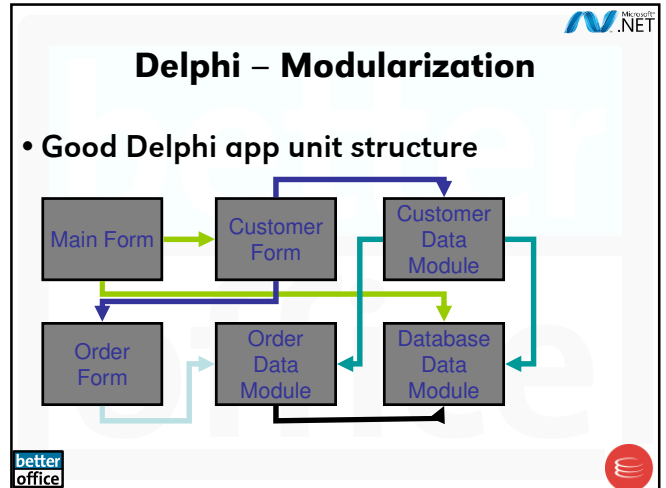
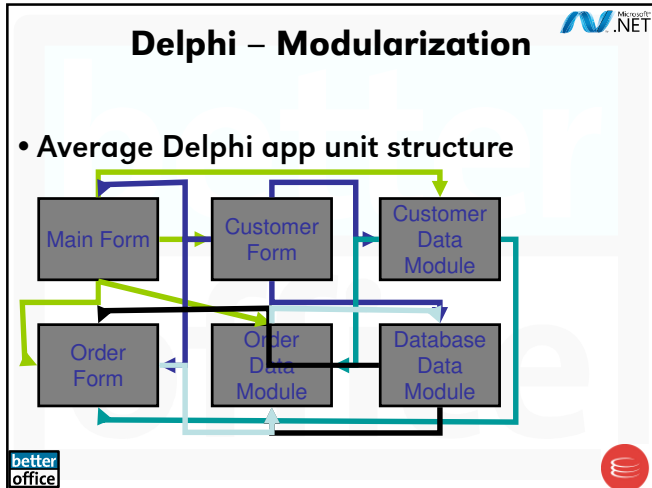


## Modularizing data layers

Cohesion, coupling, interfaces, indirection, simplify

All in one example





## How come good is better than bad?

- Look for modularization in real life...
  - Houses                      - rooms
  - Cities                        - suburbs/blocks
  - People                       - digestion system
  - Parliament                - parties
- Sometimes modularization works
- Sometimes it doesn't ☺

better office

## Modules are everywhere...

- So why does it work?
  - Internal Cohesion **HIGH**
  - External Coupling **LOW**
    - If also 'uniform': great!
    - If also 'directional': even better!

The illustration shows a train with four cars, representing modularization. The train is moving from left to right, and the cars are connected in a sequence, illustrating the concept of directional coupling.

better office

## Modularization – database apps

- Let's apply this knowledge to our database apps
- Especially: where to put the DataSource...
- Where is your DataSource?

better office

## Modularization – DataSource

The diagram compares two database app designs. The 'Bad' design shows a form with multiple data sources (CustomerQuery, CustomerDataSource, OrderQuery, OrderDataSource) and a data module (CustomerDataModule) with many internal links (3) and external links (6). The 'Good' design shows a form with a single data source (CustomerDataSource) and a data module (CustomerDataModule) with fewer internal links (6+2) and external links (2). This illustrates the importance of minimizing external links and maximizing internal cohesion.

**Bad** 3 Internal Links  
6 External Links

**Good** 6+2 Internal  
2 External

better office



## Modularization – the datasource

- **DataSource** has **two** goals

- Binding GUI controls
- Providing Master-Detail relations

– GUI binding:

put **DataSource** on **Form**

– MD-relations:

put **DataSource** on **DataModule**



## Modularization – gain

- **Flexibility**

- Change of GUI
- Change of Data Access
- Re-use of modules across projects



## Know your VCL

[Delphi 5 Object Hierarchy.pdf](#)  
[Delphi 7 - VCLHierarchyPoster.pdf](#)



## Know new features

for ... in  
 Generics  
 Anonymous methods



## for ... in

- for Win32, it was introduced  
 in D2005

– On lots of built-in data types

- Arrays
- Strings

– On many data types

- Lists, Collections, Trees,
- Components, Actions, Menus, Fields,
- many, many more ...



## Data types supporting for ... in

– These data types

- provide either of these:

- **function** GetEnumerator: T...Enumerator;
- **function** GetEnumerator: IEnumerator;

- and the result provides these functions:

- **constructor** Create(  
     **const** AObject: T...Object);
- **function** GetCurrent: T...;
- **function** MoveNext: Boolean;
- **property** Current: T... **read** GetCurrent;

– The compiler then recognizes it supports  
 for ... in



## Business logic versus glue...

```
procedure TXokumDataModule.GetMinMaxAbonneeNumberOldStyle(
var MinAbonneeNumber: Integer;
var MaxAbonneeNumber: Integer);
var
wasActive: Boolean;
begin
MinAbonneeNumber := High(Integer);
MaxAbonneeNumber := Low(Integer);
wasActive := XokumClientDataSet.Active;
XokumClientDataSet.Open;
XokumClientDataSet.First;
while not XokumClientDataSet.Eof do
begin
if XokumClientDataSet.abonneenummer.Value > MaxAbonneeNumber then
MaxAbonneeNumber := XokumClientDataSet.abonneenummer.Value;
if XokumClientDataSet.abonneenummer.Value < MinAbonneeNumber then
MinAbonneeNumber := XokumClientDataSet.abonneenummer.Value;
XokumClientDataSet.Next;
end;
if not wasActive then
XokumClientDataSet.Close;
end;
```

better  
office

## Wouldn't it be nice to...

```
procedure TXokumDataModule.GetMinMaxAbonneeNumber(
var MinAbonneeNumber: Integer;
var MaxAbonneeNumber: Integer);
var
Index: TDataSetEnumerationRecord;
begin
MinAbonneeNumber := High(Integer);
MaxAbonneeNumber := Low(Integer);
for Index in XokumClientDataSet do
begin
if XokumClientDataSet.abonneenummer.Value > MaxAbonneeNumber then
MaxAbonneeNumber := XokumClientDataSet.abonneenummer.Value;
if XokumClientDataSet.abonneenummer.Value < MinAbonneeNumber then
MinAbonneeNumber := XokumClientDataSet.abonneenummer.Value;
end;
end;
```

better  
office

## Helpers

- Introduced in Delphi to support .NET
  - The .NET class hierarchy differs from Win32 VCL
  - In the .NET framework, VCL methods and properties were different or missing
- Helpers can make extensions at function level
  - Yes: methods and properties
  - No: instance data
- They also work in Delphi for Win32:
  - Class helpers since Delphi 2005
  - Record helpers since Delphi 2006

better  
office

## Helpers

- Helpers (class or record):
  - function as long as the helper is visible to the user
- So:
  - Helper in the same unit,
  - or helper in a unit in the uses list

better  
office

## Generics

- Great document:  
<ftp://ftp-developpez.com/sjrd/tutoriels/delphi-generics/delphi-generics.pdf>
- Basically:


```
type
TList<T> = class(...)
var
MyIntegerList = TList<Integer>;
MyStringList = TList<string>;
```
- Lots of examples:
  - see demo later on

better  
office

## Anonymous methods

- Inline block of code
- Like a regular method
- With parameters
- Optional function result
- But without a name
- <http://stackoverflow.com/questions/256146/can-someone-explain-anonymous-methods-to-me>
- <http://blogs.embarcadero.com/abauer/2008/09/25/38870>

better  
office





## Anonymous example


```

class procedure TScreenSupport.Executewithwaitcursor(const Proc: TProc);
var
  OldCursor: TCursor;
begin
  OldCursor := Screen.Cursor;
  try
    Screen.Cursor := crHourGlass;
    Proc();
  finally
    Screen.Cursor := OldCursor;
  end;
end;

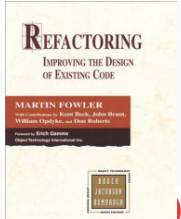
procedure TMyForm.Button1Click(Sender: TObject);
begin
  TScreenSupport.Executewithwaitcursor(
    procedure
    begin
      TExcelSupport.Export(OverzichtEnsembleStringGrid);
    end
  );
end;



```


 



## Code Refactoring






 



## Testing



<http://stackoverflow.com/questions/540617/best-way-to-test-a-delphi-application>


 



## Testing



- Makes you find bugs faster
- Helps “Fix the bug when it’s found”
- Makes your total process cheaper
  - Prepare to move some budget from “maintenance” to “development” cycle


 



## Source testing



- By humans only:
  - Pair programming (immediate feedback)
  - Peer review (feedback at a later time, for instance right before checkin)



## Continuous integration

- Build all your targets
  - By hand once every while
  - On each checkin, automatically
- Helps find the “colleague A modifies X, but it now breaks at colleague B in application Y” sooner
- Tools:
  - Makefiles
  - FinalBuilder
  - CruiseControl

## Unit testing

- Comes from eXtreme Programming and Agile Programming
- Tests methods of classes
- Built-in Delphi since Delphi 2005



## UI testing

- Use TestComplete from AutomatedQA
- Good, not cheap (USD 999-1999)



## General guidelines

- Develop more, smaller routines
- Reduce the number of globals
- Improve your programming style
- Manage changes
- Review code changes
- Retest



## Constructing new methods

- Reduce complexity
  - by shortening
  - by reducing nesting
- Share code



## Code Tuning Issues

- Delivering on-time, on-budget
- Providing a clean user-interface
- Avoiding downtime
- Constructing maintainable code

...may be more important to the user than raw speed



## Old Wives' Tales

- Reducing the lines of code improves the size or speed of the executable
  - false
- Certain operations are probably faster or smaller than others
  - false
- You should optimize as you go
  - false

"You can get 80 percent of the result with 20 percent of the effort"

The Pareto Principle



Microsoft.NET

## Strategies

- Use a profiler
  - <http://stackoverflow.com/questions/368938/delphi-profiling-tools>
  - AQTime
  - ProDelphi
  - SamplingProfiler
- Use the compiler optimizations
- Use iteration

better office

Microsoft.NET

## Where's the fat ?

- Database / Network traffic
- Other Input/output operations
- User interaction
- Speed in decreasing order
  - Network
  - USB
  - Hard drive
  - Memory

All programming is an exercise in caching  
Terje Mathisen

better office

Microsoft.NET

## Techniques

1. Use a good modular design
2. Measure the system, if performance is poor
3. Determine the reasons for slow speed
4. Tune the bottleneck, if there is one
5. Iterate

better office

better office

## Wrap-it-up

Microsoft.NET

Microsoft.NET

## Share information

- Make the targets visible to everyone on the team
- Store documentation in one central place
- Always (!) document errors that you found and the resolution to solve it
- Log bugs in one central place

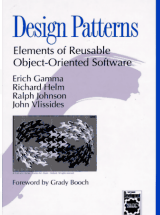
better office

Microsoft.NET

## Short Overview of Design Patterns

- Design Patterns is about reusing approaches to problem-solving
- A Design Pattern consists of
  - The pattern name
  - The problem
  - The solution
  - The consequences

better office



## Ten essentials for building good software

1. A product specification
2. A detailed user interface prototype
3. A realistic schedule
4. Explicit priorities
5. Active risk management
6. A quality assurance plan
7. Detailed activity lists
8. Software configuration management
9. Software architecture
10. An integration plan

from Steve McConnell

## 12 Classic Mistakes

1. Undermined motivation
2. Uncontrolled problem employees
3. Noisy, crowded offices
4. Abandoning planning under pressure
5. Shortchanging upstream activities
6. Shortchanging quality assurance
7. Lack of feature-creep control
8. Silver-bullet syndrome
9. Wasting time in the 'fuzzy front end'
10. Insufficient user input
11. Overly aggressive schedules
12. Adding developers to a late project

from Steve McConnell

## Sample projects

Demo time...

The good, the bad and the ugly

## Tests

Belbin (xls) – the human factor

Joel – the team factor

## Discussion time

## Q & A

Jeroen Pluimers  
better office benelux  
[jpluimers@better-office.com](mailto:jpluimers@better-office.com)

If you have questions after the workshop, please mail me

my blog: <http://wiert.wordpress.com>