Enbugging

Wie wir Fehler machen und wie wir sie vermeiden

Christian Rehn aka R2C2

Delphi-Treff

Delphi-Tage 2011

Vorstellung

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

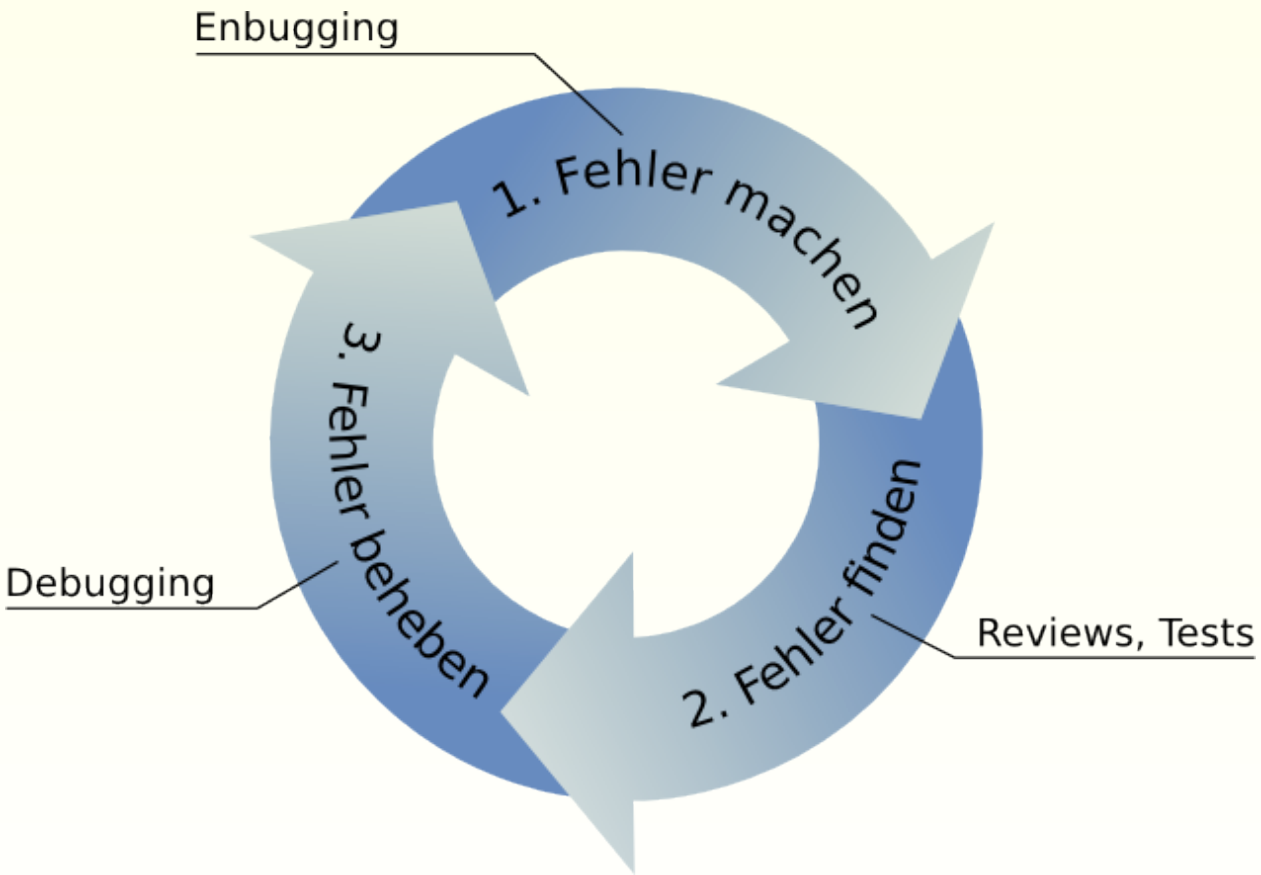
Überblick

Einführung und Motivation

Debugging und Enbugging

„Wenn Debugging der Vorgang ist, Fehler aus einem Programm zu entfernen, dann ist Programmierung der Vorgang, Fehler in ein Programm einzubauen.“

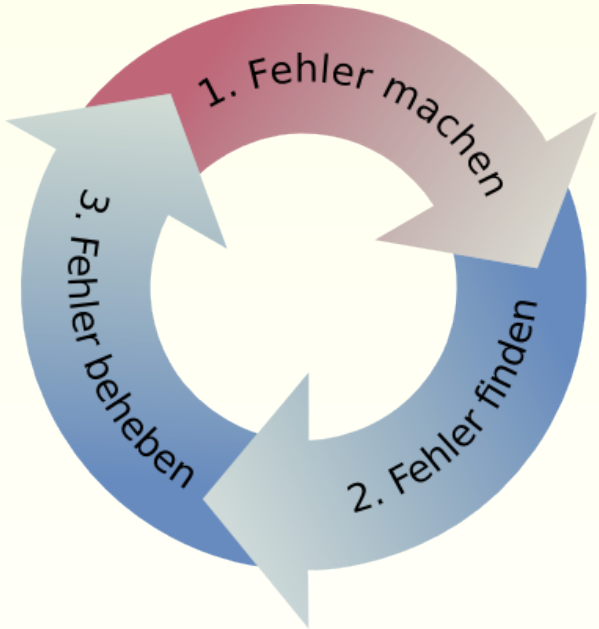
Was wir tun


Enbugging
1. Fehler machen
2. Fehler finden
Reviews, Tests
3. Fehler beheben
Debugging

Ist das gut?

- Debugging kostet Zeit
- Debugging kostet Geld
- Debugging kann nervig sein
- Aber wir sind selbst schuld an den Bugs

$\Rightarrow$ Besser Bugs vermeiden als Bugs fixen



Bugschleudern

Aus der WinAPI: GetEnvironmentVariable[1]

If the function succeeds, the return value is the number of characters stored in the buffer [...], **not including the terminating null character**.

If lpBuffer is not large enough to hold the data, the return value is the buffer size, in characters, required to hold the string **and its terminating null character** [...].

- Auch wir produzieren solche Bugschleudern – nur nicht immer so offensichtliche
- Um diese subtileren Bugschleudern geht es hier

[1] `http://msdn.microsoft.com/en-us/library/ms683188`

Was wir tun können

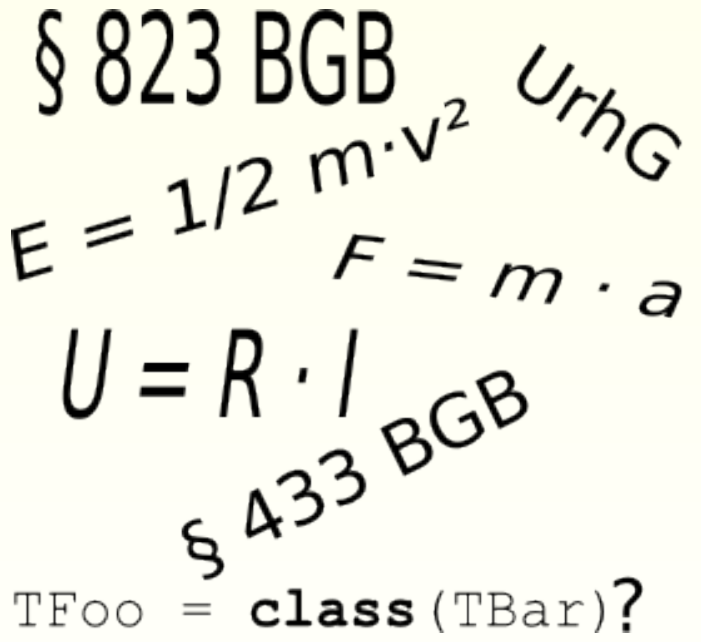
Drei Schritte

1. Die Gesetze des Codes kennen
   - Wie funktioniert Code?
   - Welche Probleme und Fallstricke existieren?
   - Welche Fehler kann man machen?
2. Die eigenen Fehler kennen
   - Welche Fehler mache ich persönlich?
3. „Bug-feindlich“ programmieren
   - Was kann ich tun, um das zu ändern?

Die Gesetze des Codes

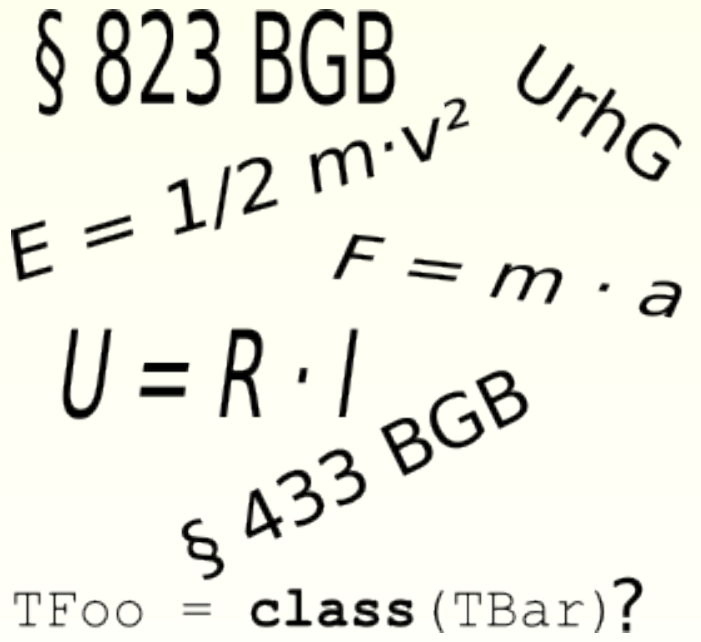
Gesetze im Code

- Herrschen im Code Gesetze?
  - Ja, aber keine physikalischen
  - Kognitive Gesetze
- Nicht nur im Code
  - In Architektur und Design
  - In Tests und Reviews
  - In Team und Organisation
  - ...



Gesetze im Code

- Herrschen im Code Gesetze?
  - Ja, aber keine physikalischen
  - Kognitive Gesetze
- Nicht nur im Code
  - In Architektur und Design
  - In Tests und Reviews
  - In Team und Organisation
  - ...



Physikalische und kognitive Gesetze

Physikalische Gesetze

Wenn man sie missachtet, raucht die Platine ab.

Kognitive Gesetze

Wenn man sie missachtet, funktioniert trotzdem alles. – Und ein halbes Jahr später kriegt man ernsthafte Probleme.

Physikalische und kognitive Gesetze

Physikalische Gesetze

Wenn man sie missachtet, raucht die Platine ab.

Kognitive Gesetze

Wenn man sie missachtet, funktioniert trotzdem alles. – Und ein halbes Jahr später kriegt man ernsthafte Probleme.

Ripple Effects

Ripple Effects

Änderungen an einer Stelle machen Änderungen an weiteren Stellen nötig, die wiederum weitere Änderungen nötig machen.

Analogie: Ein Wassertropfen erzeugt sich allmählich ausbreitende kleine Wellen (engl. *ripples*).


[2]

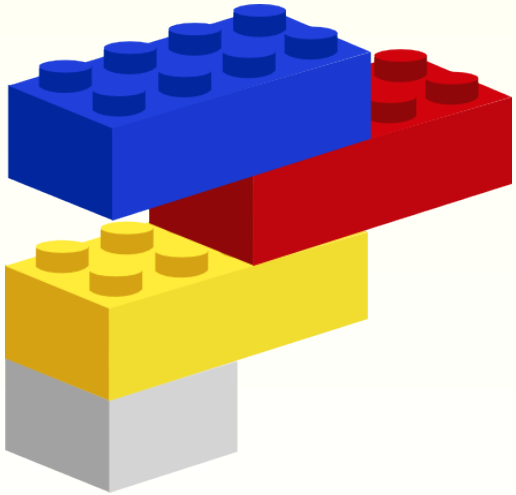
[2] CC-BY-SA by Rainer Zenz
http://commons.wikimedia.org/wiki/File:2006-01-14_Surface_waves-2.jpg

Kopplung

Die Ursache von Ripple Effects: Zu starke Kopplung

Kopplung

Kopplung ist ein Maß für die Abhängigkeit zwischen Modulen. Sie definiert sich aus den Annahmen, die ein Modul A über ein Modul B trifft.

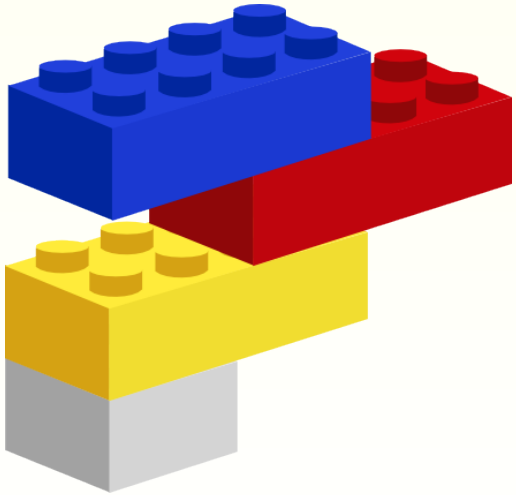


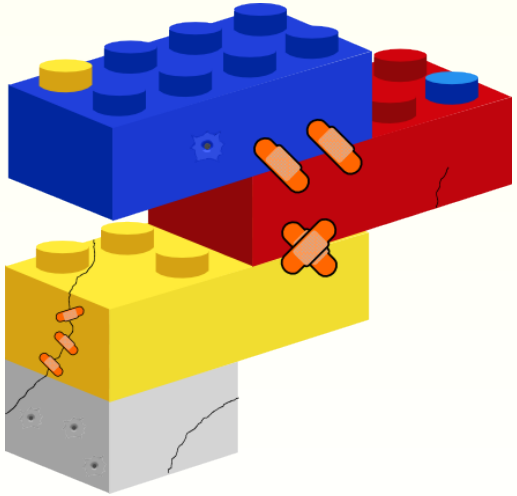
Kopplung

Die Ursache von Ripple Effects: Zu starke Kopplung

Kopplung

Kopplung ist ein Maß für die Abhängigkeit zwischen Modulen. Sie definiert sich aus den Annahmen, die ein Modul A über ein Modul B trifft.





Kopplungsarten

Kopplungsarten (von stark bis schwach)

1. Content coupling (Inhaltskopplung)
2. Common coupling (Bereichskopplung)
3. External coupling (Externdatenkopplung)
4. Control coupling (Kontrollkopplung)
5. Stamp coupling (Datenstrukturkopplung)
6. Data coupling (Datenkopplung)
7. Call coupling (Aufrufkopplung)

Kopplungsarten

Kopplungsarten (von stark bis schwach)

1. Content coupling (Inhaltskopplung)
2. Common coupling (Bereichskopplung)
3. External coupling (Externdatenkopplung)
4. Control coupling (Kontrollkopplung)
5. Stamp coupling (Datenstrukturkopplung)
6. Data coupling (Datenkopplung)
7. Call coupling (Aufrufkopplung)

Content Coupling (Inhaltskopplung)

Beispiel:

```
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;
```

- Auch: „Pathological coupling“ (pathologische Kopplung)
- Abhängigkeit von konkreter Implementierung
- Besser: Propertys verwenden (siehe `TOpenDialog`)

Control Coupling (Kontrollkopplung)

Beispiel:

```
procedure TMyFeedReader.DeleteItem(item: TFeedItem; recycle: Boolean);
begin
  if recycle then
    // throw in dust bin
  else
    // delete completely
end;
```

- Parameter ist Kontrollflussinformation
- Besser: zwei getrennte Methoden
- „Gute“ Variante der Kontrollkopplung: Rückgabewert ist Kontrollflussinformation

Kopplungsarten – Sonderformen

Sonderformen

- Tramp coupling
- Hybrid coupling (Hybridkopplung)
- Temporal coupling (zeitliche Kopplung)
- Logical coupling (logische Kopplung)
- uses-Kopplung

Kopplungsarten – Sonderformen

Sonderformen

- Tramp coupling
- Hybrid coupling (Hybridkopplung)
- Temporal coupling (zeitliche Kopplung)
- Logical coupling (logische Kopplung)
- uses-Kopplung

Tramp Couplung

Beispiel:

```
constructor TMyHtmlParser.Create(const url: string);
begin
  ...
  input := idHTTP.Get(url);
  ...
end;
```

- Ein Parameter wird einfach „durchgereicht“
  $\Rightarrow$ Unnötige Kopplung durch Kenntnis einer Information, die gar nicht benötigt wird
- Hier besser: String oder Stream übergeben und Download auslagern

Logische Kopplung

Beispiel:

```
// GUI Layer
CardImage.LoadFromFile(Path + card.ToString + '.bmp'); // e.g. C:\Some\Path\HerzAss.bmp

// Application Logic Layer
function TPlayingCard.ToString: string;
begin
  Result := SuitToStr(suit) + ValueToStr(value); // e.g. 'Herz' + 'Ass'
end;
```

- Nicht technische, sondern logische Annahmen
- Hier trifft `TPlayingCard` Annahmen über die Visualisierung

Vorbedingungen und Nachbedingungen

Beispiel:

```
procedure TMyList.Add(myItem: TMyItem);
begin
  {myItem <> nil} // Vorbedingung
  ...
  {Self.Contains(myItem) and (FCount = old(FCount) +1)} // Nachbedingung
end;
```

- Vor- und Nachbedingungen beschreiben, was eine Methode tut
- Nachbedingungen beschreiben auch Seiteneffekte
- Alle Methoden haben Vor- und Nachbedingungen
  - Auch, wenn wir sie nicht spezifizieren

Invarianten

Beispiel:

```
// wieder Beispiele im Kontext einer Liste
FCount >= 0
FCount <= FCapacity
FInnerList <> nil
forall i in 1..FCount-1 : FInnerList[i-1] <= FInnerList[i]  // Liste ist immer sortiert
```

- Invarianten sind Bedingungen die immer erfüllt sein müssen
- Arten: Kontrollfluss-, Schleifen-, interne Invarianten
- Klassen-Invarianten beziehen sich auf die Integrität eines Objekts
- Unser Code ist voll von Invarianten
  - Wir denken nur nicht darüber nach
  - Gefahr: Invarianten werden unabsichtlich verletzt

In der Praxis

- Spezifikation von Vorbedingungen, Nachbedingungen und Klassen-Invarianten nicht immer einfach
- Dennoch manchmal sinnvoll
- Auf jeden Fall sollten wir uns aber bewusst sein, dass sie existieren; ob wir sie aufschreiben oder nicht

Code und Semantik (1/2)

- Es reicht nicht, dass Code das tut, was er soll
- Nicht nur das „Was“, sondern auch das „Warum“ ist entscheidend

$\Rightarrow$ Kontext, Semantik

Beispiel: Auto mit Anfahrassistent und Klimaanlage

- Anfahrassistent löst beim Anfahren automatisch die Handbremse
- Klimaanlage wird bei niedriger Drehzahl und heißem Wetter virtuell aufs Gas-Pedal treten

$\Rightarrow$ ...

Code und Semantik (2/2)

Beispiel: Auto mit Anfahrassistent und Klimaanlage

- Situation: Heißes Wetter, Auto will in die Tiefgarage
- Tür auf, um an Ticket-Automaten zu kommen
- Resultat
  - Warme Luft strömt ins Auto
  - Klimaanlage drückt aufs Gas
  - Anfahrassistent löst die Handbremse

$\Rightarrow$ Unfall

Fehler: Klimaanlage wollte eigentlich die Motorleistung erhöhen und nicht aufs Gas treten

Semantik von OnClick-Handlern

Nicht gut:

```
ApplyButton.Click;
// oder
ApplyButtonClick( nil );
```

Besser:

```
SaveSettings; // das wird auch im OnClick-Handler aufgerufen
```

- Nicht das aufrufen, was zufälligerweise der Button tut
- Sondern das tun, was man tun will

Eigene Fehler

Die drei Feinde des Programmierers

- Frische Luft, Tageslicht und dieses unerträgliche Gebrüll der Vögel ;-)
- Murphy, der Kollege und man selbst
- Man selbst, man selbst und man selbst

Murphy, der Kollege und man selbst (1/2)

Murphys Gesetz (so wie zitiert)

„Whatever can go wrong, will go wrong.“

Murphys Gesetz (original)

„If there's more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way.“

Murphy, der Kollege und man selbst (2/2)

- Murphy ist eh immer an allem schuld
  - Gesetz bedenken, nicht als Ausrede benutzen
- Der Kollege
  - Angenommen in einem halben Jahr wird irgend ein Halbidiot an unseren Code gesetzt
  - Wenn selbst dieser Halbidiot nicht mehr Bugs einbaut als fixt, ist der Code gut
- Man selbst
  - Typischerweise ist man selbst dieser Halbidiot

Murphy, der Kollege und man selbst (2/2)

- Murphy ist eh immer an allem schuld
  - Gesetz bedenken, nicht als Ausrede benutzen
- Der Kollege
  - Angenommen in einem halben Jahr wird irgend ein Halbidiot an unseren Code gesetzt
  - Wenn selbst dieser Halbidiot nicht mehr Bugs einbaut als fixt, ist der Code gut
- Man selbst
  - Typischerweise ist man selbst dieser Halbidiot

Murphy, der Kollege und man selbst (2/2)

- Murphy ist eh immer an allem schuld
  - Gesetz bedenken, nicht als Ausrede benutzen
- Der Kollege
  - Angenommen in einem halben Jahr wird irgend ein Halbidiot an unseren Code gesetzt
  - Wenn selbst dieser Halbidiot nicht mehr Bugs einbaut als fixt, ist der Code gut
- Man selbst
  - Typischerweise ist man selbst dieser Halbidiot

Man selbst, man selbst und man selbst

- Unachtsamkeit
- Unverständnis der Problemdomäne
- Unverständnis der Technik
  - Cargo-cult programming
- Mangelndes Erinnerungsvermögen
- Faulheit
  - Sonderfälle, Randfälle
  - „Kann nicht passieren!“
    - Aber Murphy...
- ...

Eigene Fehler

- Jeder macht andere Fehler
- Nur wenn man das Problem kennt, kann man es abstellen
- Wer kennt seine eigenen Fehler?

> **Tom DeMarco**
>
> „I can only think of one metric that is worth collecting now and forever: defect count. Any organization that fails to track and type defects is running at less than its optimal level.“

Eigene Fehler

- Jeder macht andere Fehler
- Nur wenn man das Problem kennt, kann man es abstellen
- Wer kennt seine eigenen Fehler?

Tom DeMarco

„I can only think of one metric that is worth collecting now and forever: defect count. Any organization that fails to track and type defects is running at less than its optimal level.“

Meine Fehlerliste (1/2)

Ein kleines Beispielprojekt:

- ca. 1900 Zeilen Delphi-Code
- Einfache Aufgabenstellung
- Anforderungen durch vorheriges Projekt schon bekannt

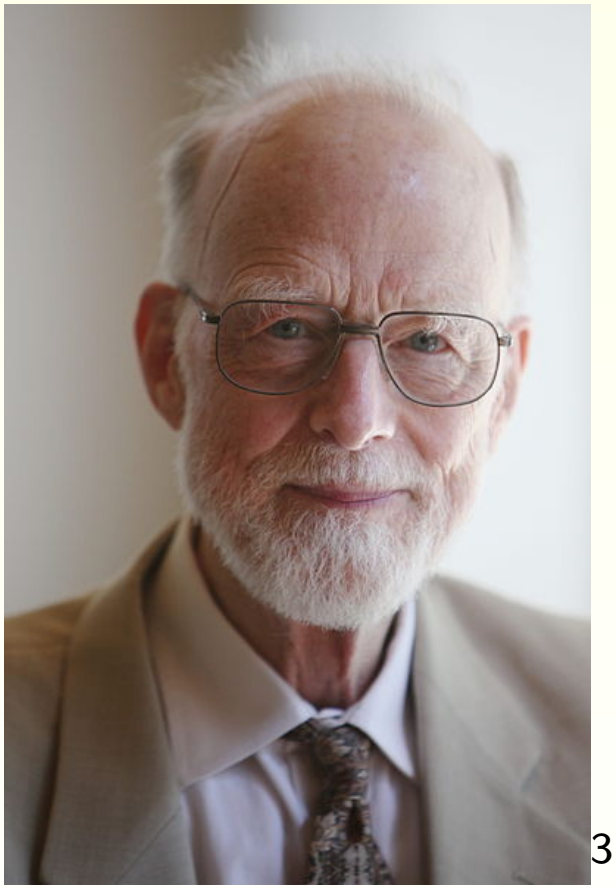
Meine Fehlerliste (2/2)

	gefunden in		
	Impl.	Test	beim Kunden
Fehler in Anforderungen und Design	keine (Anf. durch vorheriges Projekt bekannt)		
Copy'n'Paste-Fehler	–	2	–
Reihenfolge vertauscht	–	2	–
Tippfehler	1	–	–
nil setzen vergessen	3	–	–
zu frühes Freigeben	–	1	–
lok. Variable nicht initialisiert	–	1	–
Fehler in Formel	1	1	–
DecimalSeparator nicht angepasst	–	1	–
Button nicht disabled	1	–	–
Wert nicht gesetzt	–	1	1
case vergessen	–	1	–
Füllen der Pipe vergessen	–	1	–
+ ';' vergessen	–	–	1

Bug-feindlich programmieren

Simplicity

„There are two ways of constructing a software design: One way is to make it so simple that there are **obviously no deficiencies**, and the other way is to make it so complicated that there are **no obvious deficiencies**.“ – Tony Hoare



[3]

[3] CC-BY-SA by Rama http://commons.wikimedia.org/wiki/File:Sir_Tony_Hoare_IMG_5123.jpg

Daumenregeln

Princliple of Least Surprise

Never surprise the user! (And never surprise the developer!)

KISS

Keep it simple, stupid!

Was heißt einfach?

einfach $\neq$ kurz $\neq$ schnell zu schreiben

„Elegant“ vs. einfach (1/3)

Aufgabe:

Zuweisung an `CheckBox.Checked` löst `OnClick` aus.
Wie setzt man `Checked` ohne, dass das Event ausgelöst wird?

„Elegant“ vs. einfach (2/3)

Gute Entwickler neigen dazu, Probleme möglichst kunstvoll und elegant zu lösen...

Die elegante Lösung

```
procedure SetCheckBoxChecked(ACheckBox: TCheckBox; AValue: Boolean);
const
  States: array[Boolean] of wParam = (BST_UNCHECKED, BST_CHECKED);
begin
  SendMessage(ACheckBox.Handle, BM_SETCHECK, States[AValue], 0);
end;
```

„Elegant“ vs. einfach (3/3)

. . . aber kunstvoll und elegant ist nicht unbedingt „einfach“.

Die einfache Lösung

```
procedure SetCheckBoxChecked(ACheckBox: TCheckBox; AValue: Boolean);
var
  tmpOnClick: TNotifyEvent;
begin
  tmpOnClick := ACheckBox.OnClick;
  try
    ACheckBox.OnClick := nil;
    ACheckBox.Checked := AValue;
   finally
    ACheckBox.OnClick := tmpOnClick;
  end;
end;
```

Design by Contract

Beispiel (Prism):

```
method MyList.Add(myItem: MyItem);
 require
   myItem <> nil;
begin
    ...
ensure
   Self.Contains(myItem);
   FCount = old FCount +1;
end;
```

- Wir betrachten Methodenaufrufe als Verträge
  - Wenn der Aufrufer seinen Teil der Abmachung erfüllt ($\rightarrow$ Vorbedingung)
  - Dann garantiert die Methode ihren Teil ($\rightarrow$ Nachbedingung)
- Ursprung: Eiffel; mittlerweile auch in Prism; Librarys für diverse Sprachen

Assertions (1/3)

```
procedure Assert(Condition: Boolean; [ Message: String ]);

{$ASSERTIONS ON}
{$ASSERTIONS OFF}
```

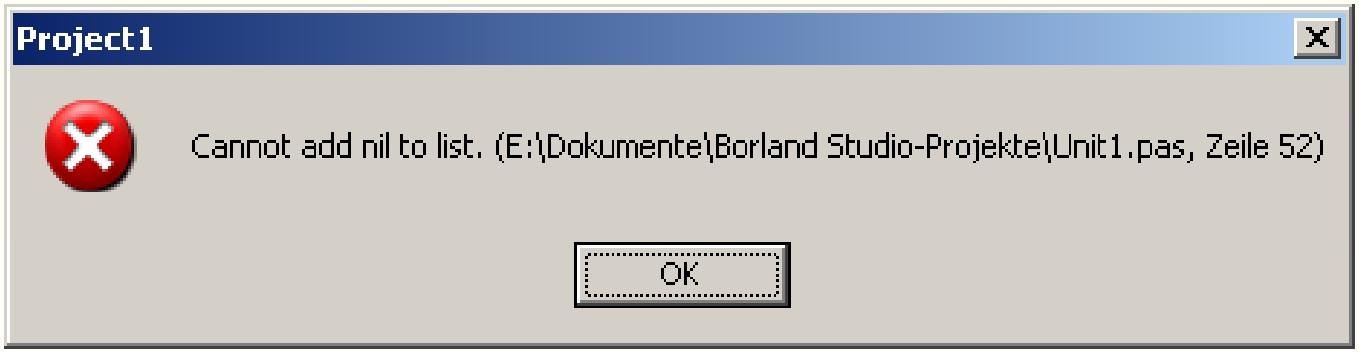
- Assertions dokumentieren, was wahr sein *muss*
  - Alles andere ist ein Bug
  - Beispiele: Vorbedingungen, Nachbedingungen, Invarianten, ...
- Wenn `Condition false` ist, wird eine `EAssertionFailed`-Exception ausgelöst
- Assertions können selektiv für jede Unit ein- oder ausgeschaltet werden

Assertions (2/3)

Achtung!

- Assertions sind kein Ersatz für normale Parameterprüfung (sondern Ergänzung)
- `Condition` darf keine Seiteneffekte haben

Assertions (3/3)



Assertions für Vor- und Nachbedingungen

Beispiel:

```
procedure TMyList.Add(myItem: TMyItem);
begin
  Assert(myItem <> nil, 'Cannot add nil to  list .');
  ...
  Assert( Self .Contains(myItem));
end;
```

- Mit Assertions kann man DBC teilweise simulieren
  - Keine alten Werte (`old Count`)
  - Keine Klassen-Invarianten
  - Keine Vererbung von Verträgen
  - Workarounds teilweise unschön

else?

Nochmal der CheckBox-Code. Eigentlich stand er so da:

```
procedure SetCheckBoxChecked(ACheckBox: TCheckBox; AValue: Boolean);
const
  States: array [Boolean] of wParam = (BST_UNCHECKED, BST_CHECKED);
begin
  if Assigned(ACheckBox) then
    SendMessage(ACheckBox.Handle, BM_SETCHECK, States[AValue], 0);
end;
```

Und `else`...?

Rule of Repair

When you must fail, fail noisily and as soon as possible.

else?

Nochmal der CheckBox-Code. Eigentlich stand er so da:

```
procedure SetCheckBoxChecked(ACheckBox: TCheckBox; AValue: Boolean);
const
  States: array[Boolean] of wParam = (BST_UNCHECKED, BST_CHECKED);
begin
  if Assigned(ACheckBox) then
    SendMessage(ACheckBox.Handle, BM_SETCHECK, States[AValue], 0);
end;
```

Und `else`...?

Rule of Repair

When you must fail, fail noisily and as soon as possible.

else? – Exception

Beispiel:

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

  ...
end;
```

- Bei öffentlichen Methoden
- Insbesondere an Schichtengrenzen, bei APIs, Frameworks, etc.

else? – Kann nicht passieren (1/2)

Beispiel:

```
procedure SetCheckBoxChecked(ACheckBox: TCheckBox; AValue: Boolean);
const
  States: array [Boolean] of wParam = (BST_UNCHECKED, BST_CHECKED);
begin
  Assert(Assigned(ACheckBox));

  SendMessage(ACheckBox.Handle, BM_SETCHECK, States[AValue], 0);
end;
```

- Vorbedingung
- Insbesondere bei privaten Methoden

else? – Kann nicht passieren (2/2)

Ein Auszug aus meiner Bachelorarbeit:

```
double tripleProduct = displayYAxis.crossProduct( planeYAxis ).dotProduct( planeNormal );
if ( tripleProduct > 0 )
{
    // right-handed system ==> rotate acording to right hand grip rule
    // nothing to do here
}
else if ( tripleProduct < 0 )
{
    // right-handed system ==> rotate against right hand grip rule
    angle = -angle;
}
else
{
    assert( false ); // can never happen ... oder etwa doch?
}
```

else? – Neutraler Rückgabewert

Beispiel:

```
function  TMyMap.GetValuebyKey(key: string): TMyValueType;
begin
   if  not  Contains(key)  then
      exit ( nil );

   ...
end;
```

- Wenn `nil` eine sinnvolle Bedeutung hat
- Oft sind leere Listen besser als `nil`
- Achtung! Bei falscher Anwendung: Hybridkopplung

else? – Idempotenz

Beispiel:

```
procedure TMyStringSet.Add(item: string);
begin
  if FInnerList.Contains(item) then
  begin
    FInnerList.Add(item);
  end
  else
  begin
    // nothing to do
  end;
end;
```

- Zwei Aufrufe haben den selben Effekt wie einer
- Leerer `else`-Teil

DRY – Don't Repeat Yourself

DRY – Don't Repeat Yourself

Don't Repeat Yourself! Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

- Keine Codedopplungen
- Keine doppelten Informationen (auch nicht in unterschiedlicher Repräsentation)
- Wenn doch, dann andere Repräsentationen aus einer generieren
  - Write code that writes code

A DRY ComboBox (1/3)

Beispiel:

```
case SomeComboBox.ItemIndex of
  0:  ...;
  1:  ...;
  ...
end;
```

- Und wenn sich die Reihenfolge ändert?

A DRY ComboBox (2/3)

Auch nicht besser:

```
if SomeComboBox.Text = 'Bla' then
  ...
else if SomeComboBox.Text = 'Blubb' then
  ...
```

- Und wenn sich die Beschriftung ändert?

A DRY ComboBox (3/3)

Objects

```
SomeTypeCast(SomeComboBox.Items.Objects[SomeComboBox.ItemIndex]) ...
```

Was sollte man stattdessen tun?

- Die VCL implementiert leider kein MVC
- Aber man kann Objects verwenden
  - Um Objekte darin anzulegen
  - Oder Methodenzeiger
  - Oder Array-Indizes
- Damit ist kein `if` mehr nötig

Zusammenfassung

- Wir sind selbst schuld an unseren Bugs
- Drei Schritte um dem zu begegnen:
  1. Die Gesetze des Codes verstehen
     - Starke Kopplungen erzeugen Ripple Effects
     - Vorbedingungen, Nachbedingungen und Invarianten
     - Semantisch richtig programmieren
  2. Die eigenen Fehler verstehen
     - Murphys Gesetz und der Halbidiot in einem halben Jahr
     - Persönliche Fehler
  3. Bug-feindlich programmieren
     - Simplicity: einfach $\neq$ kurz $\neq$ schnell geschrieben
     - Assertions verwenden
     - DRY

Fragen?

Anhang

Kopplungsarten

Kopplungsarten (von stark bis schwach)

1. Content coupling (Inhaltskopplung)
2. Common coupling (Bereichskopplung)
3. External coupling (Externdatenkopplung)
4. Control coupling (Kontrollkopplung)
5. Stamp coupling (Datenstrukturkopplung)
6. Data coupling (Datenkopplung)
7. Call coupling (Aufrufkopplung)

Content Coupling (Inhaltskopplung)

Beispiel:

```
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;
```

- Auch: „Pathological coupling“ (pathologische Kopplung)
- Abhängigkeit von konkreter Implementierung
- Besser: Propertys verwenden (siehe `TOpenDialog`)

Common Coupling und External Coupling

Common Coupling (Bereichskopplung)

- Kommunikation über einen geteilten Speicherbereich
- Kurz: Globale Variablen sind bööööse!

External Coupling (Externdatenkopplung)

- Kommunikation über externe Daten
- Dateien, Datenbank, Netzwerk, etc.

Control Coupling (Kontrollkopplung)

Beispiel:

```
procedure TMyFeedReader.DeleteItem(item: TFeedItem; recycle: Boolean);
begin
  if recycle then
    // throw in dust bin
  else
    // delete completely
end;
```

- Parameter ist Kontrollflussinformation
- Besser: zwei getrennte Methoden
- „Gute“ Variante der Kontrollkopplung: Rückgabewert ist Kontrollflussinformation

Stamp Coupling (Datenstrukturkopplung)

Beispiel:

```
function Sanitize(edit: TEdit): string;
begin
  Result := edit.Text;
  Result := StringReplace(Result, '<', '&lt;', [rfReplaceAll]);
  Result := StringReplace(Result, '>', '&gt;', [rfReplaceAll]);
end;
```

- Unnötige Kopplung an TEdit
- Besser: String-Parameter nehmen

Data Coupling, Call Coupling, No Coupling

Data Coupling (Datenkopplung)

- Stinknormaler Aufruf mit Parametern

Call Coupling (Aufrufkopplung)

- Aufruf ohne Parameter

No Coupling (keine Kopplung)

- Überhaupt keine Kommunikation zwischen den Modulen

Kopplungsarten – Sonderformen

Sonderformen

- Tramp coupling
- Hybrid coupling (Hybridkopplung)
- Temporal coupling (zeitliche Kopplung)
- Logical coupling (logische Kopplung)
- uses-Kopplung

Tramp Couplung

Beispiel:

```
constructor TMyHtmlParser.Create(const url: string);
begin
  ...
  input := idHTTP.Get(url);
  ...
end;
```

- Ein Parameter wird einfach „durchgereicht“
  $\Rightarrow$ Unnötige Kopplung durch Kenntnis einer Information, die gar nicht benötigt wird
- Hier besser: String oder Stream übergeben und Download auslagern

Hybrid Coupling (Hybridkopplung) (1/2)

Beispiel:

```
procedure TMyFeedReader.SetIntervall(minutes: Integer);
begin
  if minutes = 0 then
    FTimer.Active := False
  else
    FTimer. Intervall := minutes * 60 * 1000;
end;
```

- Parameter gleichzeitig Datum und Kontrollflussinformation
- Besser: zwei getrennte Methoden

Hybrid Coupling (Hybridkopplung) (2/2)

Beispiel:

```
position := Pos('bla', s);
if position <> 0 then
  ShowMessage('Gefunden an Position ' + IntToStr(position));
```

- Rückgabewert gleichzeitig Datum und Kontrollflussinformation
- Weniger schlimme Variante der Hybridkopplung

Temporal Coupling (zeitliche Kopplung)

Beispiel:

```
myParser.load(input);
myParser.parse;
while myParser.hasMoreTokens do
begin
  WriteLn(myParser.nextToken);
end;
```

- Die Semantik eines Aufrufs hängt vom Zeitpunkt ab
  - bzw. die Reihenfolge der Aufrufe ist wichtig
- Hier muss nach `load` noch `parse` aufgerufen werden
- Besser `parse` intern aufrufen
  - Entweder in `load` (eager)
  - Oder in `hasMoreTokens` und `nextToken` (lazy)

Logical Coupling (logische Kopplung) (1/2)

Beispiel:

```
// GUI Layer
CardImage.LoadFromFile(Path + card.ToString + '.bmp'); // e.g. C:\Some\Path\HerzAss.bmp

// Application Logic Layer
function TPlayingCard.ToString: string;
begin
  Result := SuitToStr(suit) + ValueToStr(value); // e.g. 'Herz' + 'Ass'
end;
```

- Nicht technische, sondern logische Annahmen
- Hier trifft `TPlayingCard` Annahmen über die Visualisierung

Logical Coupling (logische Kopplung) (2/2)

Beispiel:

```
SomeApplicationLogicClass.OnUpdateGUI := UpdateGUI; // Event verweist schon im Namen auf die GUI
```

- Hier wurde versucht über Events technisch zu entkoppeln
- Logisch gesehen besteht die Kopplung aber weiterhin

uses-Kopplung

Beispiel:

TypeDefs.pas, Constants.pas, Utils.pas etc.

- Gruppierung nach Syntax
- Unnötige Kopplung an semantisch nicht fassbare Units
- Bei Wiederverwendung: Datei mitschleppen ($\rightarrow$ dead code) oder nötige Deklarationen herausoperieren ($\rightarrow$ aufwändig)
- Besser: semantisch gruppieren
  - Wenn in nur einer Unit sinnvoll verwendbar: genau da rein
  - Wenn in mehreren Units verwendbar, semantisch weiter gruppieren: `MyVersionUtils.pas`, `MyCurrencyUtils.pas`
    - Teilweise kann und sollte man dann das `-Utils` weglassen
  - Wenn wirklich (fast) überall verwendet: ggf. Struktur ändern

DBC und Unit Testing

- DBC und Unit Testing scheinen sich zu überschneiden
  - Aus DBC-Spezifikationen lassen sich sogar automatisch Unit Tests generieren
- Andere Sichtweise auf Methoden, anderer Fokus[4]
- Aber miteinander vereinbar
  - Unit Tests und Exceptions für `public`/`published`
  - Assertions für private
  - Bei `protected` abhängig vom konkreten Fall

[4] `http://onestepback.org/index.cgi/Tech/Programming/DbcAndTesting.html`

Weiteres

Weitere Themen, auf die hier nicht eingegangen werden konnte:

Weiteres – Die Gesetze des Codes

- Fehler in unterschiedlichen Phasen, Änderungskostenkurve[5]
- Aliasing[6] und die daraus entstehenden Fehler
- Seiteneffekte[7], Command-Query-Separation[8]
- Wiederverwendung und die dadurch gelösten und entstehenden Probleme
- Fehleranfällige Sprachkonstrukte, Code Smells
- Durch OOP gelöste und entstehende Probleme (Fragile Base Class Problem[9], Kreis-Ellipse-Problem[10], . . .)
- . . .

[5] http://xprogramming.com/articles/cost_of_change/
[6] http://de.wikipedia.org/wiki/Aliasing
[7] http://forum.delphi-treff.de/showthread.php?29292
[8] http://martinfowler.com/bliki/CommandQuerySeparation.html
[9] http://de.wikipedia.org/wiki/Fragile_Base_Class_Problem
[10] http://www.parashift.com/c++-faq-lite/proper-inheritance.html#faq-21.6

Weiteres – Eigene Fehler

- Qualitätsmanagement
  - CMMI, TQM, PSP[11], TSP, ...
  - Quality Improvement Paradigm[12]
  - Metriken
  - Experience Factory
- Psychologische Aspekte
  - Proofreader's error, etc.
- ...

[11] http://en.wikipedia.org/wiki/Personal_Software_Process

[12] http://wwwagse.informatik.uni-kl.de/teaching/pia-scenario/welcome.html

Weiteres – Bug-feindlich programmieren

- Software-Architektur und Design
  - Kapselung, Abstraktionsschichten
  - Law of Demeter, Shy Code, „The Paperboy and the Wallet“[13]
- Qualitätssicherung
  - Regressionstests, Reviewtechniken, Statischen Analyse
- Lesbarkeit
  - Methodengröße[14], Verschachtelungstiefe
  - Syntactic Noise, sprechende Bezeichner, Konstanten[15]
- Weiteres zu DRY[16], mehr über Assertions[17]
- (API-)Design[18]: Immutability, Parameterlisten, ...

[13] http://www.ccs.neu.edu/research/demeter/related-work/pragmatic-programmer/jan_03_enbug.pdf
[14] http://www.gigamonkeys.com/book/practical-a-simple-database.html (Fußnote 10)
[15] siehe auch http://www.christian-rehn.de/2010/12/31/magic-values/
[16] http://pragprog.com/book/tpp/the-pragmatic-programmer (sehr empfehlenswert!)
[17] http://download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html
[18] http://www.youtube.com/watch?v=aAb7hSCtvGw