ITDevCon 2017

European Delphi Conference

Non aprite quella Unit!

Scrivere "codice pulito" in Delphi

Chi sono

Marco Breveglieri

Software & Web Developer, Teacher and Consultant

@ ABLS Team Software & Web (Reggio Emilia)

Blogger (www.compilaquindiva.com)

Host *@ Delphi Podcast* (www.delphipodcast.com)

...and Sushi eater!

Agenda

Contenuti

- Riconoscere il "codice cattivo": esempi pratici
- Scrivere "codice pulito" (*Clean Code*)
- Q & A
- Conclusioni

Obiettivi

- L'architettura del software è importante, ma ci sono anche altri aspetti, più piccoli ma non meno importanti
- Ridurre i tempi di manutenzione successiva (80% dello sviluppo software)

Galleria di esempi non adatta alle persone sensibili, ai programmatori deboli di cuore, alle donne in stato interessante.

Il "Codice Cattivo" 😈

Singleton alla massima potenza

```
procedure Execute;
begin
  // 2 milioni di righe di codice...
end;
```

Ripetitore di eccezioni

```
try
  // ... codice che produce errori nel 99% dei casi...
except
  on E:EYourFavouriteException do
  begin
    raise;
  end;
  on E:EAccessViolation do
  begin
    raise;
  end;
  on E:Exception do
  begin
    raise Exception.Create('Errore: ' + E.Message);
  end;
end;
```

Commento "Grazie al..."

```
function CheckSomething: Boolean;
begin

  // ...

  // ...

  // Restituisce un esito positivo.
  Result := True;

end;
```

Argomenti... fallaci

```
Azienda := GetAzienda('ABLS', True, 1, True, False);

SendMail('Test invio mail', 'Corpo del messaggio',
  'marco@abls.it', True, False, 1, True, 3, False);
```

```
function GetAzienda(const ACompanyName: string;
  IsCustomer: Boolean; ALevel: Integer;
  VisibleOnly: Boolean; ExcludeUnreliable: Boolean): TAzienda;
begin
  // ...
end;

procedure SendMail(const ASubject, ABody, ARecipient: string;
  AConfirmRead, AConfirmSent: Boolean; APriority: Integer;
  RemoveAttachmentAfterSend: Boolean; MaxRetryCount: Integer;
  UseHtml: Boolean);
```

Du iu spik Inglish?

```
interface

type
  TFileStatus = (Confirmed, Backupped, Annulled);
```

No comment...

```
function GetSignatureProviderTypeFromString(const AType: string): TSignatureProviderType;
begin
  if AType.ToLower() = TRttiEnumerationType.GetName<TSignatureProviderType>(TSignatureProviderType.Aruba) then
    Result := TSignatureProviderType.Aruba
  else
    if AType.ToLower() = TRttiEnumerationType.GetName<TSignatureProviderType>(TSignatureProviderType.ArubaVerifyVol) then
      Result := TSignatureProviderType.ArubaVerifyVol
    else
      if AType.ToLower() = TRttiEnumerationType.GetName<TSignatureProviderType>(TSignatureProviderType.PkBox) then
        Result := TSignatureProviderType.PkBox
      else
        if AType.ToLower() = TRttiEnumerationType.GetName<TSignatureProviderType>(TSignatureProviderType.InfoCert) then
          Result := TSignatureProviderType.InfoCert
        else
          if AType.ToLower() = TRttiEnumerationType.GetName<TSignatureProviderType>(TSignatureProviderType.Namirial) then
            Result := TSignatureProviderType.Namirial
          else
            Result := Aruba;
end;
```

Scusa ma...

Perché scriviamo codice cattivo?

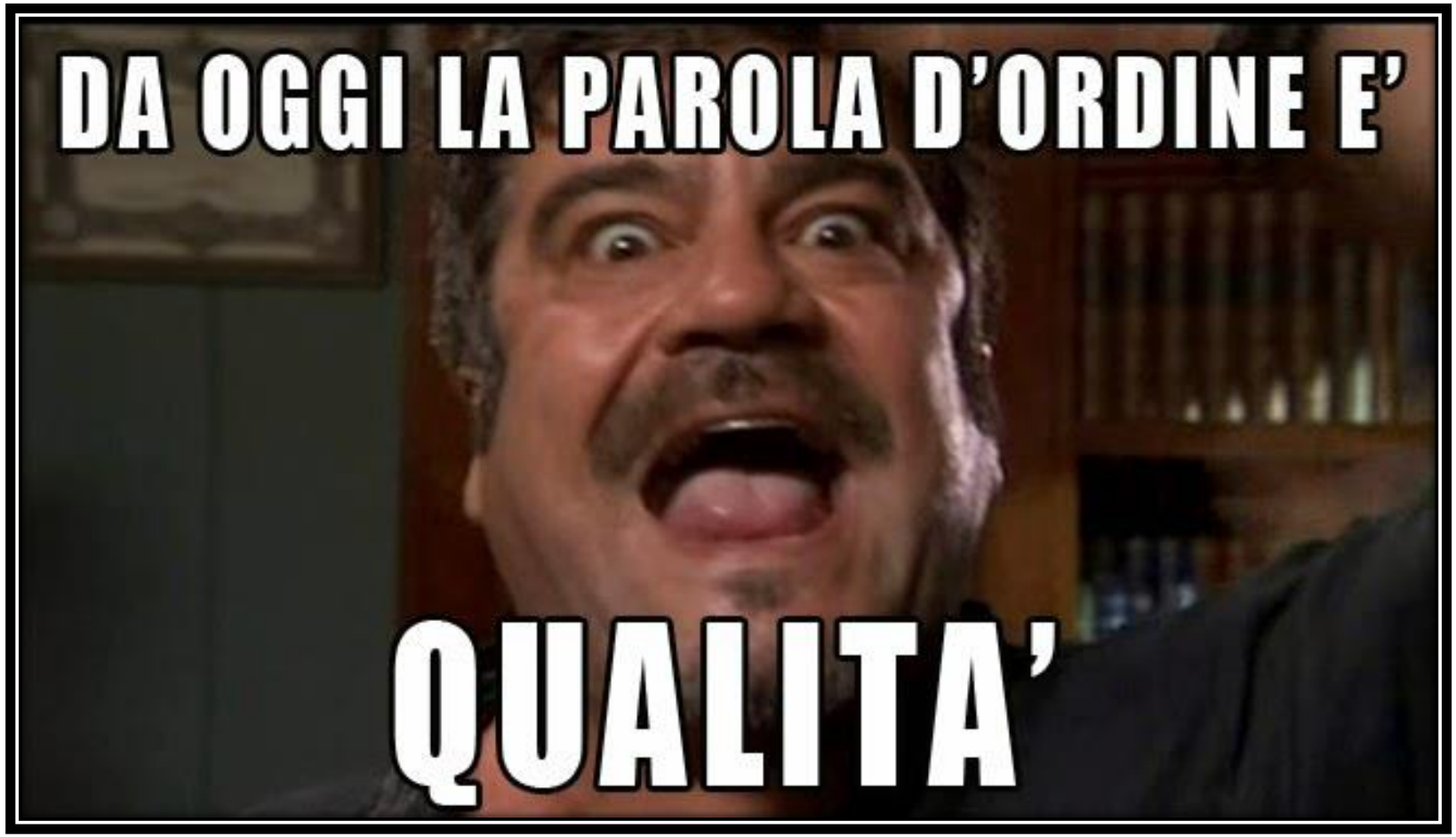
- Frenesia di rilasciare funzionalità il più velocemente possibile
- Scappare a casa alla fine della giornata
- Cattive abitudini acquisite in passato
- Analisi oscura o fuorviante, nessuna riflessione preventiva
- Imporsi di ristrutturare in seguito il codice, violando la legge di LeBlanc (*)

(*) Later equals never

Rule of Thumb

«Always code as if the guy who ends up mantaining your code is a violent psychopath who knows where you live».

Come fare?



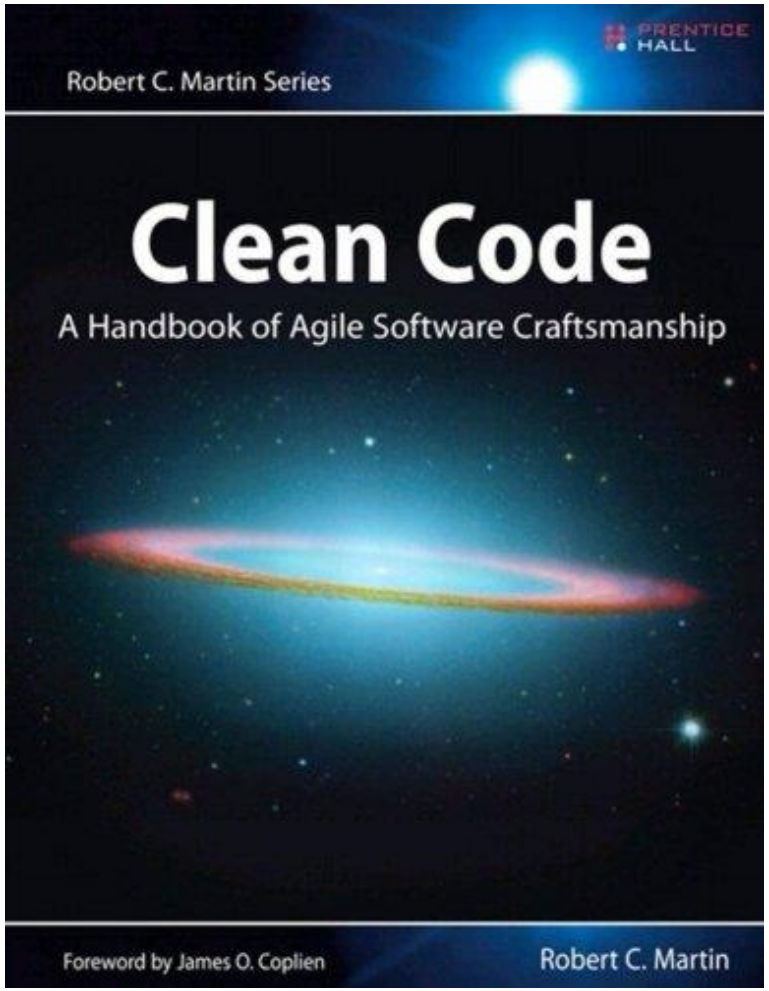
Clean Code

«Clean Code: A Handbook of Agile Software Craftsmanship»

Robert C. Martin

(prima pubblicazione: 11 ago 2008)

"Making your code readable is as important as making it executable."



Quindi


I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code

ITDevCon 2017
European Delphi Conference

Clean Code

Meaningful Names

L'importanza nel nome

I nomi sono ovunque all'interno del codice!

- *variabili*
- *costanti*
- *metodi*
- *parametri*
- *classi*
- *package*

Rivelare le intenzioni

Usare nomi che rispettino il criterio di "Intention Revealing"

- Il nome deve rispondere a tutte le possibili domande
- Usare nomi significativi richiede tempo, ma la scelta si ripaga in futuro
- In caso di dubbio, scegliere un nome temporaneo
  (ma ricordarsi di fare il refactoring quanto prima!)

Rivelare le intenzioni

```
// WRONG

var
  d: Integer; // elapsed time in days

// OK

var
  ElapsedTimeInDays: Integer;
  DaysSinceCreation: Integer;
```

Rivelare le intenzioni

```
// WRONG

function GetThem(): TArray<Integer>;
var
  List: TList<Integer>;
  I: Integer;
begin
  List := TList<Integer>.Create;
  try
    for I := 1 to 100 do
      if I mod 2 = 0 then
        List.Add(I);
    Result := List.ToArray;
  finally
    List.Free;
  end;
end;
```

```
// OK

function GetArrayOfEvenNumbers:
 TArray<Integer>;
const
  RangeMin: Integer = 1;
  RangeMax: Integer = 100;
var
  EvenNumberList: TList<Integer>;
  CurrentNumber: Integer;
begin
  EvenNumberList := TList<Integer>.Create;
  try
    for CurrentNumber := RangeMin to RangeMax do
      if CurrentNumber mod 2 = 0 then
        EvenNumberList.Add(CurrentNumber);
    Result := EvenNumberList.ToArray;
  finally
    EvenNumberList.Free;
  end;
end;
```

Evitare disinformazione

Evitare tutto ciò che fornisce indizi devianti a chi legge il codice.

- Non nascondere il motivo o il tipo di valori, entità o elementi
- Non usare nomi troppo simili tra loro, o che usano caratteri equivocabili
- Evitare precisazioni che suggeriscano una struttura dati diversa da quella reale
- Non usare nomi diversi per concetti sostanzialmente identici

Evitare disinformazione

```
// WRONG

function GetIntegerList(): TArray<Integer>;

var
  EvenNumberList: TArray<Integer>;
  Numbers: TList<Integer>;

function ControllerForEfficientHandlingOfStrings(): TArray<Integer>;
function ControllerForEfficientStoringOfStrings(): TArray<Integer>;

// OK

function GetArrayOfEvenNumbers: TArray<Integer>;

var
  EvenNumberList: TList<Integer>;
  EvenNumbers: TList<Integer>;
  BunchOfNumbers: TList<Integer>;
  Accounts: TArray<TAccount>;
```

```
// WRONG

type

{ TProduct }
  TProduct = class
    // ...
  end;

{ TProductInfo }
  TProductInfo = class
    // ...
  end;

{ TProductData }
  TProductData = class
    // ...
  end;
```

Usare nomi pronunciabili

- Non usare abbreviazioni che rendano i nomi "cacofonici"
- Se un nome non può essere pronunciato, non è possibile discuterne senza sembrare perfetti idioti 😬
- Un nome non pronunciabile non può essere cercato con facilità

Usare nomi pronunciabili

```
type

// WRONG

  TDataRcrd102 = class
  strict private
    FGenYMDHMS: TDateTime; // (generation date, year, month, …)
    FModYMDHMS: TDateTime; // (modification date, year, month, …)
    FPszqInt: string; // ???
  end;

// OK

  TCustomer = class
  strict private
    FGenerationTimestamp: TDateTime;
    FModificationTimestamp: TDateTime;
    FUniqueID: string;
  end;
```

Altri consigli utili…

- Evitare nomi con una sola lettera: non sono ricercabili facilmente e costringono al cosiddetto "mental mapping"
- Evitare l'uso di encoding e di caratteri speciali
- Non inserite il tipo di dato all'interno del nome della variabile (per intero o come prefisso)
- Evitare le convenzioni per membri privati e interfacce (forse un pochino "borderline" come requisito)
- Nei nomi delle classi, usare sostantivi o azioni concrete (es. *Customer, Page, Account, AddressParser, …*) ed evitare termini generici (es. *Manager, Processor, Data, Info, …*). Non deve essere un verbo!
- Nei nomi dei metodi, usare verbi e complemento oggetto

Altri consigli utili…

- Non usare nomi aneddotici, slang o "fare i simpatici"
  (ad esempio, chiamare un metodo *SchiantaClienti()*)
- Scegliere un termine univoco per le stesse operazioni
  (ad esempio, evitare di usare *get*, *retrieve*, *fetch*, …)
- Analogamente, non usare un termine per due o più scopi diversi
- Usa termini tecnici: il codice verrà letto da programmatori

Non temere critiche se vuoi modificare nomi (in meglio) quando necessario.

Funzioni

Funzioni

Devono essere corte!

- Le funzioni dovrebbero essere il più corte e piccole possibile
- Evitare di concentrare logica in chilometri e chilometri di codice
- Ogni piccola funzione dovrebbe raccontare una (breve) storia
- Una funzione può essere fattorizzata se è possibile estrarne una da essa con un nome che non coincide con le istruzioni
- Nel dare il nome alla funzione, rispettare le regole già viste per "meaningful names"

"Functions should do one thing.
They should do it well.
They should do it only."

Funzioni

Un solo livello di astrazione per ogni funzione.

```
procedure RenderPage(const APath: string);
var
  HtmlText, PageContents: string;
begin
  HtmlText := GetHtml();
  PageContents := RenderPage(HtmlText);
  TFile.WriteAllText(APath, PageContents);
end;
```

Case

- Non si dovrebbe abusare dei costrutti **case…of**
- Devono apparire solamente in classi di basso livello
- Non devono essere ripetuti più volte nel codice
- E' preferibile sostituirli (usando polimorfismo o altro)

Case (semplice ma limitante)

```
function CalculatePay(Employee: TEmployee): Currency;
begin
  case Employee.Kind of
    Commissioned:
      Result := CalculateCommissionedPay(Employee);
    Hourly:
      Result := CalculateHourlyPay(Employee);
    Salaried:
      Result := CalculateSalariedPay(Employee);
    else
      raise Exception.Create('Invalid employee kind');
    end;
end;
```

Case (alternativa) 1/4

```
{ IPayCalculator }

  IPayCalculator = interface
    ['{1C993E82-A3EF-43B3-8F05-5C0500D5CFE3}']
    function CalculatePay(Employee: TEmployee): Currency;
  end;

{ Routines }

function CalculatePay(Employee: TEmployee): Currency;
procedure RegisterCalculator(AKind: TEmployeeKind;
  ACalculator: IPayCalculator);
procedure UnregisterCalculator(AKind: TEmployeeKind);
```

Case (alternativa) 2/4

```
implementation

var
  Calculators: TDictionary<TEmployeeKind, IPayCalculator>;

procedure RegisterCalculator(AKind: TEmployeeKind; ACalculator: IPayCalculator);
begin
  Calculators.Add(AKind, ACalculator);
end;

procedure UnregisterCalculator(AKind: TEmployeeKind);
begin
  if Calculators.ContainsKey(AKind) then
    Calculators.Remove(AKind);
end;

initialization
  Calculators := TDictionary<TEmployeeKind, IPayCalculator>.Create();

finalization
  if Calculators <> nil then
    Calculators.Free;
  end.
```

Case (alternativa) 3/4

```
{ TEmployeeCommissionedPayCalculator }

  TEmployeeCommissionedPayCalculator = class (TInterfacedObject, IPayCalculator)
  public
    function CalculatePay(Employee: TEmployee): Currency;
  end;

{ TEmployeeHourlyPayCalculator }

  TEmployeeHourlyPayCalculator = class (TInterfacedObject, IPayCalculator)
  public
    function CalculatePay(Employee: TEmployee): Currency;
  end;

{ TEmployeeSalariedPayCalculator }

  TEmployeeSalariedPayCalculator = class (TInterfacedObject, IPayCalculator)
  public
    function CalculatePay(Employee: TEmployee): Currency;
  end;
```

Case (alternativa) 4/4

```
RegisterCalculator(TEmployeeKind.Commissioned, TEmployeeCommissionedPayCalculator.Create);
RegisterCalculator(TEmployeeKind.Hourly, TEmployeeHourlyPayCalculator.Create);
RegisterCalculator(TEmployeeKind.Salaried, TEmployeeSalariedPayCalculator.Create);
```

```
function CalculatePay(Employee: TEmployee): Currency;
var
  PayCalculator: IPayCalculator;
begin
  if not Calculators.TryGetValue(Employee.Kind, PayCalculator) then
    raise Exception.Create('Calculator not found or not registered');
  Result := PayCalculator.CalculatePay(Employee);
end;
```

Nomi delle funzioni

Assegnare un "buon nome" alla funzione

- Rispettare le regole dei "meaningful names" già visti
- Non temere di dare alla funzione un nome lungo
- Aiutarsi con verbi e nomi composti (sfruttando il "Pascal Case")
- Mantenere consistenza fra i termini usati per le azioni eseguibili

"You know you are working on clean code when each routine turns out to be pretty much what you expected."

Parametri delle funzioni

Alcune regole di base…

- Il numero ideale dei parametri di una funzione è zero
- Un parametro è tollerato, due sono tanti, tre devono essere evitati, quattro devono essere realmente giustificati
- Ogni parametro aggiunge complessità e un effort mentale nella lettura
- Rendono difficoltosi gli Unit Test: si dovrebbe creare un "case" per ogni combinazione di parametri possibile
- Prediligere i valori di ritorno rispetto a parametri passati per valore
- Racchiudere parametri multipli in strutture (record o classi)

Parametri delle funzioni

Quante volte vi è successo?

```
// Declaration
procedure AssertEquals(Expected, Actual: Extended);



// Right usage?
procedure Assert(100, Result);
procedure Assert(Result, 100);
```

Parametri di tipo "flag"

I parametri "flag" (es. booleani) sono illeggibili!

- Passare un booleano a una funzione va sempre evitato
- Complica immediatamente la firma di una funzione
- E' già un proclama del fatto che la funzione fa più di una cosa

```
Render(True);
```

ForceUpdate: Boolean

Parametri di tipo "flag"

```
// WRONG
procedure Render(ForceUpdate: Boolean);
```

```
// BETTER
procedure RenderAndForceUpdate();
procedure RenderNotForcingUpdate();
```

Evitare i "side effect"

Una funzione con effetti collaterali (magari nascosti)

- Fa più di ciò che dovrebbe aumentando l'entropia del sistema
- Mente allo sviluppatore (dichiara un compito diverso da quello eseguito)

Possibili soluzioni

- Evitare la modifica di variabili globali o di istanza
  (a meno che non sia il compito specifico assegnato alla funzione)
- Separare le letture dalle scritture (vedi paradigma CQRS)

Evitare i "side effect"

```
function TUserValidator.CheckPassword(const AUserName: string;
                                      const APassword: string): Boolean;
var
  User: TUser;
  CodedPhrase, ClearPhrase: string;
begin
  Result := False;
  User := TUserGateway.FindByName(AUserName);
  if User <> User.Null then
  begin
    CodedPhrase := User.GetPhraseEncodedByPassword();
    ClearPhrase := FCryptographer.Decrypt(CodedPhrase, APassword);
    if ClearPhrase = 'Valid Password' then
    begin
      TUserSession.Initialize();
      Result := True;
    end;
  end;
end;
```

Gestione degli errori

- Preferire l'uso delle eccezioni al posto di "error code" di ritorno
- Isolare in funzioni distinte la logica di gestione delle eccezioni
  - Ogni funzione ha un solo compito, gestire gli errori è uno di questi
  - La funzione che produce l'eccezione rimane più leggibile
  - La logica di gestione dell'errore (es. log eccezione) può essere riutilizzata

```
function TUserValidator.Login(const AUserName,
  APassword: string): Boolean;
begin
  try
    // ...login dell'utente...
  except
    on E:Exception do
    begin
      LogWebException(E);
    end;
  end;
end;

procedure TUserValidator.LogWebException(
  AnException: Exception);
begin
  FLogger.Log(AnException.Message);
end;
```

Altri consigli utili

- **Don't repeat yourself!** Evitare di scrivere più volte la stessa logica
  - Utilizziamo tutti paradigmi a nostra disposizione (OOP, AOP, COP, …)
- Attenzione alla struttura!
  - Usare un solo return (Exit) nel corpo della funzione
  - Ridurre i Break e i Continue
  - Mai - ripeto *mai* - usare Goto
- Rifattorizzare le funzioni prima che "esplodano"
  - Usare i tool di Refactoring disponibili in Delphi
  - Creare Unit Test per le parti di codice isolate

Toxicity Metrics

Delphi fornisce informazioni sulle **metriche di tossicità** dei metodi.

Method Name	Length	Parameters	If Depth	Cyclomatic Complexity	Toxicity
TUserValidator.CheckPassword	10	2	2	3	0,433
CalculatePay	5	1	0	4	0,271
GetThem	7	0	1	3	0,263
GetArrayOfEvenNumbers	7	0	1	3	0,263
CalculatePay	3	1	1	2	0,213
UnregisterCalculator	2	1	1	2	0,200
TUserValidat Login	3	2	0	1	0,163
RegisterCalcu	1	2	0	1	0,138
Rend	1	1	0	1	0,121
RenderPage	1	1	0	1	0,096
CalculateCommissionedPay	1	1	0	1	0,096
CalculateHourlyPay	1	1	0	1	0,096
CalculateSalariedPay	1	1	0	1	0,096
TEmployeeCommissionedPayCal...	1	1	0	1	0,096
TEmployeeHourlyPayCalculator....	1	1	0	1	0,096
TEmployeeSalariedPayCalculator....	1	1	0	1	0,096
TSensors.GetByIndex	1	1	0	1	0,096
TUserValidator.LogWebException	1	1	0	1	0,096
Render	0	1	0	1	0,083
TSensors.Destroy	3	0	0	1	0,079
TSensors.Create	2	0	0	1	0,067
GetHtml	1	0	0	1	0,054
RenderAndForceUpdate	0	0	0	1	0,042
RenderNotForcingUpdate	0	0	0	1	0,042

Versione "bad"

Versione "clean"

Commenti

Commenti

- Non commentare il "codice cattivo": riscrivilo!
- Commentare il codice con buon senso
  - Un commento nel punto giusto può essere utilissimo
  - Riempire di commenti frivoli e dogmatici il codice non serve a nulla
- I commenti sono sempre un fallimento
  - Se appaiono, significa che il codice non è in grado di esprimere ciò che accade
- Evitare commenti destinati a "perdersi"
  - Quando si fa refactoring, il codice viene spostato e il commento può rimanere "orfano"
  - Difficilmente vengono manutenuti dagli sviluppatori: è troppo oneroso
- Non commentare il codice sorgente
  - Ma non ce l'hai un buon sistema di Source Control?!
- Si possono usare per inserire dei "TODO" (ma ricordarsi di risolverli!)
- La verità è la fuori… ma è nel codice!

Commenti

Qual è la versione migliore?

1)

```
// Check if the employee is eligible for full benefits
if (Employee.Flags and HOURLY_FLAG > 0)
  and (Employee.Age >= 65) then
begin
end;
```

2)

```
if Employee.IsEligibleForFullBenefits() then
begin
end;
```

ITDevCon 2017
European Delphi Conference

Formattazione

La formattazione è importante

Essa dimostra la nostra cura per i dettagli,
il nostro ordine, la nostra professionalità.

Se il codice appare come scritto da marinai ubriachi,
si potrebbe pensare che la stessa disattenzione e
sciatteria venga applicata a qualsiasi altra parte
del progetto affidatoci dal cliente.

Delphi ha il comando "Format Code" (Ctrl+D)
che ci viene in aiuto.

Ordine verticale

Ogni modulo di codice sorgente è come un articolo di giornale.

- Non deve essere troppo corto, ma nemmeno troppo lungo
- Deve raccontare una "storia"
- Il titolo (cioè il nome) deve essere rappresentativo
- Deve favorire una lettura "top down"
  - Gli elementi di alto livello nella parte superiore
  - I dettagli implementativi nella parte inferiore

Ordine orizzontale

Quanto deve essere lunga una riga di codice?

- I programmatori preferiscono le linee corte
- La difficoltà di lettura rispetto alla lunghezza aumenta esponenzialmente
  - Superati gli 80 caratteri (*Hollerith Limit*), la leggibilità cala in modo drastico
- Usare in modo proficuo gli spazi
  - Isolare le parti dell'istruzione, unire elementi correlati
  - Non incolonnare i nomi, i tipi di dato, i livelli di accessibilità

Indentazione

- E' di importanza fondamentale e imprescindibile!
- Evidenzia i blocchi logici del codice e i suoi costrutti
- Agevola la lettura del codice scorrendo le sue varie parti

- E' efficace quando condivisa in team
  - Garantisce consistenza alla formattazione del codice
  - Delphi permette di personalizzare le regole e condividere la configurazione con gli altri sviluppatori del team (*Formatter -> Profiles and Status*)

ITDevCon 2017
European Delphi Conference

Strutture dati

Data Abstraction

- Si usa l'Information Hiding per nascondere i dettagli implementativi: facciamolo!
  - Non prevedere accesso automatico ai campi privati della classe tramite proprietà
- Non limitarsi all'uso di getter/setter per accedere ai valori
  - Prevedere metodi che indichino nel nome il motivo per cui si impostano i campi

```
type

  IVechicle = interface
    function GetFuelTankCapacityInGallons(): Double;
    function GetGallonsOfGasoline(): Double;
    function GetPercentFuelRemaining(): Double;
  end;
```

Law of Demeter

Un metodo "M" di una classe "C" può chiamare solo metodi che appartengono a

- *"C"*
- *Un oggetto creato da "C"*
- *Un oggetto passato come parametro a "M"*
- *Un oggetto memorizzato in un campo di "C"*

Questo codice è errato

```
OutputDir := Context.GetOptions().GetScratchDir().GetAbsolutePath();
```

(a meno che i valori restituiti non siano DTOs).

Data Transfer Objects

- C'è differenza tra classi e strutture dati generiche
  - Gli oggetti racchiudono dettagli implementativi
  - Le strutture dati (es. record) sono gruppi di campi informativi senza logica
  - Gli oggetti *non* espongono la propria struttura interna
- I DTOs sono ottimi per creare oggetti/record con parametri da passare
  - Idealmente si possono usare nelle funzioni con molti parametri
  - Si può aggiungere un nuovo membro in qualsiasi momento
  - Sono "leggeri" (es. record Delphi sono allocati sullo stack) e non richiedono memory management

ITDevCon 2017
European Delphi Conference

Gestione degli errori

Creare le eccezioni

- Utilizzare enumerativi al posto di costanti numeriche per indizi sull'errore
  - Abbiamo un linguaggio di alto livello a disposizione: sfruttiamolo!
- Programmare con il "consumatore" dell'eccezione in mente
  - Creare classi in grado di fornire informazioni di contesto sull'errore
  - Strutturare le classi affinché siano comode a chi dovrà gestire l'eccezione
  - Venire incontro alle esigenze del metodo chiamante
- Il codice deve essere robusto, oltreché leggibile
  - La gestione degli errori e la logica applicativa sono ambiti distinti

Non restituire `nil`

Gli errori, oltre ad essere gestiti, devono anche essere evitati:

una delle cause principali è restituire nil come valore di ritorno.

Poche (valide) ragioni:

- Impone di scrivere molto codice per testare se un valore è diverso da **`nil`**.
- Rende difficile risalire alla chiamata che ha restituito **`nil`** in prima istanza.

Analogamente, una funzione non deve accettare un valore **`nil`** come parametro.

Se avessi un centesimo per ogni "Access Violation" che ho affrontato…

Confini (Boundaries)

Stabilire i confini

- L'uso di librerie, framework (RTL/VCL/FMX) e package esterni richiede attenzione
- **Non sempre il codice di terze parti rispetta i dettami del "Clean Code"**
  - Non tutti i nomi di metodi sono espressivi
  - Il codice può subire modifiche inattese

```
var
  Sensors: TList<TSensor>;
begin
  SensorItem := Sensors.Items[0];
```

Possibili soluzioni:

- Creare una classe wrapper sul tipo di terze parti
- Applicare gli opportuni design pattern, es. *Adapter*

Stabilire i confini

```
TSensors = class
private
    FSensors: TList<TSensor>;
public
    constructor Create;
    destructor Destroy; override;
    function GetByIndex(AIndex: Integer): TSensor;
end;

constructor TSensors.Create;
begin
  inherited;
  FSensors := TList<TSensor>.Create;
end;

destructor TSensors.Destroy;
begin
  FSensors.Free;
  inherited Destroy;
end;

function TSensors.GetByIndex(AIndex: Integer):
TSensor;
begin
  Result := FSensors.Items[AIndex];
end;
```

```
var
  Sensors: TSensors;

begin
  Sensors := TSensors.Create;
  SensorItem := Sensors.GetByIndex(0);
end;
```

Vantaggi

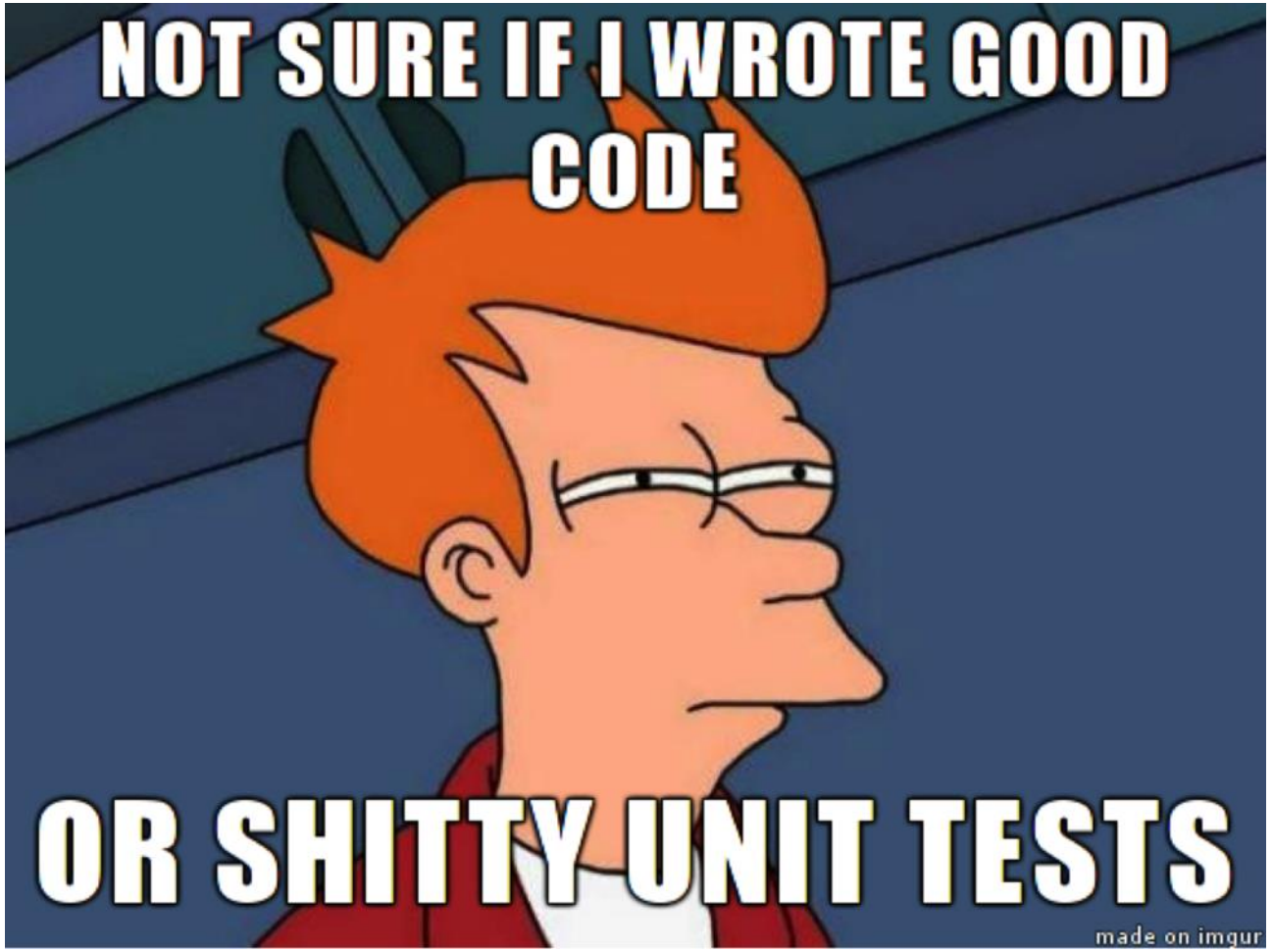
La creazione di "confini" per separare il proprio codice da quello di terze parti

- Rimuove la necessità di modificare codice esistente e già testato
- Consente di mascherare il proprio "codice pulito" dalle modifiche esterne
- Permette di connettere codice a implementazioni non ancora esistenti
- Stabilisce e rende esplicito ciò che ci aspettiamo dal codice
- Rende il codice pulito testabile separatamente dalle altre dipendenze

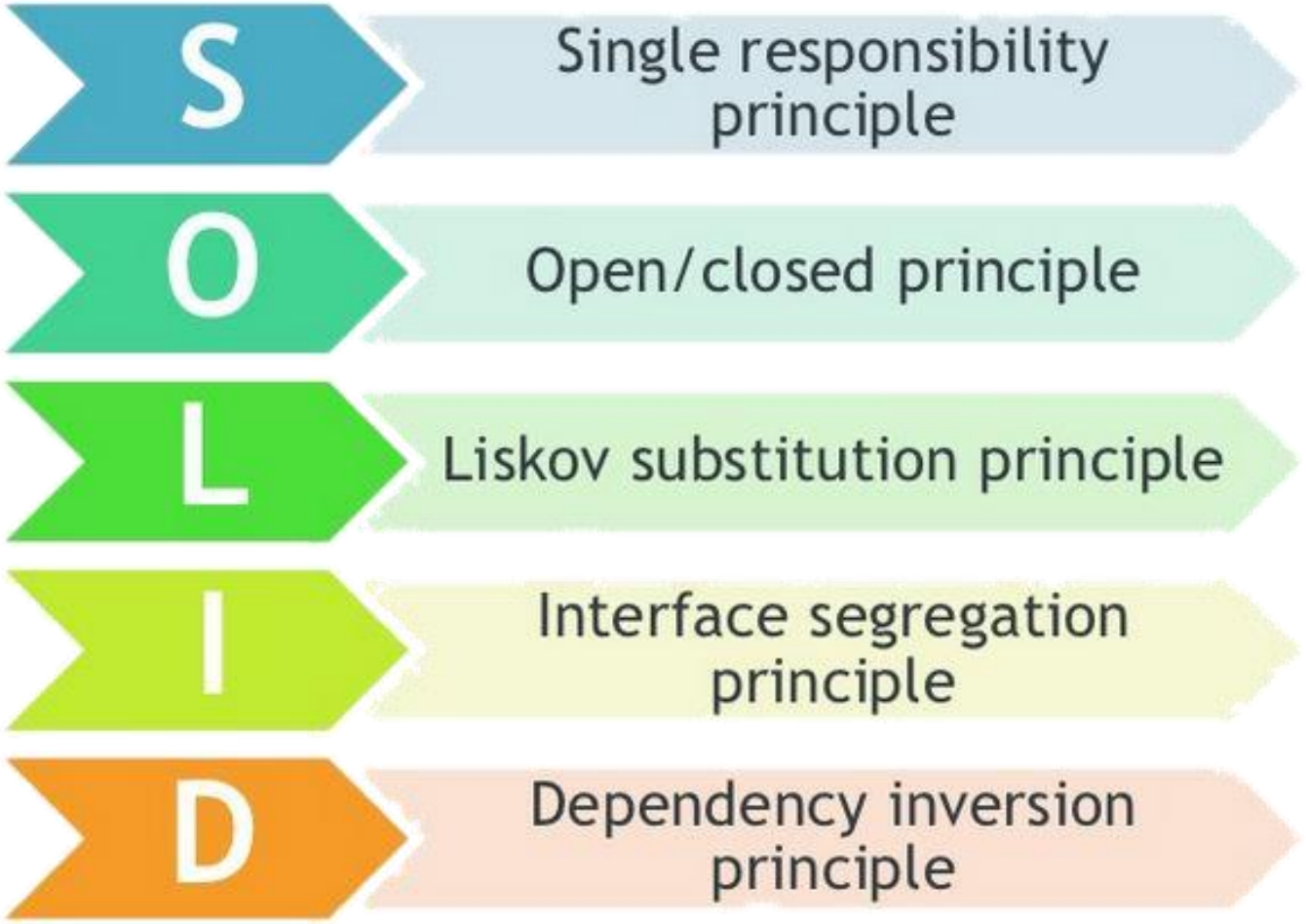
Corollari

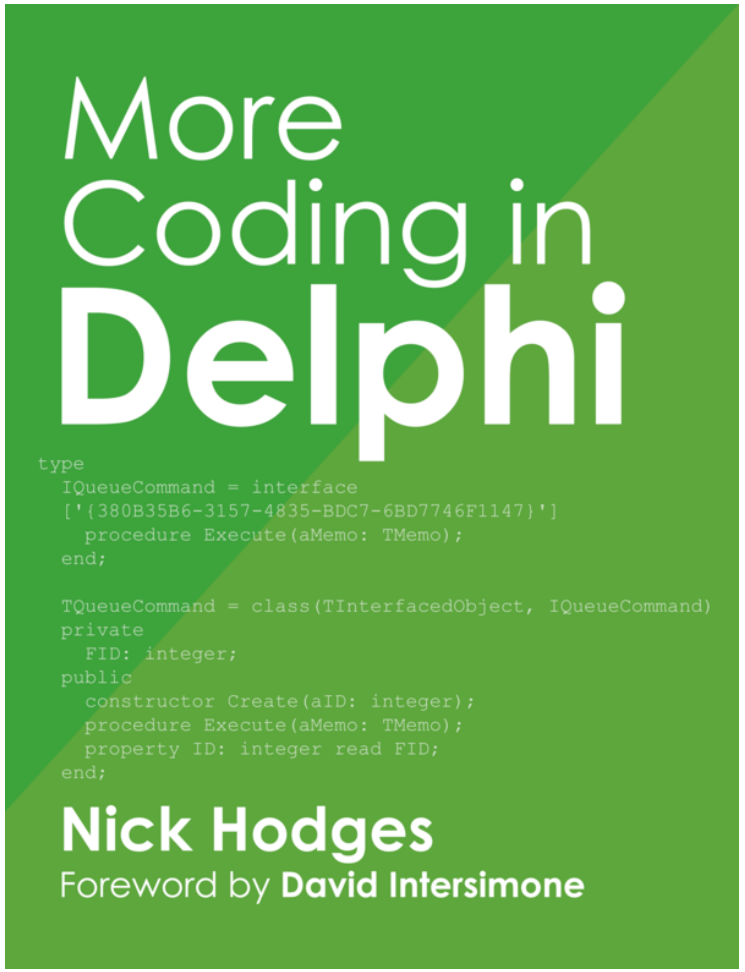
Unit & Integration Testing

Seguendo le norme per scrivere "codice pulito", si ottiene anche **codice testabile**.



SOLID Principles





Domande?

Grazie!