6-7 Giugno 2018 - Piacenza

Clean Code

Scrivere "codice pulito" con Delphi

Chi sono

Marco Breveglieri

Software Developer, Teacher and Consultant

@ ABLS Team (Reggio Emilia)

Blogger (www.compilaquindiva.com)

Podcaster (www.delphipodcast.com)

…and Sushi eater!

Agenda

- Prefazione
- Riconoscere il "codice cattivo": i code smell
- Le pratiche del Clean Code
- Principi SOLID
- Q & A
- Conclusioni

Introduzione

Perché parlare di Clean Code?

- Ogni anno una quantità enorme di tempo e risorse significative viene sprecata a causa di codice "scritto male"
- Gli sviluppatori spesso lavorano in fretta perché sentono la pressione dei boss e dei clienti per portare a termine il lavoro rapidamente, sacrificando la qualità
- La scarsa qualità del codice costituise un ostacolo per l'azienda: aumenta i costi preventivati per lo sviluppo di nuove funzionalità
- I costi di gestione del codice scritto male dilatano i tempi e costituiscono un ostacolo alle possibilità di marketing

Infine: fare bene il nostro lavoro non è forse un requisito indispensabile? 🤔

8 ragioni per scrivere codice pulito

1. **Chiarezza**: poter capire facilmente il codice velocizza le tempistiche di modifica e soprattutto risoluzione dei bug
2. **Metodologie**: le pratiche moderne quali Unit Testing, TDD, Continuous Integration e altre ancora sono facilmente applicabili quando il codice è scritto bene
3. **Logica**: la facilità di comprendere il codice permette all'occorrenza di ottimizzarlo e renderlo più performante
4. **Manutenzione**: il codice chiaro semplifica la stima e la pianificazione di espansioni e modifiche
5. **Testabile**: l'adozione delle pratiche di codice pulito rende automaticamente "testabile" il codice
6. **Semplicità**: la scrittura di codice pulito contribuisce a mantenere il codice semplice, rispettando quindi i classici e conosciutissimi principi KISS e DRY
7. **Consistenza**: la scelta di termini e nomi contribuisce a creare un vocabolario di base con cui semplificare il colloquio con utenti, operatori, clienti e amministratori, adottando metodologie come DDD (Domain Driven Design)
8. **Risparmio di denaro e tempo**.

Non adatta alle persone sensibili, ai programmatori deboli di cuore, alle donne in stato interessante.

Il "Codice Cattivo"

Una galleria di esempi

Singleton alla massima potenza

```
procedure Execute;
begin
  // 1 milione di righe di codice...
  // ...
  // 2 milioni di righe di codice...
  // ...
  // 10 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 stile "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…

Il "Codice Cattivo"

Cos'è e come riconoscerlo

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

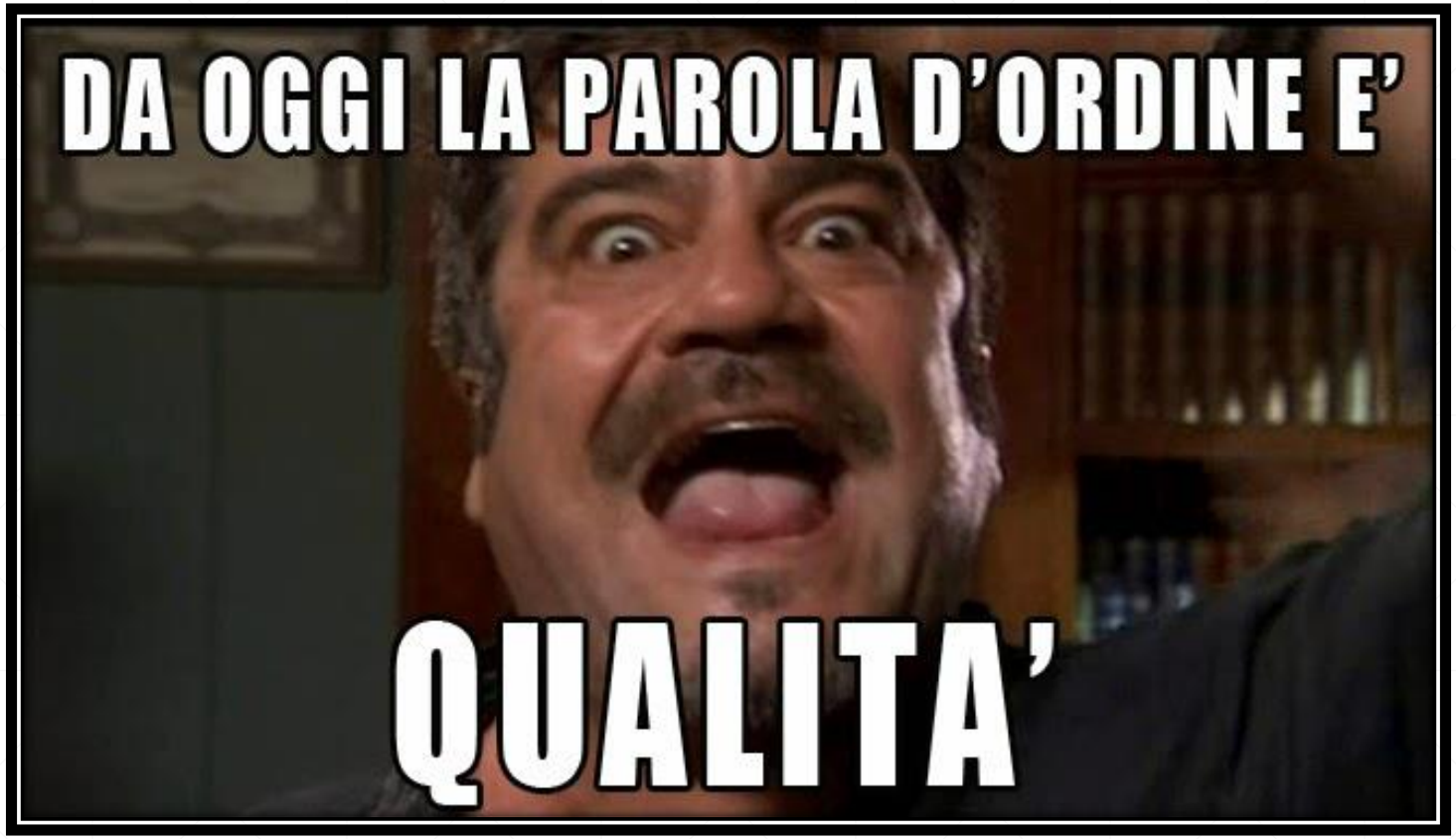
Imporsi di ristrutturare in seguito il codice, violando la legge di LeBlanc (*)

(*) Later equals never

Regola d'oro

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

Come fare?



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

Clean Code

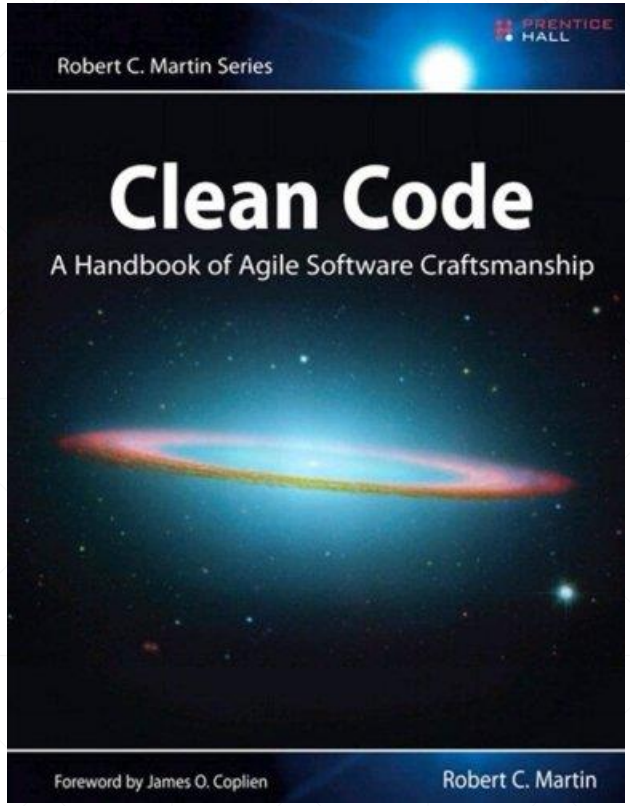
I principi del Clean Code: scrivere codice pulito

Clean Code

«Clean Code: A Handbook of Agile Software Craftsmanship»

Robert C. Martin

(prima pubblicazione: 11 ago 2008)



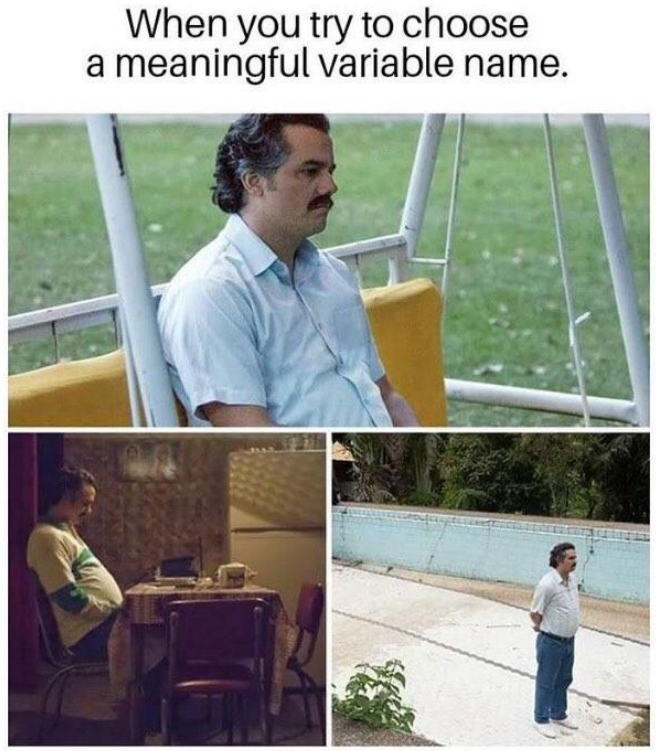
"Making your code readable is as important as making it executable."

Meaningful Names

L'importanza del nome

I nomi sono ovunque all'interno del codice!

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



Evidenziare 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!)

Evidenziare le intenzioni

```
// WRONG
var
   d: Integer; // elapsed time in days


// OK
var
   ElapsedTimeInDays: Integer;
   DaysSinceCreation: Integer;
```

Evidenziare 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

- 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

- 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, fallo.

Implementazione

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."

Livelli di astrazione

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

```
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

```
{ 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

```
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

```
{ 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

```
RegisterCalculator(TEmployeeKind.Commissioned,
  TEmployeeCommissionedPayCalculator.Create);

RegisterCalculator(TEmployeeKind.Hourly,
  EmployeeHourlyPayCalculator.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)

Ordine dei parametri

Ambiguità dei parametri

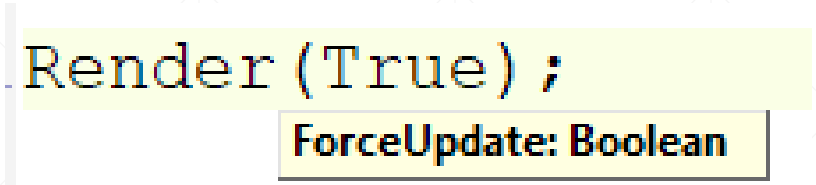
```
// Declaration
procedure AssertEquals(Expected, Actual: Extended);


// 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



Parametri di tipo flag

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

```
// BETTER
procedure Render();
procedure RenderForcingUpdate();
```

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)

Quindi occorre

- 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;
```

Creare eccezioni nel modo corretto

- 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.

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

Commenti

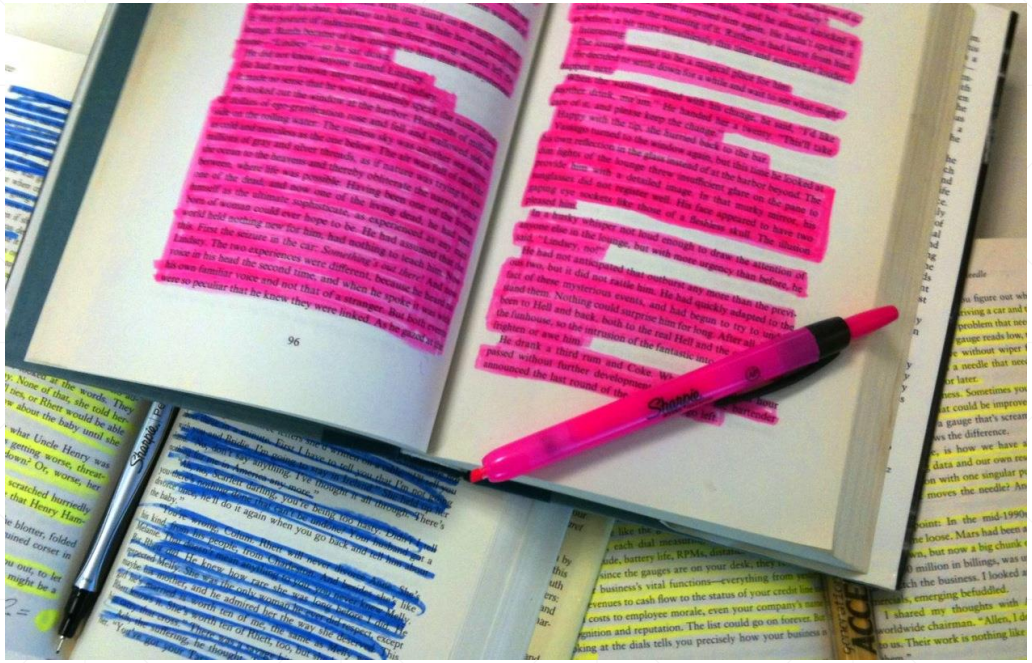
Riscrivi il codice

Non commentare il codice cattivo: **riscrivilo!**

Commentare con buon senso

Commenta il codice ma <u>usa il buon senso</u>

- Un commento nel punto giusto può essere utilissimo
- Riempire di commenti frivoli e dogmatici il codice non serve a nulla



Commento = fallimento

I commenti rappresentano sempre un **fallimento**.

Se appaiono, significa che il codice non è in grado di esprimere ciò che fa.

Quale versione è 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;
```

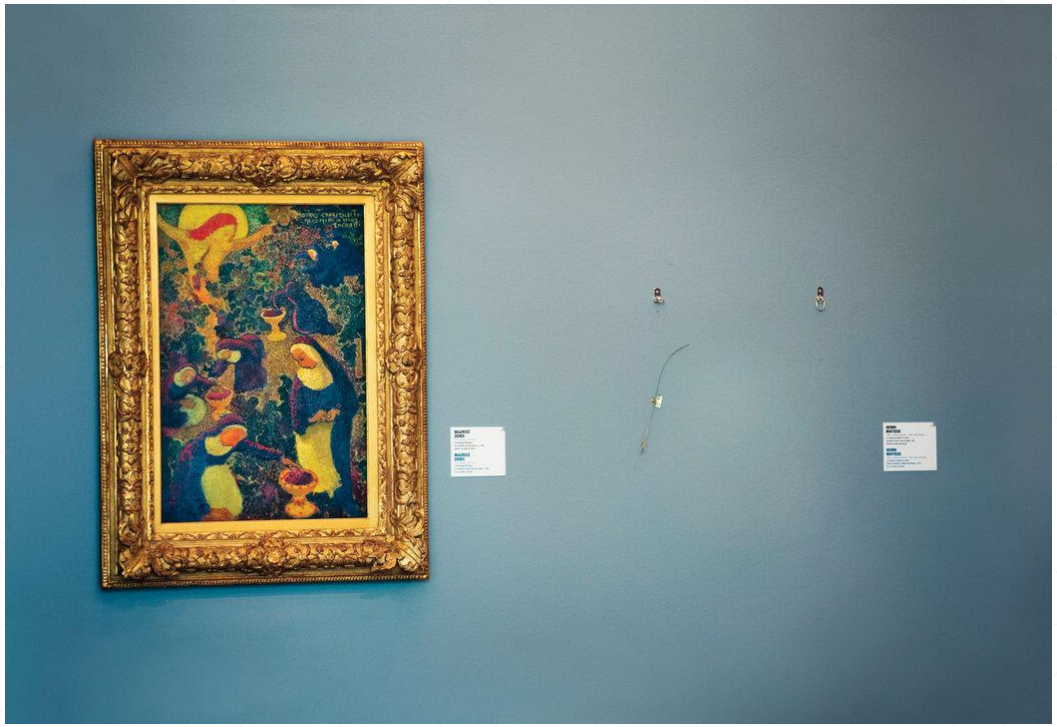
2)

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

Evitare commenti "a perdere"

Evitare commenti destinati a perdersi o a rimanere non aggiornati

- Quando si fa refactoring, il codice viene spostato e il commento può rimanere "orfano"
- Difficilmente vengono manutenuti dagli sviluppatori: è troppo oneroso



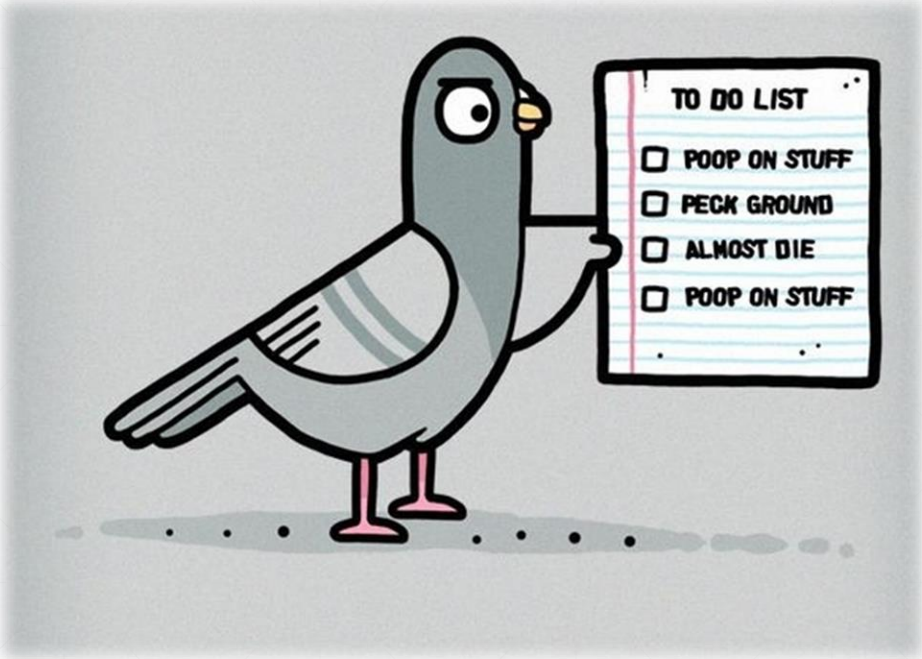
Non commentare il codice!

Non commentare il codice sorgente stesso.

Ma non ce l'hai un buon sistema di Source Control? 🤔

Attento ai TODO

Poni attenzione all'uso dei TODO.



Usa i TODO con parsimonia, ma non abusarne.

Ricordati che prima o poi devi risolverli. 😉

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.

Usa il comando "Format Code" (Ctrl+D) in Delphi.

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*)

Strutture dati

Data Abstraction

- «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 Object

- 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

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

Strumenti

Misurare il grado di tossicità

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

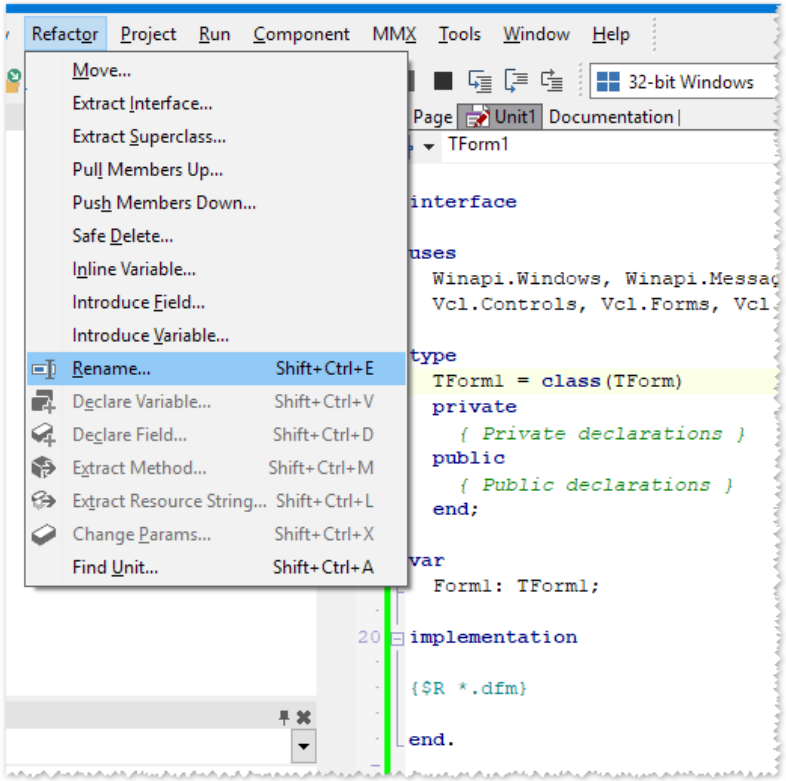
Method Name	Length	Parameters	If Depth	Cyclomatic Complexity	Toxicity
TUserValidator.CheckPassword	10	2	2	3	0,433
CalculatePay	5	1	0	4	0,271
GetThem	7	0	1	3	0,263
GetArrayOfEvenNumbers	7	0	1	3	0,263
CalculatePay	5	1	1	2	0,213
UnregisterCalculator	2	1	1	2	0,200
TUserValidat Login	3	2	0	1	0,163
RegisterCalcu	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"

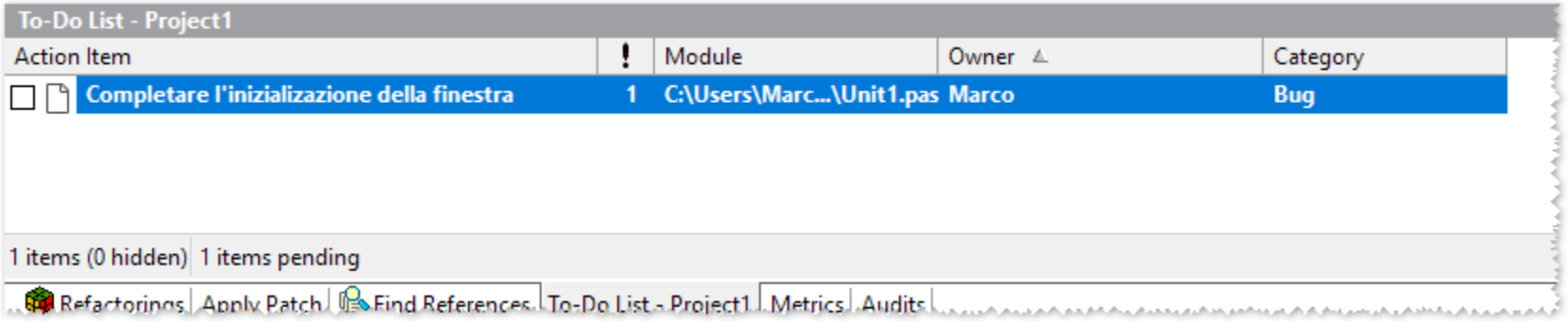
Refactoring

Delphi include i tool necessari per il Refactoring del codice sorgente.



To-Do List

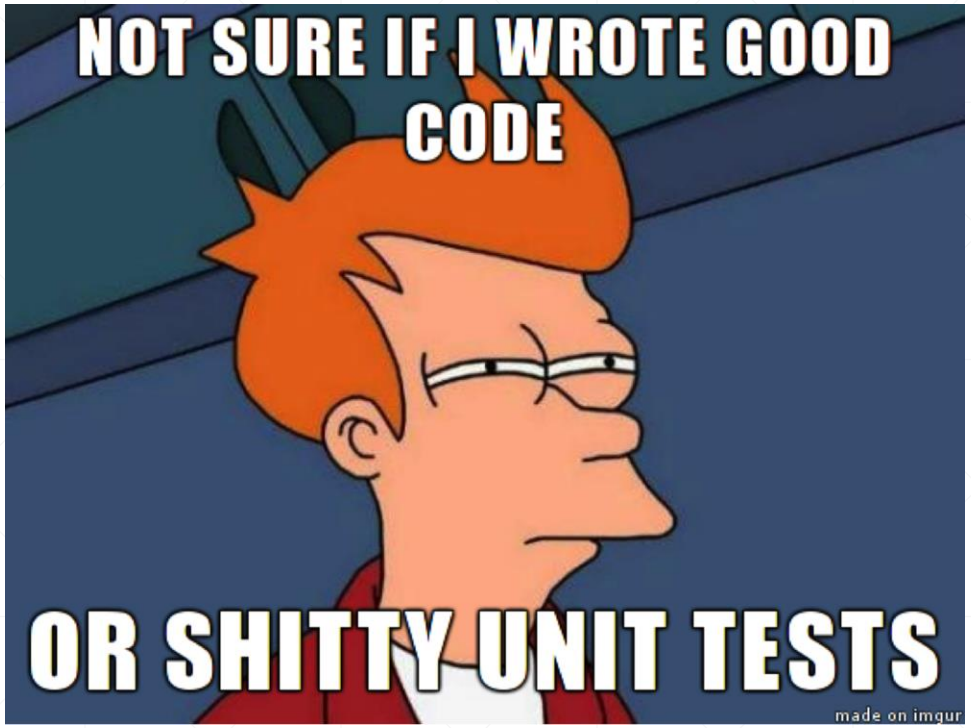
Utilizzare la lista dei To-Do per contrassegnare le parti di codice da completare.



Corollari

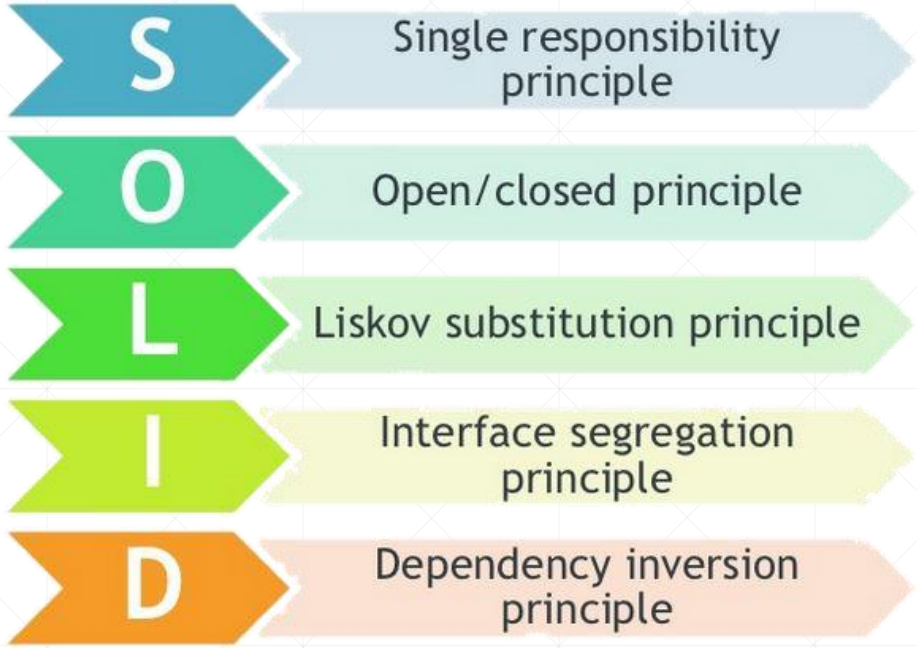
Unit & Integration Testing

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



I principi SOLID

I principi SOLID sono linee guida per lo sviluppo di software **estendibile** e **manutenibile**.



Il «bello» della OOP

Premesse

Incapsulazione

Nascondere i dettagli implementativi di una classe

Incapsulazione

Information Hiding:
il caso più semplice

```
type
  TPerson = class
  private
    FFirstName: string;
    FLastName: string;
  public
    property FirstName: string read FFirstName write FFirstName;
    property LastName: string read FLastName write FLastName;
  end;
```

Incapsulazione

Implementation Hiding:
più complesso da gestire

```
type
  TCustomers = class
  private
    FList: IList<TCustomer>
  public
    constructor Create; // will create the list.....
    property CustomerList: IList<TCustomer> read FList;
  end;
```

```
type
  TCustomers = class
  private
   FList: IList<TCustomer>;
   function GetCustomerList: IEnumerable<TCustomer>;
  public
    constructor Create; // list will be created here
    procedure AddCustomer(aCustomer: TCustomer);
    property Customers: IEnumerable<TCustomer> read GetCustomerList;
  end;
```

Incapsulazione

Gestire nel modo corretto i riferimenti agli oggetti usati internamente

- Non restituire **nil**
- Favorire la "lazy initialization"

```
function GetListOfWidgets: TWidgetCollection;
begin
  if Widgets.Count = 0 then
  begin
    Result := nil
  end else
  begin
    Result := Widgets; // Of type TWidgetCollection
  end;
end;
```

Incapsulazione

Guard Pattern:
verifica sempre la validità dei riferimenti

```
constructor TWidgetProcessor.Create(aWidgetVerifier: TWidgetVerifier);
begin
  if aWidgetVerifier = nil then
  begin
    raise ENilParameterException.Create('Cannot pass a nil parameter');
  end;
  inherited Create;
  FWidgetVerifier := aWidgetVerifier;
end;
```

Coupling

La misura in cui il tuo codice dipende da altri moduli, e viceversa

Coupling elevato

Un coupling elevato è male!

- I moduli non sono facilmente separabili
- I moduli non sono facilmente sostituibili
- Le implementazioni sono intersecate e mescolate tra loro
- Una modifica in qualsiasi punto di un modulo può produrre effetti devastanti in un altro
- Le relazioni tra i moduli sono meno comprensibili

Tuttavia, affinché il software funzioni, vi saranno senz'altro moduli in contratto tra loro *mai* non potranno esistere.

Loose coupling

Scrivi codice con un basso accoppiamento.

Esso sarà

- poco complesso da leggere
- facile da comprendere
- riusabile, perché connesso solo da strati sottili (es. interfacce)
- più manutenibile (perché le modifiche sono isolate)
- testabile!

Coesione

Il grado con cui gli elementi di un modulo possono stare assieme

Coesione

Una **elevata coesione** è bene.

Si raggiunge quando le singole parti del sistema, sebbene con caratteristiche e funzionalità eterogenee tra loro, collaborano realizzando qualcosa di utile (ad alto valore per il cliente o per lo sviluppatore).

NON deve essere ottenuta a discapito di un maggiore accoppiamento.

Pattern

Command/Query Principle

e suoi derivati

Separare ciò che "legge" i dati in un sistema da ciò che "scrive" i dati

Command/Query Principle

Letture e scritture devono essere separate!

- Si ottiene separando in classi differenti ciò che legge dati da ciò che scrive
- La lettura e la scrittura dei dati non devono avvenire nello stesso momento

Command/Query Principle

Query

E' l'operazione che restituisce lo stato di un sistema o di un oggetto.

- Non deve modificare lo stato dell'oggetto quando viene eseguita
- Eseguendola più volte sugli stessi oggetti deve dare lo stesso risultato
- Il suo risultato deve poter essere sostituito da entità che lo rappresentano
- Non possono invocare comandi

Command/Query Principle

Command

E' l'operazione che produce un "effetto collaterale" (*side effect*) nel sistema.

- Normalmente (in Delphi) è una procedura
- Non restituisce un valore di ritorno
- Non deve avere parametri di output
- Può richiamare delle query (*)

(*) In genere, non è bene mescolare Query e Command: si rischia di rompere il *Single Responsibility Principle* (SRP).

Command/Query Principle

Esempio

```
procedure ProcessWidgets(
    aCollectionOfWidgets: TWidgetCollection;
    out aNumberOfProcessedWidgets: Integer);
```

```
procedure ProcessWidgets(aCollectionOfWidgets: TWidgetCollection);

function CurrentProcessedWidgetCount: Integer;
```

Postel's Law

«Be conservative in what you do, be liberal in what you accept from others».

aka "Robustness Principle"

Postel's Law

Scritto da Jon Postel nella specifica iniziale del protocollo TCP.

Un altro modo di enunciarlo:

«Be conservative in what you send; be liberal in what you accept».

Postel's Law

Send

Quando invii o trasmetti dati

- sii rigido e restrittivo
- rispetta le specifiche e i tipi di dati
- sii accurato e preciso
- usa specifiche chiare e ben definite

Postel's Law

Receive

Quando ricevi dati

- sii tollerante e aperto
- accetta quanti più tipi possibile
- se puoi capire la richiesta, fallo
- applica meno eccezioni possibile (*)

(*) quelle fondamentali per la sicurezza non sono negoziabili!

Composition over Inheritance

Quale delle due si deve preferire?

Composition over Inheritance

Inheritance

- Incoraggia la creazione di innumerevoli classi discendenti
- Non consente un controllo completo su ciò che la classe base può (o potrà fare)
- Una volta messa in opera, rende più difficili le modifiche
- Non puoi modificare una classe base con l'assoluta certezza di non "rompere" una classe discendente

Composition over Inheritance

Composition

- Supporta meglio la scalabilità del sistema
- Consente di sfruttare "Dependency Injection" e altri paradigmi
- Fornisce una maggiore flessibilità
- Aiuta il "decoupling" e agevola la manutenibilità

E' preferibile all'ereditarietà (ma non vuol dire che vada usata sempre e comunque).

Principi SOLID

Codice solido

C'è codice solido… e codice SOLID.

I principi SOLID

Single Responsibility Principle (SRP)

Open/Closed Principle (OCP)

Liskov's Substitution Principle (LSP)

Interface Segregation Principle (ISP)

Dependency Inversion Principle (DIP)

Code & Design Smells

- Codice troppo difficile da modificare
- Codice "facile da rompere" alla prima variazione
- Codice non riutilizzabile in altre situazioni
- Difficoltà nel far fare al codice il proprio compito
- Codice inutilmente complicato per lo scopo

Single Responsibility Principle (SRP)

Ogni classe deve svolgere uno e un solo compito

Single Responsibility Principle

```
type
  TBook = class
  private
    FCurrentPage: integer;
    FTitle: string;
    FAuthor: string;
    procedure SetTitle(const Value: string);
    procedure SetAuthor(const Value: string);
  public
    procedure DisplayPage;         // Book Stuff
    function TurnPage: integer;    // Book stuff
    procedure PrintCurrentPage;    // Prints itself
    procedure SaveCurrentPage;     // Saves itself
    property Title: string read FTitle write SetTitle;
    property Author: string read FAuthor write SetAuthor;
  end;
```

Single Responsibility Principle

```
type
  TBookManager = class
  private
    FCurrentPage: integer;
    FTitle: string;
    FAuthor: string;
    procedure SetTitle(const Value: string);
    procedure SetAuthor(const Value: string);
  public
    function TurnPage: integer;
    procedure PrintCurrentPage(aPage: IPrintable);
    procedure SavePageNumber(aPageNumberSaver: IPageNumberSaver);
    property Title: string read FTitle write SetTitle;
    property Author: string read FAuthor write SetAuthor;
  end;
```

Single Responsibility Principle

```
type
  IPrintable = interface
    procedure Print(aString: string);
  end;

  TConsolePrinter = class(TInterfacedObject, IPrintable)
    procedure Print(aString: string);
  end;

  IPageNumberSaver = interface
    procedure Save(aPageNumber: integer);
  end;
```

Open/Closed Principle (OCP)

Ogni classe deve essere aperta a estensioni ma chiusa al cambiamento

Open/Closed Principle (OCP)

```
{ TRectangle }

  TRectangle = class
  private
    FHeight: Integer;
    FWidth: Integer;
  public
    property Height: Integer read FHeight write FHeight;
    property Width: Integer read FWidth write FWidth;
  end;
```

```
{ TAreaCalculator }

function TAreaCalculator.Area(
  ARectangles: array of TRectangle): Integer;
var
  R: TRectangle;
begin
  Result := 0;
  for R in ARectangles do
    Result := Result + (R.Width * R.Height);
end;
```

Open/Closed Principle (OCP)

```
{ TCircle }

  TCircle = class
  private
    FRadius: Integer;
  public
    property Radius: Integer
      read FRadius write FRadius;
  end;
```

```
{ TAreaCalculator }

function TAreaCalculator.Area(AShapes: array of TObject): Integer;
var
  S: TObject;
  R: TRectangle;
  C: TCircle;
begin
  Result := 0;
  for S in AShapes do
  begin
    if S is TRectangle then
    begin
      R := TRectangle(S);
      Result := Result + (R.Width * R.Height);
    end;
    if S is TCircle then
    begin
      C := TCircle(S);
      Result := Result + Round(C.Radius * C.Radius * Pi());
    end;
  end;
end;
```

Open/Closed Principle (OCP)

```
{ TShape }

  TShape = class abstract
    public function Area: Integer; virtual; abstract;
  end;
```

Open/Closed Principle (OCP)

```
type

{ TRectangleShape }

  TRectangleShape = class (TShape)
  private
    FHeight: Integer;
    FWidth: Integer;
  public
    public function Area: Integer; override;
    property Height: Integer read FHeight write FHeight;
    property Width: Integer read FWidth write FWidth;
  end;

implementation

{ TRectangleShape }

function TRectangleShape.Area: Integer;
begin
  Result := Width * Height;
end;
```

```
type

{ TCircleShape }

  TCircleShape = class (TShape)
  private
    FRadius: Integer;
  public
    public function Area: Integer; override;
    property Radius: Integer read FRadius write FRadius;
  end;

implementation

{ TCircleShape }

function TCircleShape.Area: Integer;
begin
  Result := Round(Radius * Radius * Pi());
end;
```

Open/Closed Principle (OCP)

```
type

{ TShapeAreaCalculator }

  TShapeAreaCalculator = class
  public
    function Area(AShapes: array of TShape): Integer;
  end;

implementation

{ TShapeAreaCalculator }

function TShapeAreaCalculator.Area(AShapes: array of TShape): Integer;
var
  S: TShape;
begin
  Result := 0;
  for S in AShapes do
    Result := Result + S.Area;
end;
```

Liskov's Substitution Principle (LSP)

"Let q(X) be a property provable about objects X of type T. Then q(y) should be provable for objects Y of type S where S is a subtype of T"

Liskov's Substitution Principle (LSP)

«I sottotipi dovrebbero essere sostituibili ai loro tipi di base»

Parafrasando...

oppure, per dirla ancora più semplice:

«Un client dovrebbe consumare qualsiasi implementazione di una interfaccia senza modificare la correttezza del sistema»

Liskov's Substitution Principle (LSP)

```
interface

type

{ TAnimal }

  TAnimal = class
  public
    procedure MakeNoise(); virtual;
  end;

implementation

{ TAnimal }

procedure TAnimal.MakeNoise;
begin
  Writeln('I am making noise');
end;

end.
```

```
interface

type

{ TDog }

  TDog = class (TAnimal)
  public
    procedure MakeNoise(); override;
  end;

implementation

{ TDog }

procedure TDog.MakeNoise;
begin
  inherited;
  Writeln('Bau bau!');
end;

end.
```

Liskov's Substitution Principle (LSP)

```
interface

type

{ TCat }

  TCat = class (TAnimal)
  public
    procedure MakeNoise(); override;
  end;

implementation

{ TCat }

procedure TCat.MakeNoise;
begin
  inherited;
  Writeln('Meeeeow!');
end;

end.
```

```
interface

type

{ TFish }

  TFish = class(TAnimal)
  public
    procedure MakeNoise(); override;
  end;

implementation

{ TFish }

procedure TFish.MakeNoise;
begin
  inherited;
  raise Exception.Create('I cannot make noise');
end;

end.
```

Interface Segregation Principle (ISP)

I client non devono essere forzati a dipendere da metodi che non utilizzano

Interface Segregation Principle (ISP)

```
{ IClickListener }

  IClickListener = interface
    procedure OnClick;
    procedure OnDoubleClick;
    procedure OnLongClick;
  end;
```

```
{ TSimpleClickListener }

  TSimpleClickListener = class(TInterfacedObject, IClickListener)
  public
    procedure OnClick;
    procedure OnDoubleClick;
    procedure OnLongClick;
  end;


procedure TSimpleClickListener.OnClick;
begin
  Writeln('Clicked!');
end;

procedure TSimpleClickListener.OnDoubleClick;
begin
  // ignored
end;

procedure TSimpleClickListener.OnLongClick;
begin
  // ignored
end;
```

Dependency Inversion Principle (DIP)

Occorre sempre dipendere da una interfaccia e non da una implementazione

Dependency Inversion Principle (DIP)

Parafrasando...

- Scrivere codice basandosi su astrazioni, ossia **interfacce**, non su implementazioni
- I moduli di alto livello non devono dipendere da quelli di basso livello
- Le astrazioni non devono dipendere dai dettagli, e i dettagli non devono dipendere da astrazioni

Conclusioni

Clean Code

- La qualità del codice che scriviamo ci rappresenta a livello professionale
- Applicare le pratiche del "Clean Code" aiuta a risparmiare tempo e denaro
- Scrivere codice, all'inizio di un progetto ma soprattutto in seguito, deve diventare un'attività piacevole
- Il codice pulito apre le porte a innumerevoli opportunità
  - Semplificazione di Unit/Integration Testing
  - Estensione delle soluzioni di deploy e abbraccio delle architetture (Cloud, Microservices, ecc.)
  - Applicazione di Design Pattern e metodologie
  - Aiuto nell'apprendimento di nuovi linguaggi e tecnologie

Il futuro te stesso ti ringrazierà. 😉

Principi SOLID

Scrivere codice SOLID non è così difficile come sembra.

- E' sufficiente vedere i problemi da una differente prospettiva nella stesura del codice sorgente
- Il risultato sarà codice facile da mantenere e ben scritto
- Ottenere molte classi e interfacce è un effetto collaterale, ma non costituisce mai un reale problema, anzi…
- Classi più piccole e mirate facilitano i test, sono più facili da riutilizzare e si possono combinare per creare sistemi complessi

Non male, no? 😉

Domande?

Grazie!