



Disaccoppiamo il nostro codice

Mettiamo mano allo “spaghetti-code”



# Omar Bossoni

**BCI Solutions Srls**

**Delphi Senior developer since  
1995**

**for Windows, MacOS  
and mobile applications.**

**Expert in porting code  
to the latest version of Delphi.  
Develope native applications for  
iOS**



**19 Novembre 2025  
Padova**





# Omar Bossoni

**BCI Solutions Srls**

@ info@bcisw.com

S omarbossoni

✈ omarbossoni

in linkedin.com/in/omar-bossoni-b0800b123



19 Novembre 2025  
Padova



# Delphi Force



Omar Bossoni



Fabio Codebue



Maurizio del Magno



Marco Mottadelli



Antonio Polito



Carlo Narcisi



Thomas Ranzetti



19 Novembre 2025  
Padova





# AGENDA

---

1. Introduzione
2. Disaccoppiamento in Delphi
3. Dependency Injection
4. TMessaging: Event bus modulare
5. Disaccoppiare con interfacce e factory
6. Q & A

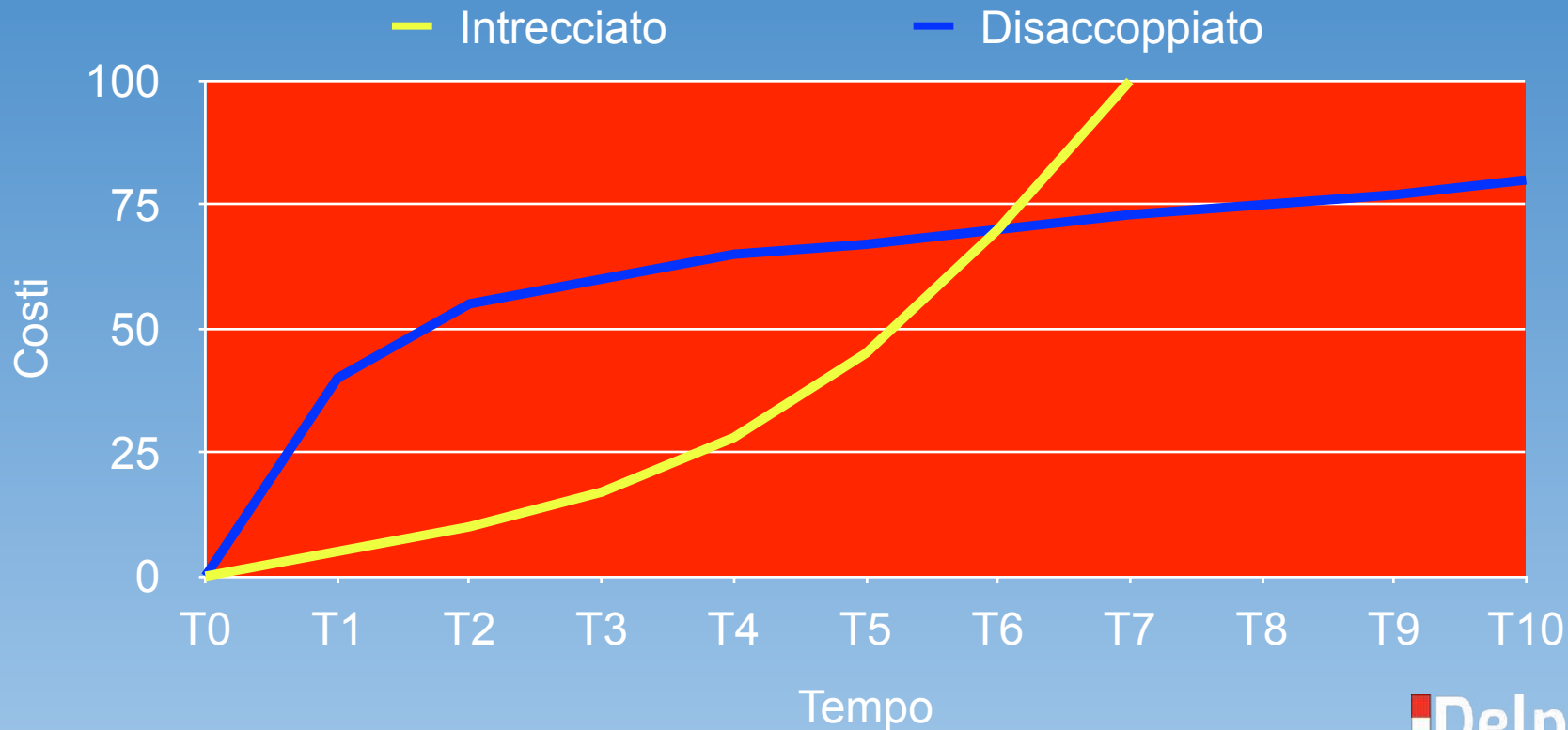


# Introduzione

1



# Introduzione





# Codice intrecciato

---

## La velocità che si paga dopo

Scrivere codice “intrecciato”, con molte dipendenze incrociate, spesso nasce da un obiettivo legittimo:

*“Consegniamo in fretta, poi sistemiamo.”*



# Codice intrecciato

---

All'inizio, infatti:

- colleghi direttamente moduli tra loro;
- accedi ai dati dove capita;
- riusi pezzi di logica copiandoli e adattandoli;
- non ti preoccupi troppo di separazione dei ruoli e responsabilità.

Effetto immediato...



# Codice intrecciato

All'inizio...

- co
- ad
- ri
- no
- re

*Tempi di sviluppo iniziali bassissimi:  
il prodotto esce in fretta, il cliente è contento,  
il business vede risultati.*

Sembra una vittoria:  
*“Funziona, l’abbiamo messa giù in due giorni.”*

oli e

Effetto immediato...



# Codice intrecciato

---

Ma cosa succede dopo col passare dei mesi?

- Ogni nuova funzionalità deve toccare **punti diversi** del sistema;
- Ogni modifica rischia di **rompere qualcos'altro** perché tutto è collegato a tutto;
- Il codice è difficile da testare in modo isolato, quindi:
  - Più bug
  - Più regressioni
  - Più tempo speso a capire “cosa è successo”



Risultato:

Il **tempo per fare una modifica** cresce in modo non lineare.

Quello che all'inizio richiedeva 2 ore dopo un anno può richiederne 2 giorni, poi 2 settimane..

Arrivi quindi a un punto in cui il costo di manutenzione supera ampiamente i ricavi generati dal prodotto o il valore che il cliente è disposto a pagare.

È la classica situazione in cui si sente dire:

*“Questo software è diventato ingestibile,  
conviene riscriverlo da zero.”*



# Codice disaccoppiato

---

**Investire subito per risparmiare dopo**

All'estremo opposto c'è l'approccio **ben progettato**:

- moduli separati,
- responsabilità chiare,
- interfacce definite,
- dipendenze ridotte e controllate.



# Codice disaccoppiato

---

All'inizio:

- ci metti di più a scrivere:
  - definire le interfacce;
  - separare logica di business, infrastruttura, UI, accesso dati;
  - evitare “scorciatoie”
  - scrivere test dove ha senso.
- Il cliente (o il capo) può avere la percezione:.  
*“Ma ci state mettendo più tempo rispetto a farlo al volo...”*
- l'avvio è più lento e più oneroso.



# Codice disaccoppiato

---

Ma cosa succede quando il sistema cresce?

- Aggiungere una funzionalità nuova spesso significa toccare 1–2 moduli ben definiti senza dover capire “tutto il sistema”;
- Modificare un comportamento è più semplice, perché la logica è localizzata, le dipendenze sono note, il resto del codice non esplode a catena.
- I test aiutano a capire subito se hai rotto qualcosa.



# Codice disaccoppiato

---

Risultato:

Il **tempo per fare una modifica** rimane più o meno stabile nel tempo, e spesso si riduce perché conosci meglio l'architettura, hai riuso sano (non copia/incolla), puoi intervenire a blocchi.

Da un punto di vista economico:

- Il costo di manutenzione rimane sostenibile.
- Il prodotto può vivere anni, essere esteso, integrato, adattato;
- Il margine tra **costi** e **ricavi** resta positivo nel lungo periodo.



# Conclusione

---

Scrivere codice intrecciato è come correre col fiato corto parti fortissimo, ma ti fermi dopo pochi metri.

Scrivere codice disaccoppiato è come impostare un passo di corsa regolare parti più piano, ma arrivi molto più lontano, con meno fatica.



# Conclusione

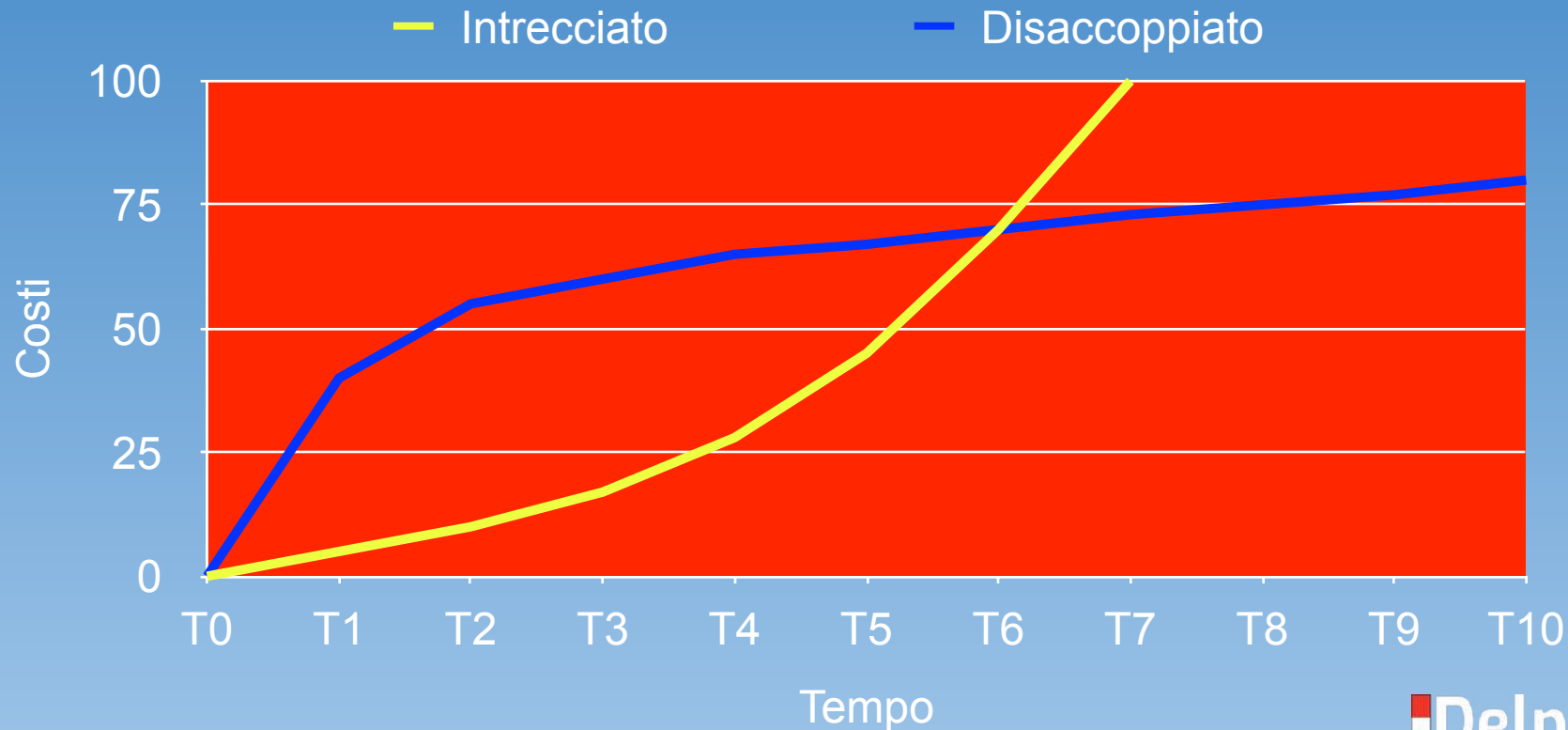
---

In sviluppo software, quindi la **velocità vera** non è quella della prima release, ma la capacità di continuare a cambiare il sistema rapidamente e senza dolore.

Investire in buona progettazione, disaccoppiamento e regole sane di programmazione è, in sostanza, una **scelta economica**, prima ancora che tecnica paghi un po' di più oggi per non dover pagare carissimo domani.



# Conclusione





# Che cosa vedremo oggi

---

Mostreremo come portare Delphi verso un'architettura più moderna, modulare e facilmente manutenibile attraverso tecniche di disaccoppiamento.

Approfondiremo la Dependency Injection con gli strumenti nativi del linguaggio, così da separare logica, servizi e infrastrutture senza introdurre librerie esterne.

Analizzeremo TMessaging come event bus interno per ridurre le dipendenze tra form e unit, rendendo la comunicazione più flessibile e scalabile.

Infine, vedremo come usare interfacce e factory per creare contratti chiari e implementazioni sostituibili, migliorando testabilità, estensibilità e qualità complessiva del codice Delphi.



# Disaccoppiamento in Delphi

# 2



# Cos'è il “disaccoppiamento” nel software?

Il disaccoppiamento è la capacità di un sistema di funzionare **in modo indipendente**, senza che le sue parti siano rigidamente legate le une alle altre.

In altre parole, significa progettare il codice affinché:

- una modifica in una unit non obblighi a cambiare tutto il resto;
- i componenti possano essere riutilizzati in contesti diversi;
- sia possibile sostituire una parte dell'applicazione senza riscrivere tutto;
- il software sia **testabile**, perché i componenti possono essere isolati.



# Cos'è il “disaccoppiamento” nel software?

In Delphi questo è particolarmente rilevante perché, per tradizione, si tende a scrivere applicazioni dove:

- le form conoscono tutto;
- si accede direttamente a controlli altrui;
- si istanziano classi concrete ovunque, in un ecosistema basato sul *drag&drop* e sul *rad-sviluppo rapido*.



# Perché l'accoppiamento è un problema?

## **Rigidità**

- se cambi una unit, devi toccare file, form o unit che non dovrebbero essere correlate;
- i uses esplodono: centinaia di unit incluse per usare una sola classe.

## **Fragilità**

- un piccolo cambiamento può rompere parti inaspettate del codice;
- aumentano gli errori casuali, specialmente nelle UI di grandi dimensioni.



# Perché l'accoppiamento è un problema?

## Scarsa testabilità

- Unit test e mock diventano difficili se i componenti si creano da soli:

```
FService := TMyService.Create;  
// Test impossibile → implementazione hardcoded
```

## Riutilizzo limitato

- Le classi diventano monolitiche, non si possono riusare in contesti diversi (VCL, FMX, servizi, console, plugin)



# Tipi di accoppiamento (e perché evitarli)

## **Accoppiamento strutturale**

È il più evidente: due classi si conoscono direttamente

- Dipendenza hardcoded da classi concrete
- Uso eccessivo del `uses` nella *interface*

`Uses`

`FormMain,`

`DataModule1;`



# Tipi di accoppiamento (e perché evitarli)

## Accoppiamento logico

Un modulo sa **troppo** sul funzionamento interno di un altro.

```
Form2.Edit1.Text := 'Ciao';
```

La UI esterna usa dettagli interni (controlli) di un'altra form → errore comune.



# Tipi di accoppiamento (e perché evitarli)

## **Accoppiamento temporale**

Quando due componenti devono essere inizializzati o chiamati in un certo ordine obbligatorio.

- devi creare il DataModule *prima* di tutto
- devi caricare configurazioni *prima* di creare certi form

Con architetture accoppiate l'ordine è rigido e fragile.



# Tipi di accoppiamento (e perché evitarli)

## **Accoppiamento sui dati**

Condividere variabili globali, singleton o record comuni:

```
GlobalSettings.Theme := 'Dark';
```

Questo genera “bottleneck” globali difficili da tracciare e testare.

... dipende dal contesto



## Perché Delphi tende naturalmente all'accoppiamento?

Non è un difetto del linguaggio, ma del suo stile classico RAD

### **Form auto-create**

L'Application crea automaticamente tutte le form, quindi tutto è accessibile ovunque

### **Accesso diretto ai controlli**

Gli elementi della UI sono accessibili come variabili pubbliche. È comodo... ma pericoloso.



# Perché Delphi tende naturalmente all'accoppiamento?

## **Drag&Drop impone dipendenze implicite**

- DataSource lascia dipendenze non visibili
- Timer, ActionList, ImageList su form che dovrebbero essere “logiche”

```
DBGrid1.DataSource = DataSource1
```

```
DataSource1.DataSet = Table1
```



Pe

Quando lavori in Delphi con l'approccio RAD, trascinare componenti sulla Form è comodissimo... ma introduce **dipendenze invisibili** nel codice.

Esempio:

- Trascini un TTable, un TDataSource e un DBGrid sulla Form;
- Imposti le proprietà di collegamento tra di loro;

### Problema

La Form non è più solo una UI ("view"), è anche:

- UI
- Business Logic
- Data Access

È **accoppiamento violento**, ma invisibile nel codice.

Non vedi un "uses" verso il DB, ma in realtà c'è eccome.

mento?

Drag

• D

• T

es

I

I



## Perché Delphi tende naturalmente all'accoppiamento?

### **Codice scritto dentro l'interfaccia della form**

Spesso la logica applicativa è scritta direttamente nel form:

```
procedure TFormMain.btnSaveClick(Sender: TObject);  
begin  
    SaveToDatabase(Edit1.Text); // logica dentro la UI  
end;
```

Questo accoppia **UI** e **business logic**, rendendo impossibile testare o riusare la logica altrove.



# Principi per ridurre l'accoppiamento

## **DIP – Dependency Inversion Principle**

Il modulo ad alto livello non deve dipendere da quello a basso livello.

Entrambi devono dipendere da **astrazioni**, non da implementazioni.

In pratica:

- usare interfacce;
- usare factory o dependency injection;



# Principi per ridurre l'accoppiamento

## **Separation of Concerns (SoC)**

Ogni modulo deve fare una sola cosa.

- La UI mostra dati;
- I servizi li elaborano;
- I repository li gestiscono;

Niente form che accede al database direttamente.



# Decomporre il codice Delphi moderno

Per ottenere un'app disaccoppiata dobbiamo suddividere i livelli:

- **UI** (Form, controlli);
- **Presenter / ViewModel** (connessione tra logica e UI);
- **Servizi** (elaborazione logica);
- **Repository / Data Access** (DB, file, API);
- **Modello dati** (classi POCO o record);



# Design Patterns Moderno

Per  
livelli

- U
- P
- S
- R
- M

Le classi **POCO** (Plane Old Class Object) sono semplicemente classi “semplici”, senza dipendenze, senza logica interna complessa e senza legami con framework o componenti esterni.

```
type
  TPerson = class
  private
    FAge: Integer;
    FName: string;
  public
    property Name: string read FName write FName;
    property Age: Integer read FAge write FAge;
  end;
```

ridere i

e UI);



# Decomporre il codice Delphi moderno

Per ottenere un'app disaccoppiata dobbiamo suddividere i livelli:

- **UI** (Form, controlli);
- **Presenter / ViewModel** (connessione tra logica e UI);
- **Servizi** (elaborazione logica);
- **Repository / Data Access** (DB, file, API);
- **Modello dati** (classi POCO o record);
- **Messaging / Event bus** (comunicazione interna)



# Decomporre il codice Delphi moderno

Questa separazione permette di:

- Sostituire parti dell'app senza toccare le altre;
- Testare i servizi senza UI;
- Riutilizzare logiche in progetti multiplatforma (FMX, Console, servizio Windows);



## Gli strumenti Delphi più utili per disaccoppiare

- **TMessaging** - Event bus interno a Delphi, potente, leggero, perfetto per separare UI e logica;
- **Interfacce e polimorfismo** - La base per DIP e architetture pulite;
- **Dependency Injection**;
- **Pattern architetturali** (MVP / MVVM / ecc..)



# In conclusione

---

Con codice disaccoppiato ottieni:

- Meno dipendenze, meno bug;
- Maggiore stabilità nel tempo;
- Possibilità di unit test;
- Riduzione dello “Spaghetti Code”;
- Codice riutilizzabile;
- Possibilità di cambiare tecnologia senza riscrivere tutto (es. VCL → FMX);

**E soprattutto...**



# In conclusione

Con codice disaccoppiato ottieni:

- Meno dipendenze, meno bug;
- Maggiore stabilità nel tempo;
- Possibilità di unit test;
- Riduzione dello “Spaghetti Code”;
- Codice riutilizzabile;
- Possibilità di cambiare tecnologia senza riscrivere tutto (es. VCL → FMX);

**E soprattutto...**



**le modifiche  
non fanno più paura!**



# Dependency Injection

3



# Cos'è la Dependency Injection?

In Delphi (come in molti linguaggi), la **Dependency Injection (DI)** è una tecnica per:

- evitare che le classi creino da sole le proprie dipendenze;
- “iniettare” dall'esterno gli oggetti di cui hanno bisogno;
- rendere l'applicazione più modulare, testabile, sostituibile;

È una forma di applicazione pratica del principio SOLID  
**DIP – Dependency Inversion Principle.**



# Cos'è la Dependency Injection?

In Delphi (come in molti linguaggi) la **Dependency**

**Injection (DI)** è una tecnica

- evitare che le classi abbiano troppe dipendenze;
- “iniettare” dalle astrazioni quelle implementazioni che hanno bisogno;
- rendere la logica modulare, testabile, sostituibile;

**“Non dipendere dalle implementazioni.  
Dipendi dalle astrazioni.”**

È una forma di applicazione pratica del principio SOLID  
**DIP – Dependency Inversion Principle.**



# Perché serve in Delphi?

Perché Delphi, se usato in modo tradizionale, favorisce questo schema

```
procedure TFormMain.SaveData;  
var  
    lService: TCustomerService;  
begin  
    lService := TCustomerService.Create; // ✗  
    lService.Save(Edit1.Text); // ✗  
end;
```



# Perché serve in Delphi?

---

Problemi:

- La Form **dipende** dalla classe concreta TCustomerService;
- Non puoi sostituire il servizio (per test, mock, versioni future);
- Non puoi riusare la Form con un altro servizio
- Hai un coupling diretto a implementazione e costruzione



# Perché serve in Delphi?

## Problemi:

- La TC
- No  
ve
- Non puoi riusare la Form con un altro servizio
- Hai un coupling diretto a implementazione e costruzione

*Con la DI, si passa da:*

*UI → concrete implementation*

*a*

*UI → interface → implementation injected from outside*



# Gli strumenti che Delphi offre di serie

## 1. Interfacce

***Il modo più potente  
per disaccoppiare.***



# Gli strumenti che Delphi offre di serie

1. Interfacce
2. Costruttori con parametri



# Gli strumenti che Delphi offre di serie

---

1. Interfacce
2. Costruttori con parametri

***Si possono iniettare dipendenze  
semplicemente con parametri  
nel costruttore.***



# Gli strumenti che Delphi offre di serie

1. Interfacce
2. Costruttori con parametri
3. Factory e Factory methods



# Gli strumenti che Delphi offre di serie

1. Interfacce
2. Costruttori con parametri
3. Factory e Factory methods

***Per centralizzare la creazione di  
oggetti e splittare i livelli.***



# Gli strumenti che Delphi offre di serie

1. Interfacce
2. Costruttori con parametri
3. Factory e Factory methods
4. Isola la logica dalla UI
5. Supporta architetture a plugin



# Gli strumenti che Delphi offre di serie

1. Interfacce
2. Costruttori
3. Factory e Factory methods
4. Isola la logica dalla UI
5. Supporta architetture a plugin

*Per mantenere le  
dipendenze tutte in un  
punto.*



# Gli strumenti che Delphi offre di serie

1. Interfacce
2. Costruttori con parametri
3. Factory e Factory methods
4. Isola la logica dalla UI
5. Supporta architetture a plugin
6. Non serve altro



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI



# Vantaggi pratici

---

## 1. Zero dipendenze concrete nella UI

***La Form non usa più  
classi concrete.***



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI
2. Facilità di test



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI
2. Facilità di test

***Puoi passare un servizio finto:***

```
FormMain := TFormMain.Create(nil,  
    TMockCustomerService.Create);
```



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI
2. Facilità di test
3. Gestione del ciclo di vita più chiara



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI
2. Facilità di test
3. Gestione del ciclo di vita più chiara

***Decidi tu dove vengono  
creati gli oggetti.***



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI
2. Facilità di test
3. Gestione del ciclo di vita più chiara
4. Modularità



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI
2. Facilità di test
3. Gestione del codice
4. Modularità

***Puoi sostituire la classe  
senza cambiare le Form.***



# Vantaggi pratici

---

1. Zero dipendenze concrete nella UI
2. Facilità di test
3. Gestione del ciclo di vita più chiara
4. Modularità
5. Riutilizzabilità



# Quando NON serve la DI?

---

- Per oggetti semplici (stringhe, record...)
- Per classi senza dipendenze
- Per codice temporaneo o isolato



# Quando NON serve la DI?

- Per oggetti semplici (stringhe, record...)
- Per classi
- Per codice

```
procedure TFormMain.FormCreate(Sender: TObject);
var
  lTempId: string;
begin
  // Codice temporaneo usato solo qui
  lTempId := FormatDateTime('yyyymmddhhnnsszzz', Now);

  ShowMessage('ID temporaneo: ' + lTempId);

  // Non ha senso usare DI perché:
  // - nessuna dipendenza
  // - serve solo qui
  // - è logica banale e non riutilizzata
end;
```



# Quando NON serve la DI?

---

- Per oggetti semplici (stringhe, record...)
- Per classi senza dipendenze
- Per codice temporaneo o isolato
- Per logica molto semplice che non richiede sostituibilità



# TMessaging Event bus modulare

# 4



# Cos'è TMessaging?

---

TMessaging è un **Event Bus interno a Delphi**, introdotto in FireMonkey ma utilizzabile ovunque (anche in VCL), progettato per inviare messaggi da un punto all'altro dell'applicazione senza creare dipendenze dirette tra gli oggetti, usando un modello **publish/subscribe**.

In pratica permette una comunicazione **molto disaccoppiata**, ideale per separare moduli, evitare riferimenti tra form, eliminare uses circolari, ridurre codice spaghetti.



# Perché è un “Event Bus”?

---

Un **Event Bus** è un canale centralizzato attraverso il quale i le unit:

- **Pubblicano** messaggi
- **Si iscrivono** a certi messaggi
- **Si disiscrivono** da certi messaggi
- **Ricevono** notifiche senza conoscere il mittente

Zero uses, zero dipendenze, zero chiamate dirette.



# Come funziona un TMessaging

**TMessageManager** è la classe responsabile della gestione dei messaggi dell'applicazione e si occupa dell'instradamento dei messaggi.

La sua proprietà **DefaultManager** restituisce un oggetto che funge da centro notifiche a livello di applicazione ed è ampiamente utilizzato nello sviluppo mobile per ascoltare gli eventi generati dal sistema.



# Come funziona un TMessaging

## Sottoscrizione ad un messaggio

```
TMessagingManager.DefaultManager.SubscribeToMessage(TMessage<String>,  
procedure(const Sender: TObject; const AMessage: TMessage)  
begin  
    ShowMessage(TMessage<String>(AMessage).Value);  
end);
```

Dopo aver sottoscritto un metodo a un tipo di messaggio, ogni volta che viene effettuata una chiamata a **TMessagingManager.SendMessage** con un messaggio del tipo previsto, i metodi sottoscritti vengono chiamati.

```
TMessagingManager.DefaultManager.SendMessage(nil,  
    TMessage<String>.Create('Hello World'));
```



# Come funziona un TMessaging

## Disiscriversi da un messaggio

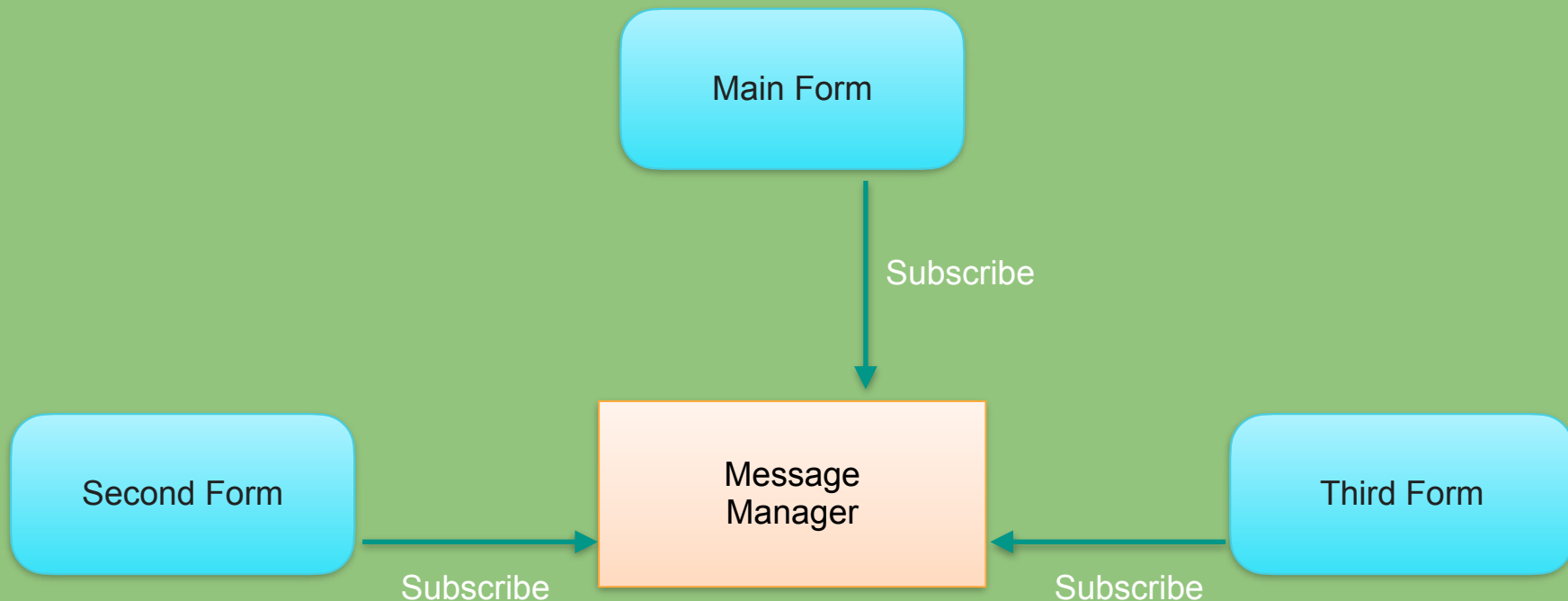
```
FIdMessage :=  
TMessageManager.DefaultManager.SubscribeToMessage(TMessage<String>,  
procedure(const Sender: TObject; const AMessage: TMessage)  
begin  
    ShowMessage(TMessage<String>(AMessage).Value);  
end);
```

Dopo aver sottoscritto un metodo a un tipo di messaggio il `SubscribeToMessage` restituisce un ID; questo ID va utilizzato nel metodo di disiscrizione.

```
TMessageManager.DefaultManager.unsubscribe(TMessage<String>, FIdMessage);
```



# Come funziona un TMessaging





# Come funziona un TMessaging

Main Form

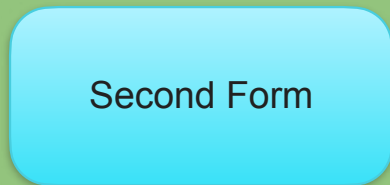
Second Form

Message  
Manager

Third Form



# Come funziona un TMessaging





# Come funziona un TMessaging

Main Form

Second Form



Message  
Manager

Third Form



# Come funziona un TMessaging

Main Form



Second Form

Message  
Manager

Third Form



# Come funziona un TMessaging

Main Form

Second Form

Message  
Manager

Third Form



# Come funziona un TMessaging

Main Form

Second Form

Message  
Manager

Third Form



# Come funziona un TMessaging

Main Form

Second Form

Message  
Manager



Third Form



# Come funziona un TMessaging

Main Form

Second Form

Message  
Manager



Third Form



# Come funziona un TMessaging

Main Form

Second Form

Message  
Manager



Third Form



# Come funziona un TMessaging



Main Form



Second Form

Message  
Manager

Third Form



# Come funziona un TMessaging

Main Form

Second Form

Message  
Manager

Third Form



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli

***Form A non deve  
conoscere Form B***



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli
2. Niente più accesso ai controlli di altre form



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli
2. Niente più accesso ai controlli di altre form

***Non più codice tipo***

*FormB.Edit1.text := ...*



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli
2. Niente più accesso ai controlli di altre form
3. Riduce i uses nella sezione interface



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli
2. Niente più accesso ai controlli di altre form
3. Riduce i uses nella sezione interface
4. Isola la logica dalla UI

***Il flusso diventa***

*UI → invia messaggio → logica → risponde → UI*



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli
2. Niente più accesso ai controlli di altre form
3. Riduce i uses nella sezione interface
4. Isola la logica dalla UI
5. Supporta architetture a plugin



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli
2. Niente più accesso ai controlli di altre form
3. Riduce i uses nella sezione interface
4. Isola la logica dalla UI
5. Supporta architetture a plugin
6. Migliora la testabilità
7. Permette comunicazione one-to-many
8. Funziona anche in thread separati



# Perché aiuta il disaccoppiamento?

1. Elimina riferimenti diretti tra moduli
2. Niente porm
3. Riduce i
4. Isola la l
5. Support
6. Migliora la testabilità
7. Permette comunicazione one-to-many
8. Funziona anche in thread separati

***Ottimo per***  
***Download async***  
***Task paralleli***  
***Operazioni in background***



# Disaccoppiare con Interfacce e Factory

5



# La Factory

---

Le **factory** (fabbriche) sono pattern di progettazione (creazionali GoF) che centralizzano la creazione di oggetti, permettendoti di disaccoppiare il codice che usa gli oggetti da quello che li istanzia.



# La Factory

Le **factory** (fabbriche) sono pattern di progettazione (creazionali GoF) che centralizzano la creazione di oggetti, permettendoti di disaccoppiare il codice che usa gli oggetti da quello che li istanzia.





# La Factory

---

Quando crei oggetti direttamente nel codice con **TClasseA.Create** o **TClasseB.Create**, il tuo codice diventa *fortemente accoppiato* a quelle classi specifiche.

Se domani devi cambiare implementazione, devi modificare tutti i punti dove hai fatto **Create**.

Una factory è sostanzialmente un metodo o una classe che si occupa di creare istanze.



# La Factory

---

Invece di scrivere

```
var
    lSensore: TSensoreTemperatura;
begin
    lSensore := TSensoreTemperatura.Create;
    // usa il sensore
end;
```

Usi una factory

```
var
    lSensore: ISensore;
begin
    lSensore := TSensoreFactory.CreaSensore('temperatura');
    // usa il sensore
end;
```



# Le Factory e le interfacce

---

Le **factory** e le **interfacce** sono una coppia perfetta perché risolvono insieme un problema fondamentale: come creare oggetti senza sapere (e dipendere da) la loro classe concreta.

*Il concetto chiave*

Le interfacce definiscono "**COSA**" fare,  
le factory decidono "**CHI**" lo fa.



# Le Factory e le interfacce

---

Quando dichiarare una variabile come interfaccia, stai dicendo "mi serve qualcosa che sappia fare X, Y, Z" ma non stai specificando quale classe. La factory è quella che decide quale implementazione concreta restituire.

```
var
  ILogger: ILogger;  // so COSA fa, non so CHI è
begin
  ILogger := TLoggerFactory.Crea('file');  // la factory decide CHI
  ILogger.Scrivi('messaggio');  // uso COSA sa fare
end;
```



# Perché funziona così bene in Delphi

## Reference Counting Automatico

Le interfacce in Delphi hanno il reference counting. Quando restituisci un'interfaccia da una factory, non devi preoccuparti di liberare la memoria

```
function TLoggerFactory.Crea(const aTipo: string): ILogger;
begin
    if aTipo = 'file' then
        Result := TFileLogger.Create // l'oggetto si distrugge automaticamente
    else
        Result := TConsoleLogger.Create;
    // nessun Free necessario!
end;
```



# Perché funziona così bene in Delphi

## Nessun cast, nessuna dipendenza

Se restituissi oggetti concreti, dovresti fare cast o conoscere le classi

### MALE - devi conoscere le classi concrete

```
var
  lLogger: TObject;
begin
  lLogger := TLoggerFactory.Crea('file');
  if lLogger is TFileLogger then
    (lLogger as TFileLogger).Scrivi('...')
  else if lLogger is TConsoleLogger then
    (lLogger as TConsoleLogger).Scrivi('...');
  lLogger.Free;
end;
```

### BENE - non conosci le classi concrete

```
var
  lLogger: ILogger;
begin
  lLogger := TLoggerFactory.Crea('file');
  lLogger.Scrivi('...'); // funziona con qualsiasi
  implementazione
end;
```



# Perché funziona così bene in Delphi

## Dependency Injection facilitata

Le factory con interfacce permettono di iniettare dipendenze facilmente:

```
type
  TServizioEmail = class
  private
    FLogger: ILogger;
  public
    constructor Create(const aLogger: ILogger);
    procedure InviaEmail(const aDestinatario, aMessaggio: string);
  end;

var
  lLogger: ILogger;
  lServizio: TServizioEmail;
begin
  lLogger := TLoggerFactory.Crea('database');
  lServizio := TServizioEmail.Create(lLogger);
  // il servizio non sa quale tipo di logger sta usando
end;
```



# Perché funziona così bene in Delphi

## Testing semplificato

Con le interfacce puoi creare una factory di test che restituisce mock

```
{ $IFDEF DEBUG }  
// Nei test restituisce un mock  
lEmailService := TEmailServiceFactory.Crea(TMockLoggerFactory.Crea);  
{ $ELSE }  
// In produzione  
lEmailService := TEmailServiceFactory.Crea(TLoggerFactory.Crea('file'));  
{ $ENDIF }
```



# Perché funziona così bene in Delphi

## Cambio implementazione senza rompere nulla

Passi da file a database per i log

```
/// Prima versione
class function TLoggerFactory.Crea: ILogger;
begin
    Result := TFileLogger.Create('app.log');
end;

// Nuova versione - cambi solo la factory
class function TLoggerFactory.Crea: ILogger;
begin
    Result := TDatabaseLogger.Create(FConnectionString);
end;

// Tutto il codice che usa ILogger continua a funzionare!
```



# Perché funziona così bene in Delphi

```
type
  ISensorRepository = interface
    procedure Salva(const aSensor: TSensorData);
    function Carica(const aID: Integer): TSensorData;
  end;

  TSensorRepositoryFactory = class
  public
    class function Crea: ISensorRepository;
    [...]
  end;

// Implementazione
class function TSensorRepositoryFactory.Crea: ISensorRepository;
begin
  if FUseCache then
    Result := TCachedSensorRepository.Create(FDBConnection)
  else
    Result := TDirectSensorRepository.Create(FDBConnection);
end;
```



# Perché funziona così bene in Delphi

Il punto fondamentale è che

- **Senza interfacce**, le factory restituirebbero TObject o classi base, obbligandoti a fare cast e dipendere dalle classi concrete.
- **Con le interfacce**, le factory possono restituire contratti stabili, permettendo al codice client di essere completamente ignaro delle implementazioni sottostanti.

È il cuore dell'inversione delle dipendenze (Dependency Inversion Principle).



6



THANK YOU