



Domain Driven Design: Overview

Speaker: Giancarlo Sudano



GUISA
Gruppo Utenti Italiani
Solution Architect

About me:



Giancarlo Sudano (*alias janky*)

- Software Architect in Objectway
- Fondatore di GUISA
www.guisa.org
- Blog su Ugidotnet:
<http://blogs.ugidotnet.org/janky>
- Email:
giancarlo.sudano@gmail.com
giancarlo.sudano@objectway.it

2a serie Webcast: "Aspire Architect"

- **Domain Driven Design: Overview**
(23/01/2007 14:30) [[Giancarlo Sudano](#)]
- **UML Reloaded**
(30/01/2007 : 14:30) [[Riccardo Golia](#)]
- **Design Principles**
(06/02/2007 14:30) [[Riccardo Golia](#)]
- **Architecting Layered Applications**
(13/02/2007 : 14:30) [[Giancarlo Sudano](#)]
- **Software Architecture: oltre il design**
(20/02/2007 14:30) [[Lorenzo Barbieri](#)]
- **Service Oriented != Object Oriented**
(22/02/2007 11:00) [[Andrea Saltarello](#)]
- **Software Architecture: soluzioni del mondo reale**
(22/02/2007 14.30) [[Andrea Saltarello](#)]

Agenda

- Domain Driven Design, definizione
- Domain Driven Design, considerazioni
- Domain Model, principi
- Domain Model, esempi
- Domain Model, layering

Domain Driven Design



...tutto cominciò...

- Formalizzazione fatta da Eric Evans 2003/04 nel suo libro "Domain Driven Design: Tackling Complexity In The Heart Of Software"
- Scopo del libro:
 - **"...To describe and build a vocabulary about the very art of domain modeling..."**

Domain Driven Design: Il portale

- <http://domaindrivendesign.org>
 - Informazioni
 - Interviste
 - Articoli
 - Discussioni
 - Case Stories
 - Esempi



The screenshot shows the homepage of the Domain-Driven Design website. At the top, the title "Domain-Driven Design" is displayed in a stylized blue font, with the tagline "INFORMATION, EXCHANGE, DISCUSSION" below it. The site is sponsored by Domain Language, Inc. A navigation menu includes links for Home, Discussion, Books, Practioners, Events, Examples, and About. The main content area is divided into two columns. The left column, titled "What's New", lists recent articles such as "Eric's interview with infoQ on 'Why DDD Matters Today'" and "Domain-Driven Design track at The Spring Experience 2006 Conference!". Below the text is a photograph of a group of people sitting around a table in a meeting. The right column, titled "What's HOT!", features a "Happy New Year!" message with a small Christmas tree icon and a link to a report titled "Strategic Design at StatOil". At the bottom of the right column, there is a section titled "This Site" which describes the website as a non-commercial forum for sharing ideas.

Qualche definizione

- Modello analitico: concettualizzazione del problema di business
- Domain Layer: Strato di software che rappresenta l'implementazione del modello analitico
- Domain Layer patterns: metodologie per implementare i problemi di business
 - Transaction script
 - Table Module
 - **Domain Model**(catalogati da Martin Fowler)

Domain Driven Design: Definizione

La DDD è un **mindset**, cioè una serie di principi e priorità, atte ad accelerare la progettazione software che ha a che fare con domini di particolare complessità.

Le premesse fondamentali sono:

- La maggior parte dei progetti, dovrebbe basarsi su un Dominio, e su una Logica di Dominio.
- La progettazione di Domini complessi dovrebbe basarsi su un modello analitico.

Considerazioni

- Il **modello analitico** (cioè il risultato del lavoro degli analisti) è uno strumento di **sola comprensione**.
- Un analista poi può usare UML per la **visualizzazione** del modello stesso.
- Il modello non porterà nessun dettaglio implementativo ai fini di non inquinare la comprensione.

Quindi

L'**implementazione** di un modello molte volte può allontanarsi notevolmente dalla sua iniziale descrizione.

La Domain Driven Design è una disciplina di progettazione atta quindi a tenere **costantemente vicini/connessi** il modello analitico che il modello implementativo.

One team...one language

- Nella **DDD** l'uso di un **linguaggio comune** tra gli sviluppatori, architetti, analisti ed esperti di dominio è essenziale alla riuscita del progetto
- Questo linguaggio sarà utilizzato oltre che tra le varie figure, anche nei diagrammi, nella documentazione e non per ultimo **nel codice**.
- Il modello analitico è quindi la spina dorsale dell'**ubiquitous language** (cit. *"Use the model as the backbone of a language"*).

Pregi e limiti dell'UML

- Ottimo come rappresentazione del **modello analitico** ma...
- Può diventare poco leggibile al complicarsi del modello stesso

"...The diagram's purpose is to help communicate and explain the model.

The code can serve as a repository of the details of the design. Well-written Java is as expressive as UML in its way..."

Eric Evans

Pregi e limiti dell'UML

- I diagrammi rappresentano una vista del modello analitico. Ma **non sono** il modello analitico.
"Per dirla con un detto zen...se qualcuno vi indica la luna con il dito...non confondete il dito con la luna..." [janky]
- Il codice di un progetto riesce a catturare il modello analitico molto più finemente.
- Certe volte è meglio leggere codice parlante che quintali di documentazione (Ogni buon developer nel suo intimo conosce questa verità! 😊).

Modelliamo in “stile” Domain Model...

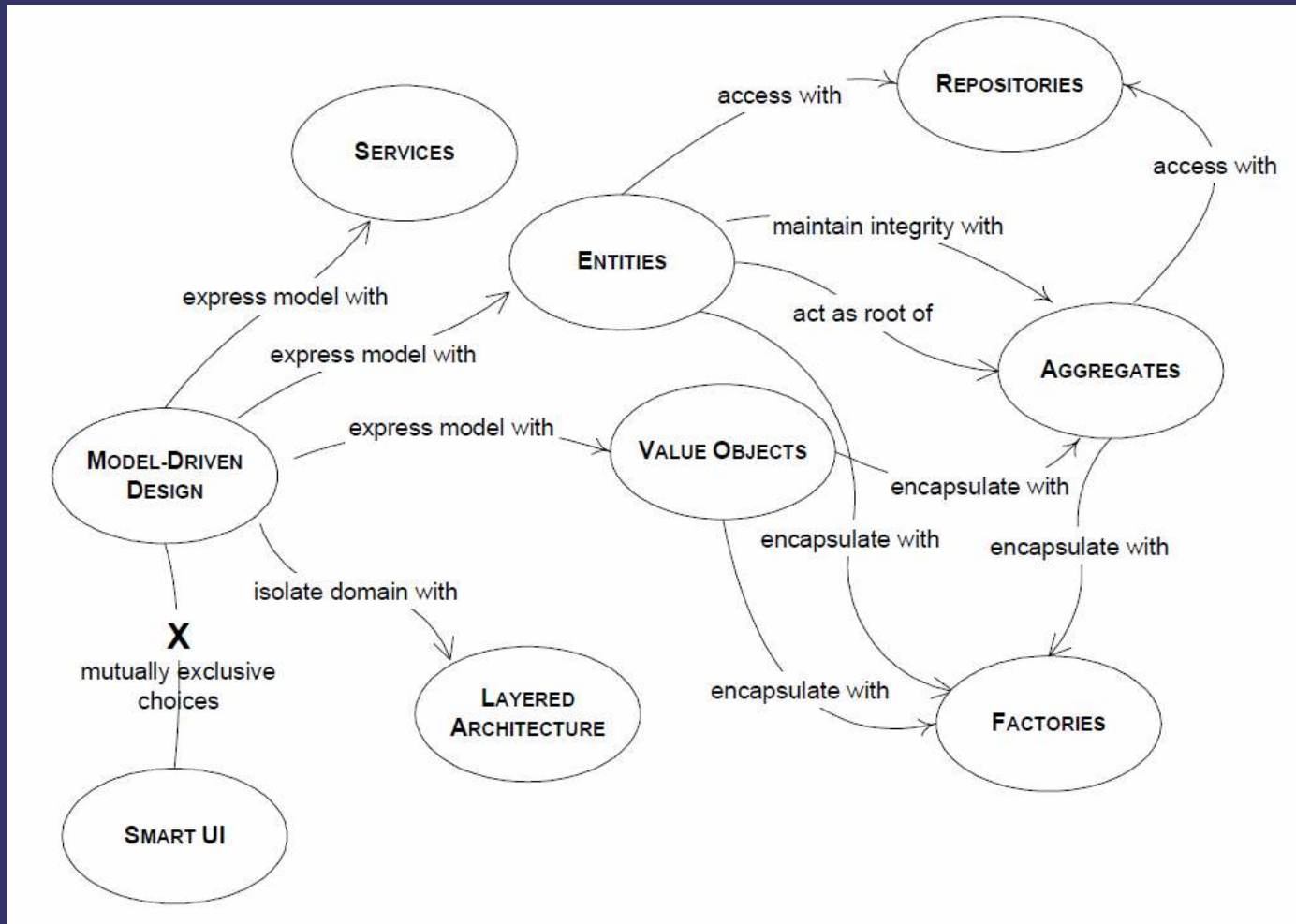


Domain Model: i principi

- Il Domain Model è un **object model** in cui ogni singola classe ha uno specifico significato di business, e può incorporare sia **dati** che **comportamento**.
- Le classi del Domain Model possono avere **associazioni** fino a formare una vera e proprio **grafo** (rete di interconnessioni).
- Le classi del Domain Model possono modellare sia le **Entità** che le **Regole di Business**.
- La logica di Business del Domain Model interagisce a sua volta con **istanze di classi**.

Ps: attenzione la schematizzazione di questi principi, è "personale" [janky]. Deriva da raccolta di informazioni da molte fonti ed esperienza sul campo.

Model Driven Design Pattern (navigation map)



Eric Evans: Domain Driven Design [2006]

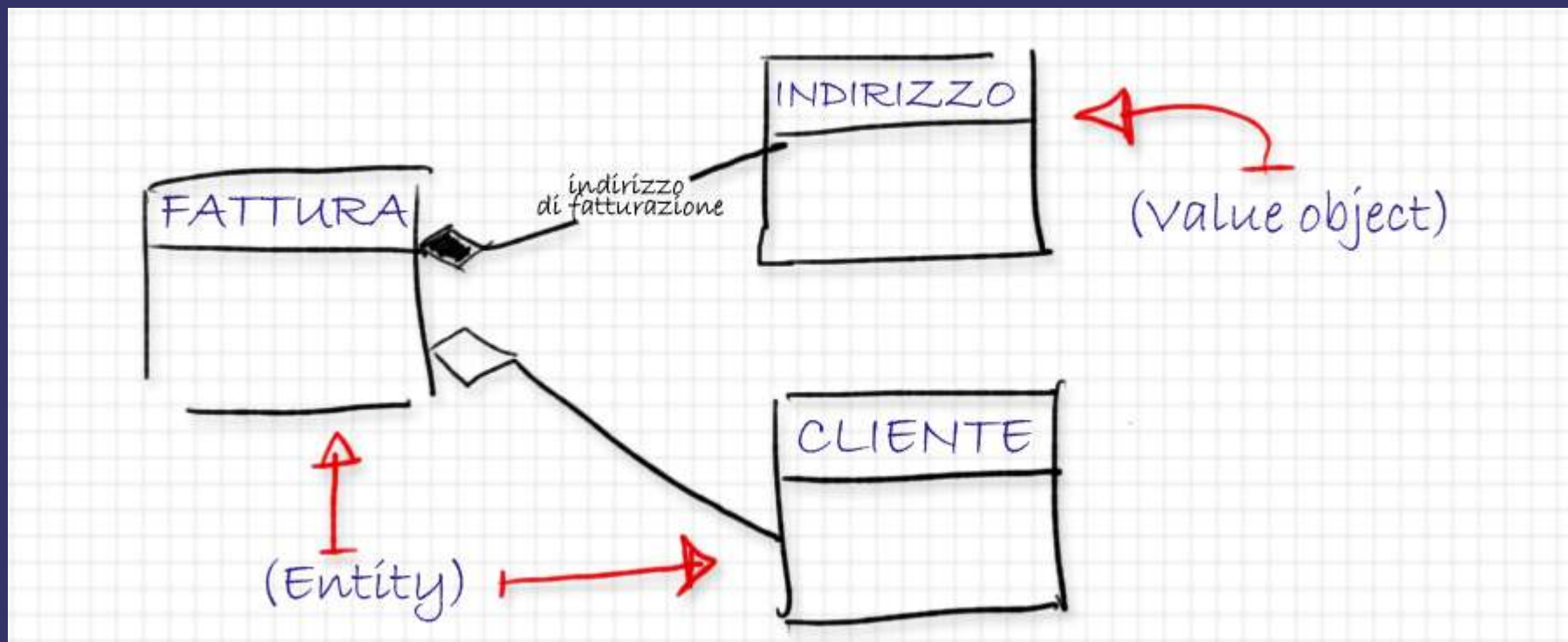
Primo principio: definizione

- *Il DM è un "object model" in cui ogni singola classe ha uno specifico significato di business, e può incorporare sia "dati" che "comportamento".*

Una applicazione che faccia uso di Domain Model non lavora direttamente con la rappresentazione tabulare dei dati, ma usa una proprio modello Object Oriented che rappresenti al meglio le entità di business.

Questo uso prepotente del paradigma Object Oriented favorisce il riutilizzo dei componenti quando si aggiungono nuove funzionalità.

Primo principio: esempio



Primo principio: alcuni elementi

	Entity	Value Object
Semantica	Semantica di Reference Es: una Fattura, un Ordine, un Cliente	Semantica di Valore: Es: un Indirizzo, una Coordinata 3D, un Numero Complesso, una Matrice
Life cycle	Indipendente	Coincidente con l'oggetto a cui appartiene
Uso	Una Entity tende a memorizzare uno stato, esporre comportamenti e coordinare altre entity/value object associati	Un value object tende a memorizzare uno stato
Uguaglianza	Basata sull'identità. Cioè dal confronto di uno (o più) attributi specifici che la rappresentano univocamente.	Basata sul confronto di tutti i suoi attributi.
Rappresentazione nel modello Object Oriented	Una Classe	Una Classe (o uno Struct)
Rappresentazione nel modello relazionale	Una (o più) tabelle	Una (o più) colonne di una tabella

Primo principio

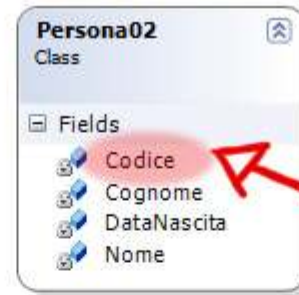
Quando si pensa agli oggetti del DM viene naturale pensare ai loro attributi, ma questi oggetti possono anche esprimere un comportamento:

- In generale si tende ad aggiungere solo i metodi essenziali ad una entity, cercando di evitare quante più dipendenze possibili dall'ambiente esterno.
- Inserire troppa logica all'interno di una entity ne può aumentare notevolmente la complessità a sfavore della espressività, chiarezza e manutenibilità

Primo principio: identità



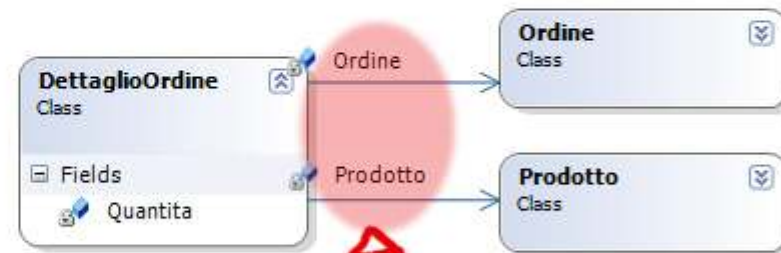
NATURAL
KEY
(STRING)



SURROGATE
KEY
(GUID)



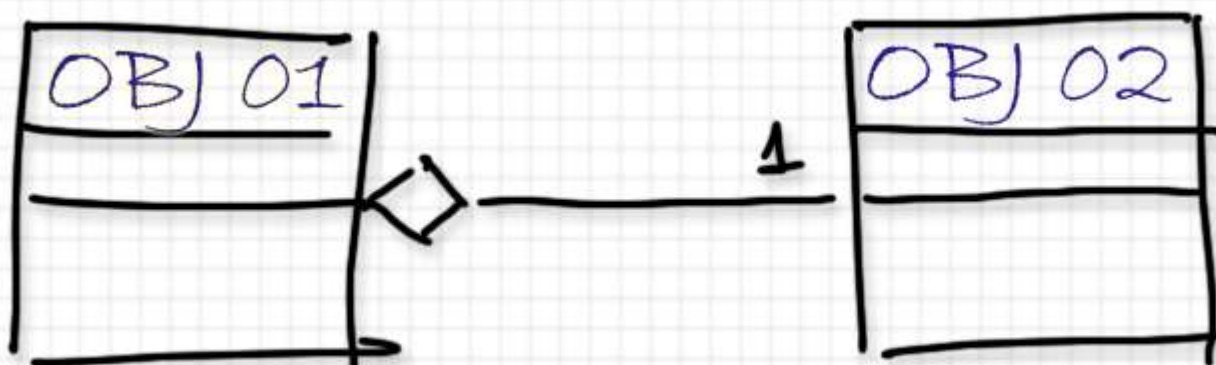
SURROGATE
KEY
(INT generato sul server)



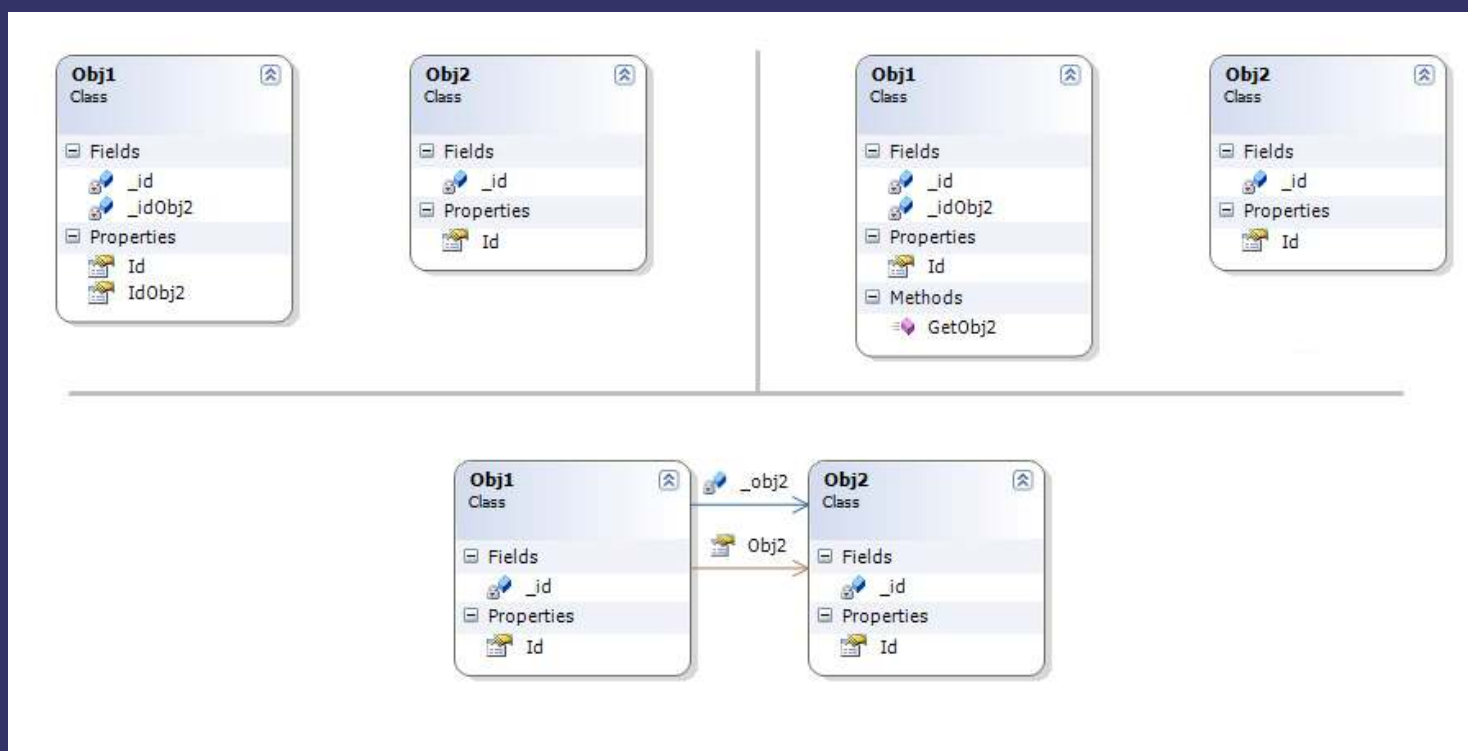
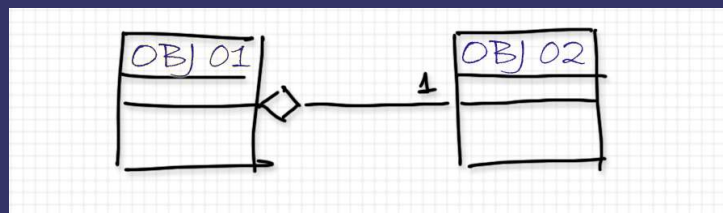
COMPOSITE
KEY
(una o più reference)

Secondo Principio

- *Le classi del Domain Model possono avere "associazioni" fino a formare una vera e propria "rete di interconnessioni".*



Secondo Principio

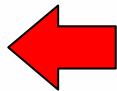


Secondo Principio

Associazione Bidirezionale



Associazione Unidirezionale



Terzo principio

- *Le classi del Domain Model possono modellare sia le "Entità" che le "Regole di Business".*

Ci sono particolari aspetti del Dominio applicativo che vengono più chiaramente espressi come operazioni che come oggetti veri e propri.

Qualcosa che il software "deve fare" e che non necessariamente coincide con uno "stato"

Forzare tutti i "comportamenti" all'interno di una entity non sempre è la scelta migliore.

Una classe Service modella, non tanto una entità di dominio, ma una regola di business (molte volte si segue il Command Pattern)

Terzo principio

Tipologie di Servizi	Esempi
Infrastrutturali	<ul style="list-style-type: none">•Spedizione di una Mail•Persistenza di una entity in uno storage•Tracciamento delle operazioni di una Entity
Applicativi (o di dominio)	<ul style="list-style-type: none">•Trasferimento di valuta tra conti bancari•Approvazione di un Ordine d'acquisto•Validazione di Entity a fronte di una operazione di business

Terzo principio

Come distribuire le regole tra Service e metodi nelle Entity?

- Una Entity con troppe responsabilità rischia di perdere in chiarezza concettuale.
- Entity troppo cariche sono di difficile manutenzione, refactoring e testing.
- Le operazioni di business, sono spesso legate ad altre entity. Esprimere queste operazioni come metodo di una Entity, aumenta la sua dipendenza da altri oggetti

Quarto principio

- *La Logica di Business del Domain Model interagisce a sua volta con "istanze di classi".*

TransazioneBancaria
Class

Methods

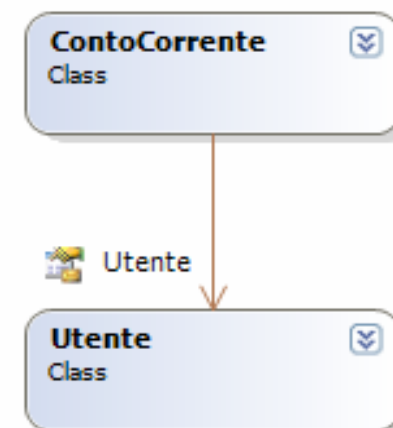

- Execute() : bool
- TransazioneBancaria(ContoCorrente CondoDare, ContoCorrente ContoAvere)
- Validate() : bool



TransazioneBancariaDataCentrica
Class

Methods

- Execute() : bool
- TransazioneBancariaDataCentrica(int CondoDare, int ContoAvere)
- Validate() : bool

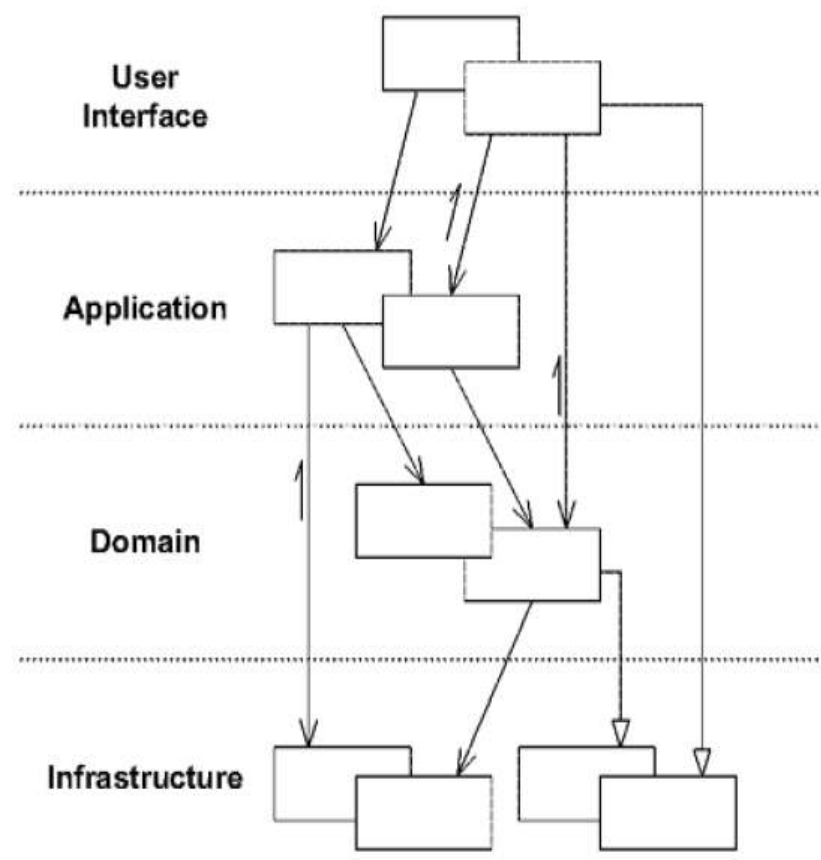


Isolamento

- Il DM deve essere quanto più indipendente da tutte le altre parti del sistema.
- E' buona norma rendere il DM facilmente isolabile, testabile, modificabile.
- La logica di business deve essere ortogonale ai servizi infrastrutturali. Evitare di inserire codice per aspetti infrastrutturali (persistenza, gestione delle transazioni, autorizzazioni e autenticazioni) nelle classi che costituiscono il Domain Model.
- Minimizzare la dipendenza anche dalle API di .NET. Renderà più facile il test senza il bisogno di ricreare a runtime particolari contesti come server, container etc.

Isolamento: Layering

Layered Architecture



Eric Evans: Domain Driven Design [2006]

User Interface (Presentation Layer)

Responsible for presenting information to the user and interpreting user commands.

Application Layer

This is a thin layer which coordinates the application activity. It does not contain business logic. It does not hold the state of the business objects, but it can hold the state of an application task progress.

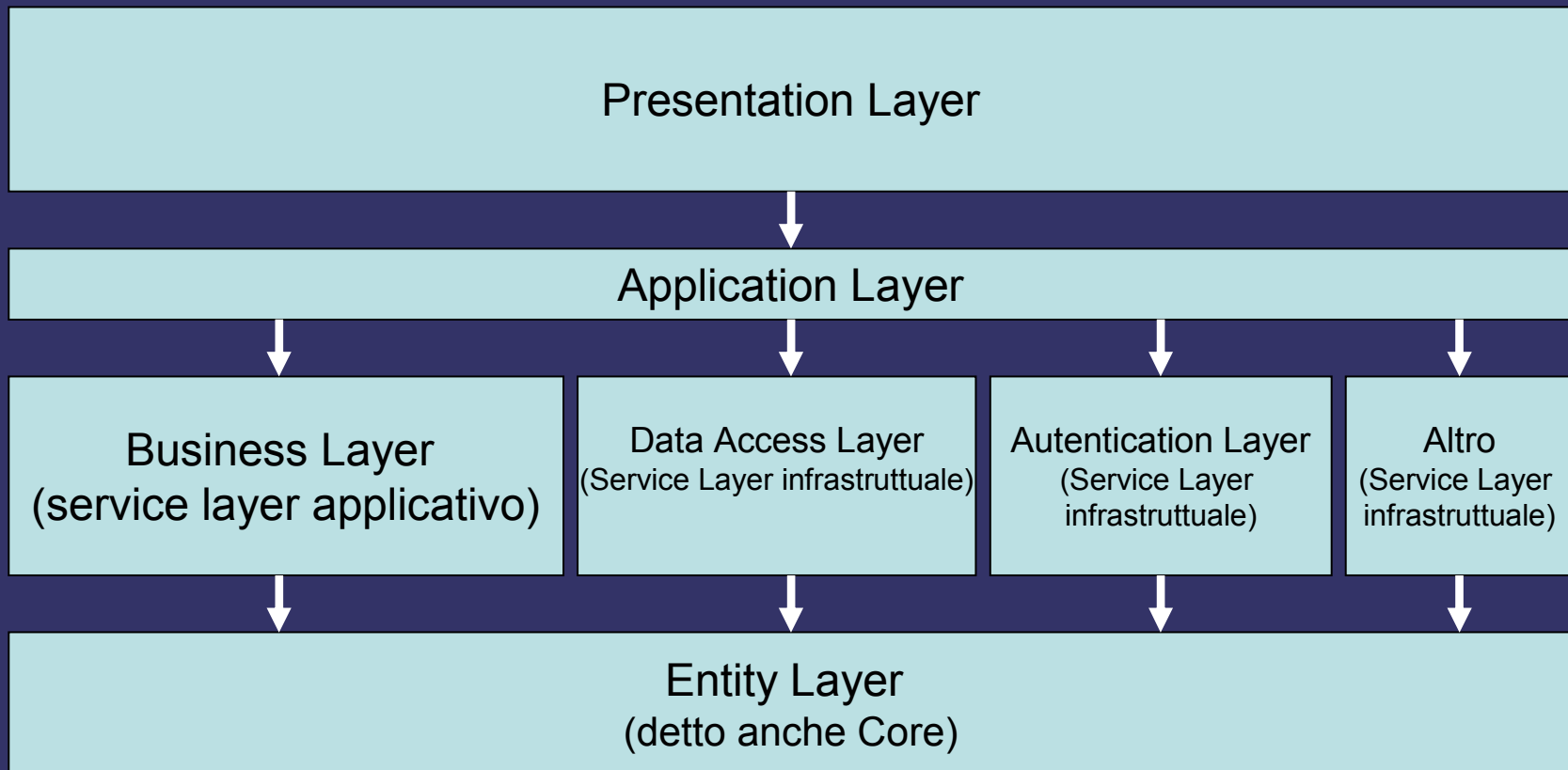
Domain Layer

This layer contains information about the domain. This is the heart of the business software. The state of business objects is held here. Persistence of the business objects and possibly their state is delegated to the infrastructure layer.

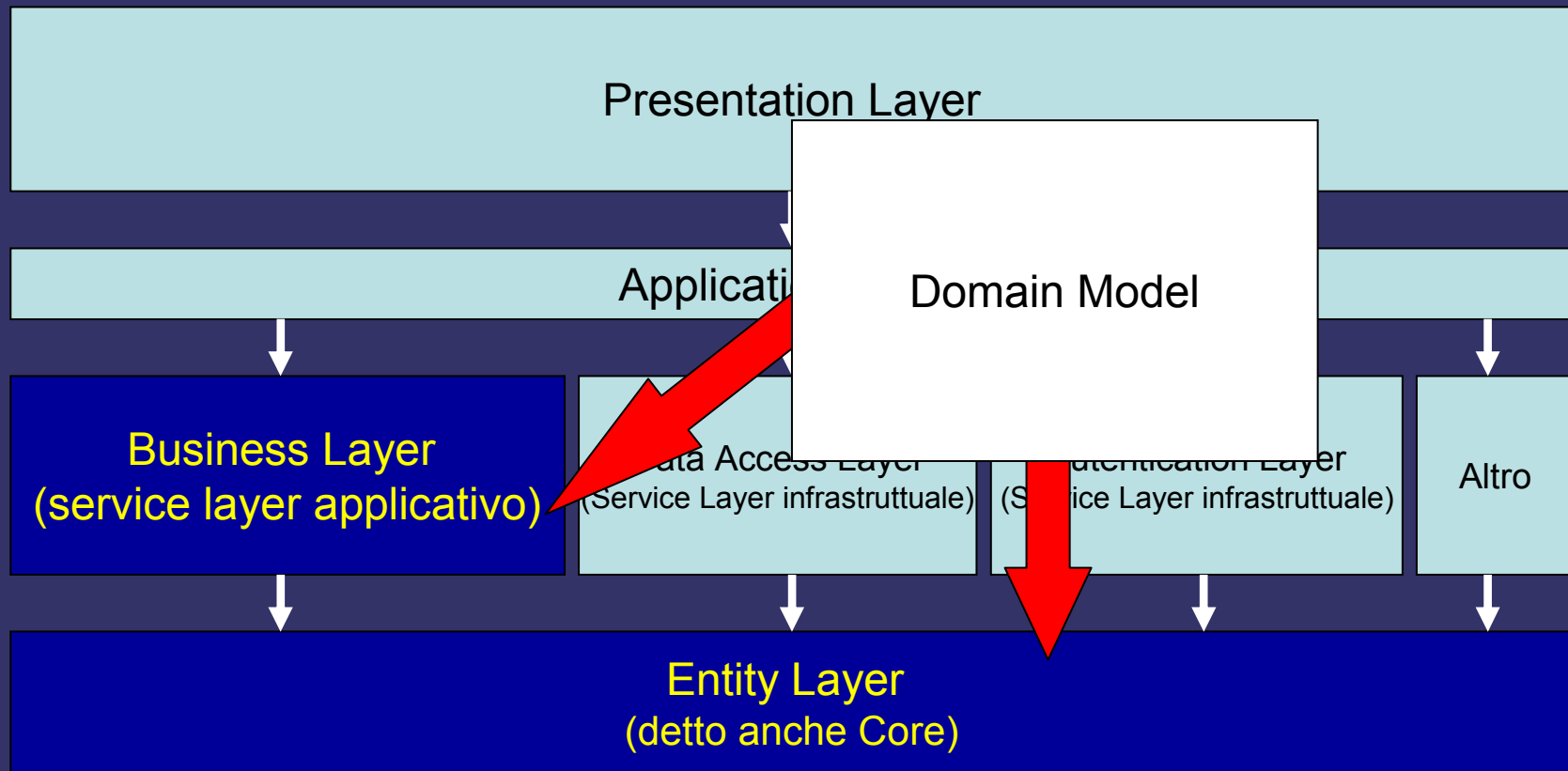
Infrastructure Layer

This layer acts as a supporting library for all the other layers. It provides communication between layers, implements persistence for business objects, contains supporting libraries for the user interface layer, etc.

Layering isolation più spinto...



Layering isolation più spinto...



Maggiori informazioni sul webcast sul Layering (sempre mio... ☺)

Cenni su Factory e Repository

Factory

Responsabilità di creare e assemblare oggetti particolarmente complessi.

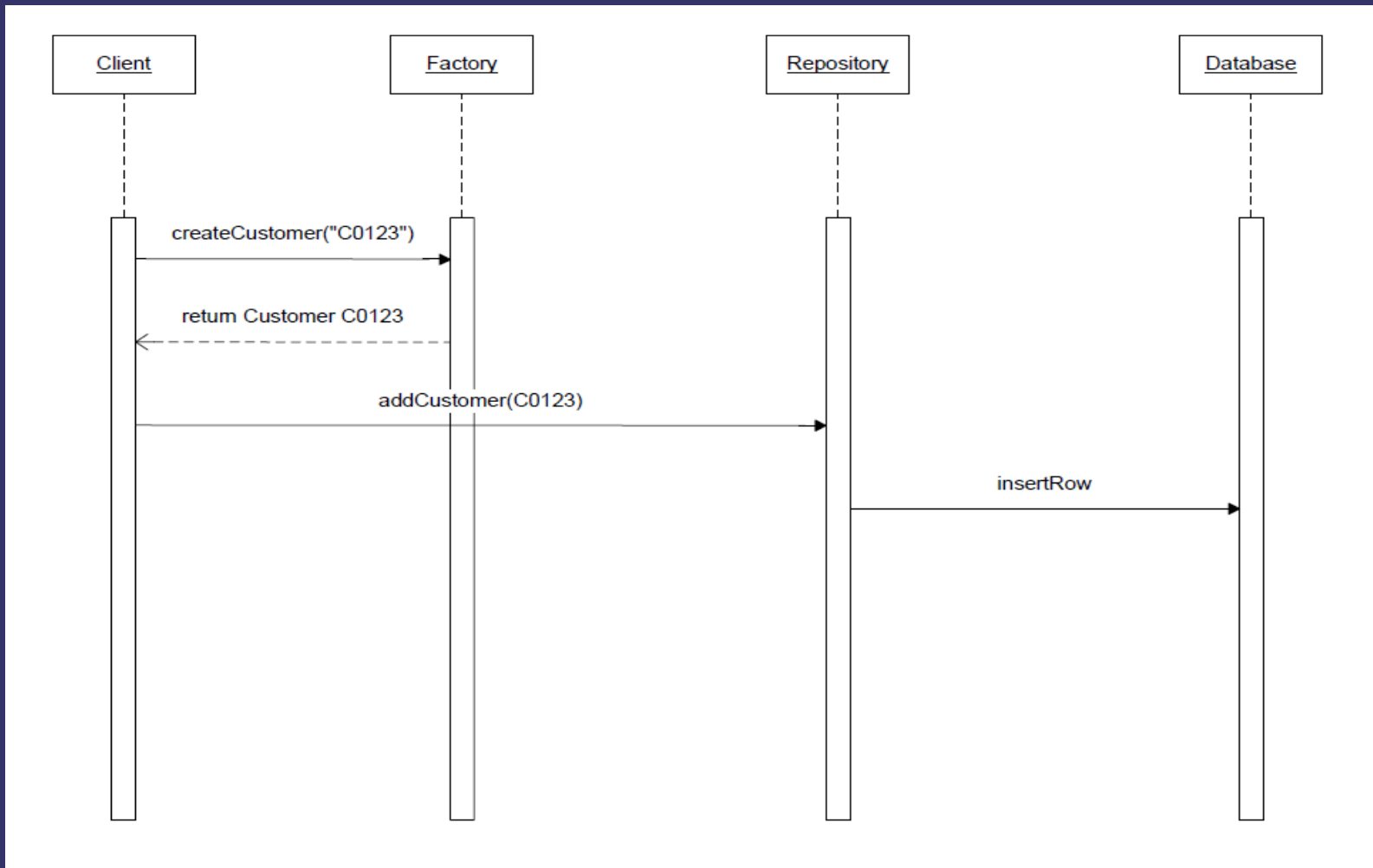
La logica di creazione di grafi molto immersa nel costruttore stesso, potrebbe non essere una “buona idea”.

Repository

Responsabilità del ciclo di vita delle Entity.

Responsabilità infrastrutturali di persistenza quali Identity Map, Cache.

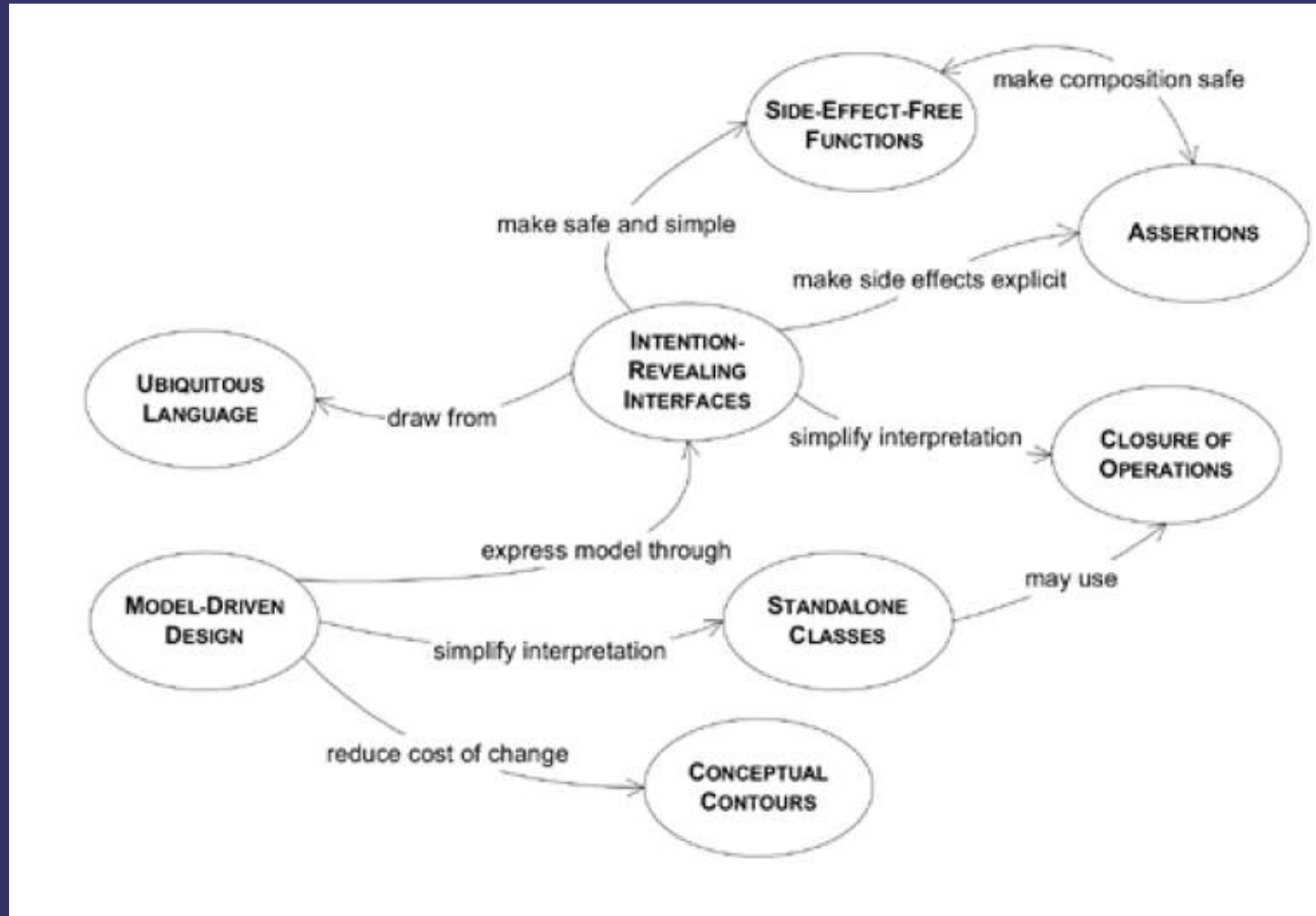
Cenni su Factory e Repository



Con la DDD abbiamo solo iniziato...

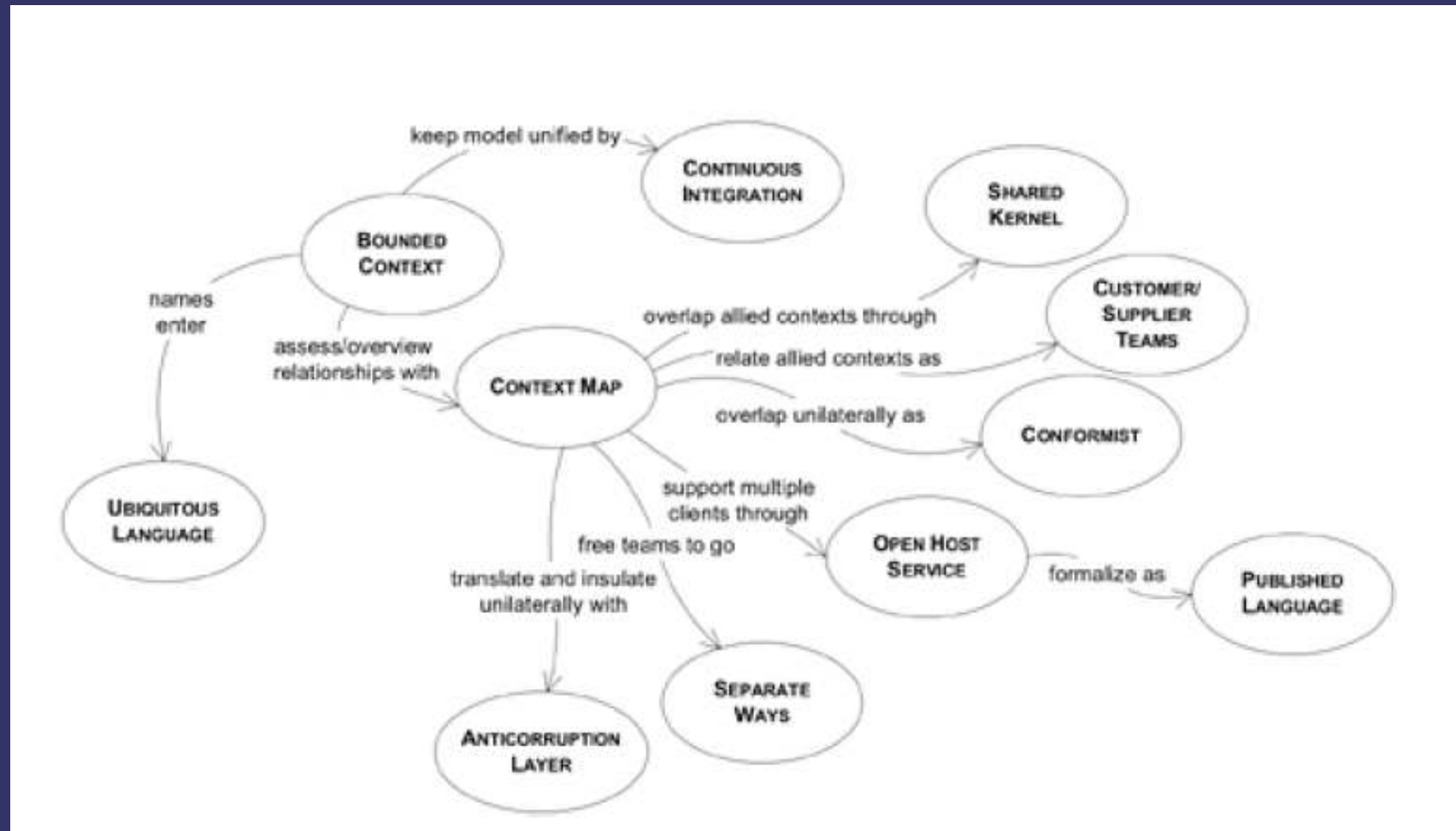


Supple Design Pattern (navigation map)



Eric Evans: Domain Driven Design [2006]

Model Integrity Pattern (navigation map)



Eric Evans: Domain Driven Design [2006]

Riferimenti

- Libro: Domain Driven Design: Tackling Complexity In The Heart Of Software
[Eric Evans 2003]
- Libro: Applying Domain-Driven Design and Patterns
[Jimmy Nilsson 2006]
- Libro: Domain Driven Design Quickly
[InfoQ 2006] (gratuito, scaricabile da internet)
- <http://domaindrivendesign.org/>
- Seguite il mio blog:
iniziativa: **“Domain Model e Enterprise Application”**

Visual Studio 2005 è disponibile
scegli il prodotto per te www.microsoft.it/msdn/vs2005/

– **Visual Studio 2005 Team Edition**

- Visual Studio Team Edition con MSDN Premium (for Architects, Developers, Testers, **Database Professionals**)
- Visual Studio Team Suite con MSDN Premium



– **Strumenti professionali**

- Visual Studio 2005 Professional
- Visual Studio 2005 Professional con MSDN Professional
- Visual Studio 2005 Professional con MSDN Premium
- Visual Studio 2005 Tools for Microsoft Office System

– **Strumenti di base**

- Visual Studio 2005 Standard
- Visual Studio 2005 Express Edition

– **Altri strumenti**

- Visual SourceSafe 2005
- VisualFox Pro 9.0



Dove acquistare:

www.microsoft.it/msdn/rivenditori/

Per informazioni:

itamsdn@microsoft.com

Domande?
(facili facili...)

