TAP - Progetto

Riassunto delle puntate precedenti

- Trovate tutti i dettagli nell'introduzione al corso (registrata)
- Punti essenziali
  - un **unica** valutazione per ciascun anno accademico
    - al momento in cui per la prima volta superate la soglia di 14/25 allo scritto
    - si valuta quello che è disponibile su GitHub alla mezzanotte prima dello scritto
      - fa fede data del commit + presenza su GitHub quando vado a scaricare (non necessariamente all'ultimo rintocco di mezzanotte)
      - se non c'è nulla o quello che c'è non passa i test minimali perdete il voto dello scritto e si applica una penalità da 0 a 7 punti sul voto dell'appello successivo
  - a fine anno il progetto scade
  - la valutazione può essere
    - solo (semi)automatica $\Rightarrow$ voto in [0,4]
    - con code inspection e discussione in sede di orale $\Rightarrow$ voto in [-3,6]
  - quello che dovete consegnare è una solution di Visual Studio 2022
    - per .Net 6
    - che usa EF Core 6.0.12 per interfacciarsi con un (arbitrario) database SQL-Server
    - non è un problema se
      - suddividete la vostra implementazione su più progetti
      - ci sono progetti ulteriori, ad esempio di testing

Componenti in che senso

- In generale
  - Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system.
  - A [software] component is a unit of composition with contractually specified interfaces and explicit context dependencies only
- Quindi nel mondo .NET la componente più naturale è una DLL (o un insieme di DLL)
- Manca un chiaro supporto linguistico per le interfacce

Interfacce fai da te

- Il costrutto interface non basta
  - descrive un livello di granularità inferiore: classi non DLL
  - descrizione solo sintattica
  - mancano dipendenze esplicite
    - in minima parte si possono esprimere usando generici
- I metadati degli assembly ci si avvicinano di più
  - granularità corretta
  - dipendenze esplicite (almeno da altri assembly)
  - continua a mancare semantica
  - scarsa leggibilità per umani

Modello di componenti (in parte già visto)

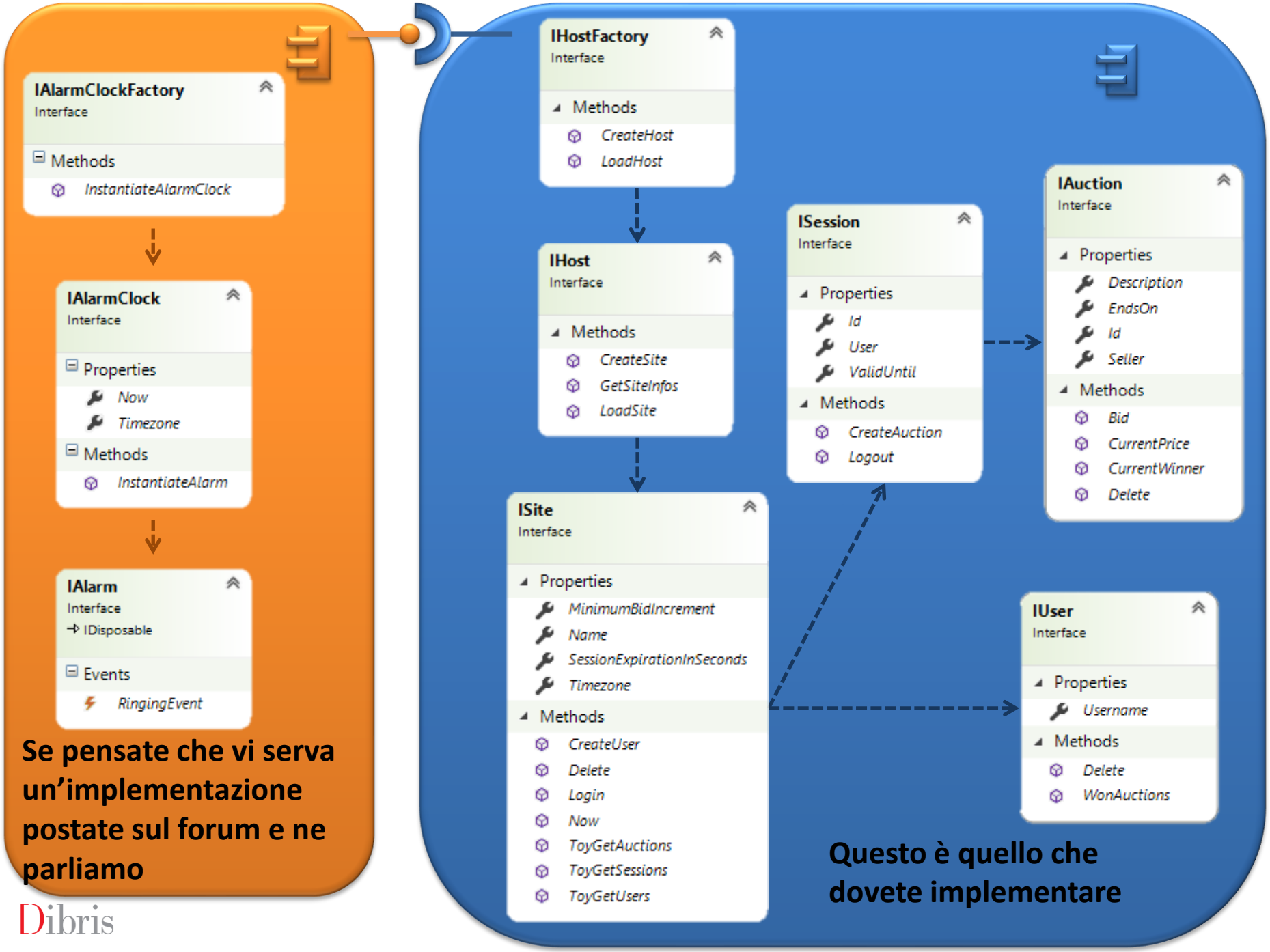
- Interfaccia modellata come class library che contiene
  - Interface C#
  - Commenti che descrivono la semantica
  - (dipendenze da altre componenti=riferimenti alle loro DLL di interfaccia)
- Componente modellata come
  - DLL di interfaccia
  - DLL con implementazioni relative
    - implementazione delle interfacce
    - …inclusa la factory
- Composizione supportata da Dependency Injection Containers
  - non nel progetto per semplicità
    - il DIC di default di .NET è pensato per contesti di programmazione più complicati dei test (es. una web application)
    - third party DIC non pienamente compatibili

Testo Componente

- Versione **giocattolo** della componente di back-office per hosting di siti di vendite all'asta stile ebay
- Per ogni sito
  - solo operazioni essenziali: mettere in vendita e fare offerte
  - nella variante più semplice possibile = senza ripensamenti
    $\Rightarrow$ non serve storico
- Concetto essenziale: temporizzazione
  - scadenza delle aste
  - validità delle offerte
  - demandata ad una *componente richiesta*: IAlarmClock
    - usiamo dependency injection per lasciare a chi usa la nostra componente la responsabilità/libertà di sceglierne un'implementazione
- Dovete realizzare la componente per la logica delle aste
  - TAP22-23.AuctionSite.AuctionSite.Interface.dll

TAP22-23.AuctionSite.AuctionSite.Interface.dll

- IHostFactory
  - Setup di un servizio di Hosting, caratterizzato dal DB su cui memorizzare i dati dei siti
    - se il DB non esiste viene creato, se esiste già viene cancellato e ricreato
  - Load di un servizio di Hosting
- IHost
  - Gestione dei siti di aste (creazione, ricerca, loading...) usando un DB (che si assume inizializzato mediante l'operazione precedente)
- ISite
  - un'instanza di ISite rappresenta un sito di aste
    - ci possono essere molte diverse istanze dello stesso sito o di siti diversi in contemporanea
  - tutti gli altri concetti sono relativi ad un ISite e possono essere creati/acceduti solo attraverso di esso
    - IUser (venditori e acquirenti)
    - IAuction (le aste)
    - ISession (sessione di un particolare utente)
      - peculiarità: ogni utente ha al più una sessione valida alla volta
  - è la radice di un aggregate
    - https://martinfowler.com/bliki/DDD_Aggregate.html

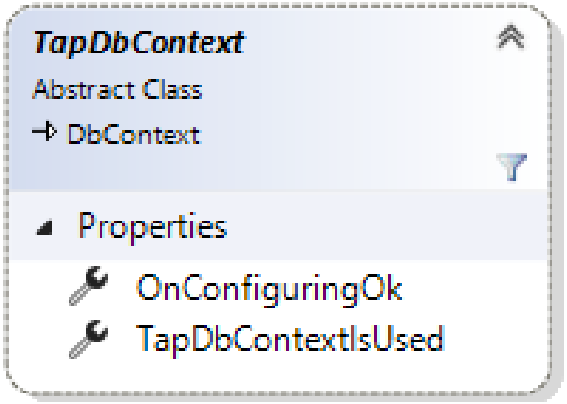

IAlarmClockFactory
Interface
Methods
InstantiateAlarmClock
IAlarmClock
Interface
Properties
Now
Timezone
Methods
InstantiateAlarm
IAlarm
Interface
IDisposable
Events
RingingEvent
Se pensate che vi serva un'implementazione postate sul forum e ne parliamo
IHostFactory
Interface
Methods
CreateHost
LoadHost
IHost
Interface
Methods
CreateSite
GetSiteInfos
LoadSite
ISession
Interface
Properties
Id
User
ValidUntil
Methods
CreateAuction
Logout
IAuction
Interface
Properties
Description
EndsOn
Id
Seller
Methods
Bid
CurrentPrice
CurrentWinner
Delete
ISite
Interface
Properties
MinimumBidIncrement
Name
SessionExpirationInSeconds
Timezone
Methods
CreateUser
Delete
Login
Now
ToyGetAuctions
ToyGetSessions
ToyGetUsers
IUser
Interface
Properties
Username
Methods
Delete
WonAuctions
Questo è quello che dovete implementare

Dibris

Eventi

- La componente IAlarmClock fornisce gli strumenti per settare una *sveglia* che suona a intervalli fissi
- Tecnicamente la sveglia è un evento e non abbiamo visto gli eventi a lezione
- Non dovrebbe essere un problema perché
  - avete usato eventi a SAW
  - avete visto il pattern observer (di cui gli eventi sono un'implementazione predefinita) a FIS
- Per sicurezza due puntatori
  - https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/
  - ho caricato un breve video con il riassunto delle cose essenziali per usare eventi in C# sul canale di TAP

Persistenza



- Il testo richiede di
  - usare Entity Framework Core
  - estendere TAPDbContext (che estende DbContext) invece di estendere direttamente DbContext
    - per semplificare il testing
- Il Db viene creato da IHostFactory
  Schema e metodo di accesso decisi da chi implementa

  ⇒ impossibile fare test generali e *usabili* che popolando direttamente il DB

  ⇒ i test usano l'implementazione anche per il setup

  → Assert.Inconclusive quando fallisce...a meno di eccezioni
- Gli oggetti in memoria possono o meno corrispondere a ciò che sta sul DB
  - Quasi tutti i metodi possono sollevare:
    - **AuctionSiteUnavailableDbException**: il DB non risponde
    - **AuctionSiteInvalidOperationException**: l'oggetto è in uno stato inconsistente (per es., è stato cancellato)
  - Fra una verifica dello stato del DB (es. *esiste il sito?*) e un uso (es. *recupero l'elenco degli utenti*) può succedere di tutto…se avvengono con query distinte
    - impatto sulle eccezioni che possono essere sollevate dal DB (e devono essere gestite da voi)
    - impatto sul design del DB (che deve fare quanti più controlli possibili in autonomia)

Eccezioni

Prassi standard

- ogni componente/libreria definisce una gerarchia di eccezioni
  - la radice denota generica eccezione causata dalla componente
  - i tipi più specifici raffinano
  - senza reinventare l'acqua calda
- Eccezioni sollevate da altre componenti/librerie usate e non gestite non si intercettano
  - es. ArgumentException

Nel progetto *esageriamo*

- gestite ogni eccezione sollevata (tipicamente da EF)
  - di solito semplicemente cambiate tipo
    - spesso scopo didattico (in un progetto vero non si farebbe)
- sollevate eccezioni direttamente voi
  - spesso di tipo generale come *argomento non accettabile* (in un progetto vero si userebbero i tipi di eccezione del sistema)
- tutte le eccezioni che escono dalla vostra implementazione hanno tipi definiti nell'interfaccia TAP22-23.AuctionSite.AuctionSite.Interface.dll
  - per ragioni didattiche
  - si evitano test che passano per falsi positivi

E ora?

- Su AulaWeb trovate il messaggio con l'invito per creare il vostro repo per il progetto su GitHub
  - seguite **SCRUPOLOSAMENTE** le istruzioni
- Accettando l'invito viene creato il vostro repo (non serve fare esplicitamente fork), a cui abbiamo accesso solo voi e io
- Potete cominciare a lavorarci subito
- Un paio di consigli
  - molti problemi del progetto sono simili a cose già viste nell'ultimo laboratorio
    $\Rightarrow$ provate a svolgerlo prima di affrontare il progetto e sfruttate la soluzione
  - per chi ha già fatto il progetto dello scorso anno
    - struttura e problematiche sono uguali
    - tecnologia cambiata (da .NET 5 a .NET 6)
    - modificate dll delle interfacce e un paio di test obbligatori
    - l'uso di GitHub non è più elemento di valutazione (perdere l'history non è penalizzante)

    Potete creare una solution per .NET 6 e aggiungere i file dello scorso anno ai progetti, mettere le reference nuove e dovrebbe funzionare al 99%
    - se non avete passato TAP lo scorso anno, probabilmente non eravate stra-preparati…riguardare e migliorare il progetto potrebbe aiutarvi a perfezionare la preparazione (e prendere qualche punto in più di progetto che non fa mai male)
- Se avete dubbi, potete fare domande sul forum in qualsiasi momento e cercherò di rispondere rapidamente

NON, ripeto NON usate questo link per nessuna ragione

Join the classroom:

TAP2022-23

To join the GitHub Classroom for this course, please select yourself from the list below to associate your GitHub account with your school's identifier (i.e., your name, ID, or email).

Can't find your name? Skip to the next step →

Identifiers
..... altri nominativi.....
Vostra matricola

Dibris