

## Architetture Software

# Architetture basate su componenti

**Dispensa ASW 440**  
ottobre 2014

*Affida le cose speciali  
agli specialisti.*

*Robert Spinrad*



## - Fonti

- [POSA4] Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing, 2007
- [UML Components], Cheesman, Daniels, UML Components – A simple process for specifying component-based software, Addison-Wesley, 2000
- [UML Components], Cheesman, Daniels, UML Components – un semplice processo per la specifica di software basato su componenti, Addison-Wesley, 2002
  - <http://www.umlcomponents.com/>



## - Obiettivi e argomenti

### □ Obiettivi

- introdurre alcuni aspetti relativi alle architetture basate su componenti e alle relative tecnologie
- presentare alcuni pattern architetturali fondamentali utilizzati nelle architetture a componenti

### □ Argomenti

- componenti
- nozioni legate ai componenti
- architetture a componenti
- tecnologie a componenti
- pattern architetturali per contenitori
- discussione



## - Wordle





## \* Componenti

- Nel contesto delle architetture software, esistono più accezioni per il termine “componente software” – tra cui
  - un’accezione più astratta e generica
    - nelle architetture software ci sono due tipi principali di elementi software – *componenti* (responsabili dell’implementazione di *funzionalità* e della gestione di *dati*) e *connettori* (responsabili delle *interazioni* tra componenti)
  - un’accezione tecnologica
    - le *tecnologie a componenti* rappresentano un’evoluzione delle tecnologie a oggetti distribuiti
  - un’accezione metodologica
    - lo *sviluppo del software basato su componenti* è un approccio alla costruzione di grandi sistemi software basato sullo sviluppo e sull’integrazione di componenti software
  - queste accezioni sono in qualche modo correlate tra loro

5

Architetture basate su componenti

Luca Cabibbo – ASw



## . Componenti come elementi architetturali

- L’architettura di un sistema software è la struttura ... del sistema, che comprende *elementi software* ...
  - due tipi principali di elementi software: componenti e connettori
  - un *componente software* è una entità architetturale che
    - incapsula un insieme di funzionalità e/o di dati di un sistema
    - restringe l’accesso a quell’insieme di funzionalità e/o dati tramite delle interfacce (fornite) definite in modo esplicito
    - ha un proprio contesto di esecuzione – anche le sue dipendenze sono specificate tramite delle interfacce (richieste) definite in modo esplicito
  - i *connettori* sono elementi responsabili delle *interazioni* tra componenti
- I componenti sono l’incarnazione dei principi dell’ingegneria del software di incapsulamento, astrazione e modularità

6

Architetture basate su componenti

Luca Cabibbo – ASw



## Componenti come elementi architeturali

- L'architettura di un sistema software è la **struttura** ... del sistema, che comprende elementi software ...
  - un sistema software può essere realizzato mediante la **composizione** di componenti software
    - la composizione è basata sulla “connessione” di più componenti, realizzata sulla base delle interfacce dei componenti e mediante l'ausilio di connettori, per dar luogo a un'opportuna struttura – che dovrebbe presentare le caratteristiche di qualità desiderate



## . Tecnologie a componenti

- Le **tecnologie a componenti** rappresentano un'evoluzione delle tecnologie a oggetti distribuiti
  - un **componente** è un'entità software runtime
    - implementa un insieme di funzionalità
    - offre i suoi servizi mediante un insieme di interfacce con nome (interfacce fornite)
    - può richiedere servizi ad altri componenti, sempre sulla base di interfacce con nome (interfacce richieste)
  - in pratica, un'applicazione è formata da un insieme di componenti, che vengono composti (al momento del rilascio) sulla base delle loro interfacce
  - alcuni esempi di tecnologie a componenti concrete
    - CORBA – Microsoft COM, DCOM e .NET – Java EE



## . Sviluppo del software basato su componenti

- Lo *sviluppo del software basato su componenti* (CBSD o CBSE) è un approccio alla costruzione di grandi sistemi software basato sullo sviluppo e sull'integrazione di componenti software
  - alcuni componenti potrebbe essere preesistenti (acquisiti da terze parti o sviluppati in precedenza) oppure sviluppati appositamente come tali
  - in questo contesto, rivestono un ruolo fondamentale le attività di specifica dei componenti e quella dell'integrazione di componenti
    - in pratica, la specifica dei componenti è basata sull'uso di interfacce, e le interazioni tra componenti avvengono anch'esse con riferimento alle interfacce dei componenti



## Discussione

- Alcuni aspetti comuni tra le diverse accezioni
  - un componente è un elemento architeturale – astratto o concreto – con responsabilità funzionali
  - ciascun tipo di componente è caratterizzato (è specificato) in termini di un insieme di interfacce fornite e di interfacce richieste
  - i componenti sono pensati per essere composti – per formare una qualche “configurazione” o “struttura”
  - la composizione dei componenti – ovvero, la specifica delle loro interazioni – è basata sulla connessione delle loro interfacce
- Nel seguito di questa dispensa, faremo principalmente riferimento all'accezione più astratta e a quella tecnologica dei componenti
  - che possono essere considerate, rispettivamente, una generalizzazione e una concretizzazione di una stessa nozione



# Componenti

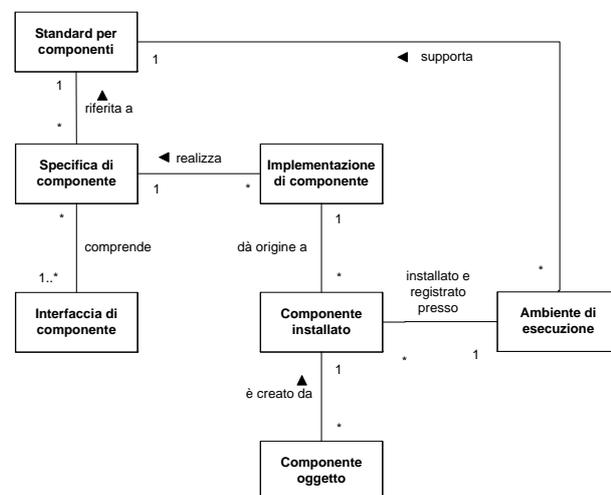
- Per riassumere, ecco una possibile definizione di “componente” [Szyperski]
  - un **componente software** è una unità eseguibile di composizione – con interfacce specificate in modo contrattuale e dipendenze dal contesto esplicite
  - un componente software può essere rilasciato (deployed) indipendentemente, ed è soggetto a composizione, anche da terze parti



## \* Nozioni legate ai componenti

- Ci sono diverse nozioni legate ai componenti – poiché la parola “componente” può essere riferita ad alcune di queste nozioni, i cui significati sono correlati ma diversi, è bene comprendere tutte queste nozioni

- standard per componenti
- specifica di componente
- interfaccia di componente
- implementazione di componente
- componente installato
- componente oggetto





## Standard per componenti

- Uno **standard** (o **modello**) **per componenti** definisce uno standard per l'implementazione, la documentazione e il rilascio di componenti
  - alcuni esempi notevoli
    - Corba Component Model
    - modello EJB – della piattaforma Java EE
    - modello .NET – della tecnologia Microsoft
  - si noti che uno standard è una specifica (“cartacea”) – e non un'implementazione (software)
  - questa specifica è di solito composta da due parti
    - specifica per lo sviluppatore – che vuole realizzare componenti aderenti allo standard
    - specifica per chi vuole realizzare un'implementazione dello standard – ovvero, una sua piattaforma di esecuzione



## Piattaforma per componenti

- Un **ambiente** (o **piattaforma**) **di esecuzione** fornisce ai componenti i servizi previsti dal relativo standard
  - alcuni esempi notevoli
    - un **application server** Java EE, nel caso della piattaforma Java EE – ad es., IBM WebSphere AS
    - un sistema operativo Windows, nel caso della piattaforma .NET
  - l'ambiente di esecuzione funge da **contenitore** per i componenti eseguibili
  - il relativo standard definisce, tra l'altro, le interazioni (mutue) previste e/o consentite tra componenti e contenitori



## Specifica e interfaccia di componente

- La **specifica di un componente** definisce in modo preciso il comportamento di un certo componente
  - ovvero, le modalità di interazione del e con il componente
  - il comportamento di un componente è definito da un insieme di interfacce – ad esempio, da un insieme di operazioni e dai relativi contratti
- Gran parte della specifica di un componente consiste nella definizione delle **interfacce del componente**
  - questo insieme di interfacce è anche chiamato collettivamente come l'**interfaccia del componente**
  - definisce l'insieme dei comportamenti forniti da un componente



## Implementazione di componente

- Un'**implementazione di un componente** è la realizzazione (software) di una specifica di componente
  - si tratta di un componente autonomo, che può essere distribuito e installato in modo (possibilmente) indipendente da altri componenti
  - la separazione netta tra specifica e implementazione è una caratteristica fondamentale dei componenti
  - per una certa specifica di componente, possono esistere più implementazioni di componenti diverse che la realizzano
- Inoltre, un'implementazione può esistere in diverse “forme”
  - codice sorgente
  - codice binario
  - unità di distribuzione



## Componente installato

- Un **componente installato** è una copia di un'implementazione di componente che è stata installata/rilasciata (deployed) su un particolare computer/ambiente di esecuzione
  - l'installazione avviene su un ambiente (piattaforma) di esecuzione – l'installazione comprende la configurazione e la registrazione del componente in tale ambiente
  - un'implementazione di componente può dar origine a più componenti installati

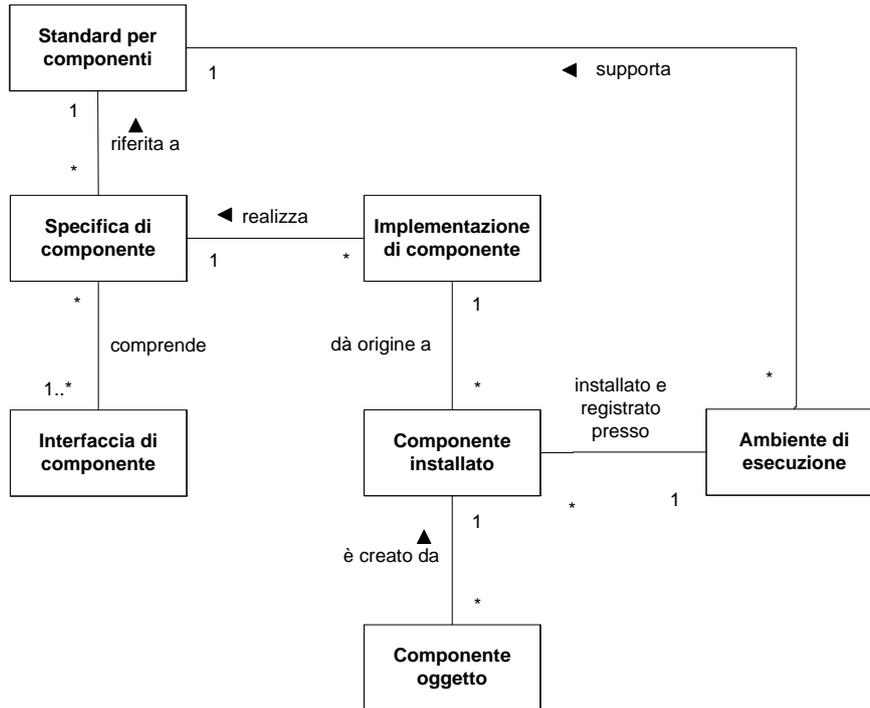


## Componente oggetto

- Per **componente oggetto** si intende un'istanza creata a partire da un componente installato, nel contesto di un ambiente di esecuzione
  - i componenti oggetto esistono solo durante l'esecuzione – ovvero, a runtime
  - sono solo i componenti oggetto a essere in grado di offrire servizi concretamente
  - i componenti oggetto hanno un'identità univoca, nonché un proprio stato (ovvero, dei dati)
  - un componente installato può avere zero, uno o più componenti oggetto da esso creati



## Nozioni legate ai componenti



## Esempio: MS Word

- MS Word comprende diversi componenti, di cui due abbastanza ovvi
  - lo *standard* per componenti è COM
  - l'*ambiente di esecuzione* è un s.o. Windows, che supporta COM
  - i due *tipi di componenti* “ovvi” sono quelli che rappresentano rispettivamente l'applicazione (WordApp) e un documento (WordDoc)
    - non conosciamo né le loro *specifiche* né le relative *interfacce*
  - il file winword.exe “impacchetta” le *implementazioni* di queste due specifiche di componenti
  - al momento dell'installazione, il file winword.exe viene copiato sul disco, dando origine a due *componenti installati*, che vengono registrati nell'ambiente COM
  - quando viene eseguita l'applicazione, vengono creati due *componenti oggetto*: uno di tipo WordApp per l'applicazione e uno di tipo WordDoc per un nuovo documento
    - ogni nuovo documento aperto sarà rappresentato da un ulteriore *componente oggetto* di tipo WordDoc



## Esempio: un'applicazione web Java EE

- ❑ Lo *standard* per componenti è Java EE
- ❑ L'*ambiente di esecuzione* è un application server (AS) Java EE – ad es., Tomcat (che implementa Java EE per la parte web)
- ❑ L'*implementazione* (unità di distribuzione) è il file war – costruito anche sulla base di un descrittore di deployment
- ❑ Per essere usata, l'unità di distribuzione deve essere *installata/deployata* sull'AS
- ❑ Richieste HTTP all'applicazione causano/possono causare
  - la creazione di *componenti oggetto* coinvolti – ad es., la creazione di oggetti istanza delle classi servlet
  - la richiesta di esecuzione di operazioni a questi oggetti – queste richieste vengono fatte dall'AS agli oggetti servlet
- ❑ Il ciclo di vita dei vari oggetti dell'applicazione è gestito dall'AS
  - attenzione – lo stesso oggetto servlet potrebbe essere usato da più sessioni oppure, in una stessa sessione, in interazioni diverse, potrebbe venire usate istanze diverse di una stessa classe servlet



## \* Architetture a componenti

- ❑ Nell'*architettura a componenti*
  - un sistema software viene realizzato come composizione di un certo numero di componenti
  - la composizione avviene in sede di deployment – ovvero, di rilascio (o installazione)
  - la composizione viene specificata connettendo le interfacce richieste dei componenti con interfacce fornite da altri componenti



## Componenti e interfacce

- Le interfacce hanno un ruolo rilevante nelle architetture a componenti – evidenziato dalle varie definizioni di componente
  - c'è una separazione netta tra interfaccia e implementazione
  - le dipendenze tra componenti sono espresse solo in termini di interfacce
  
- In generale, per “interfaccia” si intendono le assunzioni che i componenti possono fare circa gli altri componenti
  - queste assunzioni comprendono nomi delle operazioni e parametri – ma anche i loro contratti, in termini di pre- e post-condizioni, effetti collaterali, consumo di risorse globali, vincoli di coordinamento, service level agreement, ...



## Componenti e interfacce

- Perché le interfacce dei componenti sono tanto importanti?
  - le interfacce dovrebbero descrivere, in modo chiaro
    - le responsabilità di un componente, e i servizi offerti
    - il suo protocollo d'uso (e di composizione)
  - inoltre, i client di un componente dovrebbero poter implementare (in modo semplice) una collaborazione efficace e corretta solo sulla base dell'interfaccia del componente
  - per questo, la specifica dei componenti (delle loro interfacce) è un'attività chiave nella definizione delle architetture software



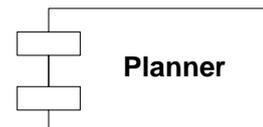
## Componenti e interfacce

- I componenti possono essere dotati di due tipi di interfacce
  - **interfacce fornite**
    - specifica dei servizi forniti da un componente ad altri componenti
  - **interfacce richieste**
    - specifica dei servizi che devono essere resi disponibili ad un componente, affinché esso possa funzionare come specificato
    - se non sono disponibili, il componente non funzionerà



## Notazione UML per i componenti

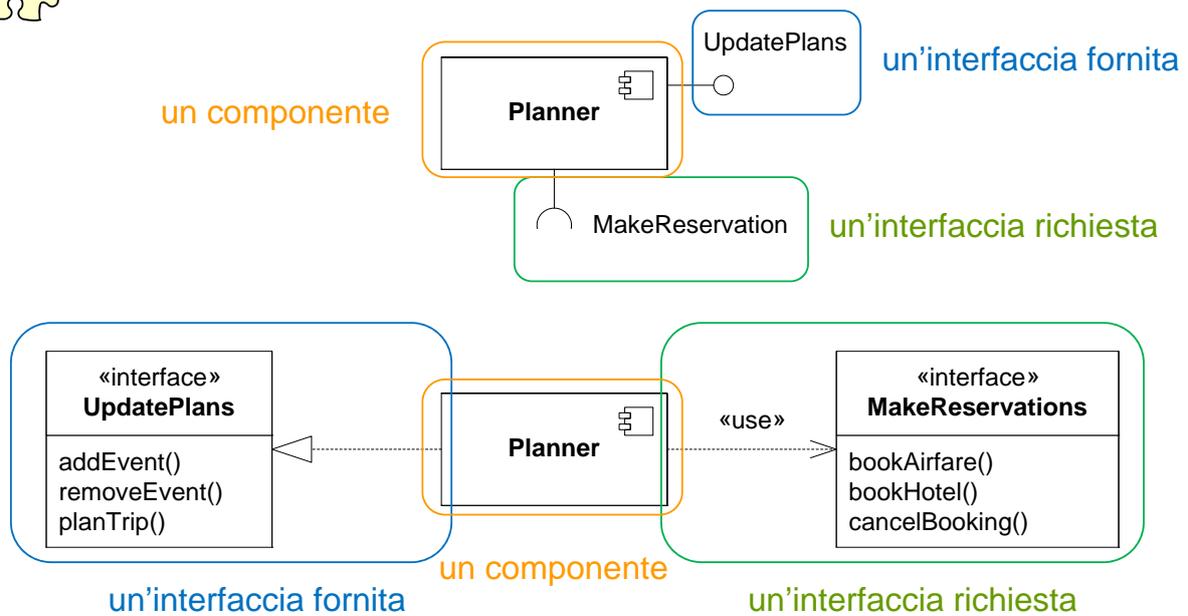
- Definizione di componente secondo UML
  - un **componente** è una parte modulare di un progetto di un sistema che nasconde la sua implementazione dietro un insieme di interfacce esterne [UML]



UML 1.x



## Notazione UML per le interfacce



### □ Un componente

- può fornire zero, una o più interfacce (dunque anche più di una)
- analogamente, può richiedere zero, una o più interfacce

27

Architetture basate su componenti

Luca Cabibbo – ASw



## Stili per le interfacce

### □ I componenti possono interagire tra loro sulla base di due modalità principali (“stili”) di comunicazione

- **stile procedurale**
  - nello stile RMI – comunicazione sincrona
  - un'interfaccia in questo stile dichiara un insieme di operazioni (fornite oppure richieste)
- **stile orientato ai messaggi/documenti**
  - nello stile del messaging – comunicazione asincrona
  - un'interfaccia in questo stile dichiara un insieme di tipi di messaggi (accettati oppure inviati)

### □ In corrispondenza, l'interfaccia sarà espressa secondo modalità diverse

### □ Sono possibili anche componenti dotati sia di interfacce procedurali che di interfacce orientate ai messaggi

28

Architetture basate su componenti

Luca Cabibbo – ASw



## Composizione

- La composizione di un'applicazione può avvenire, in sede di deployment
  - sulla base di un certo numero di componenti
  - connettendo le interfacce richieste dei componenti con interfacce fornite da altri componenti

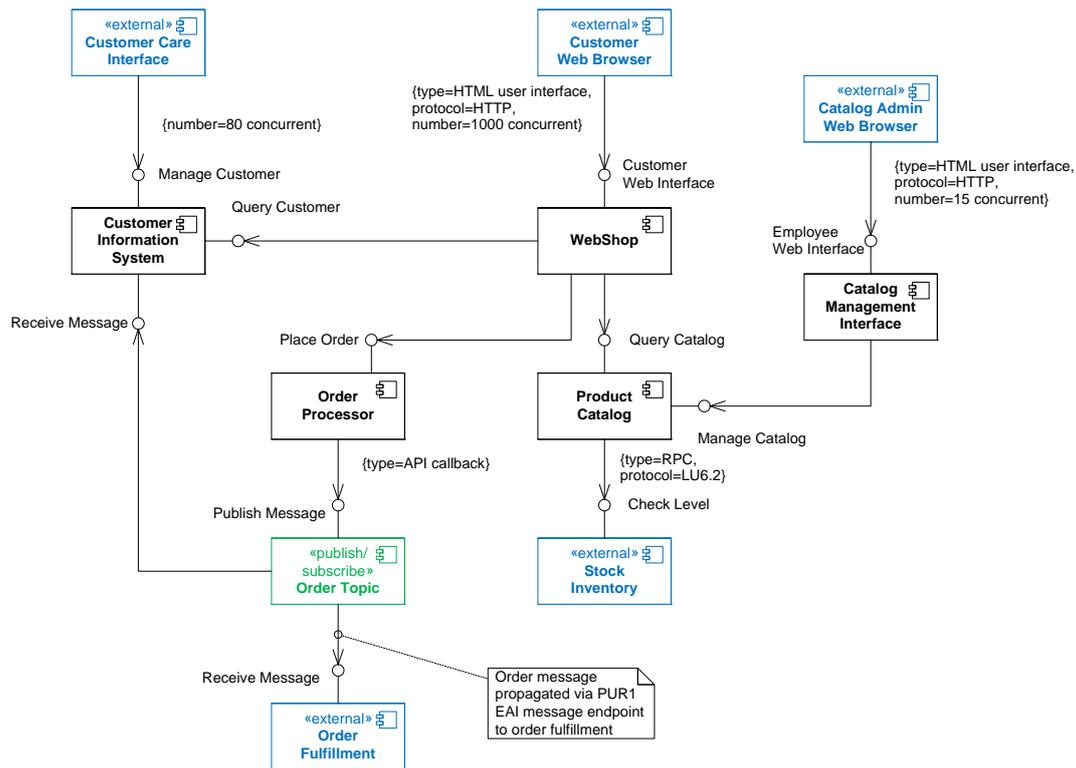


## Composizione

- In modo più specifico, la composizione di un'applicazione – basata su certo numero di componenti – avviene
  - in sede di *specifica*, sulla base delle interfacce fornite e richieste dei componenti
    - specificando quali componenti oggetto servono, ciascuno con la sua specifica
    - specificando come sono tra loro collegati – in termini di collegamenti/dipendenze tra componenti, con riferimento alle loro interfacce fornite/richieste
  - in sede di *deployment*, specificando inoltre per ciascuno dei componenti la particolare implementazione prescelta
    - la composizione sulla base delle interfacce consente infatti la sostituzione di componenti – con altri componenti in grado di fornire le stesse interfacce



## Esempio



## \* Tecnologie a componenti

- Le **tecnologie a componenti** rappresentano un'evoluzione delle tecnologie a oggetti distribuiti
  - le architetture client/server, a oggetti distribuiti e di messaging – che sono alla base di molte tecnologie per sistemi distribuiti – si sono evolute per semplificare ulteriormente lo sviluppo dei sistemi distribuiti
- Middleware per componenti
  - evoluzione del middleware per oggetti distribuiti
  - i componenti vivono in contenitori (application server) – che sono un'evoluzione significativa dei broker – in grado di gestire la configurazione e la distribuzione dei componenti, e fornire ad essi servizi di supporto
  - possibile sia la comunicazione nello stile procedurale che basata sullo scambio di messaggi



## Evoluzione storica

- Le tecnologie a componenti nascono, a metà degli anni '90, per superare alcuni limiti delle tecnologie a oggetti distribuiti
  - mancanza di meccanismi per separare le dipendenze tra gli oggetti dalle loro implementazioni
    - se un oggetto deve fruire di un servizio fornito da un altro oggetto, deve scoprire e connettersi a questo oggetto in modo esplicito
  - mancanza di strumenti di deployment e configurazione
    - l'allocazione di un oggetto remoto ad un host richiede la scrittura e l'esecuzione di uno script ad-hoc per creare, configurare e registrare l'oggetto
  - mancanza di servizi standardizzati di supporto agli oggetti
    - gli oggetti devono gestire direttamente ed esplicitamente aspetti come persistenza, notifica di eventi, transazioni, sicurezza, replicazione, bilanciamento del carico



## Alcune tecnologie a componenti

- Alcune tecnologie a componenti concrete
  - CORBA
    - Common Object Request Broker Architecture, 1991
    - definito dall'OMG, definisce un modello a componenti
  - Microsoft
    - COM Component Object Model, 1993
      - è una tecnologia specifica – ma in senso generale COM viene usato anche per indicare alcune delle sue evoluzioni, tra cui COM+, DCOM, OLE, ActiveX, ...
    - DCOM Distributed Component Object Model, 1996
    - .NET Framework, 2002
  - Java
    - EJB definisce un modello a componenti, nell'ambito della piattaforma Java EE, 1999



## Contenitori

- Un aspetto fondamentale delle tecnologie a componenti è che i componenti sono pensati per “vivere” dentro dei contenitori
  - “non esiste niente che, di per sé, è un buco [Kurt Tucholsky] – allo stesso modo, non esiste niente che, di per sé, è un componente – i componenti esistono solo grazie ai contenitori che li definiscono” [Bruce Wallace]
  - un contenitore
    - consente la comunicazione tra componenti
    - gestisce il ciclo di vita dei componenti installati
    - fornisce alcuni servizi importanti ai suoi componenti
- Descriviamo, intuitivamente, l’evoluzione che ha portato ai contenitori e, poi, i pattern architetturali che guidano l’organizzazione di un contenitore

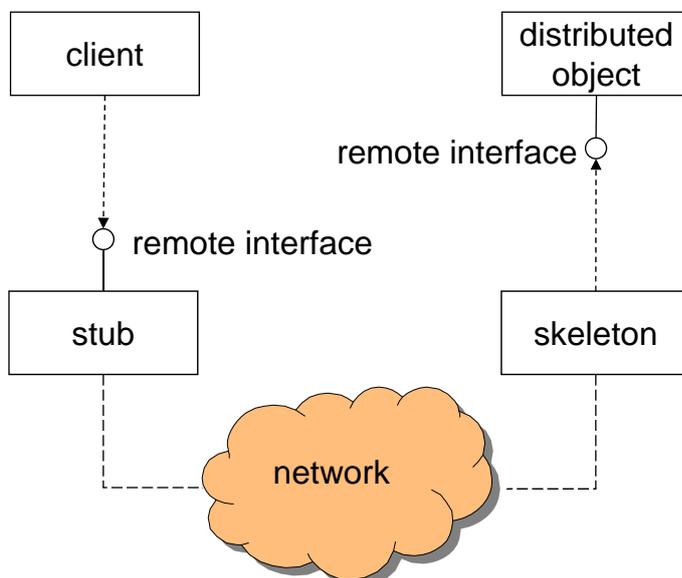
35

Luca Cabibbo – ASw



## Oggetti distribuiti

- Le DOA consentono l’interazione tra oggetti distribuiti



tutte le richieste  
remote passano  
attraverso il  
middleware

36

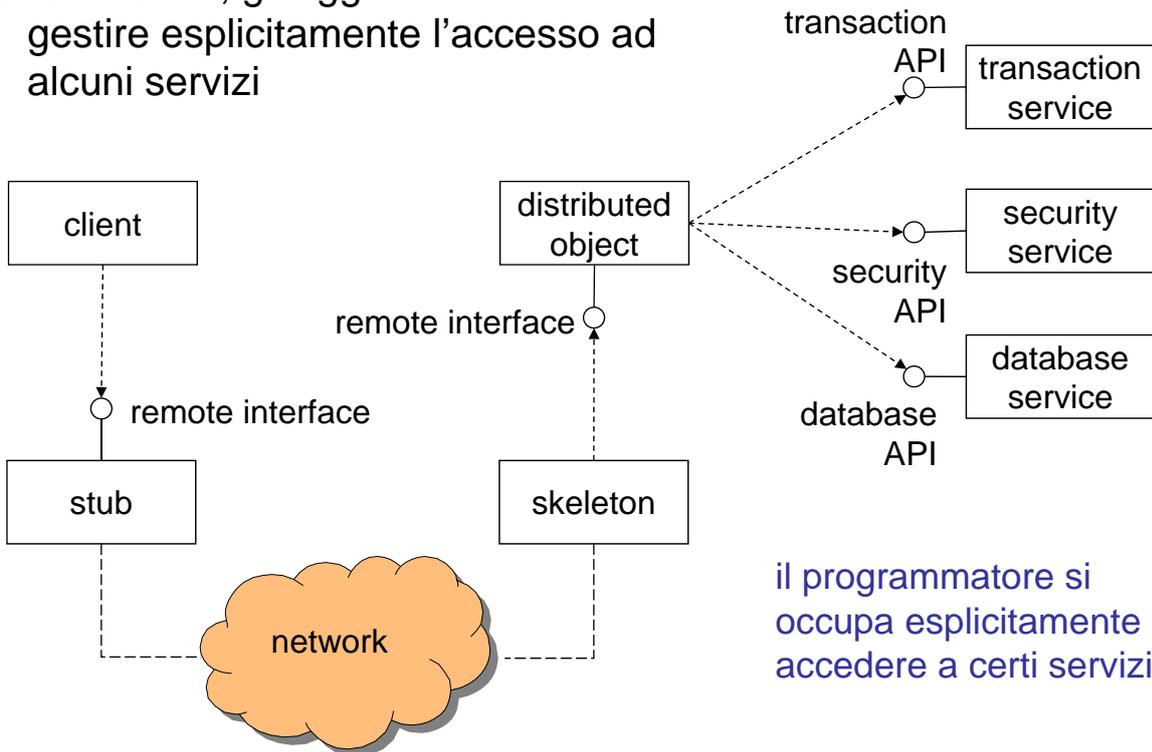
Architetture basate su componenti

Luca Cabibbo – ASw



# Oggetti distribuiti e servizi

- Nelle DOA, gli oggetti devono gestire esplicitamente l'accesso ad alcuni servizi

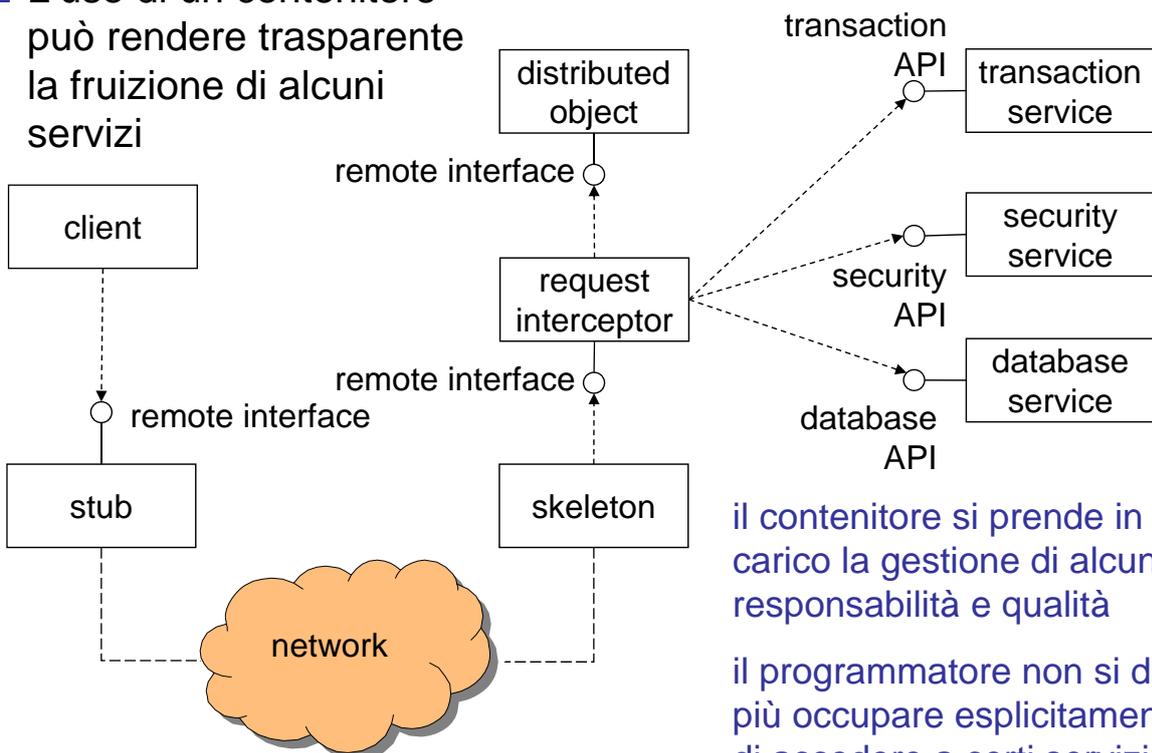


il programmatore si occupa esplicitamente di accedere a certi servizi



# Gestione trasparente dei servizi

- L'uso di un contenitore può rendere trasparente la fruizione di alcuni servizi



il contenitore si prende in carico la gestione di alcune responsabilità e qualità

il programmatore non si deve più occupare esplicitamente di accedere a certi servizi



## Componenti e contenitori

- Le tecnologie a componenti sostengono lo sviluppo e la gestione di componenti software che possono essere rilasciati, composti ed eseguiti entro ambienti di esecuzione specializzati, chiamati contenitori
  - un **componente** è un'entità software runtime
    - implementa un insieme di funzionalità
    - offre i suoi servizi mediante un insieme di interfacce con nome (interfacce fornite)
    - può richiedere servizi ad altri componenti, sempre sulla base di interfacce con nome (interfacce richieste)
  - in pratica, un'applicazione è formata da un insieme di componenti, che vengono composti (al momento del rilascio) sulla base di un'interconnessione delle loro interfacce



## Componenti e contenitori

- Le tecnologie a componenti sostengono lo sviluppo e la gestione di componenti software che possono essere rilasciati, composti ed eseguiti entro ambienti di esecuzione specializzati, chiamati contenitori
  - un **contenitore** è un ambiente runtime, lato server, per la gestione di componenti – un contenitore estende le capacità di un broker
    - i componenti, per essere eseguiti, devono essere configurati, assemblati e rilasciati in un contenitore
  - il contenitore si occupa del ciclo di vita dei componenti in esso rilasciati
    - fornisce ai suoi componenti servizi come sicurezza, transazioni e persistenza – per sostenere qualità come sicurezza, affidabilità, prestazioni e scalabilità
    - grazie a questo, lo sviluppo dei componenti può essere focalizzato sull'implementazione di funzionalità



## \* Pattern architetturali per contenitori

- Gli ambienti di esecuzione per componenti – chiamati genericamente contenitori oppure application server, specifici per un certo modello a componenti – sono realizzati sulla base dello stile architetturale **Container**
  - [POSA4] presenta questo pattern architetturale in un capitolo dedicato alla gestione delle risorse
    - per *risorsa* si intende una qualunque risorsa software, tra cui componenti, servizi distribuiti e loro istanze – ma anche connessioni di rete, sessioni nell'accesso a basi di dati, token per la sicurezza, ...
    - i pattern di questo capitolo sono relativi alla gestione del ciclo di vita e della configurazione delle risorse
    - questi pattern consentono di controllare alcune qualità importanti di un'applicazione o sistema software – tra cui prestazioni, scalabilità, flessibilità, affidabilità, sicurezza, ...



## Gestione delle risorse

- La gestione delle risorse è una capacità critica nella realizzazione di sistemi software distribuiti
  - molte proprietà di qualità di un'applicazione – tra cui prestazioni, scalabilità, flessibilità, stabilità, disponibilità, sicurezza – dipendono da come le sue risorse vengono gestite – ad esempio, da come sono create, ottenute, accedute, utilizzate, rilasciate o distrutte
    - ad esempio, le prestazioni di un server possono degradare se vengono mantenuti in memoria troppi oggetti inutilizzati
  - tuttavia, gestire le risorse in modo corretto ed efficiente è difficile – ciò che rende particolarmente difficile la gestione delle risorse è tentare di raggiungere un buon compromesso tra le diverse qualità, poiché la soddisfazione di una può essere in conflitto con la soddisfazione delle altre
    - questo motiva l'uso di pattern architetturali, provati nel tempo, per gestire le risorse in modo efficace ed efficiente



## Gestione delle risorse e qualità

- Alcune considerazioni relative alla gestione delle risorse
  - *prestazioni* – per ridurre latenza e tempo di risposta, è importante minimizzare le attività di creazione, inizializzazione, acquisizione, rilascio, distruzione e accesso alle risorse – e in generale le attività che introducono un overhead o un ritardo significativo
  - *scalabilità* – riguarda la capacità del sistema di gestire un numero di client maggiore di quello inizialmente previsto, con un impatto accettabile sulle prestazioni – anche questo richiede l'applicazione di opportune tattiche per la gestione delle risorse
  - *affidabilità/disponibilità* – per fornire un servizio senza interruzioni o inconsistenze, è importante gestire alcune risorse in modo opportuno – ad esempio, se una transazione coinvolge più risorse, il loro stato finale deve essere consistente e persistente – inoltre, le ottimizzazioni per le prestazioni e la scalabilità non devono influenzare negativamente l'affidabilità



## Gestione delle risorse e qualità

- Alcune considerazioni relative alla gestione delle risorse
  - *flessibilità* – le proprietà di qualità di un'applicazione devono poter essere selezionate al tempo di compilazione, deployment oppure di esecuzione
  - *aggiornamenti* – le applicazioni devono evolvere durante la loro vita – in alcuni casi, non è tollerabile interrompere e riavviare un'applicazione per effettuare un aggiornamento, soprattutto se ci sono requisiti stringenti di disponibilità
  - *gestione trasparente del ciclo di vita delle risorse* – i client di una risorsa dovrebbero poter usare le risorse in modo semplice – senza dover conoscere i dettagli del loro ciclo di vita (come creare le risorse? come rilasciarle?) – poiché una gestione errata del ciclo di vita delle risorse può avere effetti negativi sulle qualità del sistema, a maggior ragione è preferibile che questa attività non sia lasciata ai client delle risorse



## Pattern per la gestione delle risorse

- [POSA4] propone circa venti pattern per la gestione delle risorse – per descrivere l'infrastruttura realizzata da un contenitore – suddivisi in diversi gruppi
  - i pattern principali sono *Container*, *Object Manager* e *Component Configurator* – descrivono l'intera infrastruttura per la gestione delle risorse
  - altri pattern si occupano di diversi aspetti realizzativi per una gestione delle risorse efficace – ad es., *Resource Pool*, *Lifecycle Callback*, *Lookup*, ...
  - altri pattern ancora riguardano strategie per l'acquisizione di risorse – ad es., *Eager Acquisition* e *Lazy Acquisition* – e per il rilascio di risorse – ad es., *Automated Garbage Collection*
  - infine, alcuni pattern sono relativi ad aspetti della creazione e distruzione di risorse – ad es., *Factory Method* e *Disposal Method*



## - Container [POSA4]

- Contesto
  - ambiente di esecuzione per componenti
  - è in generale opportuno che i componenti siano disaccoppiati dai dettagli tecnici circa il loro ambiente di esecuzione



## Container [POSA4]

### □ Problema

- i componenti implementano logica di business (o infrastrutturale) – da usare per comporre una o più applicazioni
- per favorire la loro composizione, i componenti non devono essere accoppiati direttamente a particolari scenari di esecuzione né al loro ambiente tecnico di esecuzione
  - ad es., uso transazionale o meno, sicuro o meno
  - ad es., la locazione o il particolare application server
  - un componente che gestisce tali aspetti è necessariamente accoppiato alla piattaforma – inoltre la sua implementazione è più complicata del necessario
- deve piuttosto essere possibile comporre componenti in diverse applicazioni, con differenti scenari di deployment, da eseguire su differenti piattaforme – senza un esplicito intervento del programmatore



## Container [POSA4]

### □ Soluzione

- definisci un *container* (*contenitore*) per fornire un ambiente di esecuzione opportuno ai componenti
- il contenitore realizza l'infrastruttura tecnica necessaria per integrare (comporre) componenti in applicazioni – ciascuna con uno scenario di utilizzo specifico e per una piattaforma specifica – senza però accoppiare in modo stretto i componenti con le applicazioni né con gli scenari di utilizzo né con le piattaforme



## Container [POSA4]

### □ Soluzione

- il contenitore
  - fornisce un mezzo per registrare/configurare/integrare dinamicamente i componenti nel contenitore, sulla base di una specifica dichiarativa
  - inizializza e fornisce il contesto runtime per l'esecuzione dei componenti che gestisce
  - definisce operazioni per consentire le interazioni tra componenti
  - definisce e fornisce inoltre servizi di supporto ai componenti – ad es., persistenza, notifica di eventi, transazioni, replicazione, bilanciamento del carico, sicurezza, ...



## Discussione

- Il pattern *Container* sostiene e semplifica lo sviluppo e la composizione di componenti
  - consente di separare gli aspetti relativi allo sviluppo dei componenti da quelli relativi alla loro composizione e al loro deployment
  - consente al programmatore di concentrare il proprio lavoro sugli aspetti funzionali e sulla logica applicativa
    - altri interessi (di qualità) vengono invece gestiti soprattutto in sede di deployment – spesso mediante approcci dichiarativi



## Discussione

- Ma come vengono sostenuti in modo effettivo gli interessi di qualità del sistema?
  - è responsabilità del contenitore, che agisce da gestore (“manager”) per i componenti registrati
  - le diverse qualità sono sostenute nell’ambito della gestione del ciclo di vita dei componenti – garantendo le qualità richieste come specificato dalle informazioni di deployment
  - per soddisfare queste importanti responsabilità, la realizzazione di un *Container* richiede normalmente anche l’uso di ulteriori pattern, più specifici – alcuni dei quali sono descritti nel seguito



## - Component Configurator [POSA4]

- Contesto
  - nella realizzazione di un’architettura a componenti, è necessario un supporto flessibile alla configurazione a runtime dei componenti
- Problema
  - configurare prematuramente un’applicazione, in termini di un particolare insieme di implementazioni di componenti, può essere poco flessibile
  - alcune decisioni devono poter essere prese tardi nel ciclo di vita di un’applicazione – addirittura anche dopo il deployment
    - ad es., non è accettabile che un’applicazione non possa avvantaggiarsi dall’uso di componenti più nuovi o migliori
  - inoltre, nelle applicazioni in cui è richiesta alta disponibilità, il cambiamento della configurazione dell’applicazione deve avere un impatto basso sul sistema in esecuzione



## Component Configurator [POSA4]

### □ Soluzione

- separa le interfacce dei componenti dalle loro implementazioni
- organizza i componenti in opportune unità di deployment – che possono essere caricate dinamicamente
- fornisci un meccanismo (*component configurator*) per configurare i componenti in un'applicazione (e riconfigurarli in modo dinamico) senza dover interrompere e riavviare l'applicazione
- consenti la configurazione dei componenti e la loro integrazione in applicazioni sulla base di specifiche dichiarative (*declarative component configuration*) – e non sulla base di meccanismi programmatici



## Discussione

- Un *Component Configurator* offre flessibilità – consentendo il rilascio e la sostituzione delle implementazioni dei componenti per tutto il ciclo di vita di un'applicazione
  - internamente, questo elemento deve gestire un repository di componenti – ovvero, deve agire da gestore (*object manager*) dei componenti rilasciati
  - inoltre, affinché sia possibile un'effettiva riconfigurazione delle applicazioni e dei componenti, i diversi componenti devono offrire un'interfaccia per la loro amministrazione (*lifecycle callback*)



## - Object Manager [POSA4]

### □ Contesto

- nell'implementazione di un contenitore, è spesso necessario gestire l'accesso a tipi di oggetto specifici, il loro ciclo di vita, nonché le loro risorse e relazioni

### □ Problema

- per un utilizzo corretto ed efficiente, alcuni oggetti di un'applicazione (come le istanze dei componenti, ma anche risorse come thread e connessioni) richiedono un controllo dell'accesso accurato e un'opportuna gestione del loro ciclo di vita
- implementare queste funzionalità direttamente negli oggetti stessi (o in altri oggetti di supporto oppure nei loro client) ne aumenterebbe le responsabilità e la complessità, e ne renderebbe difficile l'uso e l'evoluzione



## Object Manager [POSA4]

### □ Soluzione

- separa l'utilizzo di un oggetto dal controllo del suo ciclo di vita e del suo accesso
- introduci un gestore degli oggetti (*object manager*) separato, la cui responsabilità è gestire e mantenere un insieme di oggetti
- i client, nell'effettuare l'accesso alle funzionalità degli oggetti, fruiranno in modo trasparente dei servizi del gestore degli oggetti – ovvero, cercheranno e useranno gli oggetti – ma non si occuperanno né della loro creazione né della loro distruzione
  - se un oggetto a cui è richiesta una funzionalità non esiste ancora o non è stato ancora allocato, allora il gestore degli oggetti lo creerà (o lo selezionerà da un pool di oggetti)
  - in modo analogo, il gestore degli oggetti può anche controllare la distruzione e la deallocazione degli oggetti



## - Altri pattern per la gestione delle risorse

### □ Lookup

- fornisce un servizio di registry, per registrare i riferimenti ai servizi/componenti (quando divengono disponibili) e per de-registrarli (quando non sono più disponibili)

### □ Virtual proxy

- definisce un proxy per un oggetto, anche se esso non esiste ancora in memoria – consente la creazione e l’inizializzazione dell’oggetto richiesto solo quando serve



## Altri pattern per la gestione delle risorse

### □ Resource pool

- mantiene un certo numero di istanze di una risorsa in un pool di risorse in memoria – piuttosto che creare ogni volta (e ripetutamente) le risorse da zero, le risorse vengono prelevate dal pool in modo prevedibile – quando una risorsa non è più necessaria a un’applicazione, viene rimessa nel pool, affinché possa essere disponibile per acquisizioni successive

### □ Activator

- minimizza il consumo di risorse attivando i servizi su richiesta – e disattivandoli quando non sono più acceduti dai loro client

### □ Evictor

- introduci uno “sfrattatore” per monitorare l’uso delle risorse e controllare la loro vita – le risorse che non vengono utilizzate per un certo periodo di tempo vengono rimosse – per liberare spazio per altre risorse più utili



## Altri pattern per la gestione delle risorse

### □ Lifecycle callback

- si basa su un'interfaccia che definisce un insieme di eventi fondamentali (e un insieme correlato di metodi di callback) nel ciclo di vita degli oggetti gestiti dal contenitore (o da un object manager)
  - gli oggetti (componenti) gestiti dal contenitore devono implementare questa interfaccia – per definire comportamenti specifici relativi alla loro gestione
  - il contenitore poi usa questi metodi di callback per controllare il ciclo di vita degli oggetti in modo esplicito
- il modello di programmazione dei componenti deve sostenere l'applicazione dei diversi pattern per la gestione delle risorse da parte del container – per questo motivo, è basato anche sulla definizione di questi metodi di callback



## - Discussione

- È possibile riconoscere in questi pattern architetturali alcuni aspetti relativi alla programmazione e alla gestione degli enterprise bean – una tecnologia a componenti che è stata presentata nella dispensa su componenti (middleware)
  - una conoscenza più approfondita della programmazione degli enterprise bean e della loro semantica e implementazione consente di comprendere meglio la relazione tra intercettazione delle richieste, gestione del ciclo di vita dei componenti e gestione di qualità specifiche – come prestazioni, affidabilità, scalabilità, disponibilità, sicurezza, interoperabilità, ...



## Discussione

- Alcune qualità sostenute da un contenitore EJB
  - prestazioni
    - i meccanismi indicati per la gestione degli oggetti consentono di applicare diverse tattiche per le prestazioni
    - ad es., un resource pool è un'applicazione della tattica Maintain multiple copies of computations – e supporta l'applicazione di Reduce overhead



## Discussione

- Alcune qualità sostenute da un contenitore EJB
  - sicurezza
    - basata tattiche per la sicurezza – come cifratura dei dati, autenticazioni e autorizzazioni – nonché di specifiche dichiarative e meccanismi di programmazione semplificati
    - ad es., un bean può essere protetto da credenziali, che sono controllate dal contenitore quando qualcuno effettua un accesso al bean – questo è possibile perché tutte le interazioni tra componenti sono gestite e intercettate dal contenitore



## Discussione

- Alcune qualità sostenute da un contenitore EJB
  - affidabilità
    - ad esempio, sulla base di transazioni, anche distribuite, che possono coinvolgere diversi partecipanti, come operazioni di enterprise bean, basi di dati e messaggi – il contenitore può anche agire da coordinatore di transazioni
    - di solito, il contenitore inizia una nuova transazione per ciascuna esecuzione di metodo di un bean – ma il metodo di un bean può essere eseguito anche nell'ambito della transazione del metodo che l'ha invocato – sono possibili anche transazioni multiple e annidate



## Discussione

- Alcune qualità sostenute da un contenitore EJB
  - scalabilità e disponibilità
    - sulla base di un'implementazione distribuita del contenitore su un cluster, a cui è possibile aggiungere dinamicamente nodi – e di meccanismi di replicazione



## \* Discussione

- Un uso efficace delle tecnologie a componenti richiede ulteriori considerazioni
  - considerazioni di natura *metodologica*, per guidare quanto meno le seguenti attività – da svolgere in modo congiunto
    - progettazione “statica” – identificazione dei componenti – quali componenti? in quali strati? con quali responsabilità?
    - progettazione “dinamica” – specifica dei componenti – quali interazioni tra componenti? quali interfacce per i componenti?
    - collettivamente, questa attività progettuale viene chiamata di “specifica” dell’architettura a componenti
  - considerazioni di natura *fisica*, per guidare il deployment delle architetture basate su componenti
    - per sostenere concretamente qualità come prestazioni, scalabilità e alta disponibilità



## Discussione

- Per aspetti relativi alla programmazione di componenti, vedi anche
  - dispensa su componenti (middleware)
- Per aspetti metodologici, vedi anche
  - dispensa su una metodologia per la specifica di software basato su componenti
- Per aspetti relativi al deployment delle architetture a componenti, vedi anche
  - dispensa su cluster per architetture a componenti