Java EE (Java enterprise edition)

Java EE nasce alla fine degli anni ’90 e portò il linguaggio java come un software robusto necessario per sviluppare enterprise platform.

Java EE runtime environment definisce 4 tipi di componenti che supportano:

1. Applets(Finestre) sono GUI
2. Applications sono programmi che sono eseguiti su un client
3. Web applications (fatti da servlets, filters, listener) sono eseguiti in un web container e rispondono a un http request
4. Enterprise applications sono eseguite in un EJB container. EJB container è responsabile dell’esecuzionedegli enterprise beans contenenti la logica di business della tua java ee application(servizi di sicurezza transazione)

**Package Java ee**

Per deployare un’applicazione web in un container i componenti devono essere compressi in uno specifico formato di archivio. Java se definisce come java Archive (jar)

1. In un’applicazione web java classes e gli altri file sono contenuti in file.jar. Come ogni altro formato di archivio, il jar file contiene una cartella per le meta info del progetto all’interno di questa cartella si trova un file:META-INF/manifest.mf. Il manifest.mf viene usato per definire le estensioni e il package in cui si trova la risorsa.
2. Le informazioni relative agli EJB sono contenute nel file: META-INF/ejb-jar.xml
3. Una applicazione web contiene servlet, jsp ecc. Tutti questi file sono contenuti in un apposito jar file con estensione .war. Tutti questi componenti sono definiti nel deployment descriptor nel web.xml .

Tutti questi componenti sono deployati in package (.ear)

**Paradigma della programmazione**

Nel paradigma della programmazione ci sono due approcci: programmazione imperativa e programmazione dichiarativa. La prima definisce l’algoritmo per raggiungere lo scopo (cosa deve essere fatto) e l’altra definisce come raggiungere lo scopo (come deve essere fatto).

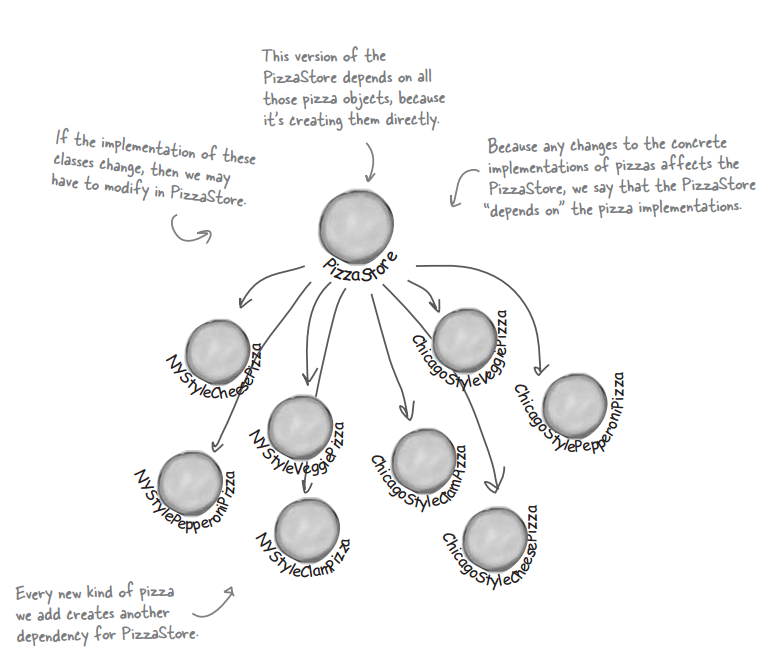
POJO Plain old java object

**Inversion of control**

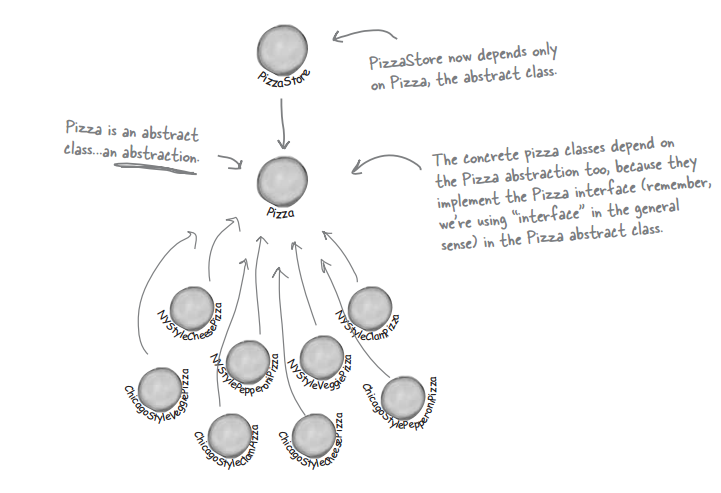
La prima versione di java ee ha introdotto il concetto dell’inversion of control (Ioc). Il principio dell’ ioc è: dipendi dall’astrattezza. Non dipendere dalle classi concrete.

Questo principio a primo impatto suona come “programmare a interfacce e non a estensione”. Tuttavia la dependency injection principle suggerisce che un componente di più alto livello non può dipendere da uno di più basso livello.

Nell’esempio sottostante indica come un high level component dipende da un low level component. In questo caso si ha un pizza store che dipende da n pizze; si ha sempre delle relazioni “has a” invece dovrebbe essere “is a” (programmazione basta sull’utilizzo delle interfacce).



L’esempio sottostante indica come sia applicato il principio dell’inversion of controller. Ora il pizza store dipende solo dall’oggetto pizza e non si sa quante pizze abbia. A loro volta le pizze custom non impongono più un dipendenza verso un’altra classe, anzi sono proprio quest’ultime che hanno una dipendenza verso l’oggetto astratto “pizza”. Infatti se si vuole aggiungere una pizzaCustom deve solo estendere pizza.



Per seguire questo concetto bisogna:

* Nessuna variabile deve riferire una classe concreta (quando si deve fare la new usare la factory per creare l’oggetto)
* Nessuna classe deriva da una classe concreata
* Nessun metodo deve overaidare un metodo già implementato in una classe base

**Dependency Injection**

DI è un design pattern che disaccoppia componenti dipendenti.

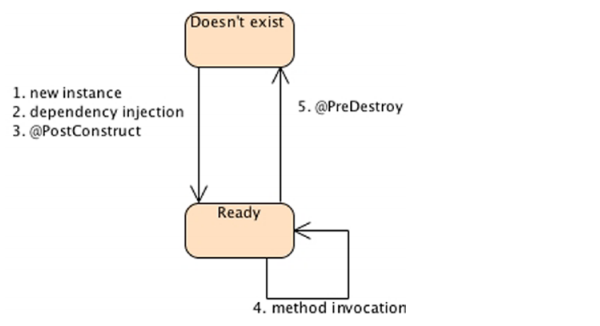


Questo è un esempio di dependency injection in quanto l’oggetto “repositoryService” viene istanziato tramite costruttore e non all’interno della classe. Questo significa che la dipendenza della classe proviene dall’esterno e non sono create internamente alla classe.

**Gestione del Life-cyle**

Il ciclo di vita di un pojo è semplice: un java developer crea l‘ istanza di una classe usando la parola new e aspetta che la garbage collector si sbarazzi di questo oggetto e liberi un po’ di memoria. Se vuoi eseguire un CDI bean dentro a un container, non ti è permesso usare la parola new. Per far ciò, è necessario iniettare il bean e il container fa il resto. Questo significa che il container è responsabile del ciclo di vita del bean: si occupa della sua creazione ed elimina l’istanza quando non è più necessaria.

Il container crea un CDI bean grazie ad una complessa infrastruttura che al suo interno fa la new dell’oggetto, richiama i metodi annotati con @postconstruct e prima di eliminare l’oggetto richiama i metodi annotai con @predestroy.

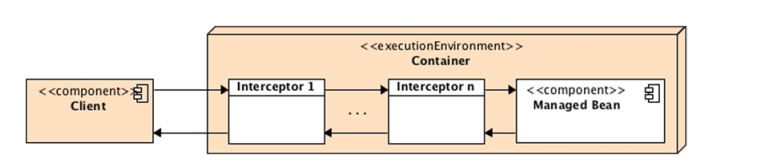


Il container gestisce automaticamente un CDI bean finchè il suo scope termina. Quando lo scope di un bean termina il bean viene distrutto.

Un session bean può mantenere attivo lo stato della conversazione con il client conservando i valori delle sue variabili interne fino al termine dello scope (session, application, requst ecc…); al contrario un bean si può limitare a fornire delle regole di business senza preoccuparsi di mantenere attivo delle informazioni relative allo stato. Nel primo caso si parla di stateles bean e nel secondo caso di stateful bean.

**Interception**

Gli interceptor sono usati per” interferire” ai metodi di business invocati. Il Managed bean supporta l’AOP e provvede la capacità di intercettare l’invocazione dei metodi. Gli interception possono essere creati prima dell’invocazione di un metodo oppure successivamente. Servono anche a eliminare parti di codice duplicati; infatti la forza degli interceptor è di disaccoppiare i concetti con la logica di business. Contestualmente la gestione del life-cycle permette a ciascun bean di gestire soltanto il proprio ciclo di vita. I beans possono usare event, notifiche per separare eventi effettuati dai consumatori di eventi.



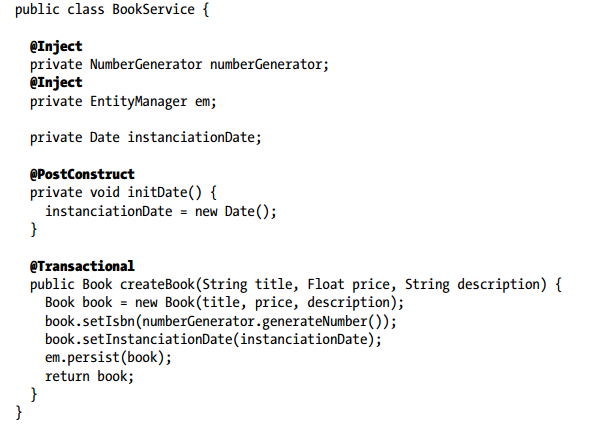
**Deployment descriptor**

Il deployment descriptor serve per definire la configurazione di un’applicazione web, in cdi il deployment descriptor è presente nel bean.xml.

**Scrittura di un CDI bean**

Un Bean di cdi può essere un genere di classe che può contenere della logica di business. Un bean è un pojo che non eredita o estende niente; possono essere iniettati altri bean al suo interno attraverso @Inject. Il suo ciclo viene gestito attraverso il container (@PostConstruct) e l’invocazione del metodo viene intercettata da @Transactional. CDI container gestisce ogni classe che soddisfa le seguenti condizioni:

1. Non è una static inner class
2. È una classe concreta oppure non ha l’annotazione @Decorator
3. Non deve avere parametri nel suo costruttore oppure si dichiara il suo costruttore con l’annotazione @Injected



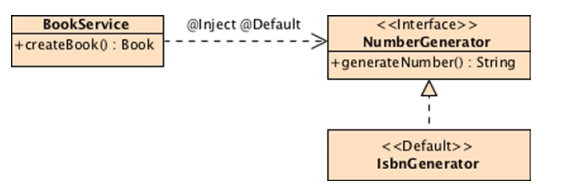
**@Inject**

Attraverso cdi non è necessario creare l’istanza di una classe a mano attraverso la new, ma sarà il container a iniettare un riferimento dell’oggetto. Questo può essere fatto annotando l’attributo con @Inject oppure con @resurce (si vedrà più avanti). L’annotazione @Inject può essere usata sia a livello di costruttore oppure a livello di set oppure a livello di attributo.

**Default Injection**

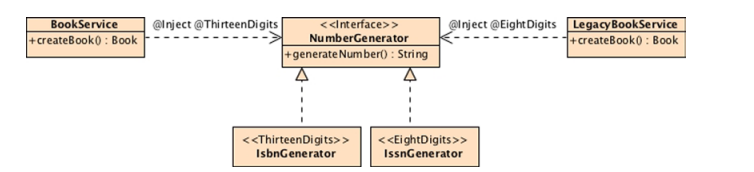


Number generator ha solo un’implementazione (IsbnGenerator). Quindi CDI è in grado di iniettare la dipendenza facilmente usando @Inject. Questo contesto è chiamato default injection. Ogni volta che un bean o un punto di injection non dichiara espressamente un qualifier, il container assume il qualifier @Default. @default è un integrato qualifier che informa CDI di iniettare l’implementazione di default del bean.



**Qualifiers**

Durante la fase di startup di un processo, il container deve validare che un bean soddisfi ciascun punto esistente di injection. Questo significa che se non esistono implementazioni di “Number generator”, il container informerà che sarà soddisfatta la injection dependency e l’applicazione non sarà deploiata. Se il container trova una sola dipendenza all’interfaccia “Number generator”, usare @default qualifier. Se esistono più dipendenze verso l’interfaccia “Number generator”, il container informerà di una dipendenza ambigua e non deplorerà l’applicazione. Questo avviene quando il container non è in grado di indentificare in modo univoco un bean da iniettare. CDI per evitare l’ambiguità di due bean che utilizzano la stessa interfaccia utilizzano @Qualifier che permette un’iniezione sicura e non ambigua.



@Qualifier è un’annotazione di java che aggiunta in un’altra annotazione, permette a quest’ultima di essere gestita dal container di CDI.

Quindi per evitare l’ambiguità bisogna:

1. Creare una nuova annotazione @EightDigits.
2. L’annotazione @EightDigits deve contenere l’annotazione @Qualifier. Questa annotazione permette all’annotazione @EightDigits di essere rilevabile dal container di CDI.
3. Annotare istanza di NumberGenerator con @Inject e @EightDigits. Attraverso queste annotazioni il container crea l’istanza della classe IssnGenerator.
4. Annotare la classe IssnGenerator con @EightDigits.

**Multi Qualifiers**

Fino ad ora, se si ha bisogno di scegliere tra due implementazioni, bisogna creare un qualifier specifico per ciascuna classe. Sopponiamo che si ha un’interfaccia che viene estensa da N classi e si vuole iniettare l’istanza utilizzando il container di CDI bisogna creare N qualifier per ciascuna classe per evitare l’eccezione di dipendenza ambigua. Per ovviare a questo problema CDI ci dà la possibilità di aggiungere delle informazioni alle annotazioni (utilizzando i parametri). Questo meccanismo creare un qualifier univoco permettendo al CDI container di iniettare la dipendenza.

In sostanza bisogna creare l’univocità dell’iniezione. Questo può essere effettuato in varie maniere:

1. Aggiungendo alle annotazione che hanno l’annotazione @qualifier delle meta-info (metodi che restituisco numeri, stringhe oppure oggetti)

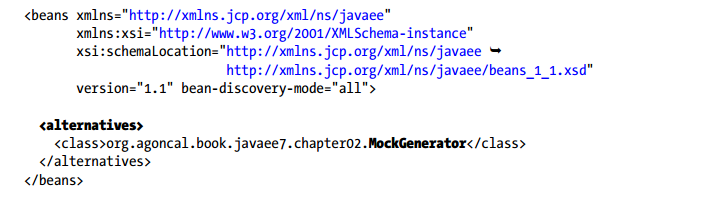


1. Aggiungendo alla classe più annotazioni



**Alternatives**

Sono dei qualificatori che permetto di scegliere tra più implementazioni di un’interfaccia quella di default. Alternatives sono dei bean annotati con @Alternative. L’implementazione della classe annotata con @Alternative non viene recepita di default da CDI. Per far si che CDI istanzi questa classe bisogna indicarla nel file xml: bean.xml nel seguente modo:



**Produces**

Fino ad ora si ha visto come iniettare CDI bean. Tuttavia con CDI non è possibili iniettare anche array, tipi primitivi, oggetti di default. Questo è dovuto al fatto che queste classi sono situate nel package rt.jar e questo package non contiene nessun bean.xml nel deployment descriptor. Per ovviare a ciò CDI ha introdotto i produces. Essi permetto di iniettare i tipi primitivi ma bisogna rispettare sempre l’univocità dell’iniezione. Ciò significa che non si può dichiarare due tipi stringa annotati con @produces. Bisogna dichiararne uno di tipo stringa e l’altro ad esempio di tipo Integer, oppure si aggiunge un altro qualifier (@custom) che permette di ottenere l’univocità. Per attivare questo meccanismo bisogna:

1. Dichiarare le variabili e annotarle con @produces

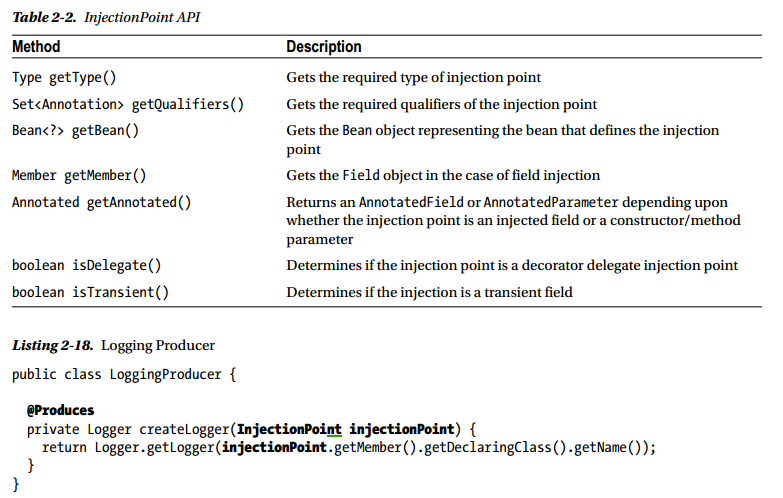


1. Iniettarle dove si necessita



**Injection Point API**

CDI ha un injectionPoint API che provvede l’accesso dei metadati del punto di iniezione.



**Disposers**

Fino ad ora si è usato produces per creare tipi di dati o POJOs per essere successivamente iniettati. Questi oggetti una volta creati e una volta usati non si è provveduto a distruggerli o a chiudere le risorse utilizzate. Alcuni metodi producer ritornano oggetti che richiedono di essere distrutti come ad esempio JDBC connection, JMS session. Quindi CDI usa i produces per creare gli oggetti e i disposes per la loro distruzione. Un metodo dispose permette a un’applicazione di provvedere la personalizzazione della pulizia di un oggetto restituito da un metodo producer.

Ogni metodo di dispose, annotato con @Disposes, deve avere un Disposed parametro dello stesso tipo che corrisponde al producer che ritorna dal metodo annotato con @Produces.



**Scopes**

Con CDI, un bean è legato a un contesto e questo legame rimane fino a quando il bean viene distrutto dal container. Non c’è modo di rimuovere manualmente un bean dal context. CDI definisce i seguenti scopes e dà delle aperture per poterli estenderli.

Appication scope(@ApplicationScoped): Si estende per tutta la durata di un’applicazione. Il bean viene creato una sola volta e ha valenza per la durata di un’applicazione e viene scartato quando l’applicazione viene chiusa.

Sessione scope(@SessionScoed): Si estende per tutta la durata della sessione di un’utente.

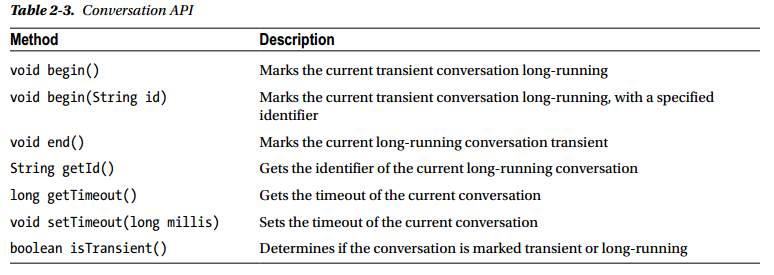
Request scope(@RequestScoped): Corrisponde a una singola http requests o alcuni metodi invocati da un singolo utente.

Conversation scope(@ConversationScoped) Si estende tra invocezioni multiple dentro la sessione; l’inizio e la fine vengono determinati dall’applicazione.

Dependent pseudo-scope (@Dependent): Il ciclo di vita è lo stesso del client. Un bean dipendente è create ogni volta che è iniettato e la referenza è rimossa quando il target dell’iniezione è rimossa. @Dependent è il default scope di CDI.

**Conversation**

La conversation scope è leggermente differente rispetto all’application, session o request scope. Essa mantiene lo stato associato con un utente, si estende su più richieste, ed è delimitata dall’applicazione. Un bean @ConversationScoped può essere utilizzato per un processo a lunga durata dove c’è un inizio definito e una fine che avviene quando si verifica l’evento atteso come ad esempio il check-out di un negozio.



**Beans in Expression Language**

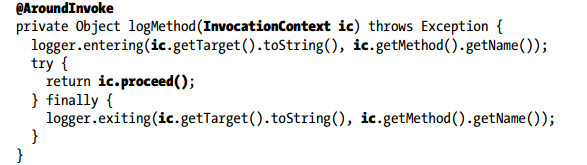
CDI è typesafe e non dipende dai nomi delle classi. Quindi i bean non sarebbero risolvibili nelle EL nelle pagine di JSF. CDI viene incontro a questa necessità attraverso l’annotazione @Named. Questa annotazione permette di dare un nome al managed Bean: il nome potrà essere utilizzato all’interno del container, per esempio da JSF via Unifield EL. Il parametro da dare a Named è una stringa; questo parametro è opzionale: se non è presente allora il nome dell’istanza è il nome della classe con la prima lettera minuscola.

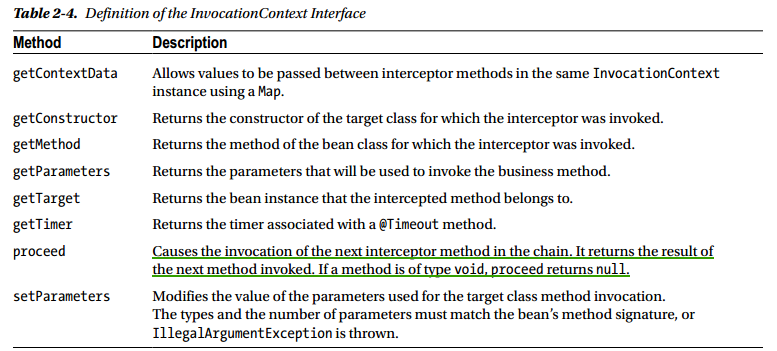
**Interceptor**

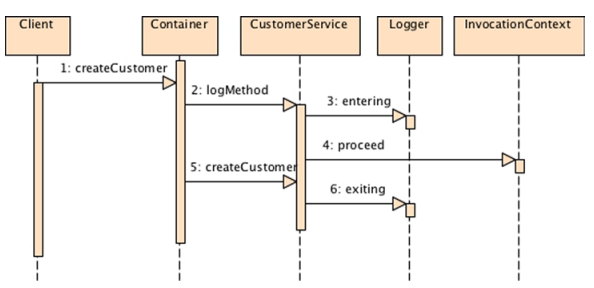
Gli interceptor vengono usati per separare l’implementazione dei cosiddetti crosscutting concepts dai concetti di business.

Gli interceptor servono ad eseguire operazioni indipendenti dai controller in maniera completamente disaccoppiata evitando di ripetere codice nel momento in cui una funzionalità deve essere resa disponibile a più URL. Esistono 4 interceptor:

1. Livello di costruttore: Interceptor associati con il costruttore della classe target @AroundConstruct
2. Livello di metodo: Interceptor associati con uno specifico metodo di business @AroundInvoke
3. Metodi di timeout: Interceptor che si frappone sul timeout dei methodi @AroundTimeout (usati solo con servizi EJB di timer)
4. Life –cycle successivo alla creazione dell’istanza @PostConstruct oppure prima della distruzione dell’istanza @PreDestroy



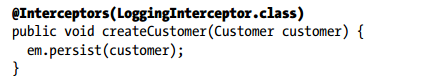
Il container intercetta le chiamate al metodo (createCustomer) e successivamente chiama questo metodo. Prima invoce il metodo (logMethod). Questo metodo usa l’invocationContext per ottenere il nome del bean invocati (metodo getTarget) e il nome del metodo intercettati (getMethod). Successivamente viene invocato il metodo proceed. Attraverso questo metodo il container invoca il successivo interceptor oppure il metodo di business; se non viene invocato questo metodo la catena di interceptor viene bloccata e non viene invocato nessun metodo di business. Quando viene invocato il metodo CreateCustomer e quando finisce l’esecuzione di questo metodo viene invocato logger.exiting solo perché viene indicato nel finall



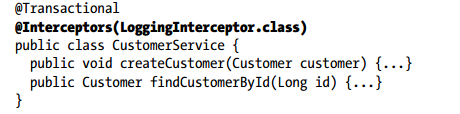
Notes: @Transactional è usata per controllare i “confini” della transazione dei bean di CDI.

**Class Interceptor**

Fino ad ora si è definito un interceptor che è solo disponibile per la classe CustomerService, in quanto è stato dichiarato al suo interno. Tuttavia molte volte è necessario isolare i concetti in una specifica classe e quindi bisogna chiedere al container di CDI di intercettare le chiamate a molti beans. Logger è il tipo esempio: è una situazione nella quale si vuole loggare uno specifico messaggio a tutti i metodi di un bean.

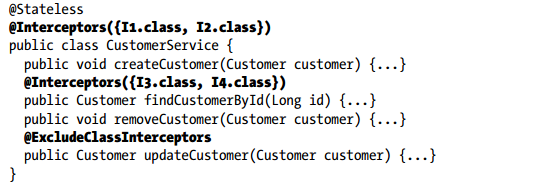


L’esempio sopra esposto indica come dice al container di intercettare un singolo metodo utilizzando un interceptor che si trova in un’altra classe

Se si vuole intercettare tutti i costruttori e i metodi bisogna indicare @Interceptors sul nome della classe.

**Catene ed esclusione di interceptor**

Fino ad ora si è visto come intercettare le chiamate ad un singolo bean (con @AroundInvoke) e a più beans (@Interceptors). Infatti, l’annotazione @Interceptors permette di aggiungere più di un interceptor, inserendo una lista di interceptor come parametri in questa annotazione. L’ordine in cui vengono invocati è determinato dall’ordine in cui vengono specificati.



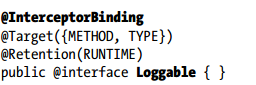
Quando un client chiama il metodo updateCustomer, nessun interceptor viene invocato in quanto il metodo in esame è annotato con @ExcludeClassInterceptors. Quando viene invocato il metodo createCustomer, Interceptor I1 e I2 viene invocato. Quando viene chiamato il metodo findCustomerById vengono richiamati gli interceptor I1, I2, I3, I4.

**Interceptor Binding**

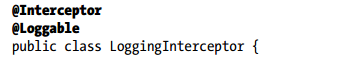
CDI ha esteso gli interceptor e ha aggiunto gli interceptor binding. Questi possono essere usati soltanto quando CDI è attivo. Con questi interceptor CDI ha introdotto l’eliminazione di dipendenze fra classi. Fino ad ora si è visto come implementare gli interceptor in modo diretto sull’implementazione del bean (@Interceptors(LoggingInterceptor.class)). In questa maniera non viene perso il legame tra le classi.

Per creare un interceptor Binding bisogna:

1. Creare un’annotazione che abbia al suo interno (@InterceptorBinding)



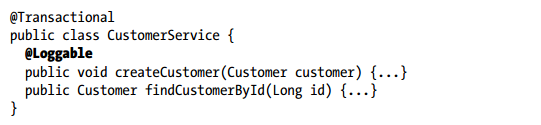
1. Annotare la classe che si vuole usare come interceptor con l’annotazione precedentemente create e con @Interceptor



1. Inserire all’interno del file bean.xml la classe che si vuole utilizzare come interceptor

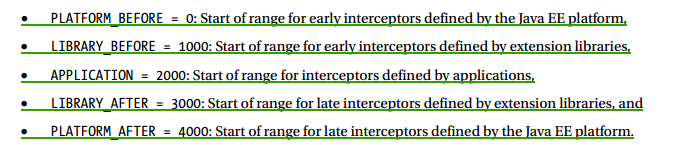


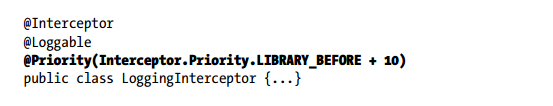
1. Annotare la classe che si vuole intercettare con l’annotazione (@Loggable quella creata) sui vari livelli: metodo se si vuole far intercettare solo la chiamata al singolo metodo, oppure sulla classe se si vuole intercettare (tutti i suoi metodi).



**Priorità Interceptors Binding**

Interceptor binding porta a un livello di astrattezza, tuttavia si perde la possibilità di ordinare l’esecuzione degli interceptor. CDI viene in contro a questa necessità attraverso l’annotazione @Priority, attraverso la quale si dà una priorità agli interceptor. Questa annotazione ha come parametro un tipo Integer. Gli interceptor con priorità più bassa sono richiamati per prima. Esistono 4 tipi di priorità di interceptor:





**Decorator**

Decorator è un design pattern. L’idea è prendere una classe e wrappare un’altra classe attraverso questa classe. In questa maniera, quando si chiama una classe decorata, si passa sempre attraverso il decoratore prima di raggiungere la classe di destinazione. I decoratori sono destinati ad aggiungere della logica a un metodo di business.

I decoratori devo avere le seguenti caratteristiche:

1. I decoratori devono avere lo stesso supertipo dell’oggetto che si vuole decorare
2. Si possono usare uno o più decorator per wrappare un oggetto
3. Nel decoratore bisogna avere l’istanza dell’oggetto che si vuole decorare che viene istanziata attraverso il meccanismo di delegate (tramite costruttore set)
4. Il decorator aggiunge metodi prima e/o dopo il meccanismo di delegate

Per Realizzare un Decorator bisogna:

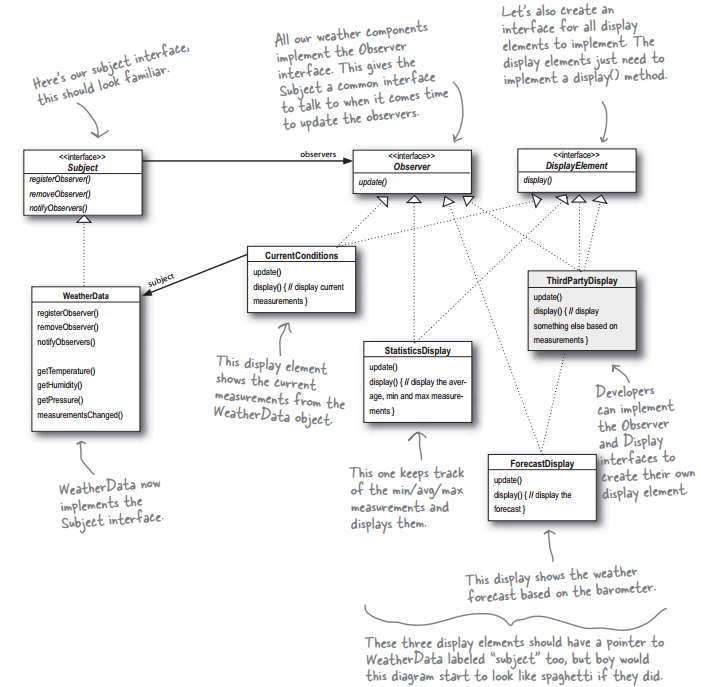
1. Annotare la classe che deve decorare un’altra classe con @Decorator
2. Annotare l’istanza della classe decorata con @Injection @Delegate
3. Indicare nel file bean.xml la classe decorator

Se si hanno più istanze che implementano la stessa interfaccia e si vuole decorare una sola di queste classi bisogna ovviamente utilizzare un qualifier. Quindi nel delegate per ottenere questa istanza bisogna annotarlo con l’annotazione qualifier appena creata.

**Events**

DI, alternatives, interceptors e decorators aggiungono un livello di astrattezza eliminando la dipendenza fra classi, aggiungendo metodi sia al runtime che a livello di compile. Gli events permettono ai bean di emettere e consumare eventi: in questa maniera è possibile sincronizzare lo stato di differenti bean in maniera completamente disaccoppiata.

Un bean può definire un event, un altro bean definisce un altro event. I beans possono essere in diversi package e l’event in un’altra parte dell’applicazione. Questo schema base segue l’observer/observable design pattern.

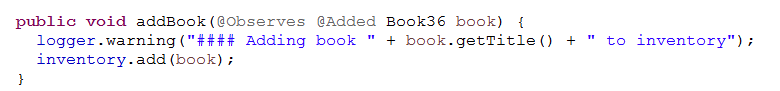


Gli event sono prodotti utilizzando l’interfaccia Event. Un producer chiama un event attraverso il metodo fire() passandogli l’oggetto event, e non deve dipendere da un observer.

Gli eventi sono prodotti dai event producer e contengono event observers. Un observer è un bean con uno o più metodi observer. Ciascuno di questi observer method prende un evento di un tipo specifico come parametro che è annotato con @Observers e opzionalmente con un qualifier. Il metodo observer è notificato da un evento se l’oggetto dell’evento meccia con il tipo dell’evento e dei qualifiers.

Per realizzare un event bisogna:

1. Dichiarare il tipo dell’evento come parametro e annotarlo con @Observes



1. Iniettare la dipendenza

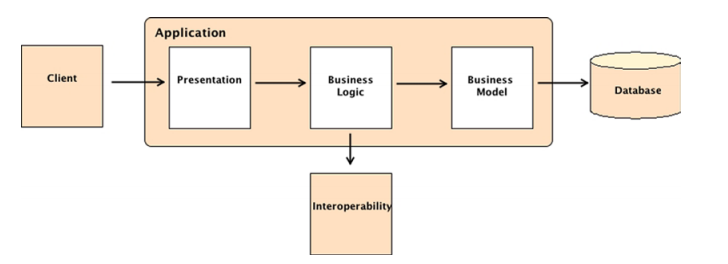


1. Notificare quando si verifica l’evento specifico gli observer con il metodo fire()

**Bean validation**

La validazione dei dati è fondamentale nelle applicazioni web in quanto ogni volta che si creare un form bisogna controllare quali campi devono essere obbligatori e se devono rispettare delle determinate regole. Molte volte la stessa logica di validazione è implementata in molti layer (presentation / database) è questo provoca la duplicazione del codice. Java ee viene incontro a questa necessità e a creato i bean validation utilizzando le metainfo.

Gli sviluppatori spendono molto del loro tempo ad essere certi che i dati processati siano validi. Gli sviluppatori scrivono dei vincoli ai dati, questi vincoli vengono applicati per la logica di business e per il modello dei dati, e devono assicurarsi che i diversi strati soddisfano questi vincoli in modo coerente.



**Application-level validation** può essere presente in molti posti per garantire che i dati siano corretti.

* Presentation layer: In questo layer si valida i dati perché i dati possono essere stati inviati da diversi clients (web browser, dall’ url)
* Business logic layer: Questo strato orchestra le chiamate ai propri servizi interni ed esterni, per

il vostro modello di dominio. . . così i dati elaborati devono essere valido.

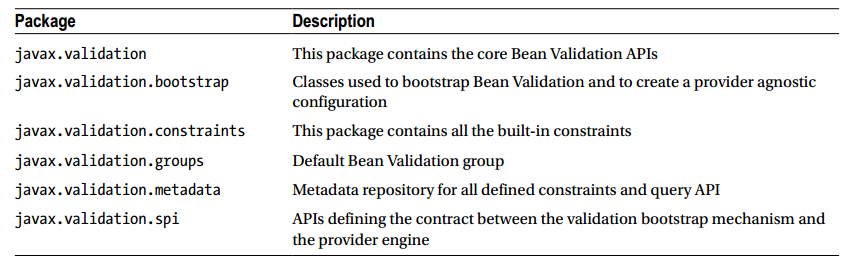
* Business model layer: Questo layer di solito viene utilizzato per validare i bean prima di essere salvati sul database

**Database**

La validazione dei dati prima di renderli persistenti è importante in quanto si evitano di far effettuare controlli di validata al db che causano errori sql. Questo provoca anche un peggioramento delle performance dell’applicazione web.

**Interoperability**

Spesso le applicazioni enterprise hanno bisogno di scambiare dati con sistemi esterni. Queste applicazioni business-to-business ricevono dati in qualsiasi tipo di formato, li elabora, li conserva e li rimanda indietro. Oggi gli XML è il linguaggio utilizzato per lo scambio di dati tra sistemi eterogenei. L’XML può usare XSD per vincolare la struttura dei documenti XML. L’XSD esprime un insieme di regole a cui un documento XML deve essere conforme al fine di essere considerato valido in base a tale schema.

Il package delle api validation è composto nel seguente modo:

**Informazioni varie**

Per ricavare le classi che implementano un’interfaccia / una classe astratta bisogna fare questo:

ClassLoader classloader = Thread.currentThread().getContextClassLoader();

ServiceLoader<ValidationProvider> loader = ServiceLoader.*load*( ValidationProvider.**class**, classloader );

creare un file con il nome e il package dell’interfaccia e aggiungere le classi che implementano quell’interfaccia

caricare un file xml vedere classe ValidationXmlParser

inputPath= META-INF/validation.xml

classloader.getResourceAsStream( inputPath );

**Constraint annotation**

Un vincolo di un java bean si esprime con una o più annotazioni. Se il retention policy contiene runtime e se l’annotazione è annotato con @Constraint (che referenza quali classi di vincoli implementa).

Un’annotazione constraint contiene alcune meta-annotazioni:

* @Target: specifica il target a cui l’annotazione può essere utilizzato.
* @Retention: specifica come verrà gestito l’annotazione (è obbligatorio l’uso del paramento runtime per consentire al provider di ispezionare gli oggetti in fase di esecuzione).
* @Constraint: specifica la classe che specifica l’algoritmo di convalida.
* @Documented: questa meta-annotazione opzionale specifica che questa annotazione sarà inclusa nella javadoc oppure no.

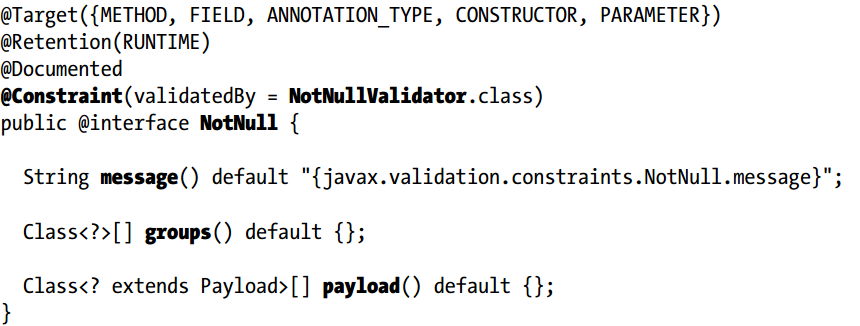
Le specifiche dei Bean validation richiedono tre attributi aggiuntivi

* Message: questo attributo permette di restituire un messaggio specifico quando la condizione del vincolo non è soddisfatta.
* Groups: Groups sono tipicamente usati per controllare l’ordine con cui i vincoli sono valutati, o per permettere una validazione parziale.
* Payload: Questo attributo è usato per associare meta informazioni con un vincolo

**Constraint implementation**

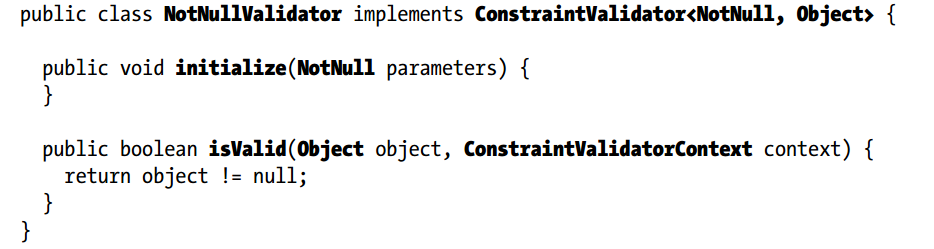
Constraint sono definiti da una combinazione di un’annotazione e zero o più classi di implementazioni. Le classe di implementazione sono specificate attraverso l’annotazione @Constraint che accetta il parametro vaidateBy in cui viene specificata la classe che implementa il vincolo, nella quale viene specificato la regola del vincolo.

L’esempio sottostante mostra come viene implementata il vincolo not null. Se crea l’annotazione e successivamente la classe che specifica come implementare il vincolo. La classe NotNullValidator implementa ConstraintValidator e utilizza i generics per passare il nome dell’annotazione e il tipo di annotazione da applicare.



The ConstraintValidator interface definisce due metodi che devono essere implementati dalla classe concreta:

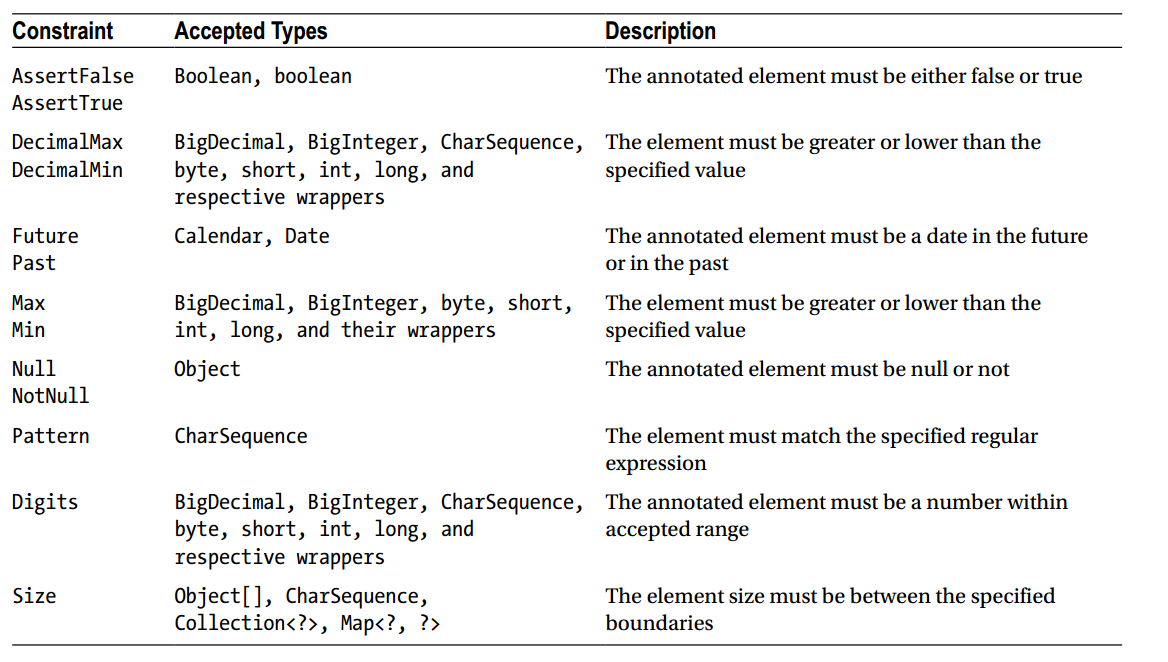
* Initialize: questo metodo è chiamato dal provider bean validation prima dell’uso di qualsiasi vincolo.
* isValid: questo metodo definisce l’algoritmo di validazione. Viene richiamato dal provider bean validation che valida un valore. Ritorna true se il valore soddisfa i controlli e ritorna false se non li soddisfa.



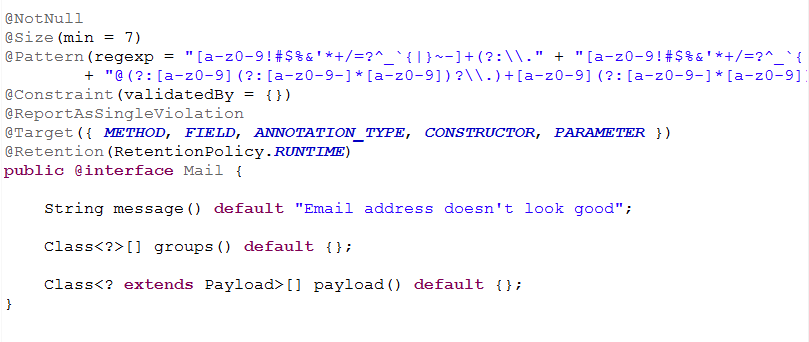
Una volta che si ha un’annotazione e un’implementazione, possono essere applicate a un vincolo o a un tipo di elemento (attributo, getter, costruttore, parametro return value, bean, interfaccia o a un’annotazione). Questa è una decisione di progettazione che i programmatori devono effettuare mediante l’uso dell’annotazione @Target(ElementType.\*) meta-annotation:

* FIELD: attributi di una classe
* METHOD: getter vincolati e i valori di ritorno del metodo vincolante
* CONSTRUCTOR: vincoli legati alla costruzione dell’oggetto
* PARAMETER: vincoli applicati ai parametri dei costruttori e dei metodi
* TYPE: vincoli dei beans, interfacce e alle superclassi
* ANNOTATION\_TYPE: un’annotazione può essere applicata a un’altra annotazione

Le annotazioni che possono essere utilizzate con le api validation (vedi comunque la documentazione):



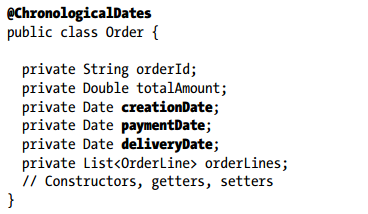
**Composizione di un vincolo**

Per creare un nuovo vincolo è aggregando quelli già esistenti senza avere una classe di implementazione. Questo è possibile utilizzando @Target(ElementType.ANNOTATION\_TYPE), un’annotazione può essere applicata a un’altra annotazione. Questo meccanismo è chiamato “constraint composition” e permette di creare vincoli di più alto livello.

Se si vuole avere un singolo errore per il vincolo @Mail, invece di averne più di uno si utilizza l’annotazione @ReportAsSingleViolation.

**Class-Level Constraint**

Fino ad ora si è visto applicare i vincoli sugli attributi o sui getter. Ma si possono creare vincoli validi per un’intera classe. L’idea è di creare un vincolo che è basato a diverse proprietà di una classe.



**Method-Level Constraint**

Questi vincoli sono dichiarati sul metodo. Questi metodi possono essere aggiunti al parametro del metodo oppure al valore di ritorno del metodo. I getter non sono considerati method-level constraint ma come filed.

**Groups**

Quando un bean è valido significa che tutti i vincoli a cui è legato sono validi nello stesso tempo. Se invece si ha bisogno di una validazione parziale di un bean oppure di controllare l’ordine con cui i vincoli sono richiamati come bisogna procedere? Questo è reso possibile utilizzando i groups che permetto di aggiungere delle restrizioni ai vincoli quando vengono eseguiti.



**Contraint Inheritance**

Molte volte un modello di business è ereditario. L’ereditarietà dei vincoli funziona come l’ereditarietà in java. Questo significa che quando un bean estende un altro bean, i vincoli del bean esteso verranno ereditati.

