PC Ready: Iterazione 2

Sommario

[Introduzione alla seconda iterazione 1](#_Toc31843787)

[Modello di dominio 3](#_Toc31843788)

[Diagramma delle classi di progetto 3](#_Toc31843789)

[Applicazione del pattern Composite 4](#_Toc31843790)

[Logica di compatibilità delle componenti 5](#_Toc31843791)

[Introduzione della classe catalogo 8](#_Toc31843792)

# Introduzione alla seconda iterazione

Terminata la prima iterazione si è ottenuto un programma in grado di far assemblare una configurazione, sia ad un cliente sia ad un amministratore, grazie alla scelta di componenti raggruppate per categoria, mediante la logica esposta nei primi due casi d’uso.

Tuttavia, la funzione di controllo della compatibilità delle componenti non è stata ancora sviluppata: uno degli obbiettivi fondamentali di tale iterazione è dunque la definizione della “logica di controllo” della configurazione creata da un amministratore/cliente.

Un altro punto fondamentale della suddetta iterazione è nato dopo un’attenta analisi del diagramma delle classi di progetto ottenuto al termine della prima iterazione: si è ritenuto opportuno ridurre le responsabilità della classe SistemaPCReady.

La classe SistemaPCReady con l’implementazione di nuovi casi d’uso potrebbe possedere un numero eccessivo di funzioni, e di conseguenza un massiccio carico di responsabilità tipico di una programmazione “artigianale” (problema di bassa coesione).

La soluzione proposta per risolvere il problema esposto in precedenza è un diverso “modus operandi” del pattern “Controller”: si è pensato di generare due controllori detti Handler di caso d’uso (invece della precedente soluzione di tipo facadeController) per UC1 e UC2, i quali si prenderanno la responsabilità dell’esecuzione delle funzioni di assemblaggio della configurazione, rendendo la classe SistemaPCReady più snella e quindi si evita un conflitto con il pattern “High Cohesion”.

In tale iterazione sono state apportate modifiche anche al precedente modello di dominio, in particolare alla classe Categoria, la quale è stata ridotta ad un semplice attributo della classe Componente.

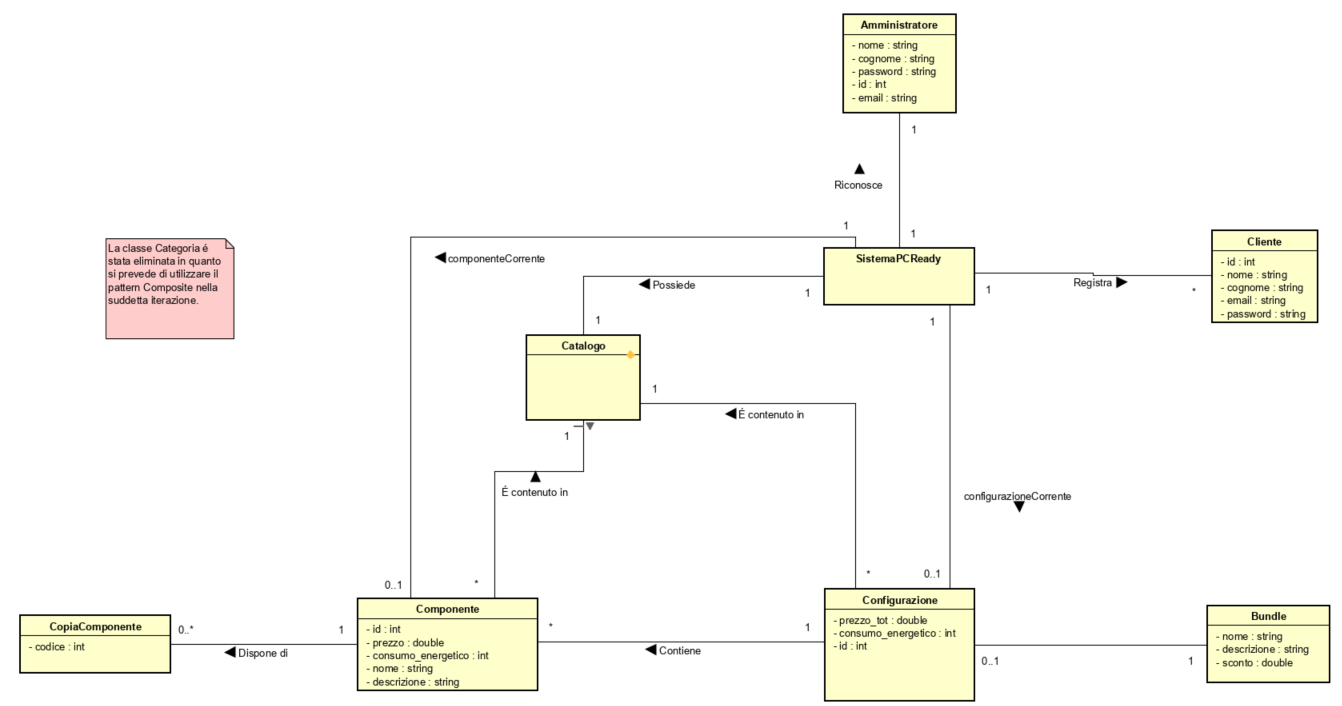
Le precedenti responsabilità della classe Categoria, tra le quali vi era il compito di restituire al sistema tutte le componenti di una determinata categoria, sono state affidate ad una classe Catalogo, la quale ha il compito di conoscere tutti i componenti che sono disponibili per l’acquisto o per la creazione di una configurazione.

Infine, in tale iterazione si è deciso di applicare il pattern Composite per gestire la Configurazione: quest’ultima difatti è composta da classi specializzate di Componente permettendo un uso più che logico di tale pattern.

Ecco un breve elenco puntato dei contenuti previsti per tale iterazione:

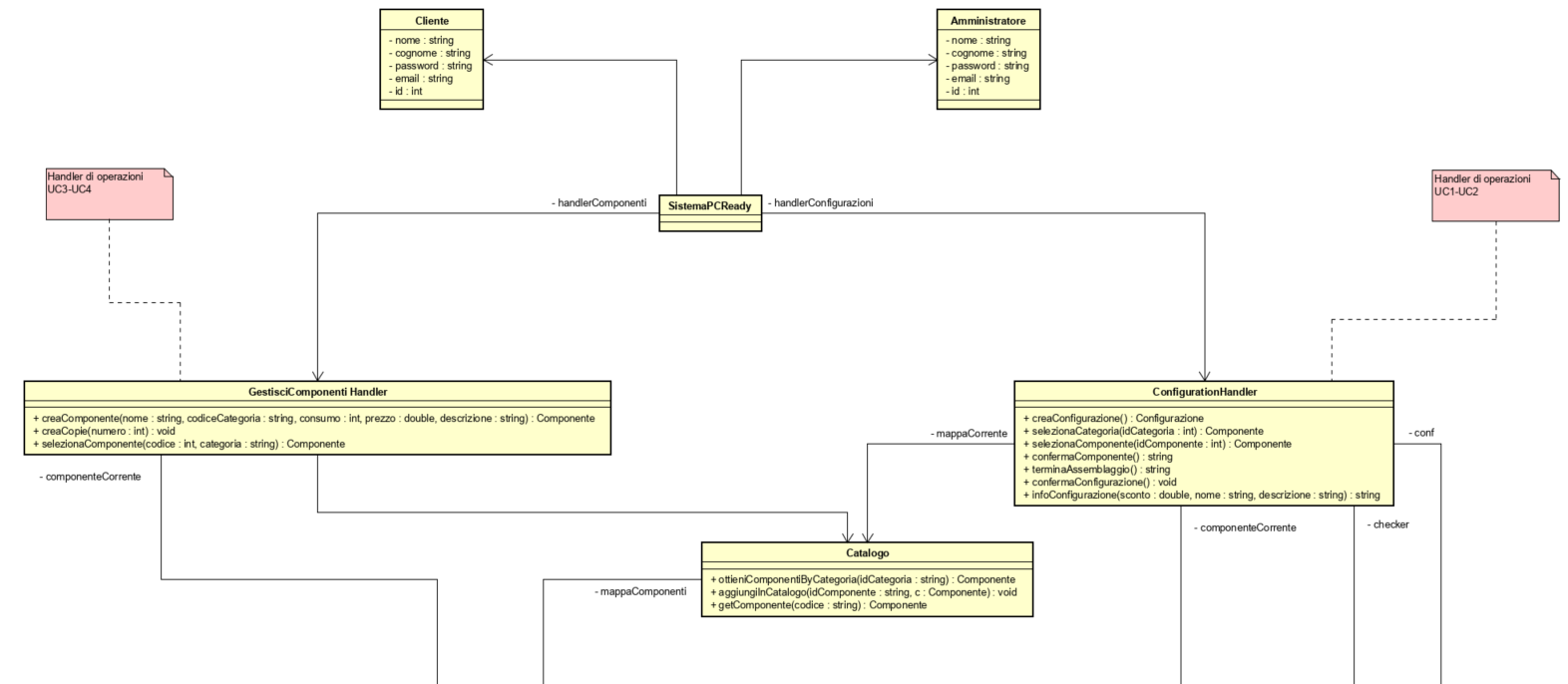
* Sviluppo della logica di compatibilità delle componenti di una configurazione così come previsto in UC1 e UC2.
* Riduzione delle responsabilità della classe SistemaPCReady mediante l’uso di controller di caso d’uso (by pattern Controller).
* Rimozione della classe Categoria e introduzione della classe Catalogo, con conseguente introduzione dell’attributo “categoria” per la classe Componente.
* Applicazione del pattern Composite per la gestione della Configurazione e delle specializzazioni della classe Componente.
* Correzioni e migliorie del codice dell’iterazione precedente.

## Modello di dominio

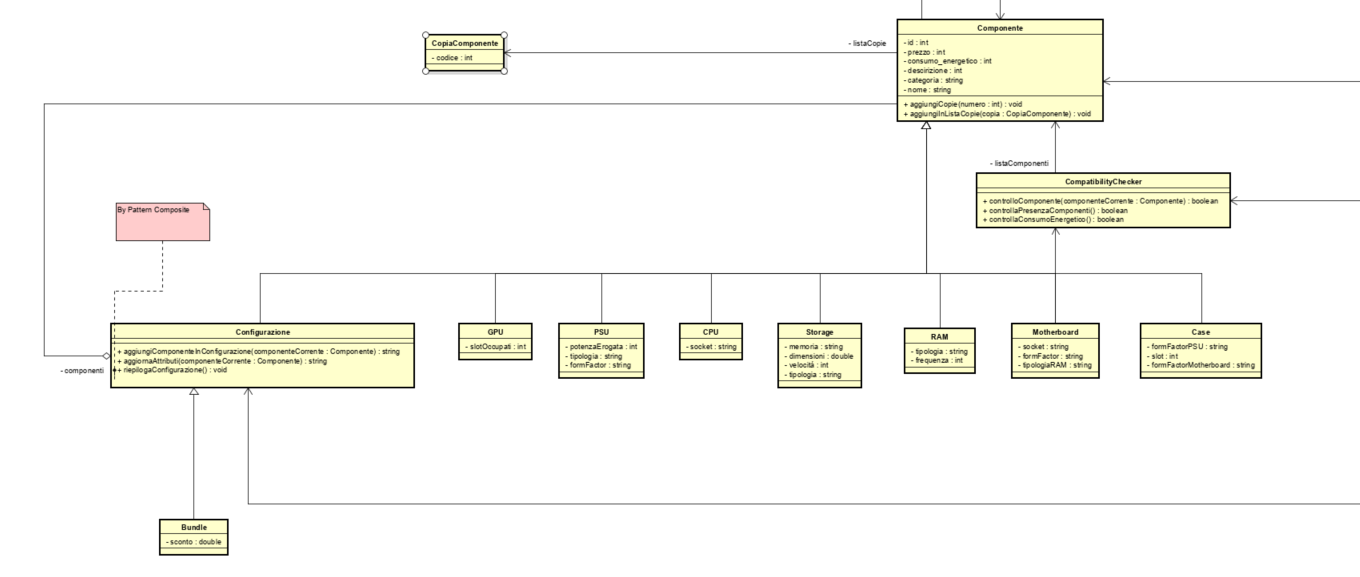


## Diagramma delle classi di progetto

Ecco la parte superiore del diagramma delle classi di progetto.



Ecco il dettaglio della parte inferiore.

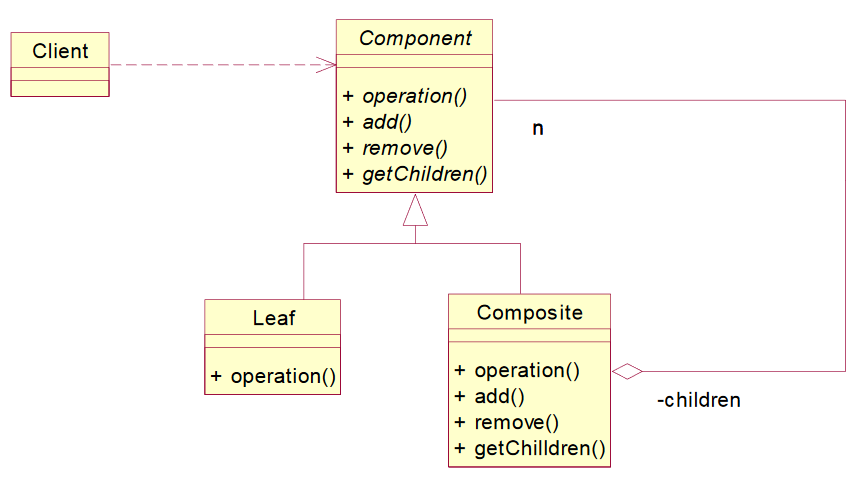


Per osservare nel dettaglio il diagramma si consiglia di consultare il file Astah presente nella directory della seconda iterazione.

# Applicazione del pattern Composite

Così come si può osservare dal diagramma delle classi di progetto della prima iterazione, la classe Configurazione è data dall’unione di più componenti necessarie per il funzionamento reale di un vero PC Desktop.

Dunque, mediante il pattern Composite si consente al cliente di trattare in modo uniforme la configurazione, come se essa fosse un oggetto semplice!



Nel nostro caso la classe “Leaf” corrisponde alle classi specializzate di componente come “CPU” e “RAM”, la classe Component risulta essere la classe “Componente” (classe generica), e la classe Composite risulta essere la classe “Configurazione”.

Mediante l’uso di questo pattern si sottolinea come “Configurazione” contiene una lista di “Componente”, e allo stesso tempo “Configurazione” diventa una sottoclasse di “Componente” e può essere trattata come tale.

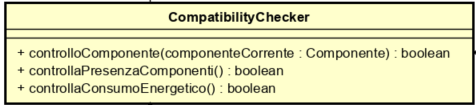
Questo pattern, oltre ad avere un beneficio attuale dato dalla gestione semplificata di una struttura composta, consente di semplificare la gestione futura del Catalogo: tale classe tratterà una configurazione e una componente come se fossero la stessa cosa, e quindi potrà usufruire di un unico set di funzioni generiche per ottenere sia componenti di una categoria specifica sia delle configurazioni/bundle.

# Logica di compatibilità delle componenti

Una configurazione per potersi definire funzionante deve soddisfare i seguenti requisiti:

* Ciascun componente deve essere compatibile con il resto dei componenti presenti nella configurazione.
* La configurazione deve possedere almeno una delle seguente componenti:
  + CPU
  + GPU
  + Motherboard
  + RAM
  + PSU
  + Storage
  + Case
* La configurazione deve possedere delle componenti il cui consumo energetico sia inferiore o uguale alla potenza fornita dal PSU selezionato.

Tali punti fondamentali sopra evidenziati sono stati tradotti in delle “funzioni di controllo compatibilitá”, le quali hanno portato una modifica ai precedenti diagrammi di sequenza, e all’introduzione di una classe a cui viene delegata l’esecuzione di tali funzioni di controllo denominata “CompatibilityChecker”



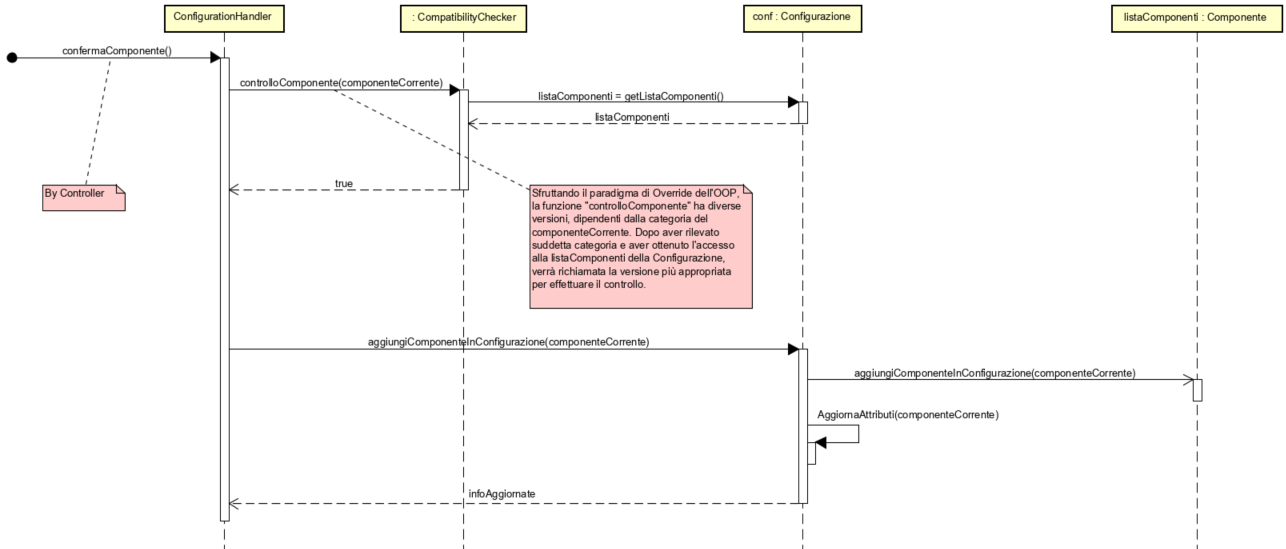
Tale classe è stata collegata con tutte classi che estendono “Componente”, con “Componente” e con “Configurazione”, in quanto deve essere in grado di riconoscere un componente specializzato e la configurazione attuale, in modo tale da poter richiedere a quest’ultima la lista di componenti attualmente selezionati.

Ecco una rapida spiegazione delle funzioni di controllo presenti in tale classe:

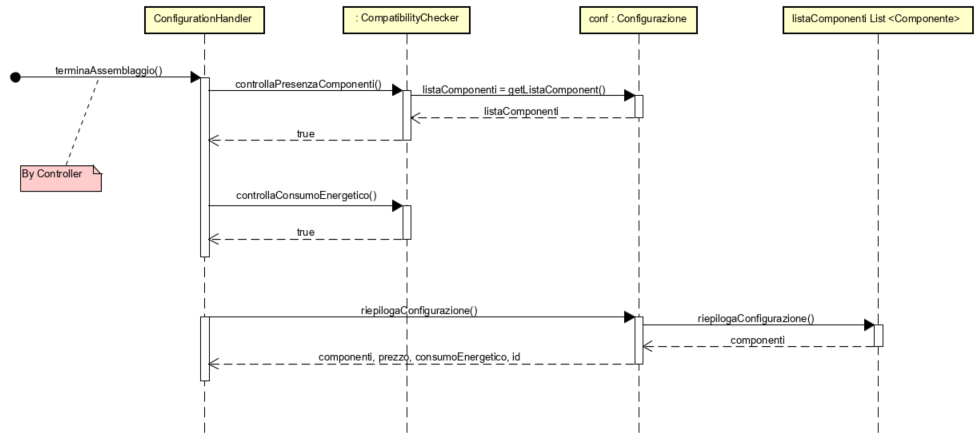
* **controlloComponente(componenteCorrente : Componente):** tale funzione ha il compito di eseguire una serie di controlli di compatibilità (eseguiti in base alla classe del componente corrente) del componente attuale con i componenti presenti nella “listaComponenti” della configurazione attuale.
  + Tale funzione viene eseguita alla conferma del componente da inserire prima dell’inserimento di quest’ultimo nella configurazione
* **controllaPresenzaComponenti():** tale funzione ha il compito di controllare l’esistenza di almeno un’istanza di ciascuna delle classi fondamentali per il funzionamento della configurazione.
  + Tale funzione viene eseguita alla conferma della configurazione attuale, prima che essa venga salvata in memoria.
* **controllaConsumoEnergetico():** tale funzione ha il compito di calcolare il consumo energetico totale di tutti i componenti presenti nella listaComponenti della configurazione per comprendere se esso puó essere soddisfatto dalla potenza fornita dal PSU scelto.
  + Tale funzione viene eseguita alla conferma della configurazione attuale, prima che essa venga salvata in memoria.

A seguire i diagrammi di sequenza nel caso di successo, con le funzioni sopra citate:

* confermaComponente



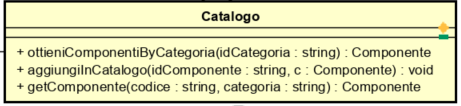
* terminaAssemblaggio



# Introduzione della classe catalogo

La classe catalogo nasce dalla necessità di dover mostrare al cliente e all’amministratore un pool di componenti disponibili per assemblare le configurazioni.

Dunque la classe catalogo avrà la responsabilità di reperire mappe di componenti in base alle richieste prevenute mediante l’handler di caso d’uso.

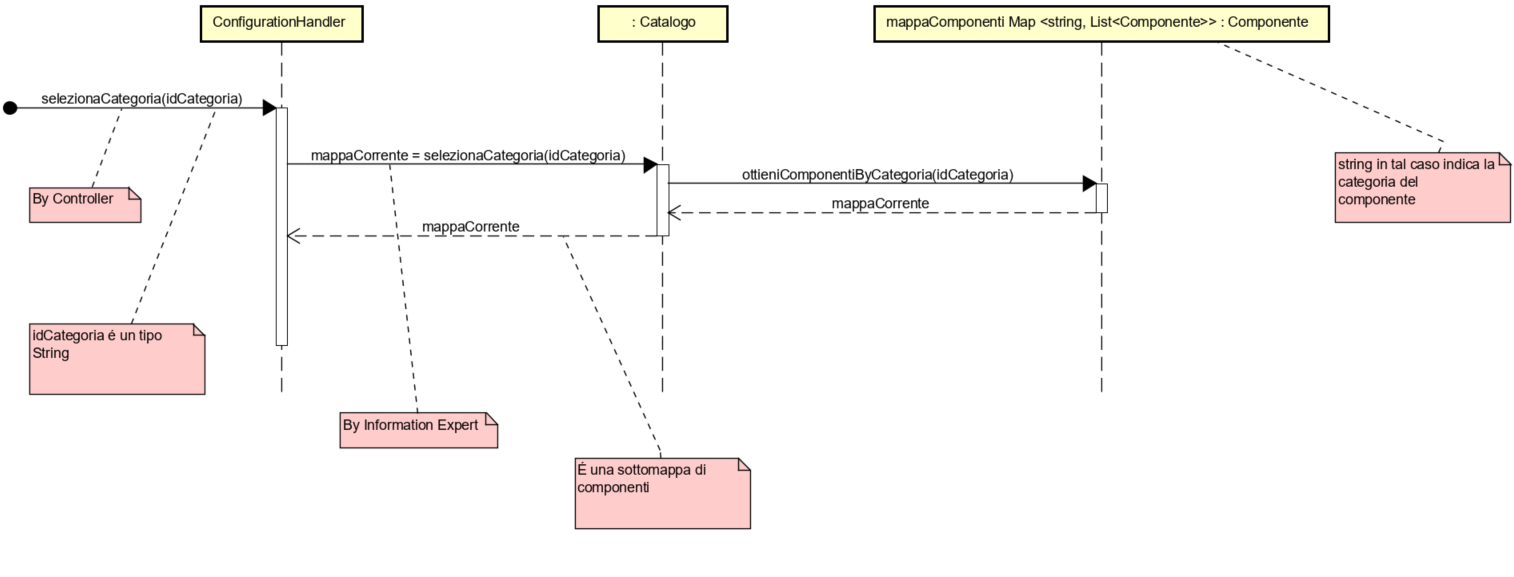


Ecco una rapida spiegazione delle funzioni di controllo presenti in tale classe:

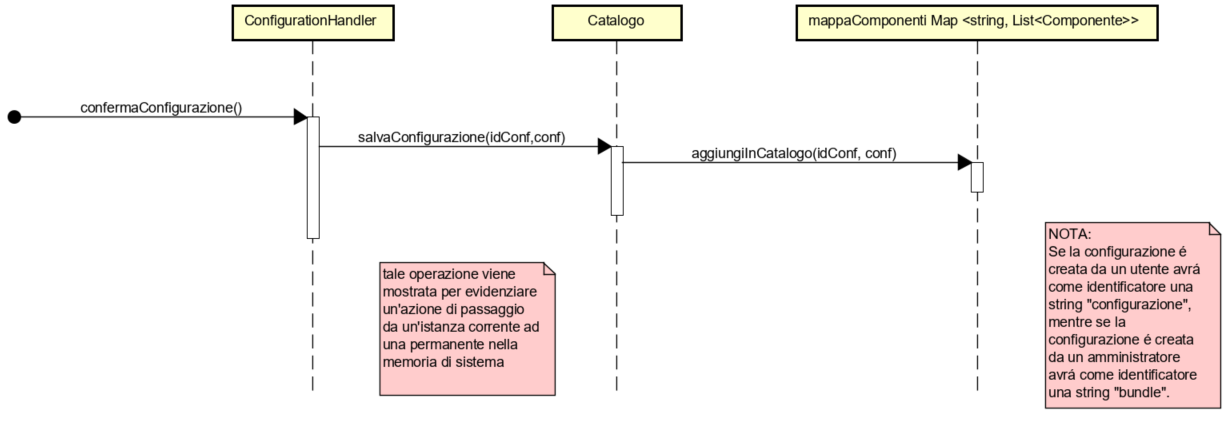
* + **ottieniComponentiByCategoria(idCategoria : string):** tale funzione ha il compito di restituire una mappa “Map <string, List<Componente>>” in base alla categoria fornita dal cliente o dall’amministratore.
  + **aggiungiInCatalogo(idComponente : string, c : Componente):** tale funzione ha il compito di aggiungere nella mappa di componenti del catalogo “Map <string, Componente>” un componente appena creato da un amministratore.
  + **getComponente(codice : string, categoria : string):** tale funzione ha il compito di restituire un componente specifico in base alla suo codice identificativo e alla sua categoria.
    - Si osserva che tale funzione se utilizzata con categoria di tipo “bundle” o “configurazione” può ritornare una configurazione o un bundle, grazie all’applicazione del pattern Composite visto in precedenza.

A seguire i diagrammi di sequenza nel caso di successo, con le funzioni sopra citate:

* selezionaCategoria



* confermaConfigurazione



* creaComponente

