Università degli Studi di Bologna

FACOLTA' DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica
Ingegneria del Software L-A

PROGETTO E REALIZZAZIONE DI UN COMPONENTE SOFTWARE PROGRAMMABILE PER LA PIANIFICAZIONE DI COMMISSIONI DI LAUREA

Tesi di Laurea di :
Roberto Rossi

Relatore:
Chiar.mo Dott. Ing. Giuseppe Bellavia

Anno Accademico 2001 – 2002

Parole chiave:

Commissioni
Laurea
Pianificazione
Componente
Software

I - Analisi preliminare del progetto

Progetto: inserito nell'ambito di un sistema software per la generazione delle commissioni di laurea, realizza un componente che incapsula la logica algoritmica per la funzione di autocreazione delle commissioni.

Scopo del sistema: il componente implementa un flusso algoritmico che, a partire da dati passati all'interfaccia secondo un contratto predefinito, genera un elenco di commissioni rispettando le regole indicate nei parametri passati. Tale componente è dunque strutturato in modo da soddisfare le possibili diverse esigenze dell'utilizzatore, che di volta in volta potrà cambiare la struttura delle commissioni che vorrà ottenere.

Caratteristiche del sistema

Strutturali:

Il componente è realizzato *per contratto,* ovvero implementa un'interfaccia predefinita all'interno del sistema in cui si inserisce e sfrutta tale interfaccia, insieme alle sue regole di comunicazione, per scambiare dati da/verso il sistema principale.

I dati trattati sono dunque vincolati dalle specifiche di interfaccia e in ogni caso descrivono il dominio definito dal dipartimento DEIS con le sue regole, le sue qualifiche, e le sue disposizioni in materia di commissioni di laurea e di laureandi.

Comportamentali:

Le due funzioni principali del componente sono la *valutazione* e la *creazione* delle possibili commissioni a partire dai dati in ingresso. Ovviamente si richiedono tempi di calcolo accettabili e soluzioni sufficientemente vicine agli ottimi del problema, anche se *non è richiesto* l'ottimo globale, visto che esso cambia a seconda dell'importanza attribuita alle singole specifiche e non è dunque possibile definirne a priori uno univoco.

Interazione con altri sistemi/componenti:

L'interazione con il sistema invocante è fondamentale per il componente, esso non ha *vita propria* ma dipende strettamente dalle richieste che gli vengono inoltrate. Tutta la gestione della persistenza dei dati, della raccolta e della visualizzazione non è contemplata in questo progetto; il componente realizza esclusivamente una parte dell'*application logic* del sistema, curandosi ovviamente di gestire la visualizzazione a video e su file dei logging, se avviato in modalità debug.

Analisi dei Requisiti

La richiesta principale formulata dall'utente in fase di discussione preliminare è stata quella di poter sfruttare l'algoritmo ad ampio raggio, per creare commissioni indipendentemente dal fatto che variazioni di politiche interne al dipartimento potessero modificare i numeri e le tipologie docenti che prendono parte nella creazione delle commissioni.

È stato dunque richiesto di poter rendere variabili questi parametri:

- Numero di docenti in commissione.
- Tipi di docenti in commissione
- Corsi di Laurea associati alle commissioni da creare
- Tipi di tesi associati alle commissioni da creare (e dunque tipologie di laureandi)

Inoltre la creazione non deve rispettare numeri precisi riguardo alle tipologie docenti, ma deve prendere le opportune decisioni per massimizzare il numero di commissioni ottenibili: si è parlato di vincoli di minimo e non di uguaglianza; per esempio potrebbe essere assolutamente necessaria la presenza di *almeno* un *professore ordinario* in commissione, ma se questo permettesse per qualche motivo di aumentare il numero di commissioni creato ne potrebbero essere inseriti anche due.

Altra richiesta formulata dall'utente è stata quella di poter ottenere commissioni il più bilanciate possibile in termini di orario di proclamazione, compatibilmente col fatto che un relatore debba comparire in una commissione assieme ai suoi laureandi se questo è possibile (cioè se il relatore è disponibile e i laureandi non sono troppi). L'utente ha inoltre espresso la volontà di poter gestire gli orari di proclamazione imponendo un tetto massimo.

L'algoritmo deve settare automaticamente presidente e segretario delle commissioni create seguendo opportune politiche (vedi dizionario dei termini).

Infine è stato richiesto di gestire in modo opzionale il numero di presenze dei docenti nelle sessioni precedenti, preferendo nell'inserimento docenti con numero di presenze minore.

Il sistema deve operare in ambiente Microsoft Windows 2000 o superiore e il linguaggio concordato per lo sviluppo è C#.

Analisi dei Rischi

I rischi principali che la progettazione di un componente di questo genere presenta riguardano la complessità del problema. Il numero dei vincoli, l'ampio raggio di variazione delle specifiche, specie di quelle sulle commissioni, e la mancanza di una precisa scala di priorità tra le richieste[1] rendono sicuramente complessa la computazione del problema, che può essere classificato come *NP-hard* alla pari di problemi di *packing* bidimensionale. L'analogia tra le due

[1] Ad esempio è importante tenere ogni docente con i propri laureandi in commissione, ma anche non creare commissioni che superino il tetto massimo di durata; tra queste due opzioni non ve n'è una univocamente prioritaria.

classi di problemi è particolarmente evidente se si pensa al fatto che i relatori devono stare con i propri laureandi: abbiamo dei limiti di tempo entro cui far stare blocchi di laureandi, a ciascuno dei quali e' associata una certa durata. Le commissioni stesse sono poi vincolate da un numero massimo di docenti e da altre caratteristiche definite nel descrittore; tipicamente problemi di questo genere hanno complessità esponenziale e vengono risolti con algoritmi approssimati per evitare tempi di calcolo proibitivi, dunque anche nel nostro caso le scelte progettuali andranno in questo senso.

Dizionario:

- **Componente:** libreria software che incapsula le strutture dati e i flussi di computazione.
- **Programma invocante:** rappresenta il sistema software al quale il componente verrà collegato e che effettuerà le chiamate alle funzionalità del componente sviluppato.
- **Utente:** rappresenta colui che utilizza il programma invocante.
- **Descrittore di commissione:** insieme di regole e caratteristiche che definiscono la struttura che una commissione deve rispettare per essere ben formata.
- **Commissione di laurea:** commissione, creata rispettando un determinato descrittore, incaricata della valutazione delle tesi di laurea proposte dai laureandi ad essa associati.
- **Sessione di laurea:** data in cui le commissioni valuteranno le tesi presentate, in pratica può essere vista come un insieme di commissioni.
- **Docenti:** rappresentano il corpo docenti del D.E.I.S. Vengono classificati in base al loro ruolo, scelto tra quelli previsti nelle specifiche di dipartimento e di ateneo.
- **Numero di matricola docente:** numero di cinque cifre che identifica univocamente ogni docente.
- **Laureando:** studente iscritto all'Università di Bologna che sta svolgendo una tesi presso il dipartimento D.E.I.S ed è in procinto di presentarla.
- **Numero di matricola laureando:** è composto da un campo di quattro cifre che identifica il Corso di Laurea e da un campo da dieci cifre che identifica univocamente ogni studente.
- **Presidente:** professore ordinario con maggiore anzianità all'interno della commissione.
- **Segretario:** solitamente è il ricercatore non confermato o il ricercatore con meno anni di servizio all'interno della commissione.
- **Relatore:** docente che ha seguito il laureando durante la stesura della tesi.
- **Tipo Tesi:** descrive le proprietà della tesi: durata dell'esposizione e nome della tipologia. Tale descrittore è importante in quanto tesi di un certo tipo devono essere associate a commissioni opportune (es. tesi del

nuovo ordinamento richiedono commissioni con cinque docenti, mentre quelle del vecchio richiedono commissioni con sette docenti).

- **Disponibilità di un docente:** viene richiesta prima di ogni sessione di laurea, indica se il docente può partecipare o meno; a volte, per politiche interne, il docente può essere inserito d'ufficio.
- **Problemi di packing:** vedi bibliografia[2].
- **Problemi *NP - hard***: vedi bibliografia[3].
- **Algoritmi approssimati:** vedi bibliografia[2].

II - Analisi statica del sistema

Il progetto si propone di modellare un sottoinsieme dei dati modellati dal sistema software a cui il componente verrà collegato. Non è compito del componente gestire la persistenza dei dati su disco, recuperare dati salvati e visualizzare a video risultati; dunque il dominio concettuale dei dati utilizzati dal componente sarà sensibilmente diverso rispetto a quello del sistema software invocante.

Utilizzando la notazione UML possiamo evidenziare la divisione, all'interno del componente, tra classi che regolano il flusso computazionale e classi che contengono dati e su cui tale flusso agisce:

[2] A.Lodi,S.Martello,D.Vigo – Recent Advances on Two-Dimensional Bin Packing Problems.
[3] S.Martello – Lezioni di Ricerca Operativa.

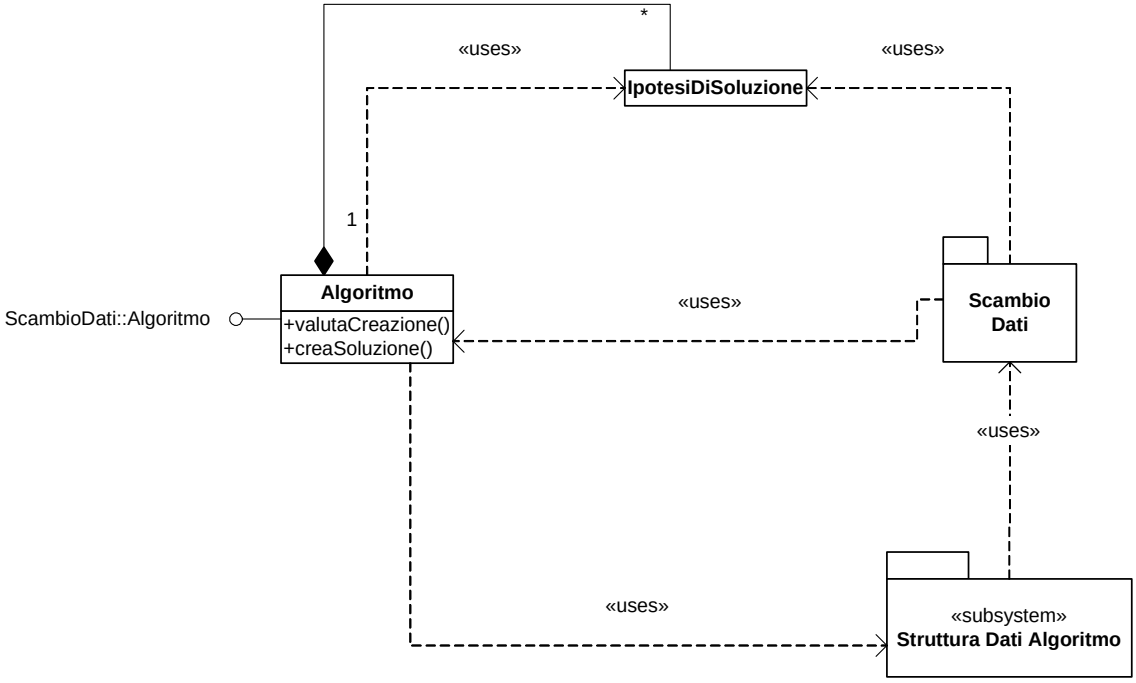


figura 2.0

Se ci poniamo al livello più alto possibile di astrazione, osserviamo che la classe Algoritmo modella tutto quanto concerne le operazioni invocabili sul componente e costituisce l'interfaccia a cui il sistema software si potrà collegare. Tale classe è l'unica visibile all'esterno (public), essa sfrutta il package esterno *scambio dati* e le funzioni che esso mette a disposizione; tale package costituisce l'interfaccia di comunicazione tra il sistema software invocante e il componente. La classe Algoritmo agisce sul sottosistema *struttura dati algoritmo,* invisibile all'esterno, che modella la realtà di interesse per l'algoritmo.

La compatibilità fra il formato *scambio dati* e *struttura dati algoritmo* è totale, in tal modo è possibile passare da una rappresentazione all'altra utilizzando le opportune conversioni messe a disposizione dal componente. In pratica, una volta elaborate le commissioni in formato *struttura dati algoritmo*, basterà invocare l'opportuno metodo e sarà possibile ottenere le stesse commissioni rappresentate in formato *scambio dati*.

È da notare che la classe Algoritmo implementa la relativa interfaccia AlgoritmoSD di *scambio dati;* questo deve essere fatto per rispettare il contratto di interfaccia tra componente e sistema.

Il package *scambio dati* è condiviso tra sistema e componente, in tal modo si assicura la comunicazione tra i due mondi.

Riassumendo dunque, il contratto di comunicazione tra componente e sistema è costituito da un package: *scambio dati,* le cui caratteristiche verranno trattate nel seguito di questo capitolo e anche nella sezione riguardante l'implementazione, e da regole di connessione che verranno evidenziate nell'analisi dinamica del sistema, dove saranno elencati i passi necessari per connettersi al componente.

Il sottosistema ***struttura dati algoritmo***

Tale sottosistema costituisce la struttura dati su cui l'algoritmo agisce, esso modella il dominio del problema e contiene le informazioni che vengono elaborate per creare le commissioni. Le responsabilità delle classi individuate nel sottosistema dati, che verranno elencate in modo preciso nell'apposita sezione di questo capitolo e anche nel capitolo sull'implementazione, si possono riassumere in pochi punti generali:

- Modellare ogni dato oggetto di interesse per l'elaborazione, trascurando tutti i particolari che non sono direttamente utilizzati dall'algoritmo e che concorrerebbero a rallentare la computazione occupando memoria. Questo è infatti il motivo principale per cui si è scelto di creare una struttura dati ad hoc e di non sfruttare quella preesistente di *scambio dati.*
- Controllare che i dati rimangano sempre consistenti durante l'elaborazione attraverso opportuni controlli.
- Offrire funzionalità specifiche per ogni dato, in modo da snellire il flusso algoritmico, che delegherà a tali funzionalità alcune operazioni ove necessario.

A livello concettuale il diagramma UML che modella tale dominio di dati è il seguente:

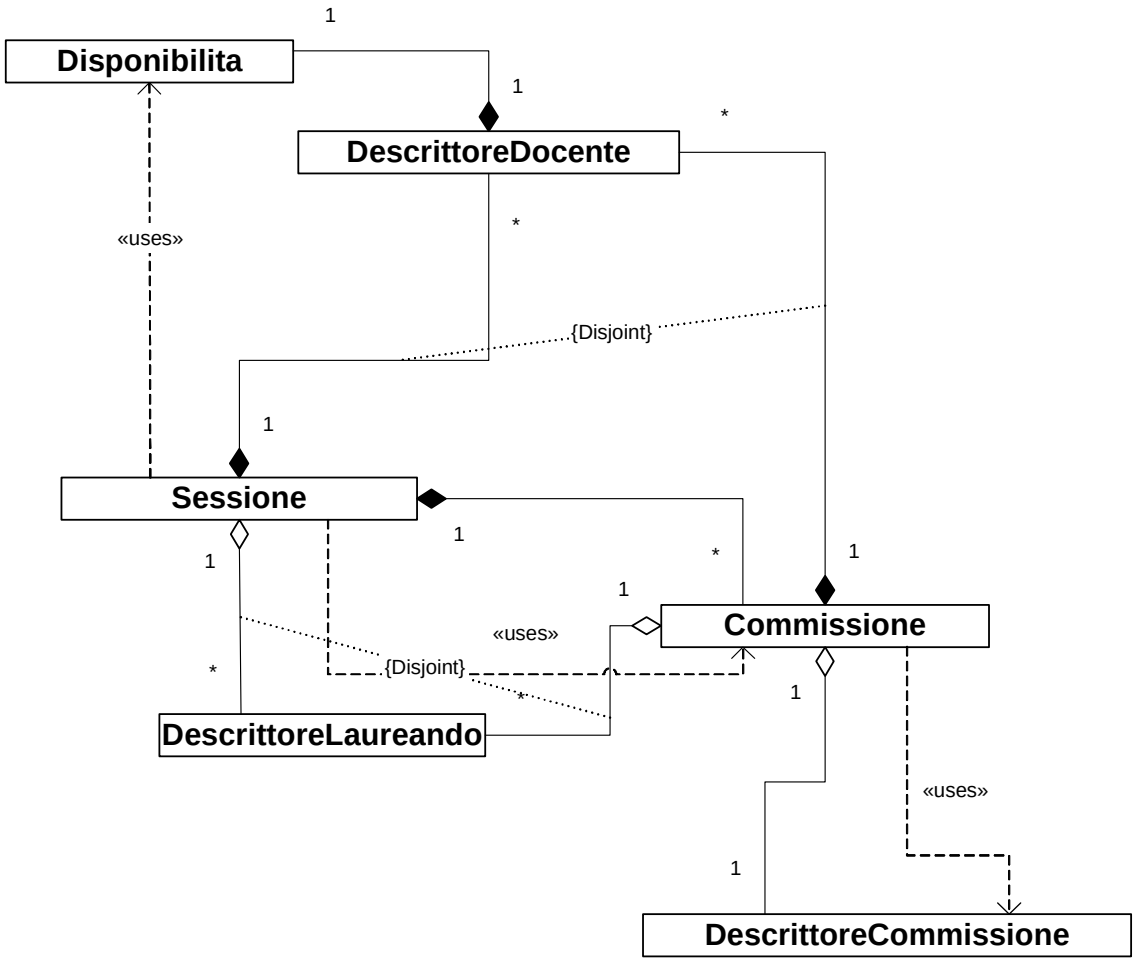


figura 2.1

Notiamo subito che le classi principali individuate nel modello concettuale sono poche; ovviamente ogni classe presenterà caratteristiche più o meno complesse che poi verranno evidenziate in sede implementativa attraverso altre classi associate o attributi di istanza. Ai fini di questa spiegazione tuttavia il modello presentato è sufficientemente esaustivo per rappresentare la realtà di interesse su cui l'algoritmo opera.

La classe principale attorno a cui tutto "ruota" è *Sessione.* Un'istanza di Sessione modella un vero e proprio *contenitore* per docenti, laureandi e commissioni, che, come è possibile notare, costituiscono le altre uniche classi che concorrono nel modellare tutto il dominio di interesse dell'algoritmo. Come vedremo in fase implementativa, la classe Sessione è creata seguendo il pattern *façade*; essendo essa infatti un contenitore, sarà necessario gestire il suo stato in ogni istante, in modo da evitare configurazioni errate dei dati contenuti. Sarà dunque l'interfaccia di sessione a offrire tutte le funzionalità di gestione dello stato interno (es. inserisci un laureando, crea una commissione, muovi un laureando in commissione e così via) ed offrirà inoltre le opportune funzioni di controllo (check) che verificheranno lo stato a fronte di ogni operazione, impedendo quelle illecite (es. inserisci un laureando in sessione quando questo è già presente). Particolarmente utile sarà la possibilità di clonare una sessione con un certo stato. Ciò è possibile grazie al fatto che sessione, come poi tutte le altre classi del sottosistema *struttura dati algoritmo,* implementa il pattern *prototype* (opportunamente modificato come vedremo in fase implementativa) che permette di replicare oggetti complessi attraverso un efficiente gestione delle deleghe.

La classe DescrittoreDocente modella invece un docente del dipartimento con le sue caratteristiche statiche: età, data entrata in ruolo, tipo etc., e dinamiche, che cioè possono variare nel tempo: disponibilità. Non è stato necessario modellare la disponibilità di un docente come classe associazione in quanto l'algoritmo lavora su uno *snapshot* dello stato del sistema principale. Non è dunque possibile che a fronte di un invocazione un docente abbia più di una disponibilità a suo carico. Ovviamente in due invocazioni successive l'una all'altra la disponibilità potrà essere diversa.

La classe DescrittoreLaureando raccoglie tutti i dati concernenti un laureando, dunque il tipo della tesi, la matricola etc.

Da notare sono i vincoli *disjoint* tra le associazioni DescrittoreLaureando - Sessione e DescrittoreLaureando – Commissione, così come tra DescrittoreDocente – Sessione e DescrittoreDocente – Commissione. Questo descrive, in pratica, uno dei tanti controlli eseguiti dai check di Sessione, non tutti esprimibili ovviamente in UML concettuale. Il vincolo fa riferimento al fatto che un laureando possa trovarsi in Sessione, non assegnato ad alcuna Commissione, o in Commissione; in quest'ultimo caso, essendo già stato assegnato, esso non deve comparire tra i non assegnati in Sessione. Tale esempio è stato riportato per mostrare il significato e l'utilità dei vari check analoghi, che ritroveremo in fase implementativa in Sessione e in Commissione.

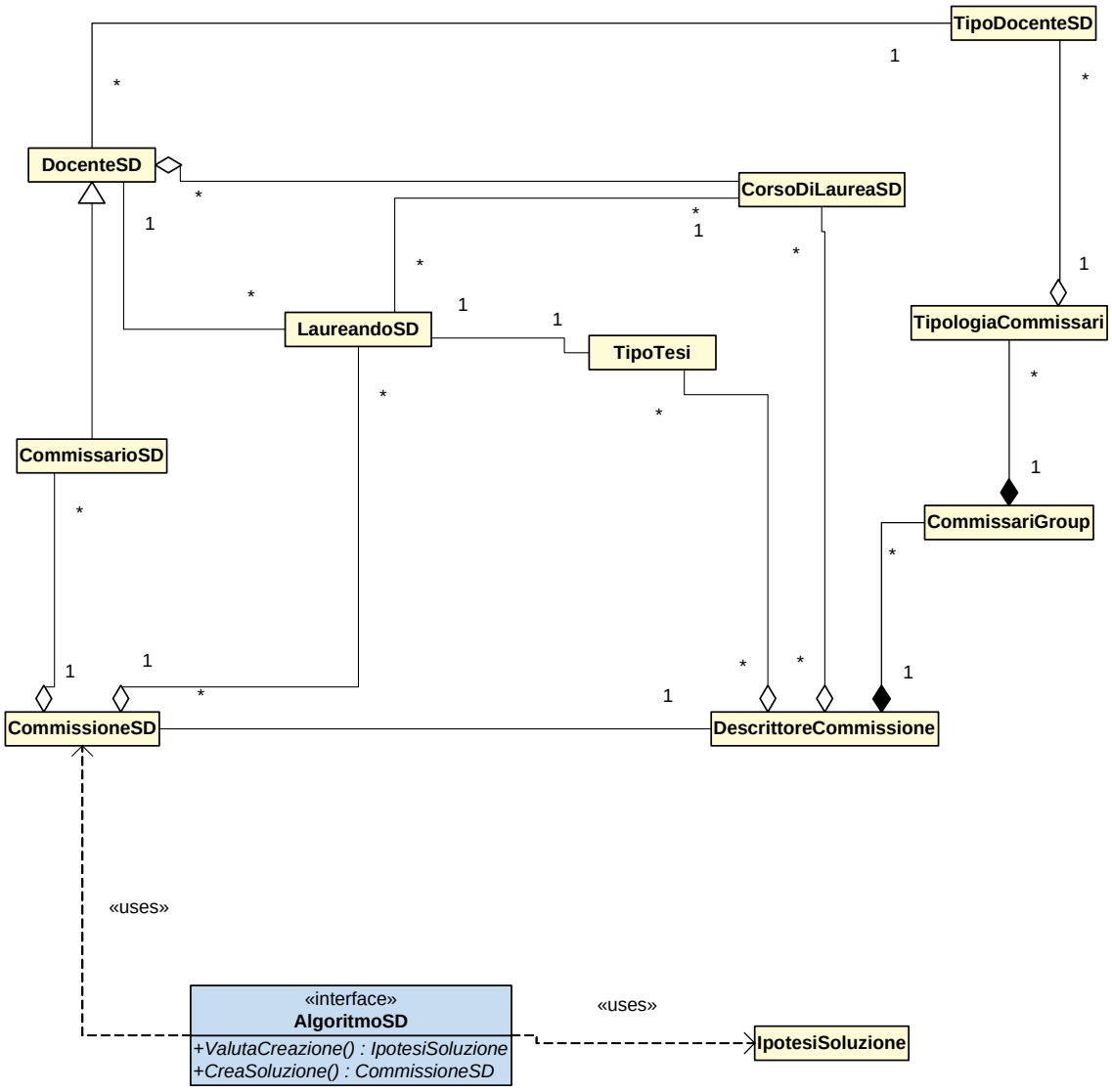
La funzione di DescrittoreCommissione, infine, è quella di raccogliere tutte le informazioni relative alla struttura che una commissione dovrà rispettare per

essere ben formata (tipologie e numero docenti, tipologie di tesi etc.). Tale classe ha importanza centrale, in quanto è largamente utilizzata nelle funzioni di check come fonte dei dati su cui basare i controlli sia in Commissione, sia in Sessione.

*Il package **scambio dati***

Come evidenziato in precedenza, tale package costituisce il contratto di interfaccia per comunicare con il sistema software principale. Utilizzando le classi di tale package è possibile scambiare dati da/verso il sistema principale. È dunque possibile linkare un qualsiasi algoritmo che implementi le funzionalità dell'interfaccia AlgoritmoSD e sfrutti le classi presenti in tale package per ricevere docenti, laureandi e descrittore di commissione e restituire le commissioni elaborate. Il sistema di comunicazione tra componente e sistema software realizza quindi l'idea del pattern *View,* in quanto sfruttando il package ScambioDati è possibile creare in ogni momento una view dello stato del componente o del sistema, permettendo ai due di *vedere* l'uno lo stato dell'altro.

Di seguito è possibile osservare il diagramma UML del package:



*

figura 2.2

Verranno commentate solo le classi più importanti ai fini della spiegazione del contratto stabilito dall'interfaccia per la connessione algoritmo-sistema.

<<*Interface*>> *AlgoritmoSD:* dichiara i due metodi che devono essere implementati per soddisfare il contratto di *linking*. Tali metodi saranno opportunamente trattati nell'analisi dinamica del componente e nella sezione dedicata all'implementazione. Per ora basti sapere che il primo, cioè *valutaCreazione,* ha il compito di avviare un'elaborazione preliminare, in modo da offrire all'utente un ventaglio di scelte tra cui selezionare la soluzione

* Nell'immagine in blu compaiono le interfacce, mentre in giallino le classi.

migliore per le sue esigenze; tale soluzione viene valutata in base alla durata e al numero di commissioni creabili. Le ipotesi così valutate sono poi reperibili attraverso un opportuna istanza della classe IpotesiSoluzione.

Il secondo metodo, cioè *creaSoluzione,* restituisce le commissioni create a partire da una determinata istanza di IpotesiSoluzione.

Come è possibile notare, le altre classi del package risultano analoghe a quelle modellate nel sottosistema *struttura dati algoritmo,* infatti, anche se le classi presenti in quest'ultimo offrono funzionalità maggiori e risultano maggiormente ottimizzate, i dati modellati sono in sostanza gli stessi, visto che sono presenti funzioni di conversione bidirezionali.

Schede di responsabilità delle classi:

sottosistema *struttura dati algoritmo*

Classe	*Responsabilità*	*Collaboratori*
Sessione	Descrivere una sessione di laurea e tutto quanto è ad essa legato: commissioni, docenti, laureandi. È in pratica un contenitore.	Commissione Disponibilita DescritttoreDocente DescrittoreLaureando
Commissione	Descrivere una commissione di laurea all'interno di una sessione, assieme a docenti e laureandi presenti in essa, nel rispetto dei vincoli imposti dal rispettivo DescrittoreCommissione.	Sessione DescrittoreCommissione DescrittoreDocente DescrittoreLaureando Disponibilita
DescrittoreDocente	Descrivere un docente e le sue caratteristiche statiche (cioè che non variano nel tempo o che variano raramente)	Sessione Commissione Disponibilita
DescrittoreLaureando	Descrivere un laureando e le sue caratteristiche statiche.	Sessione Commissione
Disponibilita	Descrivere la disponibilità di un docente	DescrittoreDocente Sessione Commissione
DescrittoreCommissione	Descrive le caratteristiche strutturali di una Commissione: numero docenti, tipologie presenti, tipi di tesi etc.	Commissione

package *scambio dati*

AlgoritmoSD	Dichiarare l'interfaccia standard che ogni algoritmo dovrà implementare per risultare compatibile con il sistema software.	---
Altre classi	Fornire una struttura per lo scambio dati tra il sistema software e il componente che implementa l'algoritmo.	vedi diagramma UML

classi principali del componente *algoritmo*

Algoritmo	Implementa l'interfaccia di comunicazione come da contratto.	Classi del sottosistema *struttura dati algoritmo* Classi del package *scambio dati* IpotesiSoluzione
IpotesiSoluzione	Proxy per l'omonima classe nel package *scambio dati*. Fornisce funzioni di conversione in entrambi i sensi e altre funzionalità utili ai metodi di Algoritmo. Modella un'ipotesi di soluzione per la creazione di commissioni, indicando il tempo e il numero di commissioni da creare.	Algoritmo IpotesiSoluzioneSD

III - Analisi dinamica del sistema

L'algoritmo si inserisce nell'ambito dell' *application logic* del sistema software DeisLauree; in realtà, a causa della complessità delle operazioni di cui è incaricato, esso è incapsulato in un componente indipendente dagli altri che costituiscono il sistema ed è accessibile attraverso un'interfaccia dichiarata nelle specifiche di interazione tra i componenti. Implementando tale interfaccia l'algoritmo diviene disponibile run time al sistema che lo può invocare passando

gli opportuni parametri richiesti nell'interfaccia. In questa sede verrà modellato esclusivamente il componente Algoritmo, mentre il sistema software principale costituirà una *black-box* di cui si considera nota esclusivamente l'interfaccia.

Scenario Principale:

L'utente richiede al software di creare nuove commissioni in base ai dati inseriti e selezionati per l'operazione, il sistema elabora i dati e restituisce tutte le possibili soluzioni che riesce ad elaborare. L'utente sceglie una soluzione tra quelle proposte e il sistema restituisce le commissioni relative a tale soluzione.

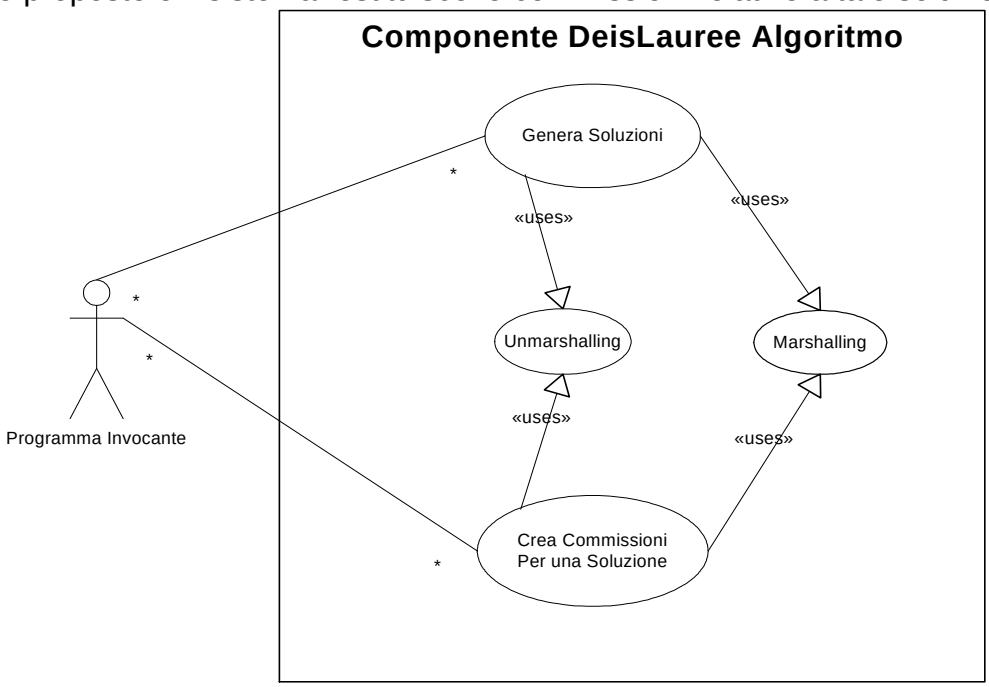


figura 3.0

Osservando il diagramma UML dei casi d'uso è possibile osservare che lo scenario principale precedentemente descritto è costituito in realtà da due scenari:

1. Genera soluzioni
2. Crea commissioni per una soluzione

Queste sono in sostanza le due principali funzionalità che l'algoritmo mette a disposizione del programma invocante.

Genera soluzioni

a. Il programma invocante richiede la valutazione delle ipotesi di soluzione per un determinato elenco di docenti e laureandi e per un dato descrittore di commissione.
b. L'algoritmo accetta i parametri in formato ScambioDati ed esegue la conversione a formato locale.
c. Vengono avviate le procedure di valutazione delle soluzioni e vengono creati due elenchi: uno con le ipotesi di soluzione, contenenti le

informazioni riassuntive riguardanti ogni soluzione (durata…), e uno con le soluzioni vere e proprie.

d. L'algoritmo converte i due elenchi in formato ScambioDati

alternativa 1 - fallimento

b. L'algoritmo rileva inconsistenze nella struttura dati, e la conversione fallisce.

alternativa 2

c. L'algoritmo analizzando i dati si accorge di non poter creare alcuna soluzione con i dati ricevuti e ritorna un elenco vuoto di soluzioni.

alternativa 3 - fallimento

c. L'algoritmo rileva inconsistenze nella struttura del descrittore e fallisce.

alternativa 4 – fallimento

b. Almeno uno dei parametri è nullo, la valutazione fallisce.

Crea commissioni per una soluzione

a. Il programma invocante, passando un'ipotesi di soluzione, richiede la restituzione delle commissioni relative a tale soluzione.
b. Il componente restituisce l'opportuno elenco di commissioni.

alternativa 1 – fallimento

b. Il componente non trova un elenco di commissioni che coincide con la soluzione richiesta e la restituzione fallisce.

In entrambi i casi l'attore è il componente invocante.

L'obiettivo di *Genera soluzioni* è quello di esplorare le soluzioni possibili e di creare gli elenchi che le contengono, eventualmente restituendo errori con informazioni relative al motivo del fallimento dell'elaborazione.

L'obiettivo di *Crea commissioni per una soluzione* è semplicemente quello di restituire l'opportuno elenco di commissioni associato all'ipotesi di soluzione passata come parametro; nel caso in cui non sia stato possibile nel caso d'uso precedente creare alcuna soluzione, per motivi legati a carenze di disponibilità o comunque non legati ad errori, verrà restituito un elenco vuoto.

La frequenza delle operazioni è legata alle richieste che provengono da chi invoca i metodi; tuttavia bisogna notare che, spesso, accadrà che dopo aver compiuto una richiesta di *generazione delle soluzioni* l'utente sarà tipicamente interessato a visitare le soluzioni proposte valutando i docenti inseriti e il numero di laureandi per commissione; quindi probabilmente spesso ad una chiamata di *generazione delle soluzioni* potranno seguire più chiamate di *crea commissioni.*

Di seguito sono stati inseriti i diagrammi UML di sequenza legati ai due scenari principali, ciascuno con i rispettivi scenari alternativi di fallimento.

Genera soluzioni

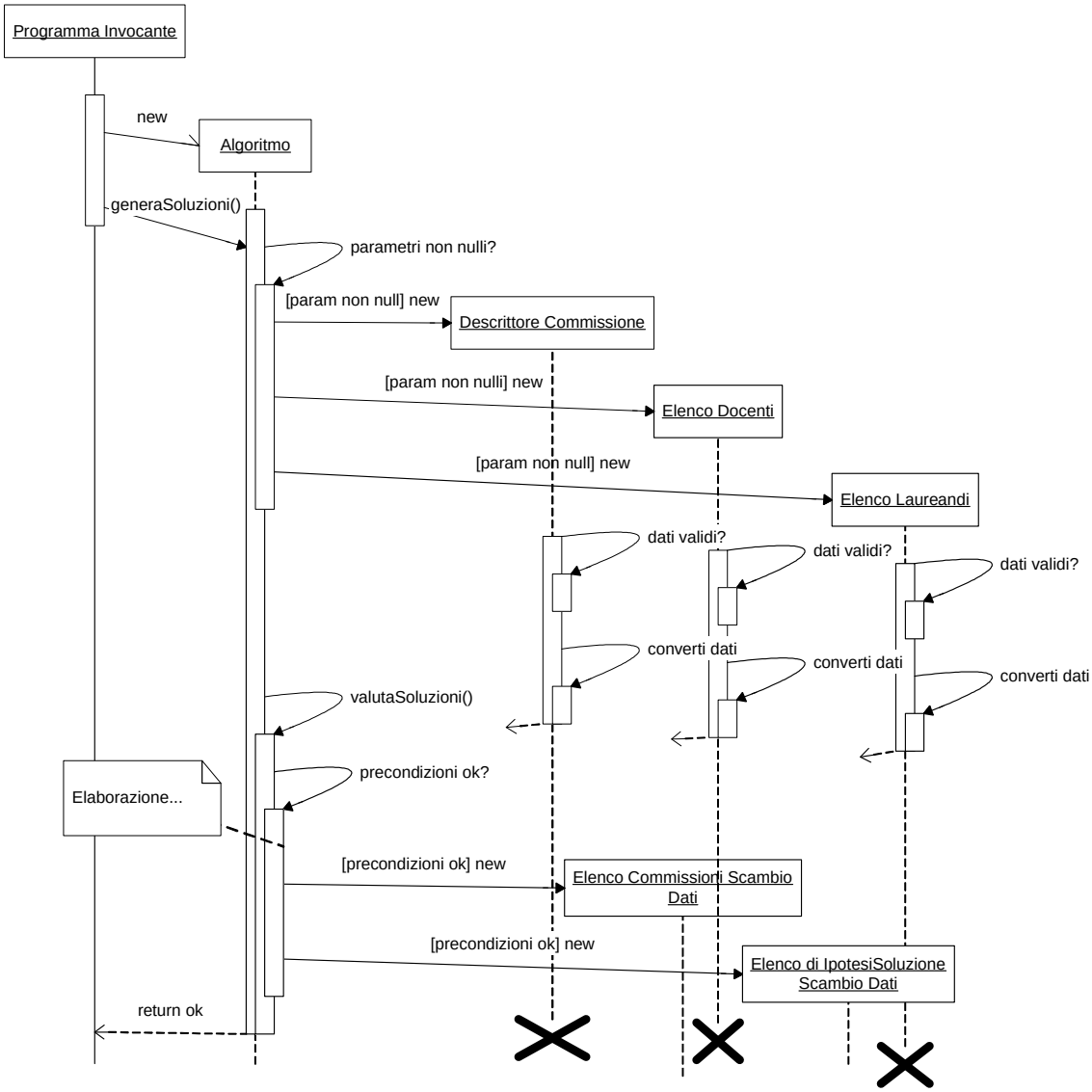


figura 3.1

generaSoluzioni() è una chiamata che deve passare tre parametri all'algoritmo in formato *scambio dati*:

- Elenco di descrittori di commissione, ciascuno dei quali contiene le informazioni per una determinata tipologia di commissioni che deve essere creata.
- Elenco di docenti, con le rispettive disponibilità, con i quali devono essere create le commissioni.
- Elenco di laureandi da inserire nelle commissioni.

Innanzi tutto viene verificato che i parametri non siano nulli, poi vengono istanziati tre nuovi elenchi; questo perché quelli passati come parametri sono in formato scambio dati, mentre quelli che l'algoritmo elabora sono dei *proxy* in formato dati interno. Durante la creazione viene verificata la validità dei dati

passati in formato scambio dati e, in caso di inconsistenze, vengono lanciate eccezioni. Tutti i *proxy*, e quindi anche l'elenco di commissioni elaborato dall'algoritmo, vengono poi riconvertiti al formato *scambio dati* prima della restituzione. Le conversioni sono necessarie perché il formato interno è appositamente studiato per migliorare le prestazioni durante l'elaborazione.

Le precondizioni sono vincoli imposti ai dati in ingresso e controllati prima di avviare l'effettiva elaborazione delle soluzioni:

- Gli elenchi passati come parametri devono essere non vuoti.
- Devono essere presenti docenti e laureandi per i corsi di laurea e le tesi indicate nel descrittore, se non ce ne sono, si restituisce un messaggio di errore (vedi successivi scenari alternativi di fallimento).
- Il massimo numero di commissioni ottenibili deve essere maggiore di zero considerando i docenti, le loro caratteristiche e la loro disponibilità.
- La durata delle commissioni che si stanno per creare deve essere minore di quella massima specificata come vincolo.

Di seguito vengono presentati i tre possibili scenari alternativi a quello principale.

alternativa 1

Fallimento dovuto a precondizioni violate:

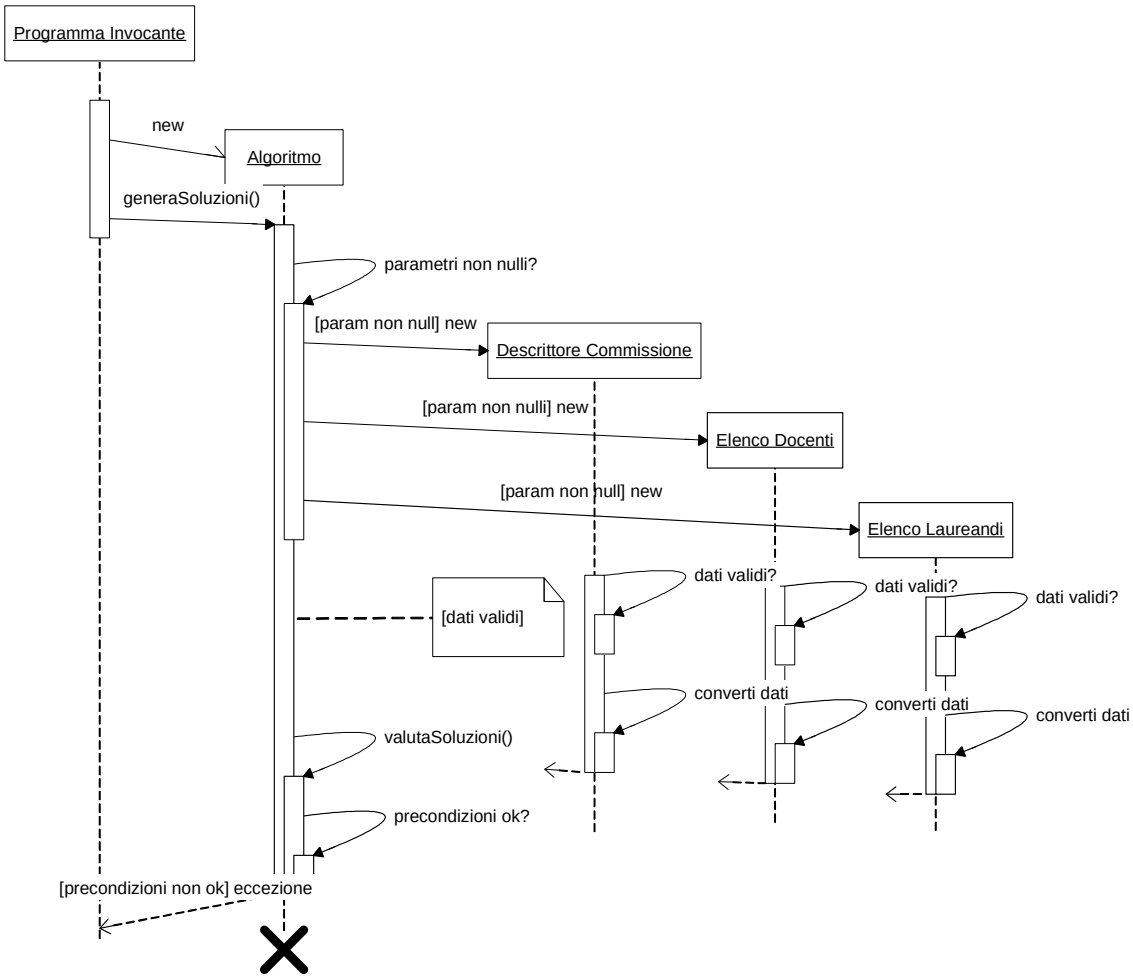


figura 3.2

Da notare che mentre la validità dei singoli dati è controllata prima dell'avvio dell'elaborazione, la validità dei dati nel loro complesso può essere inferita solo durante o all'inizio dell'elaborazione che comunque deve essere già stata avviata.

alternativa 2

Fallimento dovuto a dati non validi:

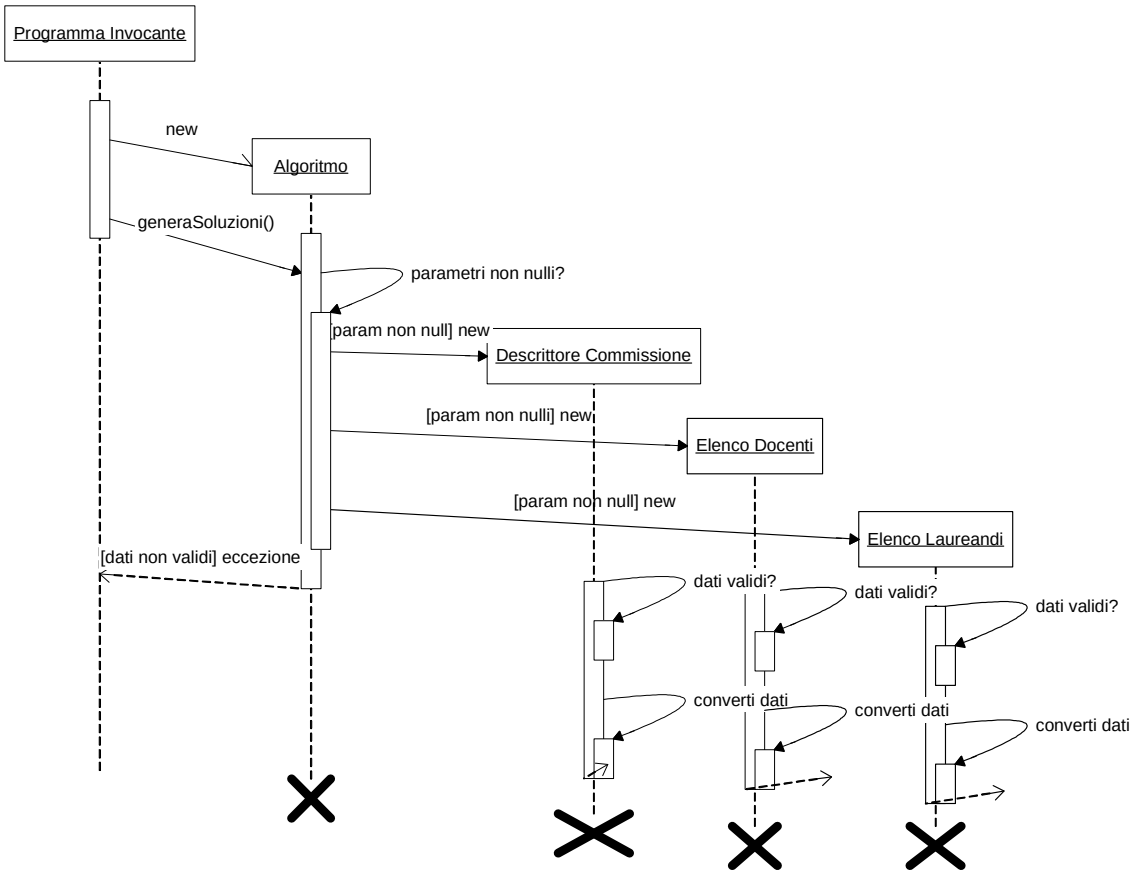


figura 3.3

La validità dei dati è strettamente legata alle specifiche di progetto e verrà discussa successivamente, possiamo però notare fin da ora cosa si intende per validità: consideriamo valido un dato (es. un docente) se tutti i suoi campi obbligatori sono definiti e non nulli, e se i dati complessi, cioè strutturati in sottoparti, in esso contenuti sono anch'essi validi.

alternativa 3

Fallimento dovuto alla presenza di almeno un parametro nullo.

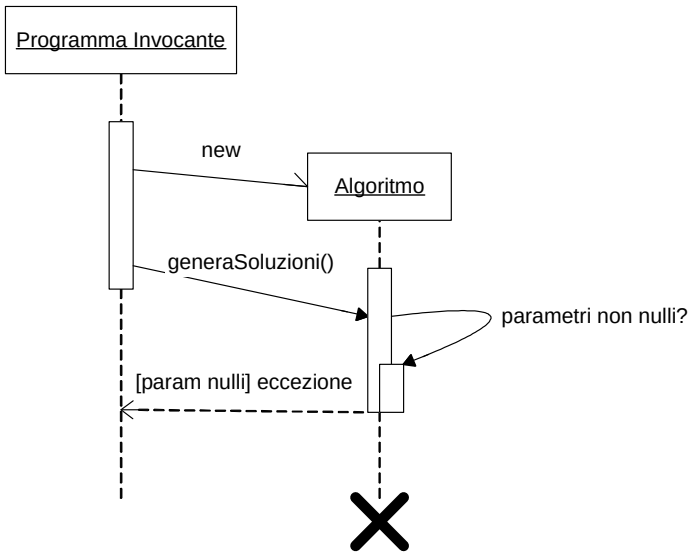


figura 3.4

Sarà compito del programma invocante gestire le eccezioni in modo opportuno e restituire a video o su file di log i messaggi relativi.

Crea Commissioni

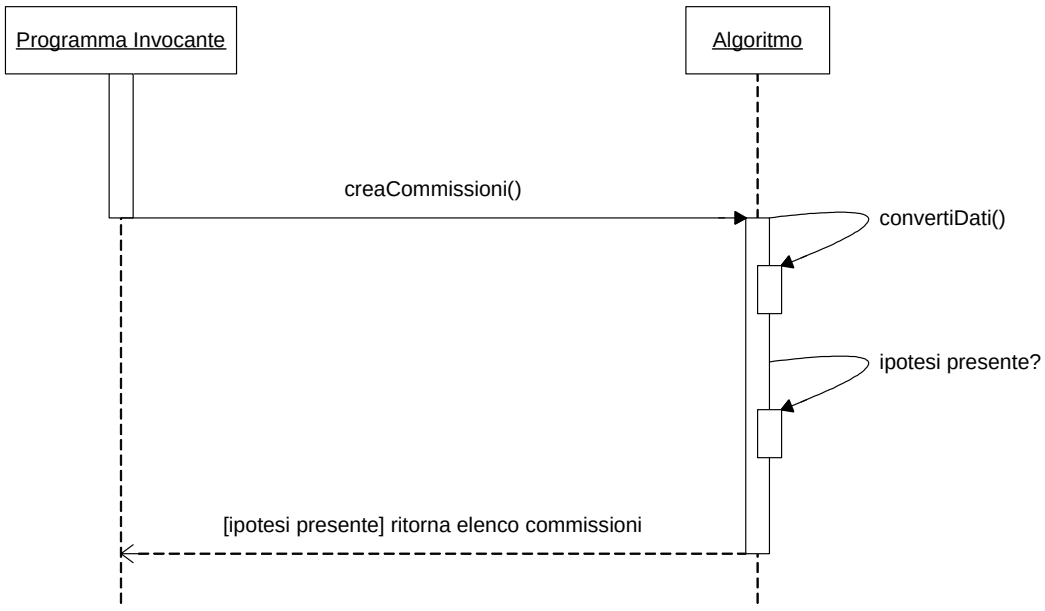


figura 3.5

Tale chiamata non fa altro che prendere l'ipotesi di soluzione passata in formato *scambio dati* e verificare che ve ne sia una equivalente nell'elenco memorizzato nell'algoritmo e creato nella precedente chiamata di *genera ipotesi.* In caso positivo, l'elenco di commissioni associato a tale soluzione verrà restituito.

Lo scenario alternativo in caso di fallimento è invece il seguente:

alternativa 1

Fallimento dovuto al fatto che non è stato possibile trovare nell'elenco di soluzioni elaborate una equivalente a quella richiesta:

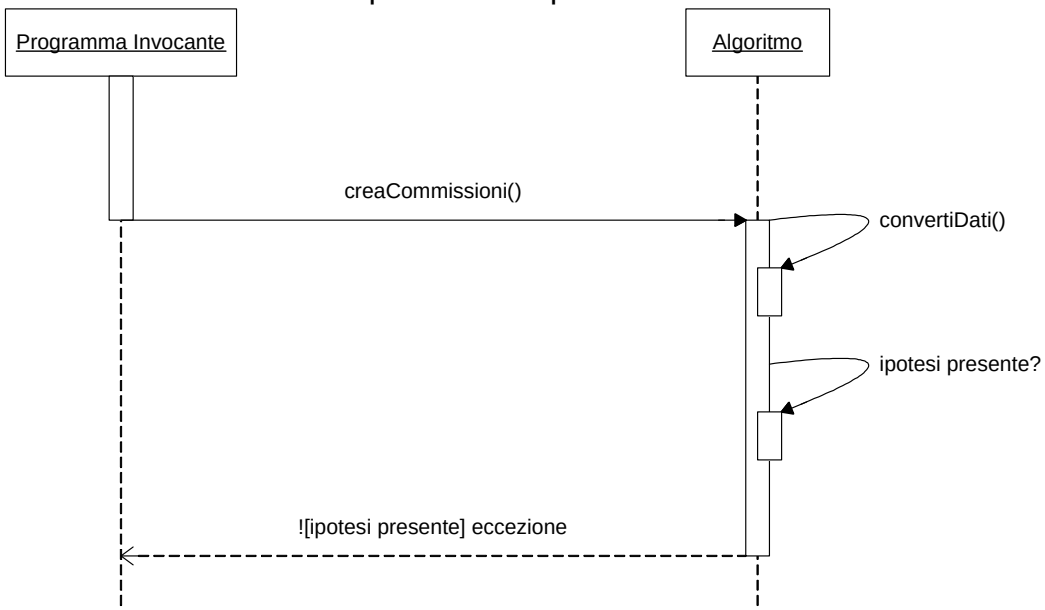


figura 3.6

I messaggi di errore devono essere opportunamente gestiti dal programma principale in quanto sono la sola fonte di dati attraverso cui è possibile risalire all'errore che ha portato ad un fallimento; ogni messaggio contiene una stringa che permette di risalire facilmente alla causa dell'errore stesso.

Il contratto di interfaccia – UML diagrammi di collaborazione

Tale paragrafo è stato inserito alla fine del capitolo in quanto necessita di tutto quanto è stato fin ora spiegato, per descrivere l'interfaccia di comunicazione nella sua interezza.

Nella spiegazione si farà riferimento al seguente diagramma di collaborazione:

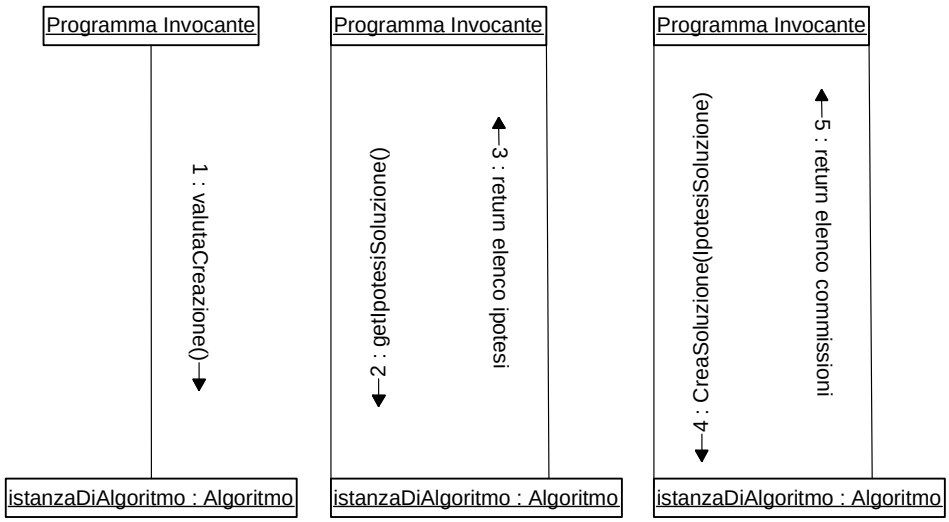


figura 3.7

Si è scelto di usare i diagrammi di collaborazione per enfatizzare l'ordine di invio dei messaggi:

1. il programma invocante richiede la valutazione delle ipotesi di soluzione per i parametri passati, che sono:
    - elenco di istanze di DescrittoreCommissioneSD, che servono per specificare le tipologie di commissioni da creare.
    - elenco di istanze di DocentiSD da usare per creare le commissioni.
    - elenco di istanze di LaureandiSD per i quali creare le commissioni.
2. il programma invocante richiede la restituzione dell'elenco di istanze di IpotesiSoluzioneSD, già creato a seguito dell'invocazione 1.
3. viene restituito un elenco di istanze di IpotesiSoluzioneSD al programma invocante.
4. il programma invocante richiede la creazione delle commissioni relative ad una certa istanza di IpotesiSoluzioneSD precedentemente valutata e restituita.
5. il componente restituisce l'opportuno elenco di istanze di CommissioneSD.

Arrivati al punto 5 è possibile iterare più volte ripartendo dal punto 4 e recuperando elenchi di commissioni a piacere.

Questa descrizione completa il quadro delle azioni necessarie per soddisfare il contratto di interfaccia stabilito per la comunicazione; tale contratto è inoltre costituito dalle regole di implementazione già spiegate nel capitolo sull'analisi statica (vedi implementazione dell'interfaccia AlgoritmoSD e utilizzo del package *ScambioDati*).

IV - Progettazione del componente e scelte di implementazione adottate

I problemi critici evidenziati dalla fase progettuale e di implementazione del componente sono stati soprattutto quelli legati all'ottimizzazione delle strutture dati utilizzate e dei workflows su queste applicati.

Si è cercato di massimizzare il più possibile il parallelismo nella computazione, introducendolo ovunque fosse possibile, in modo da ottenere tempi di risposta accettabili anche a fronte di richieste operate su liste di docenti e laureandi con più di cento elementi. Sono stati introdotti controlli nei punti cruciali della valutazione delle soluzioni, in modo tale da fermare la ricerca di soluzioni alla prima utile, cioè ammissibile e ottima, trovata; vedremo in seguito come è stato possibile garantire che la prima soluzione ammissibile trovata fosse anche quella ottima e cosa precisamente si intende per *soluzione ammissibile.*

Per quel che riguarda le strutture dati, l'attenzione è stata principalmente rivolta all'occupazione di memoria. Non essendo stato possibile utilizzare pattern di tipo *Flyweight* e *Singleton Factory* per minimizzare il numero di istanze in memoria, si è scelto di sfruttare l'idea del pattern *Prototype;* è stato possibile in tal modo evitare di istanziare dei gestori ad ogni nuova chiamata da parte del programma invocante e gestire in modo *decentralizzato* la creazione di nuove istanze, sfruttando una struttura a puntatori e le proprietà del *garbage collector.* Il problema principale del *Flyweight* è infatti quello di dover creare dei gestori e di dover far passare attraverso questi tutte le chiamate per ottenere nuove istanze di determinate classi; questa scelta non sarebbe totalmente compatibile con il parallelismo presente e *pesante*, visto che il numero medio di chiamate per ottenere nuove istanze risulta elevato.

Infine una particolare attenzione verrà rivolta all'analisi del componente che sta alla base di tutto il processo di esplorazione delle soluzioni. Implementato in una classe col nome DocentiTree, tale struttura permette di ottenere a partire da una lista di docenti che possono essere confrontati tra loro (ovvero che implementano l'interfaccia *Comparable*), un'altra lista contenente un numero di disposizioni di docenti pari a quello richiesto: è possibile infatti richiedere un numero di disposizioni minore dell'effettivo *n!* associato ad una lista di *n* docenti, in modo da ottenere *x* disposizioni distribuite tra quelle possibili[4]. La

[4] Vedi paragrafo sull'implementazione del componente per maggiori dettagli.

particolarità sta nel fatto che le disposizioni sono ordinate in modo tale da rispecchiare la confrontabilità tra i docenti: compaiono cioè prima le quelle che presentano nelle prime posizioni i docenti che dal confronto risultano minori[5].

Analisi progettuale delle strutture dati

Come già osservato nell'analisi statica le strutture dati utilizzate dal componente sono divise in tre gruppi principali:

- Strutture dati su cui l'algoritmo opera (*subsystem* Struttura Dati Algoritmo)
- Strutture dati per lo scambio (*package* Scambio Dati)
- Strutture dati contenenti il workflow

L'accesso al componente dall'esterno può essere effettuato tramite la classe Algoritmo che implementa l'interfaccia di comunicazione AlgoritmoSD messa a disposizione nel package Scambio Dati, che peraltro contiene tutte le classi necessarie per la comunicazione componente algoritmo - sistema principale.

Subsystem Struttura Dati Algoritmo

Abbiamo già elencato, in fase di analisi statica, le responsabilità delle classi principali di questo sottosistema su cui l'algoritmo opera. Analizzeremo dunque ora le caratteristiche legate all'implementazione di tali classi e i pattern utilizzati.

Osserviamo innanzitutto le classi del diagramma UML in figura 2.1 e confrontiamole con quelle del diagramma modificato:

[5] Nel caso in questione il confronto è fatto sul numero di presenze in commissione, ma il componente è versatile e presenta numerose applicazioni: può infatti essere sfruttato per ottenere combinazioni ordinate su altri fattori, tutto dipende da come si implementa l'interfaccia *Comparable.*

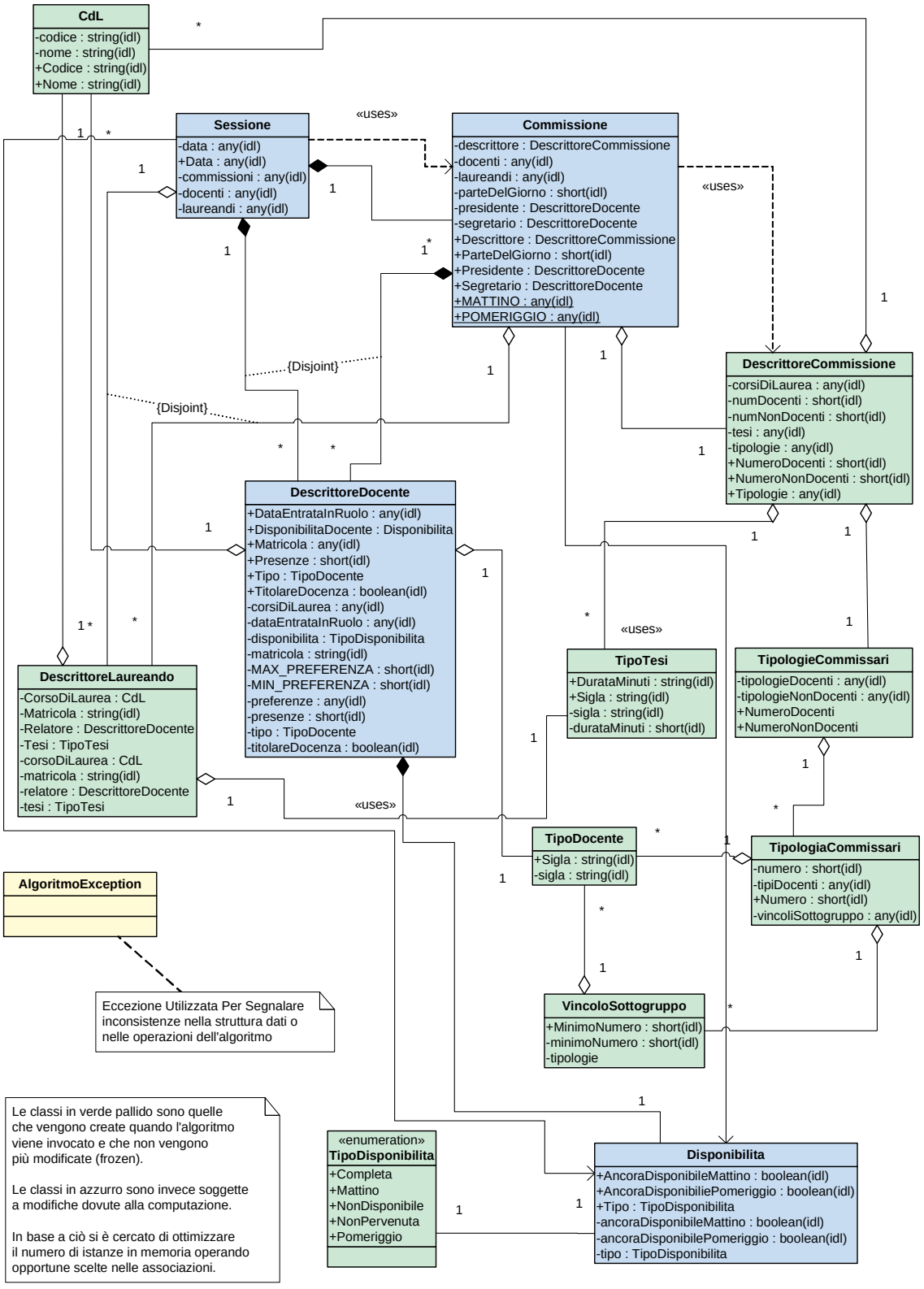


figura 4.0

Nel passaggio alla fase progettuale tutte le relazioni tra le classi principali sono rimaste le stesse, mentre sono state aggiunte nuove classi per modellare tutte le caratteristiche di ogni entità rilevante nel dominio dei dati.

Notiamo subito una differenza importante già messa in risalto dalla nota nella figura: le classi appaiono in due colori differenti, questo per mettere in risalto la soluzione adottata per eliminare eventuali ridondanze in fase di duplicazione. Come già osservato nell'introduzione uno dei problemi cruciali è stato quello di gestire in modo efficiente ed economico il numero di istanze in memoria, evitando replicazioni inutili; la soluzione adottata è stata quella di implementare il pattern *Prototype*[6] discriminando tra istanze *frozen* (notazione comune in UML), in verde pallido nell'immagine, che cioè non sono *mai* soggette a modifiche dopo la creazione, e istanze non *frozen*, cioè soggette a modifiche durante il workflow, in azzurro nell'immagine. Durante la duplicazione di un'istanza le istanze *frozen* ad essa connesse sono gestite tramite riferimenti mentre le istanze normali vengono clonate (si crea cioè una nuova istanza con strutture dati separate). Il modello così creato è a delega e segue il pattern *Chain of Responsability*[7]: se DescrittoreDocente, che è soggetto a modifiche e contiene un riferimento a TipoDocente (che invece è *frozen*), viene duplicato, durante la duplicazione le strutture dati primitive di DescrittoreDocente sono duplicate, mentre l'istanza di TipoDocente non viene duplicata e viene associata tramite riferimento anche alla nuova istanza di DescrittoreDocente, evitando così di avere due istanze della stessa classe *frozen* in memoria. Possiamo immaginare quanto spazio si possa risparmiare nel momento in cui viene clonata un'intera Sessione.

Di seguito vengono inserite le *schede di responsabilità* delle nuove classi introdotte:

Classe	***Responsabilità***	***Collaboratori***
CdL (*frozen)*	Tiene traccia del corso di laurea di un docente o di un laureando e viene utilizzata anche nei descrittori di commissione.	DescrittoreDocente DescrittoreLaureando DescrittoreCommissione
TipoTesi (*frozen*)	Tiene traccia del tipo di tesi presentato da un laureando o che può comparire in una commissione.	DescrittoreLaureando DescrittoreCommissione
TipoDocente	Indica la tipologia di un docente e quali tipi di docenti possono comparire nelle commissioni associate ad un certo descrittore	DescrittoreDocente TipologiaCommissari VincoloSottogruppo

[6] Vedi Appendice A – Design Patterns Utilizzati
[7] Vedi Appendice A – Design Patterns Utilizzati

TipologieCommissari	Contenitore che riunisce tutti i gruppi di tipi docenti con il numero ad essi associato di docenti che devono essere presenti in una commissione.	DescrittoreCommissione TipologiaCommissari
TipologiaCommissari	Rappresenta un gruppo di tipi docenti con un numero ad essi associato: nella commissione dovranno comparire docenti tra i tipi elencati in numero pari a quello indicato.	TipologieCommissari VincoloSottogruppo
VincoloSottogruppo	È utilizzato per introdurre un vincolo di minimo in sottogruppi all'interno dei gruppi definiti da TipologiaCommissari.	TipologiaCommissari

La struttura dati presentata costituisce un *proxy,* locale all'algoritmo, della struttura che il programma invocante passa durante la chiamata; è proprio su tale proxy che l'algoritmo può lavorare per trovare la soluzione da restituire alla fine dell'elaborazione. Il pattern *Proxy*[8] è dunque alla base dell'idea che è stata sviluppata nella creazione del subsystem Struttura Dati Algoritmo: tale sottosistema riproduce fedelmente il dominio dei dati presente nel package Scambio Dati, preesistente allo sviluppo del componente in quanto concordato in fase di discussione preliminare del progetto; inoltre il sottosistema introduce le opportune ottimizzazioni in termini di occupazione di memoria e, per ogni classe in esso contenuta, le operazioni a cui il workflow può delegare compiti per svolgere le sue funzioni. La compatibilità totale tra i due formati è dimostrata dalla presenza di costruttori, per le classi in Struttura Dati Algoritmo, che accettano i dati in formato ScambioDati, e di metodi toSD() che, come si può immaginare, restituiscono un'istanza dell'oggetto corrente in formato ScambioDati.

Se ci si addentra più a fondo nelle caratteristiche del diagramma UML mostrato in figura 4.0, si può osservare una struttura rigorosamente gerarchica: al vertice della gerarchia troviamo la classe Sessione. Il motivo per cui è stata sviluppata una soluzione di questo genere va ricercato nella complessità degli stati che è necessario gestire per una sessione di laurea: un istanza di Sessione può presentare una varietà enorme di configurazioni, alcune lecite e alcune non lecite; vi è dunque necessità di avere sotto controllo in ogni istante e per ogni operazione la liceità della stessa e la consistenza dello stato dell'istanza. Le due soluzioni possibili sono un controllo a priori o un controllo a posteriori. Lo svantaggio del controllo a posteriori consiste nel fatto che, se un operazione, dopo essere stata eseguita, porta in una configurazione errata,

[8] Vedi Appendice A – Design Patterns Utilizzati

questa deve essere annullata; di contro anche il controllo a priori genera problemi, in quanto necessita di opportuni checks, che prima dello svolgimento di un'operazione, possano affermare se essa è lecita o meno[9]. Si è scelto in fase implementativa di realizzare un controllo a priori dello stato di sessione, realizzando gli opportuni controlli che monitorano lo stato, impediscono lo svolgimento di operazioni non lecite, e inoltre permettono di sapere a priori se un'operazione sarà lecita o meno; questo è stato fatto in quanto presenta notevoli vantaggi durante il processo di autocreazione delle commissioni: è evidente che avere un valore booleano indicante la validità o meno di un'operazione durante un workflow permette di prendere la giusta strada e di effettuare le giuste mosse per ottenere una soluzione ammissibile. Se ritorniamo dunque al problema della struttura gerarchica creata, capiamo subito come sia assolutamente utile sfruttare il vertice della gerarchia, per delegare ad esso tutti i compiti di gestione della consistenza dello stato corrente; il procedimento è semplice: nel momento in cui si vuole compiere un'operazione su un'istanza di sessione, si richiede ad essa se l'operazione è lecita, l'istanza fa gli opportuni controlli ed eventualmente delega agli oggetti contenuti il compito di verificare se le operazioni che essa sta per compiere su di essi sono a loro volta lecite; fatto ciò, se le operazioni nel loro complesso risultano corrette essa da il via libera allo svolgimento dell'operazione, altrimenti ne blocca l'esecuzione. I concetti mostrati sono mutuati dal pattern *Façade*[10] e Sessione, nel nostro caso, svolge appunto la parte della facciata, fornendo tutti i metodi e i controlli, eventualmente con delega, per lo svolgimento di ogni operazione possibile sulla gerarchia di classi presente sotto di essa e gestendo la validità dello stato di ogni istanza.

Attenzione particolare va rivolta all'implementazione del DescrittoreCommissione e delle classi ad esso associate. Come evidenziato nelle schede di responsabilità il descrittore ha il compito di determinare le caratteristiche di un'istanza di commissione, in modo tale da poter stabilire se lo stato di quell'istanza è corretto o meno. Ci sono caratteristiche esprimibili con dati primitivi: per esempio il numero minimo di docenti che devono essere presenti viene rappresentato con un intero; ma anche caratteristiche più complesse, che sono state rese attraverso altre classi associate al descrittore. TipologieCommissari è una di queste. In pratica essa è solo un contenitore per la più importante TipologiaCommissari: quest'ultima ha il compito di tenere traccia del numero a cui diverse tipologie di docenti presenti in commissione devono dare luogo; per esempio è possibile esprimere tramite TipologiaCommissari il concetto che in una commissione debbano essere presenti tre docenti scelti tra due tipi (es. Professori Ordinari o Professori Associati). Purtroppo in questo modo possiamo solo esprimere vincoli di uguaglianza e quindi, per risolvere questa limitazione, è stata inserita un'altra classe, VincoloSottogruppo, che permette di modellare concetti come: all'interno di una tipologia strutturata in un certo modo (es. possiamo mantenere

[9] Creare checks di questo tipo potrebbe sembrare banale per problemi semplici, ma se si pensa alla complessità e alla varietà degli stati che si possono ottenere utilizzando i descrittori di commissione, si comprende subito come un controllo a posteriori sia più semplice da realizzare, ma meno efficace per i fini che ci proponiamo in questo progetto.

[10] Vedi Appendice A – Design Patterns Utilizzati

quella vista in precedenza), vogliamo che un gruppo di tipi di docenti soddisfi un vincolo di minimo. Per esempio nel caso precedente potremmo volere che nella TipologiaCommissari compaia un vincolo tale, che il numero di Professori Ordinari debba essere come minimo uno, oppure che tra i tre compaia almeno un Prof. Ordinario e almeno un Prof. Associato. In tal modo si hanno tutti gli strumenti per generare commissioni di ogni tipo e ovviamente tale rappresentazione è totalmente compatibile con quella presente in Scambio Dati, in modo da poter effettuare conversioni in ambo i sensi.

In conclusione possiamo fare delle considerazioni sulle scelte riguardanti i tipi di associazioni utilizzati nel diagramma UML in figura 4.0. Mentre nella maggior parte dei casi sono state utilizzate delle aggregazioni, tra Sessione e Commissione, Sessione e Docente, Commissione e Docente, Docente e Disponibilità troviamo invece delle composizioni. Il motivo di ciò è evidente, se si pensa al fatto che l'algoritmo ha necessità di gestire più istanze di uno stesso docente con diverse disponibilità. Ciascuna di queste istanze può essere legata ad una determinata sessione o (or – esclusivo) commissione all'interno di una sessione (motivo del *disjoint*), mentre ovviamente una commissione è associata ad una e una sola sessione e se la sessione scompare la commissione non ha più motivo di esistere.

Il Package Scambio Dati

Le classi mostrate in figura 2.2 già hanno mostrato con sufficiente precisione le caratteristiche dell'interfaccia concordata in fase preliminare per il collegamento componente – sistema principale. A livello implementativo non si può fare altro che sottolineare i punti salienti che caratterizzano la connessione:

- Implementazione dell'interfaccia AlgoritmoSD con i suoi metodi: valutaCreazione() e creaSoluzione().
- valutaCreazione() restituisce un elenco di IpotesiSoluzioneSD che contiene tutte le possibili soluzioni ottenibili a partire dai dati forniti.
- creaSoluzione() restituisce l'elenco di commissioni associato all'ipotesi di soluzione passata come parametro.

Mostriamo il diagramma UML delle classi modificato secondo le esigenze implementative:

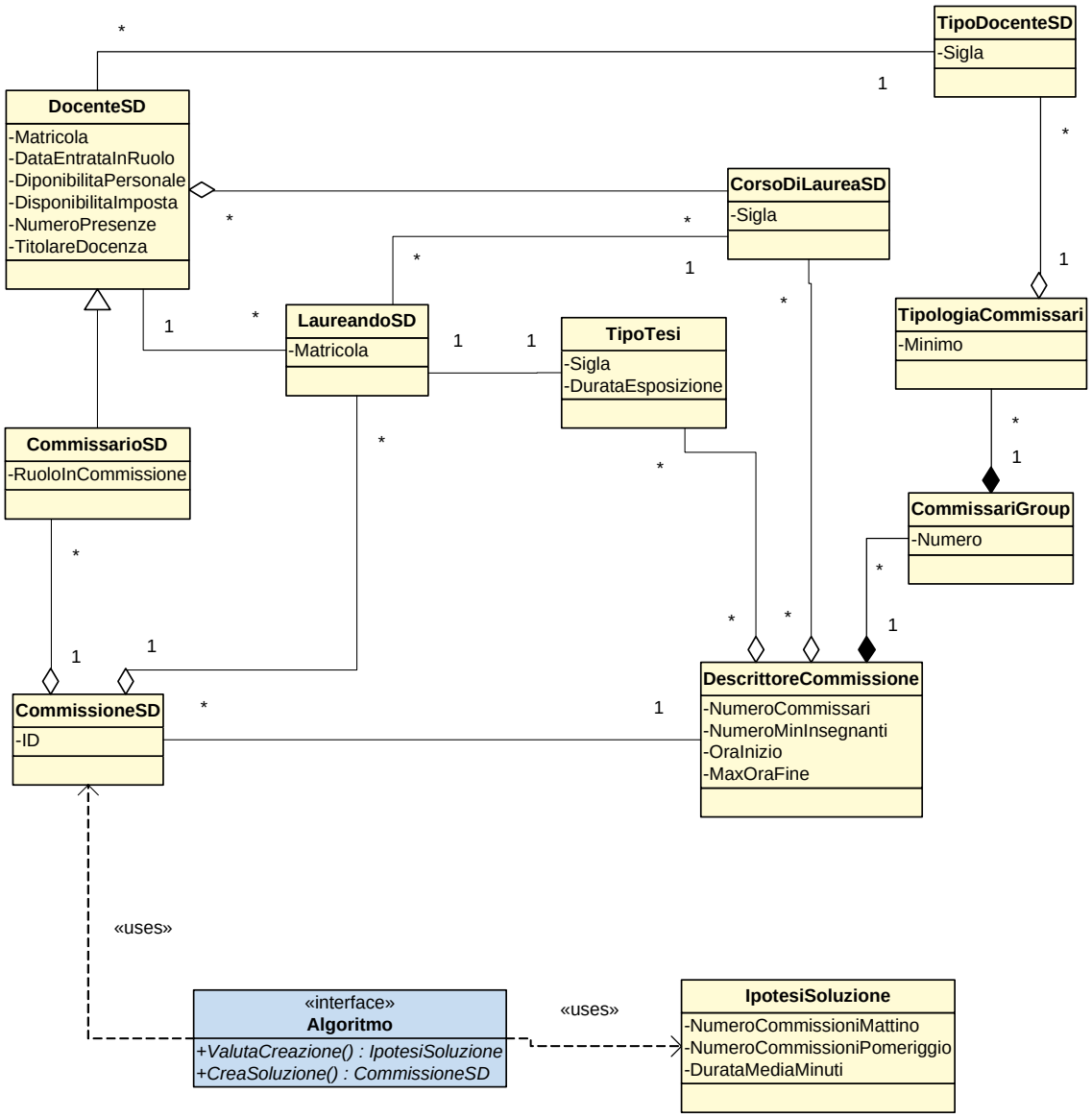


figura 4.1

Le classi qui presentate sono esattamente le stesse che erano presenti in figura 2.2. Vengono ora mostrati gli attributi e i metodi salienti di ogni classe. Non è stato modificato nulla in fase implementativa, in quanto queste classi costituiscono un <<middleware>> non modificabile (a meno di errori evidenti o di cambiamenti nelle specifiche), pattuito in fase di discussione e progettazione preliminare e stabilito immutabile assieme alle regole di interfacciamento.

È possibile notare che il protocollo di connessione segue un *pattern* ampiamente utilizzato in situazioni di questo genere, ovvero il *Bridge*[11]*.* Tale pattern crea appunto un *ponte* tra due mondi, nel nostro caso il sistema principale e il componente algoritmo. Utilizzando tale *pattern*, coadiuvato dal *Template Method*[12]*,* che prevede la dichiarazione di metodi *template* poi implementati in opportune classi derivate, risulta possibile per il sistema

[11] Vedi Appendice A – Design Patterns Utilizzati
[12] Vedi Appendice A – Design Patterns Utilizzati

principale variare a piacere il componente algoritmo e per il componente algoritmo operare in modo indipendente, dopo aver implementato l'interfaccia dichiarata dal sistema; questo dal momento che i metodi sono *template* e possono essere ridefiniti a piacere senza vincoli. L'interfaccia *bridge* è AlgoritmoSD nel package Scambio Dati e i due *template methods* sono i suoi due metodi che vengono poi ridefiniti in Algoritmo, classe presente tra le principali coinvolte nel workflow e altro attore nell'implementazione dei due pattern presentati.

Infine ricordiamo che le classi presenti in scambio dati modellano solo la realtà di interesse per un qualunque algoritmo destinato alla connessione con il sistema principale, non modellano tutto il dominio dei dati trattato dal sistema principale. Si rimanda dunque alla documentazione di quest'ultimo per una più completa visione del dominio dei dati in tale sistema e delle problematiche relative alla persistenza su disco e alla presentazione.

Strutture dati contenenti il workflow

Verrà condotta in questo paragrafo un'analisi *data-driven* delle principali classi contenenti il workflow. Per un'analisi dettagliata di quest'ultimo si rimanda al paragrafo successivo (*Analisi Progettuale del Workflow*).

Il diagramma delle classi concettuale mostrato in figura 2.0 dà già un'idea di come il componente gestisca le proprie strutture dati e di come la classe Algoritmo e la classe IpotesiDiSoluzione interagiscano con le altre classi contenute nel package Scambio Dati e nel subsystem Struttura Dati Algoritmo.

Osserviamo ora il diagramma delle classi modificato:

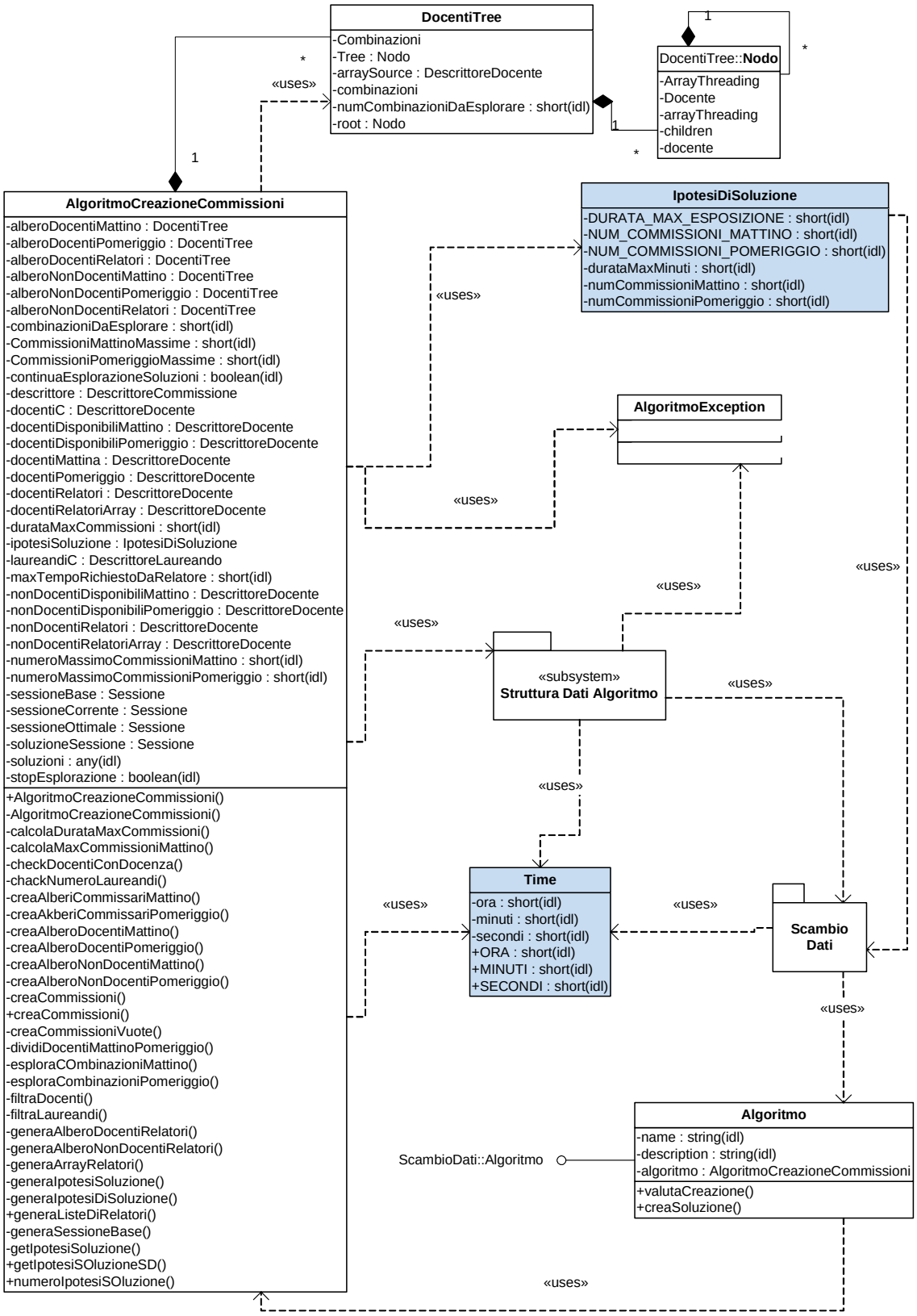


figura 4.2

È possibile osservare quali nuove classi entrano in gioco:

- *Time*. Classe adibita alla gestione delle strutture temporali.
- *AlgoritmoCreazioneCommissioni*. Il core dell'elaborazione.

- *DocentiTree*. Altro punto centrale nell'elaborazione.

Le classi che appaiono in azzurro sono quelle non direttamente coinvolte nell'elaborazione, mentre le altre sono quelle che contengono il *main flow*.

Notiamo subito che la classe Algoritmo, che avevamo indicato come il fulcro dell'elaborazione, dato che essa implementava l'interfaccia esterna AlgoritmoSD, risulti ora completamente sgravata da ogni compito che non sia quello di invocare AlgoritmoCreazioneCommissioni. Questo è stato fatto per due motivi: il primo è una sorta di assicurazione contro i cambiamenti che fa uso del pattern *Adapter*[13]; si cerca cioè di evitare che cambiamenti nell'interfaccia Scambio Dati possano portare a modifiche nella classe principale dell'elaborazione, che una volta testata non dovrebbe essere più modificata per motivi che non siano legati a correzioni di bugs o a modifiche strutturali dell'algoritmo. Il secondo deriva dal fatto che durante l'implementazione, in realtà, non ci si è neppure curati di implementare l'interfaccia di AlgoritmoSD, grazie al pattern *Adapter* si è potuto lavorare in completa libertà, delegando poi ad Algoritmo (appunto classe *adapter*) il compito di implementare l'interfaccia e adattare ad essa la struttura di AlgoritmoCreazioneCommissioni.

Se osserviamo l'interfaccia di AlgoritmoCreazioneCommissioni notiamo una quantità enorme di metodi e attributi. Il motivo di questa abbondanza non sta in un sovraccarico di responsabilità, ma semplicemente nell'uso sistematico del pattern *Template Method* durante la progettazione dell'algoritmo. In realtà i metodi fondamentali (e pubblici) sono pochissimi: il costruttore, e i metodi getIpotesiSoluzioneSD e creaCommissioni(IpotesiSoluzioneSD), che poi coincidono con le esigenze di interfaccia in AlgoritmoSD. Tutti gli altri non fanno altro che implementare sottoprocedure contenute nel workflow principale e presente nel metodo getIpotesiSoluzione(). Dunque ecco spiegata l'abbondanza di metodi nell'interfaccia; ma cosa dire del numero spropositato di attributi di classe? Purtroppo questo è un problema legato al modo in cui C# gestisce il multithreading: nel momento in cui si invoca un metodo generando un nuovo thread è necessario rispettare un interfaccia per il metodo stesso, che stabilisce la totale assenza di argomenti e il tipo di ritorno a void. Dunque non è possibile passare parametri ed è necessario far lavorare i processi su variabili di istanza[14].

La classe IpotesiDiSoluzione era già stata trattata nell'analisi statica, in ogni caso essa contiene informazioni legate ad un'ipotesi generata dall'algoritmo

---

[13] Vedi Appendice A – Design Patterns Utilizzati

[14] Probabilmente questo è dovuto al modo in cui si è deciso di gestire il multithreading: fonte MSDN Jan 2002 – gestione del multithreading tramite utilizzo della classe Thread.
La classe Thread presenta il seguente costruttore (interfaccia):
public Thread(
ThreadStart start
);
ThreadStrat è un delegate che presenta la seguente interfaccia:
public delegate void ThreadStart();
Parameters
The declaration of your event handler must have the same parameters as the **ThreadStart** delegate declaration.

dopo la valutazione iniziale: numero di commissioni ottenibili per il mattino, per il pomeriggio e durata massima di una commissione. È compito di AlgoritmoCreazioneCommissioni tenere traccia dei collegamenti tra un'IpotesiDiSoluzione e la relativa lista di commissioni da restituire nel caso in cui venga richiesta la creazione delle commissioni associate a tale ipotesi.

Infine insieme ad AlgoritmoCreazioneCommissioni, la classe più importante nell'economia del workflow è DocentiTree. Tale classe implementa una struttura ad albero multi-radice incaricata di generare, a fronte di una lista di docenti, un certo numero (passato come parametro) di disposizioni degli elementi di tale lista ordinate secondo una certa politica: tale politica varia a seconda di come è implementato il metodo *Compare* in DescrittoreDocente, o in generale negli elementi della lista passata. Vedremo in seguito come viene implementata la creazione e l'esplorazione dell'albero, per ora basti sapere che la creazione è parallela, mentre l'esplorazione è sequenziale. La classe è in grado di restituire disposizioni distribuite tra tutte quelle possibili e ordinate. La politica di ordinamento permette di avere la certezza, esplorando all'interno del workflow tali disposizioni seguendo l'ordine dalla minore alla maggiore (in termini di indici), che la prima soluzione ammissibile trovata sia anche la migliore. Ovviamente questo è vero a causa delle politiche di esplorazione adottate e dei vincoli progettuali imposti: una soluzione è migliore di un'altra se è più bilanciata in termini di orario e se contiene docenti che sono stati presenti meno volte in commissione rispetto agli altri disponibili. Abbiamo la garanzia che la prima soluzione ammissibile sia anche l'ottima, perché esploriamo prima le disposizioni che hanno nei primi slots docenti con meno presenze (e che quindi vengono inseriti per primi, se possono esserlo) e perché il bilanciamento viene valutato in maniera piuttosto complessa: le commissioni vengono bilanciate nell'orario valutando differenti disposizioni (ottenute sempre con DocentiTree) dei relatori assieme ai loro laureandi, e poi inserendo i laureandi privi di relatori come bilanciamento finale nelle commissioni in cui l'inserimento dei docenti non relatori ha portato a soluzioni ammissibili.

Analisi progettuale del workflow

È stato messo in risalto nel precedente paragrafo come la classe in cui risieda il workflow principale sia AlgoritmoCreazioneCommissioni e come un ruolo importante venga tuttavia svolto anche dal codice di DocentiTree, a cui è delegato il compito di generare le combinazioni poi esplorate al fine di generare le commissioni partendo dai docenti.

In questo paragrafo verranno dunque affrontate le problematiche legate non ai dati ma al flusso della computazione all'interno del componente.

Il seguente diagramma UML degli stati mostra il comportamento della classe DocentiTree, nel momento in cui viene istanziata, ovvero le azioni svolte dal suo costruttore.

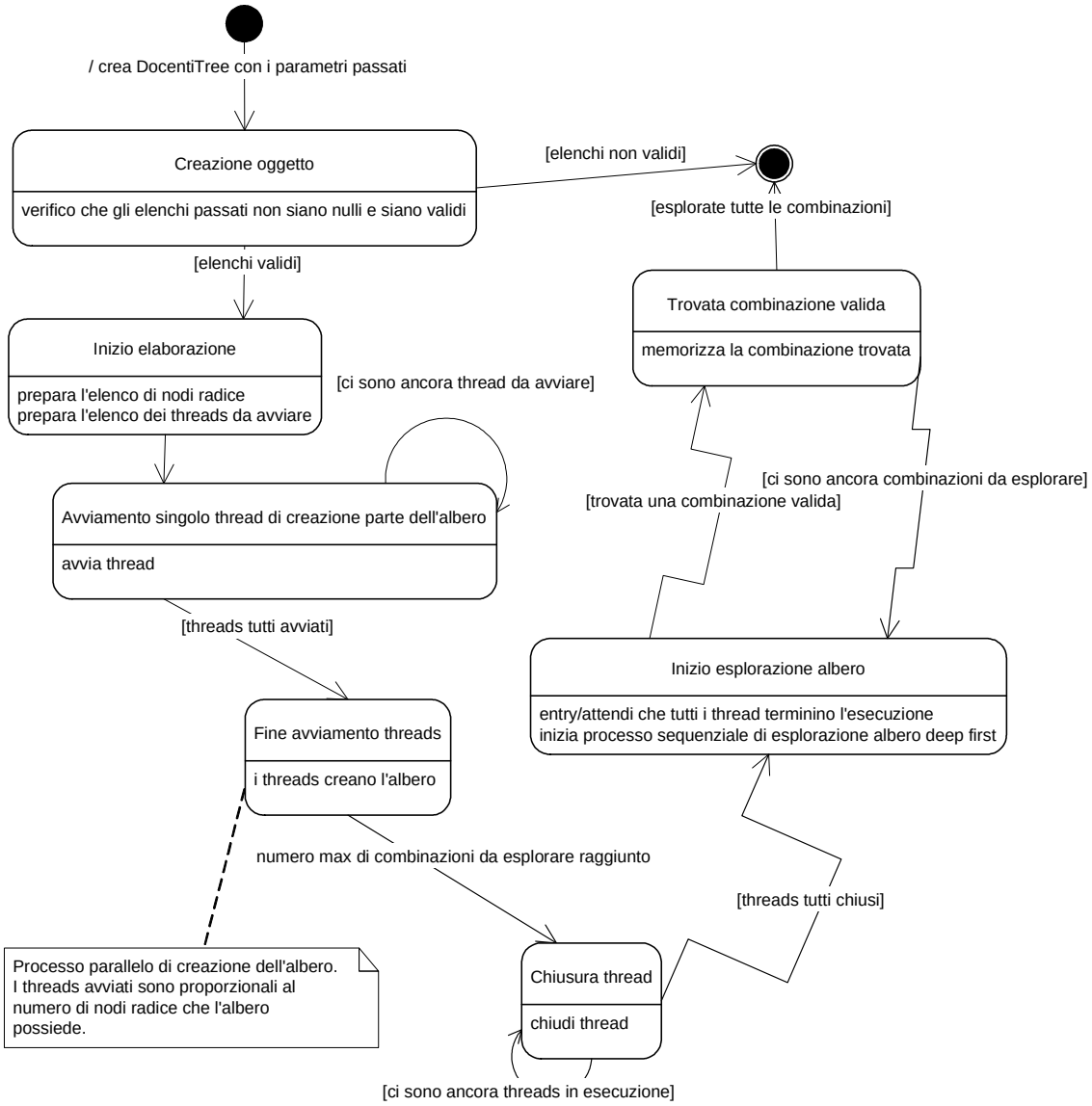


figura 4.3

Le responsabilità di tale classe sono già state messe in evidenza nel paragrafo precedente, focalizziamo quindi ora l'attenzione su come vengono ottenute le disposizioni a partire dalla lista iniziale di Docenti (o Oggetti se si preferisce). La figura 4.3 dà un'idea generale di come procede il flusso di lavoro e di quali azioni vengono svolte: si noti l'avviamento dei thread nel processo parallelo di generazione dell'albero; è questa una delle caratteristiche più notevoli di questo flusso. Sfruttando il parallelismo messo a disposizione da ambienti come Windows 2000 o Windows Xp, si generano *parallelamente* i sottoalberi relativi ai nodi radice creati in precedenza; vi è un nodo radice per ogni docente nella lista passata all'inizio. Generando tali sottoalberi in parallelo e bloccando i vari thread creati ad un istante preciso, è possibile ottenere una serie di alberi parzialmente generati. L'istante preciso è controllato attraverso una variabile monitor che indica il numero di foglie massimo complessivo a cui arrivare nella creazione degli *n* alberi.

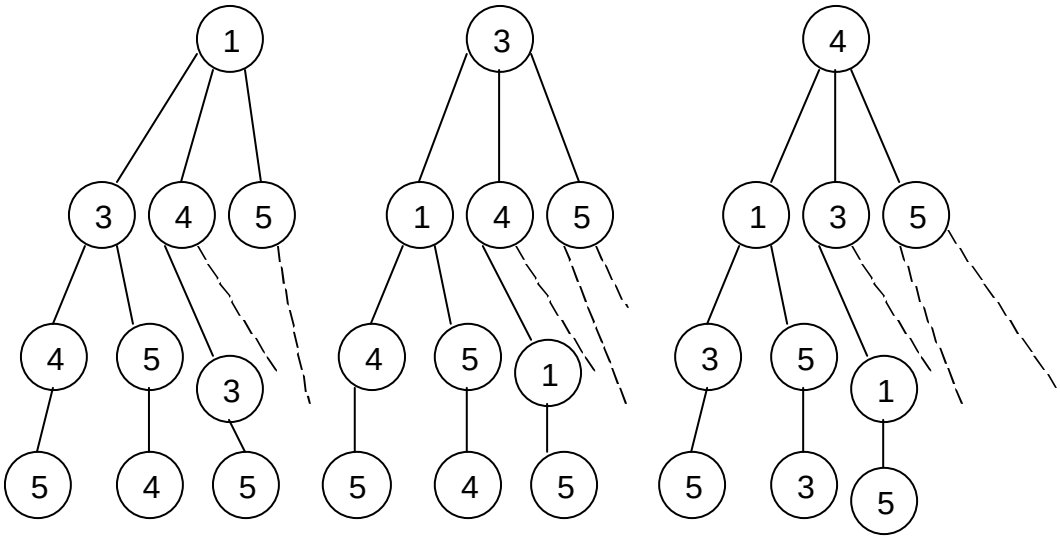


figura 4.4

Osservando l'immagine 4.4 si può osservare la logica attraverso cui gli alberi sono creati: l'immagine si riferisce al caso di 4 docenti con associate le presenze 1,3,4,5 e sviluppa solo 3 dei 4 alberi richiesti per esaurire tutte le n! permutazioni. Notiamo che gli alberi sono parzialmente sviluppati e ciò non a caso visto che questa è la tecnica usata per limitare le combinazioni esplorate. Immaginando che il numero in ogni nodo sia il numero di presenze associato ad un certo docente, si noti che i nodi figli di un nodo padre vengono sempre generati rispettando il loro ordine, quindi viene generato prima il nodo figlio con numero di presenze più basso nel nostro caso. Ad ogni nodo inoltre è associata una lista, che è parte della lista iniziale di docenti: essa è la lista iniziale meno il docente nel nodo corrente e i docenti che compaiono nei nodi padre del nodo corrente fino alla radice. Ogni volta che viene generato un nodo figlio, a questo è associato un docente (sempre rispettando l'ordine di presenze) della lista parziale corrente, dalla quale viene tolto, poi al nodo figlio viene associata la lista parziale risultante. Si prosegue così fintanto che vi sono docenti nella lista, quando la lista è vuota, si è raggiunta una foglia e si incrementa di conseguenza la variabile monitor. È da notare che diversi docenti possono avere lo stesso numero di presenze ma vengono distinti comunque per matricola e vengono conseguentemente generate tutte le opportune disposizioni. Bloccando la creazione degli alberi tramite la variabile monitor, si ottengono alberi parziali che, se esplorati in modo sequenziale e deep first, generano una sottoparte di tutte le possibili disposizioni, che resta ordinata secondo il criterio messo in risalto nel paragrafo precedente (ovvero rispecchiando quanto implementato nel metodo Compare degli oggetti trattati).

Ad esempio esplorando nel modo corretto gli alberi parziali creati sopra è possibile ottenere le seguenti combinazioni.

1	3	4	5
1	3	5	4
1	4	3	5
3	1	4	5

3	1	5	4
3	4	1	5
4	1	3	5
4	1	5	3
4	3	1	5

figura 4.5

Le permutazioni in figura 4.5 non esauriscono tutte le *n!* possibili, ma ne coprono solo una parte e risultano ordinate, in quanto per prime vengono quelle in cui compaiono per primi i docenti con meno presenze.

Ovviamente come si nota nell'elenco di permutazioni in figura 4.5 vengono considerate valide solo le quelle di *n* oggetti e non i rami parzialmente sviluppati dell'albero, per ottenere ciò si controlla che arrivati ad una foglia il numero di nodi dalla foglia alla radice sia *n.*

Infine dopo aver recuperato tutte le permutazioni, è possibile recuperarne l'elenco tramite l'opportuna proprietà di istanza: Combinazioni.

Possiamo ora dare uno sguardo al diagramma UML delle attività del costruttore di DocentiTree in figura 4.6, che pone più enfasi sulle peculiarità dell'implementazione. Anche qui sono messi in risalto i processi paralleli ed è mostrato quello sequenziale di esplorazione. Viene mostrata la gestione della variabile monitor con maggiore precisione e anche il parallelismo di *n* processi (se *n* sono gli elementi della lista iniziale).

Analizziamo ora invece il diagramma di stato della classe AlgoritmoCreazioneCommissioni, ovvero il fulcro del workflow, presentato in figura 4.7. Anche qui l'attenzione è focalizzata principalmente sui passi fondamentali che l'algoritmo compie, senza approfondire i dettagli implementativi. Nella stesura di questo diagramma è stato molto utile l'apporto della metodologia usata per stendere l'algoritmo (il *template method* di cui in precedenza si è parlato). I nomi dei vari stati, infatti, spesso rievocano i metodi dichiarati nello scheletro dell'algoritmo e solo in seguito implementati.

Osserviamo dunque i passi fondamentali:

- Conversione dei dati
- Analisi del numero massimo di commissioni ottenibile
- Generazione delle combinazioni di relatori
- Esplorazione delle combinazioni di relatori
- Generazione delle combinazioni di docenti
- Esplorazione delle combinazioni di docenti
- Inserimento finale dei laureandi se viene trovata nei passi precedenti una soluzione ammissibile.
- Decremento del numero massimo di commissioni ottenibile e iterazione finche il numero massimo di commissioni ottenibili non va a zero.

Questi passi sono ovviamente solo concettuali e mostrano la logica astratta che l'algoritmo segue.

Più utili ai fini della comprensione sono sicuramente i due diagrammi delle attività per il main flow della classe AlgoritmoCreazioneCommissioni (ovvero per il suo metodo generaIpotesiSoluzione()) mostrati nella figura 4.8 e seguente.

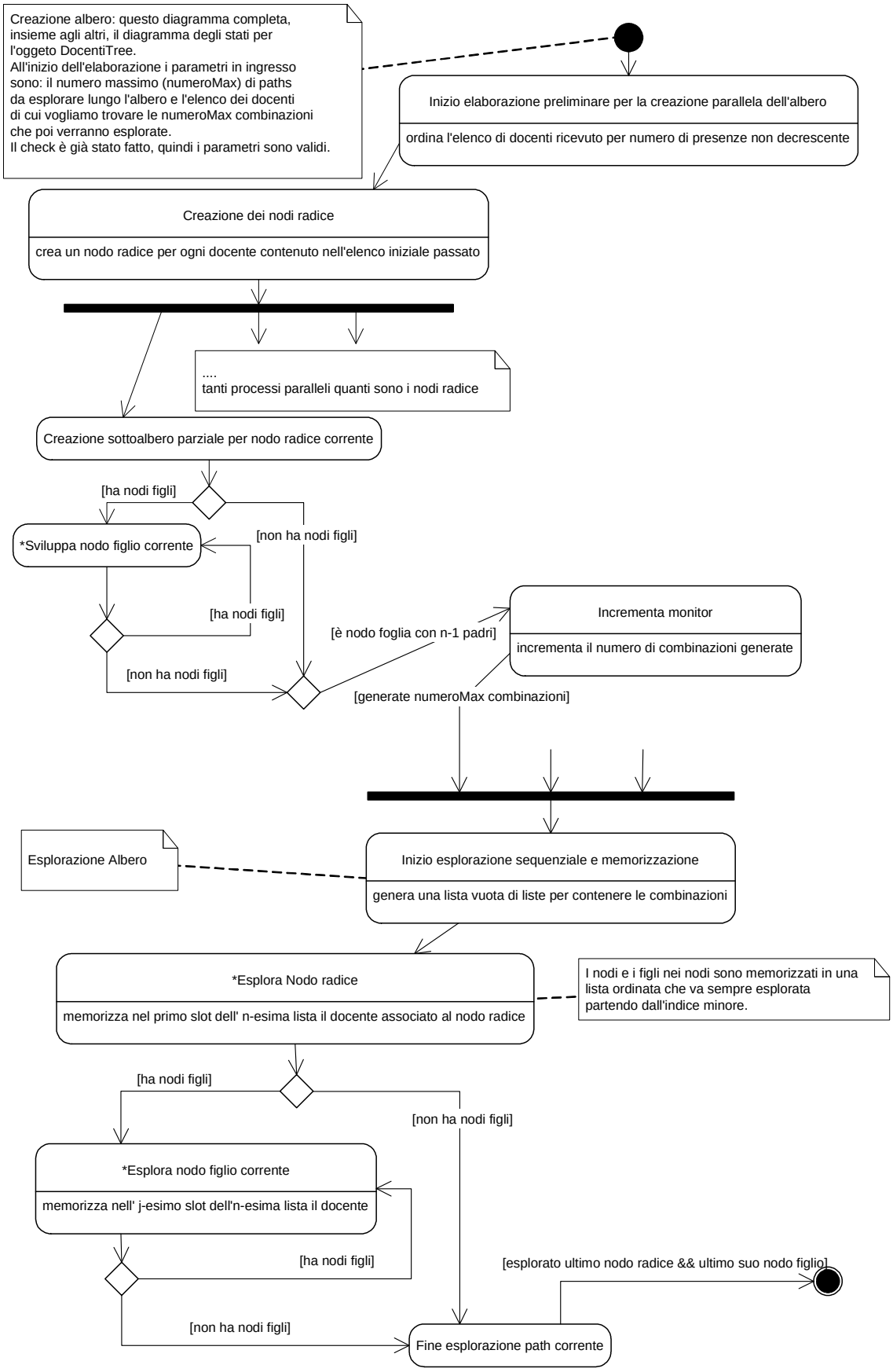

Creazione albero: questo diagramma completa, insieme agli altri, il diagramma degli stati per l'oggeto DocentiTree.
All'inizio dell'elaborazione i parametri in ingresso sono: il numero massimo (numeroMax) di paths da esplorare lungo l'albero e l'elenco dei docenti di cui vogliamo trovare le numeroMax combinazioni che poi verranno esplorate.
Il check è già stato fatto, quindi i parametri sono validi.
Inizio elaborazione preliminare per la creazione parallela dell'albero
ordina l'elenco di docenti ricevuto per numero di presenze non decrescente
Creazione dei nodi radice
crea un nodo radice per ogni docente contenuto nell'elenco iniziale passato
....
tanti processi paralleli quanti sono i nodi radice
Creazione sottoalbero parziale per nodo radice corrente
[ha nodi figli]
[non ha nodi figli]
*Sviluppa nodo figlio corrente
[ha nodi figli]
[non ha nodi figli]
[è nodo foglia con n-1 padri]
Incrementa monitor
incrementa il numero di combinazioni generate
[generate numeroMax combinazioni]
Esplorazione Albero
Inizio esplorazione sequenziale e memorizzazione
genera una lista vuota di liste per contenere le combinazioni
*Esplora Nodo radice
memorizza nel primo slot dell' n-esima lista il docente associato al nodo radice
I nodi e i figli nei nodi sono memorizzati in una lista ordinata che va sempre esplorata partendo dall'indice minore.
[ha nodi figli]
[non ha nodi figli]
*Esplora nodo figlio corrente
memorizza nell' j-esimo slot dell'n-esima lista il docente
[ha nodi figli]
[esplorato ultimo nodo radice && ultimo suo nodo figlio]
[non ha nodi figli]
Fine esplorazione path corrente

figura 4.6

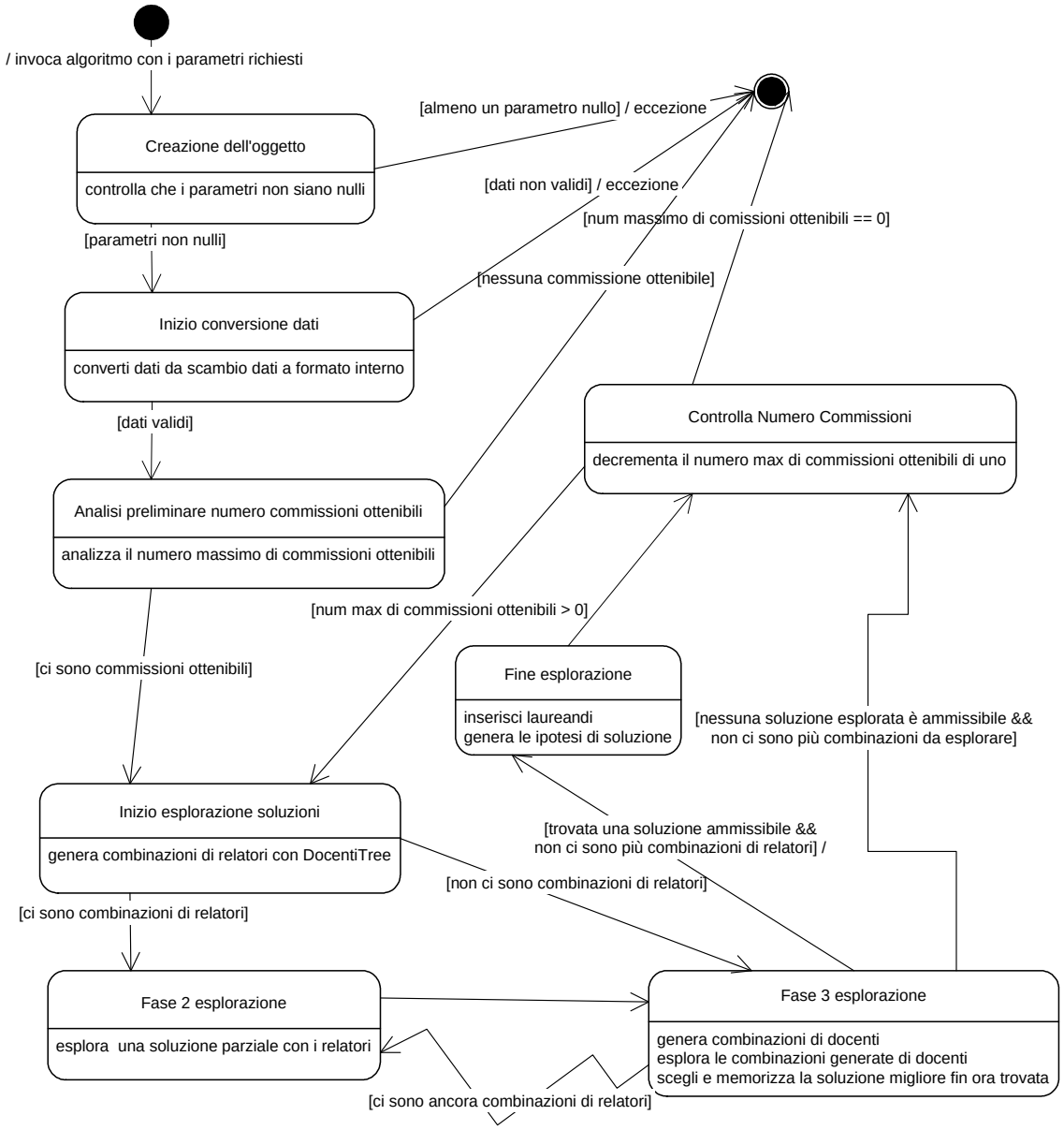

/ invoca algoritmo con i parametri richiesti
Creazione dell'oggetto
controlla che i parametri non siano nulli
[almeno un parametro nullo] / eccezione
[parametri non nulli]
Inizio conversione dati
converti dati da scambio dati a formato interno
[dati non validi] / eccezione
[num massimo di comissioni ottenibili == 0]
[nessuna commissione ottenibile]
[dati validi]
Controlla Numero Commissioni
decrementa il numero max di commissioni ottenibili di uno
Analisi preliminare numero commissioni ottenibili
analizza il numero massimo di commissioni ottenibili
[num max di commissioni ottenibili > 0]
[ci sono commissioni ottenibili]
Fine esplorazione
inserisci laureandi
genera le ipotesi di soluzione
[nessuna soluzione esplorata è ammissibile &&
non ci sono più combinazioni da esplorare]
Inizio esplorazione soluzioni
genera combinazioni di relatori con DocentiTree
[trovata una soluzione ammissibile &&
non ci sono più combinazioni di relatori] /
[non ci sono combinazioni di relatori]
[ci sono combinazioni di relatori]
Fase 2 esplorazione
esplora una soluzione parziale con i relatori
Fase 3 esplorazione
genera combinazioni di docenti
esplora le combinazioni generate di docenti
scegli e memorizza la soluzione migliore fin ora trovata
[ci sono ancora combinazioni di relatori]

figura 4.7

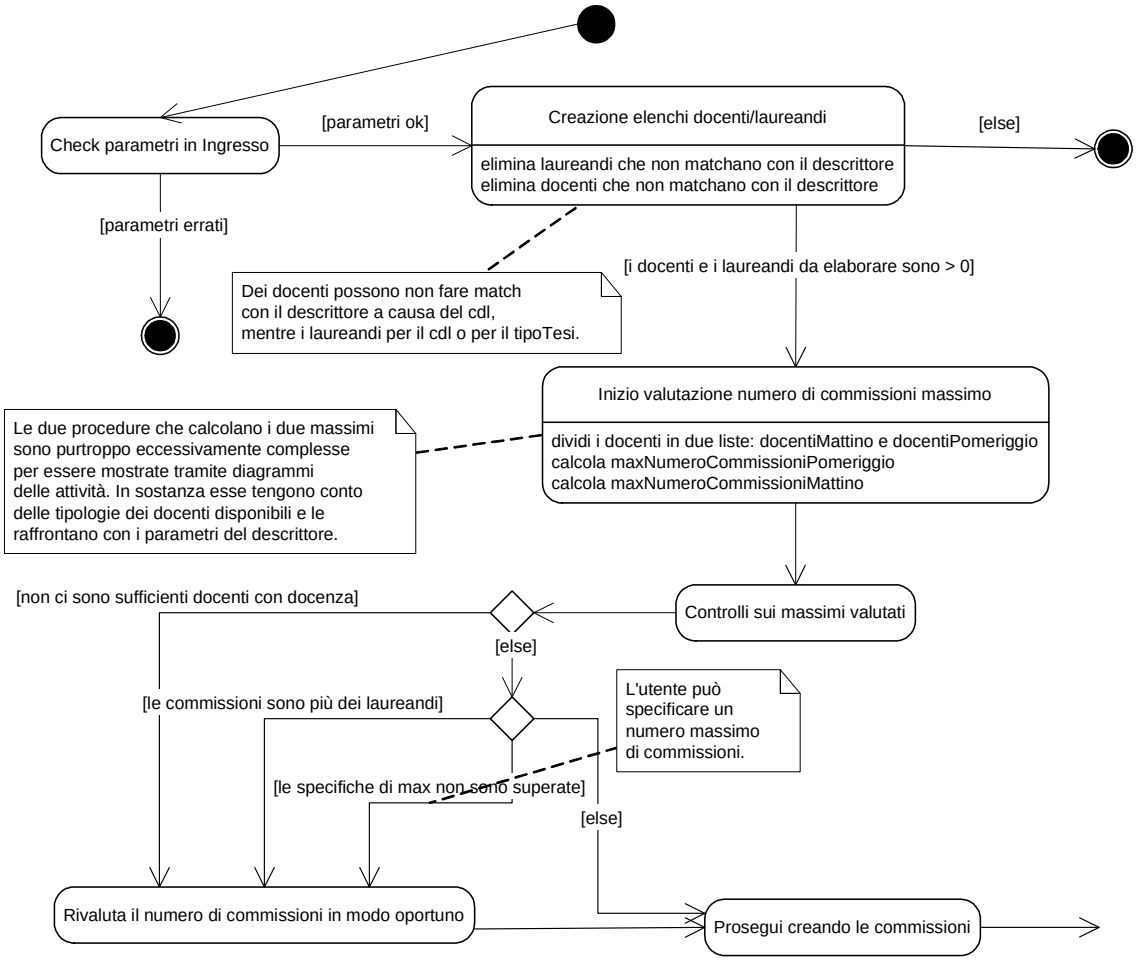


figura 4.8

L'immagine 4.8 mostra le operazioni compiute dal metodo generaIpotesiSoluzione() di AlgoritmoCreazioneCommissioni per compiere i controlli iniziali sui dati passati e per valutare il numero massimo di commissioni ottenibile con i dati (docenti) passati.

Ricordiamo che i parametri richiesti dal metodo sono:

- Un'istanza di DescrittoreCommissione.
- Un array di istanze di DescrittoreDocente
- Un array di istanze di DescrittoreLaureando
- Un valore indicante il max numero di commissioni al mattino
- Un valore indicante il max numero di commissioni al pomeriggio
- Un valore indicante il max numero di combinazioni da esplorare nelle istanze di DocentiTree usate nell'elaborazione.

Purtroppo, come evidenziato nella nota in figura, non è stato possibile inserire il flusso che effettivamente calcola il numero massimo di commissioni ottenibile. Si rimanda al codice vero e proprio in tal caso a causa della complessità delle procedure scritte.

Notiamo che alla fine del diagramma non vi è il consueto simbolo di chiusura, ma una freccia che *idealmente* punta al diagramma della figura successiva dove è presente il diagramma UML delle attività per le parti successive dell'elaborazione del metodo precedente (fig. 4.9).

Il diagramma risulta piuttosto intricato e presenta i principali passi compiuti nell'elaborazione delle soluzioni, cioè delle liste di commissioni.

Da notare è la creazione delle combinazioni di docenti, che sfrutta DocentiTree, il calcolo della durata massima delle commissioni, che può portare all'uscita nel caso in cui il valore trovato superi quello indicato come massimo nelle specifiche, e la creazione di una sessione iniziale, che sarà poi quella clonata ed elaborata ad ogni ciclo per ottenere una nuova soluzione da confrontare con quella ottima eventualmente già trovata. La parte iniziale dell'esplorazione, quella cioè in cui vengono esplorate le combinazioni di relatori non docenti prima e di relatori docenti poi, è sequenziale, come pure la seconda parte, quando si esplorano le combinazioni di docenti disponibili al mattino e di docenti disponibili al pomeriggio. Se la soluzione trovata ad un certo punto dell'esplorazione risulta ottima, cioè migliore delle precedenti (in termini di bilanciamento di orario o numero di laureandi inseriti), ma non ammissibile, cioè non accettabile in termini di tempo massimo per una commissione, si prosegue nell'esplorazione, altrimenti se è ammissibile ci si ferma, si memorizza la soluzione trovata e si prosegue nell'esplorazione di soluzioni con un numero minore di commissioni fino a che il numero non arriva a zero o la durata delle commissioni di una soluzione non è in media troppo alta. La politica di decremento delle commissioni è tale da diminuire prima il numero delle commissioni al pomeriggio fino a zero e solo dopo il numero di quelle al mattino.

Dopo che sono state esplorate e memorizzate tutte le possibili soluzioni, è possibile accedere alle liste di commissioni create tramite il

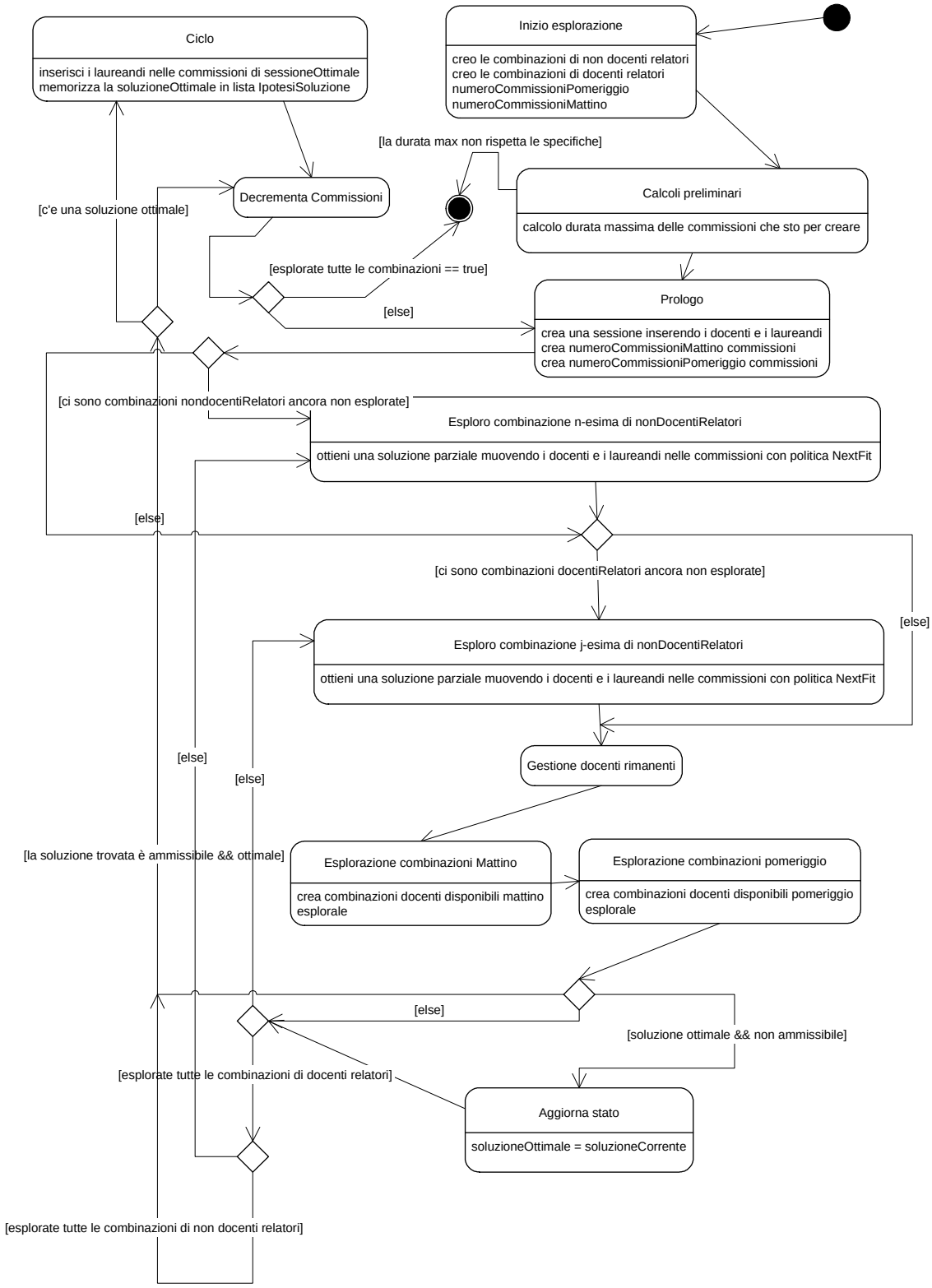


figura 4.9

metodo creaCommissioni() che riceve come parametro un'istanza di IpotesiDiSoluzione e restituisce un array di istanze di Commissione.

Il processo di esplorazione delle combinazioni di docenti nella generazione di una soluzione[15]

Viene riportato di seguito uno stralcio del codice invocato nella procedura generaIpotesiSoluzione(). È questa la parte più significativa del metodo, che quindi andremo a realizzare. Notiamo l'estrema semplicità: in solo circa 40 righe è concentrato tutto il codice che *genera* le commissioni. Ovviamente nei punti chiave sono inserite invocazioni che delegano certe operazioni ad altri metodi privati. Il codice iniziale è contenuto in un ciclo while che esplora le soluzioni possibili finché è necessario continuare l'esplorazione (variabile monitor continuaEsplorazioneSoluzioni); abbiamo detto in precedenza che si prosegue nell'esplorazione decrementando il numero di commissioni secondo la politica che è stata illustrata: questo compito è svolto dalla procedura invocata nella riga 53. Una procedura analoga è però invocata anche nella riga 49. Perché dunque vengono invocate queste due procedure? Quali sono i criteri per invocare l'una piuttosto che l'altra? In base alla struttura dell'algoritmo è possibile che dopo aver esplorato le disposizioni relative ad una certa configurazione di commissioni (mattino/pomeriggio), non sia stato possibile reperire alcuna soluzione ammissibile per tale configurazione; questo sarà allora il caso in cui verrà invocata la procedura in riga 53, che decrementa il numero di commissioni in configurazione in modo tale da diminuire in modo alterno le commissioni prima al mattino e poi al pomeriggio, valutando cosi tutte le possibili combinazioni e non perdendo alcuna soluzione, se questa esiste. Ad esempio se si parte con 2 commissioni al mattino e 2 al pomeriggio e non è possibile trovare soluzioni per tale configurazione allora si esplora la successiva: 1 mattino e 2 pomeriggio; poi, sempre se non vi sono soluzioni, 2 mattino e 1 pomeriggio e via dicendo… 1 mattino e 1 pomeriggio, fintantoché non ne viene trovata una ammissibile. Diverso è invece il caso in cui venga trovata una commissione ammissibile; se accade ciò allora la procedura invocata è quella in riga 49, che decrementa prima le commissioni al pomeriggio fino a 0 e poi quelle al mattino, finché entrambi i valori non sono 0 o non viene trovata una configurazione di commissioni che in media non permette di inserire i laureandi rispettando i limiti massimi di tempo imposti dal descrittore. Se ad esempio abbiamo 20 laureandi, le cui tesi in media durano 10 minuti, avremo bisogno almeno di 200 minuti da distribuire tra le varie commissioni; se invece le commissioni sono 4, allora perché la soluzione sia ammissibile sarà necessario che la durata massima di una commissione sia settata su un valore superiore ai 50 minuti, altrimenti la soluzione sarà scartata a priori, come è intuibile osservando le prime due istruzioni del ciclo.

```
1. while(continuaEsplorazioneSoluzioni)
2. {
3. ///Calcolo durata max commissione
4. this.calcolaDurataMaxCommissioni();
5. if(durataMaxCommissioni < maxTempoRichiestoDaRelatore)
6. durataMaxCommissioni = maxTempoRichiestoDaRelatore;
7. if(durataMaxCommissioni > descrittore.DurataMassimaInMinuti)
```

[15] Questo paragrafo si riferisce al componente definitivo, già dotato delle modifiche di cui si parla nell'Appendice B

```
8. break;

9. ///Creo commissioni
10. sessioneCorrente = new Sessione(sessioneBase);
11. this.creaCommissioniVuote();

12. ///Inizio ciclo piu esterno
13. sessioneOttimale = null;
14. stopEsplorazione = false;
15. soluzioneSessione = new Sessione(sessioneCorrente);
16. if(alberoNonDocentiRelatori.Combinazioni.Count > 0)
17. {
18. this.esploraNonDocentiRelatori();
19. }
20. else
21. {
22. sessioneIntermedia = new Sessione(sessioneCorrente);
23. if(alberoDocentiRelatori.Combinazioni.Count > 0)
24. {
25. this.esploraDocentiRelatori();
26. }
27. else
28. {
29. ///Completamento inserimento Docenti in commissioni
30. sessioneCorrente = soluzioneSessione;
31. soluzioneSessione = null;
32. this.esploraDocenti();
33. }
34. }

35. ///Non uso le combinazioni di docenti relatori && non docenti
    relatori
36. ///invocato solo se non è già stata trovata una soluzione
    ammissibile.
37. if(sessioneOttimale == null || !sessioneOttimale.check())
38. {
39. this.sessioneCorrente = new Sessione(this.sessioneBase);
40. this.creaCommissioniVuote();
41. this.esploraDocentiFinale();
42. }

43. ///Controllo che le commissioni in sessioneOttimale siano ok
44. if(sessioneOttimale != null && sessioneOttimale.check())
45. {

46. ///Inserisco i laureandi
47. this.inserisciLaureandi();

48. //Setto Presidente e Segretario
49. for(int com = 0; com < sessioneOttimale.getNumeroCommissioni();
    com++)
50. {
51. sessioneOttimale.getCommissione(com).setPresidente();
52. sessioneOttimale.getCommissione(com).setSegretario();
53. }

54. ///Calcola la durata max effettiva
55. ///e il numero di commissioni mattino/pomeriggio
56. this.creaIpotesiValutandoDurataMax();
```

```
57. #if TRACE
    DebugHandler.writeDebugMessage("Esplorazione Soluzione OK");
58. #endif
59. this.decrementaCommissioniMaxIpotesiGenerata();
60. }
61. else
62. {
63. this.decrementaCommissioniMaxIpotesiNonGenerata();
64. }
```

In seguito viene creata una soluzione vuota con l'opportuno numero di commissioni e viene iniziata l'esplorazione delle disposizioni correnti al fine di creare una soluzione. Le disposizioni a questo punto disponibili sono:

- disposizioni dei non docenti relatori
- disposizioni dei docenti relatori

Vengono dunque esplorate queste disposizioni: esplorare significa provare ad inserire docenti in commissione secondo l'ordine in cui essi appaiono in una determinata disposizione; si provano le varie combinazioni che possono essere ottenute a partire dai due elenchi di docenti e non docenti relatori, e viene infine tenuta la soluzione in cui sono stati inseriti più docenti insieme ai loro laureandi e che risulta più bilanciata delle altre in termini di orari. Si valutano poi le disposizioni dei docenti rimasti, e si esplorano anche queste finché non si ottiene una soluzione ammissibile (procedure da riga 16 a riga 34). Se questa non viene trovata, allora si tenta in altro modo (riga 37): si prova ad ottenere una soluzione ammissibile esplorando le sole disposizioni di docenti (non tenendo conto di chi è relatore e di chi non lo è), ed inserendo solo in seguito i laureandi. Tale soluzione ha un notevole inconveniente, rischia cioè di non trovare inseriti tutti i relatori, ed è infatti l'ultimo tentativo che viene fatto prima di abbandonare l'esplorazione della configurazione corrente per passare alla successiva. Infine le ultime operazioni compiute nelle righe finali sono l'inserimento dei laureandi che non sono già stati inseriti con i loro relatori, valutazione dei tempi reali per ogni commissione e creazione di una IpotesiDiSoluzione relativa alla soluzione ammissibile trovata, se questa esiste.

Testing del componente

Sono state effettuate prove di vario genere sul componente: prove con dati che rispecchiano le reali condizioni di utilizzo (in termini di numero di istanze etc.), prove di stress (con dati in numero sovradimensionato rispetto al reale utilizzo) e prove con dati insufficienti in modo da testare i controlli e la prontezza di risposta in assenza di soluzioni.

Particolarmente interessanti sono i dati scaturiti da una serie di prove effettuate con dati generati a partire da funzioni random (native in C#). È possibile osservare nei grafici e nei tabulati di tali prove il comportamento del componente a fronte di situazioni di stress. Tali prove sono state particolarmente utili come fonte di riflessione per apportare miglioramenti significativi al componente inizialmente progettato, senza modificarne eccessivamente le caratteristiche e il funzionamento. Le modifiche fatte prevedono l'introduzione di controlli in punti chiave del flusso dove l'elaborazione risultava lenta in presenza di moli di dati superiori alla media[16].

Benchmarking effettuato con configurazioni random di dati in ingresso[17]

Le prove sono state svolte generando docenti con disponibilità, tipologia e cdl scelti in modo random tra un pool di scelta. Il descrittore di commissione è stato mantenuto fisso, i laureandi sono stati anch'essi generati in modo random, scegliendo il tipo tesi e il cdl in modo casuale.

L'algoritmo è stato invocato con liste di docenti contenenti un numero via via crescente di entry, tenendo invece fisso a 20 il numero dei laureandi generati; il numero massimo di commissioni da generare al mattino è stato settato a 4 come anche quello al pomeriggio. I due nuovi parametri di cui si parla nell'Appendice B, cioè il limite massimo di esplorazione delle combinazioni di docenti e non docenti relatori è stato fissati rispettivamente a cinque e a due.

Di seguito sono riportati i grafici con le risposte temporali dell'elaborazione in rapporto a variazioni del numero di docenti presentati in ingresso al componente; in ogni casella è presente un numero (minuti.secondi) indicante il tempo di risposta e il numero di soluzioni trovate a fronte della configurazione random presentata in ingresso.

[16] L'elenco completo dei miglioramenti è inserito nell'Appendice B – Considerazioni sull'ottimizzazione del workflow.

[17] Il benchmark è stato efettuato su un personal computer con queste caratteristiche: PIII 800, 512 Kb cache, bus 133 Mhz, 540 MB RAM.

	Numero Docenti = 10	Numero Docenti = 20	Numero Docenti = 40	Numero Docenti = 60	Numero Docenti = 80	Numero Docenti = 100
Prova 1	0.0 (no soluzioni)	0.0 (2 soluzioni)	0.1 (4 soluzioni)	0.4 (4 soluzioni)	0.10 (7 soluzioni)	0.13 (8 soluzioni)
Prova 2	0.0 (no soluzioni)	0.4 (no soluzioni)	0.4 (4 soluzioni)	0.1 (2 soluzioni)	0.28 (6 soluzioni)	0.11 (8 soluzioni)
Prova 3	0.0 (no soluzioni)	0.0 (no soluzioni)	0.2 (4 soluzioni)	3.8 (1 soluzione)	0.7 (7 soluzioni)	0.15 (8 soluzioni)
Prova 4	0.0 (no soluzioni)	0.0 (1 soluzione)	0.1 (3 soluzioni)	4.26 (1 soluzione)	6.30 (2 soluzioni)	0.48 (8 soluzioni)
Prova 5	0.0 (no soluzioni)	0.0 (no soluzioni)	0.2 (4 soluzioni)	0.1(3 soluzioni)	0.15 (7 soluzioni)	2.49 (1 soluzione)
Prova 6	0.0 (no soluzioni)	0.0 (2 soluzione)	0.1 (3 soluzioni)	0.6(4 soluzioni)	0.5 (6 soluzioni)	0.9 (5 soluzioni)
Prova 7	0.0 (no soluzioni)	0.0 (no soluzioni)	0.10 (2 soluzione)	0.4 (6 soluzioni)	0.7 (5 soluzioni)	0.14 (8 soluzioni)
Prova 8	0.0 (no soluzioni)	0.0 (1 soluzione)	0.5 (4 soluzioni)	0.3 (4 soluzioni)	4.29(3 soluzioni)	0.18 (8 soluzioni)
Prova 9	0.0 (no soluzioni)	0.0 (1 soluzione)	0.1 (3 soluzioni)	0.2(3 soluzioni)	10.48(3 soluzioni)	1.51 (7 soluzioni)
Prova 10	0.0 (no soluzioni)	0.0 (no soluzioni)	0.1 (3 soluzioni)	0.2(3 soluzioni)	1.28(7 soluzioni)	0.14 (7 soluzioni)

tabella 4.0

La zona verde mostra i test effettuati oltre i limiti di stress del componente, e che ovviamente hanno presentato picchi notevoli anche se rari. Per controllare questi picchi si può agire sui due nuovi parametri di cui si parla nell'Appendice B, cioè il limite massimo di esplorazione delle combinazioni di docenti e non docenti relatori, riducendo questi parametri è possibile ottenere risposte più pronte, anche se probabilmente verranno perse alcune soluzioni.

Nelle figure 4.10 e 4.11 sono presenti due immagini che graficano i dati raccolti: la prima (fig. 4.10) rappresenta i dati della tabella 4.0, la seconda il tempo medio di risposta per ogni quantità di docenti in ingresso: 10, 20, 40, 60, 80, 100.

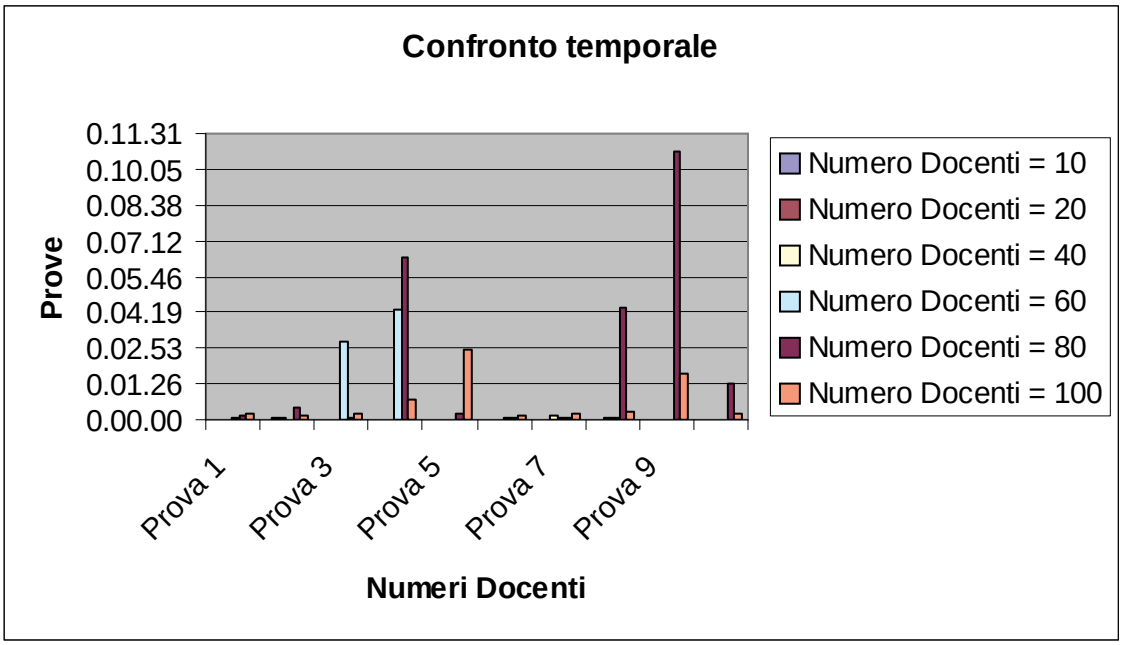


figura 4.10

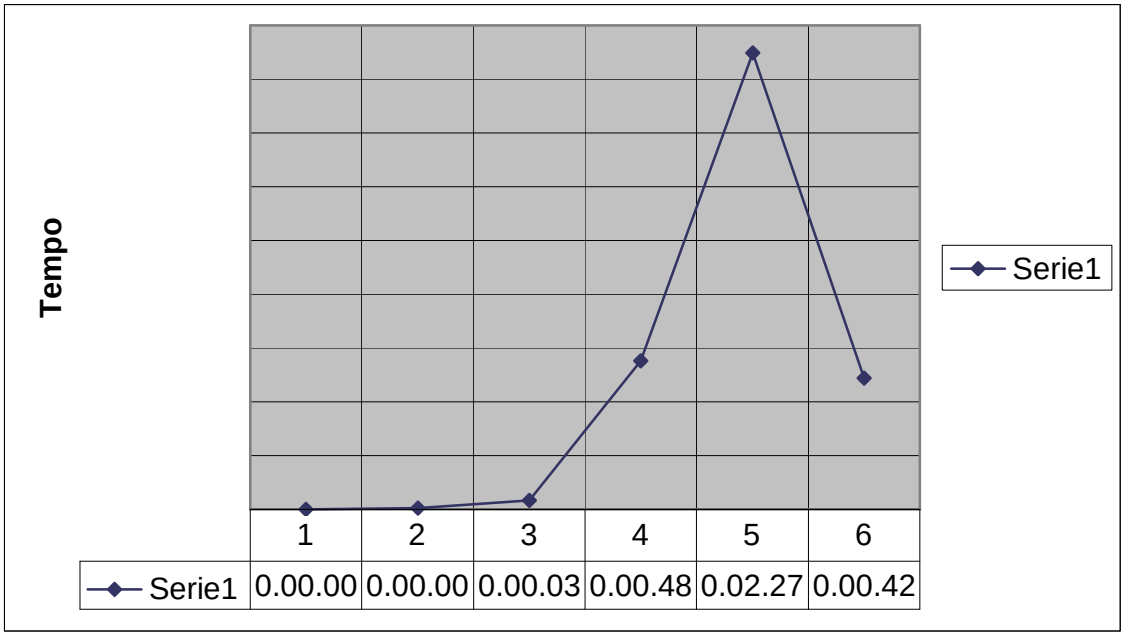


figura 4.11

Osserviamo ora cosa succede se agiamo sui due nuovi parametri, abbassandoli rispettivamente ad 1 e a 2; avremo quindi:

- limite massimo per le combinazioni di docenti relatori = 2
- limite massimo per le combinazioni di non docenti relatori = 1

Aggiorniamo solo le zone del grafico e della tabella inerenti alla zona di stress del componente:

	Numero Docenti = 80	Numero Docenti = 100
Prova 1	1.30 (4 soluzioni)	0.15 (6 soluzioni)
Prova 2	0.6 (6 soluzioni)	0.35 (5 soluzioni)
Prova 3	0.26 (6 soluzioni)	0.55 (6 soluzioni)

Prova 4	0.7 (8 soluzione)	0.14 (7 soluzioni)
Prova 5	0.7 (6 soluzioni)	4.02 (7 soluzioni)
Prova 6	0.7 (8 soluzione)	0.33 (6 soluzioni)
Prova 7	0.8 (6 soluzionI)	0.13 (6 soluzioni)
Prova 8	1.6 (6 soluzioni)	0.15 (6 soluzioni)
Prova 9	0.5 (6 soluzione)	0.17 (6 soluzioni)
Prova 10	0.18 (3 soluzioni)	3.2 (4 soluzioni)

tabella 4.1

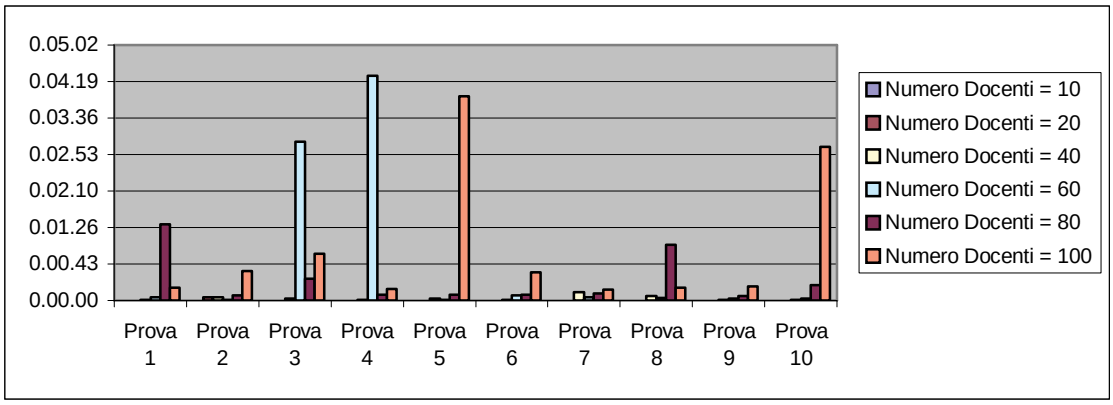


figura 4.12

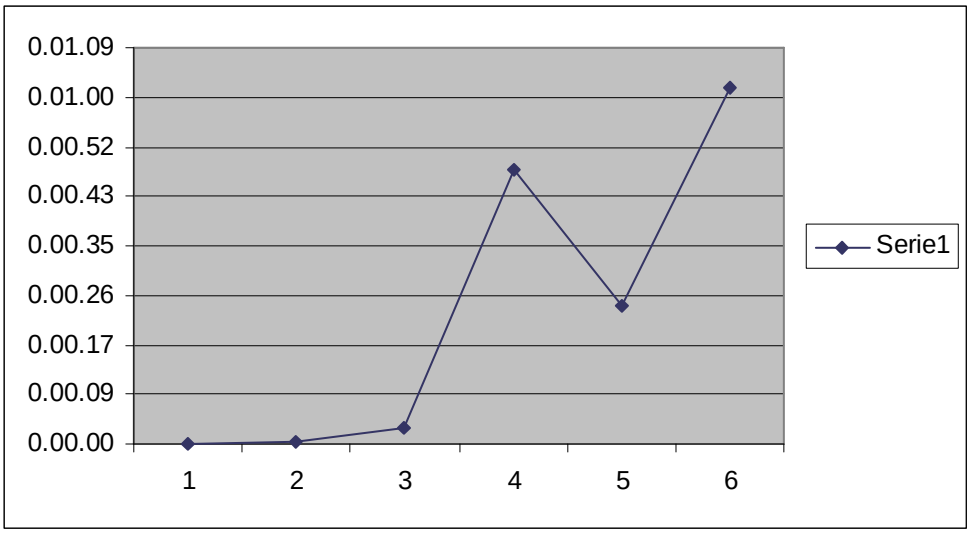


figura 4.13

Sicuramente l'immagine più interessante ai nostri fini è la 4.13 dove possiamo notare che la forma del grafico è cambiata rispetto alla 4.11, e in particolare notiamo la riduzione dei tempi di risposta introdotta dalla limitazione imposta al calcolo grazie ai due nuovi parametri introdotti e utilizzati. Il motivo per cui la curva assume la nuova forma che possiamo osservare è intuibile: l'algoritmo avendo sempre più docenti da gestire dovrà esplorare strutture dati sempre più consistenti e questo è il motivo della crescita dei tempi medi di risposta dal punto 5 (80 docenti) al punto 6 (100 docenti). Viceversa con pochi

docenti a disposizione diminuiscono vertiginosamente le combinazioni da esplorare e dunque ecco il motivo della velocità del punto 3 (40 docenti).

Si può aggiungere in conclusione che non sono stati rilevati crash, loops o eccezioni del componente durante il testing, segno di buona stabilità dal momento che le combinazioni erano del tutto casuali e che le prove sono state numerose.

Appendice A - Design Patterns Utilizzati

Il pattern Prototype

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

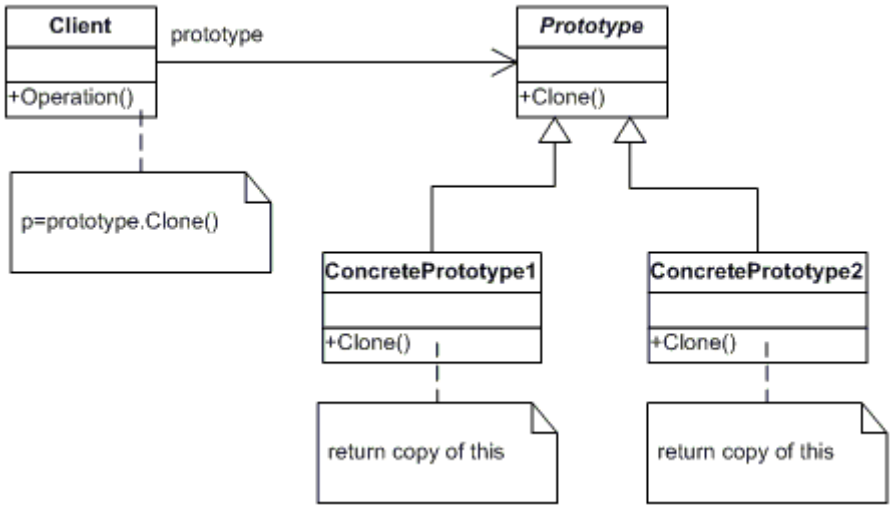


figura A.0

The classes and/or objects participating in the Prototype pattern are:

- **Prototype (ColorPrototype)**
  - declares an interface for cloning itself
- **ConcretePrototype (Color)**
  - implements an operation for cloning itself
- **Client (ColorManager)**
  - creates a new object by asking a prototype to clone itself

Questo pattern è stato ampiamente utilizzato per gestire la duplicazione di oggetti durante il workflow dell'algoritmo.

La versione qui presentata del pattern è stata opportunamente modificata in fase implementativa: non viene ridefinito il metodo clone, ma il costruttore per copia; tutti gli oggetti lo fanno, ma gli oggetti composti non sempre delegano al costruttore per copia dell'oggetto contenuto il compito di duplicarsi; quando le istanze contenute appartengono a classi *frozen* la duplicazione non viene operata e si collegano le istanze già presenti in memoria tramite riferimenti.

Il pattern Chain of Responsability

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

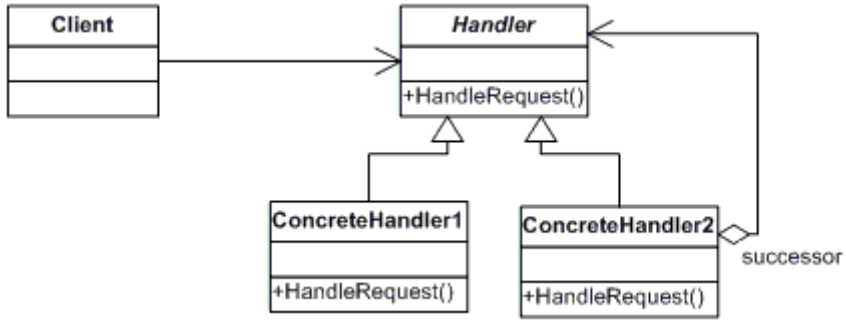


figura A.1

The classes and/or objects participating in this pattern are:

- **Handler (Approver)**
  - defines an interface for handling the requests
  - (optional) implements the successor link
- **ConcreteHandler (Director, VicePresident, President)**
  - handles requests it is responsible for
  - can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- **Client (ChainApp)**
  - initiates the request to a ConcreteHandler object on the chain

Il pattern qui mostrato è utilizzato principalmente per due scopi:

- gestire la duplicazione di oggetti complessi delegando agli oggetti contenuti il compito di duplicarsi (nel caso in cui non siano *frozen* e quindi gestiti tramite riferimenti).
- gestire la creazione dell'albero nelle istanze della classe DocentiTree e la sua esplorazione: a ogni nodo viene delegato il compito di costruire il

sottoalbero che gli compete e di esplorarlo dopo che la creazione è avvenuta.

Il pattern Proxy

Provide a surrogate or placeholder for another object to control access to it.

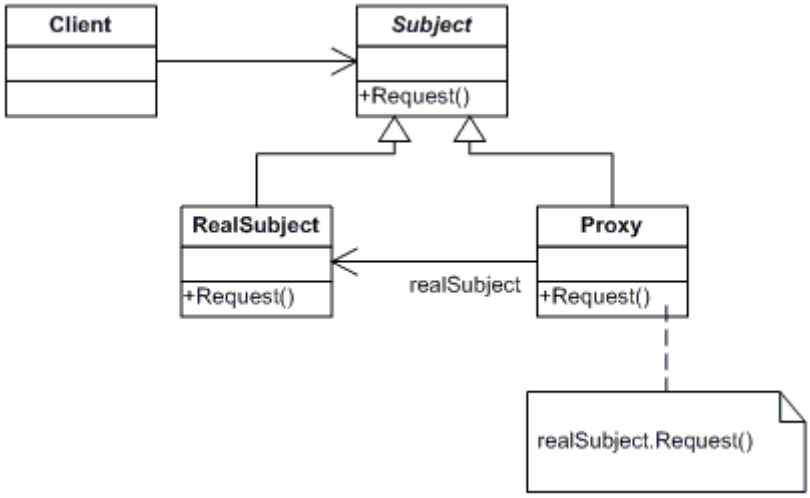


figura A.2

The classes and/or objects participating in the Proxy pattern are:

- **Proxy (MathProxy)**
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can be substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilities depend on the kind of proxy:
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images' extent.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
- **Subject (IMath)**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

- **RealSubject (Math)**
  - defines the real object that the proxy represents.

Tale pattern è alla base dell'idea di riprodurre localmente all'algoritmo le strutture dati su cui lavorare, non utilizzando quindi quelle messe a disposizione dal package ScambioDati in quanto povere di funzionalità e di ottimizzazione. Nuove strutture vengono quindi definite nel sottosistema Struttura Dati Algoritmo: queste sono perfettamente compatibili con quelle definite in ScambioDati, ma risultano ottimizzate ai fini della computazione e ricche di funzionalità a cui il workflow principale può delegare compiti durante l'elaborazione. La compatibilità totale è dimostrata dalla presenza di costruttori che accettano i dati equivalenti ma in formati ScambioDati e di metodi toSD() che come si può immaginare restituiscono un'istanza dell'oggetto corrente in formato ScambioDati.

Il pattern Façade

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

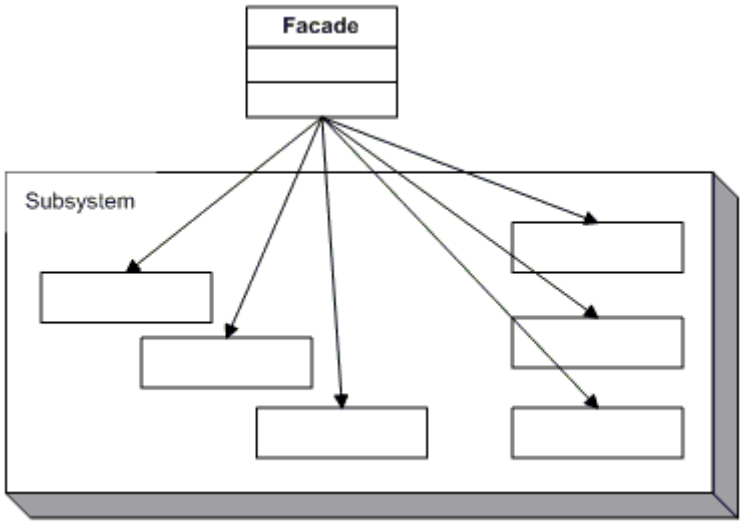


figura A.3

The classes and/or objects participating in the Façade pattern are:

- **Facade (MortgageApplication)**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem classes (Bank, Credit, Loan)**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade and keep no reference to it.

L’utilità di questo pattern per la gestione della correttezza dello stato dell’oggetto Sessione è fondamentale. Gestire lo stato di questo macro oggetto senza un’interfaccia unificata risulterebbe arduo, così come dei checks a posteriori lo sarebbero, vista la necessità, in tal caso, di tenere traccia delle operazioni svolte.

Si può intravedere un utilizzo di tale pattern anche nella struttura di tutto il componente che, nonostante la complessità, offre all’esterno un’interfaccia con solo due metodi.

Il pattern Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

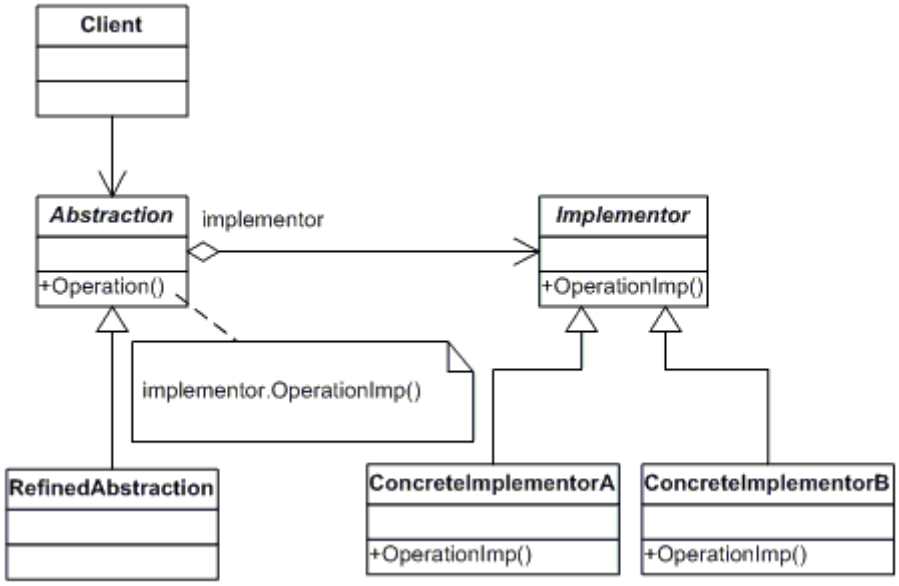


figura A.4

The classes and/or objects participating in the Façade pattern are:

- **Abstraction (BusinessObject)**
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- **RefinedAbstraction (CustomersBusinessObject)**
  - extends the interface defined by Abstraction.
- **Implementor (DataObject)**
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

- **ConcreteImplementor (CustomersDataObject)**
  - implements the Implementor interface and defines its concrete implementation.

Utilizzato nell'interfaccia di collegamento tra sistema principale e componente algoritmo. L'abstraction è la classe AlgoritmoSD nel package ScambioDati. L'implementor è la classe Algoritmo, che si trova tra le principali classi coinvolte nel workflow.

Il pattern Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

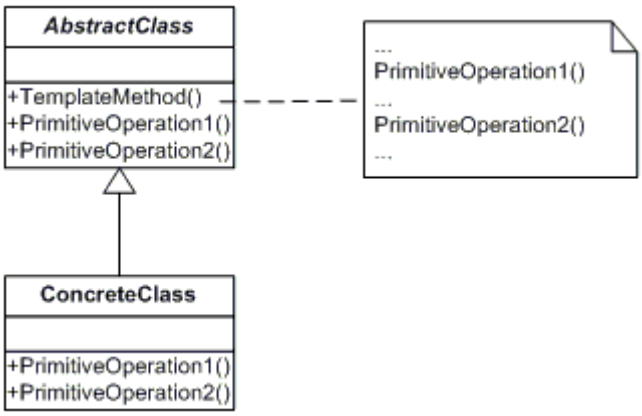


figura A.5

The classes and/or objects participating in this pattern are:

- **AbstractClass (DataObject)**
  - defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm

    - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass (CustomerDataObject)**
    - implements the primitive operations to carry out subclass-specific steps of the algorithm

Questo pattern è stato utilizzato per creare l'interfaccia di comunicazione tra sistema principale e componente algoritmo: i due template methods sono quelli dichiarati in AlgoritmoSD (package Scambio Dati) e implementati in Algoritmo (classi principali coinvolte nel workflow).

Altro impiego fondamentale è stato in fase progettuale nella creazione della classe AlgoritmoCreazioneCommissioni, da principio infatti questa classe era astratta, definiva lo scheletro dell'algoritmo e dichiarava i metodi invocati da quest'ultimo, poi in fase implementativa un collasso verso il basso ha portato all'eliminazione della classe astratta iniziale; nonostante ciò è ancora evidente l'uso di tale tecnica di progettazione visto che la classe AlgoritmoCreazioneCommissioni presenta tuttora i numerosi metodi (ora definiti) che da principio erano stati solo dichiarati.

Ritengo che tale tecnica sia preziosa in fase progettuale perché permette già da subito di immaginare il flusso algoritmico principale senza perdersi nelle sottigliezze implementative delle singole parti dell'algoritmo (da principio solo dichiarate).

Il pattern Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

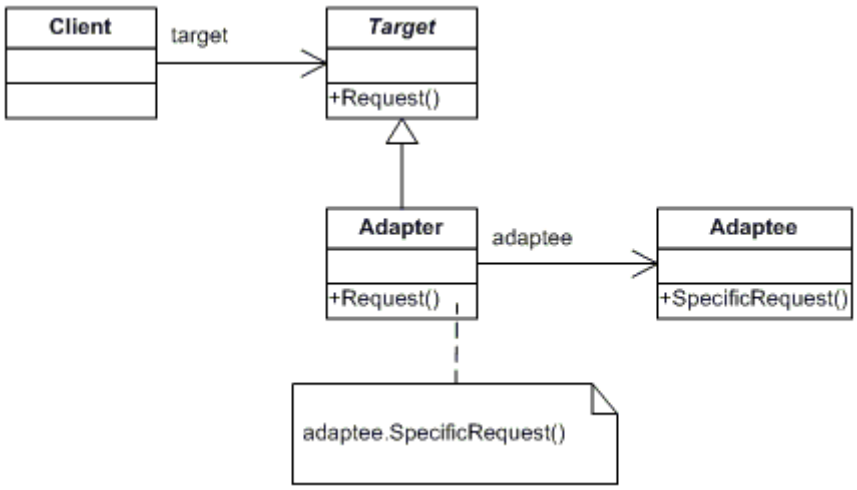


figura A.6

The classes and/or objects participating in the Adapter pattern are:

- **Target (ChemicalCompound)**
  - defines the domain-specific interface that Client uses.
- **Adapter (Compound)**
  - adapts the interface Adaptee to the Target interface.
- **Adaptee (ChemicalDatabank)**
  - defines an existing interface that needs adapting.
- **Client (AdapterApp)**
  - collaborates with objects conforming to the Target interface.

Pattern utilizzato per adattare le caratteristiche della classe AlgoritmoCreazioneCommissioni, contenente il core del workflow alla classe Algoritmo che implementa l'interfaccia AlgoritmoSD di Scambio Dati.

Appendice B – Considerazioni sull'ottimizzazione del workflow

In seguito ai test e al benchmarking a cui è stato sottoposto il componente, sono state apportate delle modifiche al componente, che qua vengono brevemente elencate.

Il punto cruciale di debolezza del sistema prima di tali modifiche era la lentezza a fronte di dati *stressanti,* ovvero a fronte di grandi molti di dati. Uscendo dalle condizioni *nominali* di operatività, stabilite attorno ai 60 docenti/chiamata e 30 laureandi/chiamata, il componente risultava, con probabilità di 1/10, lento. Ciò significa che almeno una volta ogni 10 chiamate il componente impiegava più di 5 minuti per valutare le commissioni e a volte svolgeva questi calcoli per poi non restituire soluzioni. I motivi di ciò vanno ricercati nelle scelte progettuali adottate nel workflow del componente, anche se ciò non significa che tali scelte siano state sbagliate; il fatto che il componente valutasse un certo numero di combinazioni prima di decidere che non era

possibile ottenere soluzioni era stato previsto in prima analisi, infatti non si era dato peso da principio alla complessità della computazione. Le modifiche introdotte dopo la fase di testing hanno riconfermato la validità delle scelte prese in sede progettuale, non sempre infatti un componente già progettato può essere modificato con facilità: la rigorosa struttura modulare del codice ha permesso di rilevare i punti critici in cui la computazione si soffermava a valutare sequenze inutili, e tramite opportuni controlli è stato possibile introdurre dei *jumps* che hanno snellito il flusso.

Ecco dunque l'elenco delle principali modifiche apportate in fase di testing.

- È stato introdotto un controllo alla fine di ogni valutazione di ipotesi di soluzione per una configurazione di commissioni: se non è stata prodotta un'ipotesi valida il programma non segue la consueta tecnica di decremento delle commissioni per l'ipotesi successiva, ma in modo alterno decrementa o il numero di commissioni massimo al mattino o il numero di commissioni massimo al pomeriggio.
- La valutazione delle commissioni ha ora un passo in più. Se si osserva la figura 4.9 si nota che, dopo l'esplorazione delle combinazioni di relatori (docenti e non docenti) e il tentativo di riempimento delle commissioni tramite le combinazioni dei docenti rimasti, nulla viene più tentato e si decrementa immediatamente il numero di commissioni da creare. Ora invece prima di decrementare il numero di commissioni si fa un altro tentativo di riempire le commissioni, si prendono cioè tutti i docenti senza discriminare tra relatori e non, si creano le commissioni, e solo in seguito si accorpano i laureandi ai loro relatori, finendo poi la creazione delle commissioni nel modo consueto tramite l'inserimento dei laureandi spaiati, la definizione dei segretari e dei presidenti.
- Sono stati introdotti due parametri opzionali nell'interfaccia di invocazione di Algoritmo: il primo è il numero massimo di combinazioni di relatori docenti da esplorare, il secondo è il numero massimo di combinazioni di relatori non docenti da esplorare; insieme al numero massimo di combinazioni da esplorare negli alberi prodotti durante la valutazione, questi due parametri offrono una configurabilità quasi totale del componente nei confronti del tempo di attesa dei risultati. Abbassando tali parametri, che sono di default settati ai massimi di sistema, è possibile esplorare un numero inferiore di configurazioni e ottenere risposte più pronte alle richieste. Ovviamente va tenuto in conto che esplorando meno soluzioni si ottengono risultati peggiori, ma da studi condotti su algoritmi e processi computazionali simili a questo[18] è stato dimostrato che troncare la ricerca dopo un certo numero di passi statisticamente non penalizza la bontà dei risultati ottenuti; ciò significa che in qualche caso si perderanno delle soluzioni, ma saranno più i vantaggi in termini di tempo ottenuti dal troncamento dell'esplorazione che gli svantaggi portati dalla perdita di soluzioni.

[18] A.Lodi,S.Martello,D.Vigo – Recent Advances on Two-Dimensional Bin Packing Problems.

Appendice C – Considerazioni sullo sviluppo in team e sull'uso del pattern View

Lo sviluppo in *team* di un'applicazione complessa comporta spesso ingenti problemi di organizzazione, cioè di suddivisione dei compiti, e di integrazione dei componenti prodotti. Spesso risulta infatti difficile anche solo dividere i domini di influenza di ogni componente del *team.* In fase di analisi preliminare, quando cioè si tenta di tracciare delle linee di confine per delimitare gli spazi concettuali entro i quali ognuno potrà compiere delle scelte individuali, spesso ciascuno pensa di poter dettar legge su ogni aspetto legato alle proprie problematiche e raramente ci si ricorda che un aspetto può risultare spesso e volentieri condiviso. Parlando di *aspetti condivisi*, si fa qui riferimento a parti del sistema modellato in fase di analisi concettuale, che non possono essere univocamente assegnate ad un componente piuttosto che ad un altro. Lo sviluppo, anche solo concettuale, di queste parti risulta spesso ostico, in quanto ogni persona nel *team* vorrà modellare e sviluppare secondo le proprie esigenze tali entità; accadrà così che coloro che si troveranno a modellare queste entità, sfrutteranno tutti i vantaggi del caso e modelleranno il problema secondo le loro esigenze; mentre gli altri dovranno conformarsi a modelli che probabilmente risulteranno *scomodi* da essere adattati alle entità da loro modellate secondo diversi schemi mentali. Sarebbe dunque necessario trovare un modo per evitare di vincolare gli uni alle scelte degli altri e per permettere a tutti di essere posti nelle condizioni di modellare le entità coinvolte nel loro ambito secondo le proprie esigenze, indipendentemente dalle esigenze altrui.

Come in tutti i progetti in *team* dunque, anche in questo si è presentato puntualmente il problema degli *aspetti condivisi.* In particolare, la limitazione più grande a cui si è cercato di fare fronte è stata quella relativa alla divisione degli ambiti di influenza. In fase di analisi preliminare si sono infatti individuate delle strutture dati comuni e condivise, alle quali ciascun componente del *team* doveva poter accedere per letture o modifiche. Inoltre gli usi a cui tali strutture erano destinate erano molteplici e differenti. Se si fosse dunque incaricato un solo sviluppatore del compito di modellare tali strutture, le sue scelte sarebbero state parziali e avrebbero limitato gli altri componenti del *team* nei loro modelli.

Nello sviluppo di questo progetto si è dunque rivelata particolarmente utile una fase, che identificheremo col nome di *team synchronizing.* L'importanza di questa fase è centrale: essa ha permesso a ogni individuo di operare in modo assolutamente autonomo nello sviluppo del proprio componente, e ha dato la certezza che, una volta completati tutti gli *n* componenti del sistema, questi avrebbero potuto essere integrati senza problemi.

Dove si inserisce dunque questa fase e in cosa consiste?

Nel classico modello a cascata iterativo, i componenti del *team* inizialmente si dividono i compiti, analizzano il proprio dominio di interesse e lo modellano; poi, dopo ogni fase: analisi statica, analisi dinamica etc., si incontrano per confrontare il lavoro svolto. Gli incontri sono dunque molteplici e richiedono spesso una quantità di tempo elevata. Quello che si è fatto per ovviare a tutti questi incontri (e bisogna ricordare che maggiore è il numero degli incontri maggiore è il tempo perso a discutere e a non sviluppare), ottenendo lo stesso

risultato, è stato di suddividere, in modo concettuale e non troppo preciso inizialmente, i domini di interesse, lasciando a ciascuno, da subito, un po' di tempo per analizzare le problematiche legate al proprio dominio. Ognuno in tale lasso di tempo ha dovuto seguire i classici passi dell'analisi statica e dinamica del problema, curandosi di modellare concettualmente il proprio dominio dei dati. Successiva a questa prima fase è quella di *team synchronizing.* Durante tale fase viene eseguita una vera e propria *unione* dei modelli sviluppati nell'analisi precedente da ciascuno. Di comune accordo vengono creati dei *middleware* destinati a modellare proprio quegli ambiti di dati e funzionalità condivisi tra più componenti. Tali *middleware* devono risultare opportunamente espressivi, cioè devono permettere la rappresentazione corretta dei dati nel dominio condiviso, modellando tutte le proprietà che ciascuno ha identificato; tuttavia devono anche essere essenziali, cioè non devono modellare niente che non sia semplice rappresentazione dei dati di scambio. A questo punto il problema è risolto: i dati così modellati verranno utilizzati per realizzare la struttura suggerita dal pattern *View.* Ogni componente del *team* potrà modellare la propria realtà secondo le proprie esigente, curandosi di inserirvi dei metodi che possano in qualsiasi momento convertirla nel formato del *middleware.* Sempre nel *middleware* ciascuno potrà trovare funzionalità, eventualmente implementate da altri, sotto forma di interfacce, e potrà così sviluppare il proprio codice, non curandosi di come tali interfacce vengono da altri implementate.

Avendo in mano il *middleware* ciascun componente del sistema potrà essere sviluppato in modi e tempi indipendenti da quelli degli altri. Non vi saranno priorità nello sviluppo; quando tutti i componenti saranno stati sviluppati e testati, infine, si potranno integrare con la certezza che il sistema risulterà funzionante, vista l'estrema indipendenza delle parti raggiunta con la definizione iniziale del *middleware.*

Un possibile variante alla metodologia qua esposta potrebbe essere quella che divide in modo netto dati e funzionalità. Le funzionalità potrebbero essere definite tramite opportune interfacce in un *middleware*, nel modo sopra presentato, mentre i dati potrebbero essere definiti non con strutture *object-oriented,* ma con dati in *Xml-Schema*, e potrebbero essere passate tramite dei *DOM Xml*.

Di seguito viene riportato un semplice caso di studio legato alla metodologia sopra presentata. Il caso di studio mostra un problema analogo a quello affrontato nella progettazione del sistema software presentato in questo testo. I diagrammi mostrati in seguito modellano un'astrazione che deve dare l'idea dei tratti essenziali del problema e del pattern applicato per la risoluzione.

Problema: Gestire un progetto portato avanti da un team di persone, senza che il lavoro condotto da una influisca sulle scelte di un'altra.

Soluzione: Definire a priori, cioè durante la prima iterazione del modello a cascata iterativo di sviluppo del software, una struttura di ScambioDati che costituirà un *middleware* di comunicazione tra i vari componenti del team; il *middleware* potrà essere concepito per comunicazioni 1-1, 1-*, *-1 o *-*, sarà cioè universale.

L’immagine seguente mostra un esempio di comunicazione:

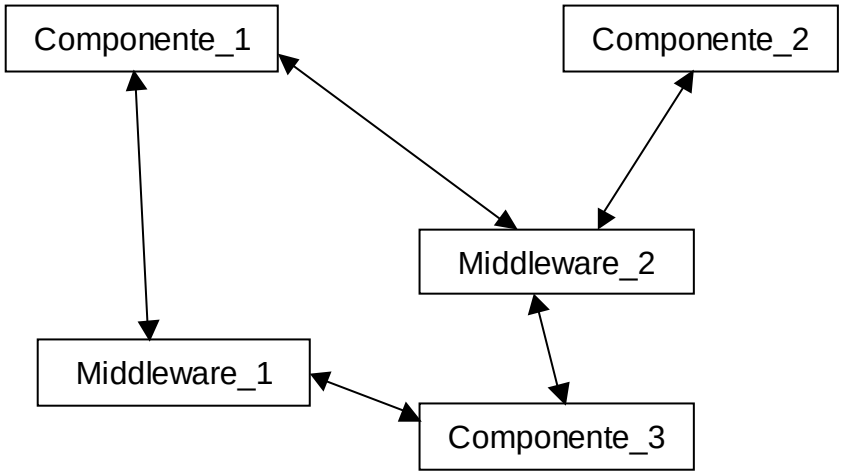


figura A.7

Come è possibile osservare i *middleware* potranno essere molteplici e con finalità differenti, sarà inoltre possibile definire *middleware* che ereditano da precedenti (cioè li incapsulano) e offrono funzionalità nuove; cosa utile soprattutto nelle comunicazioni 1-*, dove parte dei molti che partecipano alla comunicazione non dovranno essere modificati in caso di cambiamenti, mentre basterà modificare il codice del componente che partecipa con molteplicità 1 e delle classi coinvolte nelle modifiche tra quelle che partecipano con molteplicità *. Parleremo inoltre di comunicazioni tra componenti *master-master* o componenti *master-slave,* il primo caso riguarda le comunicazioni tra componenti che hanno entrambi vita propria e che recuperano alcune delle loro funzionalità da altri tramite *middleware*, mentre il secondo fa riferimento al caso in cui esistano componenti che offrono solamente funzionalità e non hanno vita propria.

Consideriamo dunque un modello di comunicazione 1-*, come quello che concerne il sistema realizzato nel contesto della creazione delle commissioni di laurea. Possiamo parlare di un sistema a comunicazione bidirezionale 1-*, nel quale esiste un componente *master* e molti *slave:* cioè quelli che offrono implementazioni diverse dell’algoritmo.

Il diagramma UML dei componenti sarà il seguente:

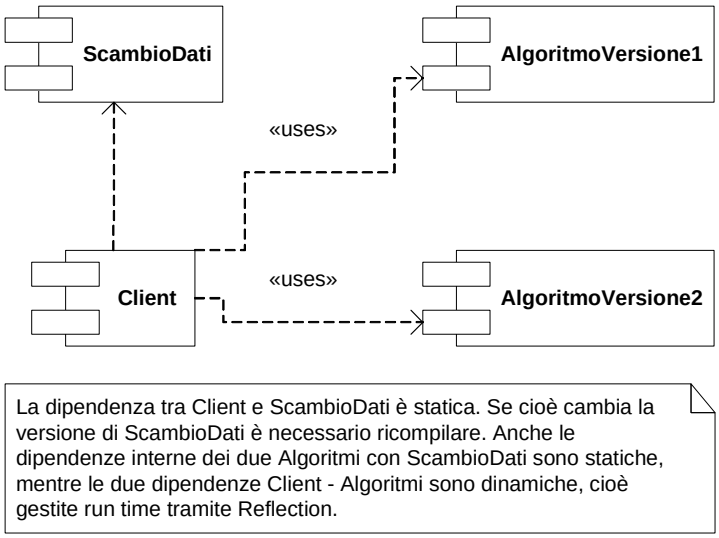


figura A.8

I diagrammi UML delle classi dei componenti sono invece i seguenti:

diagramma delle classi per ScambioDati

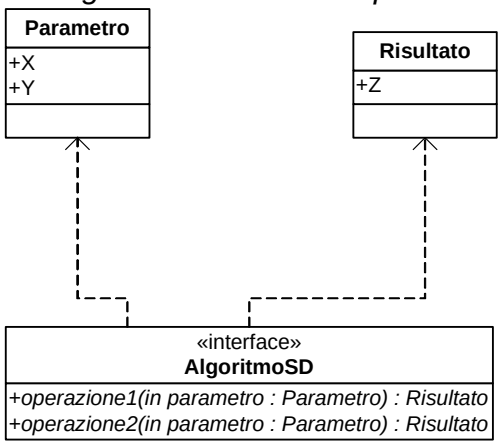


figura A.9

diagramma delle classi per AlgoritmoVersione1

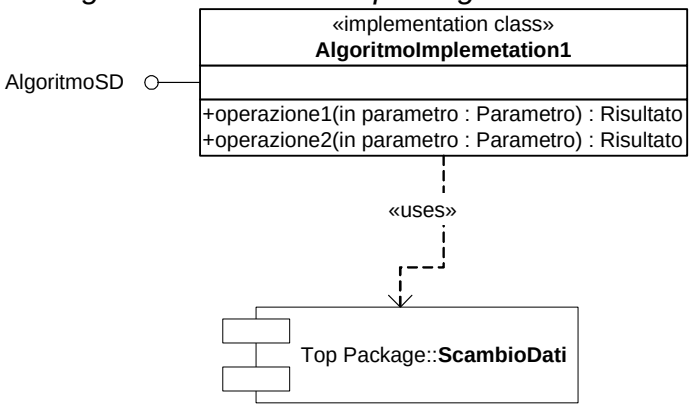


figura A.10

diagramma delle classi per AlgoritmoVersione2

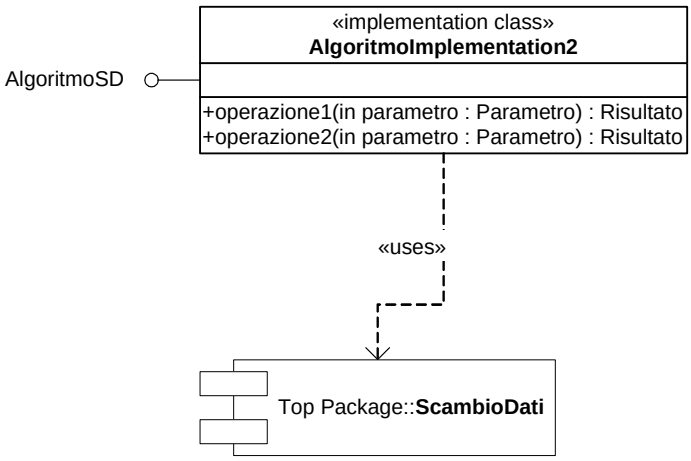


figura A.11

diagramma delle classi per il Client (cioè il componente master)

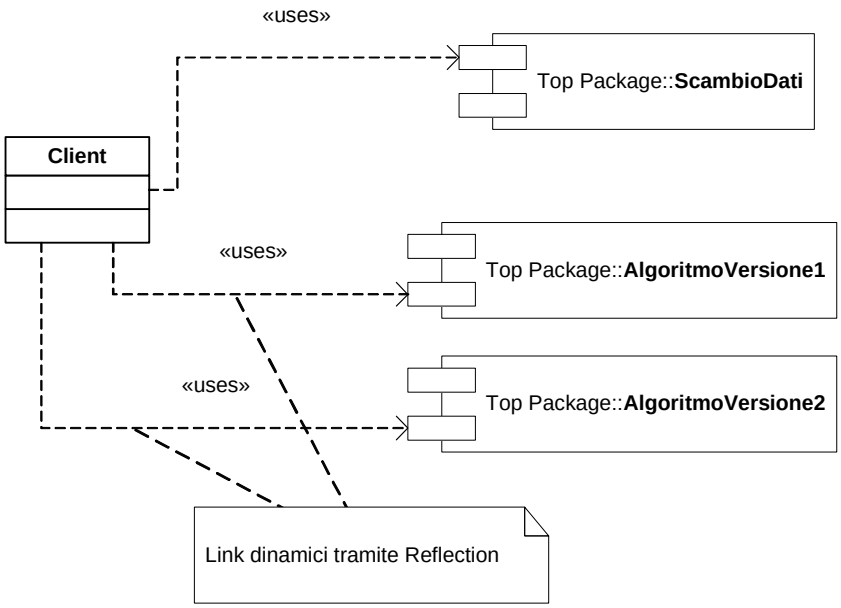


figura A.12

Osserviamo la nota nell'ultima immagine: essa ci indica i link operati r*un-time* tramite *reflection*, un'importante funzionalità messa a disposizione da C#. Una volta fissata una versione per il componente ScambioDati, sarà possibile sviluppare il Client e gli Algoritmi in modo assolutamente separato e infine linkarli indicando al componente master, cioè il Client, in quali DLL andare a cercare le implementazioni dell'interfaccia ScambioDati::AlgoritmoSD. Tali implementazioni potranno poi essere istanziate dinamicamente sotto forma di oggetti *controllabili* attraverso l'interfaccia condivisa presente nel componente ScambioDati.

Indichiamo ora i passi fondamentali che il Client deve eseguire per operare il collegamento dinamico.

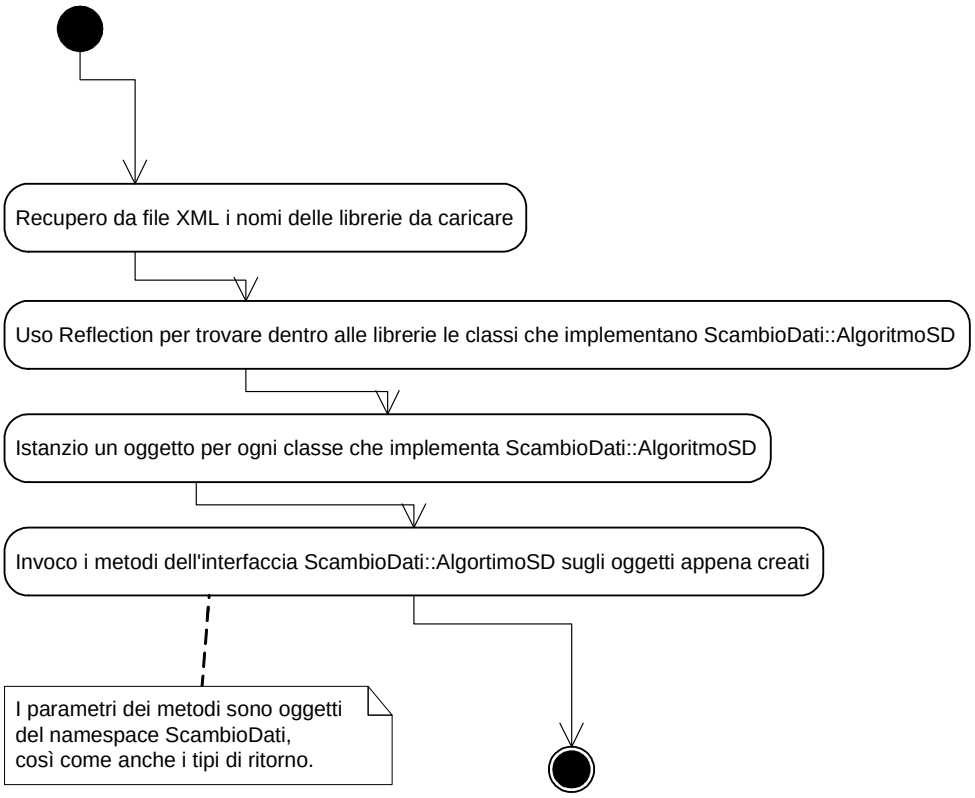


figura A.13

Dopo aver presentato i diagrammi statici e dinamici, verrà ora illustrata la sequenza dei passi che caratterizzano la soluzione progettuale. Mentre prima questi passi sono stati presentati in modo astratto, ora si procederà ad una esemplificazione pratica.

Abbiamo già mostrato il problema e la soluzione da adottare. Se supponiamo che a risolverlo sarà impegnato un *team* di 2 persone, il primo passo da compiere consisterà nel suddividere le sfere di influenza. Assumiamo dunque che la prima persona (A) si occupi dell'implementazione dell'interfaccia e della struttura dati, gestendo anche la persistenza; mentre la seconda (B) si dovrà curare di implementare l'algoritmo che opererà sui dati e che sarà richiamato dal flusso principale (implementato da A insieme all'interfaccia). Dopo una prima analisi individuale a livello concettuale, al successivo incontro A e B confronteranno le realtà da essi modellate; da tale confronto dovrà scaturire un modello concettuale del componente ScambioDati, che come abbiamo già mostrato, dovrà modellare tutti i dati e le funzionalità condivise: nel nostro caso, un'interfaccia AlgoritmoSD contenente dichiarazioni di metodi per le funzionalità che dovranno essere implementate dai componenti slave, un oggetto Parametro, che costituirà l'argomento di invocazione di un implementazione dei metodi di AlgoritmoSD, e un oggetto Risultato, appunto il risultato restituito ad ogni invocazione di tali metodi. In seguito uno dei due, o anche entrambi, dovranno implementare questo componente, che sarà rilasciato in una versione definitiva. Dopo questi primi passi, i successivi saranno completamente liberi: A potrà sviluppare tutte le sue classi basandosi su ScambioDati, come potrà fare anche B. Se il modello sarà master-slave, il master dovrà curarsi di gestire il caricamento run-time del componente sviluppato da B, altrimenti entrambi

dovranno gestire i reciproci caricamenti tramite *Reflection.* Infine il collegamento sarà immediato, quando entrambi avranno concluso lo sviluppo, in quanto consisterà solo nell'aggiornamento delle liste degli assembly che ciascun master dovrà caricare. Potrà poi succedere che ad un certo punto un terzo programmatore C si inserisca per implementare una diversa soluzione per l'algoritmo, creando quindi un nuovo componente *slave.* Avendo in mano ScambioDati, egli potrà implementare la nuova soluzione senza sapere nulla riguardo a quella già esistente o a come i dati siano trattati dagli altri componenti, anzi il grande vantaggio che egli trarrà da questa soluzione progettuale sarà quello di avere già sottomano il modello della realtà (cioè le classi in ScambioDati) su cui il suo nuovo algoritmo dovrà lavorare.

Come nota finale facciamo notare che sotto il nome di *componente* indichiamo in questa sede gli *Assembly* creati da .NET, ricordando che un assembly può comparire sotto le forme più disparate (exe, dll etc.) in quanto non è altro che un *IL*[19] del codice compilato.

[19] *Intermediate Language:* codice precompilato che viene poi interpretato dal framework .NET.

Bibliografia

- Vojin G. Oklobdzija – The Computer Engineering Handbook – CRC Press
- M. Fowler – UML Distilled – Addison-Wesley
- S. Martello – Lezioni di Ricerca Operativa – Progetto Leonardo
- A. Lodi, S. Martello, D. Vigo – Recent Advances on Two-Dimensional Bin Packing Problems – Technical Report DEIS
- A. Lodi, S. Martello, D. Vigo – Approximation algorithms for the oriented two-dimensional bin packing problems – European Journal of Operation Research
- S. Martello and P. Toth – Knapsack Problems: Algorithms and Computer implementation – J. Wiley & Sons
- A. Lodi, S. Martello, M. Monaci – Two-Dimensional Packing Problems: A Survey – Technical Report DEIS.
- Eric Gunnerson – A Programmer's Introduction to C# – Apress
- A. Turtschi, J Werry, G. Hack, J Albahari – C# .NET Web Developers Guide – Syngress
- A. Hejlsberg and S. Wiltamuth – C# language Reference – Microsoft Press
- T. Thai and H. Lain – Introducing .NET Framework – O'Reilly
- E. Gamma, R. Helm, R. Johnson, J Vlissides – Design Patterns: Elements of Reusable Object-Oriented Software – Addison-Wesley
- Brendan McCarthy – The Cascading Bridge Design Pattern – Sun Java Center
- ISO 9126 Software Product Evaluation - Quality Characteristics and Guidelines for Their Use, 1991
- Dorfman, M., and R. H. Thayer, *Software Engineering.* IEEE Computer Society Press, 1997.
- Beizer. Boris, *Software Testing Techniques,* International Thomson Press, 1990.

Indice delle figure

Sommario