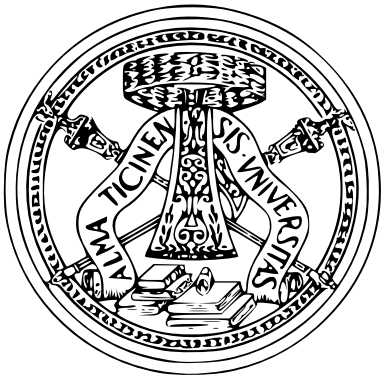

Università degli Studi di Pavia
Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Progetto di moduli per lo sviluppo di applicativi client-server basati su Shadow Framework

Relatore:
Prof. Alessandro Martinelli

Tesi di laurea di:
Luigi Pasotti

Anno Accademico 2011/2012

Indice

Elenco delle figure

Acronimi

SF	Shadow Framework
GPU	Graphic Processing Unit
UML	Unified Modeling Language
fps	frame per secondo
API	Application Programming Interface
GPGPU	General-Purpose computing on Graphics Processing Units
kB	kilobyte
MB	megabyte

Capitolo 1

Introduzione

La presente tesi si colloca nell'ambito dello sviluppo dello Shadow Framework (SF) 2.0, un framework concepito per la realizzazione di applicazioni che fanno uso di grafica tridimensionale real-time ideato e sviluppato dall'Ingegner Alessandro Martinelli.

La grafica tridimensionale, o computer grafica 3D, consiste nell'utilizzo di modelli geometrici tridimensionali da parte di un computer per il calcolo e il rendering[1] di immagini digitali.

Essa è attualmente diffusa in molteplici campi, cosa che la rende un'esperienza comune nella vita di tutti giorni. Non solo infatti viene utilizzata in modo intensivo nella pubblicità e nell'intrattenimento ma anche in settori come quello medico e nella ricerca scientifica.

La grafica tridimensionale real-time è un ramo specifico della grafica tridimensionale che si focalizza sulla generazione di simulazioni di ambienti e/o oggetti con la quale un utente può interagire osservando una reazione coerente nella simulazione. Questa percezione di interazione viene fornita da una generazione sequenziale di immagini che come nelle pellicole cinematografiche danno un'illusione di movimento, ma in cui l'effettivo contenuto dei fotogrammi non è predeterminato ed è calcolato al momento sulla base degli input forniti.

In base a quanto esplicato in [3], per poter avere un'interazione soddisfacente è molto importante che la velocità con cui vengono visualizzate le immagini, misurata in frame per secondo (fps), si mantenga stabile ed il più

[1] Con rendering, in computer grafica, si intende il processo che attraverso l'elaborazione di un modello determina il colore di ogni pixel contenuto in una immagine digitale[16, 17].

possibile elevata in modo da minimizzare i tempi di risposta ed impedire che questi interferiscano con l'interazione stessa. Ciò pone il problema di ottenere un alto valore di fps che a sua volta implica dei vincoli temporali sulla possibile elaborazione dei modelli tridimensionali: se per esempio si volesse mantenere una velocità di visualizzazione pari a 60 fps, il tempo di calcolo a disposizione per l'elaborazione di un frame rispetto al precedente ammonterebbe a circa 15 millisecondi. Questa caratteristica differenzia profondamente la grafica real-time da quella non-real-time in cui, non avendo vincoli temporali stretti per la generazione dei frame, il focus è spostato sull'applicazione di modelli di elaborazione complessi e computazionalmente onerosi che siano in grado di generare immagini il più fotorealistiche possibile.

Le problematiche della grafica real-time hanno imposto nel corso degli anni lo sviluppo di tecniche e tecnologie specifiche di settore il cui stato dell'arte è oggi rappresentato dall'ultima generazione di Application Programming Interface (API) di programmazione grafica, dalle Graphic Processing Unit (GPU)[2] a pipeline programmabile e dai Linguaggi di Shading.

Queste tecnologie hanno assunto una grande importanza perché consentono di raggiungere elevatissimi livelli di qualità e prestazioni quando sono utilizzate per la generazione di grafica 3D real-time e la ricerca su di esse ha portato allo sviluppo del General-Purpose computing on Graphics Processing Units (GPGPU) ovvero la possibilità di utilizzare la capacità di calcolo delle GPU per processare anche dati differenti da quelli grafici.

Le GPU a pipeline programmabile, in contrapposizione a quelle con pipeline fissa, permettono di adattare gli stadi della pipeline di renderizzazione mediante l'utilizzo dei linguaggi di Shading. In sostanza l'hardware viene programmato per il calcolo di algoritmi specifici da applicare ai dati grafici. Ciò consente di adattare il processo di renderizzazione agli effetti che si desidera ottenere e di sfruttare l'hardware della GPU per velocizzarne la computazione. Questo paradigma si è rivelato molto efficace ed efficiente, tanto da venire oggi applicato nella quasi totalità delle GPU moderne, sia che si tratti di dispositivi di fascia alta che di processori grafici dedicati ad architetture mobile come i cellulari.

Parallelamente le API di programmazione grafica moderne si sono sviluppate consentendo lo sfruttamento sempre più efficiente delle risorse hardware

[2]Una GPU è un microprocessore dedicato alla generazione delle immagini visualizzate sullo schermo di un dispositivo, alleggerendo da questo carico il processore principale.

delle GPU, ma non solo: la loro integrazione su piattaforme software pensate per il mercato embedded ha consentito una proliferazione di applicazioni che fanno uso di grafica tridimensionale sia su cellulari che tablet. Non è difficile trovare per questi dispositivi, dotati ormai quasi obbligatoriamente di fotocamera, applicazioni di realtà aumentata che sfruttino le API per inserire elementi grafici tridimensionali nelle immagini catturate dal sensore ottico. Un evento che nei prossimi anni porterà un impatto significativo sul mercato, consiste nel fatto che molti sviluppatori di browser per la navigazione di internet stanno attualmente lavorando per integrare l'API grafica WebGL all'interno dei loro software tramite JavaScript. Questo consentirà l'esecuzione di applicazioni 3D direttamente all'interno dei browser stessi, eliminando la transizione tra contenuti 3D e contenuti non-3D e consentendo una integrazione diretta con servizi internet di terze parti.

Di pari passo, i produttori di middleware, engine e framework[3], hanno integrato nelle funzionalità dei loro prodotti la capacità di sfruttare le pipeline programmabili e tool per testare e comporre nuovi shader[4]. Una sempre maggiore quantità di queste case produttrici supportano le piattaforme mobile (come Android e iOS) e rilasciano plugin o applicazioni "WebPlayer" per distribuire contenuti attraverso i browser web. In alcuni casi, come quello citato in [11], produttori di browser e case di sviluppo collaborano per migliorare le prestazioni di WebGL e convertire in JavaScript gli engine e i framework così da poterli eseguire direttamente all'interno delle pagine web.

In questo contesto si inserisce lo Shadow Framework, il quale è stato progettato non solo per utilizzare e supportare tutte le tecnologie che costituiscono lo stato dell'arte nel campo della grafica 3D real-time, ma anche con l'obbiettivo di farlo sperimentando un nuovo approccio nella generazione e nella gestione dei dati grafici tridimensionali. La chiave di questo approccio consiste nell'abbandonare la vecchia concezione per cui gli oggetti tridimensionali sono scolpiti generando mesh di vertici che compongono triangoli, ma utilizzando primitive parametriche più complesse per poi sfruttare le capacità delle pipeline programmabili e far generare dall'hardware stesso le

[3]Nel contesto delle applicazioni per la grafica tridimensionale con middleware di solito si intendono componenti software dedicate a compiti specifici, come la gestione della fisica o il pathfinding, che vengono affiancate agli engine ed ai framework. Per una descrizione più completa di engine e framework e le differenze fra loro, fare riferimento alla sezione 2.1

[4]Uno shader è un programma scritto con un linguaggio di shading, che viene caricato ed eseguito in hardware da una GPU.

mesh di punti necessarie per il processo di renderizzazione. Questo tipo di procedimento permette di adattare la qualità del risultato visivo finale in base alle prestazioni dell'hardware a disposizione, eliminando la necessità di produrre diverse versioni dello stesso modello la cui unica differenza consiste nel numero di vertici. È infatti sufficiente programmare la pipeline per produrre un minor numero di vertici per alleggerire il calcolo, mantenendo inalterato il modello parametrico di partenza.

Oltre ai vantaggi descritti, l'utilizzo di primitive parametriche permette di produrre file di dimensione molto contenuta rispetto ai sistemi classici in cui i file contengono l'elenco dei vertici che descrivono l'oggetto. Lo stesso paradigma viene usato all'interno del framework non solo per i modelli 3D, ma anche per la generazione delle texture da applicarvi, garantendo un meccanismo per scalarne la qualità in base alle esigenze.

Le caratteristiche principali del framework, la cui spiegazione più dettagliata verrà esposta nel capitolo 2, sono le seguenti:

- la possibilità di definire nuove primitive grafiche con un linguaggio specifico del framework;
- la grande portabilità: la versione di riferimento è sviluppata in linguaggio Java, ma è in corso di realizzazione un porting in C++ (iOS);
- il design web-oriented: i meccanismi interni sono stati progettati per supportare l'utilizzo di dati in remoto ed è in corso di realizzazione una versione del framework in JavaScript;

1.1 Obbiettivo del progetto

L'obbiettivo del progetto di tesi nasce dall'idea di produrre un'applicazione dimostrativa delle funzionalità di rete offerte dallo Shadow Framework. Ciò che si voleva ottenere era una coppia client-server in cui il server fosse in grado di gestire connessioni simultanee da parte di un numero indefinito di client. Ogni client, ottenuta una connessione con il server, doveva poter visualizzare una scena iniziale navigabile, richiedendo solamente i dati relativi all'ambiente in prossimità della posizione, all'interno della scena, dell'utente.

Successivamente si volevano analizzare due possibili approcci: il primo in cui il client richiede al server i dati aggiuntivi riguardo la scena a seconda delle necessità, ad esempio una volta raggiunti i bordi dell'ambiente. Nel

secondo il server tiene traccia degli spostamenti nella navigazione comunicando attivamente con il client, componendo in modo dinamico dei pacchetti di dati che prevedano le future richieste del client.

Le astrazioni del layer dati del framework sono state progettate specificatamente per consentire lo sfruttamento della comunicazione di rete, ma fino a quel momento non era stata fatta alcuna specifica implementazione che la utilizzasse. Si desiderava perciò produrre questo tipo di applicazione non solo a scopo dimostrativo, ma anche per individuare e correggere probabili bug presenti nel codice e dovuti a vincoli di sincronizzazione non evidenziati dai test effettuati con dati memorizzati su macchina locale.

L'obbiettivo della tesi è così diventato quello di produrre dei moduli di libreria che fossero utili allo sviluppo di applicazioni client-server.

La progettazione di questi moduli si è focalizzata su alcuni punti cardine che rappresentano la chiave dello sviluppo del progetto. La struttura del framework è stata ideata con l'obbiettivo di garantirne l'estendibilità, perciò è stato di fondamentale importanza sfruttare i meccanismi e le astrazioni previste in tal senso e lo sviluppo dei moduli è stato guidato dagli stessi principi.

1.2 Organizzazione del documento

Il presente documento è organizzato secondo la seguente suddivisione in capitoli:

- **Capitolo 2:** in cui viene presentata una panoramica generale sullo Shadow Framework 2.0 in rapporto al panorama generale sui framework di programmazione di grafica tridimensionale real-time.

- **Capitolo 3:** in cui è descritta l'astrazione di gestione dei dati interna al framework e come essa viene utilizzata dalle applicazioni SF.

- **Capitolo 4:** in cui viene descritto il progetto Sf-Remote-Connection, i moduli che lo compongono, le funzionalità offerte e i package java prodotti.

- **Capitolo 5:** in cui sono presentate le applicazioni di test ed i risultati prodotti.

- **Capitolo 6:** in cui viene presentato un riassunto del lavoro svolto, i risultati e vengono proposti alcuni sviluppi futuri.

Capitolo 2

Lo Shadow Framework 2.0

Questo capitolo presenta inizialmente una panoramica sugli attuali sistemi per la realizzazione di applicazioni che fanno uso di grafica tridimensionale real-time. Successivamente viene presentato lo ShadowFramework 2.0, le sue caratteristiche, la sua struttura e i suoi sviluppi futuri.

2.1 Panoramica sui framework per la grafica 3D interattiva

Lo scopo della presente tesi di laurea è quello di produrre dei moduli di estensione per lo Shadow Framework, utili alla realizzazione di applicativi 3D orientati al web. Per lo stesso scopo esistono sul mercato una grande quantità di soluzioni le cui caratteristiche possono essere notevolmente differenti in base al target di sviluppo a cui si rivolgono.

Prima di fornire una descrizione di alcune di queste soluzioni è bene introdurre alcuni concetti utili a comprendere quali sono i punti focali che distinguono un prodotto da un altro. L'elemento fondamentale di ognuno di questi questi software è l'**engine**, altrimenti detto rendering engine o 3D-engine. Questo è il nucleo di ogni programma grafico e fornisce una serie di meccanismi che, sfruttando le API di programmazione grafica, permettono di disegnare su schermo i modelli tridimensionali voluti. Quando un prodotto consiste nel solo engine viene fornito nella forma di una libreria con cui è possibile interagire attraverso una API specifica e che permette di configurare i passaggi della fase di rendering.

Dato che la più grande fetta di mercato per queste soluzioni software proviene dallo sviluppo di videogames sono nati una serie di software definiti **game engine**, veri e propri framework per la produzione di applicazioni 3D che, oltre al rendering engine, integrano al loro interno moduli per la gestione della fisica, del suono, delle animazioni e di tutte quelle componenti utili per lo sviluppo di giochi per computer. Questi framework possono essere forniti sia sotto forma di libreria da integrare nel proprio codice sia con tool grafici che permettono di creare la propria applicazione da zero.

Seguendo questa classificazione e dato l'obiettivo dello Shadow Framework di fornire un set di librerie completo per la creazione di applicazioni, con moduli per gestire animazioni e dati senza vincolare all'utilizzo di librerie specifiche per gli elementi non legati strettamente alla gestione grafica, esso si trova a metà strada tra un rendering engine puro e un game engine.

Vengono ora presentati alcuni dei prodotti presenti sul mercato, la scelta di quali presentare rispetto ad altri non riflette la loro qualità, ma è stata fatta su base rappresentativa.

2.1.1 Game engine per Adobe Flash

Per l'ambiente di Adobe esistono una grande quantità di game engine che sfruttano le API Flash per effettuare il rendering dei contenuti. Sebbene questo framework venga utilizzato da molti anni per la realizzazione di applicativi e che per lungo tempo abbia rappresentato l'ambiente di riferimento per applicativi web 3D, solo dalla release 11 comincia a sfruttare l'accelerazione hardware per la grafica tridimensionale.

Attualmente l'accelerazione consente di sfruttare le GPU a pipeline programmabile, ma non consente di usare le modalità legacy a pipeline fissa. Inoltre non vengono rese disponibili le API grafiche per la programmazione nativa come OpenGL o DirectX, ma una API proprietaria di nome Stage3D che fornisce un'astrazione di livello più alto e facilita la gestione delle risorse. Questo sebbene consenta di semplificare e unificare tutte le piattaforme compatibili rende necessarie delle limitazioni per conservare la compatibilità, rendendo inaccessibili le funzionalità più avanzate degli hardware moderni (ad esempio è supportato lo Shader Model solo fino alla versione 2.0 sebbene oggi sia disponibile la versione 5.0)[2].

2.1.2 Unity 3D

Unity 3D è un game engine completo che dal 2005 ad oggi ha accresciuto molto la sua popolarità. Uno dei maggiori punti di forza di questo ambiente consiste nella presenza di una discreta quantità di tool grafici integrati tra loro per il controllo del workflow di sviluppo. Sono inoltre presenti molti moduli per controllare aspetti quali le animazioni, la verifica delle performance ed il networking. Una qualità di primo piano è la compatibilità con un elevato numero di piattaforme, consentendo di pubblicare le applicazioni realizzate su piattaforme mobile (android e iOS), desktop (Windows, Mac e Linux), console e web (via browser). Riguardo quest'ultimo caso la fruizione dei contenuti via web è effettuata con due diversi metodi: o tramite un player flash o tramite l'API Google Native Client del browser Chrome che permette l'esecuzione di codice nativo all'interno del browser. In generale Unity 3D è un game engine moderno con pieno supporto alle ultime tecnologie grafiche rendendolo un prodotto molto rappresentativo. Oltre alle caratteristiche citate buona parte del suo successo è dovuto alla grande comunità di sviluppatori, anche indipendenti, ed alla numerosa quantità di asset grafici disponibili.

2.1.3 Unreal Engine e Unigine engine

Entrambi questi prodotti sono game engine completi piuttosto famosi per l'elevatissima qualità delle loro capacità di rendering e per questo vengono spesso utilizzati come benchmark per testare le capacità grafiche hardware. Questi progetti sono maturi e moderni: sono dotati di tool grafici specifici, supportano le più recenti architetture grafiche e la maggior parte delle piattaforme desktop, mobile e console. Sebbene l'Unreal engine supporti flash a differenza di Unigine, sono presentati insieme in quanto esempio di nuovo approccio alla distribuzione via web: entrambe le compagnie produttrici di questi software stanno infatti investendo nel portare i loro framework a supportare API WebGL. Gli annunci ufficiali possono essere trovati a questi riferimenti [11, 4].

2.1.4 OnLive e Gaikai

Questi prodotti non sono software per produrre applicazioni di grafica, ma due servizi a pagamento per la fruizione di applicazioni e videogiochi attra-

verso lo streaming online e vengono qui citati come esempio di una tecnologia alternativa per la fruizione di contenuti basati su grafica 3D real-time.

Entrambe le soluzioni eseguono su richiesta le applicazioni su server proprietari e all'utente viene fornito di un client, hardware o software, che attraverso una connessione ad internet permette di ricevere l'output video del server mediante un flusso in live streaming. Il client si occupa inoltre di intercettare l'input dell'utente e inviarlo ai server che stanno eseguendo l'applicazione in modo che possano processarlo per rendere interattiva la fruizione del servizio.

L'utilizzo di questo approccio consente all'utente di non doversi dotare di dispositivi hardware dall'elevata potenza di calcolo dato che il peso della computazione della grafica è totalmente a carico dei server. In questo caso il punto focale che influenza la qualità dell'interazione consiste nelle prestazioni della connessione di rete, che deve essere continuata, con una elevata larghezza di banda e una bassa latenza. Informazioni relative a questi servizi sono reperibili tramite [12, 8].

2.2 Introduzione allo Shadow Framework 2.0

Le tecnologie della pipeline di rendering programmabile e delle API 3D, tra cui lo standard più diffuso è OpenGL, hanno avuto una notevole influenza sull'industria della produzione di applicazioni di grafica tridimensionale, ma un impatto altrettanto significativo si è avuto nel campo della ricerca in cui, grazie ad esse, si è raggiunto lo sviluppo di numerose soluzioni innovative e la generazione di effetti che, senza l'ausilio delle suddette tecnologie, erano impossibili da implementare. Nonostante la bontà di molte soluzioni individuate, solamente una parte di esse ha effettivamente trovato un'applicazione nel mondo dell'industria di settore e anche in quel caso il loro utilizzo è stato indirizzato ad un miglioramento delle tecniche e delle consuetudini definite negli anni piuttosto che ad un rinnovamento radicale.

L'obbiettivo fondamentale dichiarato per lo Shadow Framework è proprio quello di ridefinire l'architettura classica delle applicazioni di grafica attraverso l'utilizzo dei nuovi paradigmi consentiti dalla nuova tecnologia: "Noi sosteniamo che cambiare si può. La tecnologia è matura quanto basta per sperimentare soluzioni alternative che siano più in linea con le tecnologie

moderne e che siano in grado di rispondere in modo migliore alle esigenze del Web 3D"[10].

Per meglio comprendere l'approccio innovativo sperimentato dal framework dobbiamo individuare dove e perché un'applicazione di questo tipo utilizza al suo interno i dati grafici tridimensionali e qual è l'impatto che ognuno di questi "momenti" ha sulla responsività del sistema.

2.2.1 I tre "momenti" di utilizzo dei dati

"Nei casi pratici ogni applicazione che fa uso della grafica tridimensionale passa per tre momenti molto importanti: **accesso** ai dati tridimensionali, la fase di **costruzione e inizializzazione** e la **visualizzazione**"[10].

Con **accesso** ai dati tridimensionali si intende il caricamento dei dati in memoria ram. In un'applicazione tradizionale questi dati sono memorizzati all'interno di file già presenti sulla memoria di massa locale, per questo motivo il loro caricamento non è mai considerato critico per quanto riguarda le prestazioni e la responsività, nonostante la dimensione di questi dati spesso raggiunge le decine di giga byte.

La fase di **costruzione e inizializzazione** consiste in quel processo in cui i dati caricati da file devono essere usati per costruire nella memoria grafica le strutture dati effettivamente utilizzabili dalle GPU, non è raro che per esaudire questo compito le informazioni abbiano la necessità di essere trasformate o decompresse per essere adattate alle suddette strutture. In alcuni casi, specialmente per dati di grosse dimensioni, questa inizializzazione viene svolta una sola volta all'inizio e prima di cominciare la visualizzazione.

La **visualizzazione** è quella fase in cui i dati grafici, caricati e inizializzati nella memoria grafica vengono processati per produrre i fotogrammi della sequenza che si desidera produrre.

Storicamente, e non senza motivo, il processo di visualizzazione è quello su cui si sono concentrati i maggiori sforzi sia per quanto riguarda la ricerca di una maggiore qualità visiva, sia per quanto concerne la reattività delle applicazioni. Queste due componenti sono parte di un complesso equilibrio che cerca di bilanciare il tempo di calcolo del processo di rendering e i tempi stretti concessi alla produzione di un singolo frame per garantire una fruizione fluida e interattiva. Va fatto notare che nel tempo che intercorre tra un frame e l'altro non deve essere eseguito il solo processo di rendering, ma

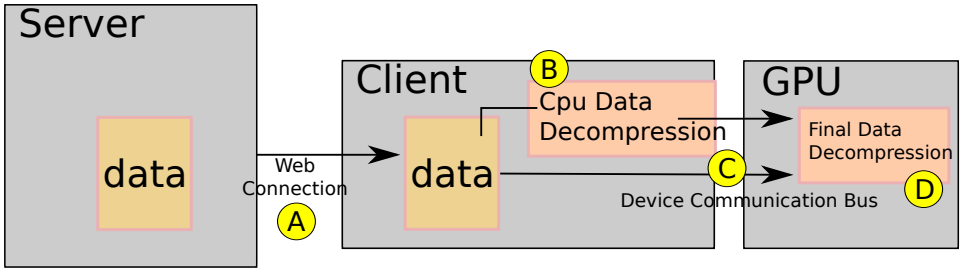

Figura 2.1: Schema degli agenti coinvolti nella distribuzione dei dati tridimensionali via rete. Sono posti in evidenza: A)la connessione di rete attraverso cui client e server comunicano; B)la "decompressione" effettuata, se necessario, dalla cpu del client; C)il bus di comunicazione tra la memoria principale e la memoria della GPU; D)l'eventuale e ulteriore "decompressione" effettuata dalla GPU stessa.

deve anche essere applicato l'input dell'utente e in alcuni casi applicata una simulazione fisica e l'intelligenza artificiale di agenti interni alla simulazione.

2.2.2 Applicazioni orientate al Web 3D

Con la diffusione di applicazioni web 3D questa prospettiva è destinata a cambiare dato che per i sistemi tradizionali l'utilizzo di una grande quantità di dati rende non più trascurabili i tempi di reperimento e caricamento. Per ovviare a questo problema le soluzioni più diffuse sono una, a volte drastica, diminuzione della qualità visiva o il rilascio di client che necessitano di installazione da parte dell'utente e che memorizzano in locale tutti i dati grafici. Quest'ultimo metodo è probabilmente il più diffuso per tutte quelle applicazioni che puntano ad avere sia elevata qualità visiva che interazione tra utenti remoti, ma è anche quello che si separa di più dalle caratteristiche del web. L'approccio del framework è invece quello di affrontare il problema aumentando la criticità delle fasi di **accesso**, **costruzione e inizializzazione**.

Se si analizzano le applicazioni di grafica distribuite sulla rete, si può risalire a tre moduli distinti: il server, il client e l'acceleratore grafico a disposizione del client. I dati contenuti sul server, per essere utilizzati, devono arrivare all'acceleratore grafico nella forma delle strutture dati proprie di quest'ultimo. Escludendo l'approccio a client con dati grafici autonomi possiamo modellizzare questo scenario come mostrato nella figura 2.1, in essa

possiamo vedere in ordine temporale da sinistra a destra, il percorso fisico attraversato dai dati. Questi partono dal server e viaggiano attraverso una connessione web (A) fino a raggiungere il client, qui vengono caricati in memoria ed eventualmente "decompressi" dalla cpu (B), successivamente vengono inviati alla memoria video attraverso il bus di connessione della scheda grafica (C) ed infine, prima di essere visualizzati, possono dover attraversare un ulteriore processo di elaborazione/decompressione da parte della GPU stessa (D). È facile identificare nella connessione (A) il collo di bottiglia del sistema, dato che in esso la banda a disposizione per il trasferimento dati è in assoluto la più piccola e non consente di sfruttare pienamente le capacità di calcolo della GPU.

Le applicazioni tradizionali si limitano ad utilizzare strutture dati datate, solitamente basate sulle mesh di vertici[1], che per le loro caratteristiche producono file di grosse dimensioni, più che sufficienti nel caso in cui i dati siano memorizzati in locale, ma poco adeguati ad un contesto di comunicazione remota. I file vengono trasferiti attraverso il canale (A), il client se necessario effettua in (B) una decompressione dei dati e carica in memoria le informazioni contenute, nello stadio (C) i dati vengono trasferiti nella memoria video all'interno di strutture dati adatte a seconda del caso. Una volta all'interno della memoria della GPU spesso i dati non vengono ulteriormente processati prima della visualizzazione, ma tutti gli effetti aggiuntivi vengono inseriti durante il rendering stesso.

Dato il collo di bottiglia della connessione web (A) è proprio la dimensione dei file a rappresentare il maggiore limite del sistema. Per questo motivo è consuetudine che anche le applicazioni che non usano client con dati grafici preinstallati dedichino una fase iniziale ad effettuare il download completo di tutti i dati.

Le applicazioni basate sul modello del live streaming hanno problemi simili: in questo caso si ha il trasferimento attraverso (A) dei dati del flusso video, la banda occupata da questo flusso per livelli di qualità medio-alta può variare da pochi Mbit/s alle decine di Mbit/s. Le informazioni video hanno bisogno di un solo stadio di decompressione che può essere effettuato sia direttamente dalla cpu in (B) che all'interno di processori grafici che

[1]Le mesh di vertici, o mesh poligonali, rappresentano i modelli tridimensionali attraverso un elenco di punti che definiscono i vertici di figure poligonali, di solito triangoli, in uno spazio tridimensionale.

implementano i decoder specifici, ma la GPU vera e propria non viene affatto utilizzata. Come già accennato il vantaggio consiste nel fatto che il client non necessita di avere a disposizione una GPU e quindi di memorizzare i dati grafici. Tuttavia è richiesta una connessione costante anche quando vengono visualizzate immagini fisse e la qualità non viene influenzata dall'hardware a disposizione, ma solo dalla banda a disposizione.

In entrambe le situazioni analizzate le possibilità offerte dallo stadio (D) non vengono sfruttate ed è proprio tramite il suo utilizzo che lo Shadow Framework cerca di risolvere il problema. Facendo riferimento alla figura 2.1, sul server sono memorizzati i dati modellizzati in una forma parametrica adatta ad essere processata dalla GPU: si pensi ad esempio ad una sfera descritta dalla posizione del proprio centro e dal suo raggio, utilizzando l'equazione parametrica della sfera è possibile calcolare dinamicamente in un secondo tempo la mesh di vertici che approssima il modello. Questa forma di "decompressione" dei dati consente di alleggerire i modelli contenuti nei file dal compito di trasportare al loro interno tutte le informazioni in maniera esplicita, similmente a quanto avviene ai formati vettoriali per le immagini bidimensionali. I file possono perciò essere di dimensioni contenute, diminuendo sensibilmente il tempo di **accesso** ai dati e scaricando parte del processo di reperimento delle informazioni anche sulla fase di **costruzione e inizializzazione** eseguita sulla GPU stessa. In figura 2.2 è possibile osservare il confronto tra il rendering di un oggetto memorizzato secondo il formato *Wavefront obj*, dove in un file testuale è elencata la mesh di vertici, e il medesimo oggetto memorizzato nel formato interno del framework. Nel primo caso l'occupazione di memoria del file obj supera i 30kilobyte (kB) mentre nel secondo la rappresentazione xml del formato SF si riduce a 3kB migliorando la qualità del rendering.

2.2.3 Funzionalità avanzate

Alla luce del principio esposto al precedente paragrafo si può giungere successivamente ad ulteriori rielaborazioni di diverse pratiche sfruttate attualmente dai framework e dalle applicazioni grafiche. Una di queste pratiche è l'utilizzo della **precomputazione**, ovvero il processo di pre-calcolare dati computazionalmente troppo onerosi per essere gestiti in real-time. Questa precomputazione può essere eseguita in fase di inizializzazione di un'appli-

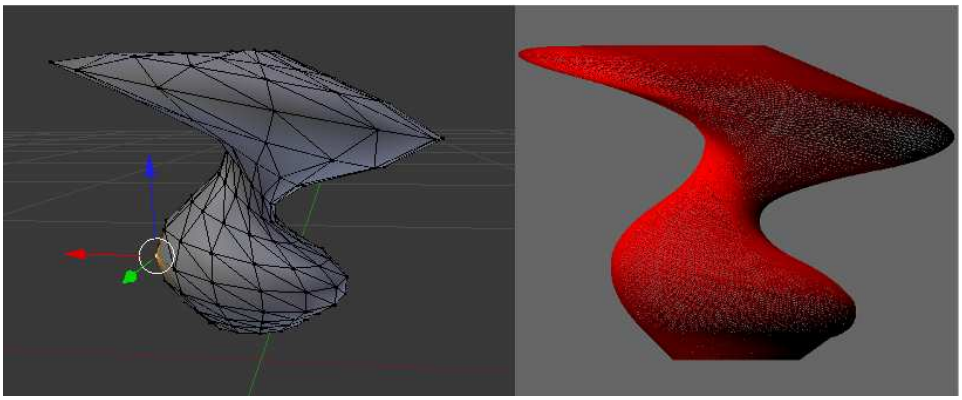

Figura 2.2: Confronto tra il rendering di un file .obj di 34kB (sinistra) e un file .sf in formato xml di 3kB (destra).

cazione oppure essere effettuata a monte e memorizzata in modo statico in file la cui dimensione può rivelarsi molto elevata.

In contesto web la seconda soluzione è la meno conveniente dato che aumenta la quantità di dati da trasferire attraverso il canale di comunicazione, ma alla luce dei moderni hardware grafici in generale anche la prima può essere ripensata sia nel senso di limitarla che nel senso di spostarne il peso computazionale direttamente sulla GPU.

Un'altra interessante considerazione riguarda la valutazione del livello di dettaglio: le applicazioni tradizionali per adattare la qualità delle immagini prodotte alla potenza dell'hardware a disposizione ricorrono all'espediente di fornire differenti varianti della stessa risorsa grafica a diversi livelli di qualità. Per il rendering della stessa scena sono perciò disponibili modelli tridimensionali con un diverso numero di vertici o varianti della stessa immagine a diverse risoluzioni. Questo consente di alleggerire il calcolo su GPU meno potenti, ma ciò provoca un aumento significativo della dimensione dei dati rendendo la soluzione insoddisfacente nel contesto web. L'utilizzo di una modellizzazione gerarchica all'interno del framework fornisce un meccanismo alternativo per gestire il livello di dettaglio che non va ad impattare sulla dimensione dei dati.

In figura 2.3 possiamo vedere come questa gerarchizzazione sia strutturata a due vie: sulla sinistra vediamo l'oggetto modellizzato ad alto livello, i file sf del framework che contengono i dati riferiti a questo livello di astrazione risultano molto compatti (dai pochi kB a qualche decina), ma comunque

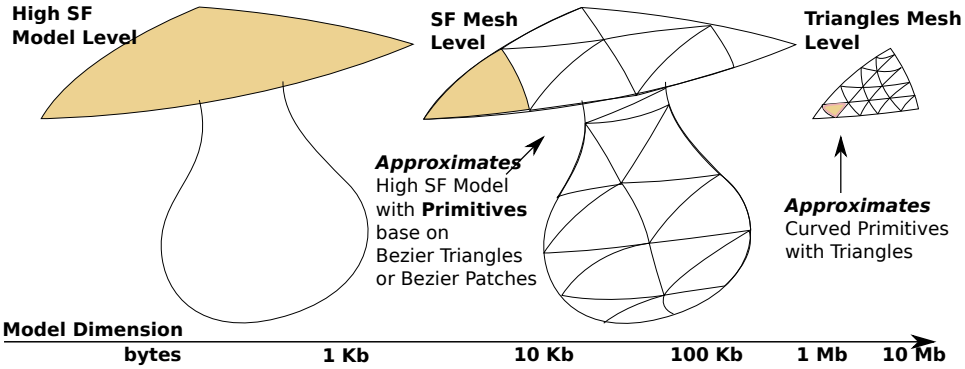

Figura 2.3: Grafico della modellizzazione gerarchica del framework.

ricchi di informazioni. La prima decompressione delle informazioni consiste nel tassellare[2] il modello parametrico con primitive di alto livello adatte allo scopo (in figura sono citati i triangoli di Bezier e le patch di Bezier). I dati estratti da questa prima fase possono raggiungere le centinaia di kB. La seconda fase consiste in una ulteriore tassellazione delle primitive di alto livello in triangoli veri e propri che approssimano le superfici curve delle precedenti primitive e che possono essere usati direttamente della GPU per effettuare il rendering. Questa operazione genera quantità di dati la cui dimensione può essere anche di decine di megabyte (MB), ma essendo eseguita direttamente all'interno della memoria grafica il trasferimento ne risulta alleggerito. Il valore aggiunto di questa gerarchizzazione è che sfruttando su più livelli i processi di tassellazione ha intrinsecamente al suo interno un modo per scalare il livello di dettaglio modificando la granularità della tassellazione stessa, evitando la duplicazione dei dati.

2.2.4 Considerazioni finali sul framework

La naturale premessa, essenziale per supportare le valutazioni espresse nei precedenti paragrafi, corrisponde alla necessità da parte del framework di utilizzare un set di strutture geometriche intelligenti adatte allo scopo. Questo set deve soddisfare una serie di requisiti fondamentali senza i quali il framework incontrerebbe serie problematiche di applicabilità e usabilità. Questi requisiti sono:

[2]La tassellazione è un processo attraverso cui una superficie viene suddivisa in poligoni non sovrapposti.[3]

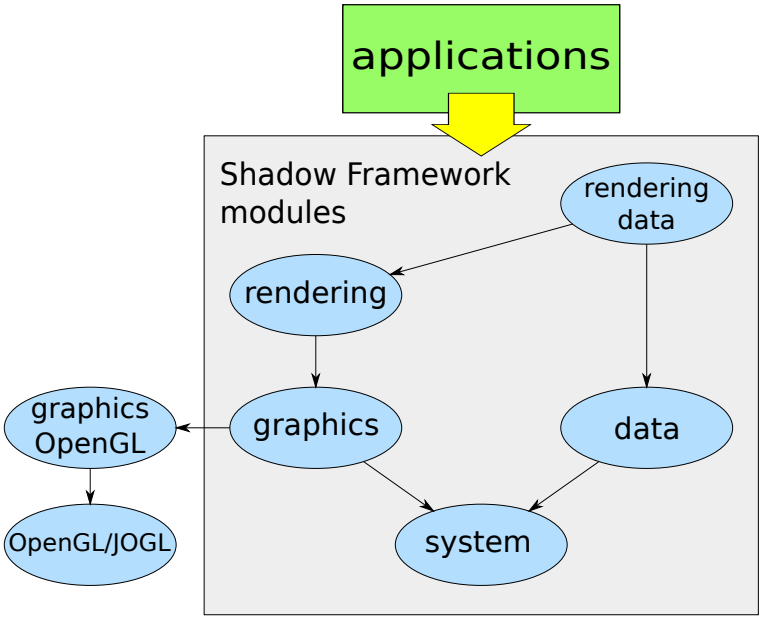

Figura 2.4: Struttura dei moduli del framework.

- la **completezza**, ovvero deve essere in grado di rappresentare ogni tipo di superficie del mondo reale;
- la **semplicità**, in quanto un set troppo complesso e articolato renderebbe il processo di selezione delle geometrie problematico;
- la **flessibilità**, nel senso che le geometrie devono essere in grado di scalare efficacemente in qualità per venire incontro alle capacità di apparecchiature differenti.

Questo è uno degli aspetti di ricerca e sperimentazione più interessanti del framework che, sebbene sia già sufficientemente maturo, è attualmente oggetto di ulteriore sviluppo.

2.3 Struttura dello Shadow Framework 2.0

Il framework è strutturato in moduli gerarchici e la figura 2.4 ne illustra i livelli e le loro dipendenze. Il livello più basso è quello denominato **system** e contiene al suo interno le astrazioni base sfruttate da tutti gli altri moduli. Ad un livello superiore troviamo il modulo **data** che si occupa di tutti gli

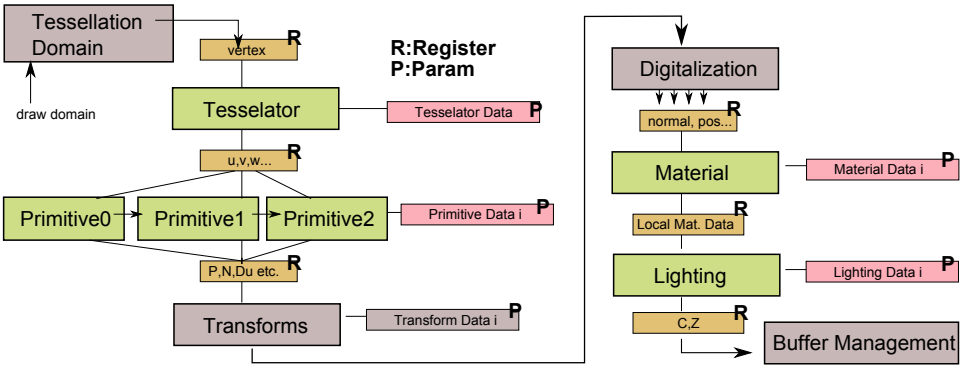

Figura 2.5: Rappresentazione della pipeline ideale implementata dal framework.

aspetti riguardanti la gestione dati all'interno del framework. Per raggiungere gli obbiettivi prefissati il progetto di tesi è incentrato sull'estensione delle funzionalità di questo livello e il capitolo 3 offre un approfondimento su meccanismi e le astrazioni su cui è fondato. Il livello **graphics** offre un'astrazione su di una pipeline di rendering ideale di cui possiamo osservare una rappresentazione in figura 2.5. Dato che non esiste una implementazione diretta di questa pipeline, il framework la realizza configurando una pipeline programmabile attraverso l'API OpenGL a cui si interfaccia tramite il modulo **graphics OpenGL**. Al di sopra del modulo graphics vi è **rendering**, che contiene il rendering engine vero e proprio che basa il suo funzionamento sulle caratteristiche della pipeline astratta del livello inferiore. Il livello **rendering data** definisce infine la struttura dei dati grafici, questa struttura dipende dai livelli inferiori perchè deve tenere conto del modulo data per quanto concerne il reperimento dei dati, e del modulo rendering per i meccanismi di costruzione dei dati.

Le applicazioni costruite basandosi sul framework non hanno necessità di operare direttamente sulle API grafiche a basso livello, ma possono essere strutturate per lavorare sulle astrazioni di alto livello offerte dal framework.

La versione del framework di riferimento, nonché quella oggetto di sviluppo in questo progetto di tesi, è quella realizzata in linguaggio Java e disponibile tramite la fonte indicata al paragrafo A.2.1. Per questa versione il modulo **graphics OpenGL** non si interfaccia direttamente con le API

OpenGL, ma sfrutta a tal scopo la libreria del progetto JOGL[3].

[3]Per informazioni sulla libreria JOGL fare riferimento al paragrafo A.2.2.

Capitolo 3

Gestione dei dati nello Shadow Framework 2.0

La gestione dei dati è un compito molto importante all'interno del framework. Attraverso l'utilizzo di un layer di gestione dati astratto, ogni mudulo del framework può essere salvato e caricato da file o trasferito attraverso un qualsiasi flusso di dati. In questo capitolo viene presentata l'astrazione utilizzata dallo Shadow Framework nella gestione dei dati, le funzionalità messe a disposizione ed i principali package e moduli coinvolti.

3.1 Dati grafici

Data la natura del framework la tipologia di dato più critica e importante è quella che descrive le informazioni grafiche. L'unità base di ogni dato di tipo grafico è l'`SFDataAsset`, questa è una classe astratta generica che serve ad automatizzare il processo di costruzione dei dati e inizializzazione degli stessi nella memoria grafica. Questo processo non può però essere effettuato in qualsiasi momento, ma deve essere correttamente sincronizzato con il processo di rendering della pipeline del framework, in caso contrario i dati potrebbero venir alterati durante il disegno della scena portando ad effetti inaspettati.

Il diagramma della figura 3.1 mostra la sequenza di operazioni necessarie affinché le informazioni del DataAsset grafico vengano costruite correttamente e inizializzate nella memoria grafica in modo sincrono con il processo di rendering, gestito dal frame grafico **SFDrawableFrame**. Partendo da in

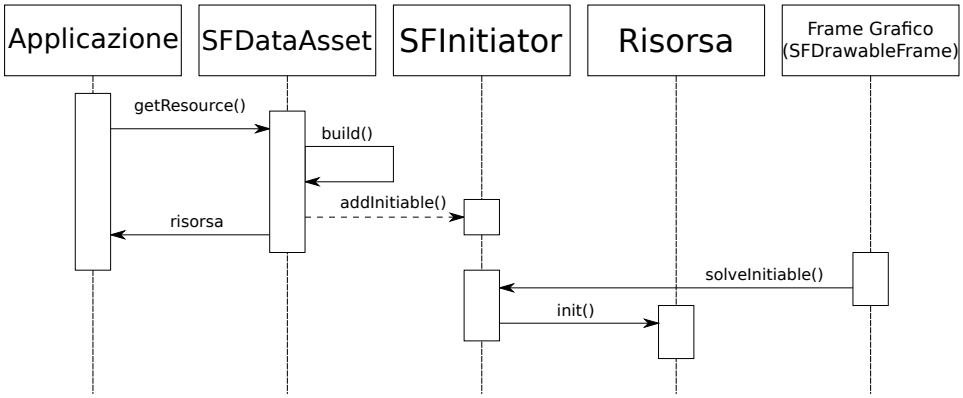

Figura 3.1: Diagramma di sequenza della fase di costruzione di un SFDataAsset.

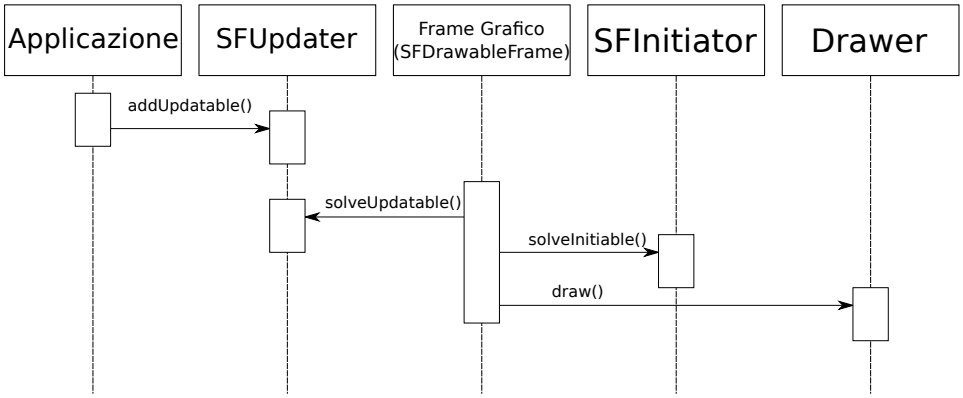

Figura 3.2: Diagramma di sequenza dell'update dei dati.

alto a sinistra l'applicazione richiede la risorsa all'`SFDataAsset`, questo la costruisce con il metodo `build()` e notifica all'`SFInitiator` che la risorsa necessita di essere inizializzata nella memoria grafica tramite la chiamata `addInitiable()`, dopo di che il riferimento alla risorsa viene restituito all'applicazione. Il processo di rendering chiede all'Initiator, ciclicamente e nel momento opportuno, di inizializzare nella memoria grafica tutti i DataAsset che sono in attesa di farlo. L'Initiator richiama allora il metodo `init()` di tutte le risorse in attesa di inizializzazione. Questo procedimento assicura che tutte le risorse necessarie al processo di rendering siano inizializzate correttamente prima di essere usate.

Internamente al framework esiste inoltre un meccanismo di sincronizza-

zione analogo pensato per effettuare un aggiornamento dei dati che sono già stati inizializzati. Questo sistema si basa sull'utilizzo di un Updater che si comporta analogamente a quanto visto con l'Initiator. Il diagramma di sequenza in figura 3.2 mostra la sequenza temporale di un update dei dati: quando l'applicazione necessita di aggiornare i dati grafici passa all'`SFUpdater` un metodo di callback che contiene al suo interno tutte le operazioni da eseguire durante l'aggiornamento. Il frame grafico che esegue ciclicamente il rendering, prima di effettuarlo, chiede all'Updater e poi all'Initiator di risolvere tutte le operazioni che hanno in sospeso. L'ordine in questo caso è di fondamentale importanza perché l'aggiornamento dei dati potrebbe comportare l'aggiunta di DataAsset che necessitano di essere inizializzati. Questo sistema è stato inizialmente pensato per gli aggiornamenti delle animazioni, ma può essere sfruttato per qualsiasi operazione critica di aggiornamento dei dati.

3.2 L'astrazione della gestione dati

All'interno di un'applicazione SF l'unità base di dati può essere identificata con quello che viene definito SFDataset la cui spiegazione verrà dettagliata nel paragrafo 3.3.3. Con Dataset si identifica quasi ogni tipo di dato, sia grafico che non, utilizzato all'interno del framework: un `SFDataAsset` ad esempio è una sottoclasse di `SFDataset`. La gestione dei dataset viene effettuata mediante un meccanismo centralizzato: ogni applicazione in esecuzione possiede un'istanza di SFDataCenter, la quale è un oggetto *Singleton* che realizza un *Bridge* tra l'astrazione di reperimento dati e la sua implementazione concreta[1]. Ogni componente può accedere al DataCenter per richiedere operazioni sui Dataset di interesse, come la lettura o la scrittura da uno stream specifico, la richiesta di una particolare istanza di un Dataset, identificata per nome, o la richiesta di una nuova istanza di Dataset, identificata per tipo. L'oggetto Singleton espone queste funzionalità traducendole internamente con chiamate ad una *factory* concreta[2] di Dataset e ad una istanza dell'interfaccia SFIDataCenter, creando un'astrazione su come vengano effettivamente costruiti e reperiti i Dataset, come mostrato

[1] Con *Singleton* e *Bridge* si intendono i design pattern omonimi descritti più in dettaglio nell'appendice B

[2] Si fa riferimento al pattern di programmazione *Abstract Factory* descritto nella sezione B.1.1.

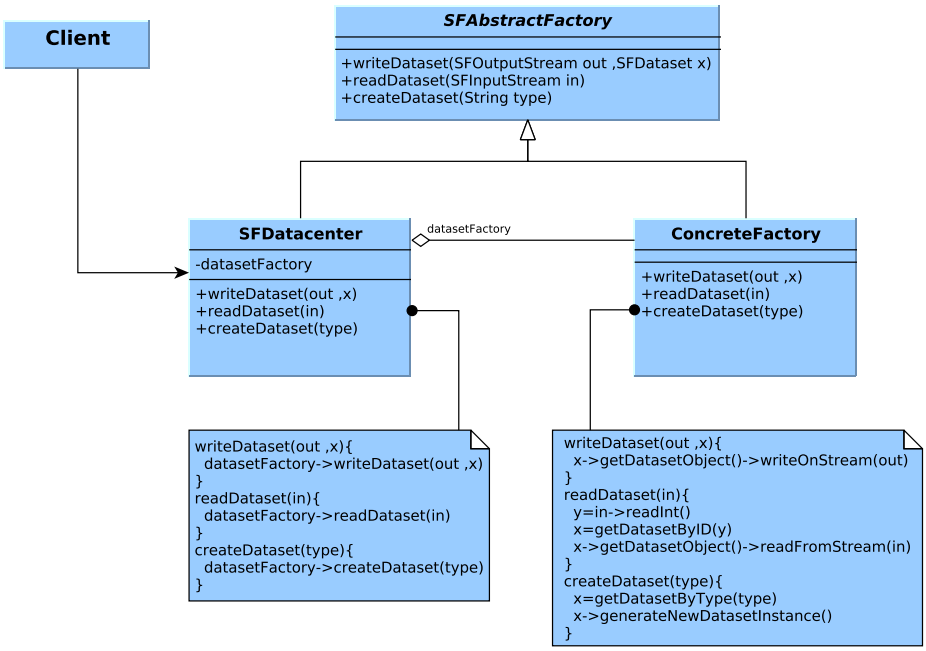

Figura 3.3: Diagramma del Bridge composto da SFDataCenter e da un'istanza concreta di SFAbstractFactory.

nelle immagini 3.3 e 3.4. La factory concreta deve essere un'implementazione dell'interfaccia SFAbstractDatasetFactory in grado di istanziare, leggere o scrivere ogni tipo di Dataset utilizzato dall'applicazione. L'istanza dell'interfaccia SFIDataCenter tiene traccia dei Dataset istanziati con nome, restituendone un riferimento a chi ne fa richiesta attraverso la chiamata a funzioni di callback.

3.3 Il package `shadow.system.data`

Questo package contiene una serie di classi ed interfacce su cui si basa l'astrazione dei dati del framework.

3.3.1 SFInputputStream e SFOutputStream

Queste interfacce definiscono le operazioni necessarie che uno stream di input o di output deve implementare affinché sia possibile leggere o scrivere su di esso dei DataObject e insieme costituiscono un elemento molto im-

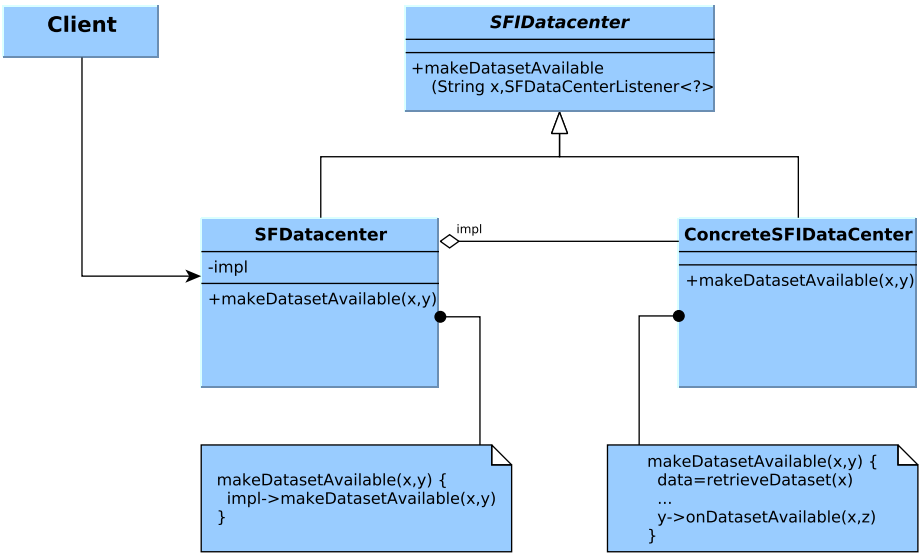

Figura 3.4: Diagramma del Bridge composto da SFDataCenter e da un'istanza concreta di SFIDataCenter.

portante per l'estendibilità del framework sui dati. Su di esse si basa infatti l'astrazione che i dati utilizzano per completare le operazioni di lettura e scrittura. Utilizzando la medesima interfaccia di astrazione è possibile far comunicare tra loro anche implementazioni diverse del framework.

3.3.2 SFDataObject

Uno dei moduli principali del package è `SFDataObject`, che rappresenta un'interfaccia con funzionalità di base comuni ad ogni oggetto che contiene dati. Ogni oggetto di questo tipo può perciò:

- essere scritto su di un `SFOutputStream`;
- essere letto da un `SFInputStream`;
- essere clonato;

I DataObject si basano sul *Composite Pattern* (B.2.2): possono essere semplici o contenere un insieme di oggetti figli. Il fatto che sia gli oggetti complessi che quelli semplici condividano la stessa interfaccia permette di trattarli in maniera uniforme. Un oggetto contenitore dovrà semplicemente richiamare

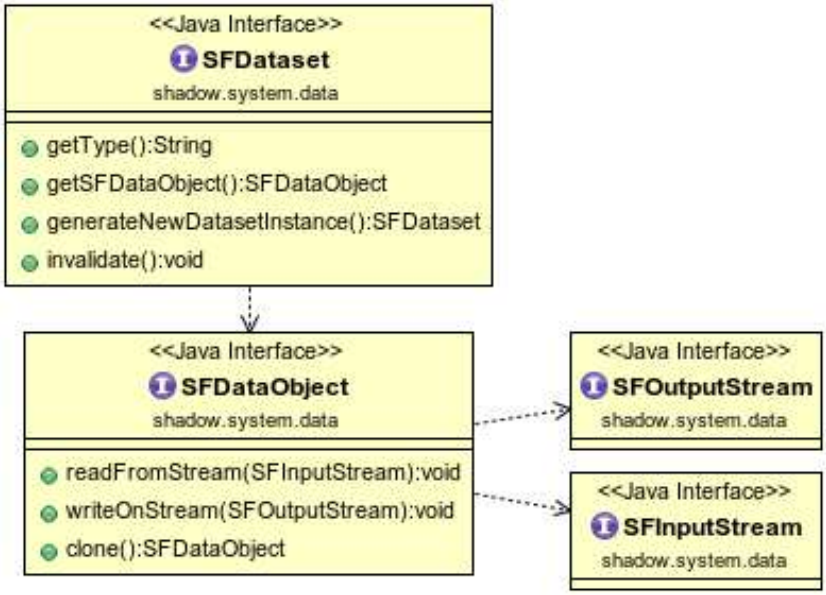

Figura 3.5: Diagramma della relazione tra le classi SFDataset, SFDataObject, SFInputStream e SFOutputStream.

lo stesso metodo di interfaccia per tutti gli oggetti figli i quali, se oggetti semplici, hanno la responsabilità di implementare l'algoritmo per leggere o scrivere se stessi da uno stream.

Tutti i componenti SF utilizzano dei DataObject per incapsulare i dati in modo che questi ultimi possano essere letti e scritti utilizzando stream appropriati.

3.3.3 SFDataset

Un altro modulo importante per la gestione dei dati è `SFDataset`. Un Dataset è un oggetto che contiene un DataObject e informazioni sul proprio tipo, rappresentato tramite una stringa. Da come si può intuire dalla figura 3.5 il DataObject viene sfruttato dal Dataset per incapsulare i suoi dati interni ed incorporarne le funzionalità di lettura e scrittura. L'interfaccia SFDataset definisce un'interfaccia per oggetti di questo tipo, la quale consente di accedere al nome del tipo specifico, al DataObject contenuto e di creare una nuova istanza delle stesso tipo. A loro volta i Dataset possono essere incapsulati in un DataObject usando un oggetto `SFDatasetObject`.

3.3.4 SFAbstractDatasetFactory

Questa interfaccia definisce le operazioni base richieste ad una DatasetFactory. Le operazioni consistono in:

- lettura/scrittura di un Dataset da uno stream;
- creazione di una nuova istanza di un Dataset specificato per tipo.

3.3.5 SFIDataCenter

L'interfaccia `SFIDataCenter` fornisce l'astrazione di una Mappa di Dataset identificati attraverso il proprio nome. Grazie ad essa possiamo chiedere ad un oggetto che la implementa di recuperare un Dataset specifico. Quest'oggetto non deve restituire direttamente il Dataset recuperato, ma deve farlo attraverso un meccanismo di callback ad una implementazione dell'interfaccia `SFDataCenterListener` passata come parametro, nel momento in cui il dato è disponibile.

3.3.6 SFDataCenterListener

Questa interfaccia definisce la callback che un componente deve implementare per effettuare una richiesta al DataCenter. La callback viene richiamata quando il Dataset richiesto è pronto.

3.3.7 SFDataCenter

Il **DataCenter** è il nodo fondamentale della gestione dei dati all'interno del framework.

É un oggetto *Singleton* (B.1.2) a cui le applicazioni accedono per richiedere i Dataset di cui hanno bisogno. Questa classe utilizza anche il pattern *Bridge* (B.2.1), fornendo un'astrazione su come i dati sono effettivamente reperiti.

Per poter funzionare, al DataCenter deve essere fornita un'implementazione per:

- `SFAbstractDatasetFactory`
- `SFIDataCenter`

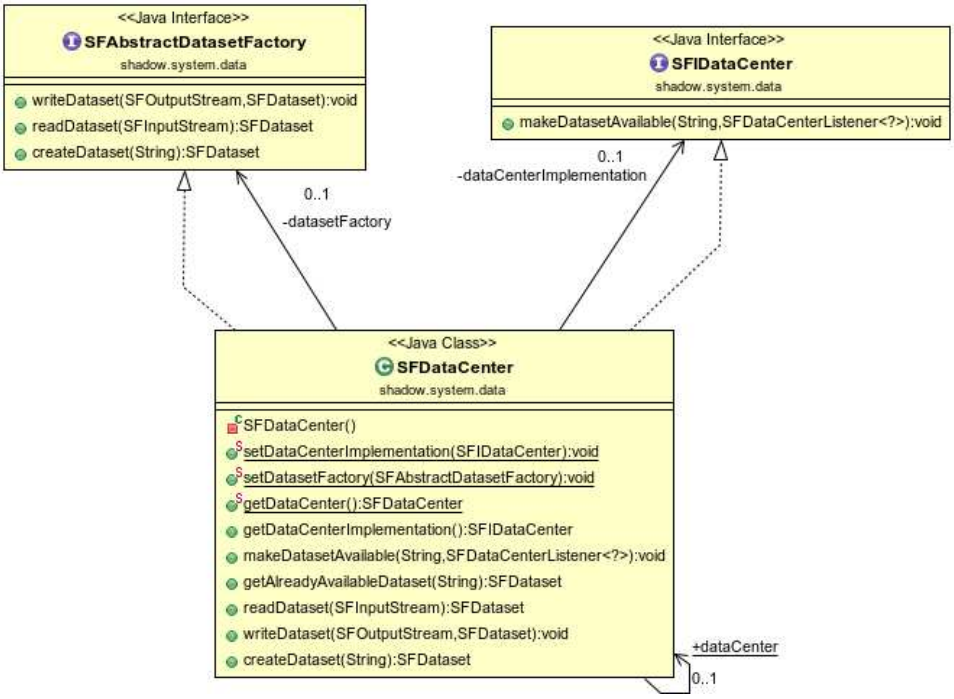

Figura 3.6: In questo diagramma viene mostrata la relazione tra le classi SFDataCenter, SFIDataCenter e SFAbstractDatasetFactory. Da notare che SFDataCenter è una classe Singleton in quanto contiene una istanza statica di se stessa (l'istanza dataCenter sottolineata) e possiede un costruttore privato (evidenziato in rosso). Inoltre, in riferimento alla sezione B.2.1, si può notare come SFDataCenter realizzi un Bridge sia con SFIDataCenter che con SFAbstractDatasetFactory

Come precedentemente esposto, l'implementazione di `SFAbstractDatasetFactory` deve essere essere una factory in grado di generare istanze di tutti i tipi di Dataset necessari all'applicazione.

Questo tipo di astrazione permette di separare la logica di utilizzo del Dataset da quella di come esso viene reperito, consentendo ad una applicazione di usare dati locali o dati di rete semplicemente cambiando l'implementazione di `SFIDataCenter`.

3.3.8 SFObjectsLibrary

É usata per memorizzare un set di Dataset ed al suo interno ogni elemento è identificato tramite un nome univoco. Un `SFObjectsLibrary` è a sua volta un Dataset, così che un ObjectsLibrary possa essere contenuta in altre ObjectsLibrary. É possibile, ad esempio, utilizzare una ObjectsLibrary all'interno di implementazione di `SFIDataCenter` per creare una mappa di Dataset necessari al funzionamento di un'applicazione.

3.4 Classi di utilità per il layer dati

3.4.1 SFLibraryreference

Un LibraryReference è un DataObject che può essere usato da qualsiasi componente per avere un riferimento ad un Dataset memorizzato in una libreria. Viene utilizzato all'interno di DataObject o di Dataset per non avere istanze doppie dello stesso dato.

3.4.2 SFGenericDatasetFactory

Questa classe di utilità consiste in una implementazione concreta di default dell'interfaccia `SFAbstractDatasetFactory`. Per consentire il riutilizzo del codice questa implementazione sfrutta il meccanismo di *factory con prototipo*: una *factory* pura deve avere un metodo specifico per ogni tipo di oggetto che può istanziare, questo rende predeterminato il numero di tipi istanziabili e rende necessario modificare il codice, aggiungendo un nuovo metodo, quando si desidera modificare le capacità della *factory*. Sfruttando invece il meccanismo dei prototipi non si ha un numero predeterminato di tipi istanziabili poiché diviene configurabile: attraverso un metodo apposito vengono passati come parametro i prototipi di ogni oggetto che la

factory deve essere in grado di creare. Questo sposta la responsabilità dalla conoscenza di come ogni singolo oggetto deve essere allocato sull'oggetto stesso, evitando di modificare la *factory* ogni volta che un oggetto viene alterato. Nel concreto la `SFGenericDatasetFactory` è resa configurabile tramite l'aggiunta di un metodo addSFDataset() che consente di generare, in un oggetto GenericDatasetFactory, un elenco di Dataset istanziabili. Quando verranno chiamati i metodi dell'interfaccia SFAbstractDatasetFactory sull'oggetto GenericDatasetFactory diviene sufficiente richiamare il metodo `generateNewDatasetInstance()` del Dataset del tipo richiesto.

Capitolo 4

Il Progetto SF-Remote-Connection

Vengono ora presentati i moduli software realizzati nel corso del progetto di tesi. Per meglio comprendere gli aspetti di sviluppo legati al progetto è importante tener conto degli obbiettivi esposti al paragrafo 1.1 e della descrizione dei meccanismi di gestione dati interni del framework, descritti nel capitolo 3.

Nella sezione 4.1 del capitolo viene fornita innanzitutto una suddivisione e una descrizione dei moduli, nella sezione 4.2 viene descritto il meccanismo dei Dataset sostitutivi e nella 4.3 l'infrastruttura di rete e il protocollo di comunicazione.

L'obiettivo principale ha riguardato la produzione di librerie e tool per lo sviluppo di applicazioni, focalizzandosi sull'estendibilità e il riutilizzo del codice. Non a caso sono infatti presenti alcuni riferimenti di appendice ad alcuni design pattern particolarmente significativi nella produzione di questo tipo di software e che sono utilizzati sia dal framework che dai moduli stessi.

Per il processo di sviluppo è stata di fondamentale importanza la produzione parallela di una serie di test, presentati nel capitolo 5. Infatti la progettazione e la realizzazione dei moduli e dei meccanismi presentati in questo capitolo non è stata svolta in maniera distinta, ma si è trattato di un processo iterativo in cui i test hanno svolto più di una volta un ruolo di guida nel refactoring del codice.

Si rimanda all'appendice A per informazioni sul codice sorgente relativo al progetto e per informazioni sulle versioni delle librerie utilizzate.

4.1 Moduli

La libreria di classi realizzata può essere suddivisa in quattro macro-moduli suddivisi in base alle finalità e alle funzionalità. Di seguito viene data una descrizione degli stessi in questo ordine:

1. **Base Communication**
2. **RemoteDataCenter Tool**
3. **Client**
4. **Server**

4.1.1 Base Communication

Questo modulo riunisce le classi che consentono la creazione e la gestione di connessioni TCP/IP tra applicazioni client/server. Ne fa parte anche la classe di utilità `GenericCommunicator` che oltre a consentire la gestione della connessione assegnatagli la utilizza per fornire funzionalità di lettura e scrittura di messaggi testuali attraverso il canale aperto.

Di grande importanza per questo modulo sono anche la classe `CommunicationProtocol` e l'interfaccia `ICommunicationProtocolTask` che, come vedremo nei casi di **Client** e **Server**, possono essere utilizzate da moduli esterni che sfruttano il design pattern *State*[1] per gestire il protocollo di comunicazione attraverso una macchina a stati.

Il modulo è composto dai package `sfrc.base.communication` e `sfrc.base.communication.sfutil`.

4.1.2 RemoteDataCenter Tool

Questo modulo raggruppa una serie di classi pensate per estendere il framework e per essere utilizzate all'interno di una applicazione client. La funzione principale consiste nel fornire uno strato di comunicazione tra l'astrazione del reperimento dati fornita dal framework e il meccanismo di effettivo reperimento dei dati.

La classe chiave del modulo è `SFRemoteDataCenter` la quale è una classe di implementazione utilizzabile nel *Bridge* realizzato da `SFDataCenter`[2].

[1] Il pattern *State* viene descritto al paragrafo B.3.1

[2] Per la classe `SFDataCenter` si rimanda al paragrafo 3.3.7 mentre per il pattern *Bridge* al paragrafo B.2.1.

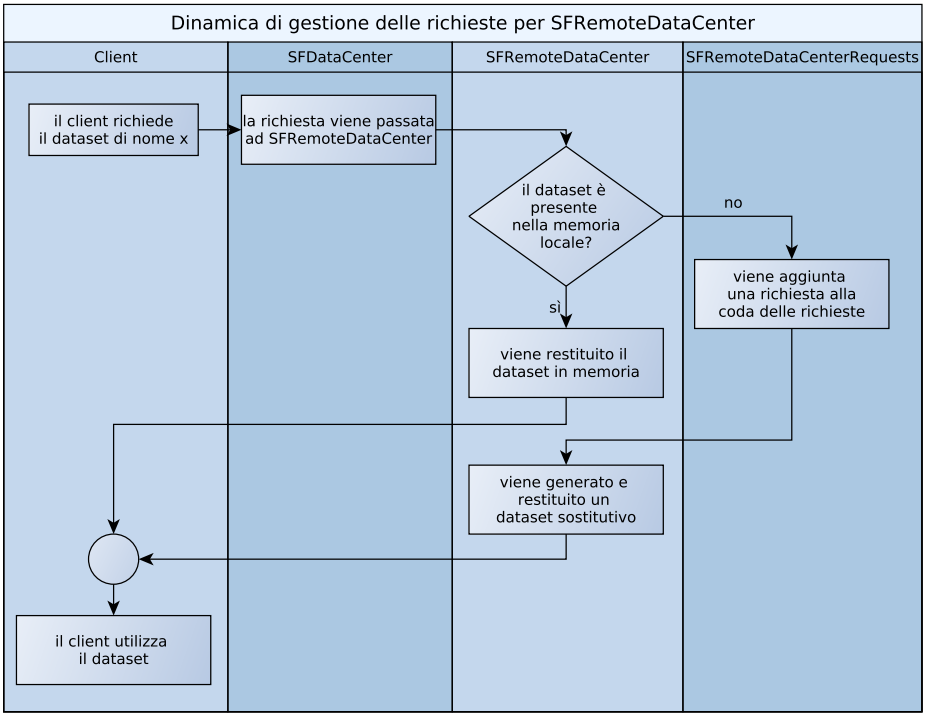

Figura 4.1: Lo schema mostra la sequenza di eventi dovuti ad una richiesta a SFDataCenter che utilizza un SFRemoteDataCenter come implementazione interna.

Dallo schema in figura 4.1 osserviamo che le richieste di Dataset effettuate al DataCenter vengono passate a questa classe che le esamina verificando che il dato richiesto sia presente nella libreria dell'applicazione. Se il Dataset non è presente viene generata una richiesta e aggiunta ad un buffer di richieste di nome `SFRemoteDataCenterRequests`, mentre al richiedente viene restituito un Dataset sostitutivo temporaneo scelto opportunamente. Il buffer, che supporta la sincronizzazione, può contemporaneamente essere utilizzato da un modulo esterno in grado di effettuare l'effettivo reperimento dei dati. Il meccanismo dei Dataset sostitutivi viene descritto esaustivamente nella sezione 4.2.

Il modulo è composto dai package `shadow.system.data.remote.wip`, `shadow.system.data.object.wip` e `shadow.renderer.viewer.wip`.

4.1.3 Client

Questo modulo raggruppa delle componenti generiche che possono essere utilizzate all'interno di una qualsiasi applicazione client e che servono ad implementare l'effettivo reperimento dei dati. Esso si pone al di sotto del modulo **RemoteDataCenter Tool** ed utilizza il modulo **Base Communication** per la gestione del canale di comunicazione e la sua implementazione è pensata per il multi-threading. Il meccanismo messo a disposizione è mostrato nella figura 4.2, dove un thread `RemoteDataCenterRequestsCreationTask` viene risvegliato periodicamente e controlla che non vi siano richieste pendenti nel buffer delle richieste. Se il buffer non è vuoto viene allocano un nuovo thread di tipo `RemoteDataCenterRequestTask` e mentre il precedente viene nuovamente sospeso questo effettua le richieste al server. Quando il secondo thread termina la comunicazione chiede al buffer di generare un update a tutti i client che avevano richiesto i dati reperiti e successivamente si chiude.

Il package che compone questo modulo è `sfrc.application.client`.

4.1.4 Server

Similmente a quello per le componenti client, in questo modulo vengono raggruppate delle componenti generiche utili alla realizzazione di una applicazione server. Queste componenti fungono da tramite tra l'applicazione e il modulo di **Base Communication** grazie al quale realizzano l'effettivo trasferimento dei Dataset verso il client connesso. L'implementazione è pensata per il funzionamento multi-thread in parallelo con l'applicazione principale che può così gestire più client connessi contemporaneamente ed eseguire altre operazioni. Vengono fornite infine anche delle interfacce utili per effettuare l'inizializzazione dei dati e per configurare il protocollo di comunicazione.

Il modulo è composto dal package `sfrc.application.server`

4.2 Dataset sostitutivi

Per evitare che, in seguito alla richiesta di un Dataset non presente nella libreria locale di un'applicazione, i moduli richiedenti rimanessero in attesa dei dati bloccando di fatto l'esecuzione, è stato realizzato un meccanismo che sostituisce i dati richiesti con dei Dataset alternativi, già presenti nella

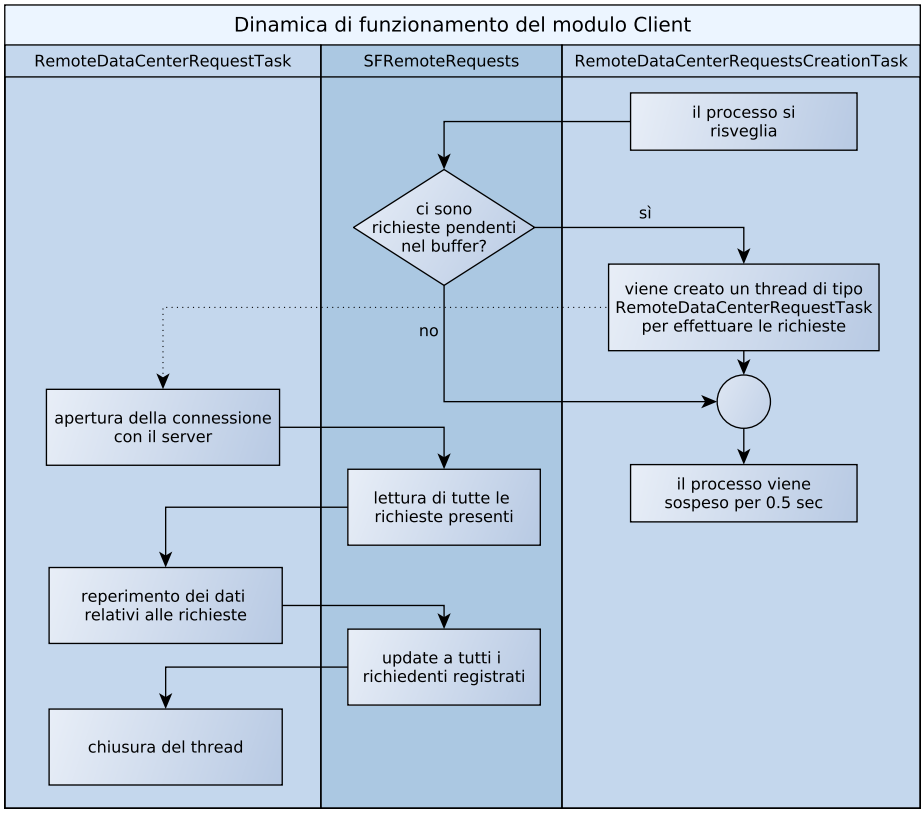

Figura 4.2: Nello schema presentato viene messa in evidenza la sostanziale indipendenza dei thread `RemoteDataCenterRequestsCreationTask` e `RemoteDataCenterRequestTask`, infatti escludendo la generazione dei thread del secondo tipo da parte del primo, non vi sono comunicazioni dirette tra essi.

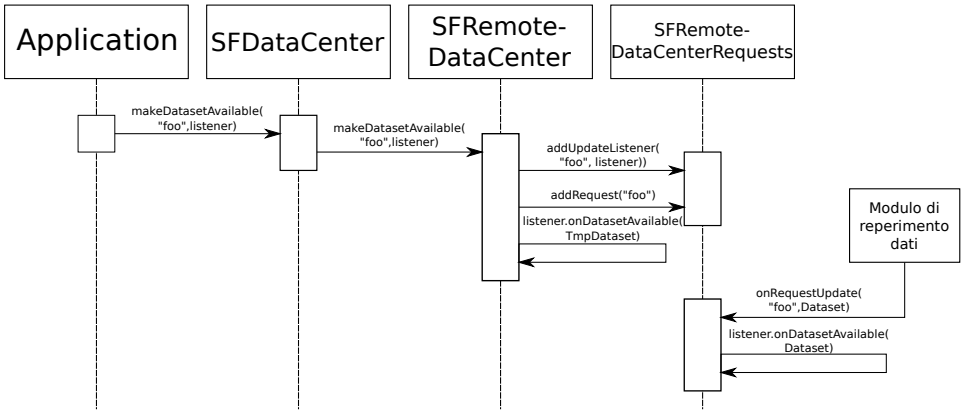

Figura 4.3: Diagramma di sequenza del meccanismo a Dataset sostitutivi.

memoria del client. Al successivo arrivo dei dati reali viene effettuato un update tramite i meccanismi di sincronizzazione offerti dal framework.

Il diagramma di sequenza in figura 4.3 mostra la dinamica temporale di funzionamento del sistema nel caso in cui l'applicazione richieda un Dataset non ancora presente in memoria: l'applicazione effettua la richiesta ad `SFDataCenter` passandone il nome e la callback necessaria per restituire il dato. Questi vengono poi passati ad `SFRemoteDataCenter` che, una volta appurata l'assenza del dato in memoria, registra alla coda delle richieste `SFRemoteDataCenterRequests` prima la callback e poi la richiesta stessa. L'`SFRemoteDataCenter` genera successivamente un Dataset sostitutivo (TmpDataset) che viene usato come parametro quando viene chiamata la callback `listener.onDatasetAvailable(TmpDataset)`. Quando il modulo di reperimento dei dati riceve il Dataset reale lo notifica alla coda attraverso il metodo `onRequestUpdate(...)`.

`SFRemoteDataCenterRequests` prende in carico il compito di effettuare l'update, richiamando tutte le callback registrate e associate al dato ricevuto.

Il passaggio critico dell'aggiornamento dei dati, soprattutto quelli grafici, viene effettuato sfruttando il meccanismo dell'Updater mostrato nella sezione 3.1.

Sfruttando il fatto di non bloccare l'esecuzione in attesa dei dati, l'utilizzo dei Dataset sostitutivi consente la visualizzazione di una scena in cui i modelli oggetto della simulazione sono temporaneamente rimpiazzati da dei segnaposto, ma in cui è già possibile navigare all'interno e avviare l'intera-

zione con l'ambiente circostante. La scelta dei dati sostitutivi da parte di `SFRemoteDataCenter` deve essere effettuata in maniera appropriata a seconda dell'oggetto che si deve rimpiazzare, ma per permettere il funzionamento di questo automatismo si è resa necessaria la realizzazione di un nuovo tipo di Dataset, l'SFDatasetReplacement, e di una libreria di Dataset sostitutivi.

Utilizzato all'interno di una ObjectsLibrary un DatasetReplacement permette di realizzare una lista di sostituzione che associa il nome di un Dataset "Alfa" richiesto, a quello di un Dataset sostitutivo di default "Beta" e ad un timestamp.

Se l'associazione tra nomi viene usata per una ricerca diretta del dato sostitutivo da utilizzare, il timestamp è stato introdotto per lo sviluppo futuro di logiche di aggiornamento della lista di sostituzione, attraverso un confronto dei timestamp locali con quelli remoti. Sul server si possono così inserire meccaniche di modifica ed espansione dell'ambiente simulato centralizzate ed automatizzate che non richiedano all'utente di bloccare la fruizione dei contenuti nell'attesa del download degli aggiornamenti.

L'utilizzo di questo automatismo richiede necessariamente una fase di inizializzazione in cui viene fatto il download o il caricamento da disco locale della lista di sostituzione e della libreria dei Dataset di default. Avendo la possibilità di integrare il protocollo di comunicazione estendendo le due librerie durante l'esecuzione, è possibile mantenere la loro dimensione iniziale contenuta.

La libreria di Dataset sostitutivi viene descritta in maggiore dettaglio nella sezione 5.3.

4.3 Infrastruttura di rete

Date le specifiche iniziali esposte nel capitolo 1, una parte fondamentale del progetto è stata decidere quale infrastruttura per la comunicazione di rete sfruttare per poter utilizzare e testare il modulo **RemoteDataCenter Tool**, che si occupa della gestione delle richieste di Dataset.

Se da lato client è ovvia la necessità di sviluppare uno strato dell'applicazione che si occupa della comunicazione di rete, da lato server si sono presentate diverse possibilità:

1. utilizzare un file server che permettesse semplicemente di accedere ai file contenenti i dati tramite la rete;

2. utilizzare un application server java, come Tomcat o Glassfish, a cui un'applicazione client potesse connettersi e che attraverso l'esecuzione di servlet realizzasse il trasferimento dei dati da server a client;

3. utilizzare un'applicazione server ad-hoc appositamente sviluppata;

La prima soluzione è probabilmente la più semplice, ma la meno flessibile poiché consente solo un accesso diretto ai file di descrizione dei dati senza alcuna possibilità di un'elaborazione lato server e spostando tutto il peso computazionale di un'eventuale interazione tra client direttamente su quest'ultimi.

La seconda soluzione offre più possibilità e flessibilità rispetto alla prima: l'utilizzo di un application server java mette a disposizione una piattaforma che consente una pre-elaborazione dei dati lato server e possiede direttamente una serie di componenti per la gestione di compiti complessi legati alle sessioni degli utenti, come ad esempio l'autenticazione e la sicurezza. Nonostante i pregi, questo tipo di soluzione presenta lati negativi: gli application server generici non sono sviluppati per questo tipo di applicazioni e non è garantita una flessibilità sufficiente per cui all'aumentare della complessità del progetto, e delle sue esigenze non sia necessario abbandonare l'architettura.

La terza soluzione è sicuramente la più flessibile ed estendibile poiché consente di modificare direttamente il server per adattarsi alle esigenze dell'applicazione. Lo svantaggio è la necessità di dover implementare da zero tutte quelle funzionalità non solo di comunicazione, ma anche di autenticazione o di sicurezza che una soluzione già pronta potrebbe possedere nativamente.

Nella scelta tra le tre soluzioni hanno condizionato la volontà di realizzare un'architettura funzionante con la scrittura di meno codice possibile e quella di mantenere una bassa complessità iniziale che non penalizzasse un'estensione futura delle funzionalità. In quest'ottica la prima soluzione è stata la prima ad essere scartata, perché sebbene garantisse un tempo di messa in opera molto basso, non consente un'estensione di funzionalità.

La scelta finale è ricaduta sulla terza soluzione che pur costringendo a rinunciare alle funzionalità avanzate della seconda, elimina difficoltà e tempistiche di una installazione e configurazione dell'application server. Inoltre, limitando inizialmente lo sviluppo alle funzionalità di base, è possibile ri-

durre la complessità del codice ad un livello non più elevato rispetto alla seconda soluzione.

4.3.1 Protocollo di comunicazione

Una volta stabilito di utilizzare un server sviluppato appositamente è stato necessario stabilire le modalità di comunicazione tra client e server.

La comunicazione tra le applicazioni viene effettuata attraverso un protocollo basato su messaggi testuali che si colloca idealmente a livello di Sessione nel modello ISO/OSI [9] mentre a livello inferiore viene utilizzato il protocollo TCP/IP. L'utilizzo del TCP, in quanto protocollo confermato, è giustificato in quanto garantisce la ricezione dei messaggi di comunicazione. Poiché questi sono principalmente dedicati allo scambio di dati grafici, la garanzia di ricezione e d'integrità risultano prioritari rispetto alle prestazioni.

I messaggi di comunicazione finora previsti sono strutturati secondo la seguente forma:

```
etichetta_messaggio[:dati:opzionali:...]
```

in cui `etichetta_messaggio` identifica il tipo di messaggio, mentre i dati opzionali dipendono dal messaggio stesso e sono divisi dall'etichetta e tra loro dal carattere " : ".

I messaggi sono utilizzati all'interno di un preciso schema di interazione tra client e server di cui possiamo vedere un esempio nella figura 4.4, in cui la numerazione dei nodi rappresenta la sequenza temporale dello scambio di messaggi. Lo schema illustra che, quando la coda delle richieste non è vuota, il client apre una connessione con il server ed invia un messaggio di tipo **request** in cui sono elencati i nomi dei Dataset richiesti. Il server a questo punto genera una sequenza di risposte attraverso cui invia, se possibile, i dati richiesti. Al termine della sequenza invia un messaggio di tipo **reply-end** a cui il client risponde con un messaggio **closing** per terminare la connessione. La gestione della comunicazione e dello schema di interazione viene effettuata tramite una macchina a stati specifica, una per il client e una per il server, ognuna delle quali si occupa di leggere ed effettuare l'analisi logico-sintattica dei messaggi generando le eventuali risposte o semplicemente modificando lo stato interno della macchina stessa.

Usare messaggi testuali per la comunicazione consente di semplificarne la gestione e rende possibile espandere il linguaggio semplicemente utilizzan-

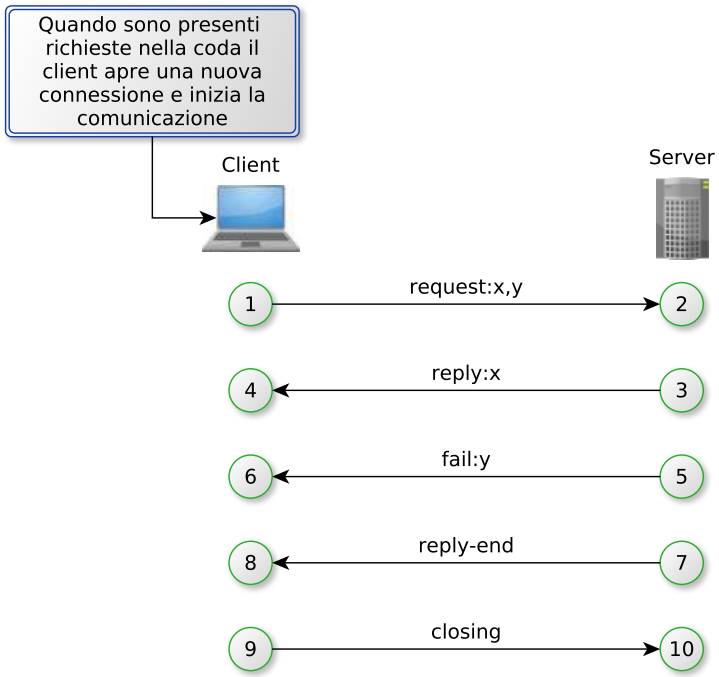

Figura 4.4: Esempio dello schema di interazione tra client e server.

do nuove etichette, tuttavia l'implementazione della macchina a stati deve essere strutturata in modo da garantire l'espandibilità.

Per questo motivo è stato preso come modello il pattern *State*, descritto al paragrafo B.3.1, e la logica di ogni stato è stata incapsulata in un oggetto differente. Questo consente di espandere la macchina a stati semplicemente definendo nuovi oggetti separando il codice della logica di gestione di ogni stato da quello degli altri.

L'immagine in figura 4.5 mostra lo schema a blocchi della macchina a stati attualmente implementata nel modulo client per la gestione del protocollo di comunicazione. Lo stato iniziale dello schema è **request** ed è innescato dalla presenza di richieste nella coda. In questo stato il processo che gestisce la comunicazione invia le richieste al server e successivamente passa allo stato **analize-reply** in attesa di riceve le risposte. Quando viene ricevuto il messaggio di risposta, viene estratta l'informazione sul tipo di messaggio determinando lo stato successivo della macchina. Se ad esempio la risposta è un messaggio di tipo `reply:nomedataset` lo stato successivo sarà quello

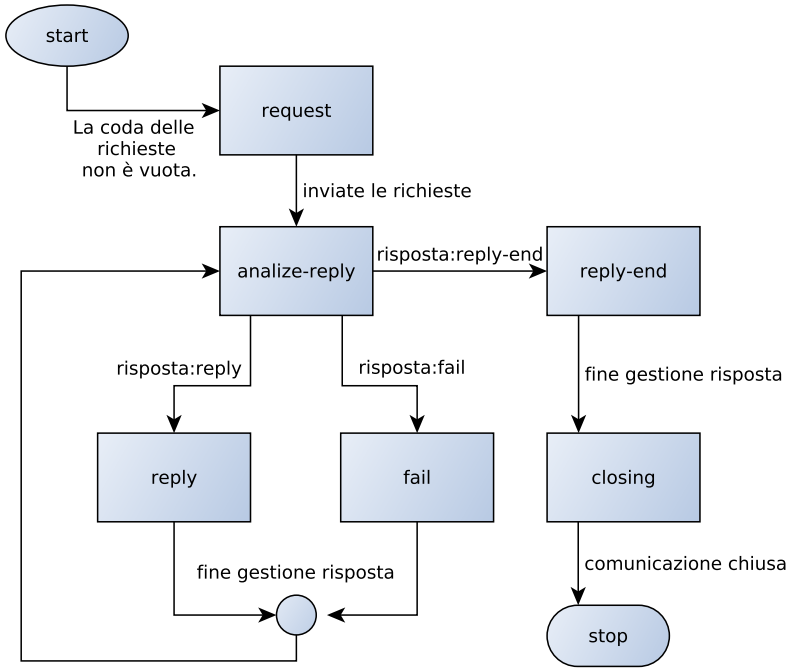

Figura 4.5: Diagramma a stati del client per la gestione delle richieste.

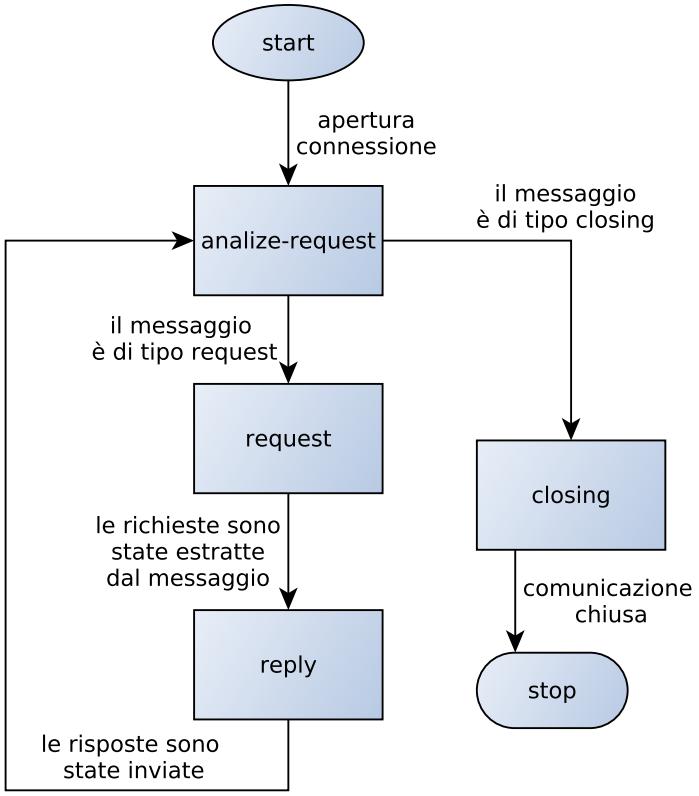

Figura 4.6: Diagramma a stati del server per la gestione delle richieste.

corrispondente alla gestione di quel tipo di messaggio. Gli stati **reply** e **fail** si occupano di gestire il tipo di messaggio corrispondente e al termine della loro esecuzione la macchina torna nello stato di **analize-reply** in attesa di altre risposte. All'arrivo del messaggio di fine risposte lo schema passa allo stato corrispondente di **reply-end** e poi a quello di **closing** dove chiede al server di chiudere la connessione.

La figura 4.6 rappresenta invece lo schema implementato nel modulo server. All'apertura di una connessione il server entra nello stato **analize-request** dove rimane in attesa di messaggi di richiesta da parte del client. Alla ricezione di un messaggio di questo tipo, passa in uno stato **request** dove estrae dal messaggio i nomi di tutti i Dataset richiesti. Successivamente entra nello stato **reply** dove uno ad uno vengono inviati i dati di ogni Dataset richiesto per poi inviare un messaggio `reply-end` al termine

della sequenza. Ritornato nello stato **analize-request** il server è in attesa di ulteriori richieste o di un messaggio di `closing` per entrare nello stato corrispondente dove viene terminata la comunicazione.

Le strutture delle macchine a stati di client e server non sono state definite per l'efficienza, ma per consentire la verifica dei meccanismi di livello più alto in differenti condizioni.

Capitolo 5

Test e Risultati

In questo capitolo vengono presentati e commentati i test effettuati sui moduli di libreria del progetto.

Nella sezione 5.1 viene presentata la struttura dell'impianto di test, sia da lato client che da lato server. Successivamente, nella sezione 5.2, vengono mostrati i risultati dell'esecuzione di alcuni test particolarmente significativi. Infine nella sezione 5.3 vi è la descrizione di alcune attività correlate emerse durante lo sviluppo.

5.1 Struttura dei test

La fase di testing del progetto ha verificato che il funzionamento dei meccanismi implementati all'interno dei moduli esplicati nel capitolo precedente producesse un corretto output visivo.

Oltre a ciò questa fase ha avuto l'obbiettivo di riprodurre i test presenti all'interno del progetto **SF20LiteTestWorld**, che consiste in un modulo di estensione del framework volto a semplificare la gestione delle finestre grafiche nelle applicazioni. Questo modulo è in via di sviluppo e per questo non è ancora parte integrante delle funzionalità di base offerte dallo SF. Tuttavia il codice relativo ad esso è disponibile alla stessa fonte della versione del framework indicata al paragrafo A.2.1. I test presenti in **SF20LiteTestWorld** sono stati pensati non solo per verificare la correttezza delle funzionalità, ma anche per mostrare alcune delle capacità del framework stesso.

Per evitare un'eccessiva duplicazione del codice le classi di implementazione dei test, sia lato client che lato server, sfruttano l'ereditarietà da una

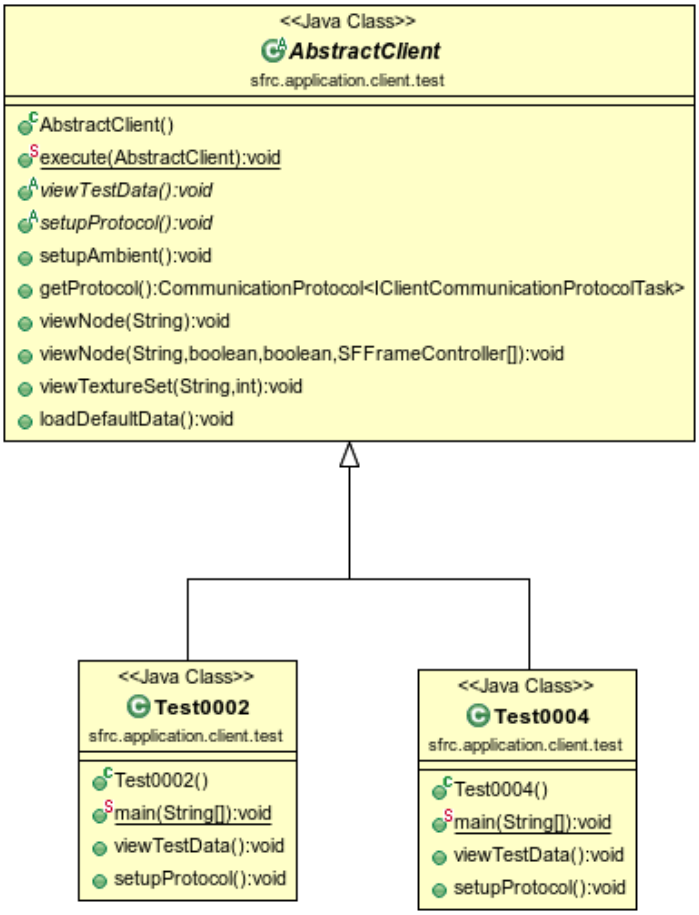

Figura 5.1: Gerarchia delle classi di implementazione dei client.

classe genitore comune per condividere tutte le parti comuni.

In figura 5.1 possiamo vedere la gerarchia delle classi dei client di test. Per implementare un qualsiasi test è sufficiente creare una sottoclasse di `AbstractClient` e implementare al suo interno i metodi astratti della classe genitore. In maniera del tutto simile è possibile vedere in figura 5.2 lo stesso principio applicato per l'implementazione di differenti server di test. Questi, opportunamente configurati, vengono usati per riprodurre le diverse condizioni a cui il client può essere sottoposto durante la comunicazione, ad esempio generando dei ritardi di risposta o l'assenza di alcuni dati. In quest'ultima figura è evidenziata la classe `ServerDataHandler`, la quale si occupa di effettuare la gestione dei dati sul server in esecuzione. Non essendoci la necessità di renderizzare i dati tridimensionali, non è richiesta

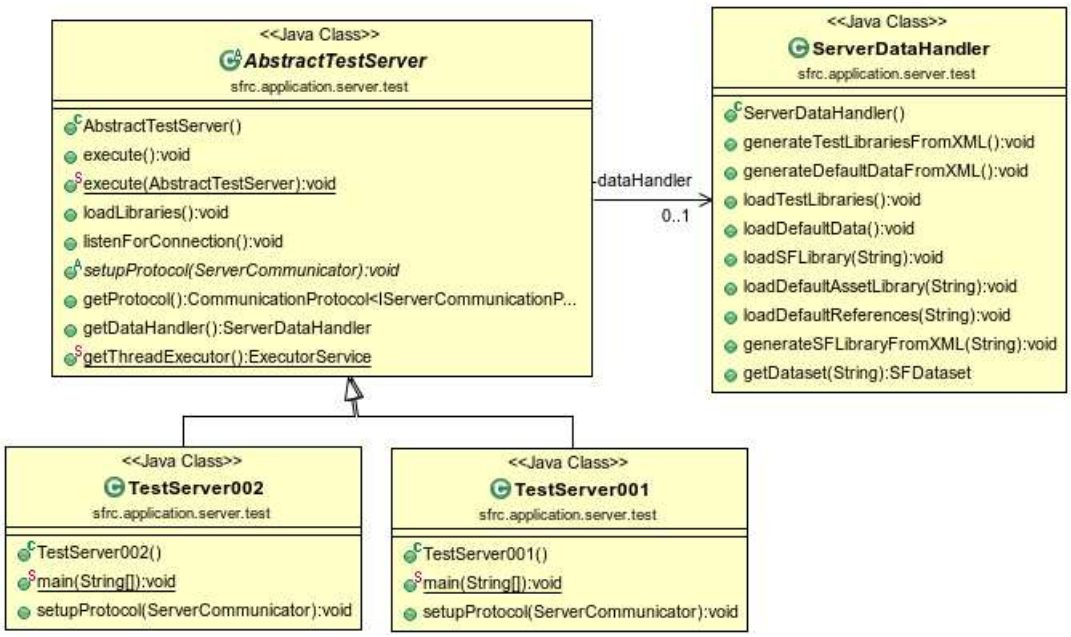

Figura 5.2: Gerarchia delle classi di implementazione dei server.

un'infrastruttura centralizzata complessa come quella del DataCenter del framework.

Le classi che implementano i test sono contenute nei package `sfrc.application.client.test` e `sfrc.application.server.test`.

5.2 Test significativi

Il server di test utilizzato nelle prove descritte di seguito è il **TestServer002** ed è stato configurato per avere 3 secondi di ritardo tra la risposta ad una richiesta ed un'altra. Questa configurazione permette di osservare nel dettaglio la reazione dei meccanismi implementati ad un arrivo progressivo dei dati.

5.2.1 Test0006

Questo test ha l'obbiettivo di visualizzare all'interno di una finestra, tre oggetti che condividono una geometria dalla forma di fungo e su cui sono applicate delle texture. Sia la geometria che le texture sono generate proceduralmente durante l'esecuzione usando la capacità di calcolo della GPU.

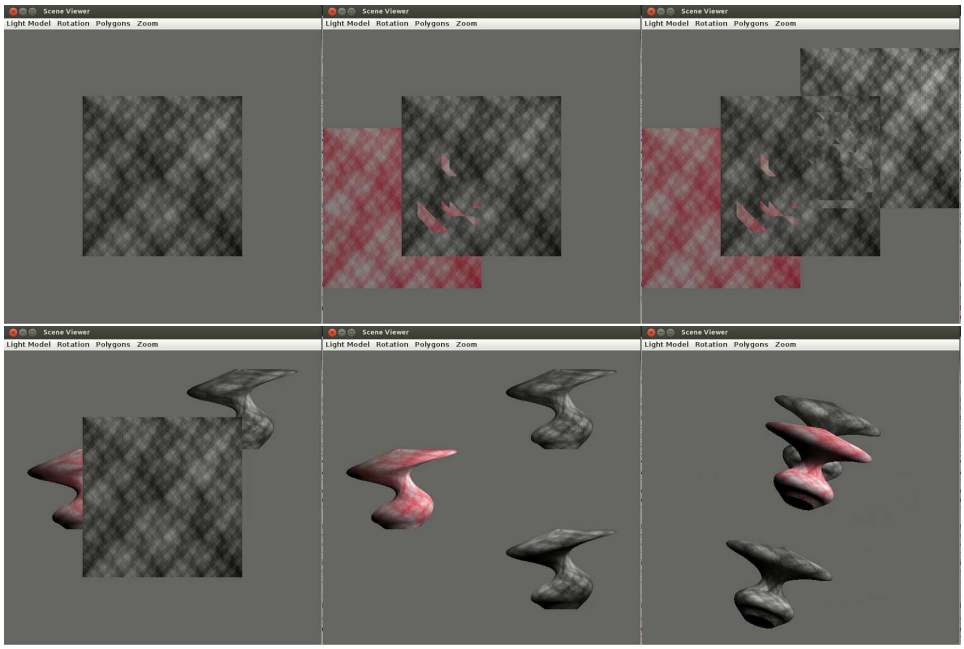

Figura 5.3: Fotogrammi estratti dall'esecuzione del test Test0006.

In figura 5.3 viene mostrata una sequenza di fotogrammi dell'esecuzione del test: al lancio dell'applicazione (fig.5.3 in alto a sinistra) la scena richiesta viene temporaneamente sostituita con un'altra che contiene un semplice cubo texturizzato. Al loro arrivo, i dati della scena reale che contengono i riferimenti ai tre oggetti vengono analizzati generando la richiesta degli oggetti verso il server. Nel frattempo ognuno di essi viene sostituito con un cubo posto al centro della scena. La sostituzione non genera un cambiamento visibile in quanto i tre cubi sono sovrapposti e posizionati esattamente come quello contenuto nella scena sostitutiva.

Quando i dati del primo oggetto vengono ricevuti (fig.5.3 in alto al centro) vengono immediatamente applicati alla scena: possiamo notare il cambiamento di colore e la traslazione di un cubo verso sinistra. L'oggetto non cambia forma perché il pacchetto di informazioni contiene un riferimento alla geometria "fungo" generica ancora non presente e per cui viene generata una richiesta. Nel fotogramma successivo (fig.5.3 in alto a destra) vediamo gli effetti dell'arrivo del pacchetto di informazioni relativo al secondo oggetto: un altro cubo viene traslato, stavolta verso destra. L'informazione sulla geometria "fungo" non è ancora arrivata, ma non viene generata una nuova

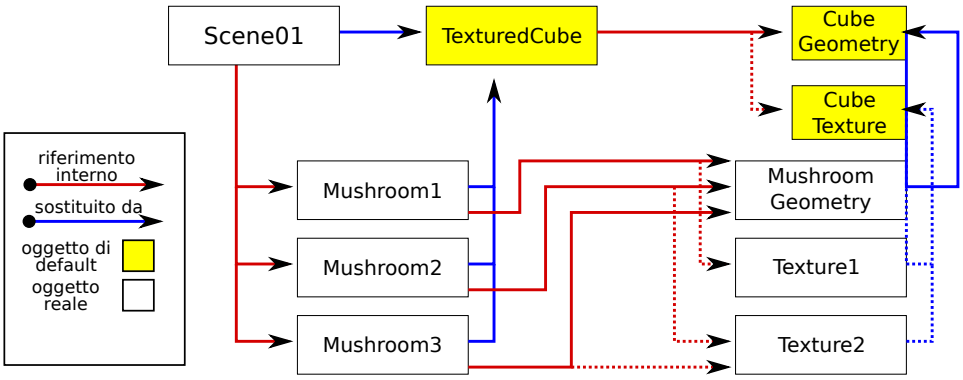

Figura 5.4: Schema delle relazioni interne tra i dati del Test0006.

richiesta dato che il sistema ha già una richiesta pendente, l'oggetto viene invece registrato per un update della geometria. Nel quarto fotogramma (fig.5.3 in basso a sinistra), viene ricevuta la geometria e tutti gli oggetti che si erano registrati per un update vengono aggiornati. Non avendo ancora ricevuto i dati relativi al terzo oggetto, esso rimane nella sua forma sostitutiva. Questo accade a causa del parallelismo dei processi che effettuano le richieste, il quale consente di non dover necessariamente ricevere i dati in un ordine specifico. Nella penultima immagine (fig.5.3 in basso al centro) si vede l'aggiornamento visivo all'arrivo delle informazioni sul terzo oggetto. Avendo già ricevuto i dati della geometria essa viene applicata immediatamente insieme alla traslazione. Nell'ultimo fotogramma vediamo la scena finale ruotata dai controlli della finestra client. Per effettuare la navigazione della scena non è più necessario richiedere altri dati al server e le connessioni vengono chiuse. In figura 5.4 sono rappresentate le relazioni tra i dati durante l'esecuzione del test.

Per non appesantire la precedente descrizione si è omesso che durante l'esecuzione sono stati richiesti e trasferiti anche i dati relativi alle texture da applicare ai modelli. La transizione non è direttamente visibile dato che le texture sostitutive hanno la stessa trama e non sono visivamente distinguibili.

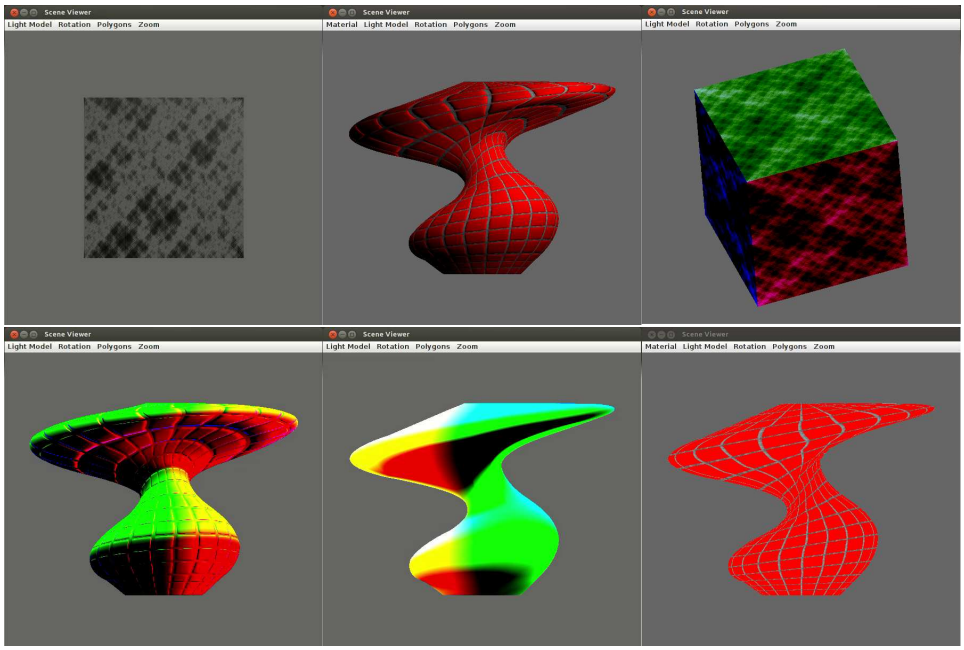

Figura 5.5: Fotogrammi estratti dall'esecuzione del test Test0019.

5.2.2 Test0019

Questo test prevede la visualizzazione di un oggetto avente geometria a fungo a cui viene applicata una texture e una *bump map*[1]. Anche in questo caso sia la geometria che le texture sono ottenute proceduralmente attraverso il rendering della GPU. Nei primi due fotogrammi in alto a sinistra della figura 5.5 vediamo le immagini della geometria sostitutiva (il cubo) e dell'oggetto reale della scena richiesta, anche in questo caso un fungo. L'obbiettivo del test è verificare che l'effetto di Bump Mapping venga applicato correttamente alla geometria del fungo una volta che essa viene ottenuta tramite la connessione. Nella seconda immagine si può infatti notare l'effetto "piastrellato" sulla superficie dell'oggetto. Sebbene dalla prima immagine non sia ben distinguibile, anche il cubo sostitutivo fa uso di questa tecnica: nel terzo fotogramma (fig.5.5 in alto a destra) esso viene disegnato in modo che ogni punto della sua superficie sia di un colore diverso in base alla

[1]Una bump map è un particolare tipo di texture che viene utilizzata tramite la tecnica del *Bump Mapping*. Questa tecnica si serve delle informazioni della bump map per descrivere la variazione da applicare al vettore normale di ogni punto di una superficie a cui è applicata. Ciò permette di ottenere l'effetto visivo di superfici irregolari senza spostare i vertici della superficie stessa

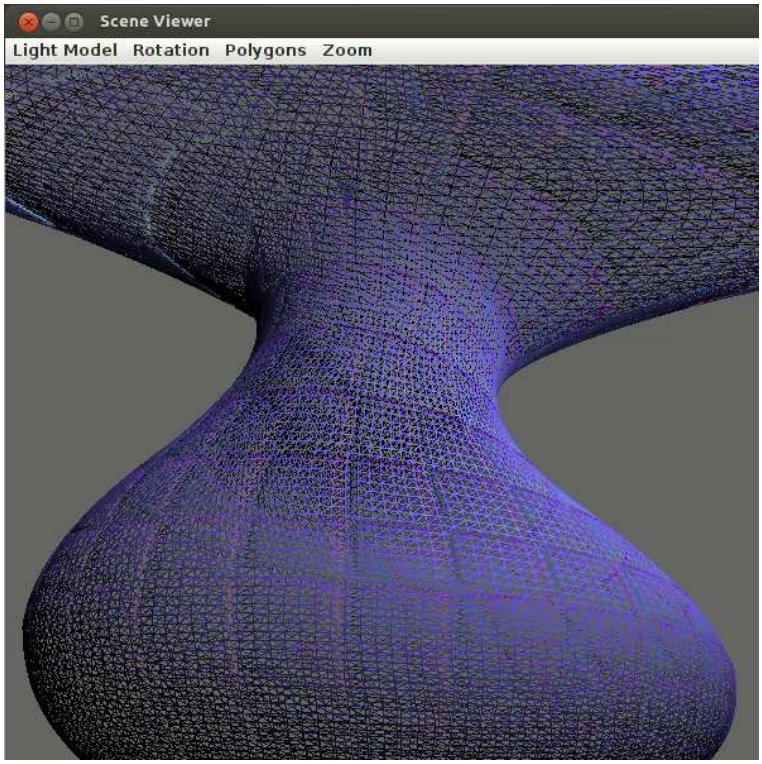

Figura 5.6: Particolare di un fotogramma estratto dall'esecuzione del test Test0019, è posta in evidenza la tassellazione in triangoli operata sul modello del fungo.

direzione del vettore normale al punto stesso. Le facce, pur essendo piatte, non hanno un colore uniforme mostrando la corretta applicazione della tecnica. Il quarto fotogramma (fig.5.5 in basso a sinistra) illustra lo stesso effetto applicato al fungo: a un colore diverso sulla superficie corrisponde una normale puntata in una diversa direzione. Gli ultimi due fotogrammi mostrano rispettivamente la geometria finale, colorata in base alla direzione della luce senza tener conto delle normali, e la geometria colorata applicando la texture in modo piatto senza illuminazione.

Come esplicato la tecnica applicata non effettua una reale traslazione dei vertici della geometria, in figura 5.6 vediamo messo in risalto questo particolare grazie ad uno zoom ravvicinato. Con questa immagine possiamo vedere anche il livello di tassellazione della geometria generata dalla pipeline programmabile.

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
    <SFDatasetReplacement name="BasicMatColours">
        <replacement>MatColours</replacement>
        <timestamp>1362586268761</timestamp>
    </SFDatasetReplacement>
    <SFDatasetReplacement name="Mushroom">
        <replacement>Cube</replacement>
        <timestamp>1362586268763</timestamp>
    </SFDatasetReplacement>
</library>
```

Figura 5.7: File xml della lista di sostituzione.

5.3 Attività correlate

La varietà dei contenuti dei test di riferimento ha reso necessario un lavoro aggiuntivo dedicato alla creazione di un insieme di Dataset sostitutivi da inserire nella libreria di oggetti di default, usati come rimpiazzo temporaneo. Questa libreria è stata realizzata progressivamente durante l'implementazione dei test, quando si rendeva necessaria la costruzione di un Dataset specifico e quelli già realizzati erano di tipo incompatibile. Queste incompatibilità dipendono da ciò che i Dataset rappresentano: geometrie, texture, etc.

Oltre a ciò è stato necessario implementare un metodo pratico per definire la lista di Dataset sostitutivi con cui effettuare gli scambi. La sola creazione della classe `SFDatasetReplacement` rendeva necessario definire a runtime la lista, che poteva essere memorizzata solamente in un file binario non leggibile ne modificabile senza un apposito tool. Per risolvere il problema è stato modificato il modulo di lettura/scrittura dei file xml preesistente nel framework, aggiungendo la capacità di codificare e decodificare i dati contenuti in un `SFDatasetReplacement`. La lista dei sostituti è così editabile a mano modificando un file testuale xml chiaramente leggibile (figura 5.7).

Capitolo 6

Conclusioni

L'obbiettivo del progetto di tesi è stato quello di progettare e produrre una serie di moduli software di supporto per lo sviluppo di applicazioni di grafica tridimensionale orientate al Web, che fanno uso dello Shadow Framework. I moduli prodotti si sono dimostrati efficaci allo scopo, consentendo di riprodurre i test effettuati con dati grafici in locale anche nella condizione in cui i dati sono memorizzati su di un server remoto. Per consentire la parallelizzazione delle richieste sono stati affrontati numerosi problemi di sincronizzazione per cui è stato di fondamentale importanza acquisire familiarità la programmazione multi-thread in Java.

Il lavoro di progettazione e sviluppo del codice è stato guidato dai principi di riutilizzo e modularità, sfruttando pratiche quali i *Design Pattern* e le metodologie di sviluppo agile.

6.1 Sviluppi futuri

I possibili sviluppi futuri del lavoro compiuto durante questa tesi sono molteplici e orientati in diverse direzioni. Innanzitutto vi è la possibilità di espandere i moduli **Client** e **Server** in modo che al loro interno vi siano degli strumenti per gestire in modo più completo le sessioni di comunicazione. Con questo si intende la possibilità di instaurare una comunicazione non legata strettamente alle richieste di dati, ma orientata allo scambio di messaggi di controllo, in modo di consentire l'interazione di utenti differenti che condividono la stessa simulazione.

Un'altra possibilità di sviluppo consiste nell'integrare nel modulo **Base Communication** lo sfruttamento di protocolli di comunicazione più complessi e potenti, come ad esempio il protocollo peer-to-peer sfruttato nella tesi [15].

La possibilità di editare via xml le liste di sostituzione dei dataset non eliminano il problema che la loro produzione sia lunga e laboriosa, soprattutto quando gli scenari diventano ricchi di modelli tridimensionali. Per questo motivo sarebbe necessario un tool in grado di esaminare i file che descrivono gli scenari e generare in automatico la lista.

È in fase di realizzazione un'estensione del protocollo di comunicazione che consenta al server di rispondere alle richieste con una libreria contenente un insieme di Dataset.

Oltre agli sviluppi direttamente collegati al progetto di tesi, i test effettuati hanno evidenziato una serie di elementi del framework che potrebbero avere la necessità di correzioni.

Appendice A

Note sul Software

A.1 Codice sorgente del progetto SF-Remote-Connection

Il progetto SF-Remote-Connection è una libreria di classi java ospitate, al momento della scrittura di questo documento, sul portale Google Code per la condivisione di codice open source all'indirizzo `http://code.google.com/p/sf-remote-connection/`. Il codice è attualmente rilasciato sotto licenza GNU GPL v3.

A.2 Versioni dei software utilizzati

A.2.1 Shadow Framework 2.0

La versione di riferimento del framework è disponibile all'indirizzo `http://code.google.com/p/shadowframework20lite/`. Il codice è attualmente rilasciato sotto licenza GNU GPL v3.

La revisione utilizzata è la: r201.

A.2.2 JOGL

JOGL è un binding Java open source e multipiattaforma per l'API grafica OpenGL. I suoi sorgenti e i binari sono disponibili sul sito `http://jogamp.org/jogl/www/`.

La versione utilizzata è la build: 2.0-b66-20121101.

A.2.3 Sviluppo del progetto

Nel corso del progetto sono stati usati questi strumenti software e linguaggi:

- **Eclipse**, IDE versione Indigo Service Release 2 [7];
- **SVN**, tool di versioning [13];
- **Git**, tool di versioning [1];
- **Java**, linguaggio di programmazione ad oggetti [5];

Utili alla comprensione degli argomenti trattati sono stati i seguenti testi tecnici [18, 14]

Appendice B

Design Pattern

Nel campo dell'Ingegneria del Software con Design Pattern si intende uno schema di progettazione del software utilizzato per risolvere un problema ricorrente. Molti di questi schemi sono stati pensati per il paradigma di programmazione ad oggetti e descrivono, utilizzando ereditarietà e interfacce, le interazioni e relazioni tra oggetti e classi.

In questa appendice vengono esposte alcune note generali sui design pattern citati nel testo, sia che siano stati implementati direttamente o semplicemente coinvolti nello sviluppo del progetto di tesi perché sfruttati dalle parti interessate del framework. Nella trattazione viene descritto l'obbiettivo di ogni schema, l'utilità, note implementative con schema Unified Modeling Language (UML) annesso e alcuni benefici della sua applicazione. Vengono inoltre raggruppati per categoria così come vengono presentati in [6].

B.1 Pattern Creazionali

I pattern creazionali sono usati per creare un'astrazione sul processo di instanziazione degli oggetti, in modo da rendere il sistema indipendente da come gli oggetti vengono effettivamente creati. Questo tipo di pattern diviene importante quando un sistema dipende principalmente da oggetti composti da aggregati di componenti più piccole, rispetto ad oggetti gerarchici organizzati in base all'ereditarietà.

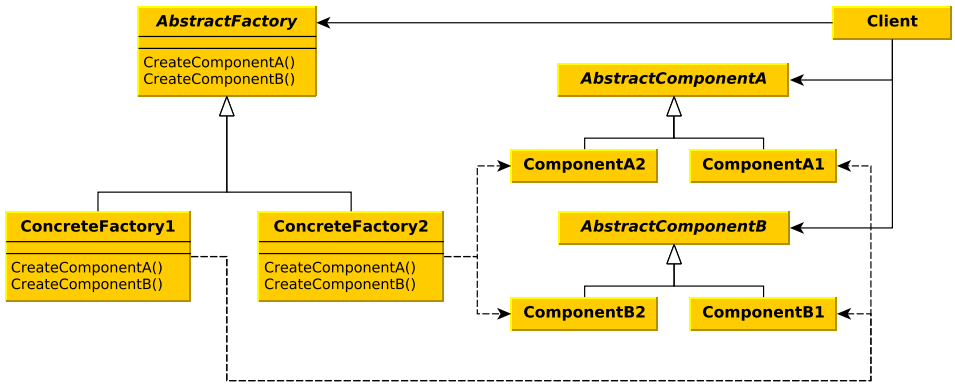

Figura B.1: Struttura UML del pattern Abstract Factory.

B.1.1 Abstract Factory

L'obbiettivo di questo pattern è fornire un'interfaccia per la creazione di famiglie di oggetti imparentati o dipendenti tra loro, eliminando la necessità di specificare i nomi delle classi concrete all'interno del codice.

In questo modo un sistema può essere indipendente da come vengono creati gli oggetti concreti e può essere configurato per utilizzare diverse famiglie di oggetti a seconda delle necessità. Un caso esemplificativo è quello di un tool per l'implementazione di interfacce grafiche che supporta diversi *look-and-feel*. L'utilizzo di un diverso look-and-feel comporta la definizione di una nuova famiglia di componenti grafiche le cui caratteristiche base sono però fondamentalmente identiche: per esempio un pulsante può essere premuto e disegnato a schermo. L'applicazione dovrebbe essere indipendente dalle modalità con cui il pulsante viene istanziato, ed essere in grado di riconoscerlo per le sue funzionalità.

Per ottenere ciò è possibile definire una classe astratta o un'interfaccia differente per ogni componente generico desiderato, come `AbstractComponentA` e `AbstractComponentB` nella figura B.1. Si definisce poi una classe `AbstractFactory` astratta la cui funzione sia quella di creare istanze delle componenti astratte generiche. Se per ogni famiglia che si desidera implementare si generano sottoclassi concrete per ogni componente astratta e si definisce una Factory concreta in grado di istanziarle, come `ComponentA1` e `ConcreteFactory1` nell'esempio raffigurato, l'applicazione potrà utilizzare la Factory concreta che preferisce in modo del tutto indipendente da quale famiglia essa gestisce.

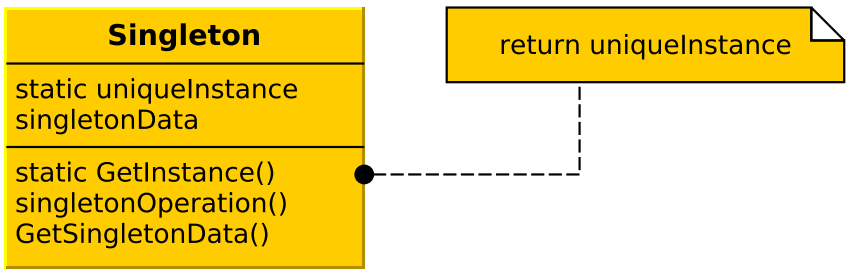

Figura B.2: Struttura UML del pattern Singleton.

Questo schema permette di cambiare le famiglie di componenti in modo semplice e consente di isolare le classi concrete, poiché le componenti vengono utilizzate attraverso la loro interfaccia.

B.1.2 Singleton

Lo scopo di questo pattern è garantire che di una determinata classe possa essere creata una unica istanza, fornendone un singolo punto di accesso globale.

L'utilità di questo pattern risiede nel fatto che spesso, all'interno di un'applicazione, ci sono servizi e funzionalità condivise a cui deve essere associata una istanza univoca. Esempi classici in proposito sono il gestore del File System o il gestore dei servizi di stampa.

Per ottenere questa proprietà spesso si delega alla classe stessa la responsabilità di controllare la creazione delle propria istanza rendendo privato il suo costruttore, impedendo di fatto l'istanziazione dall'esterno. In figura B.2 possiamo vedere lo schema UML di una classe che utilizza il pattern nella quale vi è un'unica istanza statica `uniqueIstance` accessibile solamente attraverso il metodo statico `GetInstance()`.

Utilizzare questo schema consente di avere un più stretto controllo degli accessi alla risorsa, inoltre estendendo la classe è possibile raffinare le funzionalità e configurare l'applicazione a runtime per utilizzare l'istanza adatta ad ogni caso specifico.

B.2 Pattern Strutturali

Questo tipo di pattern si occupa di come classi e oggetti vengono composte e interagiscono nella formazione di strutture più complesse. Un obbiettivo di questi pattern è permettere la composizione di oggetti complessi che uniscano le funzionalità di moduli più piccoli rendendo quest'ultimi il più possibile riutilizzabili.

B.2.1 Bridge

Questo pattern ha l'intento di disaccoppiare un'astrazione dalla sua implementazione in modo che entrambe possano evolversi in maniera indipendente.

Ciò è particolarmente utile quando si desidera evitare un legame permanente tra astrazione e implementazione, e si vuole consentire la scelta o la modifica di essa durante l'esecuzione. É utile inoltre quando sia astrazione che implementazione hanno la necessità di essere estendibili attraverso sottoclassi.

Un utile esempio per illustrare questo pattern è prendere in esame un ipotetico toolkit per interfacce grafiche. Per renderlo il più possibile portabile su piattaforme diverse esso deve utilizzare un'astrazione che descriva le finestre in modo più generico possibile. Se per ottenere queste proprietà usassimo semplicemente una classe astratta `Window` da cui, usando l'ereditarietà, costruire sottoclassi specifiche per ogni sistema da supportare, otterremmo due svantaggi:

1. In previsione di voler estendere la classe `Window` per coprire l'astrazione di altri tipi di finestra grafica esistente, per poter supportare tutte le piattaforme per cui esisteva una implementazione di `Window` dovremmo creare una sottoclasse specifica per ognuna di esse.

2. Il codice del client diventa dipendente dalla piattaforma. Quando vi è la necessità di creare una finestra, deve essere istanziata una classe concreta specifica e questo lega fortemente l'astrazione con l'implementazione utilizzata rendendo più complessa la portabilità su altre piattaforme.

Per eludere questi problemi il pattern Bridge separa la gerarchia delle classi che appartengono all'astrazione da quella delle classi di implementazione. In

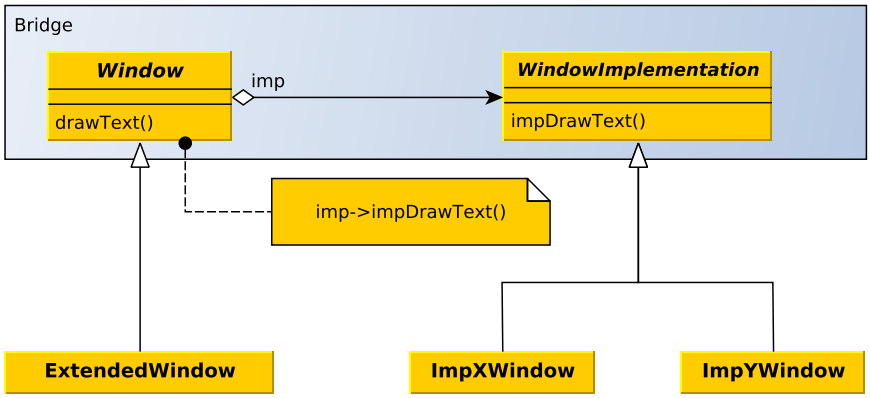

Figura B.3: Esempio di schema gerarchico dovuto all'applicazione del pattern Bridge.

cima alle due gerarchie vi sono le classi `Window` e `WindowImplementation`, che compongono il Bridge vero e proprio. La prima rappresenta l'astrazione della finestra che verrà utilizzata dal client, la seconda è invece l'interfaccia che un'implementazione concreta deve utilizzare per essere gestita dall'astrazione. Come si può vedere nella figura B.3, la classe `Window` rimappa i propri metodi su quelli dell'interfaccia `WindowImplementation`, o con una combinazione di essi. Questi vengono chiamati su di una implementazione concreta di `WindowImplementation` di cui `Window` possiede un riferimento `imp`.

L'utilizzo di questo schema permette di nascondere i dettagli implementativi ai client e di non avere effetti diretti su di esse quando l'implementazione cambia in modo che non sia necessario ricompilare il codice dei client.

B.2.2 Composite

Lo scopo di questo pattern è consentire una gestione uniforme tra oggetti semplici ed oggetti compositi. Questo si traduce in una organizzazione degli oggetti in una struttura ad albero in cui ogni nodo è un oggetto aggregato e ogni foglia è un oggetto semplice. Il nodo composito avrà al suo interno riferimenti ad oggetti figli che a loro volta possono essere semplici o compositi.

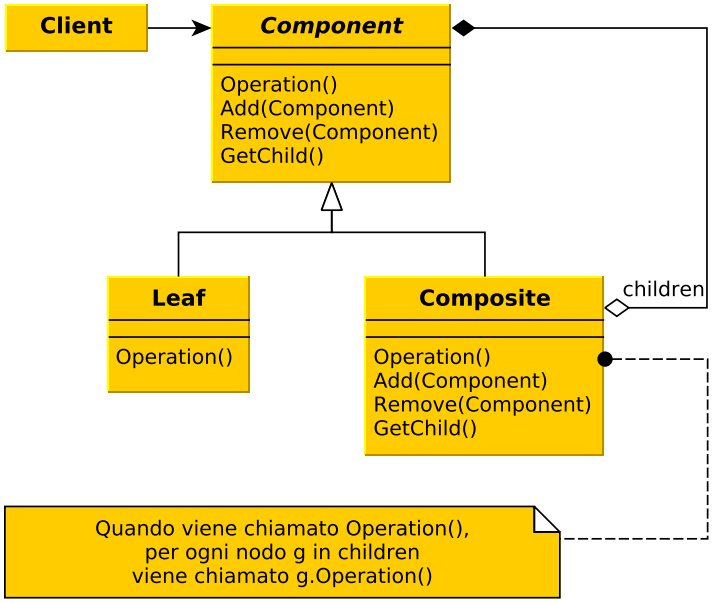

Figura B.4: Struttura UML del pattern Composite.

Spesso si desidera poter trattare oggetti composti nello stesso modo in cui gestiamo le sue componenti, ignorando come siano effettivamente implementate le loro funzionalità. Un esempio significativo sono le interfacce grafiche in cui è semplice poter gestire gruppi di componenti, come pulsanti o etichette, in modo unitario come se si trattassero di una componente singola.

Una metodologia per ottenere questo tipo di interazione è l'utilizzo di una classe astratta che rappresenti sia gli oggetti semplici, o primitive, sia gli oggetti composti, o contenitori. Nello schema UML in figura B.4, la classe astratta `Component` rappresenta sia oggetti foglia `Leaf` che nodi `Composite`. Un client può richiamare indifferentemente i metodi di interfaccia di un oggetto sottoclasse di `Component` sia che si tratti di un una foglia o di un nodo. Nel caso di una foglia il metodo risponderà direttamente in maniera appropriata mentre nel caso di un nodo verrà richiamato lo stesso metodo per tutti gli oggetti figli.

L'utilizzo di questo pattern consente di rendere meno complessi i moduli che utilizzano le strutture composite, che risultano svincolati dal conoscere se stiano trattando una foglia o un nodo dell'albero. Permette inoltre di

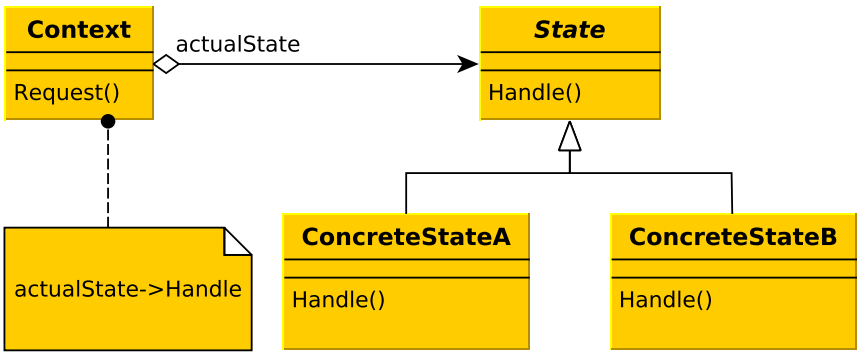

Figura B.5: Schema UML del pattern State.

aggiungere in modo semplice nuovi componenti senza laboriosi interventi sul codice preesistente.

B.3 Pattern Comportamentali

I pattern comportamentali riguardano gli algoritmi e la suddivisione delle responsabilità tra gli oggetti. Per fare ciò non si limitano solamente a descrivere i rapporti tra oggetti e classi, ma anche utilizzando schemi di comunicazione tra di essi. Questi schemi permettono di descrivere complessi flussi di controllo spostando l'attenzione sulle connessioni fra gli oggetti.

B.3.1 State

Il pattern State ha come scopo quello di consentire ad un oggetto di cambiare il proprio comportamento durante l'esecuzione, in base alle variazioni del proprio stato interno.

Esempi possono essere trovati in oggetti che gestiscono connessioni e che devono rispondere in modo diverso in base allo stato della connessione stessa, oppure nella costruzione di macchine a stati che implementano algoritmi di controllo.

La figura B.5 mostra in uno schema UML la struttura del pattern. La classe `Context` rappresenta l'interfaccia che l'oggetto a stato variabile presenta verso i client. Al suo interno mantiene una istanza (`actualState`) di una sottoclasse concreta della classe `State`. La classe astratta `State` defini-

sce un'interfaccia per incapsulare il comportamento associato ad uno stato. Le sue sottoclassi concrete sono associate ad un effettivo stato e implementano il comportamento che l'oggetto `Context` assume in quel caso. Quando i client effettuano richieste all'oggetto `Context` attraverso la sua interfaccia, viene richiamato il metodo di interfaccia dell'istanza concreta `actualState`. Il pattern non definisce dove venga inserita la logica di cambiamento di stato, ma in generale è più flessibile consentire alle sottoclassi di `State` la definizione dello stato successivo.

L'applicazione del pattern consente di eliminare lunghe e complesse parti di codice in cui una sequenza di istruzioni condizionali determinano le istruzioni da eseguire in base allo stato dell'oggetto. Inoltre isola il codice specifico relativo ad uno stato in un singolo oggetto permettendo l'aggiunta di nuovi stati e transizioni semplicemente definendo nuove classi.

Bibliografia

[1] Git - documentation.
`http://git-scm.com/documentation` [Online - controllata il 14-aprile-2013].

[2] Adobe. How stage3d works.
`http://www.adobe.com/devnet/flashplayer/articles/how-stage3d-works.html` [Online; controllata il 1-aprile-2013].

[3] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. A K Peters, Ltd., 3rd edition, 2008.

[4] UNIGINE Corp. Unigine's crypt demo is ported into a browser, 2013.
`http://unigine.com/news/2013/02/20/crypt-in-browser` [Online; controllata il 1-aprile-2013].

[5] Harvey M. Deitel and Paul J. Deitel. *Programmazione Java Tecniche avanzate*. 7th edition, 2008.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley Professional, 1994.

[7] Steve Holzner. *Eclipse*. O'Reilly Media, 2004.

[8] Gaikai Inc. `http://www.gaikai.com`.
[Online - controllata il 5-aprile-2013].

[9] Kurose J. and Ross K. *Computer Networking - A Top-Down Approach*. Addison Wesley, 6th edition, 2012.

[10] Alessandro Martinelli. Lo shadowframework, 2013.
`http://www.shadowframework.com/page/projects/`

shadowframework/mainpage&menu=shadowframework [Online; controllata il 7-aprile-2013].

[11] Mozilla. Mozilla is unlocking the power of the web as a platform for gaming, 2013. https://blog.mozilla.org/blog/2013/03/27/mozilla-is-unlocking-the-power-of-the-web-as-a-platform-for-gaming/ [Online; controllata il 1-aprile-2013].

[12] Onlive. http://www.onlive.com/. [Online - controllata il 5-aprile-2013].

[13] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion, Next Generation Open Source Version Control*. O'Reilly Media, 2nd edition, 2008.

[14] Riccardo Scateni, Paolo Cignoni, Claudio Montani, and Roberto Scopigno. *Fondamenti di grafica tridimensionale interattiva*. McGraw Hill.

[15] Davide Truzzi. Sviluppo di un'applicazione peer-to-peer soft real-time per la condivisione di esperienze virtuali 3d. Master's thesis, Università di Pavia, 2011.

[16] Wikipedia. Rendering — wikipedia, l'enciclopedia libera, 2013. http://it.wikipedia.org/w/index.php?title=Rendering&oldid=57850976 [Online; controllata il 1-aprile-2013].

[17] Wikipedia. Rendering — wikipedia, the free encyclopedia, 2013. http://en.wikipedia.org/w/index.php?title=Rendering_(computer_graphics)&oldid=542051916 [Online; accessed 1-April-2013].

[18] Richard S. Wright, Nicholas Haemel, Graham Sellers, and Benjamin Lipchak. *OpenGL SuperBible*. Addison Wesley, 2010.