Cupido

Relazione del progetto per il
corso di Architetture Software 2010/2011

Autori:

Lorenzo Belli	Mat. 602031
Marco Poletti	Mat.
Federico Viscomi	Mat. 577326

Introduzione

Questo documento contiene la descrizione di tutte le operazioni eseguite dal nostro gruppo per creare il progetto del corso Architetture software 2010/2011 presso l'università di Bologna. Il progetto realizzato è un applicazione web denominata Cupido.

Fase preliminare

La prima fase della realizzazione è stata una analisi preliminare delle richieste del committente disponibili a [1]; al termine di questa fase abbiamo individuato come caratteristica rilevante che i requisiti del software saranno molto stabili perché le richieste del committente non cambieranno in modo sensibile durante lo sviluppo.

Scelta del processo di sviluppo

In seguito si è scelto il processo di sviluppo da adottare per la realizzazione del software. Sono stati considerati vari processi di sviluppo, e per ognuno di essi sono stati considerati i pro e i contro in relazione all'analisi preliminare.

- **Processo a cascata**

Un processo che prevede analisi dei requisiti, progettazione, codifica, e testing tutti in successione.

  - Pro: è semplice da seguire ed effettuare.
  - Contro: è molto dispendioso correggere errori effettuati nelle prime fasi dello sviluppo.

Non utilizzeremo questo modello poiché prevediamo di inserire involontariamente errori nella fase di progettazione, e risulterebbe poi troppo costoso rimediare ad essi, in termini di tempo.

- **Processo iterativo-incrementale**

Processo che consiste di varie iterazioni, ognuna comprendente una fase di analisi, una di progettazione, una di implementazione ed una di testing. Ogni iterazione ha come risultato finale un prodotto testato con un numero di funzionalità aumentato rispetto all'iterazione precedente.

  - Pro: è più semplice risolvere errori introdotti nelle fasi iniziali del processo; il testing è introdotto in fasi precedenti rispetto al modello a cascata.
  - Contro: e più complesso pianificare il lavoro.

Decidiamo di non adottare questo processo poiché dall'analisi preliminare si evince che il nostro software avrà requisiti molto stabili, e risulterebbe inutilmente oneroso suddividere le attività di analisi dei requisiti e design.

- **Processo a Spirale**

Processo di tipo iterativo in cui ogni iterazione comprende analisi, design, sviluppo e testing; si sfrutta la pianificazione e l'analisi del rischio per monitorare l'andamento del processo.

Scegliamo di non adottare questo processo poiché in relazione all'analisi preliminare del software riteniamo poco rilevante la pianificazione e l'analisi del rischio ad ogni iterazione.

- **Processo a consegne programmate**

Processo con fasi iniziali di Analisi dei requisiti e progettazione, seguito da varie iterazioni di implementazione e testing. Viene programmato in anticipo il lavoro da svolgere in ogni iterazione.

    - Pro: il testing ad ogni iterazione permette di rilevare in anticipo eventuali errori. L'analisi ed il design non sono ripetuti in più fasi.

Non adotteremo questo processo perché non ci riteniamo in grado di programmare adeguatamente il lavoro da svolgere nelle varie iterazioni.

- **Processi Agili**

Scegliamo di non utilizzare questo tipo di processi poiché sono pensati principalmente per adattarsi ai cambiamenti dei requisiti; avendo previsto che il nostro software abbia requisiti molto stabili risulterebbe troppo costoso adottare un processo agile.

- **V -Model**

Processo a cascata, in cui a lavoro ultimato viene testato il risultato di ogni fase in ordine inverso.

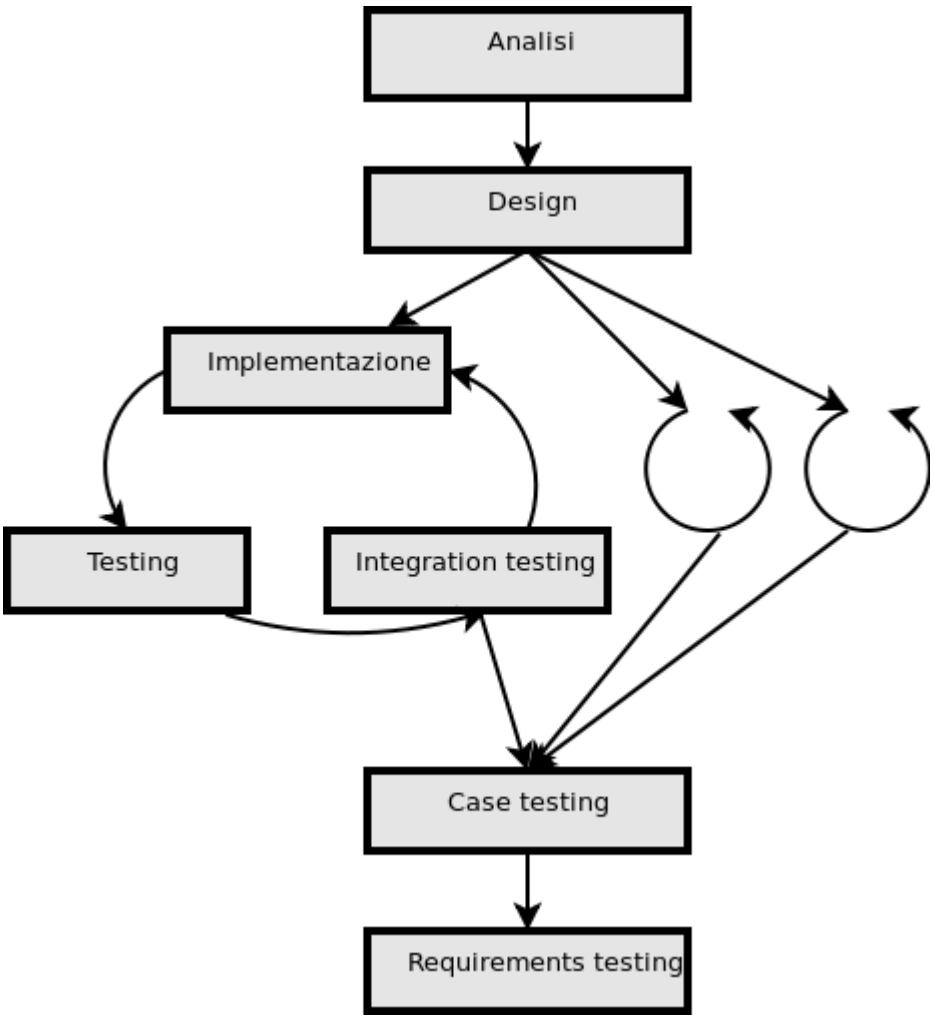
Non adottiamo questo processo poiché gli errori introdotti nella fase di analisi e progettazione vengono rilevati solo alla fine del processo, quando risulterebbe troppo oneroso correggerli.

- **RUP**
    - Pro: enfasi su una documentazione accurata; risolve in modo pro-attivo i problemi legati al rischio di progetto; alto riuso di componenti.
    - Contro: prevede che il team sia competente nel dominio del software.

Non adottiamo questo processo poiché presuppone che il team sia esperto nel dominio del software da realizzare.

Descrizione del processo di sviluppo scelto

In seguito alle considerazioni precedenti scegliamo di adottare un nostro processo di sviluppo che descriviamo qui di seguito.



Analizziamo le fasi singolarmente:

- **Analisi**

Durante questa fase saranno specificati tutti i requisiti che il software deve soddisfare. Questa fase produrrà un documento Specifica dei Requisiti Software visibile come allegato nel file SRS.pdf.

- **Design**

In questa fase sarà definita prima l'architettura del sistema. Successivamente verrà definito un design ad alto livello di dettaglio tale da permettere lo sviluppo nelle fasi successive. In particolare saranno definite le interfacce tra i componenti del software.
Tutto il gruppo deve partecipare attivamente in questa fase.
Se in questa fase emergeranno errori relativi all'analisi dei requisiti, modificheremo immediatamente i requisiti per poi modificare di conseguenza il design. Questa fase produrrà un documento Architettura e Design visibile come allegato nel file Architettura_e_Design.pdf.

- **Iterazione di sviluppo**

Il software verrà partizionato in 3 parti ognuna della quali sarà sviluppata da un diverso team di sviluppo; nel nostro caso ogni team è composto da una sola persona. Ogni team effettuerà delle iterazioni fino a raggiungere la completa implementazione delle funzionalità che lo riguardano.
Le iterazioni non sono pianificate né in senso temporale, né come lavoro da svolgere.
Ogni sviluppatore è libero di decidere la documentazione da produrre sulla parte di software da lui implementata.

Ad ogni iterazione lo sviluppatore esegue tre fasi:

- **Implementazione**

Lo sviluppatore implementa un sottoinsieme della funzionalità che deve sviluppare. Esso è libero di effettuare scelte implementative non vincolate dalla fase di design, ma nel rispetto dei requisiti individuati nella fase di analisi. In questa fase viene anche decisa la struttura delle classi di ogni modulo a discrezione dello sviluppatore.

- **Testing**

Lo sviluppatore testa le funzionalità aggiunte nella fase precedente; esso non ha vincoli su come effettuare questo test.

- **Integration Testing**

Lo sviluppatore integra ciò che ha creato nell'iterazione corrente con la parte di software che è già stata testata nelle iterazioni precedenti da lui e dagli altri team di sviluppo. Inoltre effettua un test per verificare che le interazioni fra i componenti dopo l'integrazione siano corrette. La scelta di questo tipo di test è lasciata agli sviluppatori.
La documentazione da produrre in questa fase è lasciata al libero arbitrio degli sviluppatori.
Quando durante un iterazione un team identifica errori o mancanze derivanti dalle fasi di analisi o design, tutto il gruppo si riunisce per risolvere il problema; ove occorre saranno immediatamente modificati la Specifica dei Requisiti Software, l'architettura ed il design. La correzione del problema sarà poi affidata ad uno o più dei tre team di sviluppo.

- **Case testing**

Dopo che tutti i team di sviluppo hanno terminato l'implementazione, vengono descritti ed eseguiti i test case che il software deve superare per passare alla fase successiva. Quando saranno rilevati errori, il team che ha sviluppato la parte interessata avrà il compito di risolverli.

- **Requirements Testing**

Dopo che tutti i test case sono stati superati correttamente, si verificherà che il software rispetti tutte le caratteristiche descritti nella Specifica dei Requisiti Software, comprese eventuali modifiche ai requisiti introdotti nelle fasi di sviluppo. In caso di problemi questi saranno risolti dal team di sviluppo designato, ed il software dovrà ripetere le fasi di Case Testing e Requirements Testing.
Il software che sia approvato da questa fase, sarà considerato completato.

<u>Vantaggi del processo scelto</u>:

- Tutto il gruppo parteciperà alla fase di design, dunque ognuno di noi sarà consapevole del design dell'intero sistema.
- Ogni membro del gruppo dovrà conoscere l'implementazione solo della parte di software di sua competenza.
- Le iterazioni non pianificate permettono un approccio più adattativo agli imprevisti dei singoli team di sviluppo, inoltre un approccio non pianificato in fasi di sviluppo permette di mitigare gli effetti derivanti dall'individuazione di errori risolvendoli immediatamente.
- Non è necessario che i membri del gruppo siano esperti in un particolare dominio software.
- I Test Case sono sviluppati dopo l'implementazione, quando la conoscenza del software da parte del gruppo sarà migliore; questo porterà a Test Case più accurati e precisi.
- Le fasi di Analisi e Design sono uniche e non ripetute, e questo è coerente con la stabilità dei

requisiti prevista dall'analisi preliminare.

Svantaggi:

- Gli errori introdotti in fasi di Analisi e Design possono essere identificati molto tardi nello sviluppo del software, e la loro correzione richiede la presenza di tutto il gruppo.

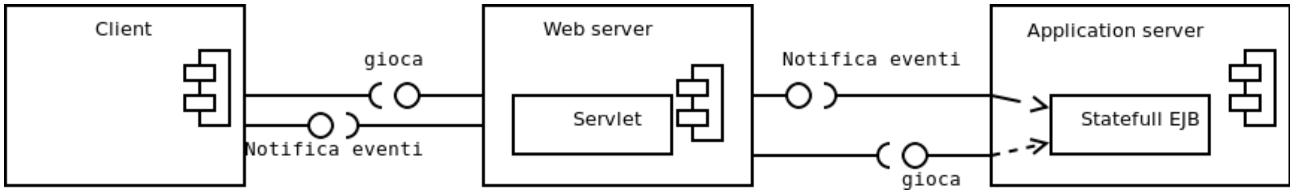
Dopo aver scelto il processo di sviluppo lo abbiamo eseguito.

La fase di Analisi ha come risultato il file SRS.pdf . Abbiamo creato i requisiti software in base ai requisiti posti dal docente ed alle osservazioni dei membri del gruppo.

La fase di Design ha dato come risultato il file architettura_e_design.pdf che descrive la sola architettura scelta.

[TODO architetture scartate. Cosa inserire qui????]
Riportiamo di seguito le operazioni che abbiamo eseguito per arrivare a definire l'architettura software.

Abbiamo subito ipotizzato un architettura basata su Java EE con utilizzo di EJB per implementare la logica di gioco. L'uso di EJB permetterebbe di non dover implementare alcune caratteristiche come la gestione della concorrenza.



In questo caso una partita dovrebbe essere gestita da un Stateful Session Bean, perché lo stato della partita deve essere sempre disponibile. Tuttavia lo stato di una partita è determinato dall'interazione di più utenti che interagiscono con il Session EJB tramite servlet; questa architettura è scorretta poiché un Stateful Session Bean non può essere condiviso tra più client.

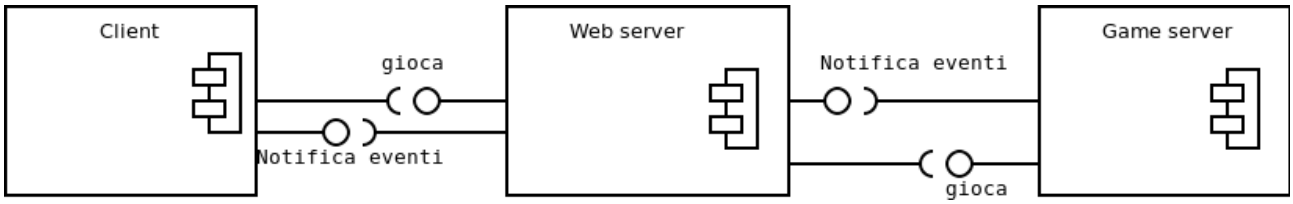
Una seconda possibilità per utilizzare EJB è di creare un componente Table che gestisce una singola partita, ed accedervi tramite Stateless EJB. In questo modo però bisognerebbe specificare ad ogni invocazione a quale partita si sta partecipando. Questo però introdurrebbe un livello di astrazione poco utile, perché una servlet potrebbe comunicare direttamente con una partita.

Decidiamo allora di distogliere lo sguardo da EJB perché non ci interessano particolarmente le caratteristiche di transazionalità e sicurezza.

Un altra opzione valutata è di utilizzare una tecnologia Web Service per implementare la comunicazione tra servlet e Application Server. Questa opzione non è valida data la natura stateless dell'interazione.

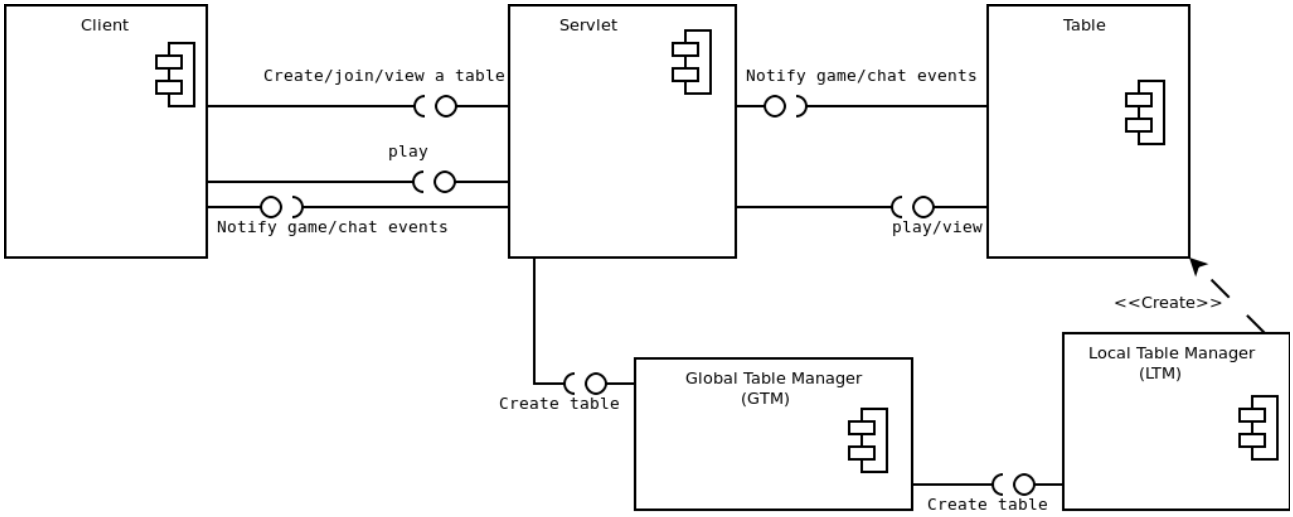
Decidiamo ora di progettare una nostra architettura partendo da zero.

Abbiamo previsto di dividere il software in tre componenti principali: un backend che gestisce la logica di gioco, un client eseguito dall'utente e da un web server che gestisce la comunicazione tra i due tramite servlet.

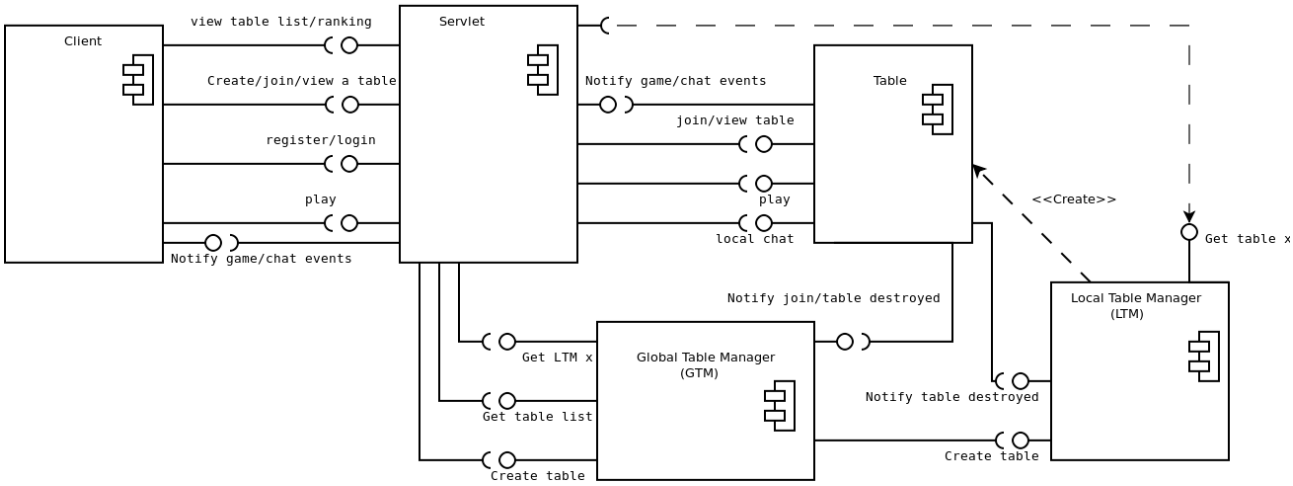


Qui ogni web server gestisce più servlet che gestiscono più client. Tutte le servlet comunicano con

un unico game server.
Avere un unico Game server può essere un collo di bottiglia, quindi per sopperire alle richieste di scalabilità decidiamo di duplicare i game server su più macchine fisiche

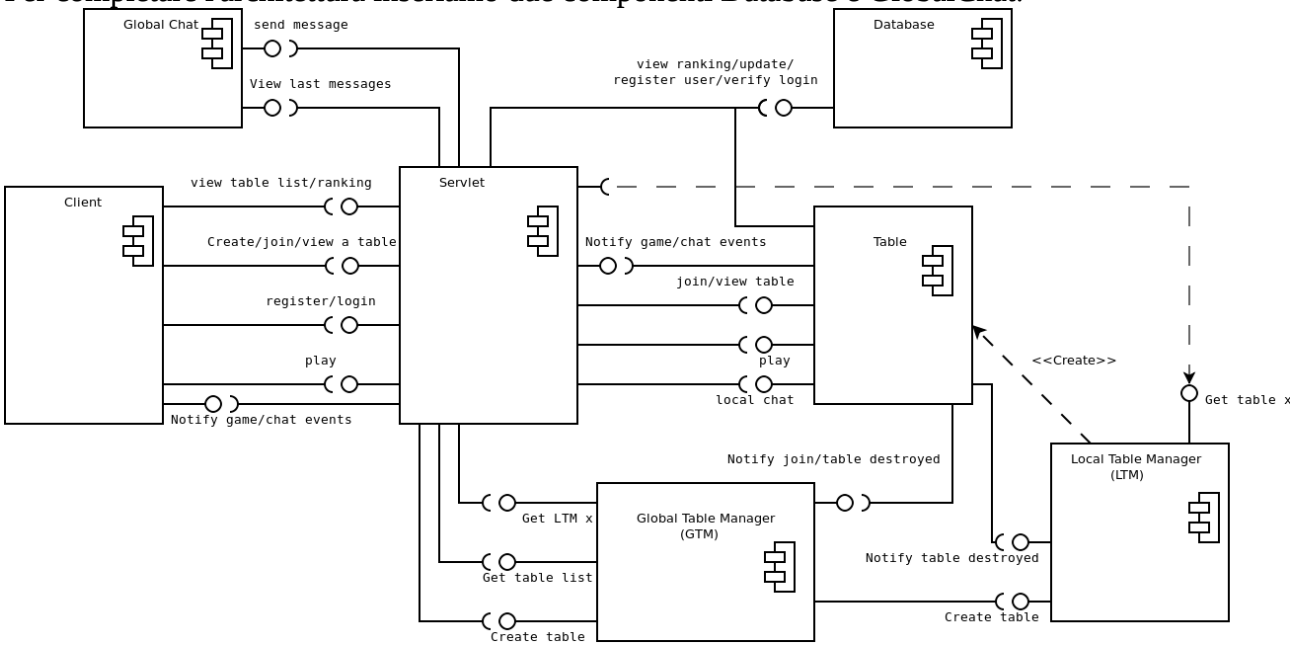


Ogni game server (ora chiamati Local Table Manager o LTM) gestisce un certo numero di componenti Table, ognuno dei quali gestisce una partita.
Global Table Manager ha il compito di suddividere il lavoro tra gli LTM: una servlet che voglia creare un tavolo deve richiedere l'interfaccia remota del Table creato a GTM, che lo creerà in un opportuno LTM secondo politiche interne. GTM deve quindi ricevere notifiche Dai LTM sul loro stato, ed in particolare sul loro carico di lavoro.
Sempre per sopperire alle richieste di scalabilità è previsto che anche i Web server siano replicabili. Il carico di lavoro su di essi può essere smistato con un DNS Name Server (non mostrato in figura).



Per richiedere la lista delle partite in corso, Servlet la richiede a GTM, che dovrà quindi mantenere la lista di tutti i Table con lo stato di ogni partita. Per mantenere aggiornato lo stato di GTM, ogni Table notifica un join o la distruzione di un tavolo attraverso l'interfaccia 'Notify join/table destroyed'. Per entrare in una partita (azione di join) o di entrare come spettatore (view) la servlet deve ottenere da GTM una reference al giusto LTM attraverso 'Get LTM x'. Similmente richiede il giusto tavolo al LTM attraverso l'interfaccia 'Get table x'. Questi due identificatori vengono restituiti con la lista delle partite in corso da 'Get Table List' di GTM.

Per completare l'architettura inseriamo due componenti Database e GlobalChat.



Database contiene gli utenti registrati ed il loro punteggio. Database è condiviso tra tutti i Table e tutte le Servlet. Servlet accede al Database per le operazioni di registrazione e login. Table accede al Database per aggiornare i punteggi dei giocatori a fine partita.
Global chat è un componente condiviso da tutte le servlet, ed è usato per la comunicazione nella chat globale.

In questa fase abbiamo anche deciso tutte le interfacce tra i moduli del sistema. Per vedere il lavoro svolto rimandiamo al file architettura_e_design.pdf ed al codice sorgente, in particolare cupidoCommon/src/unibo/as/cupido/common/interfaces/ e cupidoGWT/src/unibo/as/cupido/client/CupidoInterface.java .

Nella fase di sviluppo ogni membro del gruppo ha proceduto singolarmente a sviluppare.
Durante questa fase, soprattutto nelle prime iterazioni è stato necessario modificare più volte le interfacce. Ognuna di queste modifiche è stata apportata dopo averla discussa con tutto il gruppo.

Nella fase di Case Testing sono stati creati dei test automatici per testare il backend.
Questi test prevedono di usare dei bot più un quarto bot che simula un utente, e sono stati così progettati:

1. Gioco con un giocatore umano e 3 bot
2. Gioco con due umani e 2 bot
3. Entrare in modalità spettatore ad un gioco con un umano e tre bot
4. Gioco con un umano, tre bot ed uno spettatore. Dopo un turno il creatore del tavolo lascia la partita.
5. Gioco con due umani e due bot. Dopo un turno il giocatore non creatore lascia la partita.

Questi test sono stati effettuati anche in modo non automatico da utenti umani, per testare il client e la servlet.

In questa fase sono emersi vari errori, alcuni bug di semplice soluzione e problemi di concorrenza tra i bot. Questo ha portato in particolare ad un nuovo design e ad una nuova implementazione dei bot, per risolvere i problemi di concorrenza.

[TODO parlare meglio dei test fatti, dei risultati ottenuti, e di come sono stati corretti gli errori. Oppure vanno bene così? Sono forse pochi i test?]

Nella fase di Requirements Testing sono stati verificati tutti requisiti identificati nella fase di analisi e marcati come non opzionali. Solo un numero molto limitato di requisiti non era stato rispettato, e le modifiche richieste al codice sono state esigue.

Riferimenti

[1] http://courses.web.cs.unibo.it/cgi-bin/twiki/bin/viewauth/ArchitettureSoftware/ProgettoDelCorso v0,1