Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

CrownSouls

CrownSouls

Progetto di Programmazione ad Oggetti

Enrico Buratto
1142644

Anno Accademico 2019-2020

Indice

1 Introduzione

1.1 Abstract

Si vuole realizzare un programma per la gestione di un inventario giocatore per il gioco *Action RPG* "Dark Souls". Il giocatore possiede svariati elementi di diversi tipi; questi elementi possono infatti essere:

Armature: oggetti indossati dal giocatore per aumentare la resistenza al danno inflitto dai nemici;

Armi: oggetti utilizzati dal giocatore per infliggere danno ai nemici;

Anelli: oggetti utili al giocatore per l'aumento delle proprie statistiche; una volta indossati nel gioco, il giocatore vedrà aumentate alcune statistiche personali o delle armi;

Scudi: oggetti utilizzati dal giocatore per ridurre il danno dai colpi nemici;

Guanti: oggetti utilizzati sia come armatura, poiché aumentano la resistenza al danno, sia come arma, poiché permettono di infliggere danno;

Scudi d'attacco: oggetti utilizzati sia come scudo, poiché riducono il danno dai colpi nemici, sia come arma, poiché permettono di infliggere danno.

Un inventario è composto da un insieme di oggetti appartenenti alle diverse tipologie; ogni oggetto presente nell'inventario possiede delle caratteristiche tecniche proprie della categoria di appartenenza. Il programma deve poter simulare un inventario di questo tipo, permettendo l'inserimento, la rimozione, la modifica e la visualizzazione degli oggetti e delle loro proprietà.

1.2 Funzionalità

Per facilitare la visualizzazione degli oggetti dell'inventario, questi sono suddivisi all'interno del programma in quattro diverse schede; ogni scheda rappresenta una sottosezione dell'inventario, e mostra al suo interno solo gli elementi appartenenti alla categoria indicata dal titolo.
Per la gestione degli oggetti dell'inventario sono presenti le seguenti funzionalità:

- Caricamento ed esportazione dell'intero inventario da e su file `.xml`;
- Aggiunta di un nuovo oggetto all'inventario;
- Modifica di un elemento già presente nell'inventario;
- Rimozione di un elemento dell'inventario;
- Rimozione di tutti gli elementi presenti;
- Visualizzazione di oggetti dell'inventario divisi per categoria di appartenenza;
- Visualizzazione delle caratteristiche di ogni elemento dell'inventario, comprese alcune statistiche calcolate automaticamente dal programma;
- Visualizzazione di avvisi d'errore.

2 Breve manuale

Alla prima apertura del programma, la finestra presenta un menù superiore, seguito da una visualizzazione a schede dell'inventario. L'inventario non contiene ancora alcuna informazione, poiché è lasciato all'utente il controllo del proprio inventario.

2.1 Barra menù

Il menù presenta quattro voci:

- **File**: permette le manipolazioni *in blocco* dell'inventario. Una volta premuto su di esso comparirà un menù a tendina con le seguenti voci:
  - **Load inventory**: permette il caricamento di un inventario precedentemente salvato da un file `.xml`;
  - **Save inventory**: permette il salvataggio dell'inventario su un file `.xml`;
  - **Empty inventory**: permette lo svuotamento totale dell'inventario, ossia l'eliminazione di ogni elemente presente in esso;
- **Add item**: permette l'aggiunta di un singolo elemento all'inventario;
- **Delete item**: permette l'eliminazione di un elemento dall'inventario;
- **Edit item**: permette la modifica di un elemento già presente nell'inventario.

2.2 Caricamento e salvataggio di un inventario

Una volta premuto su **Load inventory** si aprirà un form di apertura di file, gestito dal sistema operativo sul quale si trova in esecuzione il programma. L'utente potrà quindi navigare all'interno del proprio filesystem fino a trovare un file in formato `.xml`; una volta trovato potrà selezionarlo e caricarlo quindi all'interno del programma.
Dopo l'aggiunta, la modifica o la rimozione di elementi dell'inventario, l'utente potrà salvare questo in formato `.xml` premendo su **Save inventory**. Anche in questo caso il form di salvataggio di file è gestito dal sistema operativo, e valgono le stesse regole che si hanno per il caricamento.
Il caricamento e il salvataggio prevedono l'utilizzo esclusivo del formato `.xml`; nel caso in cui l'utente provi a caricare un file mal formato o di tipo diverso rispetto a quanto richiesto del programma, quest'ultimo segnalerà un errore attraverso una finestra di segnalazione e non caricherà alcun elemento.

2.3 Aggiunta e modifica di un elemento

Una volta premuto su **Add item**, il programma mostrerà una finestra di inserimento dati. L'utente potrà quindi inserire i dati a piacimento, rispettando naturalmente le regole imposte dal programma. In merito a queste, si segnala che è possibile inserire le caratteristiche dell'elemento in base al tipo di questo; le caratteristiche non appartenenti allo specifico tipo selezionato non sono selezionabili né modificabili. Una volta inserite le informazioni desiderate, l'utente dovrà premere il pulsante per permettere così l'inserimento nell'inventario.
Per quanto riguarda la modifica, l'utente dovrà selezionare l'elemento da modificare e cliccare su **Edit item**. Nel caso in cui l'elemento sia selezionato si aprirà una finestra analoga a quella di inserimento, con la differenza che riporterà già i dati precedentemente caricati; l'utente potrà quindi modificare a piacimento le proprietà dell'elemento e salvare le modifiche con il pulsante. Nel caso in cui non vi sia alcun elemento selezionato, invece, si aprirà una finestra che segnalerà l'errore e la finestra di modifica non verrà aperta.

2.4 Rimozione di un elemento

Per rimuovere un elemento, l'utente dovrà selezionare l'elemento da eliminare e premere **Delete item**. Nel caso in cui l'elemento sia selezionato esso verrà cancellato; nel caso in cui non vi sia alcun elemento selezionato, invece, si aprirà una finestra che ne segnalerà l'errore e nessuna riga verra cancellata.

2.5 Visualizzazione delle informazioni

Una volta che l'inventario sarà riempito con un numero arbitrario di oggetti (anche solo uno), l'utente potrà cliccarvi sopra per visualizzare le relative informazioni. Una volta cliccato un elemento, si aprirà automaticamente una scheda laterale che mostrerà le informazioni dell'oggetto selezionato.

3 Progettazione

Lo sviluppo del progetto si è basato sul pattern **Model-View** di *Qt* e una metodologia mista *top-down* e *bottom-up*: si è infatti partiti con la metodologia *top-down* per la progettazione della gerarchia e del container, ma durante la scrittura di *model* e *view* si è ripensato leggermente alla struttura di questi.
Oltre alla gerarchia, è stato realizzato un container templatizzato per il contenimento degli oggetti appartenenti alla gerarchia. Sono stati realizzati inoltre:

- Una *GUI (Graphical User Interface)*, basata su classi preesistenti di *Qt*;
- Un *model*, il quale si occupa della gestione dei dati del programma, basato anch'esso su classi preesistenti di *Qt*;
- Un *filter proxy*, che funge da intermediario tra *model* e *view* e permette di filtrare i dati per la visualizzazione corretta su ogni tab;
- Una classe di Input/Output su file in formato `.xml`.

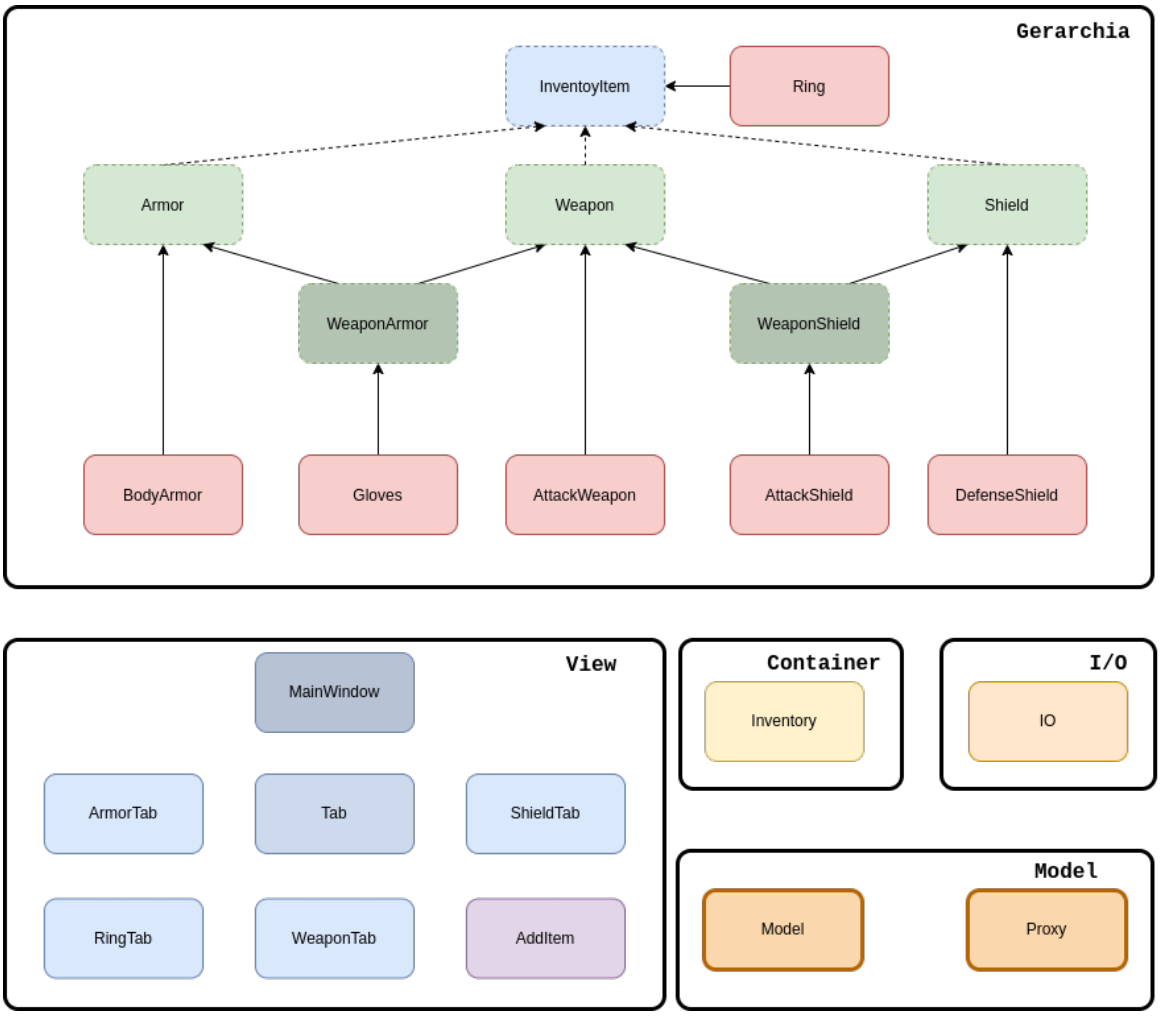

Figura 1: Diagramma delle classi di CrownSouls.

3.1 Gerarchia

La gerarchia è composta dalla classe base astratta *InventoryItem*, dalla quale derivano **direttamente** la classe concreta *Ring* e **virtualmente** le classi astratte *Armor*, *Weapon* e *Shield*. Da queste tre

classi derivano **singolarmente** e **direttamente** le tre rispettive classi concrete *BodyArmor*, *AttackWeapon* e *DefenseShield*. Viene inoltre utilizzata l'ereditarietà multipla per la definizione di classi che rappresentano oggetti appartenenti a più tipi; nello specifico, la classe *WeaponArmor* deriva direttamente da *Armor* e *Weapon*, e la classe *WeaponShield* deriva direttamente da *Weapon* e *Shield*. Questa forma di ereditarietà multipla è di tipo **is-a**, poiché un oggetto WeaponArmor è sia un oggetto Weapon che un oggetto Armor (e lo stesso vale per WeaponShield). Da queste due classi astratte derivano poi rispettivamente le classi concrete *Gloves* e *AttackShield*.
Ciascuna classe implementa dei metodi virtuali che riguardano l'impostazione e il recupero delle diverse proprietà degli elementi, e dei metodi virtuali specifici di ogni sottoclasse astratta per il calcolo e l'ottenimento di statistiche basate sulle proprietà. L'utilizzo del polimorfismo in tale contesto viene illustrato in seguito.

3.2 Container

È stata implementata una classe *Inventory* che funge da container. La classe fornisce un template di smart pointer, e simula una lista singolarmente linkata con alcuni accorgimenti: a differenza di una lista singolarmente linkata standard, infatti, essa fornisce anche un puntatore all'ultimo elemento e permette l'accesso diretto in sola lettura a un dato elemento tramite l'overloading dell'operatore accesso agli elementi del puntatore ([ ]).
La classe *Inventory* contiene al suo interno due classi annidate:

- La classe *SmartP* rappresenta uno smart pointer; è infatti questa classe a rappresentare un elemento del container di tipo T templatizzato. Oltre al contenuto effettivo dell'elemento e al puntatore all'elemento successivo, la classe è fornita anche di:
  - Costruttore e costruttore di copia profondi;
  - Distruttore profondo;
  - Assegnazione profonda;
  - Overloading degli operatori di dereferenziazione e accesso a membro;
  - Overloading degli operatori booleani di uguaglianza e disuguaglianza.
- La classe *Iterator* rappresenta l'iteratore del container. Nello specifico, un oggetto di classe Iterator è un iteratore costante, poiché non permette il *side effect* sugli oggetti a cui punta. Questa classe presenta l'overloading dei seguenti operatori:
  - Incremento prefisso;
  - Dereferenziazione;
  - Accesso a membro;
  - Uguaglianza e disuguaglianza.

La classe container *Inventory* fornisce diverse funzionalità di inserimento, cancellazione e ricerca; essa infatti permette:

- L'inserimento e la rimozione di oggetti in testa, in coda o in una posizione data;
- La modifica di un oggetto a una posizione data (sovrascrittura);
- La lettura di oggetti in testa, in coda o a in una posizione data grazie all'overloading dell'operatore [ ].

Come precedentemente detto, questa classe e le relative classi annidate sono templatizzate su tipo T; questo permette quindi il riutilizzo del container anche per applicazioni con *model* e/o *view* differenti.

3.3 Modello

Il modello, rappresentato dalla classe *Model*, fa uso delle classi della gerarchia e della classe container. Essa è derivata dalla classe *QAbstractItemModel* fornita da *Qt*; l'utilizzo di tale derivazione si

è rivelato particolarmente utile poiché permette una gestione più semplice di aggiunta, rimozione e visualizzazione degli elementi. Queste operazioni, infatti, sono svolte dai metodi *rowCount, columnCount, insertRows, removeRows, data, setData* e *headerData*, metodi virtuali o virtuali puri che sono stati ridefiniti nella classe.
Questa classe si occupa di gestire l'intero inventario presente nel programma, rappresentato dal campo dati *inventory* di classe *Inventory <InventoryItem*>*, delegando alla *view* e al *proxy model* la corretta suddivisione in tabelle di elementi dell'inventario in base al loro tipo.
La classe *Model* implementa quindi i metodi virtuali della classe base e delle diverse classi astratte derivate, aggiungendo nuovi metodi per l'inserimento, la rimozione, la modifica e la visualizzazione degli elementi.
Essa incorpora anche dei metodi *getter* che fanno uso del polimorfismo per restituire porzioni di inventario contenenti oggetti di un solo tipo; questi metodi non sono utilizzati dal programma, ma vengono lasciati per una eventuale futura espansione o modifica del programma.

3.4 GUI

La *GUI*, ossia l'interfaccia grafica, è la componente del programma che si occupa del suo aspetto grafico e dell'interazione con l'utente. Essa è composta dalle seguenti classi:

- *MainWindow*;
- *Tab*;
- *AddItem*;
- *ArmorTab, RingTab, ShieldTab* e *WeaponTab*.

3.4.1 MainWindow

La classe *MainWindow* rappresenta la finestra principale del programma; essa deriva da *QMainWindow* ed è la classe che si occupa della creazione ed istanziazione di tutte le altre componenti grafiche del programma, e della chiamata degli opportuni metodi appartenenti alle diverse classi della *GUI* attraverso opportuni *connect*.
Nello specifico, questa classe si occupa di:

- Impostare le dimensioni minime dell'applicazione;
- Creare il *widget* che si occupa della visualizzazione delle tab;
- Creare il menù superiore dell'applicazione;
- Creare e connettere le *action* dei diversi componenti grafici agli opportuni metodi delle rispettive classi. Da essa partono quindi le operazioni di:
  - Caricamento e salvataggio di un file `.xml`;
  - Svuotamento dell'inventario;
  - Aggiunta, modifica e rimozione di un elemento dall'inventario.

3.4.2 Tab

La classe *Tab*, derivata da *QWidget*, rappresenta il *widget Qt* corrispondente al modello a schede del programma. Nello specifico, essa si occupa di:

- Istanziare e impostare correttamente il *model* e il *filter proxy*;
- Istanziare le diverse schede come singoli oggetti delle rispettive classi *ArmorTab, RingTab, ShieldTab* e *WeaponTab*;
- Aggiornare la visualizzazione degli elementi dell'inventario all'interno della tabella;
- Istanziare e visualizzare gli oggetti della classe *AddItem*, ricevendo inoltre i dati inseriti in questo e aggiungendoli o modificandoli nell'inventario;

- Richiamare il *filter proxy* per la corretta suddivisione degli elementi nelle giuste schede;
- Istanziare e visualizzare la barra laterale contenente le informazioni dell'elemento selezionato.

3.4.3 AddItem

La classe *AddItem* rappresenta le finestre di aggiunta e modifica di un elemento. Essa deriva dalla classe *QDialog*, e permette l'inserimento di tutte le informazioni necessarie alla costruzione di un nuovo oggetto dell'inventario o alla modifica di uno preesistente.

3.5 Filter proxy

Il *filter proxy*, rappresentato dalla classe *Proxy*, funge da intermediario tra la *view* e il *model*. I metodi di *Proxy* vengono infatti chiamati a ogni modifica e ogni *refresh* della *view*, e fungono da filtro facendo uso di espressioni regolari e del polimorfismo (attraverso la chiamata di un metodo virtuale) per smistare correttamente gli elementi nelle diverse tab.

3.6 I/O

Il programma permette la lettura e la scrittura di interi inventari. Questa possibilità è data dalla classe *IO*, la quale fornisce i metodi necessari all'input e all'output dei dati tramite file `.xml`. Questi metodi risolvono il problema specifico del programma sviluppato, e non sono pertanto applicabili ad altri problemi.

3.7 Polimorfismo e information hiding

Il polimorfismo viene fortemente usato nei contenitori. In svariati punti del programma, infatti, sono presenti chiamate polimorfe a metodi virtuali della gerarchia.
I metodi virtuali presenti nella gerarchia sono principalmente *setter* e *getter*, metodi di cui il programma fa un largo uso. Sono inoltre presenti metodi virtuali per il calcolo delle statistiche dei singoli elementi. Entrambe queste tipologie di metodi virtuali ritornano dei valori numerici, o consistono in metodi *void* per il setting dei campi dati degli oggetti appartenenti alle classi della gerarchia.
Viene fatto uso del polimorfismo anche per la creazione degli oggetti appartenenti all'inventario; viene infatti molto sfruttata la possibilità di creare un oggetto di una classe derivata e inserirlo all'interno del container *inventory*, il quale ha tipo *Inventory<InventoryItem*>*.
Al fine di evitare il più possibile il meccanismo dell'*RTTI (run-time type information)*, si segnala la presenza di un metodo virtuale *getType()* presente nelle classi astratte derivate dalla classe base astratta pura; questo metodo ritorna tramite una stringa la classe di appartenenza dell'oggetto.

3.8 Gestione degli errori

Nel programma sono presenti blocchi *try-catch* per la gestione a *run-time* degli errori. Questi costrutti vengono usati principalmente in presenza di situazioni possibilmente critiche, dove è più importante prestare attenzione. In particolare, si fa uso della gestione di errori:

- Per l'inserimento, rimozione o modifca di elementi dell'inventario;
- Per la lettura e la scrittura dei file `.xml`, e per il conseguente inserimento degli elementi nell'inventario.

Gli errori vengono gestiti dalla *standard library*, e a seguito della gestione viene mostrato un messaggio d'errore tramite un widget *QMessageBox*.

3.9 Scelte progettuali

- Si è scelto di utilizzare una lista singolarmente linkata per la facilità e l'efficacia di questa struttura dati in un problema come quello in oggetto. Sono stati però seguiti degli accorgimenti per la semplificazione dell'accesso in sola lettura dati (tramite operatore [ ]) e per la diminuzione

dello sforzo computazionale. Un esempio di questo è la presenza di un puntatore all'ultimo elemento, che permette la riduzione di alcune operazioni, tra cui l'aggiunta e la rimozione in coda, da tempo $O(n)$ a tempo costante;

- Si è scelto di optare per una classe di Input/Output non scalabile per questioni di tempo. Una classe facilmente adattabile a più problemi, infatti, avrebbe richiesto uno studio più approfondito e un utilizzo pesante di polimorfismo sulla gerarchia; questo è certamente auspicabile, ma purtroppo incompatibile con le tempistiche reali del progetto in relazione agli sviluppatori;
- Si è optato per l'utilizzo di un unico oggetto container affiancato a un *filter proxy*, rispetto a più container, ognuno di un tipo diverso, per favorire la leggerezza del programma, la scalabilità del codice e, soprattutto, il polimorfismo;
- Si è scelto di utilizzare un metodo virtuale *getType()* per la determinazione del tipo di un oggetto per evitare l'utilizzo di *rtti*.

4 Gestione di Progetto

4.1 Suddivisione del lavoro progettuale

Il progetto è stato iniziato all'inizio del mese di giugno e si è concluso il giorno precedente alla consegna (4 luglio 2020).
Il progetto, nella la sua completezza ha richiesto circa **110** ore equalmente divise tra i due componenti del gruppo, che sono:

- Buratto Enrico - 1142644
- Ostanello Emanuele - 1143439

Entrambi i componenti hanno contribuito a tutte le parti del progetto; la progettazione iniziale dei diversi componenti del progetto è stata concepita in totale collaborazione tramite la piattaforma di comunicazione vocale *Discord*, necessaria nel periodo di distanziamento sociale che si è sovrapposto a buona parte del periodo di progettazione e sviluppo. Tuttavia, i singoli componenti hanno approfondito maggiormente un determinato ambito di progetto; nello specifico:

Enrico Buratto	**Emanuele Ostanello**
Codifica Gerarchia e Container	Apprendimento Qt
Codifica Model	Codifica View

4.2 Timeline individuale

Il lavoro individuale ha richiesto circa **55 ore** sulle 50 ore previste; le ore in eccesso sono dovute principalmente alla difficoltà di comunicazione a distanza, dovuta al periodo attuale. Nonostante questo, si può dire che il carico di lavoro è stato ben calibrato in base alla disponibilità oraria personale e a quanto sviluppato.

Fasi progettuali	**Ore utilizzate**
Analisi del problema	5
Progettazione	13
Apprendimento Qt	4
Codifica e implementazione Model	17
Codifica e implementazione View	4
Test e Debug	8
Relazione	4
Ore totali	**55**

4.3 Ambiente di lavoro

Il progetto è stato sviluppato con Qt Creator v4.11.2 e Atom v1.47.0 su sistema operativo *Arch Linux*; il compilatore usato è stato *gcc v10.1.0*. Essendo il compilatore e la versione di *Qt* del sistema di sviluppo più aggiornati rispetto alle specifiche, sono stati effettuati frequenti allineamenti con la macchina virtuale, contenente *Ubuntu 18.04 LTS* con *gcc v7.4.0* e *Qt* alla versione v5.9.5. Questo ha permesso di verificare la compatibilità di quanto prodotto con le specifiche tecniche richieste.

4.4 Istruzioni di compilazione ed esecuzione

Il progetto prevede la compilazione tramite file .pro e tool qmake. Le istruzioni per la compilazione e l'esecuzione sono quindi le seguenti:

```
$ qmake CrownSouls.pro
$ make
$ ./CrownSouls
```

Viene inoltre fornito un file `.xml`, locato nella cartella `extra`, contenente un inventario precompilato di prova.

4.5 Considerazioni finali

Il progetto ha richiesto un modesto impegno, soprattutto per quanto riguarda la progettazione e l'organizzazione del lavoro, avvenute a distanza. Una volta che il gruppo si è organizzato e il lavoro è stato diviso, però, lo sviluppo è stato abbastanza lineare, costante nel tempo e senza particolari complicazioni.
Entrambi i componenti del gruppo hanno utilizzato una parte considerevole di tempo per l'apprendimento e l'implementazione del *proxy model*, che inizialmente ha dato non pochi problemi. Una volta risolti questi, però, lo sviluppo è continuato velocemente.
Concludendo, possiamo affermare che il progetto è stato abbastanza lungo e impegnativo, ma lo studio individuale e la scrittura di codice "reale" ha portato i suoi frutti.