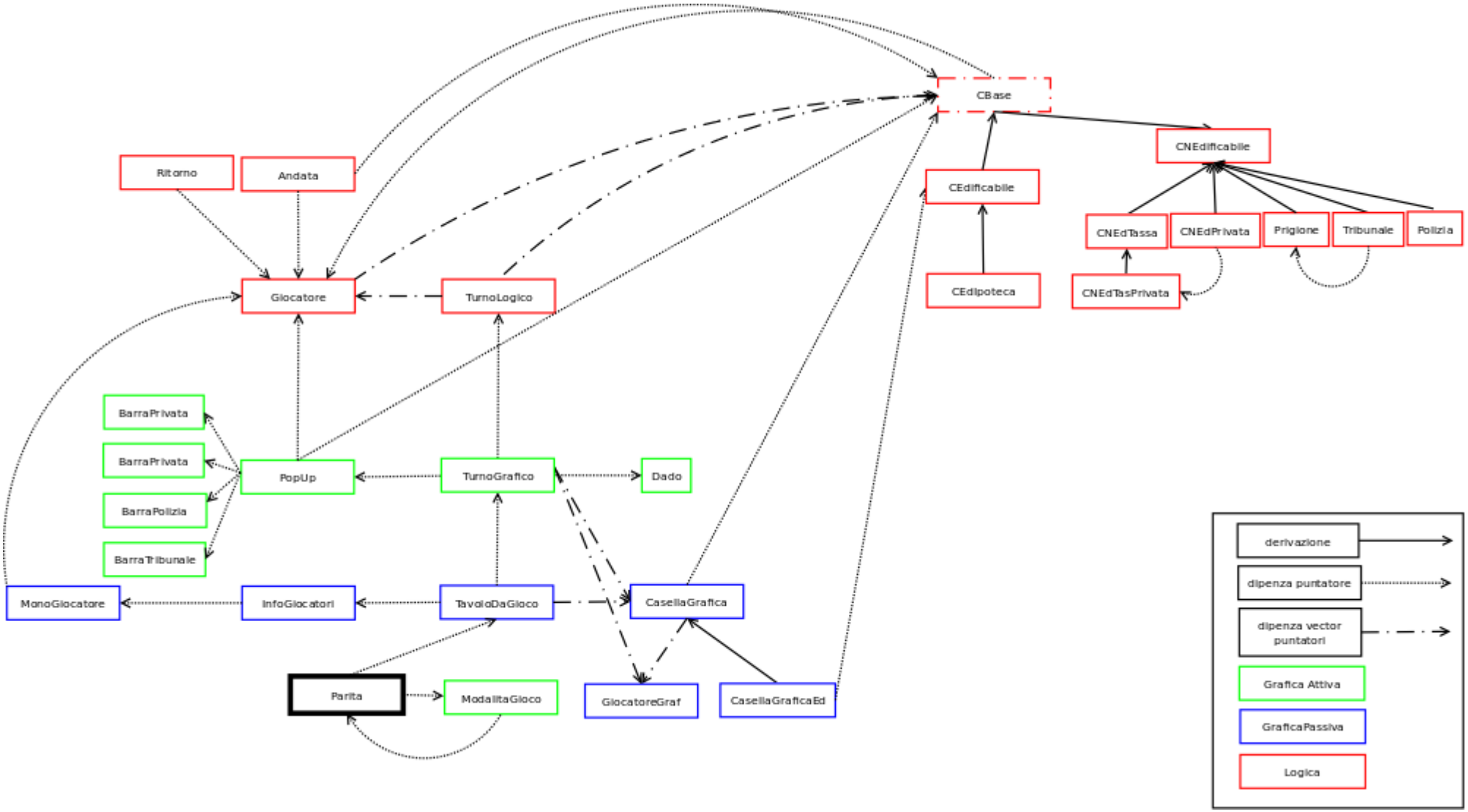
PROGETTO PROGRAMMAZIONE AD OGGETTI
A.A. 2009/2010

LUCA BONATO
MARCO ZICCARDI

Tutte le immagini contenute nel progetto sono state realizzate a mano da Luca Bonato.


CBase
CNEdificabile
CEdificabile
CNEdTassa
CNEdPrivata
Prigione
Tribunale
Polizia
CEdIpoteca
CNEdTasPrivata
Ritorno
Andata
Giocatore
TurnoLogico
BarraPrivata
BarraPrivata
BarraPolizia
BarraTribunale
PopUp
TurnoGrafico
Dado
MonoGiocatore
InfoGiocatori
TavolaDaGioco
CasellaGrafica
Parita
ModalitaGioco
GiocatoreGraf
CasellaGraficaEd
derivazione
dipenza puntatore
dipenza vector puntatori
Grafica Attiva
GraficaPassiva
Logica

INTRODUZIONE

L'intento di questa relazione è di descrivere l'intera gerarchia delle classi motivando le scelte che ci hanno condotto alla sua creazione.
Abbiamo pensato di dividere innanzitutto le classi in GRAFICHE e LOGICHE. Questa divisione è fondamentale non soltanto dal punto di vista della descrizione ma è anche uno degli scopi che la nostra gerarchia si prefigge.
Ciascuna classe sarà analizzata singolarmente nelle prossime pagine insieme a tutte le sue dipendenze e scelte cruciali, per ora ci interessa fornire soltanto qualche idea generale.
La divisione GRAFICA-LOGICA è utile a livello descrittivo ma si può fare di meglio: possiamo suddividere ulteriormente la parte grafica in due gruppi di classi: GRAFICA ATTIVA e GRAFICA PASSIVA.
Il primo gruppo rappresenta tutte quelle classi grafiche che permettono all'utente di interagire con il gioco, queste sono nell'ordine di utilizzo: ModalitaGioco, TurnoGrafico (che contiene un oggetto della classe Dado) e PopUp, PopUp che a sua volta può contenere oggetti della classe BarraIpoteca, BarraPrivata, BarraPolizia e BarraTribunale.
Il secondo gruppo rappresenta quelle classi grafiche che hanno semplicemente scopo di contenitore, come TavoloDaGioco e GiocatoreGraf oppure informativo, come infoGiocatori, CasellaGrafica e CasellaGraficaEd.
Inizieremo descrivendo le classi Logiche, passeremo poi alla Grafica Passiva e finiremo con quella Attiva. C'è però una classe che sfugge alle nostre classificazioni ed è prevedibilmente la classe Partita, che crea e raccoglie tutte le informazioni relative alla sessione di gioco. Questa classe conterrà quindi sia componenti logiche che grafiche e sarà analizzata alla fine della relazione.
Per ogni classe di una certa rilevanza sono presenti due sottosezioni: *Estendibilità* e *Indipendenza.*
Estendibilità si propone di analizzare come la classe si presti a possibili estensioni future del gioco mentre *Indipendenza* tratta il problema della divisione tra grafica e logica.
Prima di cominciare vogliamo parlare di un problema di *Estendibilità* sul quale abbiamo riflettuto e la cui decisione ha condizionato tutto lo sviluppo delle classi:
nell'ottica di ottenere la maggiore estendibilità abbiamo deciso di organizzare le caselle (sia grafiche che logiche) e i giocatori all'interno di std::vector. Vediamo le motivazioni che hanno portato a questa scelta analizzando le altre modalità che avremmo potuto adottare:

1. Array + #define:
   La prima modalità che andiamo a valutare è quella del define, avremmo potuto definire tramite la direttiva #define dei nomi (ad esempio num_caselle e num_giocatori) per il numero delle caselle e il numero dei giocatori, andando a scorrere gli array fino a num_caselle e num_giocatori.
2. Array sullo heap + variabili globali
   La seconda modalità prevedeva di creare un file HEADER che andasse a contenere due variabili globali a rappresentazione del numero di caselle e del numero di giocatori. Naturalmente questo ci avrebbe costretto ad includere il file .h in tutti i file che avrebbero usato array di giocatori o caselle e avremmo dovuto allocare tutti questi array sullo heap, essendo array di puntatori non ci sembrava essere la necessità di allocarli sullo heap.

Nessuna delle due precedenti tecniche permette di inserire una modalità di selezione del numero di giocatori, questo ci è sembrato essere molto limitante.

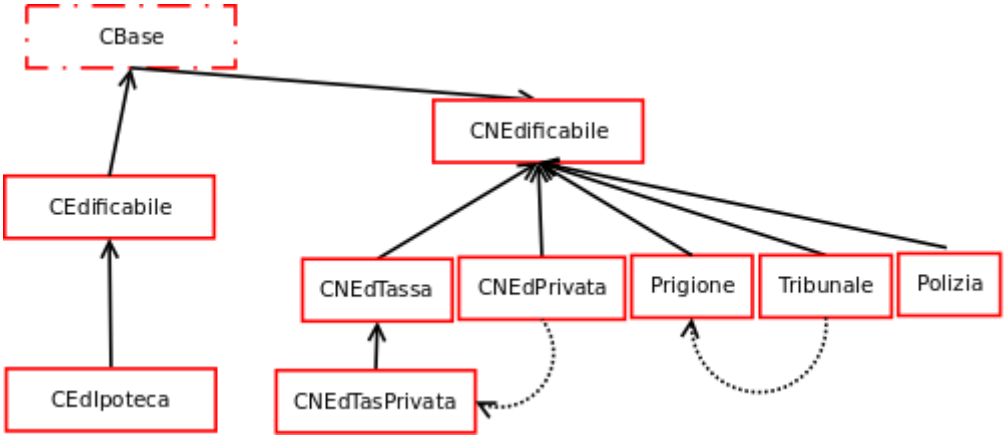
3. Vector
   L'utilizzo dei vector ci permette di estendere tranquillamente il codice per quanto riguarda il numero di giocatori (naturalmente si sta parlando della parte logica, per la grafica se si volesse estendere a più di 4 giocatori si renderebbe necessario un riadattamento delle dimensioni come vedremo in TavoloDaGioco), cosa che infatti abbiamo fatto permettendo all'utente di giocare in 2, 3 o 4 giocatori. Questo vale anche per le caselle (nella parte logica) che si possono aggiungere liberamente.

LOGICA

LA GERARCHIA DELLE CASELLE LOGICHE

CBASE

Qualsiasi casella logica deve derivare dalla classe base Cbase (che sta per casella base) che è una classe virtuale, quindi non la si può istanziare se non come sotto-oggetto.



CBase contiene i campi dati PROTECTED: string nomeCasella, Giocatore * proprietario e int indiceCasella, i nomi dicono già tutto ma è bene sottolineare che le caselle non edificabili che non hanno proprietario hanno questo campo a NULL e che indiceCasella è la posizione della casella nel Tavolo. CBase possiede inoltre un vector di booleani chiamato presenti, campo dati PUBBLICO che registra i giocatori presenti sulla casella, vedremo in CasellaGrafica perché PUBBLICO.
CBase rende disponibili i seguenti metodi:

```
1.  CBase(string, int, int);
2.  virtual Andata sceltaOpzione(Giocatore*) = 0;
3.  virtual void liberaCasella(Giocatore *) = 0;
4.  virtual int valoreCasella(Giocatore *) const =0;
5.  virtual void eseguiAzione(const Ritorno&) = 0;
6.  void aggiungiGiocatore(int);
7.  void rimuoviGiocatore(int);
8.  void setTuttiPresenti();
9.  int getIndiceCasella() const;
10. string getNomeCasella() const;
11. Giocatore *getProprietario() const;
12. virtual ~CBase();
```

Come possiamo vedere i metodi 2. 3. 4. e 5. sono virtuali PURI, di essi sarà quindi obbligatoria la concretizzazione nelle classi derivate, vediamo più dettagliatamente perché:

1. CBase(string, int, int), è il costruttore di CBase, la string inizializza nomeCasella, il primo intero inizializza l'indiceCasella mentre il secondo intero ci dice con quanti giocatori stiamo giocando, facendo sì che la casella crei e inizializzi di conseguenza il vector delle presenze;
2. *Andata sceltaOpzione(Giocatore*)*, restituisce un valore di tipo andata, che altro non è che lo strumento di comunicazione da parte logica a parte grafica, Andata raccoglie informazioni relative alle operazioni disponibili sulla casella stessa quindi, naturalmente, dato che Andata è unica per tutte le caselle, ciascuna casella avrà un diverso modo di riempire i suoi campi

dati a seconda delle operazioni possibili, in base anche alla situazione economica del giocatore e al suo rapporto con la casella (proprietario terriero, proprietario edile o semplicemente di passaggio). É ora chiara la necessita che *Andata sceltaOpzione(Giocatore*)* sia un virtuale puro nella classe Cbase;

3. *void liberaCasella(Giocatore *)* ha il compito, dato il puntatore ad un giocatore, di liberare la casella dalla sua presenza. Se ad esempio la casella è edificabile e il giocatore con cui viene invocata *liberaCasella* ne è il proprietario la casella torna libera. Tuttavia *liberaCasella* deve funzionare anche con caselle che non siano edificabili ma "partecipabili", in questi casi invocare *liberaCasella(g)* vuol dire eliminare eventuali partecipazioni del giocatore g sulla casella oggetto di invocazione;
4. *int valoreCasella(Giocatore *) const* restituisce il valore economico della casella per il giocatore parametro, per le caselle edificabili in possesso del giocatore questo valore è il costo della casella sommato all'eventuale costo dell'albergo (per quello con possedute è 0), invece per le caselle "partecipali" questo valore dipende da che quota della società è in possesso del giocatore parametro;
5. *void eseguiAzione(const Ritorno)* riceve un oggetto di tipo Ritorno e a seconda del contenuto di questo esegue determinate operazioni sulla casella, non ha senso parlarne per una CBase, vedremo nelle successive derivazioni le sue concretizzazioni;
6. void aggiungiGiocatore(int), segna il giocatore con indice uguale al parametro attuale a presente (true nel vector dei presenti);
7. void rimuoviGiocatore(int), toglie il giocatore dalla casella;
8. void setTuttiPresenti(), utilizzato all'inizio del gioco per mettere tutti i giocatori sulla stessa casella, la Partenza;
9. int getIndiceCasella() const, un semplice getter che restituisce l'indice dell'oggetto di invocazione;
10. string getNomeCasella() const, un altro semplice getter che restituisce il nome dell'oggetto di invocazione;
11. Giocatore* getProprietario const, l'ennesimo getter che restituisce il puntatore al giocatore proprietario dell'oggetto di invocazione;
12. virtual ~CBase, infine il distruttore virtuale, la sua presenza è essenziale per essere sicuri di non rimuovere soltanto il sotto-oggetto di tipo CBase quando si invoca la delete su di un puntatore a CBase con tipo dinamico diverso da CBase *.

CEDIFICABILE

La classe logica che concretizza l'idea di casella edificabile è CEdificabile, che deriva pubblicamente da CBase. Rappresenta ovviamente una casella in cui è possibile diventarne i proprietari comprando il terreno o in cui è possibile costruire l'albergo.

```
private:
 const int costoTerreno;
 const int costoAlbergo;
 const int pedaggioTerreno;
 const int pedaggioAlbergo;
 bool edificata;

public:
 CEdificabile(string, int, int, int, int, int, int);
 int getCostoTerreno() const;
 int getCostoAlbergo() const;
 int getPedaggioTerreno() const;
 int getPedaggioAlbergo() const;
 bool isEdificata() const;

 virtual Andata sceltaOpzione(Giocatore *);
 virtual void eseguiAzione(const Ritorno&);
 void compraTerreno(Giocatore *);
 void edificaAlbergo();
 void pagaPedaggio(Giocatore *);
 virtual void liberaCasella(Giocatore *);
 virtual int valoreCasella(Giocatore * g) const;
```

1. I campi dati privati racchiudono le informazioni principali appartenenti ad una casella edificabile (sempre tenendo presente che vi sono campi dati aggiuntivi che appartengono al sotto-oggetto CBase come il nome della casella, il proprietario della casella etc.):
   - *const int costoTerreno*, determina la quantità in ghiande necessaria per appropriarsi del terreno che la casella rappresenta. È costante in quanto il suo valore non cambia durante il corso del gioco.
     - *int getCostoTerreno() const* è il suo getter;
   - *const int costoAlbergo*, determina la quantità in ghiande necessaria per costruire l'albergo sul proprio terreno. È costante in quanto il suo valore non cambia durante il corso del gioco.
     - *int getCostoAlbergo() const* è il suo getter;
   - *const int pedaggioTerreno*, determina la quantità in ghiande che un giocatore deve versare al proprietario per il soggiorno nella casella in cui è stato solamente comprato il terreno. È costante in quanto il suo valore non cambia durante il corso del gioco.
     - *int getPedaggioTerreno() const* è il suo getter;
   - *const int pedaggioAlbergo*, determina la quantità in ghiande che un giocatore deve versare al proprietario per il soggiorno nella casella quando è stato edificato l'albergo. È costante in quanto il suo valore non cambia durante il corso del gioco.
     - *int getPedaggioAlbergo() const* è il suo getter
   - *bool edificata*, è un flag booleano che informa se nella data casella è stato costruito l'albergo.
     - *bool isEdificata() const* è il suo getter;

2. *virtual Andata sceltaOpzione(Giocatore *G)*[1] : è la concretizzazione del metodo virtuale disponibile in CBase.
   Ritorna un oggetto Andata che determina quello che il giocatore G può fare nella cella in cui è arrivato (quindi questa). Questo oggetto andrà poi utilizzato da PopUp per informare il giocatore su quel che potrebbe fare.
   - Determina se la casella ha di già un proprietario. In caso non lo avesse (quindi puntatore nullo == proprietario è la banca) viene creato un oggetto Andata in cui si informa che G potrebbe solamente comprare il terreno. Quindi viene pure calcolato se il giocatore G possiede abbastanza ghiande per comprare il terreno. Queste informazioni vengono inserite in Andata;
   - In alternativa determina se G è il proprietario della casella. Nel qual caso controlla se è già stato costruito l'albergo. Se la casella non è già stata edificata si costruisce Andata in modo da informare la parte grafica che G può solo edificare l'albergo[2], e informa altresì se G ne abbia l'effettiva possibilità calcolando se le ghiande in suo possesso gli siano sufficienti per l'investimento.
     Nel caso la casella fosse già edificata si costruirebbe l'oggetto Andata in modo da informare la parte grafica che in questa casella G non può fare più nulla perché ha già fatto tutto il fattibile;
   - In alternativa determina se G sia finito su di una casella posseduta da un altro giocatore. Nel qual caso G dovrebbe pagare il dazio appropriato (a seconda che la casella sia edificata o meno). Viene creato l'oggetto Andata rispecchiando tali informazioni;

3. *virtual void eseguiAzione(const Ritorno &r)* : è la concretizzazione del metodo virtuale fornito da CBase.

---

1 Poiché tale funzione restituisce un oggetto di tipo Andata, per sapere in che modo viene creato l'oggetto corrispondente si rimanda alla sezione apposita che discute la classe Andata.

2 Una piccola precisazione: il giocatore ha la possibilità teorica di compiere un'azione (che è vincolata dalla casella in cui si trova e dallo stato di quella casella: la casella non ha il proprietario, il giocatore è il proprietario e non ha ancora costruito l'albergo etc) e ha la possibilità pratica di compiere l'azione teorica solo nel caso riuscisse a soddisfare a particolari limiti (possedere abbastanza ghiande, non sforare il limite massimo di azioni privatizzate etc)

Questa funzione esegue a livello logico la scelta che il giocatore ha fatto. Per modificare le informazioni logiche si basa sulle informazioni fornite dall'oggetto Ritorno. Tale oggetto raccoglie le informazioni necessarie che l'utente ha manifestato attraverso l'utilizzo di PopUp.

- *void pagaPedaggio(Giocatore *G2),* quando r.opereazioni[3] == true
  la funzione fa in modo di decrementare le ghiande a G2 per darle al giocatore che è il proprietario della casella.
  Il proprietario della casella può ricevere meno ghiande di quante potrebbe aspettarsene poiché le ghiande vengono sottratte a G2 che potrebbe non averne abbastanza e quindi nel frattempo morire. Per ulteriori informazioni andare alla sezione Giocatore;
- *void compraTerreno(Giocatore *G2)*, quando r.operazioni[0] == true
  la funzione fa in modo di attribuire la casella a G2.
  In primo luogo decrementa le ghiande corrispettive alla somma da versare alla banca per l'acquisto del terreno.
  Imposta il proprietario della casella a G2.
  Informa G2 di esser divenuto proprietario di un bene immobile chiamando la funzione G2->aumentaPossedimenti(casellaAttuale);
- *void edificaAlbergo(),* quando r.operazioni[1] == true
  la funzione non ha bisogno di parametri esterni in quanto sa già quanto il giocatore deve pagare poiché l'informazione è propria della casella, e sa pure chi sta costruendo l'albergo poiché l'unico giocatore che potrebbe costruire in questa casella è il proprietario.
  Per prima cosa imposta il flag booleano edificata a true.
  Poi si preoccupa di decrementare del giusto quantitativo di ghiande le scorte del giocatore che è proprietario della casella;

4. *virtual void liberaCasella(Giocatore *G),* è la concretizzazione del metodo virtuale che fa in modo di liberare completamente la casella dal suo proprietario e dall'eventuale albergo costruito. Questa funzione viene chiamata quando un giocatore muore, per far in modo che le caselle da lui possedute possano venir utilizzate da altri giocatori viventi.
   Il parametro Giocatore* viene passato solo per permettere l'override della funzione a partire da CBase. In questo caso G non viene utilizzato in quanto una casella edificabile può avere un solo proprietario e non ci possono essere situazioni di ambiguità su cosa dover liberare: l'unica cosa che può fare è demolire l'albergo (edificata = false) e ridare il terreno alla banca (proprietario = 0).

5. *virtual int valoreCasella(Giocatore *G) const,* è la concretizzazione del metodo virtuale che permette di determinare il costo effettivo della proprietà.
   È stato deciso che il valore della proprietà viene data dal costo del terreno e (nel caso fosse stato costruito) anche dal costo dell'albergo.
   Questa funzione viene utilizzata per sapere quale giocatore ha vinto nel caso si sforassero il limite massimo di giri.
   Il parametro Giocatore* viene passato per riuscire a fare l'override del metodo a partire da CBase. Il parametro va utilizzato nelle caselle CNEdPriv per distinguere che giocatore possiede quote azionarie e in quale quantitativo. In questo caso G viene ignorato.

6. *Costruttore,* i primi due parametri del costruttore sono utilizzati per creare il sotto-oggetto CBase (il nome della casella e l'indice della casella). Gli altri 4 interi sono utilizzati per inizializzare il valore di: costoTerreno, costoAlbergo, pedaggioTerreno e pedaggioAlbergo.

CEDIPOTECA

La classe logica che concretizza l'idea di casella in cui è possibile modificare l'ipoteca è CEdIpoteca.
Rappresenta ovviamente una casella in cui è possibile diventarne i proprietari comprando il terreno, in cui è possibile costruire l'albergo e soprattutto, nel caso fosse costruito l'albergo, di poter ipotecarlo.
Fa ovviamente parte della gerarchia di classi che modella l'idea delle varie tipologie di caselle e deriva pubblicamente da CEdificabile.

```
private:
 int ipotecato;

public:
 CEdIpoteca(QString, int, int, int, int, int, int);
 int getIpoteca() const;
 virtual Andata sceltaOpzione(Giocatore *);
 virtual void eseguiAzione(const Ritorno&);
 void pagaPedaggio(Giocatore *);
 void aumentaIpoteca(int);
 void decurtaIpoteca(int);
 virtual void liberaCasella(Giocatore *);
 virtual int valoreCasella(Giocatore *) const;
```

1. *int ipotecato*, è l'unico valore che distingue CEdIpoteca da CEdificabile. Determina la quantità percentuale di ipoteca dell'albergo.
   - *int getIpoteca() const*, è il rispettivo getter;

2. *virtual Andata sceltaOpzione(Giocatore *G)*, è la concretizzazione del metodo virtuale disponibile in CBase e overridato da CEdificabile.
   Ritorna un oggetto Andata che determina quello che il giocatore G può fare nella cella in cui è arrivato (quindi questa). Questo oggetto andrà poi utilizzato da PopUp per informare il giocatore su quel che potrebbe fare.
   - Le operazioni che svolge sono identiche a quelle svolte da CEdificabile (si rimanda a CEdificabile per una trattazione più dettagliata)
   - In più deve solamente fornire all'utente l'informazione riguardante la modifica dell'ipoteca. Questa informazione viene mandata quando G è il proprietario del terreno e sulla casella è stato edificato l'albergo.
     Le informazioni necessarie per Andata sono: il limite minimo sotto il quale il giocatore non può riscattare l'ipoteca dell'albergo, la percentuale di ipoteca dell'immobile e il costo di una quota del 10% di ipoteca.
     L'ultima informazione viene calcolata a partire dal costo dell'albergo : costoAlbergo/10.
     La percentuale di ipoteca è definita dal valore di ipotecato.
     Il valore minimo viene calcolato a passi: prima viene calcolato il costo di un 10% di ipoteca. Quindi viene divisa la quantità di ghiande posseduta dal giocatore per il valore del 10% dell'ipoteca per scoprire quante quote di 10% potrebbe permettersi di comprare (l'arrotondamento è eseguito sfruttando l'operazione di divisione tra interi che in questo caso equivale all'operazione di ceiling(n)). Una volta calcolato il numero di quote del 10% riscattabili, questo valore moltiplicato per 10 (così si ottiene il numero di quote dell'1% di ipoteca riscattabili) viene sottratto al valore di ipotecato (che si trova già nell'unità di misura percentuale: per questo si è dovuto moltiplicare per 10 l'altro valore). Ora, se il valore appena calcolato dalla sottrazione è inferiore a 0 vuol dire che il giocatore ha abbastanza soldi per riscattare completamente l'ipoteca e quindi il limite viene impostato a 0, altrimenti il valore rappresenterà l'effettivo limite.

3. *virtual void eseguiAzione(Ritorno &r),* è la concretizzazione del metodo virtuale fornito da CBase e overridato da CEdificabile.
   Questa funzione esegue a livello logico la scelta che il giocatore ha fatto. Per modificare le informazioni logiche si basa sulle informazioni fornite dall'oggetto Ritorno. Tale oggetto raccoglie le informazioni necessarie che l'utente ha manifestato attraverso l'utilizzo di PopUp.
   - L'esecuzione di tale metodo è la medesima che ha avuto per CEdificabile
   - Si aggiunge un solo if-statement in più: se r.operazioni[2] == true significa che il giocatore ha deciso di modificare la sua percentuale di ipoteca.
     - *void aumentaIpoteca(int x)* : quando r.valore > 0
       Tale funzione aumenta la percentuale di ipoteca dell'albergo incrementando di x punti il valore di ipotecato. Inoltre fornisce al giocatore chiamante (e quindi al proprietario della casella) il valore in ghiande comprensivo della parte di albergo appena ipotecata. In questo caso basta chiamare proprietario->aumentaDenaro(x * valoreAlbergo/100) [diviso 100 perchè x è in percentuale e non in quote da 10%]
     - *void decurtaIpoteca(int x),* quando r.valore < 0
       Tale funzione diminuisce la percentuale di ipoteca dell'albergo decrementando di x punti il valore di ipotecato. Inoltre toglie al giocatore chiamante (e quindi al proprietario della casella) il valore in ghiande comprensivo della percentuale di ipoteca appena riscattata. In questo caso basta chiamare proprietario->decurtaDenaro(abs(x) * valoreAlbergo/100)

4. *void pagaPedaggio(Giocatore *G),* ha la stessa funzionalità che aveva in CEdificabile. L'unica cosa che cambia è che prima di aumentare le ghiande al proprietario, viene fatto un controllo sulla percentuale di ipoteca dell'immobile. Le ghiande ricevute sarrano proporzionali al (100-ipotecato)% di quelle versate dal malcapitato. (vale anche quando il malcapitato ha meno ghiande di quelle che dovrebbe versare. La somma viene spartita tra la banca e il giocatore equamente e non si privilegia nessuno dei due)

5. *virtual void liberaCasella(Giocatore *G),* ha lo stesso comportamento che aveva in CEdificabile, infatti in questo caso richiama semplicemente CEdificabile::liberaCasella(G) e quindi azzera il valore di ipotecato. Per una trattazione più dettagliata vedere la sezione di CEdificabile.

6. *virtual int valoreCasella(Giocatore *G) const,* ha lo stesso comportamento che aveva in CEdificabile, infatti in questo caso ritorna il valore di CEdificabile::valoreCasella(G) a cui viene sottratto un valore rispettivo a ipotecato*costoAlbergo/100 : cioè non viene calcolato il valore di albergo che è stato ipotecato e che quindi ora appartiene alla banca e non più al giocatore.

<u>CNEDIFICABILE</u>

La classe CNEdificabile deriva pubblicamente da CBase e deve essere istanziabile in quanto nella modalità di gioco normale le caselle: Municipio, Palazzo della Regione, Palazzo della Provincia, Prigione, Polizia, Tribunale e Partenza sono oggetti di CNEdificabile poiché non permettono all'utente di effettuare alcuna operazione. Nella modalità Legale invece Polizia Tribunale e Prigione si specializzano esattamente come avviene per Municipio, Palazzo della Regione e Palazzo della Provincia nella modalità Privatizzata.
CNEdificabile deve quindi concretizzare i metodi virtuali puri di CBase. Fornisce invece le concretizzazioni dei metodi:

```
1. virtual Andata sceltaOpzione(Giocatore*) ;
2. virtual void liberaCasella(Giocatore *);
3. virtual int valoreCasella(Giocatore *) const;
4. virtual void eseguiAzione(const Ritorno&) ;
```

vediamo con ordine[3]:

1. *virtual Andata sceltaOpzione(Giocatore*)*, come già questa funzione dovrebbe indicare quali sono le operazioni disponibili sulla casella, naturalmente essendo la classe delle caselle non edificabili in cui non è disponibile alcun tipo di operazione la funzione dovrà ritornare un semplice oggetto della classe Andata di tipo Null (si veda Andata);
2. *virtual void liberaCasella(Giocatore *)*, una casella della classe CNEdificabile non ha alcun tipo di proprietario, per questo la funzione non fa niente;
3. *int valoreCasella(Giocatore *) const*, come si può immaginare il valore di una casella non edificabile è 0, la funzione ritorna 0;
4. v*oid eseguiAzione(const Ritorno&)*, come già detto non ci sono operazioni disponibili su questa casella, quindi se una eseguiAzione(const Ritorno&) venisse invocata su un oggetto di questa classe non provocherebbe alcun tipo di reazione

Il costruttore CNEdficabile ha la forma: CNEdficabile(string, int, int) il suo compito è essenzialmente quello di invocare il costruttore di CBase con i parametri passatigli. Il distruttore non viene invece ridefinito in quanto non ci sono nuovi campi dati, sarà comunque virtuale per le successive derivazioni;

CNEDTASSA

Questa classe deriva pubblicamente da CNEdificabile e rappresenta quel tipo di caselle che non sono edificabili ma prevedono un pedaggio per i giocatori che ci si fermano sopra.
Oggetti di questa classe sono, nella modalità normale, Società Elettrica, Società dell'Acqua Potabile e Ferrovia.
CNEdTassa aggiunge, come logico, un unico campo dati PROTETTO che è *const int quotaPedaggio* e rappresenta quanto un giocatore che si fermi sulla casella deve pagare.
In CNEdTassa viene fatto l'override dei metodi sceltaOpzione ed eseguiAzioni, è presente inoltre un costruttore ed una funzione protetta void pagaPedaggio(Giocatore*); il distruttore non viene ridefinito perché ai nostri scopi va benissimo quello standard:

```
protected:
    1. void pagaPedaggio(Giocatore*);
public:
    2. CNEdTassa(string, int, int, int);
    3. virtual Andata sceltaOpzione(Giocatore*);
    4. virtual void eseguiAzione(const Ritorno&);
```

1. *void pagaPedaggio(Giocatore*)*, sottrae al giocatore con cui viene invocato quotaPedaggio dell'oggetto di invocazione;
2. *CNEdTassa(string, in, int, int)*, invoca con il parametro string e i primi due interi il costruttore del sotto-oggetto CNEdficabile, utilizza il terzo intero per inizializzare ed istanziare quotaPedaggio;
3. *virtual Andata sceltaOpzione(Giocatore *G)*, questo override fa in modo che la funzione

3 Abbiamo deciso di ripetere il virtual anche nelle concretizzazioni e negli override ben sapendo che non è necessario ma per permettere una più facile lettura del codice.

ritorni un oggetto della classe Andata con tipo Paga, inserendo in Andata il giocatore che deve pagare G e la quotaPedaggio;

4. *virtual void eseguiAzione(const Ritorno& r)*, il parametro della funzione conterrà sicuramente le informazioni relative al giocatore che deve pagare in r.giocotore, la funzione sottrae a r.giocatore quotaPedaggio ghiande;

CNEDTASPRIVATA

Questa classe deriva pubblicamente da CNEdTassa e rappresenta quelle caselle Tassabili di cui è però possibile acquistare una percentuale presso l'ente gestore. Per esempio nella modalità Privatizzata oggetti di questa classe sono la Società Elettrica, la Società dell'Acqua Potabile e la Ferrovia.
La classe CNEdTasPrivata aggiunge rispetto CNEdTassa dei campi dati PRIVATI[4] allo scopo di gestire le partecipazioni: *const int costoQuota const* che rappresenta quanto costa acquistare il 10% della società, *int quotaPrivata* che rappresenta per quale percentuale è stato finora privatizzato l'oggetto, *vector<Giocatore*> membriAzionisti* è un vector della dimensione del numero dei giocatori, in ogni sua cella 'i' contiene il puntatore al giocatore di indice 'i' se questo è un azionista altrimenti contiene 0, a breve spiegheremo perché non è sufficiente un vector di booleani, il *vector<int> quoteAzionist*i che contiene per ciascun giocatore la percentuale posseduta.
CNEdTasPrivata compie l'override di tutti i metodi virtuali che eredita da CNEdTassa, ne definisce di nuovi ed in più ridefinisce pagaPedaggio, non serve ridefinire il distruttore, ci è sufficiente il distruttore standard:

```
1.  CNEdTasPrivata(string, int, int, int, int);
2.  void incrementaAzionista(Giocatore*, int);
3.  void decrementaAzionista(Giocatore*, int);
4.  int getCostoQuota() const;  int getQuotaPrivata() const;  int getQuotaAzionista(int) const;
5.  virtual Andata sceltaOpzione(Giocatore*);
6.  virutal void eseguiAzione(const Ritorno&);
7.  void pagaPedaggio(Giocatore*);
8.  void liberaCasella(Giocatore*);
9.  int valoreCasella(Giocatore*);
```

1. *CNEdTasPrivata(string, int, int, int, int)*, è il costruttore, con string, il primo il terzo e l'ultimo intero viene invocato il costruttore del sotto-oggetto di tipo CnedTassa, inoltre l'ultimo parametro che rappresenta il numero di giocatori viene utilizzato per sapere il numero di elementi che dovranno contenere i due vector, ciascun elemento è inizializzato a zero, coerentemente con quanto detto sopra; costoQuota viene costruito con secondo intero, quotaPrivata è inizializzato a 0;
2. *void incrementaAzionista(giocatore *,int)*, non fa altro che aumentare di int le quote del giocatore con cui viene invocata la funzione se questo giocatore è gia un azionista, se non è già un azionista viene aggiunto, questa funziona aumenta anche il campo dati quotaPrivata;
3. *void decrementaAzionista(giocatore*, int)*, decrementa di int le quote possedute dal giocatore con cui viene invocata la funzione, se queste quote scendono a zero il giocatore viene messo a zero nella sua casella del vector degli azionisti e l'oggetto di invocazione è tolto dai possedimenti del giocatore, gli vengono inoltre restituite le ghiande della quota;
4. *int getCostoQuota() const*; *int getQuotaPrivata() const*; *int getQuotaAzionista(int) const*;

4 In questo caso i campi dati sono privati e non protetti perché nella nostra gerarchia di classi non si è rivelata la necessità di derivare da CNEdTasPrivata.

sono tutti e tre dei getters, il primo restituisce il costo della quota, il secondo la percentuale già privatizzata della società e il terzo dato l'indice di un giocatore restituisce la percentuale in possesso di quel giocatore;

5. *virtual Andata sceltaOpzione(Giocatore*G)*, sulle caselle di questa classe l'unica operazione disponibile è quella di pagare il pedaggio, così andata è del tipo Paga e conterrà G che deve pagare e la quantità da pagare;
6. *virtual void eseguiAzione(const Ritorno & r)*, l'azione da fare è quella di far pagare al giocatore contenuto in r.giocatore il costo del pedaggio, invocando pagaPedaggio su tale giocatore;
7. *void pagaPedaggio(Giocatore*)*, è un metodo PRIVATO della classe CNedTasPrivata, sottrae al giocatore con cui viene invocata quotaPedaggio ghiande;
8. *void liberaCasella(Giocatore*)*, è un override, per gli oggetti di questa classe liberare la casella vuol dire rilasciare le proprie quote, togliersi dagli azionisti e diminuire la quotaPrivata della società della quantità che abbiamo restituito al pubblico;
9. *int valoreCasella(Giocatore*)*, è un altro override in quanto in questo tipo di caselle il valore è dato dal numero di quote possedute dal giocatore moltiplicate per costoQuota ghiande.

CNEDPRIVATA

Questa classe deriva da CNEdificabile e rappresenta le caselle dalle quali è possibile acquistare o vendere quote di oggetti della classe CNEdTasPrivata, ogni oggetto di CNEdPrivata possiede un puntatore ad un oggetto della classe CNEdTasPrivata e permette di interagire con quellà società. Dobbiamo quindi aggiungere un campo dati PRIVATO[5]: *CNEdTasPrivata**. CNEdPrivata possiede anche due metodi PRIVATI che servono di appoggio ad eseguiAzione, non devono quindi essere raggiungibili dall'esterno, ma vediamo in dettaglio:

```
private:
    1.  void acquistaQuota(Giocatore*, int);
    2.   void vendiQuota(Giocatore*, int);
public:
    3.   CNEdPrivata(string, CNEdTasPrivata*, int, int);
    4.  virtual Andata sceltaOpzione(Giocatore*);
    5.   virutal void eseguiAzione(const Ritorno&);
```

1. *void acqustaQuota(Giocatore*, int)*, è un metodo PRIVATO, non fa altro che invocare sulla società che l'oggetto di invocazione gestisce il metodo aumentaAzionista(Giocatore *, int), quando invochiamo questo metodo siamo sicuri che la società si possa ancora privatizzare del valore specificato e che il giocatore abbia abbastanza denaro;
2. *void vendiQuota(Giocatore*, int)*, è un metodo PRIVATO, che invoca sulla società gestita dall'oggetto di invocazione il metodo decrementaAzionista(Giocatore*, int) per il giocatore e il valore intero con cui viene invocata, quando chiamiamo questa funzione siamo sicuri che il giocatore abbia almeno le quote specificate della società;
3. *CNEdPrivata(string, CNEdTasPrivata *, int, int)*, costruisce il sotto-oggetto di tipo CNEdificabile con string e i due parametri interi, dopodiché costruisce il puntatore alla società gestita;
4. *virtual Andata sceltaOpzione(Giocatore*)*, questa volta nell'oggetto di tipo andata da ritornare bisogna inserire più informazioni. Viene costruito un oggetto Andata di tipo Priv,

5 Anche in questo caso abbiamo un campo dati e due metodi privati, questo sempre perché nella nostra gerarchia non si è resa necessaria una derivazione da CNEdPrivata.

questo oggetto dovrà contenere tutte le informazioni per informare la grafica (e quindi l'utente) su quanto ancora può acquistare o vendere della società ed in che percentuale questa sia posseduta dagli altri utenti. Andata conterrà oltre al giocatore e alla casella di invocazione: il costo di una quota della società, la quota della società che già appartiene al giocatore, la quota massima che il giocatore potrà possedere considerando quella che già possiede e quella che le sue ghiande gli permetterebbero di compare e la percentuale già privatizzata della società. Con tutte queste informazioni il PopUp che viene costruito con un oggetto di questo tipo potrà permettere al giocatore di acquistare quote se la società è privatizzata a meno del 50%, potrà altrimenti fermarlo nel caso contrario e nel caso in cui non avesse abbastanza ghiande, come vediamo tutte le informazioni sulla fattibilità delle operazioni suddette sono calcolate dalla logica e passate alla grafica tramite Andata, grafica che non deve fare altro che interpretare queste informazioni;

5. *virtual void eseguiAzione(const Ritorno&)*, è il momento di far eseguire le operazioni selezionate dal giocatore, con lo schema di controllo adottata in sceltaOpzione siamo sicuri che l'oggetto di tipo Ritorno conterrà un giocatore e un valore in r.value (sia esso di aumento o di sottrazione azioni) accettabile, se il valore è positivo le quote vanno aumentate con acquistaQuota, altrimenti vanno diminuite con vendiQuota;

PRIGIONE

Questa classe deriva pubblicamente da CNEdificabile e come dice il nome rappresenta quelle caselle dove un giocatore può essere messo in prigionia da un'ulteriore casella della classe tribunale. Oggetti della classe Prigione vengono costruiti soltanto nella modalità legale.
Questa classe necessita come campo dati privato di un vector di interi grande quanto il numero di giocatori che rappresenti per ciascun giocatore il numero di turno che questo deve ancora scontare in prigione.
Questo *vector<int> turniPrigione* ci permette anche di conoscere quali giocatori siano imprigionati (quelli la cui casella nel vector ha valore maggiore di zero) e quelli liberi (quelli con valore a 0). Prigione ridefinisce inoltre i metodi sceltaOpzione ed eseguiAzione:

```
1. Prigione(string, int, int);
2. virtual Andata sceltaOpzione(Giocatore *);
3. void incarcera(int, int);
```

I precedenti metodi sono tutti PUBBLICI:

1. *Prigione(string, int, int)*,[6] il costruttore invoca con tutti i suoi parametri il costruttore del sotto-oggetto di tipo CNEdificabile, abbiamo deciso di utilizzare un parametro di tipo stringa per non chiamare tutti gli oggetti di questa classe "Prigione" questo allo scopo di permettere di estendere il gioco, volendo, con più di un carcere con nomi diversi o magari ottenere successive derivazioni del tipo carcere di massima sicurezza. Il secondo intero viene utilizzato anche per definire la dimensione del vector turniPrigione;
2. *virtual Andata sceltaOpzione(Giocatore *)*, è necessario descrivere questo override con attenzione, questa funzione può essere chiamata da TurnoLogico, come vedremo, in due frangenti: dopo lo spostamento di un giocatore (cioè quando un giocatore arriva ad un oggetto della classe prigione) oppure preventivamente, cioè prima che un giocatore venga fatto spostare a partire da un oggetto di tipo Prigione. Nel primo caso sceltaOpzione ritorna un oggetto Andata di tipo Null, sulla prigione non si ha niente da fare, nel secondo caso bisogna valutare il turni di prigionia: se il giocatore ha ancora turni da scontare in

6 Abbiamo voluto inserire un campo dati di tipo string per permettere la creazione di più Prigioni con nomi diversi.

vector<int> turniPrigione ne decrementiamo uno e ritorniamo un Andata di tipo Ferm con i turni ancora da scontare, se invece il turni sono a zero anche in questo caso ritorniamo un oggetto di Andata di tipo Null, viene utilizzato da TurnoLogico per essere sicuro che il giocatore si può muovere;

3. *void incarcera(int,int)*, questa funzione prende il primo intero e dopo aver controllato che sia un valore valido per un indice di giocatore setta il numero di turno che quel giocatore dovrà passare nell'oggetto di invocazione al secondo intero;

A questo punto potremmo domandarci come mai manca un override della funzione virtual void eseguiAzione(const Ritorno &), questo tipo di operazione non viene mai invocata su un oggetto di questa classe, non avrebbe senso perché non si possono svolgere operazioni su questa classe, possiamo quindi ereditare il metodo da CNEdificabile, in quanto anche su quella classe il metodo non faceva niente.

TRIBUNALE

Questa classe deriva pubblicamente da CNEdificabile e rappresenta le caselle che sono in grado di gestire l'incarceramento in una prigione. Ad ogni oggetto di tipo tribunale va quindi associata un oggetto di tipo Prigione da gestire, ecco perché Tribunale a un campo dati PRIVATO *Prigione * prigione*.
Non occorre ridefinire il distruttore, va bene quello standard[7].
Vediamo ora i metodi, tutti PUBBLICI, offerti dalla classe Tribunale:

```
1. Tribunale(Prigione *, int, int);
2. virtual Andata sceltaOpzione(Giocatore *);
3. virtual void eseguiAzione(const Ritorno&);
```

4. *Tribunale(string, Prigione*, int, int)*[8], con i parametri string ed i due interi viene costruito il sotto-oggetto CNEdficabile, mentre Prigione* è il puntatore all'oggetto di tipo Prigione controllato;
5. *virtual Andata sceltaOpzione(Giocatore * G)*, questa funzione ritorna un oggetto della classe Andata di tipo Trib contenente naturalmente G, che è arrivato alla casella, si veda Andata;
6. *virtual void eseguiAzione(const Ritorno& r)*, questa funzione prende r.giocatore, lo sposta sulla casella della prigione e lo incarcera tramite il metodo di Prigione incarcera(int,int) per il valore di turni specificato in r.value;

POLIZIA

Anche questa classe deriva pubblicamente da CNEdficabile, rappresenta caselle in cui al passaggio si deve pagare un'ammenda e si può essere inoltre tenuti una notte in stato di fermo con probabilità del 50%. Non è necessario ridefinire il distruttore quello standard va bene.
Polizia possiede quindi un campo dati PRIVATO e costante *const int ammenda*, possiede inoltre due vector di booleani sempre PRIVATI *vector<bool>fermo* e *vector<bool>appenaArrivato* e fornisce i seguenti metodi pubblici:

```
1. Polizia(string, int, int, int);
2. virtual Andata sceltaOpzione(Giocatore *);
3. virtual void eseguiAzione(const Ritorno&);
```

7 L'oggetto puntato dal campo dati di tipo Prigione* viene elimitato da TurnoLogico alla chiusura del gioco insieme a tutte le altre Caselle Logiche.
8 Anche in questo caso abbiamo voluto inserire un campo dati di tipo string per permettere la creazione di più tribunali con nomi diversi che controllino diverse prigioni.

1. P*olizia(string, int, int, int)*, con la stringa e i primi due int viene costruito il sotto-oggetto, l'ultimo intero rappresenta l'ammenda, il secondo intero è anche utile per sapere la dimensione dei vector fermo e appenaArrivato, fermo è costruito a FALSE mentre appenaArrivato è costruito a TRUE;
2. *virtual Andata sceltaOpzione(Giocatore * G)*, occorre prestare particolare attenzione a questo metodo, come valeva per sceltaOpzione nella classe Prigione questo metodo può essere invocato preventivamente da TurnoLogico per conoscere la situazione del giocatore. appenaArrivato, indica se un giocatore è appena approdato sulla casella, partiamo con appenaArrivato a true per tutti i giocatori perché sicuramente appena TurnoLogico li trova su un oggetto di tipo prigione non sarà certo perché erano imprigionati ma perché ci sono finiti. Quando invochiamo preventivamente sceltaOpzione ci interessa se il giocatore si trova già in stato di detenzione, nel qual caso viene ritornato un oggetto Andata di tipo Ferm con turni di fermo a 1 (si veda Andata). Qualsiasi altro valore dell'oggetto di ritorno non viene tenuto in considerazione dall'analisi preventiva. Se il giocatore invece ha valore appenaArrivato==true vuol dire che è appena capitato sulla polizia, dobbiamo quindi ritornare un oggetto Andata di tipo Polz. Se un giocatore non si trova in stato di fermo e non è appena arrivato sulla casella vuol dire, per l'analisi preventiva, che può essere spostato, viene infatti ritornato un'Andata di tipo Null, in questo caso la funzione mette anche appenaArrivato del giocatore a true, infatti sicuramente la prossima volta che il giocatore si troverà sulla casella sarà perché ci è appena arrivato e non perché è stato fermato;
3. *virtual void eseguiAzione(const Ritorno &r)*, questa funzione viene invocata quando il giocatore in r è appena arrivato sulla casella e ha anche lanciato la moneta per sapere se dovrà stare in Fermo e dovrà pagare l'ammenda, se r.value==1 il giocatore sarà Fermato e pagherà altrimenti sarà lasciato andare;

Estendibilità

La gerarchia delle classi logiche è completamente estendibile. La scelta di derivare tutte le caselle da un'unica classe virtuale è stata scontata. C'era la necessità di esprimere due tipo diversi di caselle, quelle edificabili e quelle non edificabili, entrambe fornivano sulla casella operazioni diverse, Cbase doveva essere ed è un punto di partenza per entrambe. Inoltre la decisione è stata presa nell'ottica di utilizzare solamente il binding dinamico in tutto il progetto, senza avere la necessità di fare type checking. Siamo riusciti nel nostro intento, anche per mezzo delle strutture Andata e Ritorno, grazie alla definizione delle funzioni sceltaOpzione ed eseguiAzione che sono virtuali pure in Cbase e costringono cosi alla concretizzazione in CEdificabile e CNEdficabile. Nelle successive classi derivate di questi metodi va fatto l'over-ride per consentire così alle caselle di permettere ed eseguire operazioni su se stesse. Estendere questa gerarchia di classi vuol dire semplicemente derivare da una delle caselle esistenti. Oppure nel caso in cui si volesse creare un interno nuovo ramo di derivazione è possibile derivare semplicemente da CBase. Per vedere come l'aggiunta di altre caselle non condizioni minimamente il gioco dobbiamo analizzare il comportamento che hanno sulle caselle POPUP e TURNOLOGICO:

- TURNOLOGICO possiede il vector di CBase* ovvero di puntatore a tutte le caselle logiche, invoca su questi puntatori la funzione sceltaOpzione(), ad occuparsi di invocare la corretta funzione ci pensa il binding dinamico, siamo sicuro che l'oggetto di tipo Andata ritornato sarà quello corretto;
- POPUP invoca, dopo che l'utente ha dato conferma, la funzione eseguiAzione() sulla casella di appartenenza, anche in questo caso POPUP invoca questa funzione su un puntatore a CBase, si preoccuperà quindi in binding dinamico a reperire la corretta funzione da invocare, siamo così sicuri che siano svolte le corrette operazioni sulla corretta casella.
- Conviene citare altre funzioni che utilizzando in binding dinamico, come vedremo nella sezione GIOCATORE questa classe possiede un vector dei puntatore a CBase per le caselle possedute e un metodo contaAveri() che invocato su un giocatore restituisce le ricchezze mobili ed immobili che il giocatore possiede. Per fare questo viene invocata la funzione

valoreCasella() su ogni proprietà del giocatore, essendo questa un metodo virtuale definito in CBase sarà il dinamic binding ad invocare il corretto override a seconda del tipo dinamico del puntatore;

- L'ultimo utilizzo del dinamic binding per le caselle logiche lo abbiamo quando un giocatore muore, la funzione liberaProprieta() di giocatore invoca su ogni singolo possedimento la funzione virtuale liberaCasella(), anche in questo caso il binding dinamico si occupa dell'identificazione della giusta funzione.

Nel derivare una nuova classe da CBase dobbiamo ricordarci di concretizzare i metodi virtuali puri mentre derivando dalle altre classi è possibile ereditarli già concretizzati.

Indipendenza
Siamo stati molto attenti a mantenere l'intera gerarchia delle classi logiche totalmente indipendente dalla grafica. In CBase abbiamo visto come il nome della casella sia memorizzato come una stringa, questo nome come vedremo viene utilizzato da CasellaGrafica per rendere la visualizzazione della casella. Inizialmente il nome era memorizzato come una QString ma ci siamo fortunatamente resi conto che questo rendeva la logica dipendente dalle librerie grafiche utilizzate, dovevamo infatti includere l'header QString. Abbiamo così deciso di abbandonare le QString (il discorso è analogo a quello fatto in GIOCATORE) e di utilizzare le std::string per il nome del giocatore. Naturalmente abbiamo bisogno di una funzione ausiliaria che converta string in QString, ma ne parleremo in CASELLAGRAFICA e successivamente nella sezione TRANSLATE.
Non ci sono e non ci devono essere dipendenze nella direzione della grafica, la gerarchia delle caselle è completamente autosussistente.

GIOCATORE

Giocatore è la classe che concretizza l'idea del giocatore logico. Sono gli oggetti di questa classe che contengono tutte le informazioni riguardanti il giocatore.

```
private:
 const string nomeGiocatore;
 int denaro;
 int casellePercorse;
 const int numGiocatore;
 bool morto;
 vector<CBase *> possedimenti;

public:
 Giocatore(string, int);
 int getDenaro() const;
 string getNomeGiocatore() const;
 int getNumGiocatore() const;
 int getCasellePercorse() const;
 bool eMorto() const;
 vector<int> getIndiciPossedimenti() const;
 void aumentaDenaro(int);
 int decurtaDenaro(int);
 void aumentaPossedimenti(CBase*);
 void decurtaPossedimenti(CBase*);
 void avanza(int);
 int contaAveri();
 void liberaProprieta();
 void cameraCommercio(int);
```

1. La parte privata della classe racimola le informazioni che costituiscono l'idea di giocatore logico:
   - *const string nomeGiocatore* è il nome che l'utente si è dato per rappresentare la sua pedina. È un campo costante in quanto non andrà mai modificato;
   - *int denaro* rappresenta la quantità di ghiande possedute dal giocatore;
   - *int casellePercorse* ricorda di quante caselle è avanzato il giocatore da quando ha iniziato la partita. In pratica continua a sommare il risultato del dado che tira.
     Unica eccezione si verifica in modalità legale in quanto ogni volta che un giocatore finisce in tribunale viene retrocesso alla prigione ed il numero casellePercorse viene decrementato (viene passato un parametro negativo invece che positivo). Il valore da decrementare può essere calcolato facendo la sottrazione tra gli indici delle caselle.
     Con questo valore è possibile risalire all'indice della casella in cui si è semplicemente calcolando casellePercorse modulo NumeroDelleCaselle, numero che non è fissato ma può essere reperito, dove necessario, dalle size del vector delle Caselle;
   - *const int numGiocatore* è un numero sequenziale assegnato all'inizio del gioco. È costante in quanto non verrà mai cambiato durante il corso del programma. Viene utilizzato per semplificare le operazioni di indicizzazione quando ne sia necessario;
   - *bool morto* è un parametro booleano che segna se un giocatore è morto oppure no;
   - *vector<CBase *> possedimenti* è un vettore che ricorda, attraverso i suoi puntatori, in quali caselle il dato giocatore è invischiato economicamente, sia per possedere terreno o albergo, sia per possedere quote di azioni. Viene utilizzato in seguito per le funzioni liberaProprieta e contaAveri.

2. Vi sono anche due valori globali che non compaiono nel corpo di Giocatore.
   Sono stati definiti esternamente in modo da rendere estremamente semplice ed immediata la loro sostituzione con altri valori (per rendere più competitivo o più facile il gioco):
   - la quantità di ghiande con cui ogni giocatore inizia la partita;
   - la quantità di ghiande che ogni giocatore riceve per ogni passaggio per il via;

3. Ci sono dei membri pubblici che sono dei getters, e poiché i dati sono privati e sono necessari anche ad altri oggetti, tali funzioni sono d'obbligo:

- *int getDenaro() const*, ritorna la quantità di ghiande posseduta dal giocatore;
- *string getNomeGiocatore() const*, ritorna il nome del giocatore;
- *int getNumGiocatore() const*, ritorna il numero che identifica il giocatore;
- *int getCasellePercorse() const*, ritorna il numero di caselle percorse;
- *bool eMorto() const*, informa se il giocatore è morto oppure è vivo;
- *vector<int> getIndiciPossedimenti() const*, ritorna un vector i cui elementi sono indici delle caselle possedute dal giocatore. Viene utilizzato per fare l'aggiornamento ponderato delle caselle quando il giocatore muore.

4. Ci sono due metodi per modificare il numero di ghiande:
   - *void aumentaDenaro(int n)*, incrementa di n ghiande le scorte del giocatore;
   - *int decurtaDenaro(int n)*, decrementa di n ghiande le scorte del giocatore.
     Questa funzione si preoccupa altresì di far morire un giocatore, esaminando se il giocatore possiede abbastanza ghiande.
     Ritorna il numero di ghiande sottratti dal giocatore: se il giocatore rimane vivo il numero di ghiande sottratte sarà uguale a quello della penale, se il giocatore muore vuol dire che il numero ritornato è necessariamente minore a quello che ci si aspetterebbe. È necessario il tipo di ritorno proprio per tal motivo: quando un giocatore muore per aver dovuto pagare un pedaggio troppo oneroso, il giocatore beneficiante del pedaggio deve ricevere la giusta quantità di denaro posseduta dal malaugurato giocatore deceduto;

5. Vi sono due metodi che gestiscono il vector possedimenti:
   - *void aumentaPossedimenti(CBase *c)*, viene utilizzata per inserire un nuovo elemento nell'array dei possedimenti.
     Viene invocata quando si compra un terreno. Non viene invocata quando si edifica l'albergo poiché per costruire l'albergo è necessario possedere il terreno, il che implica che questa funzione sia già stata eseguita per la casella in questione. Il modificare l'ipoteca non conta poiché non aumenta la quantità di beni posseduta dal giocatore.
     Viene invocata quando il giocatore acquista una percentuale di azioni di una data società. La casella che invocherà tale funzione si preoccuperà di stabilire che il giocatore non possegga già delle azioni e quindi, conseguentemente, un entrata nel vector in questione;
   - *void decurtaPossedimenti(CBase *c)*, viene utilizzata per eliminare un elemento dall'array dei possedimenti.
     Viene invocata questa funzione quando un giocatore estingue la sua partecipazione azionaria, sarebbe un'inconsistenza logica lasciare che tra i possedimenti di un dato giocatore figuri una società pubblica se non possiede più parte delle sue azioni poiché le ha rivendute.
     CBase *c è il puntatore che deve essere eliminato nel vector possedimenti.

6. *void avanza(int x)*, viene utilizzato per tenere aggiornato il numero di caselle percorse dall'inizio del gioco. Può anche farle diminuire in caso il giocatore debba retrocedere verso la prigione;

7. *int contaAveri()*, ritorna quella che è stata definita la quantità di ricchezza di un giocatore.
   L'intero ritornato indica la quantità in ghiande dei suoi beni mobili ed immobili.
   Altro non fa che sommare alla quantità di ghiande liquide in possesso al giocatore, l'intero ritornato dalle invocazioni di casella->valoreCasella(giocatorex) per ogni casella in cui il giocatorex possiede qualcosa. I puntatori alla casella sono reperiti dal vector possedimenti.
   Questa funzione viene invocata per ogni giocatore in vita solo nel caso in cui un giocatore superi il numero di giri massimo consentito;

8. *void liberaProprieta(),* elimina il giocatore come proprietario o detentore di ricchezza dalle caselle da lui possedute. La funzione viene invocata nel caso in cui il giocatore muoia.
   Riesce a reperire le informazioni necessarie dal vector possedimenti: ad ogni elemento dell'array viene applicata la funzione casella->liberaCasella(giocatorex);

9. *void cameraCommercio(int n),*: è la funzione che calcola se il giocatore è appena passato per il via. Nel caso il giocatore abbia iniziato un nuovo giro gli viene fornito il normale bonus di ghiande.

10. *Giocatore(string nome, int n),* è il costruttore. Con le informazioni passategli come parametri inizializza il nome del giocatore e l'indice del giocatore.
    Le altre informazioni sono reperite in maniera standard dalle variabili globali (vedi punto 2).

Sulla prigionia di un giocatore:
Si è presentata la scelta di inserire un flag booleano nella classe Giocatore per identificare se il giocatore fosse in prigione (o comunque dovesse saltare dei turni per qualche motivo) in modo da poter gestire la situazione: se un giocatore è in prigionia non deve tiare il dado, e questo bisogna saperlo prima di dare al giocatore la possibilità di tirare il dado.
Si è deciso che questo flag non sarebbe stato coerente con la classe poiché la possibilità di venire fermato per tot turni è vincolata da una singola modalità di gioco su 4.
Fornire a questa classe, che viene utilizzata da tutte le modalità possibili, un valore booleano che veniva ignorato per la maggior parte delle volte, se non addirittura utilizzato inutilmente ad ogni turno di ogni giocatore in qualsiasi modalità di gioco per capire se il giocatore era in satto di fermo (anche se la modalità di gioco non lo permetteva) è stato ritenuto un motivo sufficiente per non dotare la classe Giocatore di questo flag.

Estendibilità
Qualsiasi informazione aggiuntiva che si voglia fornire al giocatore non compromette in alcun modo questa struttura: basta definire il valore desiderato e possibilmente fornire gli adeguati metodi di modifica e/o getters.
Nel caso questi nuovi valori causassero modifiche ai valori preesistenti basterebbe andare a modificare gli opportuni metodi. Questo non sarebbe complicato poiché ciascun metodo già definito influisce tendenzialmente su di un solo parametro (o per lo meno su di un gruppo molto ristretto). Questa indipendenza tra parametri permette una più semplice gestione degli stesse anche nell'ottica dell'estendibilità

Indipendenza
Non v'è molto da dire: questa è una classe puramente logica e non ha nessun contatto diretto con la parte grafica. Detiene solamente dei valori sensibili di ogni giocatore e li modifica oculatamente secondo il fabbisogno.

BOLLA.H

I files Bolla.h e Bolla.cpp contengono le strutture dati che permettono il dialogo tra l'interfaccia grafica attiva e la struttura logica.

ANDATA

Andata è la classe che racimola le informazioni dalla struttura logica (le caselle su cui si ferma un giocatore) e le passa all'interfaccia grafica attiva (il PopUp).
In questa sezione si spiehgerà l'utilizzo dei vari campi di questa classe per le diverse applicazioni.
Per capire come questi dati sono stati raccolti e computati riferirsi alla sezione della struttura logica correlata.
Tutti i campi sono pubblici per risparmiare chiamate a funzioni getters: dato che questa classe è stata creata allo scopo di passare e rendere disponibili tali informazioni ad altri oggetti, non ha senso renderle private.

```
public:
 enum Tipo{Null, Paga, Edif, Priv, Ferm, Polz, Trib, Mort, Vitt};
 Tipo tipo;
 Giocatore *giocatore;
 CBase *casella;
 bool edificabili[4];
 int costo;
 int attuale;
 int limite;
 int totale;
 Andata(Tipo t, Giocatore *g=0, CBase *c=0, int opz=0, int cs=0, int att=0, int lim=0, int tot=0);
 ~Andata();
```

1. Il tipo *enum Tipo* è stato creato per far capire al PopUp che tipo di interfaccia grafica proporre all'utente:
   - *Null* : definisce che il giocatore è su di una casella in cui non ci sono azioni utili che possa fare. Probabilmente è la modalità di gioco scelta che impone questo valore.
   - *Paga* : definisce che il giocatore è su di una casella in cui deve pagare un pedaggio o un'ammenda a qualcuno (giocatore o banca);
     Vale per caselle CEdificabili, CEdIpoteca che possiedono già un proprietario diverso la giocatore in questione, CNEdTassa o CNEdTasPrivata;
   - *Edif* : definisce che il giocatore è su di una casella Edificabile di cui può diventare proprietario, su cui può costruire un albergo, in cui può modificare l'ipoteca (se la modalità di gioco lo permette) o su cui non può far nulla poiché ha già costruito l'albergo.
     Vale per caselle CEdificabili, CEdIpoteca;
   - *Priv* : definisce che il giocatore è su di una casella in cui ha la possibilità di divenire socio di una società.
     Vale per caselle CnedPrivata;
   - *Ferm* : definisce che il giocatore è in stato di fermo in prigione o in polizia. Viene utilizzata anche come informativa preventiva per TurnoLogico: è il modo per capire se lasciar lanciare il dado ad un giocatore o se bloccargli tale possibilità perché è in stato di fermo.
     Vale per caselle CLegali Prigione e Polizia;
   - *Polz* : definisce che il giocatore è sulla casella di tipo Polizia e che dovrà quindi tentare la sorte.

Vale per la casella CLegali Polizia;
- *Trib* : definisce che il giocatore è sulla casella di tipo Tribunale e che quindi dovrà tentare la sorte.
  Vale per la casella CLegali Tribunale.
- *Mort* : definisce che il giocatore è morto. Il giocatore può morire solo perché deve pagare una somma in ghiande che lui non possiede.
  Vale per le caselle CEdificabile/CEdIpoteca in cui il giocatore non è il proprietario e su caselle CNEdTassa/CNedTasPrivata;
- *Vitt* : definisce che un giocatore ha vinto. Il giocatore può vincere perché gli altri sono morti (quindi conseguente ad un tipo Mort vedi sopra) o perché è stato superato il limite massimo di giri.
  Vale per qualsiasi casella.

2. *Tipo tipo* indica che tipo di informazione grafica dovrà produrre il PopUp;

3. *Giocatore *giocator*e indica il giocatore che ha scatenato l'evento. Il puntatore viene utilizzato soprattutto per ricavare il nome del giocatore e per poi costruire un oggetto di tipo Ritorno;

4. *CBase *casella* è il puntatore alla casella su cui è presente il giocatore. Viene utilizzato da PopUp per scatenare l'aggiornamento logico dopo che il giocatore ha compiuto le sue scelte. Posso altresì recuperare il nome del proprietario della casella e il nome della casella;

5. *bool edificabili[4]* è un array di booleani per capire quale delle azioni disponibili su di una casella edificabile è possibile eseguire (una esclude le altre)
   - *edificabili[0]* : non è definito (vedere costruttore di Andata per informazioni maggiori)
   - *edificabili[1]* : è possibile comprare il terreno
   - *edificabili[2]* : è possibile comprare l'albergo
   - *edificabili[3]* : è possibile modificare l'ipoteca sull'albergo

6. *int costo* è un valore che assume diversi significati a seconda della casella su cui ci si trova e quindi è parametrico rispetto a tipo:
   - Tipo==Paga : determina il costo dell'ammenda o del pedaggio da dover pagare
   - Tipo==Edif : determina il costo dell'albergo o del terreno
   - Tipo==Priv : determina il costo di una quota del 10% della società relativa
   - Tipo==Trib : determina il numero di turni da dover saltare
   - Tipo==Polz : determina il numero di turni da dover saltare
   - Tipo==Ferm : determina il numero di turni da dover saltare
   - Tipo==Vitt : determina l'ammontare in ghiande del giocatore che ha vinto

7. *int attuale* è un valore che assume diversi significati a seconda della casella su cui ci si trova e quindi è parametrico rispetto a tipo:
   - Tipo==Edif : la percentuale attuale di ipoteca sull'albergo
   - Tipo==Priv : la percentuale attuale di azioni detenute dal giocatore
   - Tipo==Vitt : il tipo di vittoria conseguita dal giocatore (si è preferito utilizzare un campo dati preesistente invece di creare un campo booleano solo per lui), vale 1 se il giocatore ha vinto per decesso altrui, 0 se il giocatore ha vinto per superamento del limite dei giri di tabellone

8. *int limite* è un valore che assume diversi significati a seconda della casella su cui ci si trova e quindi è parametrico rispetto a tipo:

- Tipo==Edif && (edificabili[1] || edificabili[2]) : vale come valore booleano. Indica se il giocatore possiede abbastanza denaro per comprare il terreno o l'albergo;
- Tipo==Edif && edificabili[3] : indica la percentuale minima raggiungibile per il dato giocatore. Non può raggiungere un valore inferiore della sua ipoteca poiché non avrebbe abbastanza ghiande per riscattarla;
- Tipo==Priv : indica la percentuale massima raggiungibile per il dato giocatore. Non può raggiungere un valore percentuale maggiore poiché non avrebbe abbastanza ghiande per comprare tutte le azioni.

9. *int totale* determina la percentuale totale delle quote degli azionisti (giocatore attuale compreso).
   Vale solo per caselle CNEdPrivata;

10. Il costruttore assegna semplicemente i valori ai campi dati. Tutti i parametri hanno valori di default in modo da poter snellire la sua invocazione: infatti la sequenza con cui son stati dichiarati i valori ha lo scopo di racimolare i valori più utilizzati all'inizio in modo da poter utilizzare i valori di default (es: in CEdificabile è sufficiente un costruttore a 4 parametri che sono proprio i primi quattro, mentre CNedPrivata ha bisogno di 6 parametri e quindi utilizza tutti i valori meno uno che purtroppo non si trova alla fine, ma che per lo meno non son sparpagliati a casaccio).
    Una cosa da notare è che il costruttore non crea l'array edificabili sullo heap, ma che prende come valore un intero che indicizza quale elemento settare a true. Questo spiega perchè edificabili[0] non è stato dichiarato di nessuna utilità, serve solo per evitare incomprensioni o malaugurati errori: infatti edificabili[0] segna che l'array non è stato utilizzato (nel caso l'array fosse allocato sullo heap edificabili[0] equivarrebbe ad un puntatore nullo);

11. Il distruttore ha corpo nullo poiché non v'è nulla da deallocare sullo heap: il distruttore standard basta e avanza per una simile classe;

RITORNO

Ritorno è la classe che si preoccupa di fornire alla struttura logica le informazioni di cui ha bisogno per capire che scelta ha eseguito l'utente (che pulsanti ha cliccato in PopUp) per aggiornare e modificare i suoi campi dati.
Il tipo di informazioni che Ritorno trasporta è vincolato dal fatto che la struttura logica non sia completamente cieca, ma che anzi già sappia quello che il giocatore potrebbe fare e che quindi non esegua una ricerca totale bensì cerchi le informazioni necessarie dove sa per certo di trovarle: come Andata è strutturato secondo le sole operazioni che il giocatore può eseguire, allo stesso modo Ritorno fornisce il metodo di distinguere quali delle operazioni disponibili il giocatore ha voluto eseguire senza che ci siano contaminazioni di valori ambigui.

```
public:
 Giocatore *giocatore;
 bool operazioni[5];
 int value;
 Ritorno(Giocatore *g=0, int n=0, int v=0);
 ~Ritorno();
```

1. *Giocatore *giocatore* è il puntatore del giocatore che è arrivato sulla casella in questione e che ha appena eseguito la scelta di cosa fare. Questo puntatore è necessario poiché le caselle, per come sono strutturate, devono ricordarsi solo il loro stato, e non ricordarsi quali tra i giocatori che vi soggiornano sono appena arrivati o quali sono fermi da più turni.

Viene principalmente utilizzato per decrementare la quantità di ghiande che possiede.
Per gli altri scopi di questo puntatore si rimanda alla sezione che tratta la struttura logica delle caselle.

2. *bool operazioni[5]* raggruppa in se tutte le possibili azioni che un giocatore può compiere indipendetemente dalla casella in cui si trova. Ogni elemento determina se il giocatore in questione voglia eseguire l'azione disponibile sulla casella oppure di non fare nulla[9]:
   - operazioni[0] : comprare il terreno
   - operazioni[1] : comprare l'albergo
   - operazioni[2] : modificare l'ipoteca
   - operazioni[3] : pagare la tassa o il pedaggio
   - operazioni[4] : modificare la partecipazione azioniaria di una società

3. *int value* è un valore utilizzato a seconda dell'operazione confermata, quindi è parametrico rispetto ad operazioni[i]:
   - operazioni[0] == true : non definito perchè inutile
   - operazioni[1] == true : non definito perchè inutile
   - operazioni[2] == true : determina se aumentare o decrementare l'ipoteca.
     - value > 0 : aumenta l'ipoteca del value%
     - value < 0 : diminuisce l'ipoteca del abs(value)%
   - operazioni[3] == true : non definito perchè inutile
   - operazioni[4] == true : determina se aumentare o decrementare le azioni del giocatore.
     - value > 0 : compra value% delle azioni della società
     - value < 0 : rivende abs(value)% delle azioni della società (che naturalmente il giocatore deve possedere: calcolo fatto dalla casella quando crea Andata)

4. Il costruttore assegna ai campi dati semplicemente il valore corrispondente alla convenzione sopra descritta a seconda della decisione presa dal giocatore.
   Unica cosa degna di nota è che l'array operazioni può avere un solo valore true corrispondente al parametro n passato come secondo valore al costruttore. Nel caso questo parametro sia -1 nessun elemento dell'array viene settato a true (servirebbe per coerenza nel caso che si voglia chiamare da PopUp casella->eseguiAzione(...) anche quando il giocatore ha deciso di non fare nulla. Per ulteriori informazioni andare alla sezione PopUp)
5. Il distruttore ha corpo vuoto poiché nessun dato è allocato sullo heap. V'è solo un puntatore a giocatore che non deve essere distrutto in quanto l'oggetto giocatore da esso puntato esisteva prima e dovrà esistere anche dopo la distruzione di Ritorno.

Estendibilità
Nell'ottica dell'estendibilità queste strutture non pongono nessun limite.
Son state progettate per un dato schema di gioco, ma nel caso si trovassero altre azioni da svolgere basterebbe modificare tale strumento inserendo nuovi campi dati (nel caso quelli già presenti non soddisfacessero al nuovo fabbisogno). L'inserimento di nuovi dati con valori di default permetterebbe oltretutto di non dover ritoccare la lista d'inizializzazione di ogni singola precedente occorrenza di tali oggetti a patto che i nuovi valori vengano inseriti nella lista del costruttore dopo gli ultimi già presenti.
Indipendenza
Il motivo della creazione di queste strutture è proprio per tener separate la parte logica da quella grafica. Nel caso una delle due venisse modificata per qualche motivo, basterebbe attenersi alle convenzioni definite per ogni valore per non compromettere la controparte non modificata.

9 Si ricorda infatti che in ogni istante ogni giocatore ha la scelta tra al massimo due azioni: eseguire l'azione dominante della casella (comprare terreno, acquistare azioni, pagare il pedaggio etc., in cui un'azione esclude l'altra) ed in caso sia possibile non fare nulla

TURNOLOGICO

TurnoLogico è la classe che gestisce la proprietà del turno ed il susseguirsi delle azioni nel turno e sulle caselle.

Analizziamo separatamente i campi dati PRIVATI:

```
1.  int indiceGiocatore;
2.  const int lmax;
3.  vector<Giocatore*> GiocatoriLogici;
4.  vector<CBase*> CaselleLogiche;
5.  int result;
```

1. *int indiceGiocatore*, naturalmente TurnoLogico essere a conoscenza del giocatore attivo, e cioè il giocatore che per questo turno svolgerà le azioni;
2. *const int lmax*, è il parametro che rappresenta il numero massimo di giri che è possibile fare prima che il gioco si fermi e annunci come vincitore il giocatore che ha raccolto più soldi, nonappena un giocatore supera la soglia degli lmax giri il gioco viene interrotto. È costante perché non verrà mai modificato durante il gioco;
3. *vector<Giocatore *>GiocatoriLogici*, è il vector che contiene tutti i giocatori logici, se non lo possedesse TurnoLogico non potrebbe sapere a chi assegnare il turno successivo;
4. *vector<CBase *> CaselleLogiche*, questo vector permette di spostare i giocatori da una casella all'altra e di effettuare le necessarie operazioni sulla casella sulla quale il giocatore è approdato: fornire le possibili operazioni alla grafica;
5. *int result*, quando dalla grafica viene lanciato il dado il risultato è salvato per mezzo di uno slot in questo campo, in maniera tale che quando sarà invocato il metodo che muove il giocatore sapremmo di quanto spostarlo;

Passiamo ora ad analizzare i metodi della classe TurnoLogico:

```
1.  Giocatore * trovaVincitore() const;
2.   TurnoLogico(const vector<Giocatore*> &, const vector<CBase*> &);
3.   ~TurnoLogico();
4.   Andata muovi();
5.   Andata statoGiocatore();
6.   Andata testPreliminare();
7.   void passaTurno();
8.   void salvaLancio(int);
9.   Giocatore * getGiocatore();  Giocatore* trovaVincitore() const;
```

1. *Giocatore* trovaVincitore() const*, è una funzione PRIVATA e costante, il suo utilizzo è quindi esclusivamente all'interno della classe, viene invocata quando un giocatore supera la quota massima di giri lmax, permette di definire quale tra i giocatori in vita è il più ricco, fa questo servendosi della funzione contaAveri(), metodo della classe Giocatore. Calcolato il giocatore più ricco ne restituisce il puntatore;

2. *TurnoLogico(const vector<Giocatore*> &, const vector<CBase*> &)*, altro non è che il costruttore, il due parametri servono ad inizializzare GiocatoriLogici e CaselleLogiche

mentre result viene costruito e inizializzato a 0 e lmax viene costruito e inizializzato a 10;

3. *~TurnoLogico()*, è il distruttore, si occupa di rimuovere tutti i giocatori logici e tutte le caselle logiche allocati sullo heap;

4. *Andata muovi()*, ha il compito di muovere il giocatore in base al risultato del lancio del dado salvato in result, dopo aver spostato il giocatore per mezzo dei metodi delle caselle logiche rimuoviGiocatore e aggiungiGiocatore invoca sul giocatore cameraCommercio(), metodo della classe Giocatore, per controllare se questo è passato per la Camera di Commercio, nel qual caso il metodo gli accredita delle ghiande. Ma non è ancora finita, bisogna controllare che il giocatore non abbia superato gli lmax giri di tutte le caselle, se li ha superati calcola per mezzo del metodo trovaVincitore() il giocatore più ricco e ritorna alla grafica un'oggetto della classe Andata di tipo Vitt, non definendo il parametro nel costruttore che inizializza il campo dati Andata::attuale, campo dati che di default è messo a 0, indica così una vittoria per fine dei giri (si veda Andata).
   Nel caso in cui ci siano ancora giri disponibili invoca sulla casella di arrivo sceltaOpzioni che ritorna le possibilità che il giocatore in possesso del turno ha sulla cella in cui è arrivato, incapsulate nella classe Andata.
   A sua volta TurnoLogico ritorna quest'oggetto a TurnoGrafico che fornirà l'output grafico per le possibili operazioni;

5. *Andata statoGiocatore()*, questo metodo fornisce alla grafica informazioni relative alle condizioni del giocatore dopo che egli ha interagito tramite il PopUp con la casella, questa interazione potrebbe fare fallire il giocatore quindi sarebbe necessario comunicarne la dipartita, conseguentemente potrebbe essere rimasto un solo giocatore e in questo caso oltre alla dipartita va comunicata la fine dei giochi, infine potrebbe non essere successo nulla nel qual caso la grafica non ha nulla da comunicare, i tre casi sono gestiti sempre per mezzo della classe di interazione Andata:
   1. Giocatore morto, TurnoLogico costruisce una struttura dati Andata di tipo Mort contenente soltanto il puntatore al giocatore deceduto;
   2. Giocatore Morto + Vittoria, TurnoLogioco costrutisce una struttura dati Andata di tipo Vitt contenente l'ultimo giocatori rimasto in vita, i suoi soldi. Inoltre il costruttore viene invocato con il parametro che inizializza attuale a 1, questo sta ad indicare un tipo di vittoria per decesso altrui. Quando viene ritornato un oggetto di questo tipo TurnoGrafico sarà in grado di comunicare la morte del giocatore in turno e la vittoria del giocatore nell'oggetto di tipo Andata;
   3. Giocatore ancora in vita, TurnoLogico ritorna un'oggetto della classe andata con tipo Null, la grafica non fa quindi niente.

6. *Andata testPreliminare()*, questa funzione viene utilizzata dalla grafica soltanto nella modalità legale del gioco, è necessaria infatti per dirci all'inizio del turno (ancora prima di muovere il giocatore) se questo si può effettivamente muovere o se è imprigionato, se il giocatore si trova in una condizione normale viene ritornato un oggetto della classe Andata Null (vedi Andata) altrimenti abbiamo due casi: il giocatore si trova in Prigione oppure in stato di fermo presso la Polizia, come detto nella sezione dedicata a Prigione e Polizia quando invochiamo su una di essere sceltaOpzione() vengono ritornati degli specifici oggetti Andata che ci permettono di determinare univocamente lo stato del giocatore, questi oggetti della classe Andata sono passati alla grafica se indicano uno stato di arresto. Se invece l'oggetto di tipo andata non indica uno stato di fermo testPreliminare ritorna un Andata di tipo Null;

7. *void passaTurno()*, questa funzione incrementa di 1 (nell'aritmetica modulo

GiocatoriLogici.size()) indiceGiocatore, saltando naturalmente i giocatori morti;

8. *void salvaLancio(int)*, è la semplice funzione invocata da TurnoGrafico dopo il lancio del dato, utilizzata per salvare il risultato in result;

9. *Giocatore* getGiocatore()*, è un semplicissimo getter, permette di tenere privato l'indiceGiocatore e il vector dei giocatoriLogici fornendo alla sua invocazione il puntatore al giocatore in possesso del turno;

Estendibilità
Per la gerarchia di classi pensata finora TurnoLogico si comporta in maniera ottima. Se si volessero aggiungere classi di caselle che hanno effetti all'arrivo del giocatore lo si farebbe ridefinendo in ciascuna di queste classi sceltaOpzione ed eseguiAzione, caselle di questo tipo non comporterebbero alcuna necessità di modifica di TurnoLogico. Analogamente caselle che volessero un test preventivo sulla loro condizione prima dello spostamento di un giocatore (esattamente quello che richiedono le stazioni di polizia e le prigioni nella modalità legale) non richiederebbero la modifica di TurnoLogico, che per mezzo della funzione testPreliminare è già in grado di farlo.
Per eventuali altri tipi di caselle o modalità è comunque possibile effettuare derivazioni da TurnoLogico esattamente come è possibile farne da TurnoGrafico, a seconda della modalità di gioco verrebbero così create diverse coppie TurnoLogico-TurnoGrafico.

Indipendenza
TurnoLogico è il cuore del controllo del flusso della parte logica, non effettua operazioni sulle caselle e non effettua operazioni per l'utente, semplicemente si assicura che le operazioni proseguano nella maniera corretta. Per fare tutto questo TurnoLogico non ha bisogno della Grafica, ma occorre precisare questo concetto di indipendenza:
TurnoLogico non opera indipendentemente ma deve sottostare ai ritmi scanditi dall'utente, naturalmente l'utente si interfaccia con la grafica quindi è obbligatorio che la grafica detti i tempi dell'evoluzione nel flusso di gioco controllato da TurnoLogico, per spiegarci meglio alla fine della trattazione di TurnoGrafico è presente uno schema dell'interazione tra i due turni.
TurnoLogico non potrebbe in alcun modo controllare il flusso di esecuzione indipendentemente ma in questo senso ha bisogno della grafica che gli comunichi quando attivarsi per passare alla fase successiva. Questa non è una mancata indipendenza ma è semplicemente una condizione necessaria della programmazione con interfaccia grafica.
TurnoLogico sulle segnalazioni di TurnoGrafico gestisce la corretta sequenzialità delle azioni, TurnoGrafico non è alcun modo cosciente di quello che fa TurnoLogico ma si limita semplicemente a iniziare e poi seguirne l'evoluzione ed eventualmente fornire informazioni grafiche all'utente ma sempre e solo sulle informazioni raccolte e restituitegli da TurnoLogico.
TurnoLogico non dipende dalla grafica, se sostituissimo le librerie Qt con un interfaccia grafica testuale avremmo esattamente lo stesso TurnoLogico.

GRAFICA PASSIVA

GIOCATOREGRAF

Il giocatore grafico fa parte della grafica del nostro monopoli ma in realtà non rappresenta una reale finestra in quanto non deriva da QWidget, GiocatoreGraf è più che altro un contenitore di immagini del giocatore. I suoi campi dati sono due e sono i due puntatori PUBBLICI alle QPixmap che rappresentano le sue immagini;
La necessità di avere due QPixmap con immagini di diversa dimensione deriva dal fatto che utilizziamo l'immagine piccola come pedina sulle caselle mentre l'immagine più grande viene utilizzata in TurnoGrafico e in InfoGiocatori[10].
Per quale motivo utilizziamo due immagini nonostante le librerie grafiche le scalino automaticamente adattandole allo spazio dove vengono inserite? Abbiamo utilizzato due immagini perché la funzione automatica fornita dalle Qt è molto imprecisa e scalare un'immagine grande ad una molto più piccola comporta una notevole perdita di qualità. Abbiamo quindi preferito scalare manualmente alla dimensione necessaria le immagini per mezzo di un software dedicato. Discorso analogo vale per le immagini di sfondo delle CaselleGrafiche quindi non ci ripeteremo.
Detto questo possiamo passare a vedere costruttore e distruttore, che sono gli unici metodi forniti dalla classe GiocatoreGraf:

```
1.  GiocatoreGraf(QString, QString);
2.   ~GiocatoreGraf();
```

1. GiocatoreGraf(QString, QString), è il costruttore, le due stringhe servono rispettivamente a costruire avatar_piccolo e avatar_grande, entrambe le QPixmap sono allocate sullo heap;

2. ~GiocatoreGraf(); distruttore con il compito di deallocare le due QPixmap;

Estendibilità
Qualsiasi modifica del codice non intacca GiocatoreGraf, ad ogni giocatore logico è associato uno grafico, se il giocatori logici aumentano semplicemente aumenteranno anche quelli grafici.

Indipendenza
L'esistenza stessa di una classe GiocatoreGraf che contiene soltanto puntatori a QPixmap e che non deriva da QWidget è una prova degli sforzi di mantenere separate logica da grafica.
Spiegandoci meglio: oggetti di tipo GiocatoreGraf non sono visibili dall'utente servono semplicemente a contenere le immagini utilizzate dalle altre interfacce grafiche. Se non avessimo creato GiocatoreGraf avremmo potuto inserire i puntatori a Qpixmap direttamente nella classe Giocatore, questo tuttavia avrebbe voluto dire distruggere l'isolamento che cerchiamo tra grafica e logica.

10 L'utilizzo esterno alla classe appena citato è anche la motivazione per cui i due puntatori a QPixmap sono pubblici.

LA GERARCHIA DELLE CASELLE GRAFICHE

CASELLAGRAFICA

La classe CasellaGrafica deriva da QWidget e non è altro che la rappresentazione grafica di tutte quelle caselle che non sono edificabili.
Tra i campi dati PRIVATI[11] abbiamo *vector<GiocatoreGraf *> pedine*, che permette di reperire le immagini dei giocatori e di disegnarli se essi sono presenti sulla casella, *const CBase* casella*, che contiene l'array delle presenze e il nome della casella, questo puntatore è costante perché non deve essere possibile modificare casella dalla parte grafica, infine abbiamo due puntatori a Qpixmap, *sfondo* e *sfondoGrande*. Mentre lo sfondo rappresenta lo sfondo piccolo della casella sul tabellone e viene utilizzato soltanto da oggetti della classe CasellaGrafica, lo sfondoGrande come dice il nome stesso rappresenta un'immagine più grande utilizzata da TurnoGrafico per costruire il PopUp, la prima QPixmap può quindi essere privata ma la seconda deve essere pubblica.
Vediamo i metodi PUBBLICI offerti da CasellaGrafica:

```
1. CasellaGrafica(QString, CBase*, const vector<GiocatoreGraf *> &, QWidget * =0);
2. virtual ~CasellaGrafica();
3. virtual void paintEvent(QPaintEvent *);
```

1. *CasellaGrafica(QString, CBase*, const vector<GiocatoreGraf*> &, Qwidget =0)*, è il costruttore, il parametro di tipo QString è il percorso della Pixmap piccola, il percorso della Pixmap grande differisce da questo soltanto per l'ultima lettera del nome, infatti le immagini grandi hanno un 'G' alla fine del nome, così con un'unica stringa di percorso posso recuperare entrambe le immagini, il parametro di tipo QWidget* serve a costrutire il sotto-oggetto di tipo QWidget e rappresenta il parent, invece il vector di puntatori a GiocatoreGraf serve per costruire pedine;
2. *virtual ~CasellaGrafica()*, è il distruttore, viene marcato come virtuale in quanto abbiamo previsto un'ulteriore derivazione di CasellaGrafica, CasellaGraficaEd. ~CasellaGrafica() dealloca entrambe le QPixmap;
3. *virtual void paintEvent(QPaintEvent*)*, il paintEvent disegna il contenuto della CasellaGrafica sul Widget, per prima cosa scrive il nome della casella prendendolo dal campo dati casella[12], dopodiché disegna l'immagine sfondo e infine basandosi sull'array delle presenze in casella disegna i giocatori che si trovano sulla stessa, prendendone le immagini dal vector pedine.
   Come mai il paintEvent è virtuale?
   Innanzitutto come detto abbiamo creato una classe derivata da CasellaGrafica, CasellaGraficaEd, come vedremo TurnoGrafico possiede un vector di puntatori a CasellaGraf e all'avvenire di determinate modifiche deve aggiornare le caselle.
   Facciamo un esempio: è possibile che durante il suo turno un giocatore acquisti un terreno, occorre invocare su casellagrafica un update(), casellaX->update(). Se il metodo paintEvent non fosse virtuale anche nel caso in cui casellaX avesse tipo dinamico CasellaGraficaEd update invocherebbe il paintEvent per le normali CasellaGrafica. Marcando il metodo virtuale all'invocazione casellaX->update() il puntatore this dentro ad update ha tipo statico CasellaGrafica ma tipo dinamico CasellaGraficaEd, viene quindi invocato il paintEvent di CasellaGraficaEd.

11 I campi dati sono privati perché nella classe derivata CasellaGraficaEd non abbiamo necessità di accederli, come vedremo a breve.
12 Il nome della casella è una semplice stringa, occorre utilizzare la funzione di conversione string2QString definita nel file Translate.cpp e descritta nella sezione TRANSLATE.

CASELLAGRAFICAED

Questa classe rappresenta l'interfaccia grafica delle caselle che possono avere un proprietario diverso dalla banca e possono essere inoltre edificate.
È naturale che questa classe derivi da CasellaGrafica, infatti una casella grafica edificabile ha bisogno di tutte le caratteristiche di CasellaGrafica e ha inoltre la stessa base grafica, semplicemente aggiunge la possibilità di stampare a video il proprietario del terreno ed eventualmente una piccola casetta se la casella è edificata.
Per fare quanto detto CasellaGraficaEd ha bisogno di un campo dati che sia di vero tipo *Cedificabile** COSTANTE[13] in quanto con un semplice puntatore a Cbase non sarebbe in grado di capire se la casella è edificata o meno (vedere il metodo creaGioco() di PARTITA per capire come sono creati gli oggetti CasellaGraficaEd).
CasellaGraficaEd aggiunge inoltre altri due campi dati privati *QPixmap* albergo e QPixmap* timbro*. La prima viene disegnata dal metodo paintEvent se la casella è edificata, la seconda viene disegnata insieme al nome del proprietario se la casella ha un proprietario.
Vediamo ora più precisamente i metodi pubblici di CasellaGraficaEd:

```
10.  CasellaGraficaEd(QString, QString, QString, CEdificabile *,const vector<GiocatoreGraf*> &, QWidget* =0);
11.   ~CasellaGraficaEd();
12.   virtual void paintEvent(QPaintEvent * e);
```

1. *CasellaGraficaEd (QString, QString, Cedificabile *, const vector<GiocatoreGraf *>&, Qwidget*=0)*, è il costruttore, con la prima QString, il puntatore a Cedificabile, il vector di puntatori a GiocatoreGraf e il puntatore al parent costruisce il sotto-oggetto di tipo CasellaGrafica, utilizza inoltre il campo Cedificabile* per costruire il suo campo dati casella. La seconda QString serve a costruire la pixmap albergo mentre la terza la pixmap timbro.
2. *~CasellaGraficaEd()*, è il distruttore, nel suo corpo dealloca la pixmap dell'albergo e quella di sfondo, all'uscita dal suo corpo invoca automaticamente il distruttore del sotto-oggetto che fa il resto,
3. *virtual void paint paintEvent(QPaintEvent *)*, per prima cosa come abbiamo accennato prima la base grafica di una CasellaGraficaEd è esattamente una CasellaGrafica normale, quindi prima di tutto invoca il metodo paintEvent del sotto-oggetto CasellaGrafica::paintEvent e successivamente esegue i suoi compiti specifici:
   - Abbiamo deciso a causa della mancanza di spazio di far comparire il nome del proprietario della casella, se questa è edificata, come timbro sull'immagine della casella stessa, il bordi del timbro sono rappresentati dalla QPixmap Timbro che viene disegnata, successivamente dobbiamo disegnare il nome del giocatore rispettando però l'inclinazione dell'immagine Timbro, a questo scopo occorre ruotare il QPainter;
   - Se casella è anche edificata disegniamo anche la piccola immagine dell'albergo in alto a destra.

Le dimensioni delle caselle
Le dimensioni delle caselle sono dipendenti dalle dimensioni delle immagini che ne fanno da sfondo, conseguentemente essendo queste immagini di dimensione fissa anche le caselle devono essere di dimensione fissa, in modo tale da non poter essere sformate lasciando però invariata la dimensione dell'immagine di sfondo. Ciascuna casella occupa 120x130 pixel e questa dimensione condizionerà anche quella di TavoloDaGioco. La disposizione degli oggetti dentro una CasellaGrafica è la seguente: in alto compare il nome della via, in basso compaiono i giocatori che si trovano sulla casella e al centro compare lo sfondo della casella.

13 COSTANTE perché non vogliamo che la grafica modifichi anche solo per errore la logica.

Estendibilità
Le caselle grafiche viaggiano di pari passo con la gerarchia delle caselle logiche, se si dovessero creare ulteriori classi di caselle logiche con particolari necessità di rappresentazione grafica è possibile creare altrettante caselle grafiche, derivando da CasellaGrafica, che, lavorando su un puntatore COSTANTE alla nuova casella logica, rendano al meglio l'interfaccia grafica per quel tipo di casella.
La nostra derivazione si ferma a CasellaGraficaEd in quanto per le nostre necessità era sufficiente la distinzione tra caselle non edificabili ed edificabili ma niente vieta di estendere la gerarchia.

Indipendenza
Per rendere a schermo tutte le informazioni della casella logica abbiamo bisogno di un puntatore che ci permetta di accederne il campi dati. Occorre ricordare che in entrambe le classi il puntatore ad oggetti di tipo logico è un puntatore a oggetto costante, permette quindi di invocare su di esso esclusivamente metodi costanti. Questo accorgimento e l'attenzione che abbiamo prestato nel definire gli identificatori di accesso per le caselle logiche ci permette di essere sicuri che dalla grafica si possano soltanto attingere le informazioni necessarie al disegno ma non si possano in alcun modo raggiungere o ancor peggio modificare informazioni sensibili.
Crediamo quindi che il necessario accesso alla parte logica sia stato così gestito nella maniera più indolore e sicura possibile.
Vogliamo inoltre sottolineare come la logica delle caselle sia all'oscuro di questa dipendenza ed un eventuale cambiamento dell'interfaccia o delle librerie non la condiziona minimamente.

INFO GIOCATORI

InfoGiocatori è una semplice classe che riguarda l'aspetto grafico del gioco.
È stata creata per far in modo che gli utenti possano sempre avere sott'occhio chi è il giocatore attivo, quante ghiande ciascun giocatore possiede, quale è la pedina associata ad un giocatore.
È una classe completamente passiva in quanto deve solo fare il display delle informazioni e deve venire aggiornata costantemente di modo da non fornire informazioni inconsistenti.
InfoGiocatori è creata utilizzando la classe MonoGiocatore, quindi partiamo da questa classe per spiegare il resto.

MONO GIOCATORE

Rappresenta effettivamente un piccolo "Avatar" dell'utente in cui son visibili le informazioni principali: il nome, la pedina, le ghiande, se il giocatore è in vita.

```
public slots:
 void aggiorna();


private:
 Giocatore* GiocatoreLogico;
 QGroupBox *group;
 QLabel *ghiande;
 QLabel *soldi;
 QPushButton *immagine;


public:
 MonoGiocatore(Giocatore *, GiocatoreGraf *, QWidget * =0);
 ~MonoGiocatore();
```

1. I campi dati sono piuttosto banali:
   - *Giocatore *GiocatoreLogico* è un puntatore che indica costantemente un dato giocatore. Tramite questo puntatore è possibile reperire il nome del giocatore ed il quantitativo sempre aggiornato di ghiande possedute.
     È stato scelto di utilizzare un puntatore e non una struttura dati che portasse le informazioni perchè altrimenti sarebbe stato roboante per via della frequenza con cui deve essere aggiornato l'oggetto, ed inutile in quanto il puntatore è semplicemente utilizzato per reperire informazioni e non per modificarle in alcun modo. L'alternativa richiedeva la costruzione di una nuova struttura dati e di nuovi metodi, signal e slot per poter connettere la parte logica e la parte grafica: un lavoro inutile.
   - *QGroupBox *group* è utilizzato per fornire una limitazione visiva all'oggetto ed un aspetto migliore di quello che si ottiene utilizzando i contorni di rettangoli neri.
     Grazie a questo QWidget è stato altresì possibile trovare il modo di inserire il nome del giocatore in una maniera non ingombrante ed accattivante
   - *QLabel *ghiande* serve semplicemente per visualizzare la scritta "Ghiande : "
   - *QLabel *soldi* serve per fare il display della quantità di ghiande possedute dal giocatore. Questa QLabel è stata divisa da quella precedente in quanto la loro unione rendeva problematico l'aggiornamento del numeretto da visualizzare. Dividendole le cose si sono semplificate, poiché è stato possibile utilizzare su *soldi il metodo di QLabel setNum(x).
   - *QPushButton *immagine* è utilizzato come contenitore per l'immagine della pedina. Lo si utilizza con il flag 'flat' di modo che non si notino i contorni: per nascondere la sua

presenza. Perchè è stato usato un pulsante allora? Semplicemente perchè disabilitandolo l'immagine della pedina applicatagli sopra scivolava nelle tonalità di grigio e rendeva così molto bene l'idea che non era il turno di un dato giocatore: in questo modo si moltiplica lo sbalzo cromatico tra l'icona colorata del giocatore attuale e quelle grigie dei giocatori a riposo (pur spulciando nella documentazione delle Qt non siamo riusciti a trovare il modo per cambiare i colori per portarli da colore reale a scala di grigi e viceversa, se ci fossimio riusciti avremmo utilizzato il QPainter per disegnare la pedina).

2. Il costruttore è piuttosto banale. Il puntatore a giocatore viene memorizzato nell'apposito campo dati. Il puntatore a giocatoreGraf viene sfruttato per ottenere l'immagine della pedina correlata al giocatore. Il punatore QWidget serve per dichiarare l'oggetto figlio di un altro oggetto.

3. Il distruttore in questo caso si preoccupa di deallocare dallo heap gli oggetti QWidget ma non compie la delete sul puntatore a giocatore per ovvi motivi.

4. *void aggiorna()* è l'unico slot della classe. Il nome spiega eccellentemente quello che compie. Ogni volta che viene invocato distingue due casi:
   - se il giocatore è vivo aggiorna il campo soldi in quanto il numero visualizzato potrebbe essere aumentato o diminuito. Aggiorna solo quello in quanto, se il giocatore è ancora vivo, è l'unica cosa che potrebbe essere cambiata (di certo non cambia il nome e neppure la pedina che utilizza)
   - se il giocatore è morto (GiocatoreLogico->eMorto()==true) si preoccupa di sostituire l'immagine della pedina con l'immagine di una tomba e di sostituire alla scritta “Ghiande : ” la scritta “BANCAROTTA”

INFO GIOCATORI

InfoGiocatori altro non è che un accorpamento di più MonoGiocatore.
Fornisce effettivamente la disposizione grafica dei MonoGiocatore (a colonna in questo caso) e cosa molto più importante un'interfaccia utilizzabile da altri oggetti.

```
public slots:
 void aggiorna();
 void abilitaGiocatore(int);


private:
 vector<MonoGiocatore*> Schede;


public:
 InfoGiocatori(const vector<Giocatore*>&, const vector<GiocatoreGraf*>&, QWidget * =0);
 ~InfoGiocatori();
```

1. *vector<MonoGiocatore *> Schede* è l'unico campo dati presente. Effettivamente a InfoGiocatori serve solamente un modo per collezionare gli oggetti MonoGiocatore per poi farne la visualizzazione, ed il vector è il metodo migliore poiché il numero dei giocatori è variabile.

2. il costruttore abbisogna del vector dei giocatori logici e del vector dei giocatori grafici per

creare ogni singolo oggetto di MonoGiocatore da porre sullo heap: giocatori logici per il campo dati di MonoGiocatore e giocatori grafici per reperire l'immagine della pedina.
Inoltre per ogni oggetto MonoGiocatore creato, setta la sua disposizione grafica attraverso un SetGeometry parametrizzando la coodinata delle ordinate attraverso un contatore che si incrementa di uno per ogni MonoGiocatore creato.

3. Il distruttore si preoccupa di deallocare dallo heap gli oggetti MonoGiocatore creati. La delete richiamerà quindi il distruttore di MonoGiocatore che è stato appositamente ridefinito.

4. public slots:
   - *void aggiorna()*, richiama semplicemente il metodo MonoGiocatore::aggiorna() per ogni elemento presente nell'array Schede. Li aggiorna tutti e non solo uno perchè l'azione di un singolo giocatore può modificare le finanze di tutti i giocatori (per esempio nel caso tutti i giocatori detengano il 10% delle azioni dell'acquedotto e un giocatore finisca sulla casella dell'acquedotto: tutti i giocatori vedranno variare le proprie finanze, chi in meglio chi in peggio)
   - *void abilitaGiocatore(int i)*, serve per impostare a disabled tutti i MonoGiocatori tranne quello indicizzato da 'i' che verrà invece abilitato.
     In questo caso si sfrutta il fatto che MonoGiocatore possiede un metodo per abilitare o disabilitare se stesso poiché ereditato da QWidget.
     Come sopra, anche qui è necessario compiere un ciclo for su tutti gli elementi di Schede poiché pur conoscendo chi deve essere abilitato (e quindi conoscendo di chi è il turno) non si può risalire a chi apparteneva il turno appena concluso (per disabilitare solo lui in quanto ad ogni turno un solo MonoGiocatore è attivo). Questo fatto è vincolato dal fatto che i giocatori possono morire e quindi sconvolgere il ritmo del gioco : il turno appena concluso non sempre appartiene al giocatore precedente indicizzato da (i-1) modulo (num_giocatori-1) [num_giocatori-1 perchè il range varia da 0 a num_giocatori-1].

Estendibilità
La difficoltà di estendere queste struttura è legata al fatto che utilizza oggetti veramente molto piccoli: è quindi difficile trovare dello spazio per inserire qualcos'altro. Il tutto sarebbe risolto se la risoluzione globale del gioco verrebbe aumentata rendendo così disponibile più pixel liberi da gestire senza l'ansia della sovrapposizione e sbordamento di oggetti.
Essendo comunque una classe grafica prettamente passiva in cui si visualizzano semplicemente informazioni è improbabile che una modifica sostanziale al monopoli influisca più di molto su di essa.
Non è comunque arduo aggiungere qualche pulsante/immagine/scritta in più di modo da soddisfare ogni esigenza: la difficoltà rimane trovare lo spazio ed ottimizzarlo.

Indipendenza
Nonostante possa sembrare strano, la parte logica e la parte grafica restano comunque isolate l'una dall'altra. Non vi sono funzioni che mischiano aspetti logici ad aspetti grafici ad intermittenza, l'unica cosa che potrebbe trarre in inganno è il puntatore a Giocatore. Constatando che questa classe è una classe completamente passiva (le cui uniche azioni consistono nello scrivere qualche numeretto o disegnare un'immagine) la si può ritenere come un vetro che lascia filtrare l'essenza del giocatore (i suoi dati quali nome, ghiande etc) senza tuttavia rischiare di manomettere qualcosa di lui.

TAVOLO DA GIOCO

La classe TavoloDaGioco deriva da QWidget e rappresenta l'interfaccia grafica di un'intera partita, contiene tutte le CaselleGrafiche, il TurnoGrafico e InfoGiocatori.
Infatti i campi dati PRIVATI sono:

```
TurnoGrafico * t;
vector<CasellaGrafica*> caselle;
InfoGiocatori * Info;
```

TavoloDaGioco non fornisce all'utente alcuna funzione se non quella di dare coesione alle varie componenti grafiche del monopoli, può essere quindi annoverata tra le interfacce grafiche PASSIVE.
TavoloDaGioco organizza in un QGridLayout tutti i componenti grafici, fa questo lavoro nel costruttore:

```
TavoloDaGioco(TurnoGrafico * ,vector<CasellaGrafica*> &,  InfoGiocatori *);
```

Il costruttore invoca il costruttore del sotto-oggetto di tipo QWidget con parametro parent=0, assegnandosi così una nuova finestra, successivamente crea e inizializza t, caselle e Info e poi si impegna a distribuirle nello spazio disponibile che viene fissato a 1020x700 pixel.
Dopo aver fissato la dimensione il costruttore dispone gli oggetti nel layout secondo il seguente schema:

INFO GIOCATORI	CASELLA 0	CASELLA 1	CASELLA 2	CASELLA 3	CASELLA 4	CASELLA 5	CASELLA 6
INFO GIOCATORI	CASELLA 19	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	CASELLA 7
INFO GIOCATORI	CASELLA 18	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	CASELLA 8
INFO GIOCATORI	CASELLA 17	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	**TURNO GRAFICO**	CASELLA 9
INFO GIOCATORI	CASELLA 16	CASELLA 15	CASELLA 14	CASELLA 13	CASELLA 12	CASELLA 11	CASELLA 10

Appare chiaro che se si volesse estendere il codice aumentando le caselle o i giocatori questa disposizione grafica andrebbe completamente rivista, ma è comunque un piccolo prezzo da pagare.

Dobbiamo anche ricordare che il costruttore di TavoloDaGioco realizza due connessioni:

```
1. connect(tur, SIGNAL(chiudi()), this, SLOT(close()));
2. connect(tur, SIGNAL(aggiornaInfoGiocatori(int)), Info, SLOT(abilitaGiocatore(int)));
```

1. All'emissione del signal chiudi() da parte di TurnoGrafico TavoloDaGioco risponde con la chiusura di se stesso, come vedremo l'emissione di quel signal da parte di TurnoGrafico identifica la fine della partita;
2. All'emissione invece di aggiornaInfoGiocatori(int) da parte di TurnoGrafico TavoloDaGioco connette lo slot abilitaGiocatori(int) di InfoGiocatori, come vedremo in TurnoGrafico vuol dire che qualcosa ha modificato lo stato dei giocatori, occorre quindi aggiornare adeguatamente l'oggetto della classe InfoGiocatori appeso in TavoloDaGioco.

È bene ricordare che non è necessario ridefinire il distruttore di TavoloDaGioco in quanto, come spieghiamo più dettagliatamente in Partita, tutte le CaselleGrafiche, l'oggetto di tipo TurnoGrafico e

l'oggetto di tipo InfoGiocatori sono appesi sotto TavoloDaGioco, sarà quindi il distruttore standard a deallocarli, si veda Distruttori.

Estendibilità
Per quanto riguarda TavoloDaGioco occorre chiarire la scelta di fissare una dimensione: purtroppo siamo stati costretti in quanto le dimensioni degli sfondi delle caselle sono costanti, come detto precedentemente, così come gli interi widget che identificano una casella; permettere al TavoloDaGioco di allargarsi vorrebbe dire lasciare invariata la dimensione delle caselle e aumentare soltanto in maniera antiestetica le distanze tra una casella e l'altra. La necessità di fissare la dimensione del TavoloDaGioco ci ha costretto a deciderne la misura, abbiamo optato per 1020x700 pixel, una misura che stesse nella risoluzione 1024x768, risoluzione supportata dalla maggior parte delle macchine; naturalmente questa scelta obbligata ha condizionato le misure di ogni casella e conseguentemente anche di TurnoGrafico.
Siamo tuttavia convinti, magari a torto, di aver adottato la soluzione migliore partendo dal presupposto di voler personalizzare ogni casella.
Questa dimensione fissata e l'intera struttura di TavoloDaGioco pone un problema nel caso in cui si volessero aggiungere delle caselle, lo spazio per ospitarle è insufficiente, occorre un completa rivisitazione della disposizione del tavolo. Crediamo però che questa problematica sia insormontabile in qualunque tipo di interfaccia grafica e riteniamo che in caso dell'aumento del numero delle caselle una modifica dello spazio di gioco sia da ritenersi più che normale.

Indipendenza
TavoloDaGioco è un'interfaccia grafica passiva, un semplice contenitore di Qwidgets che non interagisce in alcun modo con l'utente, a causa di questo TavoloDaGioco con contiene nulla che appartenga alla logica, possiamo dichiararlo quindi completamente indipendente dalla logica e in maniera simmetrica possiamo dichiarare la sua esistenza completamente trasparente alla logica, come esattamente deve essere.

GRAFICA ATTIVA

TURNO GRAFICO

TurnoGrafico è la classe che basandosi su TurnoLogico fornisce le informazioni relative allo stato del gioco agli utenti e interagendo con l'utente detta i tempi dell'evoluzione nelle fasi del TurnoLogico.
TurnoGrafico è dotato di un puntatore all'oggetto della classe TurnoLogico e con questo interagisce, vedremo ora in dettaglio come.
È importante sottolineare che TurnoGrafico possiede un puntatore a TurnoLogico, dovendo a determinati avvenimenti nella grafica invocarne le funzioni. Ad esempio al click del QPushButton muovi prima di tutto TurnoGrafico deve far sì che TurnoLogico muova il giocatore, invocando t->muovi(). Tuttavia questa dipendenza si porta dietro dei problemi di sicurezza e indipendenza del codice, a questo scopo abbiamo prestato particolare attenzione a rendere privato in TurnoLogico tutto ciò che non era sensato fosse visibile a TurnoGrafico (si veda TurnoLogico).
È inoltre importante ricordare che TurnoGrafico deriva da QWidget e ha come parent, nella nostra confurazione del gioco il QWidget TavoloDaGioco.
Andiamo ora ad analizzare in dettaglio il contenuto della classe TurnoGrafico in modo da avere un' idea precisa del suo funzionamento.
I campi dati:

```
1.  vector<CasellaGrafica*> CaselleGrafiche;
2.  vector<GiocatoreGraf*> GiocatoriGrafici;
3.  TurnoLogico * t;
4.  QPushButton * move;
5.  QPushButton * passa;
6.  Dado * d;
7.  PopUp * p;
8.  int vecchiaCasella;
9.  int nuovaCasella;
10. QTimer *timer;
    double angolo;
    double angoloTraiettoria;
    int rotazione;
    int contatore;
    int x;
    int y;
```

1. *vector<CasellaGrafica*> CaselleGrafiche*, è un campo dati PRIVATO e non è altro che il vector delle CaselleGrafiche appese dentro al tabellone, è fondamentale che TurnoGrafico possieda le CaselleGrafiche in quanto ad ogni azione logica (come ad esempio lo spostamento del giocatore) determinate CaselleGrafiche potrebbero subire dei cambianti e quindi vanno aggiornate tramite il metodo update() che non fa altro che re-invocare paintEvent(QPainter *e);

2. *vector<GiocatoriGraf*> GiocatoriGrafici*, questo vector, campo dati PRIVATO, non è fondamentale all'esecuzione di TurnoGrafico, ci pareva tuttavia importante far comparire l'immagine del giocatore nella finestra del turno quando tocca a lui giocare, più che altro per aumentare l'intuitività del gioco;

3. *TurnoLogico * t*, questo è il sopracitato puntatore ad un oggetto della classe TurnoLogico, è

privato in quanto non deve essere disponibile a nessun altra entità eccetto TurnoGrafico. Il suo utilizzo corretto è fondamentale per il mantenimento del giusto flusso dell'esecuzione, i suoi metodi vengono invocati all'avvenire di precisi passaggi nella grafica;

4. *QPushButton* move*, è un puntatore a pulsante PRIVATO, allocato sullo heap e appeso sotto all'oggetto a cui appartiene, che permette al suo click al giocatore di muoversi, il suo click viene raccolto dallo slot *void muovi()*;

5. *QPushButton* passa*, è un puntatore a pulsante PRIVATO, anch'esso allocato sullo heap e appeso sotto all'oggetto a cui appartiene, che permette al giocatore di passare il turno al successivo, il suo click viene raccolto dallo slot *passaTurno()*;

6. *Dado * d*, è PRIVATO ed è il puntatore al dado che accompagna tutta la partita, è un widget che viene appeso sotto TurnoGrafico;

7. *PopUp * p*, è il QWidget che permette al giocatore di interagire sulla casella su cui è approdato e lo informa sulla sua condizione in caso che questa cambi, ad esempio nel caso in cui all'inizio del turno l'oggetto della classe TurnoLogico si renda conto che il giocatore è in Prigione o in stato di fermo il PopUp dovrà comunicarlo, analogamente nel caso in cui la logica si renda conto che all'inizio del turno il giocatore ha superato il massimo numero di giri e identifichi il vincitore un PopUp nell'oggetto della classe TurnoGrafico dovrà comunicare la fine della partita con relativo vincitore. Un ulteriore uso della classe PopUp si ha quando, dopo aver interagito con una casella, un giocatore sia rimasto senza ghiande, in questo caso PopUp ne comunica la morte e se è rimasto soltanto un giocatore ne comunica la vittoria (per le differenziazioni tra i vari PopUp si veda PopUp);

8. *int VecchiaCasella* è un campo dati privato, ad ogni turno contiene l'indice della casella su cui il giocatore in turno si trovava prima di effettuare lo spostamento logico;

9. *int NuovaCasella*, analogamente al precedente campo è privato e contiene l'indice della casella sulla quale il giocatore è approdato dopo lo spostamento logico, salvare questi indici per mezzo di un campo dati nella classe potrebbe sembrare inutile perché l'aggiornamento delle caselle dopo lo spostamento viene gestito da un unico slot (muovi()) dopo l'invocazione del muovi logico (t->muovi()), tuttavia la loro esistenza è cruciale per via della possibilità che sulla casella di arrivo avvengano modifiche della posizione del giocatore (ad esempio se dal tribunale passa in prigione) oppure nel caso in cui il giocatore muoia in seguito all'essere finito su di una casella in cui deve pagare una cifra che non ha;

10. *QTimer *timer, double angolo, double angoloTraiettoria, int rotazione, int contatore, int x, int y*, sono tutti campi dati che permettono l'aggiornamento della posizione dell'immagine del giocatore in TurnoGrafico ad intervalli dettati dal Timer, utilizzando lo slot movimentoImmagine();

Iniziamo con i metodi, o meglio con i signals per ora:

```
1. void chiudi();
2. void aggiornaInfoGiocatori(int);
```

1. *void chiudi()*, è un signal che suggerisce al TavoloDaGioco che la partita è finita e che si può quindi chiudere, viene invocato quando un PopUp di vittoria viene chiuso (si veda PopUp), TurnoGrafico è in grado di rendersi conto di che tipo sarà il prossimo PopUp andando ad analizzare l'oggetto di tipo Andata che gli viene passato dalla logica, in questo modo può

stabilire una connect tra il signal di chiusura del PopUp di vittoria e il suo stesso signal chiudi();

2. *void aggiornaInfoGiocatori(int)*, ad ogni operazione che avvenga sulla casella è possibile che i giocatori abbiano subito dei cambiamenti in termini di ghiande, o potrebbero addirittura averle finite ed essere quindi morti, in questo caso è necessario aggiornare il tabellone che raccoglie informazioni su di loro (appunto un oggetto della classe InfoGiocatori), anche questo oggetto è appeso in TavoloDaGioco. Questo signal viene invocato alla chiusura di ogni PopUp, non direttamente in quanto il signal di chiudi() di PopUp è senza parametri invece il signal aggiornaInfoGiocatori ha un parametro intero e questo tipo di connect non sono previste dalle librerie Qt. Abbiamo quindi bisogno di uno slot ausiliario che invochi aggiornaInfoGiocatori(int), questo slot sarà visto successivamente ed è lo slot updateInfoGiocatori(). Vogliamo anche ricordare che aggiornaInfoGiocatori(int) viene emesso quando TurnoLogico e TurnoGrafico passano il turno ad un nuovo giocatore (viene quindi invocato in passaTurno()) allo scopo di aggiornare il giocatore il possesso del turno anche nell'oggetto della classe InfoGiocatori. Finalmente possiamo chiarire lo scopo del parametro di questo signal, lo scopo è infatti quello di comunicare l'indice del giocatore in turno. Infine updateInfoGiocatori viene anche emesso nella funzione muovi di TurnoGrafico dopo che questa ha, come vedremo, invocato t->muovi(). Lo scopo è quello di aggiornare le ghiande del giocatore che si è spostato nel caso sia passato dal via.

Passiamo ora ad analizzare i metodi pubblici della classe TurnoGrafico:

```
1. TurnoGrafico(const vector<CasellaGrafica*> &, const vector<GiocatoreGraf*> &,TurnoLogico *, bool, bool, QWidget * =0);
2. ~TurnoGrafico();
3. void paintEvent(QPaintEvent *);
```

1. *TurnoGrafico(const vector<CasellaGrafica*> &, const vector<GiocatoreGraf*> &,TurnoLogico *, bool, bool, QWidget * =0)*, questo è il costruttore che prima di tutto costruisce il sotto-oggetto di tipo QWidget appendendolo al disotto del parametro di tipo QWidget* che viene passato al costruttore, dopodiché costruisce e inizializza i due vector delle caselle grafiche e dei giocatori grafici e il puntatore ad un oggetto della classe TurnoLogico. Ora è il momento di costruire i due pulsanti muovi e passa, come QPushButton figli dell'oggetto che stiamo creando, e il dado, anch'esso come QWidget figlio. Qui entra in gioco il ruolo del primo valore booleano che ci dice se stiamo giocando o meno in modalità dado truccato, nel qual caso il dado viene creato di conseguenza (si veda Dado). Il puntatore ad oggetto della classe PopUp viene creato a NULL e naturalmente presteremo particolare attenzione a non utilizzarlo prima di averlo inizializzato. vecchiaCasella e nuovaCasella sono creati inizializzati a 0.
   Per TurnoGrafico è stato deciso di utilizzare setGeometry in quanto esso è contenuto come già detto nel QWidget più esterno TavoloDaGioco che ha dimensione fissata, renderlo di dimensione variabile non avrebbe senso.
   Particolare attenzione meritano le connect definite dal costruttore:

```
connect(d,SIGNAL(lanciato(int)),this, SLOT(attivaMovimento(int)));

connect(move, SIGNAL(clicked()),this, SLOT(muovi()));

connect(passa, SIGNAL(clicked()), this, SLOT (passaTurno()));

if(legale)
  {
   connect(passa, SIGNAL(clicked()), this, SLOT(testPreliminare()));
  }
```

Per prima cosa viene creata la connect tra il signal del dado che porta con se il risultato del lancio, lanciato(int) (si veda Dado) e lo slot di TurnoLogico attivaMovimento(int) che descriveremo tra breve. Dopodiché si connette il click del tasto move con lo slot muovi() e il click del tasto passa con lo slot passaTurno(). Infine si chiarisce il significato del secondo booleano nel costruttore, questo indica se ci troviamo o meno nell'opzione legale, nel qual caso va creata un'ulteriore connect che al click del pulsante passa invoca lo slot testPreliminare() [14].
Alla fine vengono sistemati i campi dati relativi all'animazione con le relative connect:

```
1. connect(timer, SIGNAL(timeout()), this, SLOT(movimentoImmagine()));
2. connect(move, SIGNAL(clicked()), timer, SLOT(stop()));
```

La prima connect collega il timeout del timer con lo slot movimentoImmagine() mentre la seconda fa sì che l'animazione termini quando il giocatore si muove sul tabellone, ovvero quando viene creato il popup, collegando il click del pulsante muovi con lo slot stop();

3. *~TurnoGrafico()*, è il distruttore della classe TurnoGrafico, si occupa di eliminare uno ad uno i GiocatoriGrafici ed invoca anche il distruttore di TurnoLogico, si veda Distruttori per capire perché il giocatori grafici sono deallocati manualmente;

4. void paintEvent(QPaintEvent *), è il painter di turno grafico, disegna la scritta che specifica il proprietario del turno e anche la sua immagine.

Passiamo ora agli slots:

```
1. void passaTurno();
2. void muovi();
3. void attivaMovimento(int);
4. void aggiorna();
5. void testPreliminare();
6. void statoGiocatore();
7. void updateInfoGiocatori();
8. void movimentoImmagine();
```

1. *void passaTurno()*, come detto precedentemente viene invocato al click del pulsante passa, non fa altro che invocare t->passaTurno(), la funzione passaTurno di TurnoLogico, abilita il pulsante di lancio del dado tramite Dado::abilitaLancio() e disabilita quello di passaggio del turno. Emette anche il signal aggiornaInfoGiocatori() e ritorna;

2. *void muovi()*, registra il valore di vecchiaCasella, invoca muovi() in TurnoLogico (t->muovi()) salvando i valore di tipo Andata in una variabile locale e registra anche nuovaCasella, successivamente aggiorna entrambe le caselle. A questo punto controlla il valore di Andata:
   - se Andata è del tipo vittoria per esaurimento giri crea e successivamente mostra un PopUp per la suddetta Andata connettendo:

14 ATTENZIONE: una connect definita successivamente ad un altra connect sullo stesso signal farà in modo che il secondo slot sia invocato successivamente al primo, caratteristica fondamentale per la correttezza di questo passaggio.

```
connect (p, SIGNAL(chiudi()), this, SLOT(close()));
connect (p, SIGNAL(chiudi()), this, SIGNAL(chiudi()));
```

Connetto la chiusura del PopUp con lo slot di chiusura di TurnoGrafico, ereditato da QWidget, close(). La connette inoltre con l'emissione del signal chiudi();

- se Andata invece è di qualsiasi altro tipo viene costruito e mostrato lo specifico PopUp con le seguenti connessioni:

```
connect(p, SIGNAL(chiudi()), this, SLOT(updateInfoGiocatori()));
connect(p, SIGNAL(chiudi()), this, SLOT(statoGiocatore()));
connect(p, SIGNAL(chiudi()), this , SLOT(aggiorna()));
connect(p, SIGNAL(chiudi()), timer, SLOT(start()));
```

Per prima cosa si connette la chiusura con lo slot updateInfoGiocatore(), con lo slot statoGiocatore() che ha il compito di determinare se il giocatore ci ha lasciati, nel qual caso deve comunicare il tutto all'utente, con lo slot aggiorna() che risolve eventuali imprigionamenti e decessi invocando udpdate sulle caselle grafiche e infine con lo slot start() del timer, che lo fa ripartire;

5. *void attivaMovimento(int)*, questo slot viene invocato dopo il lancio del dado, come parametro attuale ha il risultato del lancio, salva il risultato tramite t->salvaLancio(int) e attiva il pulsante move;

6. *void aggiorna()*, come già detto nel caso in cui il giocatore sia morto o nel caso in cui esso sia stato spostato all'indietro dopo la fase attiva sulla casella è necessario aggiornare le caselle grafiche, vecchiaCasella diventa quella che prima era nuovaCasella e nuovaCasella viene ricalcolata, entrambe sono aggiornate da aggiorna();

7. *void testPreliminare()*, questo slot come già detto viene connesso soltanto se ci troviamo nella modalità legale, il suo compito è quello di controllare all'inizio del nuovo turno se il giocatore si trova in stato di arresto o di fermo, fa questo per mezzo della funzione di TurnoLogico testPreliminare che ritorna un valore di tipo Andata. testPreliminare grafico controlla il tipo di Andata, se questo indica lo stato di Ferm (si veda Andata) viene costruito un PopUp sull'oggetto di tipo Andata e sono disabilitati tutti i pulsanti fatta eccezione per passa, in questo modo l'utente dopo essere stato informato del suo stato potrà soltanto passare il turno;

8. *void statoGiocatore()*, come già detto questo slot viene invocato alla chiusura di un PopUp di operazioni sulla casella, il suo scopo è quello di definire se il giocatore è deceduto e inoltre se è rimasto in vita soltanto un giocatore, naturalmente per fare questo lo slot si serve della funzione statoGiocatore di TurnoLogico che ritorna un valore di tipo Andata che discrimina le condizioni precedenti, se Andata è del tipo Vitt vuol dire che il giocatore in turno è morto e ne è rimasto soltanto un altro, va creato un PopUp che ne comunica la morte per mezzo di un oggetto della classe Andata di tipo Mort contenente il puntatore al giocatore in turno deciduto (reperito tramite t->getGiocatore()) e successivamente, dopo aver passato il turno al giocatore successivo (che sarà l'ultimo in vita) tramite t->passaTurno(), un PopUp che comunica la vittoria, per il secondo PopUp possiamo utilizzare l'oggetto di tipo Antata ritornato da t->statoGiocatore(). Nel caso in cui il giocatore sia semplicemente deceduto Andata avrà tipo Mort e dobbiamo soltanto costruire un PopUp sull'oggetto Andata ritornato. Nel primo caso, che è naturalmente il più complesso vanno utilizzate delle connect per stabilire le precedenze tra PopUp:

```
connect(p, SIGNAL(chiudi()), p1, SLOT(show()));
connect(p, SIGNAL(chiudi()), this, SLOT(updateInfoGiocatori()));
connect(p1, SIGNAL(chiudi()), this, SLOT(close()));
connect(p1, SIGNAL(chiudi()), this, SIGNAL(chiudi()));
```

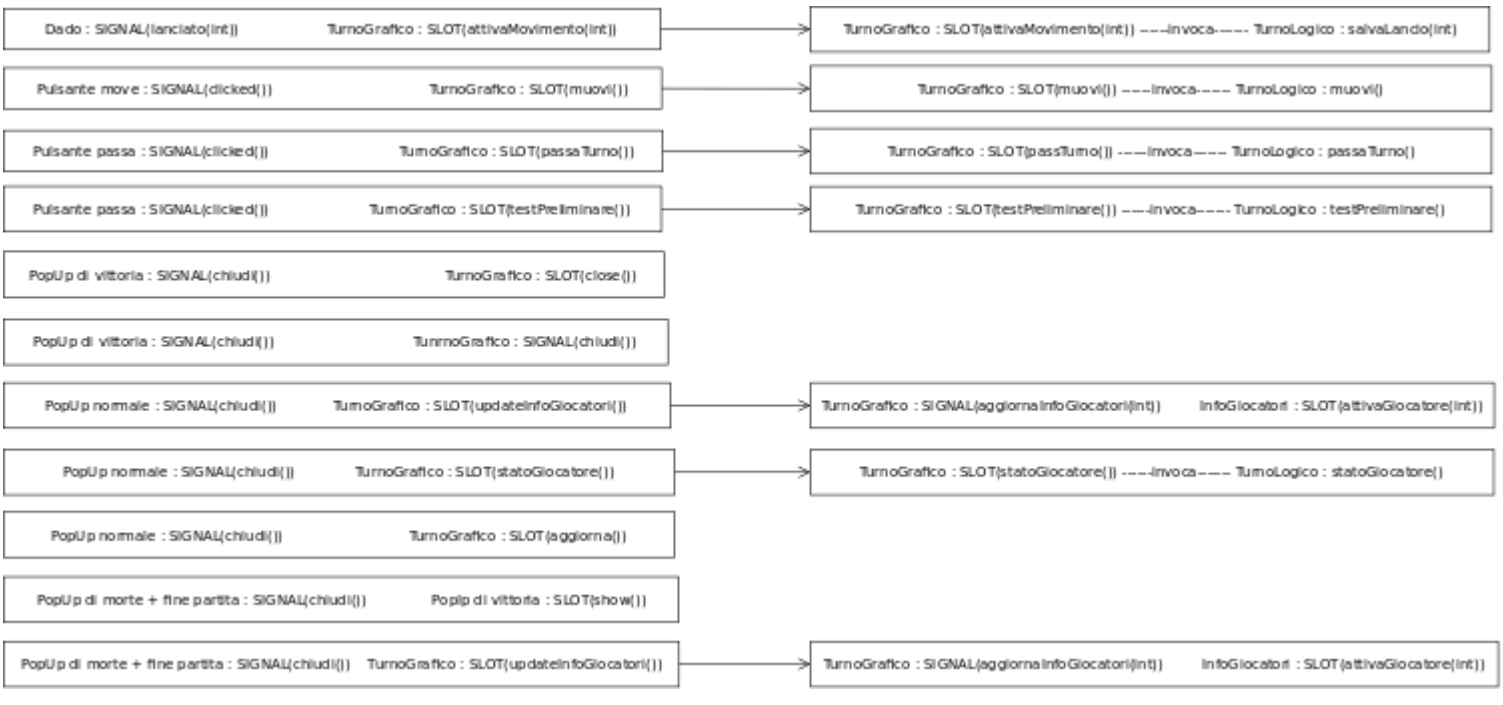
La prima connect indica che alla chiusura del primo PopUp si può rendere visibile il secondo. La seconda connect viene fatta in quanto il turno viene passato e quindi è necessario aggiornare l'oggetto InfoGiocatori. La terza e la quarta connect servono come abbiamo visto nella funzione muovi() per chiudere il gioco a vittoria comunicata.
In entrambi i suddetti casi un giocatore è morto, vanno quindi aggiornate tutte le caselle grafiche dei suoi possedimenti, che sono state liberate logicamente;

9. *void updateInfoGiocatori(),* come abbiamo già accennato questo slot fa soltanto da ponte[15] tra il signal chiudi() di PopUp e il signal aggiornaInfoGiocatori(int) che porta con se l'indice del giocatore in possesso del turno. UpdateInfoGiocatori() emette semplicemente il signal aggiornaInfoGiocatori(int);

10. *void movimentoImmagine(),* questo slot viene invocato all'emissione del timeout del timer, cambia i parametri di posizionamento e rotazione dell'avatar del giocatore in turno su cui lavora il painter, infine aggiorna l'intero oggetto.

Estendibilità
Eventuali estensioni del numero di caselle o del numero di giocatori non turbano minimamente TurnoGrafico. Questo infatti reperisce le sue informazioni da TurnoLogico e lavora esclusivamente su vector di puntatori a GiocatoreGraf e CasellaGrafica la cui dimensione può tranquillamente variare dal momento che sono sempre scorsi non oltre l'end(). Per TurnoGrafico lavorare con 100 e 1000 caselle o 100 o 1000 giocatori non fa alcuna differenza.
TurnoGrafico analogamente a TurnoLogico può lavorare con tutte le nuove caselle che prevedono operazioni all'arrivo sulla casella o controlli prima dello spostamento, per eventuali altri tipi di caselle è sempre possibile effettuare derivazioni da TurnoGrafico. secondo.

Connect e Indipendenza
Ecco uno schema riassuntivo delle connect (sono escluse per brevità quelle che gestiscono le animazioni), questo schema inoltre ci illustra come a determinati avvenimenti sono invocate funzioni di TurnoLogico per la gestione corretta del susseguirsi degli avvenimenti.



Vediamo come TurnoGrafico mette in moto TurnoLogico ma non interferisce né modifica in alcun modo la parte logica del programma:

L'inizio del turno di un giocatore avviene al click del pulsante passa, se ci troviamo nella modalità legale la TurnoGrafico chiede a TurnoLogico di informarlo sulla condizione del giocatore tramite la funzione testPreliminare(), TurnoLogico raccoglie le informazioni e comunica la situazione tramite un oggetto di tipo Andata a TurnoGrafico che agisce di conseguenza (o mostra un PopUp di tipo Ferm o lascia proseguire).
Il click del pulsante muovi provoca lo spostamento logico del giocatore, TurnoLogico raccoglie informazioni sulla nuova casella in cui è atterrato il giocatore e le passa a TurnoGrafico in modo che questo possa comunicarle a sua volta all'utente.
Alla fine dell'interazione tra utente e casella, che avviene tramite PopUp, la grafica deve nuovamente interrogare TurnoLogico circa le condizioni del giocatore, e fa questo alla chiusura del PopUp, per mezzo dello slot statoGiocatore() invoca t->statoGiocatore(), nuovamente TurnoLogico raccoglie informazioni circa la condizione del giocatore e le rimanda alla grafica sempre tramite un oggetto di tipo Andata. A questo punto TurnoGrafico comunica all'utente eventuali condizioni eccezionali altrimenti può abilitare il pulsante passa. Al click di quest'ultimo si può passare il turno, lo slot grafico passaTurno invoca il metodo di TurnoLogico passaTurno().
Come vediamo la grafica ha il compito di chiamare al momento opportuno le funzioni della logica, ovvero in base alle richieste dell'utente. Contemporaneamente la grafica non ha alcuno modo di ottenere informazioni circa lo stato del gioco e le condizioni del giocatore (che riguardano la componente logica del gioco), deve quindi sempre riferire a TurnoLogico per ottenerle.
Abbiamo così provato che la grafica non ha alcuna possibilità di accesso alle informazioni sensibili e contemporaneamente TurnoLogico dipende sì dalle scelte interattive dell'utente ma come è normale che sia, il non utilizzo delle librerie Qt o addirittura l'eliminazione dell'interfaccia grafica a fronte dell'utilizzo di un'interfaccia testuale non causerebbero il minimo problema a TurnoLogico.

TurnoGrafico è PopUp
Dal punto vista prettamente grafico TurnoGrafico occupa il centro del TavoloDaGioco, quando TurnoGrafico crea PopUp si pone la problematica della dislocazione del PopUp.
Il nostro pensiero iniziale era quello di lasciare PopUp in una nuova finestra (da qui viene anche il nome della classe). A questa idea si sono sovrapposti due problemi:

1. Se il giocatore avesse chiuso la finestra del PopUp senza dare risposta (per mezzo della x in alto a destra, non tramite un pulsante) cosa sarebbe successo al flusso di gioco?
   Questo problema si sarebbe potuto risolvere cercando nelle librerie un modo di disattivare la x in alto a destra sulla finestra, oppure adottando in turno un'opportuna logica di controllo dell'effettivo uso della PopUp.
2. Dal punti di vista grafico il gioco perdeva consistenza, dovendo utilizzare più finestre rendeva il suo utilizzo più scomodo, tenendo presente che si tratta di un software molto semplice l'utilizzo di due finestre poteva sembrare eccessivo all'utente giocatore.

Per questi due motivi ogni qual volta viene costruito un oggetto di tipo PopUp in Turno Grafico questo viene costruito con parametro parent = this. Ovvero lo stiamo costruendo sotto noi stessi! I PopUp è dimensionato per coprire esattamente TurnoGrafico.

DADO

Dado è una classe che deriva da QWidget. Viene nel nostro caso utilizzato unicamente all'interno di TurnoGrafico, fa quindi parte dell'interfaccia attiva. Dado ha un come campi dati PRIVATI: un *puntatore a Qpixmap* che punta all'immagine corrispondente al risultato dell'ultimo lancio, *un intero* che contiene il *risultato* dell'ultimo lancio e un *puntatore a QPushButton lancia*.
Dado fornisce inoltre i seguenti metodi pubblici:

```
3.  Dado(QWidget *, bool);
4.  ~Dado();
5.  void paintEvent(QPaintEvent * e);
6.  void abilitaLancio();
7.  void disabilitaLancio();
```

1. *Dado(QWidget*, bool)*, è il costruttore, il primo parametro è la finestra parent mentre il secondo parametro ci dice se il dado deve essere truccato o meno.
   Il costruttore inoltre alloca sulla pila la QPixmap della faccia[16] del risultato dell'ultimo lancio, inizialmente sarà l'immagine della faccia 1 e costruisce a 1 anche l'intero risultato. Infine costruisce il pulsante lancia sullo heap appendendolo sotto l'oggetto che sta costruendo disponendolo nel widget, sia il pulsante che il widget hanno dimensione fissa.
2. *~Dado()*, dealloca la Qpixmap;
3. *void paintEvent(QPaintEvent *)*, disegna la QPixmap alla destra del pulsante lancia;
4. *void abilitaLancio()*, questo metodo non fa altro che invocare sul pulsante lancia setDisabled(false), che lo rende cliccabile;
5. *void disabilitaLancio()*, questo metodo invoca su lancia setDisabled(true);

e i seguenti slot:

```
1.  void mostra();
2.  void mostraTruccato();
```

1. Questo slot viene connesso dal costruttore con il click sul pulsante lancia soltanto se il parametro booleano passatogli è false:

   ```
   connect(lancia,SIGNAL(clicked()),this,SLOT(mostra()));
   ```

   Per prima cosa lo slot disattiva il pulsante lancia, genera un numero casuale tramite la funzione rand(), questo valore modulo 6 +1 è il risultato del lancio del dado, occorre deallocare la vecchia QPixmap, crearne una nuova con la faccia del dado corrispondente al risultato e fare l'update (cioè re-invocare painterEvent) sul widget ed infine emette il signal lanciato(risultato);
2. viene connesso se il secondo parametro passato al costruttore è true, sta a significare che il dado è truccato

   ```
   connect(lancia,SIGNAL(clicked()),this,SLOT(mostraTruccato()));
   ```

   mostraTruccato non fa niente se non disattivare il pulsante lancia ed emettere il signal lanciato sul risultato, il risultato sarà quindi sempre lo stesso specificato nel costruttore.

Infine il signal

```
lanciato(int);
```

Viene emesso dopo che il dado è stato lanciato e dopo che il risultato è stato mostrato all'utente con il risultato del lancio.

16 I percorsi delle immagini sono definiti all'interno del codice.

POPUP

I membri di PopUp:

Il PopUp è l'interfaccia che mette in relazione il gioco con le scelte effettive dei giocatori.
La lista dei suoi membri è:

```
public slots:
 void setNulla();
 void setPagamento();
 void setCompraTerreno();
 void setCompraAlbergo();
 void setVariazioneIpoteca(int);
 void setVariazioneAzioni(int);
 void setImprigionato(int);
signals:
 void chiudi();

public:
 PopUp(const Andata&, QPixmap *, QWidget *parent=0);
 ~PopUp() {}
 void paintEvent(QPaintEvent*);
```

```
private:
 QPixmap * immagine;
 Andata::Tipo type;
 Giocatore *giocatore;
 CBase *casella;
 void initNull(const QString &);
 void initPaga(const QString &, int, Giocatore*);
 void initEdif(const QString &, bool, bool, int, int);
 void initIpot(const QString &, int, int, int);
 void initPriv(int, int, int, int);
 void initPolz(int);
 void initTrib();
 void initFerm(int);
 void initVitt(int, bool);
 void initMort();
```

La maggior parte dei membri privati e degli slots verranno spiegati in seguito poiché sono in strettamente legati al costruttore di PopUp.
In questo frangente si voleva porre l'accento su alcuni particolari:

- il distruttore di PopUp ha corpo vuoto in quanto la distruzione dell'oggetto si esegue tranquillamente eliminando i vari puntatori (ma non gli oggetti puntati da essi che sono di proprietà di altre classi).
  Per quel che riguarda tutti gli oggetti della libreria grafica allocati sullo heap, vengono distrutti tutti dal distruttore del sottooggetto di PopUp, in quanto QWidget, da cui deriva lo stesso PopUp, fa in modo di eliminare tutti gli oggetti che gli sono figli: è per questo che si è prestata particolare cura a rendere ogni oggetto grafico 'temporaneo' allocato sullo heap figlio del PopUp, in modo da non avere garbage che si accumula invocazione dopo invocazione, si veda DISTRUTTORI;
- la funzione *paintEvent(QPaintEvent*)* è stata ridefinita solo per rendere il background della finestra informativa più accattivante: la cosa principale che compie è capire se il PopUp informa il giocatore di essere arrivato su di una casella (e quindi dipingerà come sfondo l'immagine della casella), informa della morte di un giocatore (e quindi dipingerà l'immagine di una tomba per il giocatore deceduto) o informa della vincita di un giocatore (e quindi dipingerà una coppa di vittoria)
- *CBase *casella* è di fondamentale importanza, in quanto è necessario per far avviare i cambiamenti logici. Viene utilizzato come casella->eseguiAzione(Ritorno): se le classi Andata e Ritorno fanno da spoletta dall'ambiente logico a quello grafico e viceversa per trasportare dati e informazioni, è tramite questo puntatore che viene iniziato il processo di manipolazione dei dati nell'area logica;
- Tutti gli Slots di PopUp hanno tendenzialmente un'anatomia in comune[17]:
  - fanno eseguire i calcoli logici necessari per rendere consistenti i vari oggetti dopo ogni decisione presa dall'utente

17 Si dà quindi per scontato che ogni slot di PopUp segua questa struttura salvo diversa segnalazione: non verrà quindi descritta ogni volta la sequenza di eventi che scatenerà la sua invocazione, ma si evidenzieranno i tratti peculiari di ciascun slot. Gli slot sono strettamente correlati alle classi Ritorno e CBase e derivate, quindi riferirsi a tali classi per avere informazioni più approfondite o complete ove mancano

- emettono il signal chiudi() utilizzato da oggetti esterni a PopUp per capire quando la modifica dei dati è effettuata e quindi aggiornare scritte/numeri/immagini con le nuove informazioni
- richiamano lo slot close() che richiama sul PopUp il suo distruttore

Il costruttore di PopUp:

Seguendo la linea di pensiero secondo cui ogni oggetto deve rispondere ad un dato contratto e funzionalità, il costruttore di PopUp è unico poiché ogni PopUp risponde alla necessita` di:

1. fornire informazioni al giocatore, informazioni necessarie per prendere una decisione sull'azione da svolgere (l'impossibilità di comprare un terreno per la mancanza di soldi, la quantità di denaro necessaria per diventare il proprietario di un certo terreno, la necessita` di pagare la banca per il soggiorno su di una società pubblica, la quantità di denaro necessaria per acquistare un albergo etc);
2. fornire al giocatore un modo per informare il programma delle sue decisioni, decisioni comunque vincolate dall'effettiva possibilità di compiere una data azione (comprare l'albergo sempre che si possieda abbastanza denaro e diventarne il proprietario, pagare il pedaggio poiché si è finiti su di una casella appartenente ad un altro giocatore, lanciare la moneta per scoprire se si è stati incarcerati etc)

La prima necessità è globale e indirizzata al giocatore: deve essere una funzionalità onnipresente per qualsiasi casella e per qualsiasi giocatore. Questa globalità viene effettivamente concretizzata nel corpo del costruttore attraverso delle scritte informative. Ciò nonostante tale informazione viene costruita a partire dai dati provenienti dalla parte logica del programma: tutte le informazioni necessarie per creare effettivamente il PopUp sono racchiuse nell'oggetto di tipo Andata, che è a tutti gli effetti il cuore logico del PopUp e quindi punto in comune con tutte le diverse rappresentazioni grafiche dello stesso.
La seconda necessità è anch'essa globale ma indirizzata al programma: viene canalizzata attraverso dei pulsanti cliccabili dall'utente che creeranno degli oggetti di tipo Ritorno che torneranno nell'area logica portando come informazioni le scelte compiute dal giocatore stesso. A seconda del pulsante in questione (quindi quello cliccato dal giocatore), la cui visualizzazione e` strettamente legata all'aspetto grafico prodotta elaborando Andata, verrà creato un differente oggetto Ritorno. Le informazioni che possiede Ritorno sono le scelte fatte dal giocatore stesso.

Considerando la globalità di queste funzionalità, si e` preferito discernere l'aspetto grafico di PopUp all'interno del suo costruttore invece di creare molteplici classi PopUp che rispondessero allo stesso contratto.
Come già accennato, la differenziazione viene eseguita a seconda dei parametri passati attraverso Andata[18].
Il costruttore vero e proprio di PopUp altro non fa che salvare pochi campi dati ed accettare l'oggetto Andata. Al suo interno vengono quindi richiamate delle funzioni che si occupano praticamente della rappresentazione grafica del PopUp specifica per ogni casella.

```
PopUp::PopUp(const Andata& info, QPixmap *pic, QWidget *parent)
```

Poiché la differenziazione viene eseguita dai parametri contenuti nell'oggetto Andata &Info si rimanda alla sezione che tratta la classe Andata per ulteriori informazioni.
*QPixmap *pic* viene utilizzato per dipingere lo sfondo del PopUp per rendere l'aspetto grafico più accattivante: viene quindi utilizzato solamente dalla funzione paintEvent(QPaintEvent).
*QWidget *parent* viene utilizzato per creare il sottooggetto di PopUp, in quanto PopUp è derivato pubblicamente da QWidget e quindi rendere PopUp figlio di qualche altro oggetto.
A seconda delle informazioni che devono venir visualizzate (la maggior parte determinate dalla

18 E` utile notare che effettivamente PopUp non esegue nessuna decisione logica riguardo a cosa far visualizzare: tutte le decisioni sono già state prese dalle caselle logiche. Quello che fa PopUp è interpretare correttamente le informazioni che gli sono state inviate dalla parte logica, senza doverle rielaborare lui stesso una seconda volta

casella in cui si trova il giocatore) i PopUp possono venir generalizzati per categorie:

1. PopUp informativo per caselle in cui non si può compiere alcuna azione
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp è Andata di Tipo Null

void PopUp::initNull(const QString &NomeCasella)
void PopUp::setNulla()

   In tal caso il PopUp non farà altro che informare il giocatore G di essere giunto nella casella C e che in tale casella non è possibile fare nulla (per via della modalità di gioco scelta)
   - *Info verso il giocatore:*
     il nome della via su cui e` finito, sebbene non sia un'informazione necessaria non è male fornirla.
     [QString &NomeCasella e` il nome della via;
   - *info verso il programma:*
     essenzialmente nulla, basta sapere che il giocatore ha capito di non poter far nulla, e tale compito è svolto alla perfezione da un pulsante di accondiscendenza che non sarà collegato a nessuna modifica di sorta.
     - "*FATTO*" è il pulsante che informa il programma di continuare con il flusso di gioco;
   - *public slots correlati:*
     - *setNulla()*[19] in realtà non fa nulla, potrebbe invocare casella->eseguiAzione(Ritorno) poiché anche per le caselle che non hanno nulla da fare è definita tale funzione, ma ciò nonostante non sortirebbe nessun effetto se non sprecare tempo e risorse. Si evita quindi di eseguire chiamate inutili;

2. PopUp informativo per caselle in cui il giocatore è costretto a pagare
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp e` Andata di Tipo Paga

void PopUp::initPaga(const QString &NomeCasella, int DindiniDaPagare, Giocatore *g)
void PopUp::setPagamento()

   In tal caso il PopUp informerà il giocatore G di essere giunto sulla casella C e di dover pagare o la Banca o un altro giocatore G2
   - *info verso il giocatore:*
     il nome della via su cui è finito, la quantità di ghiande da dover pagare e il nome a cui il dazio andrà versato. L'identità di colui che riceverà le ghiande e` fornita mediante puntatore (nel caso il puntatore sia nullo, si e` scelta la convenzione che il quantitativo di ghiande andrà devoluto alla Banca; in caso contrario denota un reale giocatore e quindi, attraverso il puntatore e` possibile risalire al suo nome di gioco).
     [QString &NomeCasella e` il nome della via
     [int DindiniDaPagare e` l'ammontare in ghiande del dazio
     *[Giocatore *g* punta al giocatore beneficiario della somma da versare
   - *info verso il programma:*
     essenzialmente nulla, dato che il programma è già a conoscenza del fatto che in ogni caso il giocatore G deve pagare il dazio. Non essendoci una decisione arbitraria da parte del giocatore, al programma basta essere consapevole del fatto che il giocatore sappia di quante ghiande dovrà privarsi. Il singolo pulsante "PAGA" non lascia possibilità di scelta ed adempie magnificamente allo scopo
   - *public slots correlati:*
     - *setPagamento()* non fa altro che richiamare il metodo relativo alla casella che già

19 Per evitare ripetitività è da considerarsi il pulsante "FATTO" legato allo slot setNulla() ad ogni sua futura occorrenza

sapeva di dover diminuire le finanze di un giocatore.
casella->eseguiAzione(Ritorno(giocatore, 3, 0)) decurterà le finanze di giocatore di una quantità definita staticamente per via della casella su cui si è, non è necessario un parametro che lo specifichi.

3. PopUp informativo per caselle Edificabili
Il parametro fondamentale che caratterizza tale specializzazione di PopUp è Andata di Tipo Edif

void PopUp::initEdif(const QString &NomeCasella, bool primaOpzione, bool secondaOpzione, int costo, int possiedoDenaro)
void PopUp::setCompraTerreno()
void PopUp::setCompraAlbergo()

In tal caso il PopUp informerà il giocatore G di essere giunto sulla casella C e di avere la possibilità di comprare il terreno (poiché attualmente il terreno non ha il proprietario) o edificare l'albergo (in caso G sia il proprietario del terreno), una possibilità esclude l'altra.

- *info verso il giocatore:*
  come sempre le solite informazioni descrittive come nome della via, quantità di ghiande necessarie per svolgere l'investimento, la possibilità di fare l'investimento (nel caso non si abbiano abbastanza ghiande viene negata la possibilità di comprare il terreno o l'albergo).
  A seconda dell'opzione disponibile si farà il display di una frase differente (più a scopo di intrattenimento che di pura utilità pratica)
  [Qstring &NomeCasella è il nome della casella
  [bool primaOpzione identifica se il terreno è in vendita (perché non è ancora stato comprato)
  [bool secondaOpzione identifica se l'albergo è edificabile (perché non è ancora stato costruito)
  [int costo è la quantità di ghiande necessaria ad eseguire l'investimento
  [int possiedoDenaro identifica in realtà un valore booleano (è un intero per evitare di sprecare memoria aggiuntiva quanto si avevano già campi dati inutilizzati, vedi Andata) che dichiara se il giocatore possiede abbastanza ghiande per eseguire l'investimento. In caso non ne abbia viene preclusa la possibilità di comprare/costruire e viene avvertito il giocatore di tale fatto mediante le solite scritte informative.
- *info vesro il programma:*
  in questo caso il giocatore ha la possibilità di eseguire una scelta arbitraria: comprare il terreno/edificare l'albergo o non fare nulla.
  Per ciascuna possibilità è stato creato un pulsante apposito:
  - "*COMPRA*"/"*EDIFICA*": uno esclude l'altro e l'unica cosa che cambia è il nome che viene visualizzato. Viene disabilitato nel caso il valore possiedoDenaro sia false. Viene visualizzato se e solo se è teoricamente possibile eseguire una delle due opzioni (quindi la quantità di ghiande non influisce sulla sua visibilità, solo sulla sua cliccabilità). Informa il programma che il giocatore ha scelto di comprare/costruire (la distinzione e` eseguita a livello deterministico in quanto le due azioni non possono essere svolte contemporaneamente: una esclude l'altra)
  - "*FATTO*" e` il solito pulsante che informa il programma di non fare nulla.
- *public slots correlati:*
  - *setCompraTerreno()* è richiamato quando il giocatore decide di comprare l'albergo. eseguiAzione lavora con il parametro Ritorno(giocatore, 0, 0). Il valore in ghiande del terreno da decurtare al giocatore è definito staticamente dalla casella in cui si è, non è necessario un parametro che lo specifichi.
  - *setCompraAlbergo()* è richiamato quando il giocatore decide di comprare l'albergo.

eseguiAzione lavora con il parametro Ritorno(giocatore, 1, 0). Il valore in ghiande dell'albergo da decurtare al giocatore è definito staticamente dalla casella in cui si è, non è necessario un parametro che lo specifichi.

4. PopUp informativo per caselle Ipotecabili
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp è Andata di Tipo Ipot.

void PopUp::initIpot(const QString &NomeCasella, int ipotecaAttuale, int ipotecaMinima, int costoAlbergo)
void PopUp::setVariazioneIpoteca(int percentuale)

   In tal caso il PopUp informerà il giocatore G di essere arrivato nella casella C di sua proprietà e che ha la possibilità di ipotecare il suo albergo in caso lo voglia. Questo PopUp verrà visualizzato se e solo se si è scelta la modalità di gioco Ipotecabile e se si ha già costruito l'albergo. In caso non si sia già costruito l'albergo verrà visualizzato PopUp::initEdif
   - *info verso il giocatore:*
     come sempre le solite informazioni descrittive come il nome della via. Per motivi logistici di mancanza di spazio, si è separata la visualizzazione descrittiva iniziale con l'effettiva manipolazione dell'ipoteca: si può passare dalla prima alla seconda cliccando semplicemente un pulsante che non fa altro che nascondere le informazioni precedenti e rendere visibili quelle successive (elaborate indipendentemente dall'effettiva pigiatura del pulsante)
     [Qstring &NomeCasella è il nome della casella su cui si sta soggiornando
     [int ipotecaAttuale è la percentuale di albergo attualmente ipotecato
     [int ipotecaMinima è la soglia sotto la quale, considerate le risorse ghiandifere attualmente disponibili negli incavi degli alberi del giocatore, non e` possibile scendere: si hanno abbastanza ghiande solo per riscuotere (ipotecaAttuale – ipotecaMinima)% d'ipoteca
     [int costoAlbergo è il costo dell'albergo, viene utilizzato per calcolare l'ammontare di un 10% d'ipoteca dell'albergo
   - *info verso il programma:*
     in questo caso il giocatore ha la possibilità di eseguire una scelta arbitraria: ipotecare/de-ipotecare l'albergo o non fare nulla.
     Per ciascuna possibilità è stato creato un pulsante apposito:
     - "*MODIFICA IPOTECA*" che informa il programma di modificare l'ipoteca alla percentuale specificata dal giocatore. Per una trattazione più dettagliata visionare la sezione riguardante BarraIpoteca in BARRE UTILI poiché il pulsante viene creato tramite detta utilità.
     - "*FATTO*" è il solito pulsante per gettare la spugna e non fare nulla;
   - *public slots correlati:*
     - *setVariazioneIpoteca(int X)* informa la parte logica di variare l'ipoteca dell'immobile ed aumentarla/diminuirla del X%. eseguiAzione lavora con il parametro Ritorno(giocatore, 2, percentualeVariazione);

5. PopUp informativo per caselle Private
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp è Andata di Tipo Priv

void PopUp::initPriv(int costo, int attuale, int totale, int limite)
void PopUp::setVariazioneAzioni(int percentuale)

   In tal caso il PopUp informerà il giocatore G di essere arrivato nella casella C in cui sarà

possibile scegliere di acquistare delle quote della società a cui fa riferimento la casella stessa. Tale opportunità verrà visualizzata solo se la partita è giocata in modalità Privatizzata

- *info verso il giocatore:*
  oltre al solito nome della casella recuperato tramite il puntatore alla stessa, vengono forniti i dati riguardanti la società come il costo di una quota al 10%, la quantità di quote acquistate dal giocatore G e la quantità totale delle quote che sono state privatizzate in modo da far evidenziare al giocatore l'impossibilità di acquistarne di ulteriori (sia nel caso non bastassero le ghiande sia nel caso in cui la percentuale di privatizzazione della società abbia già raggiunto il 50%). Tutto questo viene fornito utilizzando l'ausilio della classe BarraPrivata che verrà discussa in seguito
  [int costo è l'ammontare in ghiande di un 10% delle azioni della società a cui fa capo il palazzo su cui si è arrivati
  [int attuale è la percentuale delle azioni già acquistate dal giocatore G
  [int totale è la percentuale totale di privatizzazione della società, serve per stabilire il limite oltre al quale un giocatore, sebbene ne abbia la possibilità, non possa acquistare partecipazioni azionarie
  [int limite è la percentuale totale di privatizzazione della società a cui il giocatore G può aspirare dando fondo a tutte le sue ghiande.
  Il limite effettivo oltre il quale sarà disabilitata la possibilità di modificare la partecipazione azionaria del giocatore G sarà il minore tra i due limiti sopra citati: la scissione dei due limiti serve oltretutto per discernere tra i due casi per cui il giocatore G non sia in grado di acquistare ulteriori partecipazioni azionarie, il giocatore viene informato attraverso una selezione di due stringhe
- *info verso il programma:*
  in questo caso il giocatore ha la possibilità di decidere di quanto modificare la propria partecipazione azionaria (sempre nei limiti consentiti tra disponibilità finanziarie e tetto massimo di percentuale privatizzabile). L'alternativa si riduce sempre a due pulsanti:
  - "*MODIFICA AZIONI*" che aggiornerà la percentuale di azioni detenuta dal giocatore, sia che sia appena stata aumentata che decrementata. Per una trattazione più dettagliata fare riferimento a BarraPrivata
  - "*FATTO*" il solito pulsante per avvertire di aver ponderato le possibilità disponibili ma non voler/poter fare nulla di propositivo
- *public slots correlati:*
  - *setVariazioneAzioni(int X)* informa la parte logica di variare la percentuale di quote azionarie del giocatore G dell'X%. eseguiAzione lavora con il parametro Ritorno(giocatore, 4, percentualeVariazione)

6. <u>PopUp informativo per la casella della Polizia</u>
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp e` Andata di Tipo Polz

void PopUp::initPolz(int ammenda) void PopUp::setImprigionato(int numTurni)

In tal caso PopUp informerà il giocatore G di essere arrivato nella casella della Polizia (e non è quindi necessario specificare il nome della via, poiché di polizia ce n'è una sola) e di dover tentare la sorte per scoprire se dovrà passare la notte in cella e pagare un'ammenda oppure di continuare indisturbato per la sua strada. Tale possibilità si verifica solamente se la partita è stata iniziata impostando la modalità Legale, altrimenti il PopUp informerà il giocatore di essere arrivato su di una casella in cui non può compiere alcuna azione (PopUp::initNull)

- *info verso il giocatore:*
  il giocatore verrà principalmente informato di quel che deve fare, tirare la moneta, e di

quanto ammonta la penale in caso sia sfortunato
[int ammenda altro non è che l'ammontare in ghiande della somma da versare nel caso, tirando la moneta, si scopri di esser stati fermati dal servizio dei gendarmi

- *info verso il programma*[20]*:*
  effettivamente il giocatore è vincolato a dover lanciare la moneta, ciò nonostante si è deciso che il risultato del lancio della moneta sia calcolato al momento del suo lancio (e non quando vengono create le informazioni dalla casella in questione). Il programma deve quindi ricevere l'informazione dell'esito della moneta che equivale a sapere se il giocatore deve essere imprigionato per un turno o non essere imprigionato.
  - "*LANCIA MONETA*" è il pulsante che dà il via al lancio della moneta e ne sancisce il risultato. Viene implementato attraverso BarraPolizia
  - "*ASSECONDA LA GIUSTIZIA*" che verrà abilitato solo dopo che la moneta si sarà fermata ed avrà prodotto il risultato. Serve al programma per sapere che il giocatore ha compreso la sorte a cui sta andando incontro e per far aggiornare la situazione penale del giocatore stesso
- *public slots correlati:*
  - *setImprigionato(int X)* informa la parte logica che il giocatore in questione dovrà saltare i successivi X turni di gioco. eseguiAzione lavora con il parametro Ritorno(giocatore, 0, numTurni).

7. <u>PopUp informativo per la casella Tribunale</u>
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp e` Andata di Tipo Trib

void PopUp::initTrib()
void PopUp::setImprigionato(int numTurni)

   In tal caso il PopUp informerà il giocatore G di essere giunto al tribunale e di dover quindi lanciare un dado per scoprire per quanti turni dovrà restare imprigionato. Tale possibilità si verifica solamente se la partita è stata iniziata impostando la modalità Legale, altrimenti il PopUp informerà il giocatore di essere arrivato su di una casella in cui non può compiere alcuna azione (PopUp::initNull)

   - *info verso il giocatore:*
     le uniche informazioni che vengono fornite al giocatore sono il nome della casella (che non ha bisogno di parametri dedicati in quanto di tribunale ce n'è uno solo) e la modalità in cui si deciderà delle sue sorti future. Al giocatore viene chiesto di lanciare il dado senza il cui lancio non si potrebbe continuare
   - *info verso il programma*[21]*:*
     sebbene il giocatore si trovi vincolato a dover lanciare un dado per poter continuare a giocare, il risultato del dado sarà calcolato all'istante in cui verrà fatto rotolare e non predeterminato da quando il giocatore è finito su questa casella. Il risultato di tale lancio sarà indispensabile al programma. Come per la polizia anche per il tribunale son sufficienti due pulsanti:
     - "*LANCIA DADO*" che è auto-esplicativo. Viene implementato effettivamente da BarraTribunale per comodità
     - "*ASSECONDA LA GIUSTIZIA*" che verrà abilitato solamente dopo che il dado si sarà fermato ed avrà quindi prodotto il risultato. Serve al programma per capire che il giocatore ha compreso la sua triste sorte di prigioniero e per aggiornare la situazione legale del giocatore in questione

20 In realtà le informazioni verso il programma sono raccolte e gestite da BarraPolizia attraverso i suoi pulsanti: PopUp fa solo da tramite alle informazioni racimolate. È comunque logicamente corretto parlarne come se fossero di PopUp dato che BarraPolizia è figlio di PopUp, è posseduto da lui ed è una sua estensione a tutti gli effetti. La divisione è stata creata per comodità.

21 Come la nota 4. In questo caso è BarraTribunale a gestire le informazioni tramite i suoi pulsanti.

- *public slots correlati:*
  - *setImprigionato(int X)* vedi PopUp::initPolz()

Sebbene i PopUp informativi relativi a tutte le categorie di caselle sono finiti, rimangono sospesi ancora 3 PopUp di evidente utilità

8. PopUp informativo per Incarceramento
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp e` Andata di Tipo Ferm

void PopUp::initFerm(int x)

   In tal caso il PopUp informerà il giocatore G di dover scontare ancora tot turni in prigione
   - *info verso il giocatore:*
     le due informazioni riferite saranno il numero di turni residui da scontare, e il luogo in cui li si sta scontando: la prigione.
     [int x è il numero di turni residui da scontare in prigione
   - *info verso il programma:*
     poiché il PopUp è puramente informativo non ci sono informazioni rilevanti da acquisire. Al programma basta sapere se il gioco può continuare.
     - "*FATTO*" è il solito pulsante che avverte il programma che l'utente ha letto quanto doveva leggere per rimanere aggiornato sulle sorti del suo alter-ego e che quindi può continuare con il proseguimento del gioco

9. PopUp informativo per il Decesso
   Il parametro fondamentale che caratterizza tale specializzazione di PopUp è Andata di Tipo Mort

void PopUp::initMort()

   In tal caso il PopUp informerà il giocatore G di essere deceduto poiché in bancarotta.
   - *Info verso il giocatore:*
     l'unica informazione riportata all'attenzione del giocatore è la sua morte
   - *info verso il programma:*
     poiché il PopUp è puramente informativo non ci sono informazioni rilevanti da acquisire. Al programma basta sapere se il gioco può continuare.
     - "*FATTO*" è il solito pulsante che avverte il programma che l'utente ha letto quanto doveva leggere le rimanere aggiornato sulle sorti del suo alter-ego e che quindi può continuare con il proseguimento del gioco

10. PopUp informativo per la Vittoria
    Il parametro fondamentale che caratterizza tale specializzazione di PopUp e` Andata di Tipo Vitt

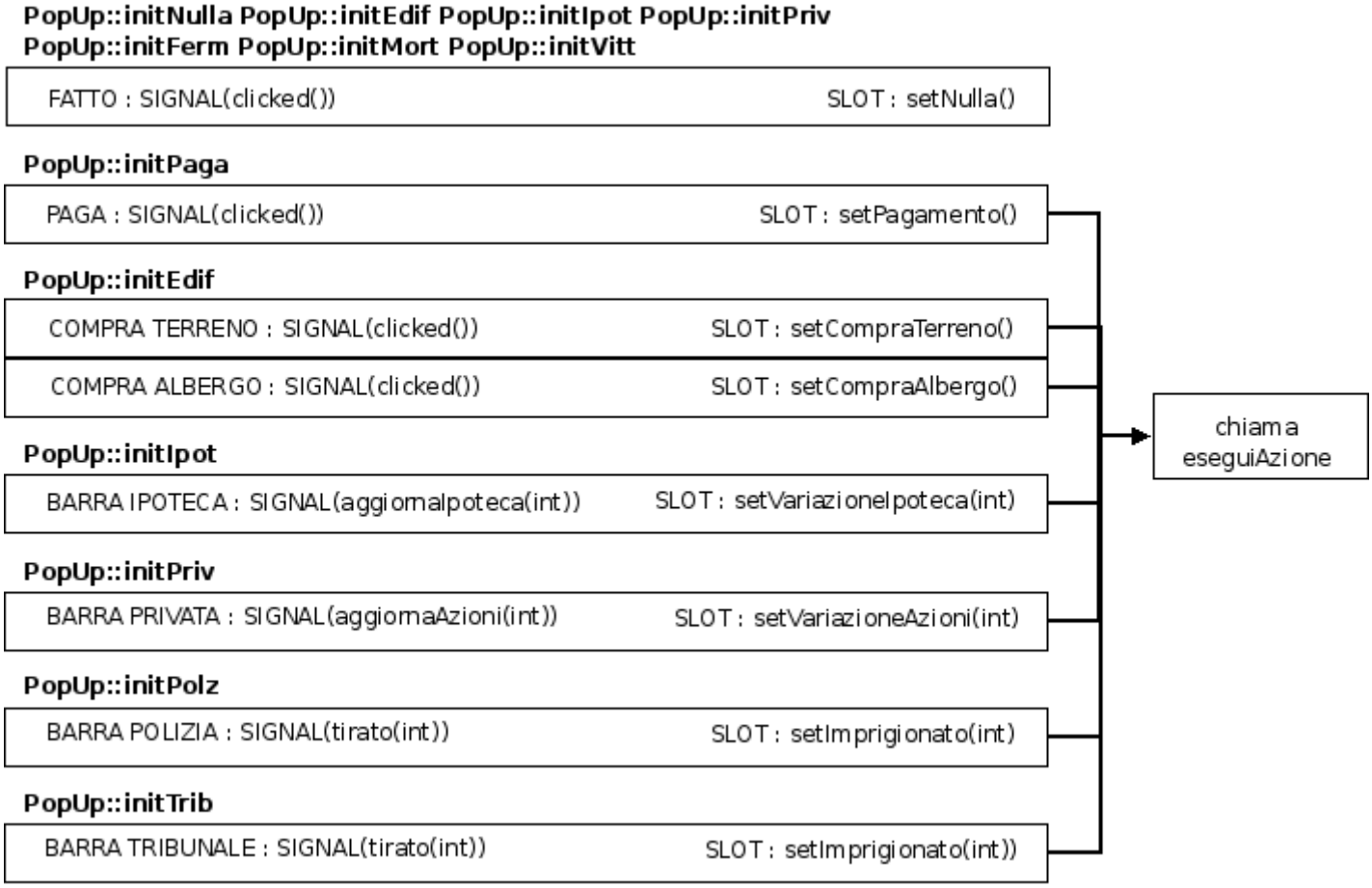
void PopUp::initVitt(int soldi, bool tipoVittoria)

    In tal caso il PopUp informerà il giocatore G di aver vinto la partita. Poiché i modi per vincere la partita sono due (o per limite massimo di giri o per decedimento generale altrui) il PopUp si preoccuperà di avvisare in quale dei due casi ci si trova
    - *info verso il giocatore:*
      essenzialmente che ha vinto e con quanti soldi ha vinto. Tramite poche parole ad effetto riuscirà ad evidenziare in che modo è stata vinta, e quindi conclusa, la partita
      [int soldi è la quantità di ghiande possedute, un'informazione puramente narcisistica e talvolta ironica

[bool tipoVittoria distingue tra i due tipi già menzionati prima. A seconda del suo valore verranno visualizzate delle frasi piuttosto che altre. Un'informazione puramente narcisistica e talvolta ironica

- *info verso il programma:*
  sapere che tutti gli utenti abbiano recepito il messaggio finale, ed avere la possibilità di terminare
  - "*FATTO*" è il solito pulsante per avvertire il programma di continuare con il suo flusso di gioco e quindi, in questo caso, di terminare

Uno schema per riassumere quali connect sono presenti in PopUp.
I signal clicked son da considerarsi riferiti ai pulsanti.
Per non complicare il diagramma è stato omesso che ogni SLOT compie in sequenza: un EMIT di SIGNAL(chiudi()) ed invoca lo SLOT(close()).



Sulla disposizione grafica:
si e` presentata la duplice alternativa di come poter gestire la grafica inerente a PopUp:

- tramite *setGeometry(int x, int y, int width, int height)* : settare la posizione con le coordinate di pixel di base di ciascun oggetto interno al PopUp e di settarne pure le dimensioni.
  Questo strumento impone principalmente di dover trattare ogni singola entità presente nel PopUp come un componente disgiunto da tutto il resto, completamente svincolato dalla disposizione complessiva. Questa caratteristica e` sia uno svantaggio che un vantaggio.
  Lo svantaggio riguarda la scarsa comodità che ne deriva nell'ottica dell'estensibilità: dover riorganizzare l'intera finestra grafica (o per lo meno buona parte) solamente per aggiungere un QPushButton, un QLabel o qualsiasi QObject. Doverlo fare a mano, pixel per pixel, senza l'ausilio di strutture che si preoccupino di trattare autonomamente la posizione (come i Layout).

Il vantaggio risiede nell'ampia maneggevolezza e duttilità dell'aspetto grafico: non dover sottostare a vincoli imposti da altre strutture (come i Layout), ma poter gestire gli spazi (esigui o ampi che siano a seconda dei casi) come si ritiene più opportuno.
Con l'aggiunta di un semplice lavoro lo svantaggio viene ridotto: con la creazione un bozzetto, anche cartaceo, del probabile aspetto del PopUp con relativa disposizione degli oggetti e misurazioni in pixel (anche approssimativi) si riesce a gestire la grafica e le riorganizzazioni degli oggetti in maniera molto più fluida e meno drammatica.

– tramite *setLayout(QLayout *l)*: delegare il compito di organizzazione spaziale alle strutture QLayout.
Il vantaggio è evidente: la grafica viene interamente gestita da una struttura cui basta essere informata sulla consequenzialità degli oggetti da disporre, e nulla di più, lei si occuperà di prendere le misure necessarie.
Lo svantaggio è più sottile: essendo i Layout una struttura dinamica, calcola la posizione dei vari oggetti al suo interno passo passo. Per esempio, la modifica del testo di una QLabel, soprattutto durante il turno di un giocatore, porta ad una riorganizzazione totale dell'intero aspetto del PopUp, distruggendo la continuità visiva dell'oggetto. Si potrebbe ovviare utilizzando gli indici di 'stretch' nella costruzione del Layout, ma programmare in anticipo cose simili risulta morboso e controproducente, in quanto può scombinare completamente l'impatto grafico. Altra limitazione portata dai Layout sta nell'armoniosità del complesso: scritte e pulsanti possono essere separate da spazi vuoti senza significato. QGridLayout non è una valida alternativa, in quanto calcolare i 'quadratini' occupati dai singoli oggetti risulta molto più tedioso e difficoltoso che trattare direttamente la loro posizione con coordinate pixel.

Si e` quindi deciso di utilizzare una combinazione pratica di entrambi gli aspetti.
Grazie ai QLayout si possono compattare assieme diversi oggetti che non hanno bisogno di modifiche o riorganizzazione durante il periodo di gioco di un singolo utente: un esempio e` l'unione di pulsanti in una barra omogenea con conseguenti QLabel, minimizzando in tal modo la necessita` di gestire ogni singolo QObject attraverso le sue coordinate.
Grazie al setGeometry si può organizzare l'intero aspetto del popUp in un'omogeneità estetica non raggiungibile attraverso i QLayout: avere la possibilità di gestire gli spazi a proprio piacimento rendendo così decisamente più accattivante l'intero popUp ed evitare la sua riorganizzazione topografica dovuta all'autogestione geometrica dei QLayout.
La scelta è stata vincolata maggiormente dal gusto estetico dei programmatori, finalizzata alla visualizzazione di un prodotto dalla grafica non banale e divertente.
È comunque possibile convertire i setGeometry in QLayout senza uno sforzo eccessivo: basta eliminare le righe di codice ove compare il QObject.setGeometry() e creare dei QLayout aggiuntivi ed inserirvi gli oggetti appena svincolati dal setGeometry. Sebbene tale modifica non sia laboriosa, lo sarebbe leggermente di più riorganizzare tutte le scritte, in modo da permettere la loro completa visualizzazione, che appunto verrebbe troncata e limitata dall'autogestione della geometria attraverso i QLayout: le scritte di informazione sono prolisse e non contengono solo le informazioni strettamente necessarie per far prendere una decisione all'utente, ma contengono pure frasi taglienti ed a volta ironiche volte a far sorridere gli utenti, scelta che ha permesso di rendere, si spera, meno banale e più simpatico l'utilizzo del gioco.

Estendibilità:
PopUp è vincolato alle regole grafiche delle librerie grafiche Qt, quindi nel caso si dovessero cambiare librerie bisognerebbe ricreare completamente il codice.
Naturalmente il suo contratto rimane ed il nuovo codice dovrebbe mantenere i contatti con le caselle in modo da poter richiamare casella->eseguiAzione(Ritorno(XXX)).
Nel caso si vogliano ampliare le proprietà del gioco inserendone di nuove si prospettano due casi:

1. la nuova opzione è da inserire in una specializzazione preesistente : in tal caso l'unica cosa da fare è trovare un modo per farci stare nel poco spazio disponibile la descrizione della nuova opzione e nuovi pulsanti (es: eliminando le scritte già esistenti e ridurre all'osso,

creare un nuovo pulsante che copra la parte vecchia e visualizzi quella nuova come se si sfogliassero delle pagine. Unico vincolo è la fantasia).

2. la nuova opzione è da ritenersi completamente distinta dalle preesistenti: in tal caso la cosa è molto più semplice in quanto basta inserire una nuova if-statement nel corpo del costruttore di PopUp, inventare un nuovo valore per enum Tipo e creare da zero la rappresentazione grafica.

In entrambi i casi non c'è molto lavoro da fare, ed è difficile pensare di riuscire a far ancora meno lavoro. L'unica cosa che aiuterebbe sarebbe avere scritte meno prolisse ma di questo si è già parlato nel paragrafo di “Sulla Disposizione Grafica”.

Indipendenza:
PopUp non interagisce con la parte logica se non quando manda l'input per iniziare ad elaborare le informazioni che si sono appena raccolte dall'utente. Questo unico contatto è ovviamente necessario altrimenti se la parte grafica e la parte logica non avessero nessun punto in comune non si riuscirebbe a far funzionare il gioco poiché il dialogo è necessario.
Questo contatto con la parte logica è altresì minimo, in quanto invoca una singola azione che è stata strutturata appositamente per elaborare i dati provenienti dall'ambiente grafico (raccolti in un oggetto Ritorno), e non è lo stesso PopUp che si permette di modificare i dati che appartengono alle strutture logiche.

BARRE UTILI

Le barre utili altro non sono che piccoli widget appositamente finalizzati per compiere una data funzione per PopUp. La loro costituzione è stata dettata sia per esigenze estetiche che per necessità semplificative. Dotare PopUp di un widget esterno rendeva il PopUp stesso meno specializzato e più comprensibile e leggero il suo codice (es: dotare ogni PopUp di una moneta, che veniva poi nascosta, solo perché la moneta era utilizzata da una singola casella)
Effettivamente le BarreUtili elaborano i dati forniti da PopUp per limitare le decisioni degli utenti: se Andata calcola i vari limiti delle azioni eseguibili dal giocatore, se PopUp informa l'utente di tali limiti attraverso delle scritte, le BarreUtili fanno in modo che tali limiti vengano rispettati (ad esempio disabilitando un pulsante ed abilitandone un altro).
Per evitare ripetizioni tutte le BarreUtili hanno il distruttore il cui copro dealloca i pochi pulsanti/scritte allocati sullo heap e che si comporta allo stesso modo in ciascuna situazione: poiché sullo heap vengono allocati anche oggetti del cui puntatore non si tiene traccia, e poiché sono oggetti derivanti da QWidget, tali oggetti saranno deallocati automaticamente dal distruttore virtuale del sotto-oggetto QWidget poiché tutti tali oggetti saranno costruiti rendendoli figli della BarraUtile che deriva da QWidget.

Poiché le BarreUtili sono a tutti gli effetti estensioni di PopUp, oggetti creati per non appesantire il codice di PopUp e per assecondare l'idea dell'incapsulamento, le motivazioni per quel che concerne Grafica, Estendibilità e Indipendenza sono le medesime descritte nella sezione di PopUp.

BARRA IPOTECA

BarraIpoteca è colei che si occupa di far visualizzare graficamente una barra scorrevole che indicizza l'effettiva percentuale di ipoteca pendente sul proprio albergo, ed offre la possibilità di incrementarla o decrementarla attraverso appositi pulsanti

```
public slots:
 void modificaIpoteca();
 void aggiornaIncremento();
 void aggiornaDecremento();
signals:
 void aggiornaIpoteca(int newValue);
private:
 int ipotecaAttuale;
 int ipotecaMinima;
 int costoAlbergo;
 QLabel *info;
 QPushButton *invia;
 QPushButton *incrementa;
 QPushButton *decrementa;
 QProgressBar *percentuale;
 void cHeckValue(int newValue);
public:
 BarraIpoteca(int, int, int, QWidget *parent=0);
 ~BarraIpoteca();
```

1. le uniche due funzionalità pubbliche sono il costruttore ed il distruttore.
   Al costruttore servono solamente i tre parametri ipotecaAttuale, ipotecaMinima e costoAlbergo che verranno poi salvati sulle apposite variabili, e sapere chi è il parent del widget in modo da diventare suo figlio e poter essere distrutto quando il padre verrà distrutto (per via delle proprietà dei distruttori dei QWidget).

2. I dati statici che si preoccupano di fornire i limiti alla percentuale di ipoteca sono:
   - *int ipotecaAttuale* che determina l'attuale percentuale di ipoteca dell'immobile
   - *int ipotecaMinima* determina la percentuale minima a cui può aspirare di arrivare il giocatore nel caso volesse riscuotere parte dell'ipoteca: non potrebbe riscuoterne di più (e quindi il valore non potrebbe essere più basso) a causa delle scarse risorse finanziarie o perché è già al minimo.
   - *int* costoAlbergo non è effettivamente necessario per tale Widget, viene utilizzato solo per aumentare le informazioni fornite al giocatore: quanti soldi spenderebbe/guadagnerebbe nel modificare del X% l'ipoteca del suo albergo. Questo dato serve solo per dare l'idea al giocatore di come si evolverà la sua situazione finanziaria, non serve assolutamente per elaborare dati in quanto la casella logica possiederà già il valore del 10% dell'ipoteca.

3. *QLabel * invia* serve per fornire al giocatore delle informazioni quali la quantità in ghiande che guadagnerebbe/spenderebbe per modificare l'ipoteca o il motivo per il quale non è possibile modificare di tanto l'ipoteca del proprio albergo. Le diverse frasi da visualizzare sono gestite dalla funzione checkValue(newValue)

4. *QPushButton *invia* è il pulsante che segnala la decisione presa e informerà PopUp di modificare a dovere l'ipoteca. La sua cliccabilità è sancita dalla funzione checkValue.
   - *void checkValue(int newValue)* è la funzione che si preoccupa di abilitare e disabilitare il tasto invia a seconda dei casi. Calcola se newValue è contenuto nel range statico di ipotecaMinima (e quindi se il giocatore possiede abbastanza ghiande per poter riscattare l'ipoteca del suo albergo fino ad ipotecaMinima) e 100 estremi inclusi. Nel caso esca dal range il pulsante di invio dei dati viene disabilitato. Il pulsante di invio dei dati viene disabilitato anche quando, dopo varie modifiche, la percentuale di ipoteca ritorna a ipotecaAttuale: ciò per evitare chiamate inutili a delle funzioni.

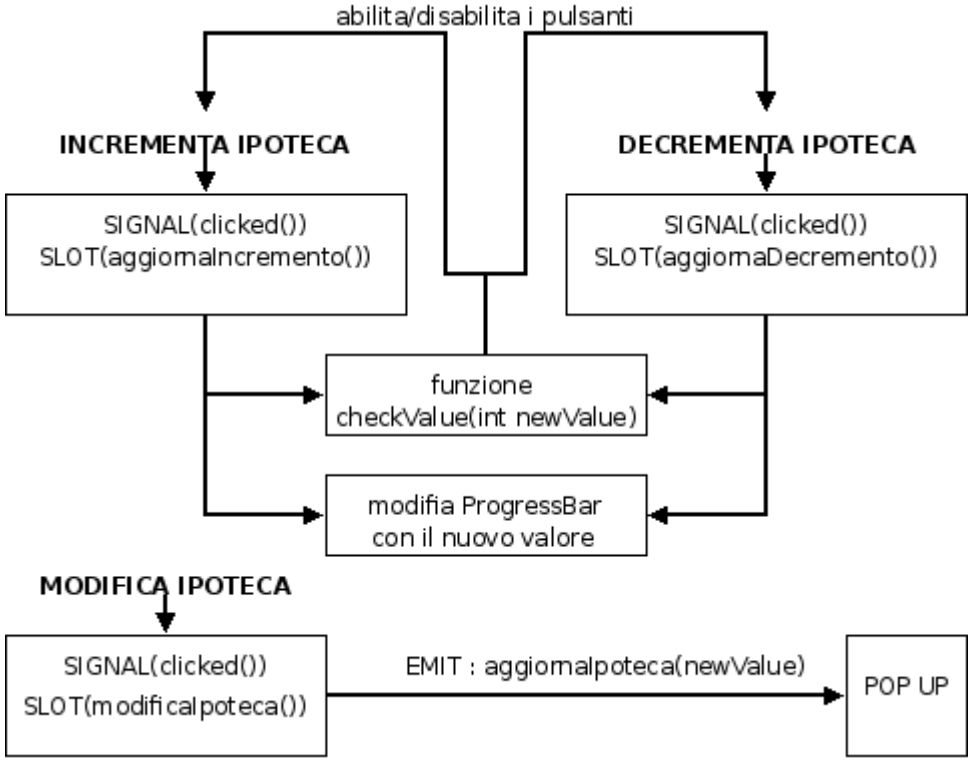
5. *QPushButton *incrementa/*decrementa* sono i pulsanti che permettono all'utente di aumentare o diminuire la percentuale di ipoteca: tali pulsanti vengono attivati e disattivati a

seconda della percentuale a cui si è arrivati. Gli slots che si occupano della loro cliccabilità sono *aggiornaIncremento* e *aggiornaDecremento* che lavorano in coppia. Entrambi gli slot chiamano la funzione checkValue, con il nuovo valore della percentuale appena calcolato, per aggiornare la cliccabilità del pulsante invia.
- *void aggiornaIncremento()*: aumenta del 10% la percentuale di ipoteca a cui si è arrivati. Il valore temporaneo è acquisito dalla QProgressBar, viene incrementato e quindi reinserito nella stessa QProgressBar per mantenerla aggiornata ad ogni click.
Se il valore incrementato è arrivato a 100 viene disabilitato il pulsante di incremento.
Se il valore incrementato è arrivato a 10 il pulsante di decremento viene abilitato.
- *void aggiornaDecremento()*: decrementa del 10% la percentuale di ipoteca a cui si è arrivati. Il valore temporaneo è acquisito dalla QProgressBar, viene decrementato e quindi reinserito nella setessa QProgressBar per mantenerla aggiornata ad ogni click.
Se il valore decrementato è arrivato a 0 viene disabilitato il pulsante di decremento.
Se il valore decrementato è arrivato a 90 viene abilitato il pulsante di incremento.

6. *QProgressBar *percentuale* è semplicemente la rappresentazione grafica della percentuale di ipoteca a cui si vorrebbe arrivare, viene modificata passo passo ad ogni pigiatura sui pulsanti di incremento e decremento. Serve anche come contenitore temporaneo della percentuale di ipoteca a cui si vorrebbe giungere (al posto di utilizzare un altro intero per tale scopo).

7. *void modificaIpoteca()* è in relazione con void aggiornaIpoteca(int newValue). ModificaIpoteca è in realtà uno slot fittizio che collega la pigiatura del pulsante INVIA con il signal aggiornaIpoteca(X). Viene fatto ciò perchè le Qt non permettono connect tra funzioni con diversa lista di parametri. X è la variazione della percentuale di ipoteca sull'albergo che si vuole apportare, e viene calcolata semplicemente con una sottrazione: valorePercentualeModificato (reperito dalla QProgressBar) – IpotecaAttuale.
Se il valore è positivo si sta cercando di aumentare l'ipoteca
Se il valore è negativo si sta cercando di decrementare l'ipoteca

La struttura schematizzata di come funziona BarraIpoteca:



BARRA PRIVATA

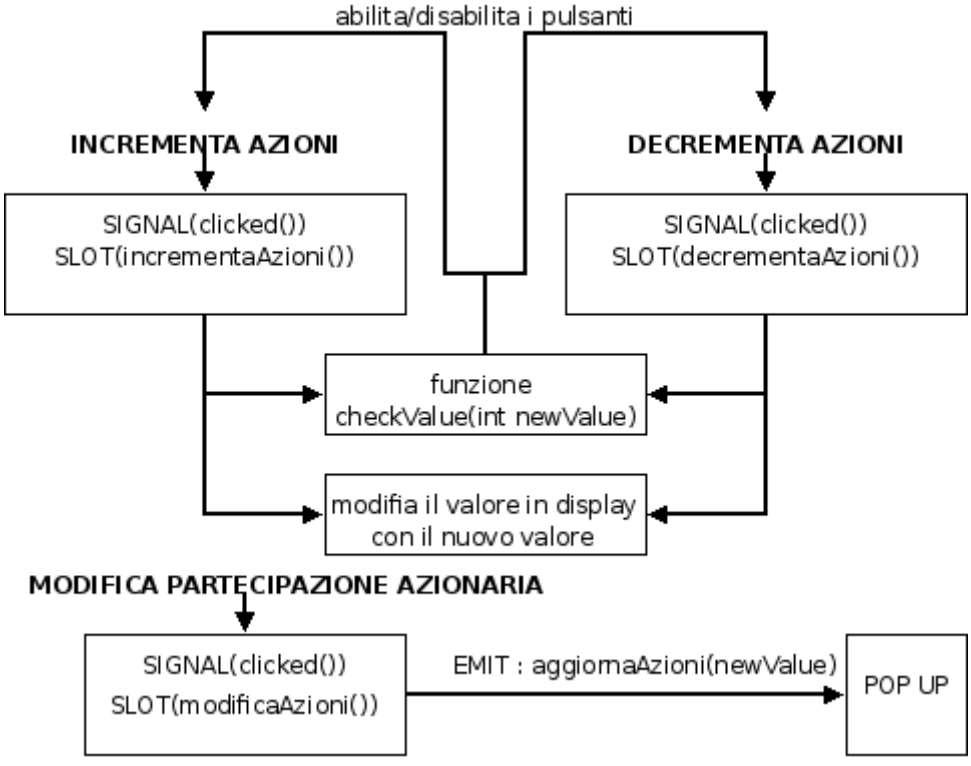
BarraPrivata è colei che si occupa di far visualizzare graficamente la percentuale che indicizza l'effettivo ammontare di azioni di una società possedute dal giocatore stesso e quelle che i giocatori hanno già privatizzato (e che quindi non sono più pubbliche), ed offre la possibilità di incrementarla o decrementarla attraverso appositi pulsanti.
La struttura è molto simile a quella di BarraIpoteca, cambia solamente il modo in cui viene visualizzato l'ammontare della percentuale: invece di una QProgressBar (che non avrebbe adempiuto efficientemente allo scopo) una semplice scritta su QLabel.

```
public slots:
 void modificaAzioni();
 void incrementaAzioni();
 void decrementaAzioni();
signals:
 void aggiornaAzioni(int newValue);
private:

 int quotaTemporanea;
 int quotaAttuale;
 int quotaTotale;
 int quotaMassima;
 QLabel *display;
 QLabel *info;
 QPushButton *incrementa;
 QPushButton *decrementa;
 QPushButton *invia;
 void checkValue(int);

public:
   BarraPrivata(int, int, int, int,  const QString&, const QString&, QWidget* parent=0);
 ~BarraPrivata() {}
```

1. Le uniche due funzionalità pubbliche sono il costruttore ed il distruttore.
   Al costruttore servono i quattro valori per inizializzare le variabili nella zona privata, il nome del giocatore (solo per rendere più coerenti le scritte), il nome della casella (per poter quindi informare a quale società fa riferimento il palazzo su cui si è giunti) e il parent a cui appendere il widget
2. I dati statici che si preoccupano di gestire i limiti della percentuale di Azioni privatizzabili sono:
   - *int quotaTemporanea* che salva la percentuale modificata a cui si vorrebbe portare la propria partecipazione azionaria della società (corrisponde al valore che memorizzava la QProgressBar di BarraIpoteca)
   - *int quotaAttuale* è la percentuale di partecipazione azionaria con la quale si è partiti giungendo nella casella prima di modificarla
   - *int quotaTotale* è la percentuale di partecipazione azionaria detenuta da tutti i giocatori
   - *int quotaMassima* è la percentuale massima a cui può aspirare di arrivare il giocatore per via della sua situazione finanziaria: non può comprare più di quella percentuale di azioni che separa quotaAttuale da quotaMassima
3. *QLabel *info* informa il giocatore del costo di un'azione del 10%, della quantità di azioni privatizzate nella società e in caso, del motivo per cui non si possano comprare tante azioni (mancanza di ghiande o per sforamento del tetto del 50% totale di azioni).
4. *QPushButton *invia* è il pulsante che segnala la decisione presa e informerà PopUp di modificare a dovere la partecipazione azionaria del giocatore alla società in questione. La sua cliccabilità è sancita dalla funzione checkValue.
   - *void checkValue(int newValue)* è la funzione che si preoccupa di abilitare e disabilitare il tasto invia a seconda dei casi. Calcola se newValue è contenuto nel range statico di 0 e min[quotaTotale+variazione apportata (non acquistare più del 50% di azioni) e quotaMassima (possedere abbastanza ghiande per acquistare ulteriori azioni)] estremi inclusi. Nel caso esca dal range il pulsante di invio dei dati viene disabilitato. Il pulsante di invio dei dati viene disabilitato anche quando, dopo varie modifiche, la percentuale di partecipazione azionaria ritorna a quotaAttuale: ciò per evitare chiamate inutili a delle funzioni.

5. *QPushButton *incrementa/*decrementa* sono i pulsanti che permettono all'utente di aumentare o diminuire la percentuale di azioni possedute: tali pulsanti vengono attivati e disattivati a seconda della percentuale a cui si è arrivati. Gli slots che si occupano della loro cliccabilità sono incrementaAzioni e decrementaAzioni che lavorano in coppia. Entrambi gli slot chiamano la funzione checkValue, con il nuovo valore della percentuale appena calcolato, per aggiornare la cliccabilità del pulsante invia.
   - *void incrementaAzioni()*: aumenta del 10% la percentuale di azioni a cui si vorrebbe giungere. Il valore temporaneo modificato è acquisito e risalvato in quotaTemporanea e aggiornata nella QLabel display per mantenere aggiornata l'informazione verso il giocatore.
   Se il valore incrementato è arrivato a 50 viene disabilitato il pulsante di incremento.
   Se il valore incrementato è arrivato a 10 il pulsante di decremento viene abilitato.
   - void decrementaAzioni(): decrementa del 10% la percentuale di azioni a cui si vorrebbe giungere. Il valore temporaneo è acquisito e risalvato in quotaTemporanea e aggiornata nella QLabel display per mantenere aggiornata l'informazione verso il giocatore.
   Se il valore decrementato è arrivato a 0 viene disabilitato il pulsante di decremento.
   Se il valore decrementato è arrivato a 40 viene abilitato il pulsante di incremento.
6. *QLabel *display* rimpiazza la funzione della QProgressBar in BarraIpoteca. Mantiene aggiornato il valore di percentuale di azioni a cui si vorrebbe giungere. Il valore viene mantenuto aggiornato ad ogni pigiatura dei pulsanti di incremento e decremento per via degli slot a loro connessi
7. *void modificaAzioni()* è in relazione con *void aggiornaAzioni(int newValue)*. ModificaAzioni è in realtà uno slot fittizio che collega la pigiatura del pulsante INVIA con il signal aggiornaAzioni(X). Viene fatto ciò perchè le Qt non permettono connect tra funzioni con diversa lista di parametri. X è la variazione della percentuale di azioni sulla società che si vuole apportare, e viene calcolata semplicemente con una sottrazione: quotaTemporanea - quotaAttuale.
   Se il valore è positivo si sta cercando di aumentare la quantità di azioni possedute.
   Se il valore è negativo si sta cercando di decrementare la quantità di azioni possedute.

La struttura schematizzata del funzionamento di BarraPrivata:



BARRA POLIZIA

BarraPolizia è colei che si occupa di far visualizzare graficamente la moneta da dover lanciare e il pulsante che segnala di aver capito la propria sorte.

```
public slots:
 void tiraMoneta();
 void inviaSegnale();

signals:
 void tirato(int);

private:
 int risultato;
 int ripetizione;
 QPushButton *tira;
 QPushButton *fatto;
 QLabel *lLabel;
 QLabel *rLabel;
 QTimer *timer;

public:
 BarraPolizia(QWidget *parent=0);
 ~BarraPolizia();
```

1. Le uniche funzionalità pubbliche sono il costruttore ed il distruttore poiché il resto delle informazioni viene scambiato con PopUp attraverso apposite connect.
   Il costruttore ha solo bisogno di sapere di chi deve diventare figlio BarraPolizia.

2. I dati statici che si preoccupano di gestire i limiti del lancio della moneta sono:
   - *int risultato*: determina quanti giri deve fare la moneta prima di fermarsi, quindi è questo numero modulo2 che determina se il lancio della moneta è testa o croce. Tale numero è casuale ed è calcolato nel costruttore ed è compreso nell'intervallo chiuso [11, 12].
   - *int ripetizione*: tiene traccia di quante giravolte ha fatto la moneta. Non si poteva decrementare risultato altrimenti una volta arrivati a 0 non si sarebbe più potuti risalire al numero iniziale. Una volta arrivato a risultato vuol dire che la moneta ha finito di girarsi.

3. *QLabel *lLabel e *rLabel* sono delle scritte che suggeriscono al giocatore di cliccare sulla moneta per poter continuare con il gioco. Sono due perché son state disposte una a destra e l'altra a sinistra della moneta per ottimizzare lo spazio e per rendere la cosa più accattivante

4. *QPushButton *tira* è il pulsante su cui è disegnata la moneta. Una volta cliccato su di lui si fa partire l'animazione (in realtà si fa partire il timer che quindi si occuperà del resto, vedi . 6)

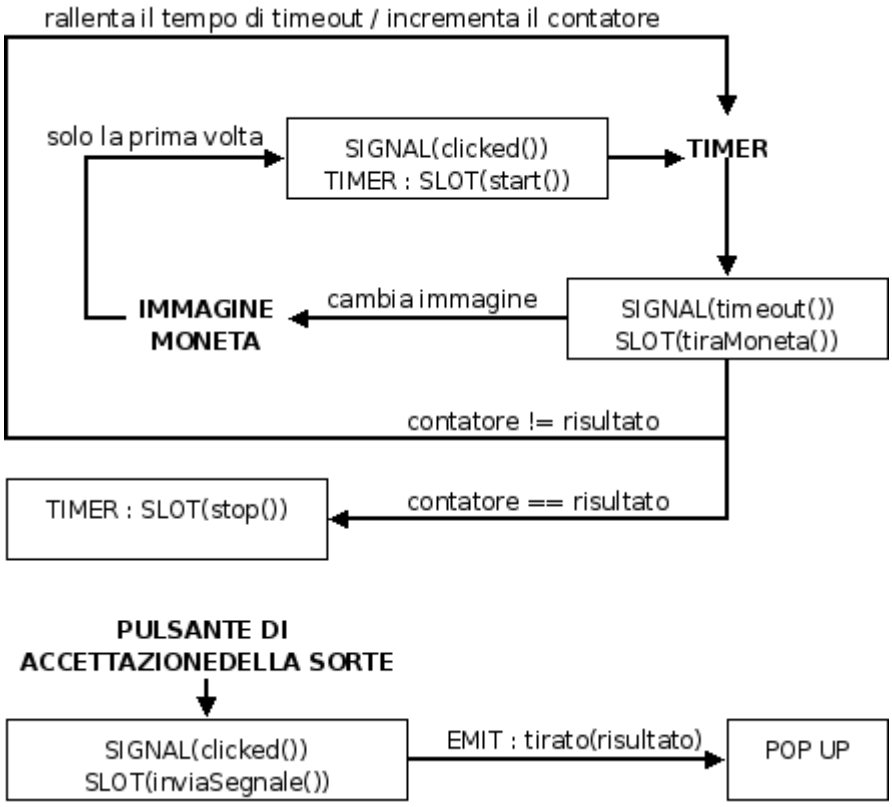
5. *QPushButton *fatto* è il pulsante che dichiara che il giocatore ha capito quale è stato il risultato del lancio della moneta. Viene abilitato solo dopo che la moneta ha smesso di girare.
   Nonostante dovrebbe rappresentare due situazioni differenti, si è riusciti a condensare in un solo pulsante l'evoluzione del gioco: la casella logica saprà che cosa fare a seconda del risultato ottenuto, e poiché i risultati possibili sono solo due non è molto difficoltoso discernere tra i due.

6. *QTimer *timer* è un semplicissimo timer utilizzato per dare un'apparenza di dinamicità al lancio della moneta e per acuire la suspance. È una infima animazione che consiste nel continuare a far vedere una faccia della moneta seguita dall'altra ad intervalli sempre più lunghi fin quanto il lancio non si ferma. Del timer sono utilizzati solo i suoi timeout per impostare i cambi di immagini e di scritte. Al timer è connesso lo slot tiraMoneta

- *void tiraMoneta()* è lo slot che viene invocato ad ogni timeout del timer. Ogni volta si preoccupa di incrementare di una unità ripetizione, di reimpostare l'intervallo successivo del timer (allungandolo per creare effetto suspance e dinamismo), di fare il corretto display della giusta faccia della moneta (dato che ogni volta devono alternarsi testa e croce), e di cambiare la scritta sul tasto FATTO per una più accurata spiegazione della futura evoluzione del gioco. Si preoccupa anche di fermare il timer quando ripetizione ha raggiunto il valore di risultato, di disabilitare il pulsante che rappresenta la moneta (così non si può ritentare la fortuna) e di abilitare il pulsante FATTO.

7. *void inviaSegnale()* è in relazione con void tirato(int risultato). InviaSegnale è in realtà uno slot fittizio che collega la pigiatura del pulsante FATTO con il signal tirato(X). Viene fatto ciò perché le Qt non permettono connect tra funzioni con diversa lista di parametri. X è il risultato del lancio della moneta: vale 1 o 0 in quanto sono solo due le possibilità
   Se il valore è 0 vuol dire che è uscita testa e che si può proseguire indisturbati
   Se il valore è 1 vuol dire che è uscita croce e che si deve passare un turno in prigione

La struttura schematizzata del funzionamento di BarraPolizia:



BARRA TRIBUNALE

BarraTribunale è colei che si occupa di far visualizzare graficamente il dado da dover lanciare e il pulsante che segnala di aver capito per quanti turni si è costretti a soggiornare in prigione.
La struttura è molto simile a quella di BarraPolizia, solo che la moneta è rimpiazzata dal dado.

```
public slots:
 void tiraDado();
 void inviaSegnale();

signals:
 void tirato(int);
private:
 int risultato;
 int ripetizione;
 QPushButton *tira;
 QPushButton *fatto;
 QTimer *timer;

public:
 BarraTribunale(QWidget *parent=0);
 ~BarraTribunale();
```

1. Le uniche funzionalità pubbliche sono il costruttore ed il distruttore poiché il resto delle informazioni viene scambiato con PopUp attraverso apposite connect.
   Il costruttore ha solo bisogno di sapere di chi deve diventare figlio BarraTribunale.

2. I dati statici che si preoccupano di gestire i limiti del lancio della moneta sono:
   - *int risultato*: determina la faccia del dado che si vedrà ad ogni istante. Viene ricalcolato per ogni giro del dado. Quando il dado avrà finito di rotolare questo campo segnerà effettivamente il numero di turni che il giocatore dovrà scontare in prigione.
   - *int ripetizione*: è il numero di giri che il dado deve ancora fare prima di fermarsi.

3. Le scritte informative non hanno puntatori, in quanto non hanno bisogno di essere modificate. Sono quindi state definite all'interno del costruttore e suggeriscono al giocatore di cliccare sul dado per poter continuare con il gioco.

4. *QPushButton *tira* è il pulsante su cui è disegnato il dado. Una volta cliccato su di lui si fa partire l'animazione (in realtà si fa partire il timer che quindi si occuperà del resto, vedi .6)

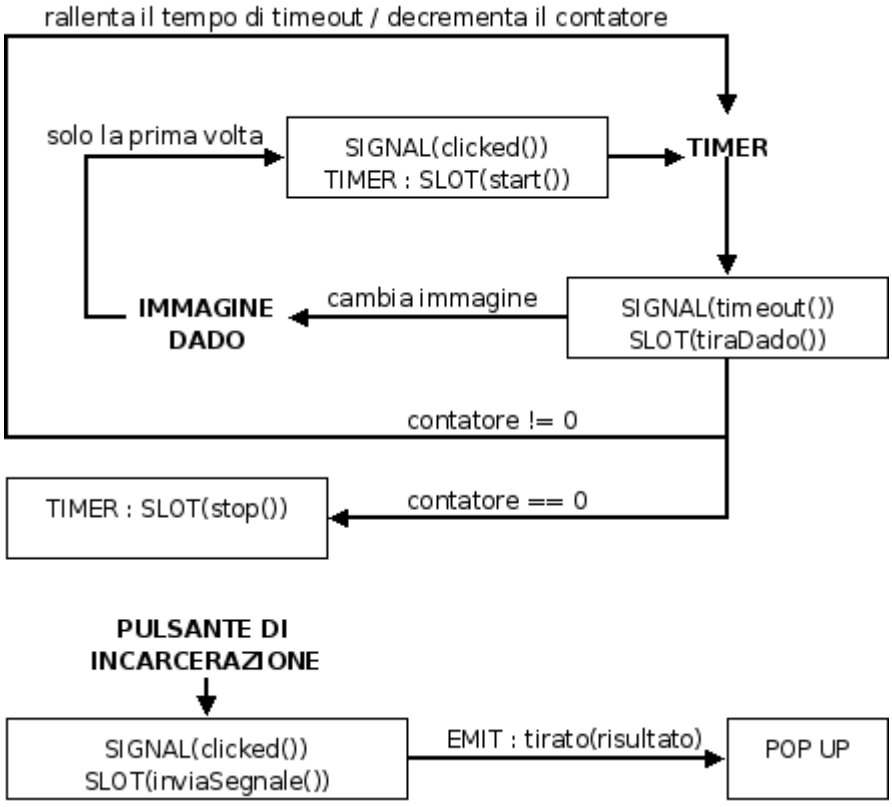
5. *QPushButton *fatto* è il pulsante che dichiara che il giocatore ha capito quale è stato il risultato del lancio del dado. Viene abilitato solo dopo che il dado ha smesso di rotolare.

6. *QTimer *timer* è un semplicissimo timer utilizzato per dare un'apparenza di dinamicità al lancio del dado e per acuire la suspance. È una infima animazione che consiste nel continuare a far vedere una faccia a caso del dado seguita da un altra faccia (differente dalla precedente) ad intervalli sempre più lunghi fin quanto il dado non si ferma. Del timer sono utilizzati solo i suoi timeout per impostare i cambi di immagini. Al timer è connesso lo slot tiraDado
   - void tiraDado() è lo slot che viene invocato ad ogni timeout del timer. Ogni volta si preoccupa di decrementare di una unità ripetizione, di affidare a risultato un nuovo valore compreso nell'intervallo chiuso [1, 6] e differente dal precedente valore, di risettare l'intervallo successivo del timer (allungandolo per creare effetto suspance e dinamismo), di fare il corretto display della giusta faccia del dado (faccia in relazione al valore di risultato). Si preoccupa anche di fermare il timer quando ripetizione ha raggiunto il valore 0, di disabilitare il pulsante che rappresenta il dado (così non si può tentare di ottenere un risultato migliore) e di abilitare il pulsante FATTO.

7. *void inviaSegnale()* è in relazione con void tirato(int risultato). InviaSegnale è in realtà uno slot fittizzio che collega la pigiatura del pulsante FATTO con il signal tirato(X). Viene fatto

ciò perchè le Qt non permettono connect tra funzioni con diversa lista di parametri. X è il risultato del lancio del dado: il suo valore rappresenterà il numero di turni che il giocatore dovrà trascorrere dietro le sbarre.

La struttura schematizzata del funzionamento di BarraTribunale:



MODALITAGIOCO

Modalità gioco deriva da Qwidget e permette all'utente di selezionare il numero di giocatori (tra 2 e 4), di impostare i nomi dei giocatori e di selezionare le modalità di gioco. Il costruttore di ModalitàGioco lascia invocare il costruttore del sotto-oggetto di tipo QWidget standard, costruttore che ha di default il campo parent a 0, questo assegna ad ogni oggetto di tipo ModalitaGioco una finestra separata.
ModalitaGioco è organizzato dal punto di vista grafico per mezzo di un QGridLayout all'interno del quale sono distribuite le varie opzioni disponibili all'utente. ModalitaGioco è inoltre fornito di un metodo paintEvent che disegna l'insegna del monopoli. A causa di questa commistione tra QGridLayout e paintEvent siamo stati costretti a definire una dimensione fissa per la finestra, 850x600 pixel.
Vediamo ora le componenti principali di ModalitaGioco:

```
QLineEdit * inputNomi[4];
QCheckBox * Modalita[4];
QLabel * Warning;
QSpinBox * sb;
```

Le componenti sopraelencate sono campi dati PRIVATI di ogni oggetto ModalitaGioco.
inputNomi è un array di QlineEdit e permettono al giocatore di inserire i nomi dei giocatori. Modalità è un array di QCheckBox, utilizzate dall'utente per selezionare le modalità di gioco (ciascuna QCheckBox è contrassegnata da un titolo che indica la modalità), Warning è una Qlabel che ricorda ai giocatori di mantenere i nomi entro un certo range, sb è una QSpinBox che permette all'utente di selezionare il numero di giocatori.
Naturalmente sono presenti anche altri widget, ma perché non sono elencati nella lista dei campi dati? I campi dati di ModalitaGioco sono quegli oggetti appesi sotto al widget a cui dobbiamo accedere anche fuori da costruttore, in questo caso sono acceduti da Partita dopo l'invocazione dello slot salvaInfo().
Vediamo gli altri widget di cui è munito un oggetto di tipo ModalitaGioco, per prima cosa abbiamo tre QLabel che rappresentano i titolo delle varie sottosezioni di personalizzazione: Numero Giocatori, Nomi Giocatori e Modalita di Gioco. Successivamente c'è anche un array di QLabes che fornisce una didascalia per ciascuna delle 4 modalità di gioco. Per concludere abbiamo ancora due QpushButton gioca e esci, inseriti in un layout orizzontale inserito a sua volta nel QGridLayout dell'intero widget.
ModalitaGioco dichiara la classe Partita come amica e possiede anche un campo PRIVATO puntatore a oggetto della classe Partita. Il costruttore:

```
ModalitaGioco(Partita *);
```

Ha bisogno di un parametro di tipo Partita* con cui crea il corrispondente campo dati, si occupa inoltre di costruire i componenti grafici che abbiamo elencato sopra nell'ordine corretto, definisce inoltre 3 importanti connect:

```
1. connect(sb, SIGNAL(valueChanged(int)), this, SLOT(sistemaGiocatori(int)));
2. connect(gioca, SIGNAL(clicked()), this, SLOT(salvaInfo()));
3. connect(esci, SIGNAL(clicked()), this, SLOT(close()));
```

La prima connette un cambiamento del valore della QSpinBox sb, rappresentato dal signal valueChanger(int), con lo slot pubblico sistemaGiocatori(int) che tra poco illustreremo.
La seconda connette il click del pulsante gioca non lo slot pubblico salvaInfo() e infine l'ultima connette il click del pulsante esci con lo slot ereditato da QWidget close() che chiude la finestra.
Vediamo cosa fanno gli slot:

```
1.  void sistemaGiocatori(int);
2.  void salvaInfo();
```

1. void sistemaGiocatori(int), quando l'utente cambia il numero di giocatori tramite la spinbox sb questa emette un signal portando con se il nuovo valore selezionato, questo slot raccoglie il signal in questione e abilita tante QLineEdit in inputNomi quanti sono i giocatori scelti dall'utente, lasciando disabilitate quelle superflue;
2. void salvaInfo(), viene invocato come detto al click del pulsante gioca, controlla che i nomi dei giocatori non siano troppo lunghi, nel qual caso mette in risalto la QLabel Warning, se tutto è corretto invoca il metodo partita->salvaInfo().

Vogliamo ancora una volta sottolineare che tutti i campi dati e tutti gli oggetti grafici all'interno dell'oggetto di tipo ModalitaInfo hanno lo stesso oggetto come parent, in quest'ottica non occorre ridefinire il distruttore, si veda Distruttori.

PARTITA

Partita è la classe che raccoglie tutti gli oggetti necessari al gioco.
Contiene il seguenti campi dati PRIVATI:

```
1. ModalitaGioco * opzioni;
2. vector<string> nomi;
3. int numGiocatori;
4. bool* modalita;
5. TavoloDaGioco* Tavolo;
```

1. ModalitaGioco* opzioni è il widget che permette all'utente di selezionare il numero di giocatori, i nomi dei giocatori e le modalità di gioco, al click del pulsante gioca di questo widget uno slot invoca il metodo pubblico di Partita salvaInfo() che salva le richieste dell'utente nei prossimi campi dati;

2. vector<string> nomi, è un vettore di stringhe che rappresenta i nomi dei giocatori, inizialmente è un vector vuoto ma viene poi riempito dalla funzione salvaInfo;

3. int numGiocatori, è il numero di Giocatori viene costruito a 4 ma poi viene modificato prima di creare il gioco dal metodo salvaInfo() a seconda delle scelte del giocatore;

4. bool* modalità gioco, è l'array booleano delle modalità di gioco, so questo array si parametrizza la costruzione dell'intero gioco, viene costruito tutto false ma come per nomi e numGiocatori viene poi modificato dalla funzione salvaInfo();

5. TavoloDaGioco* Tavolo; è il tavolo su cui si giocherà la nostra partita;

Passiamo ora ad analizzare dettagliatamente i metodi pubblici offerti dalla classe partita:

```
1. Partita();
2. void salvaInfo();
3. void CreaGioco();
4. ~Partita();
```

1. Partita(), è il costruittore, i veri valori utili per i campi dati vengono dati dalla funzione salvaInfo() tuttavia il costruttore crea gli oggetti, in particolare crea il widget delle modalità di gioco sullo heap, dopodichè crea il vector dei nomi la lo lascia vuoto, crea e inizializza il numero di giocatori a 4, alloca sullo heap l'array booleano delle modalità di dimensione 4 ma lascia indefiniti i sui valori, crea a 0 il puntatore a TavoloDaGioco[22], infine il costruttore fa lo show del widget di selezione delle modalità di gioco;

2. void salvaInfo(), questo metodo viene invocato dallo slot di opzioni salvaInfo() se tutti i campi del form sono stati riempiti correttamente, ha il compito di accedere ai campi privati[23] di opzioni inputNomi e Modalita e salvare i valori in nomi e modalità, oltre che salvare in numGiocatori il numero dei giocatori scelto;

3. *void CreaGioco()*, questo metodo viene invocato alla fine di salvaInfo(). Dati i nomi dei giocatori e le modalità di gioco crea: il vector dei puntatori a GiocatoriLogici, quello dei puntatori a GiocatoriGrafici, il vector dei puntatori a CaselleLogiche, il vector dei puntatori a CaselleGrafiche, TurnoLogico, TurnoGrafico e infine TavoloDaGioco.
   Vogliamo far notare che tutti gli oggetti sono allocati sullo heap, tutti i giocatori (Grafici e

22 Sarà poi la funzione creaGioco() a creare il vero TavoloDaGioco, fino ad allora il puntatore non sarà usato.
23 Per mezzo della dichiarazione di amicizia friend class Partita nella classe ModalitàGioco

Logici), tutte le caselle (Grafiche e Logiche), i Turni (Grafico e Logici) e infine TavoloDaGioco.
È interessante osservare più in dettaglio questo metodo:

- I giocatori sia grafici che logici sono costruiti sequenzialmente per mezzo di un for, che costruisce ogni giocatore logico con nome e indice e poi lo appende in fondo al vector dei GiocatoriLogici e ogni giocatore grafico con le due immagini corrispondenti. Per caricare le immagini utilizziamo due stringhe che memorizzano il percorso generico e poi modifichiamo per mezzo dell'indice del for questo percorso per ottenere le diverse immagini. È importante ricordare che il for in questione scorre fino a numGiocatori, siamo quindi sicuri di rispettare le scelte dell'utente;

- Per il vector delle CaselleLogiche abbiamo parametrizzato la costruzione su un file che contiene nell'ordine che abbiamo deciso di dare alle caselle sul tabellone il nome e i costi rispettivi, per quelle caselle in cui i costi non hanno senso questi hanno valore 0.
  Come visto analizzando la gerarchia delle caselle logiche abbiamo alcune dipendenze, ricordando: ogni oggetto di tipo CNEdPrivata ha bisogno di un puntatore CNEdTassa*, quindi utilizziamo una variabile locale di tipo CNEdTassa* nella costruzione, analogamente per quanto riguarda la dipendenza di Tribunale da oggetti della classe Prigione utilizziamo una variabile temporanea di tipo Prigione* che useremo per costruire oggetti di tipo Tribunale.
  L'utilizzo di un file ci permette di modificare il nome di una casella o il suo costo semplicemente andando ad editare il file in questione, tuttavia questa tecnica non è esente da problemi, nel caso ci fossero problemi con l'apertura del file siamo costretti a rinunciare alla costruzione del gioco e a comunicare un messaggio di errore.
  Naturalmente la costruzione delle caselle dipende direttamente dalle modalità selezionate dal giocatore:

Modalita[0] ~ modalità ipoteca
Modalita[1] ~ modalità privatizzata
Modalita[2] ~ modalità legale

  Se modalita[0]==true allora ogni terreno edificabile deve essere un oggetto della classe CEdIpoteca, ovvero deve potere essere ipotecato e de-ipotecato, altrimenti sono semplicemente oggetti della classe Cedificabile.
  Se modalita[1]==true allora le società non sono oggetti della classe CNEdTassa ma della classe CNEdTassaPrivata ovvero permettono la partecipazione degli utenti privati, analogamente le caselle che rappresentano enti pubblici saranno della classe NEdPrivata, ed ognuna di esse sarà associata ad una specifica società, altrimenti le società sono seplicemente oggetti della classe CNEdTassa e gli enti pubblici semplicemente oggetti di CNEdificabile.
  Se modalità[2]==true le Prigioni i Tribunali e le centrali di Polizia non saranno oggetti della classe CNEdficabile ma rispettivamente di Prigione, Tribunale e Polizia, in questo caso i tribunali saranno costruiti con il puntatore alla rispettiva prigione gestita,
  Il caricamento degli sfondi delle caselle per il vector CaselleGrafiche funziona similmente agli avatar dei giocatori, abbiamo peraltro un unico percorso sia per gli sfondi grandi che per gli sfondi piccoli:

:/immagini/sfondi/sfondoX

  La X sarà sostituita con il numero di indice della casella e poi come abbiamo detto ci penserà lo stesso costruttore di CasellaGrafica a reperire lo sfondo Grande. Ricordiamo inoltre che ogni CasellaGrafica va costruita con un vector di puntatori a GiocatoreGraf

in modo tale che sia in grado di disegnare i giocatori presenti (si veda CasellaGrafica). Si noti inoltre che tutte le CaselleGrafiche sono appese sotto al Widget Tavolo (si veda Distruttori);
Facciamo notare che è presente sempre un ulteriore dipendenza che è quella delle caselle grafiche da un puntatore alla corrispondente casella logica, abbiamo quindi per tutte le caselle logiche non edificabili una variabile temporanea che ne contiene il puntatore, di tipo CBase*, per inserirlo poi nel costruttore della corrispondente casella grafica. Per le caselle logiche edificabili abbiamo invece bisogno di un puntatore a casella edificabile, quindi il puntatore temporaneo sarà di tipo CEdificabile*;

- Al termine della costruzione dei vector di giocatori logici e grafici e delle caselle logiche e grafiche possiamo passare alla costruzione dei gestori di turno, TurnoLogico e TurnoGrafico, TurnoLogico viene costruito con i vector GiocatoriLogici e CaselleLogiche, mentre TurnoGrafico con GiocatoriGrafici e CaselleGrafiche e con parametro parent = Tavolo; inoltre TurnoGrafico raccoglie il parametro booleano modalita[3] che rappresenta la modalità Dado Truccato:

modalita[0] ~ modalità Dado Truccato

  Infatti come abbiamo già visto è TurnoGrafico a costruire il Dado.
  Costruiti TurnoLogico e Turno Grafico si crea il tabellone delle informazioni dei giocatori, InfoGiocatori, costruito con GiocatoriLogici e GiocatoriGrafici e appeso sotto a Tavolo.
- Infine viene costruito un oggetto TavoloDaGioco sullo heap e il suo puntatore sarà salvato in Tavolo, è costruito con il puntatore a TurnoGrafico, il vector CaselleGrafiche e il puntatore all'oggetto di tipo InfoGiocatori.
- Al termine della creazione del gioco viene fatta la show di Tavolo e chiuso lo stream di input.

4. *~Partita()*, il distruttore di Partita invoca esplicitamente la delete su: l'array di booleani allocato sullo heap, la schermata di selezione delle modalità e il Tavolo. La delete di Tavolo causerà la distruzione di tutto il resto (si veda Distruttori).

Come abbiamo visto Partita e ModalitaGioco sono mutuamente dipendenti, Partita allora sullo heap un oggetto della classe ModalitaGioco, oggetto che a sua volta ha bisogno per essere costruito di un puntatore a partita. In questa condizione di mutua dipendenza si rende necessaria una dichiarazione incompleta di classe, le dichiarazioni delle classi ModalitaGioco e Partita si trovano dello stesso file ModalitaGioco.h e la loro disposizione è come segue:

```
class Partita;
class ModalitaGioco{
friend class Partita; //dichiarazione di amicizia di cui abbiamo parlato sopracitato
...
Partita * partita;
...
};
class Partita{
...
ModalitaGioco* opzioni;
...
};
```

MAIN

Illustrata la classe Partita possiamo ora giustificare il nostro main:

```
QApplication app(argc, argv);
Partita partita= Partita();
return app.exec();
```

Come abbiamo visto tutti i campi dati e gli oggetti creati in Partita sono allocati sullo heap quindi possiamo tranquillamente permetterci di lasciare partita sullo stack e utilizzare il normale distruttore che il c++ invoca all'uscita dal blocco (in questo caso all'uscita dal main), uscita che viene però rimandata fino al termine di app.exec();.
Il seguente main non avrebbe in alcun modo cambiato la situazione:

```
QApplication app(argc, argv);
Partita * partita= new Partita();
return app.exec();
delete partita;
```

DISTRUTTORI

Abbiamo rimandato a questa sezione la discussione sul comportamento dei distruttori per quanto riguarda la parte grafica, riassumendo quanto detto precedentemente abbiamo:

- Il distruttore di partita dealloca l'oggetto di tipo ModalitaGioco e il TavoloDaGioco;
- ModalitaGioco utilizza semplicemente il distruttore standard;
- TavoloDaGioco non possiede un proprio distruttore, utilizza quello standard;
- Il distruttore di TurnoGrafico non si sa da chi venga invocato ma chiama la delete su tutti i GiocatoriGrafici e su TurnoLogico;
- il distruttore di TurnoLogico invoca la delete di tutti i giocatori e le caselle logiche (questo non ci interessa per il momento);
- Il distruttore di CasellaGrafica dealloca le QPixmap di sfondo e sfondoGrande;
- In TurnoGrafico PopUp non viene mai distrutto;
- Il distruttore di PopUp è quello standard.

A questo punto sorgono alcune domande: chi invoca il distruttore di TurnoGrafico, di InfoGiocatori e delle CaselleGrafiche? Chi invoca il distruttore dei widget contenuti in ModalitaGioco e PopUp? Come mai non rimane garbage sullo heap causato dalla non distruzione dei vari PopUp?
Citiamo dalla documentazione delle Qt:

QWidget::~QWidget ()

Destroys the widget.

All this widget's children are deleted first. The application exits if this widget is the main widget.

Ecco spiegato chi distrugge TurnoGrafico, InfoGiocatori e tutte le caselle Grafiche. Come detto in Partita parlando della funzione creaGioco() tutti questi oggetti sono costruiti come figli di TavoloDaGioco, alla sua distruzione anche loro vengono deallocati.
Analogamente spiega come vengono distrutti tutti i componenti di PopUp e TurnoGrafico, occorre semplicemente prestare attenzione ad appendere tutti i componenti come figli del widget superiore. Questo però non spiega come non si formi garbage sullo heap per le mancate rimozioni dei PopUps.

Citiamo nuovamente dalla documentazione delle Qt:

bool QWidget::close () `[slot]`

Closes this widget. Returns true if the widget was closed; otherwise returns false.

First it sends the widget a QCloseEvent. The widget is hidden if it accepts the close event. If it ignores the event, nothing happens. The default implementation of QWidget::closeEvent() accepts the close event.

If the widget has the Qt::WA_DeleteOnClose flag, the widget is also deleted.

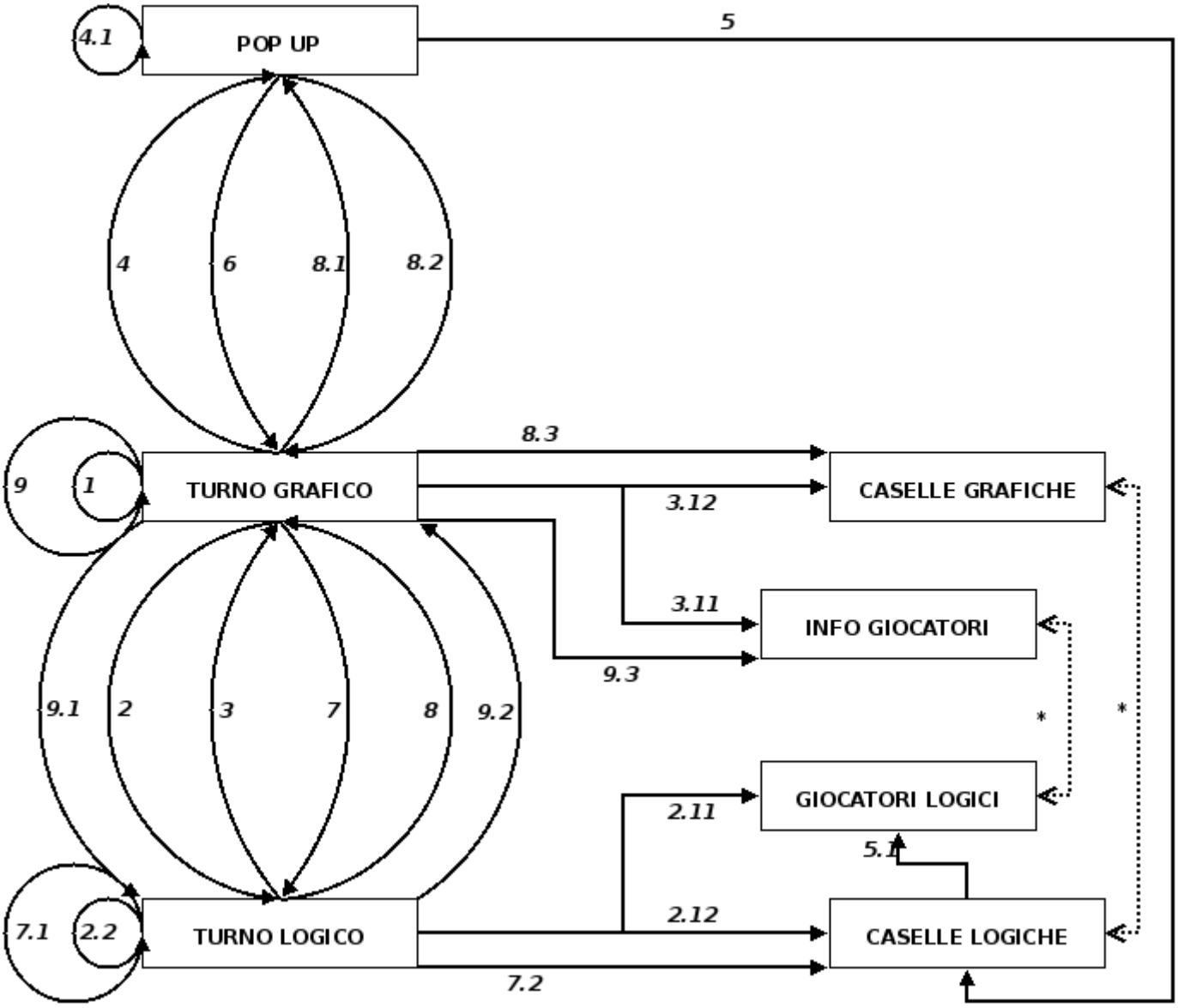
La parte interessante è quella segnata in grassetto.
Ecco che per risparmiarci di effettuare la delete ad ogni nuovo PopUp in TurnoGrafico ci siamo limitati a settare per ogni PopUp p la proprietà **Qt::WA_DeleteOnClose** tramite:

```
p->setAttribute(Qt::WA_DeleteOnClose);
```

Ecco che ogniqualvolta l'utente chiude il PopUp questo viene anche eliminato, senza creare garbage sullo heap.
Naturalmente non fidandoci poi troppo della documentazione abbiamo provato manualmente, posizionando opportune stampe dei distruttori e abbiamo così confermato quanto detto.
Resta da spiegare soltanto come mai è necessario invocare manualmente le delete per i giocatori grafici, questo è presto fatto, a differenza di tutti gli altri componenti della grafica i giocatori grafici non derivano da widget ma sono un semplice contenitore di QPixmap, ecco spiegato perché non si possono appendere come figli di un altro Widget e vanno delallocati manualmente (in TurnoGrafico).

FLUSSO DI GIOCO

Proponiamo ora uno schema semplicistico del flusso di gioco a partita avviata.
Dato che il flusso di gioco viene governato principalmente da connect, per una trattazione più specifica si rimanda alla sezione che illustra le varie classi con i vari signal, slot e connect.
Poiché era impensabile fornire uno schema esaustivo di tutte le diverse classi con la loro interazione finalizzata al flusso di gioco, nel seguente schema sono state riportate le classi principali che interagiscono tra di loro per produrre il flusso di gioco. Per capire l'effettiva relazione tra queste classi e per comprendere più a fondo in che modo le connect possano interagire tra loro e fornire il flusso di dati, si rimanda allo schema che illustra la gerarchia di classi.



Il flusso di gioco è dato in sequenza dall'arco etichettato 1 fino all'arco etichettato 9.3 con le seguenti specifiche ed eccezioni.

- **1 :** si può considerare che TurnoGrafico sia l'inizio del flusso di gioco nel momento in cui viene premuto il pulsante LANCIA DADO. Si ottiene il risultato del lancio e si abilita il pulsante MUOVI.
- **2 :** Viene premuto il pulsante MUOVI ed il risultato del lancio del dado viene salvato in TurnoLogico a cui passa il flusso di gioco.

2.11 : Viene aggiornato lo spostamento sui giocatori logici e viene eseguito il test per il passaggio per la Camera del Commercio

2.12 : Viene aggiornato lo spostamento nelle caselle logiche

2.2 : Viene eseguito il test per vedere se è stato superato il massimo numero di giri disponibili (cosa che scatenerebbe la vittoria di qualcuno)

- **3 :** il controllo ritorna a TurnoGrafico che riceve l'oggetto Andata appena calcolato da TurnoLogico tramite casella->sceltaOpzione(giocatore)

  3.11 : prima di utilizzare Andata viene aggiornato InfoGiocatori poiché nel caso si fosse passati per il via la quantità di ghiande visualizzata sarebbe minore di quella effettivamente posseduta: un'inconsistenza di cui il giocatore potrebbe non accorgersi e spingerlo ad un gioco più conservativo del dovuto

  3.12 : vengono aggiornate anche le caselle grafiche per visualizzare lo spostamento della pedina dalla casella vecchia a quella nuova
- **4 :** viene creato il PopUp grazie all'oggetto Andata appena calcolato ed il flusso di gioco passa a lui in attesa che uno dei suoi pulsanti venga pigiato

  4.1 : l'utente compie le sue scelte premendo uno dei pulsanti che son stati resi disponibili
- **5 :** viene scatenata l'azione di elaborazione dati ed aggiornamento a livello logico tramite casella->eseguiAzione(Ritorno) e quindi il flusso di gioco passa nell'area logica determinata dalle caselle logiche e dai giocatori logici

  5.1 : in caso l'elaborazione dei dati può modificare i dati presenti nei giocatori logici
- **6 :** finita l'elaborazione a livello logico il controllo ritorna a PopUp che chiudendosi cede immediatamente il flusso di gioco a TurnoGrafico, dato che PopUp era appeso ad esso, segnalandogli tra l'altro di aver eseguito delle modifiche non indifferenti a livello logico
- **7 :** TurnoGrafico ripassa il controllo a TurnoLogico per scoprire se l'azione dell'utente ha scatenato fenomeni quali morte o vittoria di qualcuno

  7.1 : TurnoLogico elabora le informazioni per capire se qualcuno è effettivamente morto o se qualcuno ha effettivamente vinto

  7.2 : nel caso qualcuno sia morto aggiorna la situazione nelle caselle logiche, in pratica si toglie la presenza del giocatore nella cella in cui era
- **8 :** TurnoLogico ripassa il flusso di gioco a TurnoGrafico passandogli inoltre un informazione incapsulata in un oggetto di tipo Andata. L'informazione che viene passata a TurnoGrafico riguarda quello che è appena stato svolto da TurnoLogico: informa se qualcuno è morto, qualcuno ha vinto o se non è successo nulla delle due cose appena citate
- **8.1 :** TurnoGrafico ora decide il dafarsi con l'informazione che ha appena ricevuto. Se qualcuno è morto o qualcuno ha vinto crea un nuovo PopUp e passa a lui il flusso di gioco

  8.2 : PopUp visualizza l'informazione che TurnoGrafico gli ha passato ed aspetta che venga cliccato il pulsante di conferma che farà ritornare il controllo a TurnoGrafico (se qualcuno è morto) o che terminerà il programma (se qualcuno ha vinto)
- **9 :** TurnoGrafico ha ora modo di ricevere la pigiatura del pulsante PASSA TURNO per abilitare il pulsante TIRA DADO

  9.1 : una volta cliccato PASSA TURNO il controllo del gioco passa a TurnoLogico per il tempo necessario a calcolare l'indice del giocatore successivo a cui spetta il turno di gioco

  9.2 : il cotrollo ripassa a TurnoGrafico

  9.3 : TurnoGrafico passa il controllo a InfoGiocatori per il tempo necessario ad aggiornare l'evidenziazione del giocatore di turno (che è appena cambiato)
- **10 :** ora che il pulsante TIRA DADO è stato abilitato è possibili ricominciare dal punto **1**
- *** :** l'asterisco informa che tra le classi con le frecce tratteggiate c'è uno scambio di informazioni, soprattutto durate il periodo di aggiornamento delle informazioni

TRANSLATE.CPP e GIOCO.QRC

TRANSLATE.CPP

Son state definite due funzioni generali che operano delle trasformazioni per facilitare l'esecuzione e la leggibilità del programma.

1. *QString num2string(int n)*, trasforma un numero intero in una sequenza di caratteri che corrisponde alla visualizzazione in cifre arabe di quel numero. Questa funzione viene utilizzata per inserire dei numeri nel mezzo di stringhe di caratteri che compaiono per inter-relazionarsi con l'utente. Viene utilizzata nel seguente modo: “stringa di caratteri” + num2string(NUM) + “altra stringa di caratteri”. Se si fosse inserito l'intero, quell'intero sarebbe stato interpretato come valore di un carattere da inserire e non come un numero di cui visualizzare le cifre.
   Se i numeri devono comparire in stringhe che contengono solo quel numero, si possono usare i metodi che le Qt mettono a disposizione per far visualizzare un intero in una Qstring.

2. *QString string2QString(string s)*, questa funzione viene utilizzata per non vincolare la parte logica alla parte grafica. In questo modo nella parte logica si possono salvare nomi e parole in string e poi farne il display nella parte grafica semplicemente richiamando questa funzione. Questo è stato fatto anche perchè le string non son compatibili al 100% con le QString: quindi inserire in una QString (utilizzata in maniera standard per far visualizzare parole e frasi nella parte grafica) una semplice string poteva causare problemi. Non era accettabile dichiarare QString nella parte logica poiché classe appartenente alle Qt, che quindi avrebbe legato la parte logica alle librerie grafiche.

GIOCO.QRC

Personalizzare il progetto con delle immagini voleva dire doverle tenere nella stessa directory dei file e inoltre un eventuale spostamento del file eseguibile avrebbe comportato l'impossibilità di rendere le immagini all'interno del gioco.
Abbiamo ritenuto questo comportamento inaccettabile per un gioco così di piccolo calibro. Si è deciso di adottare un file di tipo .qrc che contenesse tutti i percorsi delle immagini da usare secondo la sua sintassi:

```
<RCC>
    <qresource prefix="/" >
        <file>immagini/dado/faccia1.gif</file>
    </qresource>
</RCC>
```

Questo file ci permette di caricare automaticamente al momento della compilazione tutte le immagini all'interno dell'eseguibile. Questo fa sì che sia possibile specificare nel codice qualsiasi tipo di percorso per l'immagine ed inoltre uno spostamento dell'eseguibile rispetto alla cartella di compilazione non causa alcun tipo di problema.
L'unico prezzo da pagare è un eseguibile un pochino più “pesante”.

INFORMAZIONI DI GIOCO

CASELLE EDIFICABILI:

Nome Casella	Costo Terreno	Costo Albergo	Pedaggio Terreno	Pedaggio Albergo
Via Bettola	115	90	55	85
Via Ying e Yang	300	250	155	235
Via Puffosa	145	100	80	95
Via Palafitta	200	150	95	130
Via Cascate	330	250	155	235
Via Palombaro	220	160	110	160
Via Eden	400	300	180	210
Via del Campo	150	120	80	105
Via Tendopoli	100	80	50	75
Via Maya	230	185	110	170

CASELLE TASSABILI

Nome Casella	Costo Pedaggio	Costo Quota
Società Acqua Potabile	50	100
Società Elettrica	75	150
Ferrovia	100	175

AMMENDA PRIGIONE = 75

GHIANDE PER GIOCATORE = 700

BONUS PASSAGGIO DAL VIA = 200

NUMERO MASSIMO DI GIRI = 15