Simone Gentili

1977848
Teledidattica.

progetto: J-UNO

Le carte.

Dall'analisi delle carte del gioco **Uno** sorge evidente la presenza di differenti attributi che contraddistinguno i diffenti tipi di carte. Il diagramma UML in figura mostra le relazioni che intercorrono tra le diverse entità software che prendono parte nella realizzazione di un oggetto *Card*. La classe *Card,* che implementa l'interfaccia *InterfaceCard*, definisce il modello costitutivo di una carta all'interno del programma. Gli attributi di una carta del gioco **Uno** sono i seguenti:

- Attributo "Action": determina la presenza o meno di un'azione che la carta può esibire.

- Attibuto "Color": determina il colore distintivo della carta.

- Attributo "Value": determina il valore numerico della carta.

Ciascuno di questi attibuti può essere o meno presente all'interno di un oggetto della classe "Card" (nel caso in cui non sia presente un attributo, viene inserito il valore *null*).

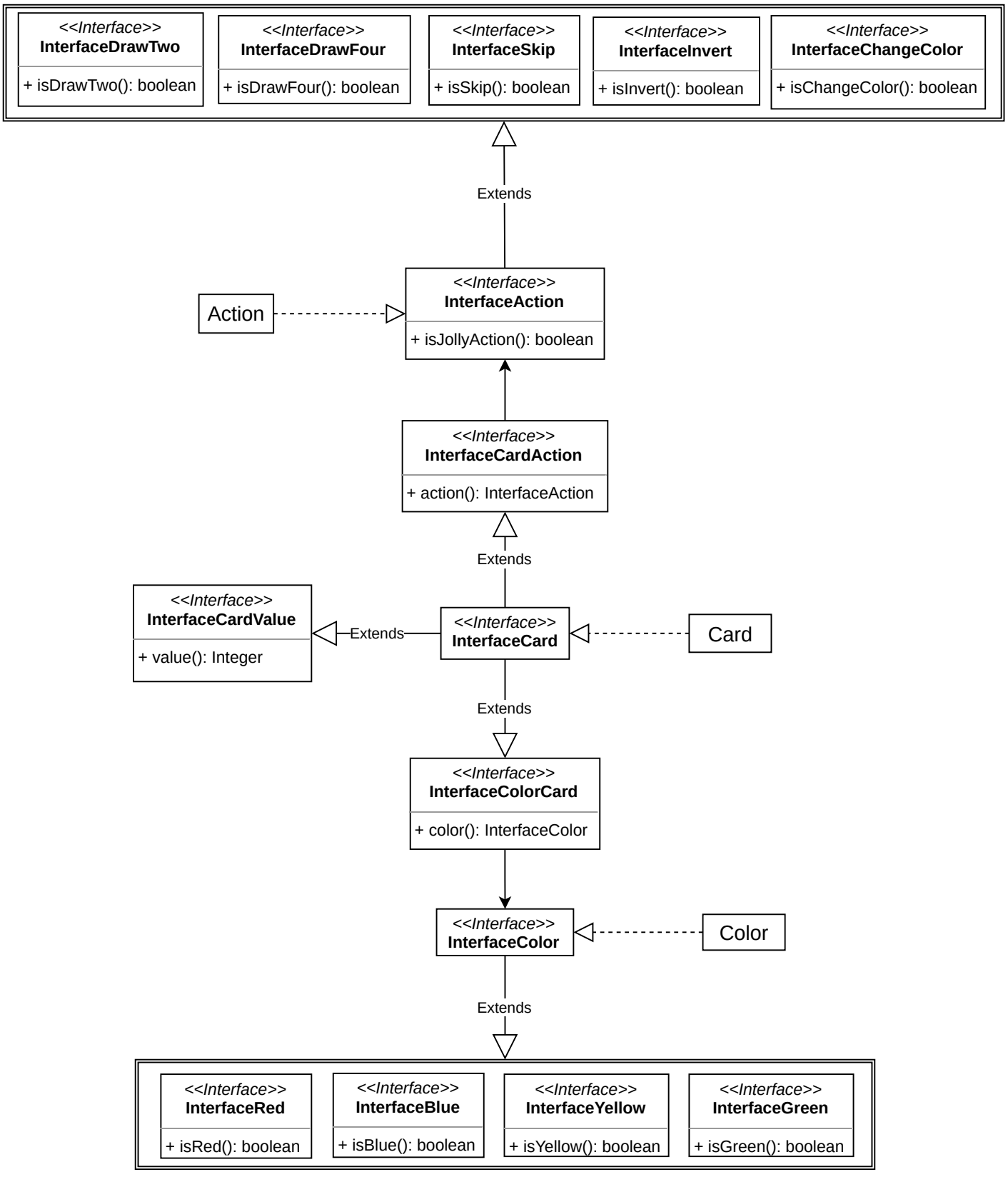
L'attibuto "Color" è realizzato dall'interfaccia *InterfaceColor* e dalla classe enumerativa *Color* che la implementa. Gli oggetti presenti all'interno di quest'ultima sono: RED, BLUE, GREEN e YELLOW, uno per ciascun colore delle carte del gioco **Uno**.

L'attributo "Action" è realizzato dall'interfaccia *InterfaceAction* e dalla classe enumerativa *Action* che la implementa. Gli oggetti presenti all'interno di quest'ultima sono: DRAW_TO, DRAW_FOUR, SKIP, INVERT e CHANGE_COLOR, uno per ciascuna azione delle carte del gioco **Uno**. L'interfaccia InterfaceAction definisce inoltre un metodo: *isJolly(); esso* permette di distinguere le azioni "Jolly" dalle azioni "Non-Jolly". Le azioni "Jolly" sono definite dagli oggetti DRAW_FOUR e CHANGE_COLOR, mentre le azioni "Non-Jolly" sono definite dagli oggetti DRAW_TWO, SKIP e INVERT della classe enumerativa *Action*.

L'attributo "Value" è realizzato mediante l'utilizzo della classe *Integer* della libreria standard di **Java (java.lang)**.

Nota sulla relazione delle carte con gli altri componenti.

Le classi concrete che utilizzano le carte per poter operare sono relazionate esclusivamente con l'interfaccia *InterfaceCard* (ad eccezione della classe *DeckFactory* presente all'interno del package juno.model.deck*,* che utilizza anche la classe concreta *Card*). Tutte le altre classi concrete (e le rispettive astrazioni) che utilizzano in qualche modo le carte a tempo di esecuzione, sono generiche sul tipo *T*. Questo implica che, anche nel caso in cui si voglia realizzare un nuovo tipo di interfaccia per le carte (ad esempio un'interfaccia che prevede un quarto attributo), si potrà riutilizzare il codice delle classi generiche, in quanto del tutto svincolate dal contratto con l'interfaccia *InterfaceCard,* ma non quello delle classi che, invece, l'hanno stretto per ovvie ragioni.


<<Interface>>
InterfaceDrawTwo
+ isDrawTwo(): boolean
<<Interface>>
InterfaceDrawFour
+ isDrawFour(): boolean
<<Interface>>
InterfaceSkip
+ isSkip(): boolean
<<Interface>>
InterfaceInvert
+ isInvert(): boolean
<<Interface>>
InterfaceChangeColor
+ isChangeColor(): boolean
Extends
Action
<<Interface>>
InterfaceAction
+ isJollyAction(): boolean
<<Interface>>
InterfaceCardAction
+ action(): InterfaceAction
Extends
<<Interface>>
InterfaceCardValue
+ value(): Integer
Extends
<<Interface>>
InterfaceCard
Card
Extends
<<Interface>>
InterfaceColorCard
+ color(): InterfaceColor
<<Interface>>
InterfaceColor
Color
Extends
<<Interface>>
InterfaceRed
+ isRed(): boolean
<<Interface>>
InterfaceBlue
+ isBlue(): boolean
<<Interface>>
InterfaceYellow
+ isYellow(): boolean
<<Interface>>
InterfaceGreen
+ isGreen(): boolean

Il Deck.

Il deck, ovvero il mazzo da cui pescare, è stato realizzato dalla classe Deck<T>, che permette mediante il metodo implementato dell'interfaccia InterfaceDeck<T>, di restituire la carta in cima al mazzo. Quando una carta viene pescata, la classe Deck<T> aggiorna tutti i suoi osservatori. Quando il deck sta per terminare (ovvero quando il numero delle carte nel mazzo è inferiore o uguale a 5), il componente DeckFiller<T>, che si occupa di riempire il mazzo utilizzando le carte contenute nella pila degli scarti, viene utilizzato invocando il suo metodo che implementa dall'interfaccia InterfaceDeckFiller<T>, ovvero fill(List<T>). Mediante il meccanismo precedente è stato possibile garantire il proseguimento delle partite durante l'esaurimento del mazzo. Il mescolatore del mazzo è stato realizzato mediante la classe Mixer<T>, che implementa l'interfaccia InterfaceMixer<T>. Quest'ultimo componente in vero è una classe capace di mescolare qualsiasi oggetto che implementa l'interfaccia List<T> del package java.lang.

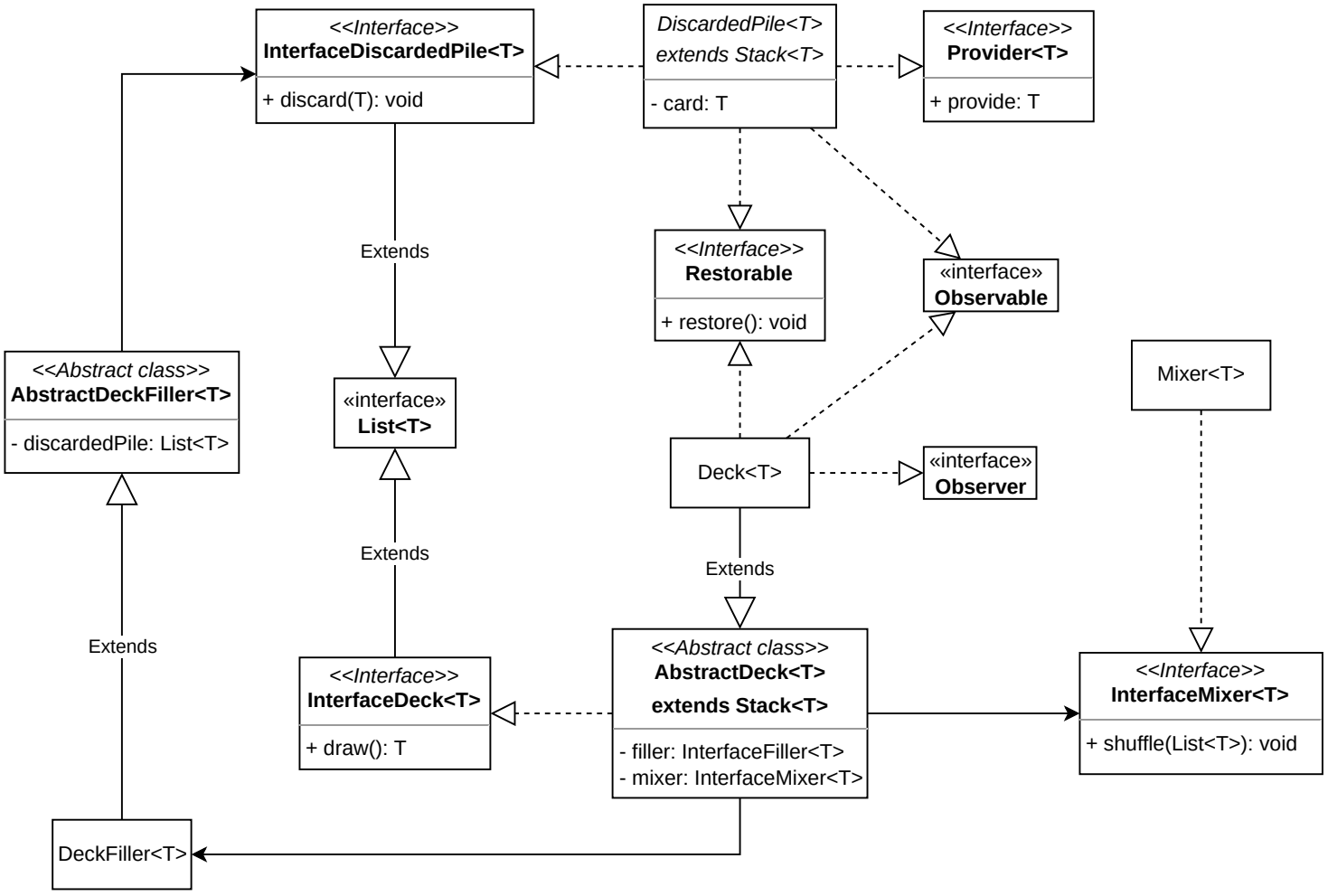
La pila degli scarti.

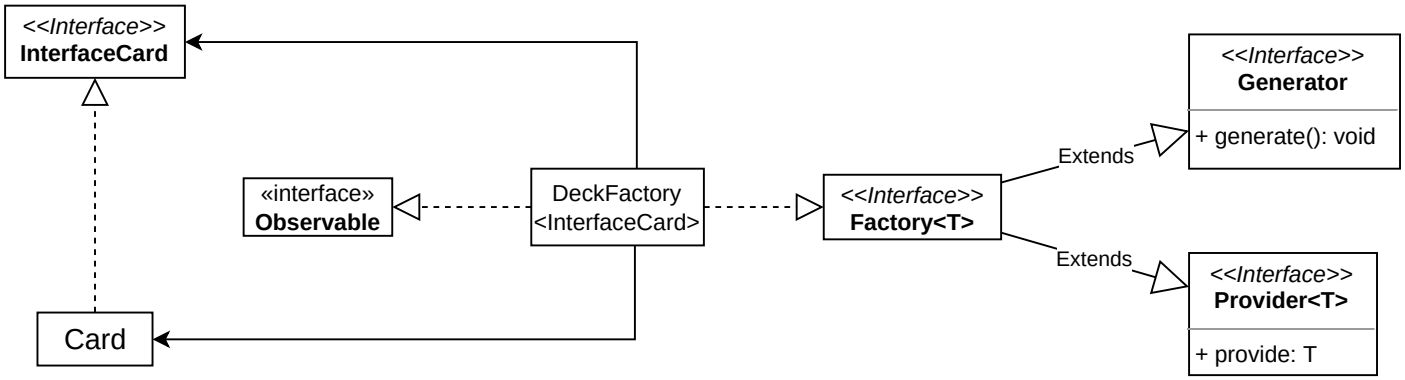
La pila degli scarti, realizzata dalla classe concreta DiscardedPile<T>, che implementa l'interfaccia InterfaceDiscardedPile<T>, permette di scartare un oggetto generico e di inserirlo in cima alla pila. Quando una carta viene scartata, oltre ad essere inserita nella cima della pila, viene anche incapsulata all'interno di un campo della classe stessa e tutti gli osservatori vengono aggiornati. Gli osservatori potranno dunque conoscere qual'è stata la carta scartata invocando il metodo provide() dell'interfaccia Provider<T>, che la DiscardedPile<T> implementa.

Il deck e la pila degli scarti implementano l'interfaccia List<T> ed estendono la classe Stack<T> del package java.lang per massimizzare il riutilizzo del codice pre esistente.

Il produttore del mazzo.

Il produttore del mazzo o DeckFactory assolve il problema della realizzazione del deck. In particolare si occupa di inserire all'interno della classe Deck<T> le carte indispensabili per poter effettuare una partita. L'interfaccia Factory<T> è composta da due sotto interfaccie: Provider<T> e Generator. Un Generator è un qualsiasi oggetto in grado di generare oggetti, quindi di instanziarli ed successivamente incapsularli. Un Provider<T> è un qualsiasi oggetto che fornisce un oggetto generico, che nel caso della classe DeckFactory<T> si tratta di un oggetto List<InterfaceCard>. La separazione dei compiti (generazione e fornitura) è stata scelta quando è apparso un problema nell'applicazione del patter Observer/Observable. Infatti se una factory aggiornasse i suoi osservatori nello stesso metodo nella quale produce e ritorna gli oggetti e, se un Observer invocasse il metodo per ricevere gli oggetti da tale factory, avverrebbe un StackOverFlowError derivato
dalla perpetua e reciproca chiamata tra l'osservatore e l'osservabile. Quando la generazione e la fornitura vengono separati, questo errore sparisce: l'osservatore viene notificato quando l'oggetto è pronto (l'updateAll() si trova all'interno del metodo generate() tipicamente) e preleva quest'ultimo invocando il metodo provide().


<<Interface>>
InterfaceDiscardedPile<T>
+ discard(T): void
DiscardedPile<T>
extends Stack<T>
- card: T
<<Interface>>
Provider<T>
+ provide: T
Extends
<<Interface>>
Restorable
+ restore(): void
«interface»
Observable
<<Abstract class>>
AbstractDeckFiller<T>
- discardedPile: List<T>
«interface»
List<T>
Deck<T>
«interface»
Observer
Mixer<T>
Extends
Extends
Extends
Extends
<<Interface>>
InterfaceDeck<T>
+ draw(): T
<<Abstract class>>
AbstractDeck<T>
extends Stack<T>
- filler: InterfaceFiller<T>
- mixer: InterfaceMixer<T>
<<Interface>>
InterfaceMixer<T>
+ shuffle(List<T>): void
DeckFiller<T>


<<Interface>>
InterfaceCard
<<Interface>>
Generator
+ generate(): void
«interface»
Observable
DeckFactory
<InterfaceCard>
<<Interface>>
Factory<T>
Extends
Extends
<<Interface>>
Provider<T>
+ provide: T
Card

Controllore di compatibilità.

Il controllore di compatibilità, ovvero il CompatibilityChecker, permette di rispondere alla domanda: è compatibile questa carta con la carta in cima alla pila degli scarti ?. Per poter rispondere a tale domanda il controllore di compatibilità è vincolato a conoscere l'interfaccia della carta (InterfaceCard in questo caso). Vengono inoltre utilizzate le classi ActualColorManager, che fornisce il colore attualmente dominante all'interno della partita e, ovviamente, la classe DiscardedPile<T> o pila degli scarti

.

Attivatore degli effetti delle carte.

Il CardEffectActivator permette mediante il metodo activate(T) implementato dall'interfaccia InterfaceCardEffectActivator di attivare l'effetto della carta specificata. Anch'esso, come il controllore di compatibilità, deve operare conoscendo l'interfaccia delle carte. L'attivatore degli effetti utilizza le classi a cui è connesso (visibili nel diagramma UML associato) per poter svolgere le sue mansioni; tra queste vi sono:

- TurnJumper: muove il turno verso il giocatore successivo. Simile al TurnMover, ma non osservabile.

- Inverter: inverte l'ordine dei turni.

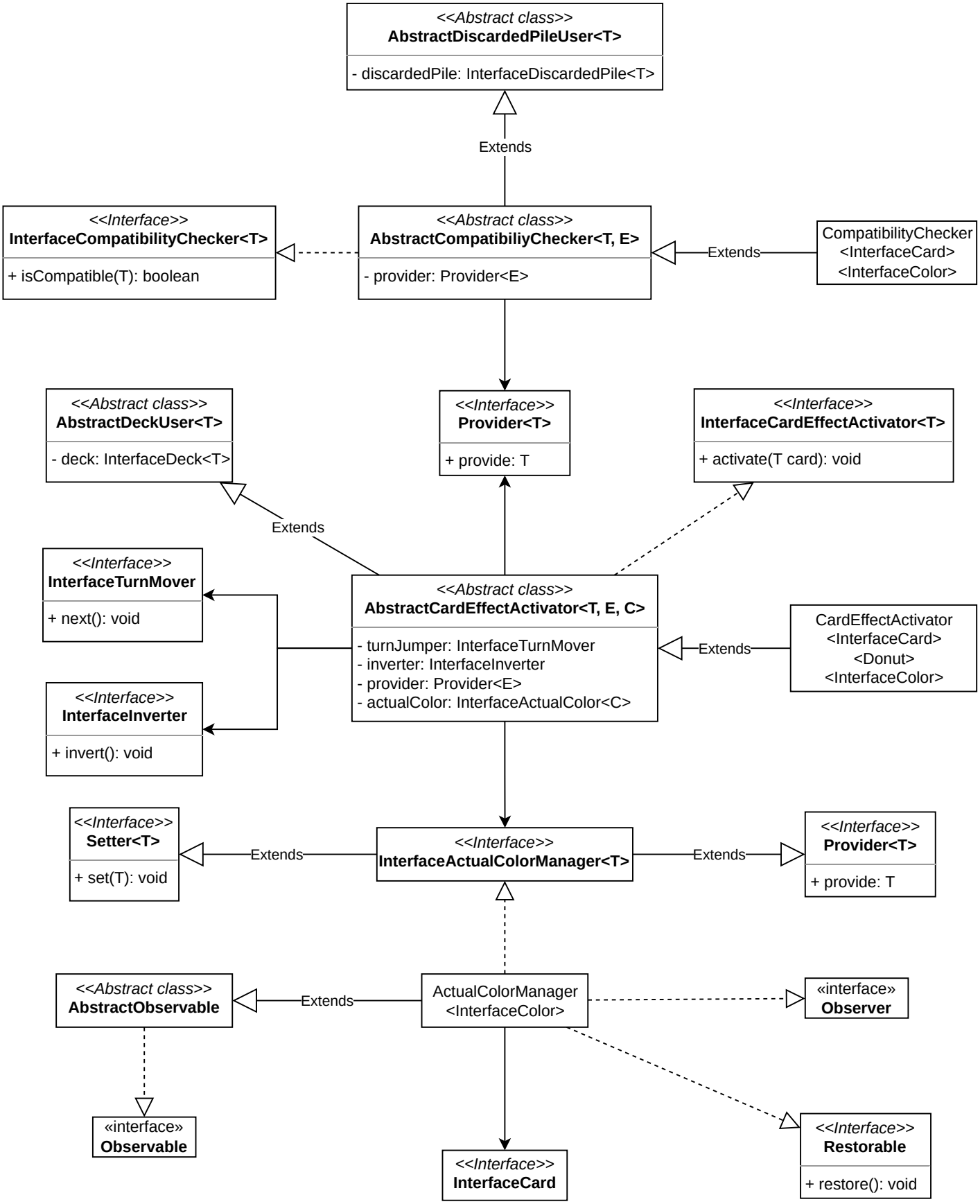
- ActualColorManager: permette di impostare un colore quando viene scartata una carta jolly.

Gli effetti delle carte che fanno pescare il giocatore successivo vengono realizzati semplicemente utilizzando la classe Deck<T> (per ottenere le carte) e un Provider<T> che fornisce i giocatori, da quale è possibile quindi ottenere il giocatore successivo a quello attuale.

Le classi TurnJumper e Inverter fanno parte del package juno.model.subjects e vengono discusse ulteriormente nella pagina dedicata.

Gestore del colore.

Il gestore del colore (ActualColorManager) si occupa di mantenere, fornire ed impostare il colore attuale durante il corso della partita. Per colore attuale s'intende il colore della carta attualmente sulla cima della pila degli scarti. Naturalmente le carte azione e numerate sono composte da colori ma, tuttavia, la realizzazione di questo componente si è resa comunque necessaria dal momento in cui alcuni effetti di carte quali pesca quattro e cambia colore permettono di modificare il colore attualmente dominante. Per non implementare un'interfaccia per l'impostazione del colore della carte, è stato scelto questo approccio: l'idea diventa decisamente più elegante quando si pensa al fatto che, introducento un'interfaccia per l'impostazione del colore al livello di InterfaceCard, anche le carte numerate ed azione non jolly avvrebbero avuto tale possibilità. L'ActualColorManager inoltre è osservatore ed osservabile: osserva la classe DiscardedPile<T> aggiornando automaticamente il colore ed è osservata dal componente della view che visualizza il colore attuale.


<<Abstract class>>
AbstractDiscardedPileUser<T>
- discardedPile: InterfaceDiscardedPile<T>
Extends
<<Interface>>
InterfaceCompatibilityChecker<T>
+ isCompatible(T): boolean
<<Abstract class>>
AbstractCompatibiliyChecker<T, E>
- provider: Provider<E>
Extends
CompatibilityChecker
<InterfaceCard>
<InterfaceColor>
<<Abstract class>>
AbstractDeckUser<T>
- deck: InterfaceDeck<T>
<<Interface>>
Provider<T>
+ provide: T
<<Interface>>
InterfaceCardEffectActivator<T>
+ activate(T card): void
Extends
<<Interface>>
InterfaceTurnMover
+ next(): void
<<Abstract class>>
AbstractCardEffectActivator<T, E, C>
- turnJumper: InterfaceTurnMover
- inverter: InterfaceInverter
- provider: Provider<E>
- actualColor: InterfaceActualColor<C>
Extends
CardEffectActivator
<InterfaceCard>
<Donut>
<InterfaceColor>
<<Interface>>
InterfaceInverter
+ invert(): void
<<Interface>>
Setter<T>
+ set(T): void
Extends
<<Interface>>
InterfaceActualColorManager<T>
Extends
<<Interface>>
Provider<T>
+ provide: T
<<Abstract class>>
AbstractObservable
Extends
ActualColorManager
<InterfaceColor>
«interface»
Observer
«interface»
Observable
<<Interface>>
InterfaceCard
<<Interface>>
Restorable
+ restore(): void

Configuratore (con reflection).

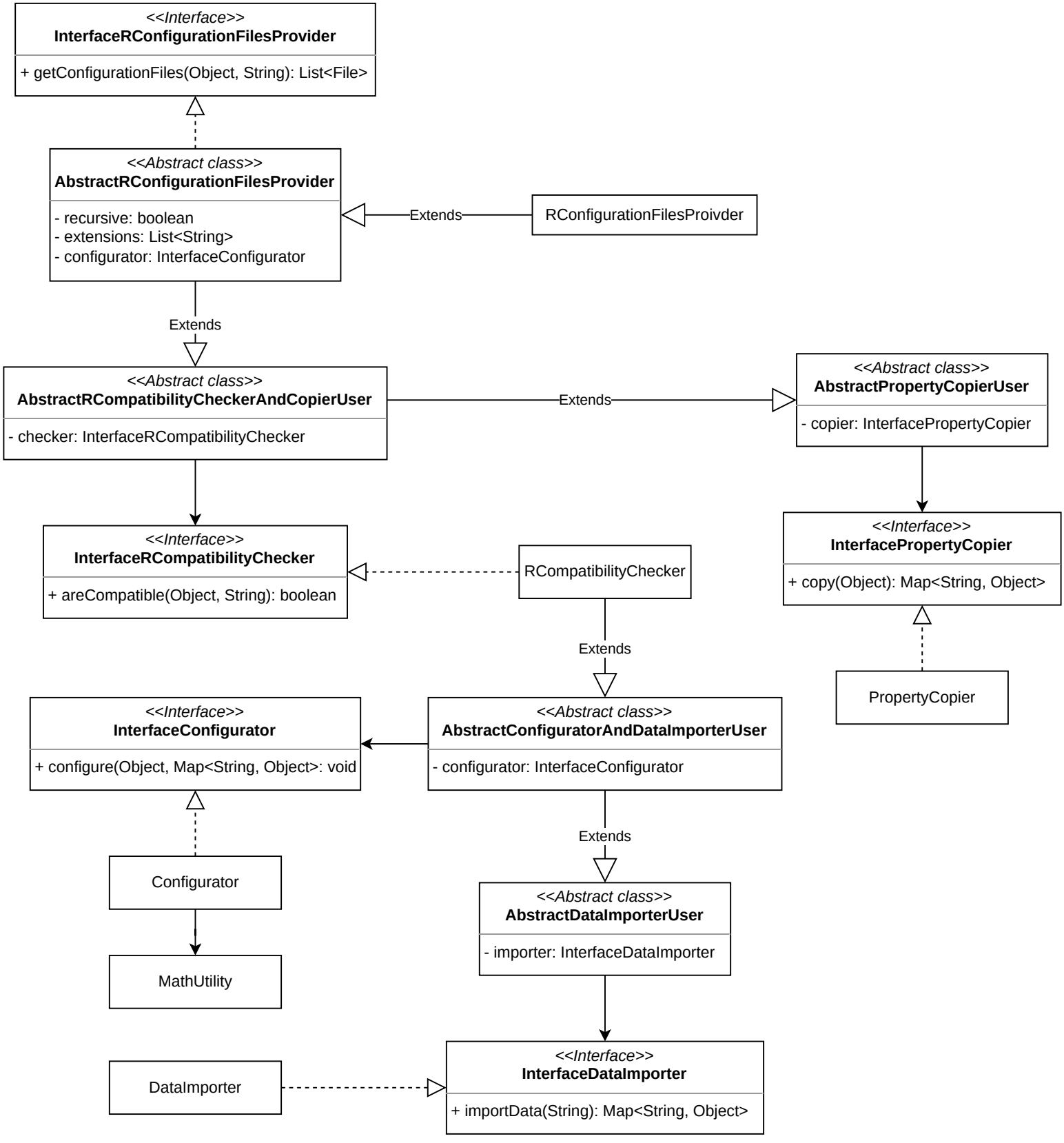
Il configuratore permette di configurare una qualsiasi classe del programma mediante l'utilizzo di un file dal quale è possibile estrarre una mappa. L'estrazione della mappa contenuta all'interno di un file viene eseguita da un *InterfaceDataImporter* che, nel diagramma UML in figura, è un'interfaccia implementata dalla classe *JSONDataImporter*[1].

La configurazione di un *Object* avviene mediante l'utilizzo del componente "Configurator" che nella figura di seguito è rappresentato dall'interfaccia *InterfaceConfigurator* e dalla classe concreta *Configurator*. Quest'ultima classe implementa il metodo configure() che, prendendo in input un *Object* ed un oggetto *Map<String, Object>*, configura l'oggetto con la mappa. Affinchè la configurazione di un *Object* termini con esisto positivo, ciascuna chiave della mappa dev'essere il nome di un differente campo dell'oggetto, mentre i valori della mappa associati a tali chiavi rappresentano i valori da inserire nei corrispondenti campi. Per permettere al configuratore di raggiungere il suo scopo è stato utilizzato il **package java.lang.reflect**. Quest'ultimo ha permesso di modificare l'accesso ai campi da modificare[2], per permette l'inseriemento dei nuovi valori. Durante l'esecuzione del metodo implementato configure() si possono verificare diverse eccezioni. La classe *MathUtility* è stata realizzata per contenere metodi di utilità utilizzati all'interno del metodo *configure()* della classe *Configurator*.

Il componente di configurazione è stato messo in funzione di un'altro componente: il controllore di compatibilità. Nel diagramma è possibile osservare che tale componente, nella sua interezza, è stato realizzato medinate l'utilizzo di un'interfaccia, la *InterfaceCompatibilityChecker*, due classi astratte, la *AbstractCompatibilityChecker* dal package *data.io.input* e la *AbstractCompatibilityChecker* dal package *data.io.input.reflect* ed infine la classe concreta *CompatibilityChecker che implementa l'interfaccia sopra citata*. Questo componente si occupa di rispondere alla domanda "è compatibile questo file con quest'oggetto ?" e per farlo si serve di un configuratore, un copiatore ed un importatore. Se durante la configurazione di un'oggetto con un certo file (da cui viene estratta la mappa mediante l'ausilio di un InterfaceDataImporter) avviene il lancio di un'eccezione, il controllore di compatibilità restituisce il valore booleano *false*, ad indicarne l'incompatibilità. Viceversa se durante il processo di configurazione non viene lanciata alcuna eccezione, il file in esame viene considerato compatibile.

Il fornitore di files di configurazione (nel diagramma UML appare come *ConfigurationFilesProvider*) implementa l'interfaccia *InterfaceConfigurationFilesProvider* ed estende la classe astratta *AbstractConfigurationFiles*. Esso si occupa di fonirne, a partite da un percorso ed un *Object*, tutti i file di configurazione per l'oggetto specificato. Utilizzando il controllore di compatibilità stabilisce se un file e compatibile e, se lo è, lo aggiunge in una lista. Una volta controllati tutti i file contenuti all'interno del percorso specificato, restituisce la lista. Tuttavia prima che il processo di fornitura dei files di configurazione venga eseguito, un'altra operazione viene effettuata. Tale operazione è la copiatura dei dati dell'oggetto mediante l'utilizzo di un "copiatore di proprietà" (nel diagramma, il *PropertyCopier*). Infatti il fornitore dei files di configurazione, dopo aver configurato l'oggetto attraverso il controllore di compatibilità per testarne la compatibilità con un certo file, deve ripristinare l'oggetto al suo stato originale. Pertanto viene effettuata prima, una copia dell'oggetto (ovvero viene generata un mappa che ne descrive il suo stato attuale), poi viene testata la compatibilità del file/files con l'oggetto ed infine viene ripristinato l'oggetto configurandolo con la mappa ricavata dal processo di copiatura.

1. ***La classe JSONDataImporter dipende da una classe del package org.json di Apache Maven: JSONObject.***
2. ***Esistoni alcune eccezioni che sono state discusse mediante alcuni commenti all'interno dei codici sorgenti del package in esame.***
3. ***Le possibili eccezioni sono: IllegalAccessException, NoSuchFieldException e InvalidTargetException. Per una descrizione più approfondita riguardo le eccezioni del metodo configure(), consultare il codice sorgente relativo.***

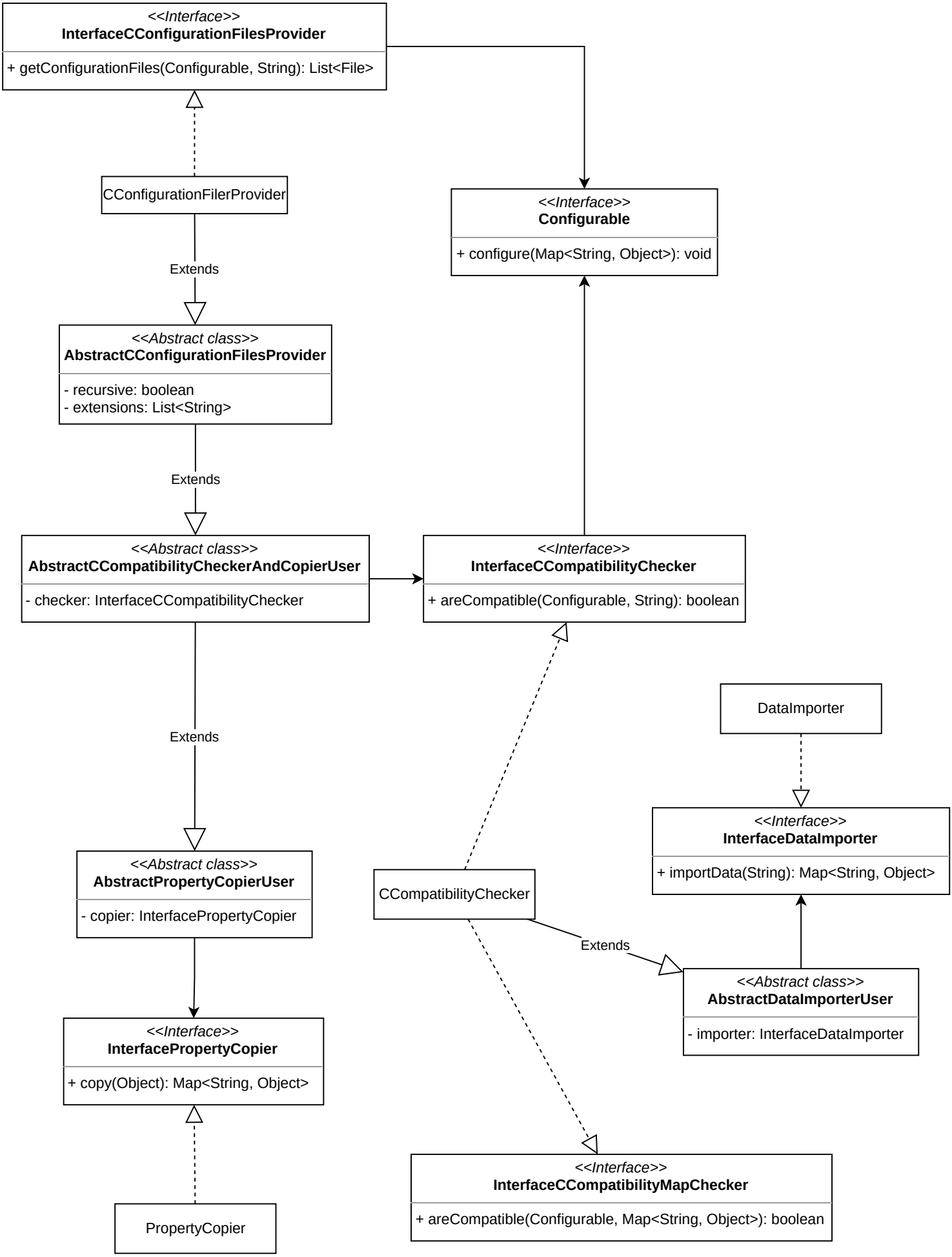

<<Interface>>
InterfaceRConfigurationFilesProvider
+ getConfigurationFiles(Object, String): List<File>
<<Abstract class>>
AbstractRConfigurationFilesProvider
- recursive: boolean
- extensions: List<String>
- configurator: InterfaceConfigurator
Extends
RConfigurationFilesProivder
Extends
<<Abstract class>>
AbstractRCompatibilityCheckerAndCopierUser
- checker: InterfaceRCompatibilityChecker
Extends
<<Abstract class>>
AbstractPropertyCopierUser
- copier: InterfacePropertyCopier
<<Interface>>
InterfaceRCompatibilityChecker
+ areCompatible(Object, String): boolean
RCompatibilityChecker
<<Interface>>
InterfacePropertyCopier
+ copy(Object): Map<String, Object>
PropertyCopier
Extends
<<Interface>>
InterfaceConfigurator
+ configure(Object, Map<String, Object>: void
<<Abstract class>>
AbstractConfiguratorAndDataImporterUser
- configurator: InterfaceConfigurator
Configurator
MathUtility
Extends
<<Abstract class>>
AbstractDataImporterUser
- importer: InterfaceDataImporter
DataImporter
<<Interface>>
InterfaceDataImporter
+ importData(String): Map<String, Object>

Configuratore (con Configurable).

Questo package comprende un insieme di classi per la configurazione mediante l'utilizzo dell'interfaccia Configurable. L'interfaccia Configurable permette all'oggetto cha le implementa di configurarsi mediante un oggetto Map<String, Object>. Il vantaggio dell'utilizzo di questo package anzichè del package di configurazione con reflection risiede nell'Information hiding delle classi che vengono configurate. Infatti, a differenza del Configurator, i valori contenuti all'interno della mappa passata in input al metodo configure dell'interfaccia Configurable non vengono iniettati direttamente mediante reflection, ma vengono gestiti direttamente dalla classe interessata alla sua configurazione. Dunque è possibile che una classe riufiti uno specifico valore contenuto all'interno della mappa anche se quest'ultimo è dello stesso tipo del campo di arrivo. Un esempio di questa applicazione è la classe Profile del package juno.data.profile, che si occupa di incapsulare i dati dell'utente connesso. I dati degli utenti sono scritti all'interno di file JSON che, trovandosi nella memoria della macchina dell'utilizzatore dell'applicazione, potrebbero subire delle modifiche inammissibili. Sfruttando quindi l'information hiding della classe Profile (nello specifico i metodi di set dei valori interessati) è possibile sottoporre ciascun dato ad un controllo scrupoloso nella quale, se qualcosa andasse storto, avverrebbe il lancio di un'eccezione. Quindi a differenza del package che sfrutta reflection fornisce maggiore solidità e consistenza al costo di una scrittura del codice decisamente più ampia. Bisogna infatti ricordare che il Configurator del package juno.data.io.reflection permette di configurare una classe senza che quest'ultima ne sia consapevole, permettendo dunque di evitare di scrivere codice aggiuntivo.

All'interno del package juno.data.io.configurable sono presenti classi analoghe a quelle presenti nel package juno.data.io.reflection, che tuttavia operano sfruttando l'interfaccia Configurable anzichè la classe Configurator.

Dai pro e contro esposti è stato possibile concludere che, all'interno di un ambiente controllato, nella quale i dati non sono forniti da un utente inconsapevole ma bensì da un programmatore, il Configurator è un componente da prendere in considerazione. Se tuttavia la sicurezza è un requisito indispensabile, l'interfaccia Configurable fornisce un livello di sicurezza maggiore permesso dall'incapsulamento ed i metodi di set.


<<Interface>>
InterfaceCConfigurationFilesProvider
+ getConfigurationFiles(Configurable, String): List<File>
CConfigurationFilerProvider
<<Interface>>
Configurable
+ configure(Map<String, Object>): void
Extends
<<Abstract class>>
AbstractCConfigurationFilesProvider
- recursive: boolean
- extensions: List<String>
Extends
<<Abstract class>>
AbstractCCompatibilityCheckerAndCopierUser
- checker: InterfaceCCompatibilityChecker
<<Interface>>
InterfaceCCompatibilityChecker
+ areCompatible(Configurable, String): boolean
Extends
DataImporter
<<Interface>>
InterfaceDataImporter
+ importData(String): Map<String, Object>
<<Abstract class>>
AbstractPropertyCopierUser
- copier: InterfacePropertyCopier
CCompatibilityChecker
Extends
<<Abstract class>>
AbstractDataImporterUser
- importer: InterfaceDataImporter
<<Interface>>
InterfacePropertyCopier
+ copy(Object): Map<String, Object>
<<Interface>>
InterfaceCCompatibilityMapChecker
+ areCompatible(Configurable, Map<String, Object>): boolean
PropertyCopier

Giocatori.

L' ntelligenze artificiali per la realizzazione delle partite in modalità singola sono state realizzate mediante l'applicazione di una forte composizione di oggetti, nella quale prendono parte principalmente gli 'esaminatori', ovvero componenti che si occupano di esaminare relativamente o meno ad un certo parametro, una lista di oggetti che, nell'ambito dell'applicazione Uno, a tempo di esecuzione, sono le carte.
L'esaminatore CardExaminer concreto estende la classe astratta MultiExaminer<T> che incapsula tre esaminatori non relativi, che a tempo di esecuzione saranno rispettivamente il componente EasyExaminer, MediumExaminer e HardExaminer. Ognuno di questi implementa in modo differente il metodo response(List<T>):

- EasyExaminer: viene utilizzato dal CardExaminer quando la difficoltà selezionata è "Easy"; la precedenza con il quale viene scelta la carta da giocare segue il seguente ordine: carte numeriche -> carte azione -> carte jolly.

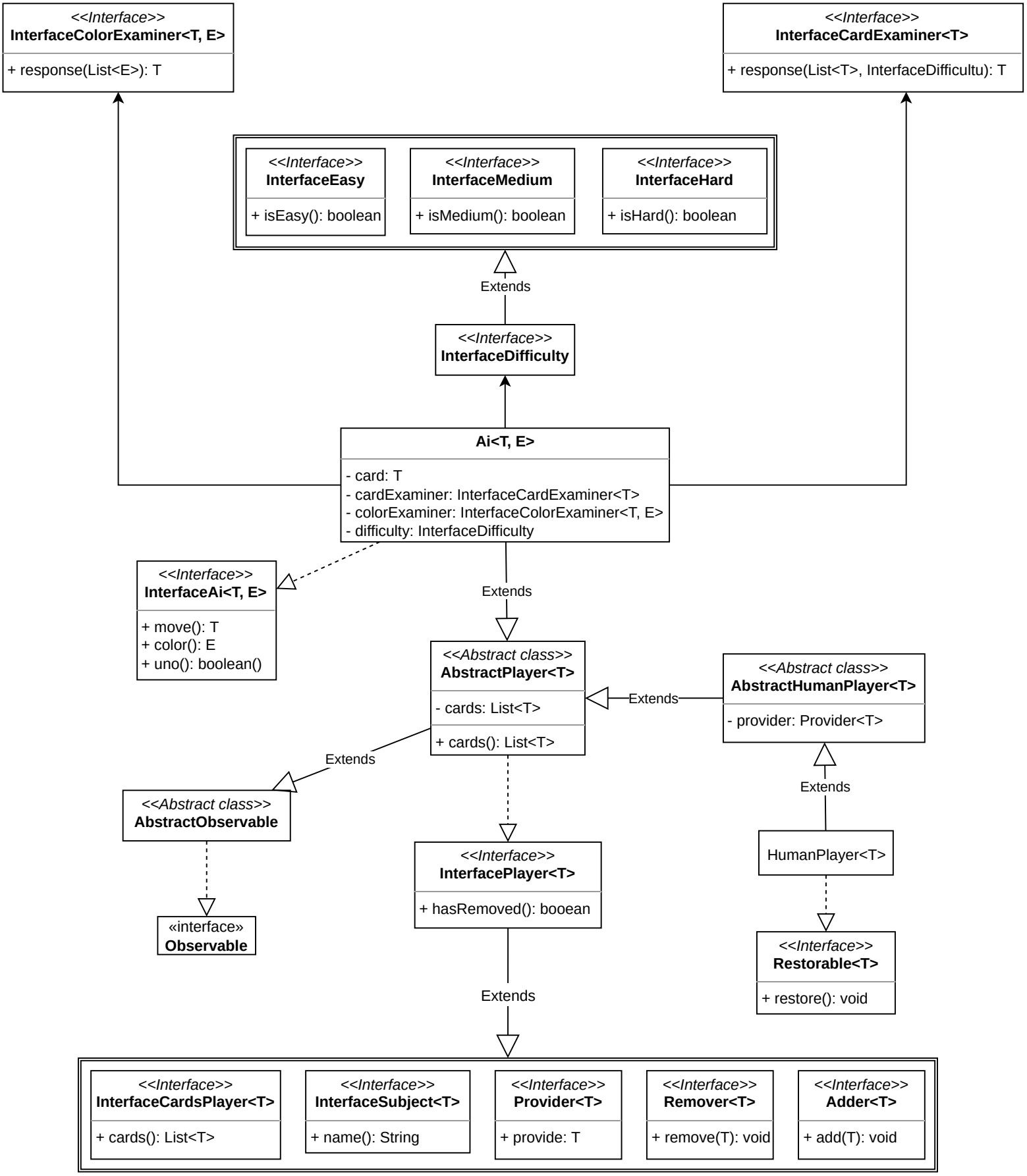
-MediumExaminer: viene utilizzato dal CardExaminer quando la difficoltà selezionata è "Medium"; la precedenza con il quale viene scelta la carta da giocare segue il seguente ordine: carte azione -> carte jolly -> carte numeriche.

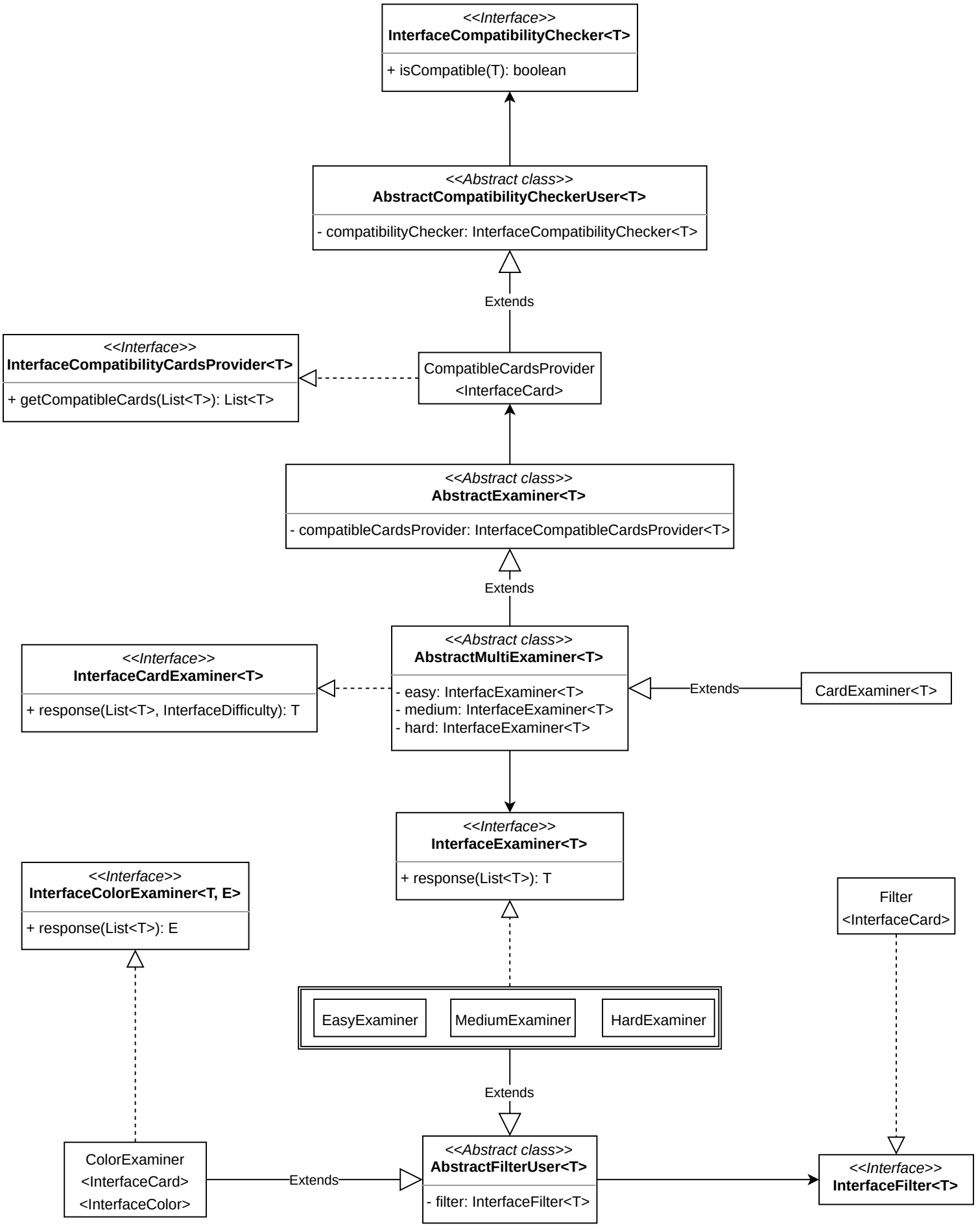
-HardExaminer: viene utilizzato dal CardExaminer quando la difficoltà selezionata è "Hard"; la precedenza con il quale viene scelta la carta da giocare segue il seguente ordine: carte jolly -> carte azione -> carte numeriche.

Il ColorExaminer è stato realizzato per permettere di decidere in modo coerente il colore che un dato giocatore AI deve dichiarare quando scarta una carta Jolly e, com'è facilmente intuibile, il metodo response(List<T>) è stato realizzato di modo che il valore di ritorno fosse il colore (InterfaceColor) con la presenza maggiore all'interno delle carte del giocatore.

Naturalmente il CardExaminer, come anche il ColorExaminer, devono poter abilmente filtrare le carte presenti all'interno della mano del giocatore al quale si stanno riferendo. Per questo motivo è stato realizzato un filtro (nel diagramma rappresentato dalla classe Filter<T>), comune per entrambi i componenti, che permette di estrarre delle carte specifiche a partire da una List<InterfaceCard>.

Il giocatore umano e le intelligenze artificiali (o AI) estendono la classe AbstractPlayer che permette ad entrambe di riutilizzare il codice relativo alla fornitura e il mantenimento delle carte.
Sono state inoltre riutilizzate le interfaccie quali Provider<T>, Remover<T> e Restorable<T> viste in diversi altri diagrammi.


<<Interface>>
InterfaceColorExaminer<T, E>
+ response(List<E>): T
<<Interface>>
InterfaceCardExaminer<T>
+ response(List<T>, InterfaceDifficultu): T
<<Interface>>
InterfaceEasy
+ isEasy(): boolean
<<Interface>>
InterfaceMedium
+ isMedium(): boolean
<<Interface>>
InterfaceHard
+ isHard(): boolean
Extends
<<Interface>>
InterfaceDifficulty
Ai<T, E>
- card: T
- cardExaminer: InterfaceCardExaminer<T>
- colorExaminer: InterfaceColorExaminer<T, E>
- difficulty: InterfaceDifficulty
<<Interface>>
InterfaceAi<T, E>
+ move(): T
+ color(): E
+ uno(): boolean()
Extends
<<Abstract class>>
AbstractPlayer<T>
- cards: List<T>
+ cards(): List<T>
Extends
<<Abstract class>>
AbstractHumanPlayer<T>
- provider: Provider<T>
Extends
<<Abstract class>>
AbstractObservable
<<Interface>>
InterfacePlayer<T>
+ hasRemoved(): booean
HumanPlayer<T>
«interface»
Observable
<<Interface>>
Restorable<T>
+ restore(): void
Extends
<<Interface>>
InterfaceCardsPlayer<T>
+ cards(): List<T>
<<Interface>>
InterfaceSubject<T>
+ name(): String
<<Interface>>
Provider<T>
+ provide: T
<<Interface>>
Remover<T>
+ remove(T): void
<<Interface>>
Adder<T>
+ add(T): void


<<Interface>>
InterfaceCompatibilityChecker<T>
+ isCompatible(T): boolean
<<Abstract class>>
AbstractCompatibilityCheckerUser<T>
- compatibilityChecker: InterfaceCompatibilityChecker<T>
Extends
<<Interface>>
InterfaceCompatibilityCardsProvider<T>
+ getCompatibleCards(List<T>): List<T>
CompatibleCardsProvider
<InterfaceCard>
<<Abstract class>>
AbstractExaminer<T>
- compatibleCardsProvider: InterfaceCompatibleCardsProvider<T>
Extends
<<Interface>>
InterfaceCardExaminer<T>
+ response(List<T>, InterfaceDifficulty): T
<<Abstract class>>
AbstractMultiExaminer<T>
- easy: InterfacExaminer<T>
- medium: InterfaceExaminer<T>
- hard: InterfaceExaminer<T>
Extends
CardExaminer<T>
<<Interface>>
InterfaceExaminer<T>
+ response(List<T>): T
<<Interface>>
InterfaceColorExaminer<T, E>
+ response(List<T>): E
Filter
<InterfaceCard>
EasyExaminer
MediumExaminer
HardExaminer
Extends
ColorExaminer
<InterfaceCard>
<InterfaceColor>
Extends
<<Abstract class>>
AbstractFilterUser<T>
- filter: InterfaceFilter<T>
<<Interface>>
InterfaceFilter<T>

AI Factory e gestione dei turni.

Per la realizzazione dei giocatori artificiali è stato applicato, come in numerose altre occasioni, il patter Factory, arricchito tuttavia da un builder (nel diagramma la classe AIBuilder) che si occupa di instanziare il singolo giocatore articiale a partire da un oggetto InterfaceDifficulty e un oggetto String che ne descrive il nome.
Il builder dunque incapsula gli esaminatori descritti nella sezione dedicata e li passa in input al costruttore della classe AI, mentre le factory, in base al numero specificato in input e la difficoltà, determina il numero di giocatore artificali da instanziare e la loro relativa capacità di gioco. Vi è inoltre la presenza di un fornitore di nomi, ovvero la classe NameProvider, che implementa l'interfaccia RelativeProvider<T, E> (fornitore relativo in quanto, in base all'input specificato, produce un certo output). Ogni volta che i giocatori artificiali vengono generati la classe NameProvider viene utilizzata, producendo nomi tipicamente differenti dalla partita precedente (se avvenuta).

Per la realizzazione della gestione dei turni è stata realizzata una struttra dati specifica definita come Donut.
La Donut è un ArrayList chiuso su se stesso (come lascia intendere appunto il suo nome). I metodi initialize(int) e initialize() permettono rispettivamente di definire l'elemento da considerare come quello corrente medinate il suo indice oppure in maniera casuale. Sono presenti inoltre i metodi next() e invert() che permettono rispettivamente di passare al successivo elemento presente all'interno della lista oppure di invertire l'ordine con il quale la si osserva. Quando l'elemento corrente si trova alla fine della lista e viene invocato il metodo next(), il nuovo elemento corrente diventà il primo elemento. Questa struttra dati è stata messa in funzione anche del riproduttore audio principale, cioè quello che si occupa di riprodurre le musiche dell'applicazione.

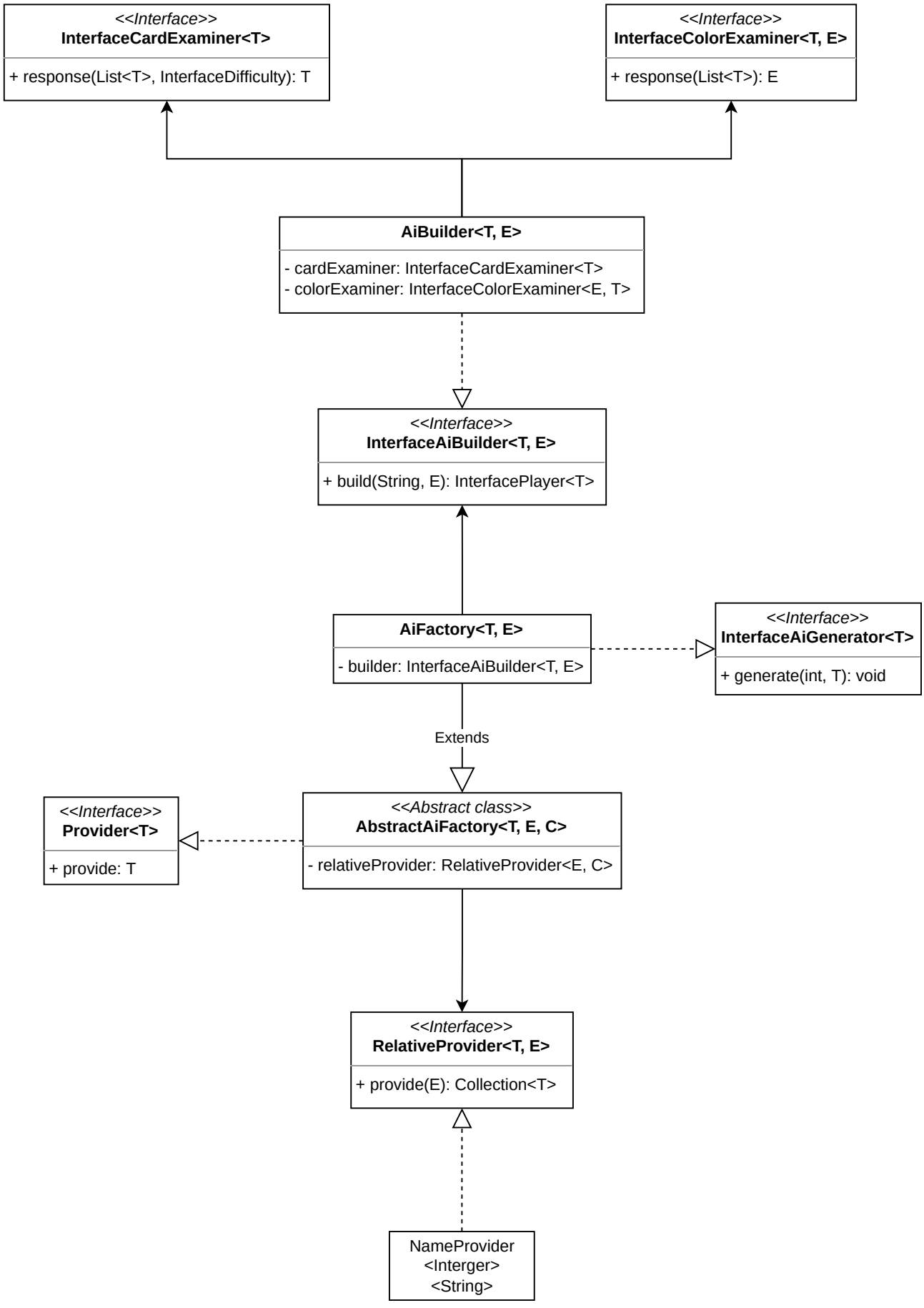
Quando i giocatori artificiali vengono realizzati, la factory aggiorna i suoi osservatori. La classe PlayerProvider viene aggiornata ed introduce all'interno di una Donut i giocatori artificiali ed il giocatore umano incapsulato in precedenza; a questo punto anche la classe PlayersProvider aggiorna i suoi osservatori che incapsulano la Donut. La classe TurnMover, TurnJumper, CurrentPlayerProvider e Inverter sono tra gli osservatori della classe PlayerProvider e incapsulano la donut non appena aggiornati. Ognuna di queste ultime classi citate fornisce una differente funzionalità all'interno dell'applicazione:

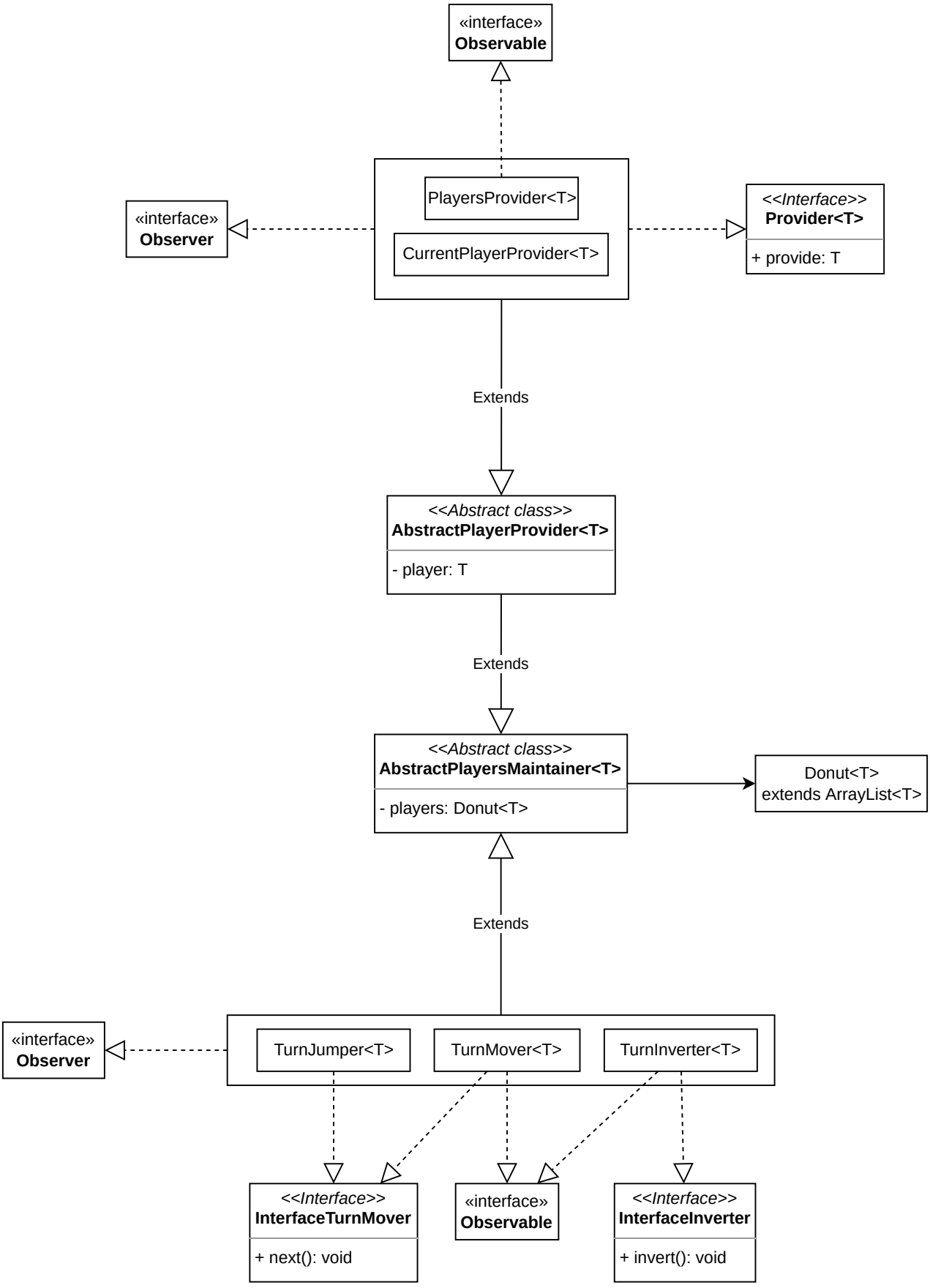
- TurnMover: implementa l'interfaccia InterfaceTurnMover e permette di muove in avanti la Donut (mediante il metodo next()) e successivamente aggiornare i suoi osservatori.

- TurnJumper: stesso comportameno della classe TurnMover ma non notifica alcun osservatore in quanto non osservabile. La realizzazione di questa classe ha permesso di risolvere problemi legati all'aggionamento degli osservatori in seguito all'invocazione del metodo next() della classe TurnMover (in pratica previene l'aggiornameto degli osservatori di TurnMover quando viene attivata una carta che fa passare il turno al giocatore successivo).

- CurrentPlayerProvider: fornisce il giocatore (InterfacePlayer a tempo di esecuzione) corrente.

- Inverter: componente che si occupa di invertire il senso con il quale prelevare i giocatori durante il proseguimento della partita.


<<Interface>>
InterfaceCardExaminer<T>
+ response(List<T>, InterfaceDifficulty): T
<<Interface>>
InterfaceColorExaminer<T, E>
+ response(List<T>): E
AiBuilder<T, E>
- cardExaminer: InterfaceCardExaminer<T>
- colorExaminer: InterfaceColorExaminer<E, T>
<<Interface>>
InterfaceAiBuilder<T, E>
+ build(String, E): InterfacePlayer<T>
AiFactory<T, E>
- builder: InterfaceAiBuilder<T, E>
<<Interface>>
InterfaceAiGenerator<T>
+ generate(int, T): void
Extends
<<Interface>>
Provider<T>
+ provide: T
<<Abstract class>>
AbstractAiFactory<T, E, C>
- relativeProvider: RelativeProvider<E, C>
<<Interface>>
RelativeProvider<T, E>
+ provide(E): Collection<T>
NameProvider
<Interger>
<String>


«interface»
Observable
PlayersProvider<T>
CurrentPlayerProvider<T>
«interface»
Observer
<<Interface>>
Provider<T>
+ provide: T
Extends
<<Abstract class>>
AbstractPlayerProvider<T>
- player: T
Extends
<<Abstract class>>
AbstractPlayersMaintainer<T>
- players: Donut<T>
Donut<T>
extends ArrayList<T>
Extends
«interface»
Observer
TurnJumper<T>
TurnMover<T>
TurnInverter<T>
<<Interface>>
InterfaceTurnMover
+ next(): void
«interface»
Observable
<<Interface>>
InterfaceInverter
+ invert(): void

Applicazione degli stream.

Quelli riportati di seguito sono alcuni esempi relativi all'applicazione degli Stream di java.

Nell'esempio riportato di seguito gli Stream sono stati applicati nella realizzazione del metodo generate(T, E) dell'interfaccia InterfaceAiGenerator all'interno della classe concreata AiFactory. Gli oggetti ritornati dal metodo provide (Sono ti tipo String a Runtime) vengono mappati mediante il metodo map() in oggetti AI, dopo lo stream viene convertito il lista.

```
@Override
public void generate(int num,
                     @NotNull E difficulty) {
    if(num < 1) throw new IllegalArgumentException("Invalid players number");
    players = Objects.requireNonNull(getRelativeProvider())
            .provide(num).stream().map(name -> builder.build(name, difficulty)).toList();
    updateAll();
}
```

La classe Filter del package juno.model.subject.ai.examiner utilizza, in ogni metodo che implementa dall'interfaccia InterfaceFilter, gli Stream. Nel caso riportato qui sotto viene utilizzato il metodo filter con il predicato specificato,che permettere di ottenere uno Stream di oggetti InterfaceCard che soddisfano tale predicato (ovvero carte jolly). Successivamente lo stream viene convertito in lista medinate l'operazione terminale toList() e restituito.

```
@Override
public List<InterfaceCard> jolly(@NotNull List<InterfaceCard> cards) {
    return cards.stream()
            .filter(card -> card.action() != null && card.action().isJolly())
            .toList();
}
```

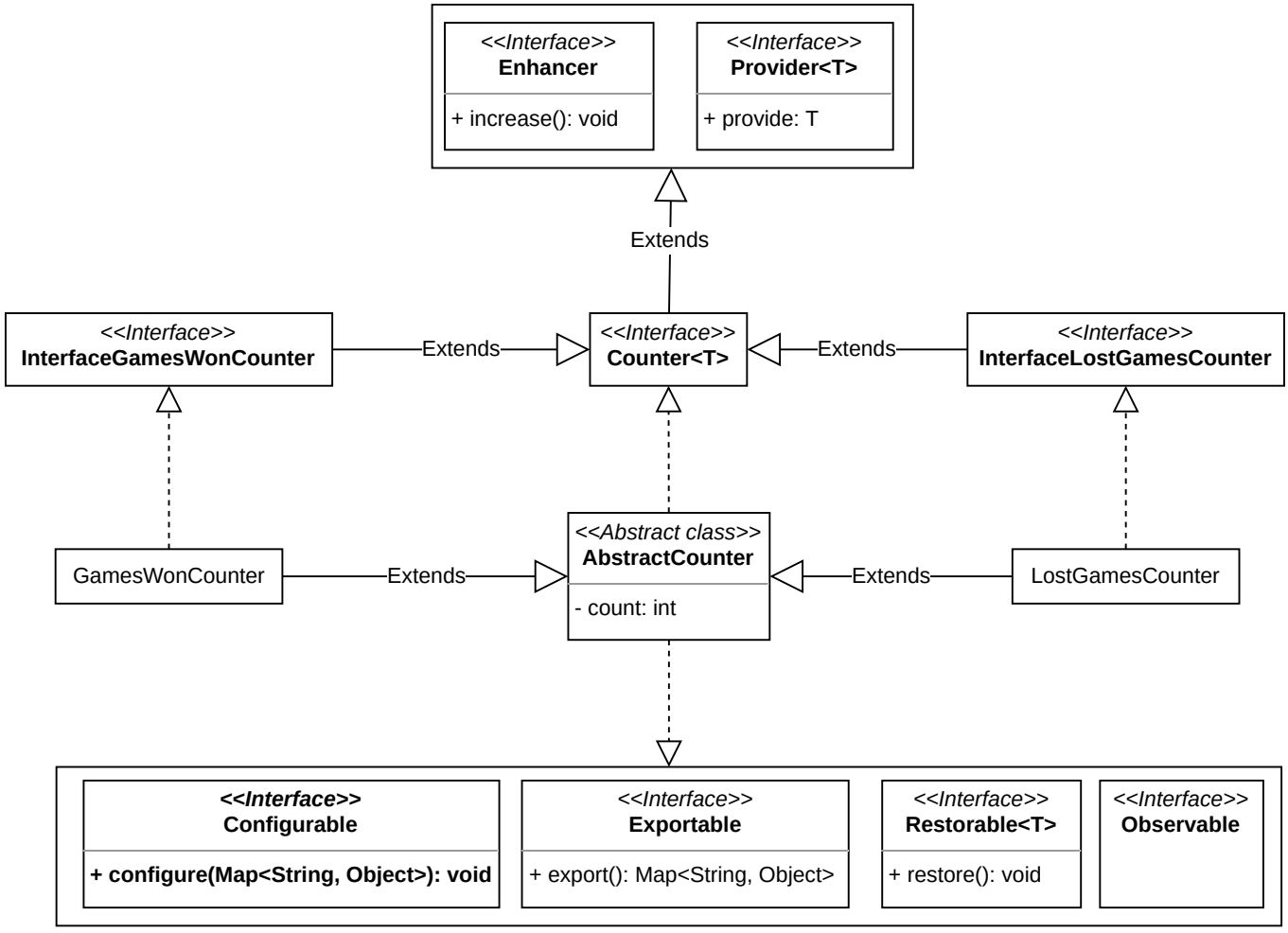
In ultimo, il metodo riportato permette di ottenere una lista di InterfaceCard compatibili. La compatibilità è determinata chiaramente dal predicato (ovvero il metodo isCompatible(T) della classe CompatibilityChecker). Lo stream di oggetti compatibili viene convertito il lista e in seguito restituito.

```
@Override
public List<InterfaceCard> getCompatibleCards(@NotNull List<InterfaceCard> cards) {
    return cards.stream()
            .filter(Objects.requireNonNull(getCompatibilityChecker())::isCompatible)
            .toList();
}
```

All'interno dell'applicazione l'utilizzo degli Stream è sparso ed è stato evitato soltanto in alcuni casi specifici nella quale è stato richiesto il lancio di eccezioni (no RuntimeException). L'implementazione del metodo configure() della classe Configurator nel package juno.model.data.io.input.reflection ne è un esempio.

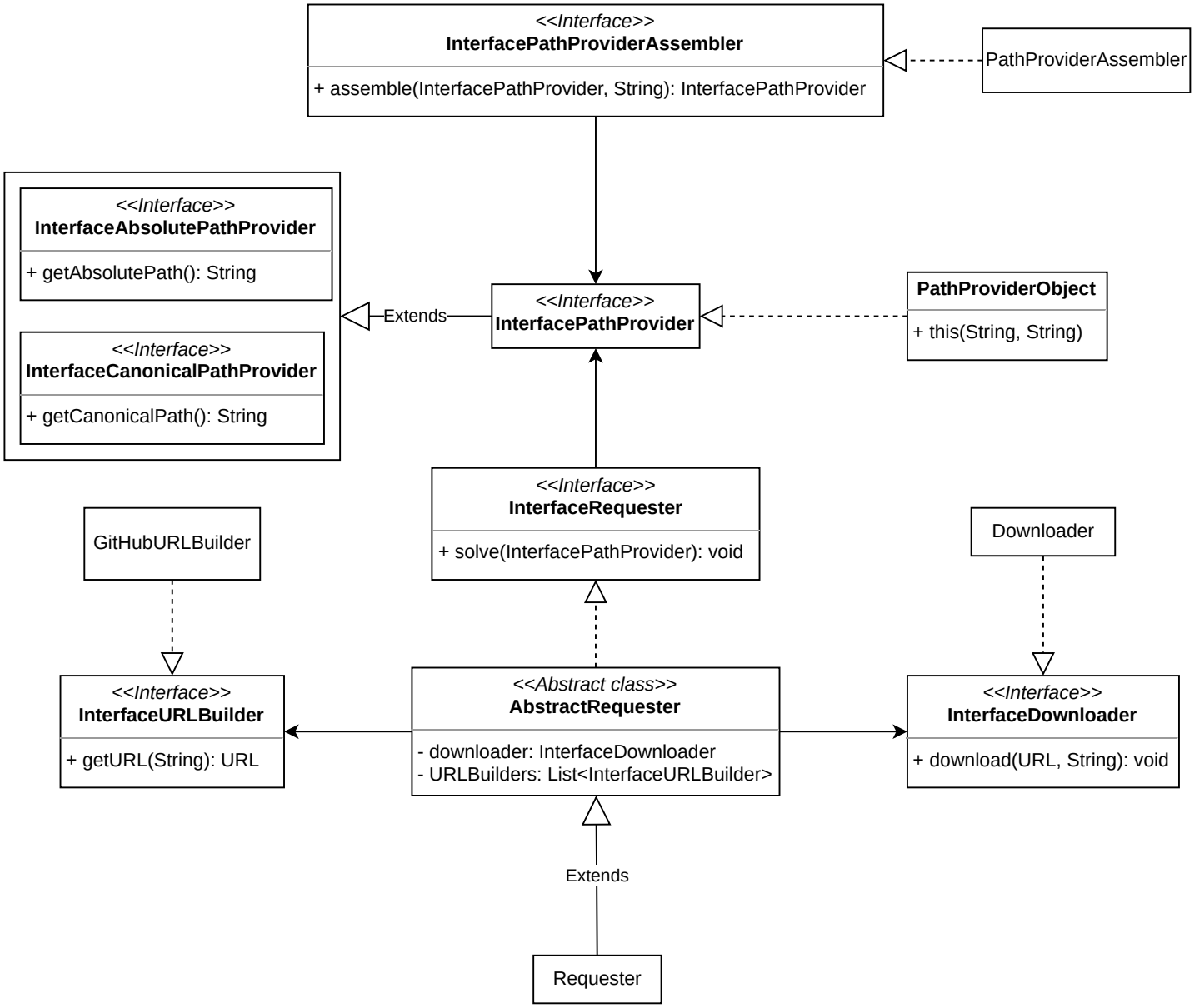
Contatori dei punteggi.

I contaggio delle partite vinte e perse è stato ottenuto mediante lo schema riportato di seguito.
L'interfaccia Counter<T>, che estende le due interfaccia Enhancer per l'incremento e Provider<T> per la fornitura è implementata dalla classe astratta AbstractCounter. quest'ultima classe contiene il codice implementativo dei due contatori: GamesWonCounter e LostGamesCounter. Entrambi i contatori aggiornano i loro osservatori, in quanto osservabili, quando il contatore subisce un cambiamento. Le due interfaccia InterfaceGamesWonCounter ed InterfaceLosGamesCounter sono state realizzate per permettere, a chi lo richieste, di effettuare un ricoscimento del contatore nel rispetto del dependency inversion. La classe Goal, infatti, sblocca i premi (ovvero oggetti di tipo Award) quando viene aggiornata da un dei due contatori. Per permette dunque alla classe Goal di distinguere tra i due contatori è stato realizzato proprio tale modello.
Entrambi i contatori sono esportabili e configurabili mediante l'implementazione delle interfaccia Exportable e Configurable. Quando un profile viene caricato infatti è necessario importare i dati relativi alle relative partite vinte e perse e configurare i contatori. Infine, viene implementata l'interfaccia Restorable che permette di azzerare i contatori quando viene effettuato un log out da un profilo.



Download.

Sin dall'inizio della progettazione dell'applicazione è stata presente l'intenzione di realizzare un meccanismo che permettesse di scaricare direttamente da internet i file mancanti nel caso in cui ne fossero stati rilevati durante il pre caricamento (o inizializzazione) dell'applicazione. Quel meccanisco è stato realizzato e lo schema riportato di seguito è un sotto componente di tale sistema. Nello specifico esso si preoccupa di gestire le richieste dei componenti software che durante il pre caricamente, lavorando con i file all'interno del disco, rilevano una mancanza.
Per poter assolvere ad una richiesta è chiaramente necessario che la connettività ad internet sia presente nella prima fase di esecuzione del programma. Un qualsiasi componente per poter richiedere un download deve fornire un oggetto che implementa l'interfaccia InterfacePathProvider. Da un tale oggetto, infatti, il Requester può estrearre il canonical path, utilizzato per generare la URL mediante l'ausilio di uno degli URLBuilder incapsulati, e l'absolute path, necessario per definire il percorso nel disco nel quale salvare il file eventualmente scaricato da internet. Il canonical path e l'absolute path vengono forniti in input alla classe Downloader che effettua il download concreto del file.

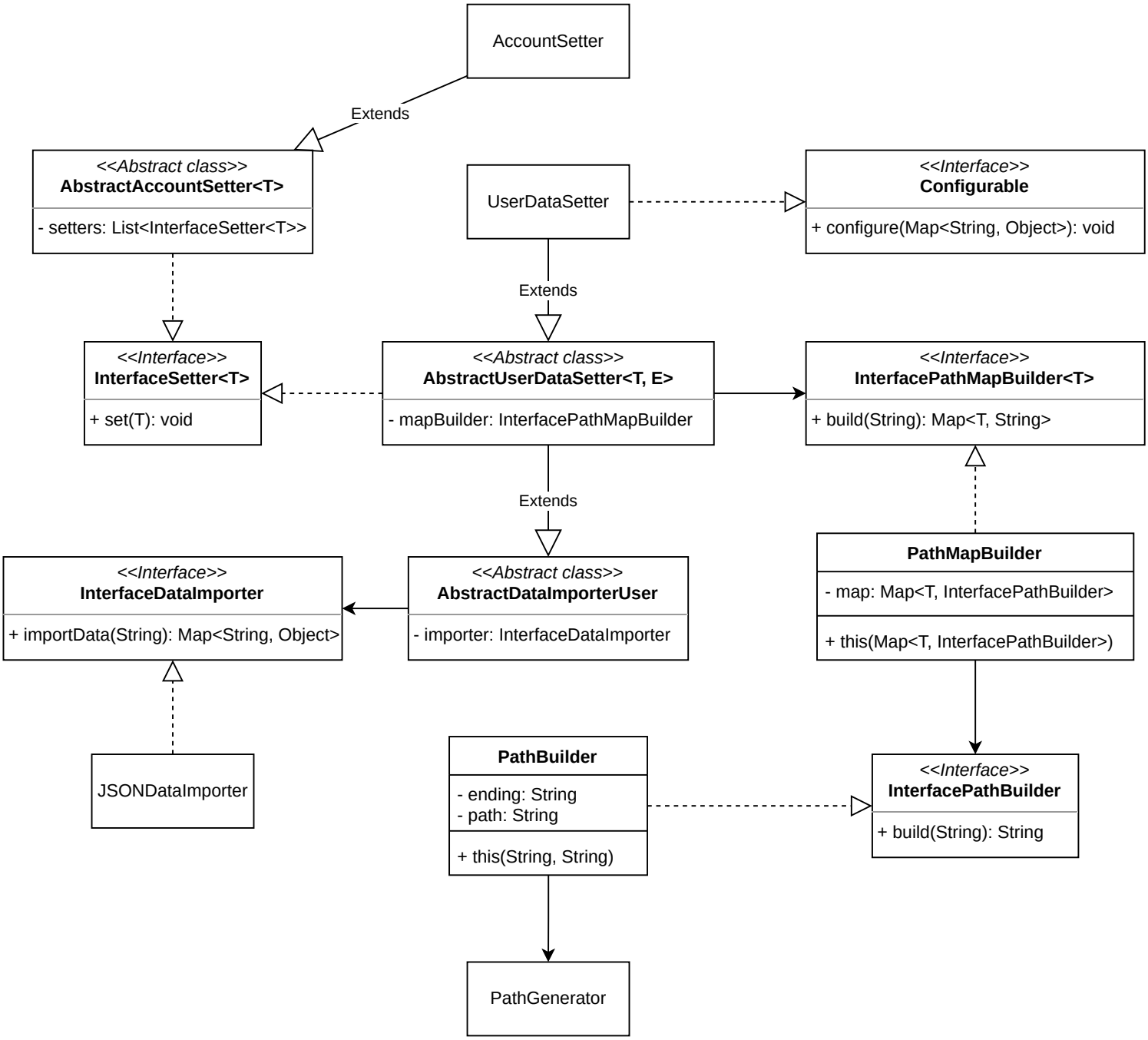


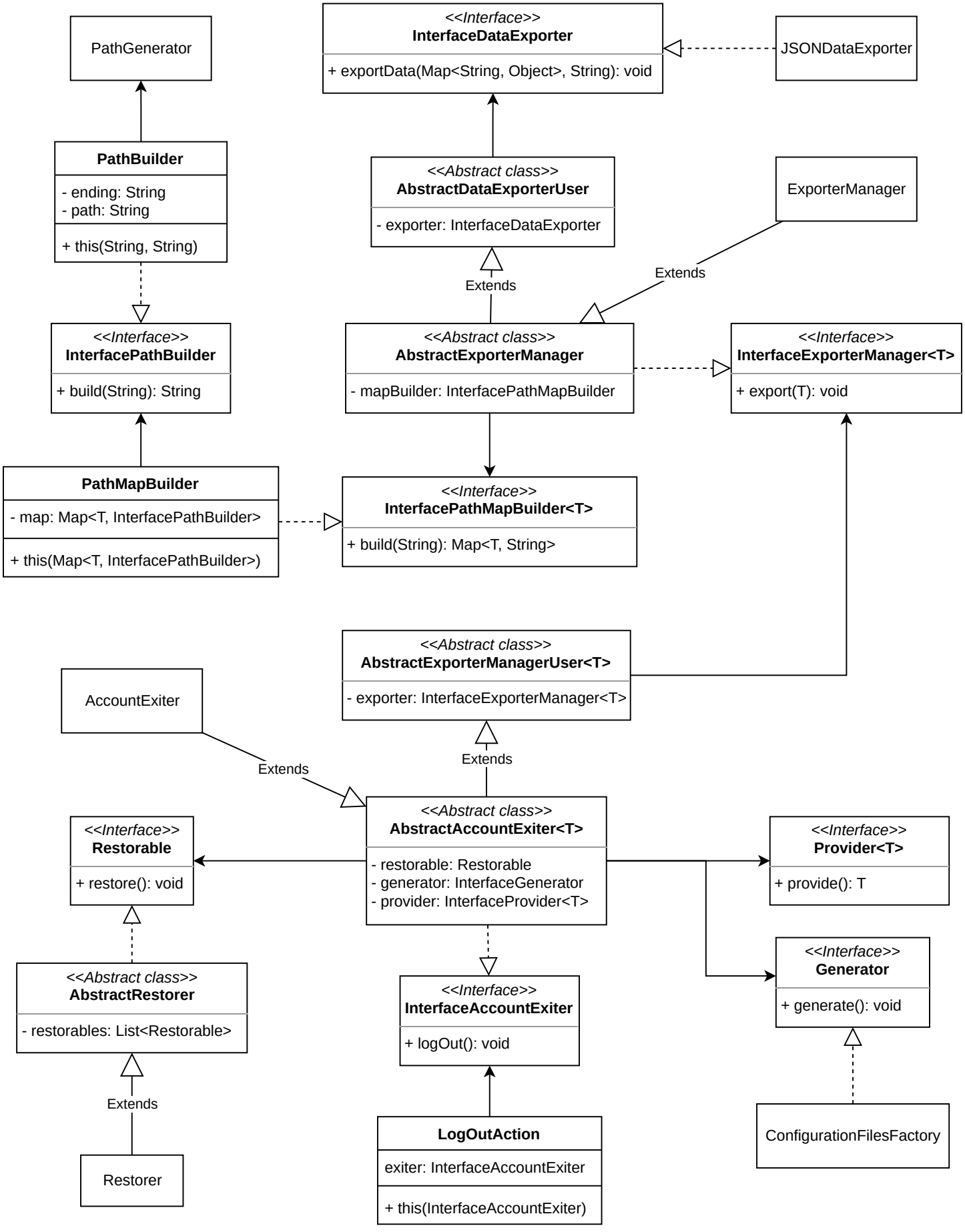
Log in & Log out.

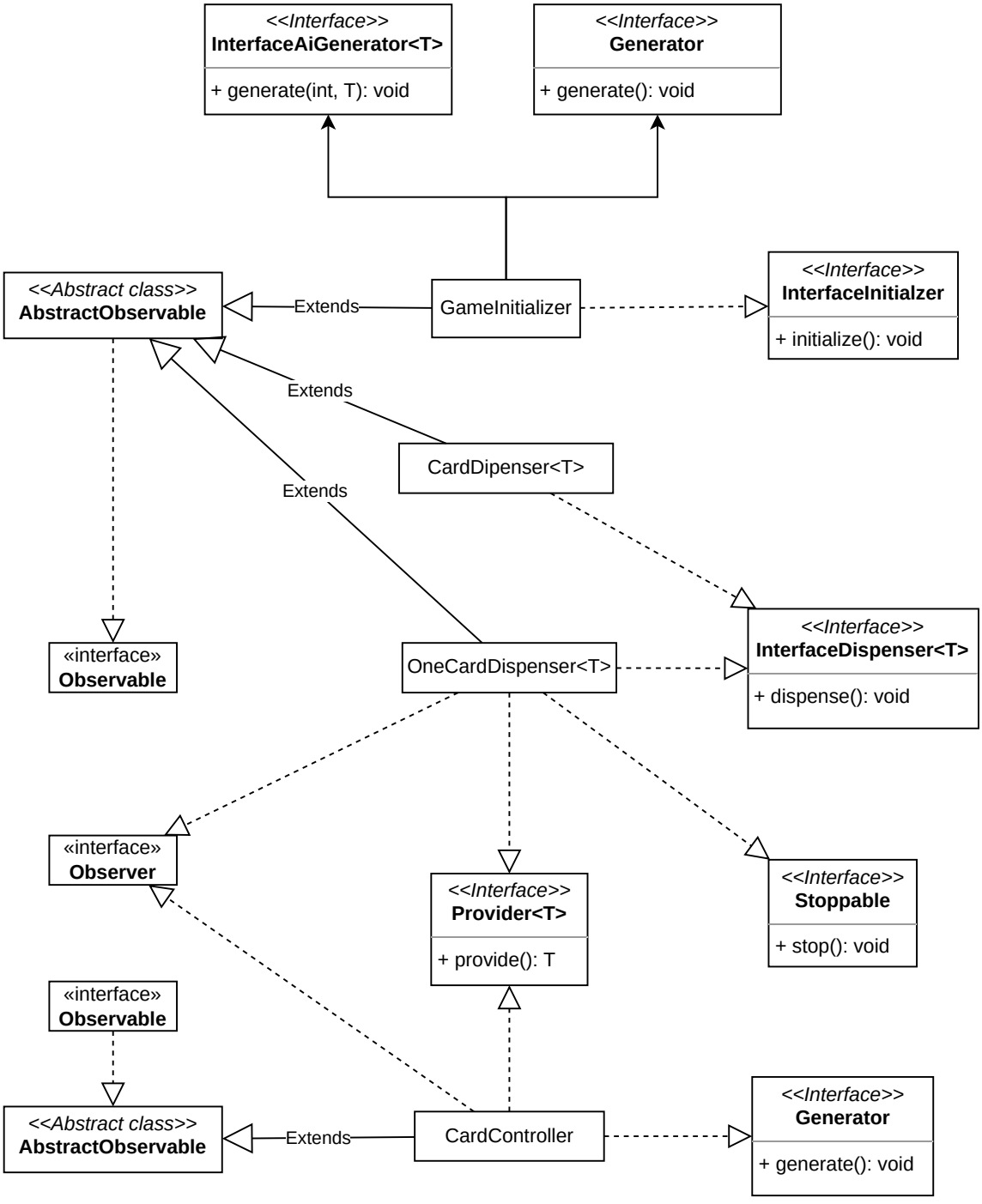
Durante il processo di pre caricamento dell'applicazione, esattamente dopo aver configurator model, view, e controller, avviene il controllo dei file di configurazione della classe Profile. I File contenuti all'interno della cartella juno/data/config/profiles vengono ispezionati uno ad uno e soltanto quelli che risultano essere compatibili vengono processati per la creazione dei relativi bottoni da inserire nell'apposito pannello nella porzione di view dedicata. Quando un bottone per l'accesso ad un profilo viene cliccato viene azionato un meccanismo mediante il quale le classe Avatar, Profile e Score vengono configurate. Le classe AvatarImage e Goal vengono conseguentemente configurate in modo automatico mediante l'aggiornamento eventualmente ricevuto da parte della classe Score. I file di configurazione vengono ottenuti accoppiando al nome profilo la parte finale ed inizialize del percorso. Per esempio, per l'ottenimento del file di configurazione legato all'avatar dell'account con nome 'Simone' viene creato il seguente percorso mediante l'asilio del PathBuilder:

../juno/data/config/avatars + Simone + -avatar.json

Gli oggetti PathBuilder vengono costruiti a runtime dagli inizializzatori (Ovvero le classi adibite alla connessione), e ne viene costruito uno per ciscuna classe che deve essere configurata quando viene effettuata una procedura di log in. La classe PathMapBuilder associa invece all'interno di una mappa incapsulata i percorsi agli oggetti Configurabili. La classe AccountSetter infine, mediante l'ausilio delle classe UserDataSetter, imposta finalmente i dati per ogni entry contenuta nella mappa fornita dal PathMapBuilder per lo specifico profile name fornito.
La procedura di Log out invece, realizzata mediante l'interconnesione dei componenti della figura successiva, si comporta in modo del tutto opposto a quella di Log in. La classe PathMapBuilder in questo classe fornisce i percorsi (sempre gli stessi) associati stavolta ad oggetti Exportable. Le mappe ottenute dall'invocazione del metodo export() dell'interfaccia Exportable vengono convertite e salvata all'interno dei percorsi indicati.




PathGenerator
<<Interface>>
InterfaceDataExporter
+ exportData(Map<String, Object>, String): void
JSONDataExporter
PathBuilder
- ending: String
- path: String
+ this(String, String)
<<Abstract class>>
AbstractDataExporterUser
- exporter: InterfaceDataExporter
ExporterManager
Extends
Extends
<<Interface>>
InterfacePathBuilder
+ build(String): String
<<Abstract class>>
AbstractExporterManager
- mapBuilder: InterfacePathMapBuilder
<<Interface>>
InterfaceExporterManager<T>
+ export(T): void
PathMapBuilder
- map: Map<T, InterfacePathBuilder>
+ this(Map<T, InterfacePathBuilder>)
<<Interface>>
InterfacePathMapBuilder<T>
+ build(String): Map<T, String>
<<Abstract class>>
AbstractExporterManagerUser<T>
- exporter: InterfaceExporterManager<T>
AccountExiter
Extends
Extends
<<Interface>>
Restorable
+ restore(): void
<<Abstract class>>
AbstractAccountExiter<T>
- restorable: Restorable
- generator: InterfaceGenerator
- provider: InterfaceProvider<T>
<<Interface>>
Provider<T>
+ provide(): T
<<Interface>>
Generator
+ generate(): void
<<Abstract class>>
AbstractRestorer
- restorables: List<Restorable>
<<Interface>>
InterfaceAccountExiter
+ logOut(): void
Extends
LogOutAction
exiter: InterfaceAccountExiter
+ this(InterfaceAccountExiter)
ConfigurationFilesFactory
Restorer


<<Interface>>
InterfaceAiGenerator<T>
+ generate(int, T): void
<<Interface>>
Generator
+ generate(): void
<<Abstract class>>
AbstractObservable
Extends
GameInitializer
<<Interface>>
InterfaceInitialzer
+ initialize(): void
Extends
CardDipenser<T>
Extends
«interface»
Observable
OneCardDispenser<T>
<<Interface>>
InterfaceDispenser<T>
+ dispense(): void
«interface»
Observer
<<Interface>>
Provider<T>
+ provide(): T
<<Interface>>
Stoppable
+ stop(): void
«interface»
Observable
<<Abstract class>>
AbstractObservable
Extends
CardController
<<Interface>>
Generator
+ generate(): void

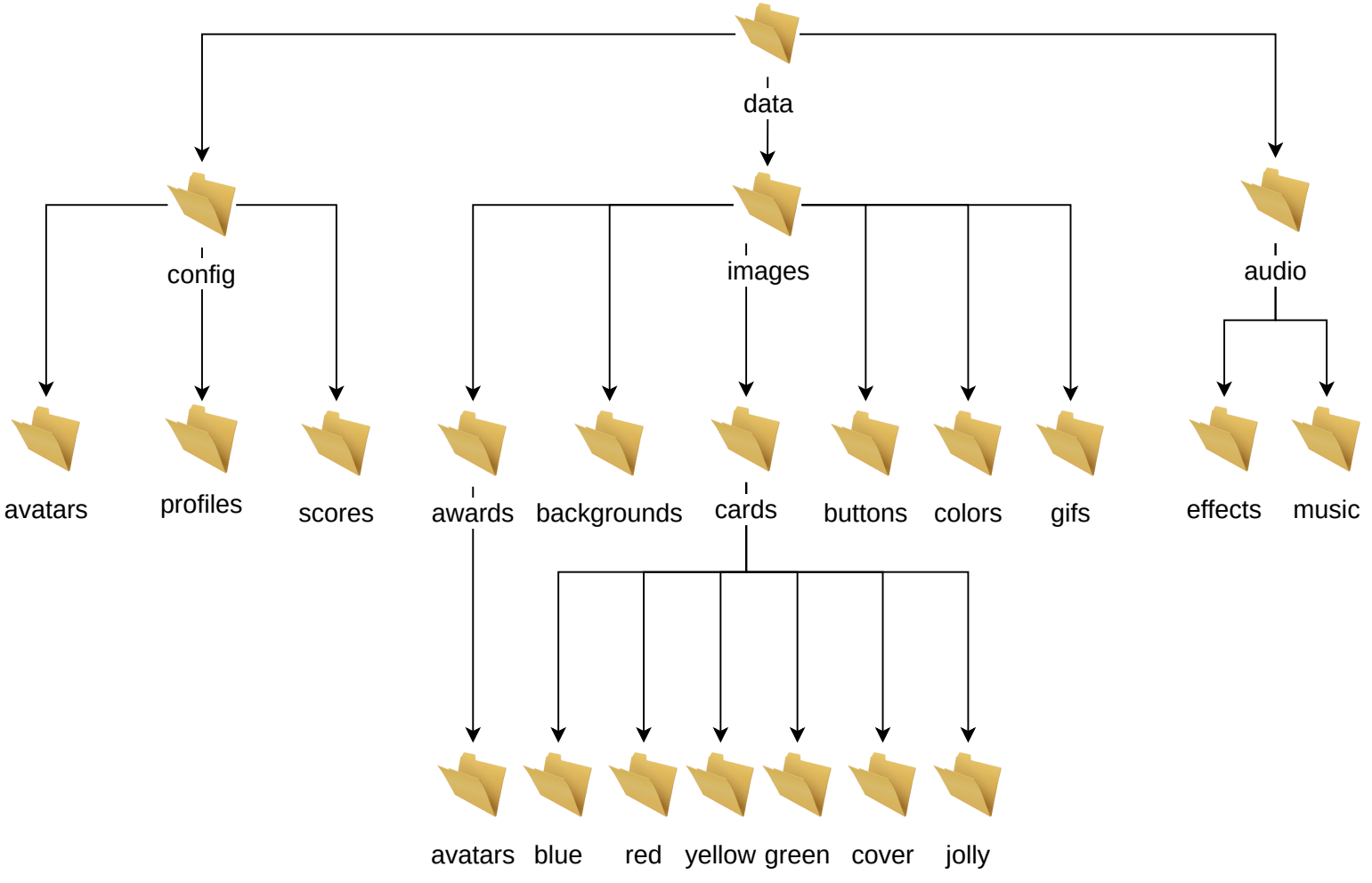
Albero delle directory.

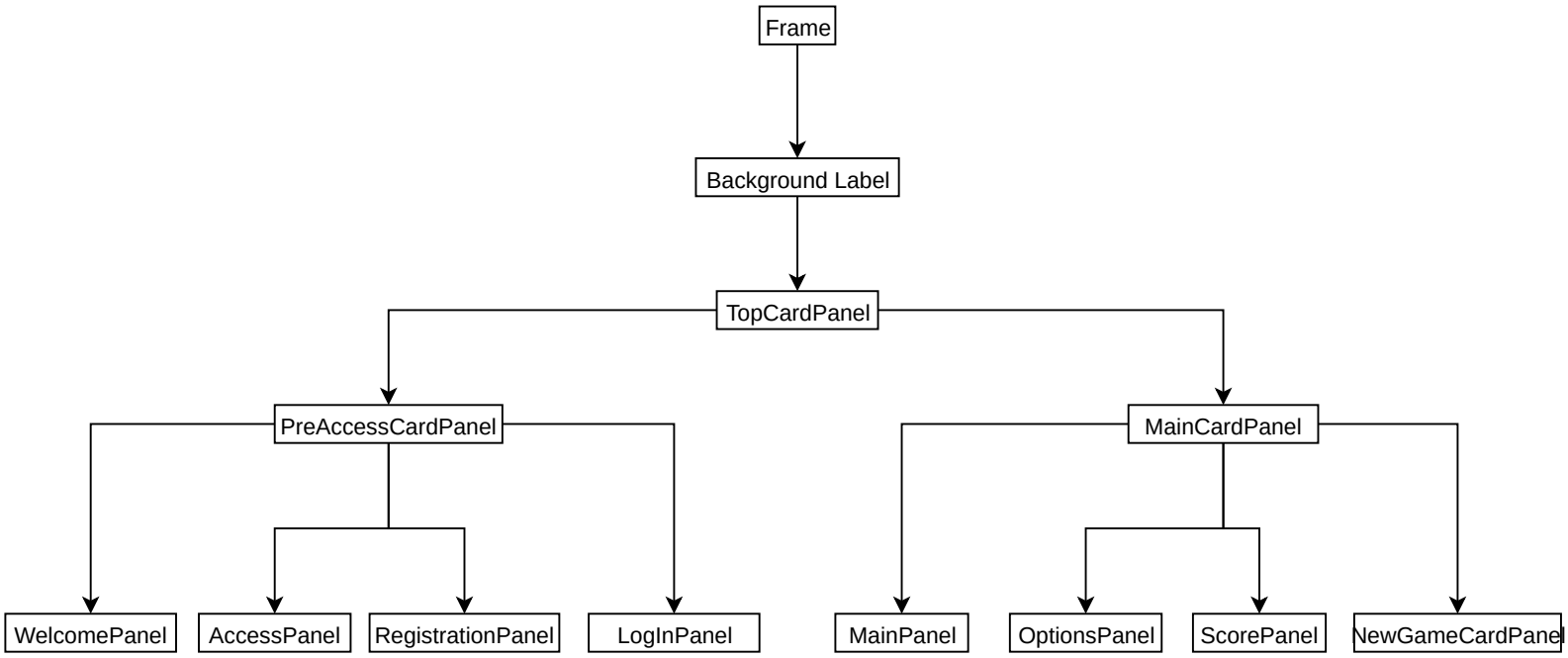
L'albero delle directory esposto nel seguito restituisce una panoramica della struttura delle cartelle dei dati dell'applicazione. Tale albero è stato mappato nella classe ProgramDirectory all'interno del package juno.init. All'interno della cartella data sono presenti le tre cartelle principali:

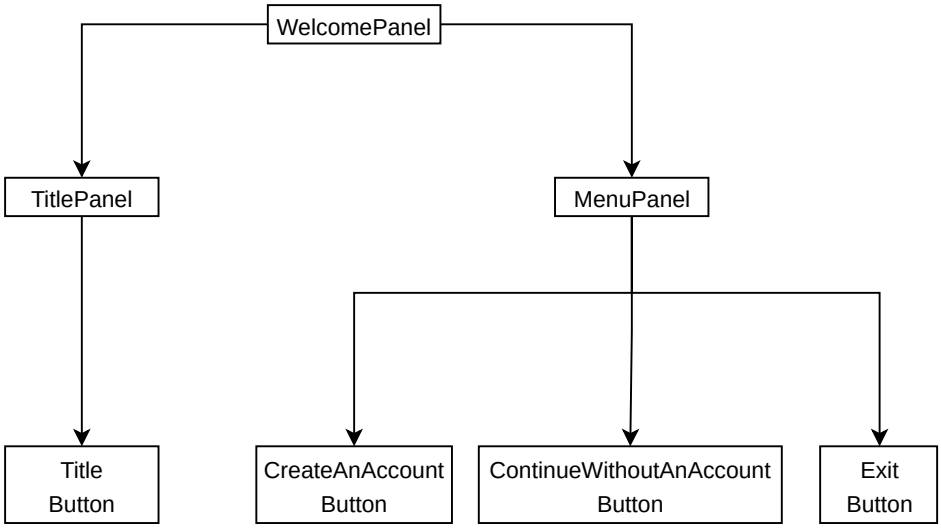
- CONFIG: cartella adibita al contenimento dei dati degli utenti.
    - AVATARS: contiene i dati relativi agli avatar degli utenti.
    - PROFILES: contiene i dati relativi al profilo degli utenti.
    - SCORES: contiene i dati relativi ai punteggi dei giocatori.

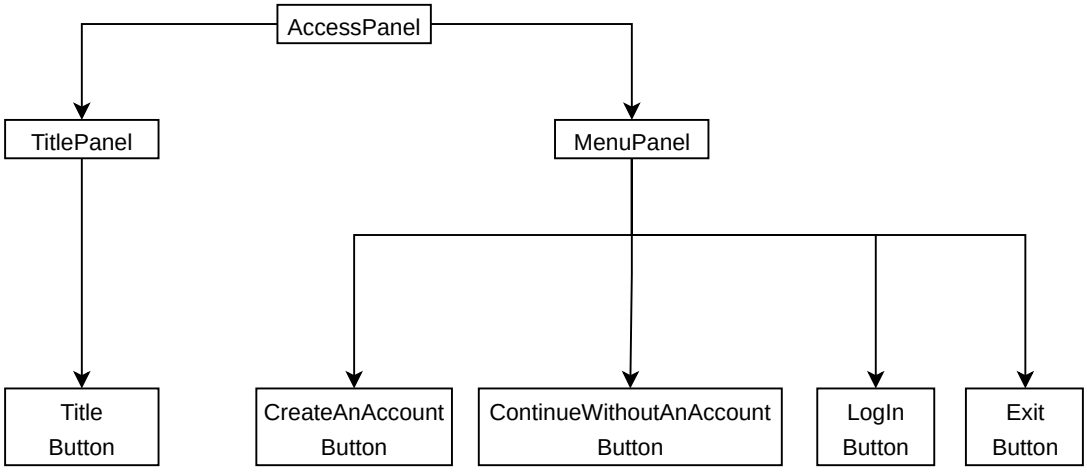
- IMAGES: cartella adibita al contenimento di tutti i file che rappresentano immagini.
    - AWARDS: cartella adibita al contenimento delle immagini dei premi.
        - AVATARS: contiene le immagini relative agli avatar
    - BACKGROUNDS: contiene le immagini relative agli sfondi dell'applicazione.
    - CARDS: cartella adibita al contenimento dei file relativi alle carte.
        BLUE: contiene le immagini delle carte blue.
        RED: contiene le immagini della carte rosse.
        YELLOW: contiene le immagini delle carte gialle.
        GREEN: contiene le immagini delle carte verdi.
        JOLLY: contiene le immagini delle carte jolly.
        COVER: contiene le immagini delle cover.
    - BUTTONS: contiene le immagini relative ai bottoni dell'interfaccia grafica.
    - COLORS: contiene le immagini relative ai quattro colori del gioco UNO.
    - GIFS: contiene le immagini con estensione .gif.

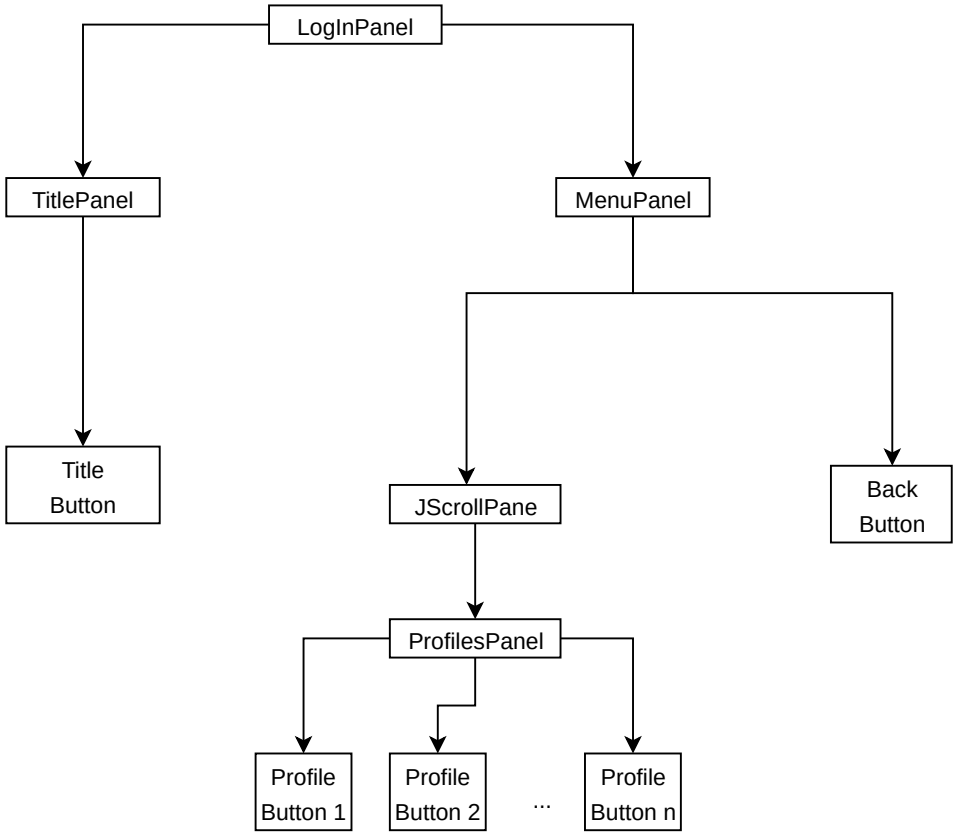
- AUDIO: cartella adibita al contenimento dei file audio.
    - EFFECTS: contiene i file audio utilizzati come effetti audio.
    - MUSIC: contiene i file audio utilizzati come musica.

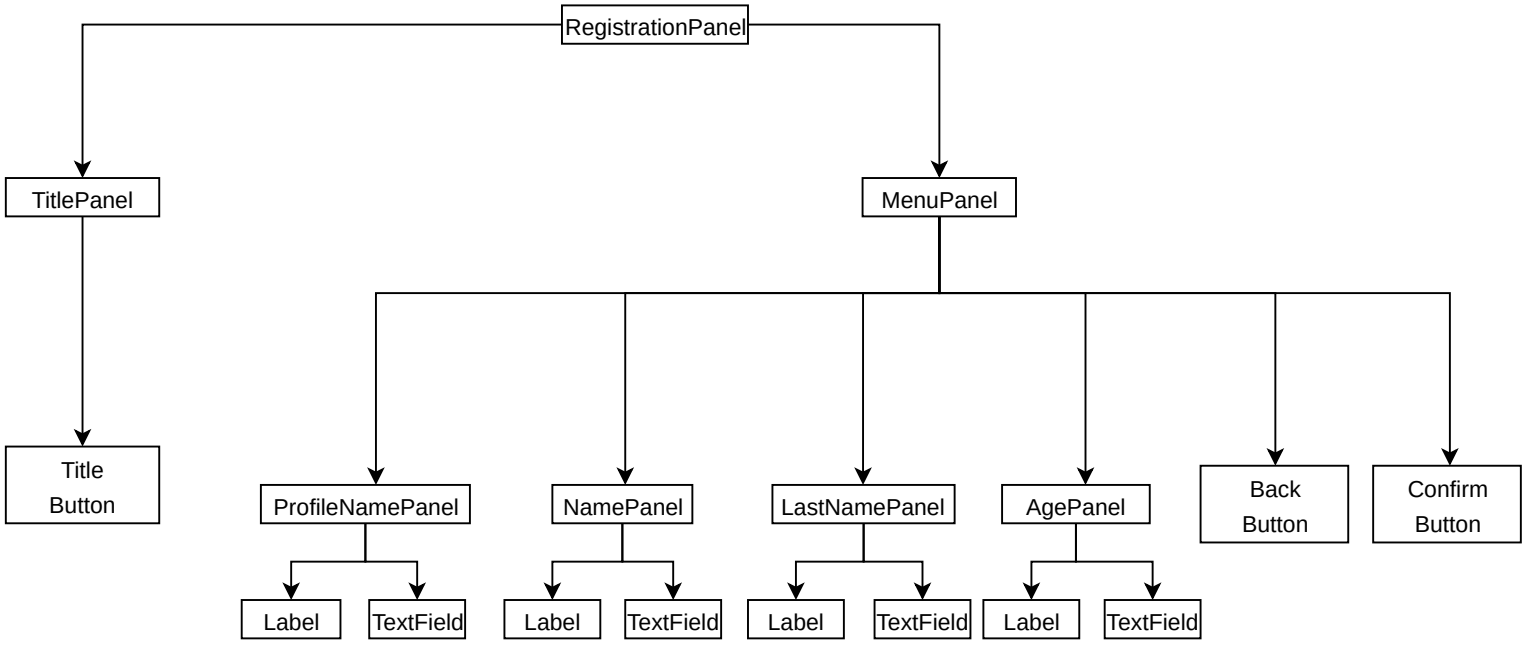


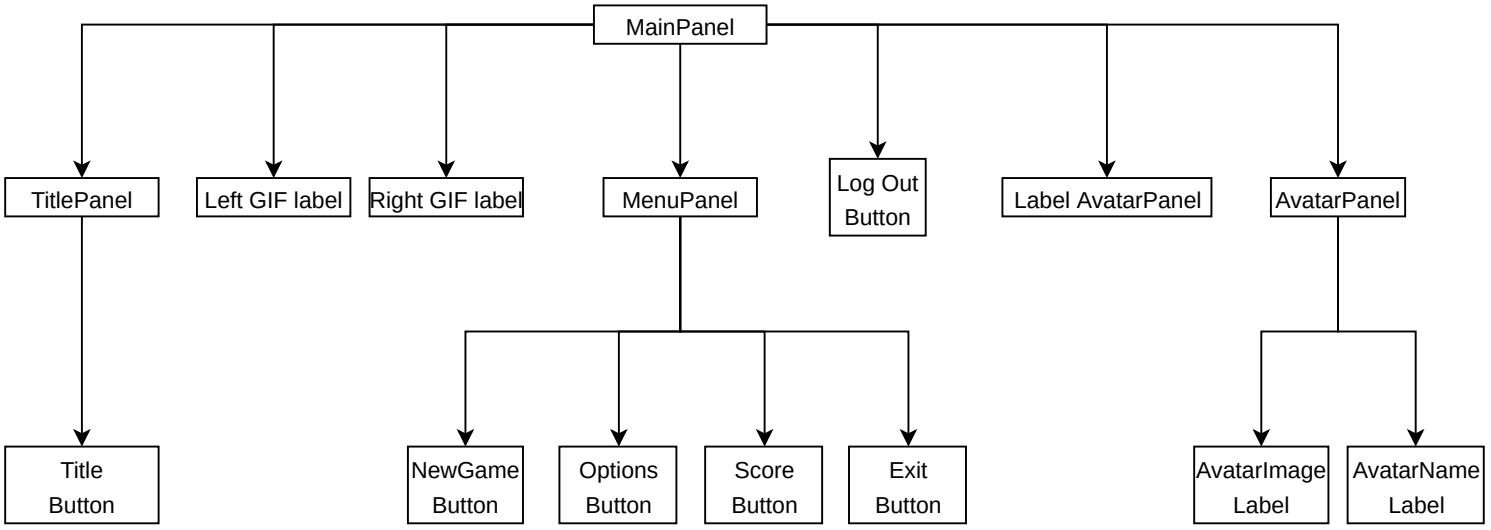

Frame
Background Label
TopCardPanel
PreAccessCardPanel
MainCardPanel
WelcomePanel
AccessPanel
RegistrationPanel
LogInPanel
MainPanel
OptionsPanel
ScorePanel
NewGameCardPanel

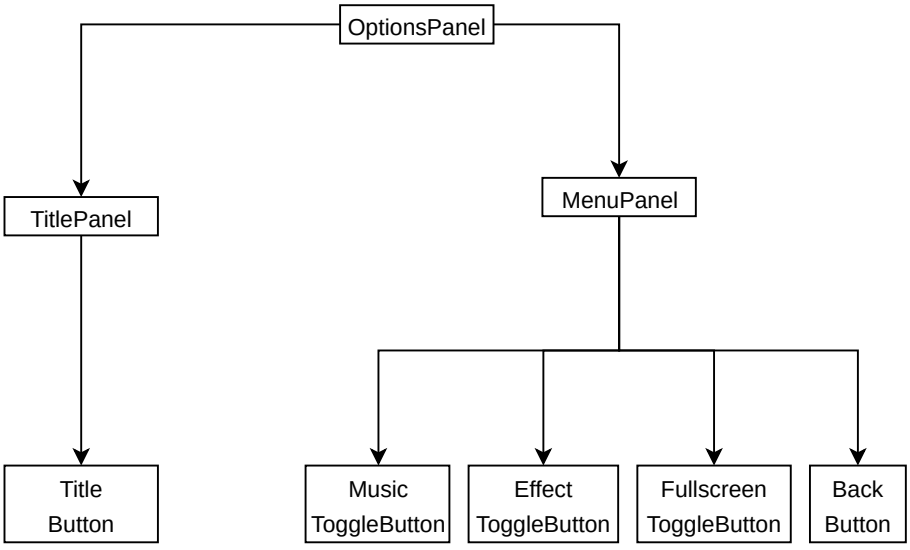

WelcomePanel
TitlePanel
MenuPanel
Title Button
CreateAnAccount Button
ContinueWithoutAnAccount Button
Exit Button

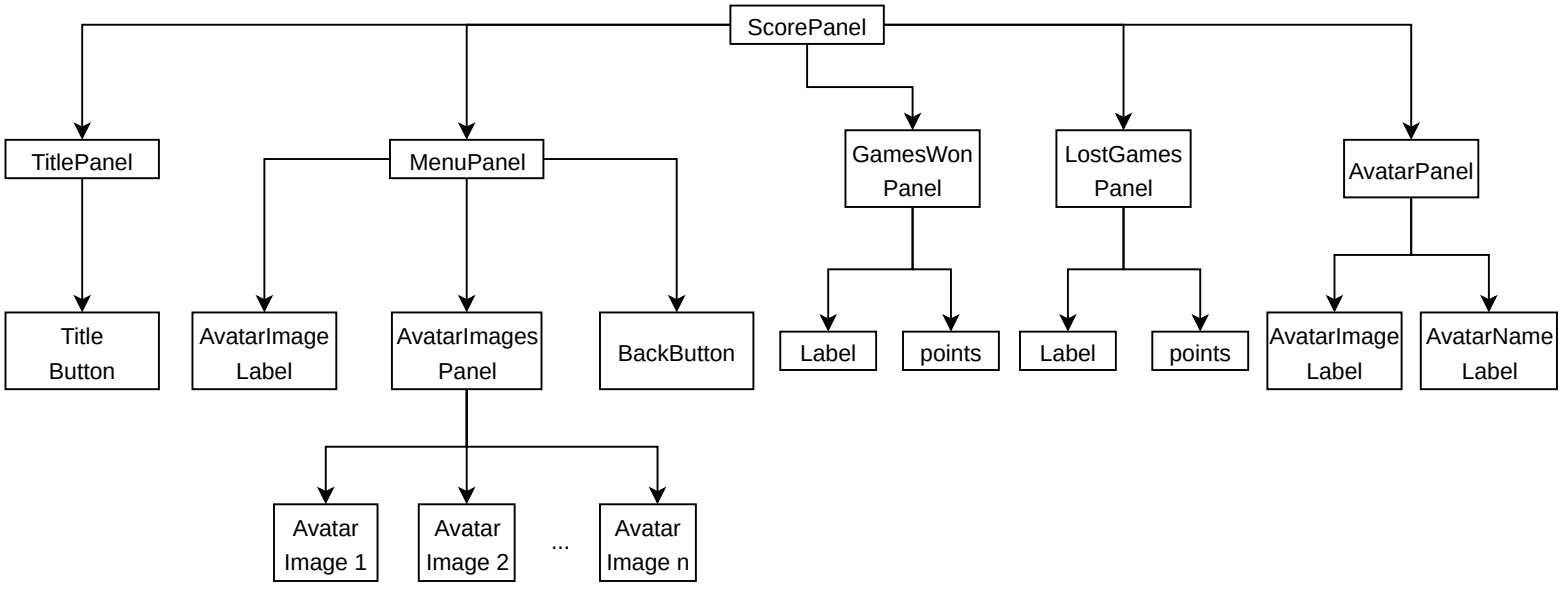

AccessPanel
TitlePanel
MenuPanel
Title Button
CreateAnAccount Button
ContinueWithoutAnAccount Button
LogIn Button
Exit Button

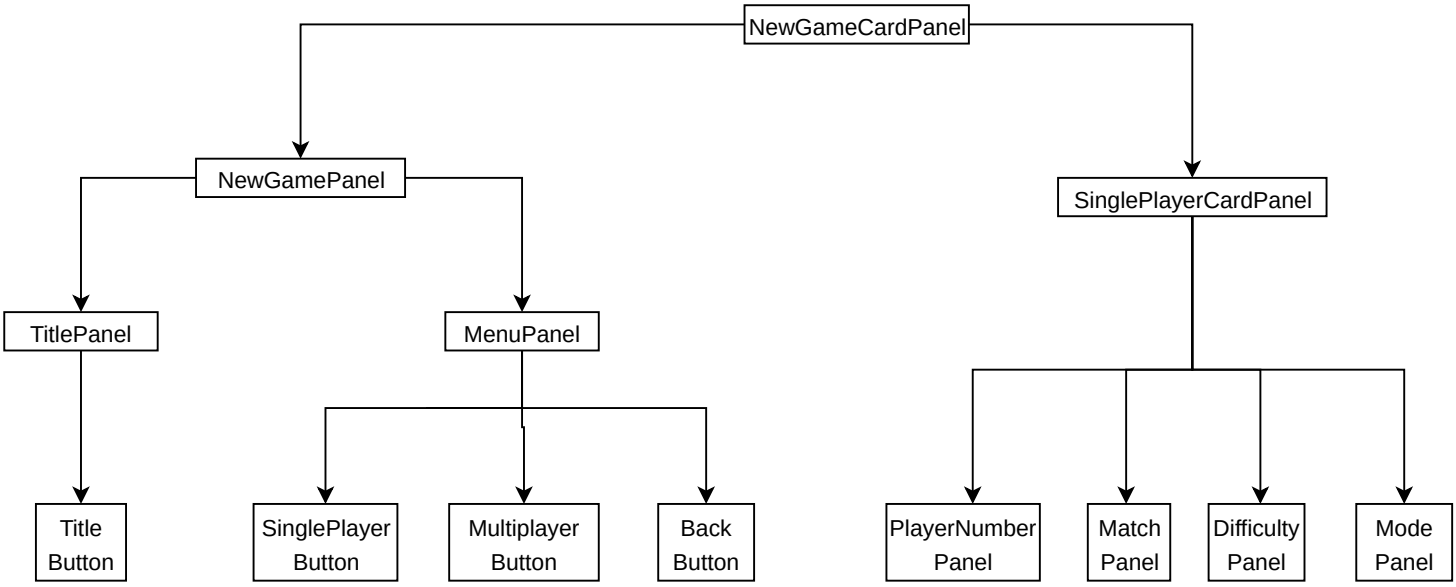

LogInPanel
TitlePanel
MenuPanel
Title
Button
JScrollPane
Back
Button
ProfilesPanel
Profile
Button 1
Profile
Button 2
...
Profile
Button n

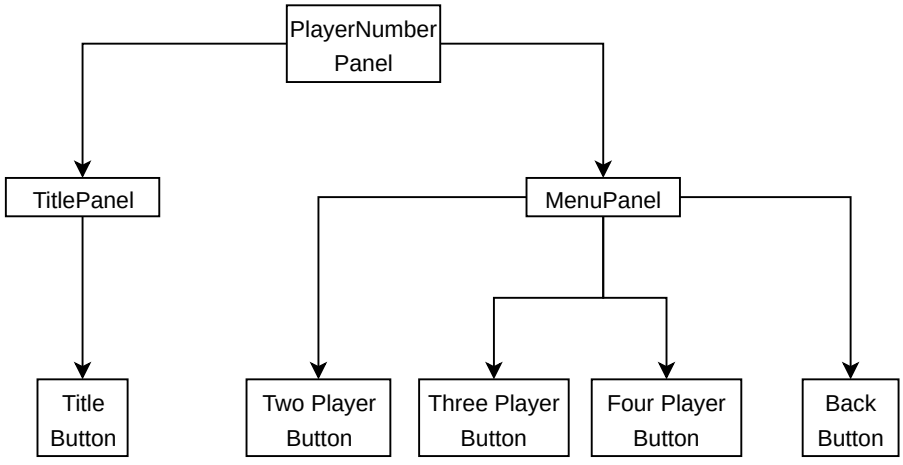

RegistrationPanel
TitlePanel
MenuPanel
Title
Button
ProfileNamePanel
NamePanel
LastNamePanel
AgePanel
Back
Button
Confirm
Button
Label
TextField
Label
TextField
Label
TextField
Label
TextField

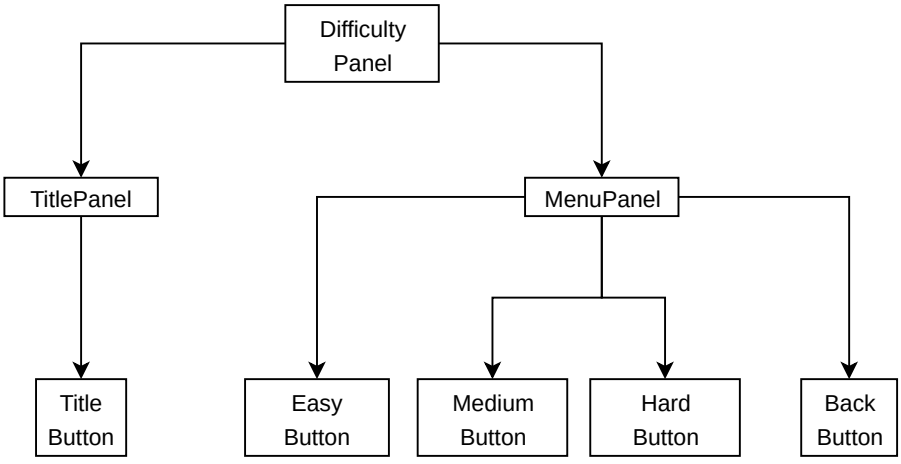

MainPanel
TitlePanel
Left GIF label
Right GIF label
MenuPanel
Log Out Button
Label AvatarPanel
AvatarPanel
Title Button
NewGame Button
Options Button
Score Button
Exit Button
AvatarImage Label
AvatarName Label

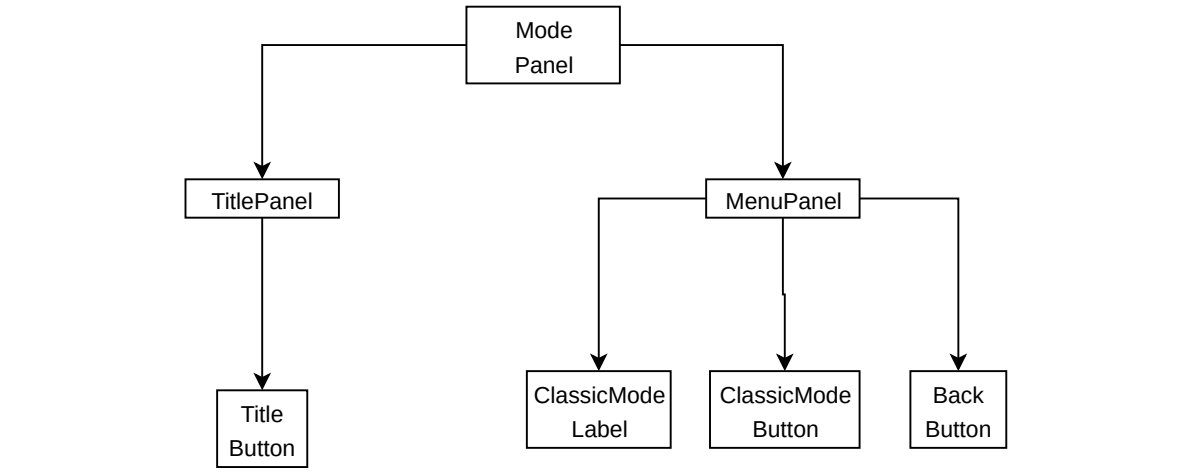

OptionsPanel
TitlePanel
MenuPanel
Title Button
Music ToggleButton
Effect ToggleButton
Fullscreen ToggleButton
Back Button

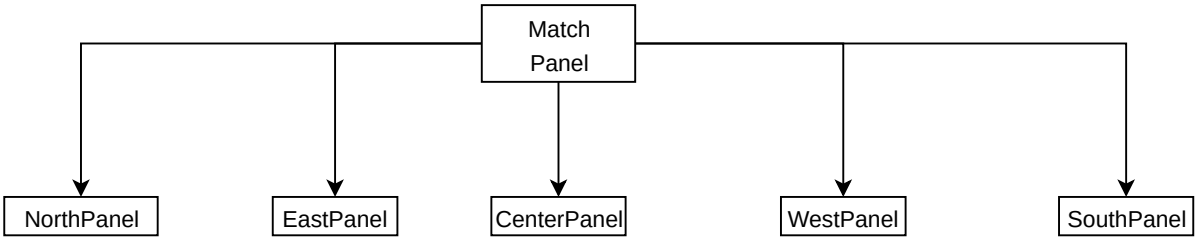

ScorePanel
TitlePanel
MenuPanel
GamesWon Panel
LostGames Panel
AvatarPanel
Title Button
AvatarImage Label
AvatarImages Panel
BackButton
Label
points
Label
points
AvatarImage Label
AvatarName Label
Avatar Image 1
Avatar Image 2
...
Avatar Image n

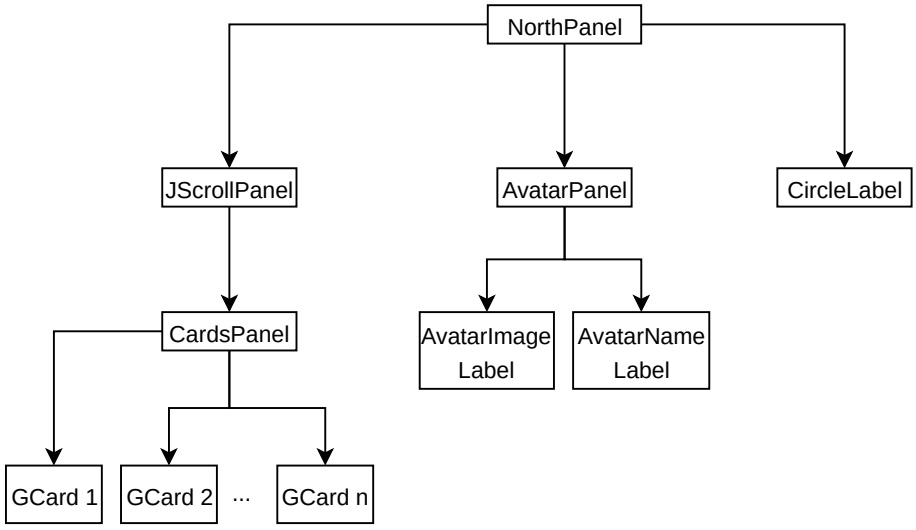

NewGameCardPanel
NewGamePanel
SinglePlayerCardPanel
TitlePanel
MenuPanel
Title Button
SinglePlayer Button
Multiplayer Button
Back Button
PlayerNumber Panel
Match Panel
Difficulty Panel
Mode Panel

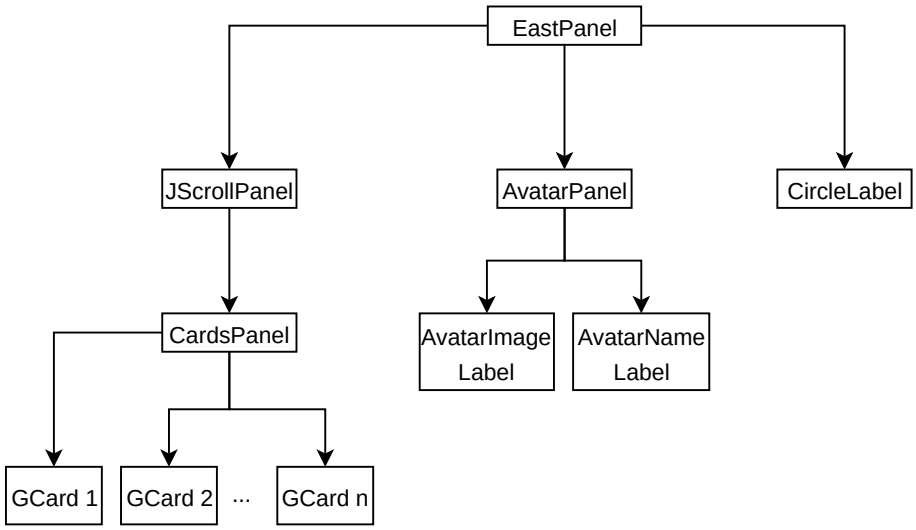

PlayerNumber Panel
TitlePanel
MenuPanel
Title Button
Two Player Button
Three Player Button
Four Player Button
Back Button

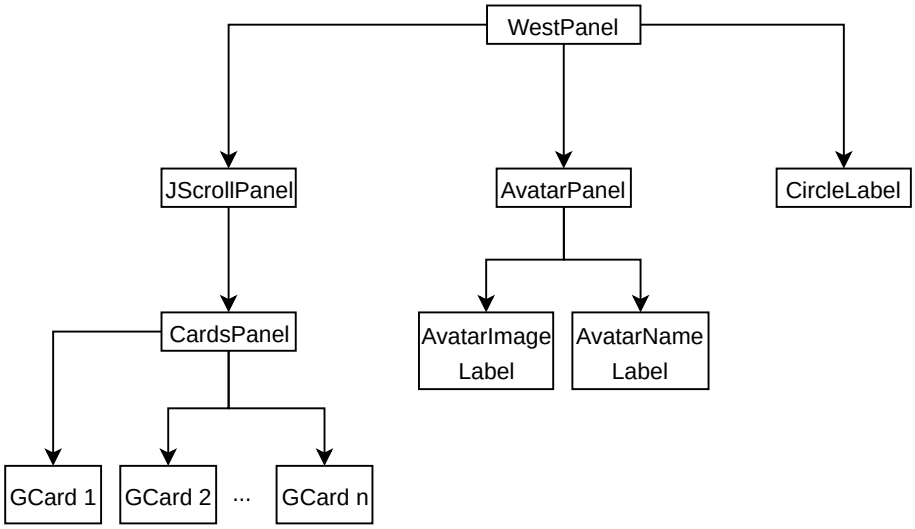

Difficulty Panel
TitlePanel
MenuPanel
Title Button
Easy Button
Medium Button
Hard Button
Back Button

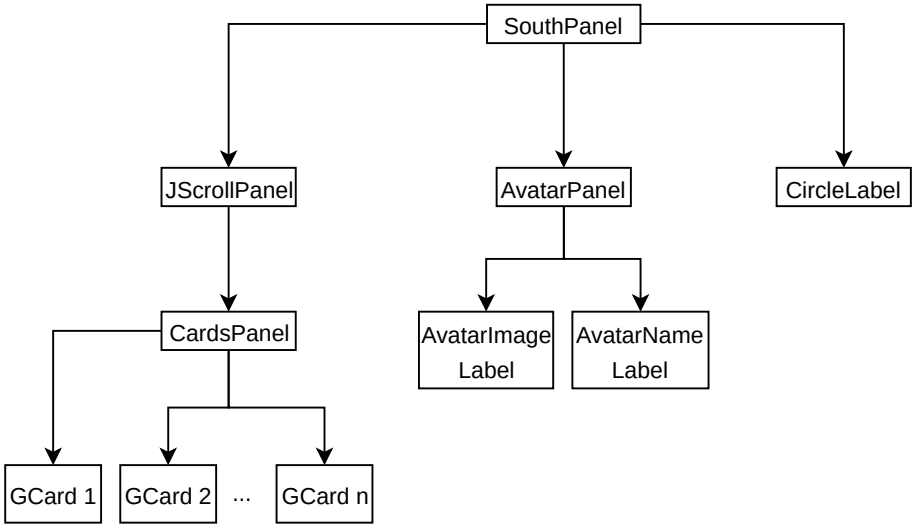

Mode Panel
TitlePanel
MenuPanel
Title Button
ClassicMode Label
ClassicMode Button
Back Button

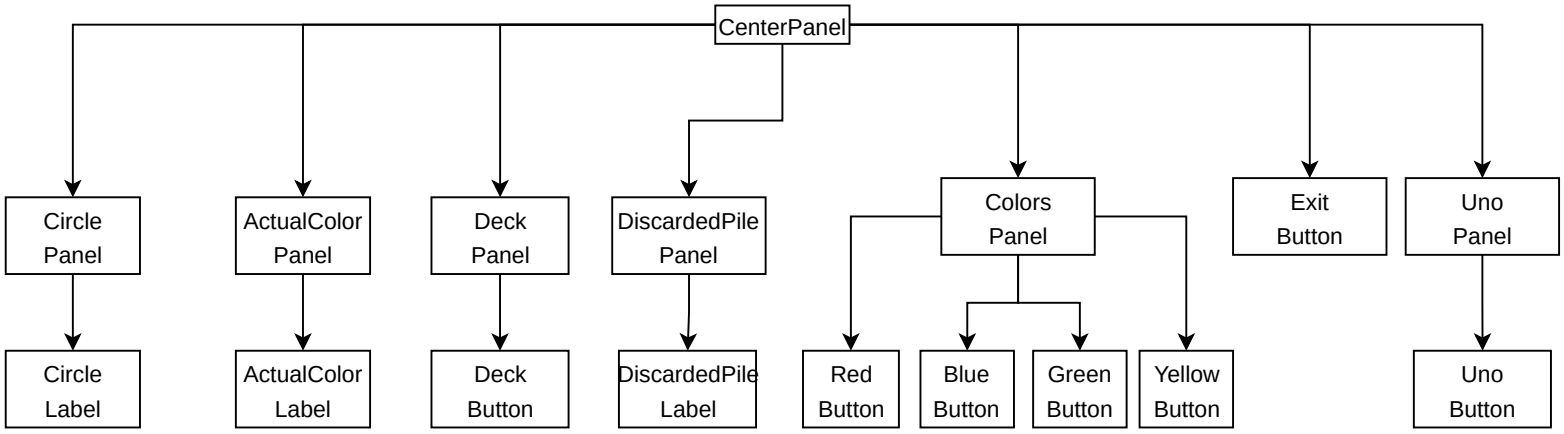

Match Panel
NorthPanel
EastPanel
CenterPanel
WestPanel
SouthPanel


NorthPanel
JScrollPanel
AvatarPanel
CircleLabel
CardsPanel
AvatarImage Label
AvatarName Label
GCard 1
GCard 2
...
GCard n


EastPanel
JScrollPanel
AvatarPanel
CircleLabel
CardsPanel
AvatarImage Label
AvatarName Label
GCard 1
GCard 2
...
GCard n


WestPanel
JScrollPanel
AvatarPanel
CircleLabel
CardsPanel
AvatarImage Label
AvatarName Label
GCard 1
GCard 2
...
GCard n


SouthPanel
JScrollPanel
AvatarPanel
CircleLabel
CardsPanel
AvatarImage Label
AvatarName Label
GCard 1
GCard 2
...
GCard n


CenterPanel
Circle Panel
ActualColor Panel
Deck Panel
DiscardedPile Panel
Colors Panel
Exit Button
Uno Panel
Circle Label
ActualColor Label
Deck Button
DiscardedPile Label
Red Button
Blue Button
Green Button
Yellow Button
Uno Button