

Progetto testing e verifica del software

ASMETA

Funzionamento del gioco

Il programma si presenta come una rivisitazione del gioco del 7 $\frac{1}{2}$.

Nel sistema sono presenti nove carte:

{ASSO | DUE | TRE | QUATTRO | CINQUE | SEI | SETTE | REGINA | RE}

Ad ognuna di queste carte è assegnato un valore da uno a sette, mentre le figure valgono $\frac{1}{2}$ punto.

Ad ogni step della macchina l'utente può decidere se pescare oppure no ed il PC decide di conseguenza se pescare.

Ad ogni pescata la macchina assegna ad ogni giocatore una carta che viene sommata a quella attualmente in mano.

Le casistiche di vittoria sono le seguenti:

- Avere in mano esattamente 7 $\frac{1}{2}$;
- Avere in mano meno di 7 $\frac{1}{2}$ ma l'altro utente ha in mano più di 7 $\frac{1}{2}$.










Il PC pesca solo nel caso in cui abbia in mano meno dell'utente (ma comunque meno di 7 $\frac{1}{2}$).

Se l'utente decide di non pescare, avendo però in mano meno del PC, allora il PC vince la giocata, in quanto è come se l'utente si fosse ritirato.

I giocatori (utente e PC) inizialmente hanno 5 euro a testa, ad ogni giocata vinta viene aggiunto un euro ed a ogni giocata persa ne viene sottratto uno.

Il gioco termina quando un giocatore non ha più soldi.

Esempio di giocata:

	Type	Functions	State 0	State 1	State 2	State 3
<input type="checkbox"/> 	M	pescaUtente	true	true	true	
<input type="checkbox"/> 	C	soldi(UTENTE)	5	5	5	6
<input type="checkbox"/> 	C	soldi(PC)	5	5	5	4
<input type="checkbox"/> 	C	mano(PC)		2.0	2.0	0.0
<input type="checkbox"/> 	C	vincitore		NONDEFINITO	NONDEFINITO	WIN_UTENTE
<input type="checkbox"/> 	C	pescaPC		true	false	true
<input type="checkbox"/> 	C	mano(UTENTE)		0.5	6.5	0.0
<input type="checkbox"/> 	C	pescata(PC)		DUE	DUE	SEI
<input type="checkbox"/> 	C	pescata(UTENTE)		RE	SEI	REGINA

Dove:

- *Stato 0*: Sia utente che PC hanno 5 euro;
- *Stato 1*: L'utente ha pescato il RE ($\frac{1}{2}$ punto) mentre il PC ha pescato il DUE, il vincitore non è ancora definito;
- *Stato 2*: L'utente ha pescato il SEI mentre il PC in questo caso non ha pescato in quanto nel turno precedente aveva in mano più punti dell'utente.
Quindi ora hanno in mano rispettivamente 6.5 punti e 2 punti, il vincitore non è ancora definito;
- *Stato 3*: L'utente ha pescato la REGINA ($\frac{1}{2}$ punto) mentre il PC ha pescato il SEI.
Quindi ora hanno in mano rispettivamente 7 punti e 8 punti, il vincitore della giocata è quindi l'utente.
I soldi dell'utente sono quindi aumentati mentre quelli del PC sono diminuiti.
I punti vengono azzerati immediatamente per la giocata successiva.

Esempio di partita completa:

	Type	Functions	State 11	State 12	State 13	State 14	State 15	State 16
	M	pescaUtente	true	true	true	true	true	true
	C	soldi(UTENTE)	1	1	1	0	0	0
	C	soldi(PC)	9	9	9	10	10	10
	C	mano(PC)	0.0	7.0	7.0	0.0	0.0	0.0
	C	vincitore	WIN_PC	NONDEFINITO	NONDEFINITO	WIN_PC	WIN_PC	WIN_PC
	C	pescaPC	true	true	false	false	false	false
	C	mano(UTENTE)	0.0	1.0	5.0	0.0	0.0	0.0
	C	pescata(PC)	DUE	SETTE	SETTE	SETTE	SETTE	SETTE
	C	pescata(UTENTE)	TRE	ASSO	QUATTRO	CINQUE	CINQUE	CINQUE

Dallo stato 14 in poi l'utente non ha più abbastanza soldi per giocare, la partita termina quindi con vincitore il PC, ovvero il vincitore dell'ultima giocata.

Scenari AsmetaV

Per creare degli scenari il codice del programma originale è stato modificato togliendo il choose durante la pesca. In questa versione le pesche sono diventate variabili monitorate dall'utente.

Gli scenari creati sono i seguenti:

- *Vincita Utente Base*: Un'unica giocata formata da tre pesche dove alla fine l'utente vince. Ad ogni pesca viene controllato il valore della mano dell'utente e del PC ed infine se i soldi sono stati modificati correttamente.

```
check succeeded: pescaPC = true
check succeeded: mano(PC) = 5.0
check succeeded: mano(UTENTE) = 4.0
check succeeded: vincitore = NONDEFINITO
check succeeded: pescaPC = false
check succeeded: mano(PC) = 5.0
check succeeded: mano(UTENTE) = 7.0
check succeeded: vincitore = NONDEFINITO
check succeeded: soldi(UTENTE) = 6
check succeeded: soldi(PC) = 4
check succeeded: vincitore = WIN_UTENTE
```

- *Vincita Completa:* Viene impostato `pescataUtente=TRE` e `pescataPC=QUATTRO`, eseguendo poi gli step necessari (tramite `step until`) affinché i soldi del PC siano 10. Viene quindi controllato se il PC ha vinto e se l'utente non ha più soldi.

```
check succeeded: vincitore = WIN_PC
check succeeded: soldi(PC) = 10
check succeeded: soldi(UTENTE) = 0
```

- *Giocata con pareggio:* In questo caso sia l'utente che il PC pescano le stesse carte. Viene controllato se alla fine della giocata il vincitore è ancora non definito ed i soldi di entrambi i giocatori invariati.

```
check succeeded: vincitore = NONDEFINITO
check succeeded: soldi(UTENTE) = 5
check succeeded: soldi(PC) = 5
```

Specifiche

Per creare e testare alcune specifiche del programma, sono stati necessari ulteriori accorgimenti. Oltre ad inserire manualmente il valore delle pescate di utente e PC, il dominio delle carte è stato modificato da reale ad intero, essendo il dominio dei reali non supportato da NuSMV.

A causa di questa modifica, sono state eliminate le figure (RE e REGINA) dal gioco, spostando il limite di vittoria da $7 \frac{1}{2}$ a 8.

Effettuate queste modifiche, le specifiche create sono le seguenti:

- La somma dei soldi di utente e PC dovrà sempre essere uguale a 10
`CTLSPEC ag((soldi(UTENTE) + soldi(PC)) = 10)`
- Esiste almeno un percorso in cui in uno stato l'utente vincerà una giocata
`CTLSPEC ef(vincitore = WIN_UTENTE)`
- Non sempre utente o il PC possono vincere una giocata
`CTLSPEC af(vincitore = NONDEFINITO)`
- Se l'utente non pesca, allora o il PC pesca oppure qualcuno vince la giocata
`CTLSPEC ag(not(pescaUtente) implies ax(vincitore=WIN_PC or vincitore=WIN_UTENTE or pescaPC))`
- Il PC pesca finché ha una mano minore o uguale a quella dell'utente
`CTLSPEC a(pescaPC, mano(PC) >= mano(UTENTE))`
- Il PC pesca finché non avrà vinto la partita oppure la sua mano è più alta dell'utente (non è detto che ciò accada)
`CTLSPEC aw(pescaPC, (vincitore = WIN_PC or mano(PC) >= mano(UTENTE)))`
- Ogni volta che il pc vince, nel prossimo stato avrà di sicuro almeno un euro
`CTLSPEC ag(vincitore=WIN_PC implies ax(soldi(PC)>0))`
- Esiste sempre la possibilità che la partita non sia possibile (qualcuno ha finito i soldi)
`CTLSPEC ef(not(possibilePartita))`

Queste specifiche sono tutte corrette, ciò viene confermato anche dalla loro esecuzione.

```
specification AG soldi(UTENTE) + soldi(PC) = 10 is true
specification EF !possibilePartita is true
specification AG (soldi(UTENTE) = 0 -> soldi(PC) = 10) is true
specification AG (soldi(PC) = 0 -> soldi(UTENTE) = 10) is true
specification EF vincitore = WIN_UTENTE is true
specification AF vincitore = NONDEFINITO is true
specification A [ pescaPC U mano(PC) >= mano(UTENTE) ] is true
specification AG (vincitore = WIN_PC -> AX soldi(PC) > 0) is true
specification AG (!pescaUtente -> AX (pescaPC | (vincitore = WIN_UTENTE | vincitore = WIN_PC))) is true
specification !E [ !(mano(PC) >= mano(UTENTE) | vincitore = WIN_PC) U !(pescaPC | (mano(PC) >= mano(UTENTE) | vincitore = WIN_PC)) ] is true
```

È stata generata anche una specifica non corretta, ovvero:

- Ogni volta che l'utente pesca allora pesca anche il PC

CTLSPEC `ag(pescaUtente implies pescaPC)`

Questa specifica è ovviamente errata, il controesempio generato per dimostrarne la falsità è il seguente:

```
-> State: 1.1 <-
  pescaUtente = false
  pescaPC = true
  soldi(UTENTE) = 5
  soldi(PC) = 5
  mano(PC) = 0
  mano(UTENTE) = 0
  pescata(UTENTE) = ASSO
  pescata(PC) = ASSO
  valoreCarte(TRE) = 3
  valoreCarte(SETTE) = 7
  valoreCarte(SEI) = 6
  valoreCarte(QUATTRO) = 4
  valoreCarte(DUE) = 2
  valoreCarte(CINQUE) = 5
  valoreCarte(ASSO) = 1
  possibilePartita = true

-> State: 1.2 <-
  pescaUtente = true
  soldi(UTENTE) = 4
  soldi(PC) = 6
  pescata(UTENTE) = DUE
  pescata(PC) = SEI
-> State: 1.3 <-
  mano(PC) = 6
  mano(UTENTE) = 2
  pescata(UTENTE) = TRE
  pescata(PC) = ASSO
-> State: 1.4 <-
  pescaPC = false
  mano(UTENTE) = 5
  pescata(UTENTE) = ASSO
```

Come si può notare, nello stato 3, il PC ha in mano 6, mentre l'utente ha in mano 2.

L'utente deciderà quindi di pescare (altrimenti avrebbe perso), ma il PC non pesca in quanto se l'utente pescasse il 7 o decidesse di non pescare, farebbe vincere il PC.

MBT – ATGT

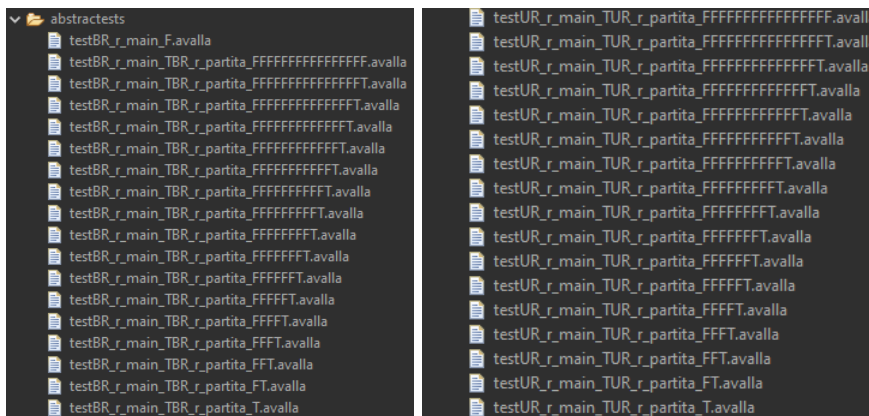
Per poter provare ad utilizzare ATGT, è stato necessario modificare il programma “principale”, essendo che ATGT dà problemi con l'utilizzo di interi, accettando solo enumerativi.

Il programma è stato quindi modificato, arrivando alla seguente versione del gioco:

L'utente pesca una carta fintanto che non riesce ad arrivare a otto (vittoria) oppure se supera l'otto (sconfitta).

Questo semplice programma è stato implementato con una serie di condizioni, le quali controllano il valore delle carte che si hanno in mano e, nel caso fosse possibile, aggiungendo la carta pescata alla mano.

Una volta implementato ciò, grazie all'utilizzo di ATGT vengono creati diversi scenari, rappresentati di seguito.



Come si può notare, il numero di scenari generati non è casuale ma ogni scenario prova uno specifico percorso.

Ecco alcuni esempi:

1. Un primo esempio è il primo scenario testa il caso in cui il primo if è falso, ovvero vittoria non è ancora definita.

Ciò viene generato nel seguente modo:

```
scenario testBR_r_main_F

load ../../SetteEMezzoATGT.asm

//// test name testBR_r_main_F
//// generated for (test goal): BR_r_main_F: vittoria != NONDEFINITA
check mano = TRE;
check vittoria = NONDEFINITA;
set pescataIniziale := TRE;
set pescata := ASSO;
step
check mano = QUATTRO;
set pescata := undef;
step
check vittoria = PERSA;
step
```

```
main rule r_main =
  if(vittoria = NONDEFINITA) then
    r_partita[]
  endif
```

Come si nota, vittoria = NONDEFINITA è il primo controllo del codice

2. Un ulteriore esempio è il seguente, dove lo scenario arriva ad un'esecuzione dove tutti gli if sono falsi tranne l'ultimo, ovvero il caso in cui l'utente vince.

Ciò viene eseguito nel seguente modo:

```
scenario testBR_r_main_TBR_r_partita_FFFFFFFFFFFFFFFFT

load ../../SetteEMezzoATGT.asm

//// test name test@BR_r_main_TBR_r_partita_FFFFFFFFFFFFFFFFT
//// generated for (test goal): BR_r_main_TBR_r_partita_FFFF
check mano = TRE;
check vittoria = NONDEFINITA;
set pescataIniziale := TRE;
set pescata := ASSO;
step
check mano = QUATTRO;
set pescata := QUATTRO;
step
```

Tutti gli scenari generati, una volta eseguiti, daranno in output tutti i check corretti, a dimostrazione della corretta generazione degli scenari.

Traduzione casi di test

Avendo ora gli scenari generati da ATGT sul modello astratto, è possibile tradurre questi scenari in casi di test reali su un programma Java.

Per far ciò è stato quindi creato un piccolo programma Java (chiamato GiocoATGT), dove il funzionamento è lo stesso illustrato per il programma in asmeta.

In questo caso, per semplicità, non sono stati utilizzati degli enumerativi ma solo delle variabili intere, dove l'utente inserendo la pescata può vincere perdere o continuare a giocare.

Il metodo principale di questo programma è il seguente:

```
//1=vinto 0=continua a giocare -1=perso
public int giocata(int pescata){
    if(partitaPersa())
        return -1;
    if(pescata <= 7 && pescata >= 1)
        mano = mano + pescata;
    if(mano == 8)
        return 1;
    if(mano > 8)
        return -1;
    return 0;
}
```

La cosa migliore da fare sarebbe ora riportare tutti gli scenari generati da ATGT in casi di test JUnit.

Per semplicità vengono creati solo due casi di test (gli stessi riportati in precedenza in scenari avalla).

Quindi gli esempi sono rispettivamente:

1. Test in cui il giocatore può continuare a giocare

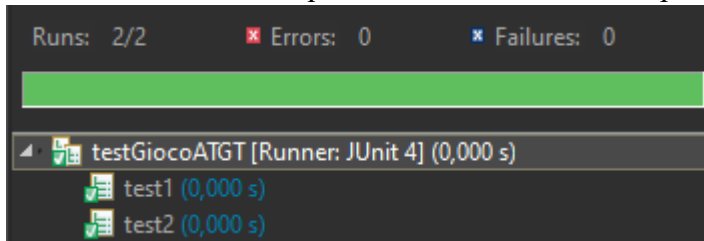
```
@Test
public void test1() {
    int result = 0;
    GiocoATGT g = new GiocoATGT();
    result = g.giocata(3);
    assertEquals(0, result);
    result = g.giocata(4);
    //Controllo che posso continuare a giocare
    assertEquals(0, result);
}
```

2. Test in cui il giocatore vince

```
@Test
public void test2() {
    int result = 0;
    GiocoATGT g = new GiocoATGT();
    result = g.giocata(3);
    assertEquals(0, result);
    result = g.giocata(1);
    assertEquals(0, result);
    result = g.giocata(4);
    assertEquals(1, result);
}
```

Come si nota, sono gli stessi casi di test riportati come scenari avalla in precedenza.

Ovviamente, entrambi questi test daranno risultato positivo, ovvero:



Questo era quindi un piccolo esempio di cui il Model Based Testing possa essere applicato a programmi Java tramite l'utilizzo di avalla e JUnit.

JAVA

Funzionamento del gioco

Anche in questo caso, il gioco si presenta come una rivisitazione del gioco del 7 ½.

Durante la creazione del gioco è necessario inserire il numero di giocatori, ciò inizierà le seguenti variabili:

- `Mano[]` -> Vettore dove ad ogni giocatore è assegnato il valore delle carte che ha in mano;
- `Vincitori[]` -> Vettore che indica lo stato dei giocatori (-1=Ha perso, 0=Può giocare, 1=Ha vinto);

Un giocatore, come nel caso di Asmeta, vince se con la propria mano arriva esattamente a otto (vengono utilizzati gli interi per semplicità), oppure se tutti gli altri superano l'otto con le proprie carte.

Per l'implementazione sono stati creati i seguenti metodi:

- *Giocata*: Prende in input un vettore di pescate (lungo quanto il numero di giocatori), aggiunge le pescate alla mano dei giocatori che possono ancora giocare ed invoca i metodi `primoControllo` e `secondoControllo`. Restituisce in output -1 se non ci sono stati vincitori, altrimenti il numero dell'ultimo giocatore ad aver vinto.
- *primoControllo*: Controlla, dopo le varie pescate chi e se qualcuno ha vinto/perso, controllando solo il numero che si ha in mano;
- *secondoControllo*: Ulteriore controllo per verificare chi e se qualcuno ha vinto/perso, considerando il caso in cui in mano si ha un numero minore di otto ma tutti gli altri giocatori abbiano un numero maggiore di otto.
- *Print*: Stampa lo stato del gioco, indicando per ogni giocatore se può giocare, se ha vinto oppure se ha perso.

Durante il gioco sono gestite tutte le casistiche particolari come, per esempio, la vincita di più giocatori (se entrambi sono arrivati a otto) oppure la perdita di tutti i giocatori (se tutti hanno superato l'otto).

Main di prova

Per capire meglio il funzionamento del programma è stato implementato un metodo `main` in cui viene creato l'oggetto e vengono invocati alcuni metodi.

Di seguito viene riportato una parte del `main`, con il relativo output:

```
Gioco g1 = new Gioco(3);
int[] pescate1 = {1,2,3};

g1.giocata(pescate1);
g1.print();
g1.giocata(pescate1);
g1.print();
g1.giocata(pescate1);
g1.print();
g1.giocata(pescate1);
g1.print();
```

Input

```
Giocatore 0 ha in mano: 1-> PUO' GIOCARE
Giocatore 1 ha in mano: 2-> PUO' GIOCARE
Giocatore 2 ha in mano: 3-> PUO' GIOCARE

Giocatore 0 ha in mano: 2-> PUO' GIOCARE
Giocatore 1 ha in mano: 4-> PUO' GIOCARE
Giocatore 2 ha in mano: 6-> PUO' GIOCARE

Giocatore 0 ha in mano: 3-> PUO' GIOCARE
Giocatore 1 ha in mano: 6-> PUO' GIOCARE
Giocatore 2 ha in mano: 9-> LOSE!

Giocatore 0 ha in mano: 4-> LOSE!
Giocatore 1 ha in mano: 8-> WIN!
Giocatore 2 ha in mano: 9-> LOSE!
```

Output

JUNIT & Codecover

I test JUnit generati permettono di ottenere la copertura di istruzioni, branch, condizioni e MCDC.

Durante tutti i test viene utilizzato il metodo assertEquals per testarne la veridicità, cercando di creare il numero minimo di casi di test.

Di seguito vengono illustrati nello specifico i test creati:

Test delle istruzioni

I casi di test generati sono i seguenti:

- *TC1*: Copre il caso in cui qualcuno riesce a vincere per il fatto che tutti gli altri giocatori superano il numero otto.
Implementazione: `Pescate=[5,3,5]`, invocato due volte il metodo `giocata` ed il metodo `print`.
- *TC2*: Copre il caso in cui qualcuno vince perché arriva ad otto preciso.
Implementazione: `Pescate=[1,1,4]`, invocato due volte il metodo `giocata` ed il metodo `print`.

Grazie a questi test è possibile ottenere la completa copertura delle istruzioni, come dimostrato con l'utilizzo di Codecover:

Name	Statement	Branch	Loop	Term
GiocoCarte	100,0 %	88,9 %	?	93,8 %
Gioco	100,0 %	88,9 %	?	93,8 %
Gioco	100,0 %	—	?	100,0 %
giocata	100,0 %	50,0 %	?	66,7 %
primoControllo	100,0 %	100,0 %	?	100,0 %
print	100,0 %	100,0 %	?	100,0 %
secondoControllo	100,0 %	100,0 %	?	100,0 %

Test dei Branch

Per avere una copertura completa dei branch è stato necessario aggiungere alcuni accorgimenti ed un nuovo caso di test ai casi di test TC1 e TC2 creati in precedenza.

Le modifiche effettuate sono le seguenti:

- Al caso di test TC2 è stata aggiunta un'ulteriore chiamata al metodo “`giocata`”, per far sì di provare il metodo nel caso in cui qualcuno abbia già vinto;
- Inserimento di *TC3*: Copre il caso in cui un solo giocatore perde, facendo in modo che lui non possa più pescare.
Implementazione: `Pescate=[1,1,5]`, invocato tre volte il metodo `giocata`.

Grazie a questi accorgimenti è possibile ottenere la completa copertura dei branch, come dimostrato con l'utilizzo di codecover:

Name	Statement	Branch	Loop	Term
GiocoCarte	100,0 %	100,0 %	?	100,0 %
Gioco	100,0 %	100,0 %	?	100,0 %
Gioco	100,0 %	—	?	100,0 %
giocata	100,0 %	100,0 %	?	100,0 %
primoControllo	100,0 %	100,0 %	?	100,0 %
print	100,0 %	100,0 %	?	100,0 %
secondoControllo	100,0 %	100,0 %	?	100,0 %

Test delle condizioni

Come è possibile notare nel test precedente, sviluppando il test del branch si è già ottenuta la copertura delle condizioni.

Questo è dovuto soprattutto al fatto che nel codice sono presenti pochi if con più di una condizione.

Ciò implica che, facendo risultare tutti gli if sia negativi che negativi almeno una volta, tutte le condizioni vengono già testate.

Test MCDC

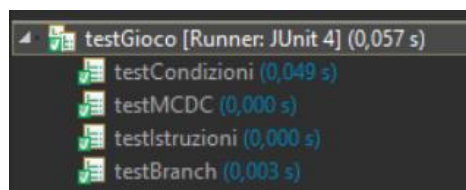
Valgono le stesse considerazioni fatte per il test delle condizioni.

Tutte le condizioni sono già state provate in tutte le combinazioni, un esempio è il seguente:

Class: Gioco		Condition: (mano[i] < 8 AND controllo == 1)				
(mano[i] < 8	AND	controllo == 1)	Result	Test Cases (Number of Executions)
	F	F	x		0	testGioco:testBranch (5) Coverage: 25,0
	T	F	F		0	testGioco:testBranch (7) Coverage: 25,0
	T	T	T		1	testGioco:testBranch (1) Coverage: 50,0

Risultati finali

Eseguendo i test eseguiti, tramite interfaccia JUnit, il risultato ottenuto è il seguente:



Ciò conferma la veridicità di tutti gli assertEquals inseriti nei vari test.

JML

Per quanto riguarda la parte di design by contract, nel programma java sono stati creati diversi contratti, descritti in seguito.

Invarianti

Sono stati creati tre contratti che andranno sempre rispettati durante l'esecuzione (a partire da dopo il costruttore), ovvero:

- Il vettore della “mano” dovrà essere sempre diverso da null

```
//@public invariant mano!=null;
```
- Il vettore dei vincitori dovrà essere sempre diverso da null

```
//@public invariant vincitori!=null;
```
- Il vettore dei vincitori dovrà sempre contenere un numero compreso tra -1 e 1

```
//@public invariant (\forall int x; 0<=x && x<vincitori.length; vincitori[x]>=-1 && vincitori[x]<=1);
```

Costruttore

Per il costruttore i contratti creati sono i seguenti:

- Il gioco deve avere almeno un giocatore, ma non più di mille, per evitare problemi di overflow

```
//@requires giocatori > 0 && giocatori < 1000;
```
- In output devono essere rispettati gli stessi contratti inseriti nelle invarianti in quanto le condizioni degli invarianti valgono da dopo il costruttore

```
//@ensures mano!= null;  
//@ensures vincitori!=null;
```
- La variabile “vittoria” deve essere impostato a -1 (nessuno ha vinto)

```
//@ensures vittoria == -1;
```
- Viene controllato che i vettori “mano” e “vincitori” siano stati creati della giusta lunghezza

```
//@ensures (mano.length == giocatori) && (vincitori.length == giocatori);
```
- Viene controllato che i vettori “mano” e “vincitori” siano stati inizializzati correttamente

```
//@ensures (\forall int x; 0<=x && x<giocatori; mano[x]==0 && vincitori[x]==0);
```

Metodo giocata

Per il metodo giocata i contratti creati sono i seguenti:

- Viene impostato `diverges true`, in modo tale da non far verificare che il programma termini

```
//@diverges true;
```
- Richiedo che il vettore delle pescate sia lungo uguale al numero di giocatori

```
//@requires pescate.length == mano.length;
```
- Viene richiesto che il vettore delle pescate contenga numeri compresi tra uno e sette

```
//@requires (\forall int x; 0<=x && x<pescate.length; pescate[x]>0 && pescate[x]<8);
```
- Viene controllato che la mano più la pescata non superi il numero 20, in quanto sarebbe il caso peggiore (aver già perso arrivando a 14 e voler pescare ancora 7)

```
//@requires (\forall int x; 0<=x && x<mano.length; (mano[x]+pescate[x])>=0 && (mano[x]+pescate[x])<=20);
```
- Viene controllato che la variabile vittoria sia compresa tra -1 (non ha vinto nessuno) ed il numero dei giocatori

```
//@ensures (vittoria >=-1) && (vittoria<=mano.length);
```

- Viene controllato che nessun giocatore in mano abbia più di 14 (caso peggiore)

```
//@ensures (\forallall int x; 0<=x && x<mano.length; mano[x]>=0 && mano[x]<=14);
```
- Viene controllato che il risultato (e quindi anche la variabile vittoria) sia un numero compreso tra -1 ed il numero di giocatori

```
//@ensures (\result >=-1) && (\result<=mano.length);
```
- Alcune di questi contratti sono stati inseriti anche all'interno del codice come `loop_invariant` in tal modo da verificare che valgano sia all'inizio che alla fine del ciclo

```
//@loop_invariant i>=0 && i<=mano.length;  
//@loop_invariant (\forallall int x; 0<=x && x<i; mano[x]>=0 && mano[x]<=20);
```
- Dopo aver eseguito i metodi primo e secondo controllo viene controllato lo stato della variabile vittoria tramite un `assert`

```
primoControllo();  
//@assert (vittoria >=-1) && (vittoria<=mano.length);  
secondoControllo();  
//@assert (vittoria >=-1) && (vittoria<=mano.length);
```

Metodo primoControllo

- Viene impostato `diverges true`, in modo tale da non far verificare che il programma termini

```
//@diverges true;
```
- Viene controllato che alla fine dell'esecuzione tutti i giocatori hanno vinto, perso o possono continuare a giocare

```
//@ensures (\forallall int x; 0<=x && x<vincitori.length; (vincitori[x]==-1) || (vincitori[x]==0) || (vincitori[x]==1));
```
- Viene controllato tramite `loop_invariant` che chi ha in mano più di otto ha perso

```
//@loop_invariant (\forallall int x; 0<=x && x<i; (mano[x]>8) ==> (vincitori[x]==-1));
```
- Viene controllato tramite `loop_invariant` che chi ha in mano esattamente otto ha vinto

```
//@loop_invariant (\forallall int x; 0<=x && x<i; (mano[x]==8) ==> (vincitori[x]==1));
```

Metodo secondoControllo

- Viene controllato che alla fine dell'esecuzione tutti i giocatori hanno vinto, perso o possono continuare a giocare

```
//@ensures (\forallall int x; 0<=x && x<mano.length; (vincitori[x]==-1) || (vincitori[x]==0) || (vincitori[x]==1));
```

Metodo print

Per questo metodo non sono stati generati contratti JML in quanto non necessari.

Verifica dei contratti

Dopo aver creato i contratti riportati, è possibile controllare il funzionamento cercando di violarne qualcuno nel main.

Per esempio, se volessimo violare il contratto tale per cui le pescate dei giocatori debbano essere comprese tra uno e sette inserendo una pescata di 22, si otterrebbe il seguente errore:

```
Gioco g1 = new Gioco(3);  
int[] pescate1 = {1,22,3};  
g1.giocata(pescate1);
```

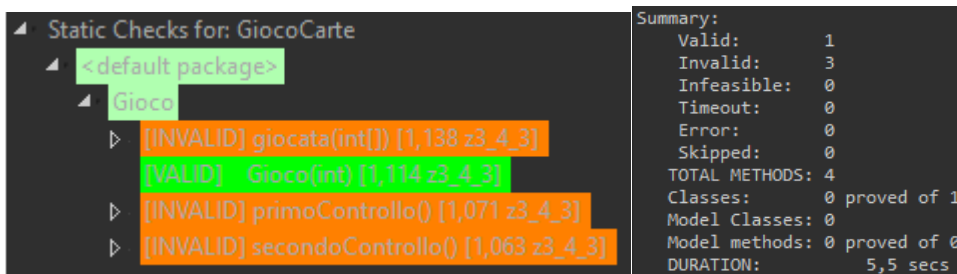
```
C:\Documenti\UniBG\Materie\Testing\Progetto_Testing\JUnit-JML\GiocoCarte\src\Main.java:32: JML precondition is false  
g1.giocata(pescate1);  
      ^
```

OpenJML

A questo punto si potrebbero validare i contratti tramite il solver z3_4_3.

Il metodo print per questa esecuzione non viene controllato per problemi di efficienza ed in quanto non serve che venga controllato.

Eseguendo il solver con i contratti creati in precedenza, si ottiene il seguente output



Come si può notare solo il metodo Gioco (il costruttore) risulta validato.

Analizzando gli altri metodi vengono di seguito riportati alcuni esempi riportati nell'output del solver:

- *Esempio 1*

```
C:\Documenti\UniBG\Materie\Testing\Progetto_Testing\JUnit-JML\GiocoCarte\src\Gioco.java:73:  assert Assert (vittoria >= -1) && (vittoria <= mano.length);
VALUE: vittoria    == ( - 2 )
VALUE: -1         == ( - 1 )
VALUE: vittoria >= -1    == false
VALUE: (vittoria >= -1) == false
VALUE: (vittoria >= -1) && (vittoria <= mano.length) == false
```

In questo caso l'errore è relativo al contratto in cui veniva richiesto il valore di vittoria compreso tra -1 ed il numero di giocatori.

Il solver afferma però che, se vittoria fosse uguale a -2, questo contratto non sarebbe rispettato.

- *Esempio 2*

```
C:\Documenti\UniBG\Materie\Testing\Progetto_Testing\JUnit-JML\GiocoCarte\src\Gioco.java:101:  controllo = controllo + 1
VALUE: controllo    == 2147483647
VALUE: 1            == 1
VALUE: controllo + 1 == ( - 2147483648 )
VALUE: controllo = controllo + 1    == 0
```

In questo caso viene fatto notare come la variabile controllo in caso di overflow viene azzerata se viene sommato uno.

Questo problema della somma con l'overflow è molto comune.

- *Esempio 3*

```
C:\Documenti\UniBG\Materie\Testing\Progetto_Testing\JUnit-JML\GiocoCarte\src\Gioco.java:44:  requires pescate.length == mano.length;
VALUE: pescate.length == 2147483646
VALUE: mano          == REF!val!16
VALUE: mano.length   == 2147483646
VALUE: pescate.length == mano.length    == true
```

In questo caso invece si può notare come il solver sia riuscito a validare un contratto creato.

La chiusura dei contratti tramite openJML in questo caso risulta molto critica in quanto la maggior parte degli errori sono generati a causa di overflow.

Probabilmente si potrebbero provare a sistemare i contratti “ensures” in quanto questi non considerano la presenza dei “requires” nel metodo, utilizzando valori in input qualsiasi.

Queste modifiche non porterebbero comunque alla chiusura completa dei contratti.

Code Inspection

Per quanto riguarda l'ispezione del codice Java implementato, è stata seguita la seguente checklist

1. Specification / Design
 - [✓] Is the functionality described in the specification fully implemented by the code?
 - [✓] Is there any excess functionality in the code but not described in the specification?
2. Initialization and Declarations
 - [✓] Are all local and global variables initialized before use?
 - [✓] Are variables and class members of the correct type and appropriate mode
 - [✓] Are variables declared in the proper scope?
 - [✓] Is a constructor called when a new object is desired?
 - [✓] Are all needed import statements included?
3. Method Calls
 - [✓] Are parameters presented in the correct order?
 - [✓] Are parameters of the proper type for the method being called?
 - [✓] Is the correct method being called, or should it be a different method with a similar name?
 - [✓] Are method return values used properly? Cast to the needed type?
4. Arrays
 - [✓] Are there any off-by-one errors in array indexing?
 - [✓] Can array indexes ever go out-of-bounds?
 - [✓] Is a constructor called when a new array item is desired?
5. Object Comparison
 - [✓] Are all objects (including Strings) compared with "equals" and not "=="?
6. Output Format
 - [✓] Are there any spelling or grammatical errors in displayed output?
 - [✓] Is the output formatted correctly in terms of line stepping and spacing?
7. Computation, Comparisons and Assignments
 - [✓] Check order of computation/evaluation, operator precedence and parenthesizing
 - [✓] Can the denominator of a division ever be zero?
 - [✓] Is integer arithmetic, especially division, ever used inappropriately, causing unexpected truncation/rounding?
 - [✓] Check each condition to be sure the proper relational and logical operators are used.
 - [✓] If the test is an error-check, can the error condition actually be legitimate in some cases?
8. Exceptions
 - [✓] Are all relevant exceptions caught?
9. Flow of Control
 - [✓] In a switch statement is every case terminated by break or return?
 - [✓] Do all switch statements have a default branch?
 - [✓] Check that nested if statements don't have "dangling else" problems.
 - [✓] Are all loops correctly formed, with the appropriate initialization, increment and termination expressions?
 - [✓] Are open-close parentheses and brace pairs properly situated and matched?

Analisi Statica – SpotBugs

Per l'analisi statica del codice, è stato utilizzato SpotBugs, ovvero una diramazione di FindBugs (il quale ormai è in disuso).

Utilizzando SpotBugs nel progetto Java, il sommario ottenuto è il seguente

FindBugsSummary	
timestamp	Mon, 3 Jul 2023 11:12:40 +0200
total_classes	4
referenced_classes	17
total_bugs	2
total_size	188
num_packages	1
java_version	1.8.0-adoptopenjdk
vm_version	25.71-b00
cpu_seconds	1.16
clock_seconds	0.52
peak_mbytes	724.04
alloc_mbytes	1024.00
gc_seconds	0.00
priority_2	2

Come si nota, il tool ha rilevato due bug e zero errori.

Analizzando nello specifico i bug, ciò che si rileva è che entrambi sono bug non rilevanti in quanto sono bug di tipo “NM_CLASS_NAMING_CONVENTION” ovvero causati dal nome delle classi.

Ciò si può evincere anche dalle seguenti bugInstance

BugInstance type=NM_CLASS_NAMING_CONVENTION	
Class classname=testGioco	SourceLine classname=testGioco
BugInstance type=NM_CLASS_NAMING_CONVENTION	
Class classname=testGiocoCT	SourceLine classname=testGiocoCT

Per convenzione, infatti, le classi non dovrebbero chiamarsi come in questo caso testGioco oppure testGiocoCT.

La convenzione per il nome delle classi è che queste dovrebbero essere sostantivi, con la prima lettera di ogni singola parola in maiuscolo (in questo caso, quindi, dovrebbero chiamarsi TestGioco e TestGiocoCT).

Azioni

Rinominando quindi le classi in TestGioco e TestGiocoCT si può notare come non vengano più rilevati bug ed errori:

FindBugsSummary	
timestamp	Mon, 3 Jul 2023 11:31:34 +0200
total_classes	4
referenced_classes	17
total_bugs	0
total_size	188
num_packages	1
java_version	1.8.0-adoptopenjdk
vm_version	25.71-b00
cpu_seconds	0.83
clock_seconds	0.35
peak_mbytes	851.82
alloc_mbytes	1024.00
gc_seconds	0.00

BugCollection version=4.0.0:
Project projectName=GiocoCarte
Errors errors=0
FindBugsSummary timestamp=Mon, 3 Jul 2023 11:31:34 +0200
ClassFeatures
History

MBT - Combinatorial Testing

Per la parte di combinatorial testing, tramite l'utilizzo di CTWedge, si è creata un'astrazione del modello nel seguente modo

```
Model Gioco

Parameters:
  pescataG1: [1 .. 9]
  pescataG2: [1 .. 9]
  pescataG3: [1 .. 9]
  vincitore: {-1 1 2 3}

Constraints:
  # (vincitore == 1) => (((pescataG1 == 8) AND (pescataG2 != 8) AND (pescataG3 != 8)) OR ((pescataG1 <= 8) AND (pescataG2 > 8) AND (pescataG3 > 8))) #
  # (vincitore == 2) => (((pescataG2 == 8) AND (pescataG3 != 8)) OR ((pescataG2 <= 8) AND (pescataG1 > 8) AND (pescataG3 > 8))) #
  # (vincitore == 3) => ((pescataG3 == 8) OR ((pescataG3 <= 8) AND (pescataG1 > 8) AND (pescataG2 > 8))) #
  # (vincitore == -1) => ((pescataG1 != 8) AND (pescataG2 != 8) AND (pescataG3 != 8)) #
```

Come si può notare sono state create tre pescate che rappresentano le pescate di 3 giocatori differenti.

Questi possono pescare numeri tra 1 e 9. Essendo però che nel gioco i numeri pescati devono essere minori di 8, il numero 9 rappresenta semplicemente un caso in cui una serie di pescate hanno superato il numero 8.

Nella parte di constraints sono state inserite le condizioni di vittoria dei 3 giocatori e la condizione di pareggio.

Ciò ha generato diverse combinazioni, come le seguenti:

Test	pescataG1	pescataG2	pescataG3	vincitore
1	1	1	2	-1
2	1	2	3	-1
3	1	3	4	-1
4	1	4	5	-1
5	1	5	6	-1
6	1	6	7	-1
7	1	7	8	3

Questi casi di test andrebbero tutti riportati in JUnit per testare la veridicità.

Per completezza sono stati riportati alcuni casi di test generati da CTWedge in JUnit.

Nello specifico i casi di test generati sono il caso 1, 31, 72 ed i risultati ottenuti sono i seguenti:

```
public class testGiocoCT {
    @Test
    public void test() {
        //Combinatorial 1
        int result = 0;
        Gioco g1 = new Gioco(3);
        int[] pescata1 = {1,1,2};

        result = g1.giocata(pescata1);
        assertEquals(-1,result);

        //Combinatorial 31
        Gioco g2 = new Gioco(3);
        int[] pescata2 = {2,2,4};

        result = g2.giocata(pescata2);
        result = g2.giocata(pescata2);
        assertEquals(3,result);

        //Combinatorial 72
        Gioco g3 = new Gioco(3);
        int[] pescata3 = {4,5,4};

        result = g3.giocata(pescata3);
        result = g3.giocata(pescata3);
        assertEquals(3,result);
    }
}
```

Runs: 1/1 Errors: 0 Failures: 0

testGiocoCT [Runner: JUnit 4] (0,000 s)

test (0,000 s)

Casi di test generati

Risultati casi di test