TESTING e VERIFICA del SOFTWARE

DOCUMENTAZIONE

# IL CODICE SEGRETO

Nel progetto sviluppato per il corso di Testing e verifica del software viene proposto una versione modificate del gioco “Codice Segreto”.

Questo prevede che un giocatore inventi, di nascosto, una combinazione di N elementi presi in modo arbitrario da un elenco prestabilito, in ordine, con possibili ripetizioni. L’altro giocatore, ovvero *il decifratore*, dovrà indovinare il codice nel minor numero di turni possibile, ottenendo di volta in volta degli aiuti da parte del giocatore 1.

Infatti, dopo ogni tentativo del decifratore, verrà assegnato un valore ad ogni elemento del codice ipotizzato:

* Se l’elemento corrisponde a quello presente nella medesima posizione del codice segreto, viene indicato un “HIT”
* Se l’elemento non corrisponde, ma è presente in una posizione diversa, viene indicato con “CLOSE”
* Se l’elemento non è presente nel codice viene indicato con “MISS”.

Il decifratore ha un massimo di 10 turni per indovinare il codice, scaduti i quali il giocatore 1 sarà dichiarato vincitore.

# JML

Nella versione in Java del gioco, sono state apportate le seguenti modifiche al regolamento del gioco:

* Il codice deve essere composto da esattamente cinque lettere.
* Al decifratore viene assegnato un punteggio ad ogni turno: 2\*turno per ogni HIT, più 1\*turno per ogni CLOSE e -2\*turno per ogni MISS, dove *turno* indica il numero del turno in corso.
* Il giocatore può in ogni istante vedere il risultato del suo ultimo tentativo
* Nel caso in cui il decifratore indovini esattamente un elemento del codice, anche nella posizione, in tutti i tentativi successivi dovrà ricordarsi quell’elemento, in quanto non gli sarà più permesso assegnarne un altro differente nel medesimo slot.

La classe presenta 6 attributi: il codice segreto (un array di 5 stringhe), i punti accumulati dal giocatore, il numero di turni trascorsi, i risultati del tentativo precedente (con true se era HIT e false altrimenti), results in cui sono salvati i risultati del tentativo in corso e endGame che indica se il gioco è finito con una vittoria per il decifratore.

Gli attributi privati hanno la specifica /\*@spec\_public@\*/

Nel codice JML sono qui inseriti due invarianti, prima del costruttore

//i turni devono essere compresi tra 0 e 10

//@ public invariant turn >=0 && turn <=10;

//ogni elemento del codice è composto da una e una sola lettera

//@ public invariant (\forall int x; x>=0 && x<5; code[x].length()== 1);

Questi impongono che il numero di turni sia compreso tra 0 e 10, dopo i quali quindi non è più possibile tentare di indovinare la combinazione. Il secondo invariante invece stabilisce che ogni elemento corrisponda ad una sola lettera (per tale motivo era possibile usare char invece che String. In questo modo è possibile modificare leggermente il codice per far sì che accetti anche parole di diversa lunghezza, ma nel codice di esempio è limitato solo a caratteri).

Come requisiti del costruttore sono richieste le seguenti caratteristiche, per far si che il codice da indovinare abbia determinati attributi.

//il codice non può essere vuoto

//@ requires c!=null;

// il codice deve essere lungo 5

//@ requires c.length == 5;

// posso passare solo 1 lettera

//@ requires (\forall int x; x>=0 && x<5; c[x].length()==1);

Al termine del costruttore, gli attributi della classe soddisferanno le seguenti caratteristiche:

// inizialmente i controlli sono settati a falso

//@ ensures (\forall int x; x>=0 && x<5; previous[x]==false);

//il codice segreto viene impostato uguale all'input

//@ ensures code == c;

Il metodo guessCode, che prende in ingresso il tentativo di indovinare il codice, richiede che anche questo tentativo rispecchi le caratteristiche del codice segreto.

// la lunghezza del tentativo deve essere 5

//@ requires c.length == 5;

// ogni parte di codice non deve essere nulla e lunga 1

//@ requires (\forall int x; x>=0 && x<5; (c[x].length())== 1 && c[x]!=null);

// se al turno precedente ho indovinato un pezzo, devo mantenere quel risultato

//@ requires isCodeValid(c);

Si noti che queste specifiche utilizzano in alcuni casi altri metodi della classe Java, illustrati di seguito. Inoltre, viene assicurato che il risultato ritornato dal metodo sia coerente con il confronto tra il tentativo di indovinare il codice segreto e il codice segreto stesso.

// se ho indovinato il codice, ritorna vero, altrimenti, ritorna falso

//@ ensures (code==c && turn<10 && isCodeValid(c)) ==> \result;

//@ ensures !\result <== (code!=c);

// se il turno è maggiore o uguale a 10 torna falso, ma se è falso non vuol dire che il turno sia >=10

//@ ensures (turn>=10 && !endGame) ==> !\result;

// i punti vengono assegnati correttamente

//@ ensures ((turn <= 10) && isCodeValid(c) && (\old(endGame)==false) && c!=null ) ==> (points == (\old(points) + tempPoints));

//@ diverges true;

Questo diverges true fa si che non sia necessario controllare che il ciclo for termini (siccome siappiamo a priori che la sua lunghezza dipende dalla lunghezza del codice segreto decisa durante la stesura del codice)

I seguenti sono invece i contratti per il metodo isCodeValid, che verifica se il codice inserito non violi la regola di ripetere obbligatoriamente ogni elemento che sia già stato indovinato nei turni precedenti.

// la lunghezza del tentativo deve essere 5

//@ requires c.length == 5;

// controllo che il risultato rispecchi la regola di correttezza

//@ ensures \result <==> (\forall int x; x>=0 && x<5; code[x].equals(c[x]) || !previous[x]);

//@ ensures !\result <== !(\exists int x; x>=0 && x<5; !(code[x].equals(c[x])) || previous[x]);

//@ diverges true;

Questo metodo è definito con /\*@ pure @\*/ poiché come visto per il metodo precedente, viene richiamato all’interno di altri contratti. Inoltre, essendo privato abbiamo bisogno di aggiungergi spec\_public.

Allo stesso modo sono definiti i contratti di guessPoints, che ritorna il numero di punti guadagnati o persi in seguito ad un tentativo.

// la lunghezza del tentativo deve essere 5

//@ requires c.length == 5;

// se ho fatto tutto giusto, devo aver guadagnato 10\*turn punti

//@ ensures (\forall int x;x>=0 && x<5; code[x].equals(c[x])) <==> \result==10\*turn;

// se ho fatto qualcosa di sbagliato, devo aver guadagnato meno di 10\*turn punti

//@ ensures (\exists int x; x>=0 && x<5; !(code[x].equals(c[x]))) ==> \result<=10\*turn;

//@ diverges true;

Infine, il metodo isLetterInCode, che ritorna vero o falso se l’elemento analizzato si trovi o meno all’interno del codice segreto, presenta i seguenti contratti.

//puoi prendere solo una lettera come input

//@ requires l!=null && l.length() == 1 ;

// se esiste una pezzo di codice uguale alla lettera, ritorna vero

//@ ensures (\exists int x; x>=0 && x<5; code[x].equals(l)) <==> \result;

// se non esiste una pezzo di codice uguale alla lettera, ritorna falso

//@ ensures (\forall int x; x>=0 && x<5; !(code[x].equals(l))) <==> !\result;

//@ diverges true;

Mentre i metodi getPoints e getResults i seguenti:

//@ requires true;

//@ ensures \result == points;

// non lo posso lanciare prima del primo turno perchè non avrei nessuna risposta

//@ requires turn>=2;

//@ ensures \result.length == 5;

All’interno dei diversi metodi troviamo anche dei contratti anche per i loop:

// @ loop\_invariant i>=0 && i<=5;

// @ loop\_invariant ((i>=0 && i<5) ==> code[i]!=null);

// @ loop\_writes i, s[\*];

e un assert per verificare lo stato di alcune variabili (in questo caso il turno):

//@ assert turn <= 10;

Il codice viene testato in un Main a parte, dove i risultati rispecchiano ciò che ci aspettavamo. Sono stati osservati:

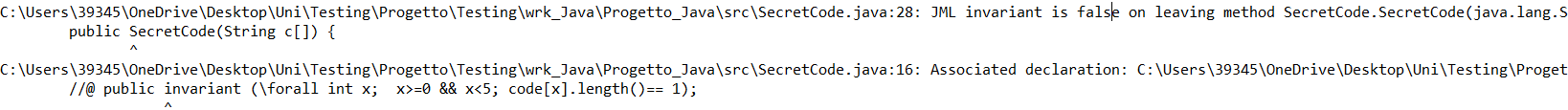
* L’inserimento di un codice errato
* Il cambiamento dei punti
* Il valore dell’array results, con eventuali “aiuti” per il decifratore
* Il cambiamento dei turni
* L’invocazione di un’eccezione nel caso in cui venga violato un invariante

Senza violazione:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Con violazione:



# CODE INSPECTION

Per fare code inspection della classe Java “SecretCode” è stata utilizzata la seguente checklist:

**1. Specification / Design**

- Is the functionality described in the specification fully implemented by the code? ✓

- Is only specified functionality implemented with no additional functionality added? ✓

- Is the program interface implemented as described in the javadocs? ✓(non c’è un’interfaccia)

- Is the code free of "smells?" (Duplicate code, long methods, big classes, breaking encapsulation, etc.) ✓

**2. Initialization and Declarations**

- 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? ✓

**3. Method Calls**

- Are parameters presented in the correct order? ✓

- Is the correct method being called, or should it be a different method with a similar name? ✓

- Are method return values used properly? ✓

**4. Arrays**

- Are there no off-by-one errors in array indexing? ✓

- Have all array (or other collection) indexes been prevented from going out-of-bounds? ✓

**5. Object Comparision**

- Are all objects (including Strings) compared with "equals" and not "=="? ✓ (non per i tipi primitivi tipo int)

**7. Output Format** (Nel main e getResults)

- Are displayed output free of spelling and grammatical errors? ✓

- Are error messages comprehensive and provide guidance as to how to correct the problem? ✓

- Is the output formatted correctly in terms of line stepping and spacing? ✓

**8. Computation, Comparisons and Assignments**

- Check order of computation/evaluation, operator precedence and parenthesising ✓

- Are all denominators of a division prevented from being zero? ✓ (Non ci sono divisioni)

- Is integer arithmetic, especially division, used appropriately to avoid causing unexpected truncation/rounding? ✓

- Are the comparison and boolean operators correct? ✓

- Is the code free of any implicit type conversions? ✓

**8. Exceptions**

- Are all relevant exceptions caught? ✓

**9. Flow of Control**

- Are all loops correctly formed, with the appropriate initialization, increment and termination expressions? ✓

# ANALISI STATICA

Per effettuare un’analisi statica del codice è stato utilizzato il tool SpotBugs, integrato in Eclipse.

Questo software controlla più di 400 bugs differenti nel codice, in questo caso ne sono stati trovati sei:

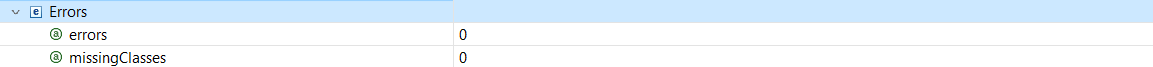
Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Ognuno di questi bugs aveva un rank pari a 15 (i valori vanno da 1 a 20, dove 20 è il meno critico ) segnalati come “of concern” e che quindi non minano il corretto funzionamento del programma ma al più rappresentano problemi di efficienza, ad esempio la creazione di variabili non utilizzate o l’importazione di librerie non richiamate.

I bugs rilevato sono nella classe Main e alcuni di questi sono presenti perché in tale classe alcune variabili sono create solo per essere stampate o per verificare che il costruttore crei l’oggetto come ci si aspetta e quindi dopo essere state create non vengono più richiamate.

Inoltre, il programma non rileva alcun errore all’interno del programma:



AZIONI

Applicando qualche modifica al main, ovvero utilizzando tutte le variabili inizializzate, i bugs vengono ridotti:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

# JUNIT

Il codice Java già descritto è stato quindi sottoposto a test di copertura tramite Junit. Il primo metodo creato puntava a fare un test delle istruzioni, per fare ciò sono stati effettuati i seguenti casi di test:

CASO DI TEST 1

* Creazione di un oggetto “SecretCode” con codice ABCDE
* Controllo del turno chiamando “getTurno”
* Test di un primo tentativo con codice ACZZZ, che risulta falso
* Controllo dei punti, con risultato 2\*1 + 1\*1 - 2\*1 - 2\*1 - 2\*1 = -3
* Controllo che i valori assegnati ad ogni slot siano veritieri
* Test di un secondo tentativo, violando la regola di ripetizione degli elementi corretti. In questo caso il turno non aumenta perché il codice non era valido.
* Test di un terzo tentativo con codice corretto, ritorna vero.

Questi test erano sufficienti per una copertura al 100% delle istruzioni e dei rami. Tuttavia, è stato necessario effettuare un nuovo metodo al fine di soddisfare anche la copertura MCDC, illustrata nella tabella seguente (anche i cicli for sono stati testati nella loro interezza, ma non sono inseriti nella tabella):

METODO GUESSCODE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Turn <10 | IsCodeValid(c) | !EndGame | Code.Equals(c) |  |
| F | Nv | nv | nv | *nuovo TestCase 3* |
| T | F | nv | nv | *TestCase 1 (punto 6)* |
| T | T | F | nv | *Nuovo TestCase 2* |
| T | T | T | F | *TestCase 1 (punto 3)* |
| T | T | T | T | *TestCase 1 (punto 7)* |
|  |  |  |  |  |

METODO GUESSPOINTS

|  |  |  |
| --- | --- | --- |
| Code.Equals(c) | isLetterinCode(l) |  |
| T | nv | *TestCase 1 (punto 3 lettera A)* |
| F | F | *TestCase 1 (punto 3 lettera C)* |
| F | T | *TestCase 1 (punto 3 lettera Z)* |

METODO ISCODEVALID

|  |  |  |
| --- | --- | --- |
| !Code.Equals(c) | Previous[i] |  |
| F | nv | *TestCase 1 (punto 3 lettera A)* |
| T | F | *TestCase 1 (punto 3 lettera Z)* |
| T | T | *TestCase 1 (punto 6 lettera C in posizione 1)* |
|  |  |  |

METODO ISLETTERINCODE

|  |  |  |  |
| --- | --- | --- | --- |
| Code[i]==l |  |  | |
| T | *TestCase 1 (punto 3 lettera A)* | |
| F | *TestCase 1 (punto 3 lettera Z)* | |

METODO GETRESULTS

|  |  |  |
| --- | --- | --- |
| Results[i]==1 | Results[i]==0 |  |
| T | nv | *TestCase 1 (punto 3 lettera A)* |
| F | T | *TestCase 1 (punto 3 lettera C)* |
| F | F | *TestCase 1 (punto 3 lettera Z)* |

Come si evince da queste tabelle, è stato necessario inserire due nuovi test case.

CASO DI TEST 2

* Creazione di un oggetto “SecretCode” con codice ABCDE
* Dopo aver indovinato il codice una volta, il metodo guessCode viene chiamato nuovamente, ma stavolta la variabile endGame è true.

CASO DI TEST 3

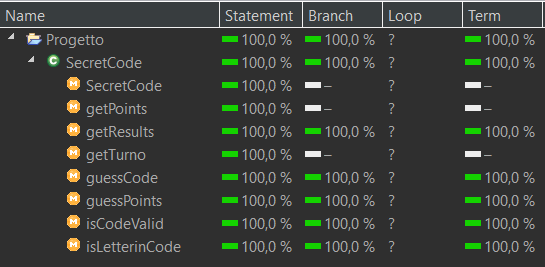
* Creazione di un oggetto “SecretCode” con codice ABCDE
* Vengono fatti trascorrere 10 turni, durante i quali non viene indovinato il codice, per poi chiamarlo un'altra volta.

Tutti i test risultano corretti:

Immagine che contiene testo, schermata, schermo, Carattere

Descrizione generata automaticamente

E garantiscono copertura di branch, istruzioni e condizioni:



# COMBINATORIAL TEST – CTWEDGE

Per effettuare test delle combinazioni è stato utilizzato il software CTWedge, dove per semplicità si è limitato il programma a soli 4 turni invece che 10 e le lettere utilizzabili solo le prime 5 dell’alfabeto, in maiuscolo.

Come parametri sono stati richiesti quindi:

* Una lettera per ogni slot del codice segreto
* Una lettera per ogni slot del codice ipotizzato al turno 1
* Una lettera per ogni slot del codice ipotizzato al turno 2
* Una lettera per ogni slot del codice ipotizzato al turno 3
* Una lettera per ogni slot del codice ipotizzato al turno 4
* Un booleano come valore per la variabile endGame

Come vincoli, si è imposto che, se uno slot di un codice ipotizzato fosse corretto, ovvero uguale al medesimo slot nel codice segreto, allora anche il medesimo slot del turno successivo avrebbe dovuto essere corretto.

Questo comporta che indovinare uno slot al turno 1, impone che esso sia corretto anche al turno 2 e poi al 3 e infine al turno 4.

Inoltre, se il valore di endGame fosse true, tutti gli slot del turno 4 devono essere uguali al codice segreto e viceversa. Questo perché tramite il vincolo precedente, se anche il codice fosse indovinato al turno 1 questo deve essere tramandato identico fino al turno 4.

Per la creazione di test è stato usato un algoritmo PairWise, che ha generato 62 casi reperibili nel file *CombinatorialTesting.xls*.

Due di questi test sono stati riportati in JUnit per testare che essi fossero effettivamente corretti, sono stati scelti un test in cui il codice viene indovinato e uno no.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteImmagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente

# ASMETA

Il gioco del codice segreto è stato modellato anche in ASMeta, con alcuni accorgimenti:

* Il codice è composto da 3 slot, in ognuno dei quali è possibile inserire un colore scelto da un enumerativo.
* Il limite di turni è 10
* Dopo ad ogni turno, viene stampato un messaggio che indica se è l’inizio del gioco (INIZIO), se il giocatore ha indovinato il codice (VITTORIA), se ha commesso qualche errore (SBAGLIATO) o se ha infranto la regola di ripetere i colori già indovinati (ALLARME)

Ad ogni iterazione, il giocatore inserirà la sua ipotesi di codice all’interno dell’array input che sarà confrontato con la variabile *controlled* code e similmente al codice Java verranno aggiornati punti e stato del gioco di conseguenza.

Una differenza con il linguaggio mostrato in precedenza la si trova nello stato 1, infatti qui all’inizio del gioco verrà chiesto al giocatore di inserire i valori di input\_code, una variabile monitored utilizzata per settare il codice segreto nell’array code.

All’iterazione successiva questa richiesta sarà sostituita dal ciclo per la richiesta e il controllo del tentativo di indovinare il codice.

Le funzioni getPoints e PointsValue sono utilizzate per modificare il valore dei punti del giocatore ad ogni turno.

La funziona checkHits controlla che non sia stato violato la regola di ripetizione dei colori corretti.

Per terminare il gioco nel caso in cui i dieci turni siano scaduti o tale regola sia stata violata, sono stati inseriti degli invarianti.

Di seguito un esempio di gioco con finale positivo, tramite l’animatore:

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

E qui una violazione di un invariante:

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

# AVALLA

Per verificare la correttezza del codice Asmeta sono stati creati tre scenari con sviluppi differenti.  
Il primo scenario è il semplice caso in cui il giocatore riesce ad indovinare il codice segreto entro i dieci turni:

Immagine che contiene testo, schermata, Carattere, menu

Descrizione generata automaticamente

Nel secondo scenario il colore indovinato non viene ripetuto al passo seguente:

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Nel terzo scenario scadono i 10 turni:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

# NUSMV e SPECIFICHE

Per verificare la correttezza del codice Asmeta sono state inserite delle specifiche CTL. Tuttavia, queste hanno richiesto delle piccole modifiche al codice precedente, per cui è stata creata una seconda versione:

* Ogni array di input è stato suddiviso in tre variabili separate.
* Il calcolo dei punti è stato aggregato in una singola funzione.
* Gli invarianti sono stati rimossi (siccome presenti come specifiche CTL).
* Molte funzioni interne che prevedevano cicli for o passaggi di variabili sono state modificate.

Le seguenti specifiche risultano corrette:

//esiste uno stato in cui il codice è sempre sbagliato fino alla fine del gioco

CTLSPEC ew( (outMess != ALLARME or outMess = INIZIO), endGame)

//se results è giusto, anche results del futuro deve essere giusto

CTLSPEC ag( results(1) = HIT implies af(results(1) = HIT))

CTLSPEC ag( results(2) = HIT implies af(results(2) = HIT))

CTLSPEC ag( results(3) = HIT implies af(results(3) = HIT))

// se ho indovinato tutti i risultati, ho vinto e il gioco finisce

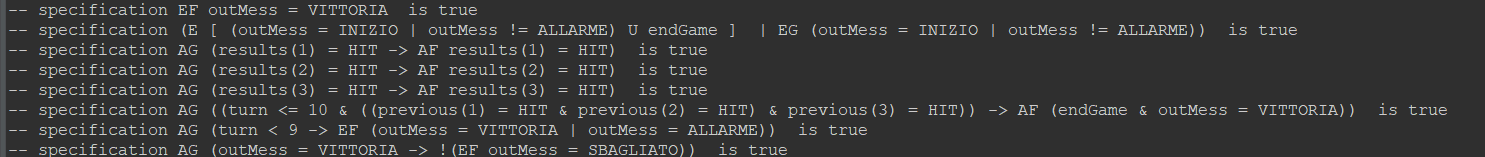
CTLSPEC ag( (previous(1) = HIT and previous(2) = HIT and previous(3) = HIT and turn<=10) implies af(endGame and outMess = VITTORIA))

// se il gioco non è finito, posso ancora vincere

CTLSPEC ag( turn<9 implies ef(outMess = VITTORIA or outMess = ALLARME))

// se ho vinto non posso cambiare scelte

CTLSPEC ag( outMess = VITTORIA implies not ef(outMess=SBAGLIATO))



*Talvolta capita che, quando viene lanciato il comando “Translate in NUSMV and execute”, alcune di queste specifiche, anche facilmente verificabili come la prima, vengano dichiarate false. Riavviare Eclipse sembra risolvere il problema.*

*Immagine che contiene testo, schermata

Descrizione generata automaticamente*

Questa invece è errata:

//uno falso: è possibile che il turno sia uguale a 15

CTLSPEC af(turn = 15)

Immagine che contiene testo, schermata

Descrizione generata automaticamente

E ne viene fornito un contro esempio in cui il codice viene indovinato immediatamente e quindi il gioco termina il turno 1, rendendo falso il fatto che si possa raggiungere il turno 15.

# ATGT – COMBINATORIAL TESTING

Oltre ai test precedenti, è stato scritto una versione semplificata del gioco (più simile a quella originale) per ottenere degli scenari generati casualmente tramite il software ATGT:

* I punti non vengono calcolati
* Il limite dei turni è stato rimosso
* Il gioco continua finchè il giocatore non indovina il codice
* Rimosso l’obbligo di ripetere i colori corretti

Ad ogni iterazione, viene indicato se il colore scelto è corretto, è presente ma in un altro slot o se è del tutto sbagliato.

Da questo codice sono stati generati 21 test, usando come criteri di coverage:

* Basic Rule
* Complete Rule
* Rule Update

I test generati sono disponibili come scenari avalla.

Questa versione del gioco è stata riportata in codice Java e sono stati scritti due test basandosi sugli scenari generati precedentemente. I casi scelti sono uno in cui il codice viene indovinato (testT) e uno in cui il codice non viene indovinato (testFFF2).

Entrambi i test sono stati superati:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

# YAKINDU – FSM

Utilizzando Yakindu è stata creata una semplice macchina a stati che rappresenta i diversi momenti del gioco.  
Sono presenti quattro stati:

* *Gioco*, lo stato principale durante il quale il decifratore deve inserire il suo tentativo.
* *checkCode*, durante il quale il giocatore uno controlla il nuovo tentativo col codice originale.
* *Alarm*, invocato se il codice non rispetta le regole.
* *Win*, invocato se il codice è corretto.
* *Lose*, invocato passati dieci turni di gioco.

A seconda della decisione del giocatore uno, il gioco raggiunge uno stato finale o torna allo stato di *Gioco.* In questo caso il turno viene aumentato di uno.

Immagine che contiene testo, schermata, diagramma, schermo

Descrizione generata automaticamente

La macchina a stati è stata testata con 3 test differenti, ognuno dei quali permette di raggiungere uno diverso degli stati finali. Come si può vedere dall’immagine tutti gli stati e tutte le transazioni sono state controllate almeno una volta, garantendo una copertura del 100%.

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente