Università Politecnica Delle Marche

Dipartimento di Ingegneria dell'Informazione

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Progettazione e Sviluppo di un Simulatore di ATM per il testing automatico

Design and Development of an ATM Simulator for Automated Testing

Relatore:
Prof. Adriano Mancini

Candidato: 1102878
Nicola Mori

Anno Accademico 2021 / 2022

"'You miss 100% of the shots you don't take.'" – Wayne Gretsky'

Michael Scott

Sommario

Le presente tesi si propone di andare a descrivere la progettazione e lo sviluppo di un simulatore di una Automated Teller Machine (ATM) per il testing del modulo software Sigma XFS dell'azienda Sigma S.p.A.

Lo scopo del progetto è quello di costruire un simulatore che sia in grado di replicare i comportamenti di un ATM fisico ed, allo stesso tempo manipolando le azioni che tale dispositivo può compiere, simulare errori, comportamenti occasionali o non previsti, in modo tale da stressare maggiormente il modulo software Sigma XFS per verificare la corretta gestione di eventi sia comuni che estremamente particolari.

Esiste già una soluzione sviluppata dall'azienda; tale soluzione prevede una fase di generazione di file all'interno dei quali viene tracciata la comunicazione che avviene tra l'ATM fisico e il modulo software, ed una successiva fase di lettura del file in modo tale che il simulatore sia in grado di rispondere correttamente al modulo XFS. L'utilizzo di un simulatore è fondamentale in quanto permette di evitare l'interazione diretta con il dispositivo fisico, risparmiando sia a livello di risorse umane sia a livello temporale, permettendo contemporaneamente la replicazione di casistiche particolari. Il dispositivo fisico non è predisposto per generare autonomamente delle condizioni di errore; per questo motivo lo sviluppo di un simulatore software, in cui ci sia la possibilità di emularle, risulta essere un passo fondamentale per il testing del modulo XFS.

La soluzione che verrà presentata all'interno della tesi prevede un simulatore procedurale il cui obiettivo è quello di emulare la comunicazione che avviene tra il *device* reale e il software Sigma XFS, tramite l'utilizzo di moduli chiamati *regole*,

i quali modificheranno lo stato del dispositivo e allo stesso tempo genereranno le risposte in base ad esso. Una ulteriore funzionalità che potrà essere sfruttata sarà quella di andare ad inserire dall'esterno delle *regole manipolatrici* che permetteranno di alterare il comportamento consueto della macchina, come per esempio andare a generare un codice di errore simulando un comportamento anomalo durante un'operazione, oppure riprodurre un comportamento non consueto della macchina, come il deposito di alcune banconote in cassetti non appropriati.

Nel primo capitolo verrà mostrata l'architettura complessiva, in cui avremo tre moduli principali, il Sigma XFS Layer, il modulo `.NET` ed infine il modulo Python. Oltre a questo verrà anche mostrata la soluzione adottata dall'azienda e, sommariamente, quella proposta per il progetto di tesi.

Il secondo capitolo è suddiviso in quattro sezioni. La prima tratta degli strumenti utilizzati per lo sviluppo: gli IDE, i linguaggi di programmazione e i framework utilizzati, andando a mostrare i motivi che hanno portato a queste scelte. Nella seconda sezione è presente la modellazione dello stato del dispositivo; ovvero vengono descritte le componenti del *device* e come sono state modellate nel codice, mostrando come vanno ad influire nella comunicazione. Nella terza sezione viene approfondita la struttura delle *regole*, simili per struttura, ma con differenze a seconda delle richieste che dovranno andare a soddisfare. Verrà mostrato successivamente un esempio per comprendere meglio sia come avvenga la generazione di una risposta da inviare durante la comunicazione, sia come si possa manipolare una regola per alterare il comportamento del *device*. Oltre a questo ci sarà un piccolo capitolo dedicato a particolari regole chiamate regole di *Idle*. Nella quarta sezione è mostrato il *Rule Engine* ovvero il motore che permette l'esecuzione delle regole dalle quali provengono le risposte che verranno poi inviate al modulo software; inoltre all'interno del motore avverrà la gestione dell'evoluzione dello stato del dispositivo dovuta all'esecuzione delle regole.

Mentre i capitoli precedenti sono dedicati alla simulazione del *device* fisico, il terzo capitolo, invece, si occupa della struttura dei test, della loro costruzione, della sintassi, di come avviene l'inserimento esterno delle regole manipolatrici e delle scelte implementative che hanno portato a semplificare la scrittura di un

test.

Infine nel quarto capitolo saranno presentante le conclusioni e i potenziali sviluppi futuri del simulatore.

Indice

Capitolo 1

Introduzione e background del progetto

1.1 Obiettivo & Architettura

L'obiettivo del progetto è quello di progettare, sviluppare e testare un simulatore software di un ATM, *Automated Teller Machine*, così che si possa testare opportunamente il Sigma XFS Layer, sviluppato dall'azienda; ovvero un software incaricato della gestione del *device* fisico sia in condizioni nominali sia in condizioni di errore. Il simulatore dovrà essere in grado di riprodurre sia casi rari, sia casi di errore e dovrà replicare particolari condizioni che si verificano solo localmente, in questo modo poi si potranno rilevare presenza di bug, mancata gestione di situazioni non comuni, errori da parte del modulo XFS.

L'architettura generale per andare ad effettuare un test del modulo, unito all'utilizzo del simulatore è mostrata in figura 1.1 e come possiamo notare è divisa in 3 parti:

- Riquadro in **giallo** abbiamo il *Service Provider* con il *Plug-In*, insieme formano il **Sigma XFS Layer**, ovvero quello strato che dovremo andare a testare grazie al nostro simulatore, ma più in generale si tratta del software il cui compito è quello di gestire il *device* in base alle condizioni in cui esso si trova. Il modulo, nel caso più comune, si interfaccia direttamente

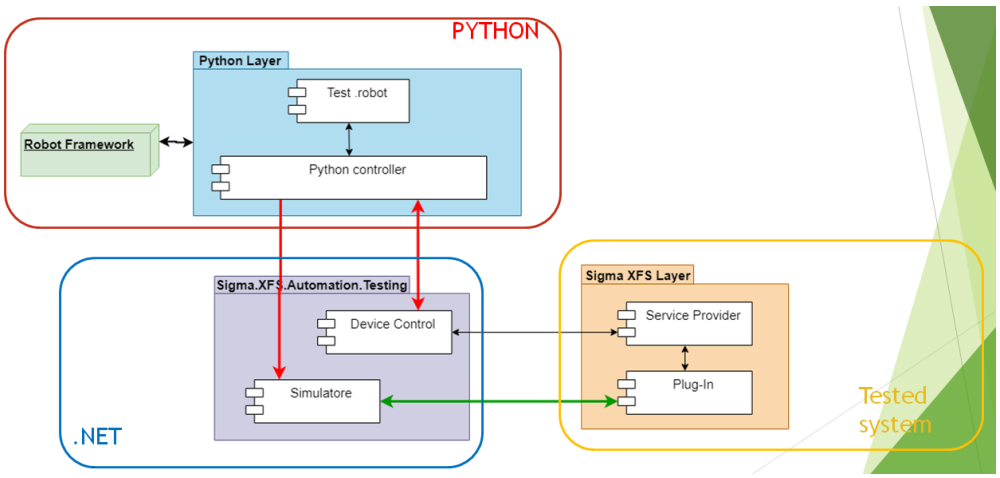

Figura 1.1: Architettura

con il *device* fisico mentre nel momento in cui dovrà interfacciarsi con il simulatore allora in quel caso sarà il *Plug-In* che permetterà ciò.

- In **blu** abbiamo il modulo `.NET` in cui abbiamo il *device Control*, ovvero il controllore software del *device* fisico reale e il *simulatore*, ovvero l'oggetto il cui compito sarà quello di imitare il comportamento del *device* fisico. In questo documento ci focalizzeremo sullo sviluppo del simulatore per il *testing* del *Service Provider*.

- In **rosso** infine il **Python Layer** che prevede l'utilizzo di un framework chiamato *Robot Framework*[1] utilizzato per la scrittura dei file con estensione `.robot` che serviranno per pilotare i test indicando quali sono le operazioni che il *device* fisico o il simulatore dovranno andare ad eseguire. Il Python Controller permette di interfacciarsi sia al *device* fisico che al simulatore, inoltrando ad uno o all'altro i comandi che vengono indicati all'interno dei file `.robot`.

1.2 Soluzione con Sniffing

La soluzione inizialmente adottata dall'azienda prevedeva di collegare il computer con il *device* fisico per intercettare la comunicazione tra il *Service Provider* e il dispositivo, tale comunicazione viene trascritta in un file con estensione `.yaml`. Le azioni che il *device* deve eseguire vengono pilotate dal file *Robot Framework*. Una volta ottenuto il file `.yaml` esso conterrà al suo interno, come mostrato in figura 1.2, diverse voci:

```
Id: PIN RXTXFILE        #identificativo lista rx-tx
Description: List of RxTx Message       #descrizione lista
RxTxDetails:
- Description: GET_SECURECORE_STATUS OPERATIVE          #descrizione rx-tx singolo
  Rx: 40-00-00-00                                       #bytes ricevuti dal simulatore
  Tx: 68-14-00-14-00-68-C2-00-0A-00-00-01-02-00-5F-00-04-00-40-00-00-00-04-00-03-00-00-00-00-16-A2-A0              #byte di risposta del simulatore
  Delay: 100                                            #ritardo nella risposta
  Name: C0                                              #nome rx-tx singolo
  Constraint: ''                                        #prende in considerazione questo rx-tx solo se gli altri singoli rx-tx contenuti in questa l
  Spontaneous: ''                                       #indica che il comando è una spontanea che verrà mandata dopo l'rx-tx indicato è stato utili
- Description: GET_VERSION
  Rx: 55-00-00-05
  Tx: 68-2C-00-2C-00-68-C3-00-56-00-00-01-02-00-60-00-04-00-55-00-00-05-04-00-00-03-03-00-04-00-00-02-01-03-04-00-00-01-07-00-04-00-55-04-00-00-04-(
  Delay: 100
  Name: C1
  Constraint: ''
  Spontaneous: ''
- Description: GetUSN
  Rx: 3F-00-00-05
  Tx: 68-20-00-20-00-68-C4-00-57-00-00-01-02-00-61-00-04-00-3F-00-00-05-10-00-04-00-99-27-20-E3-03-0B-06-10-00-00-C2-1C-0F-13-00-16-5F-5A
  Delay: 100
  Name: C2
  Constraint: ''
  Spontaneous: ''
- Description: SetBuzzerFrequency
  Rx: 25-00-01-02-04-00-00-00-00-00-04-00-00-00-00-00
  Tx: 68-0E-00-0E-00-68-C5-00-03-00-00-01-02-00-62-00-04-00-25-00-01-02-00-16-BD-DE
  Delay: 100
  Name: C3
  Constraint: ''
  Spontaneous: ''
- Description: SetBuzzerFrequency
```

Figura 1.2: Esempio file .yaml

- **Description**: indica il nome del comando che è stato appena eseguito.
- **Rx**: indica la sequenza di byte (il protocollo di comunicazione prevede uno scambio di messaggi in array di byte) che viene inviata da parte del *Service Provider* al dispositivo fisico; ogni byte all'interno del messaggio contiene un'informazione specifica su quale dovrà essere l'operazione eseguita dal dispositivo fisico, in particolare il primo byte è l'identificativo della richiesta.
- **Tx**: indica la sequenza di byte in risposta che il *device* invia al *Service Provider*, in esso sono contenute informazioni parziali sullo stato del di-

spositivo, oltre a questo è presente una porzione che si differenzia in base all'identificativo della richiesta il cui scopo è quello di indicare quale sia il comportamento del *device* e quale sia l'esito dell'operazione appena eseguita (ad esempio come è avvenuto lo smistamento delle banconote in un'operazione di deposito). La risposta deve contenere lo stesso identificativo della precedente richiesta *Rx*.

- **Delay**: ovvero il tempo in millisecondi del ritardo della risposta.
- **Name**: un identificativo per ogni scambia di messaggio *Rx-Tx* avvenuto durante la comunicazione.
- **Constraint**:indica la lista delle richieste Tx che devono essere stati inviati prima di poter prendere in considerazione questa risposta. Questo campo è necessario per quanto riguarda la prima soluzione adottata in quanto permette di ordinare le richieste dello stesso tipo.
- **Spontaneous**: indica dopo quale Tx deve essere inviata la spontanea.

Una volta raccolta tutta la sequenza di messaggi, come già detto viene ottenuto un file `.yaml`, il quale servirà al simulatore per l'esecuzione delle operazioni nel test; tutti i diversi file verranno inviati all'interno di una cartella predisposta per contenerli. Il simulatore riuscirà a connettersi con il *Service Provider* in quanto verrà simulata una connessione USB/Seriale tra i due, come se fosse effettivamente presente il dispositivo fisico. La comunicazione inizia con il *Service Provider* che invierà vettore di byte al simulatore, associabile ad una richiesta *Rx*, a questo punto il simulatore leggerà dal file `.yaml`, corrispondente al tipo di test avviato, la risposta corretta da inviare al *Service Provider* dove tra tutte le possibili scelte verrà selezionata quella che ha più *Constraint* soddisfatti. La procedura appena indicata è l'esecuzione del simulatore implementato inizialmente dall'azienda.

Lato Python, invece, abbiamo il *Python simulator controller* che permette di pilotare il test; infatti è possibile, grazie alla libreria *Robot Framework* con cui l'azienda ha modellato le richieste del modulo XFS, scrivere un test indicando

quali sono le operazioni che il Service Porvider dovrà inviare al *device*, il quale risponderà in modo opportuno come già detto leggendo il file `.yaml`. In figura 1.3 troviamo uno schema che riassume il comportamento; possiamo notare in alto a sinistra della figura il simulatore insieme al file *RxTx*(il file `.yaml`), quest'ultimo verrà letto per generare i messaggi di risposta che verranno inviati al *Service Provider*, la comunicazione tra i due viene permessa grazie ad una simulazione di comunicazione seriale o USB.

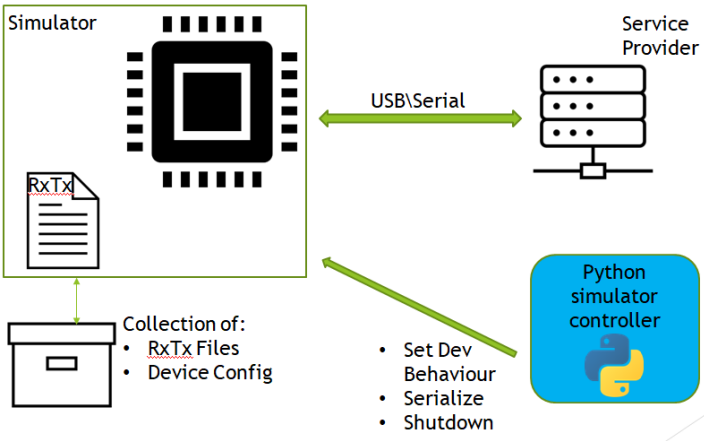

Figura 1.3: Simulatore base

Questa soluzione, che come abbiamo visto, prevede inizialmente un'operazione di *sniffing* [1], quindi, è necessario che la scrittura del test in Python corrisponda necessariamente alla sequenza di operazioni che sono state intercettate quando è stato generato il file `.yaml` del test, senza possibilità di modifiche in quando al suo interno è contenuta la sequenza di operazioni specifica per quella prova. Nel momento in cui test e file `.yaml` non combaciano l'esito del test sarà negativo, quindi sarà necessario ad ogni minimo cambiamento del test andare direttamente *device* fisico, riprodurre il cambiamento del test che si vuole testare con il simulatore e generare il file `.yaml` corrispondente al nuovo test, così che poi si potranno riprodurre le nuove condizioni anche nel simulatore. Un altro svantaggio che si riscontra con questo approccio è la riproduzione dei casi di limite o di errore, infatti per essere riprodotti prima di tutto bisognerebbe riprodurre la sequenza

[1] attività di intercettazione passiva dei dati che transitano in una rete telematica

di messaggi in cui avviene l'errore sul dispositivo reale, per poi utilizzare tale sequenza in lettura da parte del simulatore; perciò è necessario manomettere il *device* reale per riprodurre un caso particolare.

1.3 Soluzione Procedurale

L'alternativa portata avanti in questa tesi prevede lo sviluppo di un simulatore procedurale, basato sulla programmazione ad oggetti, che evita la fase di acquisizione dei dati di comunicazione tra *device* fisico e *Service Provider*, quindi evita anche la generazione del file `.yaml`, con conseguente eliminazione da parte del simulatore della lettura del file stesso, in quanto le risposte vengon generate direttamente dal simulatore rispettando il protocollo di comunicazione con cui *device* fisico e *Service Provider* interagiscono.

La soluzione sviluppata terrà traccia dello stato del dispositivo, che evolverà in base alle operazioni che verranno eseguite dal simulatore, mentre la risposta verrà generata da opportuni moduli chiamati **regole**, che terranno conto dello stato attuale e del tipo di richiesta arrivata. Grazie al simulatore inoltre è possibile modificare qualsiasi richiesta che viene inviata dal *device* al *Service Provider*, infatti è possibile manipolare a proprio piacimento il risultato dell'operazione attraverso opportune funzioni; un esempio di tale applicazione potrebbe essere simulare un errore restituendo il codice di ritorno adeguato, oppure modificare la distribuzione delle banconote in seguito ad un'operazione di deposito, ovvero l'arrivo di queste ultime in cassetti non predisposti alla loro gestione.

Un miglioramento rispetto alla soluzione che prevede come fase iniziale lo *sniffing*, è il seguente: la scrittura del test in *Python* non deve combaciare con le operazioni contenute nel file `.yaml`, infatti le risposte non vengono più lette dal file in questione ma vengono generate dal dispositivo; quindi la scrittura del test risulta essere non vincolata. Inoltre è possibile personalizzare il comportamento del *device* utilizzando metodi messi a disposizione dalla soluzione che permetteranno di generare situazioni particolari, errori, modifiche o semplicemente casi eccezionali che possono verificarsi solo in configurazioni specifiche.

Capitolo 2

Design e Sviluppo del Simulatore

Nel presente capitolo verranno analizzate le diverse componenti del simulatore; durante lo sviluppo sono state effettuate scelte di design e scelte implementative che hanno portato alla soluzione presentata nelle sezioni seguenti. Il primo capitolo presentato, 2.2, riguarda la modellazione dello stato del *device*, ovvero il dispositivo fisico è composto da diverse componenti (p.es. cassetto per il deposito delle banconote, zona d'inserimento delle banconote...), ognuna delle quali ha un proprio stato che influisce sulla comunicazione con il *Service Provider*; a causa di ciò è stato deciso di creare una classe che racchiudesse lo stato complessivo del dispositivo chiamata **DeviceStatusModel**. Nel capitolo successivo, 2.3, vengono mostrate altre tre porzioni che compongono il simulatore: il **metodo encode**, ovvero il metodo che costruisce il messaggio di risposta per il *Service Provider*, le **regole**, ovvero delle opportune classi il cui compito è quello di generare una particolare porzione del messaggio, ed infine l'ultimo aspetto presentato sono le regole chiamate **regole di idle** il cui obiettivo sarà quello di simulare il comportamento di eventi esterni al *device*, come l'estrazione o l'inserimento delle banconote. Nell'ultima sezione, 2.4, viene mostrato il motore che permette l'esecuzione delle regole, tracciando le modifiche dello stato del simulatore, prendendo in carico le nuove richieste provenienti dal *Service Provider*.

Nella figura 2.1 è possibile osservare come le diverse componenti citate precedentemente interagiscono tra di loro. La classe `DeviceStatusModel`, al suo in-

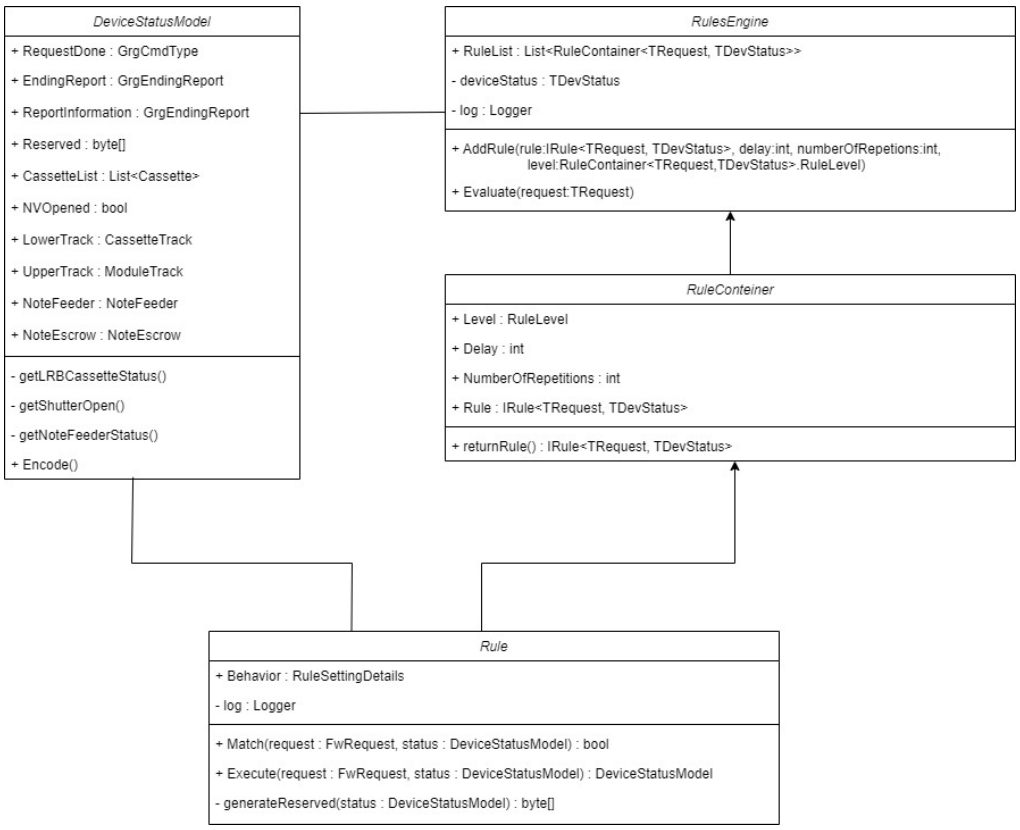

Figura 2.1: Diagramma delle classi del simulatore

terno, ha come attributi le diverse componenti del *device* fisico; interagisce con la classe `RulesEngine` per l'esecuzione delle regole e l'evoluzione dello stato. Si relaziona anche con la classe `Rule`, ovvero con una regola, il cui compito sarà quello di produrre il vettore di `byte Reserved`, attributo della classe `DeviceStatusModel` e di modificare lo stato. La `Rule`, a sua volta, sarà un attributo della classe `RuleContainer`; diverse istanze di quest'ultima, poi, andranno a "popolare" l'attributo `RuleList` del `RulesEngine`, il quale, grazie al metodo `Evaluate`, andrà ad eseguire le diverse regole che comporranno la lista.

2.1 Linguaggi & Strumenti utilizzati

Prima di analizzare le diverse componenti del simulatore è necessario dedicare una sezione ai linguaggi di programmazione e agli strumenti utilizzati per lo sviluppo del codice, chiarificando quali sono le motivazione che hanno portato a queste scelte.

2.1.1 C#

Il linguaggio utilizzato per sviluppare il simulatore del dispositivo è C#[2]; questa scelta è stata dettata dal fatto che la soluzione precedente, ovvero quella con *sniffing*, era implementata in tale linguaggio, inoltre, così facendo, l'integrazione della nuova soluzione all'interno del codice già presente è stata facilitata.

2.1.2 Python

Il linguaggio *Python*[3] è stato scelto per la scrittura dei test, per l'aggiunta di regole (vedremo successivamente cosa sono) e per le varie manipolazioni di esse. La scrittura delle operazioni che devono essere eseguite dal simulatore durante il test sono state sviluppate con il supporto di un *framework* chiamato **Robot Framework**.

Robot Framework

Robot Framework è un framework di automazione *open source*. Ha la capacità di poter essere integrato con qualsiasi altro strumento per creare soluzioni di automazione potenti e flessibili. Uno dei punti a favore di Robot Framework è la semplicità di scrittura di codice grazie alla possibilità di creare parole chiave, inoltre è possibile creare librerie per facilitare ulteriormente lo sviluppo di test. Oltre a questo è possibile strutturare i *report* dopo l'esecuzione dei test, infatti sono personalizzabili utilizzando il linguaggio di markup HTML. L'azienda ha messo a disposizione una libreria con le operazioni del modulo XFS già modellate, cosicché i test potessero essere scritti con una certa elasticità permettendone una scrittura semplice e snella.

2.1.3 IDE

Visual Studio

Il primo strumento utilizzato per lo sviluppo di codice in C# è *Visual Studio*[4], grazie al quale è stato possibile realizzare l'intera sezione inerente al simulatore del dispositivo. La scelta di *Visual Studio* è stata preferita alle altre poiché risulta essere l'*IDE* leader per scrittura del codice in C#.

Visual Studio Code

Per quanto riguarda invece la sezione in *Python* l'*IDE* scelto è stato *Visual Studio Code*[5] in quanto per tale linguaggio risulta essere un buon programma per lo sviluppo di codice; inoltre nella sezione Estensioni è possibile acquisire un *plugin* per lo sviluppo di codice in *Robot Framework*.

2.2 Lo Status Model

Il primo passo compiuto per lo sviluppo del simulatore procedurale è la modellazione del dispositivo fisico. Il *device* fisico è formato da diverse componenti, ognuna delle quali ha uno stato che influisce nel protocollo di comunicazione con il software XFS; per questa ragione è stata creata opportunamente una classe, chiamata `DeviceStatusModel`, che andasse a riunire al suo interno le diverse componenti del *device* fisico (p. es. cassetti per il deposito delle banconote, zona di prelievo e inserimento...) cosicché si potesse modellare lo stato complessivo del dispositivo unendo anche la possibilità di evolverlo durante l'esecuzione del simulatore.

In generale esistono diverse componenti fisiche, come bocchetta delle banconote, cassetti dove vengono depositate, cassetti di deposito per le banconote false..., ognuna delle quali può avere comportamenti più o meno complessi; per questa ragione in alcuni casi è stato deciso di dedicare una classe apposita per il singolo componente in modo tale che si potessero anche simulare procedure e situazioni più intricate. Nella tabella 2.1 viene riportata la lista degli attributi della classe `DeviceStatusModel` che modellano lo stato del dispositivo, nella colonna **Descrizione** tra parentesi si hanno i valori di default, ovvero lo stato al tempo t0 del simulatore.

Gli attributi elencati rappresentano gli oggetti necessari per modellare lo stato in modo tale che poi lo scambio di messaggi con il *Service Provider* sia concorde con il protocollo di comunicazione.

Gli attributi `RequestDone` ed `EndingReport` sono di tipo `enum`; nel primo caso si ha l'identificativo del messaggio, ovvero la tipologia di richiesta arrivata dal *Service Provider*, mentre nel secondo viene specificato il codice di ritorno che il *device* invia, ovvero se l'esito dell'operazione che si è conclusa è di successo, errore o *warning*. `ReportInformation` fa riferimento ad un altro `enum`, in questo caso si ha un *array* di due `byte` dove in base al valore che assume verrà indicato un *warning* specifico oppure un tipo di errore verificatosi nel processare la richiesta. L'attributo `Reserved` è un *array* di `byte` più complesso, di lunghezza variabile

a seconda del `RequestCode` che deve essere costruito diversamente richiesta per richiesta; al suo interno i `byte` indicano l'esito delle operazioni fisiche all'interno del *device*, ovvero come ad esempio in un'operazione di deposito le banconote sono state distribuite all'interno dei diversi cassetti.

2.2.1 CassetteList

La `CassetteList` è una lista in cui sono contenuti tutti i cassetti presenti nel *device*; è possibile personalizzare lo stato del dispositivo aggiungendone o rimuovendone a proprio piacimento, questo poi comporterà delle modifiche quando verranno eseguite le *regole* per processare la risposta che verrà inviata al *Service Provider*.

Cassette

La classe `Cassette` modella lo stato del cassetto, al suo interno sono presenti una serie di attributi, come per esempio la posizione all'interno del *LowerTrack*, qual è il taglio di banconote che un cassetto potrà accettare nelle operazioni di deposito e dispensazione... Nella tabella 2.2 è possibile osservare quali sono gli attributi che fanno parte di questa classe.

Per quanto riguarda la `Position` il tipo è `PositionCassette` ovvero un `enum`, 2.1, creato appositamente per specificare la posizione del cassetto.

Listing 2.1: PositionCassette enum

```
public enum PositionCassette
{
    Position01 = 1,
    Position02 = 2,
    Position03 = 3,
    Position04 = 4,
    Position05 = 5,
    Position06 = 6,
    Position07 = 7,
```

```
    Position08 = 8,
    Position09 = 9,
    Position10 = 10,
    Position11 = 11,
    Position12 = 12,
    Position13 = 13,
    Position14 = 14,
    Position15 = 15,
    Position16 = 16,
    PositionAC1 = 17,
    PositionAC2 = 18,
    PositionLRB = 19,
}
```

In particolare si possono notare le posizioni `PositionAC1`, `PositionAC2` rispettivamente dei cassetti di accumulo per operazioni particolari, mentre la posizione `PositionLRB` è destinata al cassetto adibito a contenere le banconote false.

Di seguito nella tabella abbiamo la `Denomination`, il tipo di dato è `int`; per indicare il taglio di banconote che il cassetto potrà accettare bisogna assegnare il valore della banconota, inoltre non è necessario specificare quale sia la valuta.

Per il `CassetteStatus` è stato generato, anche qui, un `enum`, 2.2, che vada a modellare lo stato del cassetto. In questo caso è stata aggiunta la parole chiave `Flags` in modo tale che potessero essere combinate le diverse opzioni, questo perché il protocollo prevede che lo stato di un cassetto può essere il risultato della combinazione in OR logico dei diversi possibili stati elencati nell'`enum` 2.2.

Listing 2.2: CassetteStatus enum

```
[Flags]
public enum CassetteStatus : byte
{
    CassettePresent = 0x01,
```

```
    CassettePressurePlateFailure = 0x02,
    CassetteFeederRollerFault = 0x04,
    CassetteAbnormal = 0x08,
    CassetteEmpty = 0x10,
    CassetteInformationFailure = 0x20,
    CassetteAlmostFull = 0x40,
    CassetteFull = 0x80,
}
```

Per indicare un cassetto vuoto, dove all'interno non sono presenti banconote, è necessario che il cassetto nella lista `CassetteList` della classe `DeviceStatusModel` abbia uno stato che risulti essere la combinazione in OR logico tra gli stati `CassettePresent` | `CassetteEmpty`, poiché il valore ottenuto a livello di `byte`, ovvero 0x11, corrisponde alla condizione di cassetto vuoto per il protocollo.

Le tipologie di cassetto sono state mappate all'interno di un `enum` chiamato `CassetteCategory`, 2.3, in questo modo è possibile personalizzare anche il tipo di cassetto che farà parte del nostro simulatore.

Listing 2.3: CassetteCategory enum

```
public enum CassetteCategory
{
    Unknown = 0,
    RC = 1,
    DC = 2,
    LRB = 3,
    AC_Reject_Withdrawal = 4,
    AC_Reject_Deposit = 5,
}
```

Con `Unknown` è possibile specificare un cassetto non riconosciuto oppure un individuo che non lo ha inserito correttamente, con `RC` può essere indicato il cassetto di ricircolo, ovvero un cassetto dal quale possono sia entrare che uscire le banconote; quindi verrà coinvolto sia per le operazioni di dispensazione sia per

quelle di deposito. Con `DC` si può definire un cassetto di deposito, ovvero un cassetto il cui compito è solamente quello di accettare banconote in entrata, quindi viene coinvolto solamente nelle operazioni di deposito. `LRB` indica il cassetto utilizzato dal *device* per confinare quelle che sono le banconote riconosciute come false durante l'operazione di deposito. Come ultima categoria si hanno i cassetti di `Reject`, i quali indicano dei cassetti in cui vengono indirizzate le banconote nel momento in cui si verificano eventi particolari nel *device*, come riconoscimento di banconote sospette di falsità, errori, *warning* oppure comportamenti anomali da parte dell'utente.

Con `BanknoteLevel` viene indicato il livello della banconota, ovvero se ad esempio la banconota risulta essere vera oppure falsa; per modellare ciò è stata adottata la stessa strategia degli attributi precedenti, quindi è stato utilizzato un apposito `enum` denominato `BanknoteCategoryLevel`, 2.4.

Listing 2.4: BanknoteCategoryLevel enum

```
public enum BanknoteCategoryLevel
{
    L4A = 0x01,
    L4B = 0x02,
    L3 = 0x04,
    L2 = 0x08,
    Unknown = 0x16,
}
```

Con `L4A` vengono indicate le banconote che sono ora in circolo nel mercato, in buone condizioni; le `L4B` invece sono banconote ancora valide nel mercato che hanno subito un eccessivo deterioramento oppure sono banconote ancora valide ma di una vecchia versione, quindi non possono essere utilizzate per il ricircolo e solitamente vengono confinate in cassetti di deposito non ricircolabili. `L3` sono le banconote false con certezza mentre con `L2` vengono classificate le banconote sospette di falsità. Per ultima si ha la categoria `Unknown` indicando oggetti non riconosciuti, ad esempio un semplice pezzo di carta.

`FirmwareVersion` è una semplice stringa utile a specificare la versione firmware del cassetto.

`NoteInfos` è un dizionario: come chiave si avrà la `Denomination` della banconote, ovvero il taglio, mentre come valore un `int` per indicare quante banconote sono state coinvolte in un'operazione di deposito per quel cassetto, questo permetterà di costruire coerentemente la risposta per il *Service Provider*, non è una vera e propria componente del *device*.

2.2.2 LowerTrack

Il *Lower Track* costituisce la zona inferiore del *device*, può essere aperta, estratta e reinserita; al suo interno è contenuta la zona di inserimento dei cassetti; si hanno 16 posizioni per i cassetti di ricircolo e deposito, mentre altre tre zone apposite per inserire i cassetti AC1, AC2 LRB. Per modellare questo oggetto è stata implementata la classe `CassetteTrack` la quale a sua volta eredita attributi e metodi della classe genitore `ModuleTrack`. Partendo dalla classe genitore essa è composta da due attributi, entrambi di tipo booleano:

- `TrackInPosition`: ovvero indica se il modulo è in posizione, cioè se non è stato estratto dal *device*.
- `Reinserted`: indica se il modulo, dopo essere stato estratto, è stato reinserito.

La classe `CassetteTrack` eredita questi attributi per poi aggiungerne un altro ovvero `CoverOpen`, anch'esso booleano, ed indica se il coperchio del *Lower Track* è stato aperto.

2.2.3 UpperTrack

L'*Upper Track* a differenza del *Lower Track* non ha coperchio quindi viene modellato solamente utilizzando la classe `ModuleTrack`, dove, come visto precedentemente, vi sono solamente i due attributi `TrackInPosition` e `Reinserted`.

2.2.4 NoteFeeder

Il *Note Feeder* è la regione di un ATM dove vengono inserite o dispensate le banconote, nel nostro caso l'interesse è stato quello di modellare solamente eventuali comportamenti che vanno ad influire all'interno del protocollo, quindi gli attributi presenti sono unicamente quelli di interesse per produrre una risposta coerente.

Il *Note Feeder* è stato implementato creando una classe chiamata `NoteFeeder` dove all'interno sono stati definiti diversi attributi, come mostrato nella tabella 2.3.

Si può notare la presenza di quattro attributi di tipo booleano che indicano rispettivamente se è avvenuto o meno un determinato evento, questo simula il comportamento dei sensori presenti all'interno del *Note Feeder*. Nella tabella 2.3 vengono elencati gli eventi che possono aver luogo all'interno del *Note Fedeer*.

- `BanknotePresentInShutterZone`: `true` se le banconote sono presenti nella *Shutter Zone*, ovvero la parte superiore rispetto alla bocchetta canonica di inserimento delle banconote.
- `BanknotePresentAtSlot`: `true` se le banconote sono state inserite all'interno dell'area canonica di inserimento delle banconote, oppure se durante l'operazione di dispensazione sono giunte all'interno della bocchetta.
- `BanknoteWrongPositionIntheSlot`: `true` indica banconote inserite in modo errato da parte dell'utente all'interno del *Note Feeder*.
- `CountInfoDeposit`: indica le banconote che sono presenti all'interno del *Note Feeder*. Come possiamo notare è una lista di oggetti della classe `CountInfo`.

CountInfo

La classe `CountInfo`, è una classe utilizzata per indicare quali sono le banconote che vengono immesse in un'operazione di deposito; un oggetto di tale classe sarà composto da tre attributi come mostrato nella tabella 2.4.

Grazie a questa classe sarà possibile creare una lista di oggetti `CountInfo`: ogni oggetto conterrà tutte le banconote di quella determinata categoria (L4A, L4B . . .), all'interno dell'oggetto nel dizionario si potranno specificare in maniera puntale il numero di banconote per ogni singolo taglio.

2.2.5 NoteEscrow

Il *Note Escrow* è un oggetto nel *device* che si trova in una posizione intermedia tra il *Note Feeder* e l'insieme dei cassetti, esso viene utilizzato nella maggior parte dei casi nelle operazioni di deposito; quando viene eseguita un'operazione di questo tipo le banconote dal *Note Feeder* prima vengono trasferite all'interno del *Note Escrow* successivamente, quando viene confermato il deposito, vengono trasferite all'interno degli opportuni cassetti presenti nel *device*. La sua capacità massima è di 300 banconote, una volta raggiunta la soglia non sarà più in grado di accumularne altre.

A differenza del *Note Feeder*, dove si hanno un numero consistente di attributi, per quest'oggetto il protocollo non prevede un numero eccessivo di informazioni da portare con sé, quindi la sua modellazione risulta essere più semplice come mostrato nella tabella 2.5

2.3 Le Rules

Il *device* comunica con il *Service Provider* attraverso un opportuno protocollo quindi, nel momento in cui viene implementato un simulatore, è necessario che questo sia in grado di riprodurre esattamente gli stessi messaggi che un *device* fisico trasmette; ciò significa che all'interno dell'*array* devono essere contenute informazioni coerenti rispetto la richiesta da soddisfare.

La comunicazione tra i due soggetti avviene tramite *array* di `byte`; ogni `byte` ha un significato rispetto alla posizione in cui si trova nella risposta, per questo motivo l'ordine è fondamentale in quanto sbagliare una singola posizione porterebbe ad imitare un comportamento diverso da quello atteso. Il *Service Provider* invia un messaggio al nostro *device*/simulatore; il primo `byte` all'interno dell'*array* indicherà il tipo di richiesta che dovrà essere soddisfatta ed anche nella risposta il primo `byte` dovrà essere lo stesso del messaggio ricevuto. In generale la struttura del messaggio viene mostrata dalla figura 2.2.

Command word	sensor information	end report	report information	Reserved	DRC sensor information	CRM 9250N status extension information	CRM 9250N status information	Reserved
(1 byte)	(25 bytes)	(1 byte)	(2 bytes)		(20 bytes)	(20 bytes)	(10 bytes)	(8 bytes)
0x								0x00

Figura 2.2: Struttura di una risposta

In ordine verranno descritte le componenti del messaggio mostrato in figura 2.2:

- **Command word**: è l'identificatore del messaggio, stabilisce qual è l'azione fisica del *device* che il simulatore dovrà andare ad eseguire.
- **Sensor information**: sono inerenti ad informazioni del sensore, solamente un `byte` è rilevante ed indica l'apertura o la chiusura dello *shutter*, ovvero della bocchetta dove entrano o escono le banconote.

- **End report**: indica qual è l'esito dell'operazione, ovvero se di successo, errore o *warning*.
- **Report information**: in base a come i due `bytes` sono valorizzati potranno essere ricavate informazioni aggiuntive sull'errore verificatosi, oppure sul *warning*.
- **Reserved**: parte riservata del messaggio, questa porzione varia da tipologia di messaggio a tipologia di messaggio, i `bytes` definiscono le azioni che avvengono all'interno del dispositivo mentre viene svolta una determinata operazione. In sostanza è l'unica frazione di messaggio che varia di lunghezza a seconda della richiesta da soddisfare.
- **DRC sensor information**: informazioni sui sensori, non sono rilevanti ai fini della costruzione del messaggio, infatti tutti i `bytes` avranno valore pari a 0x00.
- **CRM9250N status extension information**: informazioni dello stato dei cassetti, il primo `byte` riguarda lo stato del cassetto in posizione AC2 poi seguono le posizioni nel range 5-16 (in totale si hanno 16 slot disponibili per l'inserimento dei cassetti).
- **CRM9250N status information**: informazioni riguardo la presenza di banconote nel *Note Feeder*, lo stato del *Note Escrow*, lo stato del cassetto in posizione LRB, lo stato del cassetto in posizione AC1 e dei cassetti nelle posizioni nel range 1-4.

Osservando la struttura appena descritta si può dedurre che esistono due frammenti che compongono il messaggio, il primo costituito dalla porzione **Reserved**, il secondo invece da tutto il resto. Per quanto riguarda il primo, si è già detto che al suo interno sono presenti le diverse istruzioni che caratterizzano i diversi messaggi; mentre nel secondo sono contenute informazioni che riguardano lo stato del dispositivo, infatti tutto il necessario per generare questa porzione è contenuto all'interno della classe `DeviceStatusModel`.

Ragionando su questo è stato possibile dividere in due il lavoro di costruzione del messaggio finale, infatti:

- Per la costruzione della parte *Reserved* sono state create apposite classi chiamate *Rules*, **regole**, in modo tale che queste si preoccupassero solamente di andare a generare la parte riservata del messaggio. In generale l'obiettivo della *Rule* è quello di mappare ogni singola richiesta proveniente dal *Service Provider* occupandosi unicamente di due cose: cambiare lo stato del *device*, ed in base al cambiamento avvenuto generare il *Reserved* corrispondente andando a valorizzare l'attributo presente all'interno della classe `DeviceStatusModel`.

- La restante parte del messaggio, ovvero quella riguardante lo stato, ha come obiettivo di controllare in quale stato si trova il *device* dopo l'esecuzione di una regola; a livello di codice questo si traduce in analizzare le variabili presenti nella classe `DeviceStatusModel` e ricostruire i `byte` nell'ordine previsto dal protocollo durante la generazione della risposta. All'interno della classe `DeviceStatusModel` è presente il metodo il cui compito è quello di generare il messaggio finale, chiamato `Encode` al quale verrà dedicata la sezione 2.3.1.

Quindi riassumendo le fasi di generazione del messaggio si ha:

1. Arriva la richiesta del *Service Provider*.
2. Lettura del messaggio.
3. In base al valore che assume il primo `byte` viene eseguita la regola corrispondente, modificando lo stato, se la richiesta lo prevede.
4. La regola inizializza l'*array* di `byte` della classe `DeviceStatusModel` nominato `Reserved`.
5. La classe `DeviceStatusModel` analizza lo stato delle componenti del simulatore, ricostruisce l'intero messaggio grazie al metodo `Encode`.
6. Il messaggio generato viene poi inviato al *Service Provider*.

2.3.1 Il metodo Encode

Il metodo `Encode` si occupa della costruzione del messaggio che verrà inviato al *Service Provider*; data la lunghezza del metodo il codice verrà analizzato in blocchi. Di seguito, 2.5 si analizza la struttura del metodo in questione.

Listing 2.5: Encode

```
public byte[] Encode()
{
    byte[] arrayOut = null;
    try
    {
    ...
    }
    catch (Exception ex)
    {
        this.log.TraceException("DeviceStatusModel.Encode", ex);
    }
    return arrayOut;
}
```

Si può osservare che il metodo restituisce un vettore di `bytes`, all'interno del blocco `try-catch` è presente la costruzione del messaggio con relativa gestione di eventuali eccezioni che verranno trascritte in un opportuno file di `log` qualora dovessero verificarsi.

Il messaggio si compone di diverse porzioni, di seguito, 2.6, è presente quella della prima parte del messaggio.

Listing 2.6: Encode first part

```
long sizeBufferOut = this.Reserved != null ? LENGTH +
    this.Reserved.Length : LENGTH;
arrayOut = new byte[sizeBufferOut];
int i = 0;
```

```
arrayOut[i++] = (byte) this.RequestDone;
for (; i < 26; i++)
    arrayOut[i] = 0x00;
arrayOut[5] = getShutterOpen(NoteFeeder);
arrayOut[i++] = (byte) this.EndingReport;
ushort arrayByteReportInformation = (ushort)
    this.ReportInformation;
```

Si nota subito che il primo passo è calcolare la lunghezza del messaggio e salvare il valore nella variabile `sizeBufferOut`; il calcolo della lunghezza viene effettuato attraverso operatore ternario dove se il `Reserved` dello stato non è valorizzato allora la lunghezza sarà pari alla costante `LENGTH`, corrispendente al valore 87, mentre se valorizzato la lunghezza totale sarà data dalla somma di `LENGTH` e della lunghezza del `Reserved`. Come si osserva dalla figura 2.2 se il `Reserved` non venisse valorizzato allora la somma complessiva del numero di `bytes` è pari ad 87, invece se fosse valorizzato il numero di `byte` sarebbe ovviamente superiore, per questa ragione è stato scelto di utilizzare una costante a livello di classe chiamata `LENGTH` che assumesse tale valore. Successivamente viene creato un *array* di `byte` di lunghezza calcolata, dove il primo `byte` sarà quello della richiesta proveniente dal *Service Provider* mentre, poi, sarà seguito da tutti *bytes* di valore 0 ad eccezione del sesto, il quale indicherà se lo *shutter* è aperto o chiuso; tale valore verrà letto direttamente dalla componente modellata dal `DeviceStatusModel` attraverso il metodo interno `getShutterOpen`. Infine parte del messaggio verrà completata andando a leggere dallo stato il valore che assumono `EndingReport` e `ReportInfomration`; di default questi sono valorizzati indicando il successo dell'operazione per quanto riguarda il primo, mentre il secondo il fatto che durante l'operazione tutto è andato come doveva.

Segue il frammento `Reserved`, come mostra il codice 2.7.

Listing 2.7: Encode reserved part

```
if (this.Reserved != null)
{
```

```
    Array.Copy(this.Reserved, 0, arrayOut, i,
        this.Reserved.Length);
    i += this.Reserved.Length;
}
```

Si può notare come prima venga eseguito un controllo sulla valorizzazione dell'*array* `Reserved`, successivamente, se avvenuta, viene aggiunto all'*array* che comporrà la risposta e viene aggiornato l'indice per il seguente inserimento delle successive porzioni del messaggio.

Successivamente si hanno le **DRC sensor information**, le quali non influiscono sulla comunicazione tra *device* e *Service Provider*, per questo motivo i successivi 20 `bytes` vengono assegnati con un valore pari a zero come mostrato nel codice 2.8.

Listing 2.8: Encode DRC sensor information

```
for(int j = 0; j < 20; j++)
    arrayOut[i++] = 0x00;
```

Seguendo la costruzione del messaggio, poi, si ha **CRM Status Extension** che, come già detto, al suo interno presenta informazioni sullo stato del cassetto AC2 e dei cassetti nelle posizioni di range 5-16, 2.9.

Listing 2.9: CRM Status Extension

```
Cassette cassetteAC2 = CassetteList.Find(elem => elem.Position ==
    PositionCassette.PositionAC2);
arrayOut[i++] = (byte) cassetteAC2.CassetteStatus; //AC2
for (int j = 5; j < 17; j++)
{
    Cassette cassette = CassetteList.Find(elem => (int)
        elem.Position == j);
    if (cassette != null)
    {
        arrayOut[i++] = (byte)cassette.CassetteStatus;
```

```
    }
    else
    {
        arrayOut[i++] = 0x00;
    }
}
for(int j=0; j<7; j++)
{
    arrayOut[i++] = 0x00;
}
```

Si può osservare come prima operazione l'estrazione del cassetto AC2 nella variabile `cassetteAC2`, eseguita attraverso l'operazione `Find` ponendo come condizione la posizione del cassetto sulla lista `CassetteList` del `DeviceStatusModel`. La medesima cosa viene poi fatta per gli altri cassetti, qualora nello stato in una determinata posizione non è presente alcun cassetto allora a quel `byte` verrà assegnato il valore 0, indicando che lo slot è rimasto vuoto. L'ultimo ciclo `for` è di riempimento in quanto il protocollo prevede 7 `bytes` settati a 0.

Infine l'ultima porzione del messaggio è il **CRM9250N status information**, dove al suo interno sono presenti diverse informazioni riguardo lo stato del simulatore, 2.10.

Listing 2.10: CRM Status Extension

```
arrayOut[i++] = getNoteFeederStatus(this.NoteFeeder,
    this.NVOpened, this.UpperTrack);
if (NoteEscrow.IsEmpty)
{
    arrayOut[i++] = 0x80;
} else
{
    arrayOut[i++] = 0x00;
}
```

```
Cassette lrb = CassetteList.Find(elem => elem.Position ==
    PositionCassette.PositionLRB);
arrayOut[i++] = getLRBCassetteStatus(CassetteList.Find(elem =>
    elem.Position == PositionCassette.PositionLRB),
    this.LowerTrack);
Cassette cassetteAC1 = CassetteList.Find(elem => elem.Position ==
    PositionCassette.PositionAC1);
arrayOut[i++] = (byte)cassetteAC1.CassetteStatus; //AC1
for (int j=1; j < 5; j++)
{
    Cassette cassette = null;
    cassette = CassetteList.Find(elem => (int) elem.Position == j);
    else if(cassette != null)
    {
        arrayOut[i++] = (byte) cassette.CassetteStatus;
    }
    else
    {
        arrayOut[i++] = 0x00;
    }
}

arrayOut[i++] = 0x80; //DRC support command

for (int j = 0; j < 8; j++)
    arrayOut[i++] = 0x00;
```

Nel codice appena visto si nota come il primo `byte` dipenda dallo stato del *Note Escrow*, infatti se vuoto il valore sarà pari a 0x80, mentre quando non lo sarà si avrà 0x00. Segue poi lo stato del cassetto LRB, verrà ricavato grazie al modulo `getLRBCassetteStatus` che genererà il valore del `byte` del protocollo

corrispondente allo stato in cui il cassetto si trova. Segue il cassetto AC1 e i cassetti nelle posizioni di range 1-4; come nel caso precedente, ad ogni iterazione del ciclo `for` verrà assegnato il cassetto con posizione pari all'indice nella variabile `cassette`, se esiste, nell'*array* verrà inserito il `byte` corrispondente allo stato, mentre se il cassetto non è presente si avrà un valore pari 0x00. Infine abbiamo un *DRC support command* valorizzato sempre a 0x80 ed infine otto `byte` di riempimento pari a 0x00.

2.3.2 Struttura della Rule

Nel capitolo 2.3.1 si è parlato dell'utilizzo dell'*array* `Reserved` per completare il messaggio da restituire al *Service Provider*, ma non è stata affrontata la procedura che ne dà l'origine. Per permettere la costruzione di questa porzione di messaggio è stato deciso di creare quelle che vengono chiamate **regole**, ovvero classi il cui compito è quello sia di produrre la porzione `Reserved` del messaggio sia di modificare lo stato del dispositivo in base a quale sia stata la richiesta proveniente dal *Service Provider*.

Sono state create una serie di *regole* per i messaggi più complessi che devono essere restituiti, il nome delle regole è stato mappato in base al nome delle richieste previste dal protocollo:

- `RuleAcquireCRMStatus`: è la richiesta dello stato del dispositivo, si occupa di informare il *Service Provider* in quale stato il dispositivo si trovi.
- `RuleAcquireVersionNumber`: una delle prime richieste che vengono eseguite nel protocollo, utile ad informare il *Service Provider* delle versioni firmware delle componenti del *device*.
- `RuleInitialization`: viene sia invocata nella fase iniziale del protocollo, sia quando è necessario effettuare un'operazione che riporti il *device* in uno stato di normale funzionamento; richiesta per riportare il dispositivo in uno stato di corretto funzionamento.
- `RuleIoArea`: regola per la richiesta di apertura o chiusura dello *shutter*.

- `RulePrepareWithdrawal`: richiesta prima di un'operazione di dispensazione, vengono eseguite le procedure di preparazione.

- `RuleWithdrawal`: regola per la dispensazione, vengono prese le banconote dai cassetti in base all'importo selezionato dall'utente trasportandole fino al *Note Feeder* con lo *shutter* ancora chiuso.

- `RuleDeliverNote`: regola con cui viene confermata la trasmissione delle banconote, vengono azionate le testine all'interno della zona dove le banconote sono presenti dopo l'operazione di `Withdrawal`.

- `RulePrepareDeposit`: richiesta precedente ad un'operazione di deposito, vengono eseguite le procedure di preparazione.

- `RuleCount`: operazione di conteggio delle banconote che sono state immesse dall'utente all'interno del *Note Feeder* per un'operazione di deposito, l'operazione prevede sia il conteggio che il riconoscimento del taglio; le banconote valide vengono inviate nel *Note Escrow*, quelle false nel cassetto LRB mentre quelle non riconosciute, come pezzi di carta, vengono riportate nel *Note Feeder*.

- `RuleAcquireNoteValidatorInformation`: regola per ottenere informazioni riguardo l'operazione di validazione delle banconote, qui avviene il riconoscimento effettivo, viene valutata la veridicità della banconota, la corrispondenza con il numero seriale, processando 60 banconote alla volta. Se viene richiesto un deposito di 100 banconote, non importa il taglio, abbiamo due sessioni di validazione, la prima da 60 la seconda da 40.

- `RuleDeposit`: regola di deposito, una volta conteggiate e validate le banconote, trovandosi all'interno del *Note Escrow*, verranno depositate nei diversi cassetti del dispositivo a seconda del taglio e della validità della banconota.

- `RuleCancelDeposit`: operazione di cancellazione del deposito, nel momento in cui non viene confermata l'operazione di deposito le bancono-

te vengono espulse e ritornano all'interno del *Note Feeder*, presentandole all'utente.

- `RuleRecoverNF`: regola di ripristino del *Note Feeder*, le banconote che si trovano al suo interno vengono indirizzate verso un cassetto di deposito apposito.
- `RuleRecoverNE`: regola di ripristino del *Note Escrow*, le banconote che si trovano al suo interno vengono indirizzate verso un cassetto di deposito apposito.
- `RuleIdle`: regola che a differenza delle altre non corrisponde a nessuna richiesta del *Service Provider*. Queste regole servono a simulare comportamenti esterni alla macchina come inserimento o estrazione delle banconote da parte dell'utente.

Le regole implementano l'interfaccia riportata in 2.11.

Listing 2.11: Rules interface

```
public interface IRule<TRequest, TDevStatus>
{
    bool Match(TRequest request, TDevStatus status);
    TDevStatus Execute(TRequest request, TDevStatus cloneStatus);
}
```

Ogni regola deve implementare due metodi; il primo servirà per rispondere correttamente al *Service Provider* (poi verrà mostrato in modo più specifico a cosa servirà), chiamato `Match(TRequest request, TDevStatus status)`, i suoi argomenti sono la richiesta e lo stato del dispositivo; il secondo ovvero `Execute` è l'esecuzione dell'operazione effettiva, anche qui gli argomenti sono i medesimi.

Alcune regole, inoltre, possono implementare anche un'altra interfaccia, 2.12.

Listing 2.12: Rules interface behavior

```
public interface IChangeRuleBehavior<TChangeBehavior>
{
```

```
    void ChangeBehavior(TChangeBehavior behaviorVar);
}
```

In questo caso la regola dovrà implementare un altro metodo che servirà a manipolare il comportamento, ovvero a generare condizioni particolari che si discostano dal normale funzionamento, come ad esempio errori.

Esempio Regola: RuleWidthdrawal

Ora verrà mostrata una regola come esempio, in modo tale da capire internamente la loro implementazione. Si prenda come esempio la regola di dispensazione, ovvero le `RuleWidthdrawal`, in quanto risulta essere una delle regole più complesse ma, allo stesso tempo, può dare l'idea di come avvenga il processo di generazione dell'*array* che andrà poi a valorizzare l'attributo `Reserved` nello stato del dispositivo. Di seguito si nota la struttura principale, 2.13.

Listing 2.13: Withdrawal Rule

```
public class RuleWithdrawal : IRule<FwRequest, DeviceStatusModel>,
    IChangeRuleBehavior<RuleWithdawalSettingDetails>
    {
        public DeviceStatusModel Execute(FwRequest request,
            DeviceStatusModel cloneStatus)
        {
            try
            {
                cloneStatus.EndingReport = GrgEndingReport.Success;
                cloneStatus.ReportInformation =
                    GrgReportInformation.ResponseOk;
                cloneStatus.RequestDone = GrgCmdType.Withdrawal;
                cloneStatus.NoteFeeder.BanknotesPresentAtTheSlot =
                    true;
                cloneStatus.NoteFeeder.CountInfoDeposit
            .Add(toNoteFeeder(numNotesDen(request.Withdraw.Notes)));
```

```
            cloneStatus.Reserved =
                generateReserved(cloneStatus, request);
            Manip(cloneStatus);
        }
        catch (Exception ex)
        {
            this.log?.TraceException("RuleWithdrawal.Execute",
                ex);
        }
        return cloneStatus;
    }
    ...
}
```

Si osserva subito che la regola prevede l'implementazione di entrambe le interfacce `IRule` e `IChangeRuleBehavior`. Per prima cosa verrà analizzato il metodo `Execute`: vengono assegnati alle variabili di `EndingReport`, `ReportInformation` e `RequestDone` i rispettivi valori; i primi due per indicare informazioni riguardanti l'esito della richiesta mentre il terzo per specificare il tipo di richiesta. Successivamente si possono notare due cambiamenti di stato, il primo dove si assegna all'attributo `BanknotePresentAtTheSlot` dell'oggetto `NoteFeeder` il valore `true`, con questo viene simulata la dispensazione delle banconote e, allo stesso tempo, viene segnalato il cambiamento di stato del sensore destinato alla segnalazione della presenza di banconote all'interno del *Note Feeder*. Il secondo cambiamento sarà quello di andare ad assegnare, attraverso il metodo `toNoteFeeder`, le banconote all'interno dell'oggetto `NoteFeeder`. A seguire si avrà il metodo `generateReserved` che si occuperà di generare l'*array* di `byte` per la parte riservata del messaggio finale che verrà inviato al *Service Provider*. Infine è presente l'operazione di `Manip` il cui compito verrà mostrato più avanti. Il tutto viene inglobato all'interno di un blocco `try-catch` in modo tale che sia possibile tracciare eventuali eccezioni o errori in un opportuno file di *log*.

Il compito del modulo `generateReserved` è quello di andare a generare la parte riservata del messaggio finale, come già detto; per quanto riguarda l'operazione di dispensazione, quindi, si dovrà preoccupare del come deve essere prodotta. Nella tabella 2.6 si può osservare la struttura della porzione riservata del messaggio per la regola di `Withdrawal`(il `Reserved` presentato è valido solamente per l'operazione di dispensazione, le altre regole presenteranno diverse strutture).

Nome	**N° byte**	**Descrizione**
Amount of notes received in NF	42	Numero totale di banconote provenienti dai cassetti, per ogni cassetto quante banconote sono state dispensate
Compenseted amount of notes rejected	40	Non utilizzata a livello di protocollo
Amount of notes rejected by the cassette	40	Banconote rigettate (ovvero arrivate al cassetto AC piuttosto che al *Note Feeder*) durante l'operazione di dispensazione
Amount of notes rejected received in NE	2	Numero di banconote arrivate al NE piuttosto che dispensate
Amount of notes delivered from the cassette	40	Numero totale di banconote erogate dal cassetto, in quanto alcune possono arrivare al *Note Feeder*, alcune *rejected*, qui abbiamo la somma di tutto
Currency and amount of notes withdrawn	4*N+4	Per ogni *currency* viene indicato il numero di banconote dispensate, **N** indica il numero di tagli di banconote diversi coinvolti nell'operazione
Amount of note received in AC1	2	Banconote inviate in AC1

Amount of note received in AC2	2	Banconote inviate in AC2
Reserved	24	Byte con valore pari a 0x00

Tabella 2.6: Tabella della forma del Reserved dell'operazione Withdrawal

Il modulo `generateReserved` con visibilità `private` andrà ad assegnare all'attributo `Reserved` della classe `DeviceStatusModel` il vettore di `byte` corretto per comporre, poi, il messaggio finale da inviare al *Service Provider*. Ora verrà analizzato in piccoli blocchi il processo di generazione di tale messaggio partendo da 2.6, analizzando a piccoli step i passi che lo andranno a comporre.

Listing 2.14: Withdrawal Rule generateReserved()

```
private byte[] generateReserved(DeviceStatusModel cloneStatus,
    FwRequest request)
{
    byte[] reserved = null;
    try
    {
    ...
    } catch (Exception ex)
    {
        this.log?.TraceException("RuleWithdraw.generateReserved",
            ex);
    }
    return reserved;
}
```

Si osserva che il metodo restituisce un *array* di `byte`, il quale, poi, andrà ad essere utilizzato dal metodo `Execute` contenuto all'interno degli oggetti della classe `DeviceStatusModel`; quindi la prima istruzione eseguita è l'istanziazione

di un vettore di `byte` a `null` chiamato `reserved`, successivamente verrà restituito alla fine del metodo, dopo l'assegnazione all'interno del blocco `try-catch`.

All'interno del blocco `try-catch` viene valorizzato il vettore, come mostrato in 2.15.

Listing 2.15: Note in note feeder

```
List<byte> noteReceivedInNF = new List<byte>();
addPairBytes(request.Withdraw.Notes.Count, noteReceivedInNF);
Dictionary<int,int> reservedDict = null;
reservedDict = numNotesDen(request.Withdraw.Notes);
//NOTE IN NOTE FEEDER 42 BYTES
for (int j = 1; j < 17; j++)
{
    bool flag = true;
    foreach (Cassette cassette in cloneStatus.CassetteList)
    {
        if (reservedDict.ContainsKey(cassette.Denomination) &&
            ((int)cassette.Position == j) && cassette.Category ==
            CassetteCategory.RC)
            {
                flag = false;
                addPairBytes(reservedDict[cassette.Denomination],
                    noteReceivedInNF);
                break;
            }
    }
    if (flag)
    {
        addPairBytes(0, noteReceivedInNF);
    }
}
```

```
for (int j = 0; j < 8; j++)
{
    noteReceivedInNF.Add(0x00);
}
```

Si nota subito l'utilizzo di una lista per generare il `Reserved`; questa scelta è dovuta alla lunghezza variabile del messaggio causata della porzione *Currency and amount of notes withdrawn*, infatti in base a quali sono i tagli delle banconote che vengono erogate si avrà un numero di `byte` diverso. Si può osservare, nella seconda linea di codice, l'inserimento all'interno della lista del numero totale di banconote che sono richieste dall'utente per l'operazione di dispensazione (`addPairBytes` è un metodo ausiliario che permette di inserire due byte alla volta). Si può notare il dizionario `reservedDict` che aiuterà a completare l'operazione, venendo subito assegnato attraverso il metodo `numNotesDen` dove il primo `int` indicherà quale taglio di banconota ci interessa mentre nel secondo il numero di banconote per quel taglio. Le informazioni relative a quante e quali sono le banconote d'interesse per completare l'operazione vengono direttamente acquisite dal messaggio che arriva dal *Service Provider*. Si ha poi un ciclo `for` che permette di completare il riempimento della porzione del messaggio *Amount of notes received in NF*; infatti nel momento in cui vengono soddisfatte le condizioni di categoria del cassetto (di ricircolo, `RC`) e di posizione (uguale all'indice del ciclo) verrà aggiunto alla lista il numero di banconote che dovranno essere erogate da quel cassetto; viceversa se il cassetto non esiste oppure non è coinvolto nell'operazione verranno inseriti due `byte` 0x00. Infine per completare l'operazione è necessario inserire 8 `byte` con valore 0x00.

Segue poi nella composizione il codice 2.16.

Listing 2.16: Compenseted amount of notes rejected

```
//COMPENSED AMOUNT OF NOTE REJECTED BY THE CASSETTE 40 BYTES
for (int j = 0; j < 40; j++)
{
    noteReceivedInNF.Add(0x00);
```

```
}
```

Si è già visto nella tabella 2.6 che questa porzione non viene utilizzata, quindi verrà "riempita" da 40 `byte` settati a 0x00.

Seguendo l'ordine si ha poi *Amount of notes rejected by the cassette*, 2.17.

Listing 2.17: Amount of notes rejected by the cassette

```
//AMOUNT OF NOTE REJECTED IN THE CASSETTE
for (int j = 0; j < 40; j++)
{
    noteReceivedInNF.Add(0x00);
}
```

Qui viene indicato il numero di banconote per ogni cassetto che hanno subito l'operazione di *reject*, ovvero che sono state inviate nel cassetto con categoria `AC_Reject_Withdrawal`. In questo caso i `byte` sono tutti uguali a 0x00 poiché nel comportamento di default del *device* nessuna banconota uscente dai cassetti dovrebbe subire questa operazione. (Si vedrà poi che sarà possibile modificare questo comportamento grazie ad opportune regole che permetteranno la manipolazione del comportamento normale).

Successivamente si hanno i due byte per la sezione *Amount of notes rejected received in NE*, 2.18.

Listing 2.18: Amount of notes rejected received in NE

```
//AMOUNT OF NOTE RECEIVED IN NE
noteReceivedInNF.Add(0x00);
noteReceivedInNF.Add(0x00);
```

In un'operazione normale di dispensazione le banconote non dovrebbero arrivare all'interno del *Note Escrow*, per questo motivo ai `byte` del codice 2.18 viene assegnato il valore 0x00.

Di seguito si ha il codice per la porzione *Amount of notes delivered from the cassette* del messaggio di dispensazione, 2.19.

Listing 2.19: Amount of notes delivered from the cassette

```
//AMOUNT OF NOTE DELIVERED FROM THE CASSETTE (Update cassette
    status count)
for (int j = 1; j < 17; j++)
{
    bool flag = true;
    foreach (Cassette cassette in cloneStatus.CassetteList)
    {
        if ((reservedDict.ContainsKey(cassette.Denomination)) &&
            ((int)cassette.Position == j))
        {
            flag = false;
            addPairBytes(reservedDict[cassette.Denomination],
                noteReceivedInNF);
            break;
        }
    }
    if (flag)
    {
        addPairBytes(0, noteReceivedInNF);
    }
}
for (int j = 0; j < 8; j++)
{
    noteReceivedInNF.Add(0x00);
}
```

In un normale funzionamento del *device* fisico le banconote totali che fuoriescono dal cassetto dovranno corrispondere a quelle che arrivano nel *Note Feeder*, qualora ci fossero banconote che hanno subito un'operazione di *reject* oppure si sono bloccate all'interno del *device* il numero indicato in questa porzione sarà

maggiore rispetto a quello delle *Currency and amount of notes withdrawn*.

Di seguito si ha la parte variabile, ovvero quella in cui viene specificato di che tipo di banconota si tratta e di quale taglio, 2.20.

Listing 2.20: Currency and amount of notes withdrawn

```
//CURRENCY AND AMOUNT OF NOTE WITHDRAW
addPairBytes(reservedDict.Keys.Count, noteReceivedInNF);
foreach (KeyValuePair<int, int> notes in reservedDict)
{
    switch (notes.Key)
    {
        case 5:
        {
            addPairBytes((ushort)Denomination.EUR5TestType,
                noteReceivedInNF);
            addPairBytes(notes.Value, noteReceivedInNF);
            break;
        }
        case 10:
        {
            addPairBytes((ushort)Denomination.EUR10TestType,
                noteReceivedInNF);
            addPairBytes(notes.Value, noteReceivedInNF);
            break;
        }
        case 20:
        {
            addPairBytes((ushort)Denomination.EUR20TestType,
                noteReceivedInNF);
            addPairBytes(notes.Value, noteReceivedInNF);
            break;
```

```
        }
        case 50:
        {
            addPairBytes((ushort)Denomination.EUR50TestType,
                noteReceivedInNF);
            addPairBytes(notes.Value, noteReceivedInNF);
            break;
        }
        case 100:
        {
            addPairBytes((ushort)Denomination.EUR100TestType,
                noteReceivedInNF);
            addPairBytes(notes.Value, noteReceivedInNF);
            break;
        }
        case 200:
        {
            addPairBytes((ushort)Denomination.EUR200TestType,
                noteReceivedInNF);
            addPairBytes(notes.Value, noteReceivedInNF);
            break;
        }
        case 500:
        {
            addPairBytes((ushort)Denomination.EUR500TestType,
                noteReceivedInNF);
            addPairBytes(notes.Value, noteReceivedInNF);
            break;
        }
    }
}
```

Si può osservare che prima di tutto vengono aggiunti due `byte` per indicare qual è il numero totale di tagli di banconote coinvolti all'interno dell'operazione (riga 2), infatti si conteggiano il numero di chiavi presenti all'interno del dizionario `reservedDict` che venne istanziato precedentemente. Successivamente con un `foreach` si scorrono i tagli delle banconote contenuti all'interno del dizionario e con uno `switch`, per ogni chiave, verrà generata una coppia di `byte` indicando il taglio ed un altra coppia, invece, il numero di banconote coinvolte.

Per concludere nel codice 2.21 vengono presentate le ultime porzioni del metodo che completano il messaggio `Reserved`.

Listing 2.21: End of generateReserved

```
//Amount of unknow currency
addPairBytes(0, noteReceivedInNF);
//AC1
addPairBytes(0, noteReceivedInNF);
//AC2
addPairBytes(0, noteReceivedInNF);
//24 byte reserved
for (int j = 0; j < 24; j++)
{
    noteReceivedInNF.Add(0x00);
}
reserved = noteReceivedInNF.ToArray();
```

Qui si fa riferimento alle banconote non riconosciute come tali (es. pezzi di carta) per i primi due `byte`, a quelle dispensate nel cassetto AC1 e nel cassetto AC2 per le due successive coppie; queste tre sezioni avranno tutte valore pari a 0 poiché in un normale comportamento non dovrebbero essere coinvolte. Per concludere il protocollo prevede 24 byte con valore 0. Infine la lista viene trasformata in *array* che andrà poi ad essere assegnato alla variabile `Reserved` dello stato.

L'approccio utilizzato per questo esempio viene poi esteso anche alle altre

regole presenti nel simulatore, esse si occuperanno di andare a generare opportunamente l'*array* di `byte` rispettando il protocollo di comunicazione con il *Service Provider*.

Esempio: Alterazione di una Regola

Nell'esempio della regola `Withdrawal` è stata vista la presenza del metodo `Manip` di cui ancora non si è parlato; il metodo permette la manipolazione del comportamento del *device*, cosicché si possano andare a simulare casi particolari.

Per andare a modificare il comportamento innanzitutto è stato deciso di creare una classe dove sono presenti due attributi che possono essere utilizzati da ogni richiesta di manipolazione; il codice di ritorno da restituire e il codice per le informazioni aggiuntive sull'errore o *warning*. Le regole destinate a modificare il comportamento del dispositivo devono ereditare questa classe poiché per tutte le manipolazioni deve esserci la possibilità di questo cambiamento; la classe in questione è `RuleSettingDetails`, riportata in 2.22.

Listing 2.22: RuleSettingDetails

```
public class RuleSettingDetails
{
    public GrgEndingReport? EndingReport { get; set; }
    public GrgReportInformation? ReportInformation { get; set; }
}
```

Gli attributi `EndingReport` e `ReportInformation` sono facoltativi, in un'operazione di manipolazione per un test non è detto che si voglia modificare codice di ritorno ed informazioni sul codice di ritorno, per questa ragione si è lasciata la possibilità di non dover specificare obbligatoriamente tali attributi.

La classe che si occuperà di contenere le informazioni utili per la manipolazione della regola di `Withdrawal`, ovvero `RuleWithdawalSettingDetails`, è presentata nel codice 2.23.

Listing 2.23: RuleWithdawalSettingDetails

```
public class RuleWithdawalSettingDetails : RuleSettingDetails
```

```
{
    public List<CassetteInfo> CassetteInfos = new
        List<CassetteInfo>();
}
```

Si presenta con un solo attributo, `CassetteInfos`, ovvero una lista di oggetti `CassetteInfo`; questo perché in un'operazione come la dispensazione possiamo manipolare cassetto per cassetto le quantità che vengono erogate all'esterno, come per esempio il numero totale di banconote coinvolte oppure se qualcuna di esse ha subito l'operazione di *reject*. La struttura della classe `CassetteInfo` è mostrata in 2.24.

Listing 2.24: CassetteInfo class

```
public class CassetteInfo
{
    public PositionCassette Position { get; set; }
    public int? TotalDispensed { get; set; }
    public int? ToNoteFeeder { get; set; }
    public int? Rejected { get; set; }
    public int? ToAC { get; set; }
    public CassetteStatus? Status { get; set; }
}
```

La classe presenta una lista di attributi facoltativi, mentre uno solo è obbligatorio, ovvero `Position`, questo perché se si vuole modificare il comportamento di un singolo cassetto è necessario che venga specificata la posizione dove esso si trova. Per quanto riguarda invece gli altri attributi:

- **TotalDispensed**: totale delle banconote che sono fuoriuscite dal cassetto.
- **ToNoteFeeder**: banconote che hanno raggiunto effettivamente il *Note Feeder*.
- **Rejected**: numero di banconote che hanno subito l'operazione di *reject*.

- **ToAC**: banconote che dal cassetto sono fuoriuscite ed arrivate al cassetto AC.
- **Status**: indica eventuali cambiamenti per lo stato del cassetto, se si vuole, ad esempio, simulare che il cassetto sia vuoto dopo l'operazione di dispensazione.

Per ora sono state mostrate solamente le informazioni utili per modificare il comportamento della regola ma non come effettivamente avvenga; per fare ciò all'interno della classe `RuleWithdrawal` di cui abbiamo discusso prima viene utilizzato un attributo specifico, 2.25.

Listing 2.25: Attributo in RuleWithdrawal

```
public RuleWithdawalSettingDetails behavior = null;
```

L'attributo verrà assegnato dall'esterno grazie all'implementazione nella regola dell'interfaccia `IChangeRuleBehavior`, in particolare del metodo chiamato `ChangeBehavior`, 2.26.

Listing 2.26: Attributo in RuleWithdrawal

```
public void ChangeBehavior(RuleWithdawalSettingDetails behaviorVar)
{
    this.behavior = behaviorVar;
}
```

Quando il metodo viene eseguito la variabile `behavior` sarà la variabile che permetterà di modificare il comportamento del dispositivo, al suo interno saranno contenute le informazioni per manipolare il comportamento; il metodo che lo permetterà è il `Manip`. La struttura del metodo è mostrata in 2.27.

Listing 2.27: Metodo Manip

```
private void Manip(DeviceStatusModel status)
{
    try
    {
```

```
        if (this.behavior != null)
        {
            if (this.behavior.EndingReport != null)
                status.EndingReport =
                    (GrgEndingReport)this.behavior.EndingReport;
            if (this.behavior.ReportInformation != null)
                status.ReportInformation =
                    (GrgReportInformation)this.behavior.ReportInformation;
            if (this.behavior.cassetteInfos != null)
            {
                ...
            }
        }
    }
    catch (Exception ex)
    {
        this.log?.TraceException("RuleWithdrawal.Manip", ex);
    }
}
```

Dalla struttura si può vedere la presenza del `try-catch` per la gestione delle eccezioni; successivamente viene fatto un check se l'attributo `behavior` è stato valorizzato in precedenza, se così non fosse allora non andrebbe a modificare il comportamento della regola. Una volta verificato se effettivamente l'attributo è stato valorizzato allora si hanno tre `if` che permettono di controllare se sono stati inizializzati gli attributi per la manipolazione:

- Nel primo `if` viene verificato che l'attributo `EndingReport` sia stato valorizzato, se così fosse allora si andrà a modificare l'attributo dello stato con il medesimo nome.
- Nel secondo `if` viene fatta la medesima cosa ma con un altro attributo ovvero `ReportInformation`.

- Nel terzo `if` viene verificata l'assegnazione di `CassetteInfos`, al suo interno si avranno le informazioni che permetteranno di modificare il comportamento della regola.

Data la complessità della porzione di codice contenuta all'interno del terzo `if` verranno analizzati in frammenti il codice, partendo da 2.28.

Listing 2.28: Metodo Manip

```
foreach (CassetteInfo cassetteInfo in this.behavior.CassetteInfos)
{
    foreach (Cassette cassette in status.CassetteList)
    {
       if (cassette.Position == cassetteInfo.Position)
        {
        ...
        }
    }
}
```

Si itera all'interno della lista `CassetteInfos` dell'oggetto `behavior`, per poi iterare nuovamente al suo interno la lista dei cassetti presenti nel simulatore, quando verrà soddisfatta la condizione per cui la cassetta specificata nella manipolazione coincide con la cassetta all'interno dello stato del dispositivo allora verranno eseguite le operazioni.

Se la condizione `if` è stata soddisfatta si verificano una serie di ulteriori controlli `if`, i quali andranno a monitorare se il test prevede quella specifica manipolazione dell'attributo della regola. Iniziamo dal primo controllo mostrato in 2.29.

Listing 2.29: Metodo Manip

```
if (cassetteInfo.Status != null)
{
    cassette.CassetteStatus = (CassetteStatus)cassetteInfo.Status;
```

```
}
```

Per prima cosa viene verificato se la variabile `Status` è stata assegnata, se così allora verrà modificato direttamente lo stato del cassetto.

Successivamente, seguendo l'ordine con cui il protocollo struttura il *Reserved*, si ha la modifica delle banconote che effettivamente raggiungono il *Note Feeder*; ovvero come si comporta il metodo quando l'attributo `ToNoteFeeder` viene assegnato, 2.30.

Listing 2.30: Metodo Manip

```
if (cassetteInfo.ToNoteFeeder != null)
{
    int index = 2;
    for (int i = 1; i < 16; i++)
    {
        if ((int)cassetteInfo.Position == i)
        {
            addChange(status,index, cassetteInfo.ToNoteFeeder);
        }
        else
        {
            index = index + 2;
        }
    }
}
```

L'idea principale è quella di accedere alla richiesta, già generata, del comportamento normale e modificare i `byte` che la compongono per simulare un comportamento diverso da quello di default. La prima cosa fatta è utilizzare una variabile chiamata `index` per accedere direttamente ai `byte` dentro l'*array* `Reserved` e modificarli; viene assegnato il valore 2 poiché nella struttura del protocollo coincide con la posizione della porzione *Amount of notes received in NF*, vista nel capitolo precedente. A questo punto nel momento in cui l'indice del

ciclo coincide con la posizione del cassetto che vogliamo modificare si andrà ad utilizzare il metodo `addChange` per modificare il valore all'interno del `Reserved` delle banconote che sono giunte al `NoteFeeder`; se invece nella `Position` non è presente nessun cassetto allora non verrà modificato e il modulo si limiterà ad aggiornare l'`index` per la successiva iterazione. La variabile `index` viene aggiornata di due in due in quanto ogni cassetto occupa 2 `byte` nel protocollo di comunicazione, per questa ragione passando al check successivo bisognerà accedere al secondo byte dopo quello corrente.

Il successivo `if` sarà utilizzato per la manipolazione di eventuali banconote rigettate da parte di un cassetto, 2.31.

Listing 2.31: Metodo Manip

```
if (cassetteInfo.Rejected != null)
{
    int index = 82;
    for (int i = 1; i <= 16; i++)
    {
        if ((int)cassetteInfo.Position == i)
        {
            addChange(status, index, cassetteInfo.Rejected);
        }
        else
        {
            index = index + 2;
        }
    }
}
```

In questo caso la struttura è molto simile alla precedente, gli unici cambiamenti sono: `index` che viene valorizzato ad 82 ed il terzo argomento del modulo `addChange` dove si ha `cassetteInfo.Rejected`.

Segue la modifica effettuata sulla sezione in cui vengono definite le banconote totali erogate dal cassetto, 2.32.

Listing 2.32: Metodo Manip

```
if (cassetteInfo.TotalDispensed != null)
{
    int index = 124;
    for (int i = 1; i <= 16; i++)
    {
        if ((int)cassetteInfo.Position == i)
        {
            addChange(status, index, cassetteInfo.TotalDispensed);
        }
        else
        {
            index = index +2;
        }
    }
}
```

L'approccio è il medesimo dei due casi precedenti.

Di seguito si ha la modifica delle banconote che giungono nei cassetti AC, nella maggior parte dei casi in seguito ad una operazione di *reject*. Viene mostrato il codice per simulare tale operazione per il cassetto AC1, il medesimo procedimento viene poi replicato per il cassetto AC2, 2.33.

Listing 2.33: Metodo Manip

```
int numOfCurrency =
    Int16.Parse(status.Reserved[164].ToString("X2"));
int indexAC = 164 + (numOfCurrency * 2 * 2) + 2 + 2;
Cassette cassetteAC1 = status.CassetteList.Find(x => x.Position ==
    PositionCassette.PositionAC1);
if (cassetteAC1.Category == CassetteCategory.AC_Reject_Withdrawal)
```

```
{
    if (cassetteInfo.ToAC != null && cassetteInfo.ToAC != 0)
    {
        cassetteAC1.CassetteStatus = CassetteStatus.CassettePresent;
        addChange(status, indexAC, cassetteInfo.ToAC);
    }
}
else
{
    indexAC = indexAC + 2;
}
```

Si prende la variabile `numOfCurrency` in cui viene controllato all'interno del `Reserved` già generato quali sono i tagli di banconote coinvolti, successivamente si calcola la posizione da cui partire, questo perché la porzione *Currency and amount of notes withdrawn* è variabile; quindi viene calcolato l'indice e assegnato alla variabile `indexAC`. A questo punto si cerca all'interno della lista dei cassetti quello in posizione AC1 e lo si assegna alla variabile `cassetteAC1`. Seguendo il codice si controlla se, nello stato del dispositivo, il cassetto AC1 risulta essere quello di *reject* per l'operazione di dispensazione; se questo check viene superato si verifica che il `ToAC` sia stato assegnato [1], a questo punto viene mutato lo stato del cassetto, questo perché si passa dallo stato in cui il cassetto è vuoto allo stato in cui ci sono presenti delle banconote, ovvero `CassettePresent`. Infine, come nei passi precedenti, si utilizza il metodo `addChange` per modificare il comportamento. Lo stesso passo viene ripetuto poi per il cassetto AC2.

Dopo aver modificato i valori delle banconote che sono state dispensate, si ritorna ai primi `byte` i quali indicavano il numero totale di banconote che sono state dispensate, 2.34, questo perché avendo manipolato il numero delle banconote probabilmente ci sarà stato un cambiamento su quelle totali.

[1] Se si vuole specificare che la banconota rigettata sia stata trasportata nel cassetto AC1 è necessario valorizzare sia *reject* che *ToAC*, in quanto se così non fosse si simulerebbe che una banconota si è inceppata all'interno del *device* durante il trasporto

Listing 2.34: Metodo Manip

```
int newTotalFeeder = 0;
for (int i = 2; i <= 33; i++)
{
    newTotalFeeder +=
        Int16.Parse(status.Reserved[i].ToString("X2"));
}
status.Reserved[0] = byte.Parse(newTotalFeeder.ToString(),
    System.Globalization.NumberStyles.HexNumber);
```

Viene istanziata la variabile `newTotalFeeder`, variabile per indicare il nuovo numero delle banconote che arrivano al *Note Feeder* dopo le operazioni di manipolazione. Si scorre all'interno del ciclo `for` i `byte` dell'attributo `Reserved`, modificato in precedenza dalle alterazioni, infine si calcola il numero totale delle banconote indicate nella sezione *Amount of notes received in NF*. Viene sovrascritto il nuovo numero delle banconote sul vecchio nella posizione con indice 0 dell'*array* `Reserved`.

Aldilà dell'esempio l'approccio utilizzato per la regola `RuleWithdrawal` è stato esteso a tutte le regole, in modo tale che si potessero personalizzare i comportamenti del *device* a proprio piacimento, generando qualsiasi possibile di comportamento da sottoporre al *Service Provider*.

2.3.3 Idle Rule

Un tipo di regola da tenere su cui soffermarsi però è la regola di *Idle*, questo tipo di regola è necessaria poiché è necessario simulare alcuni eventi che non dipendono direttamente dal *device* ma, invece, dipendono dall'utente che interagisce con esso; in particolar modo è necessario simulare l'evento di inserimento delle banconote all'interno della bocchetta apposita e l'evento opposto, ovvero il prelievo delle stesse una volta che vengono erogate.

Per fare ciò, quindi, è stato deciso di creare una regola apposita che si attiva nel momento in cui il *device* è in stato di attesa. Come per le altre regole ha il

metodo `Execute` con il quale viene eseguita la regola, ma in questo caso l'unica cosa che deve essere eseguita è un cambiamento di stato, in modo tale si verifichi la condizione di banconote estratte o, viceversa, inserite, come mostrato nel codice 2.35.

Listing 2.35: Regola di Idle

```
public DeviceStatusModel Execute(FwRequest request,
    DeviceStatusModel cloneStatus)
{
    try
    {
        DeviceStatusModel initialStatus =
            (DeviceStatusModel)cloneStatus.Clone();
        if(cloneStatus.FlagIdle)
            cloneStatus = CheckItemInsert(cloneStatus);
        if (!cloneStatus.Equals(initialStatus))
            this.log?.TraceDbgFormat("RuleIdle.Execute", $"new
                status device = {cloneStatus.ToString()}");
    }
    catch (Exception ex)
    {
        this.log?.TraceException("RuleIdle.Execute", ex);
    }
    return cloneStatus;
}
```

La struttura del metodo presenta un `try-catch`; al suo interno, seguendo il codice, si ha una variabile `initialState` dove viene clonato lo stato attuale del *device*, successivamente con una variabile di flag [2] viene verificata la necessità di porre il *device* in attesa dell'evento. Qualora il dispositivo risul-

[2] presente all'interno dello stato del dispositivo, viene resa *true* una volta che viene aperto lo *shutter*, ovvero quando ci aspettiamo che avvenga o l'inserimento delle banconote o la presa.

tasse in attesa si entra nell'`if` e viene modificato il `cloneStatus`; utilizzando il metodo `CheckItemInserted` avverrà la modifica dell'attributo della classe `DeviceStatusModel` che modella il *Note Feeder*. Successivamente, nel secondo `if`, si verifica se lo stato ha subito un cambiamento, qualora fosse vero allora lo si trascrive sul file di *log*.

Nel codice 2.36 è mostrata l'implementazione del metodo `CheckItemInsert`.

Listing 2.36: CheckItemInsert

```
private DeviceStatusModel CheckItemInsert(DeviceStatusModel status)
{
    if (this.behavior != null)
    {
        switch (this.behavior.Action)
        {
            case ActionIdleType.Nothing:
            {
                status.FlagIdle = false;
                break;
            }
            case ActionIdleType.BanknoteInserted:
            {
                if (status.NoteFeeder.ShutterClosed == false &&
                    status.NoteFeeder.BanknotesPresentAtTheSlot ==
                    false)
                    {
                        status.NoteFeeder.BanknotesPresentInShutterZone
                            = false;
                        status.NoteFeeder.BanknotesPresentAtTheSlot
                            = true;
                        status.FlagIdle = false;
                    }
```

```
                break;
            }
            case ActionIdleType.BanknoteTaken:
            {
                if(status.NoteFeeder.ShutterClosed == false &&
                    status.NoteFeeder.BanknotesPresentAtTheSlot ==
                    true)
                {
                    status.NoteFeeder.BanknotesPresentInShutterZone
                        = false;
                    status.NoteFeeder.BanknotesPresentAtTheSlot =
                        false;
                    status.NoteFeeder.countInfoDeposit.Clear();
                    status.FlagIdle = false;
                }
                break;
            }
            default:
                break;
        }
    }
    return status;
}
```

Si nota un controllo `if` sulla variabile `behavior`, questo perché gli eventi che gestiamo con questa regola devono essere iniettati dall'esterno e bisogna verificare se effettivamente è stato assegnato un comportamento su tale variabile. Quindi nel file di test è necessario inserire un evento in modo tale che il metodo funzioni correttamente. Gli eventi che possono essere simulati sono 3, come si può notare dallo `switch`:

- `Nothing`: simula che l'utente non compie un'azione, ovvero non interagisce

con il *device*, nonostante esso si aspetti il contrario.

- `BanknoteInserted`: viene simulato l'inserimento di banconote all'interno dello *shutter*.

- `BanknoteTaken`: simulazione dell'evento di estrazione delle banconote dallo *shutter* da parte dell'utente.

Ora si considerano i casi all'interno dello `switch`, nel primo, ovvero `Nothing`, non vengono eseguite operazioni, ovvero viene lasciato il *device* nello stato corrente; solitamente questo accade quando l'utente si allontana dal dispositivo nonostante quest'ultimo abbia riportato nello *shutter* o delle banconote non riconosciute, come dei fogli di carta, oppure non sia stata completata l'operazione di deposito. Il caso successivo è l'inserimento delle banconote, `BanknoteInserted`; vengono modificati gli attributi `BanknotesPresentAtTheSlot`, assegnandogli il valore `true`, e `BanknotesPresentInShutterZone`, assegnandogli il valore `false`, entrambi appartenenti dell'oggetto *Note Feeder* dello stato. L'ultimo evento è la simulazione del ritiro delle banconote, `BanknoteTaken`, da parte dell'utente; anche qui si modifica lo stato del *Note Feeder* andando prima a modificare gli attributi `BanknotesPresentInShutterZone` e `BanknotesPresentAtTheSlot` entrambi a `false`, poi viene "ripulita" la variabile `countInfoDeposit`, ovvero la variabile dove si andavano ad inserire le banconote presenti nella zona di erogazione.

2.4 Il Rules Engine

Il cuore del simulatore è il *Rules Engine*, ovvero il sistema che permette l'esecuzione vera e propria delle regole. Il suo obiettivo è quello di scegliere opportunamente la regola che costruirà il messaggio da inviare al *Service Provider*.

Nel *Rules Engine* ci sarà un contenitore di regole, chiamato `RuleContainer`, ovvero un oggetto al cui interno verranno inserite le regole che il simulatore eseguirà. Le regole possono avere tre livelli di priorità che decreteranno quali di queste verrà eseguita per prima; ad esempio se si hanno due regole di dispensazione ma una di queste ha priorità più alta allora quest'ultima verrà presa in considerazione prima per l'esecuzione. I tre livelli sono:

- **Core**: sono le regole di default dell'*engine*, ovvero quelle regole che svolgono il proprio compito nel momento in cui il *device* fisico si comporta come da *default*, senza manipolazioni o comportamenti singolari.
- **Priority**: sono le regole invece che vengono inserite all'interno dell'*engine* dall'esterno per permettere un comportamento diverso a quello di *default* del *device*. Nel capitolo precedente si è visto il metodo `Manip` che permette l'eventuale modifica delle regole.
- **Unknown**: sono le regole la cui priorità non è definita e di conseguenza non sarà possibile una loro esecuzione.

Per implementare questi livelli di priorità è stato creato un `enum` apposito denominato `RuleLevel`, 2.37.

Listing 2.37: RuleLevel Enum

```
public enum RuleLevel
{
    Unknown = -1,
    Core = 0,
    Priority = 1,
}
```

Di seguito viene mostrata la classe `RuleContainer` che avrà l'obiettivo di fornire un contenitore di regole per la loro esecuzione, 2.38.

Listing 2.38: RuleContainer class

```
class RuleContainer<TRequest, TDevStatus> where TDevStatus :
    ICloneable
{

    public RuleLevel Level;
    public int Delay { get; set; } = 0;
    public int NumberOfRepetitions { get; set; } = -1;
    public IRule<TRequest, TDevStatus> Rule { get; set; }

    public RuleContainer(IRule<TRequest, TDevStatus> Rule, int
        Delay, int NumberOfRepetitions, RuleLevel level)
    {
        this.Level = level;
        this.Rule = Rule;
        this.Delay = Delay;
        this.NumberOfRepetitions = NumberOfRepetitions;
    }

    public IRule<TRequest, TDevStatus> returnRule()
    {
        return this.Rule;
    }
}
```

Si ha una classe generica, al suo interno può essere specificato il livello della regola con l'attributo `Level`, questo attributo permetterà di scegliere il livello, ovvero se la regola sarà `Priority`, `Core` o `Unknwon`. L'attributo successivo è `Delay` che indica il ritardo in millisecondi con cui la regola verrà eseguita. Suc-

cessivamente si ha l'attributo `NumberOfRepetitions` che indicherà il numero di ripetizioni, ovvero quante volte la regola dovrà essere eseguita; ad esempio se assume valore 1 la regola verrà eseguita una sola volta, mentre se avrà valore pari a 2 due volte e così via. Se uno volesse eseguire la regola infinite volte dovrà assegnare a `NumberOfRepetitions` il valore 0, mentre se invece da un valore superiore si arriva allo zero, come ad esempio da 1 si passa a 0, la regola verrà scartata dall'*engine*. L'ultimo attributo è la `Rule`, di cui è stato mostrato un esempio nel capitolo precedente con la `RuleWithdrawal`.

Ora verrà analizzato qual è il vero compito del *Rules Engine* attraverso una classe omonima `RulesEngine`, 2.39.

Listing 2.39: Rules Engine

```
class RulesEngine<TRequest, TDevStatus> where TDevStatus :
    ICloneable
{
    public List<RuleContainer<TRequest, TDevStatus>> RuleList =
        null;
    private TDevStatus deviceStatus;
    private Logger log = null;
    public RulesEngine(TDevStatus deviceStatus, Logger logVar)
    {
        initialize(deviceStatus, new List<RuleContainer<TRequest,
            TDevStatus>>(), logVar);
    }
    public RulesEngine(TDevStatus deviceStatus,
        List<RuleContainer<TRequest, TDevStatus>> ruleList, Logger
        logVar)
    {
        initialize(deviceStatus, ruleList, logVar);
    }
    private void initialize(TDevStatus deviceStatus,
```

```
    List<RuleContainer<TRequest, TDevStatus>> ruleList, Logger
    logVar)
{
    this.deviceStatus = deviceStatus;
    this.ruleList = ruleList;
    this.log = logVar;
}

public TDevStatus DeviceStatus
{
    get
    {
        TDevStatus status;
        lock (this)
        {
            status = (TDevStatus)deviceStatus.Clone();
        }
        return status;
    }
}

public void AddRule(IRule<TRequest, TDevStatus> rule, int
    delay, int numberOfRepetions,
    RuleContainer<TRequest,TDevStatus>.RuleLevel level)
{
...
}

public TDevStatus Evaluate(TRequest request)
{
...
```

```
    }


    }
}
```

La lista di attributi della classe:

- **RuleList**: è la lista contenente le regole, al suo interno si hanno sia regole `Core` che regole `Priority`. La visibilità è pubblica in quanto sarà possibile dall'esterno aggiungere regole e andare a modificare la lista che le contiene.
- **deviceStatus**: è lo stato del dispositivo, visibilità privata poiché lo stato potrà solamente essere modificato attraverso le regole (le variabili private vengono contrassegnate dalla lettera minuscola iniziale).
- **log**: il *logger* sviluppato dall'azienda per tracciare eventuali errori o eccezioni all'interno di opportuni file.

Successivamente si hanno due costruttori i quali richiamano il metodo privato `initialize` della classe e hanno il compito di inizializzare i tre attributi sopra citati. Un altro attributo presente è il `DeviceStatusModel`, accessibile solamente in lettura con il metodo `get`, il cui compito sarà quello di restituire lo stato del dispositivo, clonato; l'accesso a tale risorsa sarà esclusivo in quanto l'esecuzione del metodo è gestito dall'istruzione `lock`.

Per completare la descrizione del *Rules Engine* bisogna analizzare altri due metodi, ovvero:

- `AddRule`: il metodo adibito all'aggiunta delle regole nella `RuleList`.
- `Evaluate`: il metodo per l'esecuzione effettiva delle regole.

Il primo metodo `AddRule`, 2.40, permette l'aggiunta delle regole nella lista `RuleList`, dà la possibilità di aggiungere quelle componenti che consentono di costruire una risposta adeguata che deve poi essere inviata direttamente al *Service Provider*, in modo tale ci sia uno scambio di messaggi coerente con quello che è il protocollo di comunicazione.

Listing 2.40: Metodo AddRule

```
public void AddRule(IRule<TRequest, TDevStatus> rule, int delay,
    int numberOfRepetions,
    RuleContainer<TRequest,TDevStatus>.RuleLevel level)
{
   lock(this)
   {
        RuleContainer<TRequest, TDevStatus> newRule = new
            RuleContainer<TRequest, TDevStatus>(rule, delay,
            numberOfRepetions, level);
        this.RuleList.Add(newRule);
    }
}
```

Il metodo in questione si occupa di aggiungere all'interno della `RuleList` la regola; come parametri è possibile specificare il `delay`, `level`, `numberOfRepetitions`. L'aggiunta della regola può essere fatta solamente bloccando la risorsa con una `lock`, l'ultima istruzione del metodo è la `Add` per l'aggiunta effettiva nella `RuleList`.

L'altro metodo da andare ad analizzare è il `Evaluate`, 2.41.

Listing 2.41: Metodo Evaluate

```
public TDevStatus Evaluate(TRequest request)
{
    lock(this)
    {
        try
        {
            RuleContainer<TRequest, TDevStatus> ruleProcessed =
                null;
            foreach (RuleContainer<TRequest, TDevStatus>
                ruleElement in this.ruleList)
            {
```

```
            if (ruleElement.Rule.Match(request,
                this.deviceStatus))
            {
                TDevStatus newDeviceStatus =
                    ruleElement.returnRule().Execute(request,
                    (TDevStatus)this.deviceStatus.Clone());
                if (newDeviceStatus != null)
                    this.deviceStatus = newDeviceStatus;
                else
                    this.log?.TraceErrFormat("RulesEngine.Evaluate",
                        "Error. Rule has given back a null status
                        !!!!!!!!");
                ruleProcessed = ruleElement;
                break;
            }
        }

        if (ruleProcessed != null)
        {
            if (ruleProcessed.NumberOfRepetitions > 0)
            {
                ruleProcessed.NumberOfRepetitions--;
                if(ruleProcessed.NumberOfRepetitions == 0)
                    this.ruleList.Remove(ruleProcessed);
            }
        }
    }
    catch(Exception ex)
    {
        this.log?.TraceException("RulesEngine.Evaluate", ex);
    }
```

```
    }
    return (TDevStatus) this.DeviceStatus.Clone();
}
```

Anche per questo metodo viene effettuata l'operazione di `lock`, al suo interno si ha un `try-catch` per gestire eventuali errori o eccezioni che possono verificarsi nel codice sottostante. A questo punto viene istanziata una variabile nominata `ruleProcessed` in cui poi verrà salvata la regola eseguita dall'*engine*. Successivamente è presente un `foreach` che scorrerà gli elementi all'interno della `RuleList`; una volta che viene effettuato correttamente il `Match` della regola, ovvero l'*engine* ha trovato una regola corrispondente alla richiesta proveniente dal *Service Provider* allora si entrerà nell'`if`. In 2.42 viene mostrato il metodo `Match`, ovvero il metodo all'interno della regola che permette al simulatore di scegliere correttamente quella da eseguire.

Listing 2.42: Esempio metodo Match

```
public bool Match(FwRequest request, DeviceStatusModel status)
{
    bool bMatch = false;
    if (request.Command == FwRequest.RequestType.Withdrawal)
    {
        bMatch = true;
    }
    return bMatch;
}
```

Il metodo utilizza una variabile booleana `bMatch` per verificare che il match sia avvenuto, successivamente si ha un confronto tra il codice della richiesta, `request.Command` e il codice della regola, `FwRequest.RequestType.Withdrawal` (in questo caso); se il match va a buon fine viene restituito complessivamente il valore `true`, che permetterà l'entrata nell'`if` nel metodo precedente. I vari codici delle regole sono stati inseriti all'interno di un `enum` per facilitare la scrittura dei metodi di match, 2.43.

Listing 2.43: Enum Request

```
public enum RequestType
{
    Idle = 0,
    PrepareForWithDrawal = 1,
    Withdrawal = 2,
    DeliverNote = 3,
    MoveShutter = 4,
    AcquireCRMStatus = 5,
    Count = 6,
    AcquireNoteValidatorInformation = 7,
    Deposit = 8,
    CancelDeposit = 9,
    RuleRecoverNF = 10,
    RuleRecoverNE = 11,
    Initialization = 12,
    AcquireVersionNumber = 13
}
```

Proseguendo nel codice 2.41 con il metodo `Evaluate`, 2.41, si l'entrata nell'`if` dove all'interno della variabile `newDeviceStatus` viene salvato il nuovo stato del dispositivo dopo l'esecuzione del metodo `Execute`. Qualora questo metodo non restituisse nulla si andrà a scrivere nel file di *log* l'errore, mentre se invece la regola viene eseguita correttamente si andrà a valorizzare lo stato del dispositivo con il nuovo stato. Come ultima istruzione si valorizza la variabile `ruleProcessed` con la `ruleElement`.

L'ultima parte del metodo viene utilizzata per aggiornare il numero di ripetizioni una volta che la regola è stata eseguita, infatti qualora il numero di ripetizioni fosse maggiore di 0 allora si andrà a decrementare il valore della variabile `numberOfRepetitions`, quando assume valore pari a 0 la regola verrà eliminata dalla `RuleList`. Alla fine viene restituito lo stato del dispositivo clonato.

	Tipo	**Descrizione**
`RequestDone`	`GrgCmdType`	Identificatore della regola che deve essere eseguita (0x00)
`EndingReport`	`GrgEndingReport`	Codice di ritorno: Successo, Errore, Warning. (`Success`)
`ReportInformation`	`GrgReportInformation`	Tipo di errore/warning oppure risposta ok (`ResponseOk`)
`Reserved`	`byte[]`	Parte riservata del messaggio, si differenzia per ogni diversa richiesta (`null`)
`CassetteList`	`List<Cassette>`	Lista dei cassetti del device
`NVOpened`	`bool`	Indica se il Note Validator è stato aperto (`false`)
`LowerTrack`	`CassetteTrack`	Intero blocco dei cassetti
`UpperTrack`	`UpperTrack`	Blocco dei cassetti superiori della macchina
`NoteFeeder`	`NoteFeeder`	Area dove vengono inserite/dispensate le banconote
`NoteEscrow`	`NoteEscrow`	Cassetto di deposito temporaneo

Tabella 2.1: Tabella che elenca tutti gli attributi della classe che modellano lo stato del device

	Tipo	**Descrizione**
`CountryCode`	`byte`	Identificatore del *country code* del cassetto
`Position`	`PositionCassette`	Posizione del cassetto nel *device*
`Denomination`	`int`	Il taglio di banconote che il cassetto accetta
`CassetteStatus`	`CassetteStatus`	Stato del cassetto (Ok, Errore, Vuoto, Pieno ...)
`CassetteCategoy`	`CassetteCategory`	Categoria del cassetto
`BanknoteLevel`	`BanknoteCategroyLevel`	Tipo di banconote accettate dal cassetto (vere, false, sospette di falsità)
`FirmwareVersion`	`string`	Versione firmware del cassetto
`NotesInfo`	`Dictionary<Denomination, int>`	Dizionario per la singola transazione, indica le banconote e il numero di banconote coinvolte

Tabella 2.2: Tabella degli attributi per modellare lo stato di un singolo cassetto

	Tipo	**Descrizione**
`BanknotePresentInShutterZone`	`bool`	Sensore che indica la presenza di banconote nella *Shutter Zone*
`BanknotePresentAtSlot`	`bool`	Sensore che indica la presenza di banconote nella *Slot Area*
`BanknotesWrongPositionIntheSlot`	`bool`	Sensore che indica un posizionamento errato delle banconote
`ShutterClosed`	`bool`	Indica se lo shutter è chiuso (`true`) o meno (`false`)
`CountInfoDeposit`	`List<CountInfo>`	Banconote che si trovano nel Note Feeder

Tabella 2.3: Tabella degli attributi per modellare il Note Feeder

	Tipo	**Descrizione**
`NoteInfos`	`Dictionary<Denomination, int>`	Dizionario in cui viene utilizzata come chiave la `Denomination` della banconota, come valore il numero di banconote di quel taglio
`BanknoteCategoryLevel`	`BanknoteCategoryLevel`	Categoria della banconota, L4A, L4B, L2, L3
`Direction`	`NoteDirection`	Indica la direzione dove la banconota finirà

Tabella 2.4: Tabella degli attributi di CountInfo

	Tipo	**Descrizione**
`IsEmpty`	`bool`	Indica se il *Note Escrow* è vuoto (`true`) oppure no (`false`)
`CountInfoDeposit`	`List<CountInfo>`	Banconote che si trovano nel *Note Escrow*

Tabella 2.5: Tabella degli attributi del Note Escrow

Capitolo 3

Risultati: esecuzione dei test

Fino ad ora è stato presentato tutto ciò che riguarda il simulatore, ovvero le procedure e le soluzioni che sono stato implementate per imitare il comportamento del *device*. Nel presente capitolo verrà mostrata, invece, la gestione della scrittura dei test, approfondendo l'utilizzo di **Robot Framework**, del perché è stato scelto ed anche un esempio di test completo; quest'ultimo sarà composto sia dalle casistiche più comuni sia da alcuni comportamenti particolari che il *device* può assumere. Verrà mostrato, anche, come avviene l'inserimento delle regole, lato *Python*, nel simulatore. La scrittura delle regole per le manipolazioni, inoltre, ha previsto l'utilizzo del Design Pattern **Builder**, questa scelta verrà motivata successivamente.

3.1 Test con Robot Framework

Robot Framework, 3.1, è uno strumento che permette l'automatizzazione dei test, facilitarne la scrittura grazie alla presenza di parole chiave, permettendo di riassumere grandi porzioni di codice che andrebbero poi ad appesantire tutto il processo.

Oltre a questo, *Robot Framework* può essere integrato con librerie esterne, quindi non risulta essere un ambiente chiuso. Nel caso presentato l'azienda Sigma S.p.A. ha già modellato le operazioni del modulo software XFS (*Service Provider*)

Figura 3.1: Logo Robot Framework

in un'apposita libreria, ovvero le azioni, che tramite il test, il simulatore dovrà eseguire sono già riassunte in apposite parole chiave.

```
*** Settings ***
Documentation     Test suite for CRM9250 dispenser: Dispense tests
Metadata          Version             0.0.1
Metadata          Service Provider    CDM
Metadata          Service Provider    CIM
Metadata          XFS VERSION         3.20

Resource          Simulator_GRG9250N.resource
Resource          CDM_MODULE.resource
Resource          CIM_MODULE.resource
Variables         Dispense_var.py
Suite Setup       StartUp Sequence
Suite Teardown    ShutDown GRG

*** Test Cases ***

Dispense 150€
    [Tags]  SUCCESS
    [Documentation]    Dispensa 150€ (3 banconote da 50€)
    CDM_MODULE.Dispense by Amount    RequiredAmount=150                    CheckDispenseEvt=${DispenseCheckList_Test1}  CheckCashUnit=${AccountingCheckList_Test1}
    CDM_MODULE.Present               CheckXfsPosition=${PresentCheckList_Test1}
    CDM_MODULE.WaitEvent             EventName=${CdmItemTakenEvt}

*** Keywords ***
StartUp Sequence
    CDM_MODULE.Select Logical Service              LogicalService=AUTOMATION_CDMCRM9250N   ProcessToKill=${Process}
    Simulator_GRG9250N.Start GRG9250N Simulator    Dev_HwConfig=CRM9250N_CDM_Dispense        Dev_Behaviour=CRM9250N_CDM_Dispense
    CDM_MODULE.Force Kill Processes
    CDM_MODULE.Start Service Provider


ShutDown GRG
    CDM_MODULE.ShutDown Procedure
    Simulator_GRG9250N.ShutDown Simulator
```

Figura 3.2: Esempio test

In figura 3.2 possiamo notare la struttura di un test utilizzando *Robot Framework*, evidenziate in azzurro le tre zone principali in cui si divide il test:

- `Settings`: è la porzione in cui vengono *settate* le dipendenze del test, cosa deve essere importato, le versioni delle varie componenti. Si può notare la voce `Documentation` che indica il nome del test nella documentazione, di seguito sono presenti i `Metadata`, come la versione del simulatore 0.0.1, la versione del *Service Provider* da integrare, in questo caso abbiamo sia CDM sia CIM, rispettivamente il primo inerente alle operazioni di dispensazione mentre il secondo per le operazioni di deposito, per ultimo la versione del

modulo XFS, in questo caso 3.20. Proseguendo in questa sezione si hanno le `Resource`, ovvero le risorse da caricare affinché il test funzioni correttamente, in questo caso il simulatore, i moduli CIM e CDM. Di seguito ci sarà il file delle variabili definito nella voce `Variables`, in cui saranno contenute le variabili che verranno utilizzate all'interno del test. Infine come ultime due voci abbiamo `Suite Setup`, ovvero per specificare il *setup* da utilizzare, in questo caso verrà utilizzata una parola chiave appropriata, ed infine `Suite Teardown` ovvero cosa dovrà accadere una volta concluso il test, anche qui una parola chiave.

- `Test Cases`: sono i casi di test, all'interno di questa sezione verranno inseriti tutti i test che il simulatore dovrà affrontare durante la sessione. In viola si ha il nome del test; di seguito l'etichetta `Tags` viene utilizzata per assegnare un *tag* al test; con `Documentation` viene assegnata una documentazione. In seguito, in giallo, sono presenti le operazioni vere e proprie che verranno richieste al simulatore, ognuna delle quali potrà avere delle variabili di colore arancione, ovvero delle informazioni aggiuntive all'operazione oppure dei check che vengono eseguiti; questo per verificare lo stato del dispositivo o la corretta esecuzione del comando. Si può notare che le variabili hanno il simbolo $ seguito dalle parentesi graffe dove all'interno sarà presente un nome di variabile, il suo valore verrà estratto dal file specificato nella sezione precedente alla voce `Variables`.

- `Keywords`: sono le parole chiave che verranno utilizzate all'interno del test; come si può osservare sono le stesse utilizzate nel `Suite Setup` e `Suite Teardown`. Per quanto riguarda la *keyword* `StartUP Sequence` al suo interno vengono definite le operazioni per definire il *setup* del simulatore, ad esempio la configurazione hardware, mentre in `ShutDown GRG` si hanno le operazioni che permettono di chiudere il processo in cui il simulatore viene eseguito e liberare la memoria.

Quella che è stata rappresentata è la struttura generale di un test, la quale ovviamente può essere arricchita aggiungendo a proprio piacimento un numero

variabile di `Test Cases`. Ciò che però ancora non è stato visto è come viene costruito un file delle variabili, ovvero il file che permette di effettuare eventuali controlli per verificare che le operazioni si sono concluse correttamente.

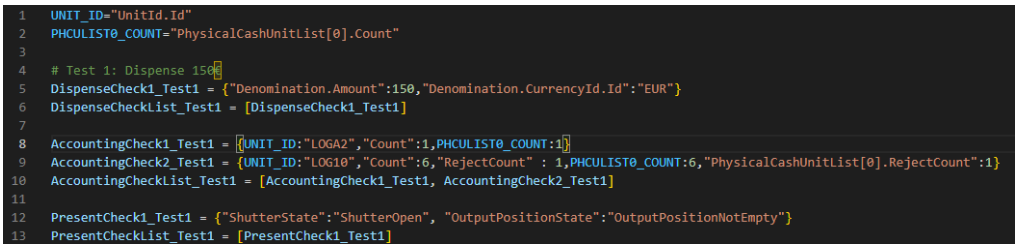

Figura 3.3: Variabili di un test

In figura 3.3 è presente un esempio di come potrebbe essere strutturato un file che contiene le variabili. Nel file si può osservare la presenza, nelle righe 1,2, di due variabili `UNIT_ID` e `PHCULIST0_COUNT` che sono un'abbreviazione di ciò che è scritto di seguito tra virgolette. Successivamente si hanno degli esempi che mostrano come sono costruite le variabili:

- `DispenseCheck1_Test1`: è un controllo che viene effettuato durante l'operazione di dispensazione, ovvero viene monitorato l'effettiva erogazione di 150€ ed anche se la valuta delle banconote è l'euro. La variabile poi viene inserita in una lista `DispenseCheckList_Test1`, la quale permetterà l'aggiunta di ulteriori controlli, se fossero necessari.

- `AccountingCheck1_Test1` e `AccountingCheck2_Test1`: controlli effettuati in seguito all'operazione di dispensazione. In questo caso si può notare, come nel primo caso, l'esecuzione un check sulla *cash unit* (cassetto) denominata LOGA2 controllando al suo interno la presenza di una singola banconota, mentre nel secondo caso viene verificato se nel cassetto LOG10, posizione 10, le banconote rimaste sono 6 e se effettivamente è avvenuto il *reject* per una singola banconota. I due elementi vengono inseriti all'interno di una lista `AccountingCheckList_Test1` in cui possiamo inserire un

numero illimitato di controlli, restituiranno successo solamente se tutti i check all'interno della lista andranno a buon fine.

- `PresentCheck1_Test1`: risulta essere un ulteriore tipo di controllo, in questo caso si verifica che lo *shutter* sia effettivamente aperto con `ShutterState` e che allo stesso tempo sia vuoto con `OutputPositionState`, ovvero che non ci siano banconote al suo interno. Anche qui vale il discorso della lista come nei casi precedenti.

3.1.1 Esempio Test Completo

Nel capitolo 3.1 è stata presentata la struttura di un test; ora verrà mostrato invece come viene effettuato un test completo, ovvero un numero elevato di test che permettano di monitorare se molte delle casistiche possibili siano effettivamente gestite dal modulo XFS. Le casistiche del test sono state accordate con gli ingegneri dell'azienda in modo tale che si potessero considerare un numero elevato di situazioni possibili.

Verranno analizzate di volta in volta, test per test, le operazioni del file per comprendere meglio quali sono i casi a cui il simulatore è stato sottoposto e al tempo stesso per verificare che il modulo XFS si sia comportato correttamente. Il primo *test case* è una normale operazione di deposito, in cui però vengono inserite diverse tipologie di banconote, come mostrato in figura 3.4.

```
Normal Deposit
    [Tags]  SUCCESS
    [Documentation]    Test inserting (50x50€ + 20x20€) L4A + (1x20€ + 3x50€) (L3)
    CIM_MODULE.CashInStart
    CIM_MODULE.OpenShutter          ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent            EventName=${CimItemInsertedEvt}
    CIM_MODULE.CloseShutter         ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.CashIn               CheckCashInEvt=${CashInCheckList}  CheckOnInfoEvt=${InfoCheckList}  CheckInputP6Evt=${InputP6CheckList}
    CIM_MODULE.CashInEnd            CheckCashUnit=${CashInEndCheckList}  CheckNotes=${CashInEndNoteNumberCheck}
    CDM_MODULE.Check Accounting     CheckCashUnit=${CheckAccountingCdmList}
```

Figura 3.4: Normale deposito

In questo caso si può notare dalla `Documentation` quali sono le banconote coinvolte nell'operazione, come scritto si hanno 50 banconote da 50€ e 20 banconote da 20€ di tipologia L4A, ovvero quelle valide e riciclabili, e l'inserimento

di 1 banconota da 20€ e 3 da 50€ di tipo L3, ovvero quelle false. Si osserva che la prima operazione è la `CashInStart`, ovvero l'operazione che permette di iniziare il deposito, successivamente viene invocata la `OpenShutter` la quale permetterà l'apertura dello *shutter*. Di seguito la `WaitEvent`, in questo caso il *device* si aspetta che l'utente inserisca le banconote da depositare (quelle indicate nella `Documentation`), quindi sarà necessario lato simulatore modificare il valore del sensore dello *shutter* all'interno del messaggio con l'uso delle regole di *Idle* che sono state presentate nel capitolo 2.3.3. Una volta che il simulatore si accerterà che le banconote sono state correttamente inserite verrà chiuso lo *shutter* con la `CloseShutter`; a questo punto verrà eseguita la `CashIn` che consiste nel conteggio e nella validazione delle banconote ed infine vengono inviate all'interno del *Note Escrow*, quelle valide, mentre quelle false vengono inviate nel cassetto LRB. Il deposito, infine, viene confermato grazie l'operazione `CashInEnd` in cui le banconote presenti all'interno del *NoteEscrow* verranno inviate negli appositi cassetti destinati ad accumulare le banconote di diversi tagli. Infine in questo caso, non è necessaria per completare un deposito, viene eseguito l'operazione di `CheckAccounting` in cui verranno effettuati i controlli sulle quantità di banconote presenti nei diversi cassetti del *device*.

Il test successivo è un'operazione di dispensazione semplice in cui non vengono effettuate manipolazioni o alterazioni, figura 3.5.

Figura 3.5: Dispensazione normale

La dispensazione è di 150€, in particolare erogati con 3 banconote da 50€. In questo caso la prima operazione è la `Dispense by Amount` per indicare che l'operazione appunto è una dispensazione, inoltre alla variabile `RequiredAmount` è stato assegnato il valore 150 per indicare l'importo richiesto dall'utente. Successivamente si ha la `Present` quindi la presentazione delle banconote all'utente;

operazione al cui interno sarà compresa l'apertura dello *shutter*. Infine questo test si conclude con la `WaitEvent`, ovvero il simulatore si aspetta che l'utente prenda le banconote appena erogate.

Seguendo l'ordine ora si ha un'altra operazione di dispensazione, 3.6, in questo caso di 70€ dove, a differenza della precedente, in due cassetti avverrà la *reject*.

```
Dispense 70€ with Reject
    [Tags]  SUCCESS
    [Documentation]    Dispense 70€, Reject from cassette 04 and cassette 10
    CDM_MODULE.Dispense by Amount    RequiredAmount=70                    CheckDispenseEvt=${DispenseCheckList_Test2}  CheckCashUnit=${AccountingCheckList_Test2}
    CDM_MODULE.Present               CheckXfsPosition=${PresentCheckList_Test1}
    CDM_MODULE.WaitEvent             EventName=${CdmItemTakenEvt}
```

Figura 3.6: Dispensazione con reject

Come descritto dalla `Documentation` nei cassetti in posizione 04 e 10 vengono rigettate alcune banconote, per il resto le operazioni che vengono imposte dal test sono le stesse rispetto ad una normale dispensazione; ciò che sarà diverso saranno i controlli che verranno effettuati, infatti nel *check* delle banconote rigettate si avrà un numero diverso da 0.

Il test successivo, anche qui, verifica l'eventuale *reject* in una deposito, 3.7 .

```
Deposit with Rejected
    [Tags]  SUCCESS
    [Documentation]    Test inserting (100€x27 + 500€x1) L4B + (200€x3) L2 with rejected in deposit: 5 banknotes cassette 2, 1 banknote cassette lrb
    CIM_MODULE.CashInStart
    CIM_MODULE.OpenShutter           ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent             EventName=${CimItemInsertedEvt}
    CIM_MODULE.CloseShutter          ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.CashIn                CheckCashInEvt=${CashIn2CheckList}    CheckOnInfoEvt=${InfoCheckList2}    CheckInputP6Evt=${InputP6CheckList2}
    CIM_MODULE.CashInEnd             CheckCashUnit=${CashIn2EndCheckList}
    CDM_MODULE.Check Accounting      CheckCashUnit=${CheckAccounting2CdmList}
```

Figura 3.7: Deposito con reject

In questo caso le operazioni sono le medesime rispetto ad una normale deposito, ciò che cambia sono le banconote coinvolte, in questo caso si hanno 27 banconote da 100€, 1 da 500€ di tipo L4B, ovvero valide ma che non possono circolare, 3 da 200€ di tipo L2, ovvero banconote sospette di falsità. Nella documentazione sono presenti anche informazioni riguardo quante banconote sono state rigettate, rispettivamente 5 nel cassetto in posizione 2 e 1 banconota nel cassetto LRB. Quello che avviene nel *device* fisico, simulato del software, è questo: le banconote valide vengono inviate nel *Note Escrow* mentre quelle sospette

di falsità arriveranno direttamente al cassetto LRB. Quando viene confermata la deposito con la `CashInEnd` allora dal *Note Escrow* si sposteranno nei cassetti adibiti a contenere i diversi tagli.

Il test successivo affrontato dal simulatore è un test di deposito dove si vuole testare il comportamento del software quando il *Note Escrow* viene riempito, ad esempio in seguito a molteplici `CashIn` prima di confermare la deposizione del denaro, 3.8.

```
Deposit with Escrow Full
    [Tags]  SUCCESS
    [Documentation]    Test inserting () and Escrow Full with Error, (200x50€ + 1 paper), (200x50€)
    CIM_MODULE.CashInStart
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent    EventName=${CimItemInsertedEvt}
    CIM_MODULE.CloseShutter  ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.CashIn  ItemRefused=True  CheckCashInEvt=${CashInCheckList_1}  CheckOnInfoEvt=${InfoCheckList_1}   CheckStatus=${StatusCashInCheckList_1}
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent    EventName=${CimItemTakenEvt}
    CIM_MODULE.CloseShutter  ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent    EventName=${CimItemInsertedEvt}
    CIM_MODULE.CloseShutter  ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.CashIn  retCode=ERRCIMTOOMANYITEMS  ItemRefused=True  CheckStatus=${StatusCashInCheckList_2}  CheckCashInEvt=${CashInCheckList_2}
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent    EventName=${CimItemTakenEvt}  Timeout=10000  ExpectedRet=False
    CIM_MODULE.Retract  CheckStatus=${StatusRetractCheckList}  CheckCashUnit=${RetractCUCheckList}
```

Figura 3.8: Deposito con Note Escrow pieno

Per prima cosa nella `Documentation` si notano le banconote protagoniste dell'operazione: abbiamo una prima `CashIn` dove vengono inserite 200 banconote da 50€ ed un foglio di carta, ed una seconda `CashIn` dove vengono inserite 200 banconote da 50€. Il limite massimo di banconote che può contenere il *Note Escrow* del *device* fisico è di 300 banconote. Quello che accade è questo: nella prima `CashIn` tutte le banconote vengono conteggiate, validate ed inviate correttamente verso il *Note Escrow* che a questo punto ne conterrà un numero pari a 200, mentre il foglio di carta viene rispedito al *Note Feeder*. Successivamente con la seconda operazione di `WaitEvent` il *device*/simulatore si aspetterà che l'utente prelevi il foglio. Di seguito l'utente indicherà che vuole depositare altro denaro prima di completare il deposito, quindi verrà invocata una seconda `CashIn`, anche qui di 200 banconote da 50€; il simulatore andrà ad accogliere le 100 banconote possibili da depositare nel *Note Escrow*, mentre le altre 100 verranno rispedite al *Note Feeder*. Il simulatore ritornerà un codice di errore `ERRCIMTOOMANYITEMS` e contemporaneamente verrà effettuato un controllo, grazie a `ItemRefused=True`, che

verificherà la presenza di banconote nel *Note Feeder*. A questo punto lo *shutter* si aprirà per fare in modo che l'utente possa prendere le banconote non accettate con la quarta `WaitEvent`, ma in questo caso viene superato il `Timeout=10000`, quindi viene invocata la `Retract`. La `Retact` è un'operazione invocata quando è necessario *ripulire* sia il *Note Escrow* sia il *Note Feeder*; quindi nel caso del test accadrà questo, prima verrà pulito il *Note Escrow* e le banconote verranno inviate in un cassetto AC apposito, poi avverrà la stessa cosa anche per il *Note Feeder*.

Dato che abbiamo visualizzato l'utilizzo della `Retract` per l'operazione di deposito, ora verrà testata la stessa per una dispensazione, 3.9.

```
Dispense 100€ With Retract
    [Tags]  SUCCESS
    [Documentation]    Dispensa 100€ (2x50€)
    CDM_MODULE.Dispense by Amount  RequiredAmount=100  CheckDispenseEvt=${DispenseCheckList_Test3}  CheckCashUnit=${CheckAccountingDisRet}
    CDM_MODULE.Present  CheckXfsPosition=${PresentCheckList}
    CDM_MODULE.WaitEvent    EventName=${CdmItemTakenEvt}  ExpectedRet=False  Timeout=8500
    CDM_MODULE.Retract
```

Figura 3.9: Dispensazione con Retract

Nella `Documentation` si nota una semplice dispensazione di 2 banconote da 50€ per un complessivo 100€. Quello però che avviene alla fine è che l'utente, quando il simulatore/*device* presenta le banconote all'utente nel *Note Feeder*, questo non le ritira ed allora viene attivata la `Retract` che si occuperà della *pulizia* del *Note Feeder* e le banconote verranno inviate al cassetto AC preposto.

Il test successivo coinvolge una nuova operazione ovvero la **Rollback**, l'operazione che viene eseguita quando l'utente decide di annullare il deposito, 3.10.

Si notano nella `Documentation` le banconote coinvolte, 2 da 20€ e 2 da 50€ L4A, 1 da 10 L3 e un foglio di carta. Quindi come si è visto precedentemente la `CashIn` prevede un oggetto rifiutato che in questo caso è il foglio di carta, inoltre ci si aspetta che il controllo `ItemRefused=True` vada a buon fine. Una volta che l'utente ritira il foglio di carta, espulso dal *Note Escrow*, decide di annullare il deposito invocando la `CashInRollback`; le banconote, che precedentemente erano giunte al *Note Escrow*, non vengono inviate nei cassetti ma vengono riportate al

```
Deposit with Rollback (after item taken)
    [Tags]  SUCCESS
    [Documentation]    Test inserting 16x20€(L4A)
    Cim CashInStart
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    Cim Wait Event  ${CimItemInsertedEvt}
    CIM_MODULE.CloseShutter  ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.CashIn  CheckCashInEvt=${CashInCheckList_Test2}  CheckOnInfoEvt=${InfoCheckList_Test2}
    CIM_MODULE.CashInRollback
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    Cim Wait Event  ${CimItemTakenEvt}
    CIM_MODULE.CloseShutter  ShutterCheck=${CloseShutterCheckList}
```

Figura 3.10: Deposito con Rollback

Note Feeder, mentre le banconote false ovvero le L3, in questo caso, già precedentemente erano state inviate nel cassetto LRB. Il tempo di timeout per recuperare le banconote dal *Note Feeder* viene superato e quindi viene attivata nuovamente la `Retract`.

Ora verrà mostrata un'altra casistica dove viene invocata la `Retract`, in un'operazione di deposito, 3.11.

```
Deposit with Retract
    [Tags]  SUCCESS
    [Documentation]    Test inserting 2x20€(L4A) + 1xPaper [all into reject/retract]
    CIM_MODULE.CashInStart
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent    EventName=${CimItemInsertedEvt}
    CIM_MODULE.CloseShutter  ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.CashIn  ItemRefused=True
    CIM_MODULE.OpenShutter  ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent    EventName=${CimItemTakenEvt}    Timeout=10000    ExpectedRet=False
    CIM_MODULE.Retract
```

Figura 3.11: Deposito con Retract

In questo caso le banconote inserite sono 2 da 20€ di tipo L4A, e un foglio di carta. La `CashIn` viene eseguita correttamente, quindi le banconote vengono portate fino al *Note Escrow* per quanto riguarda le L4A, mentre il foglio di carta viene rispedito al *Note Feeder*. A questo punto però l'utente né conferma il deposito né preleva il foglio di carta espulso, per questa ragione quello che viene fatto da parte del *device* è la `Retract`, ovvero la pulizia sia del *Note Escrow* sia del *Note Feeder*.

Un altro test previsto è l'operazione di deposito dove il cassetto prestabilito per il contenimento di un certo taglio risulta essere pieno, 3.12.

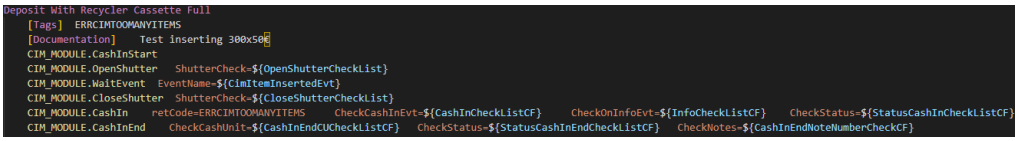

Figura 3.12: Deposito con cassetto pieno

Nella `Documentation` si osserva l'inserimento di 300 banconote da 50€. L'operazione di deposito avviene normalmente, la differenza però la si trova nel codice di ritorno dell'operazione di `CashInEnd` in cui si ha un `ERRHARDWAREERROR`, proprio ad indicare che è avvenuto un errore. Per far ritornare poi il *device*, o il simulatore, in uno stato che sia in grado di accettare ulteriori richieste è necessario che venga eseguita un'operazione di `Reset`. In questo caso l'errore è dato dall'impossibilità di un cassetto di accettare banconote al proprio interno poiché pieno; la gestione di questa casistica prevede che le banconote che non possono più essere inviate all'interno del cassetto e vengano spedite nel cassetto di deposito generico della stessa tipologia, ad esempio L4A. Successivamente il cassetto dovrà essere svuotato.

Seguendo la serie di test a questo punto si ha un'ulteriore operazione di deposito, dove viene simulato il blocco di una banconota durante il trasporto nei rispettivi cassetti, 3.13.

Dalla `Documentation` si nota che l'inserimento interessa 10 banconote da 50€. La deposito delle banconote avviene in modo corretto fino alla `CashInEnd` dove a questo punto si ha il codice di ritorno `ERRHARDWAREERROR`, ovvero viene simulato che una delle banconote si blocchi sui nastri trasportatori. Giunto tale errore il simulatore e il modulo software XFS sono bloccati, sarà necessaria anche qui un'operazione di `Reset` per riportare il *device* in uno stato di corretta esecuzione.

Data la simulazione di una banconota bloccata durante un'operazione di deposito, il passo successivo è simulare lo stesso problema però durante la dispensazione, 3.14.

```
Deposit With BanknotesBlocked
    [Tags]  SUCCESS
    [Documentation]    Test inserting 10x50€ with block in the transport
    CIM_MODULE.CashInStart
    CIM_MODULE.OpenShutter          ShutterCheck=${OpenShutterCheckList}
    CIM_MODULE.WaitEvent            EventName=${CimItemInsertedEvt}
    CIM_MODULE.CloseShutter         ShutterCheck=${CloseShutterCheckList}
    CIM_MODULE.CashIn
    CIM_MODULE.CashInEnd            retCode=ERRHARDWAREERROR
    CIM_MODULE.Reset
```

Figura 3.13: Deposito con blocco della banconota

```
Dispense 40€ With BanknotesBlocked
    [Tags]  ERRHARDWAREERROR
    [Documentation]    Dispensa 40€
    CDM_MODULE.Dispense by Amount    RequiredAmount=40      retCode=ERRHARDWAREERROR
    CDM_MODULE.Reset
```

Figura 3.14: Dispensazione con blocco della banconota

L'operazione in questione prevede la dispensazione di 40€, due banconote da 20€. Si ha un codice di ritorno durante l'operazione di `Dispense by Amount` di tipo `ERRHARDWAREERROR`, con cui verrà simulato il blocco. Anche qui sarà necessaria un'operazione di `Reset` per riportare il tutto in uno stato non di errore.

Infine come ultima operazione del test viene simulata una `Reset` a sé stante in cui viene rilevato un errore, 3.15.

```
Reset with Error
    [Tags]  ERRHARDWAREERROR
    [Documentation]    Errore alla CU CRM02
    CIM_MODULE.Reset    retCode=ERRCIMCASHUNITERROR
```

Figura 3.15: Reset con errore

Vi è un'unica operazione di `Reset` dove verrà restituito un codice di errore del tipo `ERRCIMCASHUNITERROR` che, come suggerisce la `Documentation` è un errore

alla *Cash Unit* 2, ovvero al cassetto in posizione 2.

Si può notare che tutti questi test vengono eseguiti durante un'unica sessione, impiegando circa 2 minuti. Se si dovessero eseguire questi test su un dispositivo fisico reale occorrerebbe una tempistica maggiore in quanto si dovrebbe ogni volta andare ad inserire banconote, oppure imprimere un codice di errore, oppure sbloccare i nastri di trasporto. In conclusione, il simulatore svolgendo tutti i test riesce a coprire un gran numero di casistiche possibili; qualora fosse necessario replicare un particolare evento accaduto in un dispositivo istallato non ci sarebbero problemi nell'andare ricostruire le operazioni e le condizioni di quella circostanza.

3.2 Test con Python

Avendo osservato come avviene la scrittura di un test con l'ausilio di *Robot Framework*, ora verrà mostrato come viene eseguito l'inserimento e la scrittura delle regole per concludere correttamente il test; in questo modo è possibile la simulazione dei diversi casi in cui il *device* può giungere. Prima di osservare la pure scrittura delle regole e il loro inserimento all'interno del *Rules Engine* è necessario discutere di una scelta implementativa, ovvero l'utilizzo del Design Pattern **Builder** per facilitare le scrittura del test; nel capitolo successivo, invece, verranno mostrate le diverse regole inserite nel test seguendo le operazioni descritte all'interno del capitolo 3.1.1.

3.2.1 Builder

Il **Builder** è un Design Pattern creazionale che permette la costruzione di oggetti complessi *step by step* invece che utilizzare il metodo costruttore; questo perché, nel momento in cui vi sono molteplici attributi da inizializzare andare a specificare, con il metodo costruttore, tutti gli attributi potrebbe essere non elegante. Un altra problematica è quella di istanziare un oggetto che non presenta uno o più di questi attributi; normalmente con il costruttore bisognerebbe andare ad assegnare per ognuno di essi il valore `null`, oppure è necessario creare diverse

versioni del metodo costruttore. In entrambi i casi la probabilità di errore risulta essere molto alta oltre al fatto che il codice potrebbe diventare non intuitivo e di difficile comprensione a primo impatto.

Il Builder permette di separare la rappresentazione di un oggetto dalla sua costruzione, in questo modo è possibile andare ad istanziare gli oggetti diminuendo la probabilità di errore, questo poiché per svolgere tale compito sarà presente una classe adibita solo a quello, mentre alla classe originale verrà solamente lasciato l'onere di svolgere correttamente i propri compiti. La costruzione di un oggetto verrà eseguita *step by step* con dei metodi opportuni che avranno il compito di eseguire l'inizializzazione degli oggetti.

Per vedere un esempio di applicazione di questo Design Pattern si mostra l'esempio descritto nel capitolo 2.3.2, dove è presente l'alterazione di una regola, notando come viene applicato il Builder.

Listing 3.1: Interfaccia del Builder

```
public interface IWithdrawalBuilder
{
        RuleWithdrawBuilder AddReturnCode(GrgEndingReport
            endingReport, GrgReportInformation reportInformation);
        RuleWithdrawBuilder AddCassetteInfo(PositionCassette
            position, int totalDispensed = 0, int toNoteFeeder = 0,
            int rejected=0, int toAC = 0, CassetteStatus
            cassetteStatus = 0);
        RuleWithdrawal GetRule();
}
```

Per la manipolazione della regola di `Withdrawal` è necessaria la creazione di un oggetto della classe `RuleWithdrawalSettingDetails`, la quale però al proprio interno ha un numero molto elevato di attributi da andare ad assegnare. Per questa ragione è stato deciso di utilizzare un Builder che andasse a semplificare la costruzione di tale oggetto. Per prima cosa, come riportato nel codice 3.1,

viene utilizzata un'interfaccia `IWithdrawalBuilder` per poter indicare quali sono i moduli da dover implementare all'interno del Builder. Sono presenti tre moduli:

- `AddReturnCode`: si occuperà di andare ad assegnare un codice di ritorno diverso, da quello consueto, quando la regola verrà eseguita assegnando `endingReport`, insieme alle relative informazioni sullo stato di *warning* o errore assegnando `reportInformation`.
- `AddCassetteInfo`: si occuperà di andare a specificare per ogni cassetto qual è il destino che le banconote fuoriuscite da esso avranno. Si può notare che gli argomenti del modulo sono la `position`, ovvero la posizione del cassetto, `totalDispensed`, il numero di banconote uscite dal cassetto, `toNoteFeeder` il numero di banconote che hanno raggiunto il *Note Feeder*, `rejected` il numero delle banconote rigettate, `toAC` il numero di banconote che sono state inviate al cassetto AC e per concludere il `cassetteStatus` per indicare eventuali cambiamenti sullo stato del cassetto.
- `GetRule`: è il metodo che segnala la conclusione della creazione dell'oggetto con cui verrà restituita la regola con tutte le modifiche apportate.

Si può notare che nei primi due metodi dell'elenco viene restituito un oggetto di tipo `RuleWithdrawalBuilder`, questo perché così è consentita la concatenazione dei metodi nel momento di composizione dell'oggetto.

Listing 3.2: Classe Builder

```
public class RuleWithdrawBuilder : RuleSettingDetails,
    IWithdrawBuilder
{
    RuleWithdawalSettingDetails _ruleWithdawalSettingDetails = new
        RuleWithdawalSettingDetails();

    public RuleWithdrawBuilder AddCassetteInfo(PositionCassette
        position, int totalDispensed = 0, int toNoteFeeder = 0, int
```

```
        rejected = 0, int toAC = 0, CassetteStatus cassetteStatus =
        0)
    {
       ...
    }

    public RuleWithdrawBuilder AddReturnCode(GrgEndingReport
        endingReport, GrgReportInformation reportInformation)
    {
        ...
    }

    public RuleWithdrawal GetRule()
    {
        RuleWithdawalSettingDetails ruleWithdawalSettingDetails =
            _ruleWithdawalSettingDetails;
        RuleWithdrawal result = new RuleWithdrawal();
        result.ChangeBehavior(ruleWithdawalSettingDetails);
        return result;
    }
}
```

Nel codice 3.2 si nota la struttura della classe in cui abbiamo un unico attributo `_ruleWithdrawalSettingDetails`, ovvero l'oggetto utile per la manipolazione della regola. Di seguito in modo molto semplice l'implementazione dei tre attributi dell'interfaccia del codice 3.1: `AddCasstteInfo` e `AddReturnCode` verranno trattati in seguito, mentre nel `GetRule` si può notare come venga, oltre l'oggetto citato, istanziato un oggetto della classe `RuleWithdrawal` grazie al quale verrà poi richiamato il metodo `ChangeBehavior`. Al termine verrà restituita la regola modificata, che grazie al metodo `Manip` visto nel capitolo 2.3.2, permetterà di simulare un caso particolare o un comportamento anomalo.

Per quanto riguarda il metodo `AddCasetteInfo`, 3.3, al suo interno vengono assegnate, qualora fossero specificati, gli attributi di un cassetto, questo è permesso grazie all'istanziazione dell'oggetto della classe `CassetteInfo` sul quale avverrà tale operazione. Qualora non fossero specificati gli attributi verranno valorizzati con il valore di default.

Listing 3.3: Metodo AddCassetteInfo

```
public RuleWithdrawBuilder AddCassetteInfo(PositionCassette
    position, int totalDispensed = 0, int toNoteFeeder = 0, int
    rejected = 0, int toAC = 0, CassetteStatus cassetteStatus = 0)
{
    CassetteInfo cassetteInfo = new CassetteInfo();
    cassetteInfo.Position = position;
    if(totalDispensed != 0)
    {
        cassetteInfo.TotalDispensed = totalDispensed;
    }
    if(toNoteFeeder != 0)
    {
        cassetteInfo.ToNoteFeeder = toNoteFeeder;
    }
    if(rejected != 0)
    {
        cassetteInfo.Rejected = rejected;
    }
    if(toAC != 0)
    {
        cassetteInfo.ToAC = toAC;
    }
    if (cassetteStatus != 0)
    {
```

```
        cassetteInfo.Status = cassetteStatus;
    }
    _ruleWithdawalSettingDetails.cassetteInfos.Add(cassetteInfo);
    return this;
}
```

Stesso discorso vale per il metodo `AddreturnCode`, 3.4, dove gli attributi assegnati sono `endingReport`, ovvero il codice di errore, e `reportInformation` ovvero le informazioni sul tipo di errore o *warning*.

Listing 3.4: Metodo AddReturnCode

```
public RuleWithdrawBuilder AddReturnCode(GrgEndingReport
    endingReport, GrgReportInformation reportInformation)
{
    this._ruleWithdawalSettingDetails.EndingReport = endingReport;
    this._ruleWithdawalSettingDetails.ReportInformation =
        reportInformation;
    return this;
}
```

3.2.2 Esempio Test Completo

Una volta presentata la creazione delle regole manipolate attraverso il Builder, verrà mostrato l'esempio descritto nel capitolo inerente a *Robot Framework*, 3.1, però dal lato d'inserimento delle regole nel simulatore. Per fare ciò è stata utilizzata una libreria chiamata *Python.NET*[6] che permette l'integrazione in *Python* dell'ambiente `.NET`, in modo tale che potessero essere utilizzati i metodi dei Builder per l'istanziazione degli oggetti e potesse essere invocato un particolare metodo per il conseguente inserimento. Per l'inserimento, appunto, è stato deciso di creare e di utilizzare un metodo apposito per l'aggiunta delle regole nel simulatore dall'esterno chiamato `AddExternalRule`, 3.5.

Listing 3.5: Metodo AddExternalRule

```
public void AddExternalRule(IRule<FwRequest, DeviceStatusModel>
    newRule , int numRepetition = 0, int delay = 0)
{
    try
    {
        _log.TraceDbgFormat("RecyclerSimulator.AddExternalRule",
            $"INPUT. numRepetion = {numRepetition}, delay =
            {delay}\r\n newRule = {newRule.ToString()}");
        this.rulesEngine.AddRule(newRule, delay, numRepetition,
            RuleContainer<FwRequest,
            DeviceStatusModel>.RuleLevel.Priority);
        this.rulesEngine.ruleList =
            this.rulesEngine.ruleList.OrderByDescending(o =>
            o.Level).ToList();
        _log.TraceDbgFormat("RecyclerSimulator.AddExternalRule",
            $"OUTPUT");
    }
    catch(Exception ex)
    {
        _log.TraceException("RecyclerSimulator.AddExternalRule",
            ex);
    }
}
```

Il metodo in questione ha tre argomenti, il primo la regola ovvero `newRule`, come secondo argomento `numRepetition` ovvero il numero di volte che la regola deve essere eseguita, `delay` per indicare con quanto ritardo la regola deve essere eseguita. All'interno si nota il blocco `try-catch` per catturare eventuali eccezioni che verranno scritte in un opportuno file di *log* qualora avvenissero, come mostrato nel `catch`. Nel `try` si ha, oltre alla scrittura nel file di *log* dell'esecuzione

del metodo, l'invocazione del metodo `AddRule` del `rulesEngine` per l'inserimento della regola nella lista e la chiamata del metodo `OrderByDescending` per ordinare la lista ponendo le regole con più alta priorità in cima, cosicché vengano eseguite prima.

Una volta che viene istanziato il simulatore, a quel punto ci sarà l'inserimento delle regole; prima verranno aggiunte quelle non manipolate, ovvero le **Core**, che andranno a simulare il normale comportamento del *device*, mentre quelle manipolate, ovvero le **Priority**, verranno inserite invocando il modulo `AddExternalRule`, come mostrato nell'esempio, 3.6 .

Listing 3.6: Aggiunta di una regola

```
self._simulator.AddExternalRule(ruleCount, 1, delay=0)
```

L'inserimento della regola avviene passando come primo parametro la regola, in questo caso `ruleCount`, ovvero una regola di conteggio delle banconote, il valore 1 per indicare che l'operazione deve essere ripetuta una sola volta, ed infine il ritardo pari a 0 per indicare che deve essere eseguita immediatamente. L'inserimento avverrà con la stessa modalità anche per le regole successive.

Seguendo il percorso descritto nel capitolo 3.1, la prima operazione che viene eseguita è una normale operazione di deposito; per completarla è necessario specificare le banconote che vengono conteggiate, quindi l'inserimento di una regola di **Count** nel simulatore, 3.7.

Listing 3.7: Regola di Count

```
ruleCountBuilder = RuleCountBuilder()
ruleCountBuilder.AddBanknoteCategory(BanknoteCategoryLevel.L4A)
                .AddBanknotes(BanknoteCategoryLevel.L4A,
                    Denomination.EUR50TestType,
                    50).AddBanknotes(BanknoteCategoryLevel.L4A,
                    Denomination.EUR20TestType, 20)
ruleCountBuilder.AddBanknoteCategory(BanknoteCategoryLevel.L3)
                .AddBanknotes(BanknoteCategoryLevel.L3,
                    Denomination.EUR50TestType, 3)
```

```
                .AddBanknotes(BanknoteCategoryLevel.L3,
                    Denomination.EUR20TestType, 1)
ruleCount = ruleCountBuilder.GetRule()
```

Prima viene istanziato un oggetto Builder `ruleCountBuilder` grazie al quale, attraverso i suoi metodi, vengono assegnati gli attributi della manipolazione: con l'`AddBanknoteCategory` viene aggiunta la tipologia di banconota, con `AddBanknotes` il taglio della banconota e la quantità. In questo caso abbiamo l'aggiunta di 50 banconote da 50€ e di 20 da 20€ per le L4A, mentre per le L3 vengono aggiunte 3 da 50€ e 1 da 20€. Infine per concludere la generazione della regola manipolata viene invocato il `GetRule` che assegna alla variabile `ruleCount` la regola completa.

La seconda operazione eseguita nel test è una normale operazione di dispensazione, in questo caso non è richiesto nessun intervento dall'esterno poiché può essere completata in modo autonomo dal simulatore.

Seguendo il test l'operazione successiva è una dispensazione dove in due cassetti avviene l'operazione di *reject*.

Listing 3.8: Dispensazione con Reject

```
ruleWithdrawReject = RuleWithdrawBuilder()
ruleWithdrawReject.AddCassetteInfo(PositionCassette.Position10,
    totalDispensed =2, rejected=1, toNoteFeeder=1, toAC = 1)
ruleWithdrawReject.AddCassetteInfo(PositionCassette.Position04,
    totalDispensed =2, rejected=1, toNoteFeeder=1, toAC = 1)
ruleWithdrawR = ruleWithdrawReject.GetRule()
```

Si nota che in 3.8, come nel caso precedente, viene istanziato prima un oggetto Builder. Attraverso il metodo `AddCassetteInfo`, grazie al primo argomento, viene specificato il comportamento per i due cassetti coinvolti, ovvero quello in posizione 10 e 4. In entrambi i casi vengono dispensate in totale 2 banconote, rigettate 1, 1 arriverà al *Note Feeder* ed 1 al cassetto AC. Solitamente le banconote rigettate vengono inviate nel cassetto AC preposto, quindi l'1 presente sia nel-

l'attributo `reject` sia per l'attributo `toAC` stanno ad indicare il comportamento della medesima banconota.

Successivamente si ha un'operazione di deposito in cui avviene una *reject*, quindi alcune banconote verranno inviate nel cassetto AC.

Listing 3.9: Deposito con Reject

```
ruleCountBuilder = RuleCountBuilder()
ruleCountBuilder.AddBanknoteCategory(BanknoteCategoryLevel.L4B)
               .AddBanknotes(BanknoteCategoryLevel.L4B,
                  Denomination.EUR100TestType, 27)
               .AddBanknotes(BanknoteCategoryLevel.L4B,
                  Denomination.EUR500TestType, 1)
ruleCountBuilder.AddBanknoteCategory(BanknoteCategoryLevel.L2)
               .AddBanknotes(BanknoteCategoryLevel.L2,
                  Denomination.EUR200TestType, 3)
ruleCount2 = ruleCountBuilder.GetRule()
ruleDeposit2Behavior = RuleDepositBuilder()
ruleDeposit2Behavior.ChangeDepositInCassette(PositionCassette.Position02,
    banknotesInCassette=22, rejected=5)
                  .ChangeDepositInCassette(PositionCassette.PositionLRB,
                     banknotesInCassette=2, rejected=1)
ruleDeposit2 = ruleDeposit2Behavior.GetRule()
```

In 3.9 si nota la costruzione della `ruleCount2` per inserire le banconote coinvolte nel deposito, come nell'esempio 3.7, grazie ai due metodi del Builder `AddBanknoteCategory` e `AddBanknotes`. Di seguito quello che però avviene è una modifica della regola di **Deposit** dove abbiamo delle alterazioni rispetto al normale funzionamento. Il Builder per la manipolazione della `Deposit` permette, attraverso il metodo `ChangeDepositInCassette`, di modificare la destinazione delle banconote; in questo caso per il cassetto in posizione 2, destinato a contenere le banconote di taglio che non vengono però ricircolate, e quello LRB, per le banconote false. Grazie agli gli argomenti `banknotesInCassette` viene spe-

cificato il numero di banconote che arrivano correttamente nel cassetto, mentre con `rejected` quelle destinate al cassetto AC.

Di seguito nel test si ha l'operazione di deposito in cui avviene il riempimento del *Note Escrow*, 3.10.

Listing 3.10: Deposito con riempimento del Note Escrow

```
ruleBuilderE1 = RuleCountBuilder()
ruleBuilderE1.AddBanknoteCategory(BanknoteCategoryLevel.L4A)
             .AddBanknotes(BanknoteCategoryLevel.L4A,
                 Denomination.EUR50TestType, 200)
ruleBuilderE1.AddBanknoteCategory(BanknoteCategoryLevel.Unknown)
            .AddBanknotes(BanknoteCategoryLevel.Unknown,
                Denomination.Unknown, 1)
ruleCountE1 = ruleBuilderE1.GetRule()
ruleBuilderE2 = RuleCountBuilder()
ruleBuilderE2.AddBanknoteCategory(BanknoteCategoryLevel.L4A)
            .AddBanknotes(BanknoteCategoryLevel.L4A,
                Denomination.EUR50TestType, 200)
ruleCountE2 = ruleBuilderE2.GetRule()
```

Il simulatore, in questo caso, riesce a gestire autonomamente l'errore; per questo motivo le uniche regole che dovranno essere inserite saranno quelle di **Count** per specificare le banconote inserite nella prima e seconda parte di deposito. Si può notare come nella prima `ruleCountE1` vengano inserite 200 banconote da 50€ ed una banconota `Unknown` per indicare un pezzo di carta; mentre nel con la `ruleCountE2` avviene solamente l'inserimento di 200 banconote da 50€.

Proseguendo con il test, l'operazione successiva è un'operazione di dispensazione in cui sarà presente anche la `Retract`. Non servono particolari manipolazioni delle regole poiché il simulatore è in grado di gestire il caso autonomamente.

Di seguito si presenta un'operazione di deposito dove però avviene la `Rollback`, 3.11.

Listing 3.11: Deposito con Rollback

```
ruleBuilderDepositRollback = RuleCountBuilder()
ruleBuilderDepositRollback.AddBanknoteCategory(BanknoteCategoryLevel.L4A)
            .AddBanknotes(BanknoteCategoryLevel.L4A,
                Denomination.EUR20TestType, 16)
ruleCount3 = ruleBuilderDepositRollback.GetRule()
```

In questo caso l'unica operazione che deve essere segnalata è quella di specificare quali sono le banconote coinvolte nell'operazione.

Seguendo il test viene simulata ancora un'operazione di deposito, in questo caso con inserimento di banconote contraffatte, insieme a banconote valide e un pezzo di carta, 3.12.

Listing 3.12: Deposito con banconote contraffatte

```
ruleBuilder4 = RuleCountBuilder()
ruleBuilder4.AddBanknoteCategory(BanknoteCategoryLevel.L4A)
ruleBuilder4.AddBanknoteCategory(BanknoteCategoryLevel.L3)
ruleBuilder4.AddBanknoteCategory(BanknoteCategoryLevel.Unknown)
ruleBuilder4.AddBanknotes(BanknoteCategoryLevel.L4A,
    Denomination.EUR20TestType, 2)
ruleBuilder4.AddBanknotes(BanknoteCategoryLevel.L4A,
    Denomination.EUR50TestType, 2)
ruleBuilder4.AddBanknotes(BanknoteCategoryLevel.L3,
    Denomination.EUR10TestType, 1)
ruleBuilder4.AddBanknotes(BanknoteCategoryLevel.Unknown,
    Denomination.Unknown, 1)
ruleCount4 = ruleBuilder4.GetRule()
```

Come prima anche qui l'unica regola manipolata da inserire è quella di conteggio, in modo tale che si possano specificare le banconote coinvolte nell'operazione; il resto viene eseguito da *Robot Framework* e dal simulatore.

Proseguendo la lista dei test il successivo è un deposito in cui è coinvolta l'operazione di `Retract`, senza ripetere ciò che è stato scritto prima l'unica ope-

razione che viene manipolata è quella di conteggio per specificare le banconote coinvolte nell'operazione.

L'operazione successiva mostra cosa avviene quando, durante un deposito, un cassetto si riempie, 3.13.

Listing 3.13: Deposito con cassetto pieno

```
ruleBuilder6 = RuleCountBuilder()
    ruleBuilder6.AddBanknoteCategory(BanknoteCategoryLevel.L4A)
            .AddBanknotes(BanknoteCategoryLevel.L4A,
                Denomination.EUR50TestType, 300)
ruleCount6 = ruleBuilder6.GetRule()
ruleDepositB3 = RuleDepositBuilder()
ruleDepositB3.ChangeDepositInCassette(PositionCassette.Position10,
    banknotesInCassette=29, status=CassetteStatus.CassettePresent
    | CassetteStatus.CassetteAlmostFull |
    CassetteStatus.CassetteFull, goTo =
    PositionCassette.Position01)
ruleDeposit3 = ruleDepositB3.GetRule()
```

Oltre alla `ruleCount6`, dove vengono descritte quali sono le banconote protagoniste dell'operazione di conteggio, nella `ruleDeposit3` si ha la manipolazione di una regola di **Deposit** dove, per il cassetto in posizione 10, vengono indicate quante banconote arrivano correttamente al cassetto con `banknoteInCassette`, ovvero 29, successivamente viene alterato lo stato del cassetto indicandone il riempimento ed infine viene specificato dove le restanti banconote devono finire con `goTo`, in questo caso il cassetto in posizione 1.

Continuando con le operazioni la successiva è un deposito dove le banconote si bloccano sui nastri di trasporto durante la `Deposit`, quindi solamente dopo che sono state conteggiate, 3.14.

Listing 3.14: Deposito con blocco delle banconote

```
ruleBuilderCount7 = RuleCountBuilder()
ruleBuilderCount7.AddBanknoteCategory(BanknoteCategoryLevel.L4A)
```

```
            .AddBanknotes(BanknoteCategoryLevel.L4A,
                Denomination.EUR50TestType, 10)
ruleCount7 = ruleBuilderCount7.GetRule()
ruleDepositB4 = RuleDepositBuilder()
ruleDepositB4 = ruleDepositB4.AddReturnCode(GrgEndingReport.Error,
    GrgReportInformation.SE01DetectNotesGoesIntoWrongTransportChannel)
ruleDeposit4 =
    ruleDepositB4.ChangeDepositInCassette(PositionCassette.Position10,
    banknotesInCassette=6).GetRule()
ruleInitB = RuleInitializationBuilder()
ruleInit =
    ruleInitB2.AddNotesInCassette(BanknoteCategoryLevel.L4A,
    Denomination.EUR50TestType, 4).GetRule()
```

Nel codice si notano differenze rispetto ai casi precedenti, dopo aver specificato come di consueto le banconote dell'operazione di conteggio in `ruleDeposit4`, si hanno due manipolazioni; la prima attraverso il metodo `AddReturnCode` che ci permette di segnalare un errore insieme alle informazioni relative ad esso, la seconda attraverso il metodo `ChangeDepositInCassette` grazie al quale indichiamo l'arrivo di 6 banconote nel cassetto 10 prima che il blocco avvenga. Infine è presente anche la manipolazione della regola di **Initialization**, equivalente all'operazione di `Reset`, in modo tale che venga simulato lo sblocco del dispositivo e il conseguente spostamento delle banconote bloccate in un cassetto apposito a contenerle.

Di seguito vi è l'evento analogo però per un'operazione di dispensazione, ovvero il blocco delle banconote mentre fuoriescono dall'ATM, 3.15.

Listing 3.15: Dispensazione con blocco delle banconote

```
ruleWithdrawErrorB = RuleWithdrawBuilder()
ruleWithdrawErrorB =
    ruleWithdrawErrorB.AddReturnCode(GrgEndingReport.Error,
    GrgReportInformation.SE01DetectNotesGoesIntoWrongTransportChannel)
```

```
ruleWithdrawError =
    ruleWithdrawErrorB.AddCassetteInfo(PositionCassette.Position04,
    totalDispensed = 10, rejected=3, toAC=3).GetRule()
ruleInitB2 = RuleInitializationBuilder()
ruleInit2 =
    ruleInitB2.AddNotesInCassette(BanknoteCategoryLevel.L4A,
    Denomination.EUR50TestType, 5).GetRule()
```

Si ha una manipolazione della regola di dispensazione in cui viene modificato il codice di ritorno e vengono inserite le informazioni relative ad esso grazie al metodo `AddReturnCode`, in aggiunta, grazie a `AddCassetteInfo`, viene specificato anche il comportamento che il cassetto in posizione 4 ha avuto, in questo caso che sono state dispensate in totale 10 banconote, ne sono state rigettate 3 che sono giunte al cassetto AC. Infine nella `ruleInit2` viene specificato cosa accade durante la `Reset`, in particolare che 5 banconote vengono sbloccate dai nastri e inviate all'interno del cassetto apposito.

Infine l'ultima operazione è una `Reset` in cui però avviene un errore, 3.16.

Listing 3.16: Reset con errore

```
ruleInitB3 = RuleInitializationBuilder()
ruleInit3 = ruleInitB3.AddReturnCode(GrgEndingReport.Error,
    GrgReportInformation.RC3PressurePlateFault).GetRule()
```

L'ultimo test modifica solamente la regola di **Initialization** utilizzando il metodo del Builder `AddReturnCode` con cui indichiamo la presenza dell'errore e il tipo dell'errore.

Capitolo 4

Conclusioni & sviluppi futuri

Nel presente lavoro di tesi è stato descritto la progettazione, sviluppo e *testing* di un simulatore di ATM, che sia in grado di riprodurre un comportamento identico a quello di un *device* fisico in modo tale che si potessero effettuare test automatizzati su un modulo software sviluppato dall'azienda Sigma S.p.A. L'impatto che tale simulatore può avere all'interno dei processi aziendali è sicuramente considerevole in quanto andrebbe a ridurre i costi, testare eventuali nuovi comportamenti ed eventuali aggiornamenti del modulo software, riuscendo al tempo stesso ad essere al passo con i nuovi aggiornamenti firmware delle componenti. Di seguito vengono presentati i principali vantaggi che il simulatore riesce a garantire:

- Viene velocizzato per il tester il processo in quanto non sarebbe più obbligato ad affrontare una fase di *sniffing* dei dati in un file prima di poterlo poi utilizzare nel simulatore.
- In generale il *device* fisico richiede delle tempistiche maggiori dovute ai meccanismi interni, mentre grazie al simulatore il tutto sarebbe istantaneo, le uniche tempistiche sono quelle relative al protocollo di comunicazione e alla scrittura del test, tra l'altro, come visto, facilmente riciclabile.
- La persona fisica a cui viene assegnato il compito di effettuare il test non viene più impiegata per diverse ore ad interagire direttamente con il dispo-

sitivo fisico, ma, utilizzando il simulatore, impiegherebbe solamente pochi minuti per effettuare le stesse operazioni.

- Gli aggiornamenti firmware del protocollo, i quali possono apportare modifiche nei `byte` della comunicazione, nella soluzione con *sniffing* porterebbero i file raccolti ad essere obsoleti e alla necessità di una nuova fase di acquisizione. Nella nuova soluzione, invece, basterà apportare delle modifiche all'interno delle regole per riportare il simulatore in una condizione aggiornata.

Continuando su questa strada è possibile, ovviamente, applicare delle migliorie per perfezionare il lavoro svolto. Innanzitutto alcuni moduli interni al simulatore potrebbero essere scritti più sinteticamente, cercando di rendere il codice più snello e leggibile. Il simulatore sviluppato nell'ambito di questa tesi riguarda un preciso dispositivo fisico; questo però non preclude il fatto di poter essere esteso ad altri *device*. Infatti, del codice, tutta la porzione inerente al *Rules Engine* può essere riutilizzata, così come la struttura delle regole e la modellazione dello stato. All'interno delle regole, purtroppo, non è possibile riciclare molto del codice; questo perché il protocollo di comunicazione che diversi *device* hanno con il modulo XFS è differente ed inoltre se dovessimo poi modificare lo stato alcune operazione non saranno le medesime, in quando la gestione a livello fisico potrebbe essere diversa.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] Robot framework. https://robotframework.org/.

[2] C#. https://learn.microsoft.com/it-it/dotnet/csharp/.

[3] Python. https://www.python.org/.

[4] Visual studio. https://visualstudio.microsoft.com/it/.

[5] Visual studio code. https://code.visualstudio.com/.

[6] Python.net. http://pythonnet.github.io/.