UNIVERSITÀ
DI PARMA

Corso di Laurea in Informatica

Sviluppo di un prototipo del sistema "TIS" partendo dalla sua specifica Z tramite l'utilizzo della libreria Java JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Lorenzo Galafassi

Anno Accademico 2020/2021

Indice

Obiettivi

Gli obiettivi di questa tesi sono:

Analizzare il sistema software TIS (Tokeneer Id Station), commissionato dalla agenzia per la sicurezza nazionale americana (National Security Agency) [1] a partire dalla sua specifica formale [2] scritta in notazione Z (uno dei più famosi ed utilizzati linguaggi per specifiche formali) [4].

sviluppare un prototipo di un sottosistema significativo del TIS usando la libreria (che supporta il paradigma logico-dichiarativo) JSetL di Java, sviluppata all'interno del Dipartimento di Matematica e Informatica dell'Universita‘ di Parma [5].

Capitolo 1

Il contesto

1.1 Il TIS

Il TIS è un sistema software responsabile della verifica dei permessi di una persona che voglia entrare in una stanza o luogo sicuro (definito enclave sicuro) nel quale possono trovarsi risorse il cui accesso debba essere regolato. Se la persona che vuole entrare ha tutti i permessi necessari, il TIS permette l'entrata nel luogo protetto.

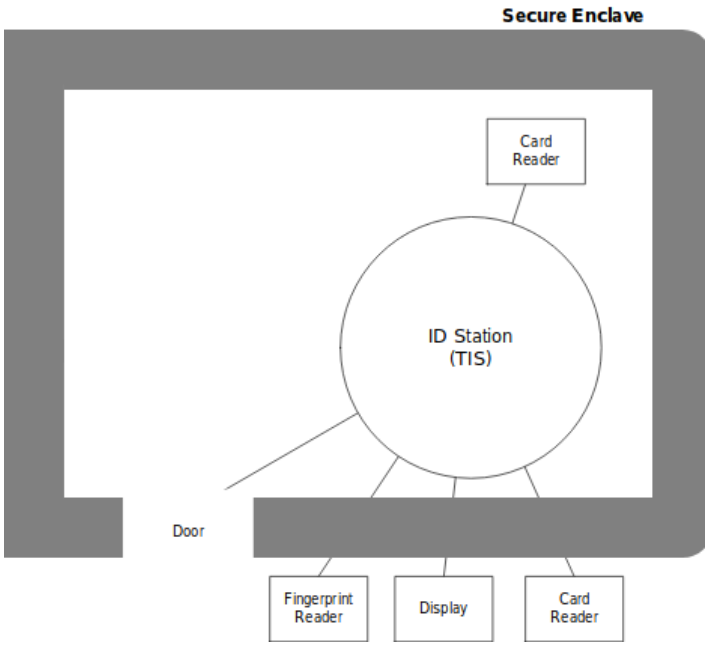


Figura 1.1: il TIS, i suoi componenti e l'ambiente dove opera

1.1.1 Figure con le quali interagisce

Le figure umane con le quali il TIS interagisce sono svariate e sono svariati i compiti che queste ultime eseguono sul TIS:

- **utente normale** utilizzatore (in modo passivo) del TIS: l'unica azione che può effettuare, se ne ha l'autorizzazione, è quella di entrare nell'enclave;

- **amministratore**: è un ruolo che ha la possibilità di eseguire operazioni (attive) sul TIS; a sua volta si divide in:
  - **guardia**: la cui unica funzione è quella di poter aprire manualmente la porta per l'enclave in caso di necessità;
  - **responsabile della sicurezza**: la cui mansione è quella di configurare il TIS (quindi decidere e poter cambiare alcuni parametri come ad esempio il tempo massimo nel quale una persona può restare all'interno dell'enclave) oppure spegnerlo nel momento in cui non è più necessario che sia in funzione;
  - **responsabile del log dei dati**: la cui unica mansione, come ci si poteva aspettare, è quella di effettuare un *dump* dei dati riguardanti lo storico delle operazioni eseguite sul/dal TIS, al fine di analizzarli e poter ricavarne delle informazioni.

1.1.2 Componenti principali con i quali interagisce

Come si può vedere nella figura 1.1, il TIS interagisce con svariati componenti hardware, che sostanzialmente forniscono l'input dal mondo esterno al sistema stesso.

- due **lettori** (uno interno ed uno esterno all'enclave) che servono **per** leggere i **token** e cioè carte magnetiche utilizzate per confermare l'identità della persona interessata all'accesso. Una volta che l'identità viene confermata viene rilasciata una certificazione registrata sul token che stabilisce alcune restrizioni sull'accesso;

- **lettore di impronte digitali**: nel caso in cui leggere e confermare l'identità dal token dell'utente non bastasse (ciò viene deciso in base alla configurazione all'interno del TIS), può sempre venire chiesta l'identificazione anche attraverso l'impronta digitale per una forma ulteriore di sicurezza; per il ruolo di amministratore non è prevista tale ulteriore prova di identità;

- **display**: utilizzato dal TIS per comunicare e/o stampare messaggi al di fuori dell'enclave;

- **porta**: usata per impedire l'accesso e controllata direttamente dal TIS.

1.2 Specifiche Z

1.2.1 Cos'è una specifica?

Con il termine specifica, intendiamo quel documento che specifica in modo chiaro, completo e preciso quali sono le caratteristiche che il sistema software che dobbiamo produrre e sviluppare deve avere.

1.2.2 Cos'è una specifica formale?

Esistono più tipologie di specifiche ed una di queste è la cosiddetta specifica formale, cioè una specifica che utilizza un linguaggio la cui sintassi e semantica è definita in modo formale, per definire tutte le caratteristiche del nostro sistema software. Dare una specifica in questo modo, non solo ci consente di eliminare tutte quelle ambiguità che possono essere presenti in specifiche non formali, ma ci permette anche di dimostrare e quindi verificare alcune proprietà (ad esempio di correttezza) sul nostro sistema software.

1.2.3 Cos'è una specifica Z?

Una specifica Z è una specifica formale scritta basandosi sulla notazione Z, cioè un linguaggio di specifica formale, basato principalmente sulla teoria degli insiemi e sulla logica del primo ordine [4]. Attraverso questa notazione possiamo definire quelli che vengono chiamati **schemi**: la parte fondante della notazione Z.

Cosa sono gli schemi?

- Sono quegli strumenti che ci permettono di definire la natura ed il comportamento del nostro sistema software (una specifica in Notazione Z è composta prevalentemente di schemi).
- Al loro interno gli schemi hanno una parte dedicata alle dichiarazioni di insiemi e una parte dedicata ai predicati (vincoli logici/matematici) su questi insiemi.

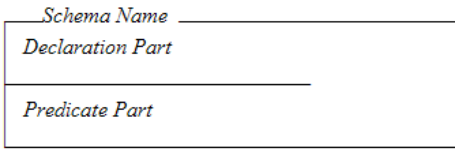


Figura 1.2: struttura di uno schema

Che tipologie di schemi esistono?

- **schemi di stato**: definiscono lo stato della nostra applicazione. Ad esempio:

Figura 1.3: schema di esempio chiamato *PERSON* con nome non vuoto

- **schemi di operazioni**: definiscono le operazioni sullo stato della nostra applicazione. Ad esempio:

Figura 1.4: l'attributo *"value"* di uno schema di stato *"Data"* dopo l' operazione deve essere 1 + il valore prima dell 'operazione

1.2.4 Specifica Z del TIS

Questa figura riportata sotto, è l'immagine di uno schema di stato che specifica tutti i componenti che formano il TIS: nella parte delle dichiarazioni abbiamo a loro volta vari schemi di stato inclusi, che verranno trattati come componenti autonomi (con le loro dichiarazioni e i predicati su di esse).

Attributi della specifica del TIS

- **UserToken**: schema di stato per il token dell'utente attualmente *loggato* nel TIS (se è presente).

- **AdminToken**: schema di stato per il token dell'amministratore attualmente *loggato* nel TIS (se è presente).

- **Finger**: schema di stato che incapsula i dati riguardanti l'impronta digitale dell'utente che in quel momento sta autenticandosi (se è presente).

- **DoorLatchAlarm**: schema di stato che si occupa del controllo della porta, del sistema di chiusura e dell'allarme associato.

DoorLatchAlarm

$$
\begin{aligned}
&currentTime : TIME \\
&currentDoor : DOOR \\
&currentLatch : LATCH \\
&doorAlarm : ALARM \\
&latchTimeout : TIME \\
&alarmTimeout : TIME \\
\hline
&currentLatch = locked \Leftrightarrow currentTime \geq latchTimeout \\
&doorAlarm = alarming \Leftrightarrow \\
&\quad (currentDoor = open \\
&\qquad \wedge currentLatch = locked \\
&\qquad \wedge currentTime \geq alarmTimeout)
\end{aligned}
$$

- **Floppy**: schema di stato per il floppy disk del TIS, utilizzato per configurare il TIS da parte dell'ufficiale di sicurezza. (se è presente).

Floppy

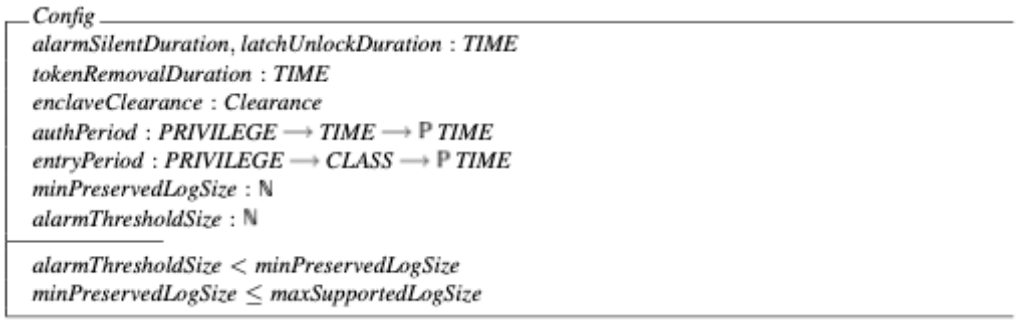
$$
\begin{aligned}
&currentFloppy : FLOPPY \\
&writtenFloppy : FLOPPY \\
&floppyPresence : PRESENCE
\end{aligned}
$$

- **Keyboard**: schema di stato per la tastiera (se è presente) del TIS, utilizzato per interagire con il TIS da parte dell'ufficiale di sicurezza dal display interno all'enclave sicuro.

Keyboard

$$
\begin{aligned}
&currentKeyedData : KEYBOARD \\
&keyedDataPresence : PRESENCE
\end{aligned}
$$

- **Config**: schema di stato che incapsula le informazioni riguardanti la configurazione del TIS.



- **Stats**: schema di stato che indica il rapporto di corrette autenticazioni sulle autenticazioni totali.

- **KeyStore**: schema di stato che incapsula le informazioni riguardanti le chiavi di accesso al TIS, indicando a chi sono state rilasciate e da chi.

- **Admin**: schema di stato che contiene tutte le informazioni dell'amministratore attualmente *loggato* nel TIS (se è presente).

Admin

$$
\begin{aligned}
&rolePresent : \text{optional}\, ADMINPRIVILEGE \\
&availableOps : \mathbb{P}\, ADMINOP \\
&currentAdminOp : \text{optional}\, ADMINOP \\
\hline
&rolePresent = nil \Rightarrow availableOps = \emptyset \\
&(rolePresent \neq nil \land the\, rolePresent = guard) \Rightarrow availableOps = \{overrideLock\} \\
&(rolePresent \neq nil \land the\, rolePresent = auditManager) \Rightarrow availableOps = \{archiveLog\} \\
&(rolePresent \neq nil \land the\, rolePresent = securityOfficer) \Rightarrow availableOps = \{updateConfigData, shutdownOp\} \\
&currentAdminOp \neq nil \Rightarrow \\
&\quad (the\, currentAdminOp \in availableOps \land rolePresent \neq nil)
\end{aligned}
$$

- **AuditLog**: schema di stato che contiene le informazioni sullo storico delle operazioni del e/o sul TIS.

AuditLog

$$
\begin{aligned}
&auditLog : \mathbb{F}\, Audit \\
&auditAlarm : ALARM
\end{aligned}
$$

- **Internal**: schema di stato che contiene le informazioni sullo stato generale del TIS.

Internal

$$
\begin{aligned}
&status : STATUS \\
&enclaveStatus : ENCLAVESTATUS \\
&tokenRemovalTimeout : TIME
\end{aligned}
$$

- **currentDisplay**: indica il messaggio corrente (se sono presenti) che è presente sul display al di fuori dell'enclave (utili soprattutto agli utenti senza privilegi che devono entrare nell'enclave).
- **currentScreen**: indica i messaggi correnti (se sono presenti) che sono presenti sul display all'interno dell'enclave (utili soprattutto agli utenti senza privilegi che devono entrare nell'enclave).

Predicati della specifica del TIS

- Se lo stato del TIS è tale che abbiamo già ricevuto ed autenticato il token dell'utente (quindi siamo negli stati evidenziati in arancione e abbiamo passato lo stato evidenziato in verde, come indicato in figura 1.5), allora quest ultimo deve essere valido/conforme alla configurazione del TIS e deve avere una autorizzazione valida in quel momento.

Figura 1.5: operazioni dell'utente

- Se nessun admin è *loggato*, un utente presente nel TIS deve avere almeno una autorizzazione da quest' ultimo.
- Se il TIS è acceso (vale a dire che non è negli stati evidenziati in arancione, come si può vedere in figura 1.6), allora deve esistere una figura (umana) che sia il responsabile di poter rilasciare le autorizzazioni ai vari token degli utenti/amministratori.

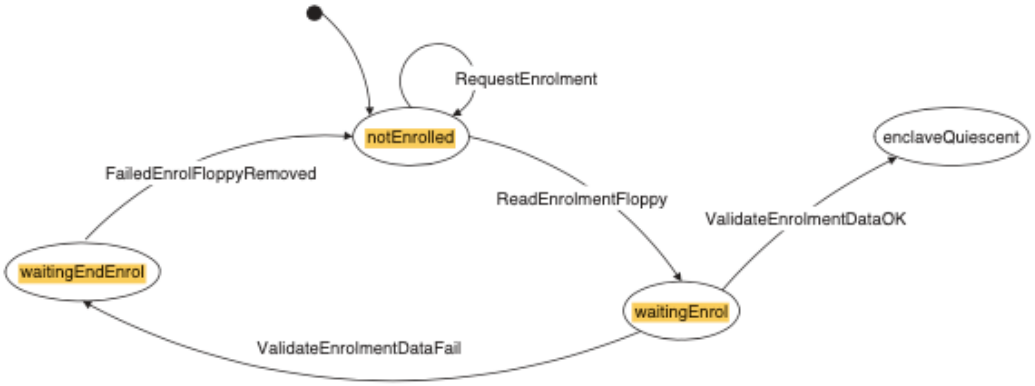


Figura 1.6: operazioni della fase di accensione del TIS

- Se il TIS sta aspettando che un amministratore esegua una operazione oppure sta aspettando che finisca una operazione, allora deve essere presente e rintracciabile una informazione che ci dica quale operazione sia.
- Se un utente esegue una operazione per la quale il TIS viene riavviato e portato ad uno stato iniziale, allora lo stato dell'enclave deve essere tale per cui quest' ultimo aspetta una operazione dell'admin.

- Se siamo in una fase dove il TIS ha appena letto il token dell'admin (come possiamo vedere nella immagine sottostante, la figura 1.7), allora ancora non possiamo capire quale sia il ruolo dell'utente dato che dobbiamo verificare prima l'autenticità dell'amministratore stesso.

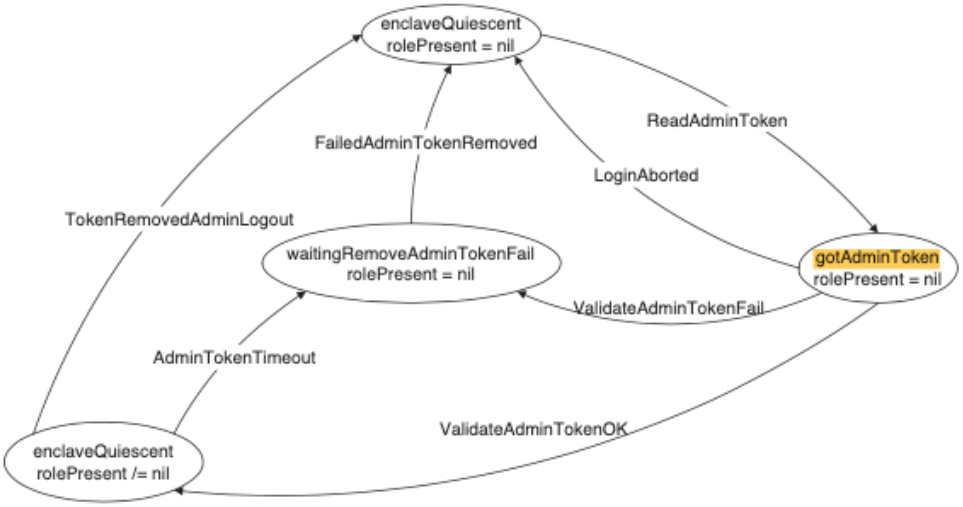


Figura 1.7: operazione di login dell'amministratore

- In qualsiasi momento, il TIS deve stampare sul display interno, i messaggi che indicano lo stato dello stesso e quale sia la sua configurazione.

1.3 JSetL

JSetL è una libreria Java che consente all'utente di utilizzare uno stile di programmazione logico - dichiarativo a vincoli. Più nel dettaglio, alcune delle principali strutture dati/astrazioni che questa libreria offre sono:

- **variabili logiche**: definite attraverso la classe, e le sottoclassi di, *LVar*. Possono assumere qualunque valore e quest' ultimo può essere assegnato come risultato della risoluzione di un vincolo che coinvolge la variabile; nel caso che l'assegnazione sia già stata fatta non è possibile modificarla.

- **liste**: una lista (logica) è un oggetto di classe *LList* i cui valori sono oggetti che possono essere presenti più volte all'interno della stessa, come per le variabili logiche, una volta assegnati i valori nella lista, quest'ultima non può più essere modificata.

- **insiemi**: un insieme (logico) è un oggetto di classe *LSet* i cui valori sono oggetti che possono essere presenti solamente una volta all'interno della stessa. Come per le variabili logiche e le liste, una volta che abbiamo inizializzato un oggetto di questa classe, non è possibile modificarlo.

- **vincoli**: un oggetto di tipo *Constraint* astrae il concetto di vincolo e nel momento stesso nel quale questi vincoli vengono risolti, allora verranno, tra l'altro, verificate le uguaglianze e nel caso fatti degli assegnamenti.

- **risoluzione di vincoli**: per risolvere i vincoli presenti nel nostro programma (la vera parte logica e il vero core di quest ultimo) utilizziamo la classe *Solver*.

1.3.1 Paradigma dichiarativo

Il paradigma dichiarativo è uno stile di programmazione dove, al contrario della programmazione imperativa, viene descritto il risultato finale desiderato invece di mostrare i passi che permettono di ottenerlo, concentrandosi quindi più sul "cosa voglio", rispetto al "come voglio farlo". Le due categorie di linguaggi più famose che derivano dal paradigma dichiarativo sono la programmazione logica (che abbiamo in parte conosciuto) e quella funzionale.

1.3.2 Perchè utilizziamo JSetL?

La notazione Z, come abbiamo visto, definisce lo stato e le operazioni di una applicazione o di un sistema attraverso la dichiarazione di insiemi e di vincoli; per cui utilizzare la libreria Java JSetL, che ci permette di creare e scrivere programmi attraverso la dichiarazione e poi la verifica/soluzione di determinati vincoli su determinati insiemi, è naturale e relativamente semplice; quasi come se i programmi scritti con questa libreria costituiscano una versione eseguibile della specifica formale Z.

Capitolo 2

Da Z a JSetL

2.1 Progettazione e sviluppo della libreria Z-JSetL

Come abbiamo in precedenza detto, i vari costrutti della notazione Z sono in un certo senso simili ad alcuni costrutti e/o astrazioni che possiamo trovare in JSetL e nella programmazione logica e/o ad oggetti in generale. Per cui pensare ad una piccola serie di funzioni e classi generali che facciano da collegamento tra le due realtà, può essere utile e soprattutto naturale.

2.1.1 Componenti

Schemi

Abbiamo visto che in una specifica Z sono definiti gli schemi, che hanno una parte di dichiarazioni di attributi e una parte di predicati, per cui possiamo ricreare questo concetto tramite l'utilizzo di una classe.

Dato quindi uno schema S con un numero di attributi *A1.....An* e con un numero di predicati *P1.....Pn*, possiamo pensare, come prima idea di scrivere una classe implementata in questo modo:

```
public final class Schema {
    A1
    .
    An

  public Schema() {
     predicates();
  }

  private void predicates() {
     P1
     .
     Pn
  }
```

A questo punto, una volta che abbiamo una idea più o meno generale di quella che può essere la nostra "astrazione" di uno schema, dobbiamo iniziare a pensare come poterla effettivamente implementare.

Possiamo dire che uno schema è sostanzialmente un insieme di vincoli che devono essere in qualsiasi momento veri. A loro volta essi formano un unico grande vincolo, formato "dall'unione" di questi ultimi. E' per questo che possiamo far estendere la classe Constraint alla nostra "classe prototipo" Schema. In questo modo, la nostra classe, ereditando tutte le operazioni di una constraint, può essere trattata come un vincolo ed inserita nelle preposizioni logiche.

A questo punto il codice del nostro schema dovrebbe assumere questa forma:

```
public final class Schema extends Constraint {
    A1 .. An

  public Schema() {
     predicates();
  }

  private void predicates() {
     P1 .. Pn
  }
```

Arrivati qui, sentiamo la necessità, però, di avere una classe specifica per ogni tipologia di schema, dato che in base alla tipologia di schema, quest' ultimo può avere un comportamento diverso da un altro:

- gli **schemi di stato** hanno sicuramente una parte di attributi ma opzionalmente una parte di predicati.

```
public class StateSchema extends Constraint {
    protected void predicates () throws Failure {}
}
```

  Dato che non abbiamo l'obbligo di un predicato, possiamo definire il metodo che si occupa di "chiamare" i predicati come metodo della classe, che se non viene ridefinito nella sottoclasse attraverso l'override, non crea, giustamente, nessuna problema.

- gli **schemi di operazioni** hanno sicuramente una parte di attributi e sicuramente una parte di predicati.

```
public abstract class OperationSchema extends Constraint {
    protected abstract void predicates() throws Failure;
}
```

  In questo caso, avendo l'obbligo di avere dei predicati (dato che le operazioni vengono comunque definite come predicati stessi), definiamo il metodo responsabile di chiamare i predicati come astratto; obbligando, in un certo senso, chi estende la classe ad implementare questo metodo.

Decoratori

Quando definiamo un attributo, nella notazione Z, abbiamo a disposizione vari modificatori per quest' ultimo:

- **decoratori per la mutabilità degli schemi di stato**: a loro volta si dividono in:
  - **decoratore per schema di stato mutabile**:

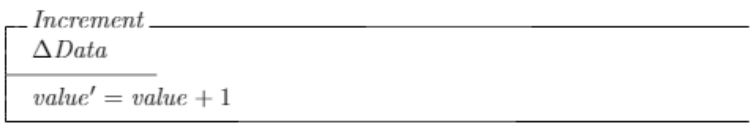


Figura 2.1: esempio di uno schema di stato mutabile

Attraverso il simbolo della lettera "Delta", stiamo dichiarando che almeno uno degli attributi dello schema di nome "Data" subirà un cambiamento. Il valore dell'attributo "value" dopo il completamento dell'operazione sarà diverso rispetto al valore iniziale.

Nella libreria Z-JSetL questo costrutto è stato riprodotto per essere (da un punto di vista della forma) il più simile alla sua controparte della notazione Z.

```
public final class FooOp extends OperationSchema {
  public Delta<StateSchema> stateSchema;
  ...
}
```

Come si può vedere, quando chiamiamo (per restare fedeli alla specifica Z) i metodi *beforeOperation* o *afterOperation* quello che viene passato è sempre e solamente il riferimento dello schema di stato originale, in modo tale da effettuare effettivamente le modifiche.

```
public final class Delta<T> {
  private T stateSchemaValue;

  public Delta(T stateSchemaValue) {
    this.stateSchemaValue = stateSchemaValue;
  }

  public T beforeOperation() {
    return stateSchemaValue;
  }

  public T afterOperation() {
    return stateSchemaValue;
  }
}
```

Esempio implementato conZ-JSetL dello schema in figura 2.1:

```
public final class Increment extends
    OperationSchema {
   public Delta<Data> data;

   public Increment() {
      predicates();
   }

   @override
   protected void predicates() {
       data.afterOperation() =
           data.beforeOperation() + 1
   }
}
```

– **decoratore per schema di stato immutabile**:

Attraverso il simbolo della lettera "Xi", stiamo dichiarando che nessuno degli attributi dello schema di nome "Data" subirà un cambiamento.

Nella libreria Z-JSetL:

```
public final class FooOp extends OperationSchema {
  public Xi<StateSchema> stateSchema;
  ...
}
```

Quando chiamiamo il metodo *immutable*, quello che viene passato è sempre e solamente una copia dello schema di stato originale, in modo tale da essere sicuri di non effettuare alcuna modifica. Nel costruttore ci preoccupiamo che la copia passata, sia di ugual valore ma non sia lo stesso oggetto.

```
public final class Xi<T> {
  private T stateSchemaValue;

  public Xi(T stateSchemaValueCopy, T
      stateSchemaValue) {
      assert stateSchemaValueCopy !=
          stateSchemaValue &&
          stateSchemaValue.equals(stateSchemaValueCopy);

      this.stateSchemaValue = stateSchemaValue;
  }

  public T immutable() {
      return stateSchemaValue;
  }
}
```

- **decoratori per il tipo degli attributi**: come i decoratori per gli schemi, esistono nella notazione Z anche costrutti atti a modificare la struttura ed il comportamento delle variabili all'interno di uno schema. Più precisamente esistono simboli (anche definibili da utente nella specifica) all'interno della notazione Z, che applicati alle variabili permettono di modificarne la forma.

  Un esempio di questo decoratore è il *Power Set*, definito matematicamente come l'insieme delle parti, cioè l'insieme formato da tutti i possibili sottoinsiemi di un insieme.

  esempio in notazione Z

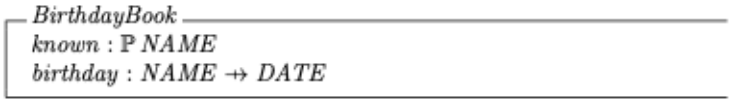


Figura 2.2: possiamo vedere l'utilizzo del decoratore *Power Set*

Come possiamo vedere nella figura 2.2, abbiamo uno schema con due variabili, di cui una ha proprio come *decoratore* applicato il *Power Set*. Quando dichiariamo una variabile di un tipo A, nella notazione Z, stiamo dichiarando che quella variabile è un elemento dell'insieme A. Per cui se stiamo dichiarando che la nostra variabile è un elemento del Power Set del tipo *Name* (o più precisamente dell'insieme Name) stiamo dicendo che allora la variabile è un sottoinsieme di *Name*.

implementazione nella libreria Z-JSetL

All'interno della libreria Z-JSetL, l'implementazione di questi decoratori viene fatta implementando l'interfaccia *Decorable* (che fa sempre parte di questa libreria) visibile in figura x, ed estendendo nel caso che ci fosse bisogno, una delle sottoclassi della classe LObject (facente parte della libreria JSetL).

```
public interface Decorator {
    public void verifyDecoratorConstraint() throws Failure;
}
```

Da come possiamo vedere dal codice soprastante, un decoratore è una classe che implementa l'interfaccia *Decorable* e quindi il metodo *verifyDecoratorConstraint*. Infatti questa interfaccia è nata per dare una definizione generale di cosa è un decoratore, per cui chiunque lo implementa, diventerà per definizione quest ultimo.

```
public final class P<T> extends LSet implements Decorator {
    public P(LSet values) throws Failure {
        super(values);

        verifyDecoratorConstraint();
    }

    @Override
    public void verifyDecoratorConstraint() throws Failure
        {
        Constraint.truec().solve();
    }
}
```

Per capire il perchè dietro al codice presentato qui sopra, bisogna prima vedere un brevissimo esempio dell'utilizzo di questo decoratore:

```
P<Name> names = new P<Name>(new
    HashSet<Name>(Arrays.asList(new Name("Lorenzo", new
    Name("Luca")))))
```

Da questo piccolo esempio, capiamo che il nostro decoratore diventerà lui la nuova variabile. In questo caso quindi un decoratore di questo tipo non sarà altro, per l'appunto, che una sottoclasse di *LSet*. L'implementazione del metodo *verifyDecoratorConstraint* è superflua ma implementata per un obbligo contrattuale con l'interfaccia. Andremo in sostanza a chiedere ad un *Solver* se la costante "true" è true, dato che non abbiamo bisogno di altri vincoli che non siano già stati implementati con la classe *LSet*.

approfondimento sulla classe P

Bisogna notare che la nostra classe P è parametrica ma al suo interno non utilizza il potere offerto da questo costrutto. Questo per due motivi:

- Il tipo parametrico è offerto solamente per una questione "estetica", per rendere più consistente e simile possibile, la sintassi dei decoratori in Z e la sintassi dei decoratori nella libreria Z-JSetL.
- Il secondo motivo è che all'interno della classe LSet tutte le variabili dell'insieme non sono altro che oggetti di classe Object perdendo così, senza utilizzare e fare dei cast, il suo tipo originale.
- Il terzo motivo, è che non abbiamo nessun bisogno di avere un controllo sul tipo delle variabili del nostro insieme, dato che per come è pensato il paradigma di programmazione logico (e quindi la libreria JSetL), una volta inizializzata una variabile non può essere reinizializzata, per cui "da contratto" il tipo del nostro insieme viene deciso in fase di iniazilizzazione ed è chiaro a tutti che tipo di variabili sono state inserite, perchè esplicito.

2.2 Una semplice applicazione

Per dimostrare un breve utilizzo della libreria prodotta, mostriamo come realizzare in Java + Z-JSetL il prototipo di una piccola applicazione partendo dalla specifica Z, in cui tutti i costrutti e le astrazioni spiegate finora si vedranno impiegate.

2.2.1 Componenti della specifica Z "BirthdayBook"

- Sono presenti due tipi la cui (come feature del "linguaggio" e possibilità data dallo stesso) implementazione non è definita, di nome Name e Date, come possiamo vedere nella figura sottostante.

$$[NAME, DATE]$$

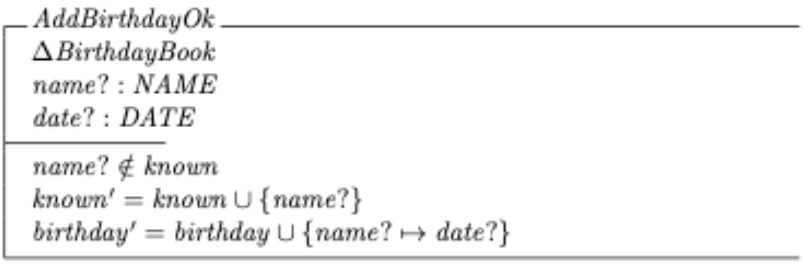
- è presente uno schema, chiamato *BirthdayBook* che realizza l'astrazione e lo schema di un libro o un quaderno dove sono indicati un insieme di nomi (cioè i nomi stessi che sono stati aggiunti a questo libro, per una forma di controllo) e per ogni nome, aggiunto, il suo giorno di compleanno. Infatti il tipo dei nomi già presenti, cioè *known* non è altro che il sottinsieme di quello che è l'insieme di tutti i nomi conosciuti. Un modo per dire che gli elementi della variabile *known* sono nomi formati in modo corretto. La variabile *birthday* è invece una funzione parziale, cioè una funzione non definita su tutto il dominio. Questo significa (dato come dominio l'insieme degli elementi appartenenti all'insieme *Name*) che non per forza di tutte le persone (come è naturale pensare) al mondo noi saremo a conoscenza del compleanno.

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

- è presente uno schema chiamato *BirthdayBookInit* che viene chiamato, come ci si può aspettare, in fase di inizializzazione della applicazione il cui scopo è quello di assegnare gli insiemi vuoti alle variabili dello schema *BirthdayBook*

- Questo riportato sotto è lo schema della prima operazione che viene implementata in questa applicazione. Dato un *name* e una *data* appartenenti rispettivamente all'insieme dei *Name* e all'insieme delle *Date* controlla che il name non sia presente tra i nomi aggiunti al *BirthdayBook*, questa è la precondizione dell'operazione per potere iniziare. In caso che questa condizione sia soddisfatta, allora possiamo aggiungere all'insieme dei nomi conosciuti anche il nostro nome "di input" (visibile dal punto di domanda alla fine del nome) ed aggiungere a quella funzione che raccoglie, per i nomi che desideriamo aggiungere, i loro compleanni. Dal punto di vista "matematico" è interessante dire che le funzioni vengono viste come insiemi di coppie. Difatti andremo ad aggiungere a *birthday* una coppia contenente il nome ed il suo compleanno.



- Nella figura sottostante è presente lo schema dell'operazione che controlla se un nome è già presente all'interno dei libro che cataloga queste ricorrenze. Come vedremo, questa è proprio la precondizione tale per cui questa operazione venga in un certo senso invocata. Successivamente possiamo vedere che non abbiamo altri vincoli. Per cui la nostra operazione viene correttamente terminata. Ciò significa che se una operazione che controlla che se un elemento è presente in un determinato insieme termina correttamente, significa allora che questo elemento è presente.

- Questa ultima operazione indicata nella figura sottostante, non è altro che la composizione delle due operazioni indicate al loro interno.

$$AddBirthday == AddBirthdayOk \lor NameAlreadyExists$$

La semantica di queste operazione è la seguente:

  - più "ad alto livello": questa operazione (che è anche a sua volta un predicato) per essere portata a termine deve risultare logicamente vera (ecco perchè le operazioni sono dei predicati), per cui deve essere vera l'operazione *AddBirthdayOk* oppure deve essere vera l'operazione*NameAlreadyExixts*.
  - più "a basso livello": questa operazione è a sua volta un nuovo schema che rappresenta una operazione dove è stata fatta l'unione (insiemistica, quindi le variabili che sono presenti in tutti e due i predicati vengono prese una volta sola) delle dichiarazioni delle variabili e la coniugazione attraverso un operatore logico *or* dei vincoli delle due operazioni su queste variabili.

2.2.2 Componenti implementati in Z-JSetL

In questa sezione, verrà presentato il codice scritto attraverso la libreria Z-JSetl della applicazione presentata precedentemente.

- Dichiarazione del tipo di dato *Name*.

```
public final class Name extends LVar {
    public Name(String value) {
        super("", value);
    }
}
```

- Dichiarazione del tipo di dato *Date*. Come anche nel tipo precedente ho deciso che la classe *LVar* potesse essere una buona e semplice candidata per poter essere estesa e quindi donare la sua implementazione ai nostri due tipi.

```
public final class Date extends LVar {
    public Date(String value) {
        super("", value);
    }
}
```

- Come ci si poteva aspettare essendo *BirthdayBook* uno state schema, quest ultimo sarà una sottoclasse proprio di *StateSchema*.

```
public final class BirthdayBook extends StateSchema {
    public P<Name> known;

    public LMap birthday;

    public BirthdayBook(P<Name> known, LMap birthday) {
        super();

        this.known = known;

        his.birthday = birthday;
    }

    public BirthdayBook(BirthdayBook birthdayBook) throws
        Failure {
       this.known = new P<Name>(new
          LSet(birthdayBook.known.getValue()));

       this.birthday = new LMap(birthdayBook.birthday);
    }
}
```

- Essendo comunque una operazione dove oltretutto lo stato del *BirthdayBook* cambiava, indicare l'operazione di inizializzazione come uno schema di operazione, fosse la scelta più naturale.

```
public final class BirthdayBookInit extends
    OperationSchema {
    public BirthdayBook birthdayBook;

    public BirthdayBookInit(BirthdayBook birthdayBook)
        throws Failure {
        super();

        this.birthdayBook = birthdayBook;

        predicates();
    }

    @Override
    protected void predicates() throws Failure {
        add(birthday().eq(LMap.empty()));

        add(known().eq(LSet.empty()));

        solve();
    }

    private LMap birthday() {
        return birthdayBook.birthday;
    }

    private P<Name> known() {
        return birthdayBook.known;
    }
}
```

- In questa classe, per aumentare la leggibilità del metodo *predicates* e renderlo più simile ancora allo schema in notazione Z, vengono introdotti alcuni metodi "di supporto".

```
public final class AddBirthdayOk extends OperationSchema {
    public Delta<BirthdayBook> birthdayBook;

    public Name name;

    public Date date;

    public AddBirthdayOk(BirthdayBook birthdayBook, Name
        name, Date date) throws Failure {
        super();

        this.birthdayBook = new Delta<>(birthdayBook);

        this.name = name;

        this.date = date;

        predicates();
    }

    @Override
    protected void predicates() throws Failure {
        add(name.nin(known()));

        add(known().union(nameInSet(), known_()));

        add(birthday().union(nameDatePair(), birthday_()));

        solve();
    }

    private P<Name> known() {
        return birthdayBook.beforeOperation().known;
    }

    private P<Name> known_() throws Failure {
        return (birthdayBook.afterOperation().known = new
            P<>(new LSet()));
    }
```

```
    private LMap birthday_() {
        return (birthdayBook.afterOperation().birthday = new
            LMap());
    }

    private LMap birthday() {
        return birthdayBook.beforeOperation().birthday;
    }

    private LSet nameInSet() {
        return new LSet(new
            HashSet<>(Collections.singletonList(name.getValue())));
    }

    private LSet nameDatePair() {
        return new LSet(new
            HashSet<>(Collections.singletonList(new
            LPair(name, date))));
    }
}
```

- Essendo il metodo this.solve() un metodo che può lanciare una eccezione di tipo *Failure* che accade nel momento stesso in cui non può essere trovata una soluzione.

```
public final class NameAlreadyExists extends
    OperationSchema {
    public Xi<BirthdayBook> birthdayBook;

    public Name name;

    public NameAlreadyExists(BirthdayBook birthdayBook, Name
        name) throws Failure {
        super();

        this.birthdayBook = new Xi<>(new
            BirthdayBook(birthdayBook), birthdayBook);
        this.name = name;

        predicates();
    }

    @Override
    protected void predicates() throws Failure {
        add(name.in(known()));

        solve();
    }

    private P<Name> known() {
        return birthdayBook.immutable().known;
    }
}
```

- Viene creato infine un "Main", dove al suo interno è presente, come si può vedere, il codice per inizializzare e quindi avviare l'applicazione. Il fulcro dell'applicazione è l'operazione (*AddBirthday*) che abbiamo visto nella notazione Z accorpa e fonde le due operazioni (*AddBirthdayOk* e *NameAlreadyExists*), ma qui all'interno di questo main, è stata chiamata come un vincolo normale (perchè infatti lo è).

```
public class Main {

  public static void main(String[] args) throws Failure {
      var birthdayBook = new BirthdayBook(new P<Name>(new
          LSet()), new LMap());

      var name = new Name("jon");

      new BirthdayBookInit(birthdayBook);

      new AddBirthdayOk(birthdayBook, name, new
          Date("03/01/1999")).or(new
          NameAlreadyExists(birthdayBook, name));
  }
}
```

Capitolo 3

Un prototipo per un'operazione del TIS

In questo capitolo viene presentato un prototipo del *TIS* che mostri in modo preciso quale sia un possibile flusso di esecuzione senza però soffermarsi su dettagli tecnici che potrebbero renderne meno evidente il funzionamento.

3.1 Descrizione dell'operazione

L'operazione, raffigurata nell'immagine appena sotto, scelta dalla specifica formale del TIS [3] è l'operazione denominata *TISAdminLogon*, cioè quella operazione che raccoglie tutto il flusso di esecuzione per quanto riguarda l'accesso al TIS da parte dell'Admin dall'interno dell'enclave sicuro. Una descrizione di questa operazione dal punto di vista dello stato del *TIS* si trova alla figura 1.7.

f

$$TISAdminLogon \mathrel{\widehat{=}} TISReadAdminToken \lor TISValidateAdminToken \lor TISCompleteFailedAdminLogon$$

3.1.1 TISReadAdminToken

Lo schema *TISReadAdminToken* è un *alias* dello schema *ReadAdminToken*. La sua definizione è la seguente:

$$TISReadAdminToken \mathrel{\widehat=} ReadAdminToken$$

ReadAdminToken, a sua volta, è uno schema (raffigurato qui di seguito) che astrae l'operazione di lettura del token.

Come possiamo vedere in figura 1.7, *ReadAdminToken* è la prima operazione che, ragionevolmente, un Admin deve compiere per poter tentare di accedere al TIS, dopo di chè se il token presentato era corretto, si può procedere alle fasi successive, altrimenti l'operazione è fallita per uno dei due motivi elencati qui di seguito.

- Il token è stato disinserito in modo doloso dall'apposito lettore.
- La validazione del token è fallita, quindi il token non era valido.

ReadAdminToken

$LoginContext$

$\Xi Admin$

$AddElementsToLog$

$status \in \{ quiescent, waitingRemoveTokenFail \}$

$enclaveStatus = enclaveQuiescent$
$rolePresent = nil$
$adminTokenPresence = present$

$enclaveStatus' = gotAdminToken$
$currentScreen' = currentScreen$

Il funzionamento dello schema è tale per cui se lo stato dell'enclave è lo stato iniziale (*EnclaveQuiescent*) e nessun admin è collegato al TIS allora il token viene inserito e viene aggiornato lo stato del TIS per indicarlo.

3.1.2 TISValidateAdminToken

Lo schema *TISValidateAdminToken* astrae e raccoglie la logica all'interno del *TIS*, atta a validare e quindi controllare la correttezza del token dell'admin. La sua struttura è la seguente:

$$TISValidateAdminToken \mathrel{\widehat=} ValidateAdminTokenOK \lor ValidateAdminTokenFail \lor LoginAborted$$

ValidateAdminTokenOK

Lo schema *ValidateAdminTokenOK*, raffigurato nella figura sottostante, si occupa di controllare e quindi successivamente convalidare che il token dell'admin sia corretto.

ValidateAdminTokenOK

$$LoginContext$$

$$AdminLogon$$

$$AddElementsToLog$$

$$enclaveStatus = gotAdminToken$$
$$adminTokenPresence = present$$

$$AdminTokenOK$$

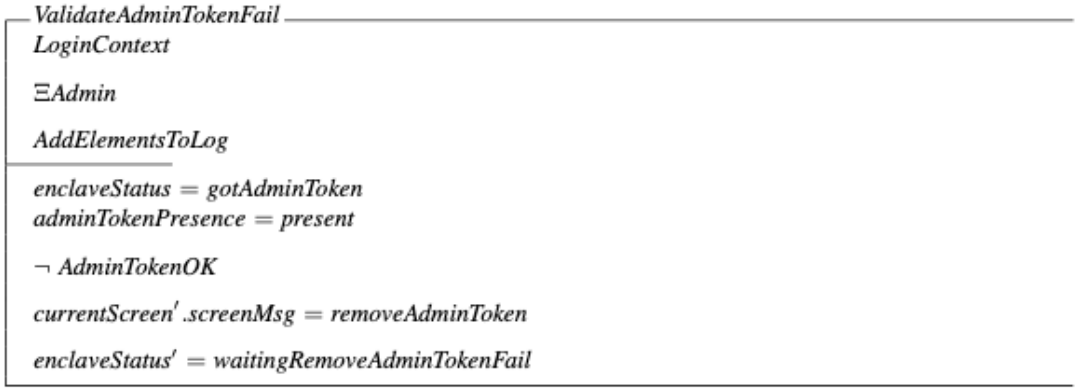
$$currentScreen'.screenMsg = requestAdminOp$$

$$enclaveStatus' = enclaveQuiescent$$

Il funzionamento dello schema è tale per cui se l'admin ha inserito il token e il token ha i permessi necessari, allora aggiorniamo lo stato del TIS indicando che effettivamente il token è corretto.

ValidateAdminTokenFail

Lo schema *ValidateAdminTokenFail*, raffigurato nella figura sottostante, si occupa di controllare e quindi successivamente verificare che il token non ha i permessi necessari per poter permettere all'admin di entrare nell'enclave sicuro.



Il funzionamento dello schema è tale per cui se l'admin ha inserito il token e il token non ha i permessi necessari, allora aggiorniamo lo stato del TIS indicando che effettivamente il token è non corretto e suggeriamo all'admin di disinserire il token, dato che l'operazione è fallita.

LoginAborted

Lo schema *LoginAborted* descrive quella operazione nella quale la procedura di login dell'admin viene interrotta e successivamente eliminata.

L'operazione soprastante, riutilizza uno schema che rappresenta un'altra operazione chiamata *BadAdminTokenTear*. Quest'ultima si occupa di descrivere l'operazione nella quale l'admin disinserisce il token quando non dovrebbe, invalidando così l'intera procedura di login.

3.1.3 TISCompleteFailedAdminLogon

Lo schema *TISCompleteFailedAdminLogon* descrive quella operazione utilizzata per la gestione della situazione nella quale l'operazione di login dell'admin fallisce, riportando il *TIS* nello stato precedente (facendo così una sorta di operazione di rollback). Nella figura sottostante possiamo osservare come questa operazione è composta:

$$TISCompleteFailedAdminLogon \mathrel{\widehat{=}} FailedAdminTokenRemoved \lor WaitingAdminTokenRemoval$$

FailedAdminTokenRemoved

Lo schema *FailedAdminTokenRemoved*, raffigurato nella figura sottostante, descrive quella situazione nella quale, successivamente al fallimento dell'operazione di login, l'admin ha disinserito con successo il suo token.

FailedAdminTokenRemoved

$$LoginContext$$
$$\Xi Admin$$
$$AddElementsToLog$$

$$enclaveStatus = waitingRemoveAdminTokenFail$$
$$adminTokenPresence = absent$$
$$currentScreen'.screenMsg = welcomeAdmin$$
$$enclaveStatus' = enclaveQuiescent$$
$$currentDisplay' = currentDisplay$$

Se l'admin ha rimosso il token aggiorniamo lo stato del *TIS* e lo riportiamo nello "stato iniziale", cioè nello stato *EnclaveQuiescent*. Per una maggiore comprensione, come specificato in precedenza, vedere la figura 1.7.

WaitingAdminTokenRemoval

Lo schema *WaitingAdminTokenRemoval*, raffigurato nella figura sottostante, si occupa di descrivere quella situazione nella quale il *TIS* sta aspettando che l'admin disinserisca il token dal lettore apposito, una volta che l'operazione di login è fallita.

3.2 Implementazione attraverso la libreria Z-JSetL del prototipo

In questa sezione verrà elencato il codice di programmazione *Java + Z-JSetL* di ogni schema mostrato nel capitolo 3. In altre parole, verrà presentata l'implementazione del prototipo sviluppato. Come prima, verranno elencate le classi ed in generale il codice in modo gerarchico, a partire dall'operazione "totale" *TISAdminLogon* e via via scendendo verso le operazioni più specifiche.

3.2.1 TISAdminLogon

Il codice Java sottoriportato è l'implementazione dello schema *TISAdminLogon* presente nel capitolo 3.1, cioè lo schema che raccoglie ed astrae la logica completa per l'operazione di login dell'amministratore.

Possiamo vedere, all'interno dell'operazione *TISAdminLogon*, le operazioni che la compongono, cioè le operazioni *TISReadAdminToken* (presente nel capitolo 3.1.1.), *TISValidateAdminToken*, (presente nel capitolo 3.1.2) e *TISCompleteFailedAdminLogon* (presente nel capitolo 3.1.3).

```
public TISAdminLogon(LoginContext loginContext, Admin admin)
    throws Failure {
  TISReadAdminToken(loginContext, admin);
  TISValidateAdminToken(loginContext);
  TISCompleteFailedAdminLogon(loginContext, admin);
}
```

3.2.2 TISReadAdminToken

Il codice Java sottoriportato è l'implementazione dello schema *TISReadAdminToken* presente nel capitolo 3.1.1.

Per una questione di chiarezza e descrizione a run time dei vari stati dell'operazione (indicati in figura 1.7) è necessario controllare che l'operazione *ReadAdminToken* (l'unica operazione che viene invocata) sia andata a buon fine e successivamente comunicarlo attraverso un messaggio a schermo. Più precisamente, nel momento in cui l'operazione *ReadAdminToken* viene valutata come soddisfatta, sta a significare che il TIS sta leggendo correttamente il token dell'amministratore.

```
private void TISReadAdminToken(LoginContext loginContext, Admin
    admin) throws Failure {
    if (new ReadAdminToken(loginContext, admin).check())
      System.out.println("i'm reading the token");
}
```

3.2.3 ReadAdminToken

Il codice Java sottoriportato è l'implementazione dello schema *ReadAdmin-Token* presente all'interno del capitolo 3.1.1.

Da come si può osservare nella sua controparte in notazione Z, la parte della dichiarazione degli attributi dello schema (quindi le eventuali variabili e gli insiemi dello schema) è riprodotta qui attraverso la dichiarazione dei dati membro di questa classe. Un discorso analogo può essere applicato per quanto riguarda la parte delle dichiarazioni dei predicati, che qui avviene nella funzione *predicates*.

Il funzionamento di questa classe (dal punto di vista della libreria Java Z+JSetL) è descritto nel punto precedente, durante la spiegazione dell'operazione *TISReadAdminToken*.

```
public final class ReadAdminToken extends OperationSchema {
  public LoginContext loginContext;

  public Xi<Admin> admin;

  public ReadAdminToken(LoginContext loginContext, Admin admin)
      throws Failure {
    this.loginContext = loginContext;
    this.admin = new Xi<>(new Admin((String)
        admin.rolePresent.getValue()), admin);

    predicates();
  }

  @Override
  protected void predicates() throws Failure {
    add(status().in(new LSet(new HashSet<>(Arrays.asList(new
        LVar("", "quiescent"), new LVar("",
        "waitingRemoveTokenFail")))));
    add(enclaveStatus().eq(new LVar("", "enclaveQuiescent")));
    add(admin.immutable().rolePresent.eq(new LVar("", "nil")));
    add(adminTokenPresence().eq(new LVar("", "present")));

    add(enclaveStatus_().eq(new LVar("", "gotAdminToken")));
  }
```

```
    private LVar adminTokenPresence() { return
        loginContext.enclaveContext.idStation.adminToken.adminTokenPresence;
        }

    private LVar enclaveStatus() { return
        loginContext.enclaveContext.idStation.internal.enclaveStatus;
        }

    private LVar enclaveStatus_() { return
        (loginContext.enclaveContext.idStation.internal.enclaveStatus
        = new LVar("")); }

    private LVar status() { return
        loginContext.enclaveContext.idStation.internal.status; }
}
```

3.2.4 TISValidateAdminToken

Il codice Java sottoriportato è l'implementazione dello schema *TISValidateAdminToken* presente nel capitolo 3.1.2, cioè lo schema che raccoglie ed astrae la logica completa per l'operazione di validazione del token dell'amministratore.

Possiamo vedere all'interno dell'operazione *TISValidateAdminToken*, le operazioni che la compongono, cioè le operazioni *ValidateAdminTokenOK*, *ValidateAdminTokenFail* e *LoginAborted* (tutte presenti all'interno del capitolo 3.1.2).

La logica di questa funzione, fa si che solo una di queste operazioni possa risultare soddisfatta, infatti solo uno dei 3 casi successivi può essere vero nello stesso momento: se il token è stato validato (ValidateAdminTokenOK) di certo il token non può essere contemporaneamente non validato (ValidateAdminTokenOK) oppure. Questo ragionamento può essere applicato anche per le due successive operazioni (*ValidateAdminTokenFail* e *LoginAborted*) all'interno di *TISValidateAdminToken*.

```
private void TISValidateAdminToken(LoginContext loginContext)
    throws Failure {
  if (new ValidateAdminTokenOK(loginContext).check())
    System.out.println("the token is correct");
  else if (new ValidateAdminTokenFail(loginContext).check())
    System.out.println("the token is not valid");
  else if (new LoginAborted(loginContext).check())
    System.out.println("login Aborted");
}
```

3.2.5 ValidateAdminTokenOK

Il codice Java sottoriportato è l'implementazione dello schema *ValidateAdminTokenOK* presente all'interno del capitolo 3.1.2. Il funzionamento di questa classe (dal punto di vista della libreria Java Z+JSetL) è descritto all'interno del suddetto capitolo: se questa operazione viene soddisfatta, allora il token inserito dall'amministratore viene considerato valido.

```
public final class ValidateAdminTokenOK extends OperationSchema {
  public LoginContext loginContext;

  public ValidateAdminTokenOK(LoginContext loginContext) throws
      Failure {
    this.loginContext = loginContext;

    predicates();
  }

  @Override
  protected void predicates() throws Failure {
    add(enclaveStatus().eq(new LVar("", "gotAdminToken")));
    add(adminTokenPresence().eq(new LVar("", "present")));
    add(AdminTokenOK(currentAdminToken()));
    add(enclaveStatus_().eq(new LVar("", "enclaveQuiescent")));
  }

  private LVar adminTokenPresence() { return
      loginContext.enclaveContext.idStation.adminToken.adminTokenPresence;
      }
```

```
  private LVar enclaveStatus() { return
      loginContext.enclaveContext.idStation.internal.enclaveStatus;
      }

  private LVar enclaveStatus_() { return
      (loginContext.enclaveContext.idStation.internal.enclaveStatus
      = new LVar("")); }

  private LVar currentAdminToken() { return
      loginContext.enclaveContext.idStation.adminToken.currentAdminToken;
      }

  private Constraint AdminTokenOK(LVar currentAdminToken) {
      var rightToken = new LVar("", "0001");

      return currentAdminToken.eq(rightToken);
  }
}
```

3.2.6 ValidateAdminTokenFail

Il codice Java sottoriportato è l'implementazione dello schema *ValidateAdminTokenFail* presente all'interno del capitolo 3.1.2. Il funzionamento di questa classe (dal punto di vista della libreria Java Z+JSetL) è descritto all'interno del suddetto capitolo: se questa operazione viene soddisfatta, allora il token inserito dall'amministratore viene considerato come non valido.

```
public final class ValidateAdminTokenFail extends OperationSchema
    {
  public LoginContext loginContext;

  public ValidateAdminTokenFail(LoginContext loginContext) throws
      Failure {
      this.loginContext = loginContext;
      predicates();
  }

  @Override
  protected void predicates() throws Failure {
    add(enclaveStatus().eq(new LVar("", "gotAdminToken")));
    add(adminTokenPresence().eq(new LVar("", "present")));
    add(NotAdminTokenOK(currentAdminToken()));
    add(enclaveStatus_().eq(new LVar("",
        "waitingRemoveAdminTokenFail")));
  }

  private LVar adminTokenPresence() { return
      loginContext.enclaveContext.idStation.adminToken.adminTokenPresence;
      }

  private LVar enclaveStatus() { return
      loginContext.enclaveContext.idStation.internal.enclaveStatus;
      }

  private LVar enclaveStatus_() { return
      (loginContext.enclaveContext.idStation.internal.enclaveStatus
      = new LVar("")); }

  private LVar currentAdminToken() { return
      loginContext.enclaveContext.idStation.adminToken.currentAdminToken;
      }
```

```
    private Constraint NotAdminTokenOK(LVar currentAdminToken) {
      var rightToken = new LVar("", "0001");
      return currentAdminToken.neq(rightToken);
    }
}
```

3.2.7 LoginAborted

Il codice Java sottoriportato è l'implementazione dello schema *LoginAborted* presente all'interno del capitolo 3.1.2. Il funzionamento di questa classe (dal punto di vista della libreria Java Z+JSetL) è descritto all'interno del suddetto capitolo: se questa operazione viene soddisfatta, allora significa che a seguito di una azione sbagliata da parte dell'amministratore, l'operazione di login viene cancellata.

```
public final class LoginAborted extends OperationSchema {
  public LoginContext loginContext;

  public LoginAborted(LoginContext loginContext) throws Failure {
    this.loginContext = loginContext;

    predicates();
  }

  @Override
  protected void predicates() throws Failure {
    add(adminTokenPresence().eq(new LVar("", "absent")));
    add(enclaveStatus().eq(new LVar("", "gotAdminToken")));
  }

  private LVar adminTokenPresence() {
    return
        loginContext.enclaveContext.idStation.adminToken.adminTokenPresence;
  }

  private LVar enclaveStatus() {
    return
        loginContext.enclaveContext.idStation.internal.enclaveStatus;
  }
}
```

3.2.8 TISCompleteFailedAdminLogon

Il codice Java sottoriportato è l'implementazione dello schema *TISCompleteFailedAdminLogon* presente al capitolo 3.1.3. Il funzionamento di questa classe (dal punto di vista della libreria Java Z+JSetL) è descritto all'interno del suddetto capitolo: se questa operazione viene soddisfatta, allora l'operazione di login dell'amministratore viene considerata completamente fallita e terminata.

```
private void TISCompleteFailedAdminLogon(LoginContext
    loginContext, Admin admin) throws Failure {
    if (new FailedAdminTokenRemoved(loginContext, admin).check())
      System.out.println("the token was removed in a bad way");
    else if (new WaitingAdminTokenRemoval(loginContext).check())
      System.out.println("i'm waiting for you to remove the
          token");
}
```

3.2.9 FailedAdminTokenRemoved

Il codice Java sottoriportato è l'implementazione dello schema *FailedAdminTokenRemoved* presente all'interno del capitolo 3.1.3. Il funzionamento di questa classe (dal punto di vista della libreria Java Z+JSetL) è descritto all'interno del suddetto capitolo: se questa operazione viene soddisfatta, allora significa che a seguito del fallimento dell'operazione di login, l'amministratore ha correttamente estratto il suo token.

```
public final class FailedAdminTokenRemoved extends
    OperationSchema {
  public LoginContext loginContext;
  public Xi<Admin> admin;

  public FailedAdminTokenRemoved(LoginContext loginContext, Admin
      admin) throws Failure {
    this.loginContext = loginContext;
    this.admin = new Xi<>(new Admin((String)
        admin.rolePresent.getValue()), admin);
    predicates();
  }

  @Override
  protected void predicates() throws Failure {
    add(enclaveStatus().eq(new LVar("",
        "waitingRemoveAdminTokenFail")));
    add(adminTokenPresence().eq(new LVar("", "absent")));
    add(enclaveStatus_().eq(new LVar("", "enclaveQuiescent")));
  }

  private LVar adminTokenPresence() { return
      loginContext.enclaveContext.idStation.adminToken.adminTokenPresence;
      }
  private LVar enclaveStatus() { return
      loginContext.enclaveContext.idStation.internal.enclaveStatus;
      }
  private LVar enclaveStatus_() { return
      (loginContext.enclaveContext.idStation.internal.enclaveStatus
      = new LVar("")); }
  private LVar status() { return
      loginContext.enclaveContext.idStation.internal.status; }
}
```

3.2.10 WaitingAdminTokenRemoval

Il codice Java sottoriportato è l'implementazione dello schema *WaitingAdminTokenRemoval* presente all'interno del capitolo 3.1.3. Il funzionamento di questa classe (dal punto di vista della libreria Java Z+JSetL) è descritto all'interno del suddetto capitolo: se questa operazione viene soddisfatta, allora significa che a seguito del fallimento dell'operazione di login, l'amministratore non ha ancora disinserito il token e quindi la situazione è contemporaneamente "ferma", in attesa di una azione da parte dell'amministratore.

```
public final class WaitingAdminTokenRemoval extends
    OperationSchema {
  public LoginContext loginContext;

  public WaitingAdminTokenRemoval(LoginContext loginContext)
      throws Failure {
    this.loginContext = loginContext;
    predicates();
  }

  @Override
  protected void predicates() throws Failure {
    add(enclaveStatus().eq(new LVar("",
        "waitingRemoveAdminTokenFail")));
    add(adminTokenPresence().eq(new LVar("", "present")));
  }

  private LVar adminTokenPresence() { return
      loginContext.enclaveContext.idStation.adminToken.adminTokenPresence;
      }
  private LVar enclaveStatus() { return
      loginContext.enclaveContext.idStation.internal.enclaveStatus;
      }
}
```

Capitolo 4

Conclusioni e sviluppi futuri

I **problemi affrontati** che hanno portato alla realizzazione del materiale prodotto in questa tesi sono stati i seguenti:

- Capire gli obiettivi della tesi e decidere il piano da seguire per portarli a termine.
- Prendere coscienza dello "stack tecnologico" da utilizzare nel corso di questo progetto, cioè la libreria Java + JSetL e la Notazione Z notando che questi ultimi sono "strumenti" complessi e per apprenderli e poi utilizzarli nel modo corretto è presente una curva di apprendimento non banale soprattutto per il cambio di paradigma che bisogna effettuare.
- Effettuare una fase di analisi sulla specifica formale [2] scritta in linguaggio Z del *TIS*, rispettando e prendendo in considerazione la complessità della sua architettura, per individuare un possibile insiemi di schemi di stati ed operazioni, per formare il prototipo da sviluppare.

I **risultati ottenuti** sono stati:

- La realizzazione della libreria *Z-JSetL* il cui scopo era aiutarmi a scrivere in modo più semplice l'implementazione del prototipo.
- La realizzazione di un prototipo attraverso l'implementazione di una operazione specifica (e quindi autonoma) del *TIS*.

gli **sviluppi futuri** di questo progetto possono dirigersi in più direzioni:

- Estendere il prototipo "orizzontalmente", quindi aggiungere altre operazioni (ad esempio l'operazione del logout dell'admin).
- Estendere il prototipo "verticalmente", quindi se lo si ritiene necessario, raffinare questa operazione ed implementare alcuni dettagli aggiuntivi.
- Estendere la libreria Z-JSetL cercando di generalizzare la logica contenuta facendo si che i costrutti che sono al suo interno possano essere applicati su tutti i concetti della notazione Z. Quindi in sostanza dare e migliorare la coerenza e la consistenza tra la libreria e Z.

Riferimenti bibliografici

[1] Janet Barnes and David Stokes. *Tokeeneer Id Station Project Plan*. Praxis High Integrity Systems Limited, 20 Manvers Street, Bath, UK. 19th August 2008.

[2] Janet Barnes and David Cooper. *Tokeeneer Id Station Formal Specifications*. Praxis High Integrity Systems Limited, 20 Manvers Street, Bath, UK. 14th August 2008.

[3] Tokeneer project. Ada Core, New York, USA, 2008.

[4] Davies, Jim; Woodcock, Jim (1996). *Using Z: Specification, Refinement and Proof*. International Series in Computer Science. Prentice Hall. ISBN 0-13-948472-8.

[5] Gianfranco Rossi, Roberto Amadini and Andrea Fois. *JSetL User's Manual.*. Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università di Parma, Parma, Italy, 2020.