RELAZIONE PROGETTO ISW2 ANNO 2019/2020

Testing on open-source Projects - Prof. De Angelis

Autore:

Alessio Mazzola, Matricola: 0279323

Indice

Capitolo 1

Introduzione

L'obiettivo dello studio da noi praticato è quello di andare ad effettuare attività di testing su classi diverse dei due progetti open-source assegnati, cioè *Bookkeeper* e *Tajo*, dove il primo risulta essere un progetto in comune con gli altri colleghi, mentre il secondo è assegnato attraverso l'algoritmo utilizzato nella deliverable del Prof. Falessi.

In un constesto generale di Verifica & Validazione l'obiettivo diventa quello di rassicurare l'utilizzatore del sistema che quest'ultimo risulta essere adatto allo scopo, quindi si intende effettuare test all'interno del sistema per aumentare il livello di fiducia che si ha intenzione di fornire.

Identificando con S una descrizione del comportamento del Sistema, allora un programma P è una funzione da un dominio D ad un codominio C. Da questa relazione possiamo determinare che il programma P risulta quindi essere corretto se e solo se, considerato ogni elemento del dominio D, allora la funzione, sotto determinati parametri, restituisce sempre un elemento del codominio C che ci aspettiamo. (La funzione P deve quindi essere *Iniettiva*).

Questo approccio risulta essere però problematico poiché, in generale, un dominio può essere estremamente grande, quindi eseguire dei test case su tutti i possibili elementi diventa un operazione estremamente onerosa. In secondo luogo l'attività di testing risulta essere una attività di tipo *Best-Effort*, quindi sarebbe impensabile impiegare una quantità enorme di risorse al fine di eseguire testing su ogni possibile elemento del dominio applicativo del programma.

Possiamo dunque definire il Software Testing come l'esecuzione di alcuni esperimenti in un ambiente controllato al fine di poter acquisire sufficiente fiducia sul suo funzionamento del programma sui quali vengono effettuati i test. Tipicamente ci soffermiamo su aspetti funzionali, comunque può riguardare anche caratteristiche extra-funzionali.

- Il vantaggio che si ha con il software testing è quindi quello di provare se il sistema reale risponde effettivamente alle esigenze per cui era stato creato, oppure se questo manifesta degli eventuali malfunzionamenti
- Lo svantaggio sta nel fatto che il testing non può dimostrare l'assenza di guasti, ma solo la loro presenza (E.W. Dijkstra)

Capitolo 2

Configurazione dei progetti & classi di Test

Per la configurazione dei progetti in locale, è stata effettuata la fork di questi utlimi direttamente dalla repository GitHub. Per quanto concerne i nostri scopi, sono stati eliminati tutti i test presenti di default all'interno dei progetti, soostituendoli con quelli creati. Inoltre, per quanto riguarda gli aspetti di *Automation&Continous Testing*, i progetti sono stati collegati rispettivamente a Travis-CI e SonarCloud.

Travis-CI si occupa dell'esecuzione della build del progetto e dell'esecuzione dei casi di test, mentre SonarCloud esegue un controllo qualitativo del progetto. All'interno del file pom.xml abbiamo inoltre impostato il plugin Jacoco, che permette di calcolare la line-coverage dei casi di test effettuati. Durante la fase di Build eseguita su Travis, quindi, verrà attivato anche Jacoco, il quale andrà a generare un report che successivamente verrà utilizzato da SonarCloud per mostrare la coverage dei casi di test eseguiti.

2.1 Individuazione delle Classi da testare

Per individuare le classi sul quale eseguire testing, è stato sfruttato l'output della milestone del Prof. Falessi, che riportava le classi dei due progetti con diverse metriche calcolate e se queste classi fossero o meno Buggy. In particolare, le metriche di maggior rilievo nella scelta delle classi sono le seguenti:

- Parametro *age* della classe.
- *Numero di Revisioni* della classe.

2.2 Bookkeeper

Per quanto riguarda Bookkeeper, le classi scelte sulle quali eseguire i Test sono le seguenti:

- **BufferedChannel**: Questa classe rappresenta un Layer di bufferizzazione dei Ledger di Bookkeeper, prima che questi vengano effettivamente

scritti su un Log. Un Ledger (unità di base di storage in Bookkeeper) rappresenta quindi una sequenza di Log entries (file che gestisce le scritture ricevute dal client Bookkeeper). Ogni Entry [1] è composta da:

- long LedgerNumber –> ID del Ledger
- long EntryNumber –> ID della entry (unico per ogni entry)
- long LastConfirmed –> ID dell'ultima entry che è stata finalizzata
- byte[] Data –> Dati della entry scritti all'interno del client Bookkeper
- byte[] AuthenticationCode –> Messaggio di autenticazione che contiene tutti i campi della entry.

Per ottimizzare la scrittura dei Ledger, viene quindi utilizzata la classe BufferedChannel, che permette di lasciare in memoria un determinato numero di Legder, fintantoché questi non dovranno essere scritti su File, in modo tale da eseguire la scrittura una sola volta piuttosto che tante scritture in tempi diversi. La finalizzazione della scrittura è determinata da una variabile speciale identificata da *unpersistedByteBound*, che appunto determina il limite massimo di bytes che possono non essere finalizzati. Una volta che questo limite viene superato, viene forzata la scrittura sul File delle entry log, in modo tale da prevenire la perdita di dati.

- **BookieStatus**: Questa classe va a determinare lo stato di un Bookie. Un Bookie rappresenta un storage server individuale di Bookkeeper, i quali contengono Ledger e comunicano in modo distributio. Lo stato di un Bookie può quindi variare, infatti questo può avere due diversi valori:

  - READ_ONLY: In questo stato il Bookie è disponibile per la sola lettura
  - READ_WRITE: In questo stato il Bookie è disponibile sia per la scrittura che per la lettura.

  In particolare una lettura permette di andare a reperire informazione scritte sul Bookie, mentre una scrittura permette di andare a finalizzare sul Bookie delle nuove informazioni. Il numero esatto di Bookie presente all'interno di Bookkeeper varia a seconda dell'istallazione, comunque risulta essere dipendente dalla grandezza del Quorum-MODE che viene scelto. Ogni Bookie [2] è composto da:

  - bookiePort: Porta TCP sul quale il Bookie é in ascolto (Default: 3181)
  - zkServers: Lista di server Zookeeper (Default: localhost:2181)
  - journalDirectory: Directory dove sono salvati i log delle write dei Bookie
  - ledgerDirectory: Directory all'interno della quale sono situate i log contenenti le LedgerEntry

[1] Riferimento alla Entry nel link: https://bookkeeper.apache.org/docs/4.5.1/getting-started/concepts/#bookies

[2] Riferimento ai Bookie nel link: https://bookkeeper.apache.org/docs/4.6.0/admin/bookies/

2.3 Tajo

Per quanto riguarda Tajo, le classi scelte sulle quali eseguire i Test sono le specializzazioni della casse astratta Datum, in particolare andiamo a testare le seguenti classi:

- **Int4Datum**
- **Float4Datum**
- **TextDatum**
- **TimeDatum**

La classe Datum rappresenta la radice della gerarchia delle tipologie di dati nativi di Oracle. Ogni specializzazione va quindi a rappresentare le operazioni per una determinata tipologia di dato. Per la creazione dei datum, viene sfruttato il pattern dell'Abstract Factory, quindi avremo una singola classe, che prende il nome DatumFactory, che si andrà ad occupare della creazione delle diverse istanze datum a seconda della tipologia scelta.

Capitolo 3

Category Partition

Nella creazione della Category Partition è stata sfruttata la procedura descritta durante il corso, quindi in particolare è stato partizionato il dominio di input includendo un elemento per ogni partizione. La qualità della partizione risulta essere determinata da:

- Chiarezza dei requisiti e delle specifiche;
- Familiarità del System Under Test (SUT);
- Conoscenza dell'implementazione degli elementi del sistema.

Da queste linee guida ricaviamo quindi la category partition per i metodi pubblici esposti dalle classi che abbiamo considerato. In particolare avremo:

3.1 Bookkeeper

3.1.1 BufferedChannel

Andiamo ad analizzare i metodi pubblici esposti all'interno della classe:

- **public void write(ByteBuf src)**: Questo metodo si occupa di scrivere il contenuto dell'oggetto complesso ByteBuf sul fileChannel associato al bufferedChannel. Trattandosi di un oggetto complesso ne possiamo considerare una istanza nulla oppure una istanza corretta. Possiamo notare che all'interno del metodo write viene chiamata la funzione readebleBytes() che va a determinare il numero di byte leggibili all'interno del ByteBuf. Considerando questo aspetto, possiamo costruire un oggetto di tipo ByteBuf nel seguente modo:
  - Istanza Null
  - Correttamente istanziato –> len = 0
  - Correttamente istanziato –> len > 0

  Notiamo inoltre che all'interno del metodo sono presenti anche due variabili ulteriori, cioè *unpersistedBytes* ed *unpersistedByteBound*. La prima

variabile individua il numero di bytes che ancora non sono diventati persistenti (e quindi che non sono stati scritti sul file) mentre la seconda variabile identifica un "Bound", cioè un limite al numero dei bytes che possono rimenere non persistenti. Una volta superato questo limite verrà dunque forzata l'operazione di Write assieme alla funzione di flush(). I valori che possono essere assunti saranno quindi:

  - unpersistedByteBound –> <= 0;
  - unpersistedByteBound –> > 0;

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - Null, unpersistedByteBound = -1
  - src.len = 0, unpersistedByteBound = 1
  - src.len = 1, unpersistedByteBound = 1

- **public int read(ByteBuf dest, long pos, int lenght)**: Questo metodo si occupa di leggere i byte dal fileChannel e scriverli nella destinazione individuata dalla variabile ByteBuf dest. Vengono inoltre considerate le varibili di input *pos* e *lenght* che vanno ad indicare rispettivamente la posizione dalla quale iniziare la lettura e la lunghezza de buffer stesso. Per la creazione di una istanza valida dobbiamo considerare anche la variabile *writeCapacity*, la quale va ad indicare la capacità di scrittura del buffer. A questo punto possiamo andare a considerare i seguenti il seguente partizionamento delle variabili:

  - WriteCapacity –> =0 , > 0
  - ByteBuf dest –> Null, Correttamente Istanzato con len = 0, Correttamente Istanziato > 0
  - long pos –> <writeCapacity, =writeCapacity, >writeCapacity
  - int leght –> <=writeCapacity-pos && <=dest.len , <=writeCapacity - pos && >dest.len, >writeCapacity-pos

  La test suite minimale sarà quindi individuata da:

  - 0, null, -2, -1
  - 0, len = 0, 0, 1
  - 1, len = 1, 2, 2

3.1.2 BookieStatus

Andiamo ad analizzare i metodi pubblici esposti all'interno della classe. In questo caso nella category partiton sono stati considerati anche quei metodi che non presentano alcun indicatore di visibilità. Questa scelta è stata effettuata in quanto nella classe di test è stato comunque possibile richiamare il metodo per eseguirne il testing.

- **public BookieStatus parse(BufferedReader reader)**: Questo metodo si occupa dell'esecuzione del parsing di un determinato BookieStatus andando a restituire come valori di ritorno il BookieStatus stesso se il parsing avviene con successo, mentre se dovessero esserci degli errori viene restituito il valore null. Essendo bufferedReader un oggetto complesso, possiamo pensare a tre diversi tipologie di istanziazione:

  – Istanza null;
  – Correttamente Istanziato con un reader che non rispetta controlli del metodo
  – Correttamente Istanziato con un reader che rispetta i controlli del metodo, quindi ottenendo un BookieStatus corretto

  In questo caso per l'istanziazione di questo BufferedReader andremo a sfruttare una istanza StringReader, la quale andrà a fornire al reader una stringa del tipo: *LayoutVersion, BookieMode, LastUpdateTime*.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  – reader = Null
  – reader = "IncorrectString"
  – reader = layoutVersion+bookieMode+lastUpdateTime

- **void readFromDirectories(List<File> dir)**: Questo metodo si occupa della lettura di un Bookie situato all'interno di un file presente nella lista delle direcotory passate come input al metodo. Se la lettura avviene con successo viene eseguito l'update dello stato del Bookie, mentre se questa fallisce o il Bookie risulta non essere leggibile, allora verrà saltato per tentare la lettura del successivo file presente all'interno delle directory. Essendo List<File> un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  – Istanza null;
  – Correttamente istanziato con directory inesistente;
  – Correttamente Istanziato con directory esistente e con un Bookie al suo interno leggibile.

  Possiamo a questo punto andare a determinare la test suite minimale, sfruttando la variabile *pathDir* [1] individuata da:

  – pathDir = null
  – pathDir= "IncorrectPathToDir"
  – pathDir = "CorrectPathToDir"

- **void writeToDirectories(List<File> dir)**: Questo metodo si occupa di andare a scrivere lo status di un Bookie all'interno di multiple directory, individuate dalla variabile di input dir. Essendo List<File> un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

[1] La variabile pathDir è una stringa e viene utilizzata con il costruttore dell'istanza File che verrà successivamente aggiunta alla lista delle directory.

  - Istanza null;
  - Correttamente istanziato con directory inesistente;
  - Correttamente Istanziato con directory esistente.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - pathDir = null
  - pathDir= "IncorrectPathToDir"
  - pathDir = "CorrectPathToDir"

3.2 Tajo

3.2.1 Int4Datum

Andiamo ad analizzare i seguenti metodi pubblici esposti all'interno della classe:

- **public Datum equalsTo(Datum datum)**: Questo metodo si occupa di verificare se il tipo di Datum preso in input sia identico al Datum che richiama il metodo. Il metodo, anche se restituisce l'oggetto complesso Datum, andrà successivamente a convertirlo in un valore Booleano, se l'istanza del datum che si aspetta il metodo è presente all'interno dello switch-case, quindi in particolare restituirà true se i due datum hanno lo stesso valore, false altrimenti. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con valori diversi;
  - Correttamente istanziato con valori uguali.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createInt4("sameInt")
  - datum = DatumFactory.createInt4("anotherInt")

- **public int compareTo(Datum datum)**: Questo metodo ritorna un valore intero a seconda che il datum preso in input sia maggiore, minore o uguale al datum che richiama la funzione. In particolare il metodo esegue uno switch-case per determinare con quale datum ha a che fare, e se quindi effettivamente il metodo può portare ad un risultato corretto. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;

  - Correttamente istanziato con datum in input maggiore rispetto al datum chiamante
  - Correttamente istanziato con datum in input minore rispetto al datum chiamante;
  - Correttamente istanziato con datum in input uguale al datum chiamante;

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createInt4("sameInt+1")
  - datum = DatumFactory.createInt4("sameInt-1")
  - datum = DatumFactory.createInt4("sameInt")

- **public Datum plus(Datum datum)**: Questo metodo ritorna un Datum con valore pari alla somma del datum chiamante e del datum preso in input. Non tutte le tipologie di datum possono essere tra loro sommate, quindi viene eseguito un switch-case all'interno del metodo che si occupa prima di tutto della determinazione del tipo di datum preso in input. Successivamente, se il tipo è presente all'interno della casistica consentita, verrà eseguito il correttamente il metodo. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con datum dello stesso tipo;
  - Correttamente istanziato con datum di tipologia diversa.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createInt4("aInt")
  - datum = DatumFactory.createFloat4("aFloat")

- **public Datum minus(Datum datum)**: Questo metodo ritorna un Datum con valore pari alla differenza del datum chiamante e del datum preso in input. Come per il metodo precedente, anche in questo caso viene eseguito un switch-case per la determinazione delle diverse tipologie di datum, in modo da capire se effettivamente sia consentita la chiamata del metodo tra il datum chiamante e quello preso in input. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con datum dello stesso tipo;
  - Correttamente istanziato con datum di tipologia diversa.

Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createInt4(“aInt”)
  - datum = DatumFactory.createFloat4(“aFloat”)

- **public Datum multiply(Datum datum)**: Questo metodo ritorna un Datum con valore pari alla moltiplicazione tra datum chiamante e del datum preso in input. Come per il metodo precedente, anche in questo caso viene eseguito un switch-case per la determinazione delle diverse tipologie di datum, in modo da capire se effettivamente sia consentita la chiamata del metodo tra il datum chiamante e quello preso in input. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con datum dello stesso tipo;
  - Correttamente istanziato con datum di tipologia diversa.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createInt4(“aInt”)
  - datum = DatumFactory.createFloat4(“aFloat”)

- **public Datum divide(Datum datum)**: Questo metodo ritorna un Datum con valore pari alla divisione tra datum chiamante e del datum preso in input. Come per il metodo precedente, anche in questo caso viene eseguito un switch-case per la determinazione delle diverse tipologie di datum, in modo da capire se effettivamente sia consentita la chiamata del metodo tra il datum chiamante e quello preso in input. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con datum dello stesso tipo;
  - Correttamente istanziato con datum di tipologia diversa.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createInt4(“aInt”)
  - datum = DatumFactory.createFloat4(“aFloat”)

- **public Datum modular(Datum datum)**: Questo metodo ritorna un Datum con valore pari al modulo del datum chiamante rispetto a quello preso in input. Come per il metodo precedente, anche in questo caso viene eseguito un switch-case per la determinazione delle diverse tipologie di datum, in modo da capire se effettivamente sia consentita la chiamata del metodo tra il datum chiamante e quello preso in input. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato, datum della medesima tipologia con valore diverso da 0;
  - Correttamente istanziato, datum della medesima tipologia con valore pari a 0;
  - Correttamente istanziato con datum di tipologia diversa.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createInt4("int!=0")
  - datum = DatumFactory.createInt4("int==0")
  - datum = DatumFactory.createFloat4("float")

3.2.2 Float4Datum

Non viene riportata la category partition di questa classe poiché risulta essere logicamente equivalente a quanto visto nella classe Int4Datum.

3.2.3 TextDatum

Andiamo ad analizzare i metodi pubblici esposti all'interno della classe:

- **public int compareTo(Datum datum)**: Questo metodo ritorna un valore intero a seconda che il datum preso in input sia uguale o diverso al datum che richiama la funzione. In particolare il metodo esegue uno switch-case per determinare con quale tipologia di Datum è stata presa come input, e se quindi effettivamente il metodo può essere generato un risultato. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con tipologia del datum presente nello switch-case;
  - Correttamente istanziato con tipologia del datum non presente nello switch-case.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createText(“aText”)
  - datum = DatumFactory.createInt4(“aInt”)

- **public boolean equals(Object obj)**: Questo metodo ritorna un valore booleano eseguendo la comprazione di obj preso in input e del datum chiamante. In particolare, se i due sono uguali, allora verrà restituito il valore true, false altrimenti. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con obj uguale al datum;
  - Correttamente istanziato con obj diverso dal datum;

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - obj = null
  - obj = (TextDatum) DatumFactory.createText(“text”)
  - datum = (Int4Datum) DatumFactory.createInt4(“int”)

- **public Datum equalsTo(Datum datum)**: Questo metodo si occupa di verificare se il tipo di Datum preso in input sia identico al Datum che richiama il metodo. Il metodo, anche se restituisce l’oggetto complesso Datum, andrà successivamente a convertirlo in un valore Booleano, se l’istanza del datum che si aspetta il metodo è presente all’interno dello switch-case, quindi in particolare restituirà true se i due datum hanno lo stesso valore, false altrimenti. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  - Istanza null;
  - Correttamente istanziato con valori diversi;
  - Correttamente istanziato con valori uguali;
  - Correttamente istanziato con tipologia non presente nello switch-case

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  - datum = null
  - datum = DatumFactory.createText(“sametext”)
  - datum = DatumFactory.createText(“difText”)
  - datum = DatumFactory.createInt4(“int”)

3.2.4 TimeDatum

Andiamo ad analizzare i metodi pubblici esposti all'interno della classe:

- **public Datum plus(Datum datum)**: Questo metodo ritorna un Datum con valore pari alla somma del datum chiamante e del datum preso in input. Non tutte le tipologie di datum possono essere tra loro sommate, quindi viene eseguito un switch-case all'interno del metodo che si occupa prima di tutto della determinazione del tipo di datum in input, successivamente, se il tipo è presente all'interno della casistica consentita in modo da produrre un risultato. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  – Istanza null;
  – Correttamente istanziato con datum avente tipologia presente nello switch-case;
  – Correttamente istanziato con datum avente tipologia non presente nello switch-case.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  – datum = null
  – datum = DatumFactory.createInterval("someTime")
  – datum = DatumFactory.createInt4("int")

- **public Datum minus(Datum datum)**: Questo metodo ritorna un Datum con valore pari alla differenza del datum chiamante e del datum preso in input. Come per il metodo precedente, anche in questo caso viene eseguito un switch-case per la determinazione delle diverse tipologie di datum, in modo da capire se effettivamente sia consentito il metodo tra il datum chiamante e quello preso in input. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  – Istanza null;
  – Correttamente istanziato con datum avente tipologia presente nello switch-case;
  – Correttamente istanziato con datum avente tipologia non presente nello switch-case.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  – datum = null
  – datum = DatumFactory.createInterval("someTime")
  – datum = DatumFactory.createInt4("int")

- **public Datum equalsTo(Datum datum)**: Questo metodo si occupa di verificare se il tipo di Datum preso in input sia identico al Datum che richiama il metodo. Il metodo, anche se restituisce l'oggetto complesso Datum, andrà successivamente a convertirlo in un valore Booleano, se l'istanza del datum che prende in input sarà di tipo "TIME". Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  – Istanza null;
  – Correttamente istanziato con datum avente tipologia presente nello switch-case;
  – Correttamente istanziato con datum avente tipologia non presente nello switch-case.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  – datum = null
  – datum = DatumFactory.createTime("someTime")
  – datum = DatumFactory.createInt4("aInt")

- **public int compareTo(Datum datum)**: Questo metodo ritorna un valore intero a seconda che il datum preso in input sia uguale o diverso al datum che richiama la funzione. In particolare il metodo esegue la comparazione solamente se il datum in input è di tipo "TIME", andando successivamente a restituire un valore booleano che va a determinare se i valori dei due datum comparati sia uguali o meno, restituendo true o false a seconda dei casi. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  – Istanza null;
  – Correttamente istanziato con tipologia del datum "TIME";
  – Correttamente istanziato con tipologia del datum diversa.

  Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

  – datum = null
  – datum = DatumFactory.createTime("someTime")
  – datum = DatumFactory.createInt4("aInt")

- **public boolean equals(Object obj)**: Questo metodo ritorna un valore booleano eseguendo la comprazione di obj preso in input e del datum chiamante. In particolare, se i due sono uguali, allora verrà restituito il valore true, false altrimenti. Essendo Datum un oggetto complesso, possiamo pensare a tre diverse tipologie di istanziazione:

  – Istanza null;
  – Correttamente istanziato con obj uguale al datum;

– Correttamente istanziato con obj diverso dal datum.

Possiamo a questo punto andare a determinare la test suite minimale, individuata da:

– datum = null
– datum = DatumFactory.createTime(“sameTime”)
– datum = DatumFactory.createTime(“differentTime”)

Capitolo 4

Implementazione dei casi di Test

Per l'implementazione dei casi di test è stato sfruttato il framework JUnit. In particolare per l'esecuzione delle diverse casistiche ottenute tramite la category partition, la classe sfrutta l'annotazione @RunWith(Parameterized.class), in modo tale da poter andare a definire i parametri di input che prende il costruttore della classe al fine di eseguire i test con differenti valori assegnati. Oltre a questo, per la preparazione dell'ambiente di test, sono stati sfruttate anche le annotazioni fornite da JUnit di @Before, @After, @BeforeClass e @AfterClass per andare a richiamare dei metodi specifici necessari alla preparazione dell'ambiente di test prima dell'esecuzione effettiva dei metodi.

4.1 Test-Case e preparazione dell'ambiente di Test

Andiamo adesso a vedere nel dettaglio i metodi di test che sono stati sviluppati.

4.1.1 Bookkeeper

BufferedChannel

Per *BufferedChannel* sono state create due diverse classi che ne implementano i test, in particolare una per la lettura e l'altra per la scrittura (quindi per i metodi *read()* e *write()* esposti). Nei casi visti attraverso la category partition abbiamo dovuto commentare i valori della seconda run dei test, in quanto va a generare un ciclo infinito, dovuto al fatto che i valori di writeCapacity e la lunghezza del writeBuffer sono entrambi pari a 0.

Per quanto riguarda la preparazione dell'ambiente di test, invece, sono stati utilizzati i seguenti metodi:

- **close()**: Questo metodo ha come annotazione @AfterClass, quindi viene eseguito solamente una volta dopo l'esecuzione di tutti i metodi di test. Il suo compito è quello di andare a chiudere il BufferedChannel istanziato precedentemente.

- **validByteBufAlreadyWrited(int length)**: Questo metodo restituisce un oggetto di tipo byteBuf con lunghezza determinata dalla variabile length che risulta essere già popolata da scritture random.
- **byteBufNoWrite(int lenght)**: Questo metodo restituisce un oggetto di tipo byteBuf con lunghezza determinata dalla variabile length.

BookieStatus

Per questa classe è stato possibile testare tutti gli input definiti dal category partition. Per quanto riguarda l'esecuzione dei metodi di test, invece, è stato necessario prepare inizialmente l'ambiente di esecuzione. Questo è stato fatto attraverso le seguenti funzioni:

- **createDir()**: Questo metodo ha come annotazione @BeforeClass, quindi viene eseguito solamente una volta prima dell'esecuzione di tutti i metodi di test. Il suo compito è quello di andare a creare una directory che verrà poi data in input ai metodi *write()* e *read()* esposti da BookieStatus.
- **deleteDir()**: Questo metodo ha come annotazione @AfterClass, quindi viene eseguito solamente una volta dopo l'esecuzione di tutti i metodi di test. Il suo scopo è quello di andare ad eliminare tutte le scritture effettuate dai metodi di test ed anche la cartella creata attraverso il metodo *createDir()* eseguito precedentemente.

una volta che l'ambiente di test è pronto, i metodi read() e write() esposti da BookieStatus potranno essere effettivamente eseguiti, quindi in particolare sarà possibile eseguire una scrittura persistente di un bookie e la sua lettura successivamente.

4.1.2 Tajo

Int4Datum

Per quanto riguarda questa classe, è stato possibile eseguire i test con i valori determinati dalla category partition, senza l'ausilio di funzioni che preparassero l'ambiente di Test. La classe Datum, per essere istanziata, sfutta la classe datumFactory, che implementa il pattern dell'abstract Factory, quindi a seconda della tipologia di Datum che si vuole, si andranno a richiamare diverse funzioni. Per ottenere una istanza corretta di Int4Datum richiameremo la funzione *createInt4(...)*.

Float4Datum

Per quanto riguarda questa classe, è stato possibile eseguire i test con i valori determinati dalla category partition, senza l'ausilio di funzioni che preparassero l'ambiente di Test. Vale lo stesso discorso fatto precedentemente, ma andremo a richiamare la funzione *createFloat4(...)*.

TextDatum

Per quanto riguarda questa classe, è stato possibile eseguire i test con i valori determinati dalla category partition, senza l'ausilio di funzioni che preparassero l'ambiente di Test. Vale lo stesso discorso fatto precedentemente, ma andremo a richiamare la funzione *createText(...)*.

TimeDatum

Per quanto riguarda questa classe, è stato possibile eseguire i test con i valori determinati dalla category partition, senza l'ausilio di funzioni che preparassero l'ambiente di Test. Vale lo stesso discorso fatto precedentemente, ma andremo a richiamare la funzione *createTime(...)*.

4.2 Adeguatezza dei Test attraverso JACOCO

Per quanto concerne la misurazione dell'adeguatezza dei casi di test, è stato utilizzato il plugin *Jacoco*. In particolare, per includerlo all'interno del progetto, è stato modificato il pom principale del progetto, andando ad aggiungere la seguente riga di comando:

Listing 4.1: pom.xml del modulo principale

```
1 <aggregate.report.dir>REFER_MODULE/target/site/jacoco-aggregate
      /jacoco.xml</aggregate.report.dir>
```

Questa infatti ha il compito di andare a comunicare a Jacoco dove andare a posizionare il report una volta generato. Per il funzionamento di Jacoco stesso, invece, è tato aggiunto il plugin all'interno del pom principale del progetto. Inoltre è stato modificato anche il pom contenente le classi di test, aggiungendo come proprità:

Listing 4.2: pom.xml del modulo delle classi da testare

```
1 <properties>
2     <sonar.coverage.jacoco.xmlReportPaths>${project.basedir
          }/../${aggregate.report.dir}</sonar.coverage.jacoco.
          xmlReportPaths>
3 </properties>
```

Questa risulta essere necessaria alla determinazione della locazione del report di jacoco utilizzato da SonarCloud per andare a determinare dove andare a prendere il report per mostrarne la coverage una volta eseguita la build del progetto su travis-CI.

Infine, nel modulo fittizio da noi creato, durante la build del progetto è stato inserito il plugin Jacoco che va ad eseguire il report quando viene eseguita la fase *Verify* di Maven. All'interno di questo pom è stata inoltre necessario immettere la dipendenza con il modulo contenente le classi da voler testare. Prendendo come esempio Bookkeeper, quindi, si avrà:

Listing 4.3: pom.xml del modulo fittizio

```
1     <dependencies>
2         <!-- TEST DEPENDECIES -->
```

```
        <dependency>
            <groupId>org.apache.bookkeeper</groupId>
            <artifactId>CLASS_TEST_MODULE</artifactId>
            <version>4.11.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
```

Per quanto riguarda i risultati stessi ottenuti dal report di Jacoco, questi tengono conto di tutti i metodi presenti all'interno della classe. Dobbiamo comunque considerare l'esistenza di metodi che hanno delle istruzioni non raggiungibili, quindi avere un risultato pari al 100% di coverage potrebbe, in alcuni casi, essere impossibile. Nelle classi di test che sono state eseguite, sono stati implementati anche altri metodi di test, differenti da quelli che abbiamo visto nella category partition, per cercare di aumentare il più possibile la coverage calcolata attraverso Jacoco.

4.3 Aumento della Coverage calcolata da Jacoco

Per l'aumento della coverage calcolata da Jacoco, sono stati eseguiti ulteriori casi di test, presenti sia all'interno di nuovi metodi creati per questo scopo, sia all'interno di metodi precedentemente esistenti. Ne riportiamo alcuni per entrambi i progetti:

4.3.1 Bookkeeper

BufferedChannel

Oltre ad altri run eseguiti con valori in input differenti rispetto a quelli visti nella category partition, sono stati aggiunti altri metodi di test. I metodi sono i seguenti:

- **testFlush()**: Questo metodo va a testare la funzione *flush()* esposta all'interno di BufferedChannel, la quale, una volta eseguita, si occupa di andare a scrivere i file all'interno del fileChannel. L'assert di questa funzione si basa proprio su questo, in quanto il metodo flush, ritorna void.
- **testForceWrite()**: Questo metodo va a testare la funzione *forceWrite(...)*, andiamo ad eseguire un Assert sul ritorno stesso della funzione, che identifica la posizione del writeBuffer al momento della chiamata.
- **testClear()**: Questo metodo va a testare la funzione *clear()*, la quale una volta chiamata va ad inizializzare la variabile writerIndex del ByteBuf associato al BufferedChannel di riferimento.
- **testGetNumOfBytesInWriteBuffer()**: Questo metodo va a testare la funzione *getNumOfBytesInWriteBuffer()* andando a verificare il ritorno della funzione tramite l'assert una volta eseguita una scrittura.
- **testGetUnpersistedBytes()**: Questo metodo va a testare la funzione *getUnpersistedBytes()*, la quale ritorna il numero di unpersistedBytes associati al BufferedChannel.

BookieStatus

Gli altri metodi di test aggiunti per aumentare la coverage sono i seguenti:

- **testWritable()**: Questo metodo va a testare la funzione *isInWritable()*, quindi andiamo ad eseguire la verifica del ritorno booleano attraverso l'assert.
- **testWritableAfterReadOnly()**: Questo metodo va a testare la funzione *isInReadOnlyMode()*, quindi ci aspettiamo un valore booleano conforme alla specifica del metodo.
- **testSetToWritable()**: Questo metodo va a testare la funzione *setToWritableMode()*, quindi andiamo a verificare che il valore booleano restituito sia conforme alla specifica del metodo.
- **writeToDirTest()**: Questo metodo va a testare la funzione *writeToDirectories(...)*, per eseguirlo correttamente diamo come input al metodo la directory precedentemente creata attraverso il metodo *createDir()*, verificando successivamente che la scrittura non restituisca eccezioni.

4.3.2 Tajo

Int4Datum

Anche in questo caso, come per Bookkeeper, andiamo ad aumentare la coverege calcolata da Jacoco andando a testare ulteriori metodi che non sono stati considerati all'interno del Category Partition. I metodi aggiunti sono i seguenti:

- **testInverse()**: Questo metodo va a testare la funzione *inverseSign()*, andando ad eseguire la verifica sul valore di ritorno, ottenuto dall'inverso del datum richiamente la funzione.
- **testEqual()**: Questo metodo va a testare la funzione *equal()*, andando quindi a verificare che effettivamente il datum chiamante e quello preso in input abbiano il medesimo valore e siano dello stesso tipo.
- **asTest()**: Questa funzione va a testare tutte le funzioni di tipo "as" esposte dalla classe.

Float4Datum

Non riportiamo i metodi aggiunti in quanto sono logicamente identici a quelli trovati all'interno della classe *Int4Datum*.

TextDatum

Per questa classe, l'unico metodo aggiunto interamente per la covarege è asTest(), che ha la medesima funzione vista all'interno della classe Int4Datum.

TimeDatum

I metodi aggiunti per l'aumento della coverage sono i seguenti:

- **asTest()**: Questa funzione va a testare tutte le funzioni di tipo "as" esposte dalla classe.
- **sizeTest()**: Questa funzione va a testare il metodo *size()*, che va a ritornare la size del datum che richiama il metodo.
- **getTest()**: Questa funzione va a testare i metodi "get" esposti dalla classe, quindi per un datum temporale andiamo a ricavare il valore delle ore, minuti, secondi e millisecondi.

4.4 Mutazioni dei test attraverso PIT

Per la misurazione dell'adeguatezza dei test rispetto alle mutazioni è stato utilizzato il plugin PIT[1]. Per la sua inclusione all'interno del progetto, come visto per il plugin Jacoco, sono stati modificati i pom del progetto stesso. In particolare sono stati aggiunti i seguenti frammenti di codice:

Listing 4.4: pom.xml del modulo delle classi da testare

```
<dependency>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.4.11</version>
</dependency>
```

Listing 4.5: pom.xml del modulo delle classi da testare

```
<plugin>
  <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.4.11</version>
    <configuration>
      <targetClasses>
            <param>PATH_CLASS</param>
        </targetClasses>
        <targetTests>
            <param>PATH_TEST_CLASS</param>
        </targetTests>
    </configuration>
</plugin>
```

Listing 4.6: pom.xml del modulo delle classi da testare

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-maven</artifactId>
      <version>1.4.11</version>
```

[1] Il report completo ottenuto da pit è presente all'interno della cartella pit-reports

```
7         <reportSets>
8           <reportSet>
9               <reports>
10                  <report>report</report>
11              </reports>
12            </reportSet>
13        </reportSets>
14      </plugin>
15    </plugins>
16  </reporting>
```

tutti gli altri pom degli altri moduli, invece, non hanno subito variazioni. Anche in questo caso, come visto per il plugin Jacoco, il numero di muzioni e la percentuali di mutanti "Killed" si riferiscono all'intera classe individuata attraverso il <targetClasses>. Nel numero di mutanti totali, sono presenti anche i mutanti che non vengono coperti, in quanto i casi di test non raggiungono quelle parti del codice. A causa dell'esistenza di porzioni di codice a volte irraggiungibili, in generale, potrebbe essere impossibile avere una copertura dei mutanti pari al 100%.

Per quanto riguarda la copertura delle mutazioni individuate da PIT, la suite test minimale non è risultata adatta a questo scopo ed è stato necessario aggiungere nuovi casi di test per andare a coprire parti del codice che prima non venivano coperte. In particolare possiamo vederle nel dettaglio:

4.4.1 Bookkeeper

BufferedChannel

L'esecuzione di Pit per la classe ha portato alla generazione di 60 mutanti, dei quali, con i test effettuati, 41 di questi risultano essere stati uccisi. Questi mutanti uccisi sono stati ricavati andando a considerare il category partion sviluppato ed andando anche a considerare dei nuovi metodi di test per aumentare la line coverage generale della classe. Le mutazioni prese in considerazione nel conteggio tengono conto anche di porzioni di codice "non coperto" o "irraggiungibili". Possiamo riportare alcuni esempi di mutanti che non sono stati uccisi attraverso i test effettuati:

- **linee 180-181** riferite al metodo *flushAndForceWrite(...)*: L'esecuzione del metodo risulta essere equivalente al SUT per quanto riguarda la strong mutation, ma non per la weak mutation, in quanto la mancata chiamata del metodo flushAndForceWrite(...) necessariamente andrà a produrre un risultato diverso da quello aspettato.
- **linea 266** riferita al metodo *read(...)*: L'esecuzione del meotodo risulta essere equivalente al SUT per quanto riguarda strong mutation, ma non per weak mutation, in quanto non considerando il branch decisionale (readBytes<=0), allora effettivamente si potrebbe avere un indice negativo, e quindi un risultato diverso rispetto a quello aspettato.

BookieStatus

L'esecuzione di Pit per la classe ha portato alla generazione di 27 mutanti, dei quali, con i test effettuati, 18 di questi risultano essere stati uccisi. Come visto per BufferedChannel, anche in questo caso i mutanti uccisi sono ricavati sia attraverso la consierazione della della category partition sia attraverso la considerazione di nuovi metodi di test effettuati per andare ad aumentare la coverage dei metodi sulla classe da testare. Riportiamo anche in questo caso alcuni esempi di mutanti che non sono stati uccisi attraverso i test effettuati:

- **linea 162** in riferimento al metodo *writeToFile(...)*: L'esecuzione del metodo risulta essere equivalente al SUT sia per quanto riguarda weak mutation che strong mutation, infatti non eseguendo il branch decisionale, comunque il risultato effettivo del metodo richiamato non verrebbe alterato.
- **linea 128** in riferimento al metodo *writeToFile(...)*: L'esecuzione del meotodo risulta essere equivalente al SUT per quanto riguarda strong mutation, ma non per weak mutation, in quanto non eseguendo il metodo write(...) effettivamente verrebbe prodotto un risultato diverso da quello aspettato.

4.4.2 Tajo

Int4Datum

L'esecuzione di Pit per la classe ha portato alla generazione di 143 mutanti, dei quali, attraverso i test effettuati, 128 risultano essere stati uccisi. Come visto per Bookkeeper, anche in questo caso i mutanti uccisi sono stati ricavando considerando test provenienti sia dal category partition effettuato che da ulteriori test per l'aumento generale della coverage. Riportiamo di seguito alcuni esempi di mutanti che non sono stati uccisi attraverso i test effettuati:

- **linea 213** in riferimento al metodo *plus(...)*: Risulta essere equivalente al SUT in riferimento alla strong mutation, in quanto viene comunque restituito un oggetto di tipo datum, ma ripetto a weak mutation risulta essere differente, in quanto rimuovendo la chiamata al metodo *plusDays(...)* verrebbe restituito un datum di differente valore.
- **linea 237** in riferimento al metodo *minus(...)*: Risulta essere equivalente al SUT in riferimento alla strong mutation, in quanto viene comunque restituito un oggetto di tipo datum, ma ripetto a weak mutation risulta essere differente, in quanto rimuovendo la chiamata al metodo *plusDays(...)* verrebbe restituito un datum di differente valore.

Float4Datum

L'esecuzione di Pit per la classe ha portato alla generazione di 142 mutanti, dei quali, attraverso i test effettuati, 126 risultano essere stati uccisi. Come visto per Bookkeeper, anche in questo caso i mutanti uccisi sono stati ricavando

considerando test provenienti sia dal category partition effettuato che da ulteriori test per l'aumento generale della coverage. Riportiamo di seguito alcuni esempi di mutanti che non sono stati uccisi attraverso i test effettuati:

- **linea 244** in riferimento al modulo *minus(...)*: Risulta essere equivalente al SUT in riferimento alla strong mutation, in quanto viene effettivamente restitutito il ritorno che il meotodo si aspetta, ma ripetto a weak mutation risulta essere differente, in quanto rimuovendo la chiamata al metodo *plusDays(...)* verrebbe restituito un risultato differente.
- **linea 220** in riferimento al metodo *plus(...)*: Risulta essere equivalente al SUT in riferimento alla strong mutation, in quanto viene effettivamente restitutito il ritorno che il meotodo si aspetta, ma ripetto a weak mutation risulta essere differente, in quanto rimuovendo la chiamata al metodo *plusDays(...)* verrebbe restituito un risultato differente.
- **linea 268** in riferimento al metodo *multiply(...)*: Risulta essere equivalente al SUT in riferimento alla strong mutation, in quanto viene effettivamente restitutito il ritorno che il meotodo si aspetta, ma ripetto a weak mutation risulta essere differente, in quanto andando ad eseguire la mutazione verrebbe restituito un risultato differente rispetto a quello che ci si aspetta.

TextDatum

L'esecuzione di Pit per la classe ha portato alla generazione di 21 mutanti, dei quali, attraverso i test effettuati, 19 risultano essere stati uccisi. Come visto per Bookkeeper, anche in questo caso i mutanti uccisi sono stati ricavando considerando test provenienti sia dal category partition effettuato che da ulteriori test per l'aumento generale della coverage. In questo caso specifico i due mutanti sopravvisuti sono derivati dalla mancanza di coverage per i metodi esposti dalla classe.

TimeDatum

L'esecuzione di Pit per la classe ha portato alla generazione di 40 mutanti, dei quali, attraverso i test effettuati, 37 risultano essere stati uccisi. Come visto per Bookkeeper, anche in questo caso i mutanti uccisi sono stati ricavando considerando test provenienti sia dal category partition effettuato che da ulteriori test per l'aumento generale della coverage. Riportiamo di seguito l'unico mutante che non è stato ucciso con i test effettuati (gli altri due mutanti risultano essere sopravvisuti per mancanza di coverage):

–**linea 143** in riferimento al metodo *minus(...)*:Risulta essere equivalente al SUT in riferimento alla strong mutation, in quanto viene comunque restituito un oggetto di tipo datum, ma ripetto a weak mutation risulta essere differente, in quanto rimuovendo la chiamata al metodo *plusDays(...)* verrebbe restituito un datum di differente valore.

Capitolo 5

Links

Di seguito riportiamo i link per GitHub, Travi-CI e SonarCloud dei progetti analizzati:

5.1 Bookkeeper

- GitHub: https://github.com/GlaAndry/bookkeeper
- Travis-CI: https://travis-ci.com/github/GlaAndry/bookkeeper
- SonarCloud: https://sonarcloud.io/dashboard?id=GlaAndry_bookkeeper

5.2 Tajo

- GitHub: https://github.com/GlaAndry/tajo
- Travis-CI: https://travis-ci.com/github/GlaAndry/tajo
- SonarCloud: https://sonarcloud.io/dashboard?id=GlaAndry_tajo