Eiffel nello sviluppo software in gruppi di lavoro complessi

Mattia Monga

28 novembre 2021

Indice

1 Perché Eiffel

> *Programming languages can be categorized in a number of ways: imperative, applicative, logic-based, problem-oriented, etc. But they all seem to be either an "agglutination of features" or a "crystallization of style." COBOL, PL/1, Ada, etc., belong to the first kind; LISP, APL — and Smalltalk — are the second kind. It is probably not an accident that the agglutinative languages all seem to have been instigated by committees, and the crystallization languages by a single person.*
>
> *— Alan C. Kay, The Early History of Smalltalk, 1993*

Spesso si impara un linguaggio di programmazione per una necessità tecnologica specifica, per realizzare più facilmente un sistema grazie alla disponibilità di librerie evolute, o per partecipare a un gruppo di lavoro che ha fatto scelte tecnologiche cui occorre adeguarsi. Nella carriera di un informatico potrà capitare senz'altro di dover familiarizzare con decine di linguaggi diversi, diffusi capillarmente ma mai incontrati nei propri percorsi formativi: PHP, Javascript, C o magari perfino il Fortran o il Cobol. Generalmente, però, non è per motivi come questi che si può aver voglia di imparare Eiffel, che rimane un linguaggio di nicchia, nonostante una solida tradizione (almeno nei circoli dell'ingegneria del *software*) e una cerchia di utenti (anche industriali) affezionati (si veda per esempio qui[1]).

Eiffel è sostanzialmente il parto della mente geniale di Bertrand Meyer, con l'integrità concettuale e tutti i limiti che ciò

[1] `https://www.eiffel.com/company/customers/`

comporta: è un gioiello concettuale che può insegnarci molto, ma è anche orgogliosamente refrattario alla maggioranza delle convenzioni più diffuse nella pratica dello sviluppo *software*. A cominciare dalla terminologia adottata: in molti casi si tratta di termini assai azzeccati (almeno secondo il mio orecchio, che condivide le radici latine di Meyer), ma che costituiscono un gergo tutto da imparare, anche quando si fa riferimento a concetti già noti (può essere utile consultare il glossario[2]).

Esistono varie implementazioni di Eiffel, tanto che nel 1997 è iniziato un processo di standardizzazione e nel 2006 ci si è accordati su un documento ECMA-367[3] (lo stesso è stato adottato anche come standard ISO/IEC DIS 25436). Si tratta di un documento molto leggibile: il capitolo 7 è un'ottima introduzione a tutto il linguaggio e il capitolo 8 permette di aver chiara la semantica di tutti i costrutti. Lo standard e il classico libro di Meyer "*Object-Oriented Software Construction*" (scritto nel 1992, non aggiornato alla sintassi moderna di Eiffel) sono le fonti adeguate per approfondire la conoscenza del linguaggio e della metodologia di progettazione *software* che sottende e promuove. Questi appunti, invece, si limitano a discutere una delle caratteristiche più peculiari di Eiffel: il cosiddetto *Design By Contract* (marchio registrato!) e la sua utilità nel contesto dello sviluppo software in gruppi di lavoro complessi.

[2] `https://www.eiffel.org/doc/glossary/Glossary`

[3] `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf`

1.1 Gli strumenti

Ci sono sostanzialmente tre comunità attive nello sviluppo di strumenti Eiffel (molte altre se ne trovano sul web, ma paiono abbandonate o, almeno, "dormienti"): *Eiffel Software* (una società legata a B. Meyer), *Gobo Eiffel* e *LibertyEiffel*. Tutte cercano di implementare lo standard, ma con qualche differenza e limitazione. Il prodotto più adatto per i nostri scopi è quello di *Eiffel Software* (`EiffelStudio`), che purtroppo però pare orientato a cambiare (in maniera un po' improvvisa e discutibile) i termini della licenza con cui è distribuito. Al momento l'ultima versione liberamente analizzabile e utilizzabile secondo le condizioni stabilite dalla GPLv2 è la 19.05: farò quindi riferimento a quella.

Per uniformità, potete utilizzare `EiffelStudio` in un contenitore `docker` che trovate qui[4]. Il contenitore permette di utilizzare l'IDE via *browser*, grazie a una connessione `VNC` (si basa su `x11vnc-desktop`[5], di Xiangmin Jiao).

Sulla mia macchina `GNU/Linux` io uso questo *script*, che potete adattare per le vostre esigenze (`tmux` serve per avere un terminale *detached* in cui monitorare i messaggi d'errore):

```
#!/bin/bash
NAME=eiffel
EPASS=eiffelruleznotwithstandingapainfulide
tmux new -d -s $NAME "docker run --rm --name $NAME \
  -p 127.0.0.1:6080:6080 \
  -p 127.0.0.1:5900:5900 \
  -e HOST_UID=$(id -u) \
  -e HOST_GID=$(id -g) \
```

[4] `https://hub.docker.com/repository/docker/mmonga/docker-eiffel`

[5] `https://github.com/x11vnc/x11vnc-desktop`

```
  -e RESOLUT=1600x900 \
  -e VNCPASS=$EPASS \
  -w /home/ubuntu/shared \
  -v $(pwd):/home/ubuntu/shared mmonga/docker-eiffel \
  'startvnc.sh >> /home/ubuntu/.log/vnc.log'"

while ! docker port $NAME >/dev/null; do sleep 1; done
x-www-browser \
 "http://localhost:6080/vnc.html?resize=downscale&aut
 ↪  oconnect=1&password=$EPASS"
```

La parte più importante è il `-v` che impone a `docker` di creare un volume che mappa la *working directory* corrente sul percorso atteso nel contenitore, permettendo così la persistenza dei *file* nel *file-system* ospitante.

2 *Design by Contract*

> *Pacta sunt servanda.*
>
> — *Domizio Ulpiano, De Pactis*

Eiffel è un linguaggio orientato agli **oggetti** che vengono creati a partire da **classi**, organizzate in ***cluster***. Le variabili sono riferimenti staticamente ed esplicitamente tipizzati che possono essere `attached` o no (in questo caso sono `Void`) a un oggetto. In una classe (`class`) si definiscono le `feature` che caratterizzeranno tutti gli oggetti, alcune delle quali (elencate in un'apposita clausola `create`) possono essere usate per **creare** le istanze (il ruolo svolto in altri linguaggi dai "costruttori"). Fra le classi si possono stabilire relazioni di ereditarietà, anche multipla: ci sono poi meccanismi molto flessibili per ridefinire, rinominare e anche nascondere le

feature ereditate, potendo così ottenere anche classi derivate non più conformi al tipo della classe base. Anche questa è una parte molto interessante di Eiffel, ma non verrà approfondita qui.

In un sistema Eiffel c'è sempre una classe ***root*** che è quella che viene istanziata automaticamente dal sistema operativo al momento dell'esecuzione, eseguendone una particolare *feature* di creazione (marcata come ***root*** e vincolata a poter essere eseguita in qualsiasi stato di partenza).

L'idea centrale nell'approccio di Eiffel è quella del **contratto**: ogni componente C del sistema (cioè le classi) realizza un **accordo** fra l'utilizzatore del componente (***client***, chi scrive codice che **usa** le *feature* di C) e il suo implementatore (***supplier***, chi scrive il codice delle *feature* di C). Questo accordo è basato su:

- il **tipo** definito dalla classe, come accade in altri linguaggi *object-oriented*: un *client* potrà creare oggetti istanza di C e attivarne le *feature* per cambiare lo stato dell'oggetto (***command***) o per conoscere lo stato dell'oggetto (***query***) o per fare entrambe le cose insieme (lo stile Eiffel suggerisce però di tenere le cose separate);

- condizioni **invarianti**, cioè valide in tutti gli stati **stabili** di un oggetto, dal momento della creazione fino alla sua eliminazione (via *garbage collection*) dalla memoria del sistema;

- per ogni *feature* $C.f$, le condizioni (dette **precondizioni**) in cui essa può essere attivata con la garanzia che l'esecuzione se terminerà lo farà in uno stato coerente con il contratto;

- per ogni *feature* $C.f$, le condizioni (dette **postcondizioni**) garantite in caso di terminazione della *feature* stessa.

Formalmente, per ogni *feature* $C.f$, le precondizioni P e le postcondizioni Q costituiscono una tripla di Hoare $\{P\}C.f\{Q\}$. Ogni esecuzione di $C.f$ che parta da uno stato che soddisfa la condizione P **termina** in uno stato che soddisfa la condizione Q. Ogni programma che termina è **corretto** se e solo se vale la proprietà precedente. La correttezza può essere poi ricorsivamente estesa a partire dalla *feature root* a un intero sistema Eiffel e lo standard (capitolo 8.9) è attento a dare le condizioni di correttezza per ogni costrutto .

Dal punto di vista dello sviluppo *software* in gruppi di lavoro complessi, quello di contratto è un concetto estremamente interessante perché esplicita il vincolo che sempre esiste fra i due ruoli che ruotano intorno a un pezzo di codice: chi lo usa e chi lo scrive. Questo vincolo esiste sempre, a prescindere dal linguaggio di programmazione usato, ed è la sorgente principale della complessità nello sviluppo *software* che coinvolge più soggetti. Naturalmente nemmeno Eiffel può riuscire a rendere completamente trasparente le implicazioni di tale vincolo (siamo di fronte a quella che è sostanzialmente una istanza del cosiddetto *frame problem*[6]), ma ci aiuta più di altri linguaggi nell'esplicitazione, che costa però creatività e fatica.

Il contratto rende esplicito:

- le responsabilità e le collaborazioni di una classe tramite il suo tipo e le condizioni invarianti;

[6] `https://plato.stanford.edu/entries/frame-problem/`

- gli obblighi dell'implementatore di $C.f$, che garantisce di portare l'oggetto in uno stato che soddisfa Q, qualora si parta da uno stato che soddisfa P;
- gli obblighi dell'utilizzatore di $C.f$, che si impegna a chiedere il servizio soltanto in un stato che soddisfa P.

Come in tutti i contratti, agli obblighi corrispondono anche dei benefici per le "parti" coinvolte:

- l'implementatore di $C.f$ si avvantaggia del fatto che può implementare considerando solo gli stati di partenza compatibili con P (per esempio se P fosse `False`... non dovrebbe implementare nulla perché non esistono situazioni in cui la *feature* può essere attivata);
- l'utilizzatore di $C.f$ sa che al termine dell'esecuzione di $C.f$ l'oggetto, oltre alle condizioni invarianti, soddisferà Q ed è quindi in grado di predire lo stato dell'oggetto che sta usando.

2.1 Un esempio: `CLOCK`

Si supponga di volere nel proprio sistema un componente che abbia la responsabilità di tener traccia delle ore del giorno. Usando i servizi della classe `INTEGER` (con cui la nostra collabora al raggiungimento delle sue responsabilità) potremmo scrivere:

```
class
  CLOCK

feature
```

```
  hours, minutes: INTEGER

invariant
  valid_hours: 0 <= hours and hours < 24
  valid_mins: 0 <= minutes and minutes < 60

end
```

Questa classe non è ancora molto utile: qualsiasi oggetto di tipo `CLOCK` avrebbe tutti i valori a `0`, visto che le *feature* `hours` e `minutes` sarebbero accessibili solo *in lettura* (sono due *query*, cioè *feature* che permettono di interrogare un oggetto per avere informazioni sul suo stato senza alterarlo) e il creatore di *default* assegna il valore `0`. Già così, però, chiarisce abbastanza bene a un possibile *client* quali sono le sue responsabilità (quindi sostanzialmente perché può essere utile al *client*) e perfino la sua strategia di massima: si tratta di un orologio basato su 24 ore e non, per esempio, su 12. Siamo certamente facilitati dalla familiarità con gli orologi della vita quotidiana e l'uso di nomi suggestivi per le *feature* è fondamentale, però gli invarianti danno un'idea delle proprietà di tutti gli oggetti di questa classe e questa informazione è efficace per capire come comporre un sistema più ampio.

2.1.1 La creazione degli oggetti

Aggiungiamo un creatore di oggetti `CLOCK`, `make`:

```
class
  CLOCK

create
```

```
  make

feature

  hours, minutes: INTEGER

  make (hh, mm: INTEGER)
    do
      hours := hh
      minutes := mm
    ensure
      hours = hh
      minutes = mm
    end

invariant
  valid_hours: 0 <= hours and hours < 24
  valid_mins: 0 <= minutes and minutes < 60

end
```

La postcondizione chiarisce molto bene l'obiettivo della *feature*. Un client però sarebbe del tutto legittimato a creare un oggetto `c: CLOCK` con `create c.make(42, -42, -42)`, ma una chiamata simile creerebbe un oggetto in cui gli invarianti non valgono! L'implementatore dovrebbe o tenerne conto nel codice che scrive (soddisfare gli invarianti è suo dovere), o prescrivere (e forse *negoziare*…) delle precondizioni appropriate che limitino la libertà del *client*. Per esempio:

```
make (hh, mm: INTEGER)
  require
    valid_hh: 0 <= hh and hh < 24
    valid_mm: 0 <= mm and mm < 60
```

```
  do
    hours := hh
    minutes := mm
  ensure
    hours = hh
    minutes = mm
  end
```

Si noti che le precondizioni non duplicano gli invarianti, infatti esse predicano su oggetti diversi (gli invarianti sulle *feature* `hours` e `minutes`; le precondizioni sui parametri della *feature* `make`). Inoltre gli invarianti non hanno senso per i creatori (l'oggetto non c'è ancora: il suo primo stato "stabile" è al termine della sua creazione).

2.1.2 *Feature* di assegnamento

Aggiungiamo anche dei *setter*. Anche qui la semantica è resa trasparente da pre- e postcondizioni, e il codice è reso elementare dalle restrizioni delle precondizioni.

```
feature

  set_hours (h_value: INTEGER)
    require
      valid_h_value: 0 <= h_value and h_value < 24
    do
      hours := h_value
    ensure
      hours = h_value
    end

  set_minutes (m_value: INTEGER)
    require
```

```
    valid_m_value: 0 <= m_value and m_value < 60
  do
    minutes := m_value
  ensure
    minutes = m_value
  end
```

Volendo si può anche rendere la sintassi più scorrevole dichiarando i *setter* come le *feature* utilizzabili per assegnare un valore, in questo caso l'operatore di assegnamento `:=` diventa un sinonimo di una chiamata al comando di assegnamento corrispondente, con i vincoli stabiliti dal suo contratto.

```
hours: INTEGER assign set_hours
-- un client di c: CLOCK
-- può scrivere c.hours := 3
-- equivalente a c.set_hours(3)
```

2.1.3 *Feature* che cambiano lo stato degli oggetti

Aggiungiamo anche due **comandi** per cambiare lo stato dell'oggetto. In questo caso non servono precondizioni: i due comandi si possono attivare in qualsiasi stato dell'oggetto e non hanno parametri da vincolare. Le postcondizioni però devono essere un po' più elaborate se vogliamo essere precisi. Potremmo anche limitarci a un condizione necessaria alla correttezza (p.es. per `increase_hours` potremmo dire semplicemente che il risultato della *query* `hours` dopo l'esecuzione del comando è diverso da quello che si otteneva prima di eseguire il comando: `hours /= old hours`), ma in questo caso non è troppo difficile essere precisi e la semantica diventa chiara: si noti anche come diventano evidenti le interrelazioni fra le due *feature*. L'operatore `implies` è un operatore

logico che rende più leggibile l'espressione: `(a implies b) = (not a or b)`.

```
feature -- Basic operations

  increase_hours
    do
      if hours = 23 then
        set_hours (0)
      else
        set_hours (hours + 1)
      end
    ensure
      normal_h: old hours < 23 implies hours = old
      ↪  hours + 1
      change_day: old hours = 23 implies hours = 0
    end

  increase_minutes
    do
      if minutes = 59 then
        set_minutes (0)
        increase_hours
      else
        set_minutes (minutes + 1)
      end
    ensure
      normal_m: old minutes < 59 implies minutes =
      ↪  old minutes + 1
      change_hour: old minutes = 59 implies minutes =
      ↪  0 and hours /= old hours
    end
```

2.1.4 Ridefinizione di una *feature* ereditata

Tutti le classi hanno una *feature* `out: STRING` ereditata da `ANY` (il vertice della gerarchia dei tipi, come `Object` in Java) che permette di ottenere la rappresentazione dell'oggetto come stringa (analogo al `toString` in Java). Per aggiungere una implementazione specifica di `CLOCK` dobbiamo ridefinirla (*overriding*). Il valore di ritorno di una *feature* in Eiffel è il valore della variabile `Result`.

```
class
  CLOCK

inherit

  ANY
    redefine
      out
    end

create
  make

feature

  hours, minutes: INTEGER

  make (hh, mm: INTEGER)
    require
      valid_hh: 0 <= hh and hh < 24
      valid_mm: 0 <= mm and mm < 60
    do
      hours := hh
      minutes := mm
```

```
    ensure
      hours = hh
      minutes = mm
    end

feature

  set_hours (h_value: INTEGER)
    require
      valid_h_value: 0 <= h_value and h_value < 24
    do
      hours := h_value
    ensure
      hours = h_value
    end

  set_minutes (m_value: INTEGER)
    require
      valid_m_value: 0 <= m_value and m_value < 60
    do
      minutes := m_value
    ensure
      minutes = m_value
    end



  out: STRING
    local
      hh, mm: STRING
    do
      -- twin crea una copia identica
      hh := hours.out.twin
      if hh.count < 2 then
```

```
        hh.prepend ("0")
      end
      mm := minutes.out.twin
      if mm.count < 2 then
        mm.prepend ("0")
      end
      Result := hh + ":" + mm
    end

invariant
  valid_hours: 0 <= hours and hours < 24
  valid_mins: 0 <= minutes and minutes < 60

end
```

Se volessimo specificare una precondizione per `CLOCK.out` però dovremmo usare cautele particolari. In effetti un *client* potrebbe usare un oggetto `CLOCK` tramite un riferimento di tipo `ANY` sfruttando il polimorfismo e il *binding* dinamico che caratterizzano la programmazione orientata agli oggetti. Il contratto di `ANY.out` (in `EiffelStudio` è facile ottenere il testo di qualsiasi classe, è sufficiente scriverne il nome nella casella di ricerca; inoltre scegliendo la visualizzazione "*contract*" non si è distratti dai dettagli implementativi) è il seguente:

```
out: STRING_8
    -- New string containing terse printable
    ↪  representation
    -- of current object
  ensure
    out_not_void: Result /= Void
```

Se in `CLOCK` ridefinissimo la *feature* `out` in questo modo:

```
out: STRING
  do
    Result := Void -- superfluo: Void è il valore di
    ↪  default
  ensure
    out_not_void: Result /= Void
  end
```

Il codice compilerebbe senza problemi, ma a *run-time* potremmo avere una situazione anomala. Infatti:

```
my_feature
  local
    aa: LIST [ANY]
    i: INTEGER
  do
    -- crea e popola aa
    from
      i := 0
    until
      i = aa.count
    loop
      if aa [i].out.starts_with ('x') then
        print ("hello: " + aa [i].out + "%N")
      end
    end
  end
```

Il codice di `my_feature` fallirebbe a causa di una chiamata di *feature* su un riferimento `Void` (il `null` di Java) se uno degli oggetti contenuti in `aa` fosse (dinamicamente) di tipo `CLOCK` e ciò nonostante il suo implementatore non citi esplicitamente alcuna *feature* specifica della classe `CLOCK` (anche nel popolamento della lista i riferimenti potrebbero essere restituiti da

feature che ottengono oggetti `CLOCK` da altre *feature*, ecc.; la *dipendenza* da `CLOCK` potrebbe essere molto ben poco evidente all'implementatore di `my_feature`).

3 Principio di Liskov

> *Socrate è uomo. Ogni uomo è animale. Dunque Socrate è animale.*
>
> *— Sesto Empirico, Schizzi Pirroniani II*

Il principio (cosiddetto, un po' impropriamente, di sostituibilità) di Liskov (che Barbara Liskov ha formulato nel 1987 in un keynote a OOPSLA e più esplicitamente espresso in un articolo del 1994 scritto insieme a Jeannette Wing: "A behavioral notion of subtyping"[7]; è anche la 'L' del popolare approccio `SOLID` per la progettazione di sistemi ad oggetti) dà le condizioni necessarie affinché un sottotipo possa essere utilizzato *come* il tipo base: con un riferimento di tipo base non si avranno perciò mai errori come quello descritto alla fine del paragrafo precedente, neanche se il riferimento punta a un oggetto del sottotipo.

> *Sia $\phi(x)$ una proprietà dimostrabile di un oggetto x di tipo T. Il tipo S sottotipo di T ha lo stesso comportamento degli oggetti di tipo T solo se $\phi(y)$ è vera per tutti gli oggetti y di tipo S.*

Con riferimento al *contratto* di un tipo, le proprietà che ci interessano sono:

[7] `https://doi.org/10.1145/197320.197383`

1. gli invarianti, che quindi dovranno valere anche per il sottotipo;

2. le "triple di Hoare" per ciascuna *feature* $\{P\}T.f\{Q\}$: anche queste devono valere per il sottotipo, trattandosi però di implicazioni che devono valere singolarmente per ogni *feature*, abbiamo un po' più di libertà, potendo indebolire l'antecedente e rafforzare il conseguente senza cambiare il valore di verità della singola implicazione.

Quindi se $\{P\}T.f\{Q\}$ è la parte di contratto del tipo base T per la *feature* $T.f$, per la *feature* $S.f$ con S sottotipo di T, dovrà valere $\{P'\}S.f\{Q'\}$ dove:

$$P \implies P' \tag{1}$$
$$Q' \implies Q \tag{2}$$

3.0.1 Principio di Liskov nei contratti Eiffel

Dimostrare le suddette implicazioni è, in generale, indecidibile automaticamente (anche "a mente" può essere piuttosto complicato) e, anche restringendosi ad ambiti decidibili come la logica del prim'ordine, può comunque essere piuttosto laborioso: in pratica quindi non è un compito che possiamo affidare con leggerezza a un compilatore. Ecco quindi che Eiffel affronta il problema obbligando il programmatore a scrivere pre- e postcondizioni derivate che verifichino **per costruzione** le implicazioni cui siamo interessati.

1. Le precondizioni devono o rimanere le stesse del tipo base, o altrimenti possono essere solo indebolite: le ul-

teriori condizioni P^* sono *disgiunte* (cioè in `or`) con quelle del tipo base e devono essere specificate tramite la clausola `require else`.

2. Le postcondizioni devono o rimanere le stesse del tipo base, o altrimenti possono essere solo rafforzate: le ulteriori condizioni Q^* sono *congiunte* (cioè in `and`) con quelle del tipo base e devono essere specificate tramite la clausola `ensure then`.

In questo modo si ha:

$$P \implies P \vee P^* \tag{3}$$
$$Q \wedge Q^* \implies Q \tag{4}$$

e le implicazioni a questo punto sono vere per qualsiasi P^* e Q^* (almeno nel dominio dell'inferenza logica, va sempre ricordato però che in Eiffel il valore di verità di un predicato è in realtà il frutto di una computazione arbitraria).

3.0.2 Ancora `CLOCK.out`

Ciò permette di risolvere il problema lasciato aperto alla fine del paragrafo precedente.

```
out: STRING_8
    -- New string containing terse printable
    ↪  representation
    -- of current object
  ensure
    out_not_void: Result /= Void
```

Dato che la precondizione è `True`, qualsiasi condizione aggiuntiva non sarà efficace per restringere il dominio di correttezza di una versione ridefinita nella *feature*.

Potremmo invece aggiungere nuove promesse a quella di non ritornare `Void`: chi usa un riferimento ad `ANY` continua comunque a veder garantita la postcondizione della classe base.

```
out: STRING
  local
    hh, mm: STRING
  do
    -- twin crea una copia identica
    hh := hours.out.twin
    if hh.count < 2 then
      hh.prepend ("0")
    end
    mm := minutes.out.twin
    if mm.count < 2 then
      mm.prepend ("0")
    end
    Result := hh + ":" + mm
  ensure then
    Result.count = 5
  end
```

Con questa aggiunta risulta chiaro che un *client* di `CLOCK` `.out` deve aspettarsi una stringa di (esattamente) 5 caratteri (non c'è dubbio dunque che per ore e minuti si usano sempre 2 caratteri). Volendo si può essere ancora più espliciti:

```
ensure then
   across Result.split (':') as token all
   ↪  token.item.count = 2 end
```

Il costrutto `across` però non è nello standard, ma è particolarmente comodo per espressioni come queste. È documentato qui[8].

Volendo evitarlo occorre fare una `feature` apposita (una *query* che ritorna un `BOOLEAN`).

```
check_tokens (s: STRING): BOOLEAN
  local
    splitted: LIST [STRING]
  do
    splitted := s.split (':')
    from
      splitted.start
      Result := True
    until
      splitted.after or Result = False
    loop
      Result := Result and splitted.item.count = 2
      splitted.forth
    end
  end
```

A questo punto è possibile scrivere la postcondizione usando questa *feature*.

```
ensure then
   check_tokens(Result)
```

3.1 Polimorfismo e parametri delle *feature*

Quando una *feature* accetta dei parametri il tipo del parametro formale è chiaramente una implicita precondizione: in-

[8] https://bertrandmeyer.com/2010/01/26/more-expressive-loops-for-eiffel/

fatti limita il dominio di ciò che l'implementatore deve trattare. Il tipo del valore di ritorno costituisce invece una promessa, esattamente come le postcondizioni. Inevitabilmente, quindi, per evitare errori di tipo occorre aderire al principio di Liskov anche nel trattamento dei parametri. C'è un modo facile di ottenerlo, adottato dalla maggior parte dei linguaggi orientati agli oggetto e staticamente tipizzati (come Java e C++): le ridefinizioni non possono cambiare il tipo dei parametri, sia di quelli in "ingresso", che di quelli in "uscita" come il valore di ritorno: nel gergo dei linguaggi di programmazione questa scelta è detta "novarianza" (o "invarianza", termine però che qui è meglio evitare).

La novarianza a volte può però essere troppo vincolante per progettisti (ricordiamoci che Eiffel ha l'ambizione di essere il linguaggio **unico** per analisi, progetto e sviluppo e di un sistema, come recita il titolo stesso dello standard ECMA-367) che, come vedremo nell'esempio che segue, troverebbero in qualche caso più "naturale" restringere il dominio dei parametri di ingresso. Per questo motivo Eiffel fa una scelta piuttosto controcorrente: la ridefinizione può essere **covariante** cioè ammettere parametri (in ingresso) di sottotipi, infrangendo così il principio di Liskov. Si noti che una varianza in senso opposto (**controvarianza**) in cui i parametri posso essere sopra-tipi sarebbe invece del tutto accettabile dal punto di vista del principio di Liskov: si tratta in effetti di una possibilità esplorata in alcuni linguaggi (p.es. Sather) ma risultata poco utile e tutto sommato meno naturale anche della novarianza.

3.2 Esempio: ANIMAL

Si supponga di avere una classe che modella gli animali che non mangiano sé stessi, nonostante gli animali siano in generale un cibo edibile. Si tratta di una classe che non ha (ancora) un'implementazione (`deferred` in Eiffel) e serve per scrivere algoritmi che si riferiscono a tutti gli animali che mangiano per aumentare le proprie riserve energetiche. Naturalmente poi serviranno concrete implementazioni che definiscono i dettagli.

```
deferred class
  ANIMAL

inherit

  FOOD

feature

  energy: INTEGER

  eat (f: FOOD)
    require
      no_autophagy: f /= Current
    deferred
    ensure
      energy >= old energy

    end

end
```

Nella precondizione si specifica che il parametro `f` non può riferirsi all'oggetto su cui la *feature* viene attivata: `Cu`

rrent è l'equivalente di this in Java. Anche FOOD è una dichiarazione priva di implementazione (deferred è analogo a abstract in Java).

```
deferred class
  FOOD

end
```

Volendo ora definire una classe COW effettivamente utilizzabile e di tipo ANIMAL dobbiamo obbligatoriamente definire un'implementazione della *feature* `eat (f: FOOD)`. In Java saremmo obbligati a mantenere la medesima *signature* con il parametro dello stesso tipo. Questo però vorrebbe anche dire che un utilizzatore di `eat` avrebbe tutto il diritto di aspettarsi un incremento delle riserve di energia anche nel caso in cui il tipo (dinamico) del parametro attuale fosse un cibo incompatibile con il metabolismo delle mucche. In altre parole, l'implementatore deve trattare il caso in cui `f` è di tipo generico FOOD, senza poter assumere nulla di più (in Java probabilmente finirebbe per utilizzare la possibilità di conoscere il tipo con instanceof). In Eiffel è possibile **restringere** il tipo del parametro, potendo avere così, per esempio, in modo molto naturale, `eat (f: GRASS)`.

```
class
  GRASS

inherit

  FOOD
    redefine
      out
```

```
    end

feature

  out: STRING
    do
      Result := "a bunch of grass (" + weight.out +
      ↪  "kg)"
    end

  consume (q: INTEGER)
    require
      q > 0
      weight >= q
    do
      weight := weight - q
    ensure
      weight = old weight - q
    end

  grow (q: INTEGER)
    require
      q > 0
    do
      weight := weight + q
    ensure
      weight = old weight + q
    end

  weight: INTEGER

invariant
  weight >= 0
```

```
end
```

Quindi, per esempio:

```
eat (f: GRASS)
  do
    f.consume (10)
    energy := energy + 1
  end
```

3.2.1 *Catcall*

La *feature* precedente può causare errori a `run-time`, come previsto dal principio di Liskov. Infatti chi ha un riferimento a un `ANIMAL` potrà fare una chiamata passando un oggetto di tipo `FOOD` che, non essendo necessariamente un `GRAS` `S`, potrebbe non avere le *feature* necessarie (`consume`). Eiffel aggiunge perciò dei controlli specifici e, nel caso, solleva l'eccezione *Changed Availability or Type* (***catcall***, ECMA-367 8.25.1).

```
my_feature
local
  a: ANIMAL
  c1, c2: COW
  do
    create c1

    a := c1
    a.eat(c2) -- errore a run-time (catcall)
    -- la catcall precede il controllo della
    ↪  precondizione
    a.eat(c1) -- errore a run-time (catcall)
  end
```

3.2.2 Pre- e postcondizioni

Volendo possiamo specificare meglio le postcondizioni di `COW.eat` (con `ensure then` naturalmente).

```
eat (f: GRASS)
  do
    f.consume (10)
    energy := energy + 1
  ensure then
    f.weight <= old f.weight
  end
```

Sarebbe molto sensato aggiungere anche una precondizione, ma il `require else` la renderebbe di fatto inutile (è in `or` con quella di `ANIMAL`, quindi è sufficiente che `f` non si riferisca alla mucca che mangia. Potrebbe però servire comunque per documentare (ma non controllare a *run-time*) il problema causato da una quantità di erba non sufficiente.

```
eat (f: GRASS)
  require else
    f.weight >= 10
  do
    -- la precondizione non impedisce che la chiamata
    -- avvenga in uno stato inopportuno
    f.consume (10)
    energy := energy + 1
  ensure then
    f.weight <= old f.weight
  end
```

Questa precondizione non verrebbe mai sollecitata e l'errore verrebbe rilevato solo al momento della chiamata di `consume`.

```
my_feature
local
  a: ANIMAL
  c1, c2: COW
  f: FOOD
  g: GRASS
  do
    create c1
    create c2
    create g

    a := c1
    f := g
    -- c1.eat(f)  -- errore a compile-time
    -- c1.eat(c2) -- errore a compile-time
    a.eat(g)  -- precondizione di g.consume non
    ↪  rispettata
    c1.eat(g) -- precondizione di g.consume non
    ↪  rispettata
    a.eat(f)  -- precondizione di g.consume non
    ↪  rispettata
  end
```

4 Violazioni dei patti contrattuali

> *Seduli in accurrendo, alacres in succurrendo*
>
> — *Motto del Corpo Italiano di Soccorso dell'Ordine di Malta*

Anche in un sistema Eiffel, naturalmente, ci possono essere dei problemi, dei *bug* come si dice spesso. In realtà è utile distinguere fra il **difetto** o **guasto** (*fault*) che ne è la **causa**

e il malfunzionamento (*failure*) che ne è l'effetto. Si noti che in generale è del tutto possibile che un difetto non si traduca in un malfunzionamento visibile: in tutti i casi però il sistema sarà in uno stato errato, anche se non riconoscibile come tale dall'utente. Il controllo dei contratti durante l'esecuzione del sistema garantisce invece che gli stati errati incompatibili con i contratti vengano segnalati e ciò facilita l'identificazione della causa. Nei casi in cui i controlli abbiano un effetto inaccettabile sulle prestazioni del sistema, possono essere disattivati in "produzione": dopotutto anche le prestazioni possono essere parte integrante di requisiti e specifiche e non soltanto nei cosiddetti sistemi *real-time*. In moltissimi casi, però, è conveniente lasciarli per aiutare la diagnosi in caso di malfunzionamento.

4.0.1 Assenza di violazioni

L'assenza di violazioni di contratto non è in generale una validazione del sistema:

- il sistema potrebbe non fare la "cosa giusta", cioè soddisfare i requisiti del committente anche quando opera nel "modo giusto", cioè in accordo con le specifiche del sistema (infatti l'ingegneria del *software* parla sempre di **verifica** e **convalida**: la prima la si persegue confrontando il comportamento del sistema rispetto alle specifiche, la seconda chiedendo al committente se il sistema soddisfa le sue esigenze);
- i contratti potrebbero non costituire una specifica completa dei componenti e del sistema nel suo complesso: alcune proprietà potrebbero essere troppo laboriose da esprimere (in particolare quelle "non funzionali" o che

dipendono da entità più complesse dello stato puntuale del sistema);

- le condizioni di esercizio del sistema potrebbero non aver mai esplorato cammini di esecuzione che portano a stati errati, anche se nel codice sono effettivamente presenti.

4.0.2 La rilevazione di una violazione

Cosa succede quando il supporto *run-time* rileva un'eccezione?

- Se dipende da una discrepanza fra quanto pattuito nel contratto e quanto rilevato a *run-time*, si ha la **certezza** della presenza di un difetto: l'implementazione o il contratto sono difettosi.
- Se il contratto è corretto, si può attribuire una "colpa":
  - quando non sono state rispettate le precondizioni, la responsabilità è del *client*;
  - quando non sono state rispettate le postcondizioni o gli invarianti, la responsabilità è dell'implementatore.
- In alcuni casi il problema è nell'ambiente di esecuzione: per esempio, per l'esaurimento della memoria o altre situazioni anomale segnalate dal sistema operativo; in Eiffel il trattamento di queste situazioni è analogo a quello delle violazioni contrattuali.

4.0.3 Il trattamento delle violazioni

La rilevazione di una situazione eccezionale non dà necessariamente luogo a un fallimento con immediata terminazione del sistema. Gli implementatori possono tentare di "salvare il salvabile" con il costrutto `rescue`.

La clausola `rescue` è una forma un po' diversa dal classico `try/catch` presente in molti linguaggi, anche se può essere usato nello stesso modo. Le differenze fondamentali sono:

- c'è un'unica clausola `rescue` per ogni *feature*: quando non viene specificata, viene chiamata la *feature* `ANY.default_rescue` che non fa nulla (il sistema termina con la segnalazione della situazione eccezionale che ha causato la violazione), a meno che non venga ridefinita;
- è possibile usare la clausola `rescue` per operazioni di pulizia/riordino in modo che la terminazione avvenga in maniera controllata (eventualmente anche distinguendo le diverse situazioni eccezionali con i servizi della classe `EXCEPTION`), senza danni per l'ambiente di esecuzione: in questo caso si parla di **panico organizzato**, una strategia comune anche ad altri sistemi;
- è possibile usare la clausola `rescue` per "aggiustare" lo stato del sistema e riprovare l'esecuzione di tutta la *feature*, ripartendo dallo stato del sistema com'era prima dell'esecuzione (ma emendato dalle istruzioni della `rescue`) che ha causato la violazione.

L'ultima possibilità non è facile da ottenere in altri linguaggi, dato che presuppone la ricostruzione dello stato iniziale (compreso lo *stack* delle chiamate): in Eiffel questo sta-

to deve comunque essere conservato per poterlo citare nelle postcondizioni con la parola chiave `old`.

Si noti che, come abbiamo già notato, una violazione contrattuale è la manifestazione dell'esistenza di un difetto: nell'implementazione o nel contratto. Se è possibile, quindi, è opportuno correggere il difetto e usare invece la `rescue` per situazioni effettivamente (parzialmente) imprevedibili o fuori del controllo del programmatore come quelle dovute ad anomalie nell'ambiente di esecuzione.

4.1 Esempio: `CLOCK.make_with_current_time`

Potremmo aggiungere a `CLOCK` un creatore che crea un orologio inizializzato con l'orario corrente. Poiché nelle librerie fornite con `EiffelStudio` non c'è (o non ho trovato…) una *query* per ottenere l'ora, uso il *file system*, creando un *file* temporaneo di cui si rileva poi l'orario di ultima modifica.

```
make_with_current_time
  local
    f: RAW_FILE
    t: INTEGER
    do
      create
      ↪  f.make_create_read_write("CLOCK_TMP_XXXX")
      t := f.date.integer_remainder (24*60*60)
      f.close
      f.wipe_out
      set_minutes (t.integer_remainder
      ↪  (60*60).integer_quotient (60))
      t := t - t.integer_remainder (60*60)
      set_hours(t.integer_quotient (60*60))
```

```
      f.delete
    end
```

Potrebbe succedere però che le operazioni sul *file system* falliscano (per esempio per mancanza di permessi di scrittura). Per rendere il sistema robusto rispetto a questa situazione anomala, si potrebbe stabilire un orario di *default*, per esempio `"00:00"`.

```
make_with_current_time
      -- if unable to get the current time, set the
      ↪  clock to 00:00
  local
    f: RAW_FILE
    t: INTEGER
    do
      create
      ↪  f.make_create_read_write("CLOCK_TMP_XXXX")
      t := f.date.integer_remainder (24*60*60)
      f.close
      f.wipe_out
      set_minutes (t.integer_remainder
      ↪  (60*60).integer_quotient (60))
      t := t - t.integer_remainder (60*60)
      set_hours(t.integer_quotient (60*60))
      f.delete
    rescue
      make (0, 0)
    end
```

A questo punto però è opportuno documentare il comportamento con un commento o cambiando il nome della *feature* in `make_with_current_time_if_possible`.

4.2 Esempio: violazioni in `COW.eat`

Abbiamo già visto che la *feature* `COW.eat` può fallire nonostante la precondizione.

```
eat (f: GRASS)
  require else
    f.weight >= 10
  do
    -- la precondizione non impedisce che la chiamata
    -- avvenga in uno stato inopportuno
    f.consume (10)
    energy := energy + 1
  ensure then
    f.weight <= old f.weight
  end
```

Potrebbe venirci l'idea di usare una `rescue` per evitare che il problema si propaghi.

```
eat (f: GRASS)
  require else
    f.weight >= 10
  do
    f.consume (10)
    energy := energy + 1
  ensure then
    f.weight <= old f.weight
  rescue
    f.grow (10)
    retry
  end
```

Questo codice "risolve" il problema: infatti un eventuale fallimento nella precondizione di `f.consume` verrebbe recuperato dalla `rescue` che prima assicura che la precondizione

sia verificata e poi riprova l'esecuzione della *feature*. Questa però è una strategia da evitare: il meccanismo di trattamento è usato in maniera impropria. Risulta evidente ragionando sulle eventuali "colpe" della violazione contrattuale: il chiamante (la *feature* `eat`) è responsabile di una chiamata di `f.consume` senza aver garantito le precondizioni. La soluzione migliore è quindi che sia proprio il chiamante ad assicurarsi che la precondizione valga, chiamando poi `f.consume` solo quando si può farlo in sicurezza. Invece nel codice appena presentato il chiamante non fa alcun controllo (a cui è tenuto, perché già sappiamo che la precondizione non è efficace a causa del `require else`) ed eventualmente "ripara" con la `rescue`; si noti, fra l'altro, che se il controllo dei contratti fosse disabilitato (come può capitare in produzione) la strategia fallirebbe, probabilmente in maniera difficile da diagnosticare. Meglio quindi agire così:

```
eat (f: GRASS)
  require else
    f.weight >= 10
  do
    if f.weight < 10 then
      f.grow (10)
    end
    f.consume (10)
    energy := energy + 1
  ensure then
    f.weight <= old f.weight
  end
```

5 Il *kata* BOWLING

> *Smokey, this is not 'Nam. This is bowling. There are rules.*
>
> *— The Big Lebowski*

Nelle comunità di sviluppo "agile" sono diffusi i cosiddetti *kata*: come è uso nel *karate*, si tratta di esercizi pensati per perfezionare la tecnica personale, ripetendo molte volte "mosse" realistiche.

Uno dei *kata* più famosi, dovuto a Robert Martin (Uncle-Bob) è il Bowling Kata[9], progettato per fare pratica con la tecnica del *Test Driven Development* (TDD).

Il gioco del *bowling* divide la gara di ciascun giocatore in 10 *frame*: in ogni *frame* il giocatore ha due possibilità di abbattere i 10 birilli (*pin*). Il punteggio ottenuto nel *frame* è il numero di birilli abbattuti, maggiorato di un premio per gli *spare* e gli *strike*. Uno *spare* si verifica quando vengono abbattuti 10 birilli usando i due tentativi. In questo caso il premio è il numero di birilli abbattuto con il tiro (*roll*) seguente (effettuato nel prossimo *frame*). Uno *strike* si verifica quando vengono abbattuti 10 birilli al primo tentativo: in questo caso il secondo tiro del *frame* non viene effettuato. Il premio per lo *strike* è il numero di birilli abbattuto con i due tiri seguenti (effettuati nel prossimo *frame*).

Se uno *strike* o uno *spare* si verificano nel decimo *frame*, il giocatore ha diritto ai tiri necessari ad acquisire il premio relativo. Il decimo *frame* può quindi dare luogo a un massimo di 3 tiri. Il punteggio massimo ottenibile è 300 punti.

[9] `http://www.butunclebob.com/files/downloads/Bowling%20Game%20Kata.ppt`

Vogliamo realizzare un componente GAME in grado di tenere il conto del punteggio.

```
deferred class
  GAME

feature

  roll (pins: INTEGER)
    deferred
    end

  score: INTEGER
    deferred
    end

end
```

Possiamo già specificarne meglio il contratto.

```
deferred class
  GAME

feature

  roll (pins: INTEGER)
    require
      valid_pins: 0 <= pins and pins <= 10
    deferred
    end

  score: INTEGER
    deferred
    end
```

```
invariant
  valid_score: 0 <= score and score <= 300

end
```

Seguiamo un approccio TDD nell'implementazione, sfruttando il *framework* per i *test* di unità presente nelle librerie di EiffelStudio: la classe da cui ereditare è `EQA_TEST_SET` (utilizzabile anche con il *wizard* `AutoTest`[10]).

```
note
  description: "[
    Eiffel tests that can be executed by testing tool.
  ]"
  author: "EiffelStudio test wizard"
  date: "$Date$"
  revision: "$Revision$"
  testing: "type/manual"

class
  BOWLING_TEST_SET

inherit
  EQA_TEST_SET

feature -- Test routines

  test_gutter_game
    local
      g: GAME
      i: INTEGER
    do
```

[10] https://www.eiffel.org/doc/eiffelstudio/AutoTest

```
      create g
      from
        i := 0
      until
        i = 20
      loop
        g.roll (0)
        i := i + 1
      end
      assert ("Gutter game has non zero score " +
      ↪  g.score.out, g.score = 0)
    end

end
```

Naturalmente il *test* inizialmente non compila, perché GAME è deferred. Per farlo fallire (il primo passo della strategia TDD *red-green-refactoring*) implementiamo GAME.

```
class
  GAME

feature

  roll (pins: INTEGER)
    require
      valid_pins: 0 <= pins and pins <= 10
    do
    end

  score: INTEGER
    do
      Result := 300
    end
```

```
invariant
  valid_score: 0 <= score and score <= 300

end
```

Per rendere il *test* "verde" è sufficiente sistemare `score`.

```
score: INTEGER
  do
    Result := 0
  end
```

Possiamo aggiungere un nuovo *test* con 20 tiri che abbattono un solo birillo.

```
test_all_ones
  local
    g: GAME
    i: INTEGER
  do
    create g
    from
      i := 0
    until
      i = 20
    loop
      g.roll (1)
      i := i + 1
    end
    assert ("All-ones game has wrong score " +
    ↪  g.score.out, g.score = 20)
  end
```

Rosso! Per superare il *test* aggiungiamo un contatore dei birilli caduti.

```
class
  GAME

feature

  roll (pins: INTEGER)
    require
      valid_pins: 0 <= pins and pins <= 10
    do
      points := points + pins
    end

  score: INTEGER
    do
      Result := points
    end

feature {NONE}
  points: INTEGER

invariant
  valid_score: 0 <= score and score <= 300
  points = score

end
```

Notate l'aggiunta dell'invariante che lega `points` e `score`: documenta l'obiettivo che ho in mente (e controlla che sia effettivamente valido in ogni momento) anche se `points` non è una *feature* pubblica.

Ora che i *test* sono tutti di nuovo verdi, possiamo fare un po' di *refactoring* dei *test*, spostando la creazione di `g` nella *routine* di *setup* di tutti i *test* (nel *framework* si chiama `on_prepare`, come si può scoprire guardando il testo della classe

EQA_TEST_SET). Precursor serve per riferirsi alla *feature* precedente la ridefinizione (quella definita da EQA_TEST_SET): la chiamata garantisce che la postcondizione promessa da EQA_TEST_SET.on_prepare (is_prepared) sia attuata. Vale la pena fattorizzare anche il tiro ripetuto in una *feature* roll_many. Naturalmente è bene verificare che i *test* restino verdi dopo il *refactoring*.

```
class
  BOWLING_TEST_SET

inherit

  EQA_TEST_SET
    redefine
      on_prepare
    end

feature {NONE}

  g: GAME

  on_prepare
    do
      Precursor
      create g
    end

  roll_many (n, pins: INTEGER)
    require
      positive_n: n > 0
      valid_pins: 0 <= pins and pins <= 10
    local
      i: INTEGER
```

```
    do
      from
        i := 0
      until
        i = n
      loop
        g.roll (pins)
        i := i + 1
      end
    ensure
      g.score >= old g.score
    end

feature -- Test routines

  test_gutter_game
    do
      roll_many (20, 0)
      assert ("Gutter game has non zero score " +
      ↪  g.score.out, g.score = 0)
    end

  test_all_ones
    do
      roll_many (20, 1)
      assert ("All-ones game has wrong score " +
      ↪  g.score.out, g.score = 20)
    end

end
```

Ora aggiungiamo un *test* per lo *spare*.

```
test_one_spare
  do
```

```
    g.roll (5)
    g.roll (5) -- spare
    g.roll (3)
    roll_many (17, 0)
    assert ("Wrong spare bonus. Total score " +
    ↪  g.score.out, g.score = 16)
  end
```

Ragionando su come rendere verde questo *test* ci si convince che l'idea di avere un semplice accumulatore per il punteggio è insufficiente: bisogna tenere traccia della *storia* dei tiri. Commentiamo quindi il nuovo *test* (ancora rosso) e teniamo traccia dei tiri in un *array*: i *test* precedenti devono rimanere verdi.

```
class
  GAME

inherit

  ANY
    redefine
      default_create
    end

feature

  roll (pins: INTEGER)
    require
      valid_pins: 0 <= pins and pins <= 10
    do
      points := points + pins
      rolls [current_roll] := pins
      if rolls.valid_index (current_roll + 1) then
```

```
        current_roll := current_roll + 1
      end
    ensure
      score >= old score
    end

  score: INTEGER
    do
      Result := points
    end

feature {NONE}

  default_create
    do
      create rolls.make_filled (0, 0, 20)
    ensure then
      rolls.count = 21
    end

  rolls: ARRAY [INTEGER]

  current_roll: INTEGER

  points: INTEGER

invariant
  valid_score: 0 <= score and score <= 300
  valid_current: rolls.valid_index (current_roll)
  valid_rolls: across rolls as r all 0 <= r.item and
  ↪  r.item <= 10 end
  points = score

end
```

Nel creatore di *GAME*, necessario per allocare la memoria dell'array, la postcondizione rende evidente il numero di elementi (uguale al numero massimo di tiri in una partita di *bowling*). A questo punto però *points* è superfluo (e l'invariante già suggeriva il pleonasmo).

```
class
  GAME

inherit

  ANY
    redefine
      default_create
    end

feature

  roll (pins: INTEGER)
    require
      valid_pins: 0 <= pins and pins <= 10
    do
      rolls [current_roll] := pins
      if rolls.valid_index (current_roll + 1) then
        current_roll := current_roll + 1
      end
    ensure
      score >= old score
    end

  score: INTEGER
    local
      i: INTEGER
    do
```

```
      from
        i := 0
      until
        not rolls.valid_index (i)
      loop
        Result := Result + rolls [i]
        i := i + 1
      end
    end

feature {NONE}

  default_create
    do
      create rolls.make_filled (0, 0, 20)
    ensure then
      rolls.count = 21
    end

  rolls: ARRAY [INTEGER]

  current_roll: INTEGER

invariant
  valid_score: 0 <= score and score <= 300
  valid_current: rolls.valid_index (current_roll)
  valid_rolls: across rolls as r all 0 <= r.item and
  ↪  r.item <= 10 end

end
```

Visto che i due *test* `test_gutter_game` e `test_all_ones` sono rimasti verdi, possiamo riattivare `test_one_spare` (rosso) e cercare di superarlo. Facendolo, però, ci rendiamo con-

to che occorre considerare i tiri a gruppi di due (un *frame*), altrimenti è difficile capire se si tratta di uno *spare*. Meglio disattivare ancora `test_one_spare` e controllare che gli altri *test* continuino a funzionare anche considerando i tiri a due a due.

```
score: INTEGER
  local
    i, frame: INTEGER
  do
    from
      frame := 0
      i := 0
    until
      frame = 10
    loop
      Result := Result + rolls [i] + rolls [i + 1]
      i := i + 2
      frame := frame + 1
    end
  end
```

Riattivando `test_one_spare`, il *test* risulta ancora rosso, ma ora è facile trovare la soluzione.

```
score: INTEGER
  local
    i, frame: INTEGER
  do
    from
      i := 0
      frame := 0
    until
      frame = 10
```

```
    loop
      if rolls [i] + rolls [i + 1] = 10 then
        Result := Result + 10 + rolls [i + 2]
      else
        Result := Result + rolls [i] + rolls [i + 1]
      end
      frame := frame + 1
      i := i + 2
    end
  end
```

Ora che è tutto verde possiamo rendere più leggibile il codice.

```
  score: INTEGER
    local
      i, frame: INTEGER
    do
      from
        i := 0
        frame := 0
      until
        frame = 10
      loop
        if is_spare(i) then
          Result := Result + 10 + rolls [i + 2]
        else
          Result := Result + rolls [i] + rolls [i + 1]
        end
        frame := frame + 1
        i := i + 2
      end
    end
```

```
feature {NONE}

  is_spare(frame_index: INTEGER): BOOLEAN
  require
    valid_index: rolls.valid_index(frame_index)
    even_index: frame_index.integer_remainder (2) = 0
  do
    Result := rolls[frame_index] + rolls[frame_index
    ↪  + 1] = 10
  end
```

Anche il *test*.

```
feature {NONE}

  roll_spare
    do
      g.roll (5)
      g.roll (5)
    end

feature -- Test routines

  test_one_spare
    do
      roll_spare
      g.roll (3)
      roll_many (17, 0)
      assert ("Wrong spare bonus. Total score " +
      ↪  g.score.out, g.score = 16)
    end
```

Aggiungiamo un *test* per lo *strike*.

```
test_one_strike
  do
    g.roll (10)
    g.roll (3)
    g.roll (4)
    roll_many (16, 0)
    assert ("Wrong strike bonus. Total score " +
    ↪  g.score.out, g.score = 24)
  end
```

Dopo aver controllato che fallisce, implementiamo la soluzione.

```
score: INTEGER
  local
    i, frame: INTEGER
  do
    from
      i := 0
      frame := 0
    until
      frame = 10
    loop
      if rolls[i] = 10 then
        Result := Result + 10 + rolls [i + 1] + rolls
        ↪  [i + 2]
        i := i + 1
      elseif is_spare (i) then
        Result := Result + 10 + rolls [i + 2]
        i := i + 2
      else
        Result := Result + rolls [i] + rolls [i + 1]
        i := i + 2
      end
```

```
      frame := frame + 1
    end
  end
```

Mi rendo conto però che l'assunzione che il controllo dello *spare* fosse solo su *rolls* di posto pari (che fortunatamente avevo codificato in una precondizione) fallisce quando ci sono degli *strike*, perché in questo caso il *frame* ha solo un tiro. Ottimo: ho scoperto che il contratto (e l'assunzione nella mia testa) era sbagliato e rimuovo la precondizione `even_index` a `is_spare`. I *test* a questo punto sono verdi e procedo con un *refactoring*.

```
score: INTEGER
  local
    i, frame: INTEGER
  do
    from
      i := 0
      frame := 0
    until
      frame = 10
    loop
      if is_strike (i) then
        Result := Result + 10 + strike_bonus (i)
        i := i + 1
      elseif is_spare (i) then
        Result := Result + 10 + spare_bonus (i)
        i := i + 2
      else
        Result := Result + pins_in_frame (i)
        i := i + 2
      end
      frame := frame + 1
```

```
      end
    end

feature {NONE}

  pins_in_frame (frame_index: INTEGER): INTEGER
    require
      valid_index: rolls.valid_index (frame_index + 1)
    do
      Result := rolls [frame_index] + rolls
      ↪ [frame_index + 1]
    ensure
      0 <= Result and Result <= 20
    end

  strike_bonus (frame_index: INTEGER): INTEGER
    require
      valid_index: rolls.valid_index (frame_index + 2)
    do
      Result := pins_in_frame (frame_index + 1)
    ensure
      0 <= Result and Result <= 20
    end

  spare_bonus (frame_index: INTEGER): INTEGER
    require
      valid_index: rolls.valid_index (frame_index + 2)
    do
      Result := rolls [frame_index + 2]
    ensure
      0 <= Result and Result <= 20
    end

  is_strike (frame_index: INTEGER): BOOLEAN
```

```
    require
      valid_index: rolls.valid_index (frame_index)
    do
      Result := rolls [frame_index] = 10
    end

  is_spare (frame_index: INTEGER): BOOLEAN
    require
      valid_index: rolls.valid_index (frame_index)
    do
      Result := pins_in_frame (frame_index) = 10
    end
```

Un ulteriore *test* sulla partita perfetta viene superato senza necessità di modificare il codice (in questi casi è sempre meglio farlo fallire con un valore errato nell'asserzione del *test*, da cambiare poi in quello giusto, in modo da essere certi che il *test* verifichi davvero qualcosa).

```
test_perfect_game
  do
    roll_many (20, 10)
    assert ("Wrong score in perfect game " +
    ↪  g.score.out, g.score = 300)
  end
```

L'esempio dovrebbe aver chiarito che i *test* di unità e le asserzioni del *Design by Contract* non sono tecniche alternative, ma del tutto complementari.

6 Limitazioni e difficoltà

> *Ma sedendo e mirando, interminati*
> *spazi di là da quella, e sovrumani*

silenzi, e profondissima quïete
io nel pensier mi fingo, ove per poco
il cor non si spaura.

— Giacomo Leopardi, L'infinito

Alla fine di questo breve excursus nel *Design by Contract* con Eiffel, è opportuno soffermarsi su alcune limitazioni intrinseche e difficoltà dell'approccio.

6.1 Proprietà *stateful*

Le proprietà di tipo *stateful*, cioè quelle predicano sulla *storia* di un oggetto sono complicate da esprimere. Se per esempio volessimo specificare che un `MY_CONTAINER` opera secondo una politica *First-In-First-Out* (è una *coda*) dovremmo scrivere un invariante che, come minimo, crea un oggetto, fa due inserimenti e un'estrazione e valuta il risultato: sarebbe piuttosto inefficiente e comunque verificherebbe la proprietà solo partendo dal contenitore vuoto. Ci sono sostanzialmente due approcci a questo problema, entrambi non del tutto soddisfacenti:

1. descrivere la proprietà nelle note della classe (se la proprietà è "classica" come la politica FIFO basta un commento molto sintetico; è l'approccio seguito dalle librerie di Eiffel per quanto riguarda, per esempio, `QUEUE`) o conformare la classe a un tipo che sappiamo già avere quella proprietà (per esempio ereditando da `QUEUE`);

2. utilizzare un oggetto **modello** che ha la proprietà desiderata e promettere nelle postcondizioni che il comportamento è analogo a quello del modello (per esempio,

aggiungendo una *feature* privata `model: QUEUE` il cui contenuto viene mantenuto "sincronizzato" con quello di `MY_CONTAINER`: inserimenti e rimozione devono perciò avere gli stessi effetti che avrebbero su `model`).

Il secondo approccio è quello che dà le maggiori garanzie dal punto di vista della correttezza ma, oltre a essere oneroso, presuppone l'esistenza di un modello adeguato.

6.2 Esecuzione e valori di verità

I contratti possono essere espressi anche su *feature* o classi `deferred`: è molto utile per esplicitare proprietà generali dei componenti. Bisogna però prestare attenzione perché il codice che calcola il valore di verità delle proprietà potrebbe riservarci delle sorprese. Non bisogna mai dimenticare che le proprietà che esprimiamo **non** devono la loro verità a una deduzione in un sistema logico formale, ma sono, appunto, il risultato di un calcolo (che può esso stesso essere sbagliato, o perfino non terminare).

Per esempio:

```
extend (x: G)
        -- Add `x' at end of list.
  require
        space_available: not full
  deferred
  ensure
        one_more:
        count = old count + 1
  end

full: BOOLEAN
```

```
    -- Is representation full?
    -- (Default: no)
  do
      Result := False
  end
```

Supponiamo che una sottoclasse ridefinisca `full`

```
full: BOOLEAN
  -- Is representation full?
  -- (Answer: if and only if number of items is equal
  ↪  to capacity)
  do
      Result := (count = capacity)
  end
```

A questo punto la `extend` della sottoclasse (anche senza bisogno di ridefinizione) non rispetta più il principio di Liskov, nonostante dal punto di vista sintattico non ci siano problemi: la precondizione, nella forma, non è cambiata (rimane `not full`). Nella sostanza, però, le condizioni in cui si può chiamare `extend` su oggetti del sottotipo sono più stringenti di quelle necessarie per il tipo base.

6.3 Proprietà non funzionali

Abbiamo già accennato al fatto che è difficile esprimere proprietà *non-funzionali*. La distinzione fra proprietà *funzionali* (una relazione fra gli stati possibili del sistema) e non-funzionali è meno precisa di quanto sembri a prima vista: in molti casi una proprietà non-funzionale può essere trasformata in funzionale arricchendo appropriatamente lo stato del componente. Per predicare sui tempi di esecuzione, per esempio,

è possibile aggiungere una variabile che conserva lo scorrere del tempo. In generale è comunque molto difficile e oneroso esprimere proprietà riguardo a tempi di esecuzione, concorrenza, sicurezza, ecc..

7 Temi d'esame risolti

> *Gli esami di profitto debbono essere ordinati in modo da accertare la maturità intellettuale del candidato e la sua preparazione organica nella materia sulla quale verte l'esame, senza limitarsi alle nozioni impartite dal professore nel corso cui lo studente è stato iscritto.*
>
> — *Regio Decreto 4 giugno 1938, n. 1269, art. 39 (*in vigore *alla data della scrittura di questa dispensa)*

7.1 Scacchi

Il codice Eiffel dell'esercizio è una bozza di un programma per giocare a scacchi: al momento le sue funzionalità si riducono alla possibilità di stampare su `stdout` la disposizione della scacchiera, l'unico pezzo disponibile è il Re. Il codice Eiffel è scritto per un compilatore *Void safe* (*default* per la versione 19.05 di `EiffelStudio`): quindi, perché un riferimento possa essere `Void` occorre dichiararlo esplicitamente di tipo `detachable` (es: `p: detachable PIECE`); per una variabile dichiarata come `p: PIECE`, il test `attached p` è sempre `True` e `p /= Void`. Dopo aver esaminato il codice a disposizione (ricordate che per vedere l'*output* è necessario lanciare il programma da terminale, con il comando `estudio`), si risponda

alle seguenti domande (il codice può essere scritto qui o nei file Eiffel):

1. Aggiungere un'opportuna precondizione alla feature PIECE.make.
2. Aggiungere una precondizione alla feature KING.make che permetta la chiamata con un parametro attuale "SINGLE". Dire qual è l'effetto delle istruzioni seguenti, discutendo il risultato.

   ```
   k: KING
     create k.make("SINGLE")
   ```

3. Aggiungere opportune precondizioni e postcondizioni alle *feature* BOARD.get, BOARD.put, BOARD.remove.
4. Migliorare le precondizioni della feature BOARD.move, specificando che la mossa deve essere valida per il pezzo da muovere.
5. Aggiungere un invariante a BOARD che specifichi che non ci possono essere più di due Re. Se l'invariante specificasse che ci devono essere sempre **esattamente** due Re, che problema occorrerebbe risolvere?

7.1.1 PIECE

```
deferred class
  PIECE

feature

  is_valid_move (board: BOARD; from_code, to_code:
  ↪  STRING): BOOLEAN
```

```
    require
      board.is_valid_code (from_code) and
      ↪ board.is_valid_code (to_code)
    deferred
    end

    -- English name of the piece

  name: STRING

  color: STRING

  make (player: STRING)
    do
      color := player.as_upper
    end

invariant
  color.is_equal ("WHITE") or color.is_equal ("BLACK")

end
```

7.1.2 KING

```
class
  KING

inherit

  PIECE
    redefine
      make,
      out
    end
```

```
create
  make

feature {NONE}

  repr: STRING

feature

  make (player: STRING)
    do
      name := "KING"
      repr := "UNICODE"
      if player.as_upper.is_equal ("SINGLE") then
        repr := "ASCII"
        color := "WHITE"
      else
        color := player.as_upper
      end
    end

  is_valid_move (board: BOARD; from_code, to_code:
  ↪  STRING): BOOLEAN
    local
      delta_x, delta_y: INTEGER
    do
      delta_x := to_code.at (2).difference
      ↪  (from_code.at (2)).abs
      delta_y := to_code.at (1).difference
      ↪  (from_code.at (1)).abs
      Result := board.is_empty (to_code) and then
      ↪  ((delta_x = 1) or (delta_y = 1))
    end
```

```
  out: STRING
    local
      u: UTF_CONVERTER
    do
      Result := u.string_32_to_utf_8_string_8
       ↪  ({STRING_32} "♔")
      if color.is_equal ("BLACK") then
        Result := u.string_32_to_utf_8_string_8
         ↪  ({STRING_32} "♚")
      end
      if repr.is_equal ("ASCII") then
        Result := "K"
      end
    end

end
```

7.1.3 SQUARE

```
class
  SQUARE

inherit

  ANY
    redefine
      out
    end

feature {NONE} -- Access

  presence: detachable PIECE assign set_presence
      -- `presence'
```

```
feature -- Element change

  set_presence (a_presence: like presence)
    do
      presence := a_presence
    ensure
      presence_assigned: presence = a_presence
    end

  get_presence: like presence
    do
      Result := presence
    end

feature

  is_occupied: BOOLEAN
    do
      Result := presence /= Void
    end

  out: STRING
    do
      Result := "_"
      if attached presence as p then
        Result := p.out
      end
    end

end
```

7.1.4 BOARD

```
class
  BOARD

inherit

  ANY
    redefine
      out
    end

create
  make

feature {NONE} -- Initialization

  matrix: ARRAY2 [SQUARE]

  make
      -- Initialization for `Current'.
    local
      empty: SQUARE
    do
      create empty
      create matrix.make_filled (empty, 8, 8)
      across
        matrix.lower |..| matrix.height as r
      loop
        across
          matrix.lower |..| matrix.width as c
        loop
          matrix [r.item, c.item] := empty.twin
        end
```

```
      end
    end

  to_row_col (code: STRING): TUPLE [r: INTEGER; c:
  ↪ INTEGER]
    require
      is_valid_code (code)
    local
      r, c: INTEGER
    do
      r := code.at (2).difference ('1') + 1
      c := code.at (1).difference ('a') + 1
      Result := [r, c]
    end

  is_available (pos: TUPLE [r, c: INTEGER]): BOOLEAN
    do
      Result := not matrix [pos.r, pos.c].is_occupied
    end

feature

  is_empty (code: STRING): BOOLEAN
    local
      pos: TUPLE [r: INTEGER; c: INTEGER]
    do
      pos := to_row_col (code)
      Result := is_available (pos)
    end

  is_valid_code (code: STRING): BOOLEAN
    local
      r, c: INTEGER
    do
```

```
    if not (code.count = 2 and code.at (1).is_alpha
    ↪  and code.at (2).is_digit) then
      Result := False
    else
      r := code.at (2).difference ('1') + 1
      c := code.at (1).difference ('a') + 1
      Result := r >= matrix.lower and r <=
      ↪  matrix.height and c >= matrix.lower and c
      ↪  <= matrix.width
    end
  end

put (where: STRING; p: PIECE)
  local
    pos: TUPLE [r: INTEGER; c: INTEGER]
  do
    pos := to_row_col (where)
    matrix [pos.r, pos.c].set_presence (p)
  end

get (where: STRING): detachable PIECE
  local
    pos: TUPLE [r: INTEGER; c: INTEGER]
  do
    pos := to_row_col (where)
    Result := matrix [pos.r, pos.c].get_presence
  end

remove (where: STRING)
  local
    pos: TUPLE [r: INTEGER; c: INTEGER]
  do
    pos := to_row_col (where)
    matrix [pos.r, pos.c].set_presence (Void)
```

```
  end

move (fromw, tow: STRING)
  require
    ok_from: not is_empty (fromw)
    ok_to: is_empty (tow)
  do
    if attached get (fromw) as p then
      remove (fromw)
      put (tow, p)
    end
  ensure
    ok_from: is_empty (fromw)
    ok_to: not is_empty (tow)
  end

out: STRING
  local
    cname: CHARACTER
    row: INTEGER
  do
    Result := ""
    across
      matrix.lower |..| matrix.height as r
    loop
      row := (matrix.height + matrix.lower) - r.item
      Result := Result + row.out + " "
      across
        matrix.lower |..| matrix.width as c
      loop
        Result := Result + matrix [row, c.item].out
        ↪  + " "
      end
      Result := Result + "%N"
```

```
      end
      Result := Result + " "
      across
        matrix.lower |..| matrix.width as c
      loop
        cname := 'a'
        cname := cname.plus (c.item - matrix.lower)
        Result := Result + " "
        Result.append_character (cname)
      end
      Result := Result + "%N"
    end

end
```

7.1.5 *Client* d'esempio

Per facilitare la sperimentazione vengono fornite anche:

```
note
  description: "chess application root class"

class
  APPLICATION

inherit

  ARGUMENTS

create
  make

feature {NONE} -- Initialization
```

```
  make
    local
      b: BOARD
      wk, bk: KING
    do
      create b.make
      create wk.make ("WHITE")
      create bk.make ("BLACK")
      print ("%N**********************************
      ↪  **********%N")
      b.put ("a5", wk)
      b.put ("h3", bk)
      print (b)
      b.move ("a5", "b6")
      print (b)
    end

end

note
  description: "[
    Eiffel tests that can be executed by testing tool.
  ]"
  author: "EiffelStudio test wizard"
  testing: "type/manual"

class
  CHESS_TEST_SET

inherit

  EQA_TEST_SET
    redefine
      on_prepare
```

```
    end

feature {NONE} -- Events

  b: BOARD

  wk, bk: KING

  on_prepare
    do
      create b.make
      create wk.make ("WHITE")
      create bk.make ("BLACK")
    end

feature -- Test routines

  test_valid_move
    do
      b.put ("b5", wk)
      assert ("b6", wk.is_valid_move (b, "b5", "b6"))
      assert ("c6", wk.is_valid_move (b, "b5", "c6"))
      assert ("c5", wk.is_valid_move (b, "b5", "c5"))
      assert ("c4", wk.is_valid_move (b, "b5", "c4"))
      assert ("b4", wk.is_valid_move (b, "b5", "b4"))
      assert ("a4", wk.is_valid_move (b, "b5", "a4"))
      assert ("a5", wk.is_valid_move (b, "b5", "a5"))
      assert ("a6", wk.is_valid_move (b, "b5", "a6"))
      assert ("b5", not wk.is_valid_move (b, "b5",
      ↪  "b5"))
      assert ("b7", not wk.is_valid_move (b, "b5",
      ↪  "b7"))
      assert ("b5 from b6", not wk.is_valid_move (b,
      ↪  "b6", "b5"))
```

```
      b.put ("b6", bk)
      assert ("b6", not wk.is_valid_move (b, "b5",
      ↪  "b6"))
    end

  test_board
    do
      assert ("Initially empty", b.is_empty ("b5"))
      b.put ("b5", wk)
      assert ("Not empty", not b.is_empty ("b5"))
    end

end
```

7.1.6 Soluzione

1. Precondizioni di `PIECE.make`

   Il nome della *feature* suggerisce si tratti di una *routine* di *creazione*. L'oggetto risultante dovrà soddisfare l'invariante della classe (`color.is_equal ("WHITE") or color.is_equal ("BLACK")`). La precondizione che garantisce questo invariante è:

   ```
   make (player: STRING)
     require
       player.as_upper.is_equal ("WHITE") or
       ↪  player.as_upper.is_equal ("BLACK")
     do
       color := player.as_upper
     end
   ```

2. Precondizioni di `KING.make`

Va innanzitutto notato che la classe `KING` è legata a `PIECE` dalla relazione `inherit`, pertanto le precondizioni di `KING.make` dovranno essere in `require else` con quelle di `PIECE.make`.

```
make (player: STRING)
  require else
    player.as_upper.is_equal ("SINGLE")
  do
    name := "KING"
    repr := "UNICODE"
    if player.as_upper.is_equal ("SINGLE") then
      repr := "ASCII"
      color := "WHITE"
    else
      color := player.as_upper
    end
  end
```

A questo punto un *client* di oggetti di tipo `KING` potrà così chiamare la *feature* `make` con un parametro stringa di valore attuale `"WHITE"`, `"BLACK"` o `"SINGLE"`. L'effetto di una chiamata come quella nel testo dell'esercizio è la creazione di un Re di colore `"WHITE"`, anche se verrà rappresentato sulla scacchiera dal carattere `"K"` anziché dal carattere Unicode per il Re bianco.

3. Pre- e postcondizioni di `BOARD.{get put remove}`

   Le tre *feature* sono molto simili: prendono tutte un parametro `where`, una stringa che, come risulta evidente dagli esempi forniti, codifica la posizione nella scacchiera. Per tutte, quindi, sarà opportuno avere una precondizione che limiti i valori di questo parametro a stringhe valide. In effetti c'è anche una *query* di `BOARD` utile

allo scopo, `is_valid_code`, che, come suggerisce il nome, valuta la validità di una stringa come codifica di una posizione sulla scacchiera.

```
require
  is_valid_code (where)
```

Ci sono altre condizioni in cui è ragionevole evitare chiamate alle tre *feature*? Per `BOARD.get` direi senz'altro di no. Anche per `BOARD.put` non sembra utile mettere ulteriori condizioni: si noti che il parametro `p` è già implicitamente vincolato a essere `p /= Void` e potrebbe essere utile mettere un pezzo dove già se ne trova un altro (per esempio in una "presa"), quindi eviterei anche `is_empty(where)`. Volendo, ma al mio gusto personale pare superflua (potrebbe essere utile poterla chiamare anche in situazioni in cui non ha effetto), per `BOARD.remove` si potrebbe invece specificare `not is_empty(where)`.

Per le postcondizioni, invece, occorre soffermarsi sull'obiettivo di ciascuna *feature*.

Per la *query* `BOARD.get` si tratta del pezzo in una data posizione: il risultato è `detachable` e sarà `Void` quando la posizione è vuota.

```
ensure
  empty_void: is_empty (where) implies Result =
  ↪ Void
  full_piece: not is_empty (where) implies
  ↪ attached Result
```

Che possiamo anche condensare in:

```
ensure
  not is_empty (where) = attached Result
```

Per il *command* `BOARD.put` si tratta di garantire che la posizione contenga effettivamente il pezzo fornito dal *client*.

```
ensure
  get (where) = p
```

Per il *command* `BOARD.remove` si tratta di garantire che la posizione risulti effettivamente vuota.

```
ensure
  is_empty (where)
```

4. Migliorare le precondizioni di `BOARD.move`

   Le precondizioni date nel testo dell'esercizio già asseriscono che la posizione di partenza deve contenere un pezzo, mentre quella d'arrivo deve essere vuota (per le "prese" servirà evidentemente un altro comando). Possiamo eventualmente aggiungere che entrambe le posizioni devono essere valide.

   Per dire invece, come richiesto, che la mossa deve essere valida per il pezzo nella posizione di partenza occorre risolvere un problema: `get (fromw)` è una *query* con un risultato `detachable` quindi non possiamo scrivere semplicemente `get (fromw).is_valid_move(Current, fromw, tow)`.

   Il modo più semplice è scrivere una *query* ad hoc:

```
is_potentially_a_valid_move(fromw, tow: STRING):
↪  BOOLEAN
   local
     p: detachable PIECE
   do
     p := get (fromw)
     if p /= Void then
       Result := p.is_valid_move (Current,
       ↪  fromw, tow)
     end
   end
```

E quindi:

```
move (fromw, tow: STRING)
  require
    is_valid_code (fromw) and is_valid_code (tow)
    ok_from: not is_empty (fromw)
    ok_to: is_empty (tow)
    valid_move:
    ↪  is_potentially_a_valid_move(fromw, tow)
```

Con `EiffelStudio` si può anche condensare in un predicato, usando però il costrutto `attached ... as` che non fa parte dello *standard*.

```
move (fromw, tow: STRING)
  require
    is_valid_code (fromw) and is_valid_code (tow)
    ok_from: not is_empty (fromw)
    ok_to: is_empty (tow)
    valid_move: attached get (fromw) as o
    ↪  implies o.is_valid_move (Current, fromw,
    ↪  tow)
```

5. Invariante sul numero di Re sulla scacchiera

   Con una *query* `king_count` che restituisce il numero di Re sulla scacchiera, l'invariante diventa immediato.

```
invariant
  king_count >= 0 and king_count <= 2
```

   Oppure, se vogliamo **esattamente** due Re:

```
invariant
  king_count = 2
```

   In questo caso però bisogna agire anche sulle *feature* di creazione di `BOARD` perché al momento permettono la creazione di scacchiere senza pezzi, che quindi non rispetterebbero l'invariante.

   L'implementazione di `king_count` può essere fatti in molti modi diversi. Per esempio:

```
king_count: INTEGER
  local
    k: KING
    col: CHARACTER
    row: INTEGER
    pos: STRING
    p: detachable PIECE
  do
    from
      row := 1
    until
      row = 8 + 1
    loop
      from
```

```
        col := 'a'
      until
        col = 'h'.next
      loop
        pos := col.out + row.out
        p := get(pos)
        if p /= Void then
          if p.name.is_equal ("KING") then
            Result := Result + 1
          end
        end
        col := col.next
      end
      row := row + 1
    end
  end
```

Un'altra possibilità è mantenere aggiornato il valore di una *feature* `king_count: INTEGER`, incrementandola a ogni `put` di un Re e decrementandola a ogni `remove` di un Re. Si tratta di una soluzione più efficiente (l'invariante viene valutato prima e dopo ogni esecuzione delle *feature* di `BOARD`), ma più difficile da manutenere perché il *concern* risulta *sparpagliato* fra *feature* diverse.

7.2 Geometria

Dopo aver letto attentamente il codice Eiffel fornito, rispondere alle seguenti domande.

1. Si consideri una variabile `s: SQUARE`, sarebbe possibile creare un oggetto riferito da `s` con l'istruzione `create s.make (3.0, 5.0)`? Se la risposta è negativa indi-

care la natura dell'errore (*compile time*, violazione del contratto, altro errore *run time*).

2. Nella classe POLYGON è definito l'invariante `area > 0`. Cosa cambierebbe se la stessa condizione apparisse invece come postcondizione (`Result > 0`) della *feature* corrispondente? Illustrare un esempio in cui la differenza è rilevante.

3. Si considerino le postcondizioni della *feature* RECTANGLE.`rotate` e la sua implementazione. Definire precondizioni che limitino i valori dell'angolo di rotazione a quelli che danno luogo a rotazioni che non cambiano la direzione dei lati e permettano di rispettare le postcondizioni senza cambiare l'implementazione.

4. Nella classe RECTANGLE, definire postcondizioni per le *feature* `area` e `perimeter`. Nella classe REGULAR_POLYGON definire un invariante che esprima la condizione che tutti i lati devono avere la stessa lunghezza.

5. Vi hanno chiesto di implementare la *feature* TILER.`tiled_square` dandovi le postcondizioni: si richiede la creazione di un *quadrato* potenzialmente tassellabile (cioè ricopribile completamente senza sovrapposizioni) con i poligoni `tiles`. L'operazione però certamente non è possibile in tutti i casi e anche quando lo è potrebbe essere necessario un algoritmo complicato. Definire adeguate precondizioni che permettano di limitare il problema a situazioni (comunque generali) in cui costruire la tassellatura richiesta sarebbe facile (non è necessario implementare la *feature*). Chi dovrà assicurare la validità di tali precondizioni?

7.2.1 POLYGON

```
deferred class
  POLYGON

feature

  area: REAL
  deferred
  end

  perimeter: REAL
  deferred
  end

  n_vertices: INTEGER

  max_edge: REAL
  deferred
  end

  min_edge: REAL
  deferred
  end

invariant
  area > 0.0
    perimeter > 0.0
    min_edge <= max_edge
    min_edge > 0.0
    max_edge > 0.0
    n_vertices >= 3
end
```

7.2.2 REGULAR_POLYGON

```
deferred class
  REGULAR_POLYGON

inherit

  POLYGON
    redefine
      is_equal
    end

feature

  make_with_edge (edge: REAL)
    deferred
    end

  is_equal (other: like Current): BOOLEAN
    do
      Result := min_edge = other.min_edge
    end

end
```

7.2.3 RECTANGLE

```
class
  RECTANGLE

inherit

  POLYGON
    redefine
```

```
      is_equal
    end

create
  make

feature

  make (height, width: REAL)
    do
      x := width
      y := height
      n_vertices := 4
    end

  area: REAL
    do
      Result := x * y
    ensure then
      Result = x * y
    end

  perimeter: REAL
    do
      Result := 2 * (x + y)
    ensure then
      Result = 2 * (x + y)
    end

  rotate (angle: REAL)
    local
      a, tmp: REAL
    do
      a := angle.abs
```

```
    from
      a := angle.abs
    until
      a <= 0
    loop
      a := a - 90.
      tmp := x
      x := y
      y := tmp
    end
  ensure
    same: x = old x implies y = old y
    swapped: x /= old x implies y /= old y
  end

max_edge: REAL
  do
    Result := x.max (y)
  end

min_edge: REAL
  do
    Result := x.min (y)
  end

is_equal (other: like Current): BOOLEAN
    -- rectangles are equals if they have the same
    ↪  edges,
    -- possibly after a rotation of 90 degrees
  do
    Result := min_edge = other.min_edge and
    ↪  max_edge = other.max_edge
  end
```

```
feature {NONE}

  x, y: REAL

invariant
  x > 0.0
  y > 0.0

end
```

7.2.4 SQUARE

```
class
  SQUARE

inherit

  RECTANGLE
    select
      is_equal
    end

  REGULAR_POLYGON
    rename
      is_equal as is_equal_as_rp
    end

create
  make_with_edge

feature

  make_with_edge (width: REAL)
    do
```

```
      make (width, width)
    end

end
```

7.2.5 TILER

```
class
  TILER

create
  make

feature -- Initialization

  make
      -- Initialization for `Current'.
    do
      create tiles.make
    end

  full: BOOLEAN
    do
      Result := tiles.full
    end

  count: INTEGER
    do
      Result := tiles.count
    end

  add (p: POLYGON)
    require
      not full
```

```
  do
    tiles.extend (p)
  ensure
    tiles.count = old tiles.count + 1
  end

tiled_square: SQUARE
  do
      -- fake implementation
    create Result.make_with_edge (42)
  ensure
    no_sovrappositions: Result.area = total_area
    tiling: Result.perimeter <= total_perimeter
  end

total_area: REAL
  do
    across
      tiles as t
    from
      Result := 0
    loop
      Result := Result + t.item.area
    end
  ensure
    Result > 0.
  end

total_perimeter: REAL
  do
    across
      tiles as t
    from
      Result := 0
```

```
      loop
        Result := Result + t.item.perimeter
      end
    ensure
      Result > 0.
    end

  all_squares: BOOLEAN
    do
      Result := across tiles as t all
      ↪ t.item.n_vertices = 4 and t.item.max_edge =
      ↪ t.item.min_edge end
    end

  all_equals: BOOLEAN
    do
      Result := across tiles as t all t.item.is_equal
      ↪ (tiles.first) end
    end

feature {NONE}

  tiles: LINKED_LIST [POLYGON]

invariant
  count >= 0

end
```

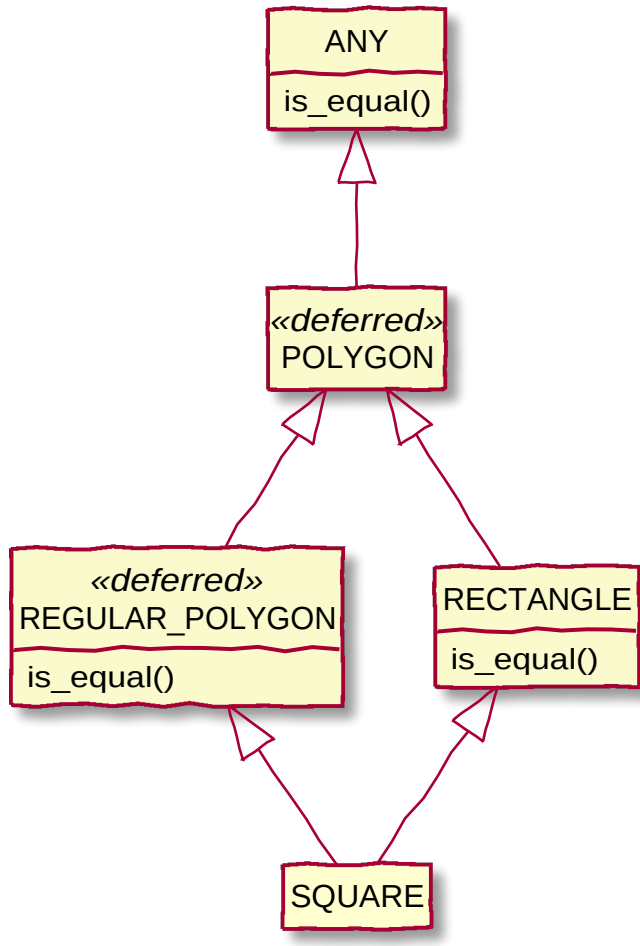
7.2.6 Soluzione

1. Creazione di uno SQUARE

   Un oggetto SQUARE non può essere creato con la *feature*

`make` (che infatti non appare nell'elenco dei `create`). Nel caso descritto, quindi, si avrebbe un errore di compilazione ("*Creation instruction uses call to improper feature.*"). Del resto non avrebbe senso creare un quadrato con due lati diversi, bisogna dunque usare `s.make_with_edge` che accetta un unico parametro `REAL`. Si noti, invece, che la chiamata della *feature* `s.make(3.0, 5.0)` una volta creato l'oggetto riferito da `s` sarebbe sintatticamente lecita, anche se probabilmente inopportuna (per esempio, finirebbe per violare l'invariante richiesto alla domanda 4.)

A questo proposito vale la pena di notare che il meccanismo dell'ereditarietà in Eiffel è molto flessibile: permette ereditarietà multiple, per esempio, e ciò obbliga a considerare il caso in cui vi siano *feature* con lo stesso nome. Nelle classi dell'esercizio le due *feature* con lo stesso nome sono in realtà la stessa *feature* (almeno semanticamente, visto che si tratta di implementazioni differenti) ereditata via percorsi parentali differenti, come si può vedere dal diagramma delle classi; in generale è però possibile che ci siano *feature* con lo stesso nome, ma del tutto scorrelate.



Le *feature* in conflitto possono essere rinominate, ottenendo così due *feature* differenti: occorre quindi *selezionare* quella che verrà utilizzata nel *binding dinamico*, quella cioè effettivamente disponibile nella classe con il nome imposto dal tipo.

```
class
  SQUARE

inherit

  RECTANGLE
    select
      is_equal
    end

  REGULAR_POLYGON
    rename
      is_equal as is_equal_as_rp
    end

end
```

Si immagini di avere un riferimento `p: POLYGON`. Se a `p` viene attaccato uno `SQUARE`, `p.is_equal` sarà quella ereditata da `RECTANGLE`, come precisato dalla clausola `select`. Si potrebbe ulteriormente specificare che il nome `is_equal_as_rp` (cioè il nome con cui `SQUARE` ha accesso a `REGULAR_POLYGON.is_equal`) non viene esportato a nessun *client*, evitando quella che è semanticamente una duplicazione (anche se il codice eseguito nei due casi è diverso); similmente può essere utile evitare di esportare `make`, che come abbiamo detto, è inutile per i *client* di `SQUARE` (ma usata da `SQUARE` internamente). Si noti che in altri casi l'esclusione dall'esportazione può causare i consueti problemi dovuti alla violazione del principio di Liskov: in questo esempio però la *feature* `make` serve alla creazione di `RECTANGLE` ed è inadeguato per creare `SQUARE` (bisogna usare `make_with_e`⌋

dge) quindi non ci sono problemi con il polimorfismo, perché non sarebbe utile chiamare `make` su oggetti di tipo RECTANGLE una volta creati (e che potrebbero essere SQUARE).

```
class
  SQUARE

inherit

  RECTANGLE
    export
      {NONE} make
    select
      is_equal
    end

  REGULAR_POLYGON
    rename
      is_equal as is_equal_as_rp
    export
      {NONE} is_equal_as_rp
    end
end
```

2. Invarianti e postcondizioni

   Un invariante è una proprietà valida per ogni stato stabile di un oggetto, mentre la postcondizione deve valere solo al termine dell'esecuzione della *feature* cui si riferisce. Considerando la proprietà `area > 0`: come invariante deve valere non appena viene creato un poligono (per esempio, la creazione di uno SQUARE di lato 0.0 genera un'eccezione); come postcondizione, invece,

un poligono di potenziale area nulla avrebbe diritto di esistere, a patto di non chiedere, appunto, di calcolarne l'area.

3. Precondizioni di `RECTANGLE.rotate`

   Perché la *feature* funzioni correttamente occorre che `angle` non sia tale da alterare il modo in cui il rettangolo è rappresentato, cioè solo dalla lunghezza dei suoi lati. Va bene qualsiasi multiplo (anche negativo, visto che l'implementazione lavora sul valore assoluto) di 90 gradi.

   ```
   rotate (angle: REAL)
     require
       is_integer: angle.floor = angle
       keep_axes: angle.abs.floor.integer_remainder
       ↪ (90) = 0
   ```

4. Postcondizioni di `RECTANGLE.area` e `RECTANGLE.perimeter`, invariante di `REGULAR_POLYGON`

   Sia per `RECTANGLE.area` che per `RECTANGLE.perimeter`, valgono già gli invarianti ereditati da `POLYGON`: devono avere sempre valori positivi. Al termine del calcolo, però, possiamo verificare che il risultato non solo sia positivo, ma anche quello che ci si aspetta per un rettangolo con i lati `x` e `y`.

   ```
   area: REAL
     do
       Result := x * y
     ensure then
       Result = x * y
     end
   ```

```
perimeter: REAL
  do
    Result := 2 * (x + y)
  ensure then
    Result = 2 * (x + y)
  end
```

Per esprimere il fatto che un generico poligono regolare (di cui quindi non conosciamo il numero di lati, né le eventuali *feature* per ottenerne le lunghezze), possiamo usare le *query* `POLYGON.min_edge` e `POLYGON.max_edge`: in un poligono con tutti i lati uguali dovranno dare lo stesso risultato.

```
invariant
  min_edge = max_edge
```

5. Precondizioni di `TILER.tiled_square`

   Costruire la tassellatura diventa molto facile se `tiles` è una serie di n^2 quadrati tutti uguali (con lato L): in questo caso il quadrato di lato $n \cdot L$ risolve il problema. La classe dispone già di *feature* per verificare se i `tiles` sono tutti quadrati (`all_squares`) e se sono tutti uguali (`all_equals`); bisogna poi assicurarsi che il numero dei `tiles` (accessibile tramite `count`) sia un quadrato perfetto.

```
tiled_square: SQUARE
        require
                all_squares
                all_equals
```

```
            count_is_square: count.power
            ↪ (.5).truncated_to_integer *
            ↪ count.power
            ↪ (.5).truncated_to_integer =
            ↪ count
    local
            edge: REAL
    do
            edge := tiles.first.min_edge *
            ↪ tiles.count.power
            ↪ (.5).truncated_to_real
            create Result.make_with_edge
            ↪ (edge)
    ensure
            no_sovrappositions: Result.area
            ↪ = total_area
            tiling: Result.perimeter <=
            ↪ total_perimeter
    end
```

La validità di queste precondizioni sono un onere per i *client* della classe `TILER`: per chiamare `tiled_squa⌋re` dovranno assicurarsi di avere un riferimento a un oggetto a cui siano stati aggiunti (via `add`) a `tiles` poligoni che rispettino la precondizione (nella soluzione data dovranno essere un numero quadrato di quadrati tutti uguali).

7.3 Conti bancari

1. Siano `a` e `b` due riferimenti `attached` ciascuno a un oggetto di tipo `ACCOUNT`. Per ognuna di queste istruzioni, indicare se sono permesse o vietate dalle precondizio-

ni, spiegando il motivo e fornendo, se necessario, le ulteriori condizioni su `a` e `b` che permetterebbero un'esecuzione conforme al contratto.

```
a.transfer(-1, b)
```

```
a.transfer(100, b)
```

```
a.transfer(100, a)
```

```
a.transfer(100, a.twin)
```

```
b.transfer(b.balance, a)
```

2. Si consideri la sequenza di istruzioni:

```
create a.make
a.set_credit_limit (X)
a.withdraw (W)
a.set_credit_limit (Y)
a.withdraw (Z)
```

   Per quali valori `X`, `Y`, `W`, `Z`: `INTEGER` (con `X /= Y`) la sequenza è ammessa dal contratto della classe `ACCOUNT`? Quale sarà il valore di `balance` al termine della sequenza?

3. Si scriva una *feature* `merge` che unisce due `ACCOUNT` differenti: il `balance` risultante deve essere la somma e il `credit_limit` il maggiore dei due. Indicare le precondizioni e postcondizioni opportune, tenendo conto degli invarianti della classe, che non devono essere alterati.

4. Si vuole progettare una classe `SAVING_ACCOUNT`, molto simile ad `ACCOUNT`, di cui si intende riutilizzare il più

possibile il codice: non è previsto però che il conto possa andare in rosso e non è prevista la possibilità di operazioni a credito (non è quindi opportuno che fornisca ai suoi *client* le *feature* relative a queste operazioni). Fornire un contratto adeguato per SAVING_ACCOUNT, chiarendo anche la sua relazione con ACCOUNT.

5. La *feature* ACCOUNT.log_balance fa una chiamata che potrebbe fallire (al momento simulata da un'asserzione che fallisce sempre). Gestire questa possibilità di fallimento riprovando tre volte. Quale problema occorre risolvere? Si supponga pure che le altre *feature* utilizzate non falliscano mai.

7.3.1 ACCOUNT

```
class
  ACCOUNT

create
  make

feature {NONE} -- Initialization

  make
      -- Initialize empty account.
    do
      balance := 0
      credit_limit := 1000
    ensure
      balance_set: balance = 0
      credit_limit_set: credit_limit = 1000
    end
```

```
feature -- Access

  credit_limit: INTEGER
      -- Credit limit of this account.

  available_amount: INTEGER
      -- Amount available on this account.
    do
      Result := balance + credit_limit
    end

  balance: INTEGER
      -- Balance of this account.

feature -- Element change

  set_credit_limit (limit: INTEGER)
      -- Set `credit_limit' to `limit'.
    require
      limit >= (0).max (- balance)
    do
      credit_limit := limit
    ensure
      credit_limit_set: credit_limit = limit
    end

  deposit (amount: INTEGER)
      -- Deposit `amount' in this account.
    require
      amount_non_negative: amount >= 0
    do
      balance := balance + amount
    ensure
      balance_set: balance = old balance + amount
```

```
    end

  withdraw (amount: INTEGER)
      -- Withdraw `amount' from this account.
    require
      positive_amount: amount > 0
      may_withdraw: amount <= available_amount
    do
      balance := balance - amount
    ensure
      balance_set: balance = old balance - amount
    end

feature -- Basic operations

  transfer (amount: INTEGER; other: ACCOUNT)
      -- Transfer `amount' from this account to
      ↪ `other'.
    require
      positive_amount: amount > 0
      may_withdraw: amount <= available_amount
      no_aliasing: other /= Current
    do
      balance := balance - amount
      other.deposit (amount)
    ensure
      withdrawal_made: balance = old balance - amount
      deposit_made: other.balance = old other.balance
      ↪ + amount
      same_credit_limit: credit_limit = old
      ↪ credit_limit
      other_same_credit_limit: other.credit_limit =
      ↪ old other.credit_limit
    end
```

```
feature

  log_balance
    local
      log: PLAIN_TEXT_FILE
    do
      create log.make_open_append ("account.log")
      log.put_string ("Balance is: " + balance.out +
      ↪  "%N")
        -- The next operation fails
      check
          -- failure
        False
      end
    end

invariant
  credit_limit_not_negative: credit_limit > 0
  balance_not_below_credit: balance >= - credit_limit

end
```

7.3.2 Soluzione

1. Conformità delle operazioni di `transfer`

   Per valutare la conformità delle operazioni occorre prendere in considerazione il contratto che regola le entità coinvolte:

   - `a` e `b` sono due oggetti `ACCOUNT`, sappiamo quindi (dagli invarianti) che valgono le seguenti condizioni:

```
credit_limit > 0
balance >= - credit_limit
```

- per la *feature* `transfer` vale:

```
transfer (amount: INTEGER_32; other: ACCOUNT)
    -- Transfer `amount` from this account
    ↪  to `other`.
  require
    positive_amount: amount > 0
    may_withdraw: amount <= available_amount
    no_aliasing: other /= Current
  ensure
    withdrawal_made: balance = old balance -
    ↪  amount
    deposit_made: other.balance = old
    ↪  other.balance + amount
    same_credit_limit: credit_limit = old
    ↪  credit_limit
    other_same_credit_limit:
    ↪  other.credit_limit = old
    ↪  other.credit_limit
```

Possiamo ora considerare le istruzioni della domanda:

- `a.transfer(-1, b)`

  Il primo parametro di `transfer` deve essere positivo, l'istruzione è vietata.

- `a.transfer(100, b)`

  Supponiamo innanzitutto che `b /= a`, in questo caso l'istruzione è permessa se `100 <= a.availa⌋ble_amount`. Volendo (questa informazione non è però parte del contratto, ma deve essere dedotta dall'implementazione) possiamo ulteriormente

espandere la condizione in `100 <= a.balance + a.credit_limit`.

- `a.transfer(100, a)`
  L'istruzione è sempre vietata, per la precondizione `other /= Current`, valida solo se `a /= a`.
- `a.transfer(100, a.twin)`
  La *feature* `ANY.twin` duplica un oggetto, quindi si ha certamente `a.twin /= a`. L'istruzione è quindi permessa se `100 <= a.available_amount`.
- `b.transfer(b.balance, a)`
  Anche qui supponiamo che `a /= b`, in questo caso l'istruzione è permessa se `b.balance > 0 and b.balance <= b.available_amount`. Guardando l'implementazione potremmo espandere la condizione in `b.balance > 0 and b.balance <= b.balance + b.credit_limit`, che si riduce a `b.balance > 0 and 0 <= b.credit_limit` e, per l'invariante `credit_limit > 0`, basta quindi che `b.balance > 0`.

Da queste considerazioni emerge che sarebbe molto utile esplicitare nel contratto anche la relazione fra `available_amount`, `balance` e `credit_limit`, aggiungendo l'invariante `available_amount = balance + credit_limit`.

2. Conformità di una sequenza

   In questo caso è opportuno seguire il flusso:

```
create a.make
-- a.balance = 0 and a.credit_limit = 1000 and
↪  a.credit_limit > 0 and a.balance >= -
↪  a.credit_limit

-- valido solo se: X >= (0).max (0)
-- cioè solo se: X >= 0
a.set_credit_limit (X)
-- a.credit_limit = X and a.credit_limit > 0 and
↪  a.balance >= - a.credit_limit
-- quindi deve essere X > 0 (la precondizione è
↪  imperfetta)

-- valido solo se: W > 0 and W <=
↪  a.available_amount
-- cioè solo se: W > 0 and W <= a.balance +
↪  a.credit_limit
-- cioè solo se: W > 0 and W <= 0 + X
a.withdraw (W)
-- a.balance = 0 - W and a.credit_limit > 0 and
↪  a.balance >= - a.credit_limit
-- quindi deve essere -W >= -X (cioè W <= X)

-- valido solo se: Y >= (0).max (-(-W))
-- cioè solo se: Y >= W (perché W > 0)
a.set_credit_limit (Y)
-- a.credit_limit = Y and a.credit_limit > 0 and
↪  a.balance >= - a.credit_limit

-- valido solo se: Z > 0 and Z <=
↪  a.available_amount
-- cioè solo se: Z > 0 and Z <= a.balance +
↪  a.credit_limit
-- cioè solo se: Z > 0 and Z <= -W + Y
```

```
a.withdraw (Z)
-- a.balance = -W - Z and a.credit_limit > 0 and
↪  a.balance >= - a.credit_limit
-- quindi deve essere -W - Z >= -Y (cioè W + Z
↪  <= Y)
```

Riassumendo: `X > 0 and W > 0 and W <= X and Y >= W and Z > 0 and Z <= -W + Y`; cioè: `W > 0 and X >= W and Z > 0 and Y >= Z + W`. Al termine: `a.balance = -(W + Z)`

Detto a parole: il primo limite al credito deve essere positivo, il primo prelievo deve essere minore o uguale al limite al credito (il saldo iniziale è zero); il secondo limite al credito deve essere maggiore o uguale a quanto prelevato (a credito) e il secondo prelievo deve essere minore o uguale al limite al credito a cui va sottratto il primo prelievo. Il saldo finale è un valore a credito pari alla somma dei prelievi.

3. *Feature* `merge`

```
merge (other: ACCOUNT)
  require
    no_aliasing: other /= Current
    total_credit: other.balance + balance >= -
    ↪  credit_limit.max (other.credit_limit)
  do
    balance := balance + other.balance
    credit_limit := credit_limit.max
    ↪  (other.credit_limit)
  ensure
    set_balance: balance = old balance +
    ↪  other.balance
```

```
    set_credit_limit: credit_limit >= old
    ↪  credit_limit and credit_limit >=
    ↪  other.credit_limit
    other_unaffected: other.balance = old
    ↪  other.balance and other.credit_limit =
    ↪  old other.credit_limit
  end
```

Oppure si può calcolare un terzo ACCOUNT.

```
merge (other: ACCOUNT): ACCOUNT
  require
    no_aliasing: other /= Current
    total_credit: other.balance + balance >= -
    ↪  credit_limit.max (other.credit_limit)
  do
    create Result.make
    Result.set_credit_limit (credit_limit.max
    ↪  (other.credit_limit))
    if balance < 0 then
      Result.withdraw (-balance)
    elseif balance > 0 then
      Result.deposit (balance)
    end
    if other.balance < 0 then
      Result.withdraw (-other.balance)
    elseif other.balance > 0 then
      Result.deposit (other.balance)
    end
  ensure
    set_balance: Result.balance = balance +
    ↪  other.balance
    set_credit_limit: Result.credit_limit >=
    ↪  credit_limit and Result.credit_limit >=
    ↪  other.credit_limit
```

```
    other_unaffected: other.balance = old
    ↪  other.balance and other.credit_limit =
    ↪  old other.credit_limit
    pure: balance = old balance and credit_limit
    ↪  = old credit_limit
  end
```

Si noti che in questo caso siamo obbligati a utilizzare le *feature* disponibili ai *client* di ACCOUNT e quindi siamo tenuti al rispetto delle precondizioni.

4. SAVING_ACCOUNT

   Dai requisiti descritti risulta che un SAVING_ACCOUNT non può essere utilizzato in alcune delle situazioni in cui può essere utilizzato un ACCOUNT: è quindi da evitare una relazione di ereditarietà, che comporterebbe problemi con il principio di Liskov. Per riutilizzare comunque il codice di ACCOUNT si può comunque associare un oggetto ACCOUNT a ciascun SAVING_ACCOUNT: a questo oggetto verranno poi delegate le operazioni; nel codice che segue l'implementazione è però omessa perché siamo interessati solo al contratto. Pre- e postcondizioni delle *feature* rimangono sostanzialmente le stesse; è importante però chiarire nel contratto che, al contrario di quanto potrebbe succedere con ACCOUNT, il saldo è invariabilmente non negativo (e la cifra disponibile per il prelievo uguale al saldo).

```
deferred class
  SAVING_ACCOUNT

feature {NONE}
```

```
  internal: ACCOUNT

feature -- Access

  available_amount: INTEGER
      -- Amount available on this account.

  balance: INTEGER
      -- Balance of this account.

feature -- Element change

  deposit (amount: INTEGER)
      -- Deposit `amount' in this account.
    require
      amount_non_negative: amount >= 0
    deferred
    ensure
      balance_set: balance = old balance + amount
    end

  withdraw (amount: INTEGER)
      -- Withdraw `amount' from this account.
    require
      positive_amount: amount > 0
      may_withdraw: amount <= available_amount
    deferred
    ensure
      balance_set: balance = old balance - amount
    end

feature -- Basic operations
```

```
transfer (amount: INTEGER; other:
↪ SAVING_ACCOUNT)
    -- Transfer `amount' from this account to
    ↪ `other'.
  require
    positive_amount: amount > 0
    may_withdraw: amount <= available_amount
    no_aliasing: other /= Current
  deferred
  ensure
    withdrawal_made: balance = old balance -
    ↪ amount
    deposit_made: other.balance = old
    ↪ other.balance + amount
  end

transfer_account (amount: INTEGER; other:
↪ ACCOUNT)
    -- Transfer `amount' from this account to
    ↪ `other'.
  require
    positive_amount: amount > 0
    may_withdraw: amount <= available_amount
  deferred
  ensure
    withdrawal_made: balance = old balance -
    ↪ amount
    deposit_made: other.balance = old
    ↪ other.balance + amount
    other_same_credit_limit:
    ↪ other.credit_limit = old
    ↪ other.credit_limit
  end
```

```
invariant
  balance >= 0
  available_amount = balance

end
```

Potrebbe essere utile avere la possibilità di *convertire* (vedi standard 8.15) oggetti SAVING_ACCOUNT in ACCOUNT (si noti che la conversione inversa non è invece sempre possibile).

```
deferred class
  SAVING_ACCOUNT

convert
  to_account: {ACCOUNT}

feature -- Conversion

  to_account: ACCOUNT
    deferred
    end

-- come prima

end
```

Questo permetterebbe di usare direttamente oggetti SAVING_ACCOUNT negli assegnamenti a riferimenti di tipo ACCOUNT.

```
local
   a: ACCOUNT
   s: SAVING_ACCOUNT
```

```
do
    -- ...
    a := s
    -- o equivalentemente
    a := s.to_account
end
```

5. Trattamento del fallimento

   Il modo standard con cui si tratta un fallimento con una serie di tentativi è qualcosa del tipo che segue. Bisogna scegliere dove mettere il controllo sul numero di tentativi: nel corpo principale della *feature* e non nella clausola `rescue` è la soluzione più opportuna se il chiamante può continuare il suo processo di calcolo (ossia se le postcondizioni sono soddisfatte, come in questo caso).

```
log_balance
  local
    log: PLAIN_TEXT_FILE
    tentativi: INTEGER
  do
    create log.make_open_append ("account.log")
    log.put_string ("Balance is: " + balance.out
    ↪  + "%N")
      -- The next operation fails
    if tentativi < 3 then
      check
          -- failure
        False
      end
      print("OK!%N")
    else
      print("Giving up...%N")
```

```
    end
  rescue
    tentativi := tentativi + 1
    retry
  end
```

Va notato, però, che la `retry` non ripristina esattamente lo stato iniziale perché c'è un effetto collaterale nell'ambiente di esecuzione, la scrittura di una riga nel *file* `account.log`: ogni tentativo aggiungerebbe quindi una riga al *log*, rendendolo inaffidabile (per esempio, se l'operazione fallisce sempre, avremo quattro righe nel *log*). Bisogna quindi accertarsi che la riga venga scritta solo se l'operazione che può fallire è andata a buon fine. Se quest'ultima non dipende da quelle precedenti, questo effetto si può ottenere spostando le operazioni con effetti collaterali dopo l'operazione problematica.

```
log_balance
  local
    log: PLAIN_TEXT_FILE
    tentativi: INTEGER
  do
      -- The next operation fails
    if tentativi < 3 then
      check
          -- failure
        False
      end
      create log.make_open_append ("account.log")
      log.put_string ("Balance is: " +
      ↪  balance.out + "%N")
    else
      print("Giving up...%N")
```

```
    end
  rescue
    tentativi := tentativi + 1
    retry
  end
```

In generale, però, sarà necessario *annullare* le operazioni con effetti collaterali, se si vuole riprovare senza alterazioni. Purtroppo l'annullamento spesso non è possibile (se l'operazione non è reversibile, come nel caso dell'invio di un messaggio *email* o l'invio di un comando a un attuatore meccanico) o è troppo laborioso. A volte un'altra possibilità da prendere in considerazione è la *compensazione*: non è possibile annullare un'*email* ormai inviata, ma è possibile mandarne un'altra di rettifica.

Nel nostro caso, l'annullamento potrebbe essere qualcosa di simile al codice che segue e che ricrea `account.log` senza la riga aggiunta prima del fallimento. Per semplicità i metadati del *file* (ora di ultimo accesso, ecc.) sono ignorati.

```
log_balance
  local
    log, cp: PLAIN_TEXT_FILE
    tentativi, lines: INTEGER
  do
    if tentativi < 3 then
      create log.make_open_append ("account.log")
      log.put_string ("Balance is: " +
      ↪  balance.out + "%N")
        -- The next operation fails
      check
```

```
        -- failure
      False
    end
    print ("OK!%N")
  else
    print ("Giving up...%N")
  end
rescue
  tentativi := tentativi + 1
  lines := 0 -- utile nei tentativi successivi
  ↪  al primo
  if attached log then
    log.close
  end
  create log.make_open_read ("account.log")
  create cp.make_open_temporary
  from
    log.read_line
  until
    log.exhausted
  loop
    lines := lines + 1
    cp.put_string (log.last_string + "%N")
    log.read_line
  end
  log.close
  cp.close
  cp.open_read
  create log.make_open_write ("account.log")
  from
    cp.read_line
  until
    lines = 1
  loop
```

```
        log.put_string (cp.last_string + "%N")
        cp.read_line
        lines := lines - 1
      end
      log.close
      cp.close
      cp.delete
      retry
    end
```

7.4 La struttura dati RING_BUFFER

Si consideri la classe RING_BUFFER:

1. Proporre pre- e postcondizioni adeguate per la *feature* count. Proporre inoltre pre- e postcondizioni adeguate per le *feature* is_empty e is_full senza fare uso delle *feature* start e free.

2. Proporre ulteriori invarianti per la classe RING_BUFFER.

3. Proporre postcondizioni adeguate per le *feature* extend e remove. In particolare si specifichi che a seguito di una extend(x), l'oggetto x è contenuto nel RING_BUFFER.

4. Sia b: RING_BUFFER[CHARACTER], si considerino le sequenze di istruzioni:

   ```
   create b.make (X)
   b.extend ('a')
   b.extend ('b')
   ```

   e

```
create b.make (Y)
b.remove
```

Per quali valori (se esistono) di `X` e di `Y` le due sequenze sono compatibili con il contratto di `RING_BUFFER`? (rispondere separatamente per ciascuna sequenza, fare riferimento al contratto risultante dopo aver risposto alle domande 1–3).

5. Le attuali postcondizioni di `item` non sono sufficienti per garantire che la struttura dati `RING_BUFFER` adotti una politica *First-In- First-Out*. Usare una *feature* aggiuntiva `model: BOUNDED_QUEUE[G]` (`BOUNDED_QUEUE` si trova nelle librerie di base, `Libraries/base/elks/structures/dispenser`) per esprimere questa specifica nella postcondizione di `item`, aggiungendo le istruzioni necessarie affinché essa venga coerentemente inizializzata e aggiornata dalle altre *feature* della classe. Se si volesse inoltre dire che la *feature* `extend` aggiunge un elemento che verrà estratto solo dopo tutti quelli già contenuti nel `RING_BUFFER`, che problema s'incontrerebbe?

7.4.1 RING_BUFFER

```
note
  description: "[
        FIFO queue implemented as a {RING_BUFFER} of
↪  elements of type {G}.

        The ring buffer uses two indexes for the
↪  first element and the
        next free slot of the buffer. The "Always
↪  keep one slot open"
```

```
        technique is used to distinguish between
↪  empty and full buffers.
  ]"

class
  RING_BUFFER [G]

create
  make

feature {NONE} -- Initialization

  make (n: INTEGER)
      -- Initialize empty buffer with capacity `n'.
    note
      status: creator
    require
      n_positive: n > 0
    do
      create data.make_empty
      data.grow (n + 1) -- one slot more
      data.trim
      start := 1
      free := 1
    ensure
      empty_buffer: is_empty
      capacity: capacity = n
    end

feature -- Access

  item: G
      -- Current item of buffer.
    require
```

```
      not is_empty
    do
      Result := data [start]
    ensure
      Result = data [start]
    end

  count: INTEGER
      -- Number of items in buffer.
    do
      if free >= start then
        Result := free - start
      else
        Result := data.count - start + free
      end
    end

  capacity: INTEGER
      -- Maximum capacity of buffer.
    do
      Result := data.count - 1
    ensure
      Result = data.count - 1
    end

feature -- Status report

  is_empty: BOOLEAN
      -- Is buffer empty?
    do
      Result := (start = free)
    end

  is_full: BOOLEAN
```

```
      -- Is buffer full?
    do
      if start = 1 then
        Result := (free = data.count)
      else
        Result := (free = start - 1)
      end
    end

feature -- Element change

  extend (a_value: G)
      -- Add `a_value' to end of buffer.
    require
      not is_full
    do
      data [free] := a_value
      if free = data.count then
        free := 1
      else
        free := free + 1
      end
    end

  remove
      -- Remove current item from buffer.
    require
      not is_empty
    do
      if start = data.count then
        start := 1
      else
        start := start + 1
      end
```

```
    end

  wipe_out
      -- Remove all elements from buffer.
    do
      start := free
    ensure
      is_empty
    end

feature {NONE} -- Implementation

  data: ARRAY [G]
      -- Array used to store data.

  start: INTEGER
      -- Index of first element.

  free: INTEGER
      -- Index of next free position.

invariant
  data_not_void: data /= Void

end
```

7.4.2 Casi di *test*

```
note
  description: "[
    Eiffel tests that can be executed by testing tool.
  ]"
  author: "EiffelStudio test wizard"
  date: "$Date$"
```

```
  revision: "$Revision$"
  testing: "type/manual"

class
  NEW_TEST_SET

inherit
  EQA_TEST_SET
    redefine
      on_prepare
    end

feature {NONE} -- Events

  on_prepare
      -- <Precursor>
    do
      create b.make(10)
    end

   b: RING_BUFFER [INTEGER]

feature -- Test routines

  test_buffer
    do
          assert("Initial capacity", b.capacity = 10)
          assert("Initial count", b.count = 0)

      b.extend (5)
      b.extend (8)

          assert("Capacity", b.capacity = 10)
          assert("Count", b.count = 2)
```

```
        assert("Item", b.item = 5)

    b.remove

        assert("Capacity after remove", b.capacity
        ↪  = 10)
        assert("Count after remove", b.count = 1)
        assert("Item after remove", b.item = 8)

    b.remove

        assert("Capacity after remove 2",
        ↪  b.capacity = 10)
        assert("Count after remove 2", b.count = 0)
  end

test_queries
  do
    assert("Empty", b.is_empty)
    assert("Not full", not b.is_full)

    across 1 |..| 10 as i loop b.extend (i.item) end

    assert("Not Empty", not b.is_empty)
    assert("Full", b.is_full)

    b.remove
    assert("Not Empty", not b.is_empty)
    assert("Not Full", not b.is_full)

  end

test_wipe_out
  do
```

```
    b.extend (-1)
    b.extend (-2)
    assert("Not Empty", not b.is_empty)
    b.wipe_out
    assert("Empty", b.is_empty)
  end

  test_char
  local
    bs: RING_BUFFER[CHARACTER]
    do
      create bs.make (3)
      bs.extend ('M')
      assert("Count", bs.count = 1)
    end

end
```

7.4.3 Soluzione

1. Contratto di `count`, `is_empty`, `is_full`

   La *feature* `count` fornisce il numero di elementi (*item*) nel *buffer*. Si tratta quindi di una *query* che può essere attivata in qualsiasi stato valido dell'oggetto, pertanto non è necessario specificare precondizioni.

   Per le postcondizioni possiamo specificare che il risultato è sempre un numero non negativo e minore della capacità del *buffer*. Questa è in realtà una proprietà generale e potrebbe in alternativa essere specificato più efficacemente come invariante di classe. Si noti che come invariante la condizione è più forte, perché deve valere dopo l'esecuzione di *qualsiasi feature*. Non è logica-

mente sbagliato duplicare la condizione (come postcondizione e come invariante), ma è un appesantimento inutile per i controlli a *run-time*.

```
ensure
    -- Questo potrebbe essere anche un invariante
  valid_count: Result >= 0 and Result <= capacity
end
```

Volendo è anche possibile aggiungere una postcondizione di controllo per l'implementazione.

```
ensure
  implementation_ok: (free >= start implies
  ↪  Result = free - start) and (free < start
  ↪  implies Result = data.count - start + free)
end
```

In questo caso, però, si introduce nel contratto informazione che riguarda dettagli che un *client* di `RING_BUFFER` non dovrebbe considerare, visto che le *feature* utilizzate sono precluse ai *client*, dato che sono esportate a `NONE`. Ciò sarebbe sintatticamente vietato per una precondizione (non si può chiedere ai *client* di rispettare vincoli che riguardano dettagli inaccessibili), ma è permesso nelle postcondizioni, che vincolano l'implementatore del componente e che quindi è tenuto a conoscerne tutti i dettagli. L'efficacia dal punto di vista della documentazione è però ambivalente, come sempre succede quando si esplicitano in una parte pubblica (il contratto) dettagli privati (le *feature* inaccessibili ai *client*).

Per `is_empty` e `is_full` la scelta è molto più facile: sono *query* che possono essere attivate in ogni stato valido dell'oggetto (quindi non servono precondizioni) e la loro semantica può essere descritta in termini di *feature* pubbliche, in particolare di `count`.

Per `is_empty`:

```
ensure
  Result = (count = 0)
end
```

Per `is_full`:

```
ensure
  Result = (count = capacity)
end
```

2. Invarianti

   Il codice dell'esercizio fornisce già un invariante relativo a `data`, la *feature* usata internamente per conservare gli elementi del *buffer*. Gli invarianti, infatti, servono a esplicitare la raison d'être di un componente, le sue responsabilità e le proprietà salienti delle sue collaborazioni; si rivolgono pertanto sia all'implementatore, sia ai *client* del componente stesso.

   Possiamo quindi sfruttare gli invarianti per mettere in luce i vincoli che legano `start` e `free` a `data`. Rendere esplicito che `capacity` è sempre positiva (ed eventualmente che `count` non è mai negativa, se non abbiamo reso questo vincolo con una postcondizione rispondendo alla domanda precedente). Inoltre, chiariamo anche che `is_empty` e `is_full` non possono essere contemporaneamente vere.

```
invariant
  data_not_void: data /= Void
  start_in_bounds: data.valid_index (start)
  free_in_bounds: data.valid_index (free)
  positive_capacity: capacity > 0
  not_both_empty_full: not (is_empty and is_full)
end
```

3. Postcondizioni di `extend` e `remove`

   La *feature* `extend` aggiunge un elemento alla struttura dati. L'effetto deve quindi essere quello di aumentarne il numero di elementi di uno. Inoltre specifichiamo anche, come richiesto dalla domanda, che l'elemento aggiunto deve essere contenuto nel *buffer* interno.

   ```
   ensure
     extended: count = old count + 1
     inserted: data.has (a_value)
   end
   ```

   Simmetricamente la *feature* `remove` diminuisce di uno il conteggio degli elementi.

   ```
   ensure
     shortened: count = old count - 1
   end
   ```

4. Conformità di sequenze

   Per valutare la conformità delle sequenze descritte nella domanda, occorre considerare pre- e postcondizioni delle `feature` coinvolte, tenendo inoltre presente gli invarianti. Ai fini di questa valutazione le condizioni

rivolte solo agli implementatori (quelle che coinvolgono *feature* non accessibili ai *client*) possono in realtà essere trascurate, perché non influiscono sulla parte di contratto che vincola i *client*.

```
-- X > 0
create b.make (X)
-- is_empty and capacity = X
-- cioè count = 0 and capacity = X
-- la precondizione garantisce l'invariante su
↪  capacity

-- not is_full
-- cioè not count = capacity
-- cioè not count = X
-- cioè not 0 = X
b.extend ('a')
-- count = old count + 1
-- cioè count = 1

-- not is_full
-- cioè not count = X
-- cioè not 1 = X
b.extend ('b')
-- count = old count + 1
-- cioè count = 2
```

Quindi in generale perché la sequenza sia conforme al contratto è sufficiente che $X > 1$.

```
-- Y > 0
create b.make (Y)
-- is_empty and capacity = Y
-- cioè count = 0 and capacity = Y
```

```
-- not is_empty
-- cioè not count = 0
-- not 0 = 0
-- false
b.remove
```

Perciò non c'è nessun valore di Y per cui la sequenza è valida.

5. Politica FIFO

   Come discusso nel paragrafo 6.1 si può utilizzare un oggetto "modello" che sappiamo comportarsi secondo la politica voluta per imporre nelle postcondizioni che il comportamento deve essere conforme a quello del modello. Occorre aggiornare coerentemente tutte le *feature* rilevanti.

```
class
  RING_BUFFER [G]

create
  make

feature {NONE} -- Initialization

  make (n: INTEGER)
      -- Initialize empty buffer with capacity
      ↪ `n'.
    note
      status: creator
    require
      n_positive: n > 0
    do
```

```
      create data.make_empty
      data.grow (n + 1) -- one slot more
      data.trim
      start := 1
      free := 1
      create model.make (n) -- Aggiunto per
      ↪  imporre la politica FIFO
    ensure
      empty_buffer: is_empty
      capacity: capacity = n
    end

feature -- Access

  item: G
      -- Current item of buffer.
    require
      not is_empty
    do
      Result := data [start]
    ensure
      Result = data [start]
    end

  count: INTEGER
      -- Number of items in buffer.
    do
      if free >= start then
        Result := free - start
      else
        Result := data.count - start + free
      end
    ensure
```

```
      valid_count: Result >= 0 and Result <=
      ↪  capacity
      implementation_ok: (free >= start implies
      ↪  Result = free - start) and (free <
      ↪  start implies Result = data.count -
      ↪  start + free)
    end

  capacity: INTEGER
      -- Maximum capacity of buffer.
    do
      Result := data.count - 1
    ensure
      Result = data.count - 1
    end

feature -- Status report

  is_empty: BOOLEAN
      -- Is buffer empty?
    do
      Result := (start = free)
    ensure
      Result = (count = 0)
    end

  is_full: BOOLEAN
      -- Is buffer full?
    do
      if start = 1 then
        Result := (free = data.count)
      else
        Result := (free = start - 1)
      end
```

```
    ensure
      Result = (count = capacity)
    end

feature -- Element change

  extend (a_value: G)
      -- Add `a_value' to end of buffer.
    require
      not is_full
    do
      data [free] := a_value
      if free = data.count then
        free := 1
      else
        free := free + 1
      end
      model.extend (a_value) -- Aggiunto per
      ↪ imporre la politica FIFO
    ensure
      extended: count = old count + 1
      inserted: data.has (a_value)
      fifo: item = model.item    -- Si comporta
      ↪ come il modello
    end

  remove
      -- Remove current item from buffer.
    require
      not is_empty
    do
      if start = data.count then
        start := 1
      else
```

```
      start := start + 1
    end
    model.remove            -- Aggiunto per
    ↪ imporre la politica FIFO
  ensure
    shortened: count = old count - 1
    fifo: old item = old model.item  -- Si
    ↪ comporta come il modello
  end

wipe_out
    -- Remove all elements from buffer.
  do
    start := free
    model.wipe_out          -- Aggiunto per
    ↪ imporre la politica FIFO
  ensure
    is_empty
  end

feature {NONE} -- Implementation

data: ARRAY [G]
    -- Array used to store data.

start: INTEGER
    -- Index of first element.

free: INTEGER
    -- Index of next free position.

model: BOUNDED_QUEUE [G]
    -- Aggiunto per imporre la politica FIFO
```

```
invariant
  data_not_void: data /= Void
  start_in_bounds: data.valid_index (start)
  free_in_bounds: data.valid_index (free)
  positive_capacity: capacity > 0
  not_both_empty_full: not (is_empty and is_full)
end
```

Se si volesse inoltre dire che la *feature* `extend` aggiunge un elemento che verrà estratto solo dopo tutti quelli già contenuti nel `RING_BUFFER`, bisognerebbe predicare sulla "storia" dell'oggetto. il che, in generale, è complicato perché la storia va codificata/conservata da qualche parte per poterla esaminare in una postcondizione. Nel caso specifico, però, si può sfruttare il fatto che il modello, essendo una coda, ha la stessa proprietà che vorremmo per la `extend` di `RING_BUFFER`, quindi se l'`item` restituito dopo una `extend` è lo stesso del modello, la proprietà è conservata.

7.5 *Tic-Tac-Toe*

Dopo aver letto attentamente il codice fornito che simula il gioco del tris (noto anche come *Tic-Tac-Toe*), rispondere alle seguenti domande.

1. Proporre almeno un ulteriore invariante per la classe `TICTACTOE`.

2. Proporre pre- e postcondizioni adeguate per le *feature* `TICTACTOE.play`. In particolare si specifichi che è possibile giocare una data casella solo se non è stata già

giocata e che il giocatore cambia dopo ogni turno di gioco.

3. Si specifichi con una opportuna postcondizione che la *feature* MATRIX_WITH_SYMMETRIES.rotate90 fa sì che quella che era la prima riga diventa l'ultima colonna della matrice.

4. Si specifichi con una opportuna postcondizione che la *feature* TICTACTOE.out non produce un carattere *newline* alla fine della stringa risultante.

5. Si implementi la *feature* MATRIX_WITH_SYMMETRIES.is_equal in maniera che vengano considerate uguali anche due matrici i cui valori coincidono dopo una rotazione di 180 gradi. Si specifichi che la *feature* non altera la matrice.

7.5.1 TICTACTOE

```
class
  TICTACTOE

inherit

  ANY
    redefine
      out, is_equal
    end

create
  make, make_ongoing

feature {NONE} -- Initialization
```

```
  make
    do
      create board.make_filled ('.', DIM, DIM)
    ensure
      x_starts: is_turn ('X')
      ongoing: not is_finished
    end

  make_ongoing (b: ARRAY [CHARACTER])
    do
      create board.make_filled ('.', DIM, DIM)
      across
        b as c
      loop
        board.enter (c.item, c.cursor_index)
        if c.item = 'X' or c.item = 'O' then
          turn := turn + 1
        end
      end
    end

feature {TICTACTOE}

  board: MATRIX_WITH_SYMMETRIES [CHARACTER]

  turn: INTEGER

  DIM: INTEGER
    once
      Result := 3
    end

feature {TTT_TEST_SET}
```

```
h_full (row: INTEGER; c: CHARACTER): BOOLEAN
  local
    i: INTEGER
  do
    Result := True
    from
      i := 1
    until
      Result = False or else not is_valid_index (i)
    loop
      if board [row, i] /= c then
        Result := False
      end
      i := i + 1
    end
  end

v_full (col: INTEGER; c: CHARACTER): BOOLEAN
  local
    i: INTEGER
  do
    Result := True
    from
      i := 1
    until
      Result = False or else not is_valid_index (i)
    loop
      if board [i, col] /= c then
        Result := False
      end
      i := i + 1
    end
  end
```

```
d_dx_full (start_col: INTEGER; c: CHARACTER):
↪  BOOLEAN
  local
    i, j: INTEGER
  do
    Result := True
    from
      i := start_col
      j := 1
    until
      Result = False or else not is_valid_index (i)
       ↪  or else not is_valid_index (j)
    loop
      if board [i, j] /= c then
        Result := False
      end
      i := i + 1
      j := j + 1
    end
  end

d_sx_full (start_col: INTEGER; c: CHARACTER):
↪  BOOLEAN
  local
    i, j: INTEGER
  do
    Result := True
    from
      i := start_col
      j := 1
    until
      Result = False or else not is_valid_index (i)
       ↪  or else not is_valid_index (j)
```

```
      loop
        if board [i, j] /= c then
          Result := False
        end
        i := i - 1
        j := j + 1
      end
    end

feature

  restart
    do
      board.clear_all
      board.fill_with ('.')
    ensure
      x_starts: is_turn ('X')
      ongoing: not is_finished
    end

  is_turn (c: CHARACTER): BOOLEAN
    do
      if c = 'X' then
        Result := turn.integer_remainder (2) = 0
      else
        Result := turn.integer_remainder (2) = 1
      end
    end

  is_winner (who: CHARACTER): BOOLEAN
    do
      Result := across 1 |..| board.width as col some
      ↪  v_full (col.item, who) end
```

```
    Result := Result or else across 1 |..|
    ↪ board.height as row some h_full (row.item,
    ↪ who) end
    Result := Result or else d_dx_full (1, who) or
    ↪ else d_sx_full (board.width, who)
  end

is_tie: BOOLEAN
  do
    Result := turn = board.count and not (is_winner
    ↪ ('X') or is_winner ('O'))
  end

is_finished: BOOLEAN
  do
    Result := is_winner ('X') or else is_winner
    ↪ ('O') or else is_tie
  end

play (row, col: INTEGER)
  do
    if is_turn ('X') then
      board [row, col] := 'X'
    else
      board [row, col] := 'O'
    end
    turn := turn + 1
  end

is_available (row, col: INTEGER): BOOLEAN
  do
    Result := board [row, col] = '.'
  end
```

```
is_valid_index (i: INTEGER): BOOLEAN
    -- True if i can be used as an index to select
    ↪  a row or a column of the board
  do
    Result := board.valid_index (i) and i <= DIM
  end

is_equal(other: like Current): BOOLEAN
do
  Result := board.is_equal (other.board)
end

out: STRING
  local
    hline: STRING
  do
    Result := ""
    hline := "-"
    hline.multiply (board.width * 2 - 1)
    across
      1 |..| board.height as row
    loop
      across
        1 |..| board.width as col
      loop
        if col.item /= 1 then
          Result.append_character ('|')
        end
        Result.append_character (board[row.item,
        ↪  col.item])
      end
      if row.item /= board.height then
        Result.append_string ("%N" + hline + "%N")
      end
```

```
      end
    end

invariant
  two_players: not (is_turn ('X') and is_turn ('O'))
  square_board: board.width = board.height
  turns: board.occurrences ('X') + board.occurrences
  ↪  ('O') = turn

end
```

7.5.2 MATRIX_WITH_SYMMETRIES

```
class
  MATRIX_WITH_SYMMETRIES [G]

inherit
  ARRAY2 [G]

create
  make, make_filled

feature

  rotate90
  -- Rotate matrix items 90 degrees clockwise
  local
    tmp: MATRIX_WITH_SYMMETRIES[G]
  do
    create tmp.make_filled (at (lower), height, width)
    tmp.copy(Current)
    make_filled (tmp.at (tmp.lower), width, height)
    across 1 |..| height as r loop
      across 1 |..| width as c loop
```

```
        put (tmp.item (tmp.height + 1 - c.item,
        ↪  r.item), r.item, c.item)
      end
    end
    ensure
      width = old height and height = old width
  end

  flip_columns
  -- Flip/exchange columns: the first becomes the
  ↪  last and so on
  local
    tmp: G
  do
    across 1 |..| height as r loop
    across 1 |..| (width // 2) as c loop
      tmp := item(r.item, c.item)
      put(item(r.item, width - c.item + 1), r.item,
      ↪  c.item)
      put(tmp, r.item, width - c.item + 1)
      end
    end
    ensure
      width = old width and height = old height
  end

end
```

7.5.3 APPLICATION

```
class
  APPLICATION

inherit
```

```
  ARGUMENTS_32

create
  make

feature {NONE} -- Initialization

  make
    local
      t: TICTACTOE
      i: INTEGER
    do
      create t.make
      print ("%NNuova partita%N")
      from
        i := 1
      until
        t.is_finished
      loop
        t.play (i.integer_quotient (3) + 1,
        ↪  i.integer_remainder (3) + 1)
        print ("%N" + t.out + "%N")
        i := i + 1
      end
      print ("%NFine partita%N")
    end

end
```

7.5.4 Casi di *test*

1. MATRIX_TEST_SET

```
class
  MATRIX_TEST_SET

inherit
  EQA_TEST_SET

feature -- Test routines

  test_rotate
  local
    matrix: MATRIX_WITH_SYMMETRIES[INTEGER]
    n: INTEGER
    do
      create matrix.make_filled(0, 3, 4)
      across 1 |..| 3 as r loop
        across 1 |..| 4 as c loop
          n := n + 1
          matrix.put (n, r.item, c.item)
        end
      end
      check
        matrix.item (1, 1) = 1
        matrix.item (2, 2) = 6
        matrix.item (3, 3) = 11
      end
      matrix.rotate90
      assert("wrong 1 1 -> " + matrix.item (1,
      ↪  1).out, matrix.item (1, 1) = 9)
      assert("wrong 2 2 -> " + matrix.item (2,
      ↪  2).out, matrix.item (2, 2) = 6)
      assert("wrong 3 3 -> " + matrix.item (3,
      ↪  3).out, matrix.item (3, 3) = 3)
      assert("wrong 4 2 -> " + matrix.item (4,
      ↪  2).out, matrix.item (4, 2) = 8)
```

```
    end

  test_flip_even
  local
    matrix: MATRIX_WITH_SYMMETRIES[INTEGER]
    n: INTEGER
    do
      create matrix.make_filled(0, 3, 4)
      across 1 |..| 3 as r loop
        across 1 |..| 4 as c loop
          n := n + 1
          matrix.put (n, r.item, c.item)
        end
      end
      check
        matrix.item (1, 1) = 1
        matrix.item (2, 2) = 6
        matrix.item (3, 3) = 11
      end
      matrix.flip_columns
      assert("wrong 1 1 -> " + matrix.item (1,
      ↪  1).out, matrix.item (1, 1) = 4)
      assert("wrong 2 2 -> " + matrix.item (2,
      ↪  2).out, matrix.item (2, 2) = 7)
      assert("wrong 3 3 -> " + matrix.item (3,
      ↪  3).out, matrix.item (3, 3) = 10)
    end


test_flip_odd
  local
    matrix: MATRIX_WITH_SYMMETRIES[INTEGER]
    n: INTEGER
    do
```

```
      create matrix.make_filled(0, 3, 3)
      across 1 |..| 3 as r loop
        across 1 |..| 3 as c loop
          n := n + 1
          matrix.put (n, r.item, c.item)
        end
      end
      check
        matrix.item (1, 1) = 1
        matrix.item (2, 2) = 5
        matrix.item (3, 3) = 9
      end
      matrix.flip_columns
      assert("wrong 1 1 -> " + matrix.item (1,
      ↪  1).out, matrix.item (1, 1) = 3)
      assert("wrong 2 2 -> " + matrix.item (2,
      ↪  2).out, matrix.item (2, 2) = 5)
      assert("wrong 3 3 -> " + matrix.item (3,
      ↪  3).out, matrix.item (3, 3) = 7)
    end


end
```

2. TTT_TEST_SET

```
class
  TTT_TEST_SET

inherit

  EQA_TEST_SET
    redefine
      on_prepare,
```

```
      on_clean
    end

feature {NONE} -- Events

  t: TICTACTOE

  on_prepare
      -- <Precursor>
    do
      create t.make
    end

  on_clean
      -- <Precursor>
    do
      t.restart
    end

feature -- Test routines

    test_v_full
    local
      tmp: TICTACTOE
    do
      create tmp.make_ongoing (<<'.','.','X',
                                 '.','.','X',
                                 '.','.','X'>>)
      assert ("%N" + tmp.out + "%N v_full not
      ↪  true", tmp.v_full(3, 'X'))
      assert ("%N" + tmp.out + "%N v_full not
      ↪  false", not tmp.v_full(1, 'X'))
      assert ("%N" + tmp.out + "%N X not
      ↪  winner", tmp.is_winner ('X'))
```

```
end


test_h_full
local
  tmp: TICTACTOE
do
  create tmp.make_ongoing (<<'X','X','X',
                            '.','.','O',
                            '.','.','O'>>)
  assert ("%N" + tmp.out + "%N h_full not
  ↪  true", tmp.h_full(1, 'X'))
  assert ("%N" + tmp.out + "%N h_full not
  ↪  false", not tmp.v_full(3, 'X'))
  assert ("%N" + tmp.out + "%N X not
  ↪  winner", tmp.is_winner ('X'))
end


test_d_dx_full
local
  tmp: TICTACTOE
do
  create tmp.make_ongoing (<<'X','.','.',
                            '.','X','O',
                            '.','.','X'>>)
  assert ("%N" + tmp.out + "%N d_dx_full not
  ↪  true", tmp.d_dx_full(1, 'X'))
  assert ("%N" + tmp.out + "%N d_sx_full not
  ↪  false", not tmp.d_sx_full(3, 'X'))
  assert ("%N" + tmp.out + "%N X not
  ↪  winner", tmp.is_winner ('X'))
end
```

```
test_d_sx_full
local
  tmp: TICTACTOE
do
  create tmp.make_ongoing (<<'O','.','X',
                            '.','X','O',
                            'X','.','X'>>)
  assert ("%N" + tmp.out + "%N d_sx_full not
  ↪  true", tmp.d_sx_full(3, 'X'))
  assert ("%N" + tmp.out + "%N d_dx_full not
  ↪  false", not tmp.d_dx_full(1, 'X'))
  assert ("%N" + tmp.out + "%N X not
  ↪  winner", tmp.is_winner ('X'))
end



test_tie
local
  tmp: TICTACTOE
do
  create tmp.make_ongoing (<<'X','O','X',
                            'O','X','O',
                            'O','X','O'>>)
  assert ("%N" + tmp.out + "%N not tie",
  ↪  tmp.is_tie)
end

test_index
do
  assert ("%N" + t.out + "%N not available",
  ↪  t.is_available (1, 1))
  assert ("%N" + t.out + "%N not valid",
  ↪  t.is_valid_index (3))
```

```
        assert ("%N" + t.out + "%N valid", not
        ↪  t.is_valid_index (4))
      end


  test_is_equal
  local
    a, b, c: TICTACTOE
  do
      create a.make_ongoing (<<'O','.','X',
                              '.','X','O',
                              'X','.','X'>>)
      create b.make_ongoing (<<'O','.','X',
                              '.','X','O',
                              'X','.','X'>>)
      create c.make_ongoing (<<'O','.','X',
                              '.','X','O',
                              'X','O','X'>>)

      assert("a not equal b", a.is_equal(b))
      assert("a equal c", not a.is_equal(c))

    end

end
```

7.5.5 Soluzione

1. Invarianti di TICTACTOE

   Sono moltissime le proprietà degli oggetti TICTACTOE che potrebbe risultare utile esplicitare come invariante. Con la prospettiva che abbiamo adottato in questo testo (i contratti come mezzo per facilitare il lavoro di grup-

po, più che come strumento di supporto alla correttezza) è opportuno mettere in luce la relazione esistente fra le *query* fornite dalla classe. Per esempio, potremmo specificare che quando il gioco non è finito, ci deve essere qualche casella disponibile.

```
not is_finished implies (across 1 |..|
↪ board.width as r some (across 1 |..|
↪ board.height as c some is_available (r.item,
↪ c.item) end) end)
```

2. Contratto di `TICTACTOE.play`

   `play` prende due interi che indicano la posizione della giocata. È opportuno quindi porre come precondizione che i due interi ricadano nell'intervallo opportuno e che la casella da giocare sia effettivamente disponibile.

   L'esecuzione di `play`, poi, rende indisponibile la casella giocata e cambia il giocatore di turno.

```
play (row, col: INTEGER)
        require
                valid_index: is_valid_index
                ↪ (row) and is_valid_index
                ↪ (col)
                available: is_available (row,
                ↪ col)
        do
           -- ....
        ensure
                not_available: not is_available
                ↪ (row, col)
```

```
                next: (old is_turn ('X') implies
                ↪  is_turn ('O')) and (old
                ↪  is_turn ('O') implies
                ↪  is_turn ('X'))
        end
```

3. Postcondizione di `MATRIX_WITH_SYMMETRIES.rotate90`

   Una rotazione di 90 gradi comporta lo scambio delle due dimensioni della matrice (`width` e `height`). Esprimere in forma dichiarativa (come è stilisticamente opportuno per un predicato; per esempio un `implies` è decisamente preferibile a un `if`, che pure sarebbe equivalente) che quella che era la prima riga diventa l'ultima colonna della matrice richiede qualche equilibrismo sintattico perché la forma più ovvia `old item(1, x.item)` purtroppo è sintatticamente scorretta, perché `x` è inutilizzabile nell'espressione `old`; occorre quindi usare l'espressione `old` per ottenere una copia dell'oggetto prima dell'esecuzione, e usarla per ottenere il valore con `x.item`). Il risultato è tutto sommato molto leggibile, anche se è bene ricordare che `across` non è parte dello standard.

   ```
   ensure
           width = old height and height = old width
           across 1 |..| height as x all
           ↪  item(x.item, width) = (old
           ↪  Current.twin).item (1, x.item)  end
   ```

   Un risultato equivalente si può ottenere con un ciclo tradizionale, che però va incapsulato in una *query* opportuna, perché serve un'espressione di tipo booleano.

4. Postcondizione di `TICTACTOE.out`

   La postcondizione è di immediata scrittura, va però tenuto presente che `TICTACTOE` ridefinisce la *feature* `out`, pertanto la postcondizione deve restringere quella originale: Eiffel infatti obbliga a usare un `ensure then`.

```
ensure then
        not Result.ends_with("%N")
```

5. Implementazione di `MATRIX_WITH_SYMMETRIES.is_equal`

   Innanzitutto occorre dichiarare che si tratta di una ridefinizione, visto che `is_equal` è una *feature* che risale addirittura a `ANY`.

```
inherit

  ARRAY2 [G]
    redefine
      is_equal
    end
```

   Per l'implementazione possiamo sfruttare l'implementazione di `is_equal` ereditata, accessibile tramite la parola chiave `Precursor` e aggiungere la possibilità che possa essere uguale anche dopo una doppia rotazione di 90 gradi. Aggiungiamo anche una postcondizione che garantisce che i valori non sono cambiati.

```
is_equal (other: like Current): BOOLEAN
  local
    tmp: like Current
  do
```

```
    if Precursor (other) then
      Result := True
    else
        tmp := other.twin
        tmp.rotate90
        tmp.rotate90
        Result := Precursor (tmp)
    end
  ensure then
    across 1 |..| count as i all Current.at
    ↪  (i.item) = (old Current.twin).at
    ↪  (i.item) end
    across 1 |..| count as i all other.at
    ↪  (i.item) = (old other.twin).at (i.item)
    ↪  end
  end
```

Volendo possiamo anche aggiungere un *test*.

```
test_eq_rotated
  local
    m1, m2: MATRIX_WITH_SYMMETRIES [INTEGER]
    n: INTEGER
  do
    create m1.make_filled (0, 3, 4)
    create m2.make_filled (0, 3, 4)
    across
      1 |..| 3 as r
    loop
      across
        1 |..| 4 as c
      loop
        n := n + 1
        m1.put (n, r.item, c.item)
```

```
      m2.put (n, 3+1-r.item, 4+1-c.item)
    end
  end
  check
    m1.item (1, 1) = 1
    m1.item (2, 2) = 6
    m1.item (3, 3) = 11
    m2.item (1, 1) = 12
    m2.item (2, 2) = 7
    m2.item (3, 3) = 2
  end

  assert ("m1 not equal to m2", m1.is_equal
  ↪  (m2))
end
```

8 Ringraziamenti

Questo scritto contiene senz'altro più errori di quanto desiderassi, ma, per merito dell'acribia di alcuni dei suoi lettori, meno di quanto è stata capace la mia distrazione. Approfitto perciò per ringraziare: Gaetano D'Agostino, Giuseppe Bartiromo, Flavio Forenza e Paolo Calcagni.

Indice analitico