Grafica ad oggetti con LuaTEX

Roberto Giacomelli

Sommario

Nell'articolo si sperimentano le potenzialità espressive e l'efficacia di un *linguaggio ad oggetti* per la grafica in LuaLaTeX cercando di delineare quali vantaggi otterrebbe l'utente programmando col nuovo paradigma rispetto ai tradizionali linguaggi di comandi come PGF, pstricks o METAPOST.

L'introduzione di LuaTeX permette di scrivere codice Lua direttamente in un sorgente TeX e questo rende possibile avvalersi di librerie ad oggetti ampliando ulteriormente le potenzialità del nuovo motore di composizione.

Abstract

In this paper we will make an attempt to measure the final TeX user advantages and efficacy of an object oriented language, the programming paradigm recently added to TeX world by the new typesetting engine LuaTeX.

A case study was developed writing a new graphic object library in Lua, based on the PGF package, carried in a source TeX files with a little ad very simple package.

1 Oggetti per TeX

Se esaminiamo un sorgente LaTeX noteremo una serie di *macro*, ciascuna con un numero variabile di argomenti ed opzioni, che può assumere anche la forma strutturata degli *ambienti* tramite codici di apertura e chiusura. La sintassi di questi comandi è quella prevista dal formato creato da Leslie Lamport e dai vari pacchetti d'estensione nell'idea semplice del paradigma *imperativo*.

Possiamo riassumere queste caratteristiche di LaTeX nelle seguenti affermazioni:

- *Paradigma*: linguaggio macro con comandi ed ambienti;
- *Programmazione utente*: no.

Questo approccio linguistico ha consentito lo sviluppo ininterrotto del sistema TeX dalla fine degli anni settanta ad oggi, rimanendo sostanzialmente invariato nel supportare il notevole numero di nuove e migliorate funzionalità tipografiche a disposizione dell'utente.

Qualche anno fa, quasi per caso giocando con Scite, l'editor programmabile in linguaggio Lua, alcuni sviluppatori proposero di ripetere l'idea includendo l'interprete Lua nel motore di composizione più importante ovvero pdfTeX. Nacque così LuaTeX, ufficialmente in fase di pre-release, con l'intenzione di colmare le lacune di TeX evidenziate con il confronto con altri linguaggi di programmazione.

Con LuaTeX infatti è possibile scrivere codice Lua direttamente in un sorgente TeX così che possono essere sviluppati pacchetti im modo molto più efficiente e dalle funzionalità più varie, dalla lettura di database server all'accesso al web. Nuovamente aumentano le funzionalità a disposizione, sempre più potenti e generali, ma il modo di utilizzarle si basa ancora su un linguaggio a sequenza di comandi dove la programmazione è nascosta all'utente e celata nelle librerie.

Il progetto dei formati basati su TeX, come LaTeX e ConTeXt, fondato su un linguaggio il più possibile semplice, non cambia con LuaTeX. Secondo questa idea, come per il codice TeX incluso nei pacchetti d'estensione, anche il codice Lua dovrà rimanere nascosto all'utente finale. Tuttavia un linguaggio potente e facile da usare consentirebbe agli utenti di essere più efficienti nel loro lavoro e ciò gioverebbe all'accuratezza dei documenti.

In questo scenario il paradigma della programmazione ad oggetti potrebbe svolgere un ruolo importante nel contribuire a creare strutture linguistiche più intuitive ed anche più creative per l'utente.

Lua rende possibile ciò poiché offre un supporto di base — e per dirla tutta anche assai poco rigoroso — alla programmazione ad oggetti ma sufficiente per le caratteristiche di un linguaggio interpretato che segue una filosofia minimalista e tende ad essere essenziale in tutto.

Il linguaggio del sistema TeX si evolverebbe per divenire essenzialmente una struttura basata su questi due punti:

- *Paradigma*: linguaggio macro e ad oggetti;
- *Programmazione utente*: sì, in Lua con librerie ad oggetti.

1.1 Una grafica ad oggetti

Per sperimentare un linguaggio ad oggetti indirizzato direttamente agli utenti si è scelto di studiare il caso di una libreria grafica, che ben rappresenta da una parte le difficoltà di programmazione degli utenti e dall'altra il diretto rapporto tra enti geometrici ed oggetti.

Nell'articolo verrano illustrate alcune conoscenze di base sia di Lua sia dei concetti del paradigma di programmazione ad oggetti che introduce nei programmi un più elevato livello d'astrazione.

In seguito verrà presentata la libreria sperimentale GOL, acronimo di *Graphic Object Library* ed il modo in cui è stata implementata attraverso la grafica ad oggetti, assieme alla sintassi in Lua con cui l'ipotetico utente/programmatore svilupperebbe i propri *disegni ad oggetti*.

2 Programmazione ad oggetti

2.1 Basi della OOP

Dal punto di vista dell'utente, il paradigma della Object Oriented Programming si basa sulla manipolazione di entità indipendenti chiamate *oggetti* attraverso funzioni chiamate *metodi*. Ciascun oggetto è caratterizzato dall'insieme dei suoi metodi che ne definiscono il comportamento, il quale è identico per tutti quelli di una stessa famiglia chiamata *classe*, una sorta di prototipo che rappresenta un "tipo di dati".

Nel sorgente l'oggetto dovrà prima essere creato indicando la classe di appartenenza, e successivamente potrà essere manipolato attraverso i *metodi* che agiranno sullo stato interno del singolo oggetto in lettura o in scrittura, fornendo servizi utili.

La creazione di un oggetto avviene con un metodo particolare chiamato *costruttore* che naturalmente può prevedere dei parametri, e che restituisce un riferimento al nuovo elemento che verrà memorizzato in una variabile con un'istruzione di assegnamento, come nel seguente pseudo-codice:

```
NomeTipo var = new NomeClasse(⟨args⟩);
```

Nonostante la variabile sia tecnicamente solo un contenitore, per il programmatore rappresenta invece l'oggetto poiché il nome del riferimento è l'elemento sintattico dopo il quale egli scrive, separandolo con un carattere punto `.`, il nome dei metodi lanciandone così l'esecuzione.

Questa sintassi, nota comunemente come *dot notation* e rappresentata nella prossima linea di codice, riassume l'essenza del paradigma ad oggetti, nel quale dati ed algoritmi sono assemblati in un'unica struttura semantica.

```
var.metodo(⟨args⟩);
```

Le idee di base del paradigma ad oggetti sono concretizzate dai vari linguaggi graduandone le caratteristiche fondamentali: per esempio in Java, linguaggio completamente ad oggetti, il codice per la creazione ed il disegno di un rettangolo grafico in un documento PDF con la libreria iText (Lowagie e Altri, 2012) potrebbe essere simile al seguente frammento di codice:

```
// creazione del documento
Document doc = new Document();

// creazione di un oggetto grafico contenitore
Graphic g = new Graphic();

// aggiungiamo ad esso un rettangolo
g.rectangle(50,50,200,300);

// aggiungiamo la figura al documento
doc.add(g);
```

Nel listato è possibile notare che la creazione di nuove variabili comporta l'obbligo di specificarne il *tipo* premettendone il nome alla variabile. Questa importante caratteristica fa di Java un linguaggio a tipizzazione statica, qualità che ne incrementa la robustezza. Nei linguaggi a tipizzazione dinamica come Lua e Python, invece, la determinazione del tipo è rimandata alla fase di esecuzione sacrificando la robustezza dei programmi a favore della semplicità.

Volendo tradurre in Lua il precedente frammento in Java[1] dovremo sostituire il punto della *dot notation* con il carattere `:` — per il motivo che sarà chiaro in seguito — ed ai doppi backslash `//` i doppi trattini `--` per indicare i commenti di riga. Potremo anche eliminare i simboli di terminazione `;` perché facoltativi in Lua e dovremo eliminare qualsiasi dichiarazione di tipo per ottenere il seguente codice:

```
-- creazione del documento
doc = Document:new()
--... eccetera

-- creazione di un oggetto grafico contenitore
g = Graphic:new()

-- aggiungiamo ad esso un rettangolo
g:rectangle(50,50,200,300)

-- aggiungiamo la figura al documento
doc:add(g)
```

La differenza principale tra il codice Java e la sua traduzione in Lua è che in Lua non esiste la parola chiave `new` che istanzia il nuovo oggetto in memoria chiamandone il costruttore. Ed ancora, in Lua il costruttore risulta essere un metodo qualsiasi dell'oggetto — addirittura non obbligatorio — che lo sviluppatore della libreria potrebbe chiamare in qualsiasi altro modo mentre in Java assume il nome della classe.

Il linguaggio Lua non è quindi progettato con gli stessi obiettivi di Java: non possiede un controllo preventivo del tipo — i relativi errori di programmazione emergono solamente nella fase di esecuzione — non prevede una chiara e semplice modalità di generazione dell'oggetto essendo privo della parola chiave `new`, ma sfrutta funzionalità tortuose poco eleganti e non impone una struttura sintattica unica ma lascia al programmatore la libertà e la responsabilità che ne deriva, di sviluppare soluzioni sintattiche alternative.

Tuttavia, Lua offre pieno supporto ai principi del paradigma ad oggetti senza perdere le caratteristiche generali di un linguaggio semplice da usare.

1. Un buon testo per imparare Java è il seguente (Cornell e Horstmann, 1997) anche se l'edizione italiana risale al 2007.

2.2 Campi e metodi

Nella programmazione OOP un oggetto è un'istanza di classe indipendente, ovvero è caratterizzato da un proprio stato interno determinato dal valore assunto dai *campi*, ovvero riferimenti ad altri oggetti o tipi semplici come numeri e stringhe. I *metodi* sono le funzioni *esposte* all'utente, che possono modificare lo stato interno dell'oggetto e che compiono le elaborazioni a cui è demandata la classe stessa.

L'espressione *la classe espone i metodi* significa che essi costituiscono l'interfaccia con cui l'utente interagisce con l'oggetto. Una volta che l'insieme dei metodi è stato fissato, all'esterno non sarà visibile né lo stato interno dell'oggetto, né l'implementazione stessa dei metodi.

Questo significa che il programmatore della classe, nello sviluppo del codice, ha il solo obbligo di mantenere la sintassi prevista dall'interfaccia *pubblica*, nomi e argomenti dei metodi, ma tutta la libertà di modificare le strutture interne *private*, come il codice dei metodi, potendo così migliorare la libreria senza che gli utilizzatori debbano modificare il proprio codice.

I linguaggi OOP prevedono quindi la possibilità di dichiarare come *privati* campi e metodi. Lua non offre invece alcun meccanismo di protezione, semplicemente contando sul comportamento corretto dell'utente della libreria.

2.3 Ereditarietà

Disponendo di una classe che rappresenti un dato concetto, possiamo implementare allo stesso modo altri concetti legati alla realtà del problema, ma ben presto ci renderemo conto che esiste tra loro una relazione. L'*ereditarietà* è il meccanismo che consente di creare una classe da un'altra estendone le funzioni e modificandone il comportamento, per rappresentare una gerarchia di elementi.

Per esempio, in una libreria di disegno potrebbero essere rappresentate le figure del rettangolo, del cerchio, della linea, ecc., ma ciascun oggetto è caratterizzato da proprietà comuni come lo spessore di linea ed un metodo che lo disegna sulla tela, e dal fatto che tutti sono enti geometrici.

La proprietà comune dello spessore di linea può essere rappresentata da un campo numerico in un oggetto geometrico generico, mentre il metodo per disegnare un oggetto reale dovrà essere implementato di volta in volta a seconda del tipo di geometria rappresentata.

Invece di scrivere sempre lo stesso codice per implementare lo spessore di linea in tutte le classi, forma geometrica per forma geometrica, non conviene ereditare questa proprietà *comune a tutte* le figure da un'unica classe?

Questa classe comune è detta classe padre, o classe base, od ancora *superclasse*. Nel caso particolare dell'esempio è anche una classe *astratta* perché non rappresenta direttamente una forma geometrica.

Tutte le classi geometriche erediteranno le proprietà comuni, definendo il proprio metodo di disegno sempre con lo stesso nome, struttura che sintetizza perfettamente il concetto d'interfaccia.

2.4 Ingegneria del software

Da queste brevi descrizioni del paradigma della programmazione ad oggetti possono essere desunte alcune considerazioni generali riassuntive.

Un oggetto è una sorta di contenitore dove trovano posto sia i dati che le funzioni che li elaborano. Si tratta quindi di un componente software ideato per una più efficiente organizzazione del lavoro di sviluppo dei programmi.

Inoltre, la costruzione ad oggetti introduce la possibilità di creare strutture in grado di rappresentare concettualmente un problema. Si tratta quindi di una metodologia di comprensione e di apprendimento per gradi di sviluppo che consente di costruire su basi solide buone soluzioni software senza che la complessità ne rallenti troppo l'evoluzione.

3 Gli oggetti in Lua

Il linguaggio Lua si fonda sull'essenzialità tanto che supporta la programmazione ad oggetti utilizzando quasi esclusivamente le proprie risorse di base senza introdurre nessun nuovo costrutto. In particolare Lua implementa gli oggetti utilizzando l'unica struttura dati disponibile nel linguaggio, la *tabella*, a cui vengono affiancate particolari funzionalità dette *metatabelle* e *metametodi*.

In questa sezione esamineremo con esempi di codice come viene implementata la programmazione ad oggetti in Lua rimandando al testo di riferimento (Ierusalimschy, 2006) per la trattazione di ulteriori argomenti di cui non si farà cenno, come la gestione della memoria tramite *garbage collector* ed alcuni dettagli sulla sintassi per la definizione delle funzioni e dei suoi parametri.

Cominceremo col descrivere la tabella con le sue particolarità e caratteristiche che la fanno assomigliare ad un oggetto tanto da costituire la base per tutta la programmazione OOP in Lua, per passare sempre più nel dettaglio del linguaggio.

Per la comprensione dei meccanismi OOP di Lua è caldamente consigliato ripetere gli esempi proposti, eventualmente utilizzando la modalità interattiva al terminale (è sufficiente digitare alla linea di comando `lua` per entrarvi) o inserendo il codice in file di testo ed eseguendoli sempre al terminale con il comando `lua nomefile`.

3.1 Le tabelle di Lua

In Lua esiste solo una struttura dati complessa con cui si implementa quello che in altri linguaggi viene espresso con molte diverse entità come gli

array, le *liste*, gli *alberi*, eccetera. Questa struttura è chiamata *tabella*, ovvero un array associativo in cui possono essere inseriti valori attraverso una chiave che può essere un numero o una stringa od anche una funzione.

Se le chiavi sono numeri interi rappresenteremo un array, detto anche vettore, ovvero un insieme di valori ciascuno con il proprio indice, se invece i valori sono tabelle rappresenteremo strutture ad albero, peraltro senza limiti di complessità, se ancora sono stringhe rappresenteremo un dizionario.

Il linguaggio prevede che per creare una tabella si debba usare un costruttore; nella sua forma più semplice questo è una coppia di parentesi graffe `{}`.

Ciascun elemento di una tabella viene restituito specificandone la chiave tra parentesi quadre oppure specificando dopo un carattere punto la chiave come fosse un codice, ma solo se questa è una stringa di caratteri[2]:

```
local mytab = {}          --> nuova tabella in mytab

mytab["primo"] = 1        --> aggiunta di elementi
mytab.secondo = 2

-- ed in 'dot notation':
print(mytab.primo)        --> stampa 1
-- specificando la chiave
print(mytab["secondo"])   --> stampa 2

-- chiave espressa da una variabile
local seckey = "secondo"
print(mytab[seckey])      --> stampa 2
```

Dall'introduzione anche solo così veloce delle tabelle si può intravedere il carattere del linguaggio Lua: semplice ed essenziale ma anche piuttosto potente ed elegante.

Ironia della sorte, in Lua un particolare tipo di costruttore per le tabelle è stato ideato su ispirazione proprio della sintassi dei record di BibTeX, per cui possiamo istanziare la stessa tabella `mytab` dell'esempio precedente con questa compatta sintassi e stamparne poi i dati con un ciclo `for` che funziona con l'iteratore predefinito `pairs()`:

```
-- costruttore compatto
local mytab = {primo = 1, secondo=2}

-- iterazione su tutti gli elementi
for k, v in pairs(mytab) do
   print(k, v)
end
```

La tabella assume il ruolo di vettore con indici interi a partire da 1 — e non da zero come in molti altri linguaggi — se nel costruttore in forma compatta non si specificano chiavi. Ecco un esempio in cui costruiamo una tabella contenente le prime sette lettere dell'alfabeto per poi stamparle in maiuscolo con un ciclo `for` semplice:

2. La parola chiave `local` utilizzata nelle assegnazioni fa in modo che la variabile venga eliminata dalla memoria quando termina il blocco di codice in cui è creata.

```
-- indici:      1   2   3   4   5   6   7
local mytab = {"a","b","c","d","e","f","g"}

-- ciclo semplice
for i = 1,7 do
   print(string.upper(mytab[i]))
end
```

3.2 Funzioni di prima classe

In Lua le funzioni sono strutture di *prima classe*, possiedono cioé la proprietà di poter essere assegnate a variabili che assumeranno il tipo *function*. Ciò rende possibile memorizzare in una tabella anche funzioni e ciò è particolarmente importante non solo per dotare il linguaggio del supporto alla programmazione funzionale, ma anche per implementare facilmente i metodi di una classe nella programmazione ad oggetti.

Come esempio, creeremo una nuova funzione `print()` semplicemente assegnando alla variabile in sovrascrittura, una nuova funzione che fa uso dell'operatore di concatenazione delle stringhe `..` per la stampa personalizzata:

```
-- tipo di una funzione:
print(type(print))  --> stampa 'function'

oldprint = print    --> ref alla funzione originale

print = function (val) -- sintassi funzione anonima
   oldprint("Hello "..val.."!")
end

print("world")  --> stampa 'Hello world!'
print(2012)     --> stampa 'Hello 2012!'
```

Una funzione può essere costruita anche in *sintassi anonima* in cui non compare nessun nome di funzione, costrutto utile quando un argomento è una funzione il cui nome è ininfluente. Questo breve esempio chiarisce i due diversi modi di costruzione delle funzioni in Lua:

```
-- Definizioni di funzioni:
-- sintassi canonica
function stampa(val)
   print('-'..val..'-')
end

-- sintassi anonima della stessa funzione
-- perfettamente equivalente alla precendente
stampa = function (val)
   print('-'..val..'-')
end
```

In realtà, è il compilatore Lua che rende possibile per l'utente la creazione di funzioni per mezzo della sintassi canonica, nella quale il nome della funzione — che non è altro che il nome della variabile che la conterrà — compare tra la parola chiave `function` e le parentesi tonde che racchiudono eventuali argomenti[3].

3. Spesso ci si riferisce a queste funzioni fornite dietro le quinte dal compilatore con l'espressione *zucchero sintattico*.

3.3 Un rettangolo

Avvicinandoci agli obiettivi dell'articolo, costruiremo una classe per rappresentare un rettangolo. Si tratta di un ente geometrico definito da due soli parametri: la larghezza e l'altezza dei lati.

Un primo tentativo potrebbe essere questo:

```
-- prima tentativo di implementazione
-- di una classe rettangolo
Rectangle = {} -- creazione tabella (oggetto)

-- creazione di due campi
Rectangle.width = 12
Rectangle.height = 7

-- un primo metodo assegnato direttamente
-- ad un campo della tabella
function Rectangle.area ()
   -- accesso alla variabile 'Rectangle'
   return Rectangle.larghezza * Rectangle.altezza
end

-- primo test
print(Rectangle.area())    -->  stampa 84, OK
print(Rectangle.height)    -->  stampa  7, OK
```

Ci accorgiamo presto che questa implementazione basata sulle tabelle è difettosa in quanto non rispetta l'indipendenza degli oggetti rispetto al loro nome, ed infatti il prossimo test fallisce:

```
-- ancora la prima implementazione
Rectangle = {width = 12, height = 7}

-- un metodo dell'oggetto
function Rectangle.area ()
   -- accesso alla variabile 'Rectangle' attenzione!
   local l = Rectangle.larghezza
   local a = Rectangle.altezza
   return l * a
end

-- secondo test
r = Rectangle    -- creiamo un secondo riferimento
Rectangle = nil  -- distruggiamo il riferimento originale

print(r.width)      -->  stampa 12, OK
print(r.area())     -->  errore!
```

Il problema sta nel fatto che nel metodo `area()` compare il particolare riferimento alla tabella `Rectangle` che invece deve poter essere qualunque. La soluzione non può che essere l'introduzione del riferimento all'oggetto come parametro esplicito nel metodo stesso, ed è la stessa utilizzata — in modo nascosto ma vedremo che è possibile nascondere il riferimento anche in Lua — dagli altri linguaggi di programmazione che supportano gli oggetti.

Secondo quest'idea dovremo riscrivere il metodo `area()` in questo modo (in Lua il riferimento esplicito all'oggetto *deve* chiamarsi `self` pertanto abituiamoci fin dall'inizio a questa convenzione così da poter generalizzare la validità del codice):

```
-- seconda tentativo
Rettangolo = {larghezza=12, altezza=7}

-- il metodo diviene indipendente dal particolare
-- riferimento all'oggetto:
function Rettangolo.area ( self )
   return self.larghezza * self.altezza
end

-- ed ora il test
myrect = Rettangolo
Rettangolo = nil -- distruggiamo il riferimento

print(myrect.larghezza)      -->  stampa 12, OK
print(myrect.area(myrect))   -->  stampa 84, OK
                             --       funziona!
```

Fino ad ora abbiamo costruito l'oggetto sfruttando le caratteristiche della tabella e la particolarità che consente di assegnare una funzione ad una variabile, ma da questo momento entra in scena l'operatore due punti `:` nella chiamata di funzione. Si tratta di *zucchero sintattico*, ovvero una sorta di aiuto fornito al programmatore dal compilatore, in questo caso per rendere *implicito* il passaggio del riferimento.

L'operatore `:` fa in modo che le seguenti due espressioni siano perfettamente equivalenti, anche se le rende differenti dal punto di vista concettuale agli occhi del programmatore:

```
-- forma classica:
myrect.area(myrect)

-- forma implicita
-- (self prende lo stesso riferimento di myrect)
myrect:area()
```

Questo operatore è il primo nuovo elemento di Lua introdotto per supportare la programmazione orientata agli oggetti. Se si accede ad un metodo memorizzato in una tabella con l'operatore due punti `:` anziché con l'operatore `.`, l'interprete Lua aggiungerà *implicitamente* un primo parametro con il riferimento alla tabella stessa a cui assegnerà il nome di `self`.

3.4 Metatabelle e metametodi

Il salto definitivo nella programmazione OOP consiste nel poter costruire una *classe* senza ogni volta assemblare i campi ed i metodi ma introducendo un qualcosa che faccia da stampo per gli oggetti.

Abbiamo fatto cenno a Java in cui esiste la parola chiave `class` per la definizione di nuove classi, ma in Lua, dove la semplicità è sempre importante e dove devono continuare ad essere supportati più stili di programmazione, non esistono né nuove parole chiave né una sintassi specifica.

In Lua l'unico meccanismo disponibile per compiere questo ultimo importante passo consiste nelle *metatabelle*, normali tabelle contenenti funzioni dai nomi prestabiliti che vengono chiamate quando si verificano particolari eventi come l'esecuzione di un'espressione di somma tra due tabelle con l'operatore `+`. Ogni tabella può essere associata ad una metatabella e questo consente di creare degli insiemi di tabelle che condividono una stessa aritmetica.

I nomi di queste funzioni particolari dette *metametodi* iniziano tutti con un doppio trattino basso, per esempio nel caso della somma sarà richiesta la funzione `__add()` della metatabella associata alle due tabelle addendo, e se non esiste verrà generato un errore.

Il metametodo più semplice di tutti è chiamato `__tostring()` e viene invocato se una tabella è data come argomento alla funzione `print()` per ottenere il valore effettivo da stampare: se ovvio è memorizzato nella metatabella associata altrimenti verrà stampato l'indirizzo di memoria della variabile:

```
-- un numero complesso
complex = {real = 4, imag=-9}
print(complex) --> stampa: 'table: 0x9eb65a8'

-- un qualcosa di piu utile: metatabella in sintassi
-- anonima con il metametodo ___index
mt = {}
mt.__tostring = function (c)
   local r = string.format("%0.2f",c.real)
   if c.imag == 0 then -- numero reale
      return "("..r..")"
   end
   -- numero complesso
   local i = string.format("%0.2f",c.imag)
   return "("..r..","..i..")"
end

-- assegnazione della metatabella mt a complex
setmetatable(complex, mt)

-- riprovo la stampa
print(complex)  --> stampa '(4.00,-9.00)'
```

3.5 Il metametodo `__index()`

Il metametodo che interessa la programmazione ad oggetti in Lua è `__index()`. Esso interviene quando viene chiamato un campo di una tabella che non esiste e che normalmente restituirebbe il valore `nil`, il tipo nullo in Lua. Un esempio di codice chiarirà il meccanismo:

```
-- una tabella con un campo 'a'
-- ma senza un campo 'b'
t = {a = 'Campo A'}

print(t.a)  --> stampa 'Campo A'
print(t.b)  --> stampa 'nil'

-- la metatabella e metametodo
mt = {}
mt.__index = function ()
  return 'Attenzione: campo inesistente!'
end

-- assegniamo 'mt' come metatabella di 't'
setmetatable(t, mt)

-- adesso riproviamo ad accedere al metodo b
print(t.b)  --> stampa 'Attenzione: campo inesistente!'
```

Tornando all'oggetto `Rettangolo`, riscriviamo il codice creando adesso una tabella che assumerà il ruolo concettuale di una vera e propria *classe* definendo i campi ed i metodi:

```
-- una nuova classe Rettangolo (campi):
Rettangolo = {larghezza=10, altezza=10}

-- un metodo:
function Rettangolo:area()
  return self.larghezza * self.altezza
end

-- creazione metametodo
Rettangolo.__index = Rettangolo

-- un nuovo oggetto Rettangolo
r = {}
setmetatable(r, Rettangolo)

print( r.larghezza ) --> stampa 10, Ok
print( r:area() )    --> stampa 100, Ok
```

Queste poche righe di codice racchiudono il meccanismo un po' tortuoso della creazione di una nuova classe in Lua: abbiamo infatti assegnato ad una nuova tabella `r` la metatabella con funzione di classe `Rettangolo`. Quando viene richiesta la stampa del campo `larghezza`, poiché tale campo non esiste nella tabella vuota `r`, viene ricercato il metametodo `__index()` nella metatabella associata che è appunto la tabella `Rettangolo`.

A questo punto, il metametodo restituisce semplicemente la tabella Rettangolo stessa e questo fa sì che divengano disponibili tutti i campi ed i metodi in essa contenuti. Il campo `larghezza` ed il metodo `area()` del nuovo oggetto `r` sono in realtà quelli definiti nella tabella `Rettangolo`.

Se volessimo creare invece un rettangolo assegnando direttamente la dimensione dei lati, dovremo semplicemente crearli con i nomi previsti dalla classe: `larghezza` e `lunghezza`. Il metodo `area()` sarà ancora caricato dalla tabella `Rettangolo` ma i campi numerici con le nuove misure dei lati saranno quelli interni dell'oggetto `r` poiché semplicemente esistono e perciò non sarà presa in causa la metatabella. Questa costruzione funziona perfettamente ma lascia al programmatore una situazione di scomodità che deriva dal progetto stesso di Lua e che può essere in parte sanata con l'introduzione del costrutture, come vedremo meglio a fine sezione.

Le chiamate alle metatabelle ed ai metametodi complicano la comprensione del funzionamento del meccanismo di stampa, compito della classe. Può sembrare che ciò influisca negativamente nella scrittura di programmi ad oggetti in Lua. Tuttavia quello che è importante è comprendere il meccanismo delle metatabelle svolto dietro le quinte, poiché in fase di sviluppo il linguaggio ci apparirà concettualmente simile ad una classe.

Le cose da ricordare di scrivere nel codice sono l'impostazione del metametodo `__index()` con la tabella di classe e l'esecuzione della funzione `setmetatable()` per impostare la tabella dell'oggetto stessa come metatabella.

Riproponendo ancora il problema di rappresentare il concetto di rettangolo, completiamo il quadro introducendo quello che adesso ci appare come un

normale metodo ma che concettualmente assumerà il ruolo di costrutture della classe che chiameremo *sempre* con il nome convenzionale `new()`. Il lavoro che dovrà svolgere sarà quello di inizializzare i campi argomento in una delle tante modalità possibili una volta effettuato l'eventuale controllo di validità degli argomenti.

Il codice completo della classe `Rettangolo` è il seguente:

```
-- nuova classe Rettangolo (campi con valore di default)
Rettangolo = {larghezza = 1, altezza = 1}

-- metametodo
Rettangolo.__index = Rettangolo

-- metodo di classe
function Rettangolo:area()
  return self.larghezza * self.altezza
end

-- costruttore di classe
function Rettangolo:new( o )
   -- creazione nuova tabella
   -- se non ne viene fornita una
   o = o or {}

   -- assegnazione metatabella
   setmetatable(o, self)

   -- restituzione riferimento oggetto
   return o
end

-- codice utente ----------------------
r = Rettangolo:new{larghezza=12,altezza=2}

print(r.larghezza)   --> stampa  12, Ok
print(r:area())      --> stampa  24, Ok

q = Rettangolo:new{larghezza=12}
print(q:area())      --> stampa 12, Ok
```

Il costruttore accetta una tabella come argomento, altrimenti ne crea una vuota e la restituisce non appena le ha assegnato la metatabella. In questo modo una tabella qualsiasi entra a far parte della famiglia `Rettangolo`. Il costruttore passa al metodo `new()` il riferimento implicito a Rettangolo grazie all'uso dell'operatore `:`, per cui `self` punterà a `Rettangolo`.

Quando nel codice viene passata una tabella con uno o due campi sulle misure dei lati al costruttore, l'oggetto disporrà delle misure come valori interni effettivi, cioè dei parametri indipendenti che costituiscono il suo stato interno. Lo sviluppatore può fare anche una diversa scelta, quella di considerare la tabella argomento del costruttore come semplice struttura di chiavi/valori da sottoporre al controllo di validità e poi includere in una nuova tabella con modalità e nomi che riguardano solo l'implemenazione interna della classe.

3.6 Questa volta un cerchio

Per capire ancor meglio i dettagli e renderci conto di come funziona il meccanismo nascosto ed automatico delle metatabelle, costruiamo un oggetto `Cerchio` che annoveri fra i suoi metodi uno che modifichi il valore del raggio aggiungendo una misura specificata:

```
Cerchio = {radius=0}
Cerchio.__index = Cerchio

function Cerchio:area()
   return math.pi*self.radius^2
end

function Cerchio:addToRadius(v)
   self.radius = self.radius + v
end

function Cerchio:__tostring()
   local frmt = 'Sono un cerchio di raggio %0.2f.'
   return string.format(frmt, self.radius)
end

-- il costruttore attende l'eventuale valore del raggio
function Cerchio:new(r)
   local o = {}

   if r then
      o.radius = r
   end

   setmetatable(o, self)
   return o
end

-- codice utente ----------------------
o = Cerchio:new()
print(o)   --> stampa 'Sono un cerchio di raggio 0.00'

o:addToRadius(12.342)

print(o)   --> stampa 'Sono un cerchio di raggio 12.34'
print(o:area())   --> stampa '478.54298786.'
```

Nella sezione del codice utente viene dapprima creato un cerchio senza fornire alcun valore per il raggio. Ciò significa che quando stampiamo il valore del raggio con la successiva istruzione otteniamo 0, il valore di default del raggio dell'oggetto `Cerchio`, per effetto della chiamata ad `__index()` della metatabella.

Fino a questo momento la tabella dell'oggetto `o` non contiene alcun campo `radius`. Cosa succede allora quando viene lanciato il comando `o:addToRadius(12.342)`?

Il metodo `addToRadius()` contiene una sola espressione. Come da regola viene prima valutata la parte a destra ovvero `self.radius + v`. Il primo termine assume il valore previsto in `Cerchio` — quindi zero — grazie al metametodo, e successivamente il risultato della somma uguale all'argomento `v` è memorizzato nel campo `o.radius` che viene creato effettivamente solo in quel momento ed eventualmente utilizzato successivamente in lettura o scrittura.

4 L'idea sulla carta

Dopo questa lunga introduzione al mondo della programmazione ad oggetti possiamo addentrarci nella messa a punto della libreria GOL, scelta come campo di studio e sperimentazione con l'obiettivo di consentire all'utente finale di scrivere codice Lua ad oggetti all'interno di un sorgente LuaLaTeX.

4.1 Struttura del codice

La strategia per l'implememtazione dell'idea di base è questa: durante l'esecuzione il codice Lua rilascia, nel flusso di token per TeX, opportune macro Ti*k*Z corrispondenti alle istruzioni di disegno. Terminata l'esecuzione del codice Lua, il controllo torna a TeX che elabora le macro grafiche esattamente come se le avesse digitate esplicitamente l'utente, producendo effettivamente il disegno.

I passi da fare per generare un disegno sono i seguenti:

- Aprire un ambiente tikzpicture;
- scrivere il codice Lua che rilasci i token dei comandi Ti*k*Z;
- chiudere l'ambiente tikzpicture;

Considerata la disponibilità in LuaTeX della libreria grafica mplib, avremmo potuto più coerentemente scegliere METAPOST per la produzione effettiva del disegno. In questo modo senza la necessità di pacchetti esterni, avremmo potuto gestire la generazione della grafica vettoriale interamente in Lua, beneficiando di migliori prestazioni sia in termini di esecuzione sia in termini di memoria.

Avremmo potuto addirittura creare direttamente le primitive grafiche nel documento di uscita senza avvelerci di alcun pacchetto particolare. La scelta di utilizzare Ti*k*Z come *motore* grafico sottostante può quindi sembrare arbitraria ma rende più semplice la scrittura del codice e crea inoltre un ambiente di sviluppo più adatto alla sperimentazione.

4.2 Il pacchetto GOL

Le operazioni elencate precedentemente possono essere effettuate direttamente dall'utente, tuttavia risulta più coerente mettere a disposizione un ambiente gol all'interno di un pacchetto d'estensione.

Perché all'interno dell'ambiente gol si possa scrivere codice Lua ci si può avvalere del pacchetto luacode. Questo è in grado di gestire opportunamente i codici di categoria dei token, problema originato dai diversi significati che alcuni simboli assumono in TeX ed in Lua. Per esempio, il simbolo percentuale o i backslash (ulteriori informazioni possono essere reperite nella documentazione di questo pacchetto (Pègouriè-Gonnard, 2012)).

Dopo il caricamento dei pacchetti di base il codice prevede quello della libreria GOL, per mezzo della funzione `require()`. Questo tipo di procedura consente di posizionare la libreria in file separati dove è possibile distribuire opportunamente il codice Lua ed evitare i problemi di interpretazione dei simboli, senza comunque compromettere lo sviluppo di ulteriori librerie basate su GOL— magari programmate da contributori esterni — da caricare solo su richiesta dell'utente.

Per ultimo si definisce il nuovo ambiente gol con le macro in stile TeX di apertura e chiusura degli ambienti componenti, tikzpicture e luacode, come mostra il seguente listato completo:

```
% lualatex package
\ProvidesPackage{gol}[2012/08/01 v0.1 Graphic
     Object Library]
\RequirePackage{luacode}
\RequirePackage{tikz}

\directlua{require "gol"}
\newenvironment{gol}
  {\tikzpicture
   \luacode}
  {\endluacode
   \endtikzpicture}
\endinput
```

Un primo sorgente di controllo del pacchetto gol potrebbe essere il seguente, utile oltre che per verificare che tutto funzioni, anche per comprendere operativamente l'implementazione dell'idea di rilascio di token TeX da Lua (occorre compilare il sorgente con LuaLaTeX):

```
% direttive per l'editor TeX works:
% !TEX program = lualatex
% !TEX encoding = UTF-8 Unicode

\documentclass{article}
\usepackage{fontspec}
\usepackage[italian]{babel}
\usepackage{gol}

\begin{document}
\begin{gol}
-- funzione di disegno
-- x,y -> angolo in basso a sinistra
-- b,h -> dimensioni lati base ed altezza
local function rdraw(x,y, b,h)
   local frmt="\\draw (%d,%d) rectangle (%d,%d);"
   tex.sprint(string.format(frmt, x, y, x+b, y+h))
end

-- disegno un quadrato
rdraw(0,0, 2,2)

-- disegno un rettangolo
rdraw(3,0, 2,4)
\end{gol}
\end{document}
```

Nel listato compilabile è definita una funzione chiamata `rdraw()` che *rilascia* i token per TeX utilizzando la funzione di LuaTeX `tex.sprint()`. Il suo scopo è disegnare rettangoli con una particolare logica sulla definizione della geometria: i primi due parametri sono le coordinate assolute del vertice in basso a sinistra del rettangolo e gli ultimi due sono le dimensioni dei lati.

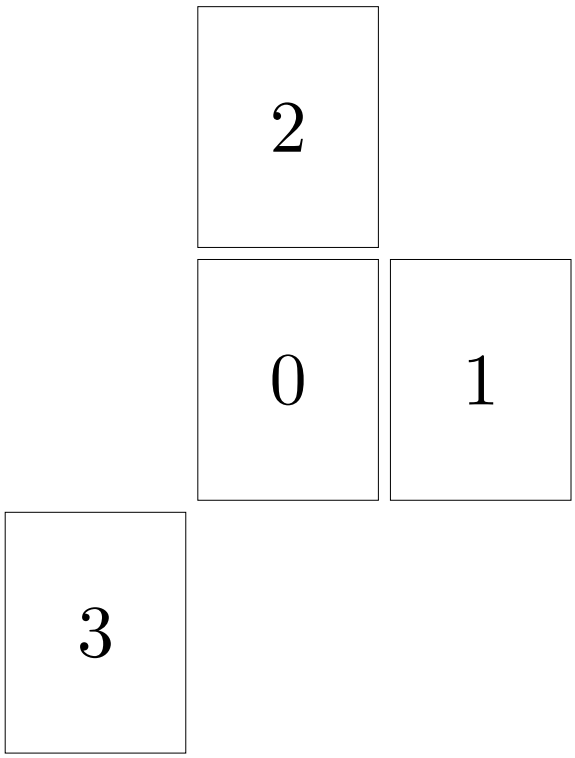


FIGURA 1: Disegno ottenuto con il sorgente LuaLaTeX riportato nel testo, che fa uso di una prima implementazione della libreria GOL a cui è stato aggiunto una numerazione dei rettangoli per facilitare la lettura del codice (immagine scalata rispetto alle dimensioni originali).

Il modello di sorgente LuaLaTeX per l'utilizzo della libreria GOL è il seguente:

```
\documentclass{⟨classname⟩}
\usepackage{fontspec}

...⟨altri pacchetti⟩
\usepackage{gol}
...⟨altri pacchetti⟩

\begin{document}
...
\begin{gol}
   ...⟨codice Lua⟩
\end{gol}
...
\end{document}
```

L'ambiente gol è molto semplice. Non fa nulla se non predisporre la programmazione Lua da parte dell'utente TEX. Il compito di rendere semplice ed efficiente questa attività è l'obiettivo principale della libreria GOL.

4.3 Primo oggetto

Il primo esempio di costruzione grafica è riportato nel seguente listato, il cui risultato è riportato nella figura 1. In esso viene costruito un rettangolo di larghezza 3 ed altezza 4, che viene poi disegnato attraverso il metodo `draw()` dell'oggetto:

```
% !TEX program = lualatex
% !TEX encoding = UTF-8 Unicode
\documentclass{standalone}
\usepackage{fontspec}
\usepackage{gol}

\begin{document}
\begin{gol}
r = Rectangle:new{x=3,y=4}

-- disegno
r:draw()                  --> 0
r:draw{dx=3.2}            --> 1
r:draw{dy=4.2}            --> 2
r:draw{dx=-3.2,dy=-4.2}   --> 3
\end{gol}
\end{document}
```

Già questa prima semplice implementazione introduce un concetto nuovo rispetto al linguaggio tradizionale e già avanzato del pacchetto Ti*k*Z: l'oggetto geometrico deve prima essere creato e solo successivamente può essere disegnato. Questa separazione tra ente geometrico e sua rappresentazione grafica caratterizza il modo in cui l'utente manipola gli oggetti ed il disegno rendendo il processo di costruzione del codice più intuitivo e flessibile.

La sintassi di Lua prevede che se si passa ad una funzione un unico argomento e questo è una stringa o una tabella allora è possibile omettere le parentesi tonde. Per questo nel sorgente di esempio d'uso, essendo il costruttore una normale funzione, può sembrare che si sia fatto uso delle parentesi graffe al posto di quelle tonde. In realtà stiamo passando un'unica tabella nella forma compatta del costruttore `{chiave1=valore1, chiave2=valore2, ...}`.

Il codice Lua che implementa il primo oggetto `Rettangolo` prevede la creazione iniziale di una serie di parametri di identificazione come data e numero di versione, la definizione di un semplice costruttore che crea i due campi *geometrici* relativi alle dimensioni dei lati del rettangolo e del metodo `draw()` che accetta come argomento una tabella opzionale con i campi `dx` e `dy` col significato di vettore di spostamento (δ_x, δ_y).

Il metodo di disegno `draw()` può fornire numerose opzioni relative sia a proprietà grafiche come colore, spessore, tipo linea e riempimento, sia a proprietà geometriche di scalatura, spostamento rispetto al sistema di riferimento locale, e rotazione. Dal lato implementativo non occorre nessuno sforzo di programmazione poiché la gestione di opzioni ⟨*chiave*⟩=⟨*valore*⟩ è insita nella tabella di Lua e nel suo costruttore in forma compatta.

Il codice seguente, prima versione della libreria GOL, dovrà trovarsi nel file di nome `gol.lua` in una delle cartelle di sistema a cui TEX può accedere:

```
-- prima implementazione dell'oggetto rettangolo
Rectangle = {}                  -- init
Rectangle.__index = Rectangle   -- metametodo
Rectangle.classname = 'Rectangle'
Rectangle.version = '0.01'      -- version number
Rectangle.date = '20120530'     -- library date
```

```
-- costruttore
function Rectangle:new(t)
   local o = {}
   o.x = t.x or 1
   o.y = t.y or 1

   setmetatable(o, self) -- metatabella
   return o
end

-- metodo di disegno
-- v -> vettore di spostamento
function Rectangle:draw(v)
   local dx, dy
   if v then
      dx, dy = v.dx or 0, v.dy or 0
   else
      dx, dy = 0, 0
   end
   -- geometria
   local x1 = -self.x/2 + dx
   local y1 = -self.y/2 + dy
   local x2 =  self.x/2 + dx
   local y2 =  self.y/2 + dy

   local frmt = '\\draw (%0.3f,%0.3f) rectangle
     (%0.3f,%0.3f);'
   tex.sprint(string.format(frmt, x1,y1,x2,y2))
end

return Rectangle
```

4.4 Griglie

Il metodo `draw()` degli oggetti grafici si presta bene per assumere funzionalità sofisticate grazie alla sintassi di specifica parametri chiave/valore supportata direttamente da Lua, anche con sottotabelle.

Si sono già incontrate le chiavi `dx` e `dy` per disegnare l'oggetto con uno spostamento nel piano ed è naturale implementare nel metodo di disegno ulteriori modalità di generazione anziché creare nuovi metodi. Per esempio, possiamo pensare di aggiungere un paramento `grid` che stampi l'oggetto su una griglia.

Questo nuovo parametro `grid` potrebbe essere una tabella contenente il numero delle righe e delle colonne, la spaziatura orizzontale e verticale e l'angolo di distorsione come nel seguente esempio il cui risultato è riportano in figura 2:

```
\begin{gol}
r = Rectangle:new{x=.75,y=.75}

-- disegno su griglia
r:draw{grid={rows=3,cols=5,dx=1,dy=2,angle=75}}
\end{gol}
```

Il metodo `draw()` assumerebbe un implementazione simile alla seguente:

```
-- metodo di disegno
-- grid -> tabella con specifiche di griglia
function Rectangle:draw(v)
   local dx, dy
   local nx, ny, sx, sy, a
   if v then
      dx, dy = v.dx or 0, v.dy or 0
```

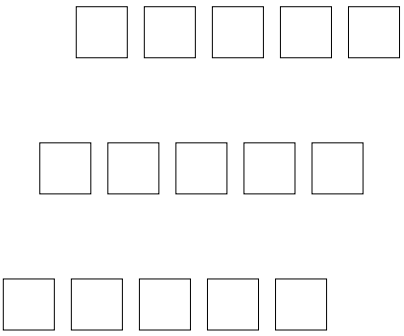


FIGURA 2: Un esempio delle potenzialità di linguaggio del metodo `draw()` di un oggetto grafico, per la stampa su una griglia parametrica (scalato rispetto alle dimensioni originali).

```
      if v.grid then
         ny = v.grid.rows or 1
         nx = v.grid.cols or 1
         sx = v.grid.dx or 1
         sy = v.grid.dy or 1
         a = v.grid.angle or 90
      else
         nx, ny = 1, 1
         sx, sy = 1, 1
         a = 90
      end
   else
      dx, dy = 0, 0
      nx, ny = 1, 1
      sx, sy = 1, 1
      a = 90
   end

   -- geometria
   local x1 = -self.x/2 + dx
   local y1 = -self.y/2 + dy
   local x2 =  self.x/2 + dx
   local y2 =  self.y/2 + dy

   local frmt = '\\draw (%0.3f,%0.3f) rectangle
     (%0.3f,%0.3f);'
   for i=0,nx-1 do
      for j=0,ny-1 do
         local vx = i*sx + j*sy/math.tan(math.pi*a
     /180)
         local vy = j*sy
         tex.sprint(string.format(frmt,
              x1+vx,y1+vy,x2+vx,y2+vy))
      end
   end
end
```

L'opzione `grid` può anche contenere *funzioni* — in virtù del fatto che in Lua le funzioni sono variabili di prima classe — per personalizzare nodo per nodo il disegno in griglia dell'oggetto. Per esempio, è possibile prevedere un campo `selection` a cui associare una funzione che restituisce un valore *vero/falso* secondo i valori del numero di riga e del numero di colonna della griglia. Di essa potremo facilmente disegnare solo i nodi delle diagonali, oppure realizzare uno schema a scacchiera.

L'implementazione del metodo `draw()` dovrebbe semplicemente interrogare nodo per nodo la funzione `selection` prima di effettuare il disegno

dell'oggetto e, nel caso in cui essa non è specificata dall'utente, creare una funzione sostitutiva che restituisce sempre il valore *vero*:

```
-- metodo di disegno
-- grid -> tabella con specifiche di griglia
function Rectangle:draw(v)
   ...
   ... codice iniziale come sopra

   -- check sulla funzione 'selection'
   local isSel
   if v then
      if v.grid then
         isSel = v.grid.selection or
                 function () return true end
      end
   else
      isSel = function () return true end
   end

   local frmt = '\\draw (%0.3f,%0.3f) rectangle
     (%0.3f,%0.3f);'
   for i=0,nx-1 do
      for j=0,ny-1 do
         if isSel(j+1,i+1) then
            local vx = i*sx +
               j*sy/math.tan(math.pi*a/180)
            local vy = j*sy
            tex.sprint(string.format(frmt,
                 x1+vx,y1+vy,x2+vx,y2+vy))
         end
      end
   end
end
```

Per costruire una scacchiera — un buon esempio di applicazione della funzionalità di selezione del disegno nodo per nodo — potremo scrivere un codice simile al seguente:

```
\begin{gol}
local function scacchiera(r,c)
   local s = (r+c)/2 - math.floor((r+c)/2)
   if s==0 then return true else return false end
end

q = Rectangle:new{}

-- disegno con griglia
q:draw{grid={rows=8,cols=8,
             dx=1.1,dy=1.1,
             selection = scacchiera,
            }
      }
\end{gol}
```

L'idea si potrebbe ampliare pensando ad un ulteriore campo funzione dell'opzione `grid`, che modifichi le proprietà dell'oggetto prima del disegno nodo per nodo, in funzione della posizione sulla griglia.

4.5 RefPoint e RefAxis

Uno dei vantaggi riconosciuti del disegno programmato è la facilità con cui gli elementi grafici possono essere affinati modificando dimensioni e proprietà procedendo per passi fino al risultato finale, o per adattare una precedente figura di cui disponiamo già del codice. Nella libreria GOL, oltre a questa essenziale caratteristica, si tenta di semplificare anche la costruzione del disegno relazionando un punto od un asse alla geometria dell'oggetto piuttosto che a coordinate fornite esplicitamente.

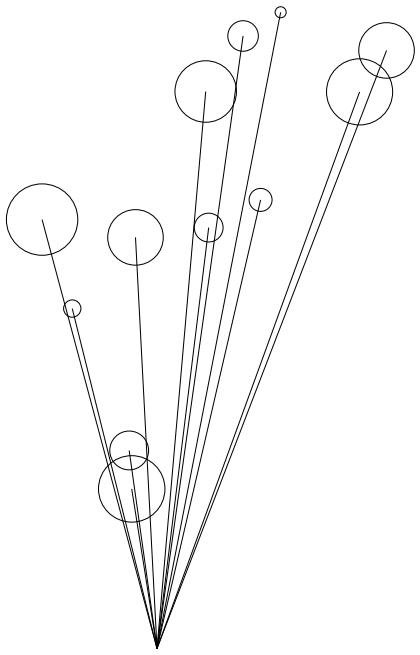


FIGURA 3: Il disegno elementare di un mazzo di fiori formati da un cerchio e da una linea come gambo, di angolo, lunghezza e dimensione casuale, ottenuto con l'utilizzo di un oggetto `RefPoint` studiato per facilitare la costruzione logica e strutturata dei disegni.

Introdurremo cioé *oggetti riferimento*, la cui geometria nel piano dipenderà da una particolare caratteristica di un altro oggetto grafico. In particolare sperimenteremo la classe `RefPoint`, che rappresenterà un punto, e la classe `RefAxis` che rappresenterà un asse.

In questo modo possiamo pensare logicamente al disegno creando dapprima gli elementi principali e successivamente altri elementi ad essi geometricamente vincolati. Per esempio, potremo rappresentare un *fiore* in modo elementare con un cerchio e con una linea che unisce un punto con il centro del cerchio. Una geometria simile si fonda sulla seguente relazione geometrica: la linea del *gambo* ha un estremo coincidente con il centro del cerchio.

Allo scopo di disegnare un quanto mai rudimentale mazzo di fiori, in un ambiente gol potremo scrivere il seguente codice il cui risultato è riportato in figura 3:

```
\begin{gol}
-- disegno di una dozzina di fiori casuale
fiore = Circle:new{r=1}
-- creo l'oggetto 'RefPoint'
centroFiore = fiore:getPoint{'center'}

gambo = Line:new{{0,0}, centroFiore}

-- parametri del mazzo
maxangle = 24
maxdim = 8
```

```
for i=1,12 do
  local alfa = math.random(-maxangle,maxangle)
  local dist = maxdim*math.random()+2

  fiore:setxy{polar={90-alfa, dist}}

  fiore:draw{scale=0.5*math.random()}
  gambo:draw()
end
\end{gol}
```

Nel codice, la relazione geometrica gambo/fiore viene espressa nella definizione dell'oggetto `centroFiore` attraverso il metodo `getPoint()` definito nella classe `Cerchio`. Successivamente viene creata una nuova linea con un vertice nell'origine del sistema di riferimento e con l'altro nel centro del cerchio. Modificando la geometria del cerchio non occorre così riaggiornare di conseguenza le coordinate del secondo estremo della linea. All'interno del ciclo `for`, per disegnare il gambo del fiore è sufficiente chiamare il metodo `draw()` dell'oggetto che lo rappresenta.

Tecnicamente, il metodo `getPoint()` restituisce un oggetto `RefPoint` che, secondo la terminologia della programmazione ad oggetti, dovremo chiamare *metodo factory*, poiché in realtà la classe `RefPoint` non possiede alcun costruttore. Questa è conseguenza diretta della relazione geometrica creata dall'utente: solamente l'oggetto `fiore` conosce la posizione nel piano del proprio centro e può così creare il corrispondente oggetto `RefPoint`.

La struttura del codice Lua che implementa la funzionalità, si basa su semplici funzioni memorizzate all'interno di una tabella che dato il raggio del cerchio, restituiscono le coordinate del punto indicato da una chiave, che può essere per esempio `'n'` per il punto a nord e `'center'` per il centro:

```
Circle.vertexKey = {
   center = function (r) return  0, 0 end,
   n      = function (r) return  0, r end,
   s      = function (r) return  0,-r end,
   e      = function (r) return  r, 0 end,
   w      = function (r) return -r, 0 end,
}
```

Nel metodo vero e proprio, si memorizza in un nuovo campo tabella chiamato `refpoints`, sulla base di un indice progressivo ed in sintassi anonima, la funzione che restituisce le coordinate effettive del punto di riferimento.

Eliminando dal codice i controlli di validità degli argomenti per favorirne la comprensione, l'implementazione del metodo `getPoint()` nella classe `Circle` è la seguente:

```
function Circle:getPoint(t)
   local trp = self.refpoints
   local id = #trp + 1
   local fxy = Circle.vertexKey[t[1]]
   trp[id] = function ()
               local x, y = fxy(self.radius)
               return x+self.x, y+self.y
             end
   -- RefPoint factory
   local rp = {}
   rp.object = self
   rp.id = id

   setmetatable(rp, RefPoint)
   return rp
end
```

L'oggetto `RefPoint` necessita di un campo tabella in cui memorizzare l'oggetto riferito ed un campo numerico dove memorizzare l'indice del punto stesso. In questo modo, possono essere creati un numero qualsiasi di punti riferimento per lo stesso oggetto.

Quando il metodo `draw()` della classe `Line` necessita delle coordinate del punto dei suoi estremi, chiama il metodo `getxy()` di ciascuno di essi. Se si tratta di un `RefPoint` il codice recupererà la funzione di calcolo delle coordinate dall'oggetto riferito che una volta eseguita restituirà le corrette coordinate, per esempio con il seguente codice:

```
function RefPoint:getxy()
   local o = self.object
   local id = self.id
   local fxy = o.refpoints[id]
   return fxy()
end
```

Come è possibile osservare, l'oggetto `RefPoint` non sa niente né di che tipo di oggetto si tratti, né dove sia effettivamente il punto riferito. Per ottenere questa generalità è necessario tuttavia che tutti gli oggetti grafici abbiano un campo `refpoints` contenente le funzioni di calcolo delle coordinate. In termini di paradigma, ciò significa che gli oggetti grafici devono implementare una stessa *interfaccia*.

Anche la classe `Line` — più correttamente si tratterrebbe della rappresentazione di una polilinea e non del più semplice concetto di segmento supposto nel codice — necessita di un campo tabella chiamato `nodes` in cui salvare nell'ordine gli oggetti punto. Ecco una versione essenziale del metodo `draw()`:

```
function Line:draw(v)
   local dx, dy
   if v then
      dx, dy = v.dx or 0, v.dy or 0
   else
      dx, dy = 0, 0
   end
   -- geometria
   local p = self.nodes
   local p1 = p[1]
   local p2 = p[2]
   local x1, y1 = p1:getxy()
   local x2, y2 = p2:getxy()
   x1, y1 = x1 + dx, y1 +dy
   x2, y2 = x2 + dx, y2 +dy

   local frmt =
      '\\draw (%0.3f,%0.3f) -- (%0.3f,%0.3f);'
   tex.sprint(string.format(frmt, x1,y1,x2,y2))
end
```

Interessante è la *flessibilità* del costruttore dell'oggetto `Line` che nel caso del gambo del fiore, deve gestire elementi diversi: una tabella con due numeri come primo argomento, ed un oggetto `Refpoint` come secondo argomento. Il costruttore può essere qualcosa di simile al seguente codice:

```
function Line:new(t)
   local l = {}
   l.nodes = {}

   for i=1,#t do
      local p = t[i]
      if p.classname then
         l.nodes[#l.nodes+1] = p
      else
         l.nodes[#l.nodes+1] = Point:new{x=p[1],
                                         y=p[2]}
      end
   end
   setmetatable(l, self)
   return l
end
```

Il campo `classname` memorizza il nome della classe ed è qui usato per discriminare tra un punto dato come semplice tabella `{`⟨*xcoord*⟩`,` ⟨*ycoord*⟩`}`, ed un oggetto `RefPoint` o `Point` (di cui per brevità non se ne riporta il codice).

Altra particolarità che riguarda l'utente, è che è necessario modificare la posizione dell'oggetto `fiore` perché poi il punto riferimento `centroFiore` possa ottenere correttamente la posizione del centro. Avremo potuto disegnare i cerchi nella posizione generata in modo casuale, sfruttando le opzioni `dx` e `dy` del metodo `draw()` ma l'oggetto `RefPoint` avrebbe puntato sempre nell'origine.

4.6 Assemblaggio di un disegno

Un disegno è un insieme di elementi come linee, tratti curvi, testo, e figure geometriche. Quello che la tecnologia GOL vuole favorire è un processo di costruzione per gruppi più semplici per poi procedere al loro assemblaggio.

Potremo quindi sviluppare una nuova classe `Drawing` che aggrega semplicemente oggetti grafici ed il cui metodo `draw()` chiami uno alla volta i metodi `draw()` degli oggetti dell'insieme.

Tuttavia a ben vedere abbiamo introdotto finora oggetti grafici per rappresentare non disegni bensì elementi geometrici. Una collezione di questi tipi non è un disegno ma piuttosto un insieme di entità geometriche anche correlate tra loro. Sulla base di questo ragionamento, la classe con cui rappresentare il concetto di gruppo di entità è un assemblaggio geometrico e pertanto dovrà fornire metodi per la gestione di questo insieme.

`Assembly` potrebbe essere il nome di questa classe che conterrebbe due tipi diversi di informazioni: la collezione degli oggetti geometrici e, per ciascuno di essi, la definizione di una o più posizioni nel piano. Stiamo quindi pensando non più ad un disegno, ma ad un contenitore in cui possono essere inseriti ed eliminati oggetti geometrici, con un effetto simile a quello dell'installazione e rimozione di un componente software in un sistema.

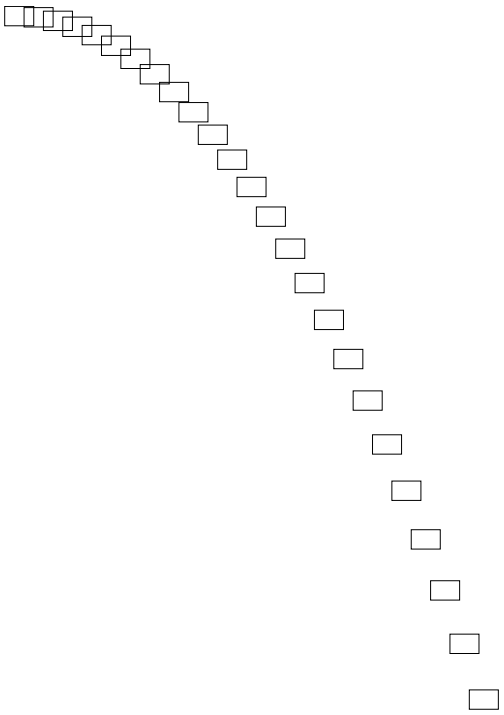


FIGURA 4: Disegno dimostrativo ottenuto utilizzando un oggetto `Assembly` contenente un rettangolo posizionato più volte a simulare il libero spostamento nel campo gravitazionale terrestre (scalato rispetto alle dimensioni originali).

L'idea è quindi questa: disporre di un contenitore in cui caricare oggetti — che nulla vieta siano essi stessi gruppi di elementi — per poi disporlo più volte sul piano, specificando i parametri di posizione, scala, e rotazione. Alle due tipologie di informazione che definiscono un oggetto `Assembly`, ovvero *l'insieme di oggetti* e la loro *distribuzione nel piano*, corrispondono due azioni riguardanti gli elementi del gruppo: *l'installazione* ed il *dispiegamento*.

L'esempio scelto per sperimentare e comprendere operativamente il concetto ideato per rappresentare l'assemblaggio di più oggetti grafici è mostrato in figura 4 dove un unico rettangolo è posto in diversi punti del piano come se fosse stato lanciato nel campo gravitazionale terrestre con una velocità iniziale non nulla, percorrendo una traiettoria parabolica.

Il codice seguente illustra il linguaggio risultato dell'idea di componente: per prima cosa viene creato un oggetto `Rettangolo` chiamato `r`, dopodiché viene installato in un nuovo oggetto `Assembly` attraverso il metodo `install()`.

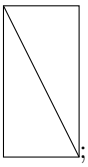
Il dispiegamento multiplo dell'unico oggetto contenuto nel gruppo è effettuato all'interno di un ciclo iterativo in cui sono calcolate le coordinate dell'oggetto attratto dalla gravità in funzione del tempo t a partire dall'istante 0, chiamando il metodo `deploy()`. Per stampare il disegno del gruppo basterà chiamare il metodo `draw()` dell'oggetto `Assembly`:

```
-- creazione rettangolo
r = Rectangle:new{x=1.2,y=0.8}

-- creazione componente
a = Assembly:new()

-- installazione del rettangolo
a:install(r)

-- parametri cinematici
local g = 9.80665    -- m/s^2
local vx, vy = 8, 0 -- m/s

for t=0, 2.5, 0.10 do
   -- calcolo coordinate
   local x = t*vx
   local y = -0.5*g*t^2+t*vy
   -- dispiegamento oggetto
   a:deploy(r, {x=x,y=y})
end

-- disegno del componente
a:draw()
```

L'implementazione della classe `Assembly` è piuttosto semplice grazie alle caratteristiche della tabella di Lua: possiamo infatti utilizzare riferimenti di tabella come chiave di campi valore. Si prevede di utilizzare una tabella chiamata `content` interna all'oggetto le cui chiavi sono i riferimenti agli oggetti grafici installabili nel componente. A queste chiavi riferimento si assocerà una tabella che a sua volta contiene la lista di tabelle ciascuna relativa alle informazioni di dispiegamento (per brevità, nel codice sono stati omessi i controlli di validità):

```
-- oggetto Assembly
Assembly = {}                    -- init
Assembly.classname = 'Assembly'
Assembly.__index = Assembly      -- metametodo
Assembly.version = '0.01'        -- version number
Assembly.date = '20120915'       -- library date

-- costruttore
function Assembly:new()
   local a = {}
    -- contenitore per oggetti
    -- e loro disposizione
   a.content = {}

   setmetatable(a, self)
   return a
end

-- caricamento oggetto
function Assembly:install(o)
   if not self.content[o] then
      self.content[o] = {}
   end
end

-- informazione di posizione
-- o : a graphic object
-- tdeploy : tables list of {x, y, angle}
function Assembly:deploy(o, tdeploy)
   if self.content[o] then
      local tpos = self.content[o]
      tpos[#tpos+1] = tdeploy
   end
end
```

In questo momento il rettangolo appare così: 

Ma può diventare anche più grande:  ;

...e di nuovo piccolo:  .

FIGURA 5: Dimostrazione che gli oggetti grafici, una volta creati come riferimento globale, possono essere riutilizzati più volte ed ovunque nel documento, chiamandone i metodi desiderati in un ambiente `gol`.

```
-- metodo di disegno
function Assembly:draw()
   for o, tdeploy in pairs(self.content) do
      for _, pos in pairs(tdep) do
         local x, y = pos.x, pos.y
         local a = pos.angle
         o:draw{dx = x, dy = y, angle = a}
      end
   end
end
```

4.7 Riuso delle figure

Gli oggetti creati come variabili globali, senza cioè che sia premessa nell'assegnazione la parola chiave `local`, rimangono nella memoria e quindi possono essere riutilizzati in altre posizioni del sorgente, così anche per gli oggetti GOL creati nel documento ed è quindi possibile chiamarne i metodi.

Nel successivo esempio un rettangolo e una delle sue diagonali vengono create all'inizio del documento per poi essere disegnate più volte modificandone le dimensioni, come dimostra la figura 5.

Nel codice dell'esempio, riportato di seguito, si può notare come la modifica delle dimensioni del rettangolo `x` ed `y` avviene in modo diretto sui campi dell'oggetto in forma di *dot notation* e, da quanto detto precedentemente sui principi della programmazione ad oggetti, ciò non risulta corretto. Infatti, nello sviluppo della libreria potrebbero cambiare i nomi delle variabili in cui sono memorizzate le dimensioni del rettangolo, o potrebbe cambiare il tipo di dato che le rappresenta.

Lo sviluppatore è tenuto a mantenere immutata solo la definizione dell'interfaccia. Nel caso descritto, il metodo `setxy()` continuerebbe comunque a funzionare per la modifica delle dimensioni del rettangolo, ma non è detto che valga la stessa cosa per l'assegnazione diretta di larghezza e altezza.

In sintesi, non accedere direttamente ai campi interni dell'oggetto è una buona regola che può essere violata in Lua, ma il codice mostra come sarebbe concettualmente più semplice e diretto agire su di essi, tanto che questi campi potrebbero rientrare nell'interfaccia dell'oggetto anziché rimanere pseudo-privati. Una scelta importante questa perché coinvolge i concetti base del linguaggio.

```
\begin{gol}
r = Rectangle:new{x=0.25,y=0.5}
a = r:getPoint{vertex='nw'}
b = r:getPoint{vertex='se'}

l = Line:new{a,b}
\end{gol}

In questo momento il rettangolo appare così:
\begin{gol}
r:draw()
l:draw()
\end{gol}

Ma può diventare anche più grande:
\begin{gol}
r.x, r.y = 1, 2

r:draw()
l:draw()
\end{gol}
;

\dots e di nuovo piccolo:
\begin{gol}
r.x, r.y = 1, 0.5

r:draw()
l:draw()
\end{gol}.
```

4.8 Il codice nascosto

Il codice Ti*k*Z che viene *iniettato* in TEX dai metodi `draw()` degli oggetti grafici creati dall'utente, è completamente inaccessibile, ma non è difficile modificare questo comportamento memorizzando in una tabella il codice nascosto prodotto man mano per poi salvarlo in un file esterno.

Questo aspetto mette in luce come il progetto della liberia sia poco modulare. Occorrerebbe scindere il codice nascosto dai metodi degli oggetti perché questo renderebbe più semplice lo sviluppo ed allo stesso tempo consentirebbe di aggiungere ulteriori motori grafici intercambiabili. Potremmo così ottenere il codice nascosto anche in formato METAPOST per esempio, od anche in formato vettoriale `dxf` (AUTODESK, 2011), formato usato nei disegni CAD.

4.9 Nuovi pacchetti

Nella sezione precedente si è accennato ad una struttura modulare più razionale per la libreria GOL per separare il codice degli oggetti geometrici da quello effettivamente necessario per disegnarli secondo un formato grafico sottostante, facendo così intravedere uno dei punti più importanti dell'introduzione di LuaTEX: la possibilità di sviluppare progetti dalle dimensioni molto superiori a quelli limitati all'uso del solo motore tipografico TEX.

Così, la libreria GOL potrebbe favorire la scrittura di librerie specifiche da parte di contributori esterni allo sviluppo di base, proprio come avviene per il pacchetto PGF, fornendo agli oggetti metodi per caricare in essi nuove funzioni e l'infrastruttura per costruire disegni complessi pronti per essere utilizzati e ricchi di proprietà grafiche modificabili dall'utente.

5 Ereditarietà

Il meccanismo dell'ereditarietà, caratteristica essenziale della programmazione ad oggetti, consiste nel creare una nuova classe a partire da una già esistente per estenderne e modificarne le funzionalità. Con la classe padre si rappresenta un concetto più generale, che è reso più specifico e dettagliato dalla classe figlia.

Ogni processo di ereditarietà crea una nuova relazione tra concetti per costruire l'intera struttura che rappresenta un particolare problema reale. Nel nostro caso l'ereditarietà può essere impiegata in tre contesti diversi: quello dello sviluppo della libreria in cui opera il programmatore, quello dello sviluppo di pacchetti d'estensione, e quello del singolo documento TEX in cui opera l'utente finale.

La libreria GOL potrebbe implementare una struttura di relazioni tra oggetti grafici a partire da una classe base ottenendo maggiore chiarezza del codice e migliore possibilità di sviluppo.

5.1 Livelli d'astrazione

Tutti gli oggetti grafici fanno uso di un sistema di riferimento locale, in particolare delle coordinate dell'origine chiamate *punto di ancoraggio*, e di alcune proprietà di disegno come il colore del tratto e del riempimento, il tipo di linea e il suo spessore. Così è naturale pensare di creare una classe base da cui ereditare per costruire tutti gli oggetti grafici reali come cerchi e linee.

Questa classe base, detta `Element`, disporrà di un metodo per la modifica delle coordinate del punto di ancoraggio e di alcuni metodi d'interfaccia privi d'implementazione che obbligatoriamente le classi derivate devono definire, come per esempio il metodo `draw()`. Lua non dispone del supporto per la creazione di questi *metodi virtuali*. Possiamo solamente inserire nei metodi dell'interfaccia un'istruzione di errore che interrompe l'esecuzione e la compilazione del documento avvertendo che il metodo reale non è stato correttamente definito nella classe derivata.

Quanto al costruttore, la classe base non ne può disporre poiché non si tratta di un oggetto reale. Valgono per esso le stesse considerazioni fatte precedentemente per qualsiasi altro metodo virtuale della classe.

Lua non dispone nemmeno di un supporto esplicito all'ereditarietà che può quindi essere implementata in diversi modi. Quello più semplice consiste nell'indurre il compilatore a percorrere all'indietro la gerarchia delle classi nel tentativo di ricerca-

re il componente chiamato attraverso la funzione `__index()` e le metatabelle.

Per esemplificare il comportamento del compilatore si consideri la classe `Rectangle` che questa volta estende la classe base `Element` dotata di un metodo `setFillColor()` per impostare il colore di riempimento degli oggetti grafici:

```
r = Rectangle:new{}
r:setFillColor('orange')
```

Poiché il metodo `setFillColor()` non è definito in `Rectangle`, Lua cerca il metodo `__index()` nella metatabella associata ad `r` — il riferimento a `Rectangle` stesso. Il metodo non è presente pertanto si ricade nella stessa situazione: Lua cerca la metatabella associata a `Rectangle` che questa volta è un riferimento ad `Element`, così in conclusione viene eseguito il metodo `setFillColor()` della superclasse con un primo parametro `self` che fa riferimento ad `r` ed un secondo argomento contenente il valore del colore.

L'effetto di questa ricerca a cascata è quello per cui la classe derivata *eredita* tutti i campi ed i metodi della classe base. In definitiva, ciò che accade nel codice precedente è la creazione — od eventualmente la modifica — del campo interno ad `r` che memorizza il colore di riempimento attraverso il metodo implementato nella classe base. Per convincerci di questo è sufficiente leggere il codice del metodo `setFillColor()` ricordando gli argomenti che esso riceve — `self` ovvero `r` ed `'orange'` — al momento dell'esecuzione:

```
function Element:setFillColor(col)
   if col then self.fillcol = col end
end
```

In generale, quando un metodo di un oggetto richiede la lettura di un campo valore o di un campo funzione della superclasse, la struttura della tabella che lo implementa non viene modificata mentre se l'oggetto richiede la scrittura degli stessi campi, la prima volta che questo accade sarà creata una copia locale che da quel momento diviene il nuovo campo interno all'oggetto in lettura/scrittura.

Dal punto di vista dell'utente della libreria ad oggetti, l'effetto della scrittura di un campo valore viene percepito semplicemente come la modifica di un elemento che concettualmente fa già parte dello stato dell'oggetto. Invece, la scrittura di un campo metodo è intesa come applicazione del *polimorfismo*, modifica il comportamento dell'oggetto con la sostituzione del metodo ereditato dalla superclasse.

In altre parole, in relazione alla modifica di campi e di metodi di una classe derivata, lo sviluppatore Lua osserva una stessa identica procedura seguita dal compilatore, mentre lo sviluppatore che adotta il paradigma ad oggetti sperimenta per essi concetti profondamente diversi.

Nella figura 6 è riporato un disegno di verifica della correttezza della nuova implementazione che fa derivare la classe `Rectangle` dalla classe

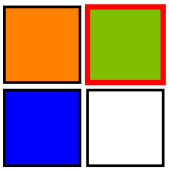


FIGURA 6: Un primo esempio per la verifica del corretto funzionamento dell'oggetto `Rectangle`, ottenuto come estensione di una classe base con il meccanismo dell'ereditarietà. Tutti gli oggetti derivati da essa condividono un insieme di funzionalità comuni senza bisogno di riscrivere codice.

`Element`. Nel relativo codice riportato di seguito vengono istanziati due oggetti `Rectangle` modificandone le proprietà di colore. La verifica ha esito positivo poiché ciascun oggetto mantiene correttamente i propri parametri perfettamente separati da quelli degli altri.

```
\begin{gol}
r = Rectangle:new{}

r:setFillColor('blue')
r:draw()

q = Rectangle:new{}
q:draw{dx=1.1}
q:setFillColor('orange')
q:draw{dy=1.1}

q:setLinewidth(2)
q:setFillColor('orange!50!green')
q:setLineColor('red')
q:draw{dx=1.1, dy=1.1}
\end{gol}
```

Gli elementi essenziali dell'implementazione sono riportati nei seguenti frammenti di codice. La classe `Element` non ha un costruttore ed è priva anche di un metodo `draw()` — che dovrebbe essere inteso come un metodo virtuale — quando invece crea una serie di campi che saranno comuni a tutti gli oggetti delle classi derivate da essa. Si noti come una delle prime assegnazioni riguardi il metodo `__index()` valorizzato con il riferimento alla stessa tabella `Element`, istruzione questa indispensabile per il corretto funzionamento della ricerca dei metodi da ereditare:

```
-- oggetto base 'Element'
Element = {}                    -- init
Element.__index = Element       -- metametodo

Element.classname = 'Element'-- class id
Element.version = '0.01'        -- version number
Element.date = '20120917'       -- library date

-- campi che saranno comuni a tutte le classi derivate
-- :coordiante del punto di ancoraggio
Element.x = 0
Element.y = 0

-- :proprietà grafiche
Element.lw = 1 -- pt
Element.linecol = nil -- means 'black'
Element.fillcol = nil -- means 'trasparent'
```

```
-- metodo d'impostazione dell' 'origine locale' oggetto
-- :x,y -> absolute coordinates
-- :dx, dy -> relative coordinates
function Element:setAnchor(t)
  if t then
    if t.x then
      if t.dx then error("Mix mode not allowed.")
     end
      self.x = t.x
    end
    if t.y then
      if t.dy then error("Mix mode not allowed.")
     end
      self.y = t.y
    end
      if t.dx then self.x = self.x + t.dx end
      if t.dy then self.y = self.y + t.dy end
    end
end

-- set property functions
function Element:setLinewidth(lw)
   if lw then self.lw = lw end
end
function Element:setLineColor(col)
   if col then self.linecol = col end
end
function Element:setFillColor(col)
   if col then self.fillcol = col end
end
```

Le righe di codice indispensabili per comprendere il meccanismo dovuto alle metatabelle sono le seguenti, riguardanti l'implementazione della classe derivata `Rectangle`. Le istruzioni fondamentali che vi compaiono sono l'assegnazione di `Element` come metatabella di `Rectangle` e la valorizzazione del metodo `__index()` con il riferimento alla tabella stessa. Nel costruttore è necessario impostare anche la metatabella degli oggetti istanza per completare le informazioni necessarie al compilatore Lua ed implementare così la derivazione:

```
-- oggetto derivato 'Rectangle'
Rectangle = {}                    -- init

-- metatabella classe figlia = superclasse
setmetatable(Rectangle, Element)
Rectangle.__index = Rectangle
Rectangle.classname = 'Rectangle'
Rectangle.version = '0.04'        -- version number
Rectangle.date = '20120917'       -- version date

-- costruttore
function Rectangle:new(t)
   local o = {}
   if t then
      o.xdim = t.x or 1
      o.ydim = t.y or 1
   else
      o.xdim, o.ydim = 1, 1
   end
   o.refpoints = {}

   setmetatable(o, self)
   return o
end

...
<campi e metodi della classe derivata>
```

5.2 Creare nuovi oggetti

Nulla vieta che anche l'utente possa creare nuovi oggetti ereditando da classi esistenti. Dovrà tuttavia rispettare l'obbligo di implementare alcuni metodi fondamentali, caretteristica propria dell'oggetto grafico che egli intende costruire.

Per facilitare la specializzazione degli oggetti già disponibili, la libreria GOL dovrebbe mettere a disposizione un metodo che si occupi dei dettagli come l'assegnazione delle metatabelle e dei metametodi.

6 Disegno parametrico

Il *disegno parametrico* è una funzionalità potente, ma complessa da utilizzare, che fa dipendere la geometria e l'aspetto di un disegno da valori indipendenti e modificabili. Questi parametri possono essere grandezze relative alla geometria di oggetti già esistenti come la lunghezza di un lato o quella del diametro di un cerchio.

Avvalersi del disegno parametrico significa poter *vincolare* le caratteristiche di un'entità a quelle di altre, per rappresentare relazioni geometriche e mettere in pratica i concetti sottostanti alla costruzione grafica. Risulterà molto efficiente, una volta stabilito il vincolo, modificare il disegno solamente nelle proprietà indipendenti.

Per esempio, è possibile vincolare il lato di un rettangolo ad essere uguale all'altro, creando un quadrato, oppure è possibile far dipendere la misura di un diametro da un parametro numerico, od ancora regolare un angolo ad assumere sempre un determinato valore.

Essendo scritta in Lua, la libreria GOL può effettivamente tentare di offrire funzionalità di disegno parametrico attraverso una nuova classe denominata `Param` che memorizza il valore del parametro e lo rende vincolo per gli oggetti grafici.

Dove per un argomento ci si aspetta una misura si può invece fornire un oggetto `Param`. Nel codice si potrebbe aggiungere un controllo sul tipo oppure trasformare ogni campo in una funzione che restituisce il valore oppure la corrispondente funzione di interrogazione del parametro.

7 Organizzare il disegno

Allo stesso modo dei disegni CAD ed anche di Ti*k*Z, il potente pacchetto per la grafica vettoriale in TEX, è possibile organizzare gli elementi grafici attribuendoli ad un dato insieme chiamato *layer*. L'idea assomiglia al modo in cui il disegnatore sovrappone tanti fogli trasparenti ciascuno con diverse porzioni grafiche, per comporre il disegno complessivo.

Un layer può essere *spento* così che i suoi oggetti non compariranno più nel disegno — per esempio per ottenere una stampa ad una scala più piccola eliminando i dettagli — ed essere caratterizzato

da uno *stile* ovvero le proprietà di colore, spessore e tipo di linea, che gli oggetti del layer possono condividere.

Cambiando le proprietà del layer cambiano di conseguenza le corrispondenti proprietà di tutti gli oggetti in esso contenuti, salvo che questi ne abbiano di proprie. Per esempio, il colore di tracciamento di un oggetto potrebbe assumere il valore di `DaLayer` assumendo quindi il parametro stabilito per il layer oppure, pur appartenendo al layer, avere una proprietà di colore persoanlizzata.

Fino ad ora l'elemento `Layer` ha due principali caratteristiche: memorizza *proprietà* grafiche per gli oggetti ad esso assegnati e risulta che un oggetto può appartenere ad un unico layer. Ciò significa che i layer adottano una struttura ad un unico livello.

Conseguenza della prima caratteristica è che l'elemento `Layer` non è un oggetto e pertanto apparirà all'utente come una classe non instanziabile dotata unicamente di *metodi statici*. Nella sintassi del linguaggio Lua significa che i metodi verranno chiamati con la notazione punto, e non compariranno in istruzioni di assegnazione ma in espressioni semplici, come nel seguente listato frammento di un ambiente gol:

```
-- creazione di un nuovo layer
-- con un metodo statico
Layer.add{name      = '20',
          linecolor = 'blue!50',
          fillcolor = 'yellow',
          linewidth = 0.2,
          linetype  = 'dashed',
}

-- spegnimento di un layer
Layer.off{'30'}

-- cancellazione di un layer
Layer.delete{'details'}
```

La seconda caratteristica porta invece ad estendere il concetto base di *layering* dei programmi CAD, in una struttura a più livelli strutturata ad albero: un layer può contenere altri sotto-layer. Un oggetto raggruppato in un layer può essere influenzato dalle proprietà associate ad un layer di livello superiore. Il livello di complessità organizzativa del disegno in layer a più livelli rimane ad esclusiva scelta dell'utente che può ignorare completamente a quale layer appartengono gli oggetti, o può invece creare un unico livello, oppure ancora può avvalersi di livelli multipli di organizzazione.

8 Conclusioni

La libreria grafica GOL descritta nell'articolo e sviluppata per sperimentare la programmazione ad oggetti da parte degli utenti finali del sistema TEX dimostra che è possibile progettare linguaggi con elevata intuitività grazie alla rappresentazione diretta di concetti.

Il linguaggio separa la definizione delle entità geometriche dal loro effettivo disegno consentendo all'utente di modificarne le proprietà a piacimento, di avvalersi di sofisticati metodi di costruzione e di riutilizzare i disegni nel documento.

Naturalmente è possibile applicare la stessa idea in molti altri settori della tipografia asincrona con TEX. Si pensi alla generazione di report in forma di tabelle compresi quelli dalla struttura molto complessa e con informazioni residenti in database esterni o memorizzati in file nei formati più vari, oppure ad un sistema di controllo dei parametri tipografici del documento.

L'evoluzione del sistema TEX continua in modo inaspettato ed originale per consentire di raggiungere un più elevato numero di utenti e migliori risultati tipografici. LATEX3, ConTEXt-`mkiv`, LuaLATEX e nuovi pacchetti basati su Lua prospettano un panorama applicativo quanto mai vasto ed in frenetico sviluppo.

Tuttavia non è mai facile concretizzare le idee e le potenzialità perché il linguaggio dovrà apparire all'utente il più possibile coerente ed equilibrato e garantire nel tempo il rapido e sicuro apprendimento da parte degli utenti ed il non meno importante fattore evolutivo delle applicazioni.

L'idea che l'utente scriva codice ad oggetti in un sorgente TEX è una recente possibilità dovuta al nuovo motore LuaTEX. Altrettanti recenti sviluppi dei linguaggi di programmazione orientati per un mondo *completamente* basato sulla rete internet — si pensi ad esempio ai sistemi operativi basati su tecnologie *cloud* ed al linguaggio Go di Google (Griesemer *et al.*, 2012) — smorzano un po' l'entusiasmo per la programmazione ad oggetti. La complessità di progettazione della OOP e la riduzione dell'efficienza in fase di esecuzione del codice sono forse i motivi principali che hanno fatto sì che questo paradigma non abbia raggiunto pienamente tutti gli obiettivi iniziali.

Si tratta del fondamentale processo di evoluzione indotto da nuove esigenze derivate a sua volta dallo spostamento delle attività di elaborazione dati dalla tradizionale *scrivania* ad un *luogo non fisico*, che nello sviluppo tende a semplificare le gerarchie nelle strutture dei linguaggi ad oggetti ed a favorire il raggiungimento delle massime prestazioni sfruttando processori a più core con l'esecuzione parallela e contemporaneamente a favorire l'aumento della produttività degli sviluppatori.

Nel mondo TEX, dove il linguaggio è lo strumento essenziale degli utenti, è sempre più importante affiancare ad una elevata qualità tipografica un'elevata produttività degli utilizzatori. La programmazione ad oggetti è un sicuro tema da esplorare per raggiungerla.

9 Ringraziamenti

Ringrazio il revisore anonimo e Gianluca Pignalberi direttore di ArsTEXnica che hanno fatto pienamente parte della *squadra* che ha portato questo articolo fino alla conclusione, squadra di cui fa certamente parte anche Agostino De Marco che ha più volte contribuito al testo con suggerimenti e correzioni e che è sicuramente pronto per il nostro prossimo lavoro. Grazie a tutti.

Ringrazio in modo particolare anche la mia famiglia (lo devo fare... altrimenti sono guai) perché mi sostiene nella scoperta di nuovi paesaggi del mondo TEX.

Riferimenti bibliografici

Autodesk (2011). *DXF Reference*. Autodesk Inc.

Cornell, G. e Horstmann, C. S. (1997). *Core Java 2 Fondamenti*. Professionale. Pearson Education Italia, 7ª edizione.

Griesemer, R., Pike, R. e Thompson, K. (2012). «Go programming language». Sito web ufficiale `http://golang.org/`.

Ierusalimschy, R. (2006). *Programming in Lua*. Lua.org, 2ª edizione.

Lowagie, B. e Altri (2012). «Libreria PDF in Java o C#». Sito web `http://itextpdf.com/`.

Pègouriè-Gonnard, M. (2012). «The luacode package version 1.2a». Disponibile sul sito web `www.ctan.org`.

Tantau, T. (2010). «pgf manual version 2.10». Disponibile su `www.ctan.org`.

▷ Roberto Giacomelli
Carrara (MS)
`giaconet dot mailbox at gmail dot com`