

Développement d'un langage de script pour des animations graphiques simples

Alain Plantec

On désire développer un prototype de moniteur pour un système de commande pour des éléments représentés graphiquement. Dans sa version minimale, un élément visuel est une forme géométrique basique (rectangle, ellipse, ligne, texte...). Le moniteur permet de les installer dans un conteneur et de les animer.

Le développement se fait en Java sous Eclipse. Le *workspace* **doit** être nommé de la façon présentée dans l'exemple suivant :

Pour le groupe des quatre membres suivants :

1. Mickaël Kerboeuf
2. Frank Singhoff
3. Laurent Lemarchand
4. Plantec Alain

le nom du *workspace eclipse* est :

Kerboeuf_Mickael__Singhoff_Frank__Lemarchand_Laurent__Plantec_Alain

Quand vous créez ce workspace, vous devez avoir un répertoire portant ce nom. C'est ce répertoire que vous rendez sur le moodle après l'avoir compressé.

Donc, le groupe exemple ci-dessus livre le fichier

Kerboeuf_Mickael__Singhoff_Frank__Lemarchand_Laurent__Plantec_Alain.zip.

La suite du sujet se constitue d'un ensemble d'exercices de difficulté croissante. Vous devez programmer systématiquement votre solution pour chaque exercice dans un package dédié. Pour cela, il faut donc systématiquement créer un nouveau package sous Eclipse pour chaque exercice. Par exemple, pour l'exercice 1, vous créez un package Java nommé *exercice1*.

Exercice 1 : Prise en main de la couche graphique

Vous devez obligatoirement utiliser le package *graphicLayer* du projet Eclipse 2DGraphicCore à prendre sur le Moodle. Cet exercice a pour objectif de vous familiariser avec cette couche graphique 2D simple. Regardez les exemples du package *graphicLayer.demos* puis programmez l'animation suivante : un robi (rectangle bleu) se déplace en suivant les bords intérieurs de la fenêtre.

Vous devez donc tout d'abord créer une fenêtre contenant un rectangle bleu en position (0 0). Pour cela vous devez créer un *GSpace* dans lequel vous allez ajouter un *GRect*. Voici le squelette qui met en place cette fenêtre avec le package *graphicLayer*.

```
import java.awt.Dimension;
import graphicLayer.GRect;
import graphicLayer.GSpace;

public class Exercice1_0 {
    GSpace space = new GSpace("Exercice 1", new Dimension(200, 150));
    GRect robi = new GRect();

    public Exercice1_0() {
        space.addElement(robi);
        space.open();
    }
}
```

```

    public static void main(String[] args) {
        new Exercice1_0();
    }
}

```

En partant de ce squelette, programmez l'animation suivante (après `space.open()`) :

- Déplacement de *robi* jusqu'au bord droit
- Déplacement jusqu'au bord bas
- Déplacement jusqu'au bord gauche
- Déplacement jusqu'au bord haut
- Changement de couleur de *robi* avec une couleur aléatoire

Le déplacement doit s'adapter automatiquement si vous redimensionnez le space (la fenêtre principale).

Exercice 2 : Première version d'un interpréteur de script

Dans cette première version, on considère un environnement composé d'un seul conteneur global, une instance de *GSpace* nommée *space*. Ce conteneur ne contient qu'un seul élément, une instance de *GRect* nommée *robi*.

On veut interpréter des scripts pour changer les propriétés de *space* et de *robi* (dimension, couleur, épaisseur du bord...) et pour animer *robi* en changeant sa position dans *space*.

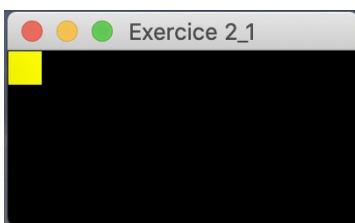
Un script est représenté comme une expression parenthésée appelée S-expression. C'est la syntaxe utilisée pour le Lisp ou Scheme par exemple. Le lecteur de S-expressions est fourni (à télécharger depuis Moodle).

2.1 Script de configuration

Ce script permet de modifier la couleur du conteneur principal nommé *space* et celle de *robi*. Ce script comprend deux S-expressions.

```
(space color black) (robi color yellow)
```

Voici une copie d'écran montrant le résultat :



La classe *SParser* permet de compiler des S-expressions (chaîne de caractères). La compilation d'un script produit une instance de *SNode*. Une instance de *SNode* peut être soit une feuille, soit un nœud intermédiaire :

- Un *SNode* feuille a un contenu récupérable sous la forme d'une chaîne de caractères
- Un *SNode* nœud intermédiaire se comporte comme une collection de *SNode*.

La classe *SParser* est disponible dans le package *stree.parser* que vous devez prendre sur le Moodle (projet *SParser*). Regardez les classes de test pour vous aider à comprendre comment s'en servir.

Voici un modele de script :

(S-expression 1) (S-expression 2) ... (S-expression n)

Un script est une liste de S-expression. Chaque S-expression représente une commande pour par exemple un changement de couleur de space ou de robi.

Voici un squelette de solution (disponible sur le Moodle). La fonction `run` analyse le script et construit une instance de `List<SNode>` (`rootNodes`) qui représente la liste des S-expression racines du script. Vous devez compléter la fonction `run` qui interprète les S-expression de `rootNodes`.

```
public class Exercice2_1_0 {
    GSpace space = new GSpace("Exercice 2_1", new Dimension(200, 100));
    GRect robi = new GRect();
    String script = "(space setColor black) (robi setColor yellow)";

    public Exercice2_1_0() {
        space.addElement(robi);
        space.open();
        this.runScript();
    }

    private void runScript() {
        SParser<SNode> parser = new SParser<>();
        List<SNode> rootNodes = null;
        try {
            rootNodes = parser.parse(script);
        } catch (IOException e) {
            e.printStackTrace();
        }
        Iterator<SNode> itor = rootNodes.iterator();
        while (itor.hasNext()) {
            this.run(itor.next());
        }
    }

    private void run(SNode expr) {
        // A compléter...
    }

    public static void main(String[] args) {
        new Exercice2_1_0();
    }
}
```

2.2 Script d'animation

Compléter votre interpréteur pour permettre l'exécution du script suivant :

```
(space color white)
(robi color red)
(robi translate 10 0)
(space sleep 100)
(robi translate 0 10)
(space sleep 100)
(robi translate -10 0)
(space sleep 100)
(robi translate 0 -10)
```

- *translate* permet de déplacer avec un décalage en x et y passé en argument
- *sleep* provoque une mise en sommeil pour un nombre de millisecondes passé en argument

Exercice 3 : Introduction des commandes

Dans cet exercice on va faire la même chose que dans l'exercice précédent mais en améliorant la qualité du codage.

La version développée jusqu'à présent est limitée et peu évolutive. En effet, pour permettre l'interprétation de nouvelles commandes dans un script, il est nécessaire de modifier la fonction *run*. De plus, au fur et à mesure que le code pour de nouvelles commandes est ajouté, l'interpréteur grossit et se complexifie.

L'idée pour cet exercice est d'éviter ces inconvénients en introduisant des classes séparées pour la modélisation des commandes. Ces classes mettent en œuvre l'interface *Command*. L'exécution d'une commande de script revient alors à envoyer le message *run* à une instance d'une telle classe.

```
public interface Command {  
    abstract public void run();  
}
```

Par exemple, à partir de la commande de script suivante :

(space color white)

on pourrait créer une instance de la classe suivante :

```
public class SpaceChangeColor implements Command {  
    Color newColor;  
  
    public SpaceChangeColor(Color newColor) {  
        this.newColor = newColor;  
    }  
  
    @Override  
    public void run() { ... }  
}
```

Voici le squelette de la classe pour l'exercice 3. Il faut mettre en œuvre :

- les classes qui mettent en œuvre l'interface *Command* (ou conformes à *Command*). On développe une classe par commande possible du script source.
- la fonction *getCommandFromExpr* qui permet de retrouver ou créer une instance d'une classe conforme à *Command* et qui correspond à la commande de script (la S-expression) passée en argument. Par exemple, pour la commande (*space color white*), la fonction analyse la *SNode* et retourne une instance de la classe *SpaceChangeColor* car :
 - on a le mot clé *space* donc c'est une commande pour *space*.
 - on a le mot clé *color* donc c'est pour modifier la couleur

```
public class Exercice3 {  
    GSpace space = new GSpace("Exercice 3", new Dimension(200, 100));  
    ...  
    private void run(SNode expr) {  
        Command cmd = getCommandFromExpr(expr);  
        if (cmd == null)  
            throw new Error("unable to get command for: " + expr);  
        cmd.run();  
    }  
  
    Command getCommandFromExpr(SNode expr) {  
        return null;  
    }  
    ...  
}
```

Exercice 4 : Sélection et exécution des commandes

La sélection d'une commande à exécuter est programmée dans la fonction *getCommandFromExpr*. Cette fonction est centrale pour notre interpréteur. La solution adoptée pour son algorithme est très souvent basée sur une double conditionnelle :

- au premier niveau, pour le choix de l'objet *space* ou *robi*
- au second niveau pour le choix de la commande à exécuter.

Voici le squelette de la fonction *getCommandFromExpr* telle qu'elle est souvent proposée comme solution à l'exercice 3 :

```
Command getCommandFromExpr(SNode expr) {
    String target = expr.get(0).contents();
    if (target.equals("space")) {
        String cmd = expr.get(1).contents();
        if (cmd.equals("setColor")) ...
        } else if (cmd.equals("sleep")) ...
        } else
            throw new Error("Invalid space command:" + expr);
    } else if (target.equals("robi")) {
        String cmd = expr.get(1).contents();
        if (cmd.equals("setColor")) ...
        } else if (cmd.equals("translate")) ...
        } else {
            throw new Error("Invalid robi command:" + expr);
        }
    } else
        throw new Error("Unknown target in command: " + expr);
}
```

Par exemple, pour la S-expression suivante : (*space setColor blue*), la première conditionnelle permet de déterminer que l'objet sur lequel s'applique l'expression est *space*. La deuxième conditionnelle permet de déterminer que la fonction à exécuter sur *space* est *setColor*.

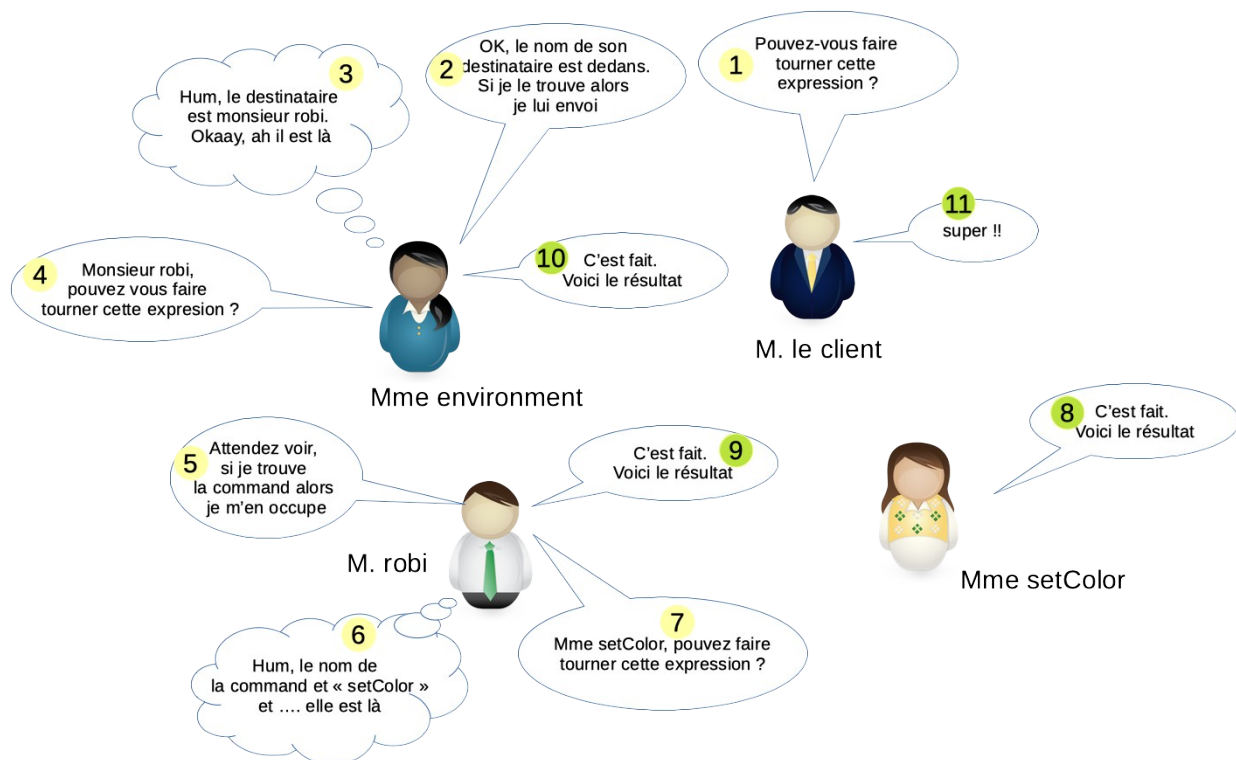
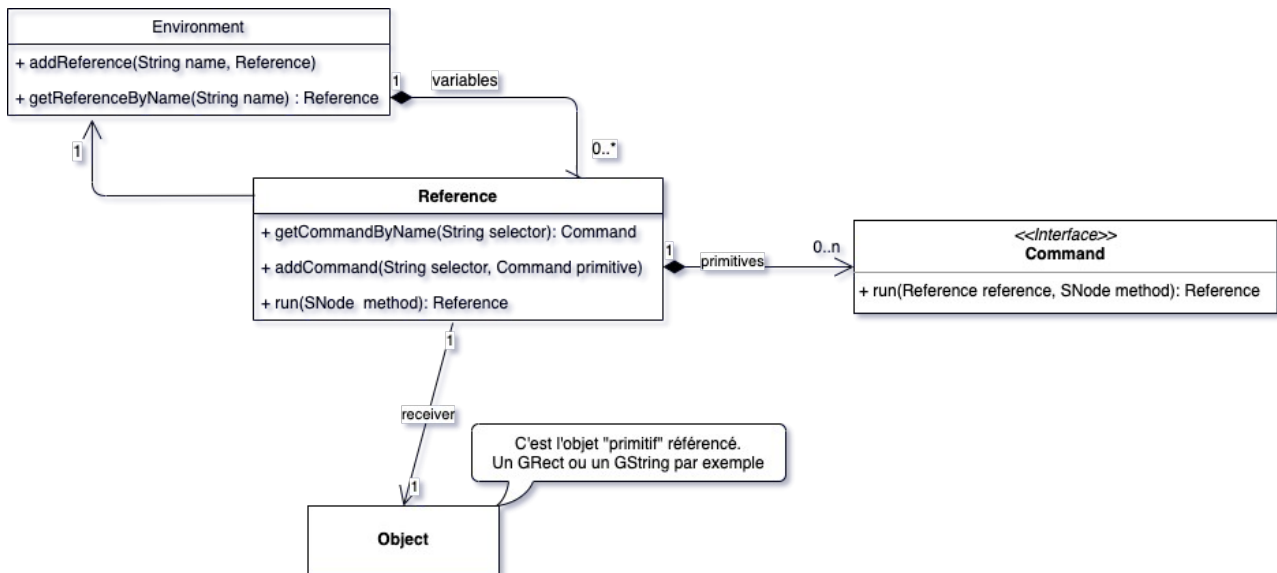
Cette solution n'est pas de bonne qualité. En effet, les choix possibles sont figés : il n'y a que deux objets possibles, *space* et *robi* et il y a un nombre fixe de fonctions. Or, ce code ne devrait pas être basé sur ces hypothèses très limitantes. Dans la suite, nous proposons de travailler à une meilleure solution qui s'appuie sur une structure à objets qui ne limite pas le nombre d'objets et le nombre de fonctions possibles.

4.1 Référencement des objets et enregistrement des commandes

Nous allons tout d'abord améliorer notre solution pour qu'il soit possible d'ajouter du comportement sans modifier en profondeur le code de l'interpréteur. La structure va donc évoluer pour pouvoir enregistrer des commandes et les retrouver pour les exécuter. Le schéma suivant présente une solution possible :

- **Command** : c'est notre interface commande de l'exercice 3 ; cependant la méthode *run* évolue quelque peu, elle prend deux arguments :
 - *reference* : c'est l'objet qui va exécuter la commande (pour l'instant, *robi* ou *space*).
 - *method* : c'est la *ExprList* complète produite par compilation d'une s-expression.
- **Reference** : permet de référencer un objet graphique à un ensemble de commandes nommées
 - *receiver* : c'est l'objet graphique (*space* et *robi* jusqu'à présent) ; seule la commande « sait » quel est son type (Grect ou Gspace) et peut donc utiliser un cast pour lui envoyer des messages.
 - *primitives* : ce sont les différentes mises en œuvre des commandes primitives disponibles pour le *receiver* (pour *robi*, pour l'instant on a *setColor* et *translate*)

- **Environment** : c'est l'environnement qui contient l'ensemble des références (deux jusqu'à présent, *space* et *robi* sont les seuls objets référencés).



Voici un dialogue qui part d'une demande *M. le client* à *Mme environment* pour exécuter une S-expression :

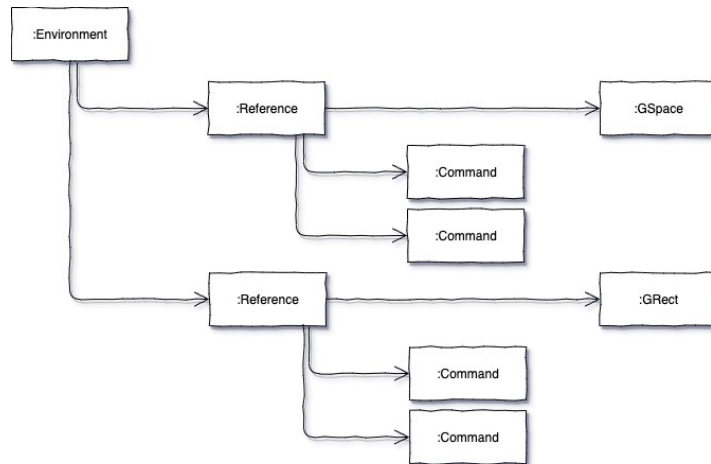
1. *M. interpreter* demande à *Mme environment* en lui passant la S-expression ;
2. *Mme environment* trouve dans son registre celui à qui est adressée la demande par son nom qui est dans la S-expression ;
3. *Mme environment* demande donc à *M. robi* d'exécuter la S-expression.
4. *M. robi* trouve le nom de la command « setColor » dans la S-expression et retrouve bien *Mme SetColor* dans ses registres ; il lui demande alors d'exécuter la S-expression ;
5. *Mme setColor* exécute la S-expression comme on lui a appris à faire puis elle retourne le résultat à *M. robi* ;

6. M. *robi* retourne le résultat à Mme *environment* qui retourne à son tour le résultat à M. *le client*.

La nouvelle solution est ainsi basée sur des objets et sur une structure de donnée arborescente qui permet de sélectionner in fine l'instance de *Command* qui va exécuter le code :

- Des instances de *Reference* sont enregistrées par leur nom dans l'instance de *Environment*
- Des instances de *Command* sont enregistrées dans chaque instance de *Reference*.

Voici une représentation de l'arbre d'instances pour un environnement et les deux références pour *space* et pour *robi* :



Voici le code (incomplet) des classes *Reference* et *Environment*. On remarque l'utilisation d'un *Map* pour stocker les instances de *Command* dans *Reference* et les instances de *Reference* dans *Environment*.

```
public interface Command {
    // le receiver est l'objet qui va executer method
    // method est la s-expression resultat de la compilation
    // du code source a executer
    // exemple de code source : "(space setColor black)"
    abstract public Reference run(Reference receiver, SNode method);
}

public class Reference {
    Object receiver;
    Map<String, Command> primitives;

    public Reference(Object receiver) {
        this.receiver = receiver;
        primitives = new HashMap<String, Command>();
    } ...
}

public class Environment {
    HashMap<String, Reference> variables;

    public Environment() {
        variables = new HashMap<String, Reference>();
    } ...
}
```

Dans cet exercice, il est demandé de mettre en œuvre ce modèle et de permettre de le tester en entrant des commandes au clavier. La boucle principale d'interprétation peut être la suivante :

```
private void mainLoop() {
    while (true) {
        // prompt
        System.out.print("> ");
```



```

// lecture d'une serie de s-expressions au clavier
String input = Tools.readKeyboard();
// creation du parser
SParser<SNode> parser = new SParser<>();
// compilation
List<SNode> compiled = null;
try {
    compiled = parser.parse(input);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
// execution des s-expressions compilees
Iterator<SNode> itor = compiled.iterator();
while (itor.hasNext()) {
    this.run((SNode) itor.next());
}
}
}

```

Voici la fonction *run* pour exécuter une s-expression compilée. Elle n'utilise maintenant plus aucune conditionnelle :

```

private void run(SNode expr) {
    // quel est le nom du receiver
    String receiverName = expr.get(0).getValue();
    // quel est le receiver
    Reference receiver = environment.getReferenceByName(receiverName);
    // demande au receiver d'exécuter la s-expression compilée
    receiver.run(expr);
}

```

4.2 Ajout et suppression dynamique d'éléments graphiques

Jusqu'à présent, l'environnement est pré-configuré avec un rectangle dont le nom est *robi*. L'idée maintenant est de créer et d'ajouter des éléments dynamiquement. Voici un exemple de script et le résultat sur la fenêtre :

```

(space add robi (Rect new))
(robi translate 130 50)
(robi setColor yellow)
(space add momo (Oval new))
(momo setColor red)
(momo translate 80 80)
(space add pif (Image new alien.gif))
(pif translate 100 0)
(space add hello (Label new "Hello world"))
(hello translate 10 10)
(hello setColor black)

```



On voit quatre objets graphiques ajoutés et manipulés dynamiquement. Prenons l'exemple de l'ajout d'un rectangle :


```
(space add robi (Rect new))
```

Le rectangle ajouté est le résultat du message *new* envoyé à une référence nommée *Rect*.

Voici comment peut être créé cette référence en 3 lignes de code :

1. *rectClassRef* est une référence sur la classe *GRect*
2. *rectClassRef* comprend la commande *new*
3. *rectClassRef* est accessible dans les scripts via le nom *Rect*

```
Reference rectClassRef = new Reference(GRect.class);  
rectClassRef.addCommand("new", new NewElement());  
environment.addReference("Rect", rectClassRef);
```

La classe *NewElement* met en œuvre la commande de création d'un objet graphique. Il est important de noter que le résultat est une référence sur un objet qui est retourné par la *run*. Cette référence est «configurée» de sorte qu'elle puisse comprendre les messages *setColor*, *translate* et *setDim*. On note aussi l'utilisation de la réflexivité pour la création d'une instance avec *newInstance*.

```
class NewElement implements Command {  
    public Expr run(Reference reference, SNode method) {  
        try {  
            @SuppressWarnings("unchecked")  
            GElement e = ((Class<GElement>)  
                reference.getReceiver().getDeclaredConstructor()).newInstance();  
            Reference ref = new Reference(e);  
            ref.addCommand("setColor", new SetColor());  
            ref.addCommand("translate", new Translate());  
            ref.addCommand("setDim", new SetDim());  
            return ref;  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        return null;  
    }  
}
```

Voici le code qui initialise l'environnement avec les références qui permettent de créer et d'ajouter des nouveaux éléments graphiques :

```
public Exercice4_2() {  
    GSpace space = new GSpace("Exercice 4", new Dimension(200, 100));  
    space.open();  
  
    Reference spaceRef = new Reference(space);  
    Reference rectClassRef = new Reference(GRect.class);  
    Reference ovalClassRef = new Reference(GOval.class);  
    Reference imageClassRef = new Reference(GImage.class);  
    Reference stringClassRef = new Reference(GString.class);  
  
    spaceRef.addCommand("setColor", new SetColor());  
    spaceRef.addCommand("sleep", new Sleep());  
  
    spaceRef.addCommand("add", new AddElement(environment));  
    spaceRef.addCommand("del", new DelElement(environment));  
  
    rectClassRef.addCommand("new", new NewElement());  
    ovalClassRef.addCommand("new", new NewElement());  
    imageClassRef.addCommand("new", new NewImage());  
    stringClassRef.addCommand("new", new NewString());  
}
```

```

environment.addReference("space", spaceRef);
environment.addReference("Rect", rectClassRef);
environment.addReference("Oval", ovalClassRef);
environment.addReference("Image", imageClassRef);
environment.addReference("Label", stringClassRef);

this.mainLoop();
}

```

Exercice 5 : Ajouter des éléments à des conteneurs

Jusqu'à présent, les éléments graphiques sont ajoutés au niveau global, uniquement dans le *space*. Cependant, *GRect*, *GOval* sont des *GContainer*, c'est à dire qu'ils peuvent eux aussi contenir des éléments.

Normalement, avec ce dont on dispose déjà, il doit être possible, avec très peu de modifications de code, d'autoriser les *GContainer* à contenir des éléments. Normalement pas plus que 2 lignes de codes supplémentaires sont nécessaires si une commande *AddElement* existe déjà et fonctionne correctement.

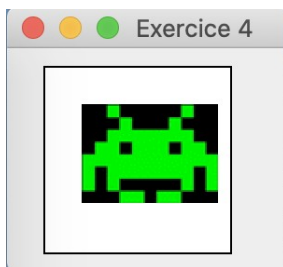
Par exemple, le script suivant permet d'ajouter l'image du fichier « alien.gif » dans *robi* qui lui est un rectangle dans *space* :

```

(space setDim 150 120)
(space add robi (Rect new))
(space.robi setColor white)
(space.robi setDim 100 100)
(space.robi translate 20 10)
(space.robi add im (Image new alien.gif))
(space.robi.im translate 20 20)

```

Voici le résultat d'exécution du script :



Pour éviter les conflits éventuels de nom, une notation pointée est utilisée. Ainsi, pour accéder à *robi* qui est ajouté à *space*, il faut utiliser le nom composé *space.robi*. Pour accéder à l'image *im* ajoutée au rectangle *robi* qui lui même est dans *space*, on utilise *space.robi.im*.

Attention, il peut y avoir plusieurs *robi* mais dans des conteneurs différents. Voici un script qui crée deux rectangles nommés *robi* l'un dans l'autre :

```

(space add robi (Rect new))
(space.robi setDim 50 50)
(space.robi add robi (Rect new))
(space.robi.robi setColor red)
(space.robi setColor white)

```

Et voici le résultat du script :



Quand *space.robi* est supprimé avec (*space del robi*) alors son contenu est aussi supprimé. Donc *space.robi.robi* est non seulement retiré de la visualisation graphique mais aussi, n'est plus accessible depuis aucun environnement.

Exercice 6 : Création et exécution de scripts

On voudrait maintenant pouvoir doter nos objets graphique de la possibilité d'enregistrer et de rejouer des scripts. Un script a toujours au moins un paramètre qui correspond à la référence dans laquelle le script a été enregistré (notez que cela correspond au mot clé *this* qui est la référence du receveur dans les méthodes d'instance en java). Par convention, ce paramètre se nomme *self* (comme en python).

Voici un exemple de script ajouté au space :

```
( space addScript empty ( ( self ) ( self clear ) ) )
```

Voici une utilisation de ce script :

```
( space empty )
```

Un script peut avoir des paramètres supplémentaire. Voici par exemple, un script qui prend en argument le nom d'un fichier pour la création d'une image. Un exemple d'utilisation est aussi montré.

```
( space addScript addImage (
  ( self filename )
  (self add im ( Image new filename ) ) ) )
( space addImage alien.gif )
```

Un script peut intégrer plusieurs opérations :

```
( space add robi (Rect new ) )
( space.robi addScript addRect (
  ( self name w c )
  ( self add name ( Rect new ) )
  ( self.name setColor c )
  ( self.name setDim w w ) ) )
( space.robi addRect mySquare 30 yellow )
```