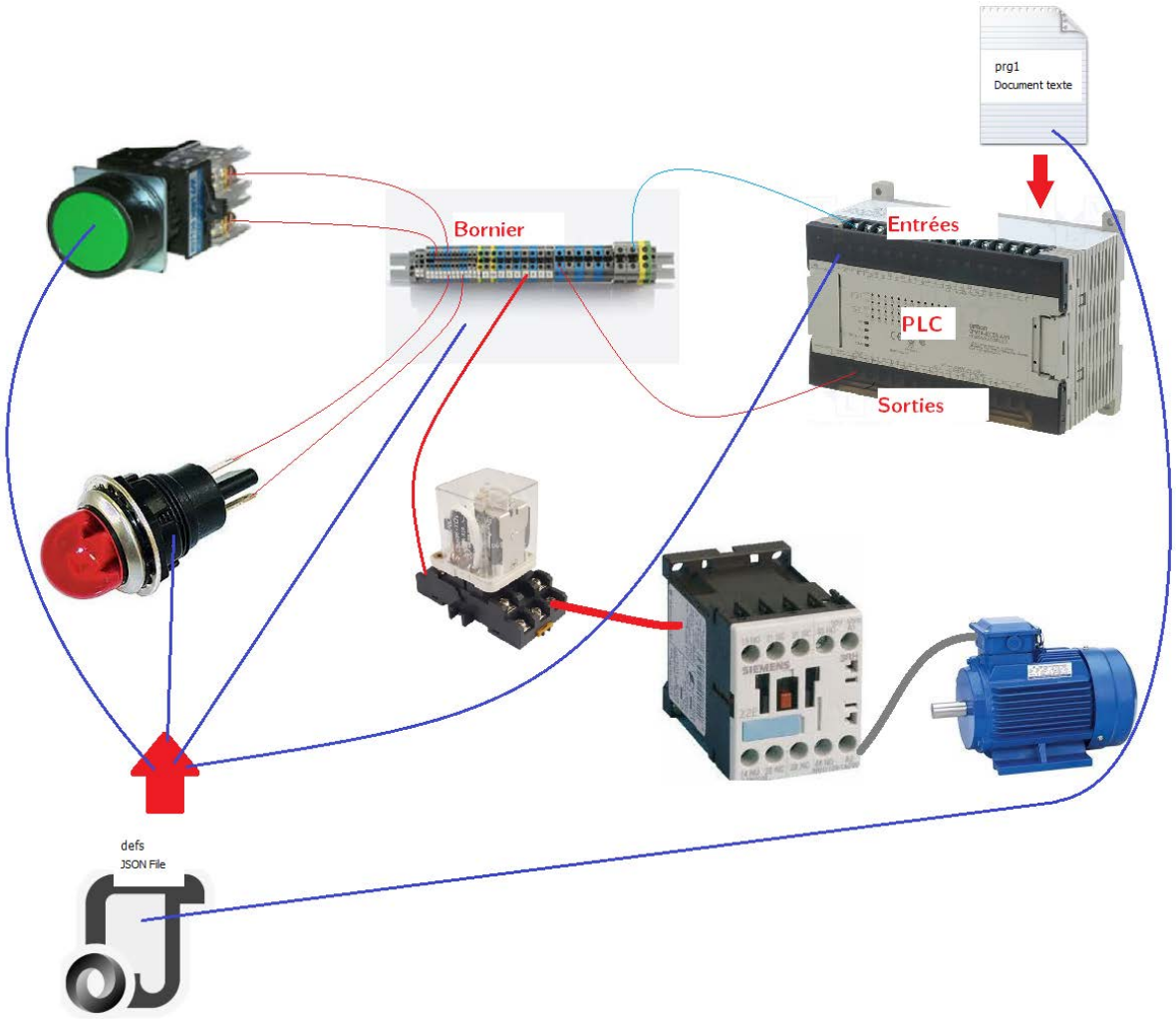
Description des éléments du programme

Un processus simulé peut être représenté par le schéma fonctionnel ci-dessous.



- Le fichier "defs.json" contient les paramètres de chaque objet à dessiner et à faire vivre.
- Le fichier "prg1.txt" contient le programme de l'automate.

Les boutons poussoir, interrupteurs, indicateurs et tous les autres acteurs dont le "PLC"[1] fournissent des informations au bornier.

Le bornier pourra :

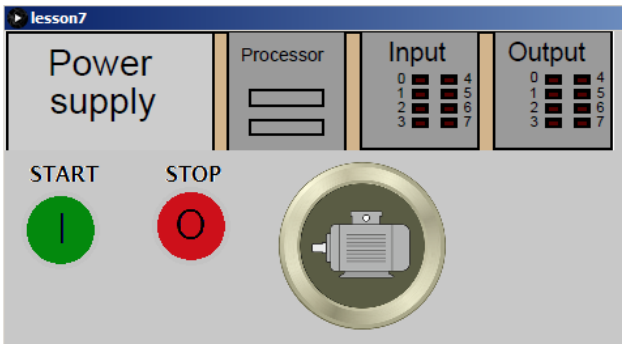
- Transmettre le signal d'un bouton poussoir à un indicateur ou (et) le PLC.
- Transmettre le signal fourni par un "producer" vers un "consumer".

Question :

Dans le schéma ci-dessus, quels sont les éléments "producer" et "consumer" ?

[1] PLC, Programmable Logic Controler

Exemple, système marche-arrêt d'un moteur



Le système simulé contient :

- Deux boutons poussoirs, start et stop.
- Un plc.
- Un indicateur de commande du moteur.

Programme de l'automate

```
// Programme d'exemple, marche arrêt avec deux boutons poussoir.
// Déclarations

defs    start,I1        // définir l'entrée start, entrée I1 du PLC
defs    stop,I2         // définir l'entrée stop, entrée I2 du PLC
defs    motor,O1        // définir la sortie motor, sortie O1 du PLC

// +---| |---+ contact normalement ouvert, contact interne ou entrée I1 à I8
// +---|\|---+ contact normalement fermé
// +---( )---> relai interne ou sortie O1 à O8

//   start       stop        motor
// <-+---| |---+---|\|---+---( )--->
//   |  motor  |
//   +---| |---+

LD      start   // lire le contact start. (entrée automate)
OR      motor   // faire un ou logique avec le contact auxiliaire de motor
ANDNOT  stop    // faire un et logique avec l'inverse du contact stop (entrée automate)
OUT     motor   // agir sur la bobine motor. (sortie automate)
```

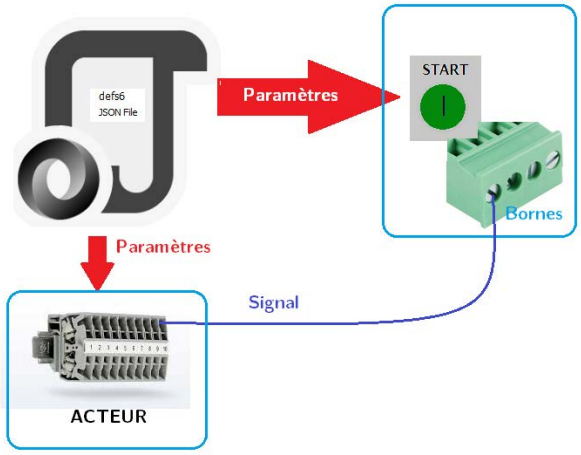
Le schéma à contacts et traduit par quatre instructions que comprend le "plc".

Le fichier "defs6.json"

```
{
  "actors": [

        { "name": "plc "type": "Plc",
          "x": 0, "y": 0,
          "images": [ "plc.png" ] ,
          "programFilename" : "prg1.txt"
        },

        { "name": "start "type": "Button",
          "x": 10, "y": 100,
          "images": [ "start_off.png", "start_on.png" ]
        },

        { "name": "stop "type": "Button",
          "x": 110, "y": 100,
          "images": [ "stop_off.png", "stop_on.png" ]
        },

        { "name": "motor "type": "Indicator",
          "x": 210, "y": 100,
          "images": ["imagesV\\icon_led_motor_off.png","imagesV\\icon_led_motor_on.png"],
          "scale" : 0.25
        }
  ],

"terminalBoard": [
        {
                "provider": { "actor": "start", "terminal": "contact" },
                "consumer": { "actor": "plc", "terminal": "I1"}
                },

        {
                "provider": { "actor": "stop", "terminal": "contact" },
                "consumer": { "actor": "plc", "terminal": "I2"}
                },
        {
                "provider": { "actor": "plc", "terminal": "O1" },
                "consumer": { "actor": "motor", "terminal": "state"}
                }
        ]
}
```

- Le fichier décrit les paramètres des acteurs du système simulé via l'objet au format JSON nommé "actors".
- Il décrit également les connexions du bornier via l'objet au format JSON nommé "terminalBoard".
- Quatre acteurs sont décrits, part :
    - Leur nom.
    - Leur type.
    - Leur position dans l'écran.
    - Leur tableau d'images nécessaires.
    - L'échelle de dessin si les images sont trop grandes.
    - D'autres paramètres propres au type d'acteur. (PLC doit déclarer un paramètre nommé "programFilename"
- Le bornier définit trois interconnexions. (Six fils)
    - Chaque borne du bornier doit être connectée à un "producer" et à un "consumer".
    - On définit "producer" et à un "consumer", via deux propriétés :
        - La propriété "actor" qui définit l'objet produisant ou consommant l'information.
        - La propriété "terminal" qui définit la borne de l'objet produisant ou consommant l'information.
- Certaines propriétés des objets au format JSON sont optionnels, par exemple "scale", lorsque que ces propriétés optionnelles ne sont pas présentes dans le fichier, une valeur par défaut est définie.

Structure d'un acteur



1. Lors de sa déclaration, un objet "actor" ou descendant, reçoit ces paramètres via le contenu du fichier "def6.json".
2. Pour communiquer entre eux, les "actors" utilisent leurs bornes telles que par exemple "contact pour un bouton poussoir.
3. Les acteurs évoluent en fonction des données transmissent ou reçues via leurs bornes.
4. Les données transmissent seront pour l'instant :
    a. Binaire pur un contact.
    b. Entière pour par exemple un niveau de réservoir.
    c. Simple précision (float) pour d'autres acteur à définir.

La première classe d'objet que nous devons décrire est la classe qui correspond au fonctionnement des bornes, Dans la suite du texte nous utiliserons le terme "terminal" pour désigner une borne.

- Une borne, "terminal".
- Une paire de bornes connectées, " TerminalConnector".
- Un borner, "terminalBoard".

La classe Terminal

La classe "Terminal" est une classe particulière elle doit simuler le fonctionnement d'une borne par laquelle trois formats de données peuvent transiter. (Boolean, Integer ou Float)

En langage Java on peut appeler ce type de classe, une classe générique[2].

Exemple:

```
public class borne<T>[3] {
 private T value;
 borne(T v) {value=v;}
 T read() { return value; }
 void write(T newValue) { value=newValue;}
}

void setup() {
 borne<Boolean> a,b; // déclaration de deux variables de class borne, contenant des booleans
 borne<Integer> c,d; // déclaration de deux variables de class borne, contenant des entiers

 a=new borne<Boolean>(false); // création de la variable avec false comme valeur initiale
 b=new borne<Boolean>(true); // création de la variable avec false comme valeur true
 c=new borne<Integer>(25); // création de la variable avec 25 comme valeur initiale
 d=new borne<Integer>(-12); // création de la variable avec -12 comme valeur true

 b.write(a.read()); // lecture de a et ensuite modification de b avec la valeur de a
 c.write(d.read()); // lecture de d et ensuite modification de c avec la valeur de d

 // c.write(a.read()); // erreur a et c doivent être du même type.
}
```

```
void setup() {
borne[] bornier=new borne[2];
  bornier[0]=new borne<Boolean>(false);
  bornier[1]=new borne<Integer>(2);
  for( borne b : bornier)
    println(b.read());
}
```

Le programme affiche false et 2.

Un tableau de bornes peut donc contenir des bornes de natures différentes. Un bornier peut donc contenir des bornes de nature différentes.

La classe "**Terminal**" contient :

- Deux propriétés, nommées "value" et "numberOfProvider".
  - "value" contient la valeur de l'information. (Boolean, Integer ou Float)
  - ""numberOfProvider", est un entier qui permet de vérifier qu'un seul "producer" et connecté à un "consumer".
- Un constructeur et quatre méthodes :
  - Le constructeur permet définir la valeur initiale. Il affecte la propriété "value".
  - Une fonction "read" qui permet de lire la valeur.
  - Une procédure "write" qui permet de modifier "value".
  - Une fonction "getType" qui fournit un String correspondant au type de variable. (Boolean, Integer ou Float)
  - Une fonction "connectedToProducer" qui retourne "true" s'il n'y a qu'un seul "producer" connecté à cette borne. (Utilisé lors de la création des connexions du bornier)

[2] https://docs.oracle.com/javase/tutorial/java/generics/types.html

[3] <T> T pour définir un type de variable à déterminer lors de l'instanciation.

Le code :

```
public class Terminal<T> {
  protected T value; // l'information, la valeur de la borne

  private  int  numberOfProducer=0; // nombre de borne lui fournissant de l'information

  protected  Terminal(T v) { value=v; } // constructeur


  public synchronized T read() { return value; } // pour obtenir la valeur

  public synchronized void write(T newValue) { value=newValue; } // pour modifier la valeur

  public String getType() {return value.getClass().getSimpleName(); }

  public boolean connectedToProducer(){ // pour vérifier qu'un seul provider est connecté
    numberOfProducer++;
    return (numberOfProducer==1);
    };
}
```

Les acteurs posséderont des variables de ce type, utilisées pour communiquer entre eux.

Un bornier est constitué de plusieurs couples de bornes, un couple de bornes réalise la fonction de transmission d'information du "producer" vers le "consumer".

Le couple de bornes est défini via la classe "TerminalConnecor".

La classe "TerminalConnector"

Propriétés			
Visibilité	Nom	Type	Rôle
`protected`	**`provider`**	`Terminal`	Source de l'information à transmettre
`protected`	**`consumer`**	`Terminal`	Destination de l'information
`protected`	**`normalyClosed`**	`boolean`	Permet de complémenter une information logique reçue.
`protected`	**`activeLow`**	`boolean`	Permet de complémenter une information logique transmise.
`private`	**`isBoolean`**	`boolean`	Vrai si le type de "value" est boolean

Méthodes			
Type de valeur retournée	Nom	Paramètres	Rôle
	`TerminalConnector`	`provider,consumer, normalyClosed,activeLow`	Constructeur
	`TerminalConnector`	`provider,consumer`	Constructeur
`void`	**`initialize`**	`provider,consumer`	Utilisé par les constructeur
`void`	**`transmit`**		Réaliser le transfert d'information

La procédure "initialize " gère deux types d'erreurs :

- L'incohérence de type, par exemple connecter un "producer" boolean à un "consumer" non boolean.
- La connexion de plusieurs "producer" vers un même "consumer".

Le programme est alors arrêté avec affichage du type d'erreur.

Le code de la classe "TerminalConnector"

```
class TerminalConnecor {
  protected Terminal provider, consumer;
  protected boolean  normalyClosed, activeLow;
  private   boolean  isboolean=false;

  public TerminalConnecor(Terminal provider, boolean normalyClose, Terminal consumer,
                          boolean activeLow) {
    initialize(provider, consumer);
    isboolean=true;
    this.normalyClosed=normalyClose;
    this.activeLow=activeLow;
  }

  public TerminalConnecor(Terminal provider, Terminal consumer) {
    initialize(provider, consumer);
  }

  private   void initialize(Terminal provider, Terminal consumer) {
    try {
    if (!provider.getClass().equals(consumer.getClass()))
      throw new RuntimeException("Erreur, provider et consumer de types différents");
    if (consumer.connectedToProducer())
      throw new RuntimeException("Erreur, plus d'un provider connecté à un consumer");
    }
    catch (RuntimeException e) {
      println(e);
      terminate=true;
      exit();
      }
    this.provider=provider;
    this.consumer=consumer;
  }

  public void transmit() {
    if (isboolean)
      consumer.write( ((boolean) provider.read() ^ normalyClosed) ^ activeLow);
    else
      consumer.write(provider.read());
  }
}
```

Quel est le rôle de normalyClosed et activeLow

- Par défaut un bouton poussoir ou un interrupteur fournira via son contact un signal actif à l'état vrai. Si "normalyClosed" est déclaré comme "true" alors l'information transmisse au "consumer" sera le complément logique de l'état du contact.
- Par défaut un objet "consumer" est actif à l'état logique "true". Si "activeLow" est déclaré comme "true" alors l'information transmisse au "consumer" sera le complément logique de celle à transmettre.
- Si "normalyClosed" et "activeLow" sont vrai, l'information sera double inversée et donc non modifiée.

(0 = NOT 1) et (1 = NOT NOT 1).

$$0 = \overline{1} \quad 1 = \overline{\overline{1}}$$

Exemple : Un bouton poussoir est connecté à deux bornes séparées (A1 et A2), un indicateur actif bas[4] est connecté en A1' et une entrée de l'automate est connectée à l'entrée active haut[5] de l'automate. Pour le couple de bornes A1-A'1 il faut prévoir que soit le "normalyClosed" soit true ou que "activeLow" soit true. (L'un ou l'autre pas les deux)

La classe "TerminalBoard"

Cette classe permet de créer un bornier constitué d'un certain nombre de bornes. Le nombre de borne est déterminé lors de la lecture du fichier "defxx.json".

Cette classe contient une propriété et deux méthodes et est "runnable".

Propriétés			
Visibilité	Nom	Type	Rôle
`private`	**`terminals`**	`Terminalconnector[]`	Les couples de bornes

Méthodes			
Type de valeur retournée	Nom	Paramètres	Rôle
	`TerminalBoard`	`Un objet JSON extrait du fichier defsxx.json`	Constructeur
`void`	**`run`**		Réaliser le transfert des informations

Vu la complexité du code du constructeur il ne sera pas publié dans ce texte. (ni étudié)

Le code de la classe "TerminalBoard"

```
class TerminalBoard implements Runnable {

  private TerminalConnecor[] terminals;

  TerminalBoard(JSONArray jterminals) {
    terminals= new TerminalConnecor[jterminals.size()];
    for (int i=0; i<jterminals.size(); i++) {
       // declare chaque couple de connexion.
    …
    }

  void run() {
    for (;; ) {
      for (int i=0; i<terminals.length; i++)
        terminals[i].transmit();
      try {
        java.lang.Thread.sleep(samplingTime);
      }
      catch (Exception e) {
        println(e);
      }
    }
  }
}
```

Nous allons maintenant nous attaquer à la structure des acteurs.

[4] Actif si False.
[5] Actif si True.

La classe "Actor"

La classe "Actor" est à la fois une classe pour un acteur passif comme un élément de décor et une classe de base pour la construction des autres classes d'objets.

Pour construire un objet du type "Actor", on puisse l'information dans une partie du fichier defsxx.json, celle qui décrit l'objet.

Pour un objet de décor comme l'image de fond le fichier defsxx.json contient :

```
{
  "actors": [
        {
          "name": "plant", "type": "Actor",
          "x": 0, "y": 0,
          "images": [ "plc.png" ]
        ]
}
```

- Le mot clé "actors" permet de déclarer un tableau. […]
- Les mots clés "name", "type", "x", "y","images" définissent les propriétés indispensables de l'objet.

L'acteur dispose d'une borne nommée "visible", via cette borne "Boolean", on peut le rendre visible ou invisible. Le signal peut être fourni par un autre acteur. (Interrupteur, sortie de l'automate …)

L'acteur de base n'est "runnable" car il n'a rien à faire. Par contre, certains de ces descendants seront "runnable".

Propriétés			
Visibilité	Nom	Type	Rôle
`protected`	**`imgs`**	`PImage[]`	Contenir les images nécessaires pour dessiner l'acteur. (Une image pour l'acteur de base)
`protected`	**`x,y`**	`int`	Coordonnées du coin supérieur gauche
`protected`	**`w,h`**	`int`	Largeur et hauteur des images.
`protected`	**`index`**	`int`	Numéro de l'image à afficher. (Toujours=à 0 pour l'acteur de base)
`protected`	**`scale`**	`float`	Échelle de dessin des images
`protected`	**`visible`**	`Terminal<Boolean>`	Borne pour rendre visible ou non l'acteur

Méthodes			
Type de valeur retournée	Nom	Paramètres	Rôle
	`Actor`	`Un objet JSON extrait du fichier defsxx.json`	Constructeur
`void`	**`draw`**		Dessine l'acteur
`boolean`	**`isX`**	`int X,int x1,int x2`	Détermine si l'abscisse de l'objet est comprise entre X-x1 et X+X2
`boolean`	**`isY`**	`int Y,int y1,int y2`	
`Terminal`	**`getTerminal`**	`String terminalName`	Fourni un objet du type Terminal à partir de son nom sous la forme d'une chaîne de caractères.

Pour voir le code, consulter dans le projet fournit l'onglet Actor.

Boutons poussoir et interrupteurs

Pour simuler les boutons poussoir et interrupteurs, une seule class existe, elle est nommée Button.

Avant de déclarer cet classe nous devons définir un "Interface" qui déclarera les méthodes que devra ajouter à "Actor" la classe "Button".

En effet, la classe "Button" sera déclarée :

```
public class Button extends Actor implements Clickable
```

La classe "Bouton" est un descendant de la classe "Actor" qui implémente les méthodes de l'interface "Clickable".

L'interface Clickable

```
interface Clickable {
         public boolean mousePressed();
         void mouseReleased();
       }
```

On doit donc prévoir deux méthodes qui gérerons l'activation et la désactivation du bouton de la souris. (Gauche ou droite, peu importe)

Les éléments de la classe "Button"

La classe "Button" dispose de toutes les propriétés "public" et "protected" de la classe "Actor".

Elle dispose également des méthodes de la classe "Actor"

On ajoute :

Propriétés			
Visibilité	Nom	Type	Rôle
protected	**interlock**	boolean	Simule le verrouillage présent dans un interrupteur et absent dans une bouton poussoir.
protected	**appOnAction**	String	Commande que doit réaliser l'application
protected	**AppOffAction**	String	Seul commande "quit" pour quitter le programme est prévue pour l'instant.
protected	**contact**	Terminal<Boolean>	Simule le fonctionnement du contact dont l'état est modifié par la souris.

On ajoute ou surcharge :

Méthodes			
Type de valeur retournée	Nom	Paramètres	Rôle
	Button	Un objet JSON extrait du fichier defsxx.json	Constructeur
void	**draw**		Dessine l'acteur
boolean	**mousePressed()**		Détermine si la position de la souris est dans la zone rectangulaire de l'image et modifie l'état de la propriété contact via contact.write(…)
void	**mouseReleased()**		

Pour voir le code, consulter dans le projet fournit l'onglet Actor.

Question d'examen

En vous inspirant que ce qui vient d'être décrit, que contient la classe "Indicator" ?

Pour voir le code, consulter dans le projet fournit l'onglet Actor.

Elle dispose également des méthodes de la classe "…"

On ajoute :

Propriétés			
Visibilité	Nom	Type	Rôle

On ajoute ou surcharge :

Méthodes			
Type de valeur retournée	Nom	Paramètres	Rôle

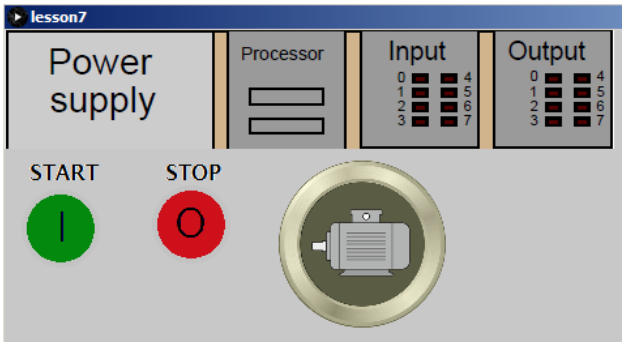
Explication …

Vous pouvez préparer pour l'examen l'explication de n'importe quel acteur du programme fournit.

Jusque la veille du jours de l'examen, vous pouvez me poser des questions.

Comment créer le fichier def.json de votre processus?

Dresser la liste des acteurs et de leurs paramètres.



Liste des acteurs

Nom	Type	X	Y	Images		scale	Program Filename
start	Button	10	100	start_off.png	start_on.png		
stop	Button	110	100	stop_off.png	stop_on.png		
motor	Indicator	210	100	icon_led_motor_off.png	icon_led_motor_on.png	0.25	
Plc	Plc	0	0	plc.png			prg1.txt

Liste des connexions du bornier

producer		consumer	
actor	terminal	actor	terminal
start	contact	plc	I1
Stop	contact	plc	I2
plc	O1	motor	state

Liste des acteurs

Voici la liste des acteurs décrites au format JSON

Les éléments marqués en jaune sont optionnels

- Actor. Pour le décor

```
{
	"name" : "",
	"type" : "Actor",
	"x" : 0,
	"y" : 0,
	"images" : [""],
	"scale" : 1.0,
	"visible" : true
	}
```

- Bouton poussoir

```
{
	"name" : "",
	"type" : "Button",
	"x" : 0,
	"y" : 0,
	"images" : ["",""],
	"scale" : 1.0,
	"visible" : true,
	"interlock" : false
	}
```

- Interrupteur

```
{
	"name" : "",
	"type" : "Button",
	"x" : 0,
	"y" : 0,
	"images" : ["",""],
	"scale" : 1.0,
	"visible" : true,
	"interlock" : true
	}
```

- Indicateur

```
{
	"name" : "",
	"type" : "Indicator",
	"x" : 0,
	"y" : 0,
	"images" : ["",""],
	"scale" : 1.0,
	"visible" : true,
	" useImageSelector" : false
	}
```

- Plc

```
{
	"name" : "plc",
	"type" : "Plc",
	"x" : 0,
	"y" : 0,
	"images" : ["plc.png"],
	"programFilename" : "prg.txt"
	}
```

- HorizontalPositionSensor

```
{
	"name" : "",
	"type" : " HorizontalPositionSensor",
	"x" : 0,
	"y" : 0,
	"images" : ["ledOff.png","ledOn.png"],
	"target" : "",
	"x1" : 0.0,
	"x2" : 1.0
	}
```

- VerticalPositionSensor

```
{
	"name" : "",
	"type" : " VerticalPositionSensor",
	"x" : 0,
	"y" : 0,
	"images" : ["ledOff.png","ledOn.png"],
	"target" : "",
	"y1" : 0.0,
	"y2" : 1.0
	}
```

- LevelPositionSensor

```
{
	"name" : "",
	"type" : " LevelPositionSensor",
	"x" : 0,
	"y" : 0,
	"images" : ["ledOff.png","ledOn.png"],
	"target" : "",
	"y1" : 0.0,
	"y2" : 1.0,
	"targetSignalName" : "level",
	"tripLevel" : 5 ,
	"span" : 1
	}
```

- VerticalColorSensor

```
{
	"name" : "",
	"type" : " VerticalColorSensor",
	"x" : 0,
	"y" : 0,
	"images" : ["ledOff.png","ledOn.png"],
	"target" : "",
	"y1" : 0.0,
	"y2" : 1.0
	}
```

- HorizontalMovingActor

```
{
        "name": "",
        "type": "HorizontalMovingActor",
        "x": 10,
        "y": 100,
        "images": [""],
        "leftSpeed": 1,
        "rightSpeed": 1,
        "scale": 0.5
        }
```

- VerticalMovingActor

```
{
        "name": "",
        "type": " VerticalMovingActor",
        "x": 10,
        "y": 100,
        "images": [""],
        "upSpeed": 1,
        "downSpeed": 1
        }
```

- Follower

```
{
        "name": "",
        "type": " Follower",
        "x": 10,
        "y": 100,
        "images": [""],
        "driver": ""
        }
```

- Ball

```
{
        "name": "",
        "type": " Ball",
        "x": 10,
        "y": 100,
        "images": ["",""]
        }
```

- HorizontalPneumaticCylinder

```
{
        "name" : "upperTrap",
        "type" :"HorizontalPneumaticCylinder" ,
        "x": 289, "y": 43,
        "images": [ "trap.png"],
        "leftSpeed" : 1,
        "rightSpeed" : 1 ,
        "direction" : "right" ,
        "extensionLength" : 55
        }
```

- Tank

```
{
        "name" : "tank",
        "type" : "Tank",
        "x": 305, "y": 252,
        "minLevel" : 0,
        "maxLevel" : 100,
        "surface" : 1,
        "inFlow" : "",
        "outFlow" : ""
        }
```

- Valve

```
{
        "name" : "",
        "type" : " Valve",
        "x": 282, "y": 0,
        "images" : ["closeValve.png","openValve.png"],
        "flow" : 10
        }
```

Les fichiers defs1.json à defs6 permettent d'étudier le fonctionnement de chaque acteur séparément. (Paramètres, positions …)
Les fichiers prg1.txt à prg5.txt permettent de découvrir le fonctionnement du plc.

Bornes des acteurs

Acteur	Producer	Consumer
Actor		visible <boolean>
Button	contact <boolean>	visible <boolean>
Indicator		visible <boolean>
		state <boolean>
HorizontalPositionSensor	contact <boolean>	visible <boolean>
VerticalPositionSensor	contact <boolean>	visible <boolean>
LevelPositionSensor	contact <boolean>	visible <boolean>
VerticalColorSensor	contact <boolean>	visible <boolean>
	ballcolor <Integer>	
	whiteBall <boolean>	
	blackBall <boolean>	
Plc	O1-O8	visible
		I1-I8
HorizontalMovingActor		visible <boolean>
		left <boolean>
		light <boolean>
VerticalMovingActor		visible <boolean>
		up <boolean>
		down <boolean>
Follower		visible <boolean>
Ball		upperTrap <boolean>
		lowerTrap <boolean>
		pusher <boolean>
		ballColor <boolean>
		whiteBall <boolean>
		blackBall <boolean>
HorizontalPneumaticCylinder	airflow <boolean>	
	released <boolean>	
	activated <boolean>	
Tank	level <integer>	
Valve	state <boolean>	
	flow <integer>	