Objets connectés

Projet | Partie I

Table des matières

Introduction

Vous devez produire un logiciel flexible pouvant réaliser différentes tâches avec le robot GoPiGo3.

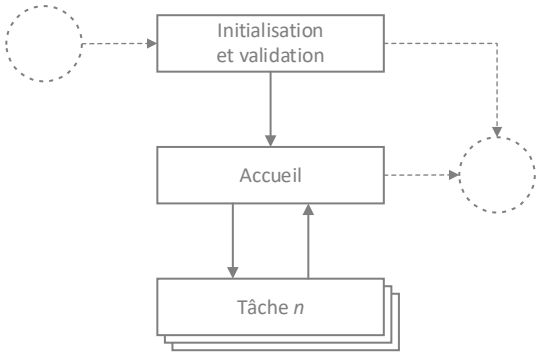
Le projet est divisé en trois parties principales :

1. Développement d'une librairie générique implémentant une machine d'états.
2. Développement de l'infrastructure logicielle du projet.
3. Développement d'une tâche applicative du logiciel :

Présentation générale du logiciel

Le logiciel à produire se découpe en 3 éléments distinctifs :

1. Initialisation et validation de l'intégrité du robot
2. Accueil
3. Tâches



Le premier élément consiste à valider si le robot est fonctionnel et disponible. Lorsque l'intégrité du robot est confirmée, le programme attend les instructions de l'usager. Ce dernier peut démarrer une tâche ou quitter le logiciel. L'opération passe par la télécommande du robot.

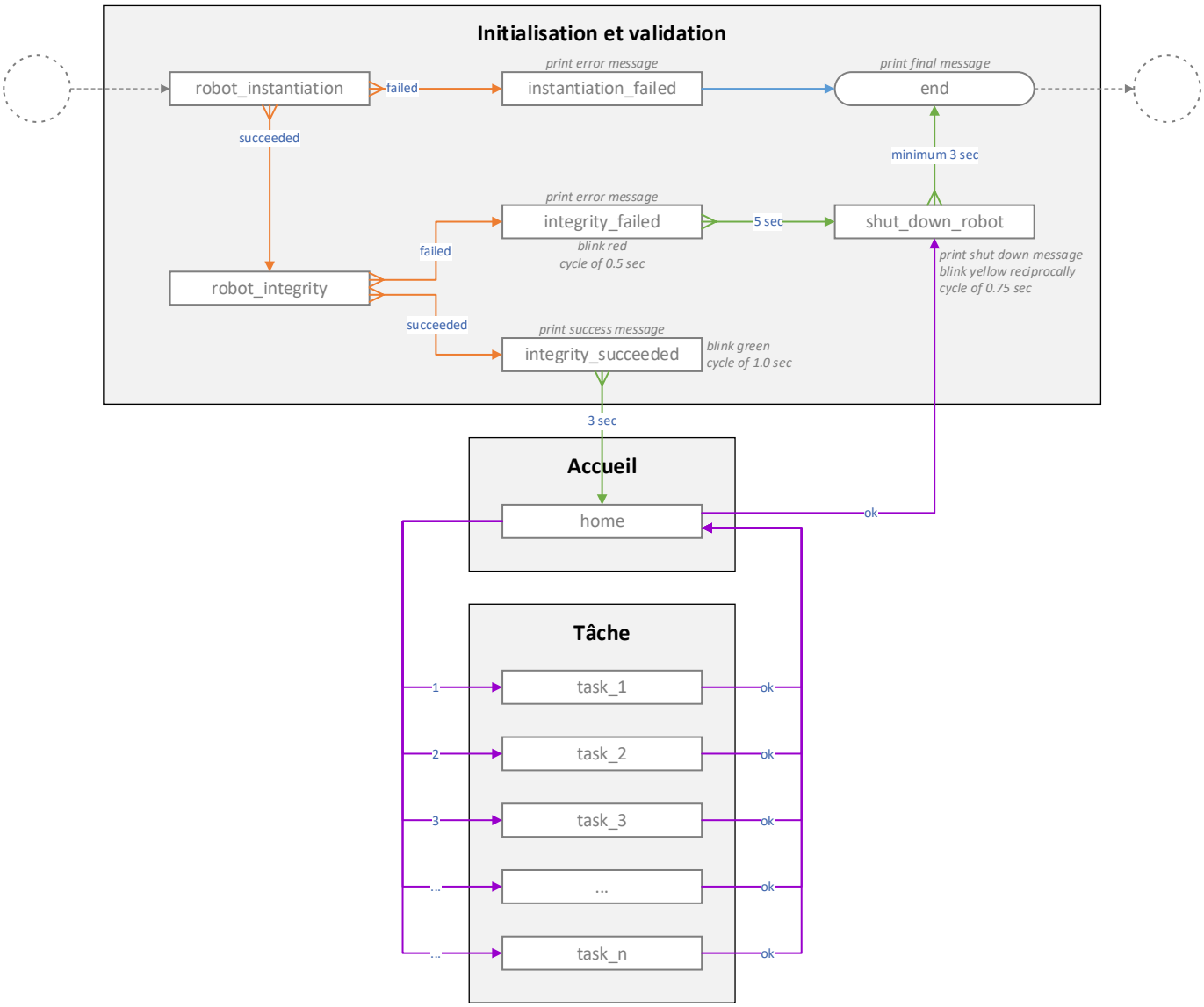
Niveau technique attendu

Ce projet doit inclure plusieurs modules différents et, rendu à ce point de votre formation, il est important de produire un programme flexible et réutilisable autant que possible.

Il est attendu que votre conception présente un excellent niveau d'abstraction. De plus, vous devrez mettre en pratique de bonnes techniques de programmation et les notions plus avancées présentées tout au long de la session.

Déroulement détaillé de l'application

Le schéma suivant présente le détail des opérations générales de l'application.



Vous trouverez ci-bas une description de chaque état, des opérations à faire et des conditions de transition.

- `robot_initialisation` : instanciation du robot
  - rôle : on s'assure que l'objet représentant le robot soit instancié et disponible
  - tâche : on tente d'instancier le robot GoPiGo3
- `instanciation_failed` : instanciation échouée
  - rôle : le robot n'est pas accessible et l'application ne peut poursuivre
  - tâche : on affiche un message d'erreur
- end : fin du programme
  - rôle : fin du programme
  - tâche : on affiche un message terminal

- `robot_integrity` : validation de l'intégrité du robot
    - rôle : le robot est instancié, on s'assure qu'il est intègre et fonctionnel
    - tâche : validation de l'intégrité du robot pour les constituants pouvant être validés
- `integrity_failed` : intégrité échouée
    - rôle : la validation de l'intégrité du robot a échoué, un composant du robot ne fonctionne pas
    - tâche : on fait clignoter les phares et les yeux en rouge pendant 5 secondes avec un cycle de 0.5 seconde (les phares et les yeux sont synchrones)
- `integrity_succeeded` : intégrité réussie
    - rôle : le robot est intègre, on poursuit le programme
    - tâche : on fait clignoter les yeux en vert pendant 3 secondes avec un cycle de 1.0 seconde (les yeux sont synchrones alors que les phares sont éteints)
- `shut_down_robot` : fin des opérations du robot
    - rôle : amorce de la fin du programme, on s'assure de laisser le robot dans un état valide et sécuritaire
    - tâche :
        - s'il y a des opérations à réaliser avec le robot avant de quitter, c'est ici qu'elles sont effectuées
        - on fait clignoter les yeux rapidement en jaune pendant les tâches de fermeture ou au moins pendant 3 secondes avec un cycle de 0.75 seconde (les yeux sont synchrones et les phares sont éteints)
- `home` : accueil
    - rôle : accueil principal du programme où ce dernier est en attente d'une instruction de l'usager
    - tâche : les yeux clignotent lentement en jaune avec un cycle de 1.5 seconde (les yeux clignotent en alternance et les phares sont éteints)
    - télécommande :
        - ok : quitte le programme
        - 1, 2, 3, … : démarre la tâche associée
- `task_`*`n`* : une tâche à faire
    - rôle :
        - sans connaître spécifiquement les tâches à venir, l'application doit considérer pouvoir démarrer *n* tâches de différentes natures – autrement dit, du point de vue de l'application principale, les tâches sont des abstractions sans conséquence pour le *main*
        - pour cette partie du développement, les tâches sont indépendantes de l'application
    - tâche : les tâches sont inconnues de l'application, mais elles ont tous un point en commun, les yeux clignotent lentement selon un patron déterminé et il est possible par l'usager de quitter la tâche et retourné à l'accueil
    - télécommande :
        - ok : quitte la tâche courante et retourne à l'accueil

Infrastructure logicielle

Ce projet doit porter une attention particulière à la structure logicielle développée. Il est important de réaliser une séparation entre l'application elle-même et les tâches à réaliser. En fait, on désire un minimum de dépendance entre le programme et chacune des tâches.

Pour y arriver, on vous demande de produire minimalement trois abstractions différentes :

1. une classe réalisant une encapsulation de haut niveau du robot
2. la possibilité d'ajouter simplement des tâches (les tâches doivent être une abstraction du point de vue de l'application)

3. une machine d'états générique utilisée dans divers contextes

Une attention particulière est portée au dernier point. En effet, on vous demande de faire un développement générique totalement indépendant de ce projet sous forme de librairie réutilisable. D'ailleurs, une conception vous est présentée pour cet élément du développement.

Pour les deux premiers points, vous devez produire vous-même ces abstractions.

Classe Robot

Vous devez faire la conception et l'implémentation d'une classe représentant le robot GoPiGo3. Cette classe doit offrir une interface de programmation performante sur le robot utilisé en classe.

Contrairement à la classe EasyGoPiGo3, votre classe doit offrir des opérations de plus haut niveau. Par exemple, il doit être possible d'utiliser des accesseurs afin de connaître l'état du robot pour différents dispositifs comme les phares, les yeux, etc.

Infrastructure de gestion des tâches

Votre infrastructure logicielle doit permettre l'insertion de tâche très simplement. Par exemple, il pourrait être possible d'ajouter la tâche par une simple fonction similaire à celle-ci : `...add_task(key, task)`

Machine d'états

Vous devez implémenter, sous forme de librairie générique, une machine d'états qui pourrait être utilisée dans n'importe quel contexte applicatif. Cette librairie devra être utilisée à divers endroits du logiciel. Sachez qu'il existe plusieurs opportunités où une telle approche est pertinente. Par exemple, les éléments suivants peuvent tirer avantage d'un tel outil : les deux phares, les deux yeux et l'application principale.

Ces exemples sont les plus évidents, mais sachez qu'il existe plusieurs autres possibilités où la machine d'état reste pertinente dans ce projet. Autrement dit, l'effort d'abstraction de la librairie et des outils que vous pouvez offrir dans la librairie profiteront à son usage.

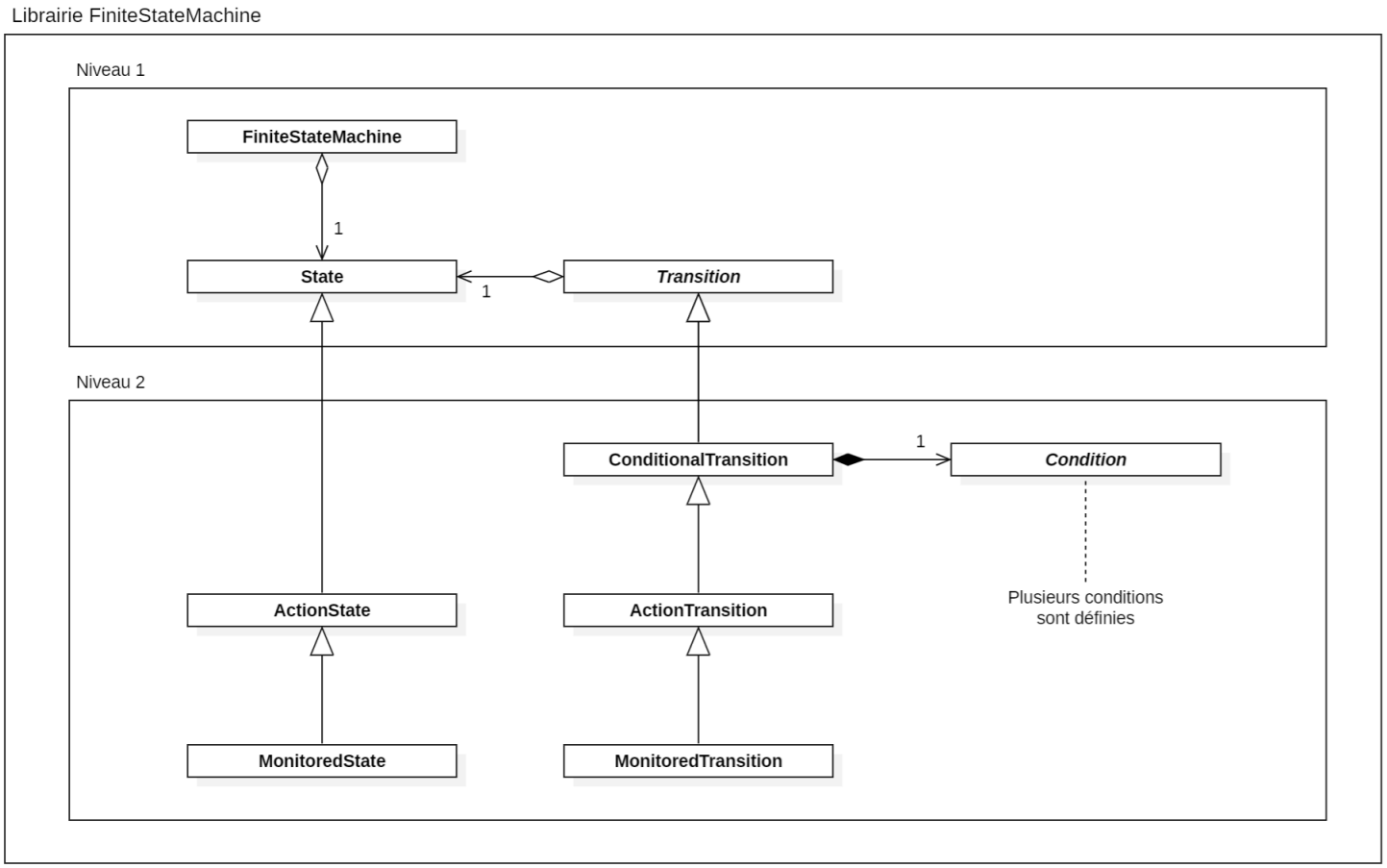
Évidemment, il existe plusieurs autres approches viables pour produire un tel logiciel, certaines sont même très pertinentes et intéressantes avec d'autres avantages et inconvénients. Toutefois, la machine d'état reste souvent un outil sous-exploité pour plusieurs types d'application.

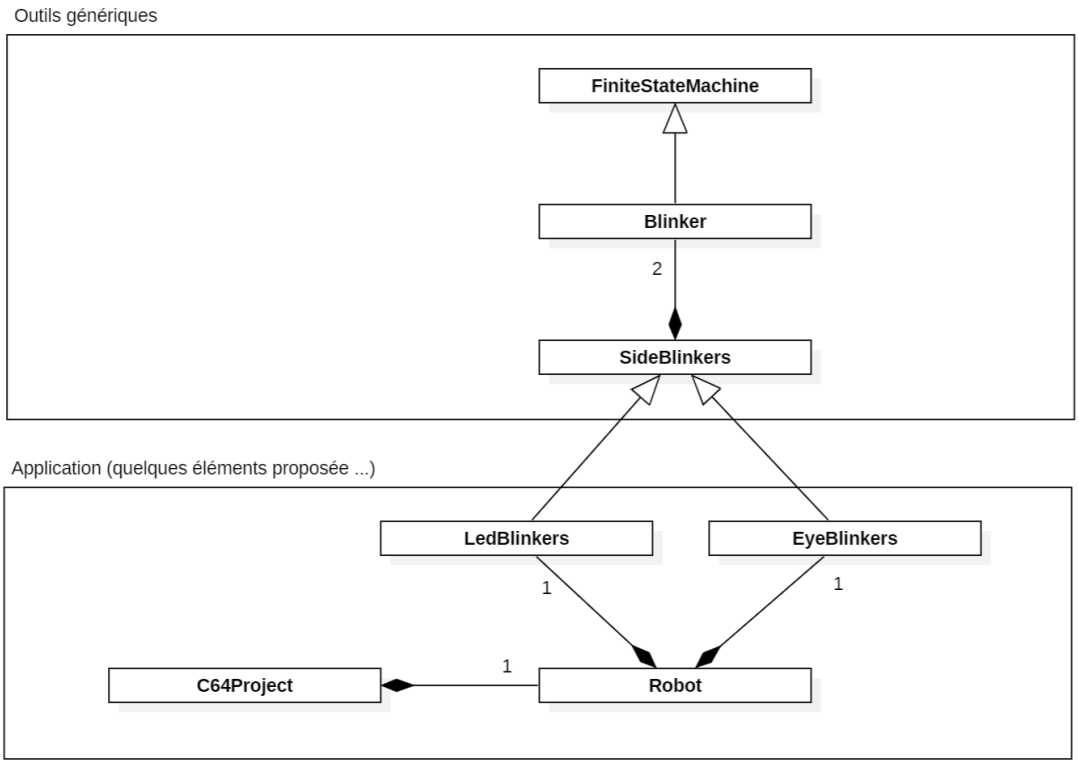
Vous avez à votre disposition la présentation d'une conception de niveau intermédiaire pour une telle librairie. Cette dernière s'appuie fortement sur le paradigme orienté objet afin de mettre l'emphase sur la modularité et flexibilité avec quelques compromis sur la performance. Vous avez une description sous forme de diagramme de classe (et un diagramme de séquence) de cette conception. Vous remarquerez la séparation complète entre les intentions de cette librairie et de l'application en cours.

Il faut comprendre que la librairie est complètement générique et n'est liée à aucun projet spécifique. La généricité est même existante au cœur de la librairie puisque cette dernière a été découpée en 2 parties fondamentales. Une autre partie est ajoutée pour se rapprocher des besoins applicatifs de ce projet; mais reste entièrement générique encore une fois. Donc, la librairie présentée est séparée en trois sous parties :

1. la partie fondamentale (nommée : librairie niveau 1)
2. une extension de la partie fondamentale avec la mise en place d'outils dont l'usage est quasi systématique (nommée : librairie niveau 2)
3. une extension supplémentaire se rapprochant des besoins de ce projet (nommé : outils génériques) réalise la conception de dispositifs clignotants :
    a. dispositif à 1 clignotant (nommé Blinker)
    b. dispositif à 2 clignotants (gauche droite, nommé SideBlinkers)

Voici une vue d'ensemble de la librairie proposée.





La classe FiniteStateMachine permet l'encapsulation globale de la machine d'état. Il est important de comprendre que cette classe est en fait l'engin d'exécution et permet la logique générique de cette approche.

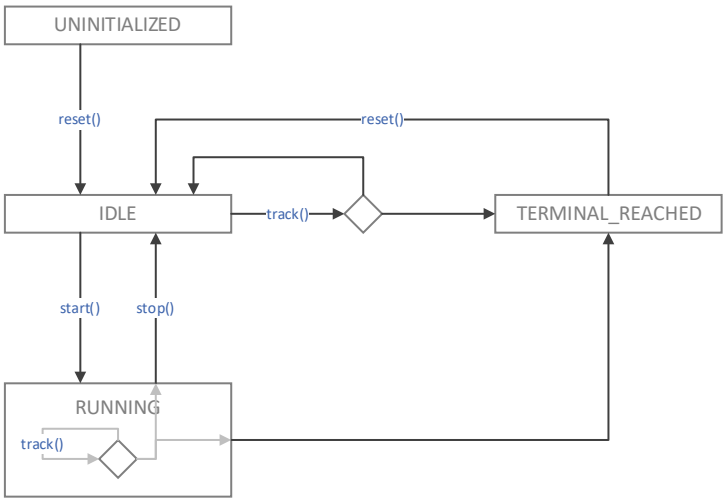
Pour son opération, cette classe fonctionne elle-même avec une structure interne de machine d'état. Une certaine confusion peut émerger de ce concept : une machine d'état gérant une machine d'état. On détermine ici une terminologie propre à cette classe permettant de lever les ambiguïtés possibles :

- état opérationnel : c'est l'action en cours de l'engin de résolution
- état applicatif : c'est l'état courant de la partie applicative

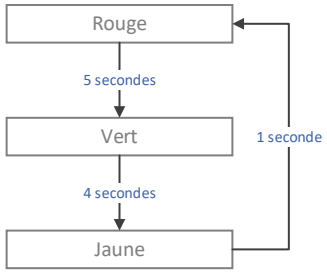
L'état opérationnel correspond à une machine d'état où 4 états existent et sont liés par les évènements de l'application :

- `UNINITIALIZED` : l'état initial n'a pas encore été engagé, l'état courant n'est pas défini et est invalide
- `IDLE` : l'état courant est valide et défini, toutefois l'engin est en attente
- `RUNNING` : l'état courant est valide
- `TERMINAL_REACHED` : l'état courant est valide et pointe vers un état terminal, l'engin ne peut plus faire de résolution supplémentaire

Ce sont les fonctions membres `track()`, `reset()`, `start()` et `stop()` qui articulent les opérations de la machine d'état.



Pour donner une exemple d'application, supposons qu'on désire réaliser une machine d'états représentant un feu de circulation. Il existe 3 états applicatifs : rouge, vert et jaune. Sur chacun de ces états, il existe une transition vers l'état approprié.



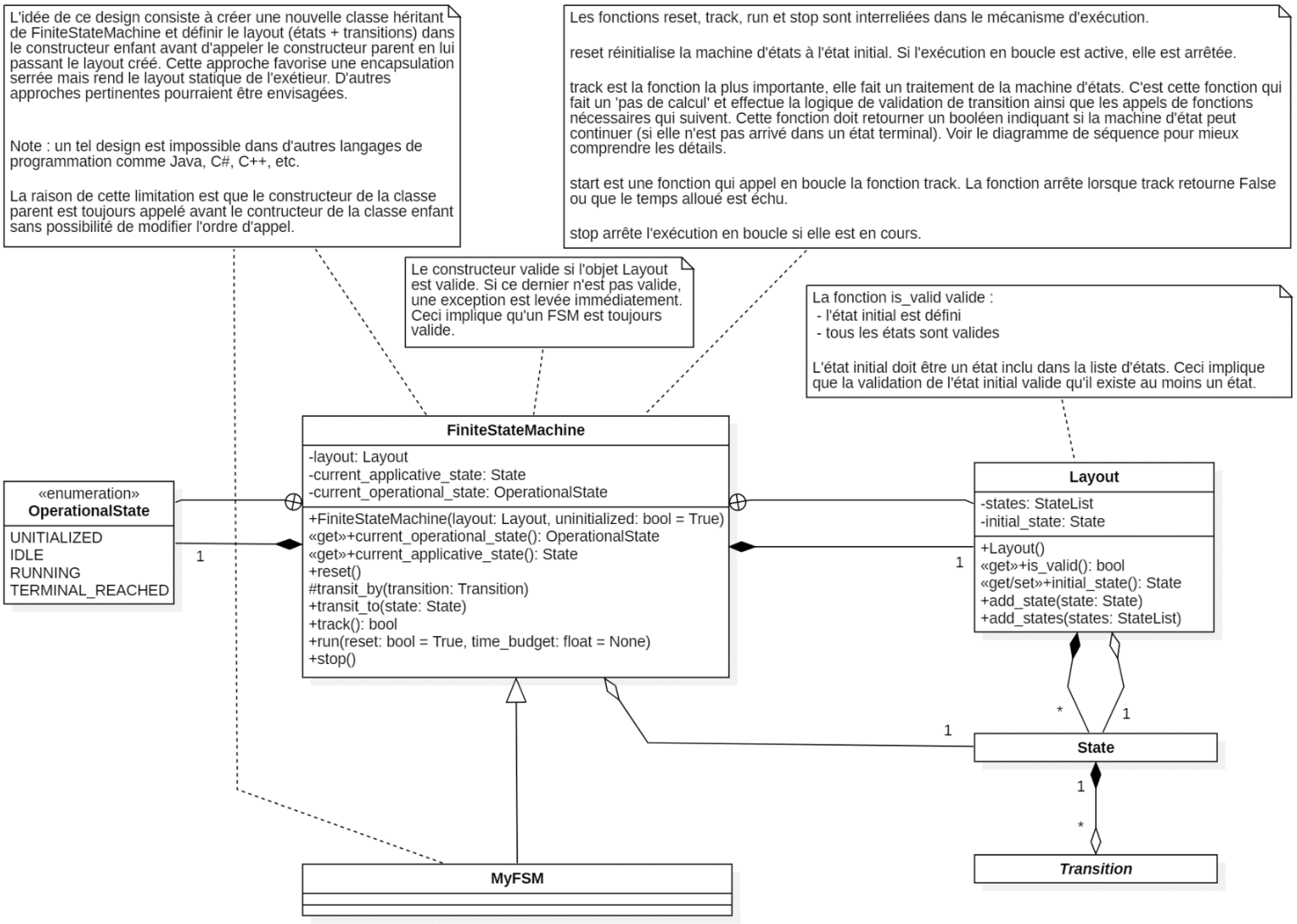
Le shéma précédent représente le `Layout` de l'application à réaliser : dans ce cas-ci, 3 états et 3 transitions.

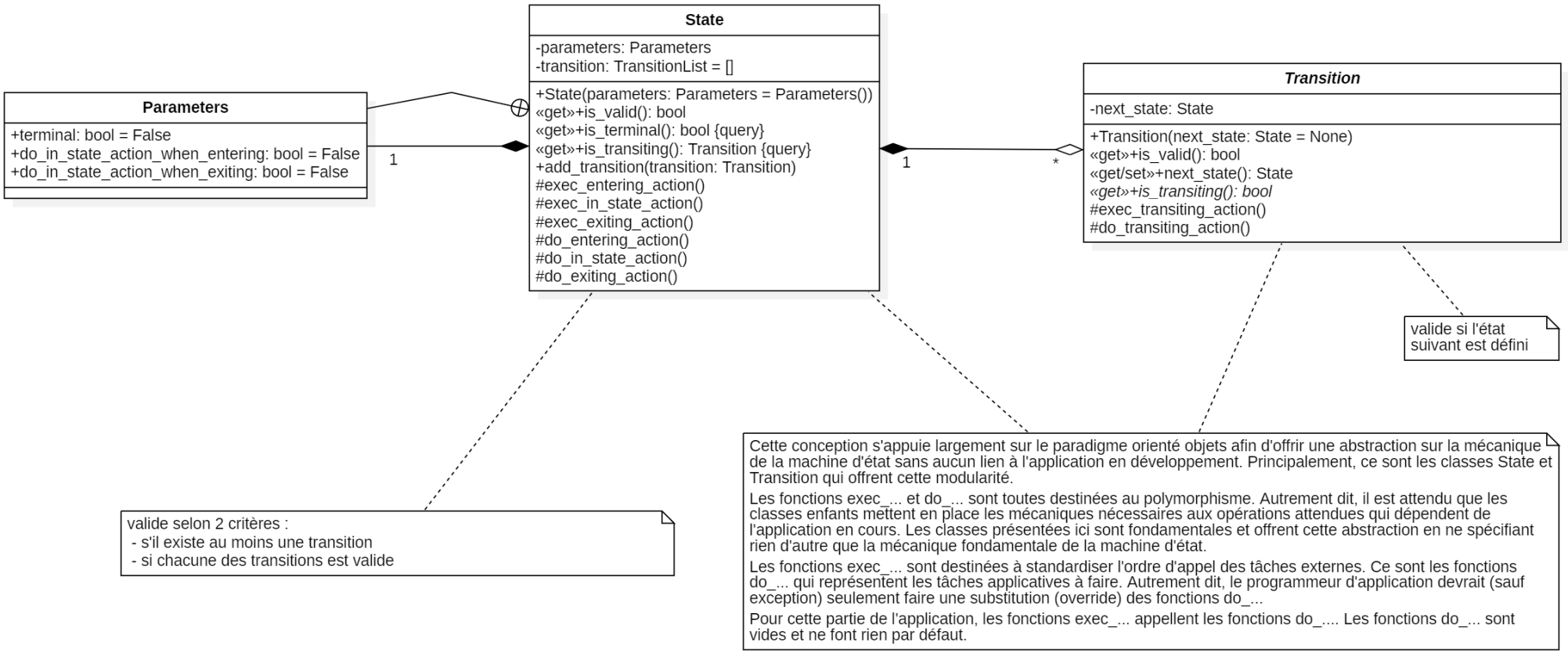
L'opération de cette machine d'état implique qu'il existe, à un temps donné, un seul état appelé état courant.

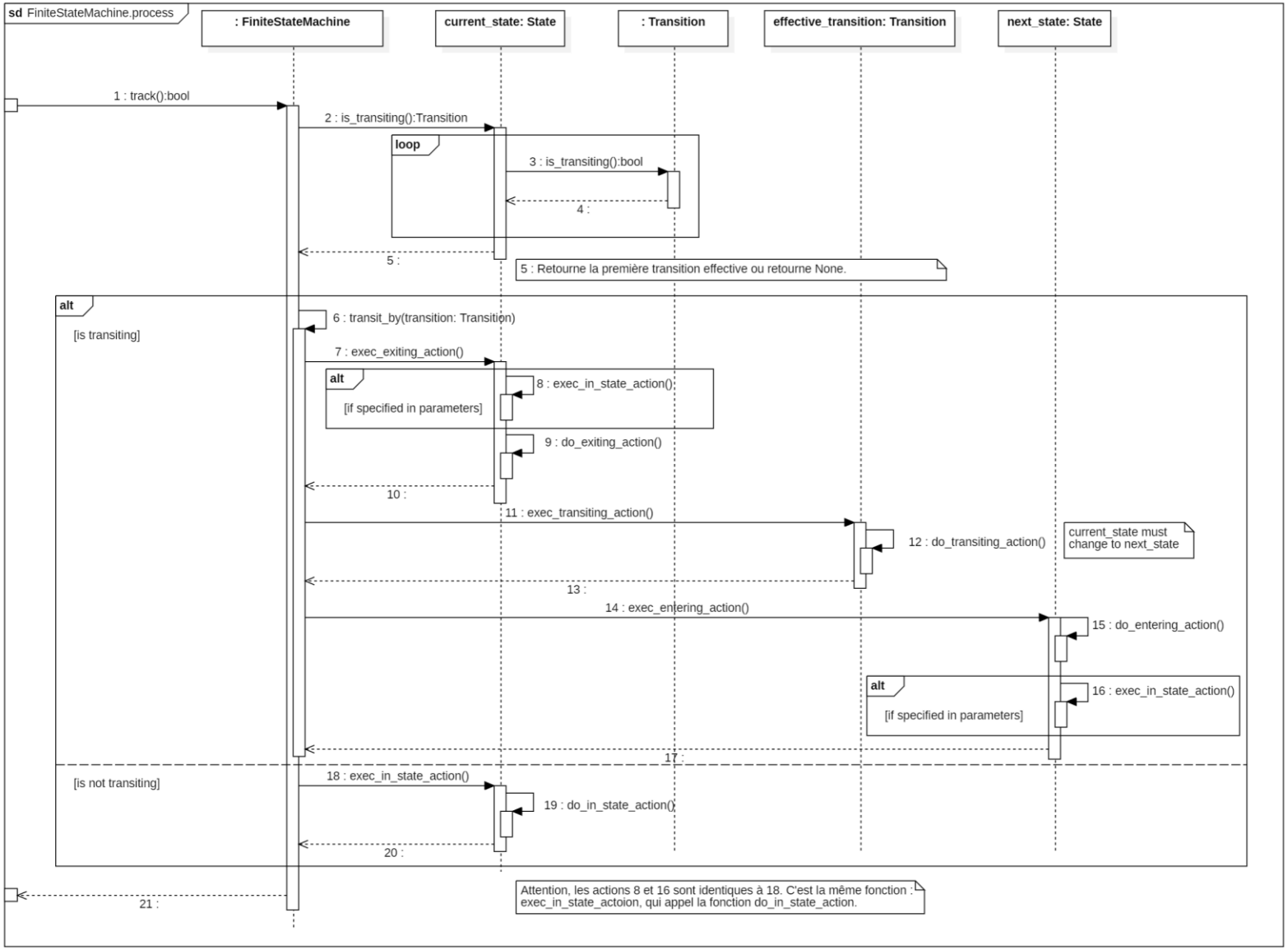
Un FSM est constitué de :

- un 'plan' (`Layout`) : définissant l'infrastructure statique de la machine d'état
- les états et leurs transitions : sont les constituants du `Layout` et définissent la partie applicative de la machine d'état
- un état applicatif initial : représente le premier état suivant l'appel de la méthode `reset`
- un état applicatif courant : représente l'état dynamique actuel
- un état opérationnel courant : représente quel est l'état interne de la machine d'état (représenté par une énumération et opéré par ces fonctions utilitaires : `reset`, `track`, `start` et `stop`)

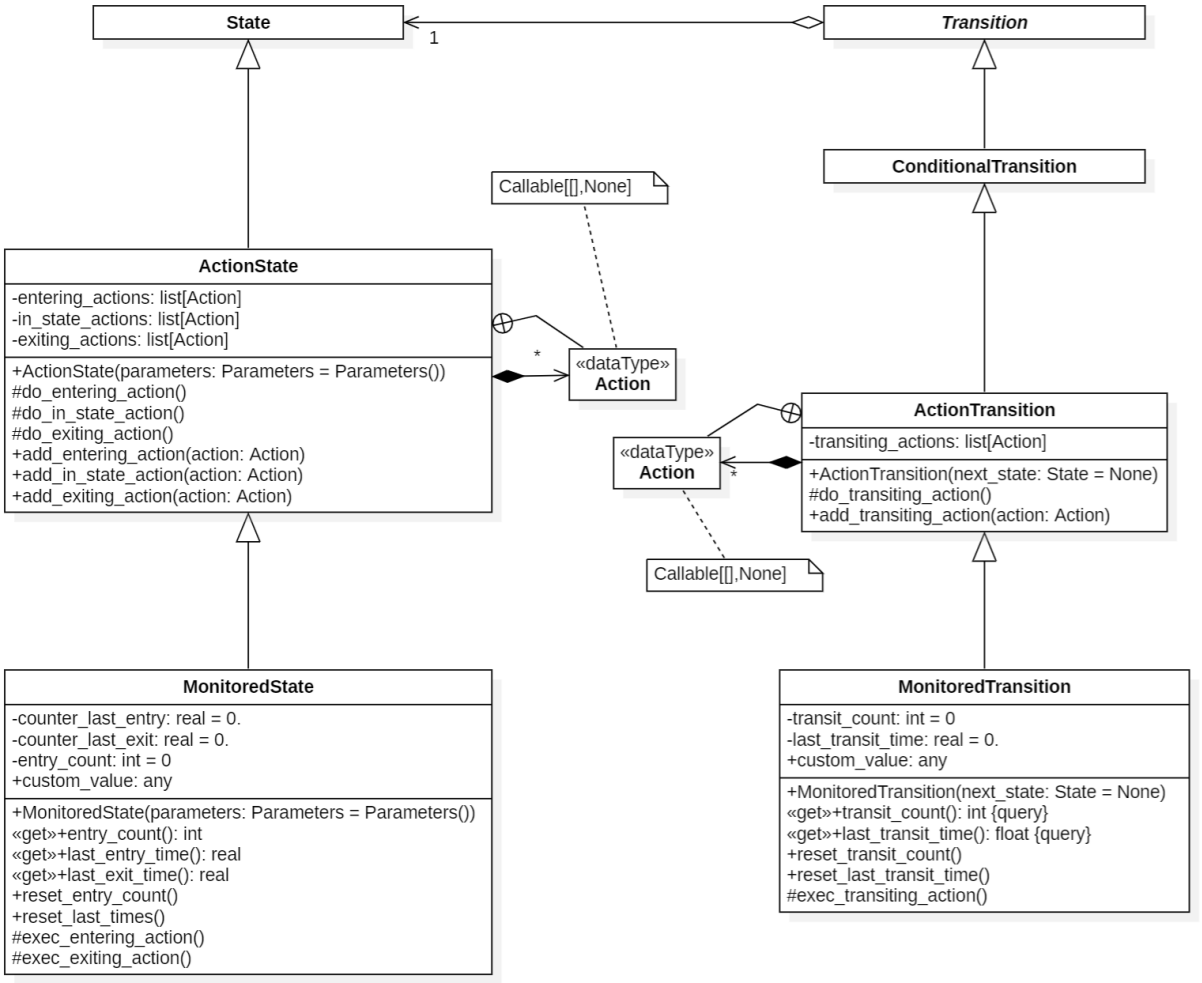
Sur les pages qui suivent, vous retrouverez les diagrammes de classes plus détaillés. Un diagramme de séquence permet de mieux comprendre le déroulement de la fonction primordiale de cette application.


L'idée de ce design consiste à créer une nouvelle classe héritant de FiniteStateMachine et définir le layout (états + transitions) dans le constructeur enfant avant d'appeler le constructeur parent en lui passant le layout créé. Cette approche favorise une encapsulation serrée mais rend le layout statique de l'exétieur. D'autres approches pertinentes pourraient être envisagées.
Note : un tel design est impossible dans d'autres langages de programmation comme Java, C#, C++, etc.
La raison de cette limitation est que le constructeur de la classe parent est toujours appelé avant le contructeur de la classe enfant sans possibilité de modifier l'ordre d'appel.
Les fonctions reset, track, run et stop sont interreliées dans le mécanisme d'exécution.
reset réinitialise la machine d'états à l'état initial. Si l'exécution en boucle est active, elle est arrêtée.
track est la fonction la plus importante, elle fait un traitement de la machine d'états. C'est cette fonction qui fait un 'pas de calcul' et effectue la logique de validation de transition ainsi que les appels de fonctions nécessaires qui suivent. Cette fonction doit retourner un booléen indiquant si la machine d'état peut continuer (si elle n'est pas arrivé dans un état terminal). Voir le diagramme de séquence pour mieux comprendre les détails.
start est une fonction qui appel en boucle la fonction track. La fonction arrête lorsque track retourne False ou que le temps alloué est échu.
stop arrête l'exécution en boucle si elle est en cours.
Le constructeur valide si l'objet Layout est valide. Si ce dernier n'est pas valide, une exception est levée immédiatement. Ceci implique qu'un FSM est toujours valide.
La fonction is_valid valide :
- l'état initial est défini
- tous les états sont valides
L'état initial doit être un état inclu dans la liste d'états. Ceci implique que la validation de l'état initial valide qu'il existe au moins un état.
«enumeration»
OperationalState
UNITIALIZED
IDLE
RUNNING
TERMINAL_REACHED
1
FiniteStateMachine
-layout: Layout
-current_applicative_state: State
-current_operational_state: OperationalState
+FiniteStateMachine(layout: Layout, uninitialized: bool = True)
«get»+current_operational_state(): OperationalState
«get»+current_applicative_state(): State
+reset()
#transit_by(transition: Transition)
+transit_to(state: State)
+track(): bool
+run(reset: bool = True, time_budget: float = None)
+stop()
Layout
-states: StateList
-initial_state: State
+Layout()
«get»+is_valid(): bool
«get/set»+initial_state(): State
+add_state(state: State)
+add_states(states: StateList)
1
*
1
1
State
1
*
Transition
MyFSM


State
-parameters: Parameters
-transition: TransitionList = []
+State(parameters: Parameters = Parameters())
«get»+is_valid(): bool
«get»+is_terminal(): bool {query}
«get»+is_transiting(): Transition {query}
+add_transition(transition: Transition)
#exec_entering_action()
#exec_in_state_action()
#exec_exiting_action()
#do_entering_action()
#do_in_state_action()
#do_exiting_action()
Parameters
+terminal: bool = False
+do_in_state_action_when_entering: bool = False
+do_in_state_action_when_exiting: bool = False
1
1
*
Transition
-next_state: State
+Transition(next_state: State = None)
«get»+is_valid(): bool
«get/set»+next_state(): State
«get»+is_transiting(): bool
#exec_transiting_action()
#do_transiting_action()
valide si l'état suivant est défini
valide selon 2 critères :
- s'il existe au moins une transition
- si chacune des transitions est valide
Cette conception s'appuie largement sur le paradigme orienté objets afin d'offrir une abstraction sur la mécanique de la machine d'état sans aucun lien à l'application en développement. Principalement, ce sont les classes State et Transition qui offrent cette modularité.
Les fonctions exec_... et do_... sont toutes destinées au polymorphisme. Autrement dit, il est attendu que les classes enfants mettent en place les mécaniques nécessaires aux opérations attendues qui dépendent de l'application en cours. Les classes présentées ici sont fondamentales et offrent cette abstraction en ne spécifiant rien d'autre que la mécanique fondamentale de la machine d'état.
Les fonctions exec_... sont destinées à standardiser l'ordre d'appel des tâches externes. Ce sont les fonctions do_... qui représentent les tâches applicatives à faire. Autrement dit, le programmeur d'application devrait (sauf exception) seulement faire une substitution (override) des fonctions do_...
Pour cette partie de l'application, les fonctions exec_... appellent les fonctions do_.... Les fonctions do_... sont vides et ne font rien par défaut.


sd FiniteStateMachine.process
: FiniteStateMachine
current_state: State
: Transition
effective_transition: Transition
next_state: State
1 : track():bool
2 : is_transiting():Transition
loop
3 : is_transiting():bool
4 :
5 :
5 : Retourne la première transition effective ou retourne None.
alt
[is transiting]
6 : transit_by(transition: Transition)
7 : exec_exiting_action()
alt
[if specified in parameters]
8 : exec_in_state_action()
9 : do_exiting_action()
10 :
11 : exec_transiting_action()
12 : do_transiting_action()
current_state must change to next_state
13 :
14 : exec_entering_action()
15 : do_entering_action()
alt
[if specified in parameters]
16 : exec_in_state_action()
17 :
[is not transiting]
18 : exec_in_state_action()
19 : do_in_state_action()
20 :
21 :
Attention, les actions 8 et 16 sont identiques à 18. C'est la même fonction : exec_in_state_actoion, qui appel la fonction do_in_state_action.

Librairie FiniteStateMachine – Niveau 2 – Extensions de la section fondamentale – Partie 1 / 2



Les classes ActionState et ActionTransition permettent une interface de programmation flexible au programmeur utilisateur.

Elles permettent de créer des états et des transitions flexibles sans héritage. On offre ici, via le concept d'action, la possibilité de réaliser des actions plutôt que d'imposer l'héritage comme mécanisme d'extension.

Ces actions sont simplement des Callable qui ne prennent aucun paramètre et ne retourne rien.

Il est donc possible de créer des états et transitions en simplement ajouter des actions à réaliser lors des opérations importantes (entering, in_state, exiting, transiting).

Chacune de ces classes offrent des listes d'actions à faire qui sont facilement accessibles et utilisables.

Il est important de comprendre que cette approche est complémentaire à l'héritage. Il reste tout à fait pertinent d'utilliser ce mécanisme et celui du polymorphisme pour étendre les possibilités des actions possibles. En fait, une bonne conception saura quand utiliser l'un ou l'autre.

Les classes MonitoredState et MonitoredTransition offrent des classes utilitaires implémentant des interfaces de programmation liées aux tâches les plus communes.

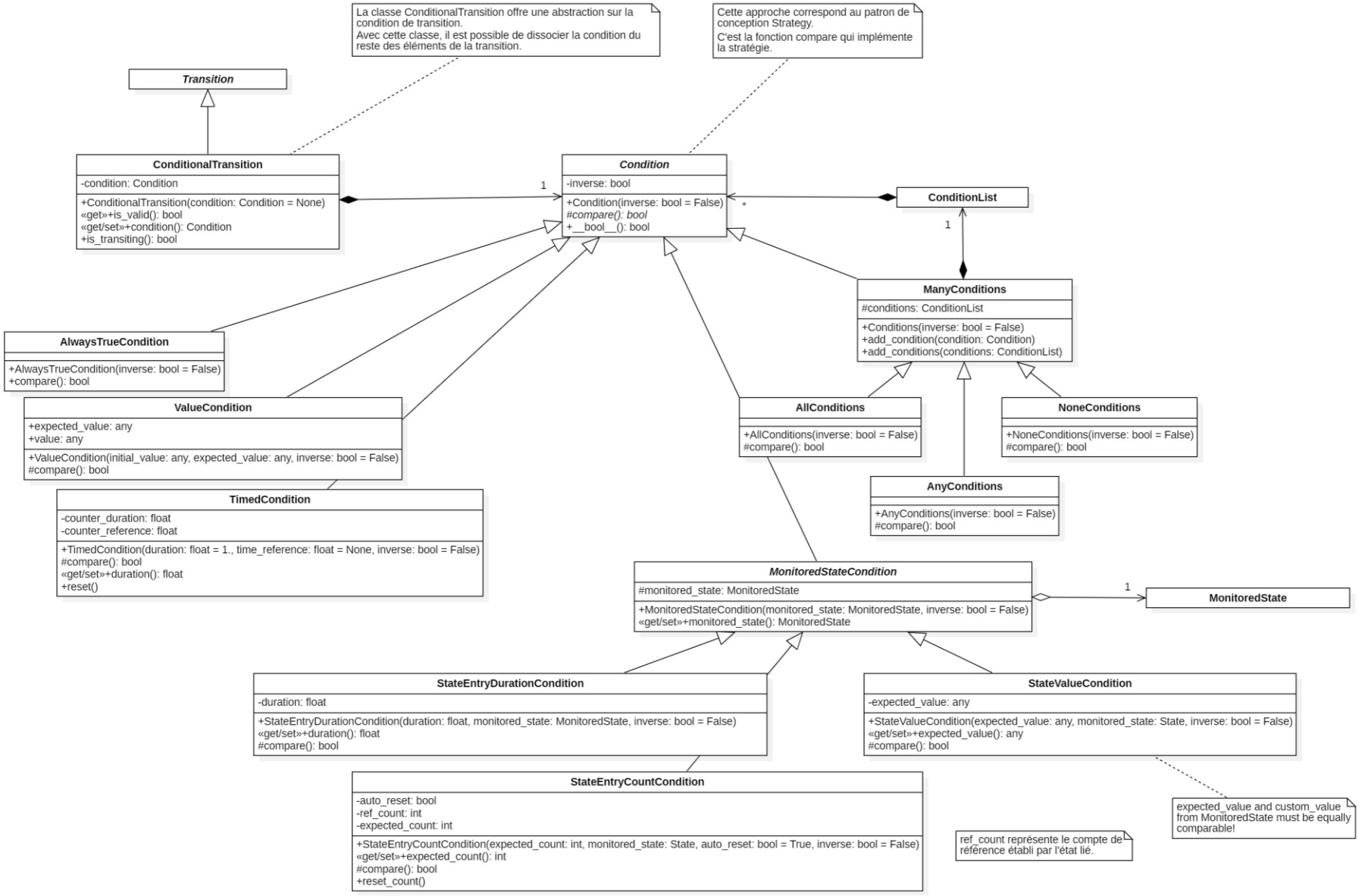
Elles offrent les mécanismes supplémentaires suivants :

- les temps d'activité :
  - la dernière fois qu'un état est devenu l'état courant :
    - temps d'entrée
    - temps de sortie
  - la dernière fois que la transition a été effective
- le nombre de fois :
  - que l'état est devenu l'état courant (le nombre d'entrée)
  - que la transition a été effective
- une valeur personnalisé

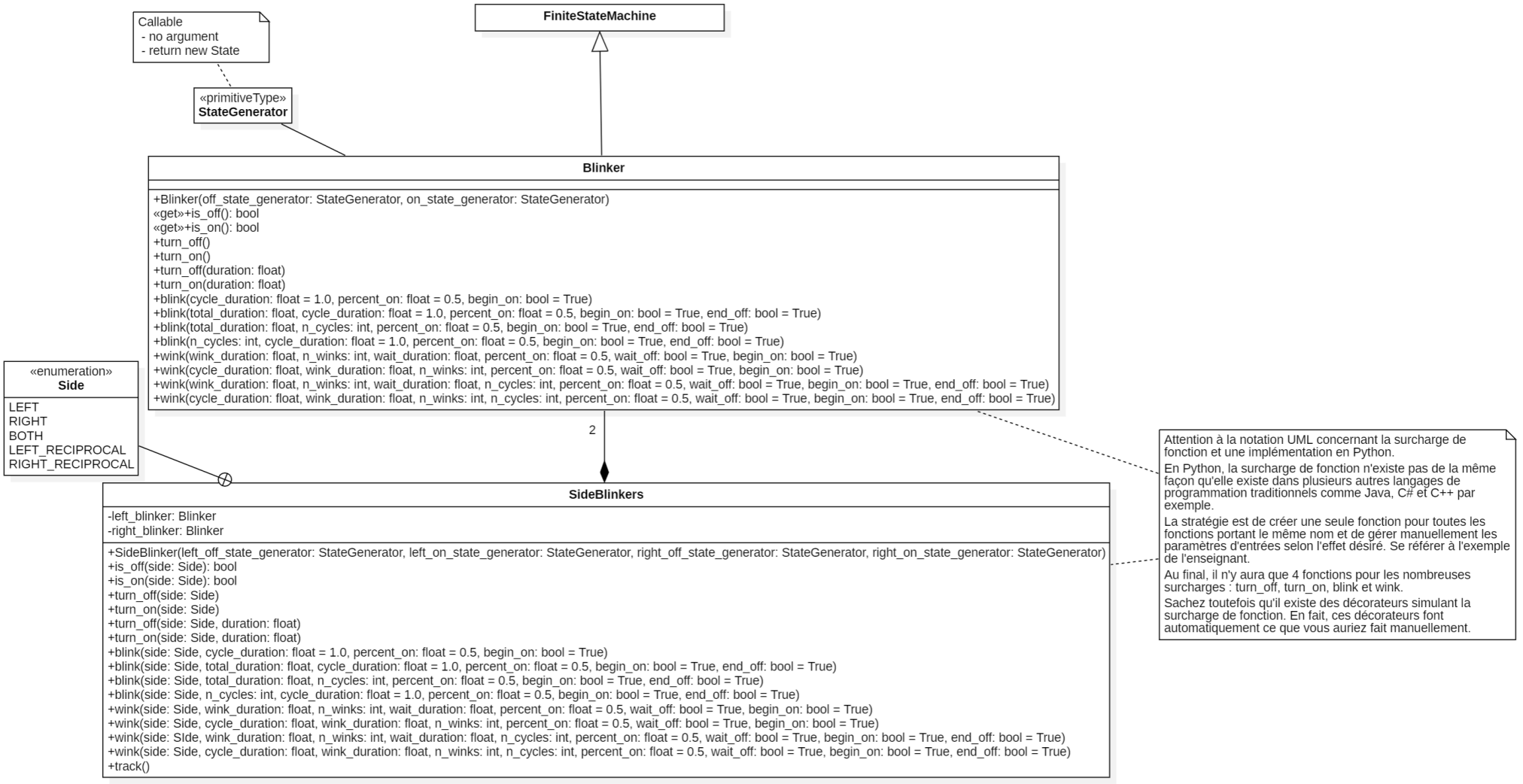
Ces outils sont essentiels pour automatiser des transitions conditionnelles à ces critères :

- transit après un certain temps (le temps)
- transit après un certain nombre d'occurrence (le nombre de fois)
- transit selon une certaine condition (valeur personnalisée)

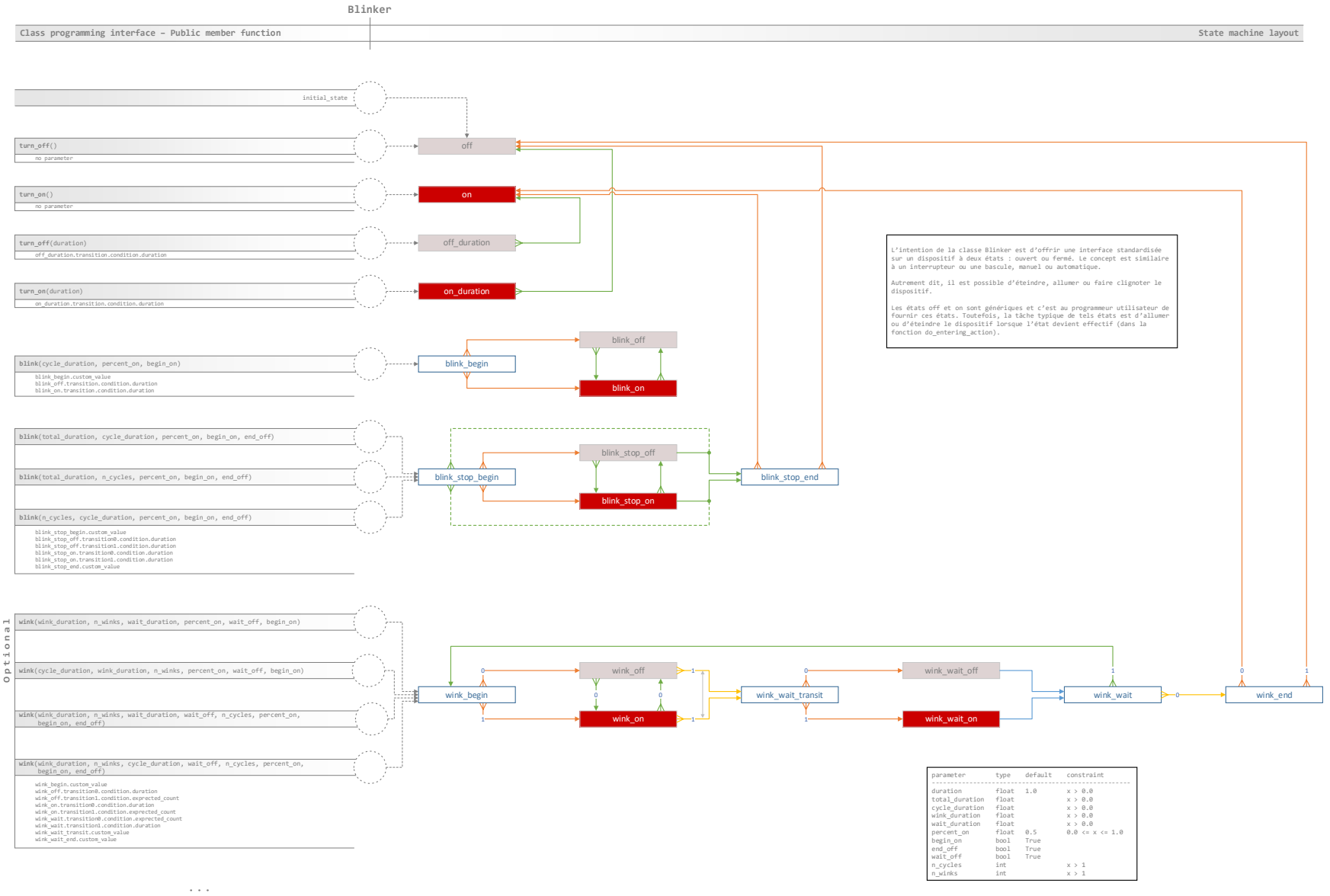
Librairie FiniteStateMachine – Niveau 2 – Extensions de la section fondamentale – Partie 2 / 2

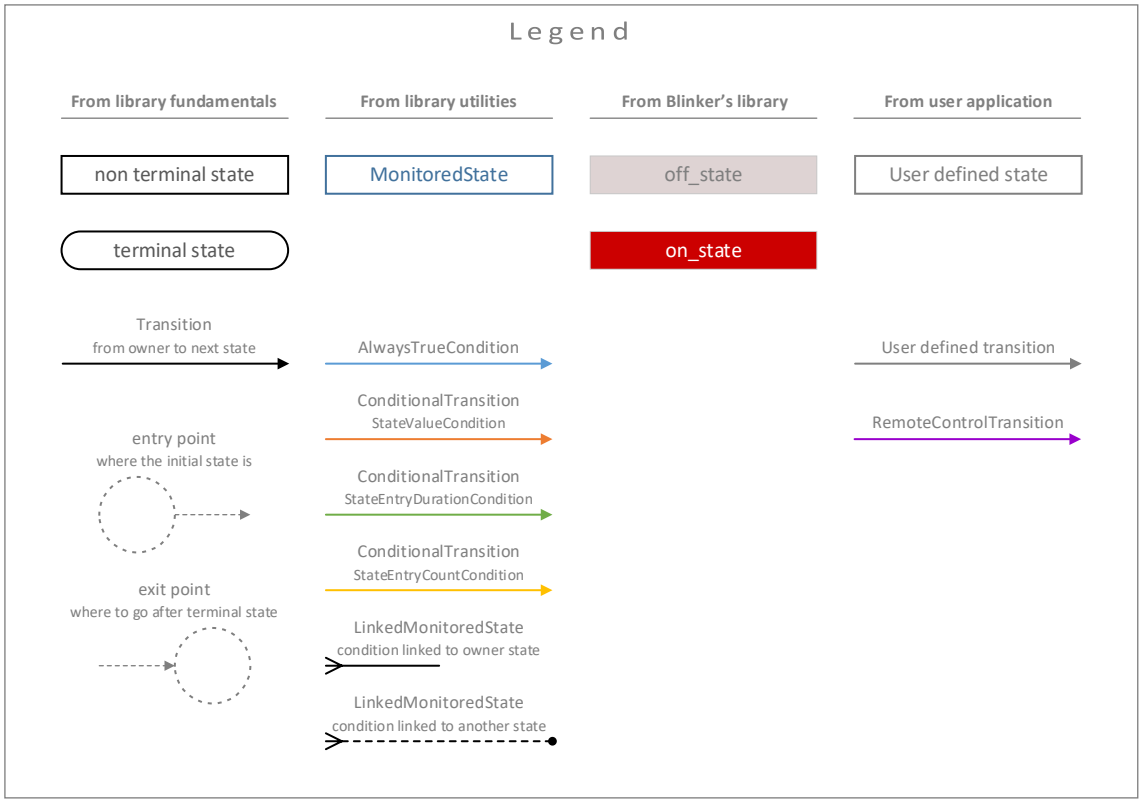


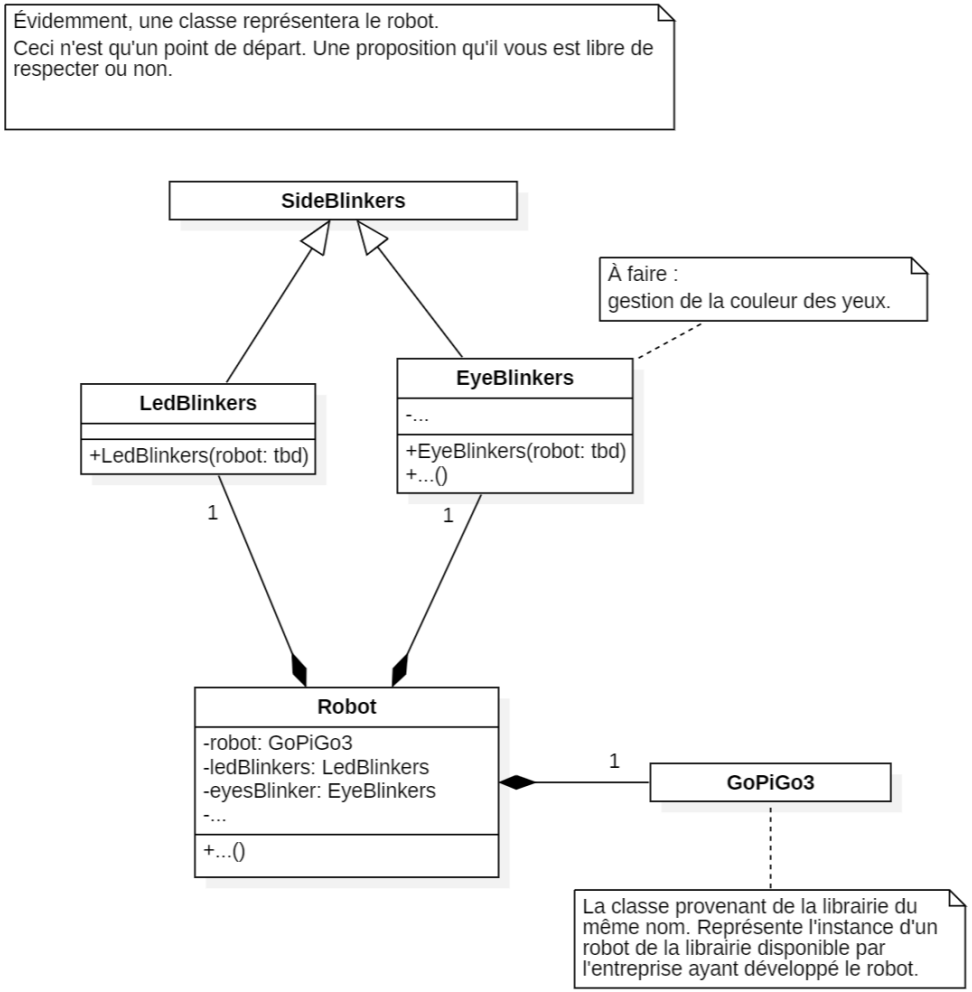
Librairie FiniteStateMachine – Niveau 3 – Extensions supplémentaires – Partie 1 / 3



Librairie FiniteStateMachine – Niveau 3 – Extensions supplémentaires – Partie 2 / 3




Legend
From library fundamentals
From library utilities
From Blinker's library
From user application
non terminal state
MonitoredState
off_state
User defined state
terminal state
on_state
Transition
from owner to next state
AlwaysTrueCondition
User defined transition
ConditionalTransition
StateValueCondition
RemoteControlTransition
entry point
where the initial state is
ConditionalTransition
StateEntryDurationCondition
ConditionalTransition
StateEntryCountCondition
exit point
where to go after terminal state
LinkedMonitoredState
condition linked to owner state
LinkedMonitoredState
condition linked to another state


Évidemment, une classe représentera le robot.
Ceci n'est qu'un point de départ. Une proposition qu'il vous est libre de respecter ou non.
SideBlinkers
À faire :
gestion de la couleur des yeux.
LedBlinkers
+LedBlinkers(robot: tbd)
EyeBlinkers
-...
+EyeBlinkers(robot: tbd)
+...()
1
1
Robot
-robot: GoPiGo3
-ledBlinkers: LedBlinkers
-eyesBlinker: EyeBlinkers
-...
+...()
1
GoPiGo3
La classe provenant de la librairie du même nom. Représente l'instance d'un robot de la librairie disponible par l'entreprise ayant développé le robot.

Application du projet – Partie 2 / 2

Vous devez créer une classe nommée C64Project.

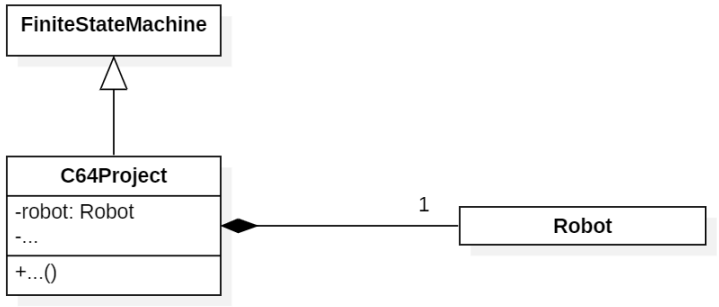
Cette classe est l'élément central du projet. C'est cet élément qu'il faut instancier.

Autrement dit, le __main__ de votre projet pourrait être aussi simple que :

```
c64 = C64Project()
c64.start()
```

Il existe plusieurs façons de créer une telle classe et c'est à vous de déterminer une bonne façon de structurer votre projet.

N'oubliez pas qu'on désire pouvoir ajouter des nouvelles tâches simplement, de façon modulaire sans modifier significativement du code principal. Autrement dit, on veut séparer entièrement l'application principale des tâches à faire.

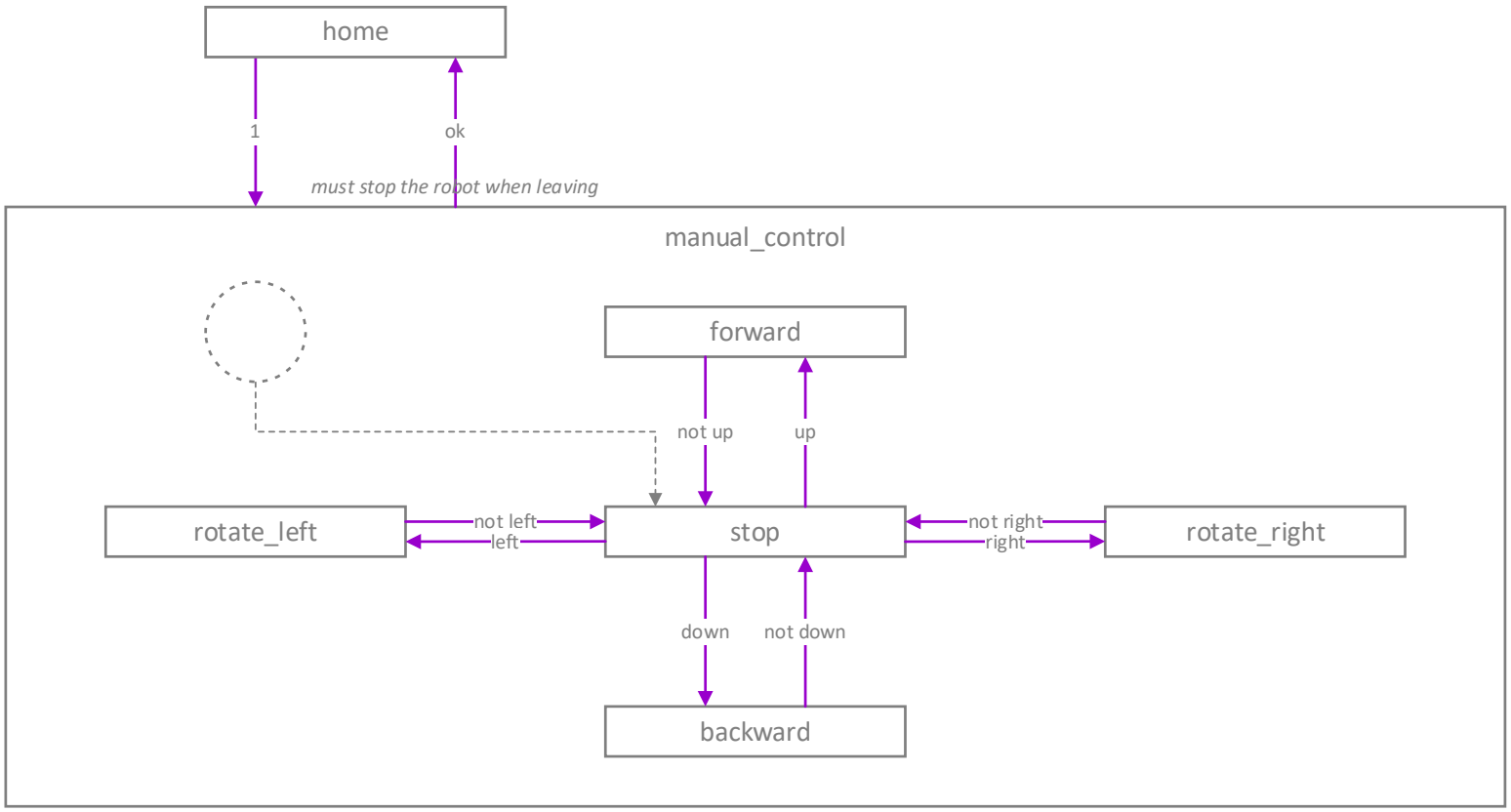


Tâche 1 – Contrôle manuel

La première tâche, nommée `manual_control`, consiste à offrir une interface utilisateur offrant la possibilité de contrôler manuellement le robot.

En fait, cette tâche consiste à contrôler le robot simplement similairement aux premiers exercices réalisés en début de session.

Plus spécifiquement, le diagramme suivant représente le comportement à implémenter.



Chaque état doit effectuer ces opérations :

- `home` : l'état accueil déjà présenté
- `manual_control` : première tâche – contrôle manuel du robot
  - rôle : l'usager peut contrôler le robot manuellement avec la télécommande
  - tâche :
    - l'utilisation des flèches sur la télécommande fait bouger le robot
    - les deux yeux clignotent en alternance de rouge à bleu à un cycle de 1.0 seconde (on désire un effet de fondu similaire à des gyrophares de police moderne)
    - ce clignotement est effectué en tout temps peu importe l'opération en cours et cesse dès que l'état `manual_control` quitte et retourne à l'état `home`
- `stop` : arrêt du robot
  - tâche :
    - le robot est mis à l'arrêt
    - les phares sont éteints

- `forward` : le robot avance
    - tâche :
        - le robot est mis en marche vers l'avant
        - les phares clignotent de façon synchronisée à une fréquence de 1.0 sec avec 25% ouvert
- `backward` : le robot recule
    - tâche :
        - le robot est mis en marche vers l'arrière
        - les phares clignotent de façon synchronisée à une fréquence de 1.0 sec avec 75% ouvert
- `rotate_left` : le robot effectue une rotation à gauche
    - tâche :
        - le robot est mis en marche et effectue une rotation vers la gauche (la roue de droite est mise en marche alors que la roue de gauche est arrêtée)
        - le phare de gauche clignote à une fréquence de 1.0 sec avec 50% ouvert
        - le phare de droite est éteint
- `rotate_right` : le robot effectue une rotation à droite
    - tâche :
        - le robot est mis en marche et effectue une rotation vers la droite (la roue de gauche est mise en marche alors que la roue de droite est arrêtée)
        - le phare de droite clignote à une fréquence de 1.0 sec avec 50% ouvert
        - le phare de gauche est éteint

Considération importante

Compréhension de la librairie

Vous devez savoir que la création d'une librairie générique est toujours plus complexe qu'il n'y paraît. Dans plusieurs situations, on doit se détacher du développement d'une application spécifique. Cette abstraction est souvent difficile à faire. De plus, la généricité de l'approche vient souvent avec des approches logicielles plus sophistiquées et ces dernières obligent le développeur à approfondir ses connaissances sur des notions plus avancées du langage de programmation.

Il est primordial que vous posiez des questions à l'enseignant dès qu'il existe des points d'ombre sur votre compréhension de la conception proposée. Le déchiffrement d'un tel schéma UML n'est pas facile et requiert en général une bonne discussion avec les personnes impliquées. En entreprise, ces discussions sont fréquentes et nécessaires. Vous devez savoir que, souvent, la conception réalisée vient avec des choix qui s'expliquent difficilement sur ces diagrammes. Certains choix dans l'approche sous-tendent des subtilités qui peuvent être importantes, autant pour la structure que pour des limitations techniques. C'est pourquoi il est attendu que vous posiez des questions le plus rapidement possible.

Réalisation de la bibliothèque ainsi que de son usage pour la réalisation du projet

Vous devez vous inspirer de la conception donnée pour la machine d'états. Toutefois, cette implémentation possède plusieurs petits défis qu'il vous est possible de réaliser différemment. Autrement dit, vous pouvez décider d'implémenter des variantes de la solution proposée. Néanmoins, sachez que toute autre implémentation sera évaluée sur son abstraction, sa séparation avec le projet en cours, sa flexibilité et sa modularité.

De plus, la conception proposée n'est pas complète pour ce projet. Après avoir réalisé votre bibliothèque, vous allez devoir l'étendre par héritage afin de produire des états et des transitions adaptés au contexte applicatif de votre projet. Autrement dit, vous devez être capable de profiter de cet outil et, par le biais du polymorphisme, offrir des outils flexibles et réutilisables pour réaliser élégamment votre projet.

Contraintes

Plusieurs contraintes sont à respecter pour ce travail :

- Vous devez travailler en équipe de 4.
- Il est très important que tous les membres de l'équipe travaillent équitablement, car l'évaluation finale tient compte de plusieurs critères, dont la répartition de la tâche de travail.

Stratégie d'évaluation

L'évaluation se fera en 2 parties. D'abord, l'enseignant évaluera le projet remis et assignera une note de groupe pour le travail. Ensuite, chaque équipe devra remettre un fichier Excel dans lequel sera soigneusement reportée une cote représentant la participation de chaque étudiant dans la réalisation du projet. Cette évaluation est faite en équipe et un consensus doit être trouvé.

Une pondération appliquée sur ces deux évaluations permettra d'assigner les notes finales individuelles.

Ce projet est long et difficile. Il est conçu pour être réalisé en équipe. L'objectif est que chacun prenne sa place et que chacun laisse de la place aux autres.

Ainsi, trois critères sont évalués :

- participation (présence en classe, participation active, laisse participer les autres, pas toujours en train d'être sur Facebook ou sur son téléphone, concentré sur le projet, pas en train de faire des travaux pour d'autres cours …)
- réalisation (répartition du travail réalisé : conception, modélisation, rédaction de script, documentation …)
- impact (débrouillardise, initiative, amène des solutions pertinentes, motivation d'équipe …)

Rapport

Vous devez produire un petit rapport en format `Markdown` répondant à ces questions :

- Qui sont les membres de l'équipe?
- Concernant chacun des trois niveau de la bibliothèque `FiniteStateMachine` :
    - donnez, en pourcentage, un estimé de la proximité de ce que vous avez réalisé et la conception donnée – autrement dit, à quel point votre travail réalise ce qui a été demandé
    - si votre pourcentage est inférieur à 100%, nommez quels sont les éléments réalisés autrement et, très succinctement, expliquez quelle est votre approche
- Donnez une courte description de l'infrastructure réalisée pour les éléments suivants (on cherche à comprendre quelles sont les abstractions réalisées) :
    - classe représentant le robot :
        - validation de l'intégrité du robot
        - gestion de la télécommande
        - gestion du télémètre et de son servo moteur
        - gestion des moteurs
        - gestion de la couleur pour les yeux (la classe `EyeBlinkers`)
    - structure générale du logiciel :
        - quelle est l'infrastructure générale du logicielle
        - capacité modulaire d'insertion d'une nouvelle tâche
- Quels sont les patrons de conception utilisés à travers ce projet. On cherche autant les patrons de conception évident et bien défini que ceux plus subtiles et utilisant des implémentations moins formelles.
- Expliquez quel est l'impact de l'utilisation du patron de conception machine d'état sur le développement du projet. On cherche à comprendre les avantages et inconvénients du patron de conception. Discutez aussi des différences et de l'impact si nous avions décidé de rien utiliser pour réaliser les mêmes fonctionnalités.
- Y a-t-il d'autres éléments d'abstraction sur lesquels que vous aimeriez porter une attention particulière?

Sachez qu'il est attendu que vous soyez extrêmement concis et précis pour ce rapport. On ne désire aucun texte philosophique ou de remplissage. On cherche un texte très technique présenté sous forme de puces pertinentes.

Remise

Vous devez créer un fichier de format zip dans lequel vous insérez :

- votre rapport en format `md`
- votre fichier de travail nommé `C64Projet1.ipynb`
- votre fichier d'autoévaluation d'équipe `C64AutoevaluationEquipe.xlsx`

Vous devez remettre votre projet une seule fois sur Lea après avoir nommé votre fichier :

`NomPrenomEtudiant1_NomPrenomEtudiant2[_NomEtudiantN].zip`