1 Simulation de comportements

L'acquisition de données réelles, concernant l'utilisation d'un système, n'est pas toujours aisée et nécessite un temps d'utilisation du système long afin d'avoir un ensemble de données représentatif de la manière dont il est utilisé. De plus, la reproductibilité de ce comportement, dans un cadre d'utilisation réelle, n'est pas stable, les utilisateurs auront toujours quelques variations comportementales qui peuvent perturber l'étude et l'analyse de ces comportements.

Afin de pallier ce problème, nous avons modélisé et implémenté un simulateur, il permet de simuler des comportements ou d'en rejouer des réels enregistrés. Ce simulateur, que nous qualifions de "Simulateur à Invocation Discrète (SID)" se base sur les concepts des Simulateurs à Évènement Discret (SED). La différence notoire entre ces deux systèmes est qu'un SED génère des événements selon des lois de probabilité, ces derniers sont stockés dans des files d'attente puis traités par différents serveurs; dans le cas de SID, des méthodes d'objets du système sont directement invoquées selon des lois de probabilité. Notons cependant, que la nuance est conceptuelle, l'un pouvant implémenter l'autre et réciproquement. La conception du SED se décompose en trois niveaux présentés ci-après: la gestion du temps, la gestion des actions et pour finir le simulateur à proprement parler.

1.1 Timer

La gestion du temps, appelée "timer", permet la génération des laps de temps à attendre avant une invocation. Cette conception est générique et se base sur un patron composite, ce qui permet d'enrichir les méthodes de génération de laps de temps. La figure 1 illustre cette conception. Un "timer" est une interface qui implémente un "iterator". Ainsi le "timer" fournit, au travers de la méthode "next()", le prochain laps de temps à attendre en fonction de sa définition. Plusieurs implémentations de "timer" ont déjà été réalisées.

- "Random" est un générateur de temps en se basant sur des lois de probabilité: Poisson, Gaussienne, Exponentielle ou uniforme. D'autres pouvant être aisément ajoutées.
- "Periodic" est un générateur de temps en se basant sur une période fixe, avec la possibilité d'un démarrage différé défini par l'attribut "at".
- "OneShot" est un générateur de temps qui ne donnera qu'une seule valeur, puis s'arrêtera.
- "Date" fournit des laps de temps définis à partir d'une liste de date, il permet notamment de rejouer l'enregistrement d'un comportement réel pour lequel, les dates des actions ont été enregistrées.

Notre modèle présente également trois compositions possibles de "timer" qui ont été implémentées "Bounded" limite la portée, début s et fin e, d'un "timer". Pour ce faire, un "Bounded timer" t_b encapsule un "timer" existant

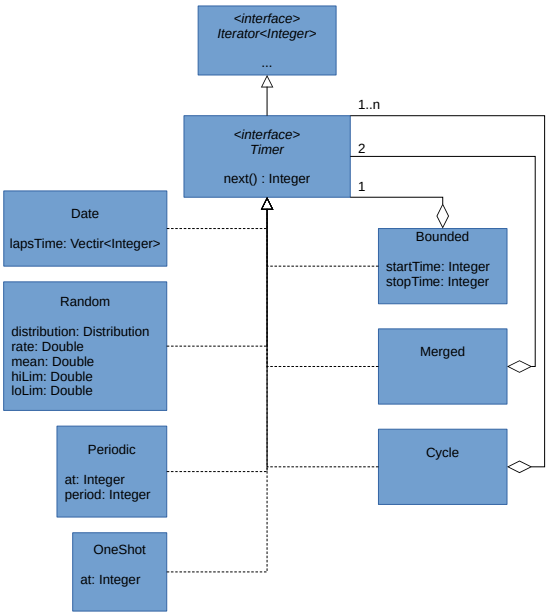

Figure 1: Diagramme UML modélisant la gestion du temps

t_w. Le premier temps t_b^0 fournit par t_b est la somme des premiers temps fournis par t_w telle que t_b^0 soit la plus petite valeur supérieure à s et inférieure à e. Il est défini par:

$$\Delta t_b^0 = \sum_{i=0}^{n_{min}} \Delta t_w^i,$$

où

$$n_{min} = argmin_n(\sum_{i=0}^{n} \Delta t_w^i | s \leq \sum_{i=0}^{n} \Delta t_w^i \leq e).$$

Le dernier temps t_b^l fourni par t_b est la somme des premiers temps fournis par t_w telle que t_b^l soit la plus grande valeur supérieure à s et inférieure à e. Il est défini par:

$$\Delta t_b^l = \sum_{i=0}^{n_{max}} \Delta t_w^i,$$

où

$$n_{max} = argmax_n(\sum_{i=0}^{n} \Delta t_w^i | s \leq \sum_{i=0}^{n} \Delta t_w^i \leq e).$$

Un "Merged time" t_m est défini à partir de deux "timers" existants, t_1 et t_2. Il fournit à chaque étape la somme des temps fournis par t_1 et t_2 pour l'étape

correspondante:

$$\Delta t_m^i = \Delta t_1^i + \Delta t_2^i.$$

Pour finir, un "Cycle timer" est défini par une liste de "timer" existants et fournit le temps de chaque "timer" les uns après les autres et recommence au début de la liste après avoir fourni le temps du dernier "timer" de la liste:

$$\Delta t_s^i = \Delta t_{i \bmod n}^k$$

où "mod" est l'opérateur de modulo, n le nombre de "timer" de la liste et k est donné par:

$$k = \left\lfloor \frac{i}{n} \right\rfloor.$$

En d'autres termes, les n premiers temps Δt_s^i sont les premiers temps Δt_i^0 des n "timer" de la liste, les n Δt_s^i suivants sont les seconds temps Δt_j^1 des "timer" de la liste, etc.

Les "timers" composés permettent de modéliser un certain nombre de comportements humains en intégrant de la variabilité. Un "Merged time" peut combiner un "timer" périodique avec un autre aléatoire afin de représenter un comportement humain périodique tout en intégrant de légères variations temporelles. Par exemple une action effectuée tous les jours à la même heure à quelques minutes près.

1.2 Action

En nous appuyant sur le modèle précédent qui permet la gestion du temps, nous pouvons modéliser les actions à réaliser en fonction du temps. Comme pour les "timer", nous nous basons sur un patron composite pour modéliser les actions, voir figure 2. L'action de base, appelée "DiscretAction", implémente l'interface "DiscretActionInterface" qui définit les caractéristiques communes des actions. C'est à dire la possibilité d'accéder à l'objet sur lequel sera effectuée l'action, la méthode à appliquer à l'objet et le laps de temps d'attente avant d'appliquer la méthode. La méthode "next()" quant à elle permet d'obtenir l'action à réaliser mise à jour. Dans le cas le plus simple, seule l'information de "timer" est mise à jour, dans les cas composés, l'objet et la méthode peuvent changer à chaque exécution.

Nous remarquerons que dans notre patron, il n'existe qu'une seule action élémentaire – "DiscretAction" – qui est définie à partir d'un seul objet et une seule méthode. Les deux compositions présentées dans la figure 2 permettent de créer des dépendances entre actions à réaliser. "DiscretActionCycle" cycle sur une liste d'action. Elle réalisera les actions les unes après les autres, puis recommencera à la première lorsqu'elle arrivera à la fin de la liste. Un exemple simple de cycle d'actions est par exemple une action d'allumage suivi d'une action d'extinction, elles se suivent toujours l'une après l'autre. "DiscretActionSeries" exécutera la première action jusqu'à ce que son "timer" se termine, c'est-à-dire qu'il ne donne plus de valeur, avant de passer à la seconde action de la série.

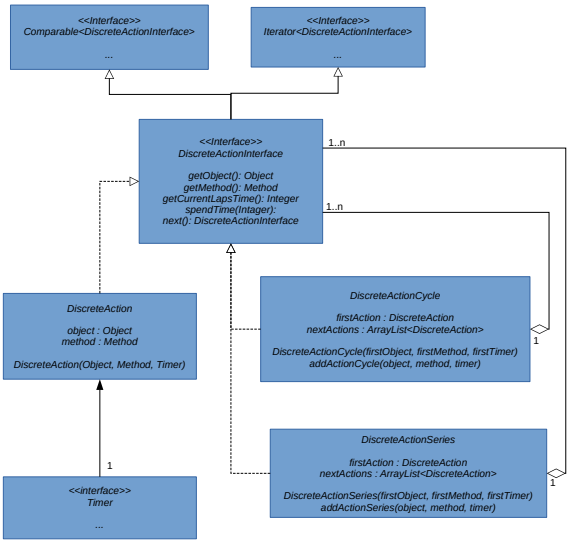

Figure 2: Diagramme UML modélisant la gestion des actions

Notons que les actions, selon la figure 2, doivent être comparables. Elles le sont par rapport à leur temps d'attente avant exécution. Ceci est indispensable au simulateur pour qu'il puisse les ordonner et les exécuter dans le bon ordre.

1.3 Simulateur

Le dernier niveau est la simulation à proprement parler. Le rôle du simulateur est donc de gérer une liste d'actions à réaliser. La figure 3 présente notre conception. Le simulateur est une entité active – un thread – qui gère à la fois une liste ordonnée d'action à réaliser et une horloge afin de simuler un temps virtuel. Continuellement, le simulateur effectue les opérations suivantes:

- extraire la première action de la liste (celle qui a le plus petit temps d'attente) en la mettant à jour
- incrémenter l'horloge avec le temps d'attente de l'action
- exécuter l'action
- réinjecter l'action à jour dans la liste

Notons que, dans cette version, nous ne prenons pas en compte le temps d'exécution de la méthode au sein de l'horloge, nous considérons qu'il est négligeable. Ceci est un point d'amélioration futur de notre simulateur.

Pour terminer cette présentation du simulateur, comme nous pouvons le remarquer au travers des différents diagrammes, ce denier n'est pas dédié aux

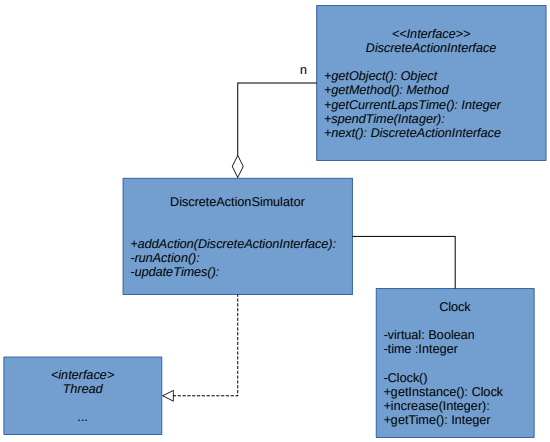

Figure 3: Diagramme UML du simulateur

WO et peut être utilisé avec n'importe quel système implémenté en Java. Outre l'utilisation que nous en faisons, l'injection d'un comportement humain afin de stimuler notre système et analyser ses réactions, le simulateur est parfaitement adapté au test logiciel. En effet, il permet de rejouer des scénarios comportementaux réels et tester un fonctionnement spécifié en phase de développement ou garantir la non-régression d'un système en phase d'évolution. Outre l'aspect simulation, l'objectif de ce SED est d'être intégré au WO afin de rejouer, pendant la phase de rêve, des séquences d'utilisations logguées et permettre au WO d'apprendre sur des variantes dans ces séquences enregistrées.