Romain Biannic (r6bianni@enib.fr)

Louis-Marie Nicolas (n6nicola@enib.fr)

Gwenaël Gicquel (g6gicque@enib.fr)

Sarah Bucquet (s6bucque@enib.fr)

Table des matières

I. Présentation

EZGantt est un programme qui vous permettra de construire vos projets de façon simple, concise et claire. Nous savons qu'un diagramme de Gantt est un outil très efficace pour gérer un plan d'actions, mais il peut être délicat à utiliser si l'on prend en compte qu'il évolue au cours du temps.

Ayant une interface ergonomique qui permet à chacun de le comprendre et de le manipuler instinctivement, EZGantt offre un large éventail d'options. De plus il s'agit d'une interface Web ainsi que d'un programme gratuit le rend accessible à tous.

Parce qu'il est Open Source, il ne fait aucun doute pour nous que notre travail sera repris pour être amélioré ou modifié afin de répondre à des besoins spécifiques. Ainsi nous accueillons la possibilité de l'évolution d' EZGantt avec anticipation sachant que cela permettra au programme de perdurer.

Mots-clefs : Gantt, Open Source, ODD, Application Web, Projet, Gestion, Organiser, UML, Javascript, Typescript

I. Presentation

EZGantt is a program that will allow you to build your projects fluently, easily and very clearly. We know that a Gantt diagram is a very effective tool to manage a plan, but it can be tricky to use as it varies in time with the elaboration of the project.

By having an ergonomic interface which allows every user to understand and manipulate it instinctively, EZGantt offers a large panel of options. The fact that it is a web interface and free makes it accessible to everyone.

Because it is opensource, it leaves no doubt in our minds that our work will be taken to be improved again or modified to meet more specific needs. Hence, we welcome with anticipation the possibility for EZGantt to evolve, knowing it will enable the program to live on.

Keywords : Gantt, Open Source, SDG, Web Application, Project, Management, Organize, UML, Javascript, Typescript

II. Préambule

Le but de notre projet d'informatique est de résoudre un problème soulevé régulièrement au sein de l'ENIB. Tous les ans, les professeurs, de mécanique notamment, sont insatisfaits des digrammes de Gantt présentés par leurs élèves. L'idée de créer un logiciel ergonomique, permettant aux étudiants de faire des diagrammes de Gantt facilement est alors née.

Mais qu'est-ce qu'un diagramme de Gantt ? Il s'agit d'un outil de gestion de projet qui permet d'ordonner et visualiser dans le temps les tâches qui composent ce projet. Ce diagramme a 3 grands avantages : déterminer les dates de réalisation d'un projet, identifier les marges existantes, et visualiser facilement l'avancement du projet.

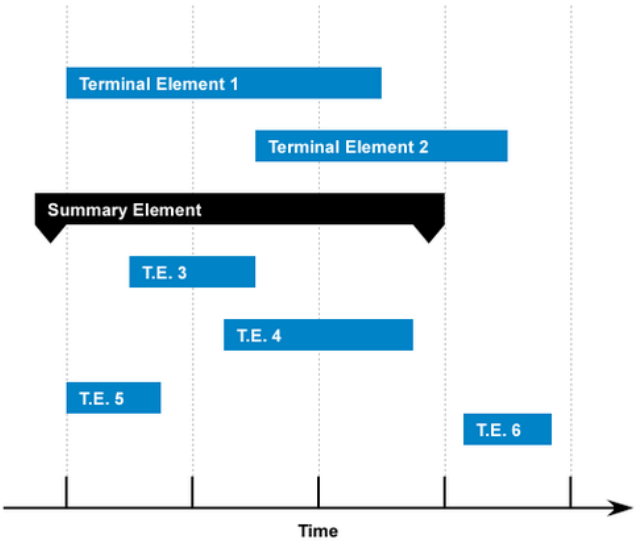


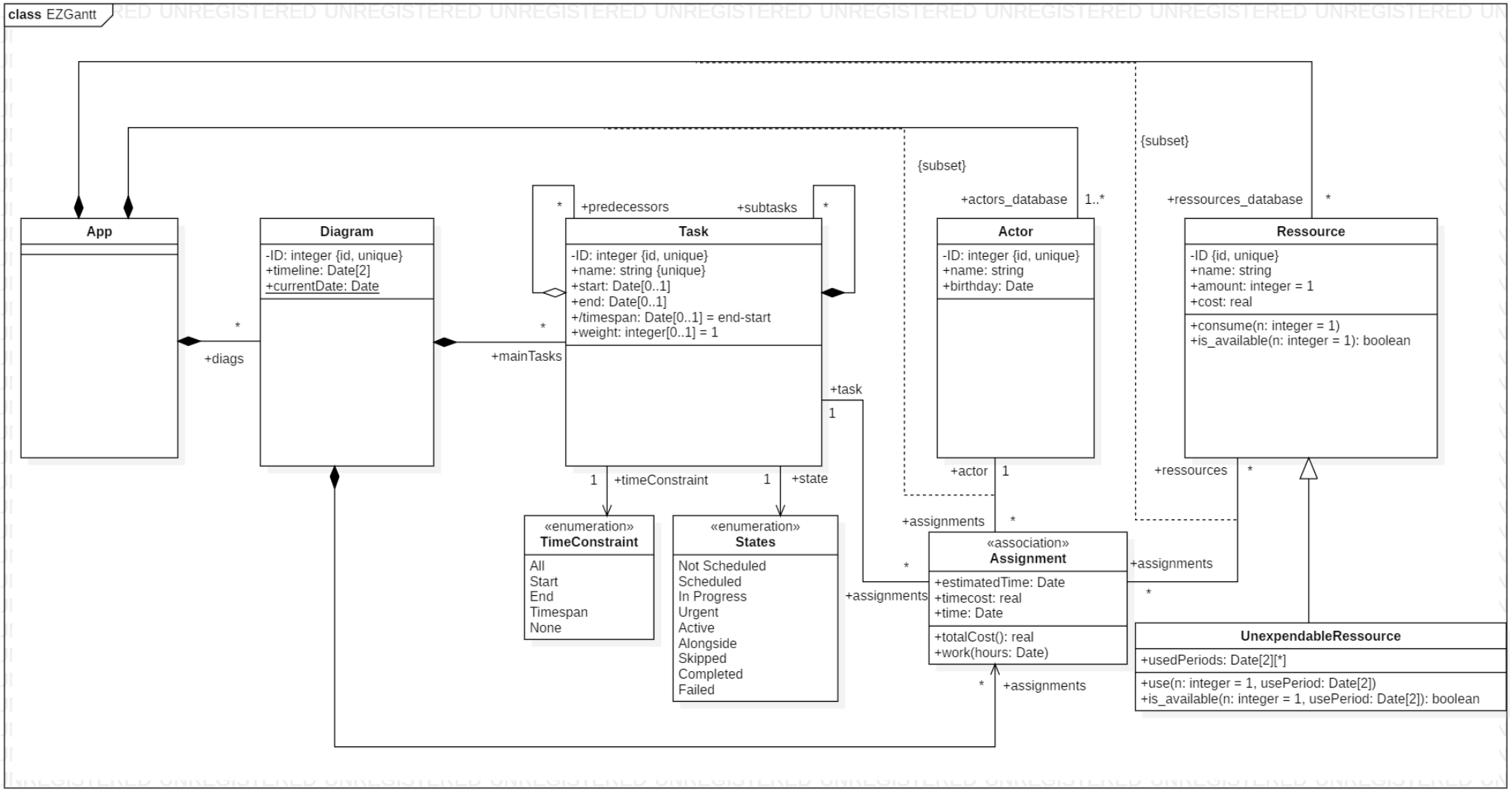
Figure 1 : Exemple de diagramme de Gantt

Étant donné l'objectif de notre logiciel, celui-ci permettra de faciliter la mise en place de projets (industriels par exemple) en aidant à l'organisation. Comme énoncé précédemment, ce logiciel sera en Open Source, il sera donc accessible de par le monde. Nous avons pris cette décision afin d'aider à « perfectionner les capacités technologiques des secteurs industriels de tous les pays » ainsi que de « soutenir la recherche-développement et l'innovation technologiques nationales dans les pays en développement, notamment en instaurant des conditions propices ». Tel que mentionné dans les paragraphes 5 et b des cibles de l'objectif 9 du développement durable intitulé : "industrie, innovation et infrastructures".

« Parvenir à un niveau élevé de productivité économique par la diversification, la modernisation technologique et l'innovation » est également une cible du huitième objectif du développement durable qui cherche à assurer un travail décent et une croissance économique à tous.

III. Diagramme de classe

1. Schéma



2. Description

a) App

Dans un premier temps, concentrons sur la classe *App*. Elle est la classe principale et représente le logiciel en lui-même. Elle est composée des bases de données – des acteurs (*actors_database*) et des ressources (*ressources_database*) - et bien sûr de l'ensemble des diagrammes de Gantt (*diags*).

b) Diagram

La classe *Diagram* est celle qui symbolise un diagramme de Gantt. Elle est, par conséquent, composée des tâches principales (*mainTasks*), c'est-à-dire celles au niveau 0 du diagramme. Elle est aussi composée par l'ensemble des affectations (*assignments*). Elle possède un attribut (*timeline*) qui représente la barre des temps qui permet de situer les tâches et un deuxième attribut (*currentDate*), statique car identique pour toutes les instances de Diagram, qui détermine la date et en temps réel.

NB : Le type Date est utilisé à beaucoup d'endroit dans le diagramme et sera implémenté par une librairie externe : nous l'avons donc sous-entendu comme type primitif pour faciliter la lecture du diagramme.

c) Task

La classe *Task* est l'une des classes les plus importantes de la modélisation. Elle désigne, naturellement, une tâche. Elle possède bon nombre d'attributs, un nom (*name*), un début (*start*), une fin (*end*), une durée (*timespan*) et un poids (*weight*). Le fait de caractériser la tâche dans le temps par 3 attributs (*start*, *end*, et *timespan*) permettra de faciliter l'implémentation de la classe. En effet, si on connaît deux de ces attributs, alors le troisième peut être déterminé par calcul ; le programme s'adapte donc tout autant à l'utilisateur qui connaît les dates précises de début et de fin, qu'à l'utilisateur qui connaît plutôt la durée approximative d'une tâche. Le poids définit simplement une « importance » à la tâche.

Une tâche dans un diagramme de Gantt possède certaines contraintes. Par exemple, elle peut nécessiter la fin d'une ou plusieurs tâches antérieures pour débuter. Nous retrouvons cette contrainte avec *predecessor*. Il faudra noter l'usage d'une agrégation pour éviter le problème simple suivant : deux tâches, contraintes mutuellement, qui doivent attendre la fin de l'autre pour débuter.

De plus, une tâche peut aussi être décomposée en plusieurs sous-tâches elles-mêmes décomposables. Nous pouvons donc identifier la composition (*subTasks*) qui permet précisément de symboliser cette fonctionnalité.

Afin de pouvoir décrire d'une meilleure façon les tâches, nous avons rajouté deux énumérations. La première (*state*) donne à la tâche un état dont le comportement est décrit plus bas. La deuxième (*timeConstraint*), permet à l'utilisateur de déterminer les contraintes temporelles qui lui sont associées.

Par exemple, si on veut raccourcir la deadline, le diagramme sera recalculé en tenant compte des contraintes.

d) Actor

La classe *Actor* permet de représenter un acteur du projet. L'ensemble des acteurs forme une base de données (que l'on retrouve dans *actors_database*). Ils ne sont modélisés ici que par un nom (*name*) et une date de naissance (*birthday*) mais pourront potentiellement posséder plus d'attributs lors de l'implémentation (comme un système d'heures supplémentaires de travail voire même d'emploi du temps).

e) Ressource & UnexpendableRessource

La classe *Ressource* permet de modéliser les ressources du projet. Tout comme *Actor* , l'ensemble des ressources forme une base de données (*ressources_database*). *Ressource* possède un nom (*name*), un prix unitaire (*cost*), et la quantité de la ressource (*amount*). L'opération *consume* permet de consommer *n* unités et *is_available* vérifie si le stock est disponible.

Elle se dérive aussi en *UnexpendableRessource* qui représente les ressources non-consommables. Elle possède alors un attribut supplémentaire : *usePeriods*, la liste des périodes d'utilisation de la ressource. L'opération *use* permet alors d'ajouter une période d'utilisation valide. La surcharge de *is_available*, avec une période en paramètre, vérifie le stock pour la période donnée. La permière fonction est toujours fonctionnelle, sans période de temps.

NB : l'opération consume peut toujours être utilisé et signifie plutôt la perte ou la casse de la ressource non-consommable.

f) Assignement

La classe-association *Assignment* fait l'association entre 3 entités (*Task*, *Actor*, *Ressource*). Elle symbolise l'affectation d'un acteur à une tâche avec une certaine liste de ressource. Cette affectation est définie par un coût horaire (*timeCost*), une estimation du temps de travail (*estimatedTime*) afin de calculer le coût total approximatif (donné par *totalCost*) et le temps de travail courant (*time*) qui augmente avec l'utilisation de la fonction *work* qui comptabilise le temps de travail d'un acteur.

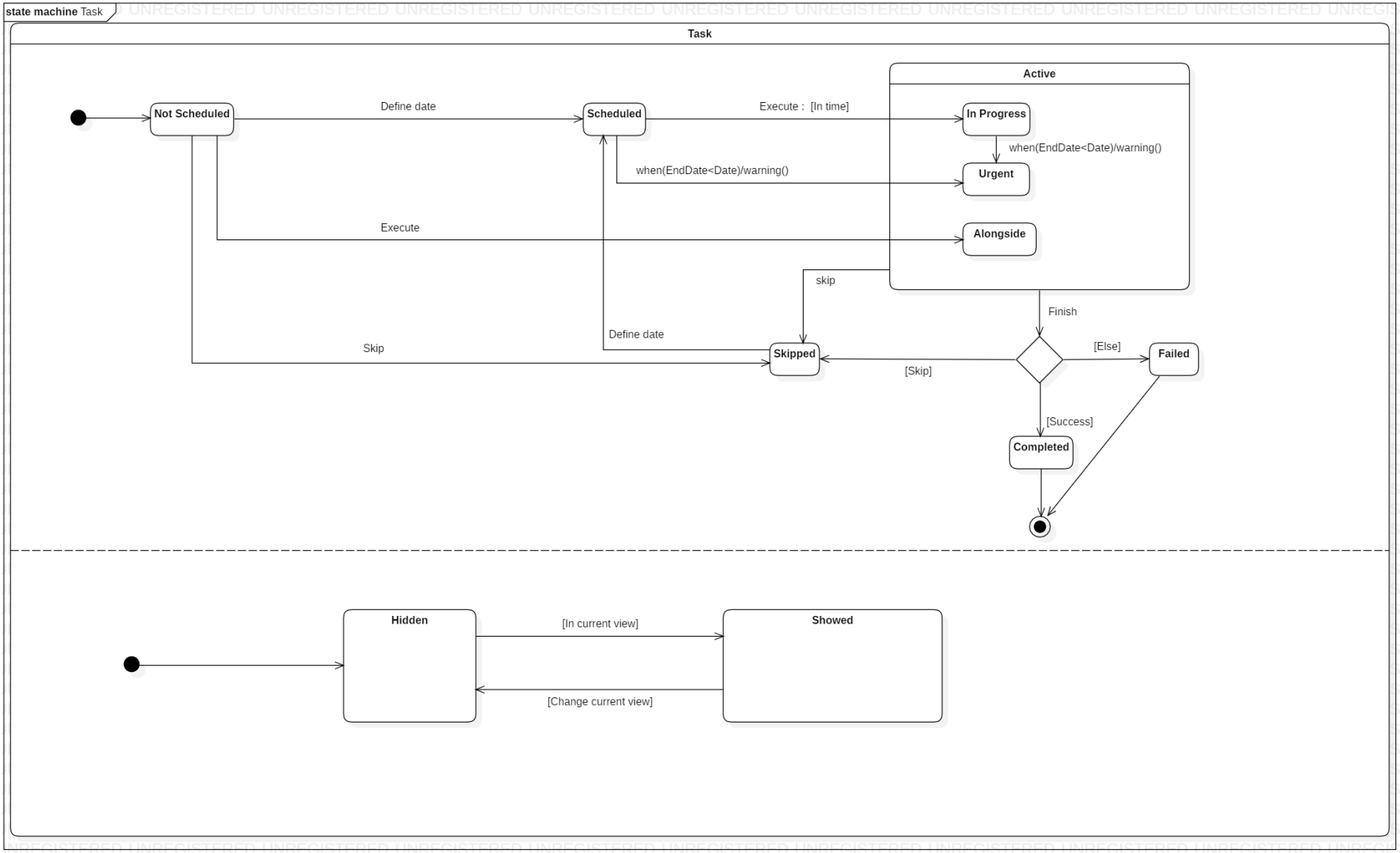
NB : L'acteur et les ressources associés font respectivement partie des bases de données des acteurs (subset de *actors_database*) et des ressources (subset de *ressources_database*).

Si plusieurs acteurs travaillent sur une même tâche, il faudra juste instancier plusieurs fois *Assignement* avec la même tâche. Nous avons choisi cette modélisation pour permettre de différencier les coûts et les ressources par acteur.

Les 3 entités ont aussi une référence vers *Assignment* pour faciliter le travail de recherche aux requêtes du type « quelles sont les ressources/acteurs de cette tâche ? » et trouver directement toutes les affectations qui satisfassent la requête.

IV. Diagramme d'état d'une tâche

1. Schéma



2. Description

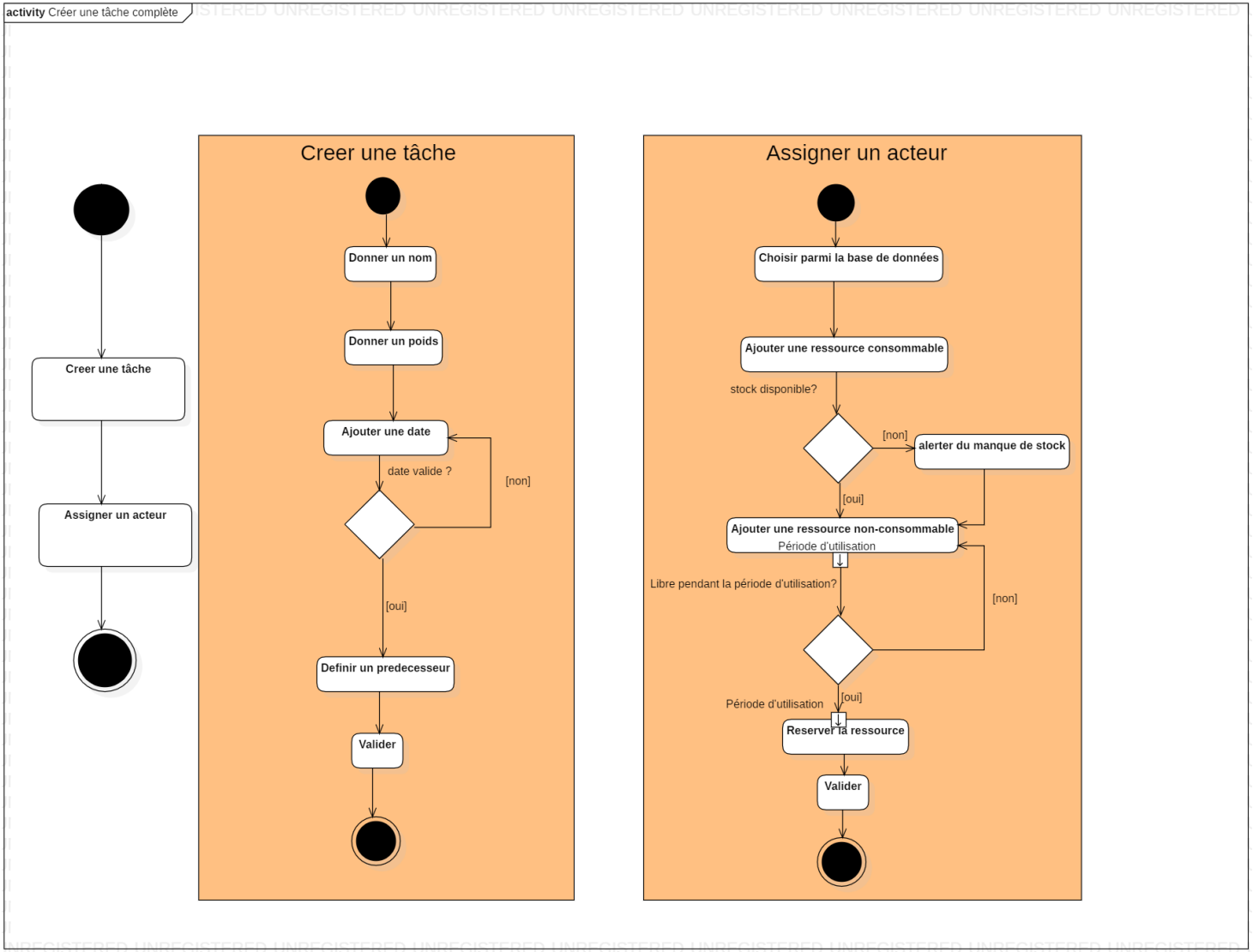
Ce diagramme de machine à état représente les différents états dans lequel peut être une tâche.

La région supérieure gère l'état *state* des tâches dans le temps. A son instanciation, une tâche est d'abord non-programée (*Non Scheduled*). Elle peut ensuite se voir affecter une date et passe alors en état *Scheduled*. Sinon, elle peut être passée (*Skipped*) ou exécutée en hors-piste (*Alongside*). Si elle est executée dans sa période, elle rentre dans l'état *In Progress.* Mais si elle n'est pas terminée avant la fin de sa période, elle est alors dans l'état *Urgent*. Les 3 états d'exécution (*Alongside, In Progress,* et *Urgent*) sont l'état actif (*Active*) de la tâche. A tout moment une tâche active peut passer à l'état *Finished* dans le cas où elle est terminée et dans l'état *Skipped* si un changement d'objectif nécessite de passer cette tâche. Cette conception se justifie par le besoin de souplesse des entreprises qui doivent s'adapter à des deadlines fixes. Lorsqu'une tâche est finie, elle est soit réussie (*Completed*) ou ratée (*Failed*)

La région inférieure contient les états *Hidden* et *Showed*. En effet, du fait de la taille finie des écrans et du système de tâche/sous-tâches, il n'est pas souhaitable d'afficher en même temps toutes les tâches. Ces deux états sont donc liés au rendu des tâches. Cette modélisation ne concerne que l'implémentation de l'interface du logiciel et est donc optionnelle dans cette représentation.

V. Diagramme d'activité

1. Schéma



2. Description

Ce diagramme d'activité représente un cas d'utilisation type de notre logiciel. Dans ce cas d'utilisation, un utilisateur définit une tâche dans un cadre temporel fixe, la met en relation avec d'autres tâches puis lui affecte un acteur ayant accès à un stock de ressources, consommables et non-consommables.

Ainsi, la première activité représentée dans le graphe est 'Créer une tâche'. Dans le cas d'utilisation étudié cette activité comprend le nommage de la tâche, la détermination de son poids c'est à dire de son degré d'importance dans le projet, puis enfin sa deadline et sa relation avec les tâches existantes.

La deuxième activité représentée est 'Assigner un acteur'. Celle-ci comprend la sélection d'un acteur dans une liste du personnel, puis l'allocation à cet acteur d'une ressource consommables et d'une ressource non-consommables. Ces ressources sont ensuite explicitement réservées.