Université de Rouen Normandie

GIL
UFR Sciences et Techniques

Projet Architecture logiciel

Étudiants :
Venance DJABIA
Michel NASSALANG

Enseignants :
Florent NICART

Mises à jour du document

Version	Date	Modification réalisée
1	27 Mars 2024	Création du document
2	29 Mars 2024	Conception
3	03 Avril 2024	Codage et modifications apportées
4	05 Avril 2024	Refonte de la conception et remise en cause
5	08 Avril 2024	Finalisation du documentet d'un analyseur.

Table des matières

1 Introduction

A la fin de cette UE dediée à l'architecture logicielle, nous avons été invités à réaliser un projet visant à développer une application de gestion de tâches (todo lists) permettant de suivre efficacement les tâches à réaliser. L'application comprend deux programmes principaux :

tsk-edit : Un éditeur de listes de tâches avec une interface graphique pour créer, ajouter, supprimer, modifier et sauvegarder/charger des listes de tâches dans un format XML personnalisé.

tsk-top : Un programme en ligne de commande qui lit une liste de tâches à partir d'un fichier et affiche les 5 tâches non terminées avec les dates d'échéance les plus proches, triées par ordre croissant.

Les tâches peuvent être simples (complétées ou non) ou complexes (composées de sous-tâches). Elles sont caractérisées par un descriptif, une date d'échéance, une priorité, une durée estimée et un état de progression. L'objectif principal est de concevoir une architecture logicielle modulaire et extensible, en appliquant les principes et patrons de conception vus en cours, tels que les patrons de création, structurels et comportementaux.

2 Environnemment et déroulement du projet

2.1 Environnement

Pour le déloppement de ce projet, nous avons opter pour un environnement java (JavaFX). A cet effet, nous avons devoir configurer un environnement propice pour cela.

Nous allons suivre les étapes suivantes pour la configuration

- Installer Java Development Kit (JDK 17 minimum)
- Installer un environnement de développement intégré (IDE) (IntelliJ IDEA version Community gratuite)
- Configurer un système de gestion de dépendances (optionnel) (Maven)
- Configurer un système de contrôle de version (recommandé) (Git)

2.2 Déroulement du projet

Nous avons commencé le projet par l'analyse et la modélisation. On est ammené à déterminer les patrons de conception à mettre en place. Et cela au départ nous a poussé à faire fausse route. Au départ on ciblait un patron de conception et essayer de le mettre en place dans le système. A la fin pour ne pas trop complexifier le système, on a pris départ sur les besoins réels du projet et on a essayé de respecter le maximum de principe en fonction de la complexité du projet. Au début nous avions eu des points de vue d'une divergence trop grande pour pouvoir travailler ensemble. Cela nous a bloqué pendant un long moment. En un moment, on a même envisagé de travailler chacun de son bord vu qu'on ne trouvait pas de point d'entente. A la fin, chacun a fait sa conception de solution avec une implémentation qui concrétise réellement l'éfficacité de nos idées. A partir de ses implémentations, on a parvenu à connaitre lequel des solutions serait la base la plus convenable pour notre solution. Et c'est en ce moment qu'on a réellement su travailler ensemble pour la conception et la mise en oeuvre de la solution définie dans ce document. Et cela nous a causé un énorme retard pour l'aboutissement de la solution.

3 Raisonnement des choix des Patrons de conception

Pour le choix des différents patrons de conception, nous avons été confronté à un défi majeur : au cours de nos discussions, nous avons eu des divergences dans le choix des patrons à utiliser. Mais finalement pour les contraintes de temps, nous avons décidé d'implementer les différents patrons suivants :

3.1 Composite

3.1.1 Description

Le patron composite uniformise le traitement d'objets et de lots d'objets. Le pattern Composite est idéal pour représenter la structure hiérarchique des tâches, où une tâche complexe peut contenir d'autres sous-tâches. Cela permettrait de traiter uniformément les tâches simples et complexes

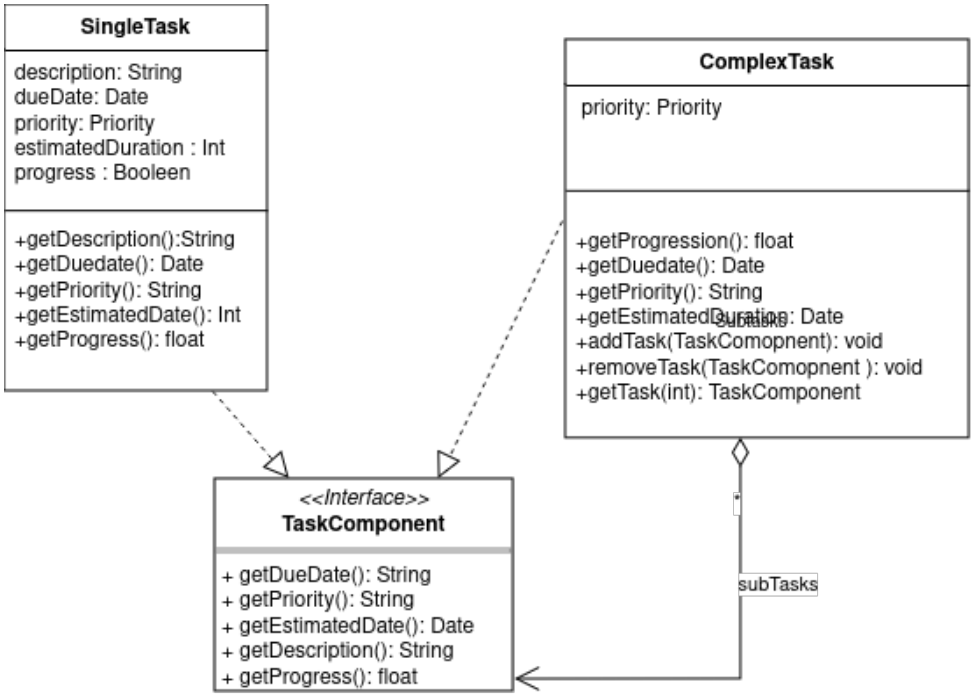
3.1.2 Motivation

Ce patron a été choisi dans notre projet, car nous avons différents types de taĉhes à gérer (tâche simple (SimpleTask) ne contenant aucune sous-tâche et les tâches complexes peuvent contenir d'autres sous-tâches) ComplexTask. Et ces différentes tâches, nous devons les traiter de manière uninforme.

3.1.3 Les entités du patron

Component : TaskComponent (*interface*)
leaf : SimpleTask
composite : ComplexTask

3.1.4 Diagramme



3.2 Singleton

3.2.1 Description

Fournir une (hiérarchie de) classe(s) admettant au plus une instance. Le pattern Singleton garantit qu'il n'existe qu'une seule instance de la liste de tâches en cours d'édition, accessible globalement dans l'application.

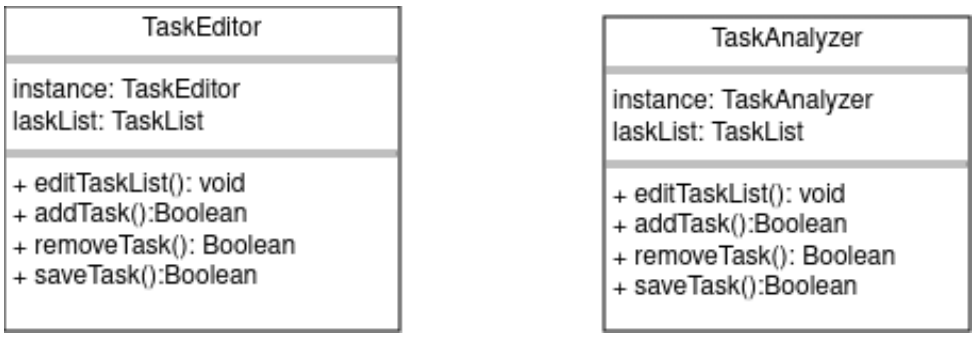
3.2.2 Motivation

Dans notre projet, l'un des spécification nous demandait de s'assurer qu'une seule instance des liste de tâche ne soit encours de traitement à un instant T. Et le patron qui nous permet de s'assurer de l'unicité d'une instance est biensûr le patron Singleton. Ainsi ce patron a été appliqué pour la réalisation de l'éditeur et l'analyseur.

3.2.3 Les entités du patron : 2 patrons singleton

Singleton : TaskEditor
Singleton : TaskAnalyzer

3.2.4 Diagramme



3.3 Observateur

3.3.1 Description

Permet de synchroniser l'état de plusieurs objets dans une relation un-à-plusieurs.

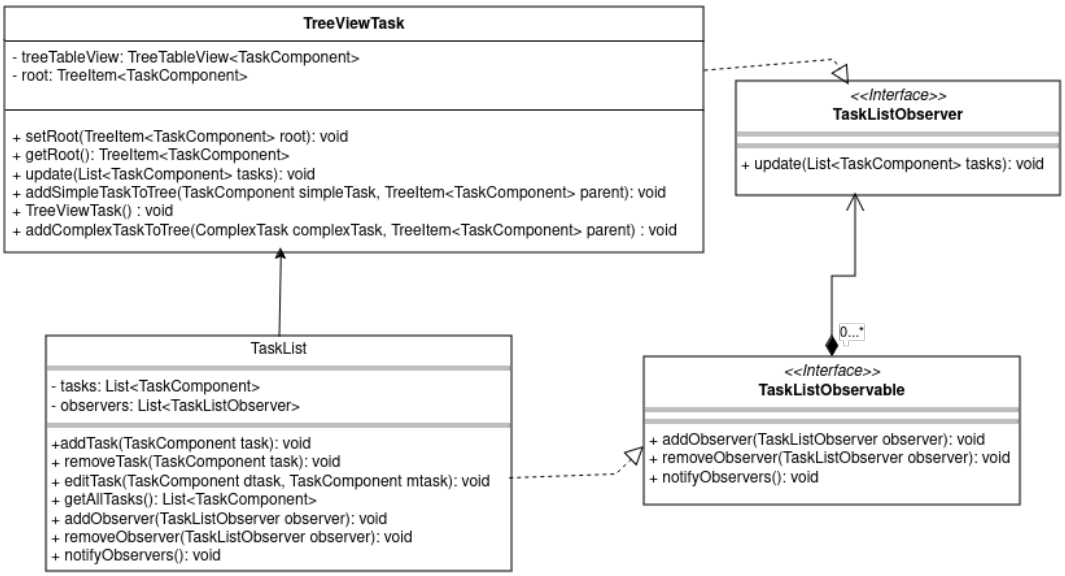
3.3.2 Motivation

Nous avons une spécification dans notre projet, nous devons faire des affichages dans une interface graphique. Afin de gérer cet aspect et de notifier les différents acteurs qui interviennent dans l'affichage des données sur l'interface graphique, nous avons besoin d'un patron qui joue ce rôle de surveillance et de notification afin de faire des mises à jour dans l'affichage. C'est dans ce sens que nous avons choisir le patron observateur qui cadre avec les objectifs que nous voulons atteindre.

3.3.3 Les entités du patron

Observer : TaskListObserver
ConcreteObserver : TreeViewTask
Subject : TaskListObservable
ConcreteSubject :TaskList

3.3.4 Diagramme



3.4 Factory Method

3.4.1 Description

Définir une interface pour créer un objet et laisser l'implémenteur en choisir le type.

3.4.2 Motivation

Pour gérer les listes de tâches, nous utilisons le pattern Factory Method. Une interface ListTaskFactory définit la méthode de création de liste de tâches, et chaque type de liste de tâche a sa propre implémentation concrète de cette interface. Ici nous avons deux type de liste de tâches : celui obtenu lorsqu'on charge un fichier et celui obtenu lorsque nous utlisons l'interface graphique pour la création.

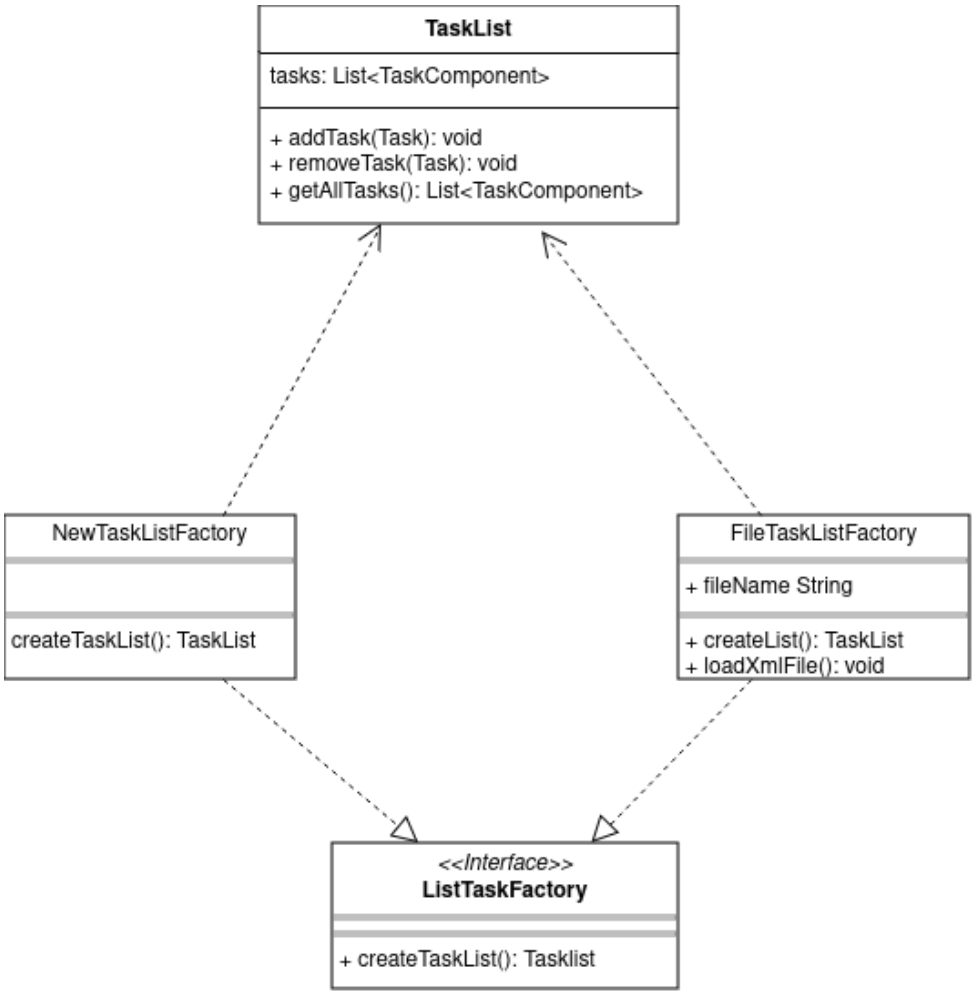
Le pattern Factory Method est utilisé pour la gestion des différents types de liste de tâches dans ce projet pour plusieurs raisons :
Encapsulation de la logique de création Le pattern Factory Method encapsule la logique de création des objets dans des classes dédiées (les factories). Cela permet de séparer la responsabilité de création des objets du reste du code de l'application, favorisant ainsi le principe de responsabilité unique.
Extensibilité En utilisant le pattern Factory Method, il est facile d'ajouter de nouveaux types de liste de tâches dans le futur. Il suffit de créer une nouvelle classe concrète implémentant l'interface ListTaskFactory pour le nouveau type de liste de tâche, sans avoir à modifier le code existant (OCP).
Découplage : Le patron Factory Method découple le code client du code de création des objets. Le client n'a pas besoin de connaître les détails d'implémentation des différents

types de liste de tâches, il interagit uniquement avec l'interface ListTaskFactory. Cela favorise un couplage faible entre les composants du système.
Cohérence : Le patron Factory Method garantit que les objets créés respectent les contraintes et les règles de création spécifiques à chaque type de liste de tâche. Cela aide à maintenir la cohérence et l'intégrité des données dans l'application.

3.4.3 Les entités du patron

Product :TaskList
Creator : ListTaskFactory
ConcreteCreator : NewListTaskFactory, FileTaskListFactory

3.4.4 Diagramme



3.5 Visiteur

3.5.1 Description

Permettre l'ajout d'opérations homogènes aux élément d'un modèle sans couplage

3.5.2 Motivation

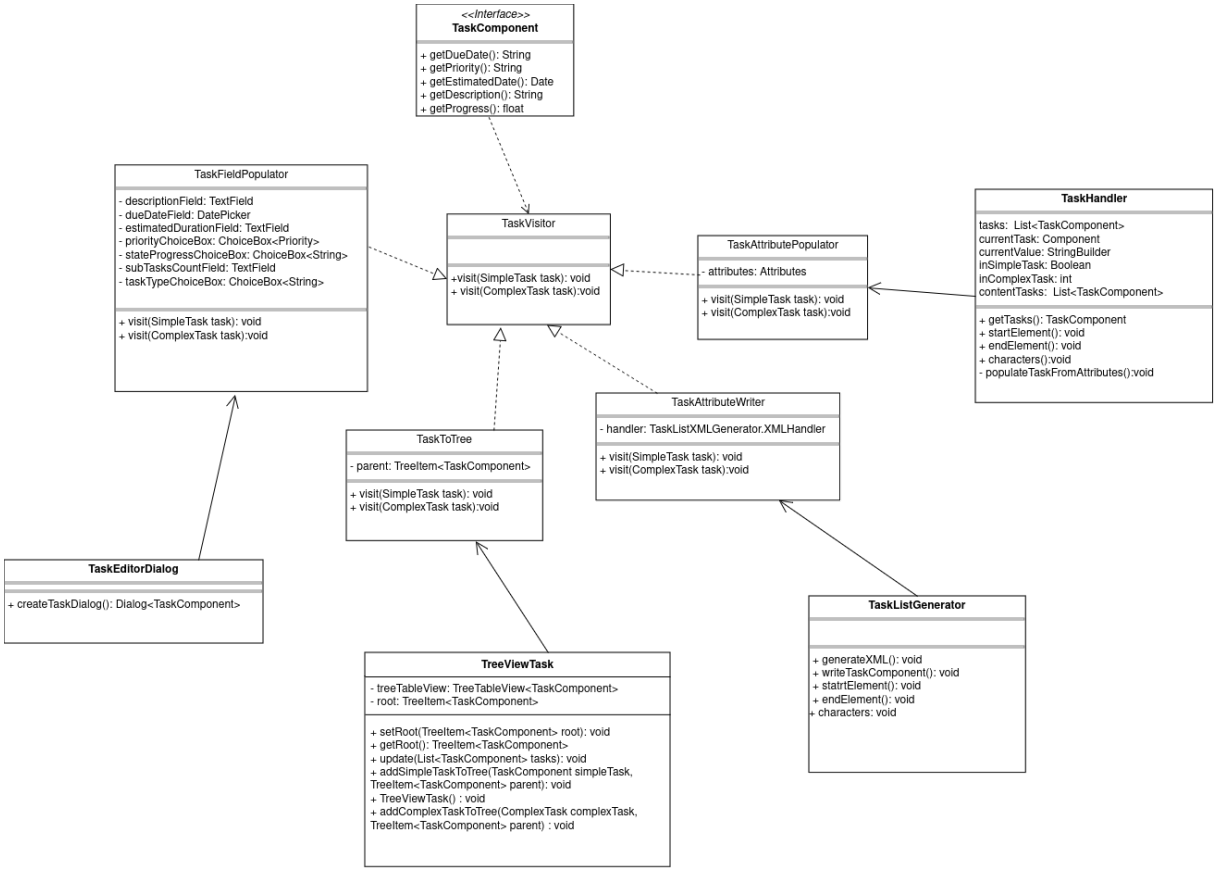
Vu les différents types de tâches à gérer dans les systèmes comme la sérialisation en XML, la mise en forme des tâches au niveau interface, nous avions dû enfreindre à plusieurs reprises le principe OCP pour implémenter ces fonctionnalités. Et cela nous a poussé à vouloir rendre le système beaucoup plus modulaire et évoluable. Pour éviter cela, nous avons jugé nécessaire d'utiliser le patron visitor

3.5.3 Les entités du patron

Client :
Visitor : TaskVisitor
ConcreteVisitor : TaskFieldPopulator, TaskToTree, TaskAttributeWriter, TaskAttributePopulator
Element : TaskComponent
ConcreteElement : SimpleTask, ComplexTask
ObjectStructure : TaskHandler, TreeViewTask, TaskEditorDialog, TaskListXMLGenerator

3.5.4 Patron envisagé mais écarté

3.5.5 Diagramme :



4 Patrons écartés et patrons éventuels

4.1 Patrons écartés

4.1.1 Builder

Le patron Builder pouvait être utilisé pour construire des objets tâches. Il permet de créer des tâches avec différents ensembles d'attributs sans avoir à utiliser de nombreux constructeurs différents. La classe TaskBuilder fournirait des méthodes pour configurer les attributs d'une tâche avant de la construire. Cependant nous avons vu que ce patron allait beaucoup plus complexifier notre système.

4.1.2 Factory Method

Le patron Factory Method a été aussi envisagé pour la création de taches(ComplexTask, SimpleTask). On a envisagé ce patron pour permettre la création de tache vu la complexité des tâches et la différence entre les tâches. On n'a écarté ce patron parceque on avait que réellement ça n'apportait rien de plus à notre architecture.

4.2 Patrons éventuels à adopter

4.2.1 Adapter

Le patron Adapter pouvait être utilisé pour convertir les données des tâches complexes en un format compatible avec l'analyseur de tâches. Cela permet à l'analyseur d'accepter des données dans un format différent de celui attendu. La classe TaskAdapter fournirait une méthode pour convertir une liste de tâches complexes en une liste de tâches simples.

Pour la persistance des tâches, nous pourrions utiliser le patron Adaptateur. Une interface TaskPersistenceAdapter définira les méthodes pour sauvegarder et charger les tâches. Différentes implémentations concrètes de cette interface pourront être fournies pour différents les formats de fichiers (XML, JSON, etc.).

4.2.2 Façade

Nous pourrions utiliser le patron Façade pour fournir une interface simplifiée pour les opérations courantes de l'éditeur (créer, modifier, supprimer des tâches, charger/sauvegarder la liste de tâches). L'interface graphique de l'éditeur utiliserait cette façade pour interagir avec la logique métier.

5 Répartition des paquetages et responsabilités

```
src.main.java.fr.univrouen
   |
   +--- todolist
   |       |-- TodolistApp.java (Point d'entrée général de l'application)
   |
   +--- analyzer
   |       |-- TaskAnalyzer.java (Implémentation de TaskAnalyzer fournissant des fonctionnalités d'analyse pour les listes de tâches)
   |       +-- TaskAnalyzerApp.java (Point d'entrée de l'analyseur de tâches)
   |
   +--- editor
   |       |-- TaskEditor.java (Imlplémentation de TaskEditor responsable de la gestion des opérations d'édition de tâches)
   |       +-- TaskTopApp.java (Point d'entrée de l'éditeur de tâches)
   |
   +--- parser
   |       |-- TaskHandler.java (Implémentation du gestionnaire d'événements SAX qui analyse un fichier XML de tâches)
   |       +-- TaskListXMLGenerator.java (Implémentation du générateur de fichier XML à partir d'une liste de tâches)
   |
   +--- task
   |       |-- TaskCompent.java (Interface représentant une tâche)
   |       |-- SimpleTask.java (Implémentation d'une tâche simple)
   |       |-- ComplexTask.java (Implémentation d'une tâche complexe)
   |       +-- Priority.java (Énumération représentant les priorités d'une tâche)
   |
   +--- taskListFactory
   |       |-- TaskListFactory.java (Interface pour la création d'une liste de tâche)
   |       |-- NewTaskListFactory.java (Implémentation de la création d'un TaskList via l'éditeur)
   |       +-- FileTaskListFactory.java (Implementation de la création d'une instance de TaskList à partir d'un fichier XML)
   |
   +--- taskListObserver
   |       |-- TaskList.java (Implémentation de TaskList représentant une liste de tâches observables.)
   |       |-- TaskListObservable.java (Interface définissant les méthodes et notifier les observateurs d'une liste de tâches.)
   |       |-- TaskListObserver.java (Interface répresentant l'observer)
   |       +-- TreeViewTask.java (Implémentation d'affiche des tâches sous forme d'arborescence dans une interface graphique)
   |
   +--- taskVisitor
   |       |-- TaskAttributePopulator.java (Implémentation de TaskAttributePopulator, visiteur qui peuple les attributs d'une tâche)
   |       |-- TaskAttributeWriter.java (Implémentation de TaskAttributeWriter, visiteur chargé d'écrire les attributs d'une tâche)
   |       |-- TaskFieldPopulator.java (Implémentation de TaskFieldPopulator, visiteur chargé de remplir les champs d'une tâche)
   |       |-- TaskVisitor.java (Interface du visiteur)
   |       +-- TaskToTree.java (Implémentation de TaskToTree, visiteur qui convertit une hiérarchie de tâches en une structure)
   |
   +--- ui
           |-- AlertUI.java (Implémentation de AlertUI fournissant une statique pour afficher des alertes dans l'interface utilisateur)
           |-- AnalyzeBox.java (Vue de l'analyser de tâches)
           |-- EditBox.java (Vue de l'éditeur de tâches)
           |-- TaskEditorDialog.java (Implémentation de TaskEditorDialog une boîte de dialogue pour éditer les détails d'une tâche.)
           +-- TaskAnalyzerDialog.java (Implémentation de TaskAnalyzerDialog une boîte de dialogue pour afficher les tâches incomplètes.)
```

5.1 analyzer

Il contient le point d'entrée de l'application Analyzer(TaskAnalyzerApp) et la classe (TaskAnalyzer) permettant d'implémenter l'analyseur de tâches.

5.2 editor

Il contient le point d'entrée de l'éditeur (TaskEditorApp) et la classe (TaskEditor) permettant d'implémenter l'éditeur des tâches.

5.3 task

Il contient les classes représentant les différents types de tâches (TaskComponent, SimpleTask, ComplexTask) et l'enumération Priority.

5.4 taskListFactory

Il contient les classes liées à la gestion des listes de tâches, notamment l'interface (TaskListFactory) et les classes NewTaskListFactory et FileTaskListFactory.

5.5 taskListObserver

Il contient les classes liées à la gestion des notifications et la mise à jours des données affichées à l'interface graphique. Notamment les deux interfaces (TaskListObservable et TaskListObserver) et les classe (TaskList, TreeViewTask).

5.6 ui

Il contient les classes liées à l'interface utilisateur de l'éditeur et de l'analyseur notamment les classes (AnalyzeBox,EditBox, TaskEditorDialog, AlertUI, TaskEditorDialog)

5.7 todolist

Il contient l'application principale du système qui unit l'interface de l'éditeur et l'interface de l'analyseur en un même (TodoListApp).

5.8 parser

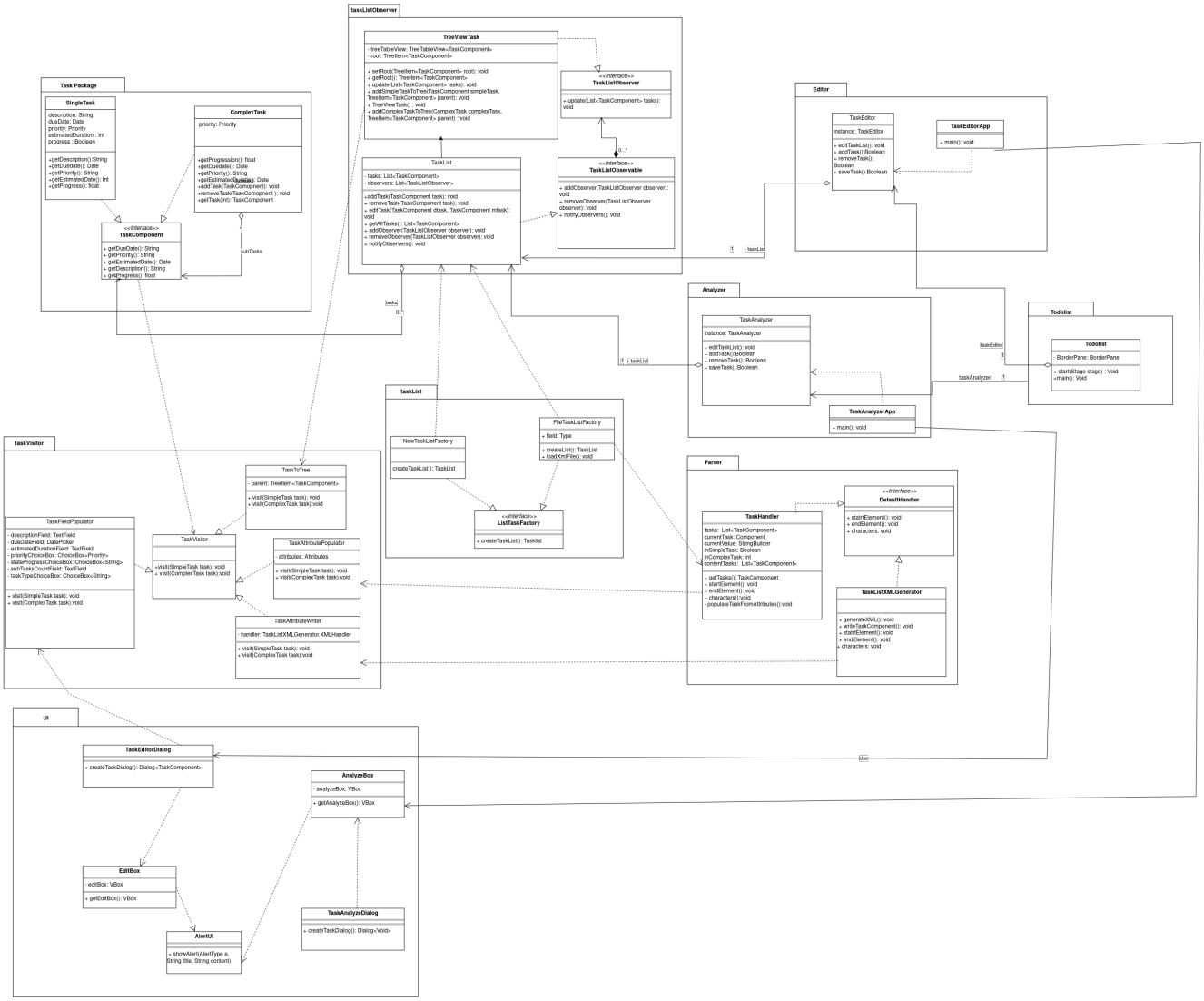
Il contient l'ensemble des les classes permettant la sérialisation en XML et la désérialisation (TaskHandler et TaskListXMLGenerator).

5.9 taskVisitor

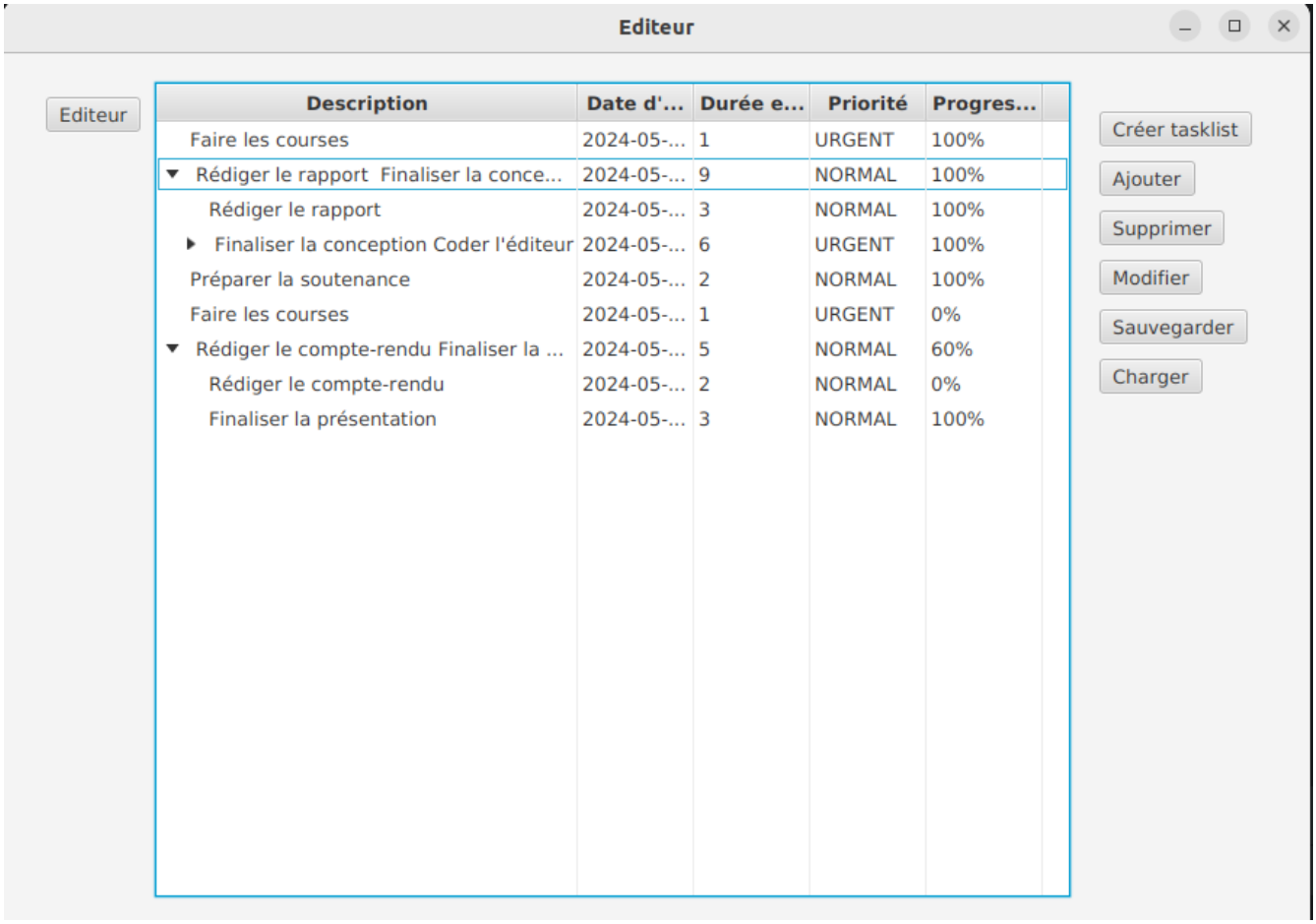
Il contient l'ensemble des acteurs du patron visiteur intervenant dans le système. Le visitor nous a réellement permis d'avoir une solution évoluable car toutes les infractions d'OCP qu'on avait remarqué, on était éliminé par le patron qu'on juge très important pour notre système. Il est composé de l'interface TaskVisitor et des classes : TaskToTree, TaskFieldPopulator, TaskAttributeWriter, TaskAttributePopulator.

6 Architecture Globale de l'Application

Nous avons la modélisation globale de notre solution ci dessous.



Nous avons les programmes d'édition et d'analyseur illustrés par ces images suivantes.



7 Système de sérialisation et désérialisation XML

tasklist : peut contenir une séquence d'éléments simpleTask et/ou complexTask. L'astérisque (*) indique que cette séquence peut se répéter zéro ou plusieurs fois.

simpleTask : c'est un élément vide, c'est-à-dire qu'il n'a pas de contenu texte ou d'autres éléments enfants.

Attributs de SimpleTask : Chaque attribut est défini avec un type de données (dans ce cas, CDATA indique que le contenu de l'attribut est du texte) et une déclaration d'obligation .

ComplexTask : peut contenir une séquence d'éléments simpleTask et/ou complexTask, similaire à la définition de tasklist. On a seulement l'attribut priority. priority est l'unique attribut avec la même déclaration d'obligation que simpleTask

priority : contient la priorité de la tâche, qui doit être l'une des valeurs définies dans l'entité (urgent, normal ou secondaire).

Ce DTD définit une structure pour un document XML contenant des tâches, où une tâche peut être soit simple (avec une description, une date d'échéance, une priorité, une estimation de durée et un progrès), soit complexe (pouvant contenir une séquence de tâches simples et/ou complexes).

Notre DTD est le suivant :

```
<!ELEMENT tasklist (simpleTask|complexTask)*>
<!ELEMENT simpleTask EMPTY>
<!ATTLIST simpleTask
    description CDATA #REQUIRED
    dueDate CDATA #REQUIRED
    priority CDATA #REQUIRED
    estimatedDate CDATA #REQUIRED
    progress CDATA #REQUIRED>
<!ELEMENT complexTask (simpleTask|complexTask)*>
<!ATTLIST complexTask priority CDATA #REQUIRED>
```

```
<!ELEMENT tasklist (simpleTask|complexTask)*>
<!ELEMENT simpleTask EMPTY>
<!ATTLIST simpleTask
    description CDATA #REQUIRED
    dueDate CDATA #REQUIRED
    priority CDATA #REQUIRED
    estimatedDate CDATA #REQUIRED
    progress CDATA #REQUIRED>
<!ELEMENT complexTask (simpleTask|complexTask)*>
<!ATTLIST complexTask priority CDATA #REQUIRED>
```

8 Bilan du respect des principes d'architecture

8.1 Principe de responsabilité unique (Single Responsibility Principle)

Chaque classe a été conçue avec une responsabilité unique et bien définie. Par exemple, l'interface TaskComponent gère le comportement des différents types de tâche, tandis que la classe TaskList gère la liste des tâches. Cette séparation des préoccupations facilite la maintenance et l'évolution du code. Par contre, nous avons enfreint ce principe dans la classe TaskInputDialog.

8.2 Principe ouvert/fermé (Open/Closed Principle)

L'architecture a été conçue de manière à ce que les classes soient ouvertes à l'extension mais fermées à la modification. Par exemple, l'utilisation de l'héritage et de l'interface TaskComponent permet d'ajouter de nouveaux types de tâches (Simple, complexes, etc.) sans modifier le code existant.En général, on a respecté le principe partout sur le projet surtout en utilisant le patron de conception Visitor.

8.3 Principe de substitution de Liskov (Liskov Substitution Principle)

Les classes dérivées (SimpleTask, ComplexTask) sont des sous-types substituables de leur classe de base (Task). Elles respectent les contrats définis par la classe de base et peuvent être utilisées de manière transparente dans le code existant. Et c'est ainsi qu'on

a fait pour toutes les autres interfaces dans notre système.

8.4 Principe d'inversion de dépendances (Dependency Inversion Principle)

Les dépendances ont été inversées de manière à ce que les modules de haut niveau ne dépendent pas des modules de bas niveau, mais plutôt d'abstractions. Par exemple, la classe TaskEditor dépend de l'interface TaskList plutôt que d'une implémentation spécifique, ce qui facilite le remplacement de l'implémentation si nécessaire.

8.5 Principe d'inversion de dépendances (Dependency Inversion Principle)

Les interfaces ont été conçues de manière à être cohérentes et spécifiques à leur rôle. En général, elles ne contiennent que les méthodes nécessaires à leur fonctionnalité, évitant ainsi les interfaces monolithiques et favorisant la réutilisabilité du code. Par contre, nous avons enfreint ce principe dans l'utilisation du patron composite. l'interface regroupe beaucoup de méthodes chose qui enfreint le principe d' ISP : nous pouvons privilégier l'utilisation de plusieurs interfaces qui n'est pas aussi évidente.

9 Conclusion

Ce projet de développement d'une application de gestion de tâches a permis de mettre en pratique les concepts et principes d'architecture logicielle vus en cours. Grâce à une conception rigoureuse et à l'utilisation judicieuse de patrons de conception, nous avons pu créer une solution modulaire, extensible et respectant les bonnes pratiques de développement.

L'application résultante, composée de l'éditeur de tâches "tsk-edit" et de l'analyseur de tâches "tsk-top", offre une expérience utilisateur intuitive et efficace pour la gestion des tâches quotidiennes. Les fonctionnalités clés, telles que la création, la modification, la suppression et la sauvegarde des tâches, ont été implémentées avec succès, nous avons également ajouter une fonctionnalité qui permet de faire l'analyse graphique des tâches en plus de l'analyseur en ligne de commande.

Au cours de ce projet, nous avons appliqué divers patrons de conception, notamment les patrons de création, structurels et comportementaux sur un projet réaliste, afin de garantir une architecture solide et évolutive. Chaque choix de patron a été soigneusement motivé et documenté, en mettant en évidence les avantages et les compromis associés.

L'utilisation d'un format XML personnalisé pour la sauvegarde des listes de tâches a permis une gestion efficace des données, tout en offrant une flexibilité pour de futures extensions. La DTD fournie garantit la validité des fichiers XML générés.

Tout au long du développement, nous avons veillé au respect des grands principes d'architecture, tels que le principe de responsabilité unique, le principe ouvert/fermé et le principe d'inversion de dépendances. Ces principes ont guidé nos choix de conception et ont contribué à la maintenabilité et à l'évolutivité du code.
En conclusion, ce projet a été une expérience enrichissante qui a permis de consolider nos compétences en architecture logicielle et en conception orientée objet. Nous avons acquis une compréhension approfondie des avantages et des défis liés à l'utilisation de patrons de conception, ainsi que de l'importance d'une architecture bien structurée pour le développement d'applications robustes et évolutives.