

TA MELOG :

Conception d'un système de placement de containers sur une plateforme de stockage

Architecture du programme :

Le programme est bâti sur une architecture MVC (Modèle Vue Contrôleur). Le travail demandé pour le TA est inclus dans le modèle. Pour assurer un développement propre des différents modules M, V, et C, le comportement des différentes classes est défini à l'aide d'interfaces et de classes abstraites. On exploite ainsi au mieux le polymorphisme.

Modèle :

Toute la partie logique est totalement intégrée dans le modèle. Le modèle peut notifier les interfaces qui l'observent grâce à une implémentation du pattern observer. À cet effet, le modèle génère des Objets d'Échange (Beans) qu'il transmet à la vue.

Vue :

La vue transmet au contrôleur les interactions avec l'utilisateur. Nous n'avons pas pu finir la vue. Elle est fonctionnelle mais très limitée. Toute la logique étant contenue dans le modèle, la fin de l'implémentation de la GUI n'est pas intéressante sur le plan du TA.

Stockage en base de données :

Nous avons utilisé le système de Gestion de Bases de Données SQLite, qui nous a paru le plus adapté pour cet exercice. Les bibliothèques Java permettent aussi une totale abstraction vis à vis du SGBD. La connexion à la base de données se fait à l'aide d'un pattern Singleton (classe DbConn). L'idéal eut été de faire un pool de connexions, mais bon...

Il est de plus à noter que les types de containers (normaux, frigorifiques, surtarifés) ne sont pas codés en dur dans le code (sauf dans le constructeur de la classe **Stockage**), ce qui garantit une évolution plus facile du programme.

On peut noter que toute la complexité algorithmique du modèle se situe dans les requêtes SQL. En effet, c'est la requête SQL qui permet de trouver, compter les places libres ou occupées.

Les requêtes sont commentées dans le code (classes **Stockage** et **Attente**). Les points récurrents de ces requêtes sont les suivants :

- Récupération des emplacements vides à l'aide d'une jointure gauche entre emplacement et container
- Récupération des containers stockés (et uniquement ceux-là) à l'aide d'une jointure naturelle
- Utilisation d'une clause ORDER BY type_id DESC pour traiter en priorité les containers les plus importants
- Utilisation d'une clause ORDER BY type_id ASC pour libérer en priorité les emplacements normaux utilisés pour des containers frigorifiques ou surtarifés
- Utilisation de clauses GROUP BY pour compter les containers par catégorie

Jeux d'essais :

Les jeux d'essais sont pris en charge par des tests unitaires. Le plugin Surefire Report de Maven permet de lister les résultats des tests, donc des jeux d'essais, sous forme plus agréable à l'œil. (cf annexe)

Versionnement et gestion du projet :

Le programme est géré par Maven et le versionnement se fait grâce à Git. Il est d'ailleurs possible de consulter le repository Git sur GitHub : <http://github.com/divarvel/TA-Melog>

Maven fournit des outils pour toutes les phases du projet, y compris la mise en place de tests unitaires grâce au framework Junit. Les jeux d'essais sont donc réalisés grâce aux tests unitaires. Maven génère ainsi un site web (*fourni en annexe*) regroupant les rapports d'erreurs et les informations utiles au projet (licence, auteurs, **javadoc**...) Maven permet de plus de générer un package jar comportant les classes compilées du programme, ainsi que les dépendances nécessaires à la compilation ou à l'exécution.

L'archive fournie contient le **package Jar généré**, les dépendances (nécessaires à la compilation et à l'exécution), un script permettant de lancer rapidement l'exécutable, **le site généré par Maven**, les **sources** (dont les tests), ainsi que le pom.





ToDo :

Il reste cependant quelques tâches (pas toutes indispensables... ou réalisables) :

- Finir la GUI
- Englober les fermetures des ResultSet dans un bloc finally (si une exception est levée avant la fermeture du ResultSet, la base est bloquée en écriture, ce qui n'est pas top)
- Optimiser les notifications : Ne générer les Objets d'échange (Bean) que si au moins un observer est attaché au modèle.
- Déporter les informations variables (telles que les types de containers et le nombre de places dans la zone de stockage) dans un fichier de conf XML. Ces données sont toutes regroupées à un seul endroit du code, le refactoring ne serait donc pas trop douloureux.
- Arranger tout le code pour obtenir un style conforme aux specs définies par Sun. (Pour l'instant, c'est pas la joie, malgré nos efforts)

Annexe :

com.ecnmelog.model

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	StockageTest	12	0	0	0	100%	2.574
	AttenteTest	7	0	0	0	100%	1.026
	StockageTest	12	0	0	0	100%	2.711
	ContainerTest	5	0	0	0	100%	0.137

com.ecnmelog.app

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	AppTest	1	0	0	0	100%	0.002

StockageTest













	testEmpty	0.066
	testRemoveContainerById	0.11
	testRemoveContainerByType	0.067
	testInitEmplacement	0.06
	testCountContainer	0.088
	testCountContainers	0.05
	testCountEmplacementsDispo	0.064
	testCountEmplacementsDispoByType	0.136
	testRemoveContainersByType	0.087
	testGetEmplacementLibre	0.115
	testTraiterAttente	0.148
	testTraiterAttenteOverFlow	1.573

Diagramme de Classes (1/3)

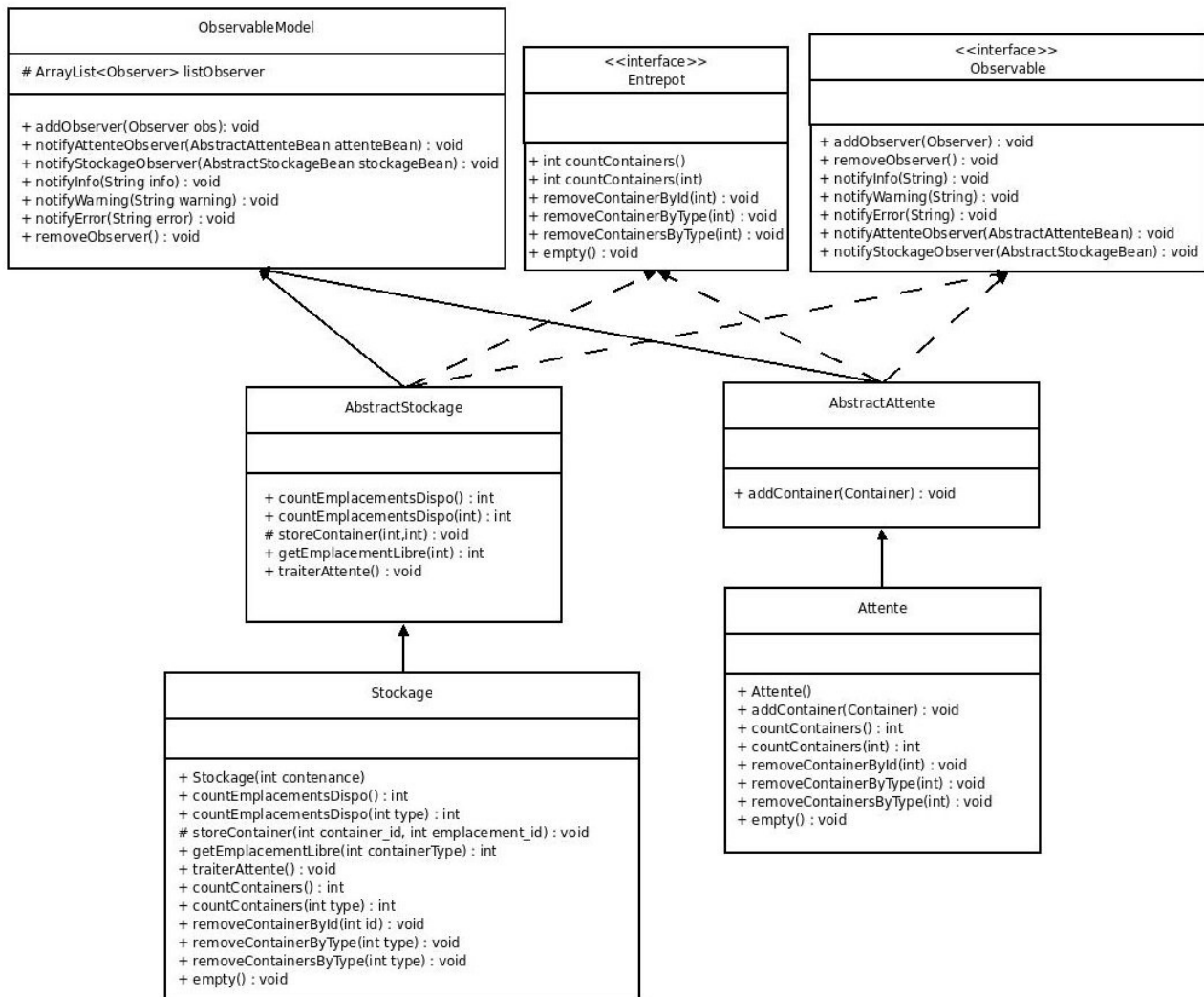


Diagramme de Classes (2/3)

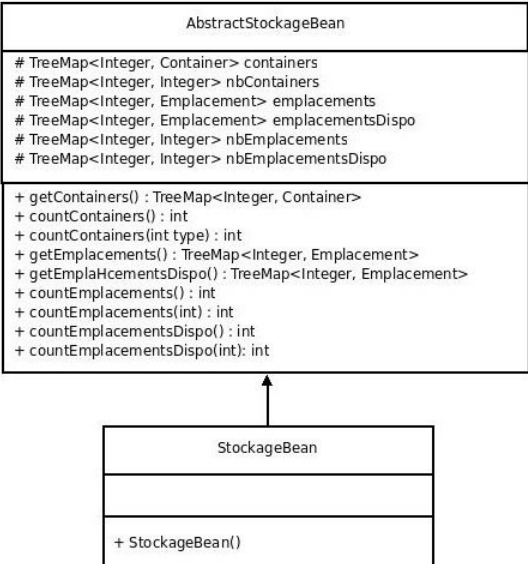
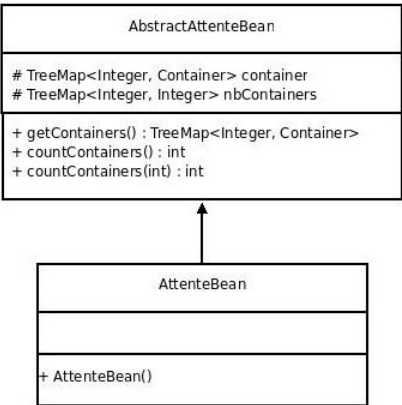


Diagramme de Classes (3/3)

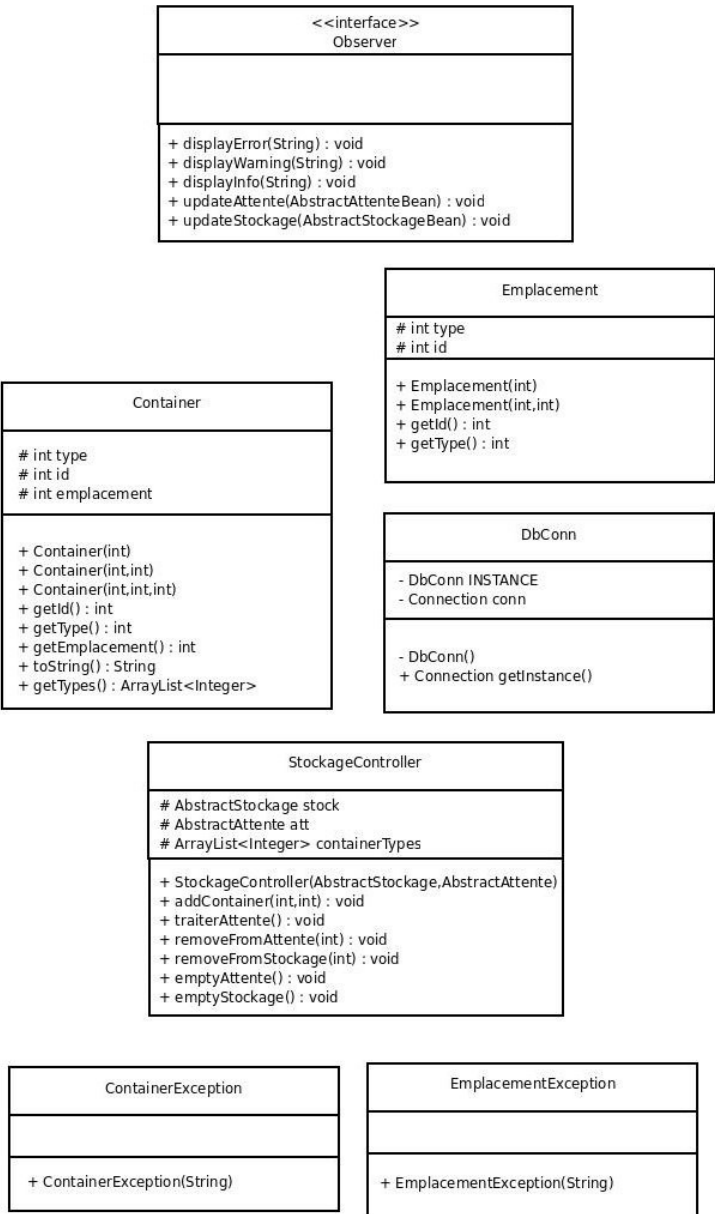


Schéma physique de la Base de Données

