Modélisation d'un framework fondé sur des patrons de conception temps réel

Naoufel Machta

UNIVERSITÉ DE TUNIS EL MANAR
FACULTÉ DES SCIENCES DE TUNIS

ECOLE DOCTORALE EN INFORMATIQUE

MÉMOIRE DE D.E.A

présenté en vue de l'obtention du

Diplôme d'Études Approfondies en Informatique

par

Naoufel MACHTA
(Maîtrise Informatique, FST)

Modélisation d'un framework fondé sur des patrons de conception temps réel

soutenu le 08 avril 2006, devant le jury d'examen

MM.	**Yahya SLIMANI**	***Président***
	Henda BEN GHZALA	***Membre***
	Samir BEN AHMED	***Directeur du mémoire***
	Adel KHALFALLAH	***Invité***

Au Maître
qui m'a montré
le chemin ...

Remerciments

Je tiens à remercier tout particulièrement Monsieur Yahya SLIMANI, Maître de conférence à la Faculté des Sciences de Tunis, pour avoir accepté de me faire l'honneur de présider le jury de mon DEA.

Je tiens à remercier tout particulièrement Madame Henda BEN GHZALA, Professeur à l'École Nationale des Sciences Informatiques, pour avoir accepté d'être membre du jury.

Je tiens à remercier tout particulièrement Monsieur Adel KHALFALLAH, Maître assistant à l'Institut Supérieur d'Informatique, pour m'avoir encadré, mais aussi pour sa confiance, ses encouragements et son enthousiasme.

Je voudrais exprimer toute ma reconnaissance à Monsieur Samir BEN AHMED, Professeur à l'Institut National des Sciences Appliquées et de Technologies pour m'avoir accueilli au sein de l'équipe, pour m'avoir facilité toutes les tâches administratives et surtout pour son soutien morale.

Je remercie également tous les membres de l'unité de recherche MOSIC qui ont su instaurer une bonne ambiance.

Je suis très reconnaissant à tous mes collègues de l'Institut Supérieur d'Informatique pour leurs encouragements.

Enfin je remercie aussi tous les membres de ma famille et tous ceux qui par un conseil, un encouragement ou un sourire m'ont soutenu pour mener ce travail à son terme.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Les systèmes temps réel embarqués sont de plus en plus utilisés dans notre vie quotidienne. Ces systèmes sont très variés car ils sont très influencés par l'architecture matérielle de leurs environnements d'application, c'est pourquoi ces systèmes sont généralement développés à partir de zéro (*from scratch*).

La modélisation orientée objet devient de plus en plus adoptée dans le développement des systèmes temps réel et UML (*Unified Modeling Language*) est de plus en plus utilisé dans ce domaine [Douglass, 1999, Douglass, 2004, Gomaa, 2000].

La combinaison de la technologie de l'orienté objet et les design patterns pour la conception et le développement des frameworks temps réel peut réduire énormément le coût et la duré de développement des applications temps réel.

Les design patterns [Gamma *et al.*, 1995] sont généralement connus comme des abstractions des solutions de problèmes récurrents. Ils sont des modèles de conception définissant les rôles, les relations entre classes et objets, et le mécanisme de résolution d'un problème donné dans un contexte bien déterminé. La structure décrivant un design pattern comporte principalement le nom, le but, la solution et la conséquence. Depuis le travail initial de Gamma qui n'été pas spécifique à la conception des systèmes temps réel, beaucoup de design patterns temps réel ont été identifiés [Douglass, 1999, Douglass, 2002].

Les frameworks ont fait l'objet de plusieurs travaux de recherche durant les deux dernières décennies. Fayad [Fayad *et al.*, 1999] pense que les frameworks seront au centre des intérêts des chercheurs dans le domaine du génie logiciel pour les années à venir. Un framework est un ensemble de classes coopérantes qui constituent une architecture réutilisable pour la conception d'une classe spécifique de logiciel [Gamma *et al.*, 1995]. Un framework peut être définit aussi comme une application semi complète (partiellement réalisée) qu'on adapte ou spécialise pour une application spécifique [Johnson and Foote, 1988]. Un framework est spécifique à un domaine d'application, son avantage est qu'il facilite le développement des applications et réduit les coûts et le temps de développement. Ce dernier, encourage la réutilisation de la conception et du code. Toutefois la réutilisation du code présente un inconvénient aussi ; le framework est très dépendant d'un langage de programmation particulier.

D'autre part, et afin de préciser notre domaine de recherche, notre travail se situe dans le cadre de développement des applications temps réel embarquées. Notre objectif est de fournir un framework temps réel qui se place dans la tendance nationale actuelle qui consiste à promouvoir les logiciels libres. La conception du framework est basée sur le principe de séparation des préoccupations. Le framework est composé de sous-frameworks où chacun traite une problématique du domaine. L'architecture des sous-frameworks est fondée sur un ensemble de design patterns traitant la même préoccupation mais d'une manière différente. A l'état actuel le framework comporte deux

sous-frameworks ; un sous-framework de concurrence et un sous framework de tolérance aux fautes.

Les travaux menés dans ce mémoire sont rapportés en quatre chapitres :

Le premier chapitre est consacré à la présentation de la notion de pattern. Nous commençons par un historique de l'idée de pattern et de design pattern. La section suivante donne une idée sur l'utilisation de cette notion dans le domaine de l'orienté objet et sur les formalismes de sa description. Nous introduisons ensuite quelques concepts liés aux patterns. L'application des design patterns dans le domaine du temps réel est discutée dans la section suivante. Ce chapitre est clôturé par la présentation des travaux de recherches s'adressant aux problématiques liées aux patterns et une conclusion.

Le second chapitre discute le concept de framework. En premier lieu nous présentons des définitions suivies par quelques avantages et quelques caractéristiques des frameworks. Dans la section suivante nous discutons leur classification. La suite de ce chapitre est consacrée à l'application des frameworks dans le domaine du temps réel. Nous commençons par présenter les spécificités des frameworks dans ce domaine, ensuite nous présentons des exemples de frameworks temps réel. La section suivante étudie l'architecture de deux frameworks temps réel différents. Nous terminons ce chapitre par une conclusion.

Dans le troisième chapitre nous présentons notre expérience dans la conception d'un framework horizontal pour les systèmes temps réel embarqués. Nous commençons par présenter l'intérêt d'un tel framework, ensuite nous présentons brièvement le processus suivi lors de la conception. Dans la suite de ce chapitre nous discutons l'architecture du framework et ses différents composants. Le framework et composé de deux sous-frameworks, nous présentons l'architecture de chacun et les design patterns qui le forment. La structure de chaque design pattern lors de son intégration au framework est aussi illustrée. En fin, nous terminons ce chapitre par une conclusion.

Le quatrième chapitre est consacré à la présentation des exemples d'utilisation du framework pour le développement des applications temps réel

embarqués. Nous présentons deux exemples : le premier illustre l'application du sous-framework de concurrence et traite le cas d'un bras de robot dans le milieu industriel, alors que le deuxième présente l'application du sous framework de tolérance aux fautes et traite le cas d'un système d'acquisition.

Nous clôturons le mémoire par une conclusion générale dans laquelle nous résumons les principaux résultats suivis des perspectives.

Chapitre 1

Patterns et design patterns

1.1 Introduction

Les patterns ont été introduits par Christopher Alexander qui voulait présenter ses expériences d'architecte sous formes de modèles à partir desquels on peut concevoir des maisons et des espaces confortables. Pour lui : « *Chaque pattern décrit un problème qui se manifeste constamment dans notre environnement et décrit le cœur de la solution à ce problème, de telle façon qu'elle puisse être réutilisée des millions de fois et jamais deux fois de la même manière* » [Alexander, 1977].

Dans la littérature on trouve deux traductions du mot anglais pattern : Modèle de conception et Patron de conception. Mais le terme anglais est aussi fréquemment utilisés dans les textes français. Nous avons choisi de le conserver le long de ce document afin de préserver tout le sens du terme.

La notion de pattern et langage de patterns présentée par Alexander a inspiré le comité de la conception orientée objet. Les patterns présentent une bonne méthode pour la documentation de l'expérience afin de l'étudier, la communiquer, et l'améliorer. L'expérience dans le domaine de la conception orientée objet pèse lourd. Elle distingue nettement les développeurs chevronnés des jeunes développeurs qui sont à leurs premiers pas dans la conception et la modélisation orientée objet. La procédure d'apprentissage quand à elle, n'est pas sans difficulté.

L'intégration de cette notion dans le domaine de conception orientée objet a commencé vers les années 90 avec plusieurs travaux de recherche sur ce thème. Le plus célèbre de ces travaux est la thèse de doctorat de E. Gamma qui s'est transformé plus tard en un livre à l'aide de trois autres collaborateurs travaillant sur ce thème. Ils formaient alors, ce qui est connu dans le comité scientifique par GoF (*Gang of Four*) et leur livre devient le livre de référence des modèles de conception orientés objet appelés design patterns [Gamma *et al.*, 1995].

1.2 Design Pattern

Les travaux sur les design patterns ont commencé vers la fin des années quatre-vingts lorsque Kent Beck et Ward Cunningham ont adopté l'idée d'Alexander pour présenter cinq patterns décrivant la conception des interfaces utilisateur basées sur des fenêtres [Beck and Cunningham, 1987]. L'idée a fasciné d'autres chercheurs intéressés par le domaine de conception orientée objet [Busmann, 1993, Gamma, 1992, Johnson, 1992] et d'autres travaux ont étés présentés. Mais les design patterns ont connus leur maturité avec la publication du GoF de leur catalogue des design patterns [Gamma *et al.*, 1995] qui a connu un grand succès dans le communauté de la conception orientée objet. L'uns des principaux atouts de ce livre est sa facilité de lecture et sa présentation pédagogique des design patterns grâce à une technique de présentation des patterns ou formalisme de design patterns.

1.2.1 Définitions

Un design pattern est une forme de documentation de l'expérience dans le domaine de la conception orientée objet. Le concepteur expérimenté résout facilement des problèmes qui se répètent et qui ne sont pas forcément identiques. Un design pattern constitue une abstraction de la solution de ces problèmes.

« *Un design pattern donne un nom, isole et identifie les principes fondamentaux d'une structure générale, pour en faire un moyen utile à l'élaboration d'une conception orientée-objet réutilisable* » [Gamma *et al.*, 1995].

Les design patterns sont des modèles de conception définissant les rôles, les relations entre classes et objets , et le mécanisme de résolution d'un problème donné dans un contexte bien déterminé.

« *Ils facilitent la réutilisation de solutions de conception et d'architectures efficaces* » [Gamma *et al.*, 1995].

Généralement, un design pattern décrit la solution du problème de conception en présentant les différentes interactions et relations entre les acteurs principaux, et en définissant le rôle de chacun. Dans le contexte de la conception orientée objet, ces acteurs sont des classes ou des instances de classes formant une collaboration. Afin d'illustrer et de clarifier la solution, les design patterns sont présentés suivant un formalisme bien établi.

1.2.2 Formalisme des design patterns

Un pattern décrit un problème, une solution et le contexte dans lequel cette solution est considérée. Il n'existe pas dans la littérature de consensus sur un formalisme unifié de description des design patterns. Mais, en général un formalisme de description des design patterns possède quatre éléments de base [Gamma *et al.*, 1995] :

- **Nom** : nom du pattern avec éventuellement d'autres noms reconnus pour le même pattern.
- **Problème** : description de la situation que doit résoudre le pattern.
- **Solution** : description de la structure, du comportement et de la résolution du problème dans le contexte donné.
- **Conséquences** : évolution du contexte après utilisation du pattern, problèmes résolus et ceux qui ne le sont pas.

Le *nom* est l'élément clé du design pattern, il permet de donner brièvement une description du problème de conception et donne un vocabulaire de conception facilitant la communication entre collègues.

Le *problème* expose le sujet à traiter et son contexte. Il permet de donner une idée claire sur la situation dans laquelle le design pattern est utilisé.

La *solution* est le plus important aspect du pattern. Elle identifie les éléments du pattern et leurs rôles dans leurs relations les uns avec les autres.

Les *conséquences* sont les effets résultants de la mise en œuvre des design patterns. Ils sont importants lorsqu'on est amené à choisir un, parmi plusieurs patterns.

Dans la littérature on distingue deux types de formalismes pour la description des design patterns ; formalismes narratifs et formalismes structuraux.

Le formalisme narratif a été utilisé par Alexander qui a présenté ses design patterns selon le modèle textuel suivant [Lea, 2000] :

SI `vous êtes dans un` **CONTEXTE**
`par exemple` **EXEMPLE**
`avec un` **PROBLÈME**
`nécessitant des` **FORCES**
Alors `pour des` **RAISONS**
`appliquer` **MODÈLE OU/ET RÈGLE**
`pour construire la` **SOLUTION**
`menant à`
NOUVEAU CONTEXTE `et` **AUTRES PATTERNS**

Les formalismes structuraux offrent plus de confort pour le lecteur et permettent une bonne compréhension des design patterns, des problèmes auxquels ils apportent solutions, et de leurs contextes d'application. La richesse du formalisme donne une meilleure vision du design pattern et une meilleure compréhension de son principe de résolution du problème donc une meilleure utilisation. Comme exemple de formalisme structural nous présentons par la suite le formalisme de GoF [Gamma *et al.*, 1995] :

Noms de modèle et classification : Le nom du pattern contient succinctement son principe. Un bon nom est vital, car il va faire partie du vocabulaire du concepteur.

Intention : C'est une courte déclaration qui répond aux questions suivantes : que fait effectivement le modèle de conception ? Quel est sa raison d'être et son but ? Quel cas ou quel problème particulier de conception concerne-il ?

Alias : Quelques autres noms reconnus du modèle, s'il en existe.

Motivation : C'est un scénario qui illustre un cas de conception, et qui montre comment la structure de classe et d'objet du modèle contribuent à la solution de ce cas. Ce scénario aide à comprendre des descriptions plus abstraites du modèle.

Indication d'utilisation : Quels sont les cas qui justifient l'utilisation du modèle de conception ? Quelles situations de conception peu satisfaisantes peuvent tirer avantage de l'utilisation du modèle ? Comment reconnaître ces situations ?

Structure : C'est une représentation graphique des classes du modèle, qui utilise une notation orientée object généralement UML [Rumbaugh *et al.*, 2004].

Constituants : Les classes et/ou les objets intervenant dans le modèle de conception, avec leurs responsabilités.

Collaborations : Comment les constituants collaborent-t-ils pour assumer leurs responsabilités ?

Conséquences : Comment le modèle de conception assume-t-il ses objectifs ? Quels sont les compromis qu'implique son utilisation et quel en est l'impact ? Quelles parties de la structure d'un système permet-il de modifier indépendamment ?

Implémentation : De quels pièges, astuces, ou techniques, faut-il être averti lors de l'implémentation du modèle, y a -t-il des solutions typiques du langage utilisé ?

Exemples de code : Ce sont des extraits de programmes, qui illustrent la façon qu'il convient d'employer pour développer le modèle en langage orienté objet (C++, Smalltalk, Java, ...).

Utilisations remarquables : Exemples de modèles appartenant à des systèmes existants.

Modèles apparentés : Quels modèles de conception sont en relation étroite, avec celui traité ? Quelles sont les différences importantes ? Avec quels autres modèles peut-ils être employé ?

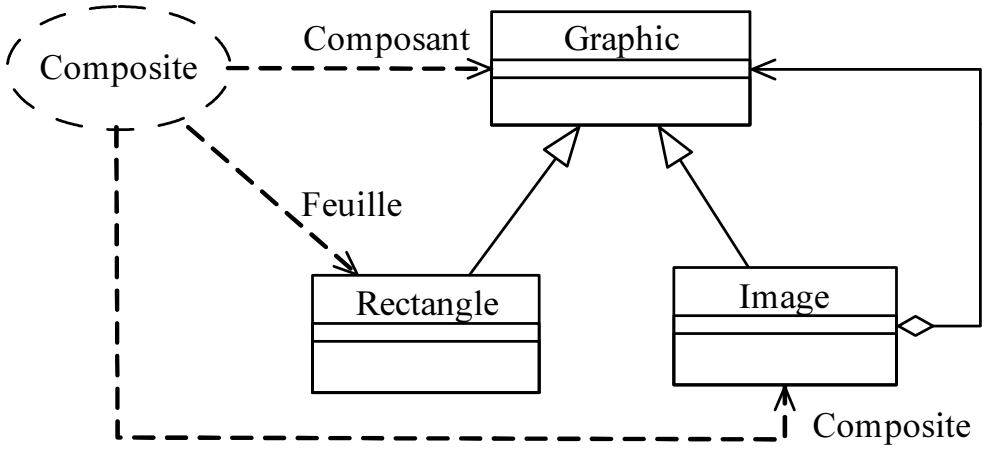

FIG. 1.1 – Notation d'une occurrence d'un design pattern avec UML [Sunyé *et al.*, 2000]

La structure ou la représentation graphique du design pattern se fait généralement par une notation de modélisation graphique orientée objet quelconque. Cependant, Le langage de modélisation unifié (UML) [Rumbaugh *et al.*, 2004] devient le langage de modélisation préféré pour les concepteurs. Par sa richesse, UML s'est imposé pour être le plus utilisé dans la description des structures des design patterns et leurs notations. Ils sont notés avec une forme ovale contenant le nom du design patterns connecté avec des flèches en lignes discontinues aux classes participantes. La figure 1.1 présente une occurrence du design pattern *Composite* [Gamma *et al.*, 1995]. Cette notation est utilisée généralement dans UML pour indiquer une collaboration de classes . L'utilisation du mécanisme de collaboration pour noter les design patterns est une extension naturelle[1] du langage UML car enfin de compte un design pattern forme une collaboration de classes. Malgré que ce mécanisme soufre de manque de précision [Sunyé *et al.*, 2000] il est très pratique et utile pour faire remarquer une application du design pattern dans un modèle sans passer du temps à observer les détails compliqués de la conception.

[1]Cette extension est naturelle car le mécanisme existe deja, il y a eu seulement une extension de son utilisation.

1.2.3 Utilisation des design patterns dans le processus de développement

L'expérience dans le domaine de développement et conception des logiciels est un facteur très important. Avec chaque programme développé des dizaines de leçons sont assimilées. Ces leçons sont les résultats des erreurs et des problèmes rencontrés dans le processus de développement et pour lesquels les développeurs ont trouvé, testé et optimisé des solutions aboutissants à des résultats acceptables. L'un des avantages les plus importants des designs patterns est la documentation de cette expérience pour la rendre disponible et accessible pour les novices et les experts du domaine. En réalité les design patterns ne sont pas des solutions prêtes à utiliser directement comme des morceaux de puzzle à coller pour construire une application. L'utilisation des design patterns nécessite un effort de la part du développeur (ou concepteur) afin d'assurer une bonne intégration dans le modèle de l'application. Douglass [Douglass, 2002] définit une stratégie de choix du bon pattern appelé encore *pattern hatching* en s'inspirant de Vlissides [Vlissides, 1998]. Sa stratégie est définie par sept étapes :

1. Se familiariser avec les design patterns.
2. Caractériser le problème de conception rencontré.
3. Modéliser le problème caractérisé.
4. Identifier une solution possible.
5. Evaluation de la solution et en fonction du résultat passer à l'étape 6 ou revenir aux étapes 3, 2 ou même 1.
6. Instancier le pattern et l'adapter avec le projet de conception.
7. Tester la solution.

Cette démarche résumé par la figure 1.2 n'est pas seulement valable pour l'application des design patterns mais en plus, elle permet d'en trouver des nouveaux. Extraire les patterns ou "*pattern mining*" est un autre terme utilisé par Douglass pour désigner un autre domaine lié au design patterns qui consiste à extraire les patterns à partir des conceptions déjà existantes.

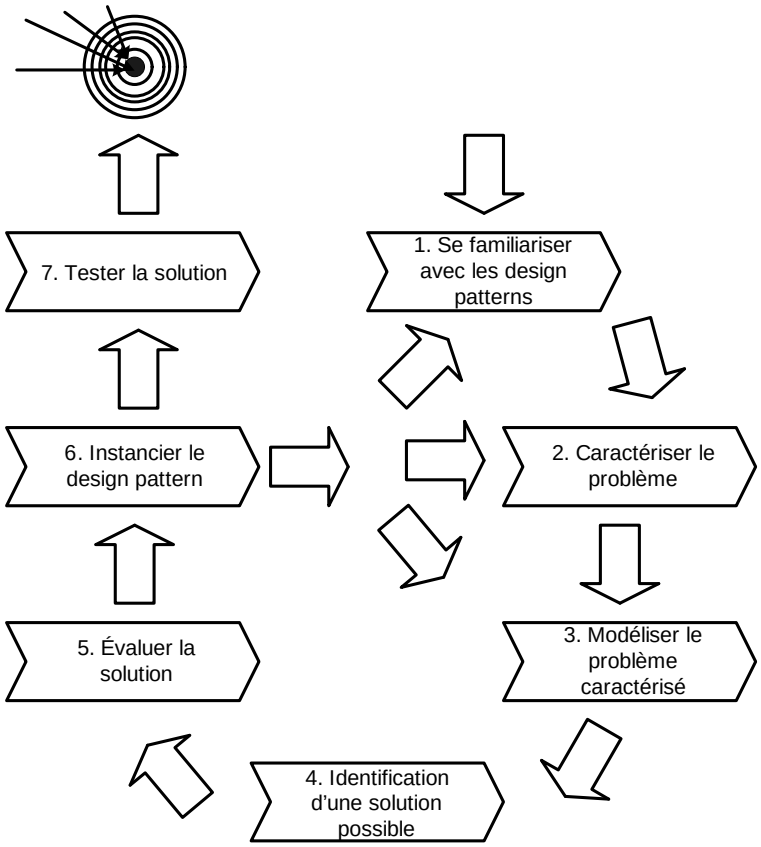

FIG. 1.2 – Pattern Hatching

1.2.4 Classification des design patterns

Dans [Gamma *et al.*, 1995] les design patterns sont classés selon deux critères. Le premier critère concerne ce que fait le design pattern, il est appelé Rôle. Les design patterns peuvent avoir un des rôles suivants : créateur, structurel ou comportemental. Les design patterns créateurs concernent le processus de création d'objets. Les design patterns structuraux s'occupent de la composition de classes ou d'objets. Les design patterns de comportement spécifient les façons d'interagir de classes et d'objets et de se répartir les responsabilité.

Le second critère précise si le design pattern s'applique à des classes ou à des objets. Il est appelé Domaine. Les design patterns de classes traitent des

		Rôle		
		Créateur	Structurel	Comportement
Domaine	Classe	Fabrication	Adaptateur	Interprète
				Patron de méthode
	Objet	Fabrique abstraite	Adaptateur	Chaîne de reponsabilité
		Monteur	Pont	Commande
		Prototype	Composite	Itérateur
		Singleton	Décorateur	Médiateur
			Façade	Mémento
			Poids Mouche	Observateur
			Procuration	Etat
				Stratégie
				Visiteur

TAB. 1.1 – Classification des design patterns de [Gamma *et al.*, 1995]

relations entre les classes et leurs sous-classes. Les design patterns d'objets traitent des relations entre objets, qui peuvent être modifiées

Le Tableau 1.1 présente la classification des design pattern du GoF dans [Gamma *et al.*, 1995].

1.3 Idiomes

Les idiomes sont similaires aux design patterns mais ils sont à un niveau d'abstraction plus bas. Au contraire des design patterns, ils sont très liés aux langages de programmation . Les idiomes sont des bonnes « astuces » de programmation permettant une meilleure détection des erreurs, ou des instructions optimisés permettant une meilleure gestion de la mémoire, ainsi qu'un temps d'exécution réduit. Les idiomes définissent des structures réutilisable tenant compte des spécificités d'un langage. Ils sont généralement illustrés avec des lignes de code.

Dans la littérature, les idiomes les plus connus sont ceux du langage C++ grâce à Coplien [Coplien, 1992]. Si ces idiomes sont spécifiques au langage C++, ils restent aussi applicables à d'autres langages qui lui sont semblables comme le langage C ou Java. L'exemple suivant illustre un idiome permettant de détecter l'erreur liée à l'utilisation de l'opérateur d'affectation au lieu de l'opérateur d'égalité :

```
if ( Constante = =  variable)
 {
     // code si " variable " vaut Constante
     ...
 }
```

Dans les langages tel que C++, C ou Java, l'opérateur d'égalité (=) est utilisé pour l'affectation, alors que l'opérateur « double égale » (==) est utilisé comme opérateur d'égalité. L'utilisation de l'opérateur d'affectation dans l'expression à évaluer avec l'instruction « if » ne génère pas une erreur lors de la compilation car l'expression est syntaxiquement correcte. L'idiome nous donne un moyen simple permettant d'éviter ce problème. Mettre l'opérande gauche comme constante permet au compilateur de signaler une erreur si on a saisi un seul « = » au lieu de « = = ».

L'utilisation des idiomes dans la programmation permet de gagner du temps et d'optimiser le code source. Mais ceci peut générer des problèmes de communication du code source dans un groupe de programmeurs ou les membres ne partagent pas les mêmes idiomes.

1.4 Langage de pattern

Lorsque Alexander a présenté ses patterns d'architecture pour la construction des espaces confortables, il les a présenté sous forme de langage. Le terme

langage pour lui est un moyen d'exprimer les espaces conçu en fonction des patterns.

Le langage de pattern donne une aide pour la conception dés le début jusqu'à la fin. Un langage de pattern est plus qu'un catalogue de pattern présentant une description individuelle des pattern. Dans le langage de pattern les relations inter patterns ainsi que leur ordre d'utilisation sont bien définis. Le terme « système de pattern » est plus réputé que le terme « langage de pattern » dans le domaine du génie logiciel [Buschmann *et al.*, 1996].

1.5 Antipattern

Les antipattern [Brown *et al.*, 1998] sont des patterns à éviter à tout prix. Au contraire des design patterns, les antipatterns sont des mauvaises pratiques, des solutions mal choisies mais qui sont très répandues pour des problèmes donnés, ils conduisent généralement à des problèmes plus sérieux que les problèmes originaux. Présentés à l'image des catalogues des design patterns, les antipattern sont accompagnés de solutions potentielles.

Au contraire des design patterns leur connaissance n'est pas importante avant la conception car il vaut mieux apprendre les bonnes habitudes pour les appliquer que d'apprendre les mauvaises pour les éviter.

1.6 Pattern processus

Un pattern processus est un pattern qui décrit une approche réussie prouvée ou/et une série d'actions pour le développement des logiciels [Ambler, 1998].

Une caractéristique importante des patterns processus est qu'ils décrivent ce qu'il faut faire sans détailler comment le faire. Les patterns processus sont donc le point de départ de la réalisation d'un projet informatique [Khayati, 2001].

1.7 Design Patterns temps réel

La réutilisation est aussi importante dans le développement des systèmes temps réel et embarqués que dans le développement d'autres applications. L'expérience aussi, joue un rôle très important dans le développement de telles applications. Les design patterns accomplissent la tâche de documentation de l'expérience et permettent la réutilisation de bonnes solutions aux problèmes de conception résolus.

Dans la littérature on rencontre plusieurs design patterns dits « temps réel ». Ce terme fait allusion à des design pattern traitant des aspects temporels, alors que ceci n'est pas généralement le cas. En effet, les design patterns temps réel traitent des problèmes rencontrés dans la conception des systèmes temps réel.

Les design patterns temps réel varient selon leurs domaines d'application et selon l'approche de conception. On distingue deux familles de design patterns selon l'approche de conception :

- les design pattern temps réel orientés objet,
- les design patterns temps réel non orientés objet.

Les design patterns temps réel orientés objet sont similaires aux design patterns du GoF, cependant ils traitent des aspects spécifiques à leur domaine d'application, comme la synchronisation , la concurrence, ...

Dans son étude des design patterns temps réel orientés objet, et relativement aux domaines d'applications des design patterns, Mckegney distingue deux grandes familles [McKegney, 2000] :

- design patterns généraux,
- design patterns spécifiques.

Les design patterns généraux sont des design patterns « largement utilisés » et qui traitent des aspects communs aux différents systèmes temps réel. Cette famille intègre les design patterns de Douglass [Douglass, 2002] et de Schmidt [Schmidt *et al.*, 2000].

L'autre famille comprend des design patterns spécifiques à des domaines temps réel particuliers comme par exemple le domaine de télécommunications ou encore le domaine de l'aviation.

D'après Douglass, la conception d'un système passe par trois phase [Douglass, 2004] :

Conception architecturelle (Architectural Design) : cette phase définit les décisions de conception stratégiques qui affectent la totalité ou la majorité de l'application. Cette phase comprend le modèle de déploiement, le modèle physique et le modèle de concurrence. Les design patterns appliqués dans cette phase sont appelés design patterns d'architecture ou « *architectural design patterns* ».

Conception mécanistique (Mechanistic design) : cette phase consiste à optimiser le comportement des collaborations formant le système ou l'application. Les design pattern appliqués dans cette phase sont nommés design patterns mécanistiques ou «*mechanistic*» design patterns.

Conception détaillée (Detailed design) : dans cette phase des optimisation de bas niveau sont ajoutées pour optimiser le système final. Les design patterns appliqués dans cette phase sont appelés design patterns détaillés (*detailed design pattern*).

Les design patterns détaillés sont très proches des idiomes. Ils caractérisent la conception à bas niveau. Au contraire des idiomes, les design patterns détaillés ne sont pas dépendants d'un langage de programmation. Ils sont illustrés avec des pseudo code ou avec des diagrammes d'états transitions.

Selon Schmidt il faut distinguer entre :

Design pattern un design pattern capte l'aspect dynamique et statique d'une collaboration de classes censée donner une solution à un problème récurrent dans le développement des logiciels.

Pattern d'architecture un pattern d'architecture exprime un schéma d'organisation structurel et fondamental.

1.8 Travaux similaires

Vu l'importance des modèles de conception et l'utilisation des solutions prouvées dans le processus de conception des applications, plusieurs axes de recherche ont été orientés vers ce domaine. Ils visent une meilleure utilisation des design patterns.

Les travaux de recherche en relation avec les design patterns peuvent être classés selon deux grands axes :

- Les travaux qui visent une meilleure intégration des design patterns dans le processus de développement,
- les travaux qui visent la reconnaissance ou la détection des design patterns dans le code source pour une meilleur documentation.

Dans le premier axe, les travaux de recherche s'orientent vers des outils facilitant l'utilisation des design patterns. Dans [Pascal, 1999] l'outil présente une bibliothèque de design patterns représentés en UML. Au moment de la conception l'utilisateur choisi un design pattern de la bibliothèque et met en relation chaque classe du pattern avec une classe de son application. Et l'outil se charge, par la suite, de l'instanciation du design pattern. L'instanciation consiste à recréer pour un ensemble de classes, les associations, la hiérarchie ainsi que les propriétés nécessaires pour que les dites classes correspondent au pattern désiré [Pascal, 1999]. Le support des design patterns est maintenant intégré dans des outils commerciaux comme Rational Rose de IBM et Together de Borland. Les travaux de recherche s'orientent aussi, dans cette famille, vers d'autres techniques d'implémentation des design patterns. Les chercheurs essayent de trouver des techniques d'implémentation autres que celles du paradigme orienté objet basées sur les mécanismes d'héritage. Dans [Quintian and Lahire, 2001], la technique utilisée est basée sur un mélange d'approche de programmation comprenant la programmation orientée objet [Lau, 2000], la programmation par aspect [Elrad *et al.*, 2001] et la programmation orientée sujet [Harrison and Ossher, 1993]. Le principe est de rendre la structure du design pattern indépendante de son utilisation tout en assurant la flexibilité et la facilité de son emploi.

Dans le deuxième axe, les travaux de recherche s'orientent vers des méthodes pour l'identification et la reconnaissance des design patterns dans du code source, ce qui peut servir comme un moyen de mesure de la qualité de ce dernier (selon le nombre de design patterns utilisés) ou pour la documentation du code source ce qui facilite sa maintenance. Parmi les techniques utilisés pour la documentation du code source, on notera l'utilisation dans [Balanyi and Ferenc, 2003] du langage DPML (Design Pattern Markup Language) basé sur XML pour la description des design patterns.

1.9 Conclusion

Dans ce chapitre la notion de design pattern a été introduite. Leur origine ainsi que leurs applications dans le domaine du génie logiciel et en particulier le domaine de la conception orientée objet ont été présentés. Les design patterns forment un bon moyen de documentation de l'expérience. Ils établissent un haut niveau de réutilisation (réutilisation d'architecture logicielle) et permettent une meilleure communication entre les concepteurs.

Chapitre 2

Frameworks

2.1 Introduction

L'un des avantages de l'utilisation de l'approche orientée objet dans la conception et le développement des logiciels est la réutilisabilité. En plus de la réutilisation du code, l'approche orientée objet permet d'utiliser d'autres techniques comme l'héritages, les classes génériques (*template classes*) et les bibliothèques de classes.

Les frameworks présentent une technique de réutilisabilité, basée essentiellement sur les caractéristiques des langages orientés objet, de niveau plus haut que celui des bibliothèques des classes. En plus de la réutilisation du code sous ses différentes formes tel que classes ou procédures, les frameworks favorisent la réutilisation de la conception.

2.2 Définition

La définition la plus simple et la plus « courte » qu'on rencontre dans la littérature lorsqu'on essaye de définir un framework est : « *Un framework est une application semi complète* ». Malgré qu'il est difficile de donner une définition précise en quelques mots, cette définition est très significative. Une application orientée objet est un ensemble d'objets réalisant l'architecture définie par les relations entre les classes desquelles les objets sont des instances. En d'autres termes un framework est un squelette d'application qui peut être personnalisé par un développeur [Johnson, 1997]. Le squelette est l'ensemble de classes et architectures couvrant une grande famille d'applications dans un domaine donné.

Généralement, un framework est implémenté avec un langage orienté objet comme C++, Java ou Smaltalk [Fayad *et al.*, 1999] ce qui lui permet de profiter du polymorphisme, de l'abstraction des données, de l'héritages et des autres caractéristiques et avantages des langages orientés objet.

	Framework	Bibliothèque de classe
Réutilisation	Large	Faible
Domaine	Dépendant	Indépendant
Taux de réutilisabilité /code de l'application	Elevé	Faible
Réutilisation d'architecture	Oui	Non
Réutilisation du code	Oui	Oui
Classes / flux de contrôle	Actives	Passives
Complémentarité	Oui	Oui

(a)

	Framework	Design pattern
Réutilisation du code	Oui	Non
Réutilisation de la conception	Oui	Oui

(b)

TAB. 2.1 – Framework Vs Autres techniques de réutilisation
(a) Frameworks vs bibliothèques des classes
(b) Framework vs design patterns.

On rencontre dans plusieurs références [Fayad *et al.*, 1999, Schmidt and Huston, 2002, Mattsson, 1996], abordant le sujet des frameworks, des comparaisons entre ces derniers et d'autres techniques de réutilisation orientées objet qui sont principalement : bibliothèques de classes, composants et design patterns . Le tableau 2.1 donne un résumé de ces comparaisons.

Ces comparaisons n'ont pas pour objectif de favoriser une technique par rapport à une autre, mais plutôt de montrer et mettre en valeurs les caractéristiques et les avantages des frameworks.

2.3 Avantages et caractéristiques des frameworks

Les avantages élémentaires des frameworks proviennent de la modularité, de l'extensibilité et de la réutilisabilité et de l'inversion du contrôle [Fayad *et al.*, 1999] :

Modularité Les frameworks renforcent la modularité par encapsulation des détails d'implémentation derrière des interfaces stables. La modularité des frameworks aide à améliorer la qualité des logiciels en concentrant l'impact des changements de conception et d'implémentation.

Réutilisabilité Les interfaces stables fournies par les frameworks renforcent la réutilisabilité en définissant des composants génériques qui peuvent être réutilisés pour créer de nouvelles applications. La réutilisabilité des frameworks influence la connaissance du domaine et l'effort antérieur des développeurs expérimentés afin d'éviter la recréation et la revalidation des solutions. La réutilisation des composants du framework peut produire une amélioration réelle dans la productivité des programmeurs en plus du renforcement de la qualité, la performance et l'interopérabilité des logiciels.

Extensibilité Un framework renforce l'extensibilité en fournissant des *hook methods* qui permettent aux applications d'étendre leurs interfaces - stables. Les hook methods découplent systématiquement les interfaces stables et le comportement d'un domaine d'application des variations nécessaires pour l'instanciation d'une application dans un contexte particulier.

Inversion de contrôle L'inversion de contrôle, réalisée grâce à la liaison dynamique est l'une des plus importantes caractéristiques des frameworks. Comme le montre la figure 2.1 l'inversion de contrôle permet

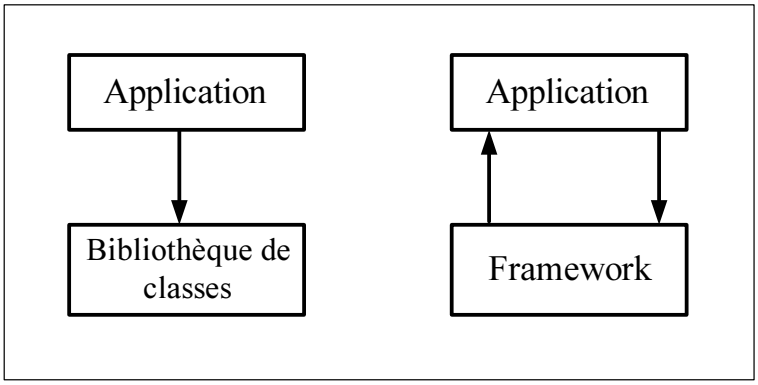

FIG. 2.1 – L'inversion de contrôle du framework

d'avoir un flux de contrôle dans les deux sens entre le framework et l'application par contre il est dans un seul sens dans le cas d'une application utilisant une bibliothèque de classes.

2.4 Classification des frameworks

2.4.1 Classification selon le domaine des frameworks

2.4.1.1 Framework d'infrastructure système

Ce type de framework simplifie le développement des infrastructures sys-tème portables et efficaces comme les systèmes d'exploitation [Campbell and Islam, 1993] et les frameworks de communication comme ACE [Schmidt, 1997]. Les frameworks d'infrastructure système sont généralement intégrés dans une organisation logiciel.

2.4.1.2 Framework d'intégration de middleware

Ces frameworks sont généralement utilisés pour l'intégration des composants et des applications distribuées . Ce type de framework est conçu pour

aider les développeurs des logiciels à étendre leurs infrastructures logicielles afin qu'il fonctionnent dans des environnements distribués. Des exemples de ce type de framework sont le framework ORB (*Object Request Broker*), le middleware orienté message comme USP (*Universal Service Processor*) de HP [Packard, 2005] et les bases de données transactionnelles, ce type de framework ne nécessite pas des langages orientés objet [Fayad *et al.*, 1999], comme exemple on peut citer Genesis [Batory, 1986].

2.4.1.3 Framework d'application d'entreprise

Ces frameworks s'adressent aux applications de domaine large comme les télécommunications, l'aviation, et les applications de finance. Relativement aux autres frameworks, les frameworks d'applications sont très coûteux à développer. Par contre, ils présentent un bon investissement car ils sont généralement développés pour l'utilisateur final et sont productifs directement après livraison. L'avantage le plus important de ces frameworks est que leurs coûts restent plus abordables que l'utilisation et l'adaptation d'un framework d'intégration de middleware et un framework d'infrastructure système. ET++SwapsManager [Eggenschwiler and Gamma, 1992] est un exemple de framework d'application d'entreprise utilisable dans le domaine de finance.

2.4.2 Classification selon la technique d'extension

2.4.2.1 Framework boite blanche

Les frameworks boite blanche *(whitebox framework)* se fondent fortement sur les caractéristiques des langages orientés objet comme l'héritage afin d'accomplir l'extensibilité. Les fonctionnalités du framework sont réutilisées et étendues par héritage des classes de base du framework et la redéfinition des méthodes virtuelles. Les frameworks boite blanche nécessitent du développeur d'applications une connaissance détaillée de la structure interne du

framework. Ces frameworks sont plus difficile à développer vu que le développeur du framework doit définir les interfaces et les *hook methods* qui anticipent une large gamme de cas d'utilisation. ACE [Schmidt and Huston, 2002] est un exemple de framework boite blanche.

2.4.2.2 Framework boite noire

L'extensibilité de ces frameworks s'effectue par la définition d'interfaces pour les composant qui peuvent être branchés (liés) au framework via la composition d'objet. Les fonctionnalités du framework sont réutilisées par la définition des composants qui sont conformes à des interfaces particulières et l'intégration de ces composants à travers leurs interfaces. Ces composants sont utilisés sans tenir compte de leur structure interne

Les frameworks boite noire (*blackbox framework*) sont structurés en utilisant la délégation et la composition d'objet plutôt que l'héritage.

Visual Basic eXtension (VBX) est un exemple réussi de framework boîte noire [Bichler *et al.*, 1998]. VBX [Cilwa *et al.*, 1994] est un composant logiciel réutilisable issu du langage de programmation Visual Basic de Microsoft (fonctionnant quasiment exclusivement avec un programme dans ce langage), contenant en général un contrôle et le code associé.

2.4.2.3 Framework boite grise

En réalité un framework n'est pas boite blanche pure ni boite noire pure [Pree, 1999] mais plutôt il forme une alliance entre les deux. Le terme « boite grise » a été utilisé pour résoudre le problème de classification lorsque l'héritage et la composition ont le même poids dans l'architecture du framework.

Les framework boite grise sont conçus pour éviter les désavantages des frameworks boite noire et des frameworks boite blanche. En effet un bon framework boite grise (*graybox framework*) a assez de flexibilité et extensibilité car il profite des avantages du framework boite noire et ceux du framework boite blanche en mixant la composition d'objet et l'héritage .

2.4.3 Classification selon la spécificité des framework

2.4.3.1 Frameworks Verticaux

Les frameworks verticaux sont organisés autour d'un domaine particulier tel que les systèmes financiers. Les frameworks verticaux sont en général plus complet pour ces types d'applications car la connaissance du domaine leur permet d'avoir un méta model plus riche.

2.4.3.2 Frameworks Horizontaux

Les framework horizontaux savent peu de leurs domaine d'applications. Ils fournissent une infrastructure de déploiement leurs permettant de fonctionner d'une manière commune (par exemple en fournissant une interface utilisateur commune) ou sur une plate forme commune.

2.5 Frameworks temps réel

Dans son livre [Douglass, 1999] Douglass donne une importance aux frameworks dans le développement des applications temps réel. Pour lui de 60% à 90% d'une application est du «code de ménage» (*housekeeping code*) qui peut être réutilisé s'il est bien structuré.

L'utilisation des frameworks appropriés peut considérablement faciliter la tache de développement d'une application car les services communs n'ont pas besoin d'être reimplémentés.

Le framework doit fournir un ensemble approprié de services pour le domaine d'application et doit également travailler dans l'environnement d'exécution du domaine-y compris les protocoles de transmission, les algorithmes de distribution, et les systèmes d'exploitation.

Un framework est une application partiellement réalisée, dont les éléments sont adaptés aux besoins du client par l'utilisateur pour accomplir l'application [Rogers, 1997]. Les frameworks fournissent un certain nombre d'avantages significatifs pour le développement rapide des applications robustes [Douglass, 1999] :

- Ils fournissent un ensemble de fonctionnalités permettant d'accomplir les tâches de programmation communes, ce qui réduit au minimum la complexité du système. Car il y a peu d'idiomes à apprendre afin de comprendre la structure et le comportement de l'application.
- Ils fournissent des moyens de réutilisation à grande échelle pour les applications.
- Ils fournissent une infrastructure architecturale commune pour les applications.
- Ils peuvent considérablement réduire le temps de développement pour des applications nouvelles.
- Ils peuvent être spécialisés pour des domaines de sorte qu'ils puissent capturer des idiomes et des modèles spécifiques au domaine.
- Ils imposent un style d'architecture.

Les frameworks forment également un ensemble de concepts dans lesquels un modèle d'application peut être exprimé. Dans ce sens, un framework est une langue pour exprimer un ensemble particulier de modèles. Les frameworks sont réalisés dans l'application exécutable par «sous-classage» (*subclassing*) des classes qui peuvent être considérées comme des points d'extension du framework.

2.6 Exemple de Frameworks temps réel

Dans cette section nous allons présenter un exemple de framework rencontré dans la littérature afin de voir sont application dans le domaine du temps réel.

2.6.1 VERTAF

VERTAF [Hsiung *et al.*, 2001] (Verifiable Embedded Real-Time Embedded Framework) est une intégration de trois technologies : la modélisation orientée objet, les composants logiciel et la vérification formelle. Ce framework est capable de produire efficacement des logiciels temps réel vérifiable, ce qui permet de réduire les erreurs et d'améliorer la conception.

Les auteurs de VERTAF ont profité de leurs travaux précédants sur les frameworks temps réel orientés objet comme SESAG [Hsiung *et al.*, 2005] et OORTSF [Kuan *et al.*, 1995] (Ces frameworks sont des simples frameworks appliqués dans le développement des logiciels d'aviation) pour présenter un framework amélioré et plus évolué.

VERTAF est un framework offrant un éditeur graphique où le développement des applications est fondé sur les modèles conçus par l'utilisateur. L'environnement graphique permet à l'utilisateur de créer facilement des modèles alors que le framework se charge de les concrétiser en faisant toutes les opérations nécessaires allant de l'instanciation et la spécialisation des classes, jusqu'à la vérification des modèles et la génération du code. Le développement est définit comme un processus de deux phases :

- phase de construction logicielle indépendante de la machine cible,
- phase d'implémentation logicielle dépendante de la machine cible.

La décomposition du processus de développement en deux phases permet de donner plus de flexibilité et d'augmenter la réutilisabilité du framework. Cette décomposition permet encore d'optimiser le code généré pour le matériel cible. Elle découple la conception et la modélisation indépendante du matériel, de l'implémentation très dépendante du matériel cible.

2.6.1.1 La conception et la vérification dans VERTAF

Dans VERTAF le développement est définit comme un processus de deux phases :

– Phase de construction logicielle indépendante de la machine (cible)
– Phase d'implémentation logicielle dépendante de la machine (cible)

La première phase elle-même est divisée en trois sous phases :
– phase de modélisation UML
– phase d'ordonnancement temps réel embarqué
– phase de vérification formelle

La deuxième phase elle aussi est divisée en deux sous phases :
– phase de « mapping » des composants
– phase de génération du code

2.6.1.2 Modélisation UML

Pour la modélisation de son application, l'utilisateur de VERTAF utilise trois des diagrammes de UML à savoir, diagramme des classes, diagramme des séquences, diagramme d'états transitions.

Le diagramme des classes peut contenir deux types de classe à savoir les classes ordinaires définit par l'utilisateur appelées aussi « classes logicielles » et les classes qui représentent les composants du matériel supportés par le framework appelées « classes materielles ». Excepté les relations ordinaires dans le diagramme de classe comme l'association, l'héritage, l'agrégation, une relation de déploiement est introduite entre les classes de l'application et les classes spécifiant un composant matériel. Par exemple une opération d'affichage est déployée sur un afficheur à cristaux liquides (LCD). La figure 2.2 présente un diagramme de classes avec déploiement d'un système de garde d'entrée. Les cadres pointillés représentent des classes matérielles alors que les cadres réguliers représentent des classes ordinaires. Les lignes pointillées représentent des déploiements, et les lignes solides représentent des associations.

Le diagramme d'états transitions (statecharts) utilisé pour spécifier le comportement d'un objet est amélioré par d'autres spécifications temporelles similaires à celles des automates temporisées. Des mots clés comme

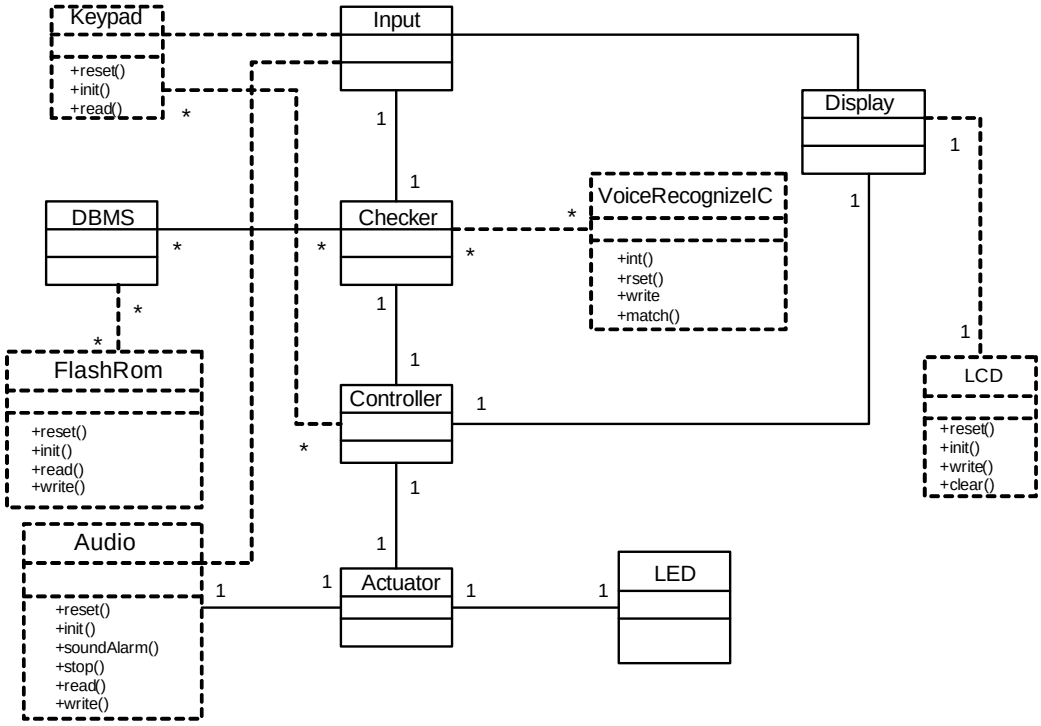

FIG. 2.2 – Diagramme de classes avec déploiement [Hsiung *et al.*, 2001]

'start', 'stop' et 'reset' associés aux méthodes déclenchées par le temps (time-triggered) sont aussi ajoutés.

Les diagrammes des séquences dans VERTAF sont utilisés principalement pour ordonnancer les tâches exécutées par les objets. Ils sont enrichis avec des structures de contrôle comprenant la concurrence, la composition et des marqueurs d'états qui sont le lien entre les diagrammes de séquences et les diagrammes d'états transitions dans VERTAF.

2.6.1.3 Ordonnancement temps réel embarqué

Pour l'ordonnancement, le framework utilise deux algorithmes différents basés sur une technique d'ordonnancement statique. L'un des deux algorithmes est utilisé pour les systèmes embarqués basés sur un RTOS, alors

que l'autre est utilisé pour les systèmes embarqués sans RTOS. L'ordonnancement des tâches dans le framework se base sur les réseaux de Petri traduits des models UML et principalement des diagrammes de séquences.

2.6.1.4 Vérification formelle

Le framework applique le «model checking» pour la vérification des modèles. Cette technique nécessite un model formel du système et une spécification formelle des propriétés temporelles. Afin d'assurer ces contraintes, les modèles UML sont traduit en automates temporisés alors que l'OCL (Object Constraint Language) en des formules TCTL. Chaque statechart du model spécifié par l'utilisateur est transformé en un ou plusieurs automates temporisés. Les automates du model générés à partir des statecharts sont fusionnés avec les automates générés par l'ordonnanceur dans la phase de l'ordonnancement. L'ensemble est vérifié par le noyau de vérification de VERTAF adapté du « State Graphe Manipulators ».

2.6.1.5 « Mapping » des composants

Cette phase consiste à configurer le matériel et le système d'exploitation à travers la génération des fichiers de configuration, les fichiers make, et les fichiers d'entête. Durant cette phase l'utilisateur est en mode interactif avec le framework qui lui permet de choisir les composants et les périphériques à partir d'une liste. D'après les auteurs une solution automatique n'est pas faisable à cause de la variété des caractéristiques physiques du matériel cible.

2.6.1.6 Génération du code

La génération du code est basé sur une approche de trois étages : couche d'abstraction matérielle, couche système d'exploitation, et middleware et moniteur temps réel. VERTAF supporte un ensemble de plateformes matériel.

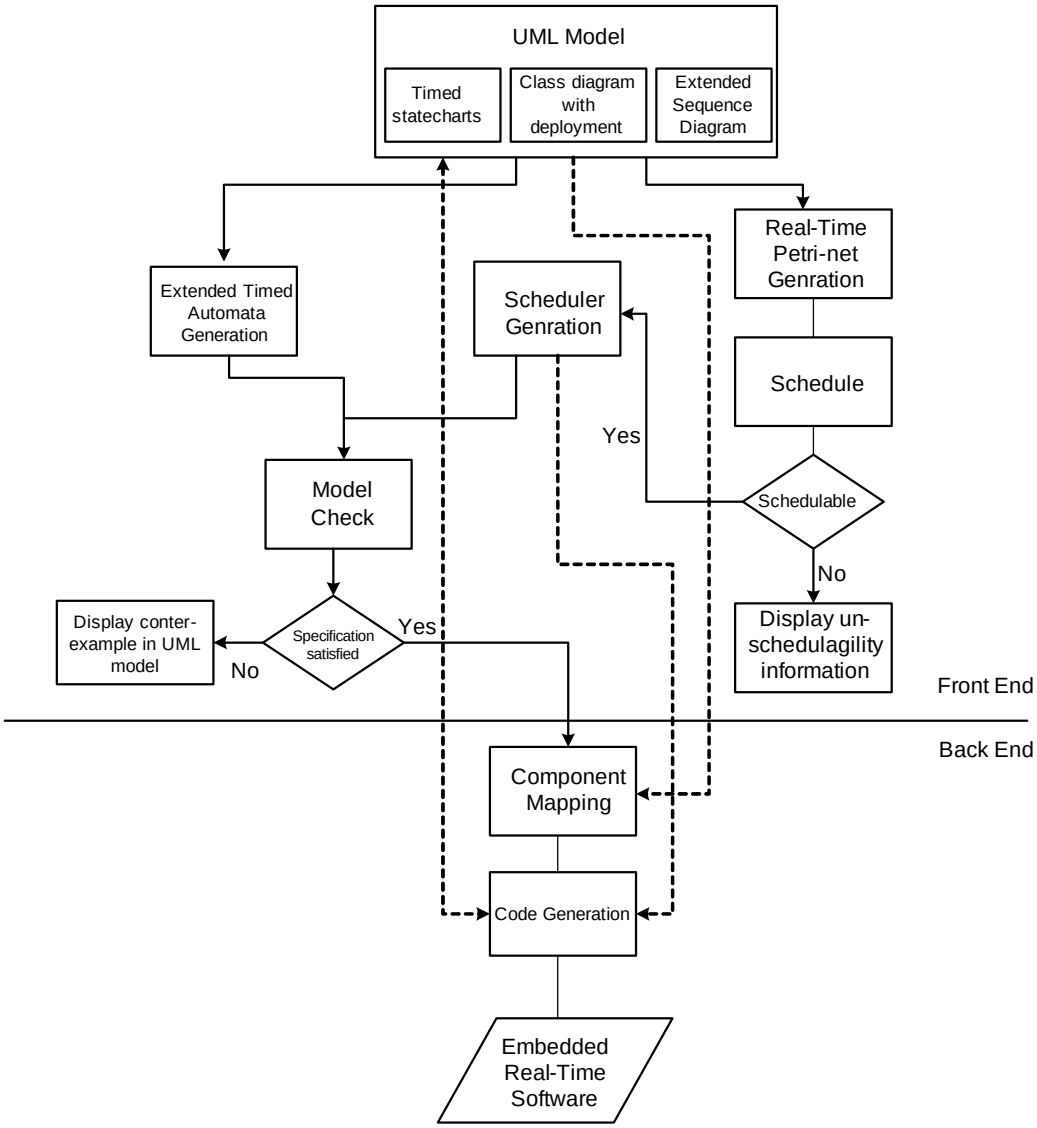

FIG. 2.3 – Le processus de conception et de vérification dans Vertaf [Hsiung *et al.*, 2001]

2.7 Exemple d'architecture des Frameworks temps réel

Dans cette section nous allons étudier deux frameworks temps réel. Ces deux frameworks sont différents dans leurs architectures, leurs domaines d'ap-

plication ainsi que la natures de leurs licenses d'utilisations (libre ou commerciale), la disponibilité du code source et la documentation.

2.7.1 Le framework ACE

ACE (**A**daptive **C**ommunication **E**nvironment) est un environnement adaptable de communication pour les applications distribuées [Schmidt and Huston, 2002]. ACE est un framework orienté objet dont l'architecture est basée sur un ensemble de design patterns traitant des problèmes de concurrence et de répartition. Cette architecture est destinée a développer des logiciels et des applications pour la communication entre systèmes.

ACE fournit un ensemble intégré de composants qui sont ciblés pour le développement d'applications temps réel de haute performance. Ces applications sont portables à une grande variété de plateforme de systèmes d'exploitation.

Afin de réduire la complexité, la conception du framework ACE est basée sur une architecture en couches développé en C++ . La Figure 2.4 illustre les différents composant du framework ainsi que les différentes couches de son architecture.

Les couches inférieures de ACE comprennent « OS adapter » ou couche d'adaptation aux systèmes d'exploitation et « C++ Wrapper » ou classes d'enrobage C++.

La couche « OS adapter » réside directement au dessus des API (**A**pplication **P**rogram **I**nterface) des systèmes d'exploitation qui sont généralement écrits en langage C. Cette couche permet de libérer les autres couches de la dépendance aux systèmes d'exploitation, elle contient des API pour le multitâche, la synchronisation et la communication inter-processus.

La couche « C++ Wrapper » fournit des interfaces C++ encapsulant les différents services des systèmes d'exploitation. Les applications peuvent

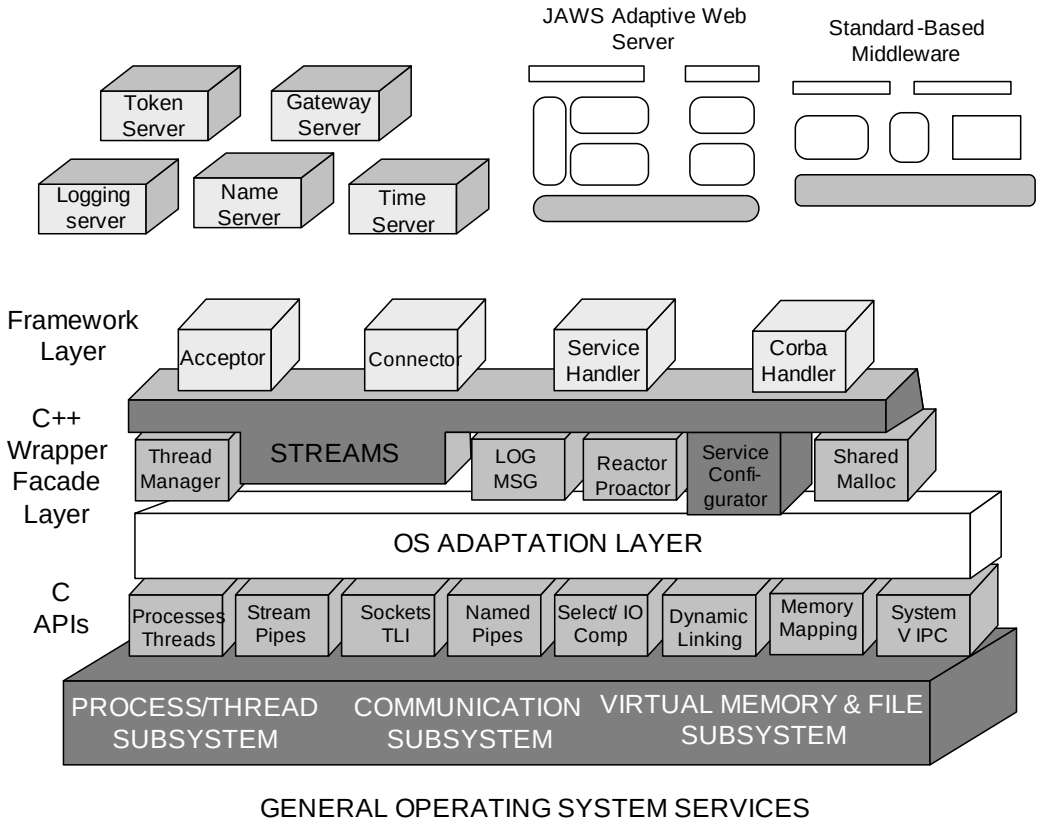

FIG. 2.4 – Architecture en couche du framework ACE

composer et combiner ces classes d'enrobage (*Wrapper*) par héritage, agrégation et instanciation. La couche « C++ Wrapper » fournit plusieurs des mêmes caractéristiques offerts par la couche « OS adapter ». Par contre, les caractéristiques et les services sont bien structurés en des objets et classes en C++ plutôt que des fonctions en C.

La couche « Framework » englobe une partie très importante du code du framework ACE. Elle est aussi le résultat des efforts les plus importants dans la conception. Cette couche est constituée de plusieurs composants (ou frameworks) dont la conception a été basée sur un ensemble de design patterns.

Ces principaux composants sont présentés dans les paragraphes suivants.

2.7.1.1 Reactor

Les applications en réseaux ont la caractéristique d'être dirigé par les événements. Le framework Reactor simplifie le développement de telles applications. Il offre des méthodes qui peuvent être spécifiées et personnalisées par les applications, alors qu'il se charge de les dispatcher pour le traitement. Le framework est aussi responsable de :

- La détection d'occurrences d'événements de différentes sources.
- Le démultiplexage des événements aux traiteurs d'événements.

La figure 2.5 illustre le diagramme des classes du framework Reactor. Le framework est constitué d'un ensemble de classes qui définissent des stratégies, de détection et de traitement des événements, indépendantes de l'application. Cet ensemble est appelé classes de couche infrastructure d'événement. Les classes ACE_Time_Value, ACE_Event_Handler, ACE_Timer_Queue et les différentes implémentation de ACE_Reactor sont les constituants de cette couche. Les traitements des événements dans les applications développées avec ACE, sont définis dans des classes descendantes de ACE_Event_Handler. Cette classe est la base de tous les traitements réactifs.

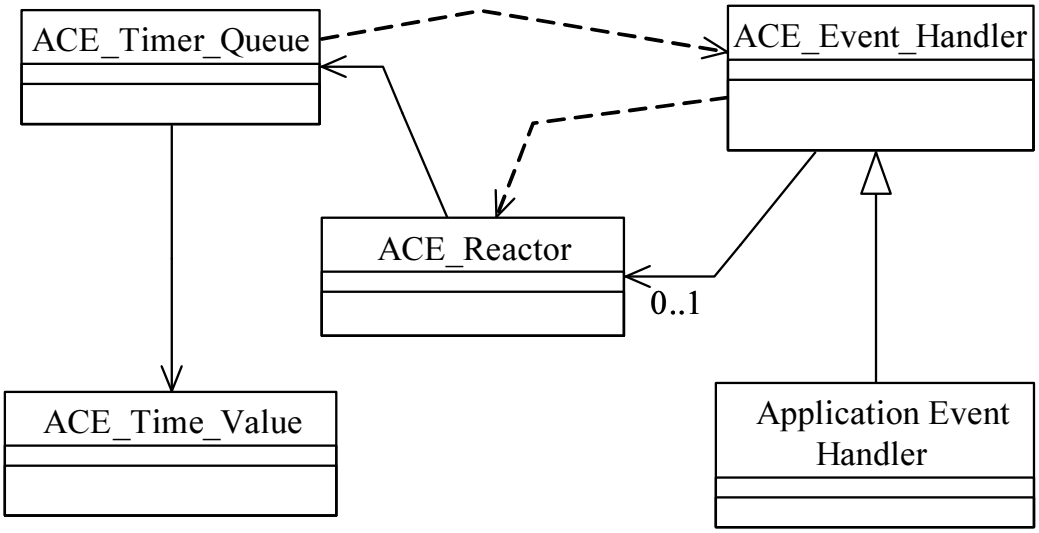

FIG. 2.5 – Diagramme des classes du framework Reactor

La classe ACE_Reactor est une application du design pattern Bridge [Gamma *et al.*, 1995]. La séparation de l'interface de son implémentation

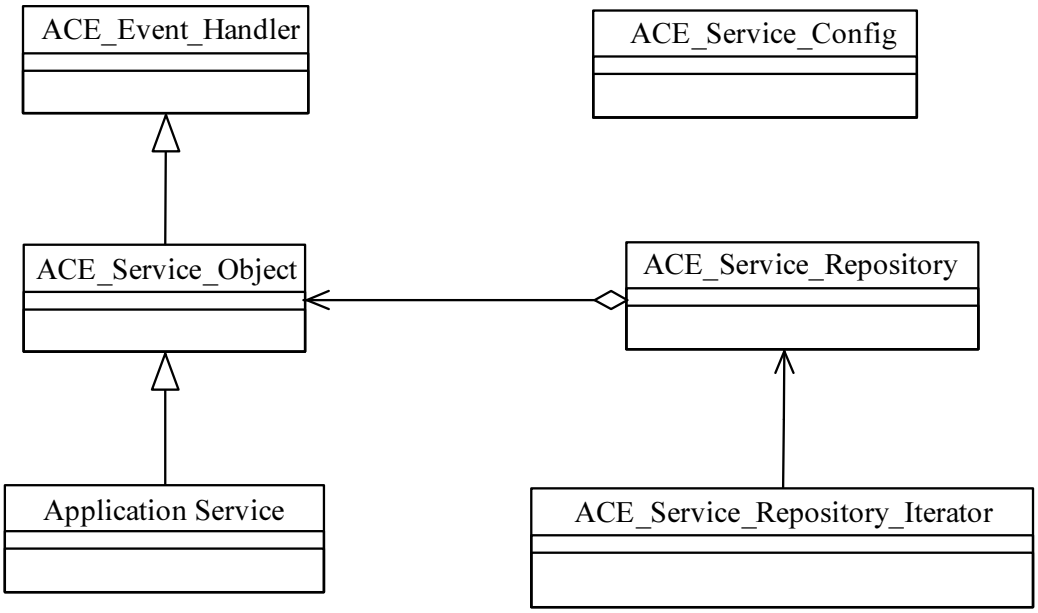

FIG. 2.6 – Les classes principales du framework Configurateur de service

donne beaucoup de flexibilité à l'utilisateur. Ceci permet de choisir une implémentation parmi plusieurs lors de l'exécution.

2.7.1.2 Configurateur de service

Ce framework est une implémentation du design pattern configurateur de composants [Jain and Schmidt, 1997]. Ce pattern aide à augmenter l'extensibilité et la flexibilité des applications. Il permet à l'application de reconfigurer ses services lors de l'exécution.

Le Configurateur de service dont le diagramme des classes est illustré dans la figure 2.6 est composé de deux ensembles principaux de classes :

Classes de la couche de gestion de la configuration : assurent des stratégies indépendantes de l'application pour initialiser et arrêter les objets de service. Cette couche comprend les classes ACE_Service_Config, ACE_Service_Repository, et ACE_Service_Repository_Iterator.

Classes de la couche application : implémentent des services concrets pour l'exécution des traitements de l'application. Ces classes sont descendantes de ACE_Service_Object qui hérite de ACE_Event_Handler.

La classe ACE_Service_Config implémente le design pattern Facade [Gamma *et al.*, 1995] pour intégrer d'autres classes dans le framework Configurateur de Services.

2.7.1.3 Task

Ce framework aide à mettre en valeur la modularité et l'extensibilité des applications orientées objet concurrentes. ACE Task forme la base commune des design patterns de concurrence dans le framework. Il fournit des capacités de concurrence orientée objet puissante et extensible. Les classes principales de ce framework sont illustrées dans la figure 2.7.

La classe ACE_Task fournit une abstraction orientée objet qui associe les processus legers (Threads) avec des objet C++. Elle implémente le design pattern Active Object [Lavender and Schmidt, 1995] qui permet à un objet d'exécuter ses méthodes dans son propre thread. Chaque invocation d'une de ses méthodes se transforme en un message stocké dans la queue de message de cette classe pour que celle ci l'exécute plus tard.

2.7.1.4 Acceptor Connector

Ce framework implémente le design pattern Acceptor-connector [Schmidt and Huston, 2002] qui sépare la connexion et l'initialisation, des services dans une application réseau, des traitements que ces services assurent après la connection et l'initialisation.

Les différentes relations entre les classes principales formant le framework sont illustrées dans la figure 2.8. Ces classes sont réparties sur trois couches :

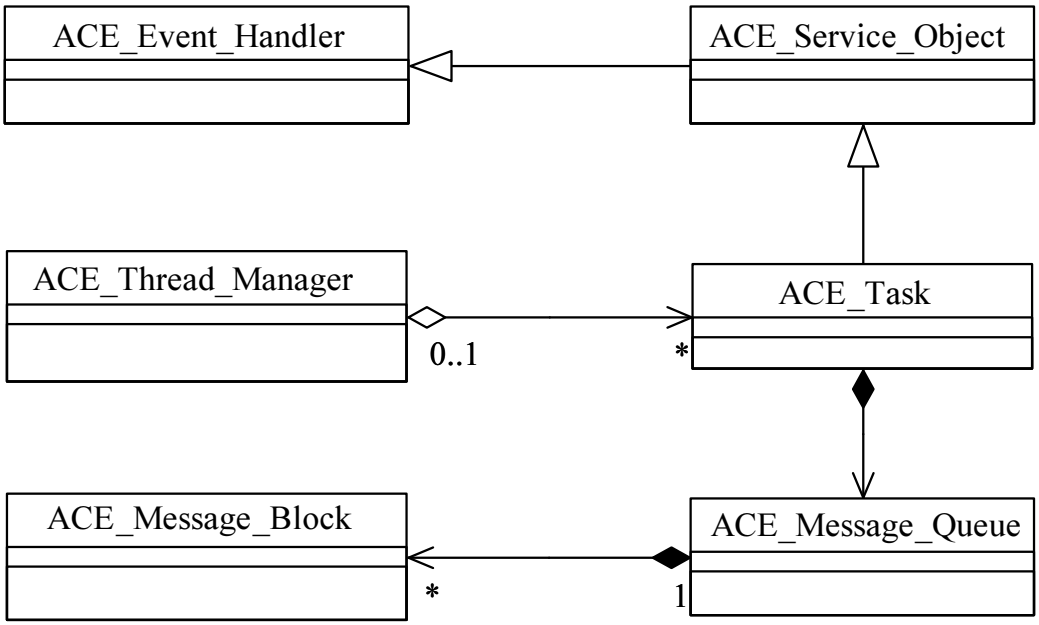

FIG. 2.7 – Les classes du framework ACE Task

Classe de la couche infrastructure d'événement assurent des stratégies génériques et indépendantes d'applications pour acheminer les événements. Le framework Reactor est généralement utilisé comme couche d'infrastructure d'événement.

Classes de la couche de gestion des connections assurent des services de connexion et d'initialisation indépendants de l'application. Ces classes comprennent ACE_Svc_Handler, ACE_Acceptor, et ACE_Connector.

Classes de la couche application personnalisent les stratégies génériques exécutées par les deux autres couches par l'intermédiaire de « sousclassage » et instanciation.

La classe ACE_Acceptor implémente le design pattern Template Method [Gamma *et al.*, 1995] ce qui lui permet de définir un algorithme d'exécution.

2.7.1.5 Proactor

Ce framework met en application le design pattern Proactor [Pyarali *et al.*, 1999] qui permet aux applications entraînées par les événements de

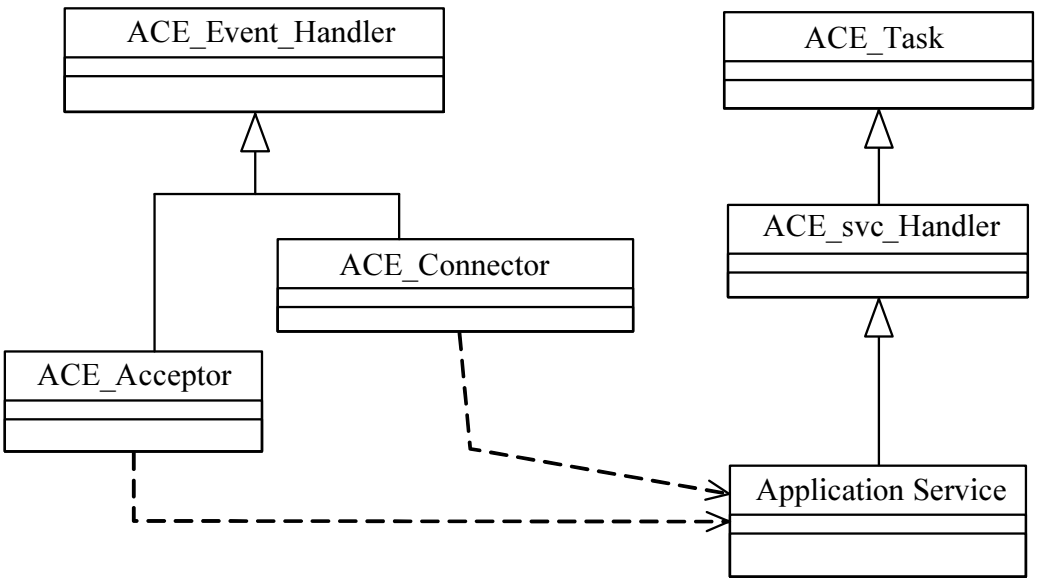

FIG. 2.8 – Les classes du framework Acceptor-Connector

démultiplexer et acheminer efficacement des demandes de service déclenchées par l'accomplissement des opérations asynchrones d'entrées/sorties.

Les relations les plus importantes entre les classes du framework Proactor sont montrées dans la figure 2.9.

2.7.1.6 Conclusion

En observant le cycle de développement du framework ACE on remarque que depuis le début de son développement (1991) ACE est en évolution continue[1]. Les développeurs de ce framework tirent des leçons de l'utilisation de ce dernier lors de développement des applications, et ils essayent d'améliorer les performances du framework en intégrant des design patterns existants, ou en généralisant des solutions à des problèmes rencontrés sous forme de design patterns qui seront intégrés dans les nouvelles versions du framework.

Ce processus de développement est dit ascendant car il part de l'architecture des applications existantes pour atteindre une architecture généralisée à un niveau d'abstraction plus élevé.

[1] A cette date la notion de design pattern orienté objet n'était pas encore évoquée.

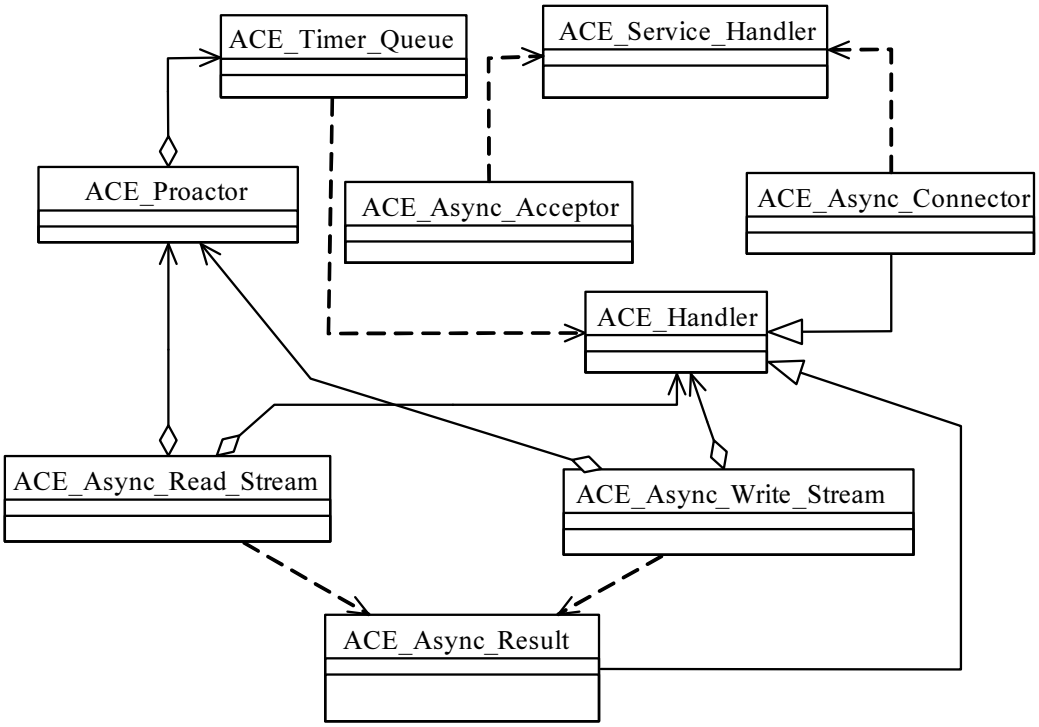

FIG. 2.9 – Les classes principales du framework Proactor

2.7.2 Rhapsody

Rhapsody [I-Logix, 2005] est un logiciel intégrant un éditeur graphique permettant de créer des diagrammes UML . Ce logiciel permet aussi de générer du code à partir des modèles UML grâce à un framework orienté objet appelé OXF .

Le framework contient un ensemble d'abstractions temps réel utiles permettant de structurer le code généré et de donner un sens concret aux concepts de UML. Les éléments du framework peuvent êtres personnalisés par héritage pour s'adapter aux besoins du développeur.

Le framework est indépendant du générateur de code . Les classes du framework peuvent être utilisées par l'utilisateur pour le développement des applications sans avoir recours au processus de génération de code. Dans ce cas, l'utilisateur est responsable des choix stratégiques des classes du framework correspondant à celle de son modèle.

L'architecture du framework OXF est basée sur des concepts illustrés dans les paragraphes suivants.

2.7.2.1 Classes Actives (Active Classes)

Une classe active est une classe qui possède un thread et peut initier une activité de contrôle. La signification concrète de la classe active dans Rhapsody est donnée par les éléments du framework (figure 2.10).

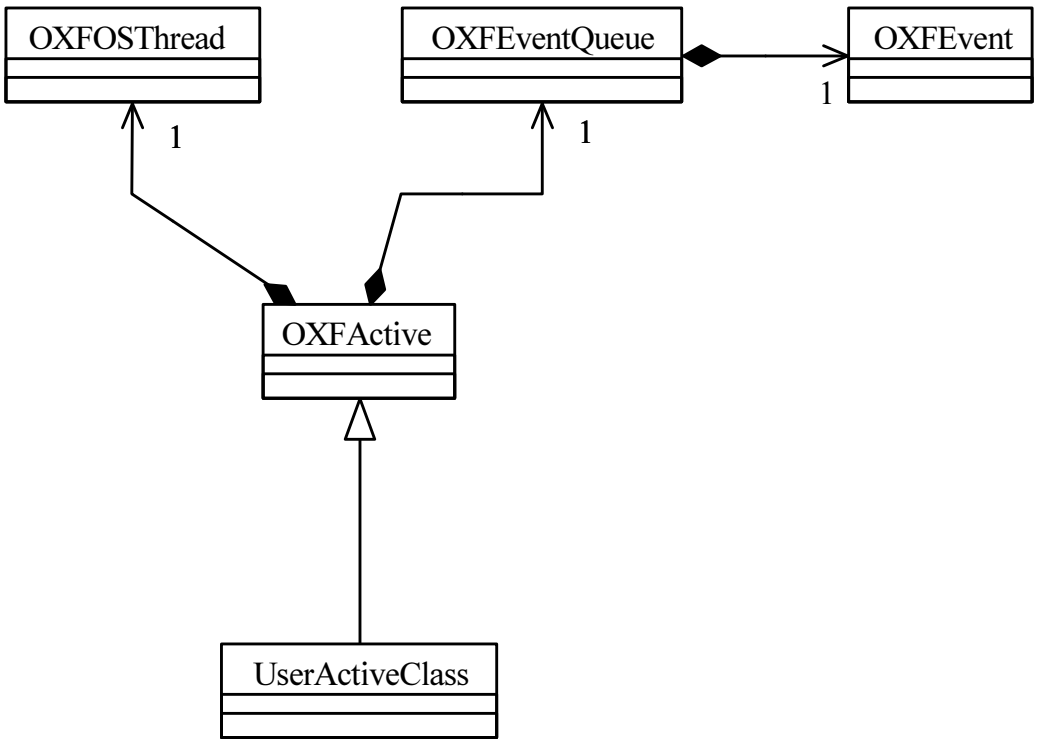

FIG. 2.10 – Eléments du framework relié à la classe active

Une classe active est une classe possédant un thread et des fonctionnalités d'acheminement des évènements. Un thread dans le framework est une instance de la classe OXFOSThrread qui est un enrobage (wrapper) du thread (ou tâche) du système d'exploitation. Comme le montre la figure 2.10 une classe active est une instance de la classe OXFActive. Cette classe gère une file (queue) d'événement.

2.7.2.2 Classes Réactives (Reactive Classes)

Une classe réactive est une classe qui réagit aux événements. Elle est consommatrice d'événements. Toutes les classes réactives héritent de la classe OXFReactive et chaque classe réactive du framework est associée à un gestionnaire d'événements. La figure 2.11 illustre les classes du framework et les classes éventuelles que l'utilisateur peut définir.

Le traitement des événements est normalement définit par des « *statecharts* » dans le modèle UML (on peut associer à chaque classe du modèle une ou plusieurs statecharts pour définir son comportement vis-à-vis des événements).

On distingue trois types de classes réactives :

Classe réactive active : une classe de ce type, gère elle-même ses événements dans son propre thread.

Classe réactive passive : une classe de ce type est gérée par le gestionnaire par défaut du framework. Toutes les classes réactives de ce type partagent ce même unique gestionnaire par défaut.

Classe réactive subordonnée : une classe de ce type est gérée par une classe réactive définie par l'utilisateur.

2.7.2.3 Evénements et Opérations

Chaque classe peut avoir des événements et des opérations. Les événements correspondent aux occurrences instantanées prédéfinies qui affectent le comportement de la classe. Les opérations correspondent aux services ou fonctionnalités fournis par la classe.

Dans Rhapsody les classes actives ne traitent que leurs activités d'acheminement d'événement, par contre elles ne traitent pas l'invocation de leurs opérations ; si par exemple un client invoque une opération d'une instance

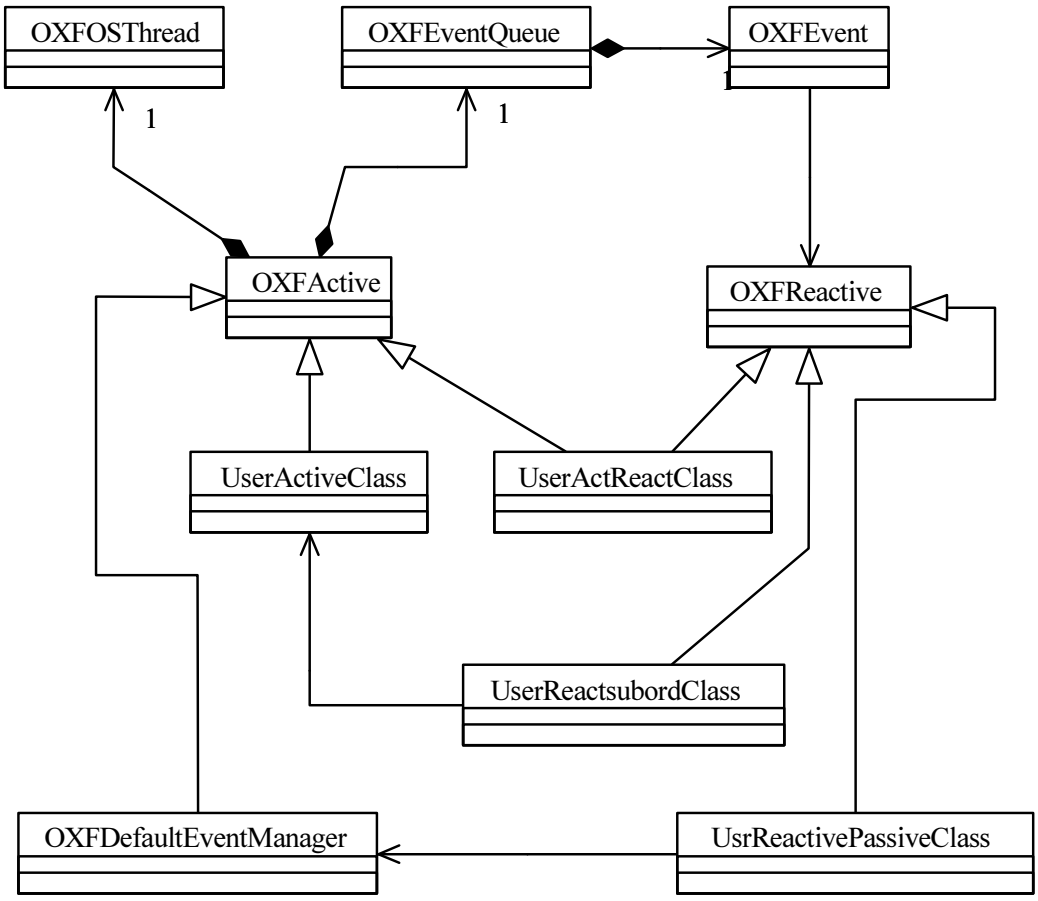

FIG. 2.11 – Eléments du framework Actifs et Réactifs

d'une classe active l'exécution de l'opération invoquée se déroulera dans le thread de l'appelant qui est dans ce cas le client.

Dans Rhapsody il y plusieurs types d'événements :

événement signal : les signaux dans le framework sont appelés simplement événements. Ils sont représentés par la classe OXFEvent de laquelle héritent toutes les instances d'événements. Chaque événement est associé à son objet (réactif) cible. Les événements signaux sont synchrones, c'est pourquoi ils n'ont pas une valeur de retour.

événement temporel : ces événements sont appelés dans le framework « *timeout* ». Un timeout est représenté dans le framework par la classe

OXFTimeout. Les instances de cette classe sont associés à un gestionnaire (instance de la classe OXFTimeoutManager) qui s'exécute dans son propre thread et possède un chronomètre (ou timer instance de OXFTimer) .

2.7.2.4 Contrôle de concurrence et protection des ressources

Le framework OXF comprend des abstractions des mécanismes de contrôle de concurrence :

- OXFOSMutex est une classe d'enrobage des sémaphores d'exclusion mutuelle fournit par le système d'exploitation.
- OXFOSSemaphore est une classe d'enrobage des sémaphores binaires ou des sémaphores à compteur fournit par le système d'exploitation.

Les méthodes dans une classe peuvent être protégées en les déclarant avec le mot clé « *guarded* ». Une classe qui possède des méthodes protégées est dérivée de la classe OXFProtected. Cette classe utilise les sémaphores pour contrôler les méthodes protégées.

2.7.2.5 Synthèse

Le but de Rhapsody est de générer du code à partir des diagrammes UML tout en prenant en considérations les concepts temps réel. Le rôle de OXF, le framework sur lequel repose Rhapsody, est de fournir une infrastructure logicielle lui permettant d'accomplir sa mission. De ce fait, la conception de OXF n'est pas basée sur des abstractions d'applications existantes, mais plutôt des abstractions des concepts temps réel. Ce processus et de développement des framework est dit descendant, car sa conception part des concepts généraux du domaine d'application du framework.

2.8 Conclusion

Dans ce chapitre nous avons introduit les frameworks, leurs avantages, leurs caractéristiques ainsi que leurs catégories. Nous avons aussi présenté quelques exemples de frameworks appliqués au domaine de développement des applications temps réel et embarquées.

Les frameworks qu'on a présentés dans ce chapitre comprennent des frameworks commerciaux (Rhapsody de i-Logix) ainsi que des frameworks issues d'instituts de recherche (ACE, VERTAF).

OXF de Rhapsody et VERTAF sont conçus pour donner une sémantique aux diagrammes UML afin de générer du code à partir d'un modèle. L'architecture de ces frameworks se limite aux concepts nécessaires pour la conception et le développement des applications temps réel et embarqué avec UML.

ACE Offre une architecture logicielle très riche intégrant un nombre important de design patterns. Le développement avec ACE nécessite une connaissance de l'architecture de ce dernier ainsi que de son domaine d'application.

En comparant les frameworks OXF de Rhapsody et ACE on peut dire que l'avantage de Rhapsody c'est qu'il s'oriente vers le développement des applications temps réel embarquées en général, alors que ACE est destiné au développement des applications temps réel distribuées et s'occupe principalement des problèmes de la distribution. D'autre part, l'architecture de Rhapsody se focalise sur la génération du code à partir des modèles UML et elle se limite à donner une sémantique aux différents diagrammes UML supportés par l'éditeur graphique de Rhapsody. Par contre, ACE offre une architecture riche en design patterns et elle est destinée à faciliter le développement d'applications distribuées indépendamment du langage de modélisation.

Nous pensons qu'il est intéressant de fournir un framework qui est d'un coté, comme OXF de Rhapsody , orienté vers le développement des applications temps réel en général et qui est d'un autre coté, comme ACE, riche

en design pattern et indépendant du langage de modélisation. Un tel framework peut réduire considérablement le coût et le temps de développement des applications temps réel grâce à la réutilisation du code du framework, comme il peut aussi augmenter la qualité de ces applications en profitant d'une architecture riche en design patterns.

Chapitre 3

Conception d'un Framework Horizontal pour les Systèmes Temps Réel Embarqués

3.1 Introduction

« *Le développement d'une application orientée objet est une tâche difficile et le développement d'un framework l'est encore plus* » [Mattsson, 1996]. C'est ce qu'on rencontre dans la littérature quand on parle du développement des frameworks. La même chose est dite pour le développement des applications temps réel : *« C'est plus difficile que le développement d'autres applications »* [McKegney and Shepard, 2000]. La difficulté parvient essentiellement des ressources limitées et des contraintes temporelles strictes auxquelles sont affrontés les développeurs d'applications temps réel embarquées. Alors que dire du développement d'un framework temps réel ?

Le développement d'un framework passe par les étapes ordinaires par lesquelles passe le développement d'une application. Mais si le processus de développement de l'application s'arrête une fois que cette application est testée et délivrée, le processus de développement d'un framework commence un autre cycle d'abstraction et de généralisation des différentes techniques et modèles architecturaux sur lesquelles se base l'application, afin qu'ils puissent êtres utilisés pour la génération d'autres applications, car un framework est conçu pour le développement d'une famille d'applications dans un domaine particulier.

Les applications temps réel embarquées sont largement utilisées et dans des différents domaines comme l'aviation, la téléphonie mobile, la robotique, etc. En fait un système temps réel est un système en interaction continue avec son environnement à travers des capteurs et des actionneurs qui sont les moyens de communication du système. Les capteurs et les actionneurs ainsi que les données manipulées par le système varient d'un environnement à un autre selon l'objectif du système et son milieu. Les contraintes temporelles aussi dépendent de la nature des données et de l'environnement.

3.2 Développement des systèmes temps réel embarqués

Le multitâches est l'une des caractéristiques les plus importantes des systèmes temps réel . «*Il fournit à l'application le mécanisme fondamental pour contrôler et réagir aux multiples événements discrets du monde réel* » [WindRiver, 1999]. Afin de traiter les événements externes, les systèmes temps réel sont multitâches. Les événements sont généralement apériodique le multitâches permet de découpler le traitement des événements au moins en deux phases [Geeter, 1999] :

- La première phase consiste à stocker toute l'information concernant l'événement.
- La deuxième phase consiste à récupérer ces informations et exécuter les travaux nécessaires suite à cet événement.

Ce concept (multitâches) permet aux systèmes temps réel de mieux contrôler les événements extérieurs et de partager le traitement entre les tâches.

Le matériel cible des systèmes temps réel embarqués diffère d'une application à une autre selon le contexte et les objectifs de cette dernière. Cette variété du matériel, rend difficile la tâche de conception et développement de tels systèmes. D'autre part les systèmes d'exploitation temps réel (RTOS) facilitent la tâche de développement en offrant des services et des routines d'interfaçage au matériel. Ces services sont appelés BSP (Board Support Package) et ils sont développés par l'éditeur du RTOS ce qui lui permet d'être portable sur une famille de materiel cible de différentes architectures.

La figure 3.1 présente l'architecture générale pour le développement des applications temps réel. Le RTOS avec le BSP présentent une couche logicielle facilitant l'interaction avec le matériel.

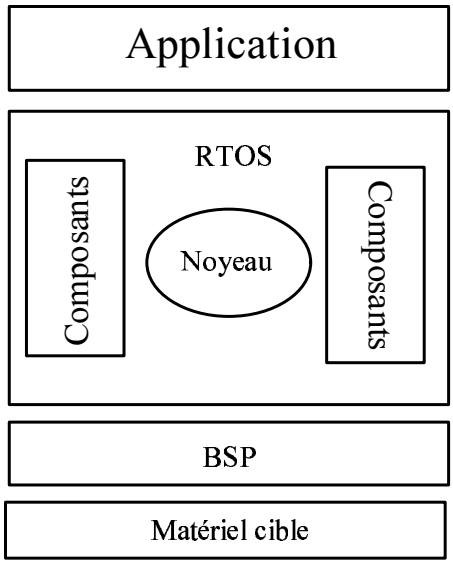

FIG. 3.1 – Architecture générale d'une application temps réel

3.3 Un framework horizontal pour les systèmes temps réel embarqués

Le coût de développement, la taille du système ainsi que sa complexité varient énormément selon le domaine d'application et le matériel cible. Les applications temps réel peuvent être des simples applications de contrôle comme elles peuvent être des applications larges et distribuées. En conséquence, l'architecture matérielle varie d'une application à une autre selon les besoins. Généralement la différence principale entre les différentes architectures concerne les points suivants : le nombre des microprocesseurs, leurs architectures (CISC, RISC, ...), leurs puissances, la nature de la mémoire (partagé ou distribué), son type et sa taille. A cause de la variété des domaines d'application des systèmes temps réel embarqué, le développement commence pratiquement à partir de zéro (*from scratch*) dans la majorité des cas.

Malgré leurs diversités, les systèmes temps réel ont plusieurs points en commun. Ils sont en interaction continue avec leurs environnements à travers

des capteurs et des actionneurs, leurs ressource sont généralement limitées avec des architectures logicielles qui se basent essentiellement sur des tâches concurrentes. Les frameworks offrent une architecture réutilisable regroupant des infrastructures logicielles concernant une famille d'applications.

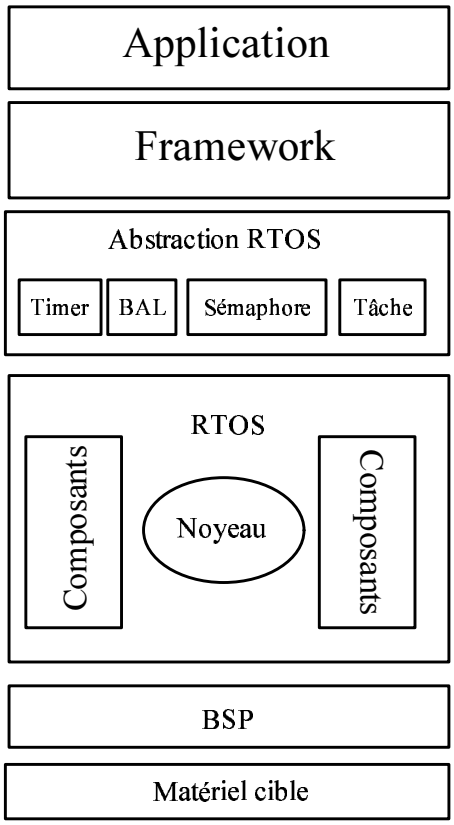

FIG. 3.2 – Architecture d'une application temps réel développée avec un framework

Un framework vertical est basé sur la spécificité du domaine et rassemble les expériences acquises durant le développement des applications dans ce domaine. Comparé à un framework vertical, un framework horizontal couvre un domaine plus large et un nombre d'applications plus important, de ce fait il est à un plus haut niveau d'abstraction.

La conception et le développement d'un framework horizontal qui englobe les abstractions des caractéristiques générales des systèmes temps réel peut

faciliter le développement de ces derniers, comme il peut être le noyau d'un framework vertical spécifique à un domaine particulier.

Au contraire du développement des frameworks verticaux où il existe des techniques de conception et de développement [Johnson, 1992, Mattsson, 1996], on a pas rencontré dans la littérature des approches similaires pour la conception et le développement des frameworks horizontaux. Les techniques de développement de ces deux types de frameworks sont totalement différentes. Le développement des frameworks verticaux nécessite une approche ascendante ; on part d'un ensemble d'applications dans un domaine particulier et on applique des abstractions pour atteindre une architecture logicielle spécifique à ce domaine et qui constitue le cœur d'un framework vertical. Par contre, le développement des frameworks horizontaux nécessite une approche descendante ; on part d'un ensemble de caractéristiques et de concepts généraux d'un domaine d'application et on essaye de les intégrer dans une architecture logicielle qui forme la base d'un framework horizontal.

A court terme, les frameworks verticaux sont plus rentables[1] car ils contiennent des concepts très spécifiques au domaine, mais à long terme les frameworks horizontaux sont plus rentables car ils sont plus génériques.

Dans [Christensen and Rn, 2000] le framework horizontal s'adresse aux problèmes de recherche des chaînes de caractères. Le framework est un éditeur graphique pour l'édition et la construction des expressions de recherche dans un domaine de donnés arbitraire (une base de données ou des message électronique par exemple). Il met en œuvre un ensemble de design patterns pour résoudre les problèmes de modélisation des données, la representation d'une expression, la recherche d'une expression, l'interface graphique du framework et les éditeurs d'attributs.

On peut dire qu'un framework horizontal peut être conçu à base d'un ensemble de design patterns traitant des problèmes récurants dans un domaine

[1] Un framework devient rentable au bout du développement de N applications, si le code nécessaire pour le développement de ces N applications est supérieur au code nécessaire pour leurs développement en utilisant le framework.

particulier, ce qui peut être le cas dans la conception d'un framework temps réel.

Le nombre grandissant des design patterns et les différents problèmes auxquels ils s'adressent, pose des difficultés dans leurs sélections pour la conception et le développement d'un framework horizontal. Comment choisir les design patterns ? Quels sont les design patterns qui peuvent intégrés dans le framework ?

Pour résoudre ce problème nous proposons un processus de conception d'un framework horizontal pour les systèmes temps réel afin de faciliter le choix et la selection des design patterns.

3.4 Processus de conception

On peut concevoir un framework horizontal pour les systèmes temps réel à partir d'un ensemble de design patterns. C'est ce qu'on peut dire en se basant sur les rares expériences concernant le développement des frameworks horizontaux. Mais la question qui se pose c'est *comment ?* En réalité nous n'avons pas rencontré dans la littérature une méthode, un processus ou une approche pour la conception d'un framework horizontal.

Notre but est de développer un framework horizontal modulaire qui accroît la qualité des applications en intégrant des design patterns. La subdivision du framework en des sous-frameworks facilite son développement, sa maintenance, ainsi que son utilisation.

Pour la décomposition du framework, nous avons appliqué le principe de séparation des préoccupations (*separation of concerns*). Ce principe nous permet de décomposer le framework en des sous-frameworks indépendants (Figure 3.3).

L'étape suivante consiste à sélectionner des design patterns pour chaque sous-framework en observant les design patterns existantes dans la littérature ou en « *détectant* » des nouveaux design patterns.

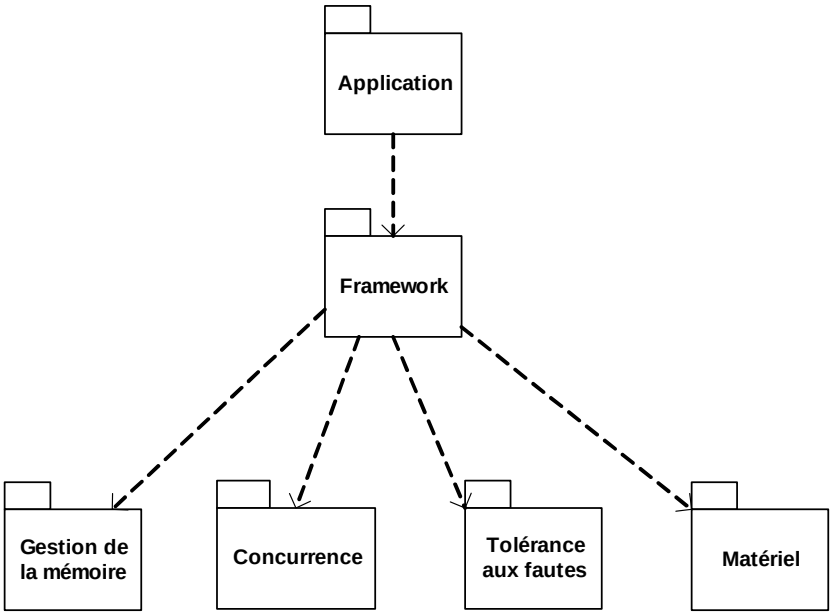

FIG. 3.3 – Application du principe de séparation des préoccupation

La troisième étape consiste à l'intégration des design patterns. La phase d'intégration comprend l'adaptation des design pattern, l'identification des classes communes afin de préserver la cohérence entre tous les composant du framework.

Dans notre cas le framework traite deux préoccupation : la concurrence et la tolérance aux fautes.

3.5 Selection des design patterns

La sélection des design patterns est, dans notre cas, l'une des tâches critiques de la conception du framework. Ceci est dû à la diversité des design patterns temps réel et à l'absence d'une classification exhaustive de ces design patterns. Dans la majorité des cas, les design patterns temps réel sont identifiés dans des contextes particulier et ne sont pas suffisamment généralisés pour être appliqués dans une large famille d'application.

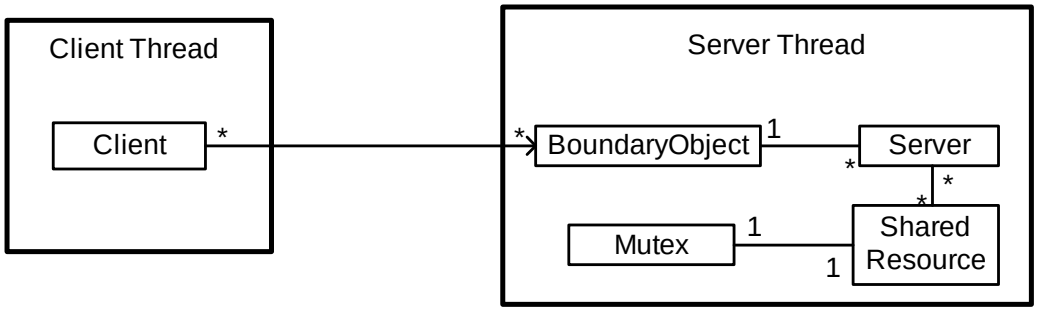

FIG. 3.4 – Guarded Call Pattern [Douglass, 2002]

Dans notre cas, la sélection des design patterns a été influencée par deux facteurs déterminants. Le premier facteur concerne la « cohabitation » du design patterns avec le RTOS qui est dans notre cas vxworks . Par exemple, on a rencontré un nombre de design patterns qui s'adressent à l'ordonnancement des tâches mais qui ne « cohabite » pas avec notre RTOS car ce dernier possède son propre ordonnanceur. Par contre ces design patterns peuvent être utiles lorsque le framework sera porté sur d'autres plateformes. Il est de même pour les design patterns de gestion de la mémoire qui peuvent être très utiles lorsque le framework sera implémenté en un langage temps réel sans RTOS . Le deuxième facteur concerne la structure et la spécificité du design pattern. Dans certains cas, le design pattern ne présente pas des classes (ou des méthodes) réutilisables à chaque application du design pattern. Dans ce cas, le design pattern présente une solution abstraite qui ne concerne que la structuration des relations entre des classes indépendamment de leurs méthodes. Comme exemple, on peut citer le design pattern de concurrence *Guarded Call Pattern* [Douglass, 2002] dont le modèle est illustré dans la figure 3.4.

3.6 Sous framework de concurrence

Les systèmes temps réel sont naturellement concurrents. Ils sont généralement conçus pour fonctionner durant de longues périodes sans s'arrêter. Ils peuvent fonctionner durant des jours, des mois, ou même des années (comme

dans le cas des satellites) sans être arrêtés ou redémarrés. Ils sont en interaction continue avec leurs environnements. Ils peuvent être vues comme un ensemble de tâches collaboratrices, chaque tâche se charge de traiter une fonctionnalité ou plus parmi celles que le système doit assurer, et collabore avec les autres en communiquant et en partageant des ressources et des données. La notion de tâche comme portion de code constitué d'instructions séquentielles impose un parallélisme (ou pseudo parallélisme) dans l'accès au microprocesseur et (ou) à la mémoire dans le cas où le système est constitué de plus qu'une tâche. Le nombre des tâches qui est, dans la majorité des cas, supérieur au nombre des microprocesseurs exige un ordonnancement d'exécution des tâches. Les communications entre les tâches se font à travers des messages ou des variables partagées, dans ce cas, des outils et des mécanismes de synchronisation sont obligatoires afin d'assurer la cohérence des données.

3.6.1 Le design pattern Message Queuing

Ce design pattern (Figure 3.5) fournit un outil simple pour la communication des données entre les tâches. Bien que la communication est une approche « *lourde*[2] »[Douglass, 2002] pour partager l'information, elle est préférée parce qu'elle est supportée par la majorité des systèmes d'exploitation et parce qu'elle est la plus facile à tester.

Le Message Queuing pattern utilise la communication asynchrone pour la communication et la synchronisation des tâches. En plus de la simplicité, cette approche permet de décharger les tâches synchronisées de la gestion de l'exclusion mutuelle pour accéder à la ressource. L'information à partager entre les tâches est envoyée par valeur ce qui permet d'éviter les problèmes de corruption reliés au partages des données passées par référence.

[2]La communication dans ce cas nécessite deux opérations (envoi et reception) et une copie de l'information partagée.

3.6.1.1 Structure

La structure de ce pattern est illustrée dans la figure 3.5. Chaque tâche possède une queue de message dans laquelle elle enregistre[3] les messages asynchrone qui lui sont envoyés. La tâche peut lire ces messages au moment de son activation. Chaque queue de message est protégée par un sémaphore d'exclusion mutuelle. Vu qu'elle est accessible par la tâche qui envoie et celle qui reçoit, elle doit être protégée contre les accès multiples.

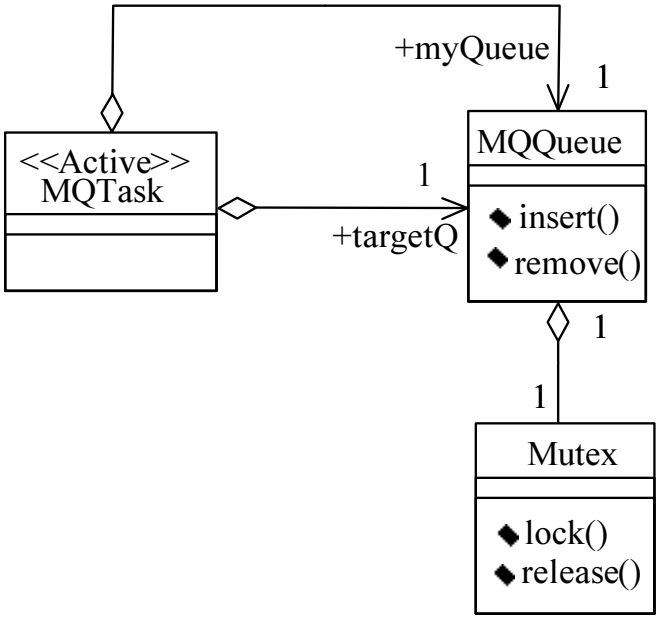

FIG. 3.5 – Message Queuing Design Pattern

3.6.1.2 Collaboration

Task

La tâche peut créer des messages pour les envoyer vers d'autres tâches comme elle peut récupérer des messages de sa queue pour les traiter.

[3]En réalité, l'opération de l'enregistrement des messages est faite par la tâche envoyante car l'exécution des méthodes de la queue de message se déroule dans le thread de contrôle de leur appelant.

Queue

La queue de message est conteneur de messages. Ses opérations s'exécutent dans le thread de contrôle de la tâche appelante. Elle fournit au moins deux méthodes insert() et remove() qui verrouillent le sémaphore d'exclusion mutuelle avant d'exécuter leurs corps et le déverrouillent après exécution.

Mutex

Le sémaphore d'exclusion mutuelle permet de protéger la queue de message des accès multiples à son contenu. Si une première tâche essaye de faire une opération d'insertion alors qu'une deuxième est entrain de faire une opération sur la queue de message alors la première sera bloquée jusqu'a ce que la deuxième termine.

3.6.1.3 Intégration au framework

La figure 3.6 présente la structure du Message Queuing design pattern dans le framework. Nous avons choisi de distinguer les noms des classes spécifiques au design patterns dans le framework par deux lettres majuscules désignant le nom du design pattern. On essaye de garder au maximum possible la même structure du design pattern dans le framework afin de faciliter son utilisation et l'exploiter dans la documentation du framework.

AbTask est une classe active abstraite pure. Elle sert comme une interface pour tous les objets actifs dans le framework.

AbQueue est une classe abstraite servant comme interface pour les types abstrait de donnée comme liste, queue, ...

MQTask est une classes active abstraite qui hérite de la classe abstraite pure AbTask. Elle possède une queue de message de type MQqueue.

MQQueue est une implémentation de la classe AQueue et utilise une queue de message de type AMQueue ainsi qu'un sémaphore de type AMutex.

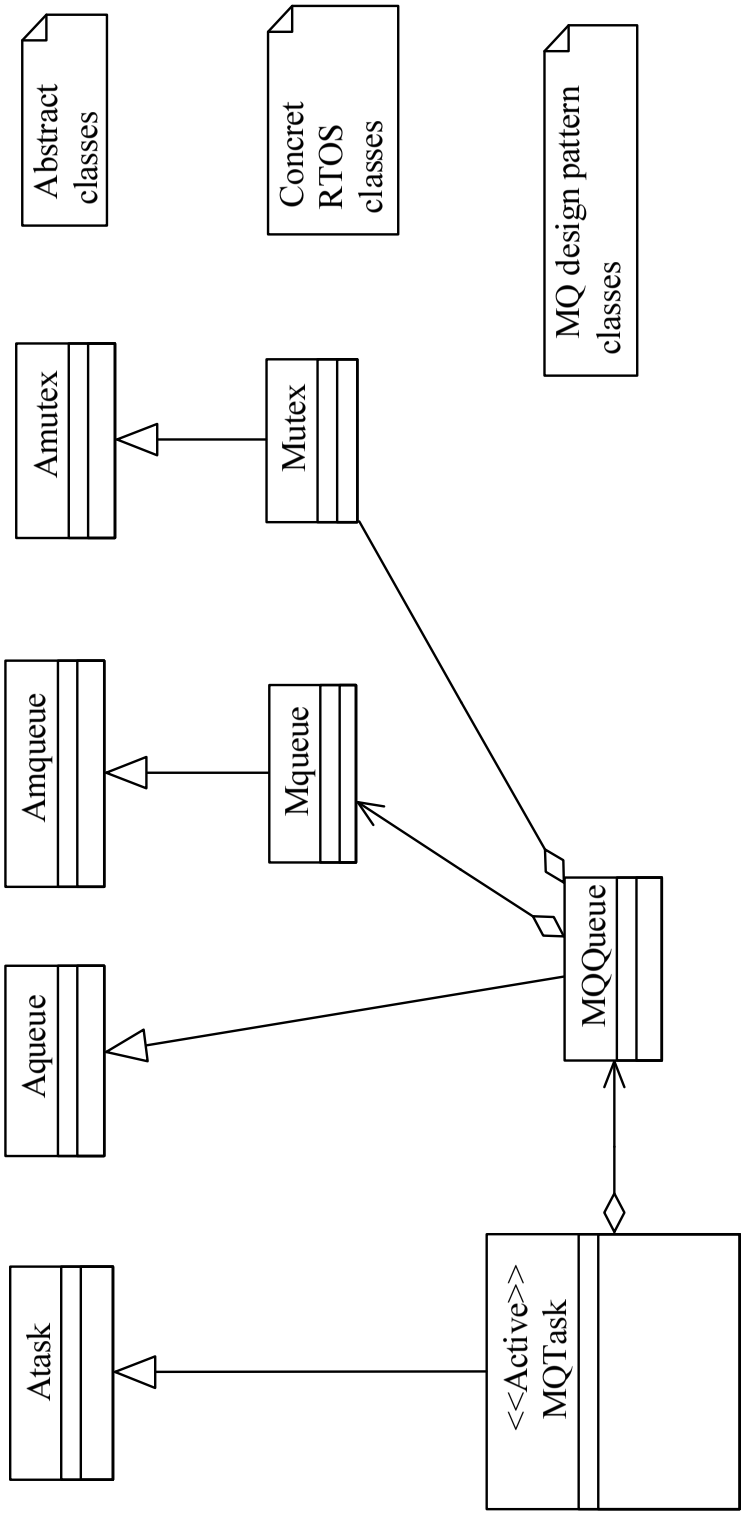

FIG. 3.6 – Integration du Message Queuing pattern dans le framework

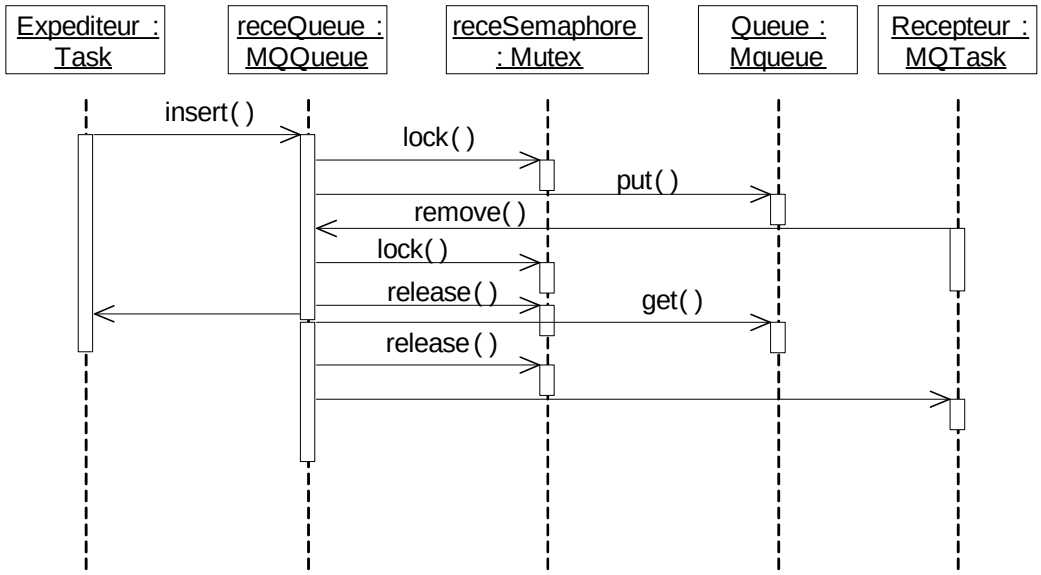

FIG. 3.7 – Diagramme de séquence illustrant un cas d'utilisation du Message Queuing Pattern.

La relation de MQQueue avec la classe abstraite AMQueue permet de l'associer avec toute classe de ce type (i.e dérivant de AMQueue). En d'autre termes, MQueue peut être une instance de l'une des classes dérivantes de la classe abstraite AMQueue (non illustrées dans la figure pour des raisons de clarté).

La figure 3.7 présente un diagramme de séquence illustrant un cas d'utilisation du Message Queuing design pattern. La tâche Expéditeur est entrain de poster un message lorsque la tâche Récepteur essaye de lire un message. Cette dernière se bloque jusqu'à la fin de l'opération de postage. Il faut noter que seulement les deux méthode insert() et remove() sont visibles aux deux tâches.

3.6.2 Le design pattern Rendezvous

Le Rendezvous pattern (Figure 3.8) est un modèle simple qui permet la synchronisation d'un ensemble de tâches. Il permet aussi de partager des

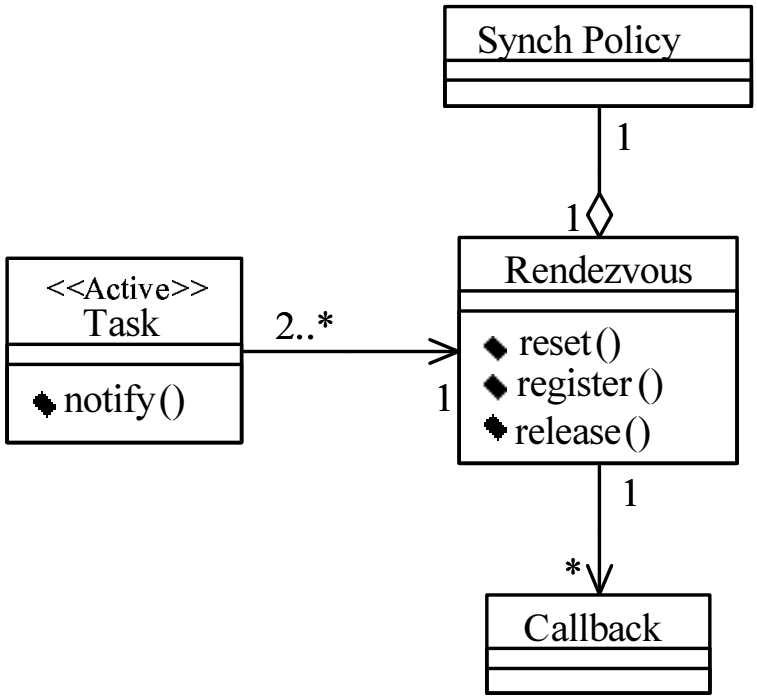

FIG. 3.8 – Rendezvous design pattern

données après synchronisation. L'avantage de ce design pattern est qu'il permet de fixer des politiques de synchronization ou des préconditions avant d'acceder à une ressource.

3.6.2.1 Structure

Le principe de ce pattern est que chaque tâche prête pour la synchronisation vient s'inscrire dans l'objet Rendezvous (figure 3.8). L'inscription entraîne un blocage de la tâche jusqu'a une notification éventuelle lui permettant de reprendre l'exécution. Cette approche présente une grande flexibilité dans la modélisation des préconditions complexes. La structure du Rendezvous design pattern est illustré dans la figure 3.8.

3.6.2.2 Collaboration

Callback
L'objet Callback possède les adresses des tâches clientes. Il permet à l'objet Rendezvous de notifier (i.e libérer) tous les tâches inscrites, une fois que la précondition est réalisée.

Task
Ce design pattern est utile pour la synchronisation d'au moins deux tâches. Lorsqu'une tâche atteint son point de synchronisation elle s'inscrit dans l'objet Rendezvous en laissant son adresse.

Rendezvous
L'objet Rendezvous gère la synchronisation entre les tâches. Normalement il possède une opération (méthode) d'inscription dans son interface. Les tâches invoquent cette méthode pour signaler qu'elles sont prêtes pour la synchronisation.

Synch Policy
L'objet Sync Policy concrétise un ensemble de préconditions dans un concept unique. Une version simple de Sync Policy consiste à inclure un compteur des tâches inscrites. Lorsque le compteur atteint une valeur bien déterminé, il le signale à l'objet Rendezvous qui libère toutes les tâches.

3.6.2.3 Intégration au framework

Ce design pattern peut être utilisé sous deux approches selon la notifications des tâches :

- Notification collective.
- Notification sélectionnée.

Notification collective : Dans ce cas les tâches inscrites attendent la réalisation de la précondition pour être notifiées et libérées toutes. La politique de synchronisation dans ce cas est simple, car elle s'occupe d'une précondition concernant toutes les tâches.

Notification selective : Dans ce cas les tâches inscrites sont notifiées individuellement. L'objet Rendezvous sélectionne les tâches à notifier suivant la politique de synchronisation. Cette dernière s'avère plus compliquée dans ce cas, car elle doit tenir compte de plusieurs contraintes et préconditions.

De point de vue implémentation il y a une différence entre les deux variétés de ce design pattern. Dans le cas de la notification collective, l'objet Rendezvous peut ne pas avoir une liste (Callback) des adresses de toutes les tâches inscrites. L'inscription dans ce cas peut être effectuée à travers un sémaphore initialement vide que l'objet Rendezvous possède et que les tâches voulant s'inscrire prennent. Ceci entraîne le blocage de toutes les tâches qui viennent de s'inscrire. Lorsque l'objet Rendezvous exécute la méthode releaseAll() il libère toutes les tâches bloquées sur ce sémaphore.

L'intégration de ce design pattern dans le framework nécessite la prise en compte de ses deux variétés. La figure 3.9 présente la structure de ce design pattern dans le framework en tenant compte de ses variétés.

AbRVclass est la classe abstraite de laquelle dérivent les classes concrétisant les deux variétés de ce design pattern.

AbRVsyncPolicy est la classe abstraite de laquelle dérive toute classe concrétisant une politique de synchronisation.

RVsyncPolicy est la classe concrète qui définit une politique de synchronisation bien déterminée.

RVclassCollect est la classe concrétisant la notification collective. Elle hérite les méthodes de AbRVclass ainsi que l'agrégation d'une politique de synchronisation et possède un sémaphore. Les tâches s'inscrivant dans ce type de Rendezvous dérivent de la classe Atask.

RVclassInd est comme RVclassCollect sauf qu'elle possède une liste (RVcallback) au lieu du sémaphore. Par contre, les tâches s'inscrivant dans ce type de Rendezvous ont leurs propres sémaphores. Elles sont des instances de RVtask.

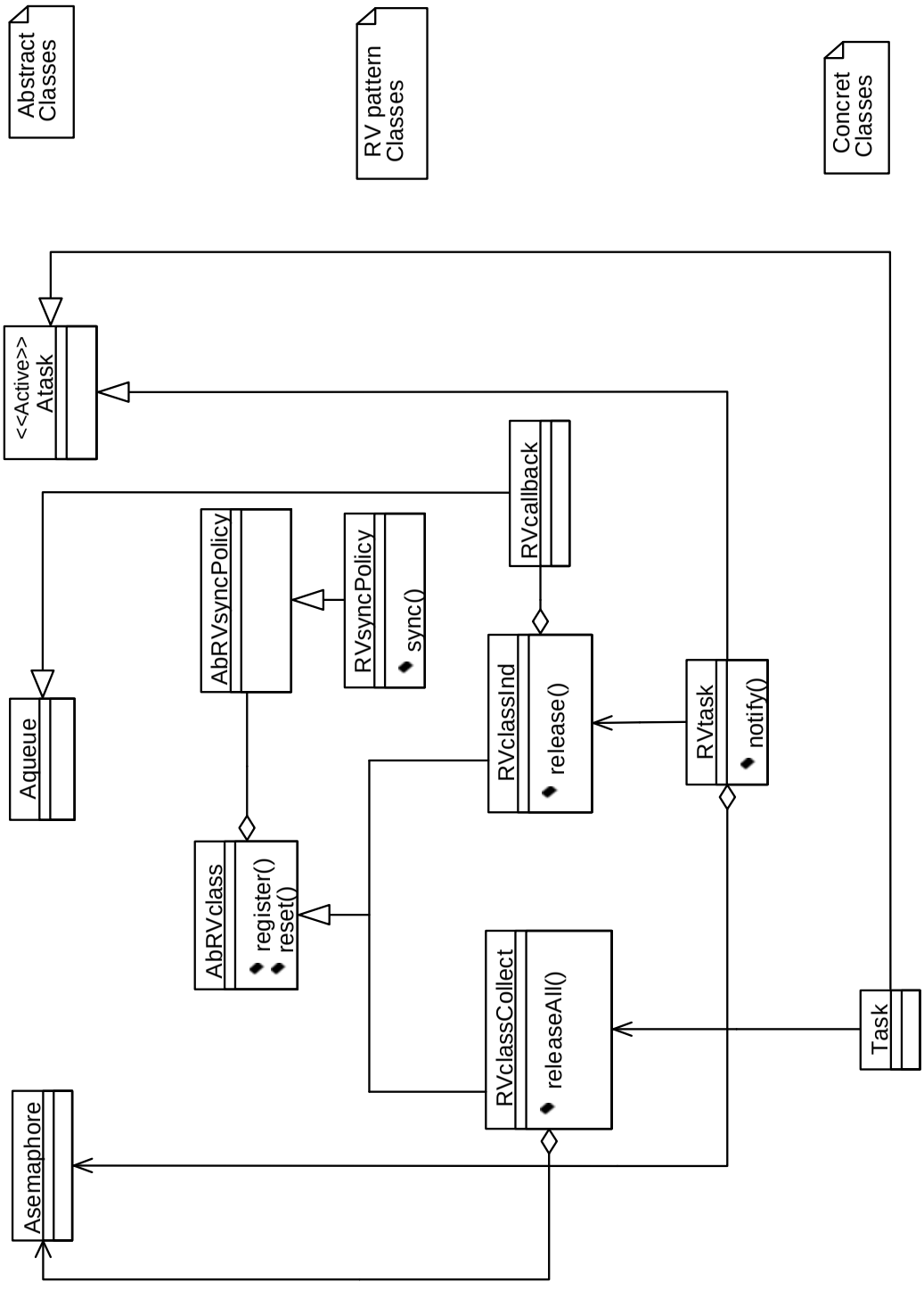

FIG. 3.9 – Rendezvous Pattern intégré dans le framework

RVcallback est une sorte de liste (elle dérive de Aqueue) qui sert à enregistrer des références sur les tâches inscrites dans un Rendezvous.

RVtask est une tâche qui possède son propre sémaphore utilisé par l'objet RVclassInd pour la bloquer.

Il faut noter que l'instance de RVclassInd est un objet passif et que son code s'exécute dans le thread de contrôle de la tâche appelante.

La figure 3.10 présente un diagramme de séquence illustrant un cas d'utilisation du Rendezvous Design Pattern. Les trois tâche : Tâche1, Tâche2 et Tâche3 s'inscrivent respectivement au Rendez-vous en appelant la méthode register() de l'objet Rendezvous. L'inscription entraîne un blocage de la tâche jusqu'à la réalisation de la condition. À chaque opération d'inscription l'objet Rendezvous vérifie la condition de synchronisation en exécutant la méthode sync() de l'objet PolSync. Une fois la condition est réalisée (dans ce cas nombre de tâches inscrites égale à trois), l'objet PolSync appelle la méthode releaseAll() de l'objet Rendezvous qui libère toutes les tâches.

3.6.3 Le design pattern Monitor

Le moniteur est un mécanisme de synchronisation plus évolué que les sémaphores et les régions critiques conditionnelles. En plus de la protection d'une ou plusieurs sections critiques, il permet à ses clients d'être mis en attente d'une condition, et d'accéder à la ressource qu'il protège, une fois que cette condition devient vraie. Comme cela a été conçu par Dijkstra [Dijkstra, 1967, Hoare, 1974], un moniteur est un module regroupant un ensemble de procédures et de variables, où chaque procédure représente une région critique et manipule des données (variables) encapsulées dans le moniteur. Les appels aux procédures sont séquentielles, le moniteur n'autorise qu'un seul appel à la fois, c'est à dire qu'à un instant donné, on ne peut avoir plus qu'une procédure en cours d'exécution. Du point de vue logiciel, un moniteur est une construction du langage de programmation qui garantit un accès approprié à des sections critiques. Le code permettant le contrôle d'accès à la section

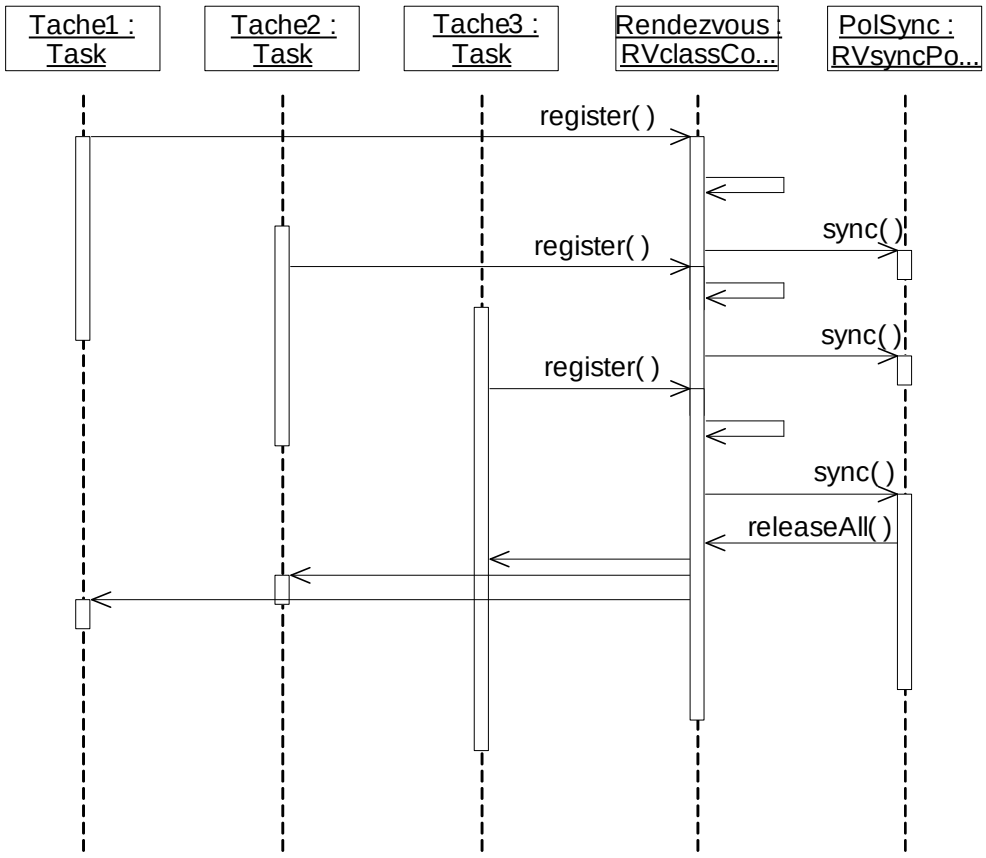

FIG. 3.10 – Diagramme de séquence illustrant un cas d'utilisation du Rendezvous Design Pattern

critique est généré par le compilateur. Tel qu'il est, on retrouve le moniteur dans différents langages de programmation temps réel comme Modula-1 , Concurrent Pascal et Mesa .

3.6.3.1 Structure

La structure de base du Monitor Pattern est illustrée dans la figure 3.11. Son principe est de protéger les données des accès multiple tout en permettant l'attente d'une condition. Il est composé d'un objet Moniteur encapsulant des données et présentant une interface facilitant l'accès à ces données, un verrou (*lock*) protégeant l'accès au Moniteur, et des conditions permettant de maintenir les clients dans une file d'attente tout en libérant le Moniteur.

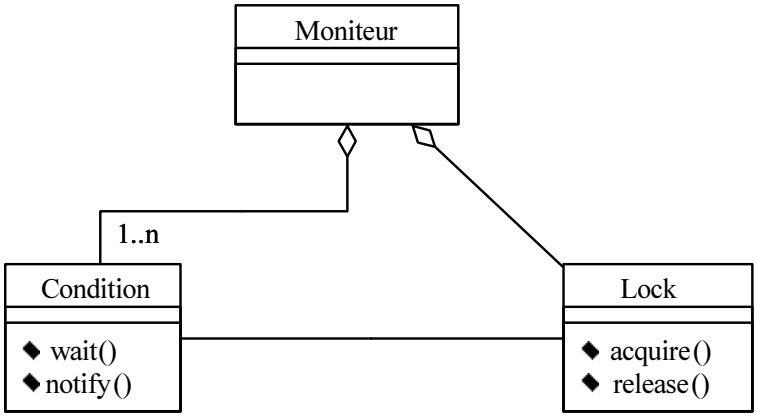

FIG. 3.11 – Monitor object pattern

3.6.3.2 Collaboration

Monitor

Un moniteur présente une ou plusieurs méthodes à ses clients. Les services offerts par le moniteur ne sont accessibles qu'à travers ces méthodes, ce qui permet de protéger l'état interne du moniteur contre la « *race condition*[4] » ou les accès non contrôlés. L'exécution de chaque méthode se fait dans le thread de contrôle du client car le moniteur n'a pas son propre thread. Par ce fait, le moniteur est un objet passif.

Lock

Chaque moniteur possède son propre verrou (*lock*). Ce dernier est utilisé par les méthodes synchronisées pour ordonner leurs accès séquentiels aux variables du moniteur. Chaque méthode doit acquérir le verrou en accédant aux ressources du moniteur et libérer le verrou en fin de traitement. Un sémaphore est généralement utilisé pour implémenter le verrou.

Condition

Les conditions permettent aux méthodes synchronisées de coopérer afin d'ordonnancer leurs exécutions. Chaque méthode peut attendre une condition ou notifier une autre méthode si une condition vient de se réaliser.

[4]La *race condition* est une condition dans laquelle un résultat dépend de l'ordre d'exécution, mais cet ordre ne peut pas être prévu [Douglass, 2002]

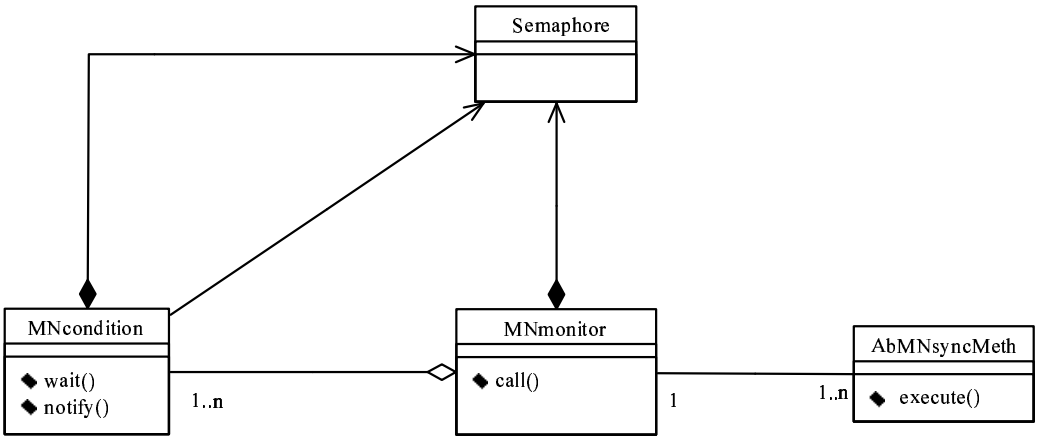

FIG. 3.12 – Classes du Monitor pattern dans le framework

3.6.3.3 Intégration au framework

L'intégration du Monitor Pattern dans le framework ne pose aucun problème si le moniteur possède une seule méthode ou un nombre de méthodes connu à l'avance. Mais en général l'utilisateur choisi les méthodes et les conditions du moniteur selon le contexte de son problème, et le framework doit lui permettre ce choix. Le moniteur dans ce cas doit avoir une structure plus flexible. Afin d'assurer cette flexibilité, les méthodes et les conditions sont spécifiées par l'utilisateur. Pour assurer ceci, on associe à chaque méthode un objet. L'objet est une instance d'une classe qui dérive de AbMNsyncMeth et qui implémente la méthode execute(). Les paramètres utilisés par la méthode sont aussi intégrés dans l'objet avec des méthodes spécifiques. L'appelle à une méthode du moniteur se fait à travers la méthode call(). Les conditions prennent, après leurs création une référence sur le sémaphore de moniteur afin de le libérer lors d'une opération wait() sur une condition.

La figure 3.12 présente la structure du Monitor Pattern aprés intégration au framework.

MNmonitor constitue l'interface permettant l'accès aux différents service du moniteur et ceci à travers la méthode call() qui selon les paramètre appelle une méthode particulière.

MNcondition permet de libérer le moniteur et de maintenir ses clients dans une file d'attente. Lorsque la condition n'est pas vérifiée la méthode wait() entraîne un blocage de la tâche appelante. Une fois la condition réalisée la méthode notify() libère les tâches bloquées.

AbMNsyncMeyh est la classe abstraite de laquelle dérivent toutes les classes qui concrétisent les méthodes du moniteur. L'utilisateur spécifie la méthode abstraite execute()qui sera appelée par le moniteur. L'utilisateur peut ajouter d'autres méthodes à cette classe mais elles ne seront pas utilisées par le framework.

La figure 3.13 présente un diagramme de séquence illustrant un cas d'utilisation du Moniteur Pattern. L'exemple illustre un moniteur (Moniteur) avec une condition (uneCond) et deux méthodes (Meth1 et Meth2). La première tâche (Client1) demande l'exécution de la méthode Meth1. La condition nécessaire pour l'exécution de cette méthode n'est pas réalisée. La tâche Client1 se bloque en attendant la réalisation de la condition (opération wait()). La tâche Client2 demande l'exécution de la méthode Meth2 qui réalise la condition attendue par Client1 (opération notify()).Une fois libérée, la tâche Client1 termine son exécution normalement.

Il faut noter que l'exécution du traitement spécifique à chaque méthode nécessite la prise du sémaphore mntSem (opération acquire()) et le libère une fois l'exécution se termine (opération release()).

La figure 3.14 constitue un exemple d'utilisation du framework pour l'implémentation d'un producteur/consommateur avec un moniteur. Le buffer dans cet exemple possède deux méthodes : Put et Get et deux conditions nonVide et nonPlein.

3.6.4 Le design pattern Active Object

En UML un objet actif est une instance d'une classe active. Cette dernière est définie comme « classe dont les instances possèdent un thread de contrôle », ou encore « classe dont les instances peuvent initier une activité

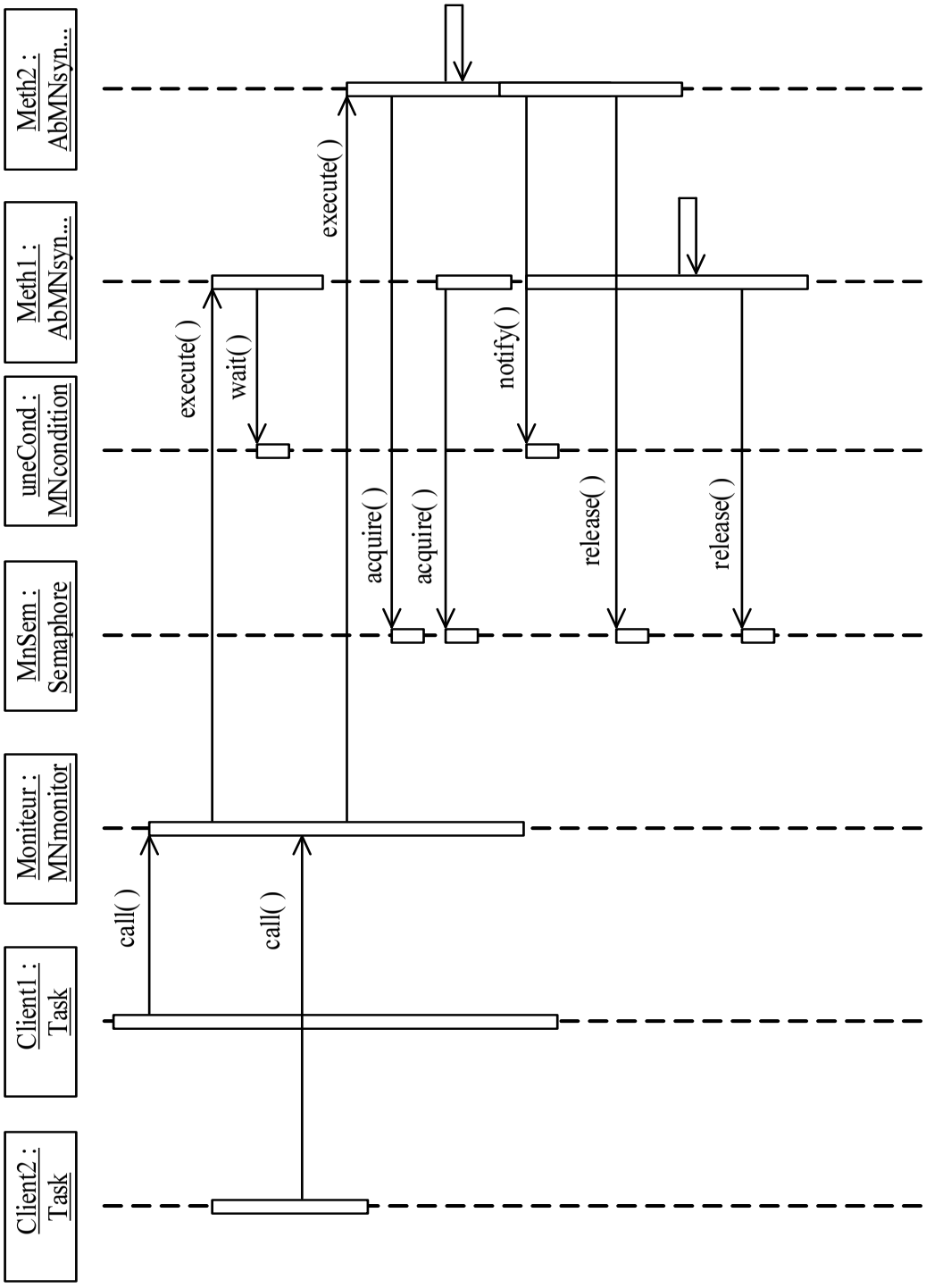

FIG. 3.13 – Diagramme de séquence illustrant un cas d'utilisation du Moniteur Pattern

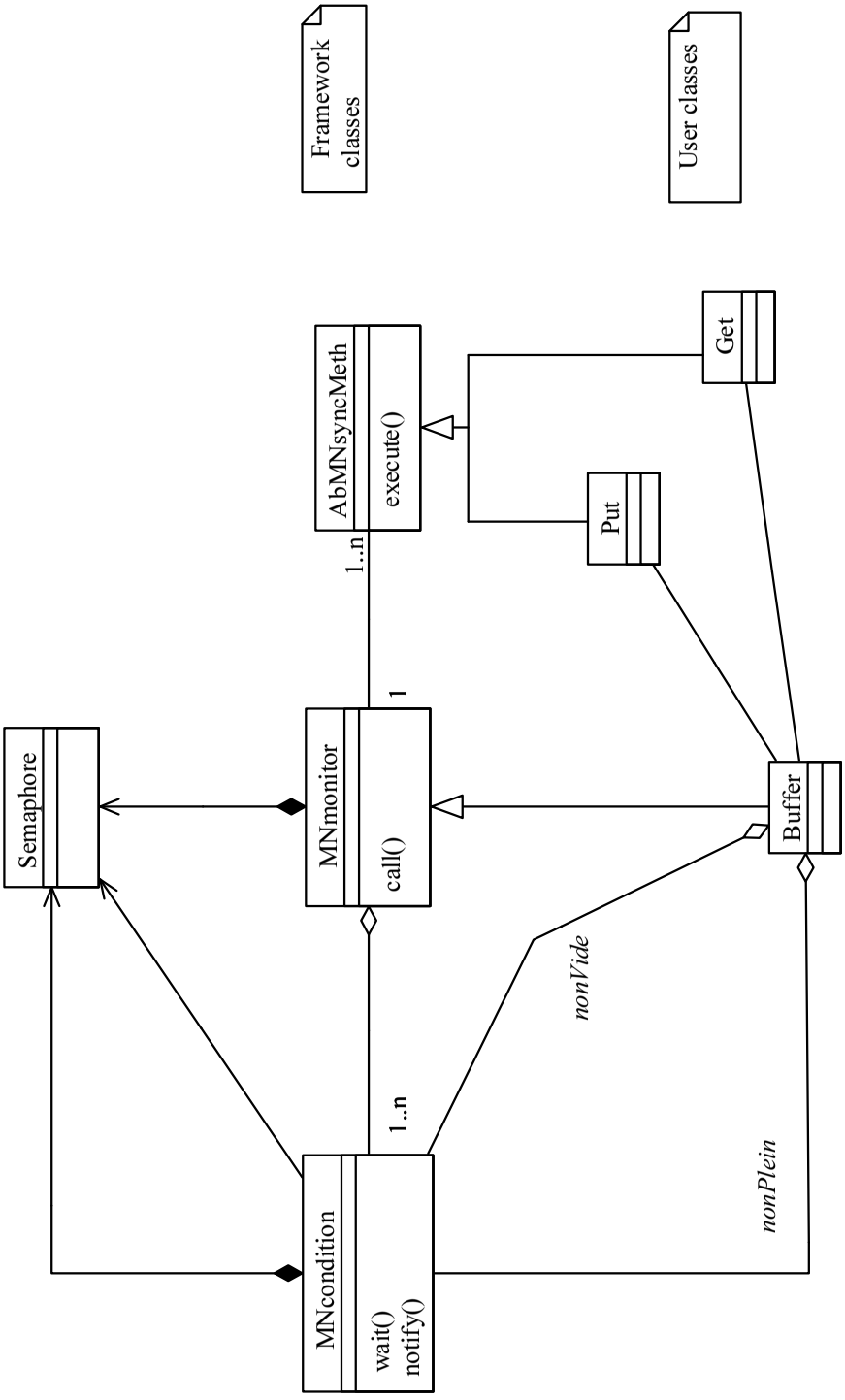

FIG. 3.14 – Exemple d'utilisation du pattern Monitor

de contrôle ». Cette définition ne précise pas la relation entre les méthodes de l'objet actif et son thread de contrôle. Autrement dit, la définition ne mentionne pas si les méthodes de l'objet actif doivent être exécutées dans son thread ou non[5]. Ce qui laisse une part d'ambiguïté dans la sémantique du modèle UML d'un objet actif . Ce modèle ne permet pas la distinction entre les méthodes s'exécutant dans le thread de contrôle de l'objet actif et les autres méthodes s'exécutant dans le thread de contrôle de l'appelant (client).

Active Object Pattern est un design pattern d'architecture permettant l'implémentation de la notion d'objet actif. L'idée de ce design pattern est de décharger le client (l'appelant) de l'exécution des services offerts par l'objet actif qui les exécute dans son propre thread.

Il y a plusieurs variétés de ce design pattern :

- Objets actifs possédants un seul thread de contrôle.
- Objets actifs possédants un nombre limité de thread. Ce nombre correspond généralement aux nombre de méthodes qu'ils possèdent, de telle sorte qu'ils assurent un thread pour l'exécution des invocations d'une méthode (thread par méthode).
- Objets actifs possédants un nombre illimité de thread de contrôle. Dans ce cas l'objet actif associe à chaque invocation de méthode un thread (thread par requête).

Dans les deux premières variétés le nombre de threads est statique, il est connu lors de la conception ce qui facilite le test et la validation de l'application. Par contre dans la troisième variété le nombre de threads est dynamique, il n'est pas connu lors de la conception. A un instant donné, on ne peu pas prévoir le nombre exact de threads contrôlés par l'objet actif.

[5]Généralement lorsque un client invoque une méthode d'un objet, l'exécution du code de cette méthode se déroulera dans le thread de contrôle du client

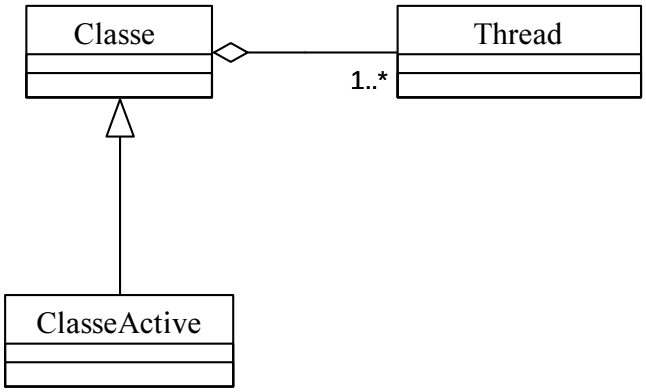

FIG. 3.15 – Modèle d'une classe active

3.6.4.1 Structure

Un objet actif est une instance d'une classe active. La figure 3.15 présente un modèle simple de la classe active. Cette dernière n'est qu'une simple classe possédant au moins un thread.

3.6.4.2 Collaboration

La classe active utilise son thread pour décharger ses clients de l'exécution de quelques services.

3.6.4.3 Intégration aux framework

Nous avons choisis de limiter l'intégration de l'Active Object Pattern aux deux première variantes où le nombre de threads est statique. La troisièmes variété où le nombre de thread est dynamique n'est pas appropriée aux systèmes temps réel embarqués car elle mène à un débordement de création de thread[6] et d'usage mémoire.

[6]Le nombre maximum de thread ou tâche qui peuvent être créés simultanément est limité dans les noyaux ou RTOS temps réel

La figure 3.16 présente un modèle du Active Object Pattern avec un seul thread de contrôle (première variété). Dans ce modèle ActiveObject est une spécialisation (par héritage) de Task. Les appels aux méthodes de l'objet actif sont stockés dans la même queue d'activation pour s'exécuter dans son thread.

Actif Object Pattern peut être implémenté avec un thread pour chaque méthode. La relation dans ce cas entre l'objet actif et les threads est assurée par agrégation. Pour chaque méthode l'objet actif associe une queue d'activation et un thread. Avec un seul thread et une seule queue d'activation ce modèle permet de retrouver un modèle equivalent au modèle précédent illustré dans la figure 3.16.

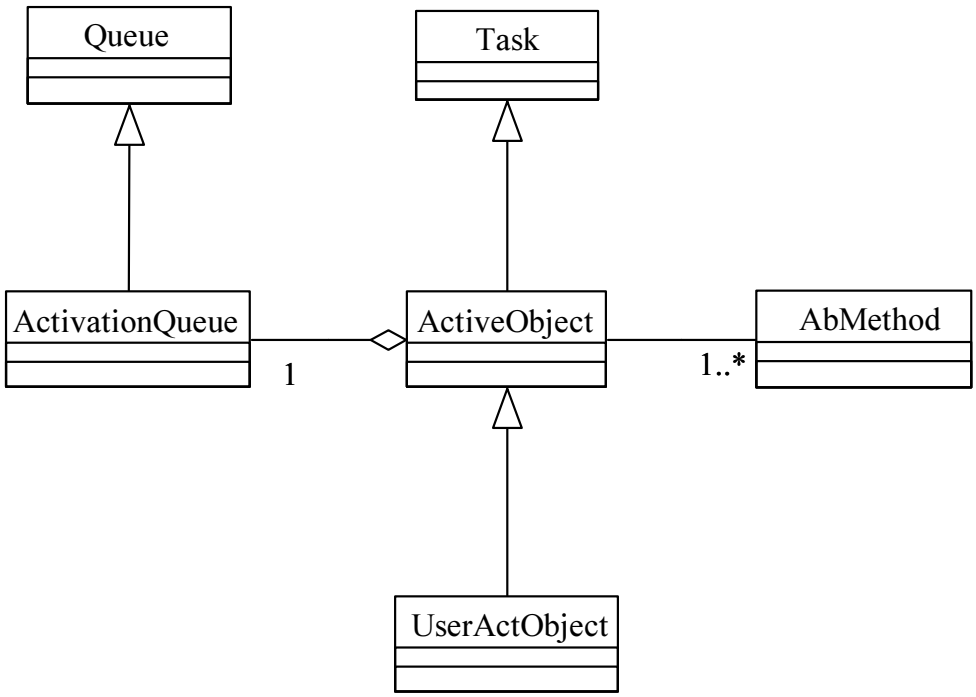

FIG. 3.16 – Modèle du Active Object Design Pattern

3.7 Sous framework de tolérance aux fautes

Les exigences de sécurité et de sûreté de fonctionnement sont plus strictes pour les systèmes temps réel et embarqués que pour d'autres systèmes [Burns and Wellings, 2001]. S'il est tolérable de suspendre l'exécution d'une application de calcul scientifique, il n'est pas de même pour un système de contrôle d'une centrale nucléaire. Dans le premier cas, la suspension ne cause que la perte du temps de calcul, alors que dans le deuxième cas, la suspension peut causer une catastrophe. Il est vital pour des systèmes temps réel de fonctionner en dépit des fautes et c'est la raison pour laquelle la conception de ces systèmes est basée sur des techniques de tolérance aux fautes.

3.7.1 Le design pattern Recovery Block

Les blocs de recouvrement [Horning *et al.*, 1974, Anderson and Kerr, 1976] sont utilisés comme un moyen destiné à assurer qu'un système remplit sa fonction en dépit des fautes. Le but de cette technique est d'augmenter la fiabilité en proposant plusieurs alternatives d'exécution où chacune doit satisfaire un test d'acceptation. Les blocs de recouvrement s'inspirent de la technique de tolérance aux pannes matérielles qui est basée sur la duplication des composants. La tolérance aux pannes est assurée par un composant maître et un certain nombre de composants esclaves. En fonctionnement normal, le composant esclave reste au repos et prêt pour prendre la relève en cas de panne éventuelle du maître. La Figure 3.17 illustre le principe et le fonctionnement des blocs de recouvrement.

Chaque bloc de recouvrement est caractérisé par un checkpoint, un traitement et un test d'acceptation. Le checkpoint symbolise l'état du système avant l'exécution du bloc de recouvrement. Le traitement est la partie critique d'un programme qui doit être exécutée correctement. Son résultat doit subir un test d'acceptation. Le test permet de valider un traitement. Si le test d'acceptation est validé, les résultats du traitement sont délivrés aux clients.

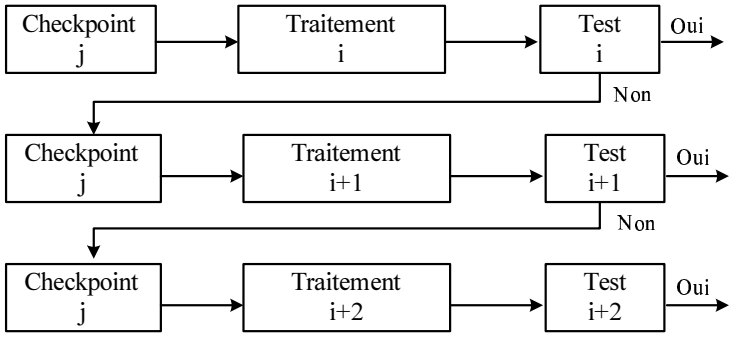

FIG. 3.17 – Principe de fonctionnement des blocks de recouvrement

Dans le cas contraire, le système doit (1) revenir son état initial avant l'exécution du bloc de recouvrement actuel (2) traiter le bloc de recouvrement suivant. Le pseudo code suivant donne la structure de base de la technique de blocs de recouvrement [Burns and Wellings, 2001] :

```
ensure  <acceptance test>
by
<primary module>
else by
<alternative module>
else by
<alternative module>
...
else by
<alternative module>
else error
```

Cette structure est implémentée dans les langages procéduraux en utilisant les instructions conditionnelles auxquelles s'ajoute un mécanisme de restauration de l'environnement car comme expliqué plus haut l'échec d'un bloc de recouvrement doit mener le système à l'état qui précède son exécution. Ceci généralement nécessite d'étendre le langage utilisé avec des fonctionnalités (macros par exemple) permettant la mise en place de ce mécanisme,

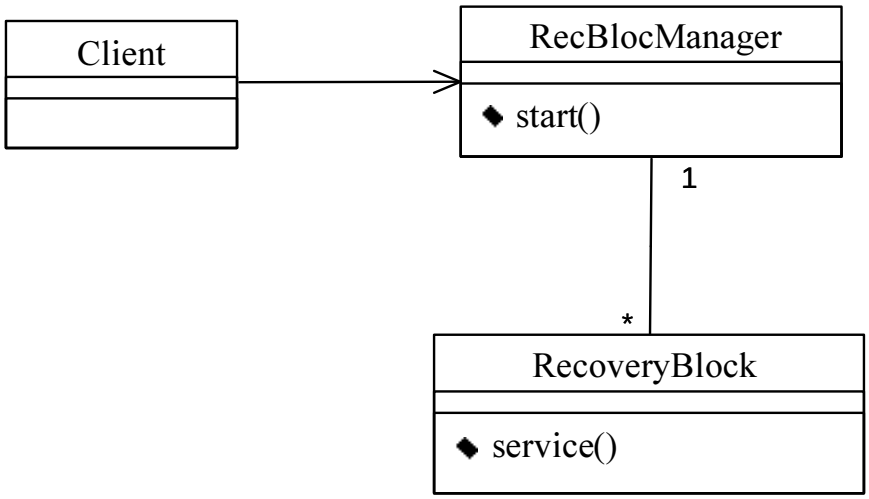

FIG. 3.18 – Structure simple du Recovery Block design pattern

rendant ainsi le code source plus difficile à comprendre et à maintenir. Dans les langages les plus évolués comme C++, Java et Ada, Cette structure est généralement implémentée avec les techniques de gestion des exceptions [Xu *et al.*, 1996, Cristian, 1982] des langages de programmation. Or l'utilisation des exceptions pour l'implémentation des blocs de recouvrement « pollue » le code source et rend son maintien difficile.

Afin d'intégrer cette technique dans notre framework nous avons été obligés de lui trouver une modélisation orientée objet et réutilisable. Finalement notre effort s'est traduit en un design pattern appelé Recovery block design pattern [Machta *et al.*, 2005].

3.7.1.1 Structure

La structure de base de ce design pattern est illustrée dans la Figure 3.18. Le gestionnaire gère des blocs de recouvrement qui doivent s'exécuter l'un à la suite de l'autre si le précédant ne réussi pas son test d'acceptation.

3.7.1.2 Collaboration

RecBlocManger

Le gestionnaire des blocs de recouvrement est le responsable de l'exécution des blocs de recouvrement. Une fois qu'il a invoqué le premier bloc, il ne lancera le suivant que lorsque le précédent n'a pas réussi son test d'acceptation.

RecoveryBlock

Une fois lancé, un bloc de recouvrement commence par exécuter son traitement avant d'appliquer le test d'acceptation et retourner le résultat au gestionnaire si le test d'acceptation est validé.

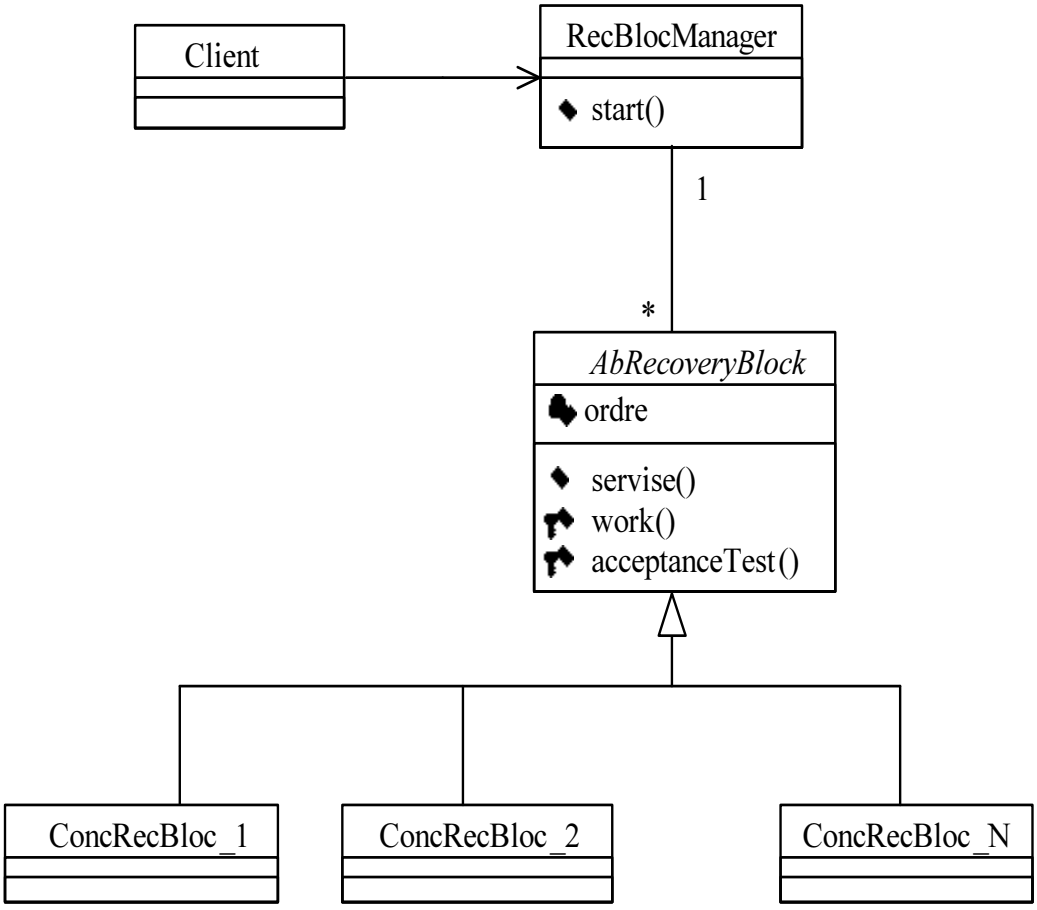

FIG. 3.19 – Diagramme des classes du Recovery Block design pattern dans le framework

La figure 3.20 présente un diagramme de séquence illustrant un cas d'utilisation du Recovery Block Pattern. Le client demande au gestionnaire des blocs de recouvrement (RecoveryBlocManager) l'exécution du traitement critique. Le gestionnaire des blocs de recouvrement commence par lancer le premier bloc de recouvrement(RB1). Une fois lancé, un bloc de recouvrement commence par exécuter son traitement (méthode work( )) avant d'appliquer le test d'acceptation et retourner le résultat au gestionnaire si le test d'acceptation est validé.

3.7.2 Le design pattern N-versions Programming

Cette technique consiste à développer plusieurs versions d'un module ou d'une tâche en espérant que la même erreur ne se répète pas dans toutes les versions. Le principe de la technique se base sur la diversité des versions dans leurs spécifications, implémentation, algorithmes ou même dans leurs langages de programmation.

3.7.2.1 Structure

La structure simplifiée de ce design pattern est illustrée dans la figure 3.21.

3.7.2.2 Collaboration

Judge : juge
Le « Judge » a pour rôle d'élire le meilleur résultat parmi les résultats fournis par les N versions. L'élection du résultat doit être faite selon un algorithme de sélection spécifique au cas traité.

Version
Chaque objet version doit exécuter sa version, du traitement demandé par le client, ensuite il doit passer voter chez le juge.

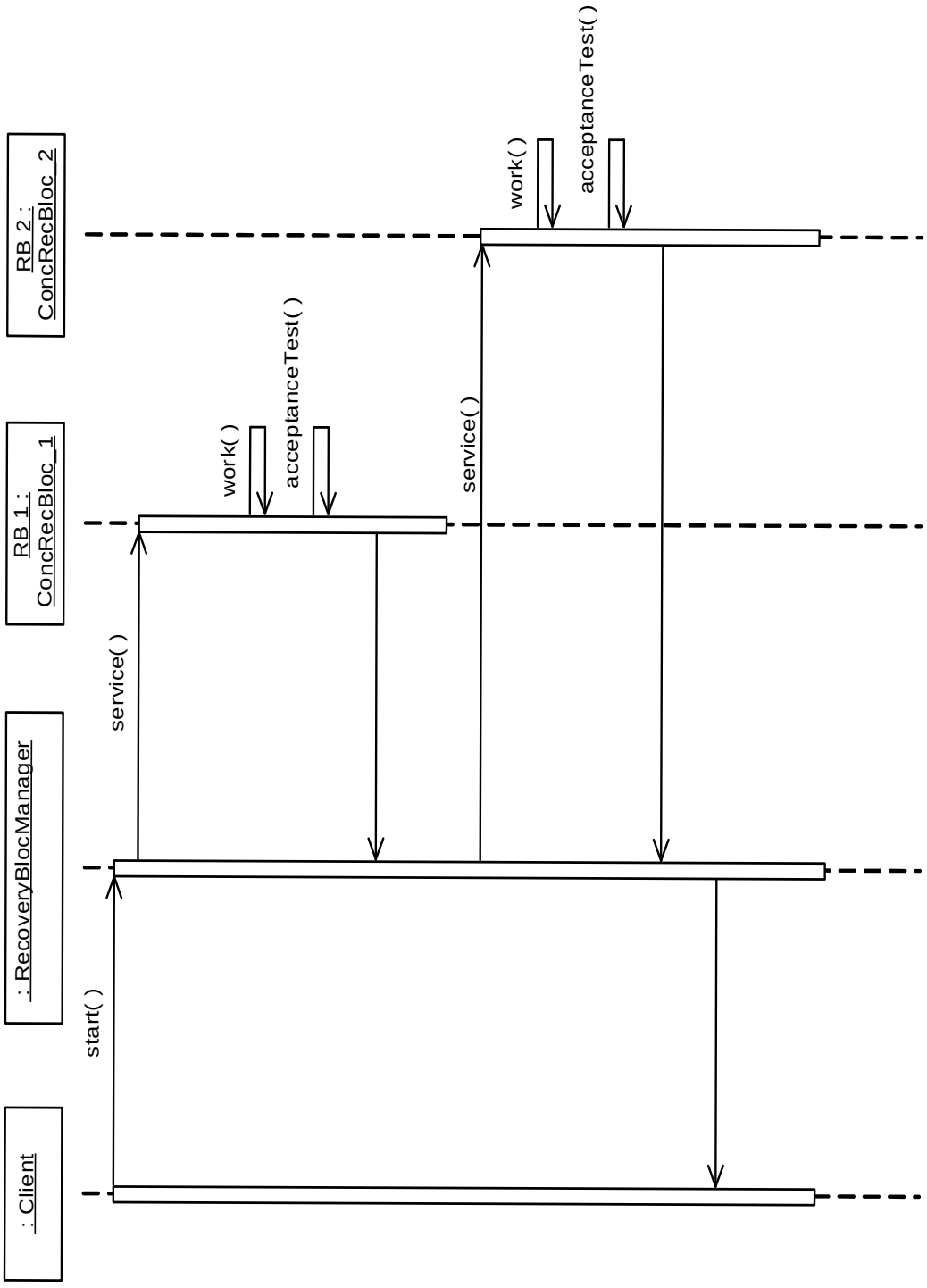

FIG. 3.20 – Diagramme de séquence illustrant un cas d'utilisation du Recovery Block Design Pattern

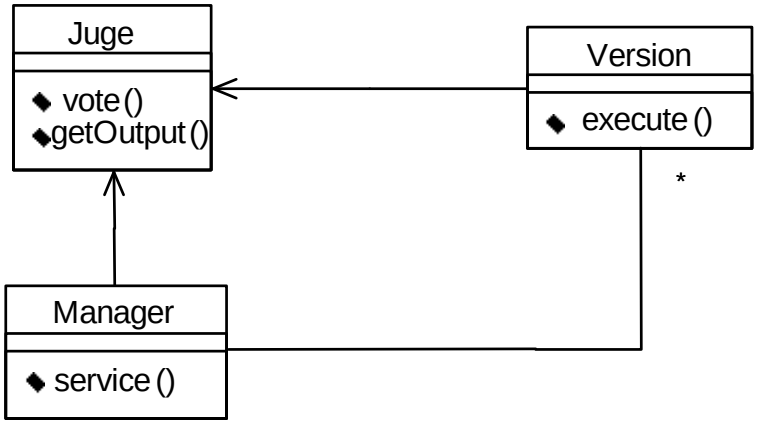

FIG. 3.21 – N-version programming design pattern

Manager
Le manager constitue une interface par laquelle le client peut accéder au service qui lui est offert d'une façon transparente. Le Manager se charge de récupérer les données sollicitées par le client au prés du Juge.

3.7.2.3 Intégration au framework

La figure 3.22 présente l'adaptation de la structure du « N-versions programming » dans le framework.

NVPJuge : Cette classe concrétise le « Juge », elle offre au moins deux méthodes ; vote( ) et getOutput( ). La première méthode permet aux différentes versions d'envoyer les résultats de leurs traitement au Juge. La deuxième méthode résume le rôle du Juge qui va élire le meilleur résultat suivant un algorithme d'élection.

NVPManger : Cette classe maintien une liste dans laquelle sont stockées « les versions ». Ces versions sont exécutées simultanément à la demande du client

AbNVPVersion : c'est la classe abstraite de laquelle dérivent toutes les versions qui spécifient la méthode execute( ). programing

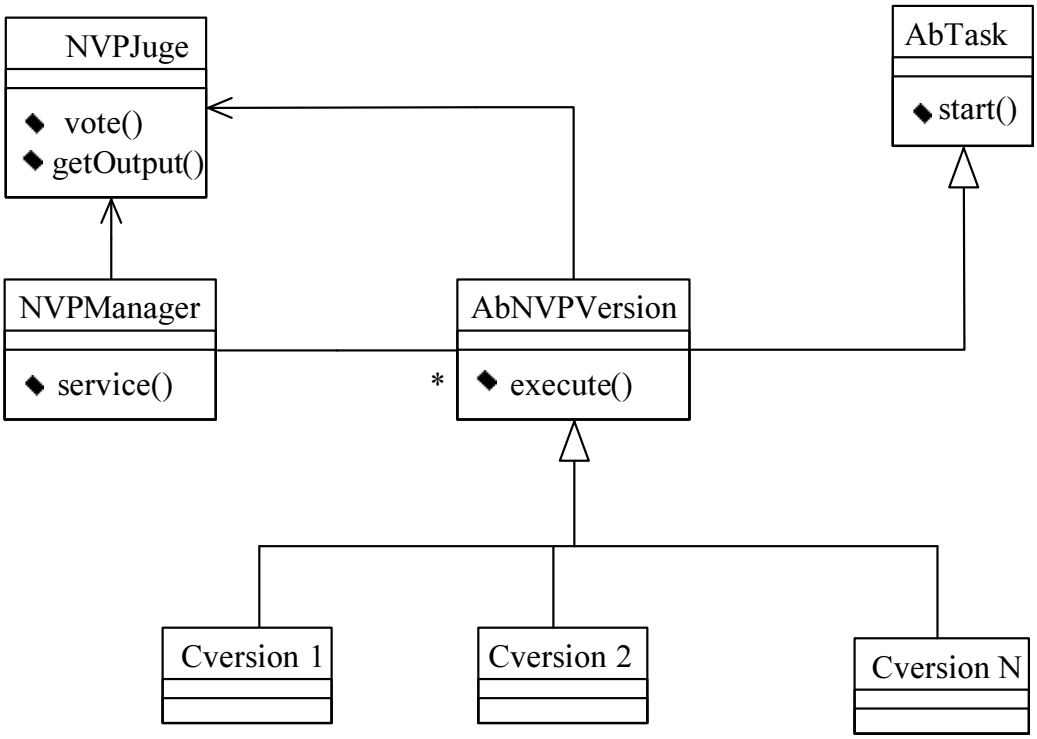

FIG. 3.22 – Structure du design pattern N-version programming dans le framework

La figure 3.23 présente un diagramme de séquence illustrant un cas d'utilisation du N-Version Programming. Le client demande le lancement du traitement en appelant la méthode service( ) du gestionnaire aManger. Ce dernier commence par lancer les versions (trois versions dans cet exemple) en appelant la méthode start( ) de chaque version (Vers1, Vers2 et Vers3). Ces versions exécutent leurs traitements (méthode execute( )) et passent les résultats au juge (aJuge) en appelant la méthode vote( ). Enfin le manager récupère le résultat sélectionné par le juge en appelant la méthode getOutput( ) pour le retourner au client.

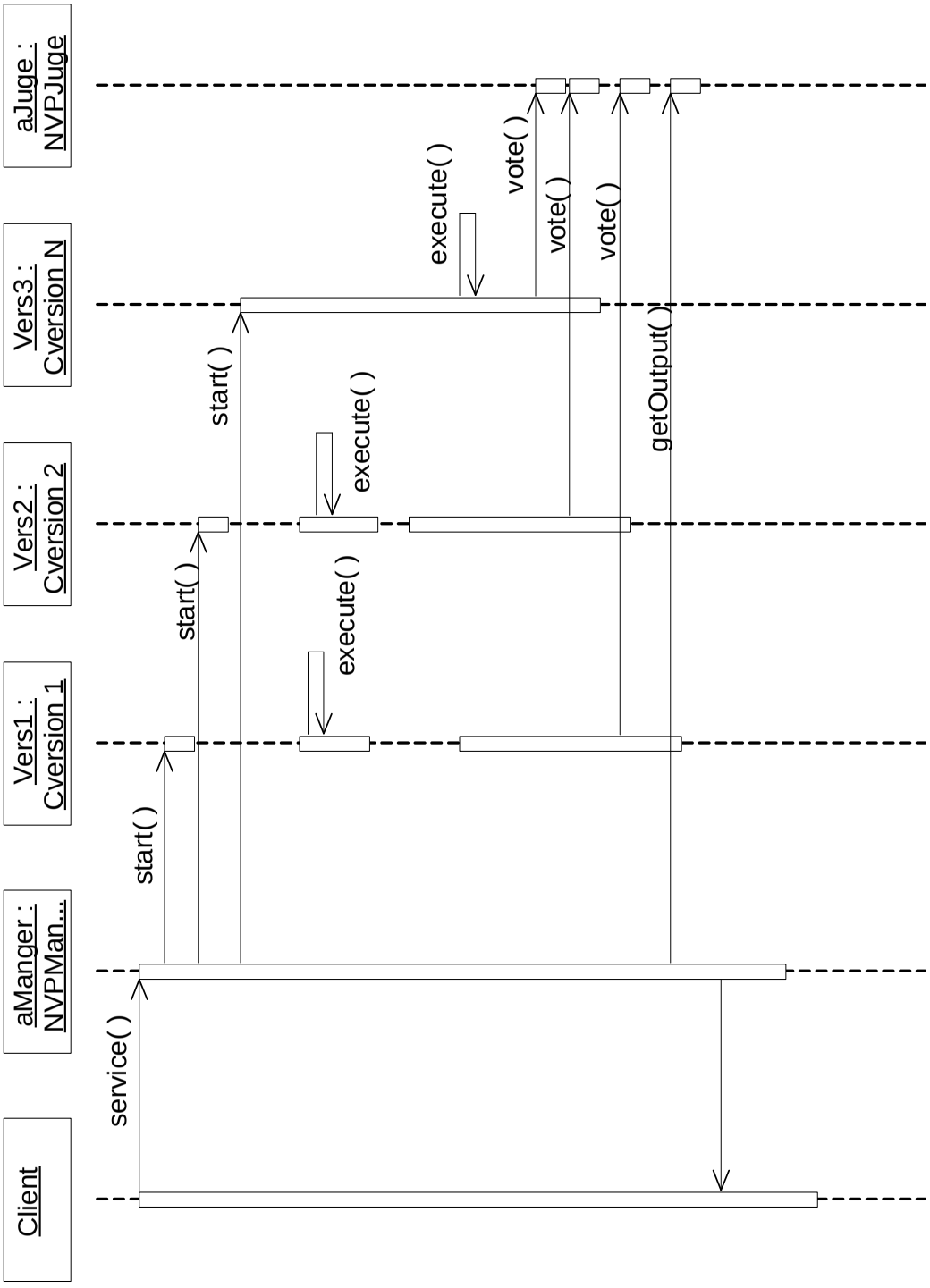

FIG. 3.23 – Diagramme de séquence illustrant un cas d'utilisation du N-Version Programming Design Pattern

3.8 Implantation

L'implémentation du framework a été réalisée sur l'exécutif Vxworks (version 5.4) intégré dans l'environnement de développement Tornado (version 2.0) pour plateforme PC. Le code développé a été testé sur le simulateur intégré VxSim. A ce stade les classes ont été développées avec des interfaces minimales (seulement les classes et les méthodes essentielles ont été implémentées).

Nous envisageons prochainement de porter le framework sur d'autres plateformes et d'autres RTOS afin de définir des interfaces (classes abstraites) communes avec des implémentations spécifiques à l'environnement d'exécution. Ceci permet de porter facilement les applications développées avec le framework d'un RTOS (ou environnement temps réel) à un autre.

3.9 Conclusion

Dans ce chapitre nous avons présenté un framework horizontal pour les systèmes temps réel embarqués. Le framework est modulaire, il est composé de deux sous-frameworks ; un framework de concurrence et un framework de tolérance aux fautes. Chaque sous framework regroupe un ensemble de design patterns en relation avec sa préoccupation. L'architecture de base de chaque sous-framework est le résultat d'intégration de ces design patterns qui entraîne des modifications dans leurs structures.

Chapitre 4

Mise en Œuvre du Framework

4.1 Introduction

Dans ce chapitre nous illustrons l'utilisation du framework pour développer une application temps réel en exploitant une architecture fondée sur les design patterns. Ce chapitre comprend deux exemples. Le premier exemple traite un problème de synchronisation et présente une solution en utilisant le sous-framework de concurrence. Le deuxièmes exemple traite un problème de tolérance aux fautes et présente une solution en utilisant le sous-framework de tolérance aux fautes

4.2 Bras de robot

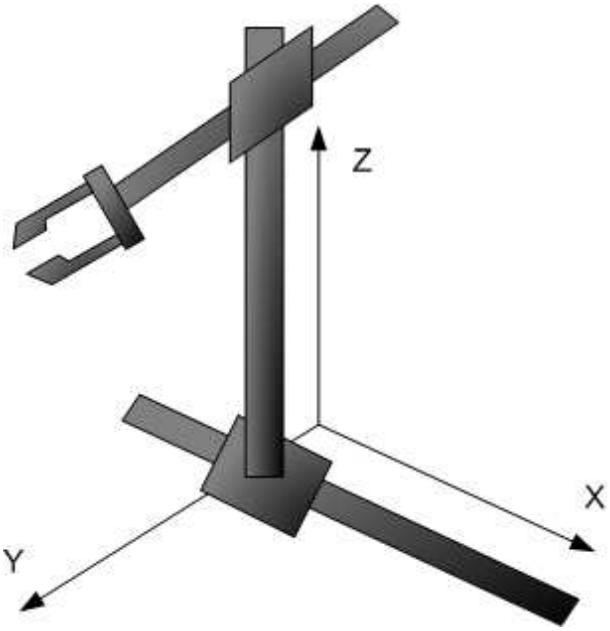

FIG. 4.1 – Un bras de robot avec trois degrés de liberté

On considère un bras de robot dans le milieu industriel (figure 4.1). Ce bras possède trois degrés de liberté, il est capable de faire des mouvements de translation dans les trois dimensions. Tout mouvement du bras est décomposé

en trois mouvements de translation selon les axes du repère cartésien : X, Y et Z.

Le robot possède trois moteurs électriques, chaque moteur assure le mouvement de translation dans les deux sens. L'inversion du sens de rotation du moteur entraîne l'inversion du sens de translation. Un moteur est caractérisé par sa puissance et par sa vitesse en nombre de tours par seconde. La translation effectuée par le bras est proportionnelle aux nombres de tours effectués par le moteur. Chaque moteur est géré par un contrôleur qui commande les translations. Le contrôleur est une tâche qui boucle indéfiniment. A chaque boucle, la tâche récupère le nouveau déplacement, l'additionne avec le positionnement actuel et ordonne les mouvements nécessaires.

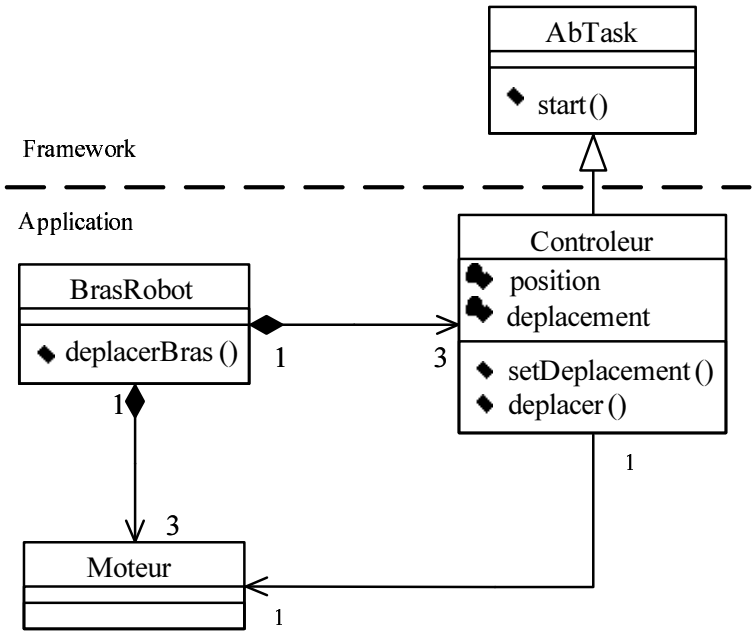

FIG. 4.2 – Modèle UML d'un bras de robot

La figure 4.2 illustre un diagramme de classes modélisant le bras de robot. La classe contrôleur hérite de la classe AbTask et spécifie la méthode start(). Elle possède deux attributs ; position et déplacement et deux méthodes ; setDeplacement() et deplacer(). L'attribut position est modifié par la méthode

setDeplacement(). La méthode deplacer() commande le déplacement vers la nouvelle position calculée en additionnant le déplacement avec la position actuelle. Le pseudo code suivant exprime une implémentation possible de la méthode start() dans la classe Contrôleur :

```
Controleur :: start() {
    ...
    position = position + deplacement ;
    deplacer();
    deplacement = 0;
    ...
}
```

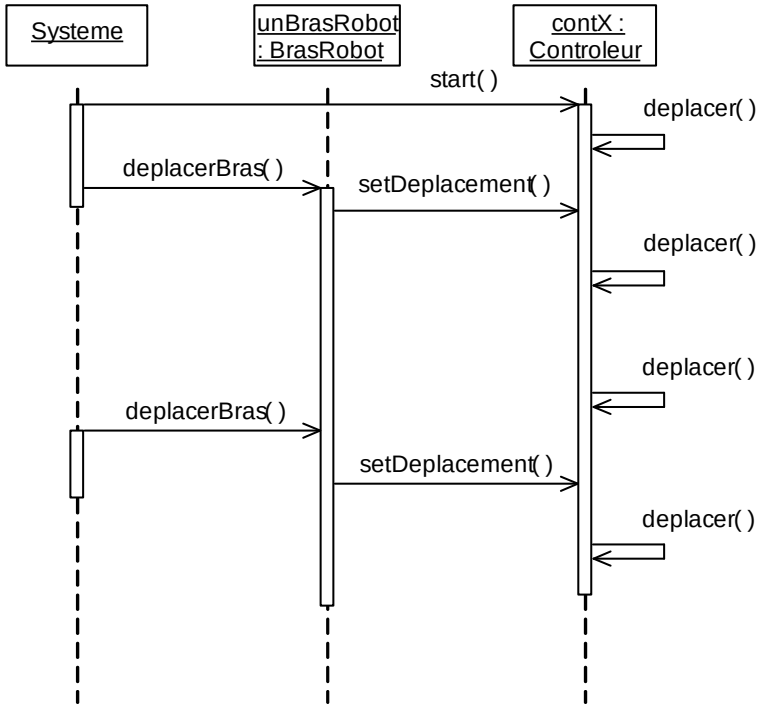

FIG. 4.3 – Diagramme de séquence illustrant les interactions avec un seul contrôleur

Le diagramme de séquence illustré dans la figure 4.3 présente un exemple de schémas d'exécution avec un seul contrôleur (contX). Le système com-

mande le mouvement du bras de robot en appelant la méthode deplacerBras(). Le bras de robot détermine les mouvements à effectuer pour chaque contrôleur et le communique à ce dernier en appelant la méthode setDeplacement(). Le contrôleur est en exécution continue ; à chaque boucle la méthode deplacer() est appelée pour effectuer le déplacement nécessaire.

Le problème avec cette solution c'est que la variable deplacement constitue une ressource critique accédée par deux tâches ; la tâche du contrôleur qui en est lectrice et la tâche dans laquelle s'exécute la méthode setDeplacement() qui en est rédactrice ce qui entraîne un problème de cohérence de données nécessitant une synchronisation. Ce problème peut être résolu en utilisant les files de messages. Chaque contrôleur possède une queue de messages dans laquelle il reçoit les déplacements. Cette solution peut être facilement implémentée avec le Message Queuing Pattern intégré dans le framework (figure 4.4).

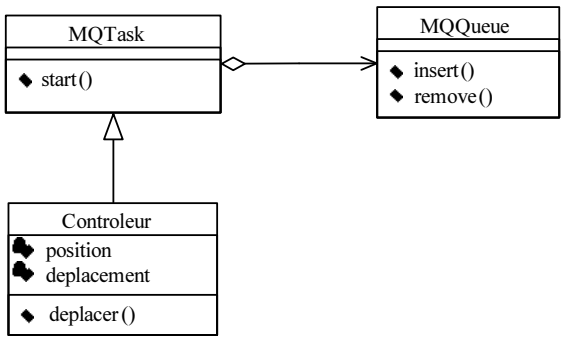

FIG. 4.4 – Contrôleur implémenté avec message queuing design pattern

Le diagramme de séquence illustré dans la figure 4.5 présente un exemple de schéma d'exécution après l'intégration du Message Queuing Pattern dans le modèle du bras de robot. Lorsque le système ordonne un déplacement au bras de robot, ce dernier utilise les messages pour communiquer le nouveau déplacement aux contrôleurs. Il envoie à chaque contrôleur un message contenant le nouveau déplacement à effectuer et ceci en utilisant la méthode insert() de la queue de message. Le contrôleur reste en écoute de l'arrivé d'un

nouveau message dans sa queue de message pour le récupérer avec la méthode remove() et effectuer le déplacement demandé.

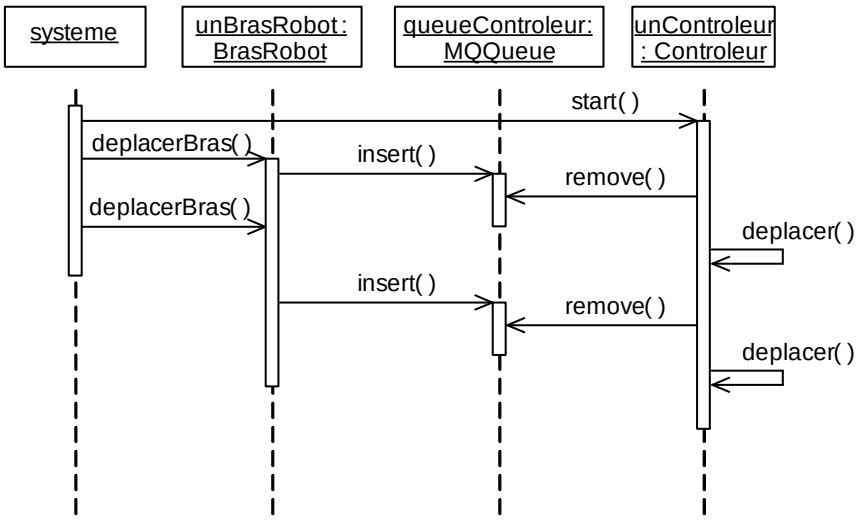

FIG. 4.5 – Diagramme de séquence illustrant les interactions avec un contrôleur en utilisant Message Queuing Design Pattern

Les bras de robots fonctionnent dans les espaces partagés. Ils ont à collaborer ensemble pour éviter les collisions. Ils sont aussi menés à collaborer pour assurer l'exécution d'une tâche comme, par exemple, le transport d'une pièce. Cette tâche nécessite la synchronisation des mouvements des bras de robots afin d'éviter la détérioration de la pièce. Le design pattern Rendezvous est une bonne solution pour la synchronisation des bras dans ce cas.

Le diagramme de séquence illustré dans la figure 4.6 présente un exemple d'utilisation des classes du framework pour appliquer le principe du Rendezvous Pattern afin de synchroniser les mouvements des bras de robots. Pour des raisons de clarté nous avons limité l'exemple à deux bras de robots. Pour les mêmes raisons nous avons aussi limité les mouvements à une seule dimension. Le diagramme de séquence présente les interactions entre les objets qui concrétisent les acteurs principaux de cet exemple. L'objet système est celui qui commande les mouvements des deux bras de robot. R1contX et R2contX

sont les deux contrôleurs responsables sur les mouvements de translation dans le l'axe X associés respectivement aux objets Bras1 et Bras2 concrétisant les deux bras de robot. L'objet Rendezvous est responsable de la synchronisation des deux tâches Contrôleur. La stratégie de synchronisation est déterminée dans l'objet Politique.

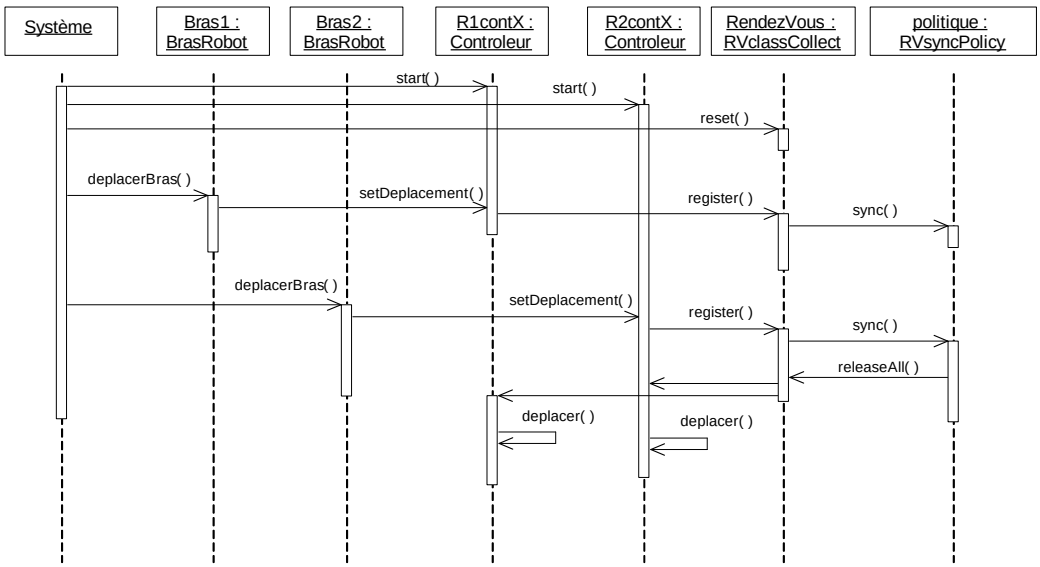

FIG. 4.6 – Diagramme de séquence illustrant l'application du Rendezvous pattern

Le système commence par lancer les deux tâches R1contX et R2contX (méthode start()) et initialiser l'objet Rendezvous (méthode reset()). Ensuite, le système commande le mouvement des deux bras (méthode deplacerBras()). Les deux bras communiquent le déplacement à effectuer à leurs contrôleurs (méthode setDeplacement()). Avant d'effectuer le déplacement, chaque contrôleur doit s'enregistrer aux prés de l'objet Rendezvous (méthode register()). L'opération d'enregistrement entraîne le blocage de la tâche jusqu'à la réalisation des conditions de synchronisation. Après chaque opération d'enregistrement l'objet Rendezvous vérifie les conditions de synchronisation selon la stratégie de l'objet Politique (méthode sync()). Dans ce cas,

la stratégie de synchronisation consiste à bloquer les tâches et ne les libérer qu'après que toutes les tâches concernées par le rendez-vous se soient enregistrées. Dans notre exemple le nombre des tâches concernées par le rendez-vous est deux. La première tâche qui s'enregistre sera bloquée pour être libérée par l'enregistrement de la deuxième tâche qui ne se bloque pas. Lorsque le nombre de tâches enregistrées atteint deux, l'objet Politique libère la tâche bloquée (méthode releaseAll()). Après la libération de la tâche bloquée les deux contrôleurs continuent leurs exécutions et effectuent les déplacements recommandés (méthode deplacer()).

4.3 Système d'acquisition

Dans ce paragraphe nous allons présenter un exemple illustrant l'utilisation du sous-framework de tolérance aux fautes et en particulier la technique des blocs de recouvrement.

Les systèmes temps réel sont en interaction continue avec leurs environnements. Ils sont amenés à déterminer des grandeurs physiques comme vitesse, température, pression, etc. Le calcul de ces grandeurs dépend de la réaction des capteurs vis-à-vis des changements des phénomènes physiques et/ou chimiques de l'environnement. La détermination de ces grandeurs revient souvent à une modélisation mathématique du problème, un système d'équations différentielles est un exemple courant que ce soit en physique ou en chimie. Le système embarqué doit résoudre numériquement le système d'équations. Il exist plusieurs méthodes de résolution numérique d'un système d'équations différentielles. Ces méthodes ne donnent pas des solutions exactes mais des valeurs rapprochées avec des précisions différentes.

Considérons une valeur sensible à calculer en résolvant un système d'équations différentielles. La valeur doit être calculée avec la meilleure précision possible. La valeur doit être calculée avec la meilleure précision possible. Elle doit être comprise entre deux bornes qui définissent la précision souhaitée (v1 $<$ valeur $<$ v2). Dans ce cas, tout calcul de la valeur dépassant ces

bornes est considéré comme faute. Afin d'assurer cela nous devons appliquer plusieurs méthodes de résolution.

Le principe des blocs de recouvrement convient à ce problème comme le montre l'algorithme suivant :

```
Assurer une bonne précision par
      méthode 1
si non par
      méthode 2
si non par
      méthode 3
si non
      Erreur
```

L'implémentation de cet algorithme avec Recovery Block design pattern consiste consacrer à chaque méthode de résolution, une classe dérivant de la classe abstraite AbRecoveryBlock.

Soit Euler, Feynman et Runge-Kutta les méthodes utilisées pour résoudre un système d'équations différentielles. La figure 4.7 illustre un diagramme de classes modélisant l'application du Recovery Block design pattern en utilisant ces méthodes.

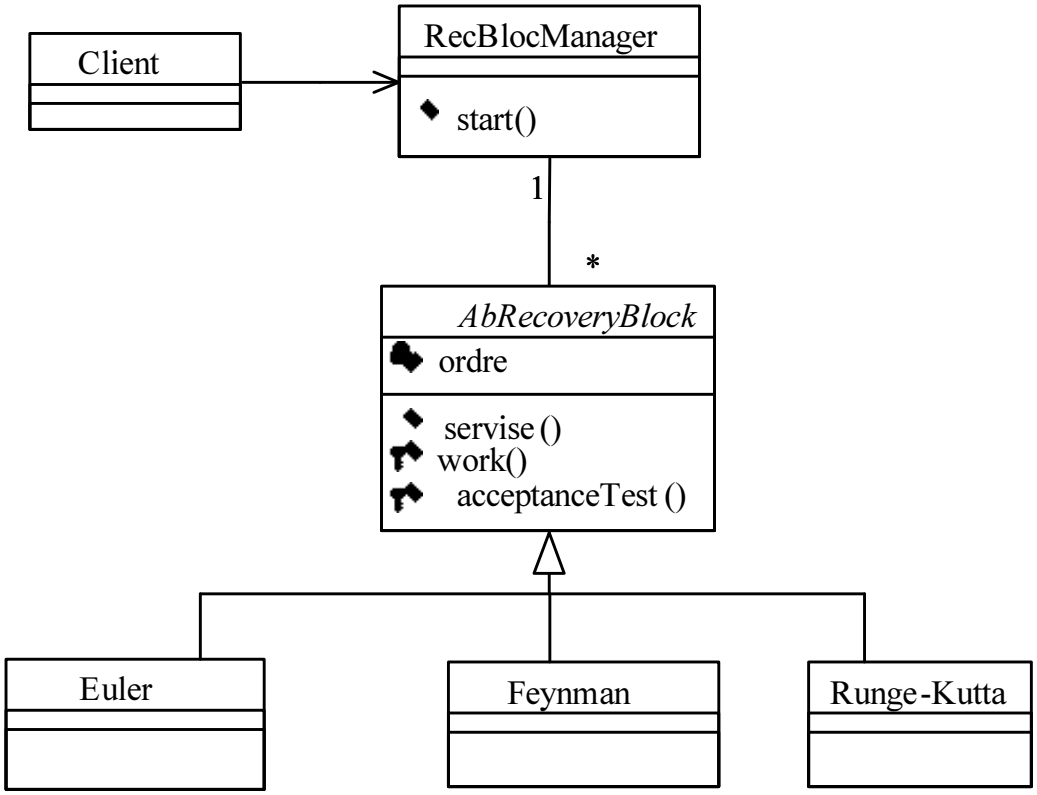

FIG. 4.7 – Diagramme des classes du Recovery Block pattern appliqué aux méthodes de résolutions numériques

La classe AbRecoveryBlock possède deux méthodes abstraites (work() et acceptanceTest()) et une méthode concrète (service()). Les trois classes Euler, Feynman et Runge-Kutta dérivant de cette classe doivent implémenter les méthodes abstraites. La méthode work() est distinguée dans chaque classe par l'implémentation de sa méthode de résolution numérique des équations, alors que la méthode acceptanceTest() est la même pour ces trois classes.

Dans l'exemple illustré par le diagramme de séquence de la figure 4.8, seulement deux des trois méthodes ont été utilisées. La première méthode (Euler) n'a pas réussi à donner un bon résultat alors que la deuxième (Feynman) a donné un résultat avec une précision acceptable.

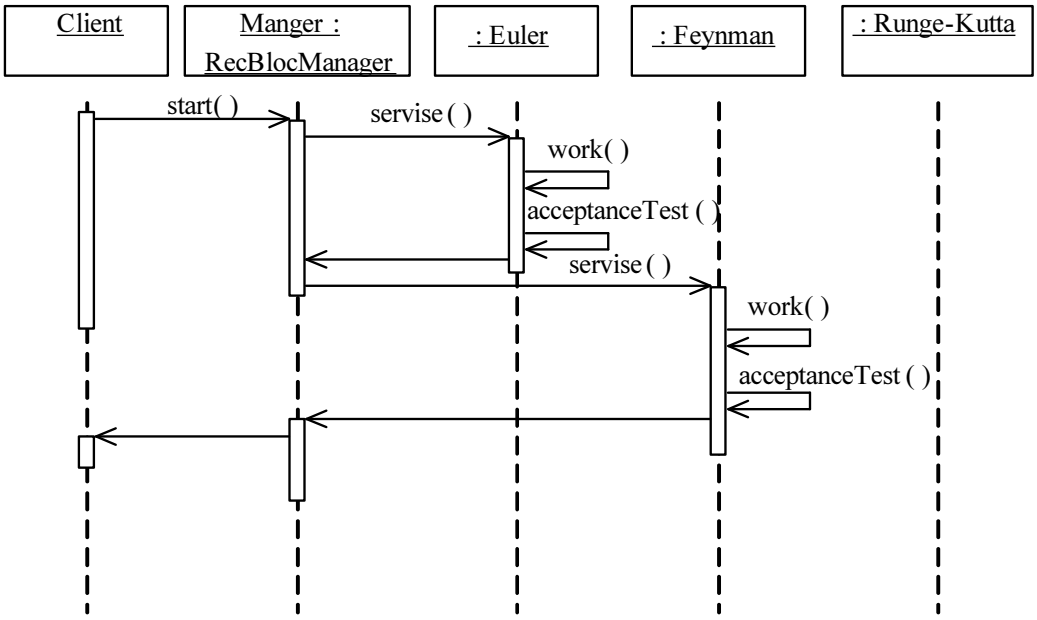

FIG. 4.8 – Diagramme de séquence

Afin d'augmenter la réutilisation, on peut modifier le modèle présenté dans la figure 4.7 en ajoutant une classe abstraite MethNumerique héritant de la classe AbRecoveryBlock et qui offre une implémentation de la méthode abstraite acceptanceTest(). Les trois classes Euler, Feynman et Runge-kutta héritent de la nouvelle classe abstraite MethNumerique et spécifient uniquement la méthode work(). Le nouveau modèle est illustré dans la figure 4.9.

4.4 Conclusion

Dans ce chapitre nous avons présenté deux exemples d'utilisation du framework pour le développement d'applications temps réel. Le premier exemple concerne un problème de concurrence et qui a étudié le cas de synchronisation des bras de robot dans le milieu industriel. Le deuxième exemple a considéré un système d'acquisition pour illustrer le cas de tolérance aux fautes.

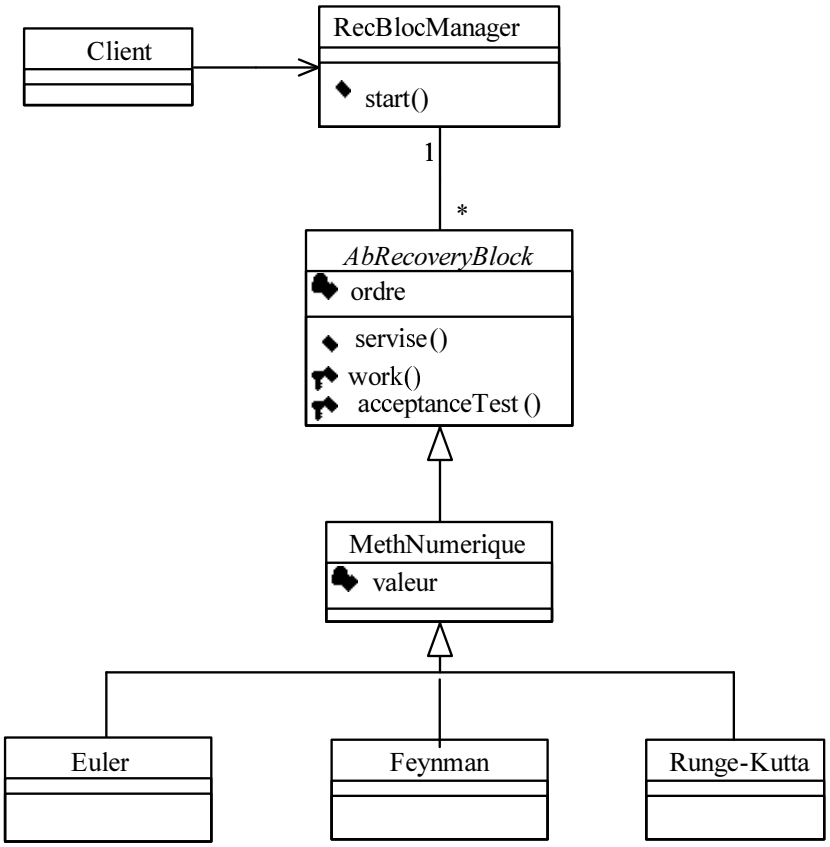

FIG. 4.9 – Diagramme des classes du Recovery Block pattern comprenant une classe spécifiant la méthode acceptanceTest()

En plus de la réutilisation du code, l'utilisation du framework permet d'avoir du code facile à maintenir et à documenter. Ceci s'explique par l'utilisation des classes du framework pour résoudre des problèmes de concurrence ou de tolérance aux fautes au lieu d'intégrer des techniques qui traitent ces préoccupations (problèmes) dans les classes de l'application.

Conclusion générale

La progression technologique était un facteur déterminant dans l'expansion des systèmes temps réel embarqués. Ce qui était, il y a une dizaine d'années, un confort pour un chercheur chevronné ne satisfait pas les besoins d'un console de jeu portable pour un enfant de nos jour.

Le génie logiciel appliqué au domaine du temps réel embarqué est un axe qui suscite un intérêt croissant des chercheurs. Ceci s'explique aisément par le nombre grandissant de ces systèmes et par le coût élevé de leur développement à cause de leur variation. Les frameworks continuent d'être un sujet actif de recherche grâce à leurs caractéristiques permettant la réutilisation du code et de la conception réduisant ainsi le coût et le temps de développement.

L'objectif poursuivi dans ce travail consiste à proposer un framework horizontal orienté objet pour le développement des systèmes temps réel et embarqués dont la conception est à base de design pattern. L'architecture du

framework proposé est modulaire. Le framework est composé d'un ensemble de sous-frameworks où chacun traite d'une préoccupation. A l'état présenté dans ce mémoire, le framework comprend un sous-framework de tolérance aux fautes et un autre de concurrence. Cette architecture devrait permettre une meilleure utilisation du framework et faciliter sa maintenance et son extension.

Dans le premier chapitre nous avons présenté les patterns et leur utilisation dans le domaine du génie logiciel ainsi que leur application dans le domaine du temps réel. Nous avons aussi présenté d'autres concepts liés aux patterns ainsi que des travaux de recherche s'intéressant à ce domaine.

Le second chapitre a été consacré à la présentation des frameworks. Nous avons commencé par les introduire et en présenter quelques avantages et caractéristiques. Nous avons aussi présenté l'utilisation des frameworks dans le domaine du temps réel. Le chapitre a été clôturé par l'étude de quelques exemples de frameworks temps réel et une conclusion.

Dans le troisième chapitre nous avons proposé un framework horizontal pour les systèmes temps réel embarqués. Nous avons commencé par présenter l'intérêt du framework ainsi que le processus suivi lors de sa conception. Le reste de ce chapitre a été consacré pour la présentation de l'architecture du framework et ses différent composants.

Enfin, le dernier chapitre a traité l'application du framework pour le développement des applications temps réel. Le chapitre a présenté deux exemples le premier a traité le cas du bras de robot dans le milieu industriel, alors que le deuxième exemple a traité le cas d'un système d'acquisition.

Bien que la phase de conception d'un framework est une étape cruciale dans le cycle de vie du framework car elle définit son architecture générale, elle reste la première étape qui doit être suivie par d'autres qui sont aussi déterminantes dans la réussite du framework. Nous sommes convaincus que notre travail laisse de nombreux points susceptibles de constituer d'intéressantes perspectives futures. Parmi ceux qui nous semblent digne d'intérêt nous pouvons envisager ce qui suit :

- L'extension du framework en intégrant d'autres sous-frameworks ayant d'autres préoccupations comme l'ordonnancement et l'aspect temporel.
- Le développement du framework pour des différentes plateformes temps réel et pour des différents langages de programmation comme Ada, java temps réel, etc.
- Fournir une documentation complète concernant l'architecture du framework et son utilisation.
- L'optimisation du code source pour les systèmes temps réel.
- La mise au point d'un outil en se basant sur l'architecture du framework pour la génération de codes à partir des diagrammes UML.

Bibliographie

[Alexander, 1977] Christopher ALEXANDER. *A Pattern Language.* Oxford University press, New York, 1977.

[Ambler, 1998] Scott W. AMBLER. *Process Pattern : Building Large-Scale Systems Using Object Technology.* Cambridge University Press, July 1998.

[Anderson and Kerr, 1976] T. ANDERSON and R. KERR. Recovery blocks in action : A system supporting high reliability. In *ICSE*, pages 447–457, 1976.

[Balanyi and Ferenc, 2003] Zsolt BALANYI and Rudolf FERENC. Mining design patterns from c++ source code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, page 305–314. IEEE Computer Society, September 2003.

[Batory, 1986] Don S. BATORY. Genesis : a project to develop an extensible database management system. In *OODS '86 : Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 207–208, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[Beck and Cunningham, 1987] Kent BECK and Ward CUNNINGHAM. Using pattern languages for object-oriented programs. In *OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming*, 1987.

[Bichler *et al.*, 1998] M. BICHLER, A. SEGEV, and J. ZHAO. Component-based E-commerce : Assessment of current practices. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(4) :7–14, 1998.

[Brown *et al.*, 1998] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis.* Wiley, 1998.

[Burns and Wellings, 2001] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages : Ada95, Real-Time Java and Real-Time POSIX.* Addison Wesley Longmain, 3 éme edition, 2001.

[Buschmann *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1 : A System of Patterns.* Software Design Pattern. Wiley, 1996.

[Busmann, 1993] Frank Busmann. Rational architecture for object-oriented systems. *Journal of Object-Oriented Programming*, 6(5), September 1993.

[Campbell and Islam, 1993] Roy H. Campbell and Nayeem Islam. A technique for documenting the framework of an object-oriented system. *Computing Systems*, 6(4) :363–389, 1993.

[Christensen and Rn, 2000] H. Christensen and H. Rn. A case study of framework design for horizontal reuse. 2000.

[Cilwa *et al.*, 1994] Paul S. Cilwa, Jeff Duntemann, and Paul Cilwa. *Windows Programming Power with Custom Controls.* Coriolis, 1994.

[Coplien, 1992] James O. Coplien. *Advanced C++ programming styles and idioms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[Cristian, 1982] Flaviu Cristian. Exception handling and software fault tolerance. *IEEE Trans. Computers*, 31(6) :531–540, 1982.

[Dijkstra, 1967] Edsger W. Dijkstra. The structure of the ẗhe¨multiprogramming system. In *SOSP '67 : Proceedings of the first ACM symposium on Operating System Principles*, pages 10.1–10.6, New York, NY, USA, 1967. ACM Press.

[Douglass, 1999] Bruce Powel Douglass. *Doing Hard Time : Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* The Addison-Wesley Object Technology Series. Addison Wesley Professional, May 1999.

[Douglass, 2002] Bruce Powel DOUGLASS. *Real-Time Design Patterns : Robust Scalable Architecture for Real-Time Systems.* The Addison-Wesley Object Technology Series. Addison Wesley Professional, September 2002.

[Douglass, 2004] Bruce Powel DOUGLASS. *Real Time UML : Advances in the UML for Real-Time Systems.* The Addison-Wesley Object Technology Series. Addison Wesley Professional, 3 éme edition, February 2004.

[Eggenschwiler and Gamma, 1992] Thomas EGGENSCHWILER and Erich GAMMA. Et++swapsmanager : using object technology in the financial engineering domain. In *OOPSLA '92 : conference proceedings on Object-oriented programming systems, languages, and applications*, pages 166–177, New York, NY, USA, 1992. ACM Press.

[Elrad *et al.*, 2001] Tzilla ELRAD, Robert E. FILMAN, and Atef BADER. Aspect-oriented programming : Introduction. *Commun. ACM*, 44(10) :29–32, 2001.

[Fayad *et al.*, 1999] Mohamed E. FAYAD, Ralph E. JOHNSON, and Douglas C. SCHMIDT. *Building Application Frameworks : Object-Oriented Foundations of Framework Design.* John Wiley & Sons, Inc., New York, NY, USA, September 1999.

[Gamma, 1992] Erich GAMMA. Object-oriented software development based on et++ : Design patterns, class library, tools. *Springer-Verlag*, 1992.

[Gamma *et al.*, 1995] Erich GAMMA, Richard HELM, Ralph JOHNSON, and John VLISSIDES.. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley,Reading, 1995.

[Geeter, 1999] Jean-Marie De GEETER. *Approche du temps réel industriel.* Technosup. Ellipses, juin 1999.

[Gomaa, 2000] Hassan GOMAA. *Designing Concurrent, Distributed, and Real-Time Applications with UML.* The Addison-Wesley Object Technology Series. Addison Wesley Professional, 2000.

[Harrison and Ossher, 1993] William HARRISON and Harold OSSHER. Subject-oriented programming : a critique of pure objects. In *OOPSLA '93 : Proceedings of the eighth annual conference on Object-oriented pro-*

gramming systems, languages, and applications, pages 411–428, New York, NY, USA, 1993. ACM Press.

[Hoare, 1974] C. A. R. HOARE. Monitors : an operating system structuring concept. *Commun. ACM*, 17(10) :549–557, 1974.

[Horning *et al.*, 1974] James J. HORNING, Hugh C. LAUER, P. M. MELLIAR-SMITH, and Brian RANDELL. A program structure for error detection and recovery. In *Symposium on Operating Systems*, pages 171–187, 1974.

[Hsiung *et al.*, 2005] Pao-Ann HSIUNG, Trong-Yen LEE, Jih-Ming FU, and Win-Bin SEE. Sesag : an object-oriented application framework for real-time systems : Research articles. *Softw. Pract. Exper.*, 35(10) :899–921, 2005.

[Hsiung *et al.*, 2001] Pao-Ann HSIUNG, Feng-Shi SU, Chuen-Hau GAO, Shu-Yu CHENG, and Yu-Ming CHANG. Verifiable embedded real-time application framework. In *Proceedings of the International Symposium on Real-Time Applications and Systems (RTAS'01), WiP session*, pages 109–110. IEEE Computer Society Press, May 2001.

[I-Logix, 2005] I-LOGIX. Uml 2.0 based model-driven development software for real-time embedded systems. [En ligne]. Adresse URL : http ://www.ilogix.com, Page consultée le 21 août 2005.

[Jain and Schmidt, 1997] Prashant JAIN and Douglas C. SCHMIDT. Service configurator : A pattern for dynamic configuration of services. In *COOTS*, pages 209–220, 1997.

[Johnson, 1992] Ralph JOHNSON. Documenting frameworks using patterns. In *Object Oriented Programming System, Languages, and Applications Conference Proceedings*, pages 63–76, Vancouver, Britsh Columbia, Canada, October 1992. ACM Press.

[Johnson, 1997] R.E. JOHNSON. Frameworks = components + patterns. *Communication of the ACM*, 40(10) :39–42, October 1997.

[Johnson and Foote, 1988] Ralph E. JOHNSON and Brian FOOTE. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2) :22–35, 1988.

[Khayati, 2001] Oualid KHAYATI. Vers des systèmes de patrons formalisés et outiés. Master's thesis, Université JOSEPH FOURIER de Grenoble, september 2001.

[Kuan *et al.*, 1995] T. KUAN, W.-B. SEE, and S.-J. CHEN. An object-oriented real-time framework and development environment. In *Proceeding of OOPSLA*, 1995.

[Lau, 2000] Yun-Tung LAU. *Art of Objects, The : Object-Oriented Design and Architecture*. The Addison-Wesley Object Technology Series. Addison Wesley Professional, 2000.

[Lavender and Schmidt, 1995] R. Greg LAVENDER and Douglas C. SCHMIDT. Active object : an object behavioral pattern for concurrent programming. In *Proc.Pattern Languages of Programs*, 1995.

[Lea, 2000] Doug LEA. Patterns-discussion faq. page web, 2000.

[Machta *et al.*, 2005] Naoufel MACHTA, Adel KHALFALLAH, and Samir Ben AHMED. Recovery block design pattern : Un design pattern pour la conception des systèmes tolérants aux fautes. In *3 ème Conférence Internationale de Sciences Electroniques, Technologies de l'Information et des Télécommunications*, Sousse Tunisie, Mars 2005.

[Mattsson, 1996] Michael MATTSSON. Object-oriented frameworks - a survey of methodological issues. Master's thesis, Department of Software Engineering and Computer Science University of Karlskrona/Ronneby SWEDEN, Soft Center SE-372 25 RONNEBY SWEDEN, February 1996.

[McKegney, 2000] Ross MCKEGNEY. Application of patterns to real-time object-oriented software design. Master's thesis, Department of Computing & Information Sciences, Queen's University, 2000.

[McKegney and Shepard, 2000] Ross MCKEGNEY and Terry SHEPARD. Design patterns and real-time object-oriented modeling (poster session). In *OOPSLA '00 : Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 55–56, New York, NY, USA, 2000. ACM Press.

[Packard, 2005] Hewlett PACKARD. Universal service processor (usp). [En ligne]. Adresse URL : http ://h71000.www7.hp.com/openvms/products/-ips/usp/usp.html, Page consultée le 8 août 2005.

[Pascal, 1999] Rapicault PASCAL. Vers une meilleure intégration des designs patterns à la conception. Rainbow Project, 1999.

[Pree, 1999] Wolfgang PREE. *Building Application Frameworks : Object-Oriented Foundations of Framework Design*, Chapitre Hot-Spot-Dreiven Framework Dévelopment. John Wiley & Sons, Inc., September 1999.

[Pyarali *et al.*, 1999] I. PYARALI, T. HARRISON, D. SCHMIDT, and T. JORDAN. Proactor – an architectural pattern for demultiplexing and dispatching handlers for asynchronous events. 1999.

[Quintian and Lahire, 2001] L. QUINTIAN and Ph. LAHIRE. Vers une meilleure réutilisation des patrons de conception. Technical Report, Laboratoire I3S (UNSA/CNRS), Sophia-Antipolis, France, janvier 2001.

[Rogers, 1997] Gregory F. ROGERS. *Framework-based software development in C++*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Rumbaugh *et al.*, 2004] James RUMBAUGH, Ivar JACOBSON, and Grady BOOCH. *UML 2.0 - Guide de référence*. CampusPress, 2004.

[Schmidt, 1997] Douglas C. SCHMIDT. *The Handbook of Programming Languages (HPL) : Object Oriented Programming Languages*, volume vol 1, Chapitre Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software. MacMillan Technical Publishing, New York, 1997.

[Schmidt and Huston, 2002] Douglas C. SCHMIDT and Stephen D. HUSTON. *C++ Network Programming : Systematic Reuse with ACE and Frameworks*, volume 2. Addison-Wesley Professional, 2002.

[Schmidt *et al.*, 2000] Douglas C. SCHMIDT, Michael STAL, Hans ROHNERT, and Frank BUSCHMANN. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[Sunyé *et al.*, 2000] Gerson SUNYÉ, Alain Le GUENNEC, and Jean-Marc JÉZÉQUEL. Design patterns application in UML. In E. BERTINO, editor,

Proceedings of ECOOP 2000, volume 1850 of *LNCS*, pages 44–62. Springer, 2000.

[Vlissides, 1998] John VLISSIDES. *Pattern Hatching : Design Patterns Applied*. Software Patterns Series. Addison Wesley Professional, June 1998.

[WindRiver, 1999] WINDRIVER. *VxWorks Programmer's Guide*, 1999.

[Xu *et al.*, 1996] Jie XU, Brian RANDELL, and Avelino F. ZORZO. Implementing software-fault tolerance in c++ and open c++. In Y. MIN and D. TANG, editors, *Proceedings of the 1996 International Workshop on Computer-Aided Design, Test, and Evaluation for Dependability (CADTED '96)*, pages 224–229, Beijing, China, 1996. International Academic Publishers.

Index

الخلاصة

تُعتبر نظم الوقت الحقِيقِي المدمجة شدِيدة التأَثّر بعتَاد بِيئة التطبِيق خَاصّتهَا لذلك فهي غَالبًا مَا تُطوّر من عدم. إِنّ استعمَال أُطر عمل كَائِنيّة التّوجّه في تطوير تطبِيقَات الوقت الحقِيقِي المدمجة يمكن أن يقلّل كثِيرًا من تكلفتهَا و مدّة تطوِيرهَا. من جهة أُخرَى تعتبر نمَاذج التصمِيم حلول مثبتة لمشَاكل مُتوَاترة. إِنَّ استعمَال هذه النّمَاذج في تطوِير أطر العمل من شَأنه أن يزيد في جودة التطبِيقَات المطوّرة. نقترح في هذَا العمل إِطَار عمل كَائِني التّوجه أُسسه نمَاذج تصمِيم الوقت الحـقِيـــقِي.
كلِمَات مفَاتِيح : أطر عمل، كَائِنيّة التّوجّه، نمَاذج التصمِيم ، اَلوقت الحقِيقِي.

Résumé

Les systèmes temps réel embarqués sont très influencés par l'architecture matérielle de leurs environnements d'application, c'est pourquoi ils sont généralement développés à partir de zéro. L'utilisation des frameworks orientés objet dans le développement des applications temps réel peut réduire énormément le coût et la duré de développement de ces applications. D'autre part les patrons de conception sont des solutions prouvées à des problèmes récurrents. Leur utilisation pour la conception et le développement des frameworks augmente la qualité des applications développées. Dans ce travail nous proposons un framework orienté objet fondé sur les patrons de conception temps réel dont.
Mots clés : framework, orienté objet, patrons de conception, temps réel.

Abstract

Real-time embedded systems are very influenced by the hardware architecture of their application environment, that's why they are often developed from scratch. The use of object oriented framework in the development of real-time embedded application can remedy the situation and reduce the development time and cost. Moreover, design pattern are proven solutions to recurrent problems. Their use for the design and the development of frameworks increases the quality of developed applications. In this work we propose a real-time design pattern-based object oriented framework.
Keywords : framework, object oriented, design patterns, real-time.