Projet de diplôme

LAMS - Laboratoire de modélisation systémique

Specifying Design Patterns in OOAD

Rapport Technique
IC/2002/039

Auteur : **Florian Bois**
Assistant : **Pavel Balabko**
Professeur : **Alain Wegmann**

SOMMAIRE

1 Introduction ... 4
1.1 Qu’est ce qu’un design pattern ? ... 4
1.2 Avantages des design patterns ... 4
1.3 Inconvénients des design patterns ... 4
1.4 Objectifs du projet ... 4
1.5 Résumé ... 4
1.6 Plan du rapport ... 5
2 Etat de l’art ... 6
2.1 Introduction ... 6
2.2 Design pattern selon GoF ... 6
2.2.1 Présentation ... 6
2.2.2 Description des patterns ... 6
2.2.3 Classifications ... 7
2.2.4 Liste des patterns référencés ... 8
2.2.5 Un Exemple [Decorator Pattern] ... 9
2.2.6 Qualités et défauts ... 10
2.2.7 Conclusion ... 11
2.3 Business pattern selon Martin Fowler ... 11
2.3.1 Présentation ... 11
2.3.2 Description des patterns ... 11
2.3.3 Classification ... 12
2.3.4 Liste des patterns référencés ... 12
2.3.5 Exemple [Accountability Pattern] ... 14
2.3.6 Qualités et défauts ... 15
2.3.7 Conclusion ... 16
2.4 Business Modeling with UML de Hans-Erik Erikson et Magnus Penker ... 16
2.4.1 Présentation ... 16
2.4.2 Description d’un pattern ... 16
2.4.3 Classification ... 16
2.4.4 Liste des patterns référencés ... 17
2.4.5 Exemple [Business Event-Result history] ... 18
2.4.6 Qualités, défauts et conclusion ... 19
3 Spécification de design patterns ... 20
3.1 Introduction ... 20
3.2 Formalisation avec LePus ... 20
3.2.1 Qu’est ce que le LePUS ? ... 20
3.2.2 Description du pattern « decorator » ... 21
3.2.3 Représentation littérale du pattern « decorator » ... 22
3.2.4 Comparaison avec le modèle UML, Avantages et inconvénients ... 23
3.2.5 Objectifs de LePUS ... 23
3.2.6 Conclusion ... 24
3.3 Problématique actuelle ... 25
3.3.1 Présentation de la problématique ... 25
3.3.2 Exemple du décorateur ... 26
3.3.3 Exemple du « business event-result history pattern » ... 27
3.4 Formalisation complémentaire au modèle existant ... 28
3.4.1 Présentation ... 28
3.4.2 Exemple du décorateur ... 29
3.3.3 Exemple du « business event-result history pattern » ... 31
3.5 Proposition d’un autre type modélisation ... 32
3.5.1 Description ... 32
3.5.2 Exemple du décorateur. ... 33
3.5.3 Exemple du « business event-result history pattern » ... 39

3.5.4 Applicabilité de la méthode ... 47
3.6 Modèles complémentaires à notre proposition ... 48
3.6.1 Exemple du décorateur ... 48
3.6.2 Exemple du « business event-result history pattern» ... 49
3.7 Comparaison, classification et conclusion [à compléter] ... 50
4 Intégration des design patterns dans les cases tools ... 52
4.1 Intégration des patterns dans TogetherJ ... 52
4.1.1 Présentation de TogetherJ ... 52
4.1.2 Quelles sont les différents patterns disponibles ... 52
4.1.3 Comment les patterns sont-ils présentés et intégrés ? ... 53
4.1.4 Exemple du décorateur : ... 56
4.1.5 Conclusion ... 59
4.2 Proposition d'un case tool idéal ... 59
5 Conclusion ... 60

TABLE DES FIGURES

Figure 1 : Diagramme de classe du pattern "decorator" ... 10
Figure 2 : Diagramme de classe du pattern Accountability ... 15
Figure 3 : Diagramme de classe du pattern "business event-result history" ... 19
Figure 4 : Le pattern "decorator" in the graphical representation of LePUS ... 21
Figure 5 : Représentation littérale du pattern "décorateur" selon LePus ... 22
Figure 6 : Comparaison entre Le décorateur selon LePUS et le décorateur selon UML ... 23
Figure 7 : Diagramme de classe du décorateur ... 26
Figure 8 : Diagramme de classe du pattern "business event-result history" [tiré du livre] ... 27
Figure 9 : Vue en bloc du décorateur ... 29
Figure 10 : Diagramme de collaboration UML du décorateur ... 30
Figure 11 : Sequence diagram montrant plusieurs applications du pattern « business event ...» 31
Figure 12 : Invocation simple ... 33
Figure 13 : Simple décoration ... 34
Figure 14 : Synthèse de deux décorateurs ... 35
Figure 15 : Resultat de la synthèse de deux décorateurs ... 37
Figure 16 : Exemple concret de décorations ... 38
Figure 17 : Business event-result history pattern ... 39
Figure 18 : Structure pour l'enregistrement d'un business event" ... 40
Figure 19 : Explication d'un contrat ... 41
Figure 20 : Application du pattern Business event-result history ... 42
Figure 21 : Deuxieme application du pattern business event-result history ... 43
Figure 22 : Business event-result history , synthèse de deux patterns ... 44
Figure 23 : Business event-result history, résultat de la synthèse ... 46
Figure 24 : Role modeling de plusieurs applications du décorator ... 48
Figure 25 : Role modeling de plusieurs applications "business event-result history pattern" ... 49
Figure 26 : Tableau comparatif des différentes modélisations ... 50
Figure 27 : Comparatifs des différentes modélisations d'éléments semblables ... 51

1 Introduction

1.1 Qu'est ce qu'un design pattern ?

Un design pattern est un élément de logiciel orienté objet réutilisable présentant une solution générique à des problèmes récurrents.
A la différence d'un framework[1], le design pattern[2] n'apporte aucune solution globale à un problème général. Il se cantonne à des problèmes de plus petites tailles plus concrets.
Il s'adresse donc aux designers afin de leur permettre de modulariser, réutiliser, et structurer leur conception.

1.2 Avantages des design patterns

Les design pattern permettent donc d'apporter une solution éprouvée, efficace et réutilisable et forcent les concepteurs à proposer une solution modulaire.
Ils permettent un gain de temps considérable ; Par exemple, il n'est pas nécessaire de gaspiller son énergie à savoir si telle ou telle solution est la meilleure alors que le problème a déjà été résolu.
Il en résulte une meilleure compréhension du design complet du problème, rendant les discussions entre concepteurs plus aisées.

1.3 Inconvénients des design patterns

Il est cependant vrai que tout n'est pas facile pour autant : un pattern de son choix ne peut en effet être implanté aveuglément dans son design. Les patterns sont pour la plupart volontairement abstraits, donc très peu formalisés, ce qui leur permet par ailleurs d'être au maximum ouverts à tous les types de problèmes.
Un effort supplémentaire est donc nécessaire pour intégrer au mieux notre pattern au problème rencontré. Ce qui suppose une très bonne connaissance de ce qui est utilisé.
Les problèmes rencontrés sont très hétéroclites, au même titre que les design pattern. Ils peuvent être très formalisables, alors que d'autres le sont beaucoup moins. Inversement, des formalismes sont adaptés à des types de pattern correspondants.

1.4 Objectifs du projet

L'analyse des différentes caractéristiques (qualités et défauts) des design pattern met en évidence un net manque de formalisation. Il convient néanmoins de veiller à ne pas perdre l'essence [2-1] du pattern afin de ne pas dévier vers des solutions non adaptables.
Ce projet s'attachera donc à proposer des solutions concrètes pour permettre aux concepteurs d'utiliser au mieux les capacités des patterns tout en limitant le temps d'intégration.

Objectifs :

- Apporter une amorce de solution aux défauts rencontrés dans l'utilisation des design patterns à travers un formalisme qui décrit aussi bien la structure que le comportement.
- Trouver une bonne formalisation tout en restant le plus proche possible de la forme originale du pattern.

1.5 Résumé

Les patterns sont :
réutilisables, modulaires, adaptables, variés, petits, …
Mais :
Peu formalisés, abstraits, peu clairs, …

1.6 Plan du rapport

Dans ce projet, nous allons tout d'abord étudier la littérature en matière de design pattern ou de business pattern, puis nous apporterons notre solution aux problèmes rencontrés, et enfin, nous étudierons sa possible efficacité à l'intérieur d'un outils de développement.

2 Etat de l'art

2.1 Introduction

Dans ce chapitre, nous allons étudier ce qu'il existe en matière de design pattern.

Nous nous sommes tout d'abord basés sur la référence en la matière, c'est à dire le livre de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (usuellement appelé GoF pour Gang of Four) intitulé « Design Patterns, elements of reusable Object-Oriented software » [2].

Par la suite, nous nous sommes penchés sur les business pattern à travers « Analysis Patterns, reusable Object Models » [3] de « Martin Fowler». et « Business modeling with UML, Business pattern at work » [4] de Hand-Erik Eriksson et Magnus Penker.

Enfin, par l'intermédiaire des multiples références du Web sur le sujet, nous avons étudié ce qu'il existait en matière de formalisation de patterns.

2.2 Design pattern selon GoF

2.2.1 Présentation

« Design Patterns, Elements of reusable Object-Oriented software » [2] est la référence en ce qui concerne les software patterns, il contient un catalogue détaillé de 23 solutions différentes groupées en 3 catégories (design créationnel, structurel et comportemental).

Ce livre se compose de 3 parties :

- La première présente les notions de programmation objets et introduit les design patterns.
- La seconde décrit d'un point de vue global un exemple complet de logiciel conçu à l'aide de pattern.
- La troisième, la plus importante, regroupe dans un catalogue un large panel de pattern couvrant un maximum de problèmes usuelles.

2.2.2 Description des patterns

Concrètement, un pattern possède quatre éléments essentiels :

- Le nom
- Le problème
- La solution
- Les conséquences

De manière plus détaillée, chacun de ces éléments à été subdivisé en sous-paragraphes. Ainsi, nous ressortons les 13 points essentiels permettant de présenter les différents aspects des patterns présents dans le livre :

- *Nom du pattern et classification :*
  Un nom bien approprié est indispensable pour permettre la meilleure compréhension possible de la solution proposée. Ce choix est primordial, car ce nom entrera dans le vocabulaire du concepteur.

- *Objectif :*
  Une phrase (courte et précise) définit la fonction du pattern et le problème traité.

- *Connus aussi sous le nom de :*
  Certains patterns sont connus sous différents noms.

- *Motivation :*
  Un scénario illustrant le problème rencontré peut être résolu à l'aide de la structure proposée. Ceci permet notamment de mieux comprendre les descriptions les plus abstraites.

- *Application :*
  Dans quelle situation le pattern peut-il être appliqué au mieux ?

- *Structure :*
  Une représentation graphique suivant la norme OMT de la structure de la solution.

- *Participants :*
  Les classes et les objets mis en relation dans la structure de classe ainsi que leurs responsabilités individuelles (sous forme littérale).

- *Collaborations :*
  De quelle manière les participants collaborent-ils entre eux (sous forme littérale) ?

- *Conséquences :*
  De quelle manière le pattern en question réussit-il ces objectifs, y-a-t-il entre eux des compromis ?

- *Implémentation :*
  Quelles techniques, outils et astuces sont utilisées pour introduire le pattern dans une conception ?

- *Exemple de code :*
  Quelques fragments de code C++ permettant de démarrer avec un squelette prédéfini.

- *Utilisation connue :*
  Des exemples de situations où le pattern est particulièrement adapté.

- *Pattern en relation :*
  D'autres patterns possédant des similitudes avec celui étudié.

2.2.3 Classifications

Nous pouvons classifier les patterns selon 2 échelles :

-Leurs types :

Patterns créationnels	Patterns structurels	Patterns comportementaux

-Leurs qualités d'interprétations :

Précise	Alternative possible	Généralisation précise	Ouvert a certaines variations	Informel, description théologique	Délibérément sans détails

2.2.4 Liste des patterns référencés

Patterns créationnels :

Abstract factory :

Fournir une interface pour la création de famille d'objets sans spécifier les classes concrètes.

Builder :

Séparer la construction d'un objet complexe de sa représentation. Le même procédé de construction peut donc créer différentes représentations.

Factory Method :

Définir une interface pour la création d'objet, en laissant les sous classes décider quelle classe instancier. (Laisser le travail d'instanciation aux sous classes)

Prototype :

Spécifier le type d'objet à créer en utilisant une instance d'un prototype, et créer de nouveaux objets en copiant ce prototype.

Singleton :

S'assurer qu'une classe n'ait qu'une instance et lui fournir un point d'accès global.

Patterns structurels :

Adapter :

Convertir l'interface d'une classe dans une autre interface compatible avec d'autres clients.

Bridge :

Séparer une abstraction de son implémentation dans le but que chacune puisse être modifiée indépendamment.

Composite :

Composer des objets en structure d'arbre hiérarchisé.

Decorator :

Ajouter dynamiquement des responsabilités à un objet.

Facade :

Proposer une interface commune à un ensemble d'interfaces d'un sous système (améliorer l'utilisation d'un sous système).

Flyweight :

Utiliser le partage afin de supporter un large panel d'objets bas niveau.

Proxy :

Proposer un clone ou un point d'entrée à un objet pour contrôler son accès.

Patterns comportementaux :

Chain of responsability :

Permettre d'associer l'émetteur et le récepteur d'une requête en donnant à plus d'un objet la chance de manipuler la requête (enchaîner des objets et parcourir cette chaîne avec la requête jusqu'à trouver quelqu'un capable de la manipuler)

Command :

Encapsuler une requête dans un objet.

Interpreter :

Proposer un langage, définir une représentation pour sa grammaire et fournir un interprète capable de manipuler cette grammaire.

Iterator :

Proposer séquentiellement une façon d'accéder aux éléments d'un objet.

Mediator :

Définir un objet qui encapsule la façon dont interagit un autre ensemble d'objets.

Memento :

Sans toucher à l'encapsulation, capturer et extraire l'état d'un objet de manière à ce que cet objet puisse retrouver son état initial.

Observer :

Proposer une dépendance entre objets de manière à mettre à jour automatiquement toutes ces dépendances lors du changement d'état d'un objet.

State :

Permettre à un objet de modifier son comportement lorsque son état interne change également (changement de classe).

Strategy :

Définir une famille d'algorithmes, l'encapsuler, la rendre interchangeable, et permettre à chaque algorithme de varier indépendamment du client qui l'utilise.

Template method :

Définir le squelette d'un algorithme en une opération et allouer certaines parties aux sous classes (les sous classes redéfinissent certaines parties de l'algorithme sans changer la structure principale)

Visitor :

Permettre de définir une nouvelle opération sans changement de classe des éléments utilisés.

2.2.5 Un Exemple [Decorator Pattern]

La forme littérale in-extenso du livre n'est pas introduite ci-après, mais seule une version plus concise avec les points essentiels du pattern.

Forme littérale simplifiée:

Nom :
Decorator (décorateur)

Problème :
Comment ajouter un (ou plusieurs) comportement(s) spécifique(s) à un objet donné tout en donnant la possibilité d'activer ou non ces nouvelles fonctionnalités ?

Solution :
Un décorateur est un objet qui se substitue à l'objet initial, avec la même interface. Il est par ailleurs transparent pour l'utilisateur. Chaque décorateur possède une référence vers un composant (un autre décorateur ou l'objet initial), permettant d'enchaîner les décorations et de transmettre tous les appels de méthodes à travers les différents décorateurs jusqu'à l'objet initial.

Conséquences :
L'objet devient flexible, permettant en temps réel d'ajouter ou de supprimer à sa guise de nouveaux comportements, de les désactiver puis les réactiver. Alors qu'une hiérarchie d'héritage engendrerait la création d'une multitude de classes correspondant à toutes les combinaisons possibles de décoration, un comportement s'intégrant dans ceux existants (en ajoutant une simple classe correspondant à la nouvelle décoration) peut être ajouté. un seul objet peut également être décoré plusieurs fois.

Déléguer les décorations à des plus petites classes annexes *évitent la gestion de classes trop lourdes*. Ainsi, à la place d'un seul objet complexe, nous disposons d'un objet simple auquel sont rajoutés des fonctionnalités simples par l'incrémentation de petits objets.

Un décorateur est différent d'un composant, il ne possède pas la compétence fondamentale dudit composant, mais joue son rôle en déléguant au composant la responsabilité d'exécution du travail. Il est cependant transparent

On se retrouve donc avec une multitude de petits objets, possédant tous l'interface de l'objet complexe, mais n'ayant que les capacités d'embellir l'objet principal.

Diagramme de classe :

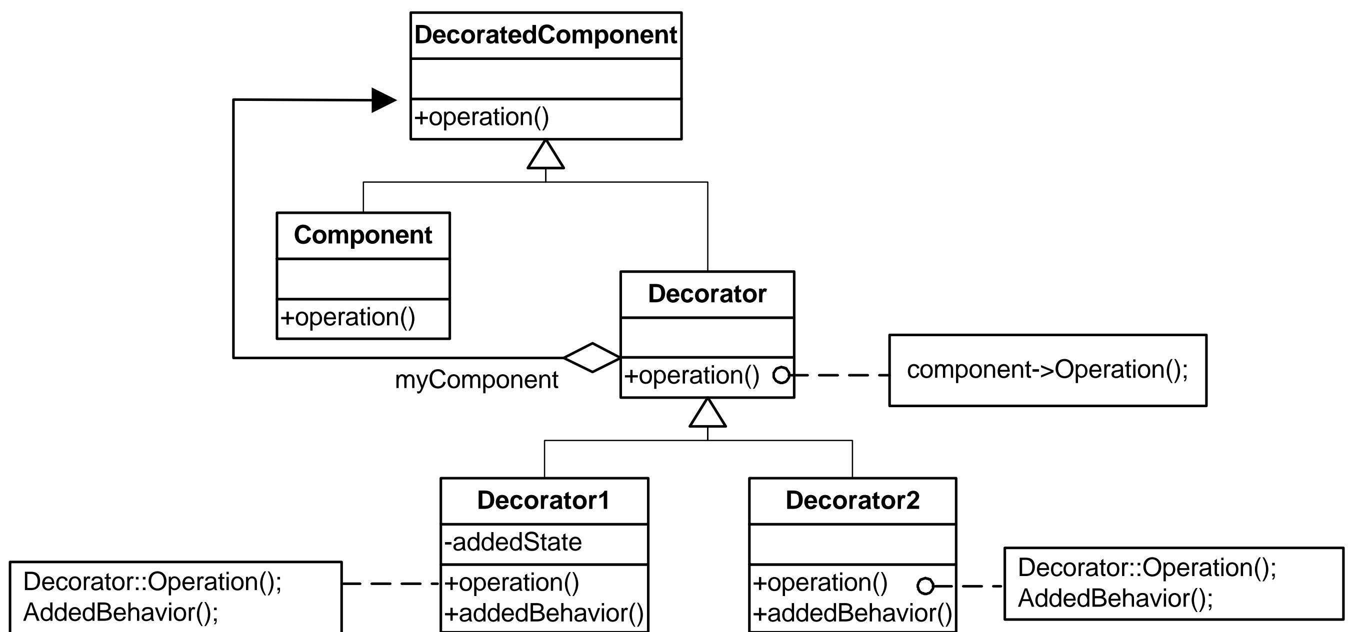


Figure 1 : Diagramme de classe du pattern "decorator"

2.2.6 Qualités et défauts

Qualités :

- une description abstraite qui laisse le concepteur libre d'interpréter les solutions proposées comme bon lui semble.
- une description littérale très détaillée qui permet de bien comprendre l'essence du pattern.

Défauts :

- le manque d'une description du comportement ne permet pas une compréhension instantanée.
- La structure des données en elle-même ne décrit pas tout, d'autant plus que certains patterns ont un comportement assez complexe.
- Impossibilité dans l'état des choses d'intégrer les patterns tel quel dans un environnement de développement.

2.2.7 Conclusion

« Design pattern » est la référence en matière de pattern. Il s'est notamment imposé grâce à son catalogue complet et détaillé. Cette notoriété lui confère un avantage considérable, puisque que chaque designer peut, en faisant référence à un pattern de ce livre , être certain que l'ensemble de la communauté du développement logiciel le comprendra.

Cependant, les modélisations existantes ne suffisent pas. En effet, la description du comportement reste très abstraite et noyée dans les explications littérales, malgré une structure bien présentée à travers les diagrammes de classes. Il est par conséquent difficile de saisir l'intégralité des finesses de chaque pattern.

Il convient donc d'espérer une mise à jour de ce livre qui intégrerait les dernières spécifications d'UML :
- Collaboration,
- use case
- Activité …

2.3 Business pattern selon Martin Fowler

2.3.1 Présentation

« Analysis Pattern » traite différents types de problèmes ayant trait à
- la gestion,
- la finance,
- et au système permettant de les mettre en œuvre.

A l'instar du livre du GoF, on retrouve un catalogue complet de patterns, divisé en 2 parties majeures :

- Les « Analysis patterns » qui présentent des solutions dans les domaines tels que la comptabilité, les mesures, les observations, les inventaires, la planification, les transactions, les contrats …
- Les « Supports patterns » qui eux font plutôt référence aux systèmes qu'utilisent les « Analysis patterns ».

2.3.2 Description des patterns

Les patterns sont décrits de manière très peu formelle et peu structurée. Sont présentés :

- Le nom du pattern
- Une explication détaillée permettant d'appréhender le problème
- Un modèle décrivant la structure des données (compatible avec le modèle Entité/Association des bases de données)
- Des exemples concrets
- quelques diagrammes d'activité

Ce livre présente rarement un pattern isolé, mais plutôt des familles. A partir d'un problème trivial, et suite à plusieurs adaptations et généralisations, nous aboutissons à des solutions plus complexes. Il est donc important lors de la lecture, de ne pas se focaliser sur un seul exemple, mais d'avoir une vision plus globale du chapitre en question ; il en va de la bonne compréhension des patterns.

2.3.3 Classification

Pour des raisons de clarté et de compréhension, les noms et types des patterns sont laissés en anglais.

2 classifications différentes correspondant aux 2 types de patterns présentés existent :

- *Les « Analysis Patterns »*

Accountability	Observations and Measurements	Observations for corporate finance	Referring to objects	Inventory and accounting
Using the accounting éodels	Planning	Trading	Derivative contarcts	Trading packages

- *Les « Support Patterns »*

Layered Architecture for information systems	Application Facade	Patterns for type model	Association Pattern

2.3.4 Liste des patterns référencés

Analysis Patterns
Accountability
- Party
- Organization Hierarchies
- Organization structure
- Accountability
- Accountability Knowlegde level
- Party type generalization
- Hierarchic Accountability
- Operating Scopes
- Post

Observations and Measurements :
- Quantity
- Conversion Ratio
- Compound Units
- Measurement
- Observation
- Subtyping Observation concepts
- Protocol
- Dual time record

Observations for corporate finance :

- Enterprise Segment
- Measurement Protocol
- Range
- Phenomenon with range
- Using the resulting framework

Referring to objects :

- Name
- Identification scheme
- Object merge
- Object equivalence

Inventory and Accounting :

- Account
- Transaction
- Summary Account
- Memo account
- Posting rules
- Individual instance method
- Posting rule execution
- Posting rules for many Accounts
- Choosing entries
- Accounting Practice
- Source of an entry
- Balance sheet and income statement
- Corresponding account
- Specialized account model
- Booking entries to multiple accounts

Using the accounting models :

- Structural models
- Implementing the structure
- Setting up new phone services
- Setting up calls
- Implementing account-based firing
- Separating calls into day and evening
- Charging for time
- Calculating the tax
- Concluding thoughts

Planning :

- Proposed and implementing action
- Completed and abandoned actions
- Suspension
- Plan
- Protocol
- Resource allocation
- Outcome and start functions

Trading :

- Contract
- Portfolio
- Quote
- Scenario

Derivative contracts :
- Forward contracts
- Options
- Products
- Subtype state machine
- Parrallel application and domain Hierarchies

Trading packages :
- Multiple access levels to a package
- Mutual visiblity
- Subtyping Packages
- Concluding thoughts

Support patterns :

Layered architecture for information systems :
- Two-tiers Architecture
- Three-tiers Architecture
- Presentation and Application Logic
- Database Interaction
- Concluding Thoughts

Applications facades :
- A health care example
- Contents of a facade
- Common Methods
- Operation
- Type Conversation
- Multiple Facades

Patterns for type model, design templates :
- Implementing Associations
- Implementing Generalization
- Object creation
- Object destruction
- Entry point
- Implementing Constraints
- Design Templates for others techniques

Association patterns :
- Associative type
- Keyed mapping
- Historic mapping

2.3.5 Exemple [Accountability Pattern]

Forme littérale simplifiée:

Traitant de la responsabilité, ce pattern est une généralisation du pattern « Organization structure » relatif à la structure organisationnelle d'une entreprise. Ainsi, un directeur peut être responsable de plusieurs chef de projets eux même responsables de plusieurs développeurs, à l'instar d'une entreprise possèdant plusieurs unités qui elles même possèdent plusieurs divisions.

Nom :

Accountability (Responsabilité)

Problème :

Comment gérer les responsabilités relatives à chaque partie (personne, groupe de personnes, laboratoires …) d'une organisation ?

Solution :

Une responsabilité est composée :

- d'une tâche.
- de deux parties :
    - le responsable, gestionnaire de la tâche à exécuter.
    - Le subordonné, celui qui l'exécute.
- D'un période de temps relatif à la durée de la tâche

Conséquences :

- assigner des tâches à chaque personne suivant une durée précise.
- le responsable d'un subordonné peut devenir le subordonné d'un autre responsable.
- retrouver toutes les tâches et toutes les responsabilité d'une personne.
- gérer au mieux la structure hiérarchique pour éviter :
    - de trop déléguer les tâches.
    - de ne plus savoir à qui s'adresser en cas de problème.

Structure de classe :

Pour permettre une meilleure compréhension du problème, la version UML des diagrammes présents dans le livre est fournie ci-après.

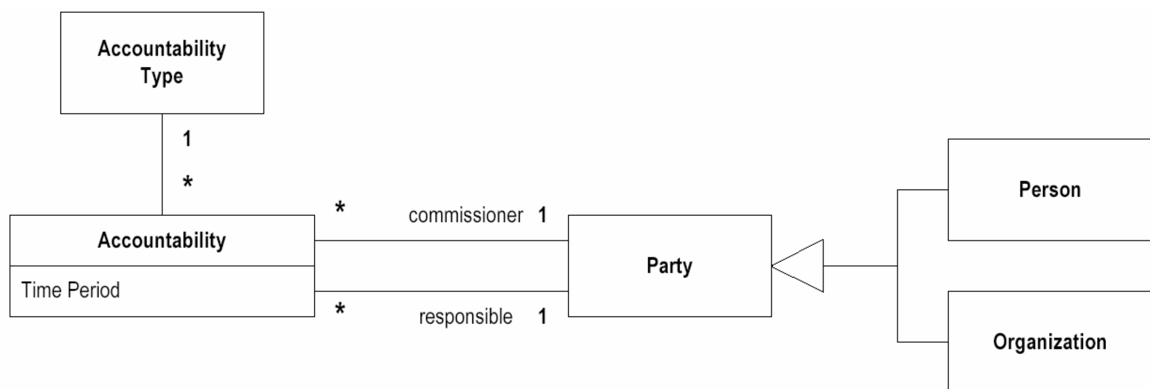


Figure 2 : Diagramme de classe du pattern Accountability

2.3.6 Qualités et défauts

Comme dans l'œuvre du GoF, le principal atout de cet ouvrage réside dans la variété et la multiplicité des patterns proposés. Ce catalogue étant directement basé sur l'expérience de l'auteur, on y retrouve des problèmes très concrets, agrémentés d'exemples facilitant grandement la compréhension.

Bien qu'aucun exemple de code ne soit fourni, les patterns présents dans ce livre sont assez clairs. Ils permettent une implémentation relativement rapide, aussi bien au niveau du software que de la base de données.

Si « Analysis patterns » a les même qualités que le livre du GoF, il présente également les même défauts. En effet, aucune description du comportement n'est fournie, malgré des modèles de la structure très détaillés, complétés d'exemples bien choisis. Par ailleurs, les descriptions littérales abstraites n'apportent rien.

2.3.7 Conclusion

Moins bien structuré que « design pattern » , « Analysis Pattern » en conserve les caractéristiques, c'est à dire une grande variété de problème traités, mais présente une faiblesse dans la modélisation du comportement. Ainsi, les patterns sont très statiques, leur comportement abstrait voire inexistant.

Une mise à jour ou l'auteur suivrait les spécifications UML en incluant les diagrammes de collaborations, de « use case » ou d'activité s'impose (cf. http://www.martinfowler.com pour de plus amples informations).

2.4 Business Modeling with UML de Hans-Erik Erikson et Magnus Penker

2.4.1 Présentation

« Business modeling in UML » aborde les mêmes thèmes que l'œuvre de Martin Fowler (exception faite de la partie d'analyse financière) ,tout en intégrant une modélisation UML. Deux types de patterns ressortent : à dominante soit comportementale, soit structurelle.

Le livre se scinde en trois parties :
- La première reprend les bases d'UML, et leur applicabilité au domaine du business. (en terme de structure, de comportement, de process …)
- La deuxième présente 26 patterns différents classés en 3 thèmes.
- Un exemple de « business model »

2.4.2 Description d'un pattern

La présentation est quasiment identique à celle du livre du GoF (cf. paragraphe 2.2.2).

- *Nom du pattern*
- *But*
- *Motivation*
- *Application*
- *Structure*
- *Participants*
- *Collaborations*
- *Conséquences*
- *Exemples*
- *Pattern en relation:*

2.4.3 Classification

Les patterns sont répertoriés en 3 parties :

Ressource & Rule Patterns	Goal patterns	Process Patterns

2.4.4 Liste des patterns référencés

Ressource & Rule Patterns :

Actor-Role pattern :

Proposer des règles afin d'utiliser des acteurs et le concept de rôle. (comment sont-ils combinés et différenciés ?)

Business Definitions pattern :

Capturer et organiser les termes d'un business , puis créer un système capable de les manipuler.

Business event-result history pattern :

Suivre des événements de type business, les connecter pour analyser les résultats.

Contract pattern :

Proposer des règles pour modéliser les points clés d'un contrat

Core-represention pattern :

Structurer les points clés d'un problème

Document pattern :

Proposer une structure capable de restituer et de présenter tout les composants d'un document

Employement pattern :

Modéliser la relation entre une personne et son organisation.

Geographic location pattern :

Modélisation d'adresses et d'endroits sur une longue période de temps.

Organization & party pattern :

Créer une structure organisationnelle flexible et qualitative.

Product data management pattern :

Capturer et organiser les relations entre documents et produits.

Thing-information pattern :

Eliminer le « focus shifting » apparaissant lors d'une modélisation.

Title-Item pattern :

Séparer le titre du corps de l'information

Type-Object-Value pattern :

Modéliser des relations entre le type de l'objet , l'objet lui-même et sa valeur.

Goal patterns :

Business goal allocation pattern :

Assigner un but à un business spécifique afin de faciliter la description et la validation.

Business goal decomposition pattern :

Améliorer le « goal modeling » en hiérarchisant les buts.

Business goal-Problem pattern :

Identifier les connexions entre « business goals » et leurs problèmes afin de corriger ces problèmes et d'accomplir les buts.

Process patterns :

Basic process structure pattern :

Montrer comment est créé un « business process ».

Process interaction pattern :

Montrer comment créer et organiser les interactions existantes entre différents « business process »

<u>Process feedback pattern :</u>

Evaluer les résultats d'un « business process » pour l'ajuster lors de sa prochaine utilisation.

<u>Time-to-customer pattern :</u>

Raccourcir le temps entre la demande d'un client et sa livraison.

<u>Process layer supply pattern :</u>

Organiser la structure complexe d'une organisation en « business process »

<u>Process layer control pattern :</u>

Aider à structurer de complexes businesses afin de pouvoir mieux les comprendre et les réutiliser

<u>Action workflow pattern :</u>

Proposer un outil capable d'analyser la communication entre plusieurs parties (client – fournisseur par exemple)

<u>Process-Process instance pattern :</u>

Clarifier la distinction entre un « process » et son instance.

<u>Resource use pattern :</u>

Structurer les ressources utilisées dans un « process » afin de modéliser et d'implémenter dans un système d'information.

2.4.5 Exemple [Business Event-Result history]

La forme littérale complète du livre n'est pas présentée in-extenso ci-après, mais une version plus concise qui expose les points essentiels du pattern.

Forme littérale simplifiée :

Nom :
Business Event-Result History (Historique des événements relatifs aux affaires financières)

Problèmes :
Dans le monde des affaires, comment suivre et enregistrer des événements et leurs conséquences lors d'une transaction entre plusieurs entités.

Solution :
Dans notre cadre, un événement (« business event ») représente une seule transaction concernant un ou plusieurs produits (« product ») . Il doit tenir compte de de plusieurs partis, dont notamment le demandeur et le fournisseur, et, le cas échéant, des intermédiaires.

Ces parties s'entendent et définissent les termes de la transaction par le biais d'un contrat (« contract »). Celui-ci sera officialisé par un rapport (« statement »).

En résumé, l'événement est matérialisé par un rapport qui sert de support à la définition de la transaction entre les différentes parties.

Conséquences :
Cette structure permet de suivre et d'enregistrer l'historique des transactions entre 2 ou plusieurs acteurs. Elle est extensible et permet d'ajouter autant d'événements que souhaité.

Par la suite, ces enregistrements formeront une base de donnés permettant d'analyser et de prévoir les besoins. Cependant, dans ce cadre la, il faudra veiller à ne pas « polluer » cette base de donnés d'informations de trop « bas niveau ». Sinon, ces analyses risqueraient d'être trop complexes à déchiffrer.

Structure de classe :

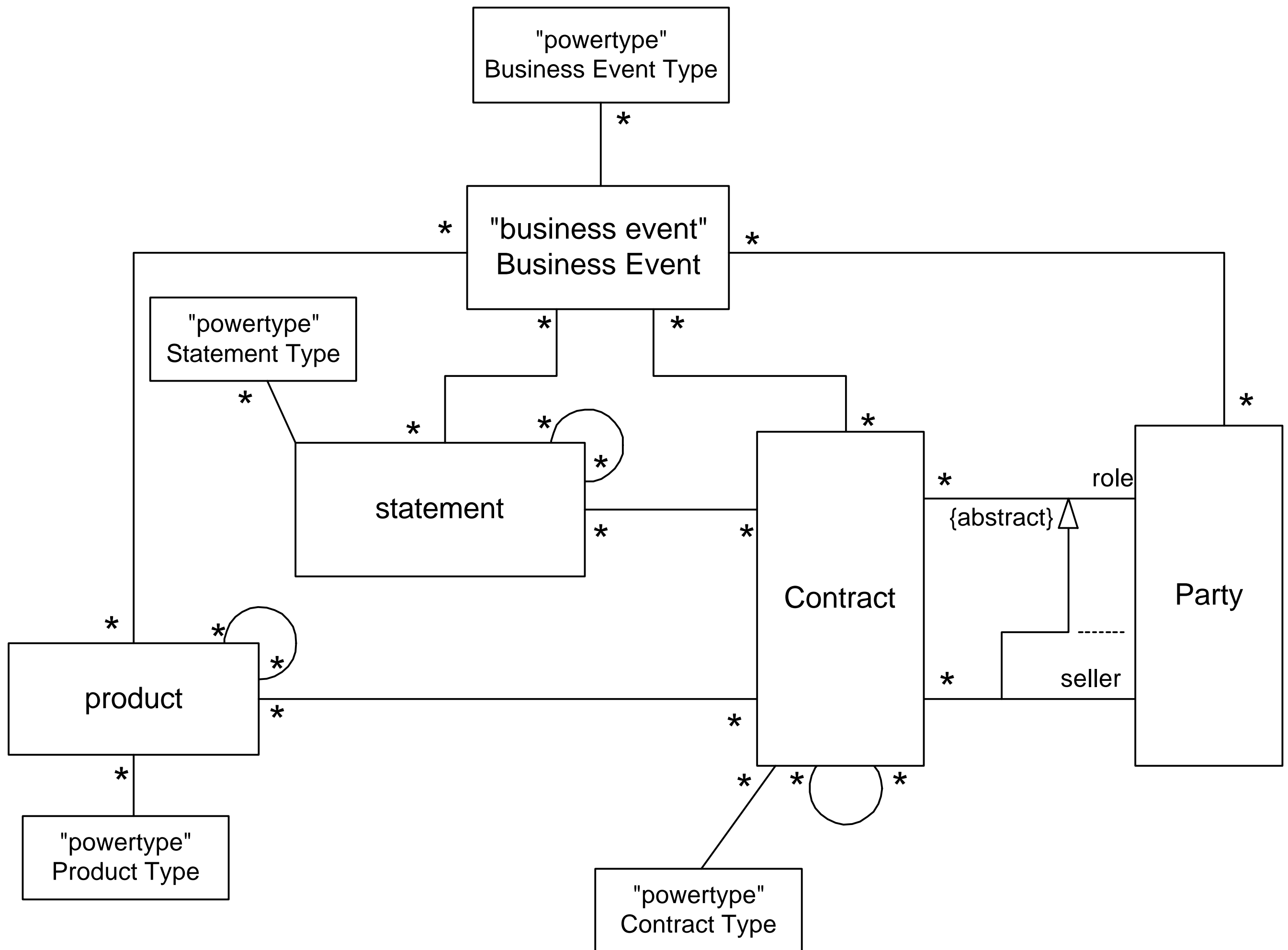


Figure 3 : Diagramme de classe du pattern "business event-result history"

2.4.6 Qualités, défauts et conclusion

Deux faiblesses paraissent toutefois devoir être relevées dans cet ouvrage :
- Au niveau des patterns orientés structure, un manque récurrent de formalisation du comportement,
- Au niveau des patterns orientés comportement, une faiblesse des descriptions statiques.

Cela paraît d'autant plus déplorable que comportement et structure sont étroitement liés, qu'il n'existe aucun pattern sans comportement ou sans structure de classe ; ces 2 types de formalisations sont donc indispensables à la compréhension.

Ainsi, ce livre est bien plus abouti que les 2 précédents, il s'exprime dans un langage familier (UML). Et malgré les lacunes mentionnées, l'ensemble est facilement abordable.

3 Spécification de design patterns

Pour mieux décrire et par conséquent améliorer la compréhension des patterns, il convient de proposer plusieurs vues différentes : tout ceci dans le but final d'arriver vers un formalisme intégrable dans un case tool tel que Together.

3.1 Introduction

Dans ce chapitre, nous proposons un formalisme différents de ce qui existe déjà. Associé a plusieurs modèles complémentaires. Tandis que certains montrent des choses plus complètes, d'autres offrent une vue complètement nouvelle.

Dans la mesure du possible et de la clarté offerte par chaque situation, nous avons essayé d'être le plus proche possible d'UML. Néanmoins, et comme le décrit Jezequel dans son document intitulé « Precise Modelling of Design Patterns », UML n'est pas toujours approprié.

Par souci de clarté, les diagrammes présentés se réfèrent aux exemples déjà étudiés dans les chapitres antérieurs ; il s'agit du design pattern « décorateur » et du business pattern « Business event-result history ».

Il convient tout d'abord de synthétiser et résumer la problématique actuelle concernant la formalisation des designs patterns (en s'appuyant notamment sur l'exemple du langage LePUS [12]). Ensuite, notre approche se penchera sur des modèles complémentaires au diagramme de classe du GoF. Puis, une description complète et détaillée d'une modélisation jamais utilisée dans le domaine des designs patterns sera effectuée. Cette modélisation sera complétée par d'autres modèles de plus haut niveau. Enfin, nous procéderons à la comparaison de tous ces formalismes, ainsi qu'à l'analyse de leurs avantages et inconvénients.

3.2 Formalisation avec LePus

LePus est un langage formel très complet permettant notamment la description de pattern ; il ne fera pas l'objet dans la suite d'une description détaillée mais d'une analyse à travers l'exemple du décorateur.

3.2.1 Qu'est ce que le LePUS ?

LePUS est un langage structuré permettant la modélisation de programmes informatiques et plus spécifiquement de designs patterns.
A la différence d'UML, LePUS permet de modéliser de manière plus détaillée le comportement et l'interaction entre différentes entités d'un modèle.
Il se présente sous 2 formes différentes. L'une littérale à base de prédicats et de fonctions, l'autre graphique retranscrivant de manière plus visuel la forme littérale.

3.2.2 Description du pattern « decorator »

Dans leur Article « LePUS –A Declarative Pattern Specification Language » [12], Amnon H. Eden,Yoram Hirshfeld et Amiram Yehudai présentent leur version du pattern « decorator » :

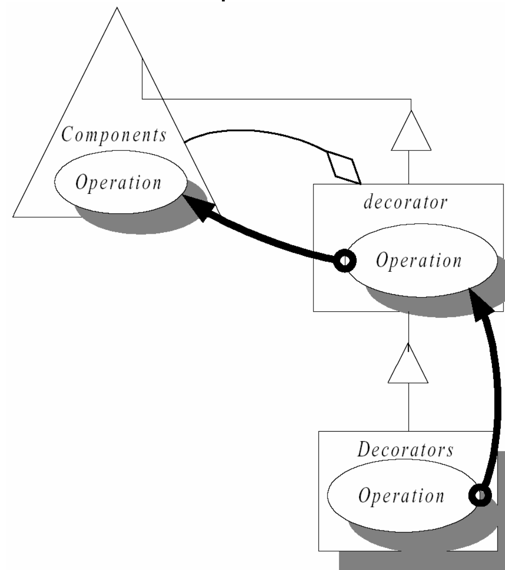


Figure 4 : Le pattern "decorator" in the graphical representation of LePUS

a. *description des classes*

Les classes de la figure 4, représentées par un triangle (hierarchy of inheritance), un rectangle (single class) et un rectangle ombré (multiple classes) fournissent les informations suivantes :

Components désigne une hiérarchie de Généralisation / Spécialisation, qui correspond, sur la base d'une classe abstraite, à un nombre non spécifié de classes dérivées qui implémentent l'interface de celle-ci. Chacune de ces classes dérivées possède un jeu de méthodes appelé *Operation*

La classe **decorator** est une classe abstraite dérivée elle-même de la classe abstraite **Components**. Elle sert d'interface au groupe de classes **Decorators**. Elle possède également un jeu de méthodes appelé *Operation.*
Le groupe de classe (set of class) **Decorators** représente les différents décorateurs disponibles. Chaque classe est un décorateur spécifique possédant son propre jeu de méthodes *operation.*

b. Relations entre classes

La relation (la ligne avec un diamant à son extrémité) entre la hiérarchie **Components** et la classe **decorator** représente une référence, signifiant ainsi que chaque instance de la classe **decorator** possède un composent. Ce composent référencé peut être un autre décorateur.

c. Relations entre les jeux de méthodes

Les cercles ombrés représentent des jeux de méthodes (ou fonctions) selon les relations suivantes :

Entre le jeu *Operation* (de **Decorators**) et le jeu *Operation* (de **decorator**) , il existe une relation « forward ». Celle-ci signifie que chaque appel à une méthode d'une des classes du groupe **Decorators** engendrera un appel identique à la méthode correspondante de la classe **decorator.**

Entre le jeu *Operation* (de **decorator)** et le jeu *Operation* (de **Components**)**,** il existe une relation « forward ». Celle-ci signifie que chaque appel à une méthode de la classe **decorator** engendre un appel identique à la méthode correspondante de la hiérarchie **Components**.

3.2.3 Représentation littérale du pattern « decorator »

La traduction littérale du modèle graphique du pattern « decorator » est donnée ci-dessous:

$$\begin{array}{l}
\exists \text{Components} \in \text{H} \\
\exists \text{decorator} \in \text{C} \\
\exists \text{Operation} \in 2^{2^{F}} \\
\hline
\text{tribes}(\text{operation} \otimes \text{Components}, \text{Components}) \\
\text{tribes}(\text{operation} \otimes \text{decorator}, \text{decorator}) \\
\text{tribes}(\text{operation} \otimes \text{Decorator}, \text{Decorator}) \\
\text{Inheritance}(\text{decorator}, \text{Components}) \\
\text{Inheritance}(\text{Decorators}, \text{decorator}) \\
\text{Reference_to_single}(\text{decorator}, \text{Components}) \\
\text{forward}(\text{operation} \otimes \text{Decorators}, \text{operation} \otimes \text{decorator}) \\
\text{forward}(\text{operation} \otimes \text{decorator}, \text{operation} \otimes \text{Components})
\end{array}$$

Figure 5 : Représentation littérale du pattern "décorateur" selon LePus

Une « tribes » représente une superposition entre une classe (ou set de classes, ou hiérarchie) et un set de méthodes. Une superposition entre une classe (ou set de classes ou hiérarchie) et une méthode est appelée « clan ».

3.2.4 Comparaison avec le modèle UML, Avantages et inconvénients

Comparé au modèle de classe UML ci-dessous, le modèle LePUS n'apporte aucune information complémentaire. En outre, les représentations graphiques des deux modèles sont très proches. Finalement, le modèle LePUS n'a de mieux que sa représentation « ombré », qui symbolise la multiplicité des entités, et évite ainsi la redondance des classes du modèle UML.

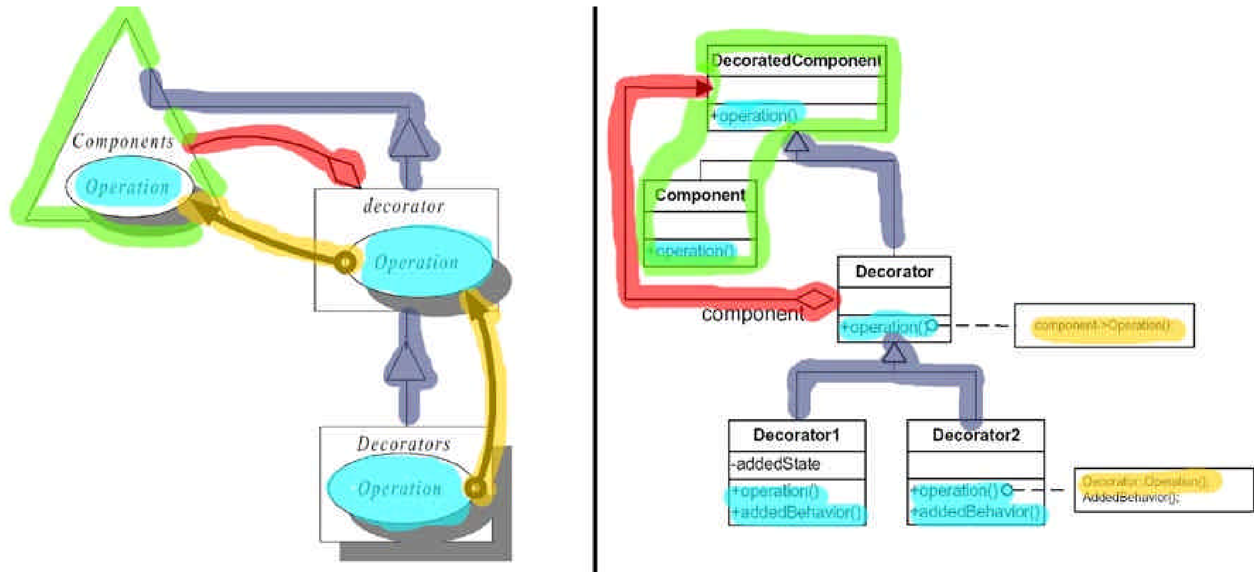


Figure 6 : Comparaison entre Le décorateur selon LePUS et le décorateur selon UML

Ainsi, les mêmes inconvénients que ceux du modèle de classe UML existent, c'est-à-dire un manque de description du comportement du à l'absence de modélisation au niveau des instances.

3.2.5 Objectifs de LePUS

S'agissant des aspects graphiques, le Langage LePUS n'apporte rien de nouveau par rapport aux schémas UML. Cependant, la version littérale permet d'entrevoir plusieurs possibilités :

- Validation :
  LePUS permet de savoir si une portion d'un certain design quelconque suit en fait les spécifications d'un pattern spécifique. Ceci peut être réalisé en unifiant les données traduites du code existant avec celles d'un pattern spécifique.

- Perfectionnement (Raffinement) :
  Le langage LePUS permet aisément de perfectionner ou simplifier le comportement et la structure d'un pattern donné. Son design est amélioré en ajoutant des clauses (prédicats, relations) à son schéma. Inversement, enlever des clauses revient à simplifier son design.
  De part ces propriétés, nous pourrons donc facilement répondre à la question :
  Est-ce que le pattern X est un cas spécial du pattern Y ?

Relation entre les patterns (projection et abstraction) :
Comme le langage est basé sur la manipulation des dimensions, un set peut aisément être remplacé par un de ses éléments pour obtenir une projection du design original.
Inversement, généraliser un élément en un set permet d'obtenir une abstraction de l'élément original.

Relation entre les patterns (point commun entre différents patterns) :
LePUS permet de détecter les ressemblances entre différents patterns. Par exemple de savoir si :
- un pattern spécifique est en fait une partie d'un autre pattern
- 2 patterns différents possèdent une partie commune, ou un comportement commun.

Tool support :
Tous les avantages précités sont importants. Cependant, il est encore plus appréciable de pouvoir automatiser ces observations par l'intermédiaire d'un outil.
Ainsi, comme la notation de LePUS est similaire aux spécifications de PROLOG, il est plus facile et automatique de valider un pattern pour qu'il soit conforme aux propriétés du modèle original, ou encore de reconnaître un pattern parmi un modèle existant.

3.2.6 Conclusion

Bien que la modélisation LePUS soit plus compacte et proche d'un modèle théorique, il n'en reste pas moins qu'elle n'apporte rien qui permettrait de comprendre plus rapidement le fonctionnement du pattern.

3.3 *Problématique actuelle*

3.3.1 Présentation de la problématique

Tel que vu dans le chapitre 2 à travers la modélisation selon le GoF ou LePus, la description de la structure statique des patterns (diagramme de classe) est très bien documentée, et agrémentée de schémas relativement explicites.

Cependant, la description des comportements n'est pas du tout traitée de manière formelle, et seules quelques explications textuelles permettent après plusieurs lectures de comprendre le fonctionnement concret du pattern.

Il est de plus très difficile à travers la structure de classes de savoir quel est le rôle de chaque objet existant. Ceci paraît d'autant plus regrettable que le rôle d'un objet peut changer au cours de son exécution.

A l'opposé, certains patterns de type comportementaux ne représentent que très peu de structure, ce qui est d'autant plus dommageable qu'il sera difficile d'introduire ce type de solution dans un design basé sur une structure existante.

3.3.2 Exemple du décorateur

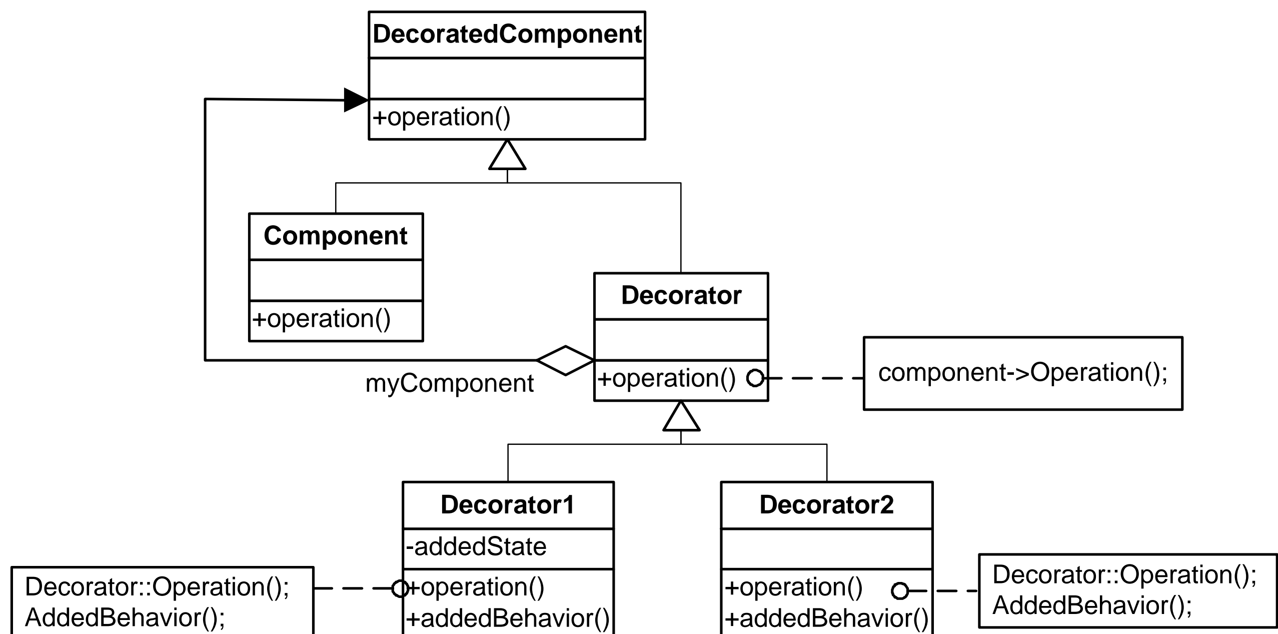


Figure 7 : Diagramme de classe du décorateur

Ce diagramme définit clairement une structure de classe à ajouter au design existant. Il est donc relativement aisé d'intégrer un mécanisme de décoration.

Il est toutefois indispensable de comprendre le comportement d'un décorateur et de son environnement pour l'utiliser. On ne sait donc pas qui appelle qui, dans quel ordre, et quel rôle possède chacun des acteurs. (Il n'est par exemple mentionné nul part qu'un « Decorator » joue le rôle de « Component » dans certain cas)

En résumé, cette vue statique des choses permet de comprendre la structure d'un décorateur. Elle n'indique cependant comment l'utiliser ni la manière dont elle fonctionne.

3.3.3 Exemple du « business event-result history pattern »

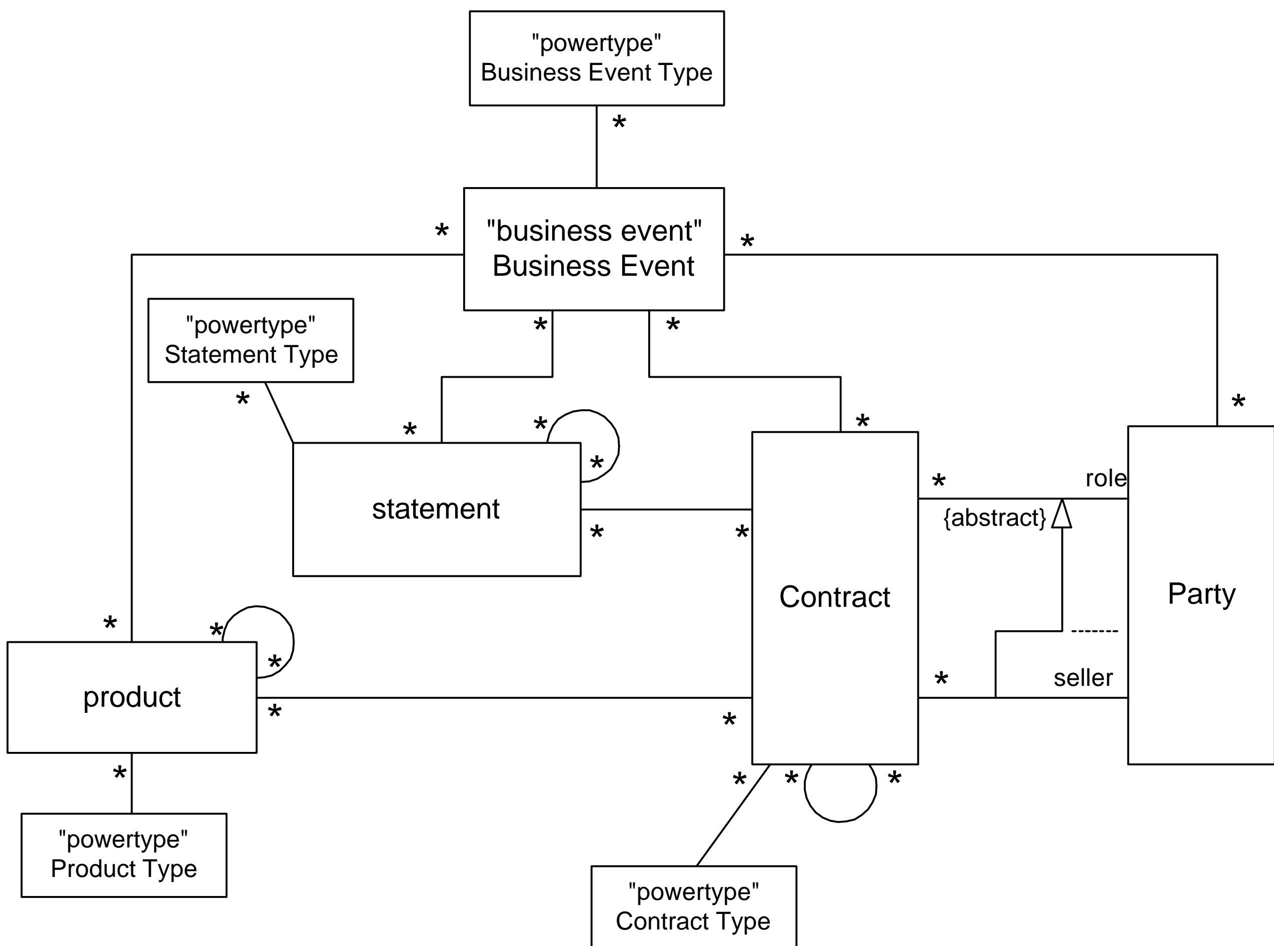


Figure 8 : Diagramme de classe du pattern "business event-result history" [tiré directement du livre]

A l'instar du décorateur, le seul diagramme qui nous est proposé est un diagramme de classe.

Les problèmes identiques se posent alors, et les questions similaires apparaissent :

« Comment fonctionne ce pattern ? »
« Qui met à jour les données ? »
« Qui envoie des messages a qui ? »
« Dans quel ordre ? »

Autant de questions qui ont trait au comportement et qui restent sans réponse.

Une description textuelle d'accompagnement de ce diagramme existe bien, elle ne concerne cependant que description individuelle de chaque participant, et non l'interaction entre acteurs. Ce dernier aspect manque cruellement.

Ce n'est finalement que dans l'exemple inclus en fin de chapitre qu'un seul élément de collaboration du pattern est abordé, et ce de manière plus ou moins précise.

3.4 Formalisation complémentaire au modèle existant

3.4.1 Présentation

Pour combler partiellement les problèmes rencontrés dans la section précédente, 2 types de vues complémentaires au diagramme de classe sont présentés. Elles décrivent ce qu'il manque dans l'état actuel des choses, à savoir le comportement.

Le problème peut être modélisé de manière très visuelle sans suivre de formalisme précis, et ce indépendamment de chaque pattern. Cette modélisation vise simplement à faire passer un message et donner un référentiel unique. Cette visualisation serait donc parfaite pour symboliser le pattern. L'exemple du décorateur de la section 3.3.2 est nettement plus parlant qu'une explication. A noter que l'auteur de ce document « Formalizing the Intent of Design Patterns», par ailleurs auteur du Langage LePus, corrobore cette vision des choses.

Bien plus qu'une description sommaire du comportement, il convient de disposer de bases solides sur lesquelles reposera l'implémentation. C'est ainsi que nous proposons une formalisation comportementale adéquate à travers un diagramme de collaboration ou un « sequence diagramme » suivant la norme UML.

3.4.2 Exemple du décorateur

« Vue en bloc »

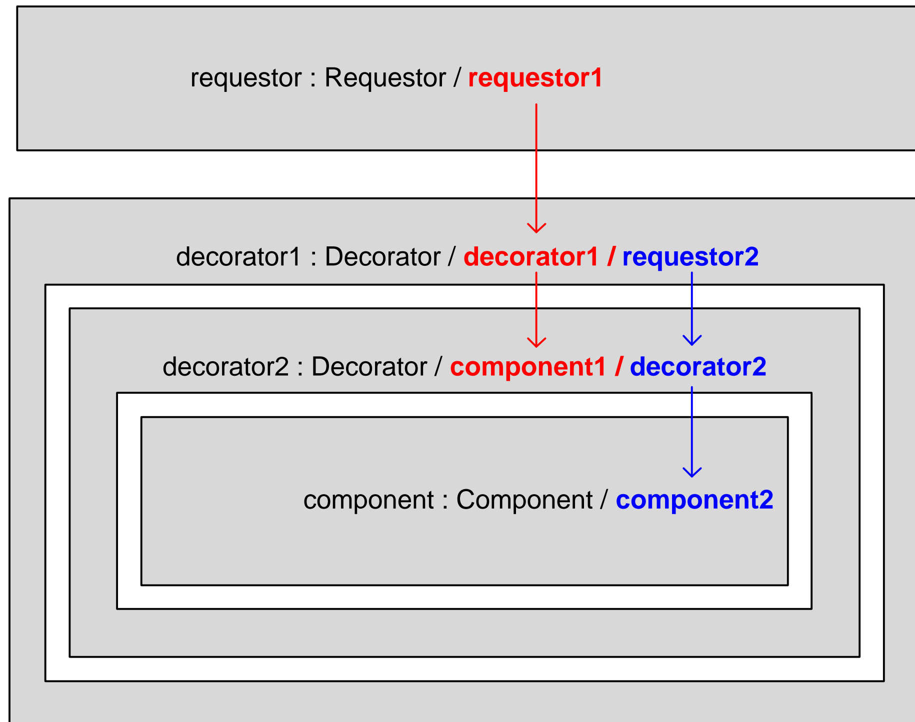


Figure 9 : Vue en bloc du décorateur

Cette vue n'a rien de formel, elle est en revanche plus concrète et plus précise qu'une situation mettant en jeu un décorateur. Elle n'a que pour seul objectif d'améliorer la compréhension.

L'imbrication des décorateurs, l'encapsulation du « component », et finalement le rôle de chaque objet dans la décoration sont ici mis en évidence.

Nous sommes bien conscients qu'une telle formalisation n'est applicable que dans ce cas précis. Cependant, il est souvent bien plus appréciable d'avoir un tel type de diagramme pour symboliser au mieux notre problème qu'une structure complexe mettant en jeu un nombre conséquent d'interfaces. On imagine donc aisément un case tool symbolisant l'utilisation d'un décorateur englobant une classe donnée grâce à ce type de vue, même si celle ci n'est finalement pas compatible UML.

« Diagramme de collaboration »

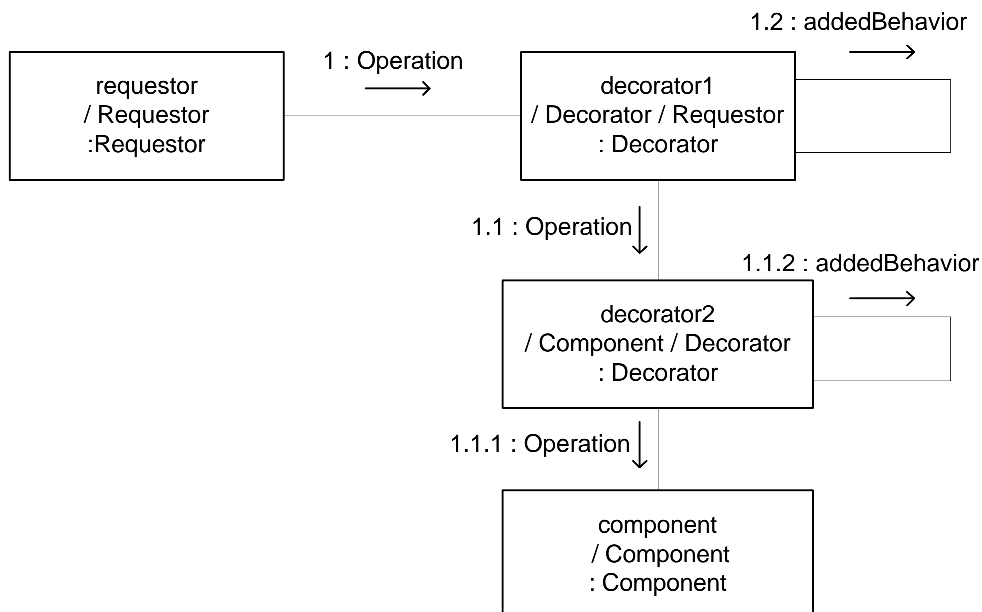


Figure 10 : Diagramme de collaboration UML du décorateur

Contrairement à la précédente, cette vue est entièrement compatible avec UML. Elle apporte une vision très instructive sur la collaboration entre instances. On retrouve donc ici le plus gros de ce qu'il manquait au diagramme de classe, à savoir la description des relations entre objets.
Grâce à ce diagramme, l'imbrication des décorateurs entre eux, leurs ordres et leurs rôles est aisément compréhensible.

3.3.3 Exemple du « business event-result history pattern »

« Sequence diagram »

Un diagramme de collaboration UML aurait pu être présenté dans cette partie ;, le « sequence diagram » paraît cependant plus adapté. Même sans description du fonctionnement à l'intérieur de chaque acteur, l'échange d'informations entre les deux participants peut rapidement être déduite.

Plutôt que de montrer un modèle correspondant à la version théorique du pattern, nous avons choisi de présenter un exemple beaucoup plus significatif.

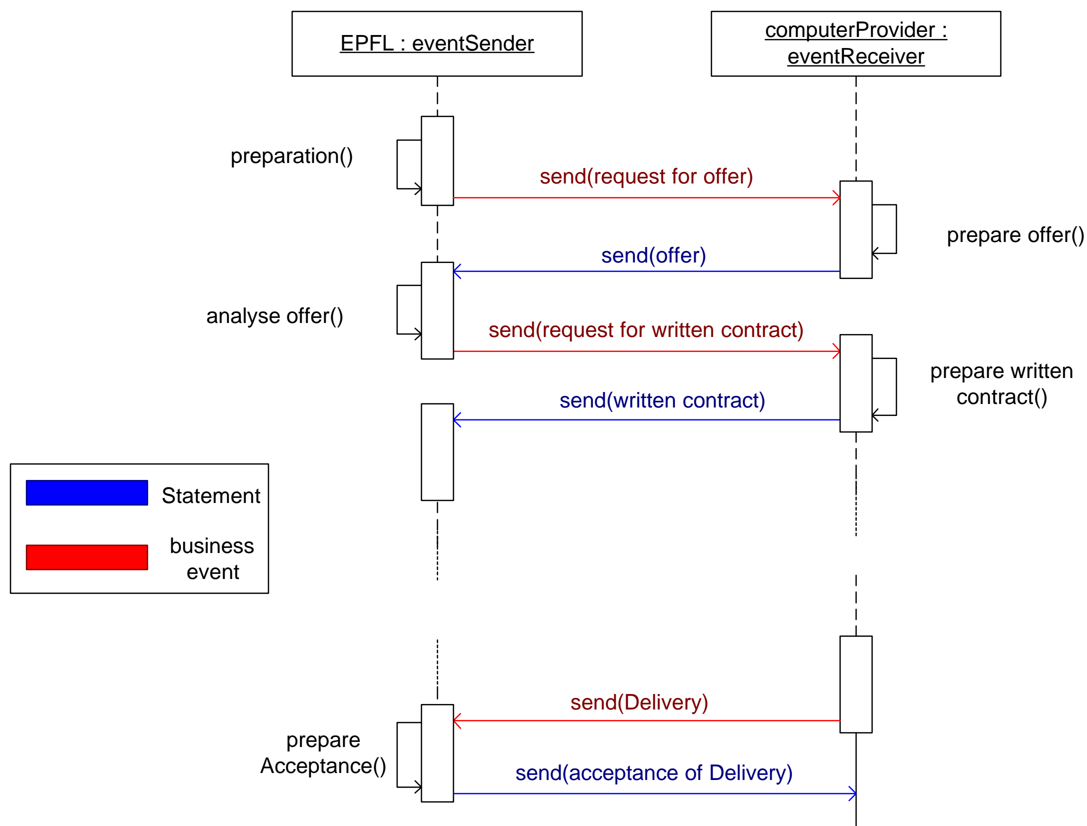


Figure 11 : Sequence diagram montrant plusieurs applications du pattern "business event-result history"

Le même pattern « business event-result history » a été appliqué à plusieurs reprises.

Ainsi, lors de la première application, l'émetteur (« eventSender ») du « business event » se trouve être l'EPFL. Il envoie une demande d'offre (« request for offer ») au fournisseur informatique (« eventReceiver »), lequel lui retournera l'offre en question (« offer »).

Lors de la deuxième application, l'EPFL envoie une demande de contrat (« request for writtern contract »), faisant suite à l'offre proposée lors du premier « business event ». Elle recevra par la suite un contrat à signer.

Après plusieurs échanges protocolaires correspondant à plusieurs applications du même pattern, intervient la livraison du produit. Les rôles sont alors inversés, et l'émetteur de l'événement (« Delivery ») devient le fournisseur informatique qui envoie ces produits a l'EPFL. Il recevra en retour une confirmation de réception (« Acceptance of Delivery »)

3.5 Proposition d’un autre type modélisation

Les propositions de la section précédente sont très utiles, il manque cependant plusieurs choses permettant de définir de manière complète et formelle notre problème.
Ainsi, il manque des informations concernant les rôles de chaque objet, de même qu’il est impossible de décrire de façon générale ce qu’il se passe avec un nombre indéfini d’applications du même pattern.

3.5.1 Description

Cette nouvelle formalisation des design patterns est basée sur les collaborations entre acteurs d'un pattern.

Chaque objet est représenté par un rectangle divisé en 2 parties ; la première partie montre l'environnement de l'objet, en quelque sorte sa vision du monde extérieur. La deuxième partie décrit son comportement, les signaux reçus, le traitement effectué, et enfin les signaux renvoyés.

L'environnement de l'objet est un simple diagramme de classe UML, dans lequel l'objet est représenté par une classe Myself.

Le comportement est décrit grâce à un diagramme d'activité (activity diagram) UML.

L'interaction entre les objets est représentée par un cercle pointillé.

Dans ce chapitre, nous allons présenter l'exemple du décorateur et business event-result history, mais également sa synthétisation dans le but de généraliser l'application de plusieurs patterns.

3.5.2 Exemple du décorateur.

Diagramme « appelant appelle appelé » sans décoration :

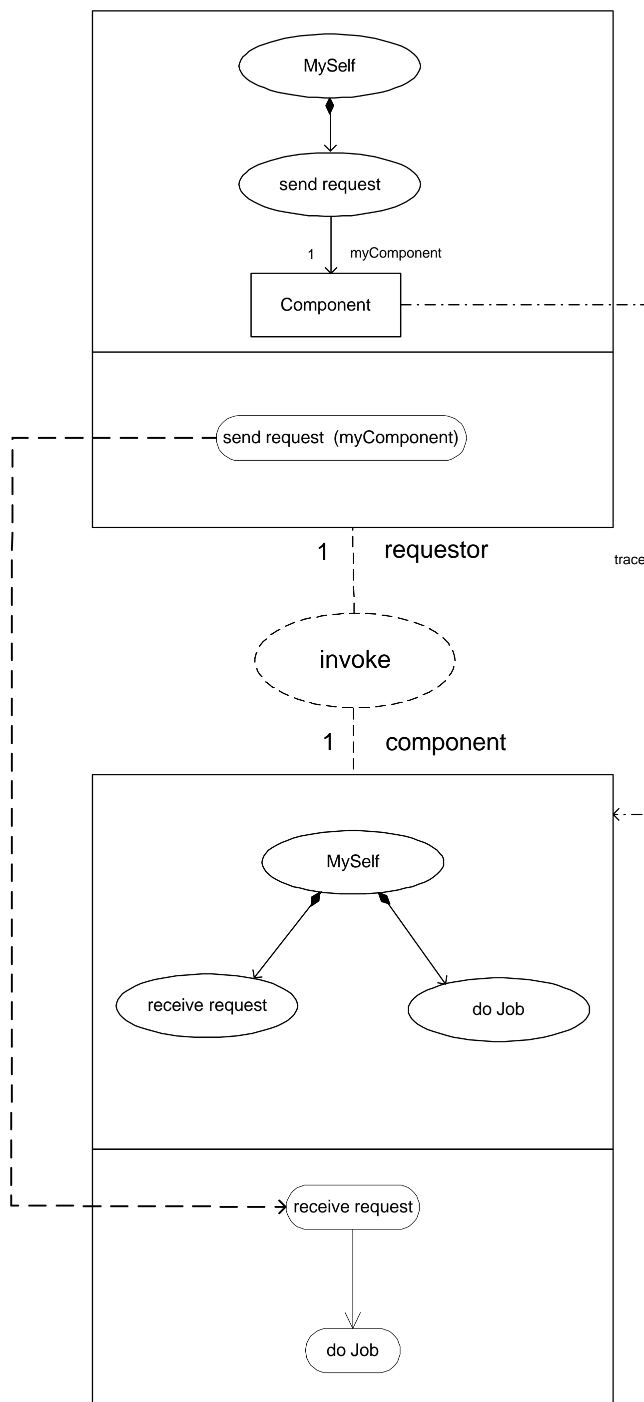


Figure 12 : Invocation simple

Afin de mieux comprendre le fonctionnement et le rôle du décorateur, nous présentons d'abord l'état des choses sans décoration. Nous avons donc 2 objets, un « requestor » et un « component ».
Le « requestor » connaît le « component » et lui adresse une requête (« request »), alors que le « component », sans savoir qui l'interroge, reçoit la requête, et l'exécute.

Note 1 : La relation « trace » entre l'élément component du requestor et l'élément component global symbolise la relation qu'il y a entre la vision qu'a le « requestor » du component et le component réel.
Note 2 : La relation entre l'événement « send request » et l'événement « receive request » est une contrainte séquentiel imposant l'ordre dans lequel les actions doivent être effectués.

Utilisation d'un seul décorateur :

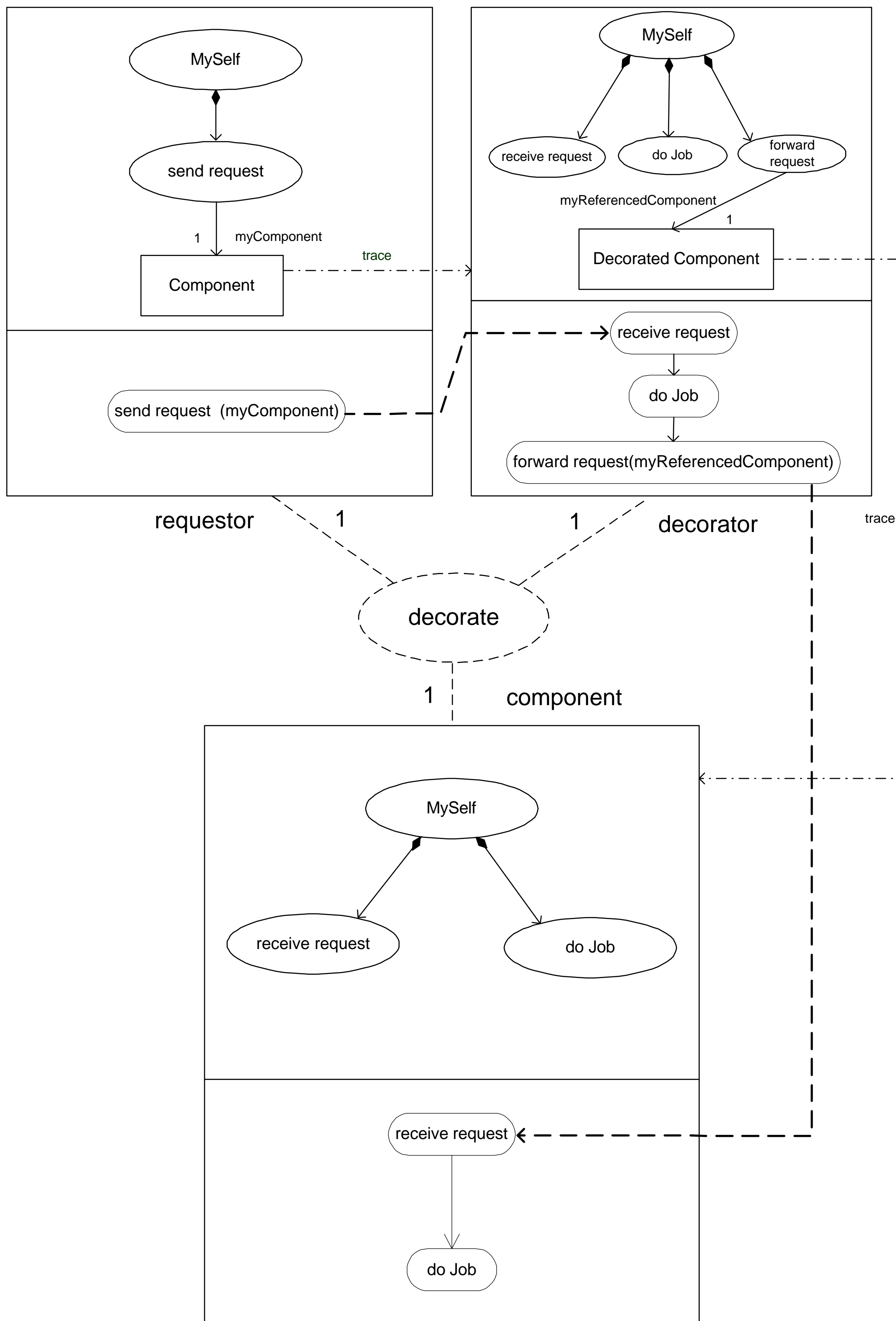


Figure 13 : Simple décoration

Nous disposons ici de 3 acteurs majeurs. A l'image du diagramme sans décoration, nous avons, le « requestor » et le « component », auxquels vient s'ajouter le « decorator ».

Le « requestor » ne s'adresse plus au « component » directement mais au « decorator », qui connaît le « component » . Ainsi, le « decorator » reçoit la requête (« request ») du « requestor », l'exécute, et la transmet au « component ». Ce dernier la reçoit et l'exécute à son tour.

Synthèse de 2 décorateurs :

[PAGE A3 avec synthèse des décorateurs]

Figure 14 : Synthèse de deux décorateurs

Nous avons mis en jeu 2 décorateurs simples, représentant 2 fois 3 acteurs :

le « requestor »,
le « decorator »
le « component »

soit 6 acteurs :

le « requestor1 »
le « decorator1 »
le « component1 »
le « requestor2 »
le « decorator2 »
le « component2 »

A l'aide notamment du diagramme en bloc de la section 3.3.2, on s'aperçoit cependant qu'un même objet peut jouer plusieurs rôles lors de l'utilisation de 2 décorateurs successivement. Ainsi, lors de la deuxième décoration le « requestor2 » n'est en fait que le « decorator1 » et le « component1 » que le « decorator2 », comme le montre la figure 9. Chaque élément du « requestor2 » a sont équivalent dans le « decorator1 », et chaque élément du « component1 » a sont équivalent dans le « decorator2 ». Le comportement du « decorator2 » enchaîne celui du « decorator1 »

Résultat de la synthèse : Double décoration d'un objet

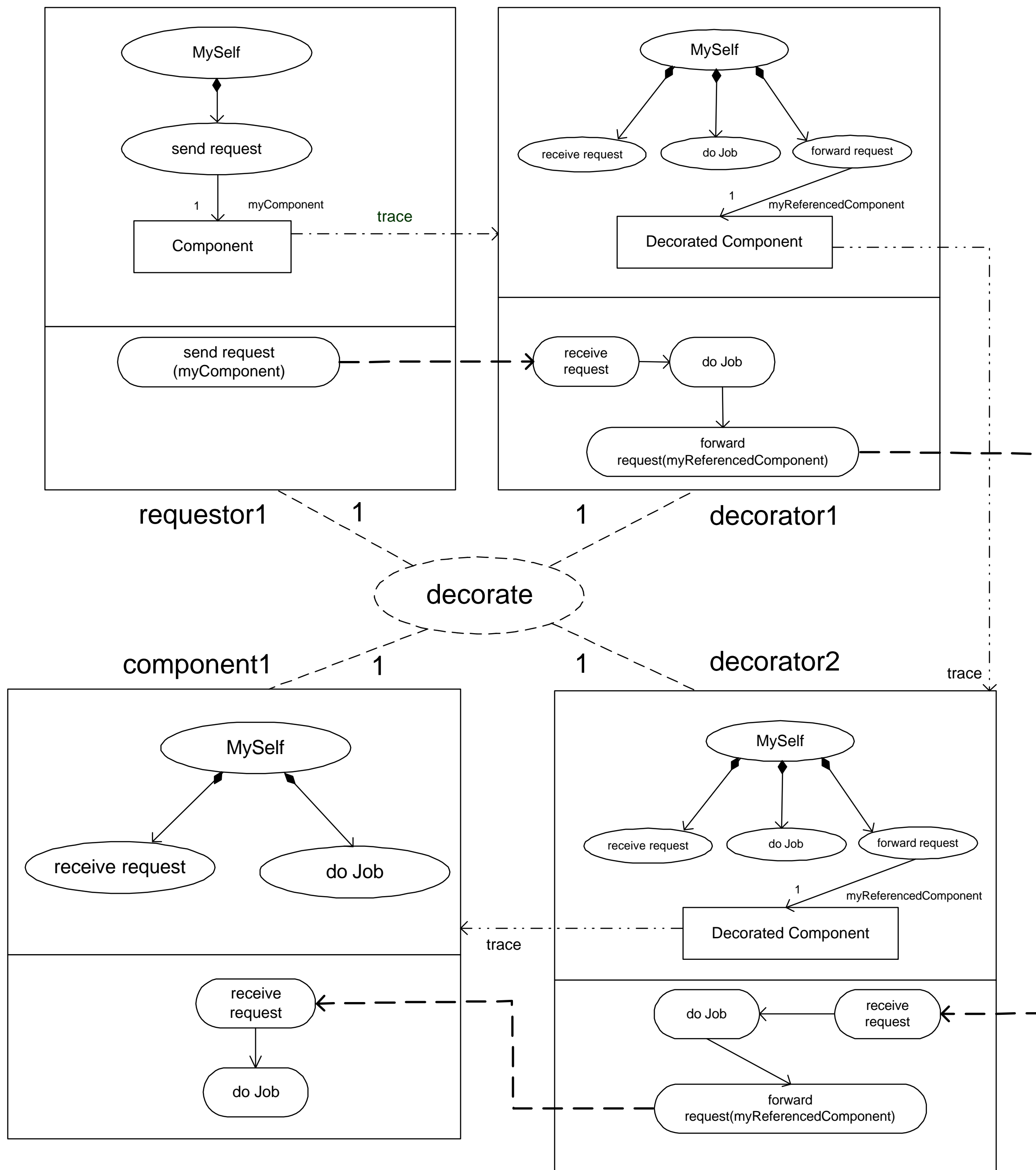


Figure 15 : Resultat de la synthèse de deux décorateurs

En rapport avec le paragraphe précédent concernant la synthèse des 2 décorateurs, le « decorator1/requestor2 » de la figure 9 garde sa dénomination de « decorator1 » dans la figure 10 puisque c'est sont rôle principal. Il possède les caractéristiques d'un requestor et en plus celles d'un décorateur. De la même manière, le « decorator2/component1 » garde la dénomination de « decorator2 » .

Ainsi, un « requestor » adresse la requête à son « decorator » (myComponent du requestor i.e. decorator1), lequel adresse la requête à son « decorator » (myReferencedComponent du decorator1 i.e. decorator2) qui lui-même la transmet au « component » (myReferencedComponent du decorator2 i.e. component). Chacun d'entre eux exécute la requête dans son propre contexte lorsqu'il la reçoit.

Mise en œuvre de décorations dans un exemple concret

Ici, les décorations consistent à ajouter deux fonctionnalités supplémentaires a un « textViewer » :

- Une barre de défilement (« scrollbar »)
- Un cadre (« border »)

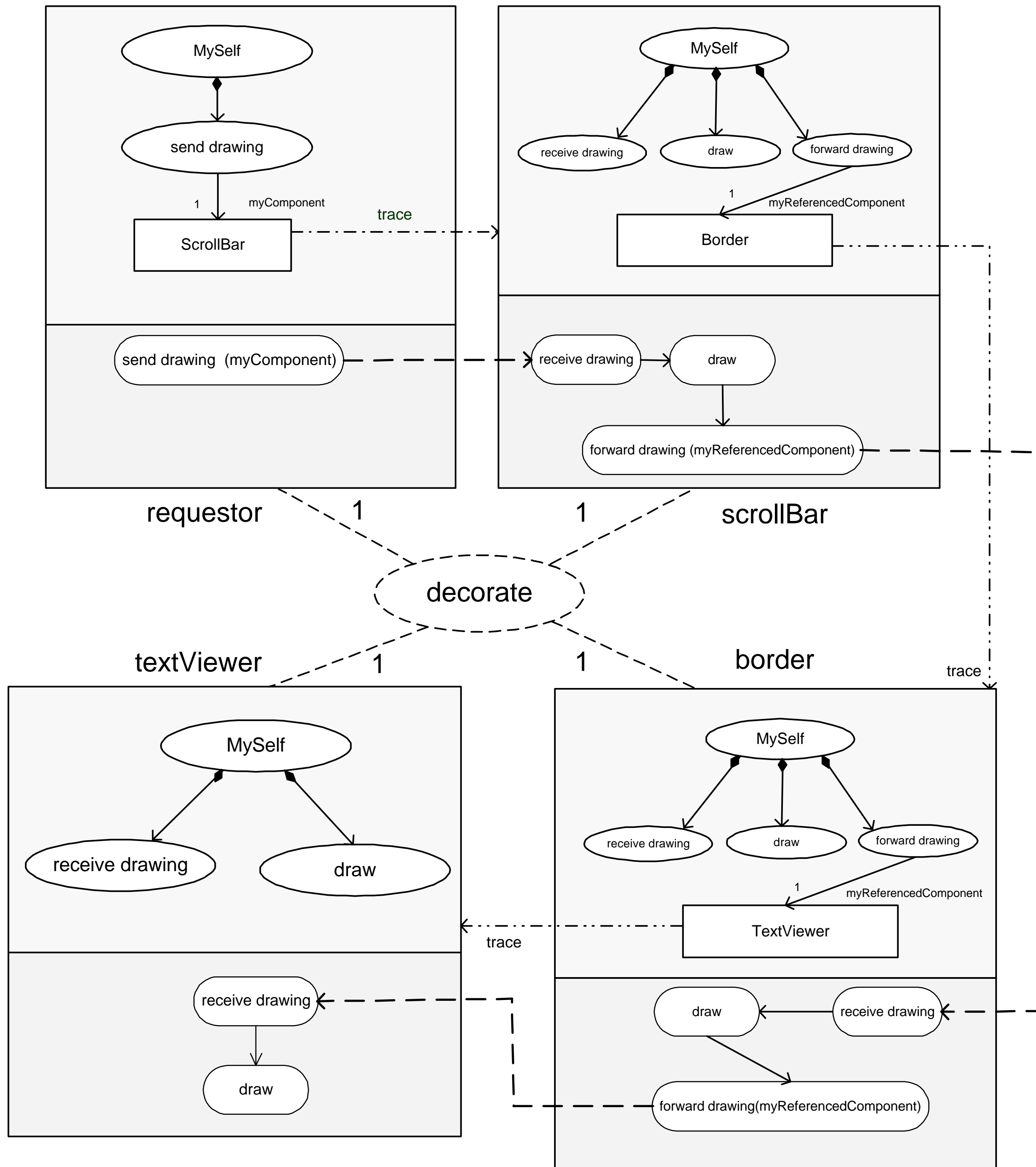


Figure 16 : Exemple concret de décorations

3.5.3 Exemple du « business event-result history pattern »

Diagramme général d'un pattern « business event-result history » :

[PAGE A3 DIAGRAMME GENERALE BUSINESS EVENT]

Figure 17 : Business event-result history pattern

Description de l'émetteur de l'événement (« eventSender »)
L'émetteur est capable de :

- Créer un événement (« create event »), en prenant en paramètre les spécifications d'un produit (« requested product specification)
- D'envoyer (« send ») des événements (avec l'événement en paramètre) à un collaborateur.
- D'accepter (« accept ») un rapport (« statement »)
- De recevoir (« receive ») un contrat provenant d'une personne donnée.
- D'enregistrer (« record ») tous les éléments participant à l'événement.

En ce qui concerne l'enregistrement des données, ce schéma est incomplet. Pour des raisons de clarté, les références entre la base de données et les différents objets enregistrés ont volontairement été omises.

Il convient donc de considérer le schéma suivant lors de l'enregistrement de l'événement.

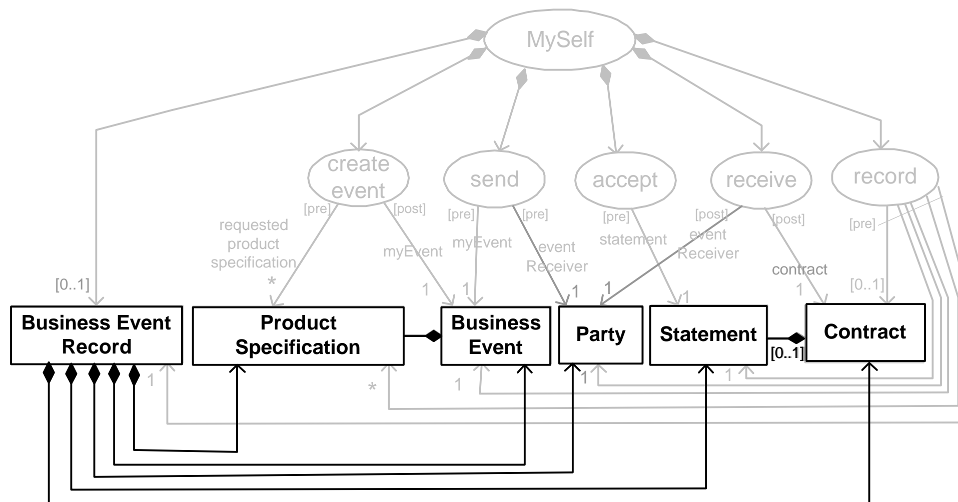


Figure 18 : Structure pour l'enregistrement d'un business event"

Ainsi, lors de la suppression dans la base de données d'un événement, toutes les relations avec les participants à cet événement seront également supprimées.

Description du récepteur de l'événement (« event receiver »
Le récepteur est capable de :

- Créer un rapport (« create statement ») relatif aux spécifications d'un produit. (une offre par exemple)
- Recevoir un événement (« receive ») de la part de l'émetteur
- Créer (« create contract ») un contrat qui possèdera un ou plusieurs rapport(s)
- Envoyer (« send ») un contrat (et les rapports qu'il contient)

-
Concernant les contrats et les rapports, la description du livre reste assez vague, puisque qu'un contrat peut avoir plusieurs rapports et vice-versa. Nous avons donc considéré le contrat comme un dossier qui suit la totalité de la transaction entre tous les partis, et dans lequel les rapports sont insérés.

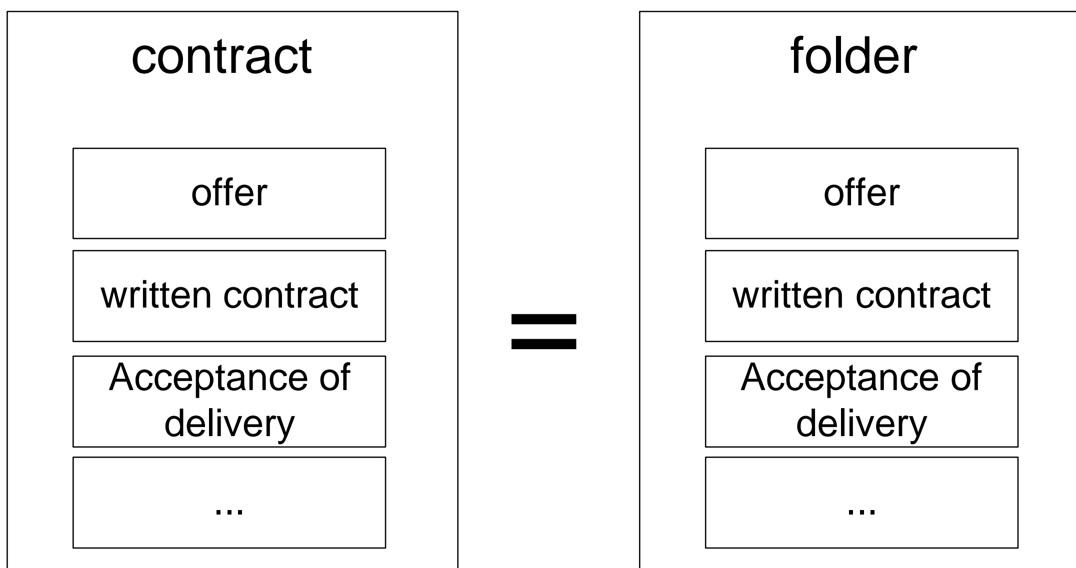


Figure 19 : Explication d'un contrat

Description de la collaboration entre l'émetteur et le récepteur
En premier lieu, l'émetteur crée l'évènement en prenant en considération les spécifications des produits qui l'intéressent, puis il envoie cet événement au destinataire.

Une fois l'événement reçu, on crée un rapport qui prend en compte les spécifications des produits voulu par l'émetteur, si un contrat n'existe pas, il est créé, et on lui associe le rapport. Le tout sera enfin renvoyé à l'émetteur de l'événement pour acceptation.

L'émetteur reçoit donc le contrat, juge le rapport qui si trouve et l'accepte ou le rejette. Puis, quelque soit le résultat de ce jugement, toute la transaction est enregistré dans la base de donnée.

Application du pattern dans deux cas concret :

Premier cas, lors d'une demande d'offre (« request for offer »):

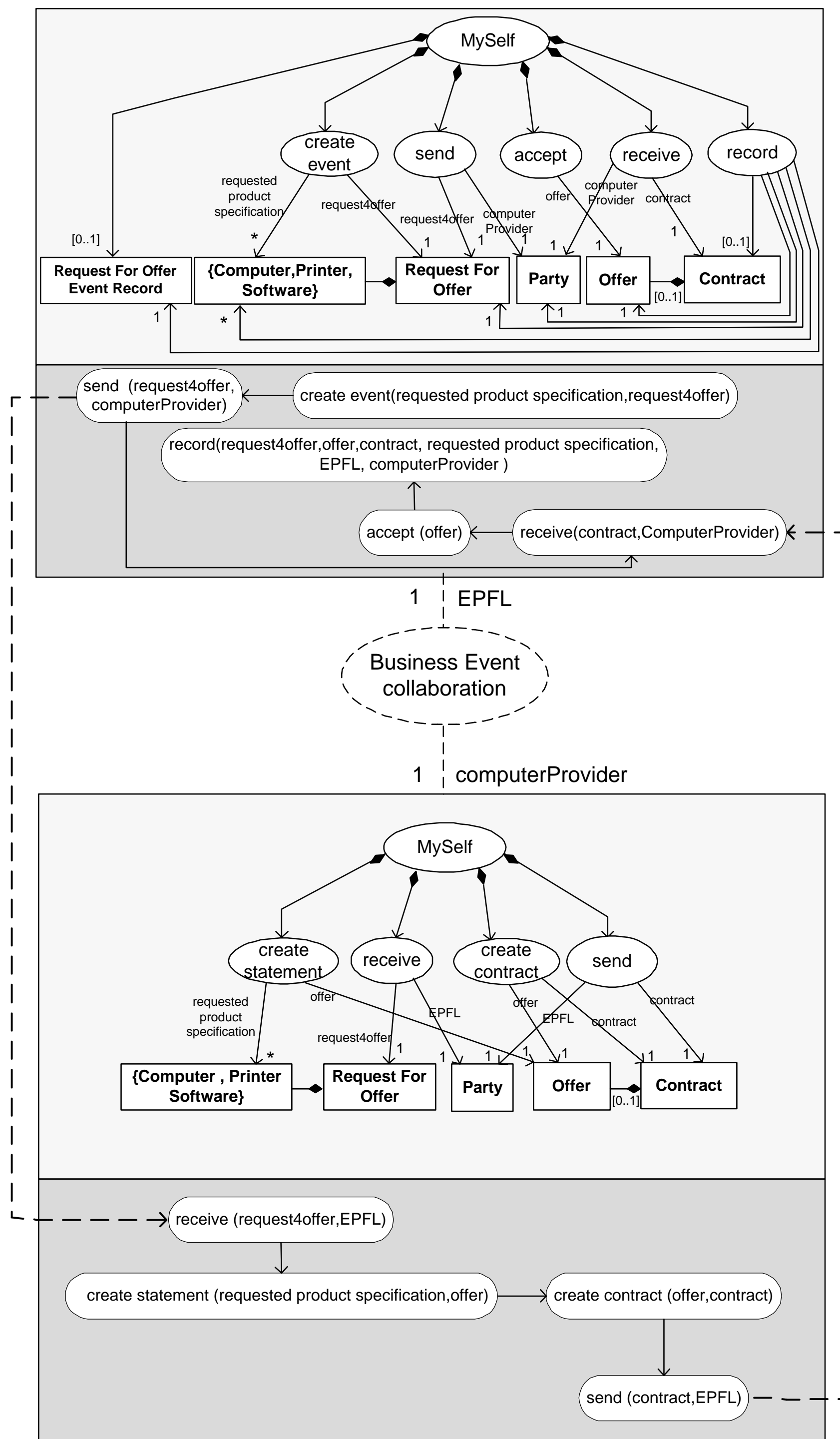


Figure 20 : Application du pattern Business event-result history

l'EPFL transmet une demande d'offre à un fournisseur de produits informatiques (par exemple des ordinateurs, des imprimantes et des logiciels). Elle reçoit en retour un dossier (« contract ») contenant une offre pour ces produits.

Nous avons donc comme:
- business event : une demande d'offre (« request for offer »)
- émetteur de l'événement : un client, l'EPFL
- récepteur de l'événement : un fournisseur informatique (« computerProvider »)
- comme produits : des ordinateurs, des imprimantes et des logiciels
- comme rapport : une offre (« offer »)

Deuxieme cas, lors de la demande de contrat écrit (« request for written contract »)

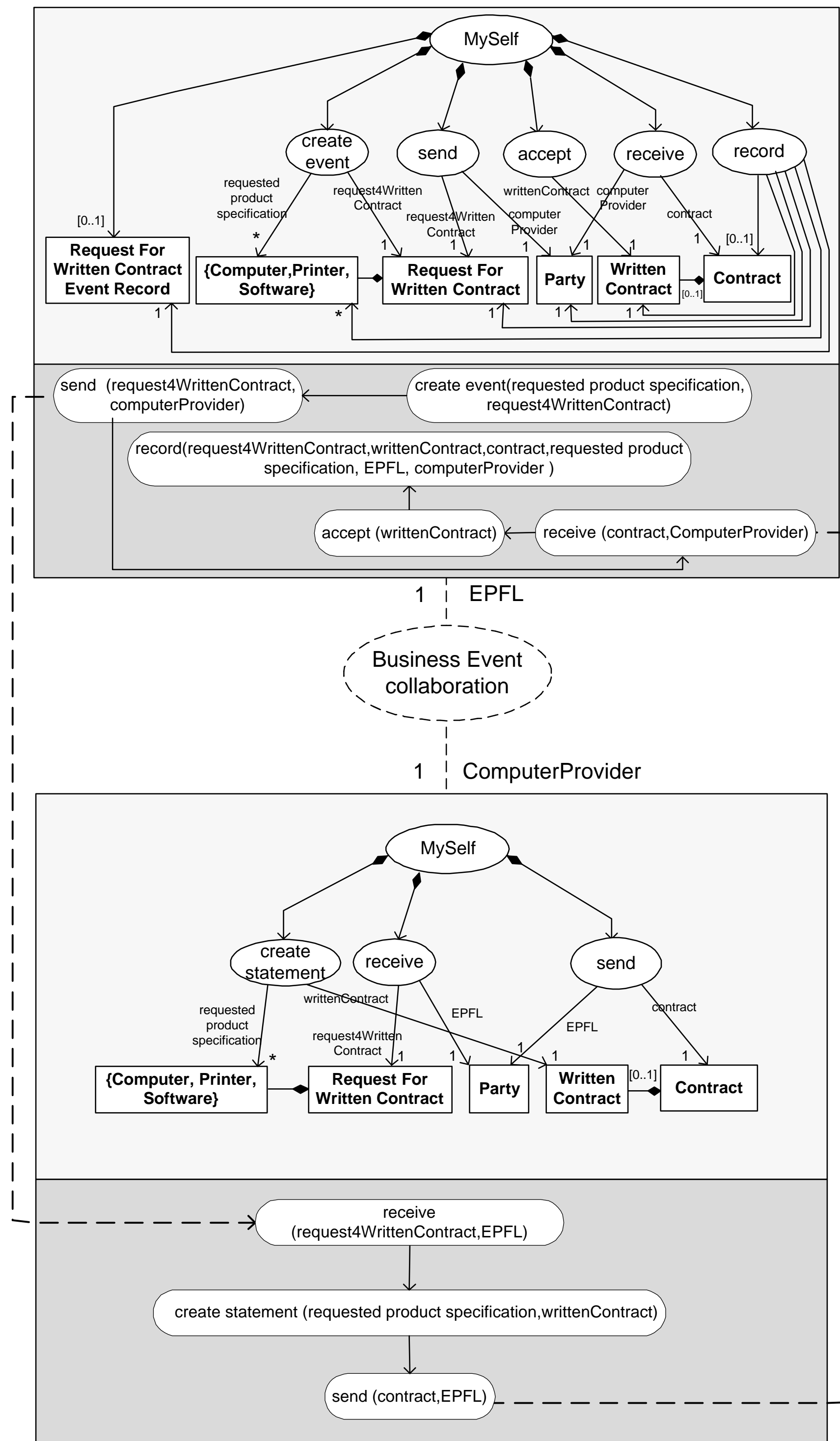


Figure 21 : Deuxieme application du pattern business event-result history

Faisant suite à la demande d'offre précédente, l'EPFL transmet une demande de contrat écrit au fournisseur informatique concernant les ordinateurs, les imprimantes et les logiciels. Elle reçoit en retour le dossier auquel est ajouté le contrat écrit à signer.

Nous avons donc comme :

- business event : une demande de contrat écrit (« request for written contract »)
- émetteur de l'événement : un client, l'EPFL
- récepteur de l'événement : un fournisseur informatique (« computerProvider »)
- comme produits : des ordinateurs, des imprimantes et des logiciels
- comme rapport : un contrat écrit (« written contract »)

-

Synthèse de 2 applications du pattern :

[PAGE A3 Diagramme synthèse]

Figure 22 : Business event-result history , synthèse de deux patterns

Nous allons maintenant synthétiser les deux patterns correspondant à la demande d'offre et à la demande de contrat écrit. Pour ce faire, nous identifions quels sont les éléments, les acteurs et actions présentes dans les deux cas de figure.

Nous avons :
- Le client : l'EPFL
- Le fournisseur informatique
- Les produits (Ordinateurs, imprimantes et logiciels)
- Le contrat (ou dossier dans notre cas)

A cela, nous ajoutons une pré-condition concernant la demande de contrat écrit et sa création qui ne peuvent exister que dans la mesure où une offre existe.

Au niveau du comportement, les actions concernant le deuxième pattern s'enchaînent naturellement aux actions du premier.

Synthèse de plusieurs pattern « business event-result history »

Comme montré dans le chapitre 3.3.3, nous pourrions appliquer une troisième fois ce pattern en inversant les rôles. Le fournisseur informatique deviendrait l'émetteur de l'événement « livraison du produit» tandis que l'EPFL, récepteur de l'événement, renverrait une quittance de livraison.

Multiplication des acteurs

On peut également imaginer que le client s'adresse à plusieurs fournisseurs informatiques, ou de manière encore plus évidente encore, qu'un fournisseur possède plusieurs clients.

Ainsi nous aurions pour de multiples fournisseurs :
- plusieurs acteurs correspondant à chaque fournisseur, jouant le rôle d'émetteur et de récepteur d'événement.
- Les spécifications des produits demandés identiques dans tous les cas.
- Un événement différent pour chaque fournisseur.
- Un contrat (dossier) défini pour chaque relation client /fournisseur.
- Une offre, un contrat écrit,… renvoyé par chaque fournisseur.

Au niveau du comportement, nous pourrions imaginer des processus indépendants qui ne peuvent être interrompus pour ce qui concerne le receveur de l'événement, et un processus en attente de réponse pour l'émetteur.

Et nous aurions pour de multiples clients :
- plusieurs acteurs correspondants à chaque client, pouvant jouer le rôle d'émetteur ou de récepteur de l'événement.
- Des spécifications de produits différents en fonctions des demandes.
- Un événement différent pour chaque requête.
- Un contrat différent par client et par affaire
- Une offre, un contrat écrit,… renvoyé à chaque client en fonction de sa demande.

Au niveau du comportement, chaque requête serait traitée indépendamment de manière ininterrompue.

[page A3 resultat de la synthèse]

Figure 23 : Business event-result history, résultat de la synthèse

De la synthèse, nous retrouvons nos deux acteurs, le client et le fournisseur informatique, qui collaborent à travers deux « business events ».

Nous avons :
- pour les mêmes produits deux événements, une demande d'offre et une demande de contrat.
- pour le même contrat (dossier) deux rapports, une offre et un contrat écrit.
- Et pour la base de donnés, deux enregistrements correspondant aux deux transactions.

3.5.4 Applicabilité de la méthode

Nous avons choisi deux exemples pour expliquer les différentes possibilités de cette méthode :
- Le « decorator », simple, académique et relativement bien connu.
- Le « business event-result history », plus complexe, peu connu et moins statique.

Deux cadres très différents où la méthode a pu être appliquée sans difficultés majeurs. Deux exemples ou nous avons réussi à décrire bien plus que ce qu'il existait déjà, et ceci même en l'absence de description formelle du comportement.

Il est donc raisonnable d'imaginer une transcription des patterns du GoF et de HH Errikson dans le but :
- d'avoir une meilleur compréhension des patterns
- de pouvoir les intégrer à un outil de développement (voir paragraphe 4.2)

Pour ce qui est des patterns plus comportementaux, les modèles seront alors plus utiles, mais sensiblement plus complexes. Les patterns seront quant à eux plus statiques, plus simples voire triviaux. L'exemple typique est le pattern « accountability » qui n'offre qu'un attrait relativement restreint car il ne met en jeu que deux acteurs et aucun comportement.

3.6 Modèles complémentaires à notre proposition

La description de notre proposition a mis en évidence qu'un décorateur pouvait avoir plusieurs rôles ; ceci n'est toutefois pas très explicite dans le diagramme concernant la synthèse, c'est pourquoi il paraît judicieux d'ajouter un nouveau diagramme explicitant les rôles de chacun.

Pour cela, nous nous sommes inspirés de la notation de Riehle consistant a modéliser les classes par des rectangles et les différents type de rôles par des ovales. Les patterns sont quant à eux représentés par des ovales pointillés, comme à l'accoutumée.

3.6.1 Exemple du décorateur

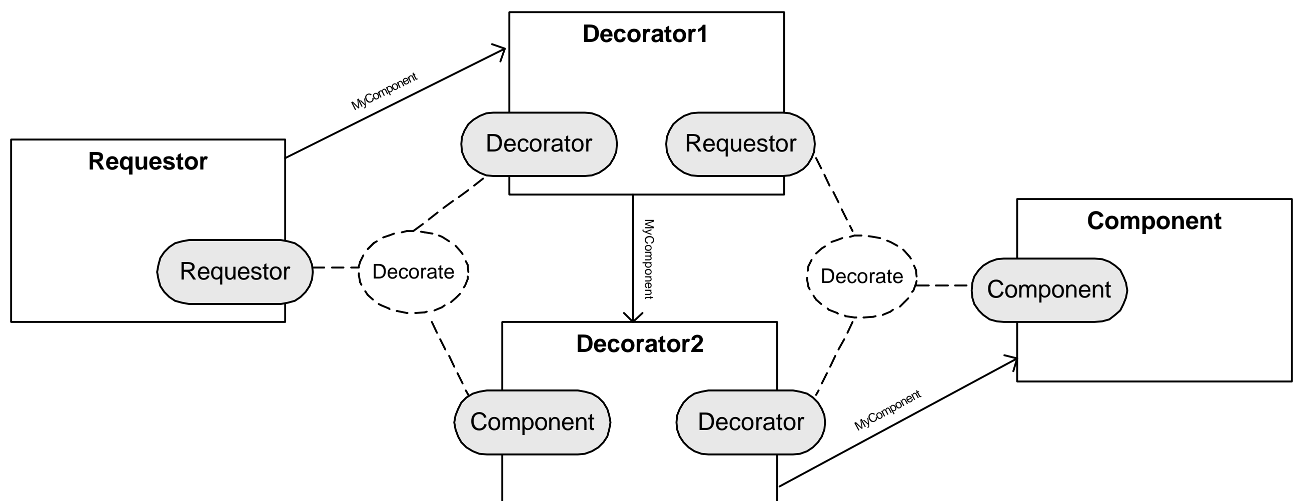


Figure 24 : Role modeling de plusieurs applications du décorator

Ce diagramme montre les différents rôles joués par chacun des acteurs. Ainsi, le decorator1 joue son rôle naturel de « decorator » pour la première application du pattern, et un rôle de « requestor » pour la seconde décoration. Le decorator2 joue également son rôle naturel de décorateur lors de la seconde application et un rôle de « component » pour la première décoration.

3.6.2 Exemple du « business event-result history pattern»

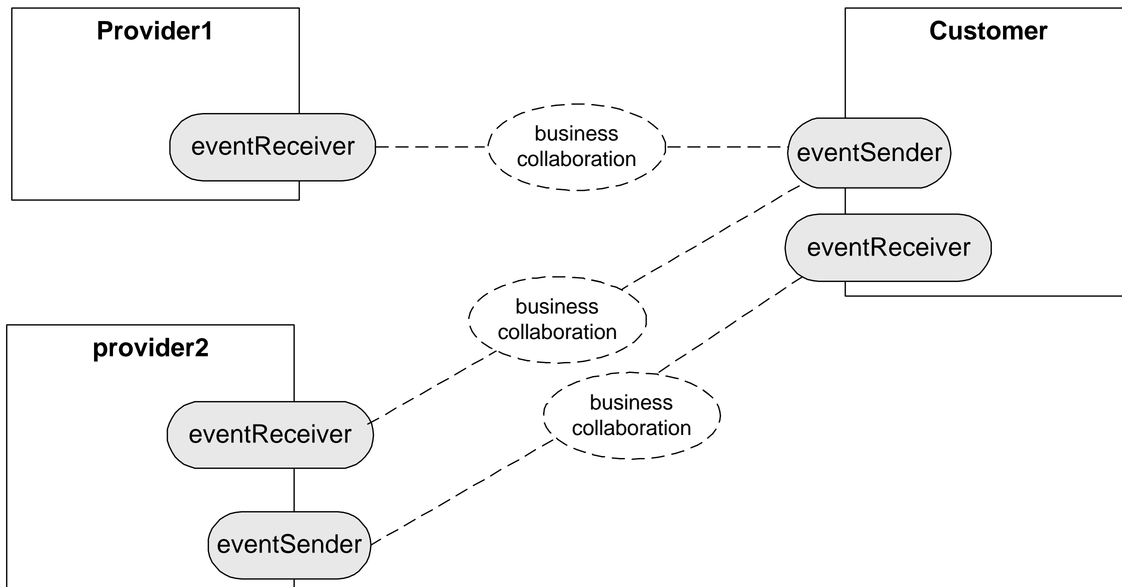


Figure 25 : Role modeling de plusieurs applications "business event-result history pattern"

Nous avons représenté trois applications du business event-result history pattern. Il apparaît que le pattern peut s'appliquer dans les deux sens, puisque chacun est capable d'émettre et de recevoir des « business event ». En effet, le client peut envoyer deux « demandes d'offre », l'une au provider1, l'autre au provider2, et ainsi tenir le rôle d'émetteur d'événements. Il peut cependant également devenir le récepteur de l'événement « livraison du produit » vis-à-vis du provider2.

3.7 Comparaison, classification et conclusion [à compléter]

Tout ces formalismes pour le moins hétéroclites ne sont pas tous utiles dans l'utilisation d'un pattern. Pour mieux comprendre l'essence d'un pattern, et ainsi pouvoir l'intégrer à un case tool, il convient connaître au mieux sa structure et son comportement. C'est ce à quoi la suite va s'employer.

Chaque pattern est une solution à un problème à part entière ; il n'est donc pas possible de traiter tous les patterns de la même façon. L'étude de chaque cas ainsi que la prise en compte des modélisations proposées permet de mieux cerner les problèmes, mieux connaître les acteurs concernés, leurs rôles et leurs comportements. La prise en considération de tous ces paramètres offre la possibilité d'intégrer correctement un pattern dans un case tool. Le tableau ci-dessous renseigne les comparaisons entre modélisations et donc celles entre les patterns.

		structure	comportement	rôles	facilité d'implémentation	compréhension intuitive	Application multiple du pattern
UML	diagramme de classe	***	*	*	**	*	*
UML	diagramme de collaboration	*	***	**	**	**	*
LePus		***	*	*	**	o	***
Litteral Form		*	**	**	o	***	**
Role Modeling / Collaboration		*	***	***	**	**	***
Role Modeling (Reihle form)		**	*	***	*	**	*

*** très bon   ** bon   * moyen   o passable

Figure 26 : Tableau comparatif des différentes modélisations

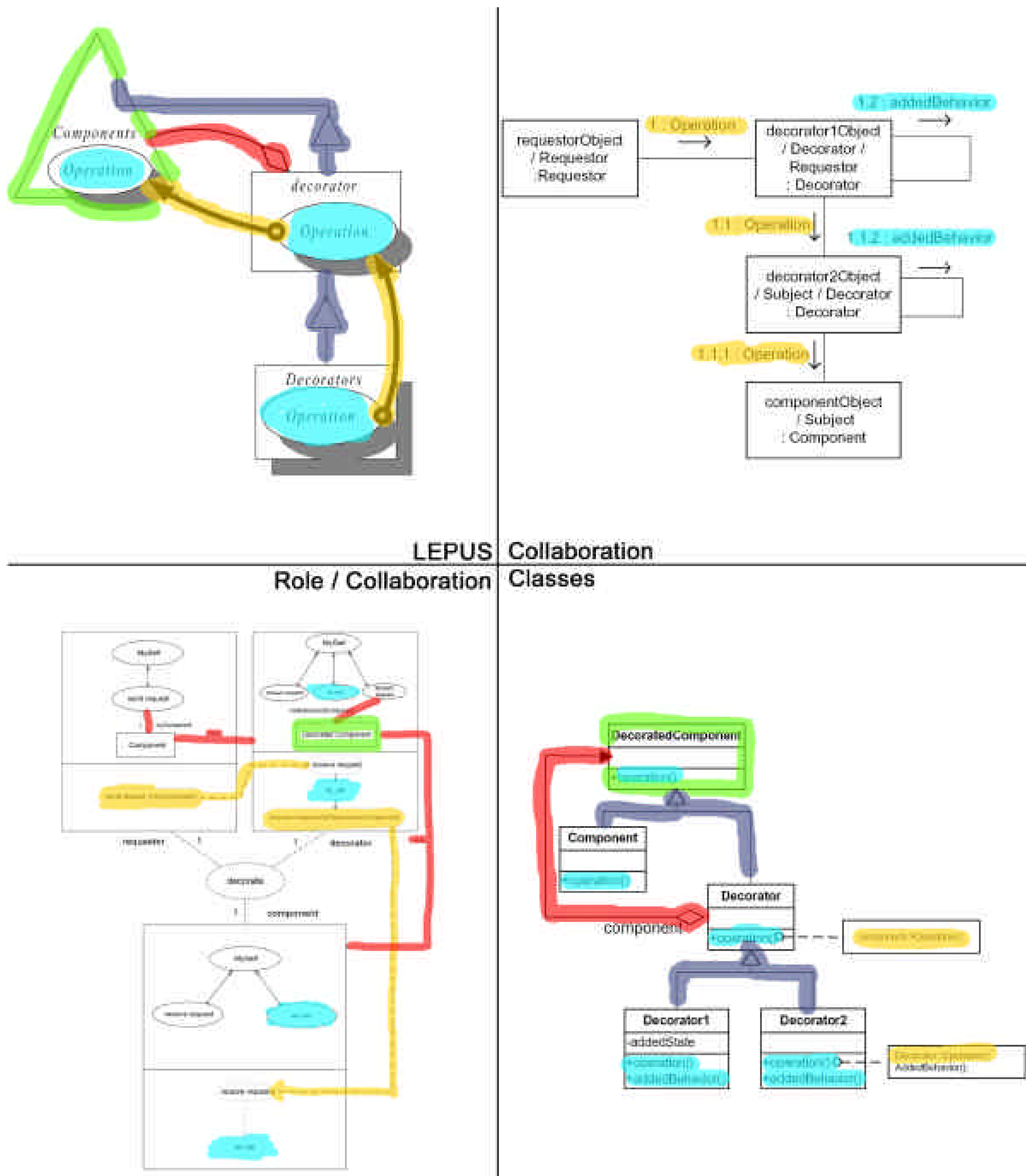


Figure 27 : Comparatifs des différentes modélisations d'éléments semblables

2 types de diagramme peuvent être relevés :

Ceux indispensables, à savoir les diagrammes de classe, de collaborations (role modeling et UML), qui apportent une vue nécessaire à la compréhension du pattern.

Ceux complémentaires (les autres) qui apportent une vue nouvelle, utile à la compréhension du pattern.

Il ressort que notre proposition de modélisation est parfaitement complétée par un diagramme de classe. A eux 2, ils couvrent l'intégralité des besoins.

4 Intégration des design patterns dans les cases tools

4.1 Intégration des patterns dans TogetherJ

4.1.1 Présentation de TogetherJ

TogetherJ est un outil d'aide au développement de logiciel. Grâce à ses multiples possibilités, la création d'un design compatible UML est aisée.

Il est un des outils les plus utilisés dans l'industrie de la production informatique.

Il possède une interface permettant de dessiner toute sorte de diagramme UML, et de générer par la suite, du code Java correspondant.

L'objectif n'est pas un exposé sur TogetherJ, mais seulement de montrer :
- La manière dont il utilise les designs patterns.
- Quelles sont les différents design patterns disponibles.
- les qualités et les défauts de la partie « pattern » de ce soft.
- Ce qu'il faudrait améliorer.

Des informations complémentaires sur ce produit sont disponibles sur le site du concepteur : http://www.togethersoft.com/

4.1.2 Quelles sont les différents patterns disponibles

Une multitude de patterns sont disponibles dans together, du simple canevas d'une applet (Une classe avec ces méthodes init() , start() , stop()…) au composant EJB, en passant par nos usuelles decorator, observer du GoF.

Composant Java :
- Bean
- Servlet
- Applet
- Main Class
- Reference http servlet
- Std Exception

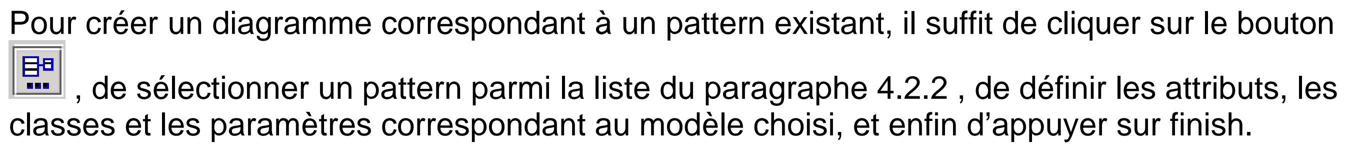
Pattern du GoF :
- Abstract Factory
- Adapter
- Chain of responsability
- Composite
- Decorator
- Factory Method
- Observer
- Proxy
- Singleton
- State
- Visitor

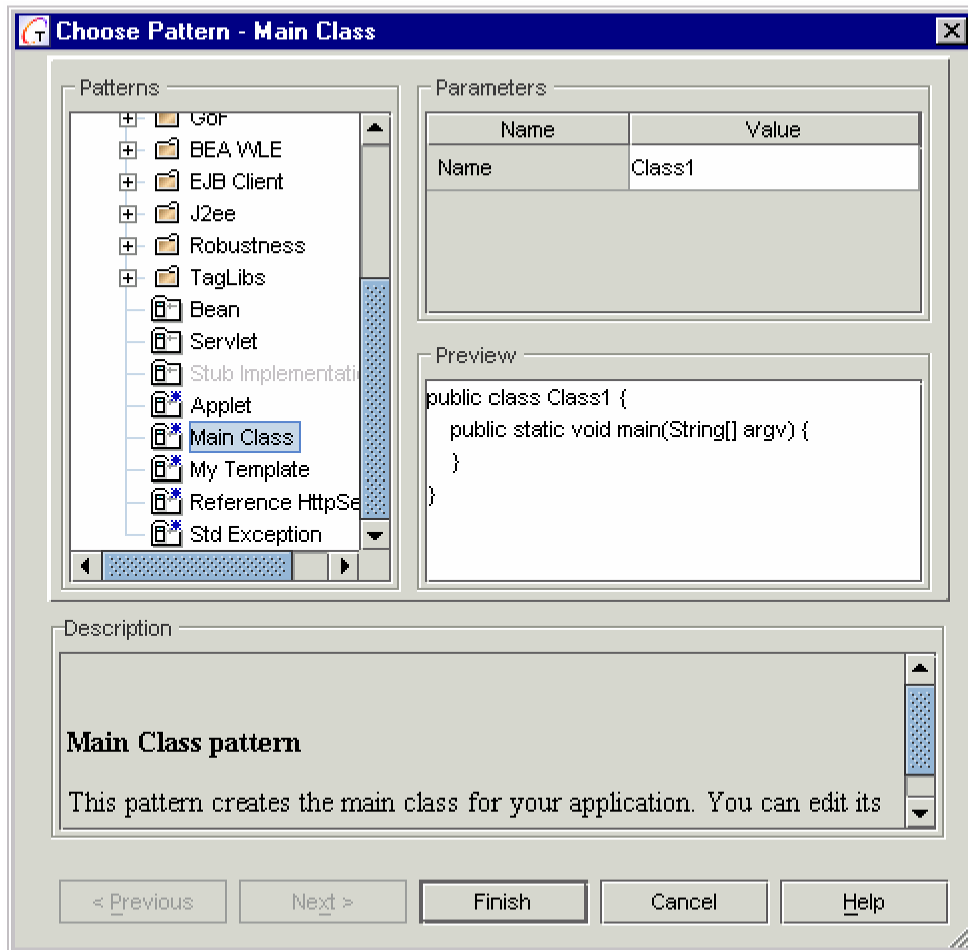
Autres pattern dédiés :

- Coad components :
    - domain neutral component
    - The four archetypes
- EJB clients (IBM, J2EE)
- HP e-speak
    - client
    - service provider
- J2EE
- JMS (Queue , Topic)
- Oracle 9i
- SQLJ object type
- Java center J2EE
    - Business tiers
    - Integration tiers
    - Presentation tiers
- XP
    - test case
    - test package
    - test proxy
- Coad classes
    - Description
    - MomentInterval from ArchetypesInColor
    - PartyPlaceThing
    - Role
- Robustness
    - Boundary class
    - Controler class
    - Entity class
    - Worker boundary class
    - Worker controler class
- Taglibs
    - BodyTag
    - BodyTagSupport
    - Tag
    - TagExtraInfo
    - Tag Support

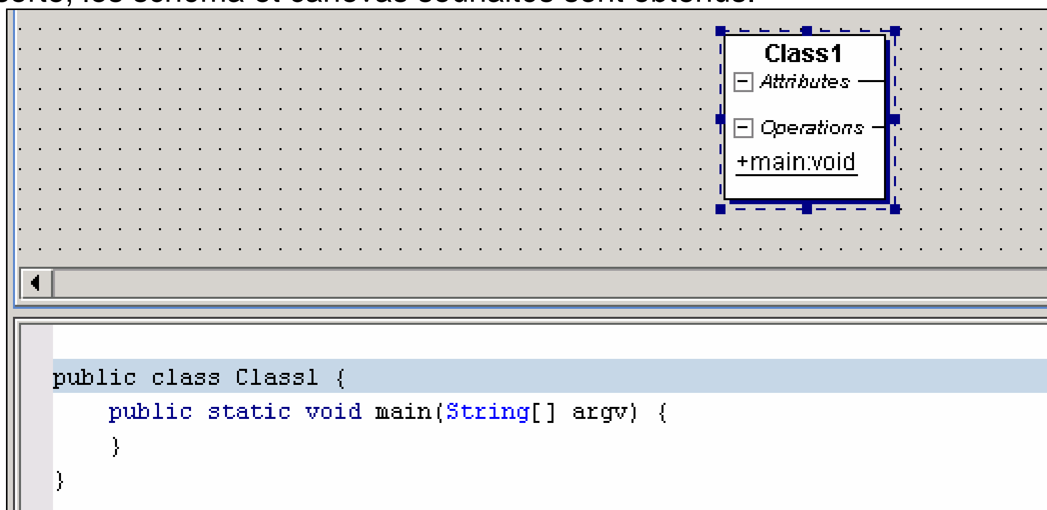
4.1.3 Comment les patterns sont-ils présentés et intégrés ?

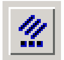
Together propose 2 manières d'intégrer des patterns, suivant que l'utilisateur désire créer un diagramme correspondant a un pattern existant ou qu'il souhaite relier 2 classes avec un pattern.
Les patterns ne sont donc pris en compte qu'au travers du diagramme de classe, l'outil étant donc totalement inefficace en matière de collaboration et de comportement, ce qui peut être regretté.

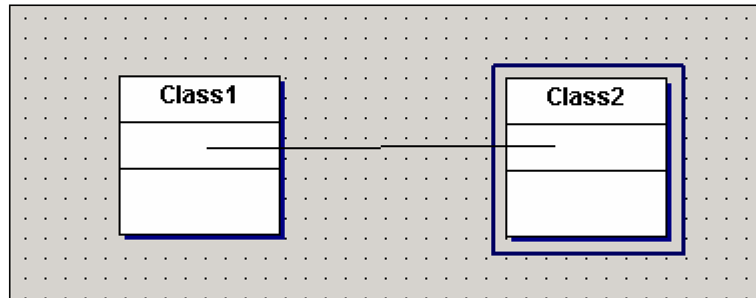
Pour créer un diagramme correspondant à un pattern existant, il suffit de cliquer sur le bouton  , de sélectionner un pattern parmi la liste du paragraphe 4.2.2 , de définir les attributs, les classes et les paramètres correspondant au modèle choisi, et enfin d'appuyer sur finish.



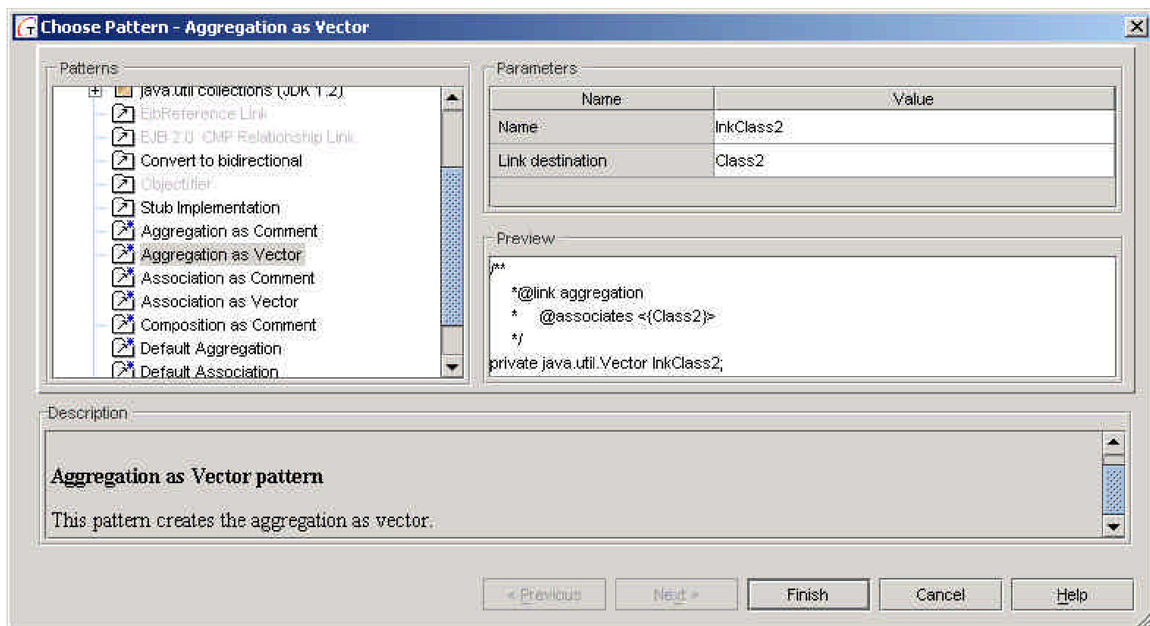
De la sorte, les schéma et canevas souhaités sont obtenus.



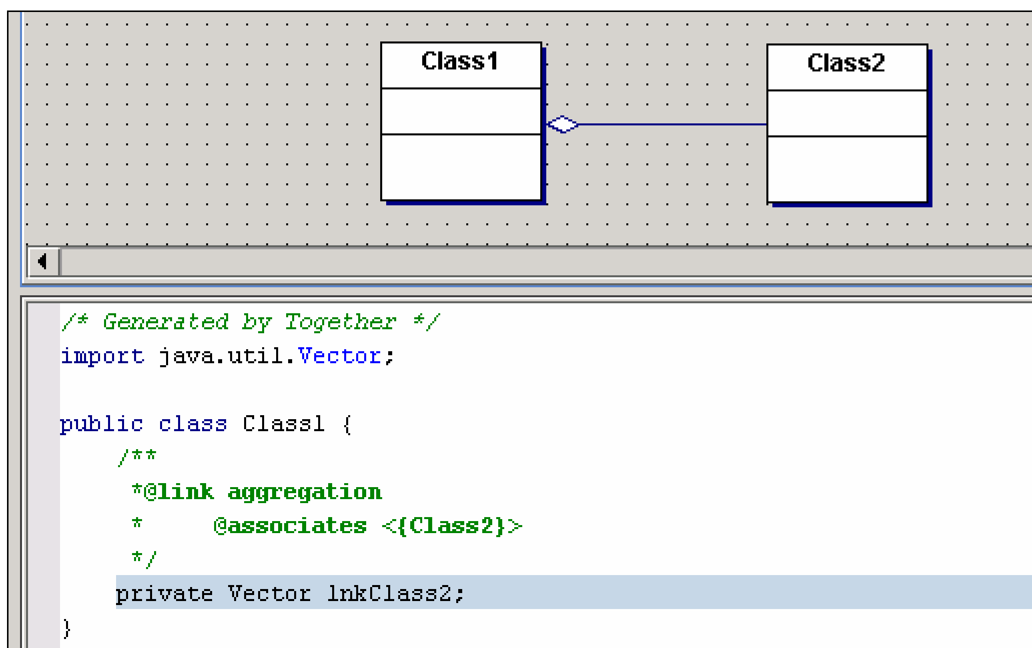
Pour relier 2 classes existantes à l'aide d'un pattern, il suffit de cliquer sur le bouton  , et de sélectionner les deux classes à associer.



Puis de sélectionner le pattern parmi la liste du paragraphe 4.2.3, de le nommer et de le paramétrer, et enfin d'appuyer sur finish.



Ceci afin d'obtenir les schéma et canevas souhaités.



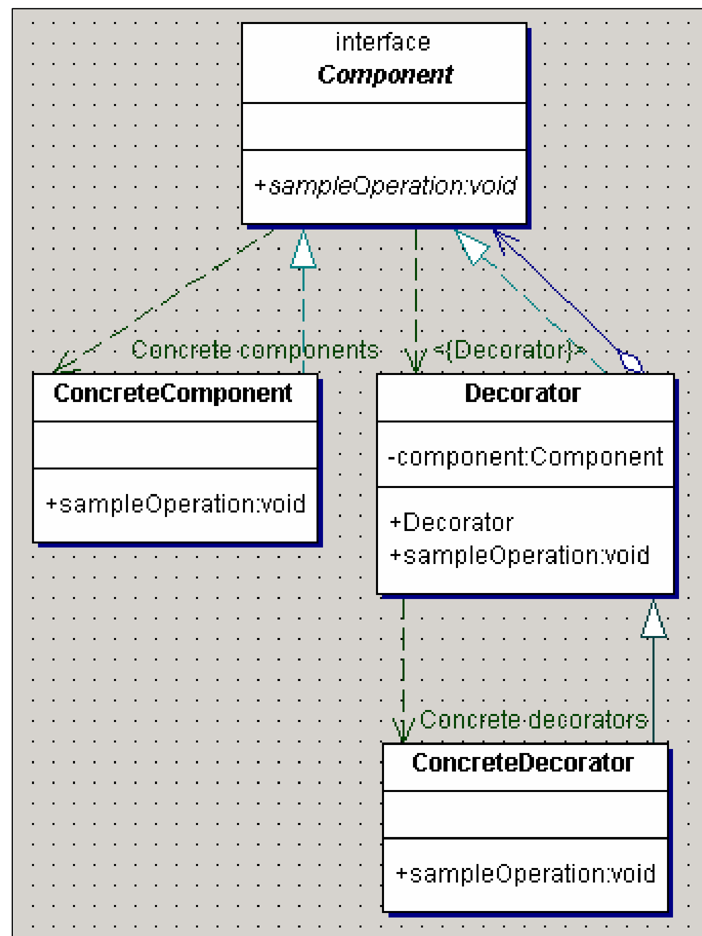
4.1.4 Exemple du décorateur :

Pour mieux comprendre le fonctionnement des patterns dans together, nous proposons un exemple déjà étudié dans les chapitres précédents : le décorateur.
Cette démarche permettra une meilleure critiquer de l'approche des patterns dans together.

Il convient dans un premier temps d'étudier les possibilités offertes par together, puis de pousser l'étude en intégrant un pattern dans un design existant.

4.1.4.1 Possiblité offerte par together :

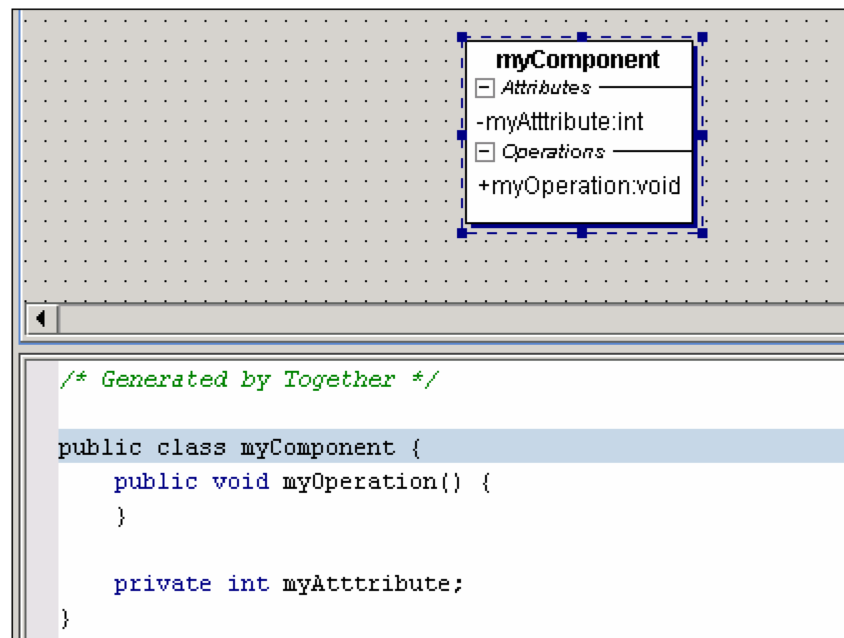
Partant d'une page blanche, il s'agit tout d'abord d'insérer le pattern « decorator » tel que réalisé dans la section 4.5.3, pour obtenir le diagramme ci-dessous :



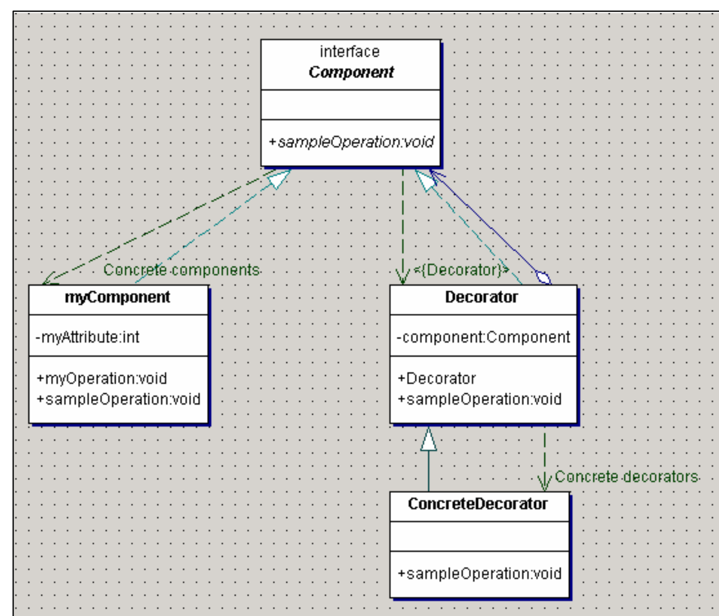
Together offre la possibilité de générer les classes correspondant à un décorateur, retrouvant ainsi la structure telle qu'elle est décrite dans l'œuvre du GoF.
Cette vue est modifiable ; cependant, rien n'est automatisé, et il y a lieu de maîtriser la structure et le fonctionnement du pattern pour pouvoir l'améliorer. En effet, rien n'empêche de faire quelque chose d'illicite !.

4.1.4.2 *Intégration d'un pattern dans un design existant*

En partant d'une classe existante « myComponent » (possédant un attribut « myAttribute » et une méthode « myOpération() ») que nous souhaiterions décorer,

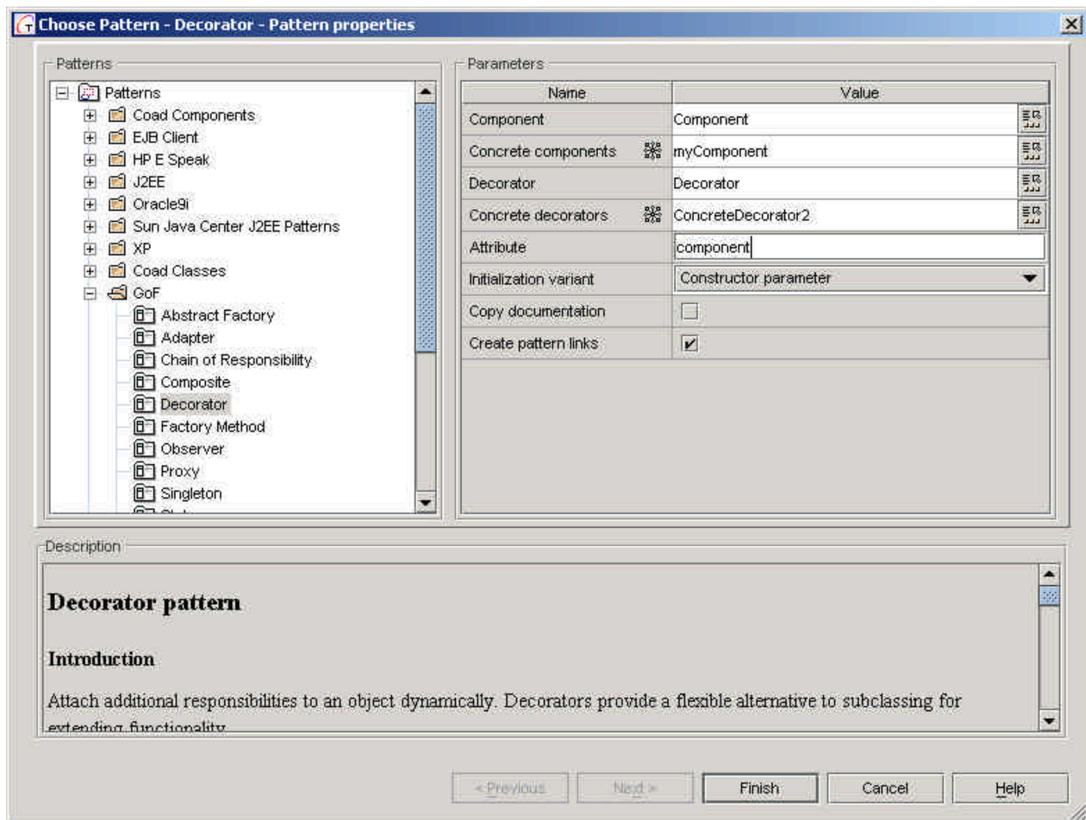


nous insérons le pattern de la manière décrite dans le point précédent, tout en précisant que le concrete component est le composent existant.

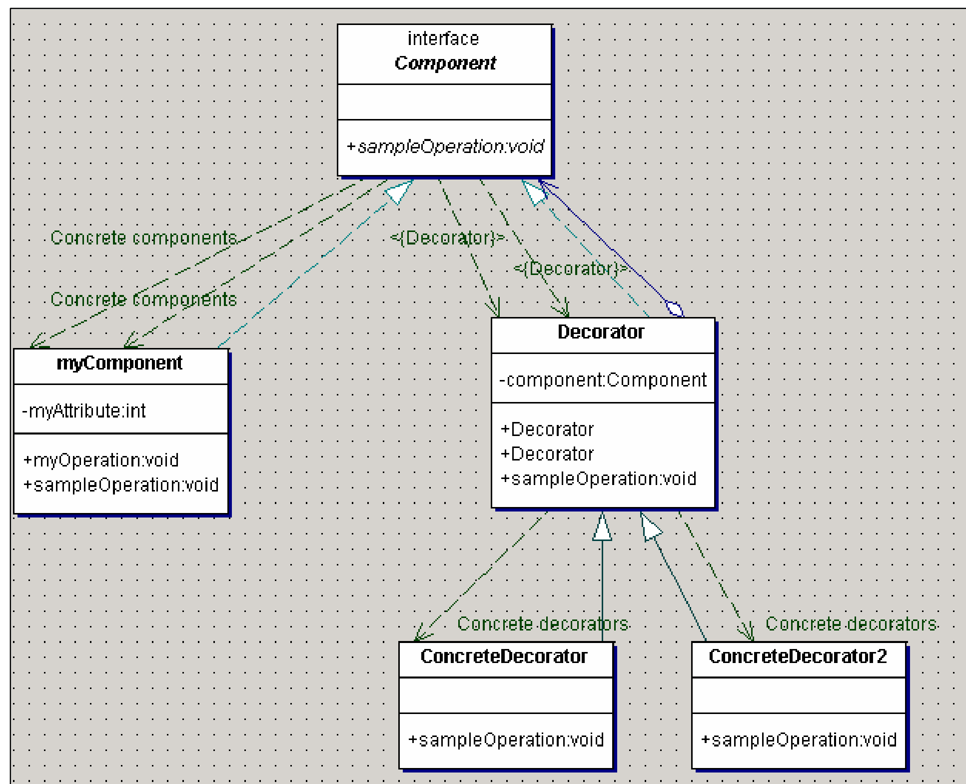


Il apparaît que together n'a procédé qu'à une substitution de la classe ConcreteComponent par la classe myComponent, sans prendre en compte les spécificités de cette dernière. C'est donc au développeur que revient cette tache parfaitement automatisable.

De la même manière, nous insérons un deuxième décorateur. Pour cela, nous configurons les différents attributs du nouveau pattern en conséquence, en prenant soin de faire correspondre les classes du nouveau pattern avec les classes existant.



Il en résulte le schéma suivant :



Plusieurs problèmes interviennent, en plus de ceux précité. En effet, les références {Decorator} et ConcreteComponent on été créé une seconde fois. Il faut donc manuellement les supprimer.

A noter que si nous avions également modélisé le « requestor » de notre composent, il aurait également fallut changer la référence qui allait vers myComponent en une référence vers l'interface Component.

4.1.5 Conclusion

Les différentes manipulations nous montrent les enseignements suivants :

- TogetherJ possède une base de données de diagramme de classe correspondant aux patterns trouvés dans la littérature.
- TogetherJ n'a aucun système « intelligent » concernant les patterns. Il est donc
    - Incapable de cacher certaines spécificités des patterns inutiles à l'utilisateur.
    - Incapable de synthétiser des patterns ensembles ou un pattern avec un design existant.
    - Incapable de reconnaître un pattern qu'il a créé.
- TogetherJ a modélisé ce qui existait dans la littérature, il a donc reproduit les mêmes erreurs, c'est-à-dire une absence complète de description des comportements.

L'utilisation des patterns dans TogetherJ n'est utile qu'à ceux qui ont de solides connaissances dans le domaine. L'exemple du décorateur montre immédiatement les limites de l'outil. La diversité des patterns en termes de structure, de comportement, de niveau formalisation, montre que le parti pris par TogetherJ (modéliser uniquement la structure de chaque pattern) est voué à l'échec.

Il convient donc de s'interroger :
« Est-il possible de créer un outil de développement capable de maîtriser l'essence même de chaque pattern »

En d'autres termes :
« Peut on réaliser une application en 3 clics ».

Ma réponse est « non ». A mon sens, aucun outil ne pourra remplacer la réflexion de l'ingénieur, qui détient seul les clés des solutions à son problème. Au mieux, un système d'aide à l'intégration de pattern est envisageable, de même qu'un système expert capable de poser les questions pertinentes pour épouser au plus prés les attentes de l'utilisateur.

4.2 Proposition d'un case tool idéal

Nous vous proposons notre vision des choses à travers cette simulation d'un case tools idéal, utilisant la méthode du point 3.2.2 et intégrant les patterns.

Consultez la à l'url : http://diwww.epfl.ch/~fbois/pattern

En ce qui concerne les patterns, cette proposition prend en compte les spécificités de la méthode que nous avons présenté afin de :

- Prendre en compte le comportement et non seulement la structure.
- Donner la possibilité à l'utilisateur de synthétiser deux instances d'un même pattern ou deux pattern différents.
- Différencier les patterns du design général de l'application.
- Donner la possibilité de masquer l'intérieur d'un pattern. et ainsi proposer plusieurs niveaux d'abstraction.

5 Conclusion

Lors de l'élaboration de ce projet, nous avons pu apprécier le formidable engouement suscité par les designs patterns. Nous avons également pu constater la multitude et la diversité des patterns proposés, à la lumière des nombreux articles, papiers et ouvrages. Est-ce pour autant une solution miracle a tout les problèmes de conception ? Ou plutôt une ressource supplémentaire d'information ? Une chose apparaît certaine, les patterns permettent d'unifier la discussion entre concepteurs ; A ce titre, il est opportun de ne pas les négliger.

Cependant le problème de la modélisation se pose. D'OMT à UML, un nombre considérable de formalisations similaires existent, et aucune ne satisfait pleinement les exigences multiples engendrées par la diversité des solutions proposées. Certaines modélisations sont pleinement adaptées à certains types de pattern, et inutilisables dans d'autres cas.

Nous avons par ailleurs pu distinguer deux grands types de patterns ; ceux plutôt comportementaux, et ceux structurels. Chaque pattern possédant des exigences diverses en matière de modélisation, nous avons donc opté pour une solution mixte, mettant en jeu aussi bien une description des comportements qu'une modélisation de la structure de chaque acteur. Cette méthode a fait ses preuves sur les deux cas que nous avons étudiés en profondeurs, et nous pouvons raisonnablement penser qu'elle est applicable à la majorité des patterns existants.

Cette théorie semble malgré tout inutile si elle ne permet pas des applications concrètes ; par exemple dans l'industrie de la production logiciel, un outil de développement capable d'intégrer les designs patterns, leurs structures, leurs comportements et leurs caractéristiques. Un outil qui serait non seulement capable d'intégrer une solution dans une conception existante, mais également de reconnaître un pattern, de le combiner avec d'autres et de le synthétiser sans limite. Nous avons imaginé les contours d'un tel outil, reste à savoir comment serait-il perçu par la communauté des développeurs.

Références

1. *Object-Oriented Application Frameworks* – M. Fayad and D. Schmidt.
2. *Design Patterns : Elements of Object-Oriented Software.* – E.Gama , R. Helm, R. Johnson, and J. Vlissides.
3. *Analysis Patterns, reusable Object Models* – M. Fowler
4. *Business modeling with UML, business pattern at work* – H.E. Eriksson and M. Penker
5. *Pattern-Oriented Analysis and Design (POAD) : A structural Composition Approach to Glue Design Patterns* - S. M. Yacoub and H. H. Ammar
6. *Role Model Based Framework Design and Integration* - D. Riehle and T. Gross
7. *A Role-BasedDesign Pattern Catalog ofAtomic and Composite PatternsStructured by Pattern Purpose – D. Riehle*
8. *Precise Modeling of Design Patterns* – A. Le Guennec, G. Sunyé and J.M. Jézéquel
9. *Formalizing the Intent of Design Patterns* – A. H. Eden, A. Gustavsson and M. Ersson
10. *Design Patterns Application in UML* - A. Le Guennec, G. Sunyé and J.M. Jézéquel
11. *Formalising Design Patterns* – K Lano and S Goldsack
12. *Precise Visual Specification of Design Patterns* – A. Lauder and S. Kent
13. *Precise Specification of Design Patterns and tool support in their application* – A. Eden
14. *LePUS –A Declarative Pattern Specification Language* – A. H. Eden , Y. Hirshfeld and M. Yehudai

Remerciements

A Mr Alain Wegmann, professeur au laboratoire de modélisation systémique de l'EPFL.
A Mr Pavel Balabko, doctorant au laboratoire de modélisation systémique de l'EPFL.