Analyse, Conception Objet

Design Patterns
Introduction

O. Boissier, SMA/G2I/ENS Mines Saint-Etienne, Olivier.Boissier@emse.fr, **Avril 2004**

Sommaire

- ✓ **Conception**
- Réutilisabilité
  - Bibliothèque de classe vs. Framework
- Design Pattern
- Historique
- Catégories de Patterns
- Bibliographie

Conception

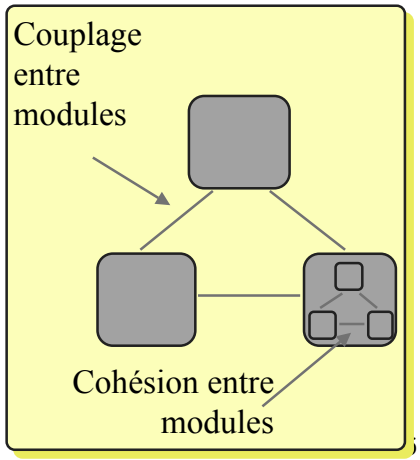
- Face à des problèmes complexes, rechercher la modularité afin d'obtenir un ensemble de modules plus simples et plus facilement gérables.
- Dans le monde des objets, la notion de ''modules'' se décline en :
  - Classes
    - abstractions des données et des services
  - Méthodes
    - Implémentation des services
  - Packages / sous-systèmes
    - Groupement de classes dans une seule collection
  - Processus physiques (client/serveur, …)

Deux questions sur la conception de modules

- Cohésion : degré avec lequel les tâches réalisées par un seul module sont fonctionnellement reliées
  - Quel est le liant d 'un module ?
  - Quel est son objectif ?
  - Fait-il une ou plusieurs choses ?
  - Quel est sa fonction ?
- Couplage : force de l'interaction entre les modules dans le système
  - Comment est-ce que les modules travaillent ensemble ?
  - Qu'ont-ils besoins de connaître l ''un sur l 'autre ?
  - Quand font-ils appel aux fonctionnalités de chacun ?

Cohésion et Couplage

- ✓Cohésion (relations logiques entre items d 'un objet)
  - ✓ Etendue de la responsabilité d 'un module (une seule/plusieurs)
  - ✓ Informelle (définie en terme d 'objectif)
  - ✓ Une forte cohésion est une bonne qualité
- ✓Couplage (force des relations physiques entre items d 'un objet)
  - ✓ Dépendance entre les modules
  - ✓ Peut-être définie formellement
  - ✓ Un couplage "lâche" est une bonne qualité



Cohésion "mauvais" exemple

Class GameBoard {

public GamePiece[ ][ ] getState()
- Méthode copiant la grille dans un tableau temporaire, résultat de l 'appel de la méthode.
public Player isWinner()
- vérifie l'état du jeu pour savoir s 'il existe un gagnant, dont la référence est retournée. Null est retourné si aucun gagnant.
public boolean isTie()
- retourne true si aucun déplacement ne peut être effectué, false sinon.
public void display ()
- affichage du contenu du jeu. Espaces blanc affichés pour chacune des références nulles.

GameBoard est responsable des règles du jeu et de l 'affichage.

}

Cohésion "bon" exemple

Class GameBoard {

public GamePiece[ ][ ] getState()
- -
public Player isWinner()
- -
public boolean isTie()
- -
}

class BoardDisplay {

public void displayBoard (GameBoard gb)
- affichage du contenu du jeu. Espaces blanc affichés pour chacune des références nulles.
}

Couplage Exemple

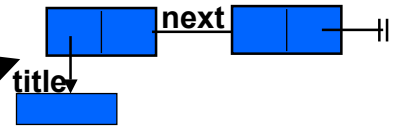
```
void initArray(int[ ] iGradeArray, int nStudents) {
        int i;
        for (i=0; i < nStudents; i++)  {
                iGradeArray[i] = 0;
}
```

Couplage entre client et initArray par le 2nd paramètre

```
void initArray(int[ ] iGradeArray) {
        int i;
        for (i=0; i < iGradeArray.length; i++)  {
                iGradeArray[i] = 0;
}
```

Couplage faible (et meilleure fiabilité) au travers de l 'utilisation de l 'attribut length

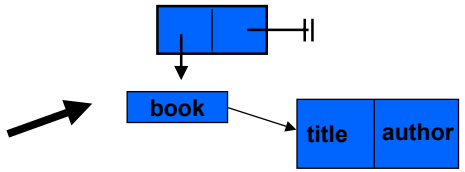
Couplage
Nœuds d'une classe Booklist




```
class BookNode  {

    private String title;
    private BookNode next;

    public BookNode (String
                        newTitle)  {
        title = newTitle;
        next = null;
    }

    public BookNode getNext () {
        return (next);
    }

    public void setNext (BookNode
                        nextBook) {
        next = nextBook;
    }

    public void print () {
        System.out.println (title);
    }

}//class BookNode
```

Que se passe-t-il
si nous ajoutons Author ?

Couplage
Nœuds d'une classe Booklist




```
class BookNode  {

    private Book book;
    private BookNode next;

    public BookNode (String title,
                     String author)  {
        book = new Book(title, author);
        next = null;
    }

    ...

}
```

```
class Book

    private String title;
    private String author;

    public Book(String title,
                String auth) {
    ...
    }

}//class Book
```


Principes de conception (1)

- Programmer une interface plus qu'une implémentation,
- Utiliser des classes abstraites (interfaces en Java) pour définir des interfaces communes à un ensemble de classes,
- Déclarer les paramètres comme instances de la classe abstraite plutôt que comme instances de classes particulières.

Ainsi :

- les classes clients ou les objets sont indépendants des classes des objets qu'ils utilisent aussi longtemps que les objets respectent l 'interface qu'ils attendent,
- les classes clients ou les objets sont indépendants des classes qui implémentent l 'interface.

Exemple
Tri d'étudiants (1) : "mauvais"

```
class StudentRecord {
    private Name lastName;
    private Name firstName;
    private long ID;
    public Name getLastName() {return lastName;}
    … // etc
}
 class SortedList {
     Object[] sortedData = new Object[size];
     public void add(StudentRecord X) {
            // …
            Name a = X.getLastName();
            Name b = sortedData[k].getLastName();
            if (a.lessThan(b)) … else … //do something
     } … }
```

Exemple
Tri d'étudiants (2) : "solution 1"

```
class StudentRecord {
    private Name lastName;
    private Name firstName;
    private long ID;
    public boolean lessThan(Object X) {
          return lastName.lessThan(X.lastName);}
    … // etc
}
 class SortedList {
     Object[] sortedData = new Object[size];
     public void add(StudentRecord X) {
           // …
           if (X.lessThan(sortedData[k])) … else … //do something
     } … }
```


Exemple
Tri d'étudiants (3) : "solution 2"

```
Interface Comparable {
    public boolean lessThan(Object X);
    public boolean greaterThan(Object X);
    public boolean equal(Object X);
}

 class StudentRecord implements Comparable {
     private Name lastName;
     private Name firstName;
     private long ID;
     public boolean lessThan(Object X) {
           return lastName.lessThan(((StudentRecord)X).lastName);}
     … // etc
}
```


Exemple
Tri d'étudiants (4) : "solution 2"

```
class SortedList {
    Object[] sortedData = new Object[size];
    public void add(Comparable X) {
          // …
          if (X.lessThan(sortedData[k])) … else … //do something
    }
    …
}
```


Principes de conception (2)

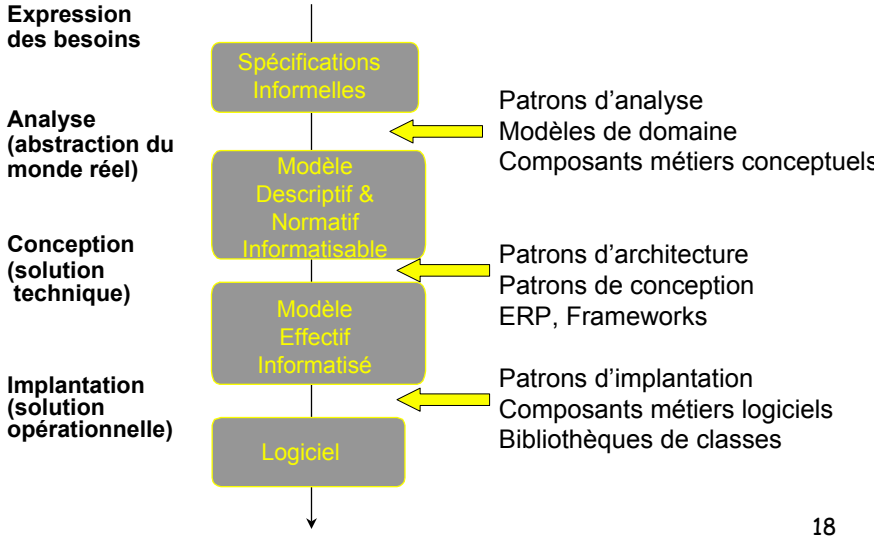
- Préférer la composition d'objet à l'héritage de classes

  Ainsi :
- le comportement peut changer en cours d 'exécution,
- les classes sont plus focalisées sur une tâche,
- réduction des dépendances d 'implémentation.

Sommaire

- Conception
- ✓ **Réutilisabilité**
  - ➢ **Bibliothèque de classe vs. Framework**
- ➢ Design Pattern
- ➢ Historique
- ➢ Catégories de Patterns
- ➢ Bibliographie

Réutilisation en Génie Logiciel

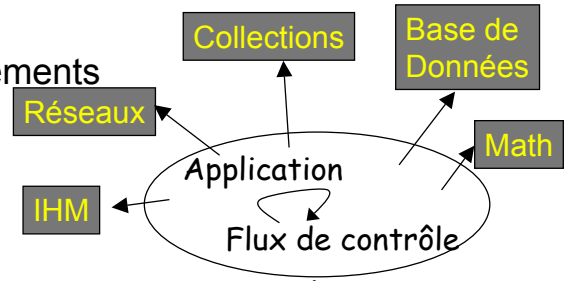


Réutilisation en Génie Logiciel

Analyse «métier commun» «métier spécifique»	Conception «architecture» «conception»	Implantation « idiome »	Documentation
- Le problème traité est issu d'une analyse de domaine - Aide à la construction de modèles objets descriptifs	- identifier, nommer et abstraire des thèmes récurrents de la conception par objet. - Aide à la construction de modèles objets effectifs	- comment implanter dans un langage particulier certains traits absents de ce langage	- technique de dialogue, de documentation, d'enseignement , etc.
Gestion des ressources Opérations Bancaires	Composite Observateur Blackboard	Simuler l'héritage multiple en Java	Documenter un framework

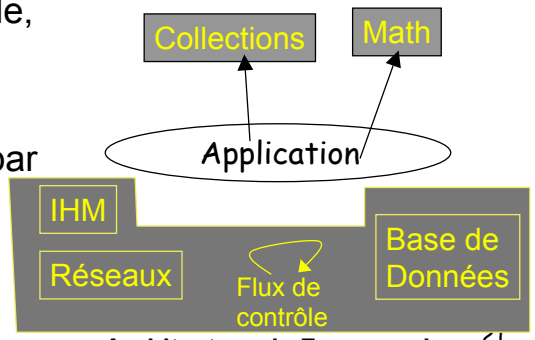
Bibliothèque de classe

- ➢ Ensemble de classes
  - ➢ le plus souvent abstraites
  - ➢ indépendantes (pas d 'interaction a priori entre elles)
  - ➢ sans comportements par défaut



Framework = squelette d'application

- Ensemble de classes :
  - concrètes ou abstraites conçues pour être utilisées ensemble,
  - en interaction,
  - fournissant des comportements par défaut



Architecture de Framework

Framework = squelette d'application (suite)

- ***Application framework*** : encapsultation d 'une expertise applicable à une large variété de programmes,
- ***Domain framework*** : encapsulation d 'une expertise dans un domaine de problèmes particulier
- ***Support framework*** : services de niveau système (accès fichier, application répartie, …)

Framework = squelette d'application (suite)

- ***Boîte blanche ou en verre*** :
  - fondé sur l 'héritage,
  - conçu par la généralisation de classes de différentes applications,
  - utilisé par dérivation de classes
- ***Boîte noire*** :
  - fondé sur la composition,
  - utilisé par composition des composants entre eux.

Sommaire

- Conception
- Réutilisabilité
  - Bibliothèque de classe vs. Framework

✓**Design Pattern**

- Historique
- Catégories de Patterns
- Bibliographie

Autres domaines : Architecture

C. Alexander : 253 patrons de conception architecturaux (64,77,79)

« Pattern #112 d'Alexander»

- **Nom** : Transition d'entrée
- **Quoi** : créer une transition du monde extérieur vers un univers intérieur, plus privé
- **Pourquoi** : L'entrée dans un bâtiment influence la façon dont on va se sentir à l'intérieur
- **Quand** : systèmes d'accès et d'entrée pour les maisons, les cliniques, les magasins, etc.
- **Comment** : Créer un espace de transition entre la rue et la porte d'entrée. Marquer le cheminement dans l'espace de transition par un changement de lumière, un changement de direction, etc.
- **Patterns en relation** : vue Zen (#134), etc.

tiré de «Introduction aux patterns» Jean Bézivin

Autres domaines : l'hydraulique

Ken Asplund en 1973

« Le volume d'eau en aval » (Downstream Water Volume)

Problème : Quand on détourne une partie des eaux d'une rivière, le volume d'eau dans le drainage décroît, ce qui conduit à un affaiblissement des contributions de la rivière.

Contraintes : Il ne faut pas que le détournement d'une partie de la rivière pénalise l'irrigation des cultures en aval ; la population ne doit pas souffrir de cette baisse du débit des eaux ; l'impact écologique doit être limité au minimum.

Solution : Il faut maintenir le niveau des eaux en aval en ramenant un maximum d'eau détournée vers son lit d'origine ; en traitant les eaux usées afin de les réintroduire en aval ; en économisant les eaux d'irrigation en évitant les systèmes de pulvérisation ou de brumisation.

Définition

Un patron décrit à la fois un **problème** qui se produit très fréquemment dans votre environnement et l'architecture de la **solution** à ce problème de telle façon que vous puissiez **utiliser** cette solution des milliers de fois sans jamais l'**adapter** deux fois de la même manière.

C. Alexander

Décrire avec succès des **types de solutions** récurrentes

à des **problèmes** communs dans des **types de situations**

Design Pattern

- **Coad** [Coad92]

Une abstraction d'un doublet, triplet ou d'un ensemble de classes qui peut être réutilisé encore et encore pour le développement d'applications

- **Appleton** [Appleton97]

Une règle tripartite exprimant une relation entre un certain contexte, un certain problème qui apparaît répétitivement dans ce contexte et une certaine **configuration logicielle** qui permet la résolution de ce problème.

- **Aarsten** [Aarsten96]

Un groupe d'objets coopérants liés par des relations et des règles qui expriment les liens entre un contexte, un problème de conception et sa solution.

Les patrons sont des composants **logiques** décrits indépendamment d'un langage donné (solution exprimée par des modèles semi-formels)

Design Pattern (suite)

- Documentation d 'une expérience éprouvée de conception
- Identification et spécification d 'abstractions qui sont au dessus du niveau des simples classes, instances
- Vocabulaire commun et aide à la compréhension de principes de conception,
- Moyen de documentation de logiciels
- Aide à la construction de logiciels répondant à des propriétés précises, de logiciels complexes et hétérogènes

Design Patterns / Frameworks

- Synergie entre ces deux concepts
- les design patterns décrivent à un niveau plus abstrait des composants de frameworks, implémentés dans un langage particulier,
- les design patterns aident à la conception des frameworks,
- la conception de frameworks aident à la découverte de nouveaux design patterns.

Présentation d'un pattern

- **Nom du pattern**
  - utilisé pour décrire le pattern, ses solutions et les conséquences en un mot ou deux
- **Problème**
  - description des conditions d 'applications. Explication du problème et de son contexte.
- **Solution**
  - description des éléments (objets, relations, responsabilités, collaboration) permettant de concevoir la solution au problème ; utilisation de diagrammes de classes, de séquences, … vision statique ET dynamique de la solution
- **Conséquences**
  - description des résultats (effets induits) de l 'application du pattern sur le système (effets positifs ET négatifs)

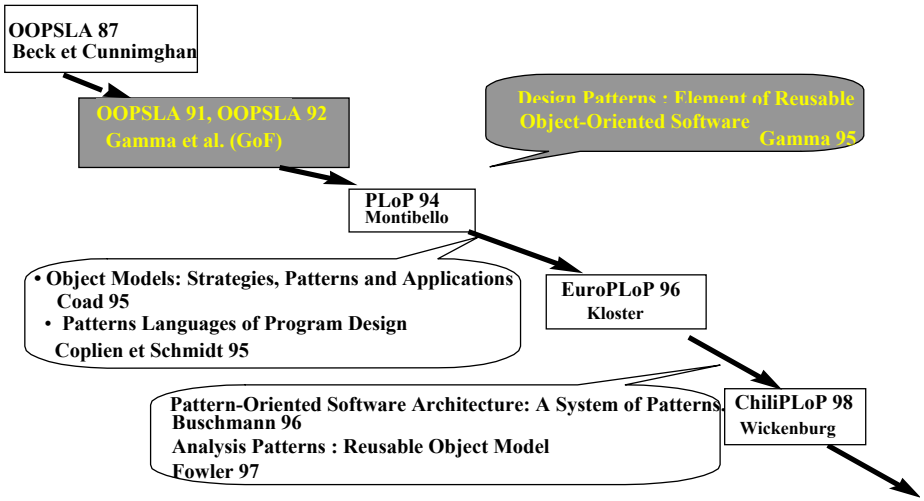
Sommaire

- Conception
- Réutilisabilité
  - Bibliothèque de classe vs. Framework
- Design Pattern

✓ **Historique**

- Catégories de Patterns
- Bibliographie

Historique



Sommaire

- Conception
- Réutilisabilité
  - Bibliothèque de classe vs. Framework
- Design Pattern
- Historique

✓**Catégories de Patterns**

➢Bibliographie

Catégories de Patterns

- Architectural Patterns :
  - schémas d 'organisation structurelle de logiciels (pipes, filters, brokers, blackboard, MVC, …)
- Design Patterns :
  - caractéristiques clés d 'une structure de conception commune à plusieurs applications,
  - de portée plus limitée que les architectural patterns
- Idioms ou coding patterns
  - solution liée à un langage particulier
- Anti-patterns :
  - mauvaise solution ou comment sortir d 'une mauvaise solution
- Organizational patterns :
  - organisation de tout ce qui entoure le développement d 'un logiciel (humains)

Catégories de Design Patterns

- **Création**
  - description de la manière dont un objet ou un ensemble d 'objets peuvent être créés, initialisés, et configurés : isoler le code relatif à la création, l 'initialisation afin de rendre l 'application indépendante de ces aspects.
- **Structure**
  - description de la manière dont doivent être connectés des objets de l 'application afin de rendre ces connections indépendantes des évolutions futures de l 'application : découplage de l 'interface et de l 'implémentation de classes et d 'objets
- **Comportement**
  - description de comportements d 'interaction entre objets : gestion des interactions dynamiques entre des classes et des objets.

Portée des Design Patterns

- Portée Classes
  - Focalisation sur les relations entre classes et leurs sous-classes
  - Réutilisation par héritage
- Portée Objets
  - Focalisation sur les relations entre les objets
  - Réutilisation par composition

Vue d'ensemble des Design Patterns

		but		
		Création	**Structure**	**Comportement**
Portée	**Classe**	Factory method	Adapter	Interpreter
				Template Method
	Objet	Abstract Factory	Adapter	Chain of responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Design Patterns de création

- **Abstract Factory** : interface pour la création de familles d'objets sans spécifier les classe concrètes.
- **Builder** :séparation de la construction d'objets complexes de leur représentation afin qu'un même processus de construction puisse créer différentes représentations.
- **Factory Method** :définition d'une interface pour la création d'objets associées dans une classe dérivée.
- **Prototype** :spécification des types d'objet à créer en utilisant une instance prototype.
- **Singleton** : comment assurer l'unicité de l'instance d'une classe.

Design Patterns de structure

- **Adapter** : traducteur adaptant l'interface d'une classe en une autre interface convenant aux attentes des classes clientes.
- **Bridge** :découplage de l'abstraction de l'implémentation pour faire varier les deux indépendamment.
- **Composite** : structure pour la construction d'agrégations récursives.
- **Decorator** : extension d'un objet de manière transparente.
- **Facade** : unification de plusieurs interfaces de sous-systèmes.
- **Flyweight** : partage efficace de plusieurs objets.
- **Proxy** : approximation d'un objet par un autre.

Design Patterns de comportement (1)

- **Chain of Responsibility** :délégation des requêtes à des responsables de services.
- **Command**: encapsulation de requêtes par des objets afin de permettre à un objet de traiter plusieurs types de requêtes.
- **Interpreter** : étant donné un langage, représentation de la grammaire le définissant pour l'interpréter.
- **Iterator** : parcours séquentiel de collections.
- **Mediator** : coordination d'interactions entre des objets associés.
- **Memento** : capture et restauration d 'états d'objets.

Design Patterns de comportement (2)

- **Observer** :mise à jour automatique des dépendants d'un objet.
- **State** : permettre à un objet de modifier son comportement lorsque son état interne change.
- **Strategy** : abstraction pour sélectionner un algorithme parmi plusieurs.
- **Template method** :définition d'un squelette d'algorithme dont certaines étapes sont fournies par une classe dérivée.
- **Visitor** :représentation d'opérations devant être appliquées à des éléments d'une structure hétérogène d'objets.

Causes de reconception (1)

- Création d'un objet par la spécification d 'une classe explicite
  - Abstract factory, Factory Method, Prototype
- Dépendances entre opérations spécifiques
  - Chain of Responsibility, Command
- Dépendances par rapport aux plate-formes matérielles et logicielles :
  - Abstract factory, Bridge
- Dépendances par rapport aux représentations et implémentation des objets :
  - Abstract factory, Bridge, Memento, Proxy

Causes de reconception (2)

- Dépendances algorithmiques :
  - Builder, Iterator, Strategy, Template Method, Visitor
- Couplage important :
  - Abstract factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer
- Accroissement des fonctionnalités par dérivation :
  - Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Impossibilité de modifier des classes facilement :
  - Adapter, Decorator, Visitor

Memento

- Lorsque l 'on utilise les Design Patterns, se souvenir que :
  - ce ne sont que des suggestions,
  - qu'ils pointent **vers** des solutions,
  - qu'ils sont implémentés différemment selon les langages de programmation,
  - qu'ils sont rarement utilisés seuls mais plutôt combinés avec d 'autres patterns,
  - que les solutions ainsi conçues sont généralement plus génériques et plus facilement réutilisables.

Anti Patterns

- Représentent une « leçon apprise »
- Erreurs logicielles que les gens font fréquemment.
- Deux catégories :
  - mauvaise solution à un problème qui a produit une mauvaise situation.
  - comment sortir d'une mauvaise situation et comment poursuivre à partir de là vers une bonne solution.
- Référence : W.J. Brown & al (1998) *Anti Patterns - Refactoring Software, Architectures, and Projects in Crisis*, Wiley.

Exemple de schéma pour les anti-patterns

- « Background »
- Forme générale
- Symptômes et conséquences
- Causes typiques
- Exception connue
- Solution « réparée »
- Variations
- Exemple
- Solutions apparentées
- Applicabilité à d'autres points de vue ou échelles

Pattern architecturaux

- Schémas d'organisation structurelle fondamentaux pour les logiciels.
- Exemples : *Pipes and Filters, Broker, MVC, Micro-kernel*
- Références :
  - F.Buschmann & al., *Pattern-Oriented Software Architecture*, Wiley 1996.
  - R.Martin & al., *Pattern Language of Program Design 3*, Addison-Wesley 1998.

Patterns d'analyse

- Question de modélisation générale en regardant un problème particulier dans un domaine.
- Martin Fowler a proposé deux catégories de patterns :
  - « Analysis patterns »
  - « Supporting patterns »
- Référence :

  Martin Fowler, *Analysis Patterns - Reusable Object Models*, Addison-Wesley, 1997.

Exemples de patterns d'analyse

- « Accountability »
  - Le concept de comptabilité s'applique quand une personne ou une organisation est responsable d'une autre.
  - notion abstraite qui peut représenter plusieurs problèmes spécifiques, comportant des structures, des contrats, et du personnel.
  - *Party, organization hierarchies, organization structure etc.*

- « Observations and measurement »
  - Des quantités peuvent être utilisées en tant qu'attributs d'objets pour enregistrer de l'information sur eux.
  - Ces patterns montrent comment cette approche échoue et suggère des approches plus sophistiquées.
  - *Quantity, conversion ratio, compound units, measurement, observation, protocol, dual time record etc*.

- « Referring to Objects »
  - *Name, Identification scheme, object merge, object equivalence*
- « Planning »
  - *Proposed and Implemented Action, Suspension, Plan, Resource Allocation etc..*
- « Trading
  - *Contract, Portfolio, Quote, Scenario*

Exemples de patterns support

- « Layered Architecture for Information System »
  - *Two-tier architecture, Three-tier architecture, Presentation and Application Logic, Database interaction*
- « Application Facades »
  - Interface simplifiée à un modèle compliqué.
  - Responsable du choix et de l'organisation de l'information pour une présentation.

- « Patterns for Type Model Design Templates »
  - comment transformer un modèle de spécification implicite en un modèle explicite et une implémentation.
  - *Implementing Associations, Object Creation, Implementing Constraints* ….
- « Association patterns »
  - *Associative Type, Keyed mapping, Historic mapping.*

Sommaire

- Conception
- Réutilisabilité
  - Bibliothèque de classe vs. Framework
- Design Pattern
- Historique
- Catégories de Patterns

✓**Bibliographie**

Bibliographie

- Erich Gamma & al (1995) *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley.
- " Pattern Languages of Program Design ", Coplien J.O., Schmidt D.C., Addison-Wesley, 1995.
- " Pattern languages of program design 2 ", Vlissides, et al, ISBN 0-201-89527-7, Addison-Wesley
- " Pattern-oriented software architecture, a system of patterns ", Buschmann, et al, Wiley
- " Advanced C++ Programming Styles and Idioms ", Coplien J.O., Addison-Wesley, 1992.
- S.R. Alpert, K.Brown, B.Woolf (1998) *The Design Patterns Smalltalk Companion*, Addison-Wesley (Software patterns series).
- J.W.Cooper (1998), *The Design Patterns Java Companion*, http://www.patterndepot.com/put/8/JavaPatterns.htm.
- S.A. Stelting, O.Maasen (2002) *Applied Java Patterns*, Sun Microsystems Press.
- Communications of ACM, October 1997, vol. 40 (10).

Sites Web

- http://st-www.cs.uiuc.edu/users/patterns/patterns.html
- http://hillside.net/patterns/patterns.html

- Portland Pattern Repository
  - http://www.c2.com/ppr

- Ward Cunnigham's WikiWiki Web
  - http://www.c2.com/cgi/wiki?WelcomeVisitors
  - http://www.c2.com/cgi/wiki?PatternIndex