Programmation Objet Avancée

David Navarre

david.navarre@ut-capitole.fr

Objectifs

- Connaître les design patterns (patrons de conception) classiques
- Utiliser des design patterns pour améliorer la maintenabilité du code

Introduction aux Design Patterns

- Un concept pour augmenter la réutilisabilité du logiciel orienté-objet
- Donner des archétypes de solution à des problèmes récurrents dans la conception de logiciel
- Solutions issues de l'expérience dans des projets réels, validées par l'usage
  - « On n'invente pas un Pattern, on le découvre »

Inspiration Architecturale

- « Un pattern décrit un problème qui se présente fréquemment dans notre environnement, et décrit l'essence d'une solution à ce problème, de telle sorte que l'on puisse réutiliser cette solution un million de fois sans jamais la refaire deux fois de la même manière »
  - Christopher Alexander.

A pattern language, town, buildings, constructions, avec Sarah Ishikawa et Murray Silverstein, 1977, Oxford University Press

Bibliographie sur les Patterns

- Design Patterns, Elements of Reusable Object-Oriented Software
  - E.Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, le «Gang of Four», 1995
- A System of Patterns : Pattern-oriented software architecture
  - Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. A Stal, Wiley 1996
- Smalltalk-80: the language and its implementations
  - Goldberg, A. and Robson, D.. Addison Wesley; 1983.
- Java

Éléments d’un Pattern

- Nom du pattern
  - augmenter le vocabulaire du design, augmenter le niveau d'abstraction
- Description du problème
  - contexte d'application
- Solution
  - en termes d'un modèle OO générique
- Conséquences
  - avantages, inconvénients, compromis

Les patterns ne sont pas :

- Des bibliothèques de classes réutilisables
  - listes chaînées, arbres, …
- Des applications toutes faites
  - traitent uniquement un sous-problème bien défini
- Des classes génériques (templates)

Catégories de Patterns

- Créationnels
  - Solutions pour l'instanciation des objets
  - Ex : Singleton, prototype, factory, …
- Structurels
  - Solutions pour la composition des classes ou objets
  - Ex : Proxy, façade, Adapter, …
- Comportementaux
  - Solutions pour l'interaction et la répartition des responsabilités entre les classes ou objets
  - Ex : Iterateur, visiteur, state, MVC, …

Les design patterns de références

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsability
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

D’autres patterns et classification

10

Buschmann F., Henney K., Schmidt D.C., Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, Vol 4, Wiley Series, 2007

Architectural Patterns

- From Mud To Structure
  - Domain Model, Layers, Model-View-Controller, Presentation-Abstraction-Control, Microkernel, Reflection, Pipes and Filters, Shared Repository, Blackboard, Domain Object
- Distribution Infrastructure
  - Messaging, Message Channel, Message Endpoint, Message Translator, Message Router, Publisher-Subscriber, Broker, Client Proxy, Requestor, Invoker, Client Request Handler, Server Request Handler
- Event Demultiplexing and Dispatching
  - Reactor, Proactor, Acceptor-Connector, Asynchronous Completion Token
- Interface Partitioning
  - Explicit Interface, Extension Interface, Introspective Interface, Dynamic Invocation Interface, Proxy, Business Delegate, Facade, Combined Method, Iterator, Enumeration Method, Batch Method
- Component Partitioning
  - Encapsulated Implementation, Whole-Part, Composite, Master-Slave, Half-Object plus Protocol, Replicated Component Group
- Application Control
  - Page Controller, Front Controller, Application Controller, Command Processor, Template View, Transform View, Firewall Proxy, Authorization
- Concurrency
  - Half-Sync/Half-Async, Leader/ Followers, Active Object, Monitor Object

Architectural Patterns

- Synchronization
  - Guarded Suspension, Future, Thread-Safe Interface, Double-Checked Locking, Strategized Locking , Scoped Locking, Thread-Specific Storage, Copied Value, Immutable Value
- Object Interaction
  - Observer, Double Dispatch, Mediator, Command, Memento, Context Object, Data Transfer Object, Message
- Adaptation and Extension
  - Bridge, Object Adapter, Chain of Responsibility, Interpreter, Interceptor, Visitor, Decorator, Execute-Around Object, Template Method, Strategy, Null Object, Wrapper Facade,Declarative Component Configuration
- Modal Behavior
  - Objects for States, Methods for States, Collections for States
- Resource Management
  - Container, Component Configurator, Object Manager, Lookup, Virtual Proxy, Lifecycle Callback, Task Coordinator, Resource Pool, Resource Cache, Lazy Acquisition, Eager Acquisition, Partial Acquisition, Activator, Evictor, Leasing, Automated Garbage Collection, Counting Handle, Abstract Factory, Builder, Factory Method, Disposal Method
- Database Access
  - Database Access Layer, Data Mapper, Row Data Gateway, Table Data Gateway, Active Record

Ce qu’il faut retenir

13

Au moins au début

Il y a beaucoup de patterns et ce n’est pas fini

- On ne peut pas tous les retenir
- Mais on doit les avoir parcourus au moins une fois

- Certains patterns sont déjà implémentés dans certains langages ou bibliothèques, ou sont construits autour
  - .net : Observer, Reactor, Iterator …
  - js : Proactor, Prototype, Iterator …
  - Java : Reflection, Introspective interface, Iterator …

- Face à un problème, le souvenir d’un pattern peut faciliter la conception et l’implémentation

Pour la suite

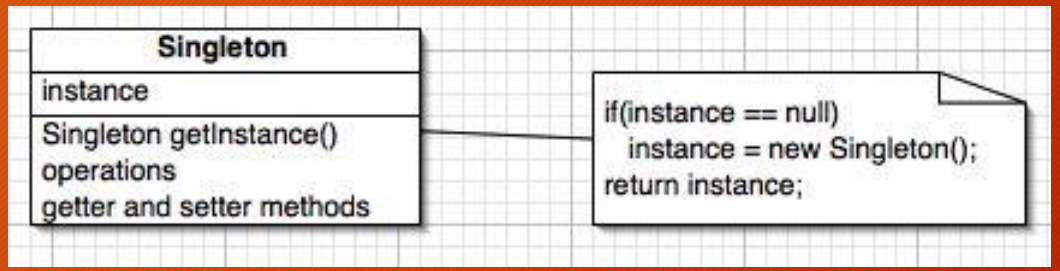
- On va revenir à la classification d'origine
  - Créationnel, Structurel et Comportemental
- On va regarder en détail quelques patterns de ces catégories
- On va en assembler plusieurs pour résoudre un problème

Design Patterns Créationnels

Singleton

Design Pattern “Singleton”

- Objectif : s'assurer qu'une classe donnée aura au plus une instance
- Grand Classique
- Utile pour créer des utilitaires partagés par plusieurs objets de classes différentes



- Par contre, des défauts
  - Non Thread safe a priori
  - Gestion de la suppression
  - Non configurable

Design Pattern “Singleton”

- Implémentation classique
  - public class ClassicSingleton {
    private static ClassicSingleton instance = null;

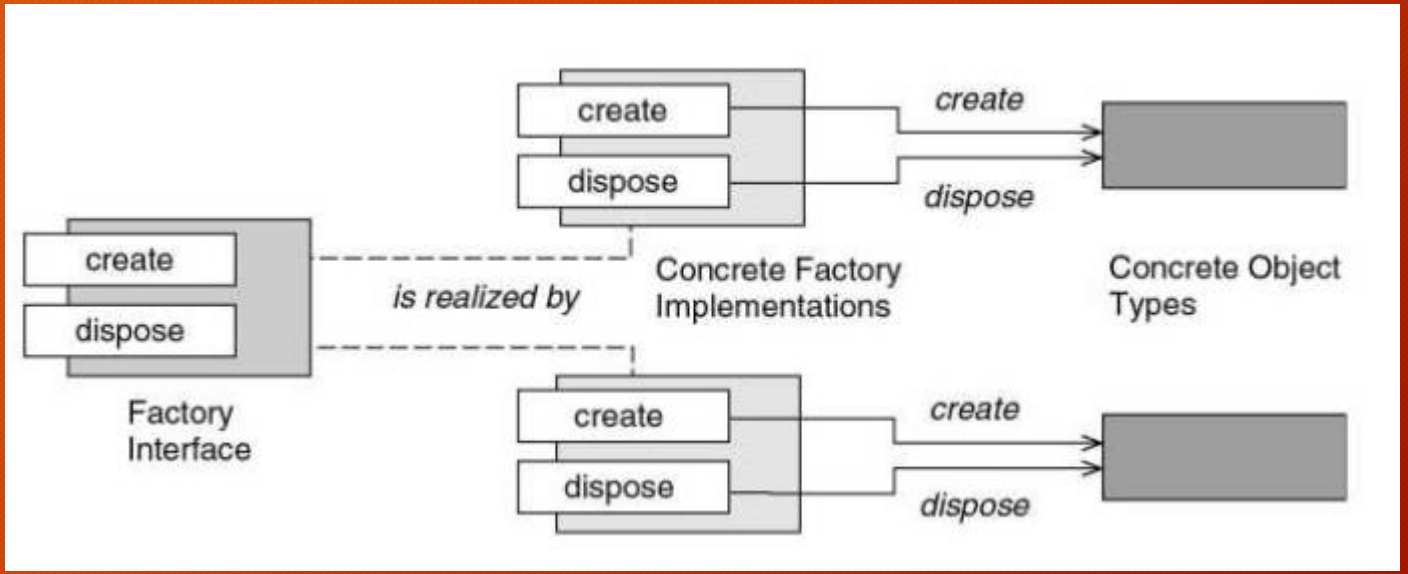
    private ClassicSingleton() {
    // interdit l'instanciation.
    }
    public static ClassicSingleton getInstance() {
    if(instance == null) {
    instance = new ClassicSingleton();
    }
    return instance;
    }
    …
  - }

Design Patterns Créationnels

Abstract Factory

Design Pattern "Abstract Factory"

- Objectif
  - Fournit une interface pour créer des objets qui ont une relation sans spécifier leur classes concrètes



Exemple d'un outil de dessin

- 3 classes sont responsables de gérer les dessins (Line, Circle, Rectangle)
- Les constructeurs sont différents
- Mias, on veut uniformiser les méthodes de création

- Solution : Abstract Factory (mixé avec Factory method)
- AbstractFactory contient une méthode de création assez large
  - Ex : Drawing createDrawing(Color aColor, Point… points)
- On crée des Factory concrètes pour Line, Rectangle, Circle, … en se basant sur le pattern « Singleton »
  - Class LineFactory implements AbstractFactory {
    - private static LineFactory defaultFactory = new LineFactory();
    - public static LineFactory getDefaultFactory() {
    - return defaultFactory;
    - }
    - public Drawing createDrawing(Color aColor, Point… points) {
      - return new Line(points[0], points[1], aColor);
    - }
  - }

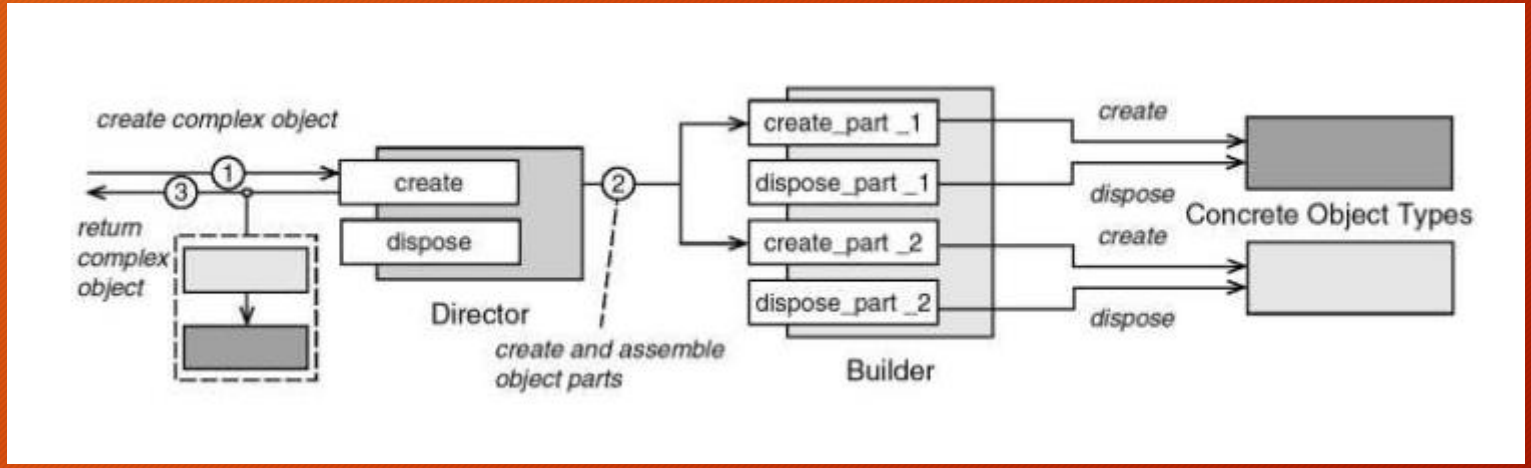
Design Patterns Créationnels

22

Builder

Design Pattern « Builder »

- Objectif
  - Support à la création d'objets complexes qui nécessitent plusieurs étapes
  - Evite le maintient de variables temporaires



- Exemple
  - StringBuilder en Java

Exemple d’un outil de dessin

- Un dessin se crée dans le temps par assemblage de formes
- Solution
  - Un builder combiné à des abstract factory

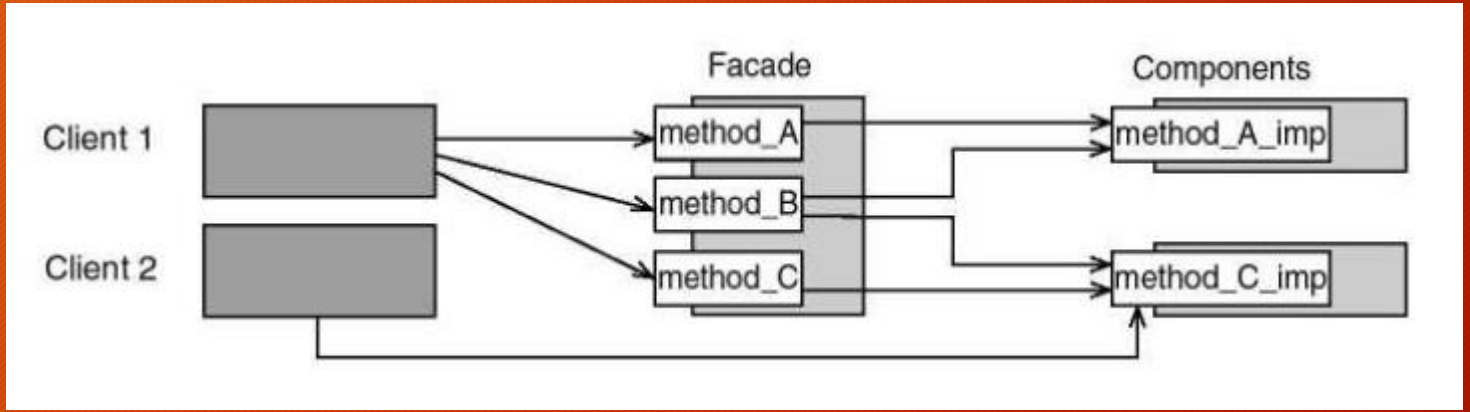
Design Patterns Structurels

25

Façade

Design Pattern "Façade"

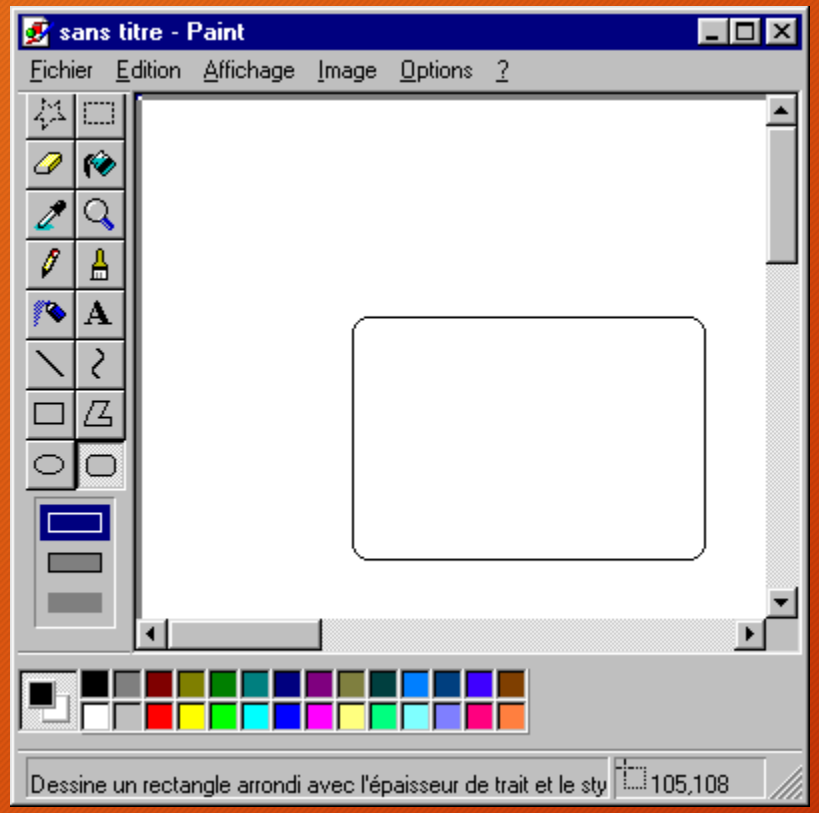
- Unifie et simplifie l'interface d'un sous-système
- Forme un point d'entrée simplifié dans une API
- Limite les points d'entrée d'une API
- Facilite l’utilisation d’un ensemble de classes



Design Patterns Comportementaux

State

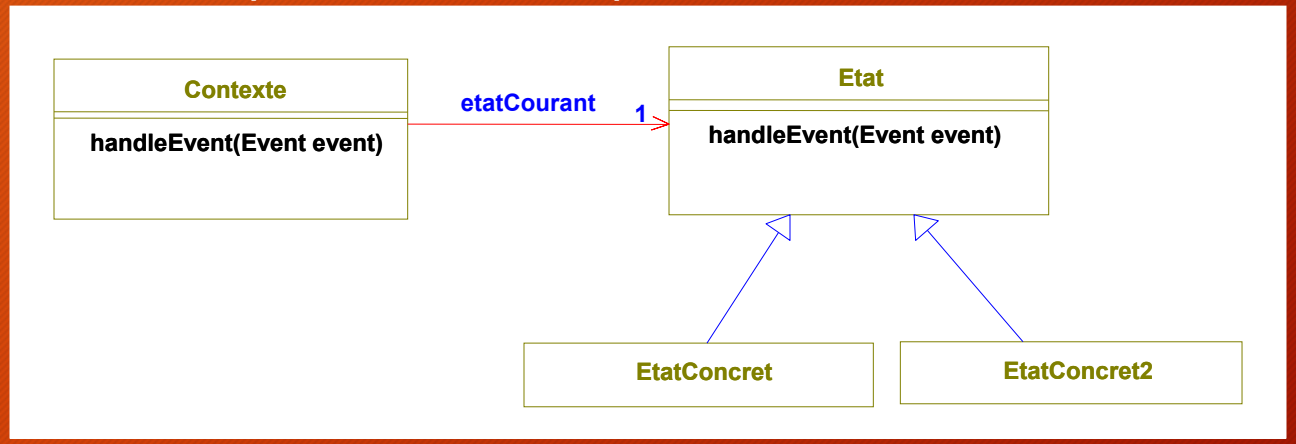
Le pattern État / Objet (State)

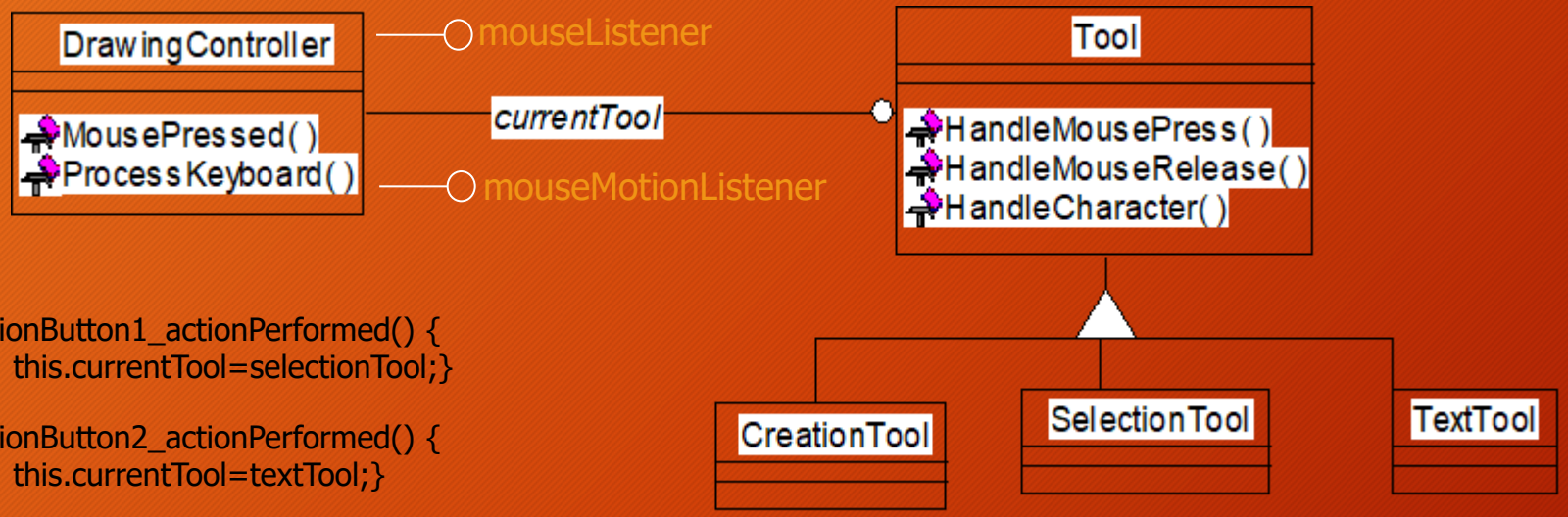


- Objectif : Permettre à un objet de changer de comportement quand son état interne change. L'objet se comporte comme s'il changeait de classe.
- Exemple typique : éditeur de dessin avec barre d'outils
  - « Modes d'opération » différents

Structure générale

- Participants
  - Contexte :
    - définit l'interface utilisable par les clients
    - Maintient une instance de Etat qui définit l'état courant
  - Etat
    - Classe abstraite qui définit l'interface de chacun des états concrets
  - EtatConcret (classes dérivées)
    - Chaque sous-classe implémente le comportement effectif associé à un état du contexte






```
selectionButton1_actionPerformed() {
        this.currentTool=selectionTool;}

selectionButton2_actionPerformed() {
        this.currentTool=textTool;}
```

mouseListener

```
mousePressed() {
        currentTool.handleMousePress();}
mouseReleased() {
        currentTool.handleMouseReleased();}
mouseClicked() {
        currentTool.handleMouseClicked();}
mouseEntered() {
        currentTool.handleMouseEntered();}
mouseExited() {
        currentTool.handleMouseExited();}
```

mouseMotionListener

```
mouseMoved() {
        currentTool.handleMouseMove();
}
mouseDragged() {
        currentTool.handleMouseReleased();
}
```

Conséquences

- Localise le comportement dépendant de l'état dans des sous-classes spécifiques
  - Évite la prolifération de clauses « if » ou « switch » dans le contrôleur
- Facilite l'introduction de nouveaux états
  - Définition d'une nouvelle sous-classe
  - Instanciation de l'objet/état
- Rend les changements d'états plus explicites

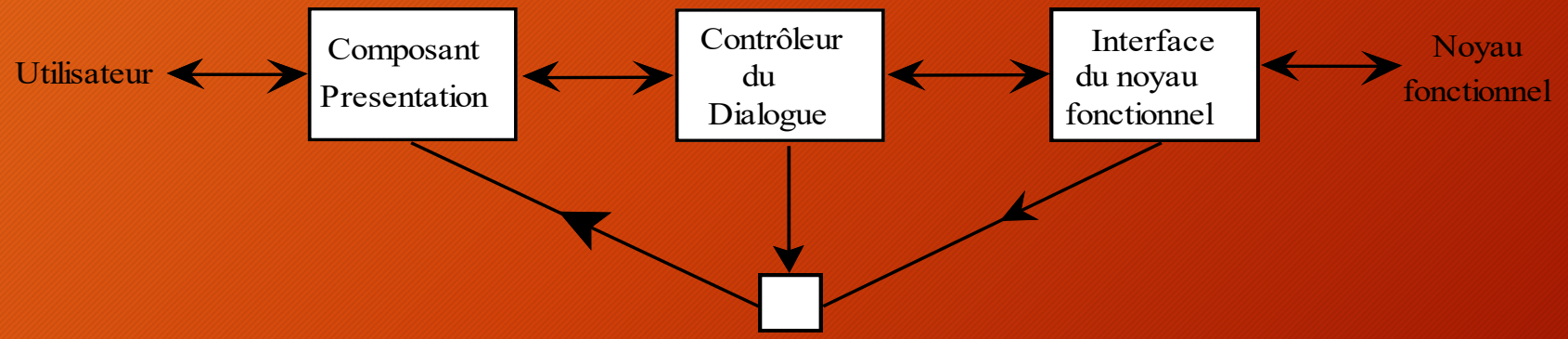
Détails spécifiques

- Comment les changements d'états sont-ils contrôlés ?
  - Gérés par le contexte (le contexte implémente un automate à états)
  - Gérés par chaque sous-état
    - Chaque état connaît son (ses) état(s) suivants
    - implémentation « distribuée » d'une machine à états
- Comment sont instanciés les objets/états ?
  - Pré-instanciation
  - Instanciation dynamique selon les besoins

Rappel : Architectures IHM classiques

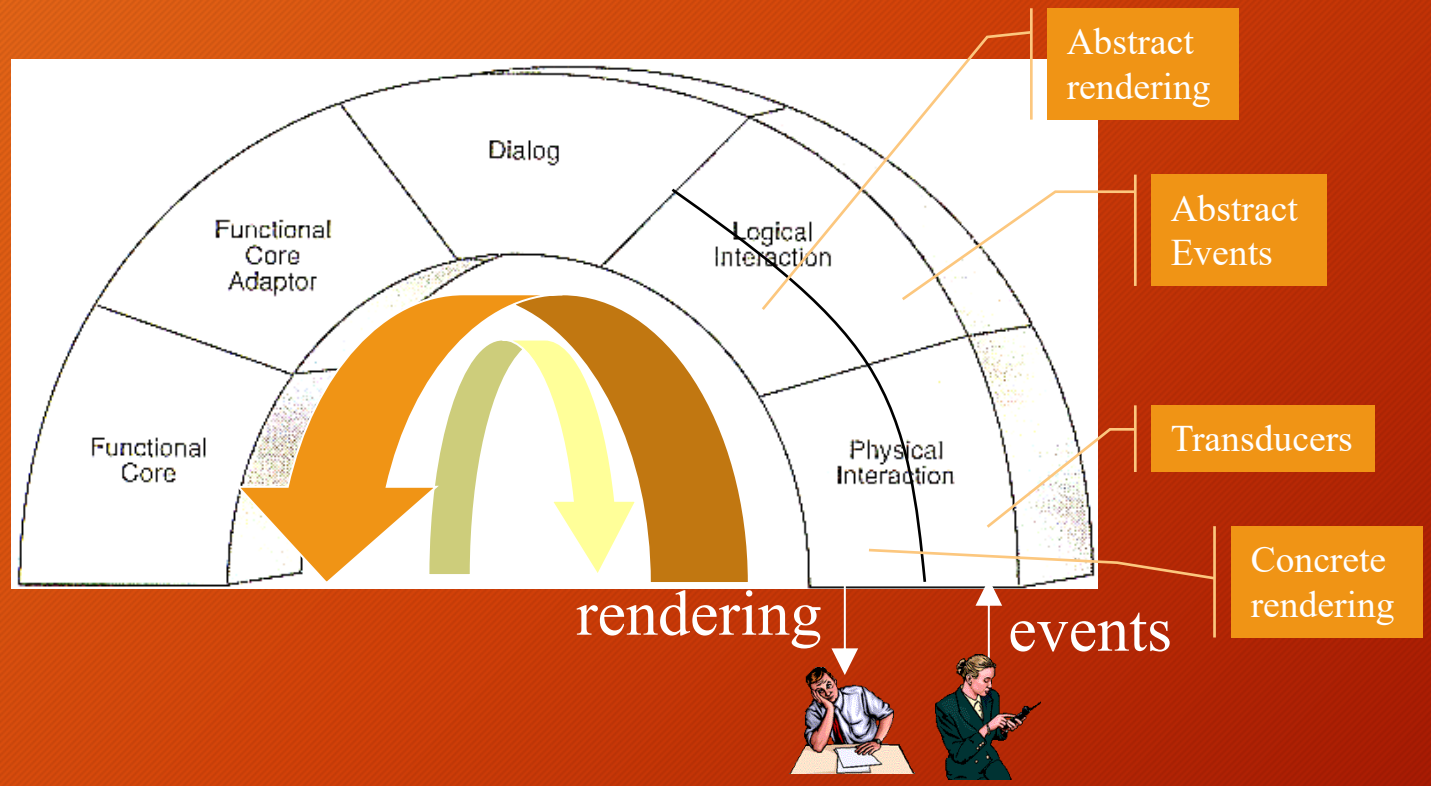
Seeheim 1983

- Pfaff, G. (ed). Seeheim Workshop on User Interface Management Systems. Eurographics Seminars, Springer-Verlag 1985.



Arch/Slinky 1992

- UIMS Tool Developers Workshop. A Metmodel for the Runtime Architecture of an Interactive System. SIGCHI Bulletin 24(1), 1992.



Avantages et Limites

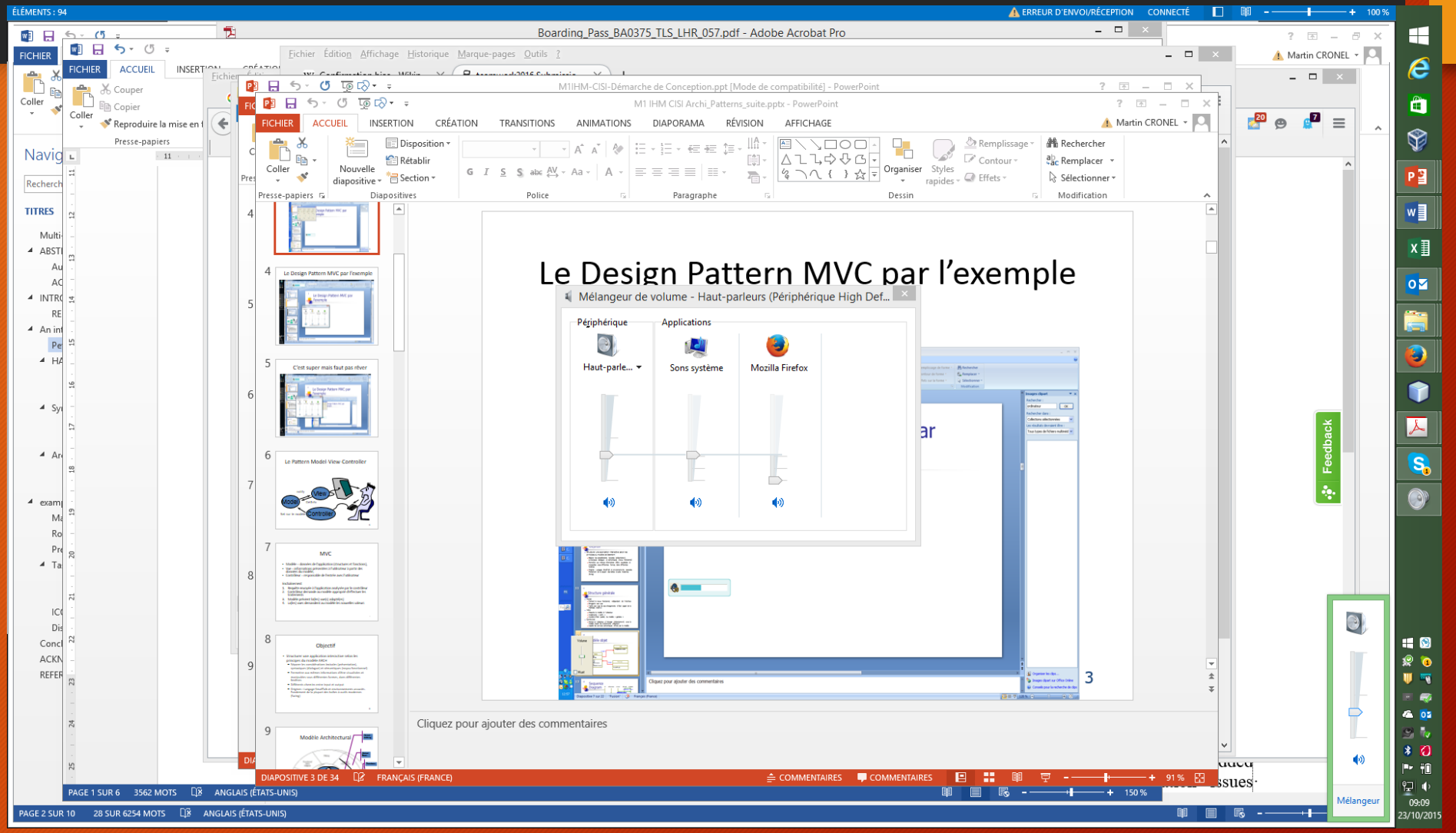
- Approche modulaire
- Mise en avant du contrôleur de dialogue
- Possibilité de développer le noyau fonctionnel en parallèle de l'interface (métiers différents)

- Couplage fort entre les briques
- Duplication de données

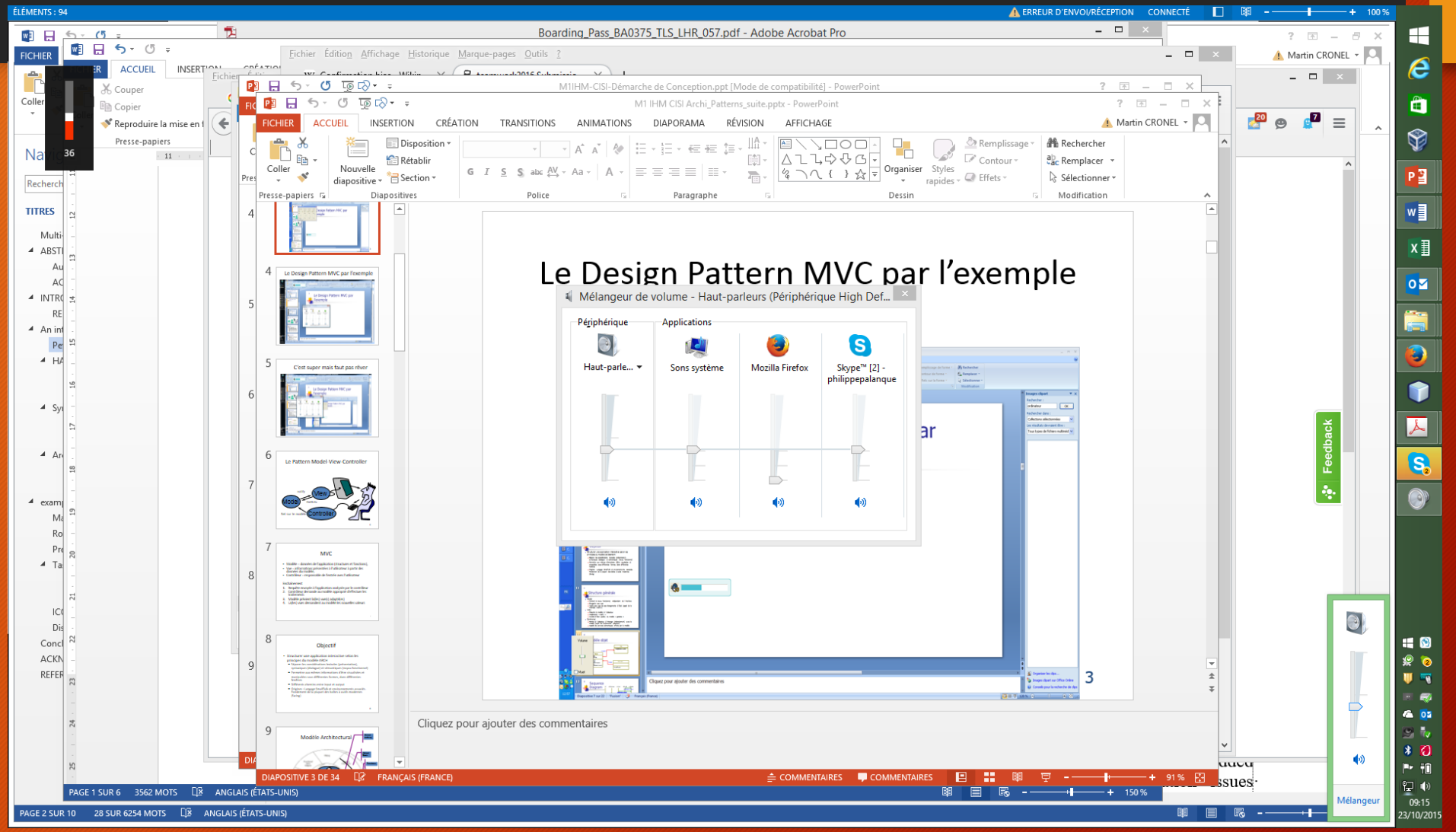
Design Patterns Comportementaux

MVC

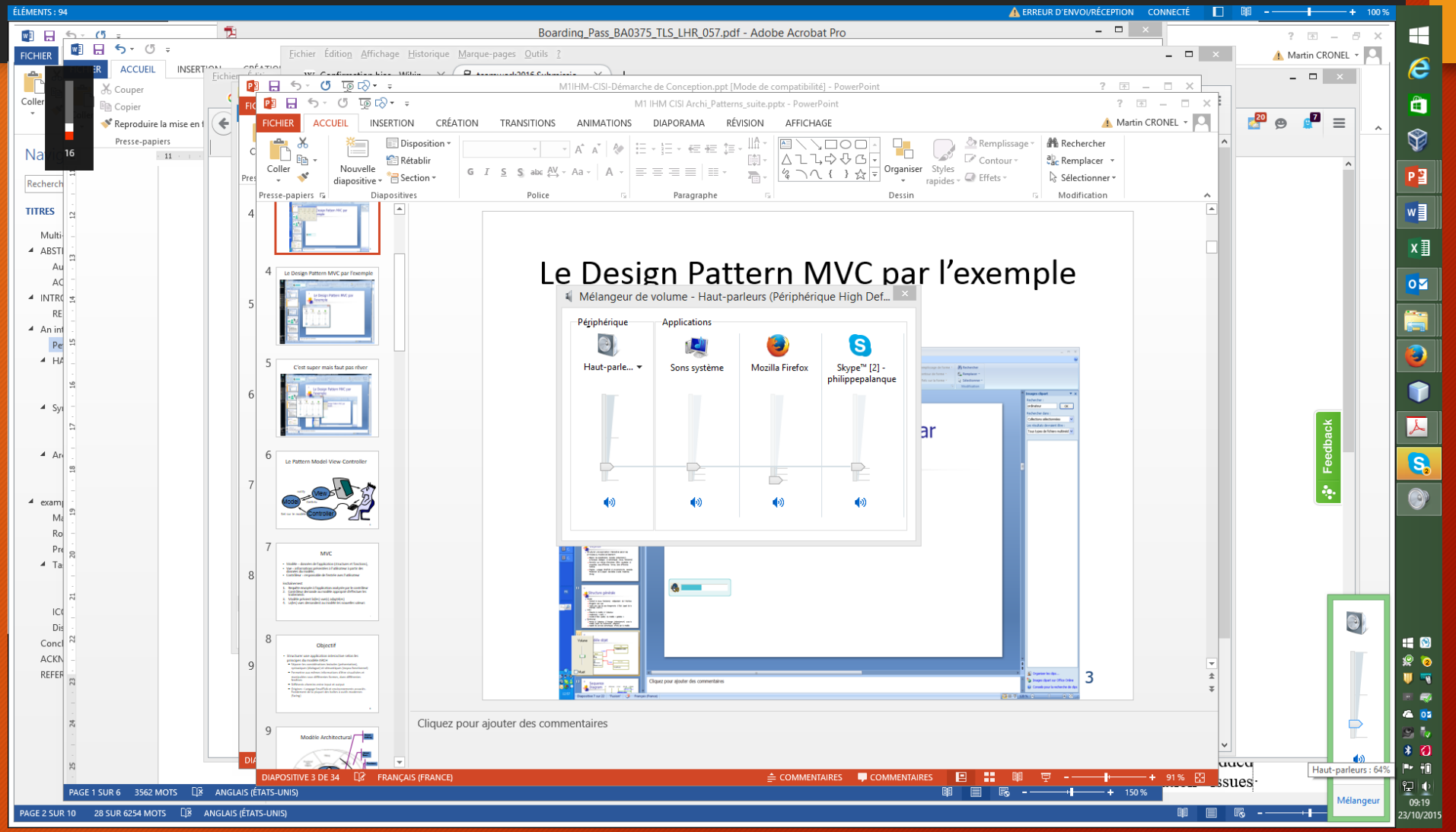
Le Design Pattern MVC par l'exemple



Le Design Pattern MVC par l'exemple

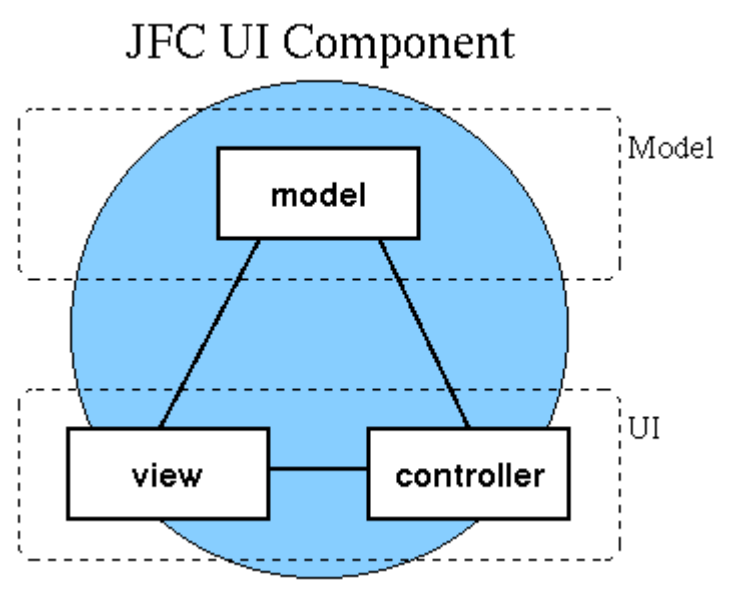


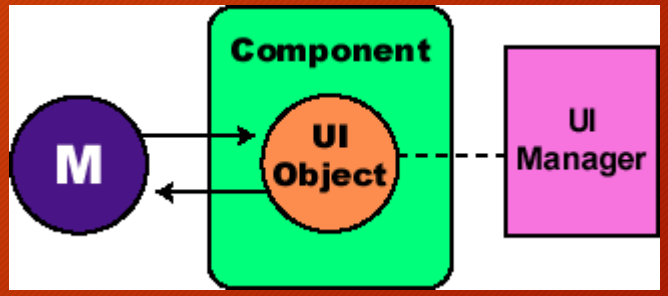
Le Design Pattern MVC par l'exemple



Les composants Java Swing

- Implémenté en respectant JavaBeans (propriétés, événements, facets)
- Architecture Swing = MVC ou M-UI





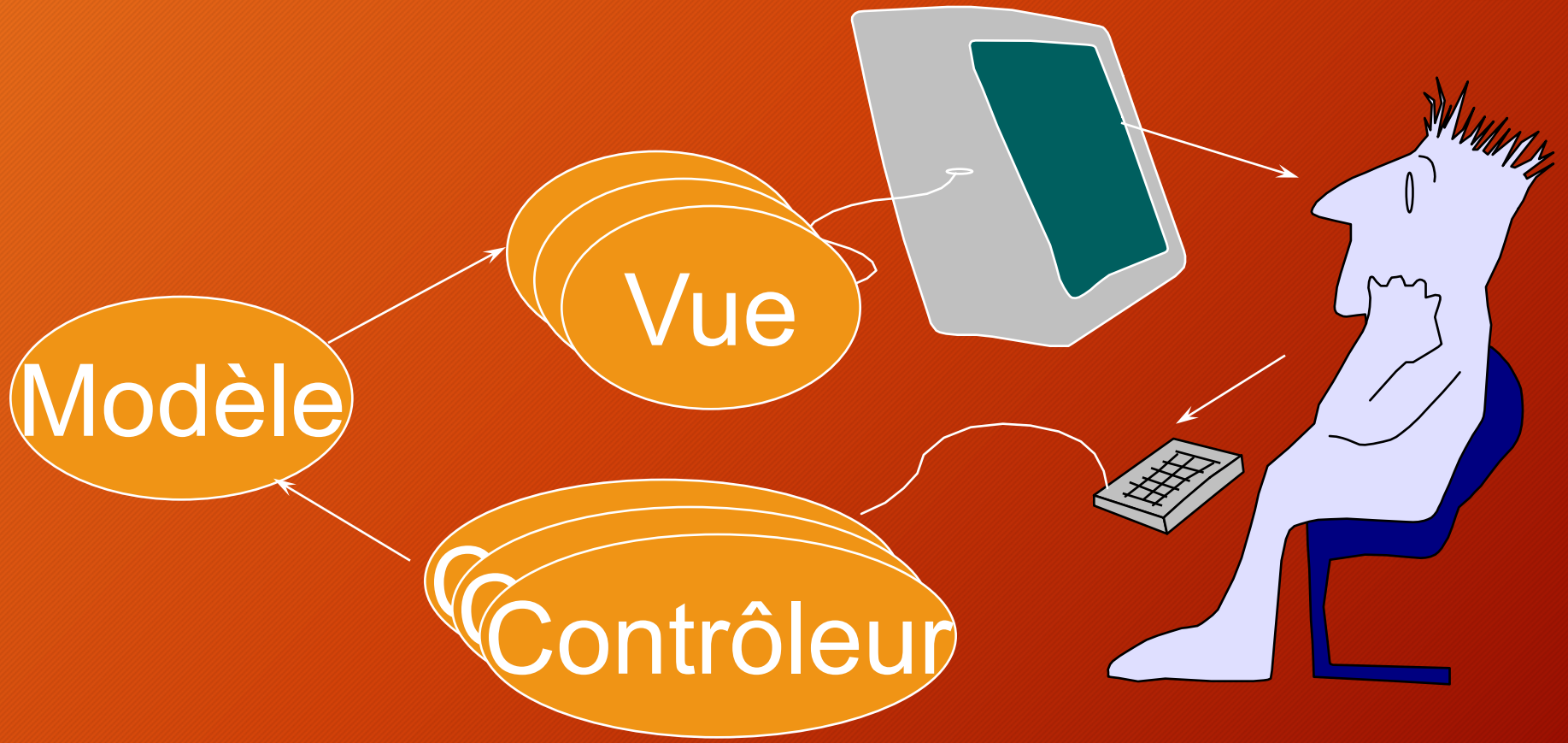
Architecture des JComponents

- Modèle
  - Gère l'état interne (enabled, pressed, armed…)
  - Gère les listeners
  - Lève les événements (actionPerformed, …)
  - Peut être lié au noyau fonctionnel (exemple : JTable et Base de données)
- UI
  - Gestion du Look&Feel
    - Gère le dessin
    - Fournit les informations sur la géométrie
    - Gère les événements AWT
  - Peut-être changé dynamiquement
- Le tout est caché dans un object qui sert de façade
  - setModel
  - setUI

Exemple du JButton

- Démo partage de modèle entre boutons
- ButtonModel
  - Gestion des propriétés pressed, armed, selected
  - Gestion des listeners (et donc événéments) changed et action
- ButtonUI
  - Responsable du dessin
  - Instance partagée par plusieurs boutons
  - Sous-classée pour les différents Look&Feel
  - Définition de la géométrie (contains, preferredSize…)
  - Changement d'état du modèle en fonction des événement reçus via un ButtonUIListener

Le Pattern Modèle - Vue - Contrôleur



Model-View-Controler

- Trygve Reenskaug: THING-MODEL-VIEW-EDITOR - an Example from a planningsystem. Technical note, Xerox PARC, May 1979.
- Trygve Reenskaug: MODELS - VIEWS - CONTROLLERS. Technical note, Xerox PARC, December 1979.
- Adèle Goldberg (Smalltalk), 1980

MVC

- But : Séparation de la partie interface de la partie donnée dans le cadre de données volumineuses et complexes

- Modèle
  - gère les aspects applicatifs
  - met à jour les vues enregistrées
- Vue
  - se déclare auprès d'un modèle
  - se charge de la visualisation en sortie
  - *redirige les entrées vers un contrôleur associé*
- Contrôleur
  - gère les événements d'entrée
  - prévient le modèle associé

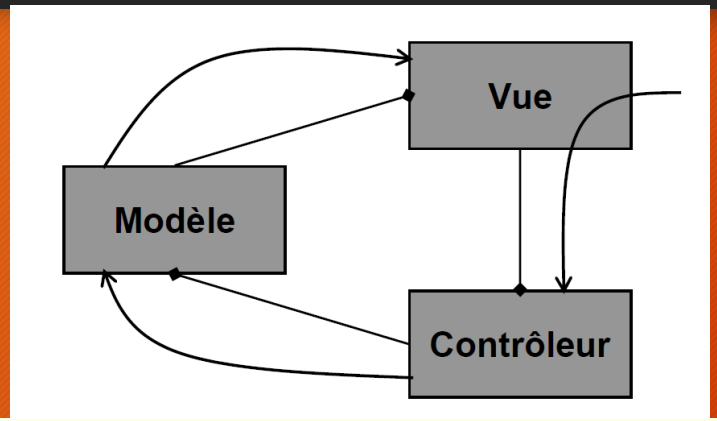
Objectif

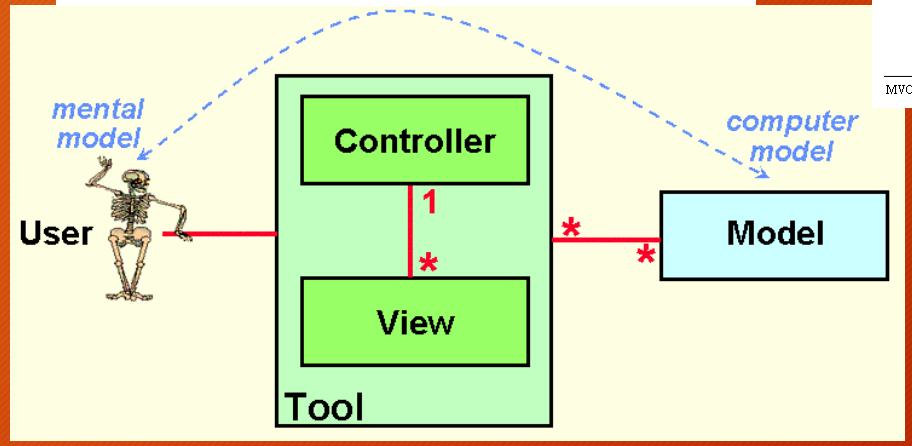
- Structurer une application interactive selon les principes du modèle de Seeheim
  - Séparer les considérations lexicales, syntaxiques et sémantiques
  - Permettre aux mêmes informations d'être visualisées et manipulées sous différentes formes, dans différentes fenêtres

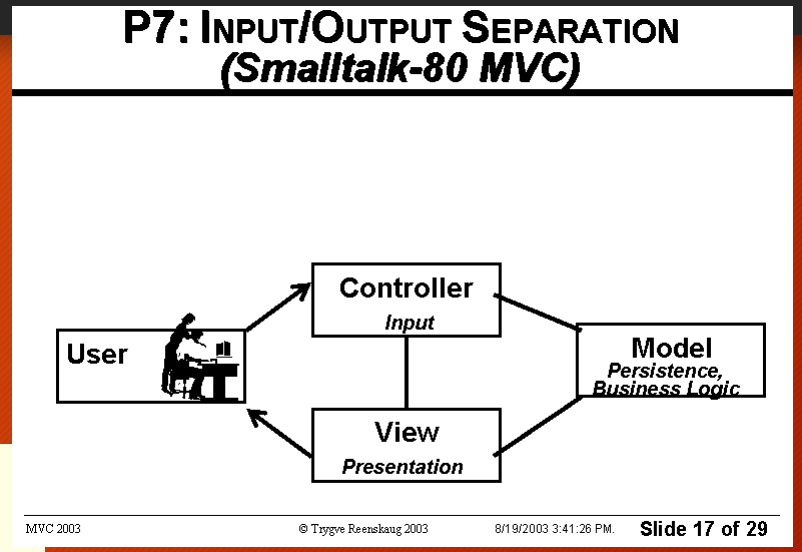
Structure générale

- Modèle
  - Contient le noyau fonctionnel, indépendant de l'interface
  - Enregistre ses vues
  - Notifie ses vues de ses changements d’état (appel de « notify »)
- Vue
  - Présente l’état du modèle à l’utilisateur
  - Implémente « notify »
  - Accède à l’état (public) du modèle (« getdata »)
- Contrôleur
  - Permet à l'utilisateur d’interagir (indirectement) avec le modèle (reçoit les événements utilisateur)
  - Appelle les services sémantiques offerts par le modèle

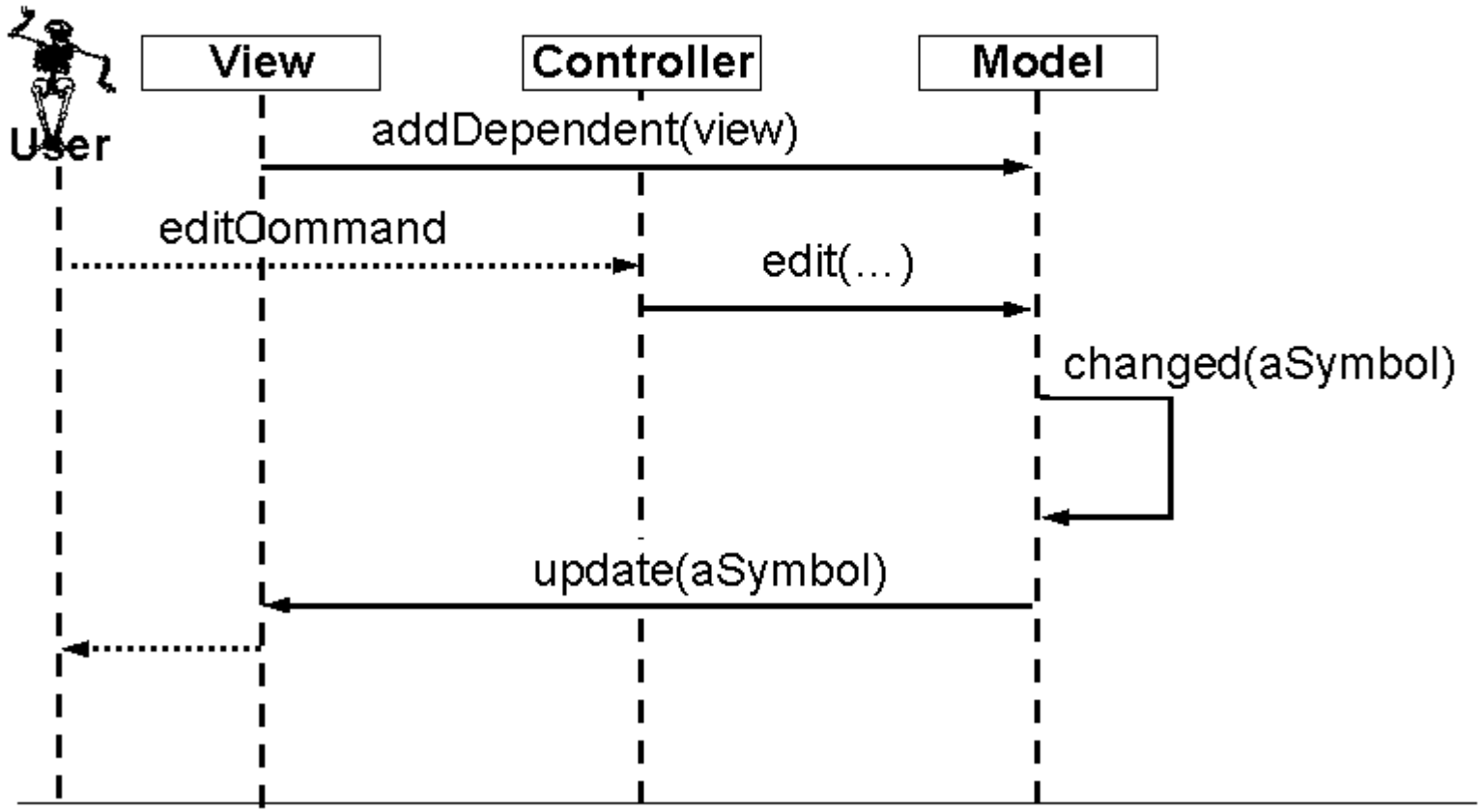
MVC : Versions Originales





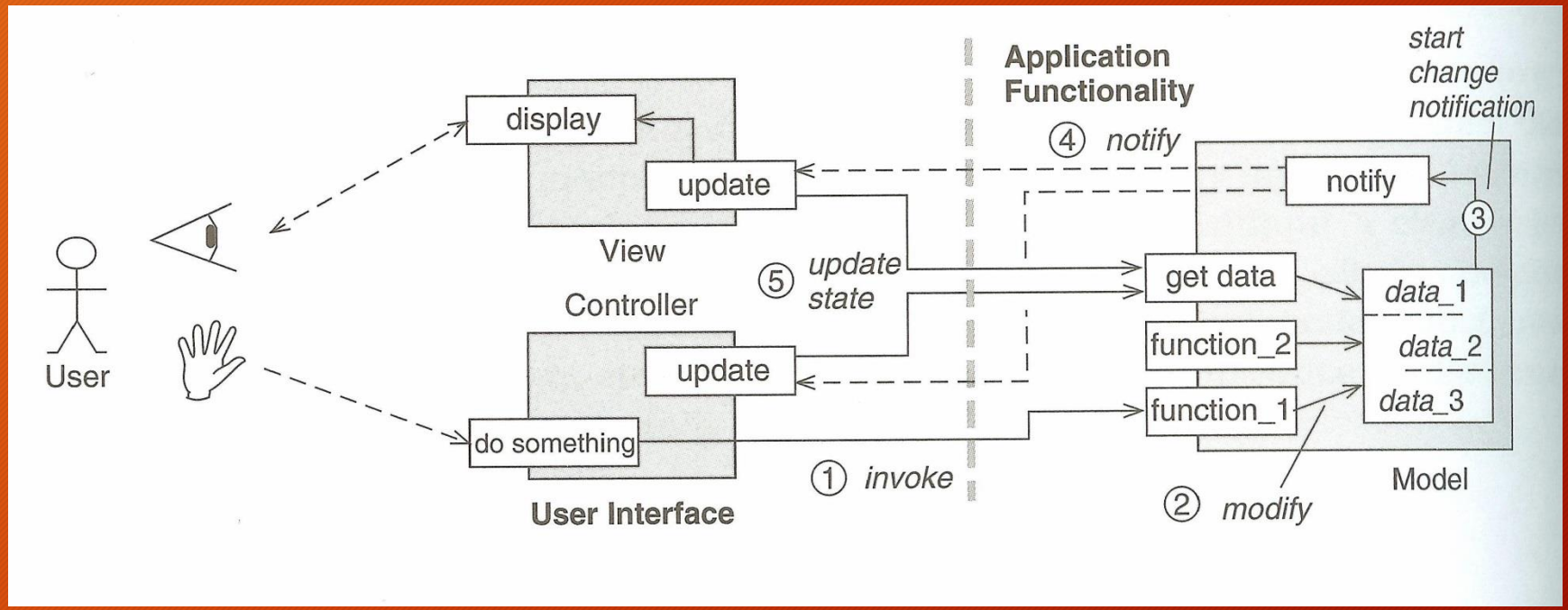


P11: Synchronize Model and View
ST-80 class library



MVC : version « moderne »

- Buschmann F., Henney K., Schmidt D.C., Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, Vol 4, Wiley Series, 2007



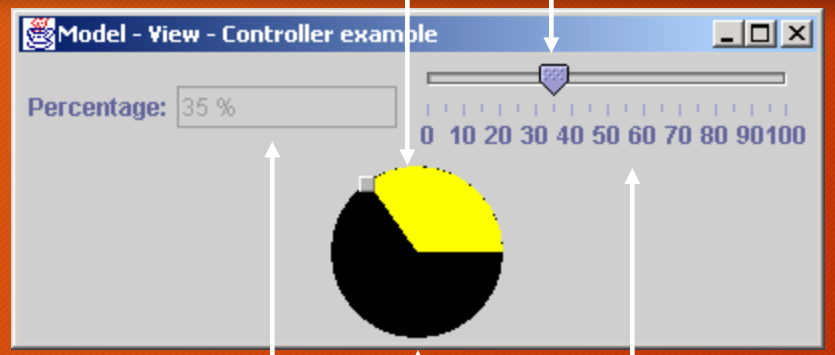
MVC

- Modèle = données de l'application (structures et fonctions),
- Vue = informations présentées à l'utilisateur à partir des données du modèle,
- Contrôleur = responsable de la gestion des entrées de l'utilisateur

Enchaînement

1. Requête envoyée à l'application analysée par le contrôleur
2. Contrôleur demande au modèle approprié d'effectuer les traitements
3. Modèle prévient la(les) vue(s) adapté(es)
4. La(les) vues demandent au modèle les nouvelles valeurs

Exemple



Contrôleurs

Vues

Modèle:

Pourcentage (valeur flottante 0 <= f < 1)

Modèle de Classe

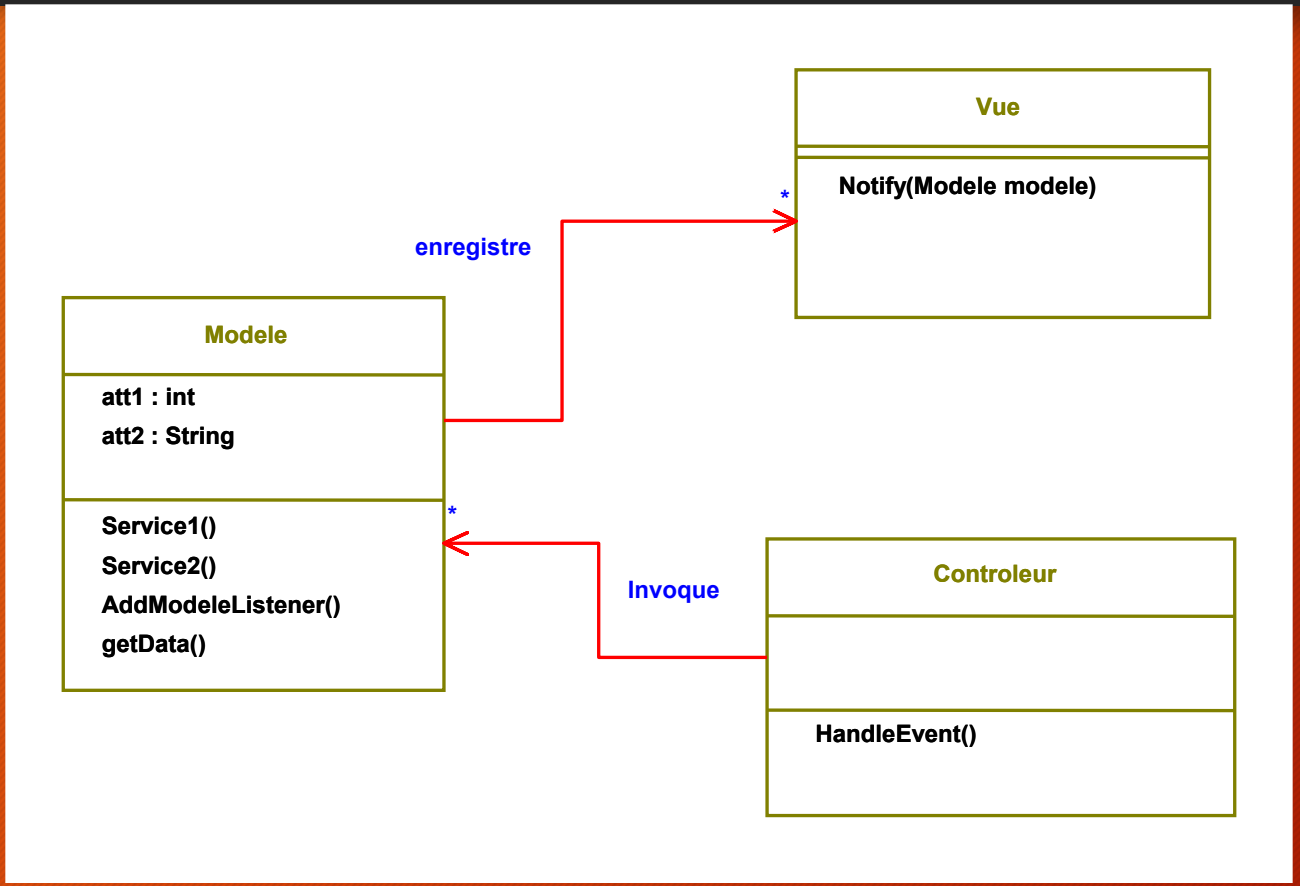
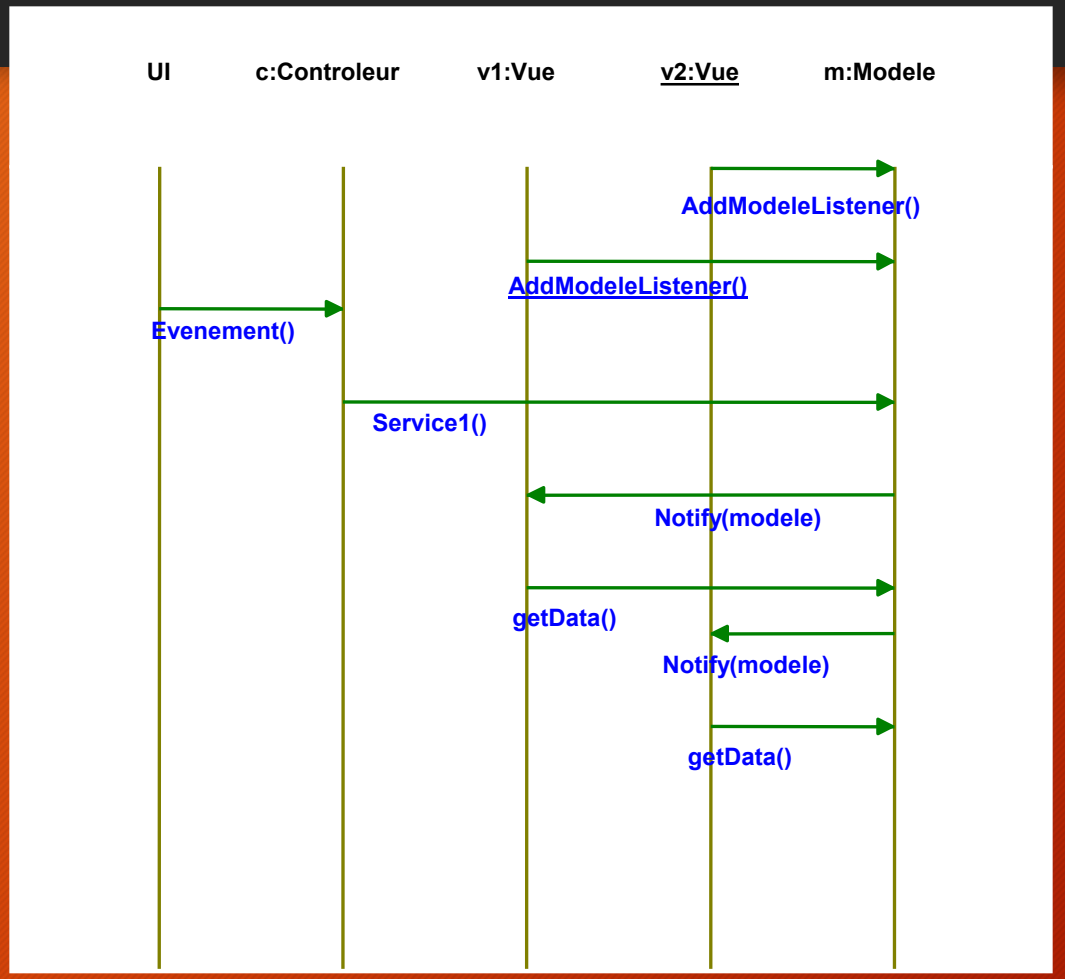


Diagramme de séquence



Conséquences

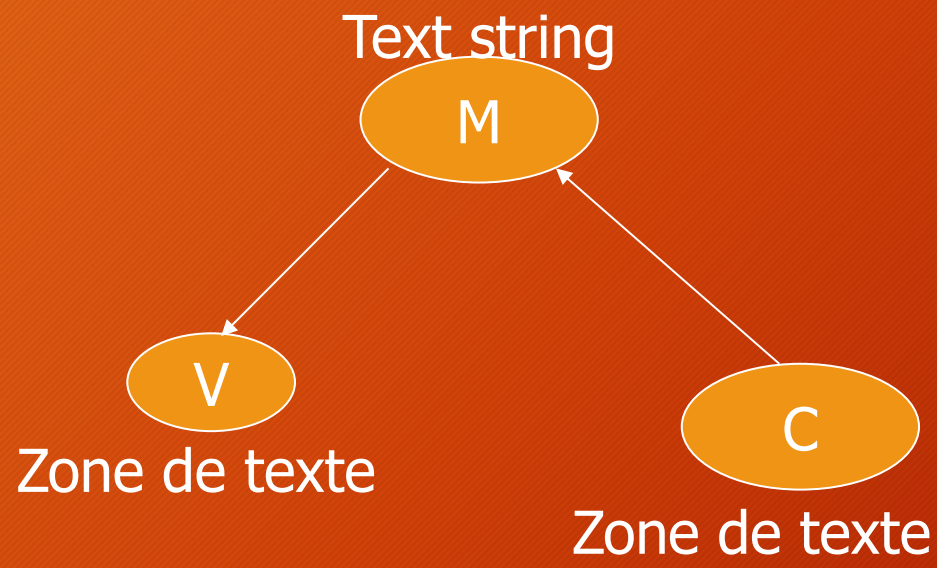
- Avantages
  - Facilité à présenter différentes vues du même objet et à les synchroniser
  - Facilité d'ajout d'une nouvelle représentation
- Inconvénients
  - Multiplication des « notify »
  - Inefficacité de l'accès au modèle
  - Couplage fort entre vue et contrôleur
    - Impossible en général de réutiliser un contrôleur indépendamment de sa vue
- Variantes
  - Document / View, PAC

Exemple particulier le zoom

- Où sont les éléments M, V et C dans un la fenêtre?
- Décrivez l'architecture ainsi que les séquences d'appel de méthodes

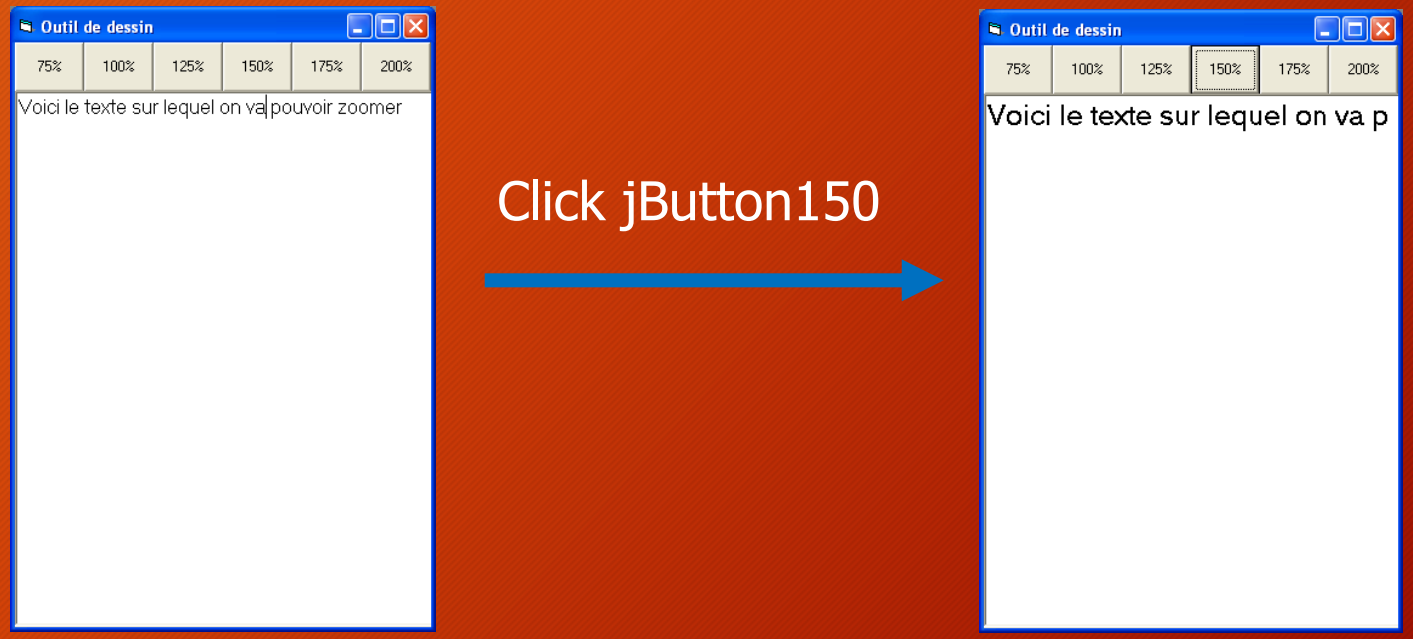
Exemple particulier le zoom

- Où sont les éléments M, V et C dans un la fenêtre?
- Décrivez l'architecture ainsi que les séquences d'appel de méthodes

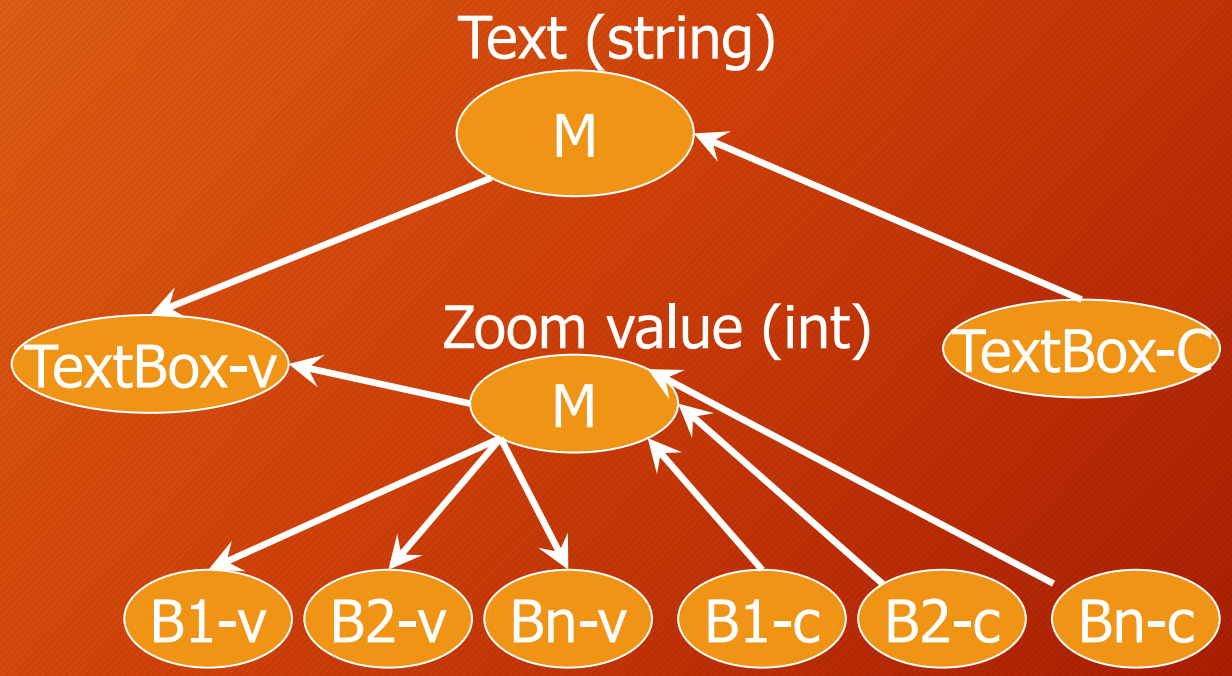


Exemple particulier le zoom

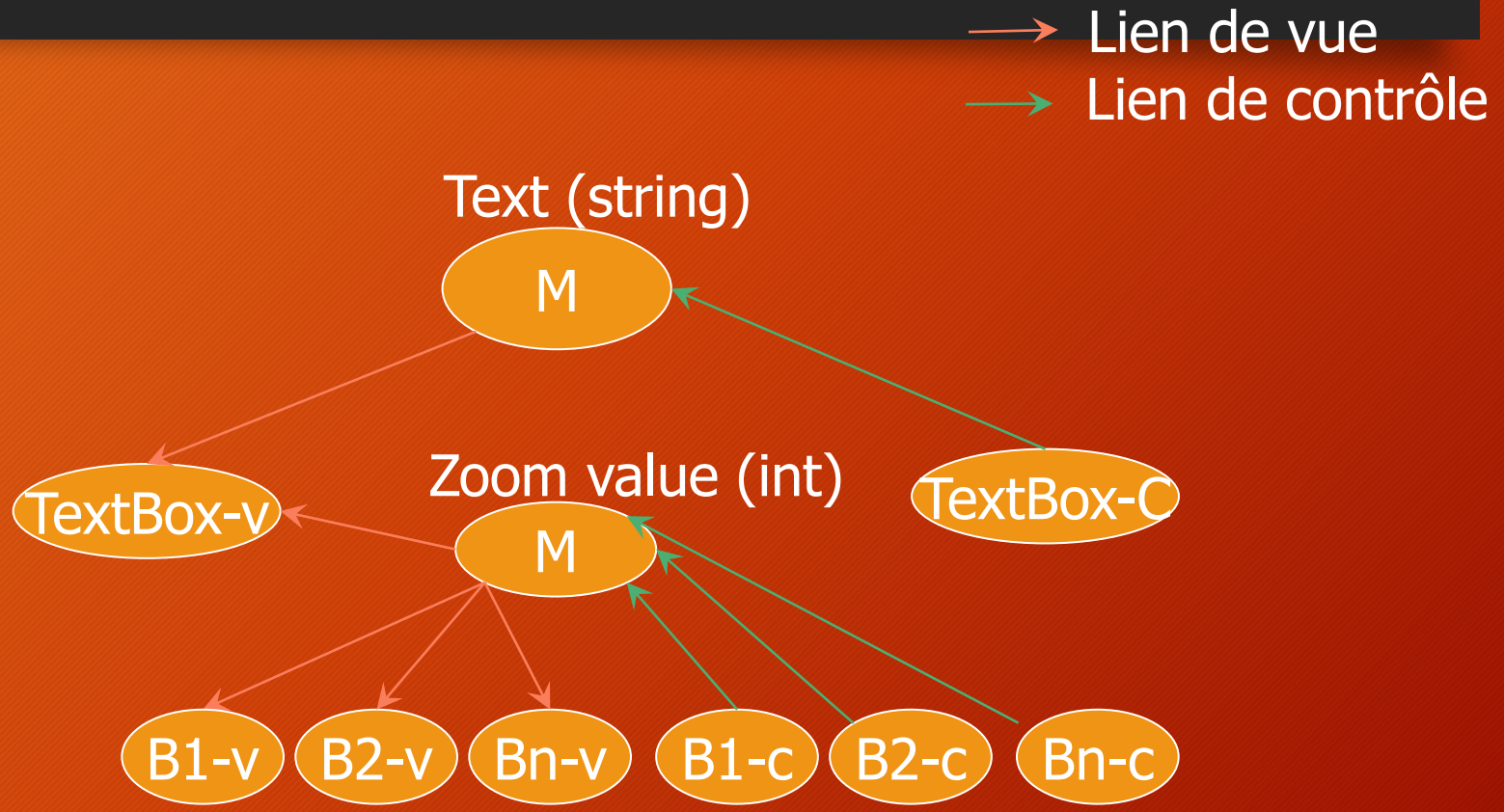
- Comment gérer plusieurs contrôleurs
- Décrivez l'architecture ainsi que les séquences d'appel de méthodes



Exemple particulier le zoom

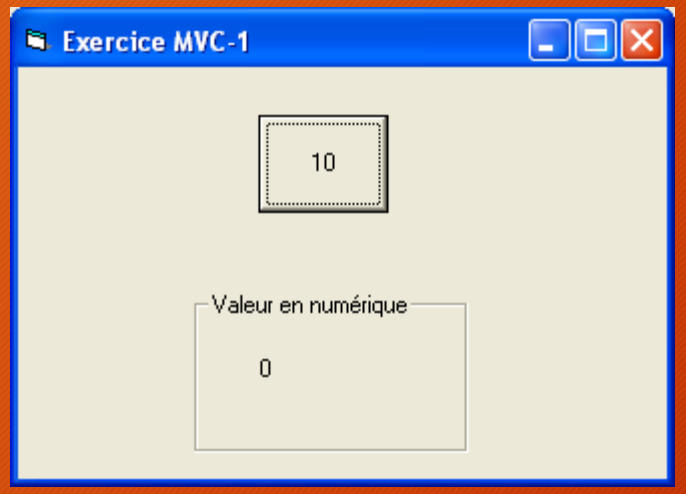


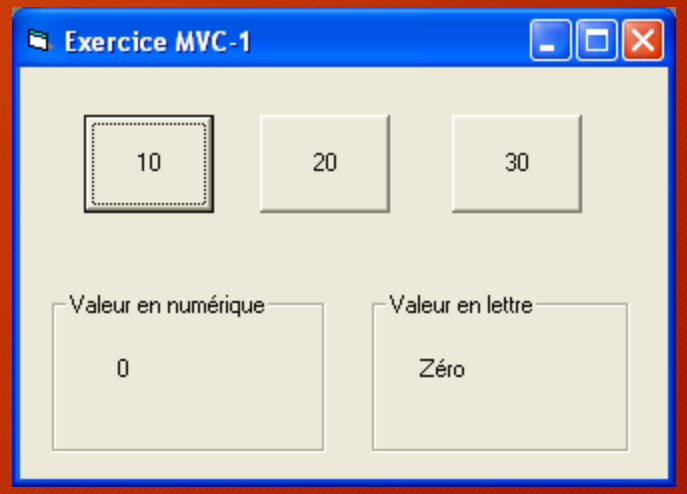
Exemple particulier le zoom



Exercice MVC

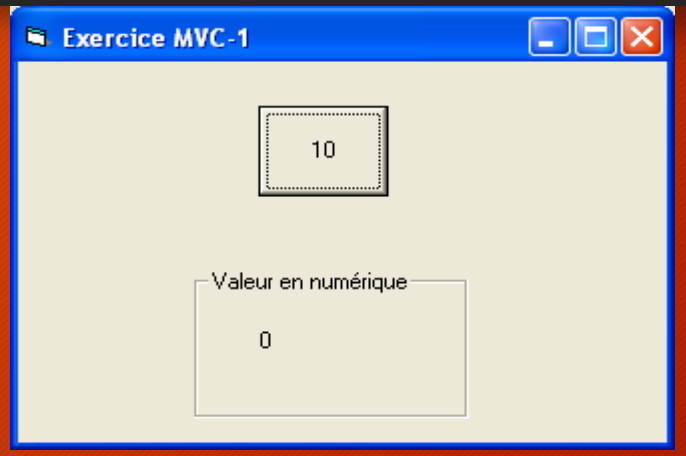
- 2 applications avec des modifications minimales grâce à une bonne structuration du code
- Objectif accroître la modifiabilité de l'application

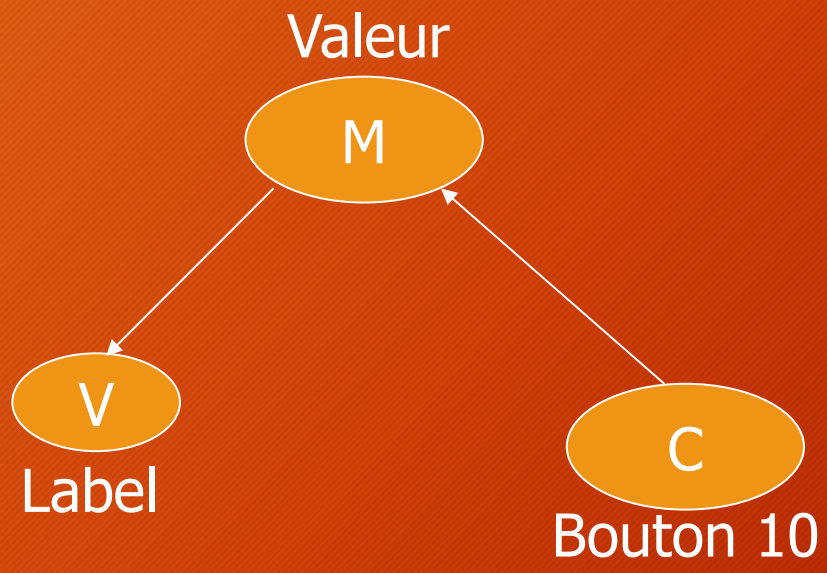




Exercice MVC

- Cas 1





Exercice MVC : Interfaces

- 2 interfaces
  - IVue : offre une méthode notification()
  - IModel : offre une méthode addVue(IVue vue)
- Les vues sont "implements" de l'interface IVue
- Les modèles sont "implements" de l'interface IModel
- Objectif
  - Rapidité de création de nouvelles vues
  - Cohérence « forcée » des vues
  - Ne pas faire plusieurs fois la même chose

Exercice MVC : Classes

- VueLabel1
  - A une référence vers l'objet qui permet l'affichage
- ModelString
  - Contient un attribut (String valeur)
    - Offre l'accesseur setValeur (String chaine) : appelé par les contrôleurs
    - Offre l'accesseur getValeur (returns valeur) : appelé par les vues
  - Connait chacune des vues (géré dans une Collection)
  - Gère les vues en offrant un abonnement/désabonnement
  - Possède un itérateur pour la notification de toutes les vues
- jButton1: contrôleur direct (pas de classe supplémentaire)
  - Connait l'instance de ModelString
  - Appelle setValeur (dans l'event handler ou "actionPerformed")

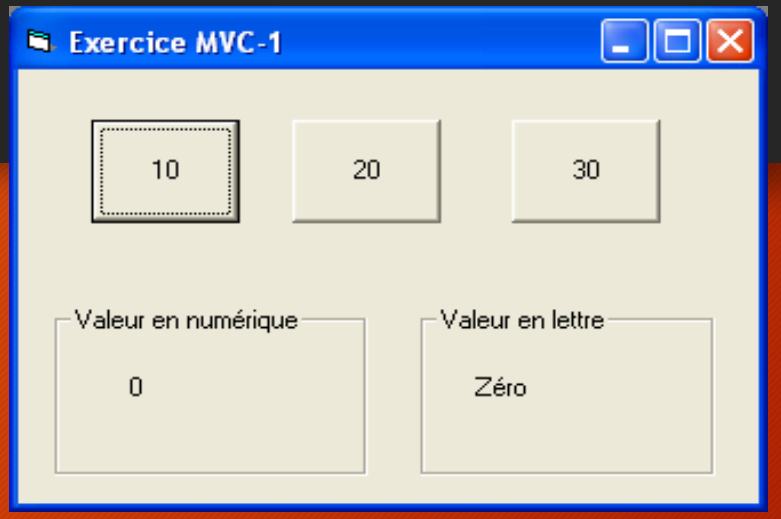
Exercice MVC : code-contrôleur

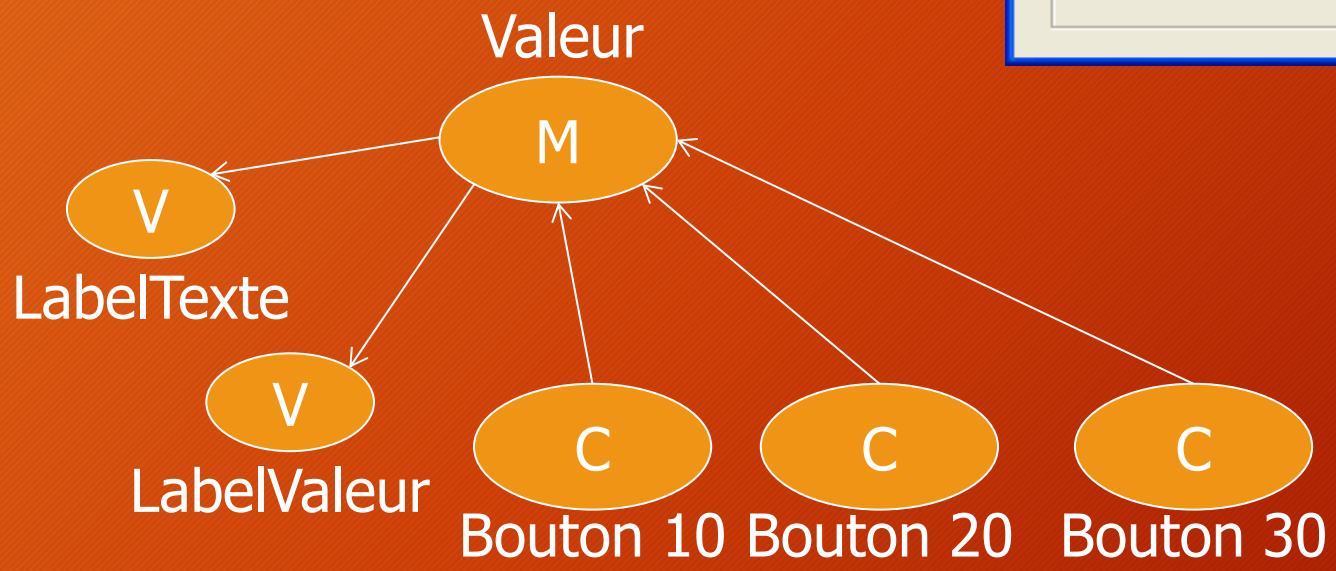
- Le contrôleur (**code mauvais :** le contrôleur connait la vue et même ici l'interface graphique)
  - jButton1.addActionListener((e) -> { jLabel1.setText("10"); }

- Le contrôleur (**code bon :** le contrôleur connait le modèle)
  - jButton1.addActionListener ((e) -> {
  - model.setValeur("10"); }

Exercice MVC

- Cas 2



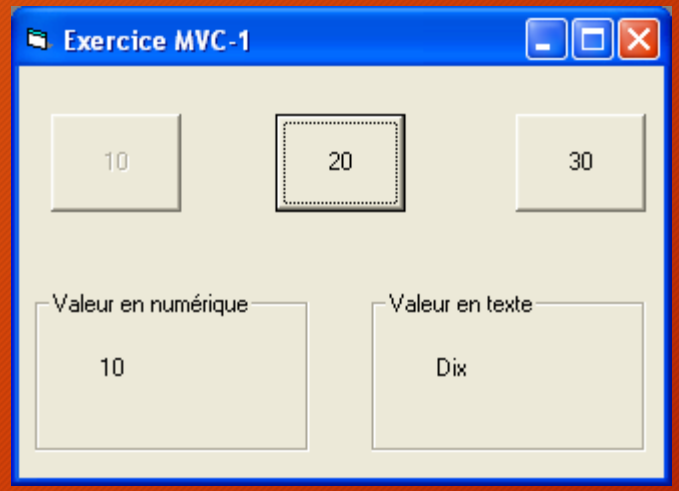


Exercice MVC: modification pour les 3 contrôleurs et les 2 vues

- …

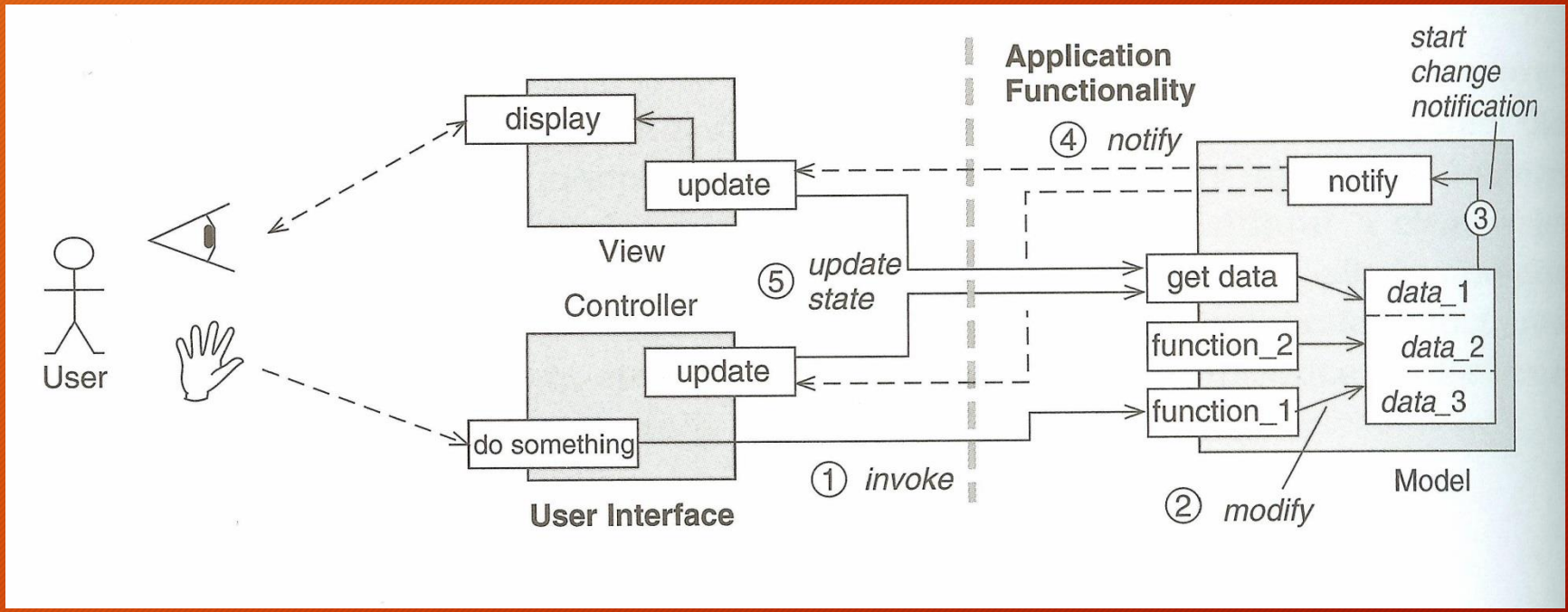
Exercice MVC-3

- Mélange états modèles et design patterns



MVC : version « moderne »

- Buschmann F., Henney K., Schmidt D.C., Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, Vol 4, Wiley Series, 2007



Design Patterns Comportementaux

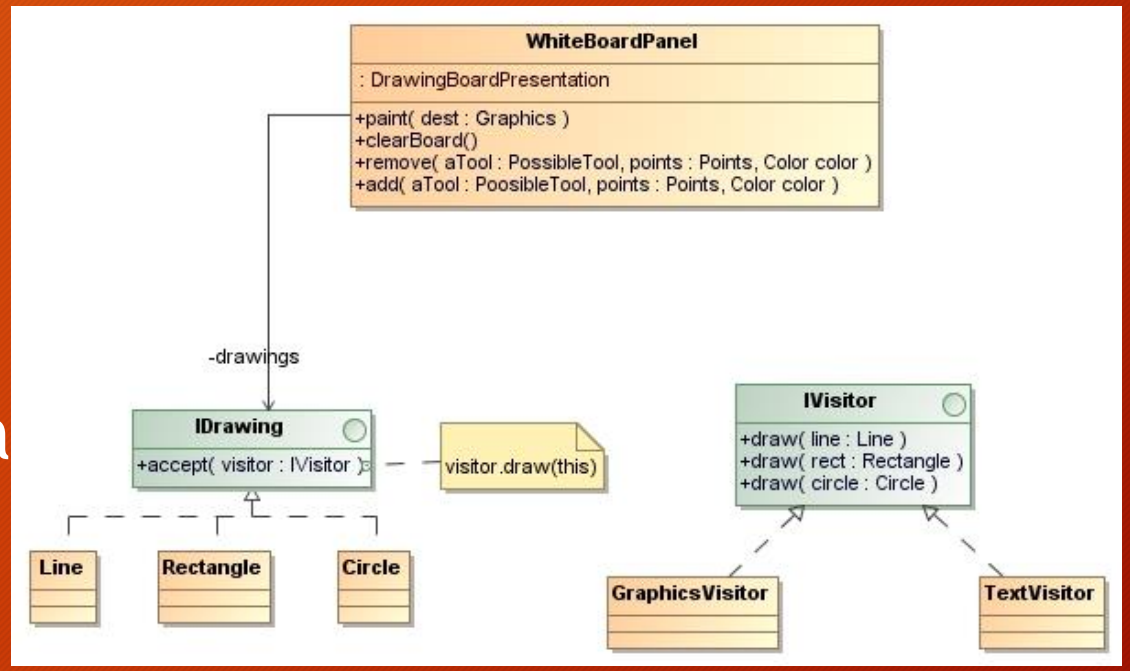
Visitor

Design Pattern Visitor

- Utilité : permettre de rajouter facilement des fonctionnalités à une hiérarchie de classes sans retoucher à la hiérarchie
- Quelle place pour visitor dans l'outil de dessin ?

Visitor : Diagramme de classe

- Visiteur Graphique utilisé pour faire le dessin swing
- Visiteur Texte utilisé en passant par un menu de la fenêtre pour simuler la sauvegarde



Design Patterns Comportementaux

Command

Undo / Redo : brutal

- Sauvegarder après chaque action l'état complet de l'application
  - Fonctionne, mais
  - Pb de capacité de stockage
- Idée, stocker le strict minimum à chaque action
  - Utilisation du pattern Command

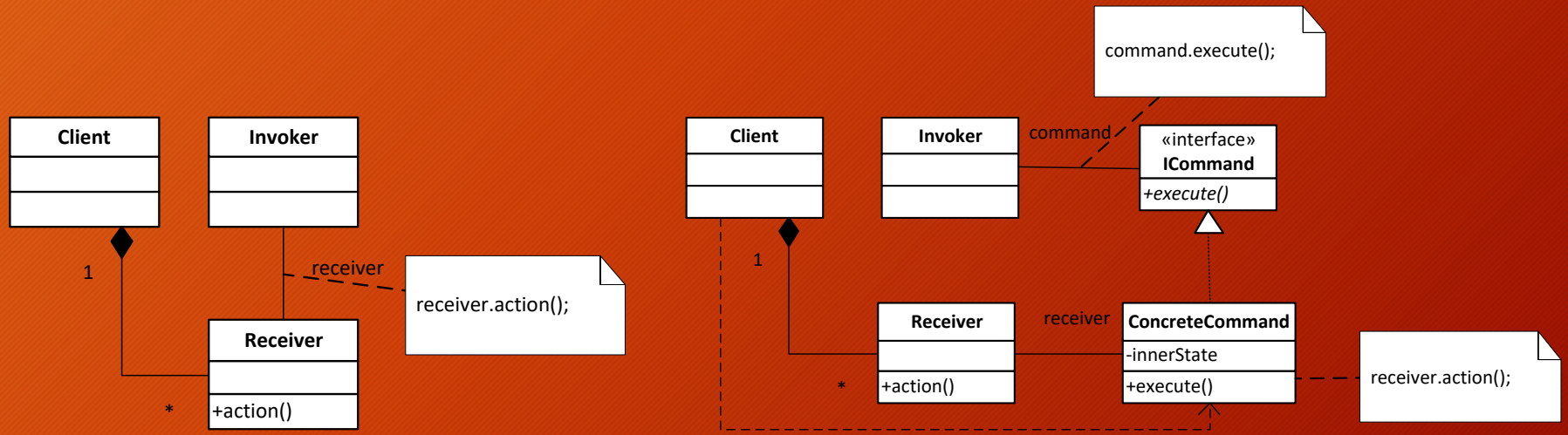
Pattern Command : principe

- Objectif : réifier une action et permettre de l'attacher à l'objet auquel elle se rapporte
- Permet la mise en œuvre de concepts transversaux aux IHM
  - Macro (liste de commandes regroupées)
  - Undo/Redo
  - Logging, Recover
  - Copy/Cut/Paste
  - Actions différées

Pattern Command : Structure

- Client = Application
- Invoker = Item de menu par exemple
- Receiver = Document ou application

Avant	Après



Undo / Redo : pattern Command

- Un objet command peut offrir un service de undo
- Pour assurer le undo, il faut que l'objet possède suffisamment d'information
  - Référence au receveur
  - Arguments de la méthode à invoquer
  - Le receveur doit fournir une méthode qui permettra de revenir en arrière

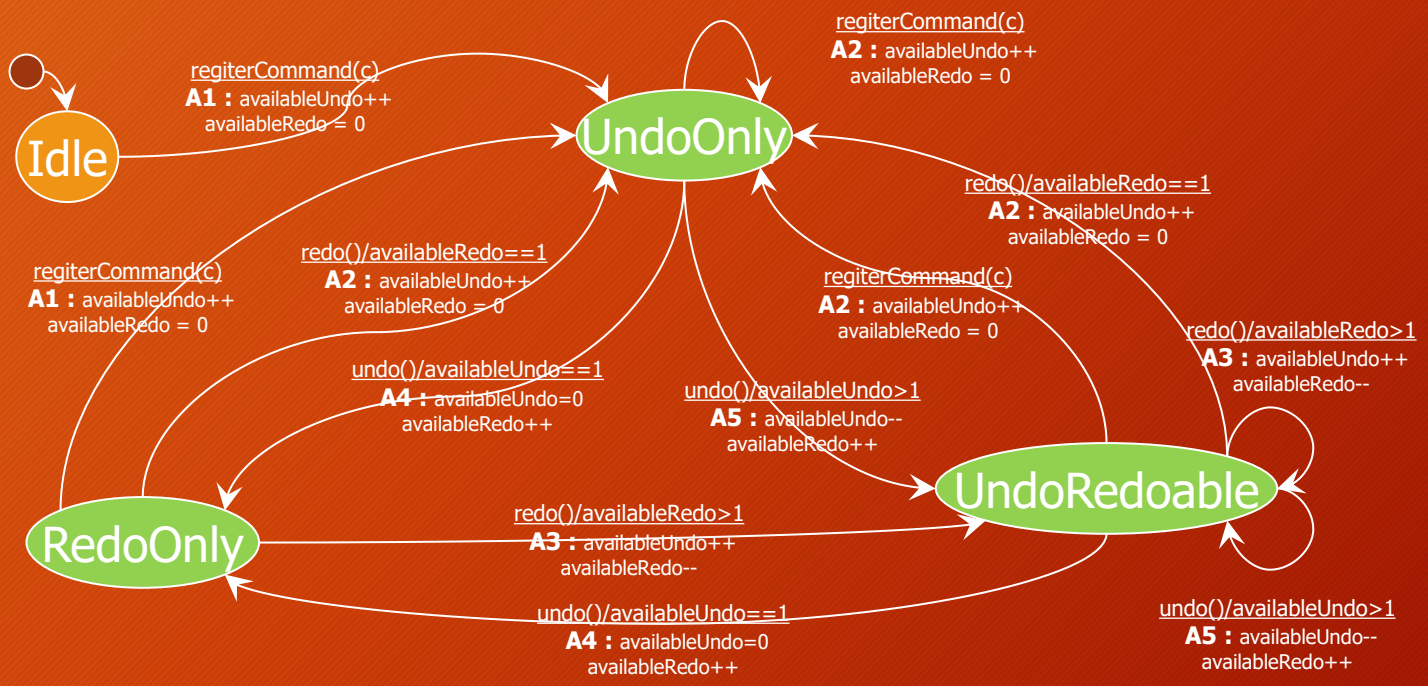
Undo / Redo : Comment ?

- La classe commande propose une méthode undo
- Le client doit gérer une historisation
  - Quelle structure de données ?
  - Quelle capacité

Undo/Redo : Comportement

Events:
- registerCommand(c)
- undo()
- redo()

Actions:
-availableUndo, availableRedo : Collection<Command>
-availableXXX++ ⇔ add c to collection
-availableXXX-- ⇔ remove c from collection
-availableXXX = 0 ⇔ clear collection



Undo / Redo : Java Swing

- public class **UndoManager**
  extends CompoundEdit
  implements UndoableEditListener

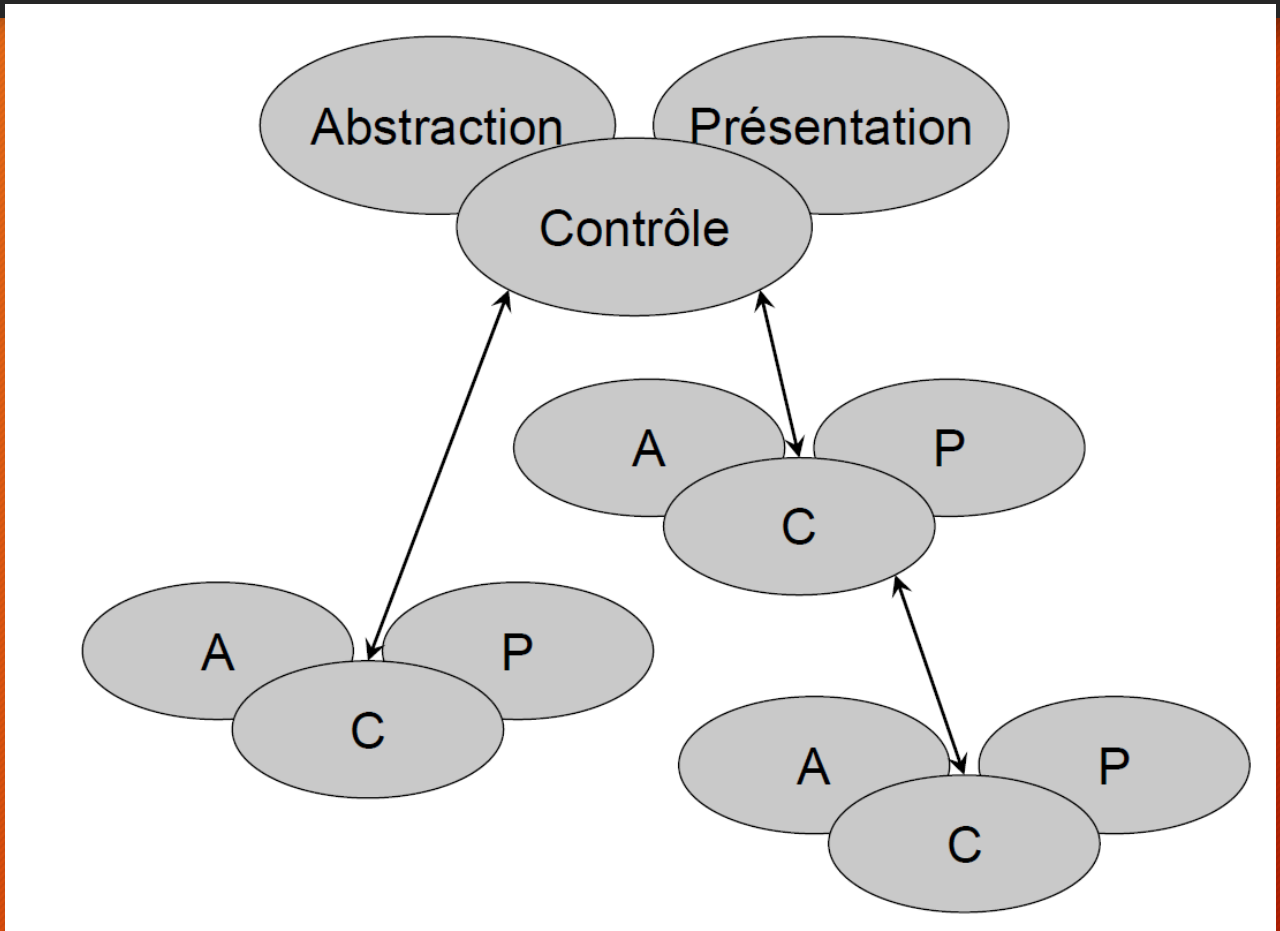
```
JTextField tf = ...;
tf.getDocument().addUndoableEditListener(undoManager);
```

Variantes de MVC

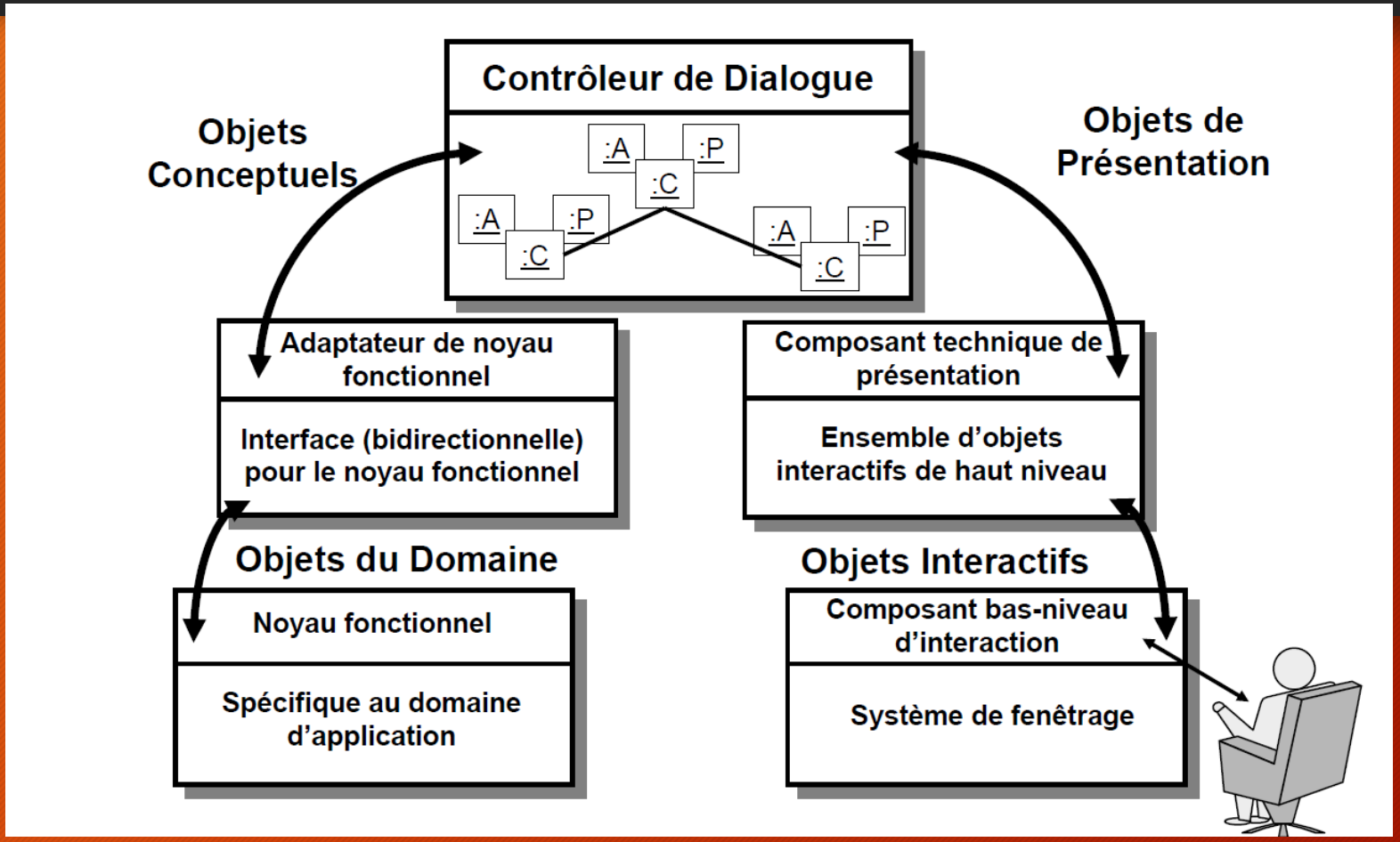
PAC : Définitions

- Joëlle Coutaz, 1987
- Présentation
  - définit le comportement, en entrée comme en sortie, de l'application par rapport à l'utilisateur
  - ne communique qu'avec son contrôleur
- Abstraction
  - les fonctionnalités de l'application
  - ne communique qu'avec son contrôleur
- Contrôle
  - maintien de la cohérence entre présentation et abstraction
  - peut communiquer avec d'autres contrôleurs

Presentation-Abstraction-Control (PAC)

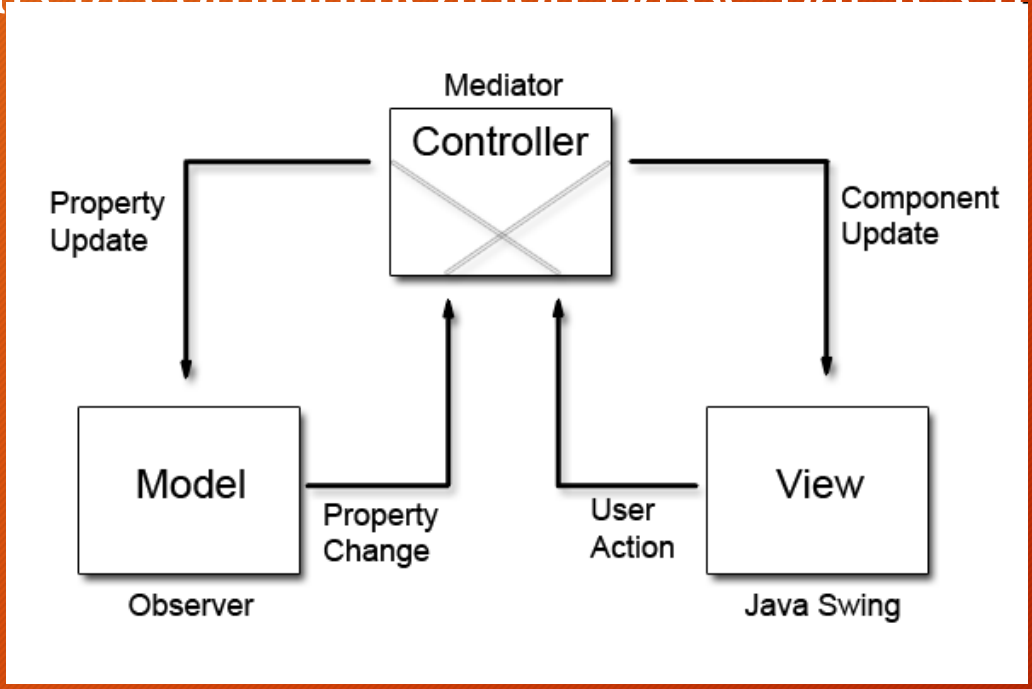


PAC-Amodeus


Contrôleur de Dialogue
:A
:P
:C
:A
:P
:C
:A
:P
:C
Objets Conceptuels
Objets de Présentation
Adaptateur de noyau fonctionnel
Interface (bidirectionnelle) pour le noyau fonctionnel
Composant technique de présentation
Ensemble d'objets interactifs de haut niveau
Objets du Domaine
Noyau fonctionnel
Spécifique au domaine d'application
Objets Interactifs
Composant bas-niveau d'interaction
Système de fenêtrage

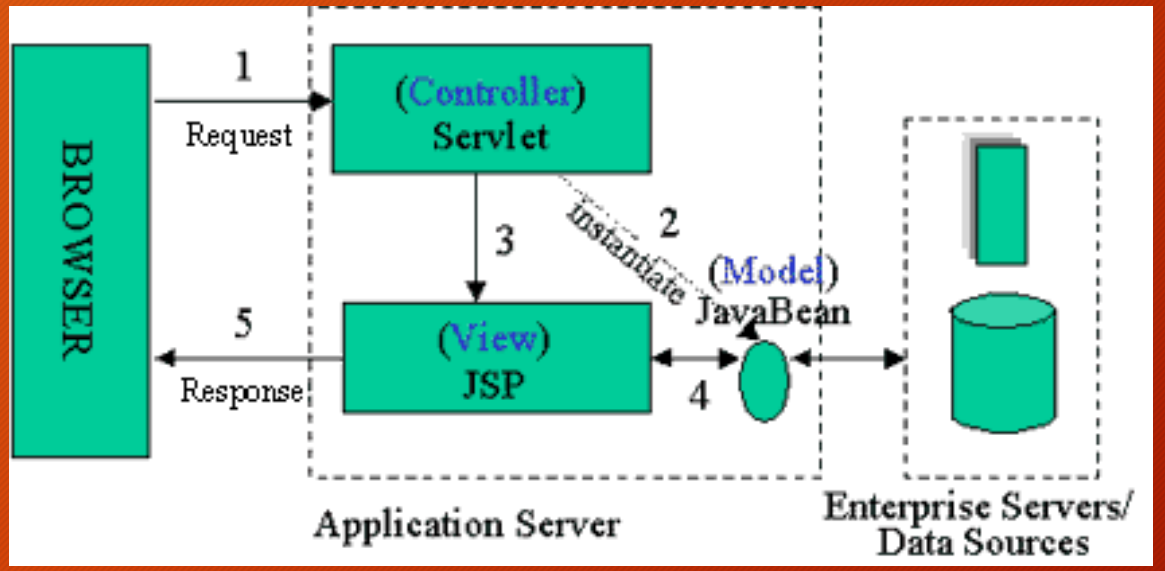
Modified MVC (Oracle/Sun)
OS X COCOA (Apple)

- Robert Eckstein, Mars 2007
- Toute communication passe par le contrôleur
- PAC mais la communication passe par les modèles



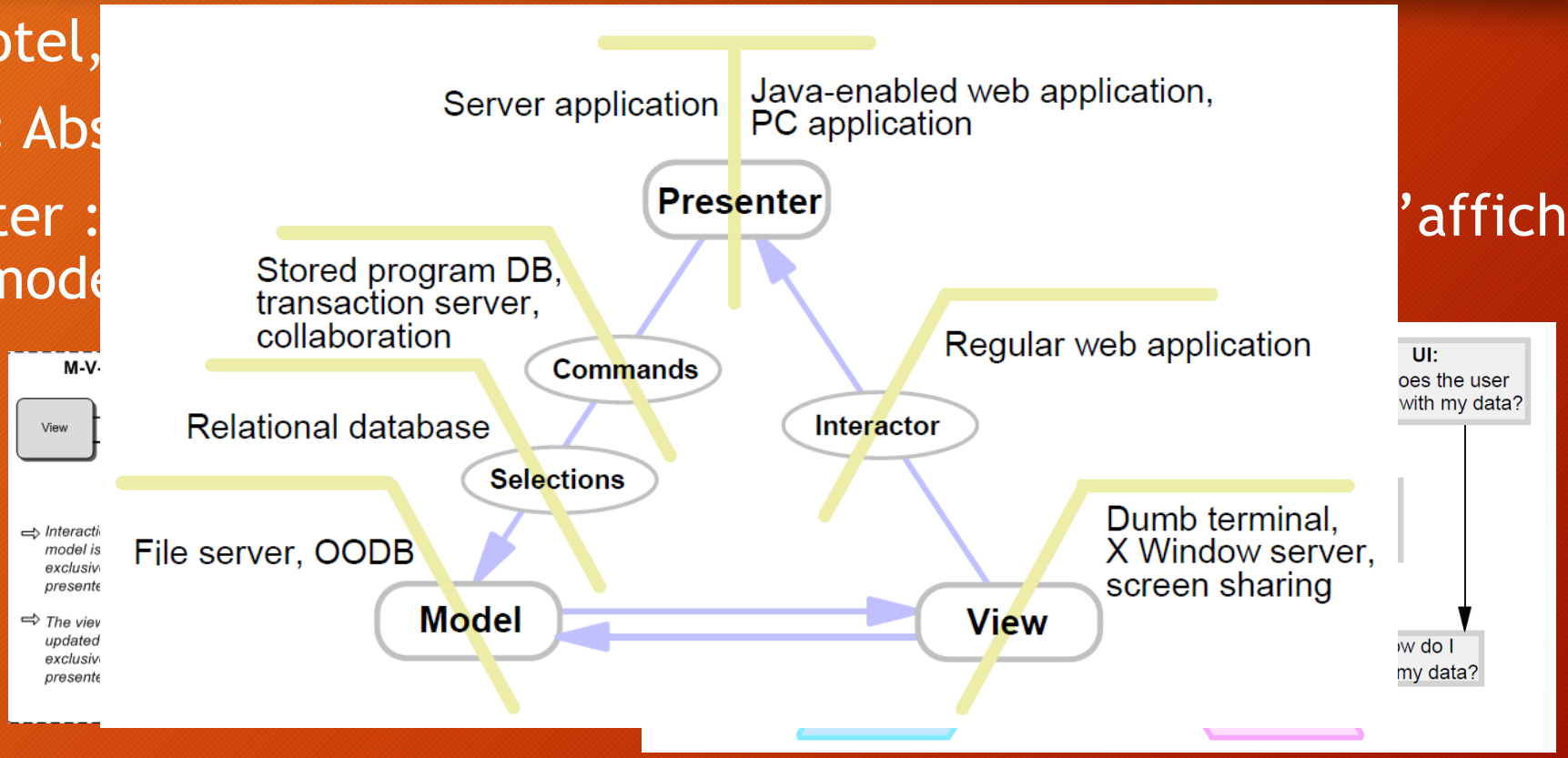
MVC 2

- JavaServer Pages
- Contrôleur unique
- Page affichée sélectionnée par le contrôleur



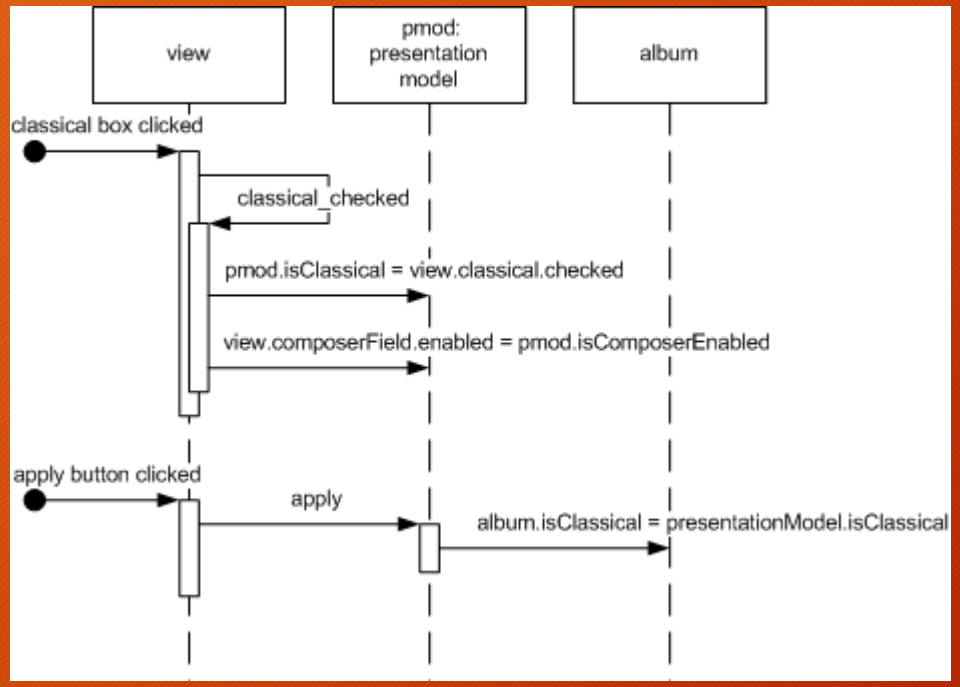
Model-View-Presenter (MVP)

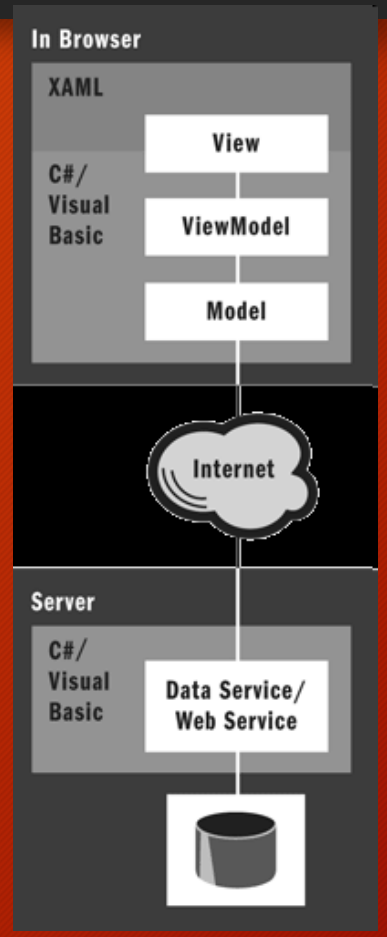
- Mike Potel,
- Model : Abs
- Presenter : ’affichage et agit sur le mode



Model-View-ViewModel (MVVM) et Presentation Model (PM)

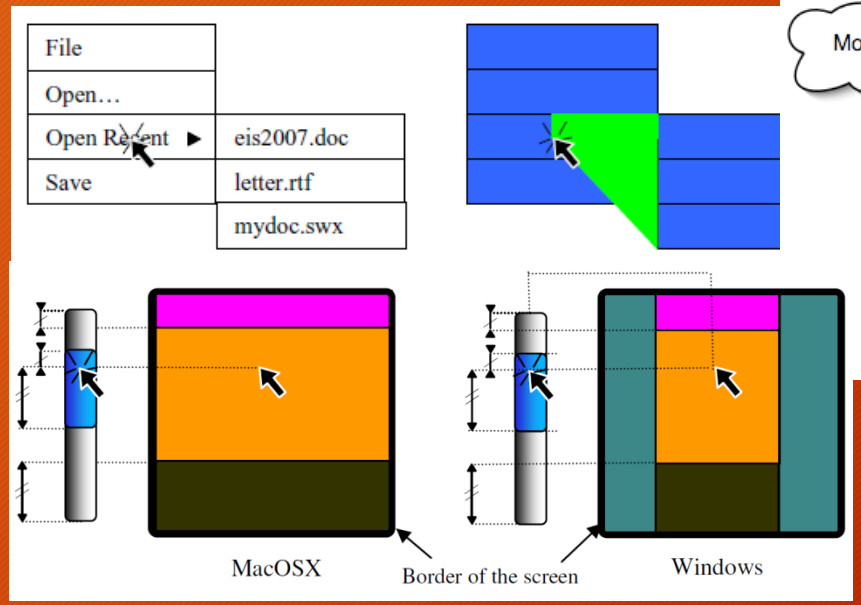
- PM Martin Fowley, 2004
- MVVM John Gossman(Microsoft), 2005, spécialisation de PM pour silverlight
- PM est une abstraction de la Vue, indépendante de la plateforme

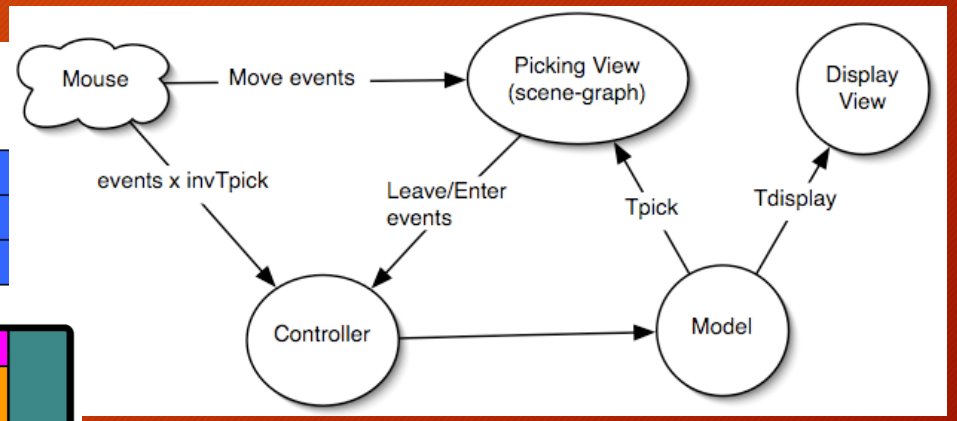




Model - Display View – Picking View – Controller (MDPC)

- Conversy, S., Barboni, E., Navarre, D., Palanque, P. "Improving modularity of interactive software with the MDPC architecture" EIS 2007.
- Vue invisible facilitant la description de l'interaction





Quelle architecture choisir ?

- Un problème, une architecture ?

- Dépend de ce qui doit être « absorbé » par l'architecture
  - Au moment de la conception du système
  - Evolution du système

Les design patterns comme support à l'utilisabilité

De nombreux design patterns

- Archétypes de solution à des problèmes récurrents dans la conception de logiciel
- Bibliographie
  - Design Patterns, Elements of Reusable Object-Oriented Software, E.Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, le «Gang of Four», 1995
  - A System of Patterns : Pattern-oriented software architecture, Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. A Stal, Wiley 1996
  - …
- Généralement regroupés en pattern touchant la création, la structure ou le comportement

Design patterns orientés Systèmes Interactifs

- MVC
- Event Handling
- Object for State
- Visitor
- Command
- Cancellation
- …

- Comment choisir ?

Achieving Usability Through Software Architecture

- Len Bass, Bonnie E. John, Jesse Kates. Technical Report, CMU/SEI-2001-TR-005, March 2001

- Principes
  - Identification de scénarios d'utilisabilité
  - Organisation de ces scénarios en terme de bénéfice pour l'utilisabilité
  - Organisation des scénarios en terme de mécanismes logiciels

- Résultat : une classification des patterns

Scénarios d'utilisabilité et pattern correspondant

- Identification de scénarios d'utilisabilité qui ont un impact sur l'architecture logicielle

- Exemple
  - L'utili
  - L'utili
  - Ex: La

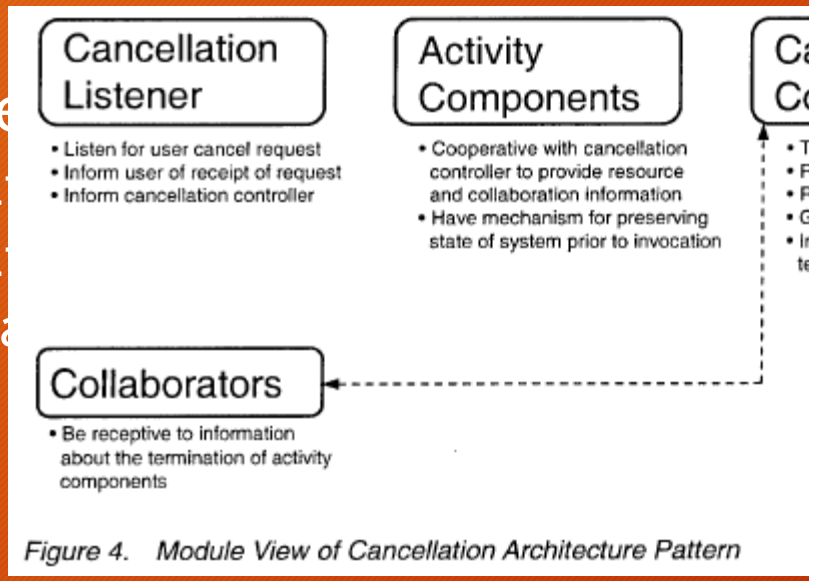


Figure 4. Module View of Cancellation Architecture Pattern

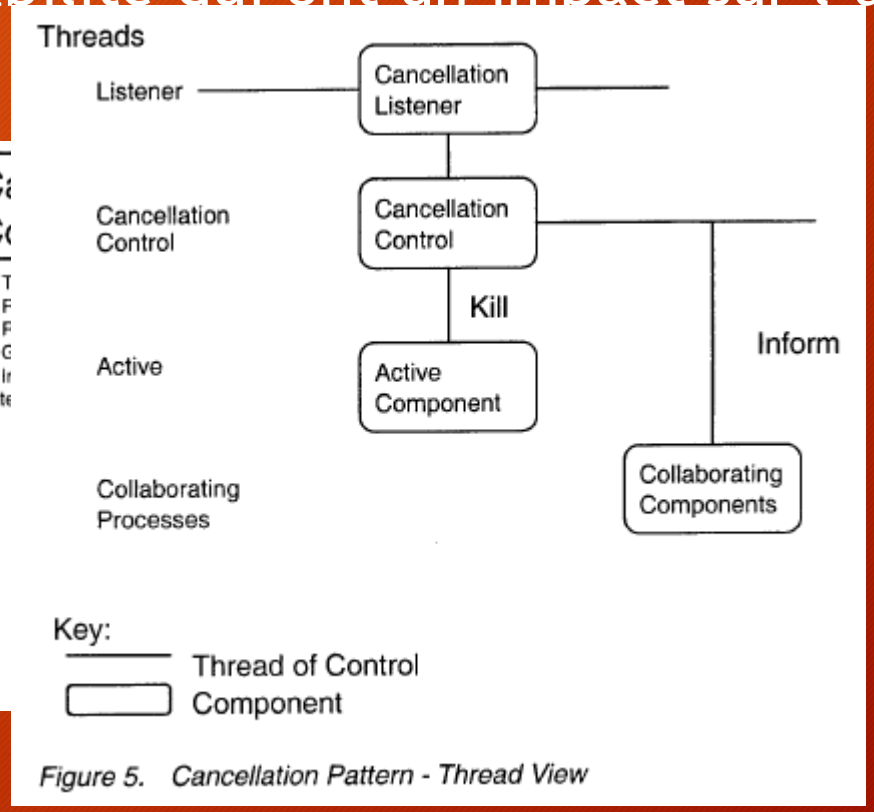


Figure 5. Cancellation Pattern - Thread View

Critère bénéfice pour l'utilisabilité

- **Les critères**

Increases individual user effectiveness
- *Expedites routine performance*
  - Accelerates error-free portion of routine performance
  - Reduces the impact of routine user errors (slips)
- *Improves non-routine performance*
  - Supports problem-solving
  - Facilitates learning
- *Reduces the impact of user errors caused by lack of knowledge (mistakes)*
  - Prevents mistakes
  - Accommodates mistakes

Reduces the impact of system errors
- *Prevents system errors*
- *Tolerates system errors*

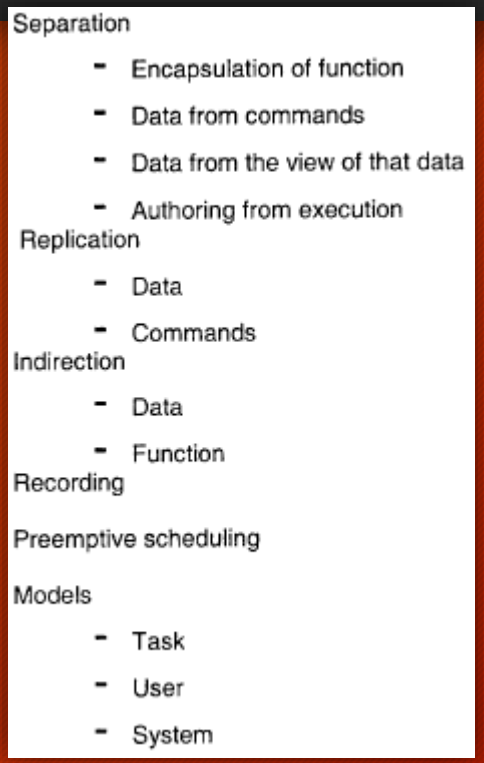
Increases user confidence and comfort

- **Ex: Reduces the impact of routine user errors (slips)**
  - The negative impact of routine user errors can be reduced in two ways. First, since users will always slip, reducing the number of opportunities for error (roughly corresponding to the number and difficulty of steps in a given procedure) will usually reduce its occurrence. Second, systems can be designed to better accommodate user slips by providing adequate recovery methods

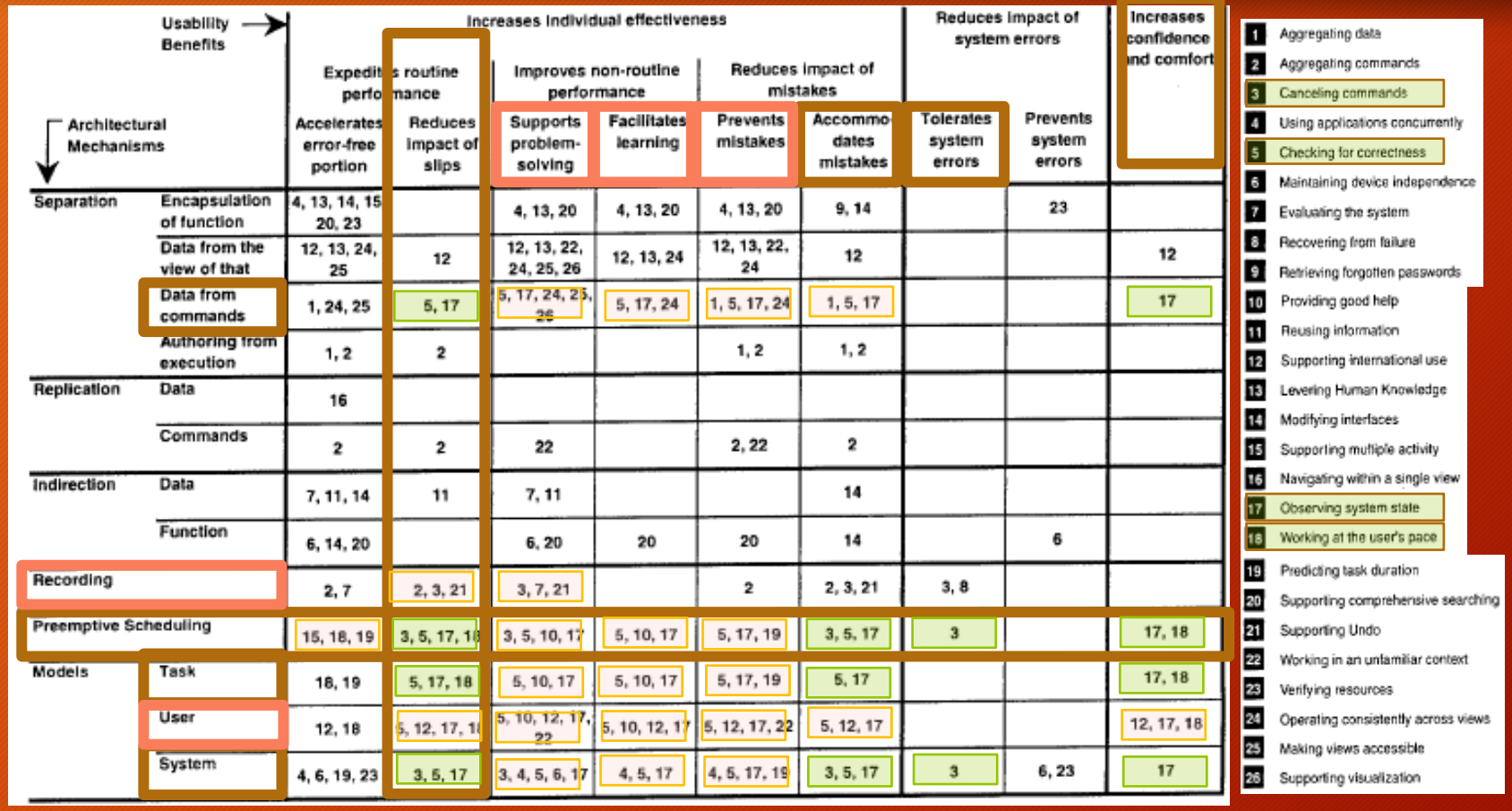
Critères mécanismes logiciels

- **Les critères**

- **Ex: Preemptive Scheduling**
  - Preemptive scheduling allows the software system to have multiple simultaneous activities. In fact, the activities are not simultaneous when examined at a tiny time scale (measured in terms of microseconds) but they appear simultaneous when examined at a larger time scale (measured in terms of 10s of milliseconds). The term thread refers to a logical sequence of activities within the computer system. At any point in time, a thread is either active (consuming the processor resource) or blocked (waiting for a resource or for some input). Having multiple simultaneous activities is expressed as having multiple threads. Having multiple threads is most often accomplished (although not exclusively) by using a preemptive processor scheduling strategy



Résultat : Classification



Architectural Mechanisms \ Usability Benefits		Increases individual effectiveness: Expedites routine performance — Accelerates error-free portion	Increases individual effectiveness: Expedites routine performance — Reduces impact of slips	Increases individual effectiveness: Improves non-routine performance — Supports problem-solving	Increases individual effectiveness: Improves non-routine performance — Facilitates learning	Increases individual effectiveness: Reduces impact of mistakes — Prevents mistakes	Increases individual effectiveness: Reduces impact of mistakes — Accommodates mistakes	Reduces impact of system errors — Tolerates system errors	Reduces impact of system errors — Prevents system errors	Increases confidence and comfort
Separation	Encapsulation of function	4, 13, 14, 15 20, 23		4, 13, 20	4, 13, 20	4, 13, 20	9, 14		23	
	Data from the view of that	12, 13, 24, 25	12	12, 13, 22, 24, 25, 26	12, 13, 24	12, 13, 22, 24	12			12
	Data from commands	1, 24, 25	5, 17	5, 17, 24, 25, 26	5, 17, 24	1, 5, 17, 24	1, 5, 17			17
	Authoring from execution	1, 2	2			1, 2	1, 2			
Replication	Data	16								
	Commands	2	2	22		2, 22	2			
Indirection	Data	7, 11, 14	11	7, 11			14			
	Function	6, 14, 20		6, 20	20	20	14		6	
Recording		2, 7	2, 3, 21	3, 7, 21		2	2, 3, 21	3, 8		
Preemptive Scheduling		15, 18, 19	3, 5, 17, 18	3, 5, 10, 17	5, 10, 17	5, 17, 19	3, 5, 17	3		17, 18
Models	Task	18, 19	5, 17, 18	5, 10, 17	5, 10, 17	5, 17, 19	5, 17			17, 18
	User	12, 18	5, 12, 17, 18	5, 10, 12, 17, 22	5, 10, 12, 17	5, 12, 17, 22	5, 12, 17			12, 17, 18
	System	4, 6, 19, 23	3, 5, 17	3, 4, 5, 6, 17	4, 5, 17	4, 5, 17, 19	3, 5, 17	3	6, 23	17

1. Aggregating data
2. Aggregating commands
3. Canceling commands
4. Using applications concurrently
5. Checking for correctness
6. Maintaining device independence
7. Evaluating the system
8. Recovering from failure
9. Retrieving forgotten passwords
10. Providing good help
11. Reusing information
12. Supporting international use
13. Levering Human Knowledge
14. Modifying interfaces
15. Supporting multiple activity
16. Navigating within a single view
17. Observing system state
18. Working at the user's pace
19. Predicting task duration
20. Supporting comprehensive searching
21. Supporting Undo
22. Working in an unfamiliar context
23. Verifying resources
24. Operating consistently across views
25. Making views accessible
26. Supporting visualization

Avantages

- Avantages
  - Design rationale
  - Evaluation d’architecture existante
  - Evaluation du coût d’une évolution
- Limites
  - Evaluation du coût en terme de performance, sécurité, sûreté non fait