Une comparaison de patrons d'architecture logicielle pour l'IHM à bas niveau

Stéphane Conversy, Mathieu Magnaudet, Vincent Peyruqueou, Mathieu Poirier

▶ **To cite this version:**

Stéphane Conversy, Mathieu Magnaudet, Vincent Peyruqueou, Mathieu Poirier. Une comparaison de patrons d'architecture logicielle pour l'IHM à bas niveau. 2025. ⟨hal-05454476⟩

HAL Id: hal-05454476

https://hal.science/hal-05454476v1

Preprint submitted on 12 Jan 2026

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY-NC-SA 4.0 - Attribution - Non-commercial use - ShareAlike - International License

A comparison of patterns of software architecture for low-level UI
Une comparaison de patrons d'architecture logicielle pour l'IHM à bas niveau

Stéphane Conversy, Mathieu Magnaudet, Vincent Peyruqueou, Mathieu Poirier

Université de Toulouse, ENAC, ISAE-SUPAERO, TSAAE Lab, Toulouse, France, fistname.lastname@enac.fr

There are many patterns of software architecture for human-computer interaction. They have been designed, improved, refined and even reinvented according to a range of considerations by researchers and practitioners, as well as technological developments. However, to our knowledge, there is no synthesis or characterization of these architectures. In this article, we propose a partial synthesis of architecture patterns, with the intentions of their inventors and a formalization of their behavior. We also provide an analysis of the concerns and the components, applying the separation of concerns principles inside the Controller itself. Such a synthesis could enable future and current practitioners to better understand and compare them, in order to make informed choices during software engineering processes.

CCS CONCEPTS • Insert your first CCS term here • Insert your second CCS term here • Insert your third CCS term here

Additional Keywords and Phrases: Software Architecture, Pattern, Human-Computer Interaction, User Interface

Il existe de nombreux patrons d'architecture logicielle pour l'interaction humain-machine. Ils ont été conçus, améliorés, raffinés, voire réinventés en fonction d'un ensemble de considérations des chercheurs et des praticiens et des évolutions des technologies. Cependant, il n'existe pas à notre connaissance de synthèse et de caractérisation de ces patrons d'architecture. Dans cet article, nous proposons une synthèse partielle de patrons d'architecture, avec les intentions de leurs inventeurs et une formalisation de leur comportement. Nous proposons ensuite une analyse des préoccupations et des composants, en appliquant notamment le principe de la séparation des préoccupations au sein du composant Controller. La synthèse et l'analyse pourraient permettre aux futurs et actuels praticiens de mieux les comprendre et de mieux les comparer, afin de les choisir en connaissance de cause lors des process d'ingénierie logicielle.

Mots-clés additionnels : Architecture logicielle, Patron, Interaction Humain-Machine, Interface Utilisateur

ACM Reference Format:
First Author's Name, Initials, and Last Name, Second Author's Name, Initials, and Last Name, and Third Author's Name, Initials, and Last Name. 2018. The Title of the Paper: ACM Conference Proceedings Manuscript Submission Template: This is the subtitle of the paper, this document both explains and embodies the submission format for authors using Word. In Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY. ACM, New York, NY, USA, 10 pages. NOTE: This block will be automatically generated when manuscripts are processed after acceptance.

1. INTRODUCTION

Les organisations, équipes ou individus qui créent des systèmes informatiques emploient souvent une architecture logicielle afin d'organiser le code de leur système [24]. Une architecture est l'ensemble des « concepts fondamentaux ou les propriétés d'une entité dans son environnement et les principes régissant la réalisation et l'évolution de cette entité et des processus du cycle de vie qui y sont liés » [24]. Ainsi, pour un système informatique donné, une architecture particulière permet d'organiser le code. Une telle architecture peut suivre des modèles ou des patrons d'architecture [11].

Pour les systèmes interactifs, il existe plusieurs de ces patrons d'architecture, dont l'un des plus répandus est « Model View Controller ». Ces patrons ont été conçus en symbiose avec l'avènement des interactions de plus en

plus sophistiquées qu'ont introduites les évolutions des technologies d'entrée/sortie comme le stylo optique, la souris, l'écran graphique ou les dispositifs multimodaux. Leurs concepteurs ont inventé ces patrons, les ont raffinés, ont conduit des ateliers [17] pour mieux les définir, d'autres se les sont appropriés, les ont détournés, ou réinventés.

A notre connaissance, il n'existe pas de synthèse de ces patrons d'architecture logicielle pour l'IHM. Une conséquence de ce fait est qu'il est parfois difficile de comprendre les similitudes et différences parfois subtiles qu'il peut exister entre ces patrons. Par ailleurs, les intentions de leurs auteurs sont peu mentionnées, ce qui nuit à leur compréhension et à leur comparaison. Enfin, les exemples illustrant ces patrons font souvent référence à des interactions avec des widgets, au détriment des interactions de type « manipulation directe », et n'aident pas les développeurs d'applications hautement interactives.

Les questions de recherche sont les suivantes :

RQ1 Quelles sont les différences entre les patrons MVC, MVC-ST80, PAC, MVVM, MVZmC, MDPC ? (Réponse dans la section 4)

RQ2 Quelles sont les préoccupations séparées par ces patrons ? (Réponse section 5.2 et 5.3)

RQ3 Quel est le rôle d'un contrôleur in fine ? (Réponse section 5.4)

Dans cet article, nous proposons une analyse d'un ensemble partiel de patrons d'architecture logicielle pour l'IHM. Cet ensemble est partiel car nous ne prétendons pas à l'exhaustivité, et que nous ne nous intéressons qu'aux interactions de bas-niveau.

Dans une première partie, qui se veut factuelle, nous analysons les intentions, exigences, approches des auteurs. Nous formalisons les interactions entre composant avec des « diagrammes de séquence », qui nous semblent appropriés pour mettre en valeur les similitudes et les différences entre modèles d'architecture. Par ailleurs, nous montrons des différences d'extraits de code mettant en œuvre chacun des modèles d'architecture, afin d'aider à la compréhension du lecteur. Dans une deuxième partie, nous proposons une analyse des préoccupations et des composants de ces patrons.

2. BIBLIOGRAPHIE

Cette section ne présente pas les articles décrivant les patrons car ces derniers sont décrits et référencés dans les sections suivantes. Nous nous appuyons évidemment sur les publications archivées, mais aussi sur des documents en ligne, ou des messages de discussion eux aussi en ligne. A des fins de préservation, nous avons inclus en annexe les documents dont nous pensons qu'ils risquent de disparaître.

Il y a eu plusieurs tentatives de synthétiser ces patrons d'architecture. La première d'entre elles est le résultat de l'atelier qui s'est tenu à Seeheim, en Allemagne, en 1983 [17]. Comme tout travail, l'atelier ne pouvait pas discuter des architectures plus récentes. Le modèle Arch/Slinky se déclare comme un modèle revisité de Seeheim, et définit plusieurs composants en formalisant leurs interactions avec une « arche », dont certains éléments peuvent avoir des tailles de dimension variable [23]. Ces documents des informations sur la séquence exacte d'interaction entre les composants, ou de code explicitant comment se traduisait ces concepts de façon pragmatique.

La notion de patron de conception (Design Patterns) [11] est antérieure à ces ateliers, même si elle sous-tendait les préoccupations des participants. Un patron de conception est « une description d'objets et de classes communicants qui sont personnalisés pour résoudre un problème de conception général dans un contexte particulier ». C'est justement la réflexion sur les architectures de type MVC qui a conduit les auteurs de Design Patterns à les abstraire sous forme de patron de conception. Plusieurs variations de MVC, mettant en œuvre des patrons comme le Mediateur, ont été analysés [1].

Reenskaug, l'auteur de MVC, a tenté de ré-analyser plusieurs architectures, mais son travail n'a pas abouti, et il ne compare pas avec les patrons plus récents. [20]

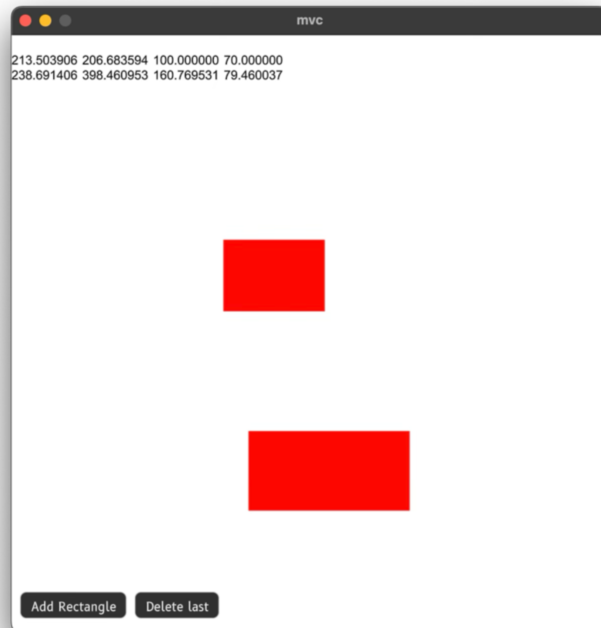
Les réflexions sur l'architecture logicielle utilisent souvent le principe de Séparation des Préoccupations (*Separation of Concerns*) [9]. Il s'agit de déterminer pour un code donné l'ensemble des préoccupations qu'il implémente, et de les séparer en modules distincts. Cette séparation a un objectif de compréhensibilité et maintenabilité. De manière similaire, la conception structurée (*Structured design*) est le « processus consistant à décider quels composants interconnectés de telle ou telle manière permettront de résoudre un problème bien spécifié » [22], en tenant compte des coûts de mise en œuvre (raisonnablement faible, soluble séparément), de maintenance (facilement lié à l'application, raisonnablement faible, corrigeable séparément) et de modification (facilement lié au problème, modifiable séparément). La mise en œuvre, la maintenance et la modification sont généralement minimisées lorsque chaque élément du système correspond exactement à un petit élément bien défini du problème et que chaque relation entre les éléments d'un système correspond uniquement à une relation entre des éléments du problème. Pour Ousterhoot, l'objectif est de minimiser la complexité, c-à-d « tout ce qui est lié à la structure d'un logiciel qui rend difficile la compréhension et la modification d'un système » [16]. Les symptômes de complexité comprennent l'amplification d'un changement. Pour Meyer, les qualités importantes d'un logiciel sont la correction, la robustesse, l'extensibilité et la réutilisabilité, ces deux dernières étant souvent évoquées pour expliquer l'intérêt d'un patron [14].

3. CADRE D'ANALYSE

Les analyses que nous produisons dans cet article se font sur la base des articles originaux. Nous retranscrivons les descriptions des auteurs des articles, ainsi qu'une partie de leurs intentions. Nous produisons les diagrammes de séquence que ces descriptions impliquent. Enfin nous donnons des exemples de code.

Le code est celui d'un petit programme utilisant la manipulation directe qui permet de créer des rectangles dans une fenêtre avec un bouton idoine, de les déplacer par « presser-tirer-relâcher » sur le centre du rectangle, et de les redimensionner par « presser-tirer-relâcher » des bords des rectangles. Par ailleurs, quatre informations sont affichées dans la fenêtre pour chaque rectangle sous forme d'une ligne de texte, représentant la position en x et y et la largeur et hauteur du rectangle. Ces quatre textes sont aussi manipulables, en positionnant le curseur de la souris au-dessus, et en tournant la molette de la souris pour augmenter ou diminuer la valeur manipulée. Les deux vues (graphique et textuelle) sont synchronisées : manipuler l'une changera l'autre et vice-versa.

Comme MVC ST-80 est le patron le plus connu d'entre tous, nous en donnons le code correspondant. Cependant, l'objectif de notre article est de comprendre les similitudes et les différences entre chaque patron. Au lieu de produire le code correspondant à chaque patron, nous le fournissons sous forme de différence avec le code de MVC, ou le code du patron dont il est directement tiré. Le code est écrit dans le langage orienté interaction Smala [13], car ce langage a été conçu pour éliminer la complexité accidentelle de description des interactions inhérente aux langages orienté fonction ou objet p. ex. Nous fournissons également une implémentation en Python en annexe.



4. ANALYSE DE PATRONS D'ARCHITECTURE

Les patrons d'architecture suivants sont décrits dans un ordre « presque » chronologique. C'est en effet par amélioration successive de patrons existants que leurs auteurs ont conçu de nouveaux patrons. Une présentation chronologique permet de mieux les comparer. Cependant, le patron MVC que nous estimons le plus connu n'est pas celui qui a été présenté originellement par Reenskaug [19], mais une réinterprétation et son implémentation dans Smalltalk 80 (ST-80) par d'autres personnes (cf annexe A5). C'est la raison pour laquelle nous partons de la version ST-80 (la plus connue), afin d'expliquer les suivantes.

Chaque sous-section a la même structure : les intentions déclarées des auteurs, les descriptions des concepts originaux, notre interprétation sous forme de diagramme de séquence, notre interprétation sous forme de code ou de différence de code. Pour ce qui est des intentions de la description des concepts originaux, nous avons choisi de garder la langue dans laquelle ils ont été exprimés. Les composants des diagrammes de séquence sont ordonnés de plus proche de l'utilisateur (donc, User puis ce qui fait office de View) au plus éloigné (souvent, le Model).

4.1.MVC ST-80 (Model – View – Controller)

4.1.1.Intentions

Nous ne connaissons pas les intentions des auteurs de ST-80 qui ont réinterprété le patron MVC original. Cependant, nous formons l'hypothèse que l'intention originale est restée la même, et est la suivante.

"The essential purpose of MVC is to bridge the gap between the human user's mental model and the digital model that exists in the computer. The ideal MVC solution supports the user illusion of seeing and manipulating the domain information directly. The structure is useful if the user needs to see the same model element simultaneously in different contexts and/or from different viewpoints."

4.1.2.Exigences et Approches

Des intentions exprimées, nous extrayons les exigences (Ex) et approches (App) suivantes :

Ex1$_{MVC}$ Rapprocher les modèles mentaux et conceptuels *: bridge the gap between the human user's mental model and the digital model.*

Ex2$_{MVC}$ Manipuler Directement : *supports the user illusion of seeing and manipulating the domain information directly.*

Ex3$_{MVC}$ Plusieurs vues du même modèle : *see the same model element simultaneously in different contexts and/or from different viewpoints.*

L'auteur original ne semble pas mettre en avant les exigences traditionnelles de génie logiciel de réutilisation et d'extensibilité, mais plutôt de l'ordre de ce que doit ressentir l'utilisateur (cf Annexe A.4: *" The user was the czar; everything done at LRG was done to support him"*)

4.1.3.Référence originale

L'article original qui décrit MVC est laconique (~1,20 page) et décrit 4 concepts [19] : Model, View, Controller et Editor. Seuls les 3 premiers sont passés à la postérité dans ST-80, en tout cas pour les termes, car il y a eu un échange de sémantique entre Controller et Editor dans ST-80 (cf la discussion dans la sous-section 5.1).

"A model could be a single object (rather uninteresting), or it could be some structure of objects.

A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter.

A view should never know about user input, such as mouse operations and keystrokes.

A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. All these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents."

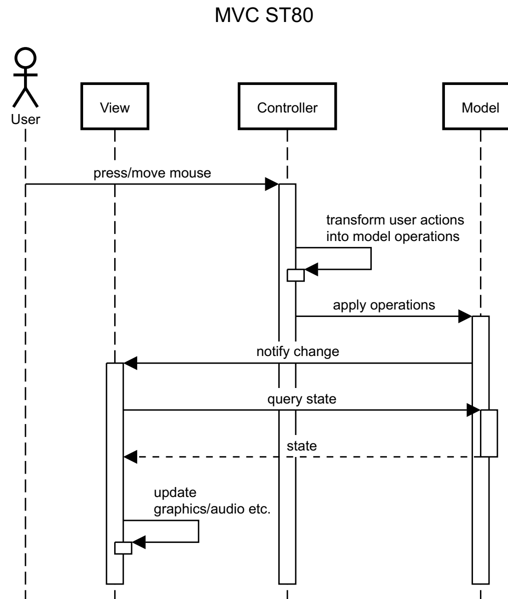
Pour ce qui est du Contrôleur ST-80, nous retranscrivons sa description à partir de la description que Reenskaug fait de l'Editor (Reenskaug : « *The Smalltalk-80 Controller is here a fourth element called Editor* »), comme suit :

"~~An Editor~~ [A (ST-80) controller] permits the user to modify the information that is presented by the view."

4.1.4.Diagramme de séquence

Le diagramme de séquence formalise le document original de 1,2p en ne retenant que les 3 premiers composants (M, V et C, pas E), et en appliquant la sémantique de E pour C. Le Controller reçoit les inputs et notifier le Modèle, qui va à son tour notifier la Vue. Avec MVC original, la notification ne transporte pas de modification : c'est la Vue qui va récupérer l'état du Modèle pour adapter ses graphismes. Un deuxième diagramme de séquence disponible

en annexe permet de formaliser l'utilisation du patron « sujet/observateur » et le fait que le Modèle a une liste d'observateurs Vue (dans ce cas au nombre de 2) qu'il notifie l'une après l'autre.



4.1.5.Code

Notre implémentation MVC ST-80 de l'application exemple « éditeur de rectangle » comprend 5 composants :

- un Modèle de rectangle
- un couple Vue/Contrôleur par type de rendu (Textuel et Graphique)

```
_define_
RectModel(double _x, double _y, double _w, double _h) {
  Double x (_x)
  Double y (_y)
  Double width (_w)
  Double height (_h)

  Spike changed                  // this will serve as a ‘signal’ for notification
  x, y, width, height -> changed // establish a notification whenever a component changes
}

_define_
TextView(Process model_, int _ty) {
  model aka model_ // retain the parameter named “model_” into a component of TextView named “model”

  Translation pos (0, _ty)

  // the 4 text graphical objects...
  Text t_x (0, 15, "0")
  Text t_y (15, 15, "0")
```

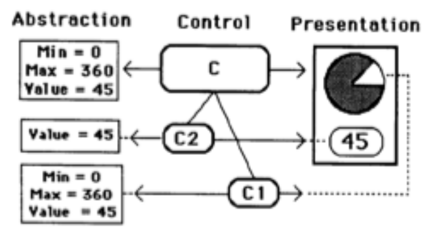
```
  Text t_width (30, 15, "0")
  Text t_height (45, 15, "0")

  // ... laid out horizontally thanks to a data-flow =:>
  t_x.x     + t_x.width     + 5 =:> t_y.x
  t_y.x     + t_y.width     + 5 =:> t_width.x
  t_width.x + t_width.width + 5 =:> t_height.x

  // 'API' to enable changing the content of the 4 graphical texts regardless of their implementation
       x aka t_x.text
       y aka t_y.text
   width aka t_width.text
  height aka t_height.text

  // update the view whenever the model changes (subject/observer pattern)
  // =: is an assignment done only once, as opposed to =:> which operates continuously
  model.changed -> { model.{x,y,width,height} =: this.{x,y,width,height} }
}

_define_
TextController(Process model, Process view_) {
  view aka view_
  // update model from interactions on the view by establishing a dataflow
       view.t_x.wheel.dy +=> model.x
       view.t_y.wheel.dy +=> model.y
   view.t_width.wheel.dy +=> model.width
  view.t_height.wheel.dy +=> model.height
[…]
_define_
GraphicsView(Process model_) {
  model aka model_
  FillColor fill (Red)
  Rectangle r (0,0,0,0)

  // update the view whenever the model changes (subject/observer pattern)
  model.changed -> {
    model.{x,y,width,height} =: r.{x,y,width,height}
  }
}
```

Le code de la Vue graphique montre l'abonnement au changement du modèle et sa transformation en graphisme (motif sujet/observateur). Le code du Contrôleur de texte consiste à retranscrire les mouvements de la molette au-dessus des textes en opération sur le modèle. Le code du Contrôleur de graphique qui régit l'interaction avec le rectangle, notamment le redimensionnement des côtés du rectangle par « presser-tirer-relâcher », est plus complexe en raison de la détermination des endroits où s'effectue l'interaction (centre ou bords). Il est montré en annexe A.3, et sera discuté dans la suite de l'article.

4.2.PAC (Presentation – Abstraction - Control)

4.2.1.Intentions

PAC [8] is an implementation model that attempts to bridge the gap between the abstract sphere of theoretical models and the practical affairs of building user interfaces. It takes as a basis the vertical decomposition of human-computer interaction into semantic, syntactic and pragmatic layers as promoted by some theoretical models. However, PAC stresses the fact that these notions do not form strict monolithic layers but are distributed across related "chunks", called interactive objects.



4.2.2.Exigences et Approches

Des intentions exprimées, nous extrayons les exigences (Ex) et approches (App) suivantes :

Ex1$_{PAC}$ Rapprocher les modèles théoriques et l'implémentation pratique

App1$_{PAC}$ Décomposer l'IHM en couches

App2$_{PAC}$ Distribuer les couches dans des objets interactifs

Par rapport à MVC, ces exigences sont davantage de l'ordre du génie logiciel.

4.2.3.Référence originale

L'article original décrit 3 composants (ou *interactive objects*) [8] : l'Abstraction, la Présentation et le Contrôleur.

the Abstraction part corresponds to the semantics of the application. It implements the functions that the application is able to perform.

the Presentation defines the concrete syntax of the application, i.e. the input and output behavior of the application as perceived by the user;

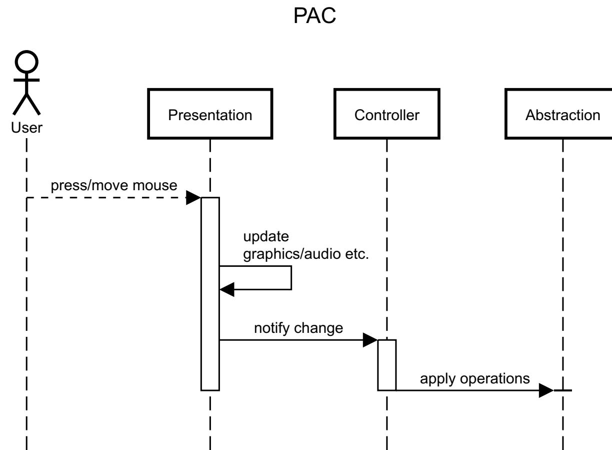
the Control part maintains the mapping and the consistency between the abstract entities (involved in the interaction and implemented in the Abstract part) and their presentation to the user. It embodies the boundary between semantics and syntax. It is intended to hold the context of the overall interaction between the user and the application.

Historiquement, PAC n'est pas une évolution de MVC, il a été conçu indépendamment. Dans l'article original, PAC est comparé avec MVC, en mettant en avant le fait que PAC n'essaie pas de décomposer une interaction bas-niveau en V+C, mais l'intègre comme un tout dans le composant P.

Un aspect important de PAC non discuté ici est celui de la hiérarchie : PAC se veut une description fractale, composés de sous-éléments eux-mêmes décrits avec PAC, ce qui permet de décrire une application interactive dans son entièreté quel que soit le niveau d'analyse.

4.2.4.Diagramme de séquence

Le diagramme de séquence de PAC est présenté ci-dessous. L’article originel ne mentionne pas comment le Controller traduit la modification de la Présentation en opération sur l’Abstraction. Le Contrôleur de PAC semble être « omniscient » pour ce qui est du contrôle (*« explicitly centralized [into C] »*) et spécifique, il sait donc comment traduire des interactions aux opérations. Un deuxième diagramme de séquence disponible en annexe illustre la synchronisation de deux vues, qui ne se fait pas par la médiation du Modèle comme MVC, mais directement par le Contrôleur.



4.2.5.Code

Le Modèle de MVC et l’Abstraction de PAC sont identiques.

MVC

```
_define_
RectModel(double _x, double _y, double _w, double _h)
{
  Double x (_x)
  Double y (_y)
  Double width (_w)
  Double height (_h)

  Spike changed
  x, y, width, height -> changed
}
```

MVC text

```
TextView(Process model_, int _ty) {
  // a textual view of a rectangle with 4 properties as 4 texts (for x, y, w and h) on
      the window
  model aka model_

  Translation pos (0, _ty)

  // the 4 text graphical objects...
  Text t_x (0, 15, "0")
  Text t_y (15, 15, "0")
  Text t_width (30, 15, "0")
  Text t_height (45, 15, "0")

  // ... laid out horizontally
  t_x.x     + t_x.width     + 5 =:> t_y.x
  t_y.x     + t_y.width     + 5 =:> t_width.x
  t_width.x + t_width.width + 5 =:> t_height.x

  // the TextView 'api' changes the textual content of the 4 graphical texts
  x aka t_x.text
  y aka t_y.text
  width aka t_width.text
  height aka t_height.text

  // update the view whenever the model changes (subject/observer pattern)
  model.changed -> {
    model.{x,y,width,height} =: this.{x,y,width,height}
  }
}
```

PAC

```
_define_
RectAbstraction(double _x, double _y, double _w, double _h)
{
  Double x (_x)
  Double y (_y)
  Double width (_w)
  Double height (_h)

  Spike changed
  x, y, width, height -> changed
}
```

PAC text

```
Text t_x (0, 15, "0")
Text t_y (15, 15, "0")
Text t_width (30, 15, "0")
Text t_height (45, 15, "0")

// ... laid out horizontally
t_x.x     + t_x.width     + 5 =:> t_y.x
t_y.x     + t_y.width     + 5 =:> t_width.x
t_width.x + t_width.width + 5 =:> t_height.x

// the TextPresentation 'api' changes the textual content of the 4 graphical texts
x aka t_x.text
y aka t_y.text
width aka t_width.text
height aka t_height.text

Spike changed
// x, y, width, height -> changed
// ^^ commented out:
// let GraphicsPresentation activate changed on interaction, not on value change.
// Besides this avoid a cycle

// update abstraction from interactions on the view
      t_x.wheel.dy +=> x
      t_y.wheel.dy +=> y
  t_width.wheel.dy +=> width
 t_height.wheel.dy +=> height

// notify
      t_x.wheel.dy,
      t_y.wheel.dy,
  t_width.wheel.dy,
 t_height.wheel.dy -> changed
```

```
_define_
GraphicsView(Process model_) {
  model aka model_
  FillColor fill (Red)
  Rectangle r (0,0,0,0)

  // update the view whenever the model changes (subject/observer pattern)
  model.changed -> {
    model.{x,y,width,height} =: r.{x,y,width,height}
  }
}
```

MVC graph

```
_define_
GraphicsPresentation(Process abstraction, Process frame) {

  // graphics
  FillColor fill (Red)
  Rectangle r (0,0,0,0)

  Spike changed
  // r.x, r.y, r.width, r.height -> changed
  // ^^ commented out:
  // let GraphicsPresentation activate changed on interaction, not on value change.
  // Besides this avoid a cycle


  // control: update presentation from interactions

  //press aka frame.press
  press aka r.press
  move aka frame.move
  //release aka frame.release
  px aka frame.move.x
  py aka frame.move.y

  // delta helpers
  Int lastx(0)
```

PAC graph

Le Contrôleur de PAC ne gère plus l'interaction par manipulation directe, mais gère la synchronisation après un changement.

```
_define_
GraphicsController(Process model, Process _view, Process frame)
{
  Spike about_to_delete
  view aka _view


  // -- update model from interactions on the view

  // abstract events away
  press aka view.r.press
  move aka frame.move
  release aka frame.release
  px aka frame.move.x
  py aka frame.move.y

  // move delta helpers // FIXME should be in move?
  Double lastx(0)
  Double dx(0)
  px -> {
    px - lastx =: dx
            px =: lastx
  }
  Double lasty(0)
  Double dy(0)
  py -> {
    py - lasty =: dy
```

MVC text

```
_define_
GraphicsController(Process abstraction, Process presentation_, Process frame)
{
  presentation aka presentation_

  // the controller is in charge of updates
  abstraction.changed -> {
    abstraction.{x,y,width,height} =: presentation.r.{x,y,width,height}
  }

  presentation.changed -> {
    presentation.r.{x,y,width,height} =: abstraction.{x,y,width,height}
  }

  Spike about_to_delete
  about_to_delete->(this) {
    delete this.presentation
    delete this
  }
}
```

PAC text

```
_define_
TextController(Process model, Process view_)
{
  view aka view_

  // update model from interactions on the view
        view.t_x.wheel.dy +=> model.x
        view.t_y.wheel.dy +=> model.y
    view.t_width.wheel.dy +=> model.width
  view.t_height.wheel.dy +=> model.height

  Spike about_to_delete
  about_to_delete->(this) {
    delete this.control
    delete this.view
    delete this
  }
}
```

MVC graph

```
_define_
TextController(Process abstraction, Process presentation_)
{
  presentation aka presentation_

  // the controller is in charge of updates
  abstraction.changed -> {
    abstraction.{x,y,width,height} =: presentation.{x,y,width,height}
  }

  presentation.changed -> {
    presentation.{x,y,width,height} =: abstraction.{x,y,width,height}
  }

  Spike about_to_delete
  about_to_delete->(this) {
    delete this.presentation
    delete this
  }
}
```

PAC graph

Le Modèle d'Application de VisualWorks [1], ou l'Abstraction de Présentation de Fowler [10], ou le Modèle de Vue de MVVM [12] sont similaires et ont toutes le même but : abstraire la Vue, et servir de médiateur entre le Modèle et la Vue, afin de découpler encore davantage ces derniers. Dans le cas de MVVM, il s'agit notamment pour le Modèle de Vue de servir de composant pivot entre designer graphique et programmeur [12, 5]. Les champs du Modèle de Vue servent de paramètres pour le graphisme (designer), et de point d'entrée pour la description de l'interaction et de la modification du graphisme (programmeur). Ainsi, les ViewModel sont des instances « Presentation Objects » du modèle Arch/Slinky [23].

Par ailleurs :

MVVM is a refinement of MVC that evolves it from its Smalltalk origins where the entire application was built using one environment and language, into the very familiar modern environment of Web and now Avalon development.

MVVM also relies on one more thing: a general mechanism for data binding.

4.3.2.Exigences et approches

Des intentions exprimées, nous extrayons les exigences (Ex) et approches (App) suivantes :

$Ex1_{MVVM}$: Permettre à plusieurs parties prenantes de participer à la réalisation d'un logiciel interactif

$App1_{MVVM}$: Abstraire la Vue avec un composant ViewModel pour servir de composant pivot en parties prenantes et langages/environnement d'exécution

$App2_{MVVM}$: S'appuyer sur le « data-binding » uni- et bi- directionnel.

Ici les exigences sont orientées vers le support au génie logiciel collectif.

4.3.3.Référence originale

4.4.2.1 AM/PA

Presentation Model pulls the state and behavior of the view out into a model class that is part of the presentation. The Presentation Model coordinates with the domain layer and provides an interface to the view that minimizes decision making in the view. The view either stores all its state in the Presentation Model or synchronizes its state with Presentation Model frequently.

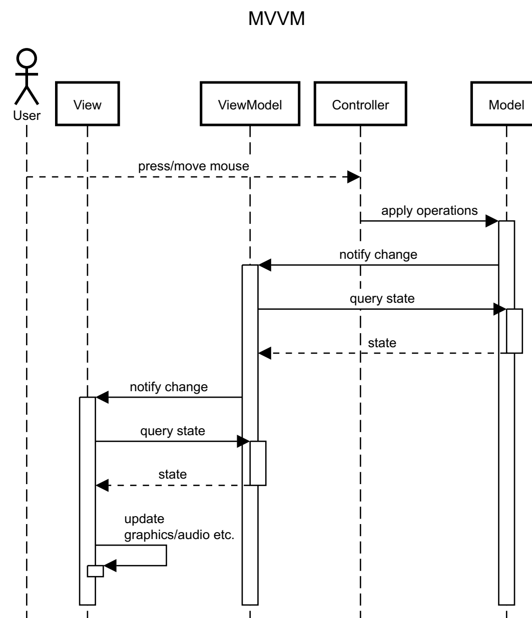
Presentation Model is not a GUI friendly facade to a specific domain object. Instead, it is easier to consider Presentation Model as an abstract of the view that is not dependent on a specific GUI framework. While several views can utilize the same Presentation Model, each view should require only one Presentation Model.

The essence of a Presentation Model is of a fully self-contained class that represents all the data and behavior of the UI window, but without any of the controls used to render that UI on the screen. A view then simply projects the state of the presentation model onto the glass.

4.4.2.2 MVVM

The term means "Model of a View" and can be thought of as abstraction of the view, but it also provides a specialization of the Model that the View can use for data-binding. In this latter role the ViewModel contains data-transformers that convert Model types into View types, and it contains Commands the View can use to interact with the Model.

4.3.4.Diagramme de séquence



4.3.5.Code

Le code du Modèle de Vue est très simple et est partagé entre la vue Textuelle et la vue Graphique.

```
_define_
ViewModel(Process view) {
  Double x(0)
  Double y(0)
  Double width(0)
  Double height(0)
}
```

Ainsi, dans la Vue Texte, il n’est plus nécessaire de proposer une API pour changer le contenu des textes et s’abstraire de leur nom ou de leur position hiérarchique dans un arbre de composant : c’est le Modèle de Vue qui s’en charge.

MVC text

```
_define_
TextView(Process model_, int _ty) {
  // a textual view of a rectangle with 4 properties as 4 texts (for x, y, w
      and h) on the window
  model aka model_

  Translation pos (0, _ty)

  // the 4 text graphical objects...
  Text t_x (0, 15, "0")
  Text t_y (15, 15, "0")
  Text t_width (30, 15, "0")
  Text t_height (45, 15, "0")

  // ... laid out horizontally
  t_x.x     + t_x.width     + 5 =:> t_y.x
  t_y.x     + t_y.width     + 5 =:> t_width.x
  t_width.x + t_width.width + 5 =:> t_height.x

  // the TextView 'api' changes the textual content of the 4 graphical texts
       x aka t_x.text
       y aka t_y.text
   width aka t_width.text
  height aka t_height.text

  // update the view whenever the model changes (subject/observer pattern)
  model.changed -> {
    model.{x,y,width,height} =: this.{x,y,width,height}
  }
}
```

MVVM text

```
_define_
TextView(int _ty) {
  // a textual view of a rectangle with 4 properties as 4 texts (for x, y, w
      and h) on the window

  Translation pos (0, _ty)

  // the 4 text graphical objects...
  Text t_x (0, 15, "0")
  Text t_y (15, 15, "0")
  Text t_width (30, 15, "0")
  Text t_height (45, 15, "0")

  // ... laid out horizontally
  t_x.x     + t_x.width     + 5 =:> t_y.x
  t_y.x     + t_y.width     + 5 =:> t_width.x
  t_width.x + t_width.width + 5 =:> t_height.x
}
```

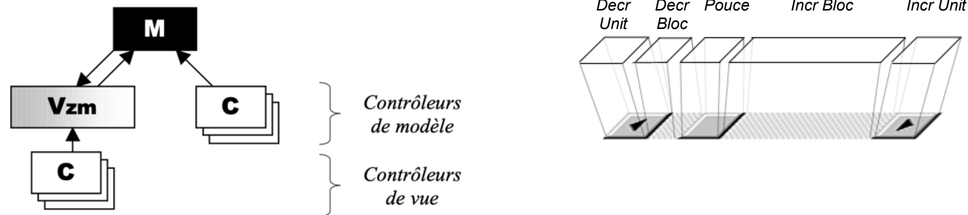
Le Contrôleur est chargé de mettre en place un lien bidirectionnel entre d’une part la Vue et le Modèle de Vue, et d’autre part le Modèle de Vue et le Modèle. Plutôt que d’utiliser un schéma de type ‘notification’, le code suivant s’appuie sur l’opérateur de flot de donnée ‘= :>’ fournit par le langage Smala, qui élimine le besoin de notification. Cet abonnement à une variable est aussi appelé Variable Active, ou ValueModel dans VisualWorks. Qt implémente ce schéma sous forme de signal/slot, avec une fonction anonyme pour particulariser le traitement (patron SASE (Self-Adressed, Stamped Enveloppe) [1]).
Le Contrôleur de texte établit les relations bidirectionnelles entre Modèle et Modèle de Vue, ainsi qu’entre Modèle de Vue et Vue.

```
_define_
TextController(Process model, Process _view, Process _view_model)
{
  view aka _view
  view_model aka _view_model

  // update the view_model whenever the model changes (subject/observer pattern)
       model.{x,y,width,height} =:> view_model.{x,y,width,height}

  // since the view_model acts as a proxy, we must update the model whenever view_model changes
  view_model.{x,y,width,height} =:> model.{x,y,width,height}

  // setup a bidirectional connector between the view and the the view_model
  // in this example, it's not strictly necessary, since the view is modified only through interaction
  // it could be useful if it was possible to edit the text with the keyboard
  view_model.{x,y,width,height} =:> view.{t_x.text,t_y.text,t_width.text,t_height.text}
  view.{t_x.text,t_y.text,t_width.text,t_height.text} =:> view_model.{x,y,width,height}

  // update the view_model from interactions on the view
       view.t_x.wheel.dy +=> view_model.x
       view.t_y.wheel.dy +=> view_model.y
   view.t_width.wheel.dy +=> view_model.width
  view.t_height.wheel.dy +=> view_model.height
[..]
```

4.4.MVzmC (Model – View/zone/manipulateur - Contrôleur)

4.4.1.Intentions

L’intention des auteurs de MVzmC est de « découpler les widgets de leur méthode d’entrée », notamment pour permettre une gestion des entrées multimodales plus modulaire, dynamique et extensible. MVzmC s’appuie notamment sur la distinction de deux types de contrôleurs : Contrôleur de Vue et Contrôleur de Modèle, et sur l’identification de Zones à l’intérieur des Vues. Les Vues sont capables de rendre un service de détermination des zones dans laquelle l’utilisateur agit.



4.4.2.Exigences et approches

Des intentions exprimées, nous extrayons les exigences (Ex) et approches (App) suivantes :

Ex1$_{MVzmC}$ Rendre plus modulaire, dynamique et extensible les composants logiciels

Ex2$_{MVzmC}$ Découpler les widgets de leur méthode d’entrée

App1$_{MVzmC}$ Distinguer les Contrôleurs de Vue des Contrôleurs de Modèle

App2$_{MVzmC}$ Réifier les « sous-Vues » dans des composants Zones

App3$_{MVzmC}$ Réifier les interactions de bas-niveau dans des composants Vzm Vue-zones-manipulateur.

4.4.3.Référence originale

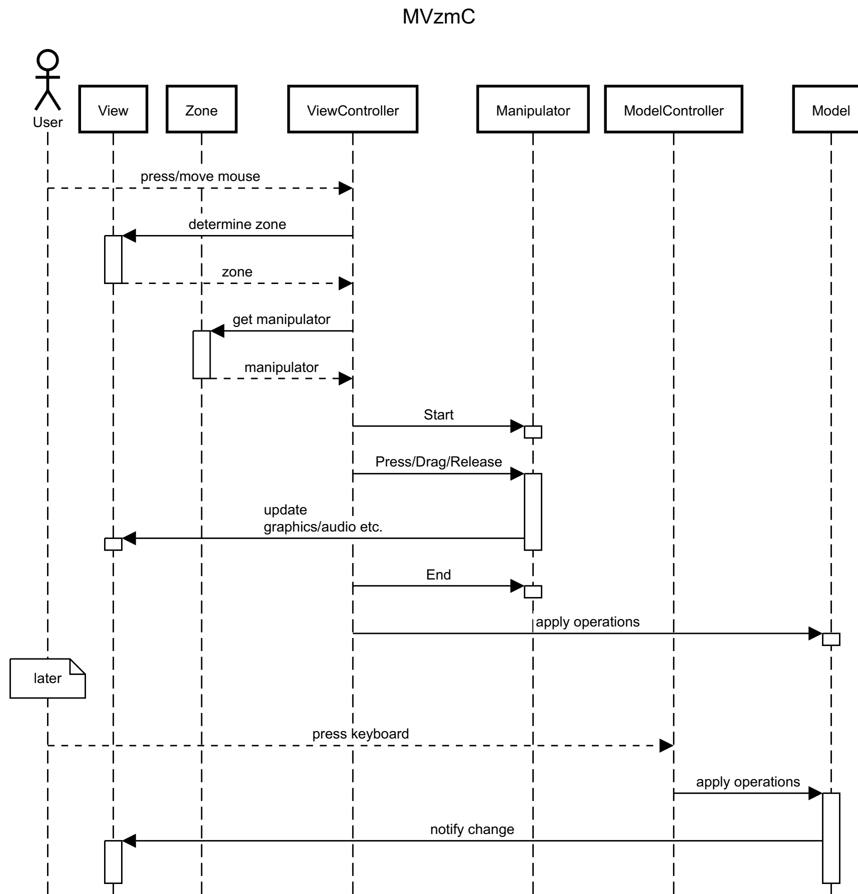
L'objet "Modèle". Cet objet est le même que celui décrit dans MVC. Il représente le type de données que le widget sert à manipuler.

Le Contrôleur de Modèle. Cet objet correspond à l'objet Contrôleur de MVC, qui traduit certains événements utilisateur en changements dans le modèle. Il est appelé ainsi par opposition au Contrôleur de Vue. L'objet Modèle peut gérer des Contrôleurs de Modèle multiples.

Le Contrôleur de Vue. Le Contrôleur de Vue traduit certains événements utilisateur (en général, des événements positionnels) en manipulations de la vue. Il communique avec la vue (ici, l'objet Vzm) à travers un protocole spécial que nous décrirons par la suite. Ce protocole autorise aussi les contrôleurs multiples.

L'objet "Vue/Zones/Manipulateurs". Il s'agit d'un objet Vue classique, mais qui implémente une [API] (Start/Press/Drag/Release/End) lui permettant d'être manipulé par des contrôleurs. Cette interface exhibe la vue comme étant un objet graphique contenant un ensemble de zones qui réagissent à des manipulations de base. L'objet Vzm joue aussi le rôle de Contrôleur pour le modèle, car il traduit les manipulations de la vue en manipulations du modèle.

4.4.4.Diagramme de séquence



4.4.5.Code

Nous n'avons pas implémenté l'exemple avec MVzmC, ce code fera l'objet de travaux futurs.

4.5.MDPC (Model – Display View – Picking View – (inverse) Transforms)

4.5.1.Intentions

L'intention des concepteurs de MDPC était de pouvoir disposer d'un ensemble de widgets entièrement basées sur des modèles [7]. Il fallait donc que le contrôle (exprimé sous forme de réseau de Petri) soit complètement séparé de la représentation (exprimé sous forme de SVG et de feuille de style), car chacun des aspects ne pouvait être codé dans un seul langage et environnement d'exécution. Le travail sur MDPC a permis d'identifier de nouvelles responsabilités, et d'en accentuer la séparation.

4.5.2.Exigences et approches

Des intentions exprimées, nous extrayons les exigences (Ex) et approches (App) suivantes :

Ex1$_{\text{MDPC}}$ Se rapprocher d'une description basée sur les modèles, écrits dans des langages différents et exécutés dans des environnements séparés

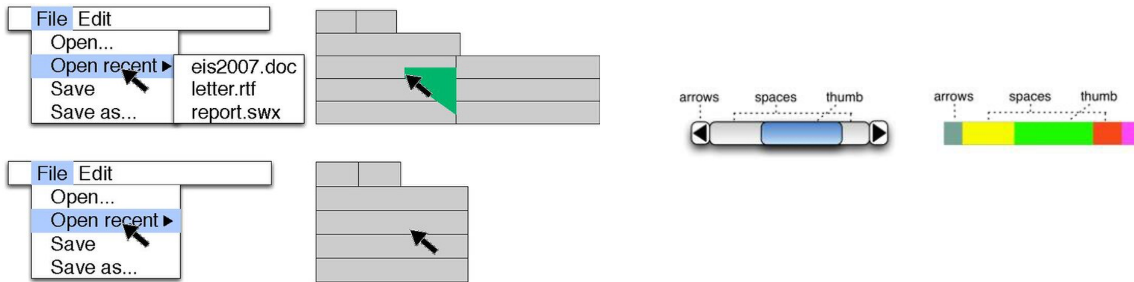
App1$_{\text{MDPC}}$ Découpler complètement le Contrôleur de la Vue

App2$_{\text{MDPC}}$ Réifier les zones d'interaction dans un composant PickingView qui va servir de base pour l'interaction

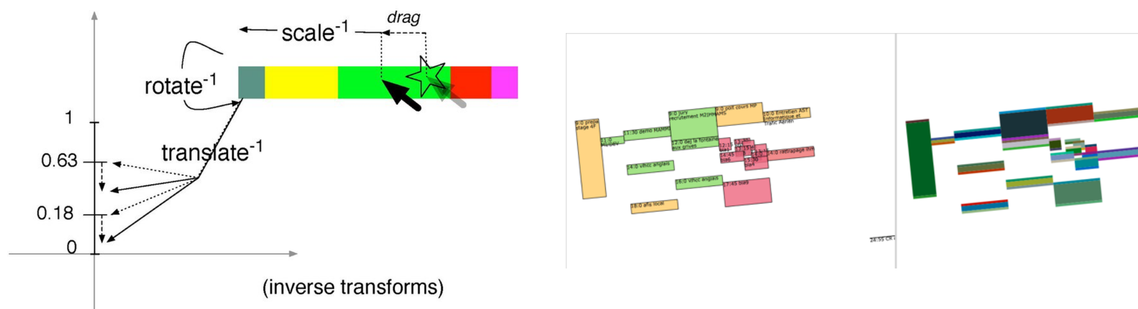
App3$_{\text{MDPC}}$ Réifier les transformations de Modèle en Vue dans un composant idoine, et utiliser les transformations inverses pour retraduire les interactions de l'utilisateur en opérations sur le Modèle

4.5.3.Référence originale

Picking views. Picking views are invisible graphical objects overlaid on visible ones, but that still react to user events. The first figure shows the "display view" of a hierarchical menu (top left) and the corresponding picking view when the user is navigating in the menu (top right). The (transient) triangle laid over the menu in the picking view enables reaching the sub-menu entries while avoiding submenu folding. Similarly, the picking view of the scrollbar displays as many shapes as spatial modes (thumb, arrows and spaces between thumb and arrows).

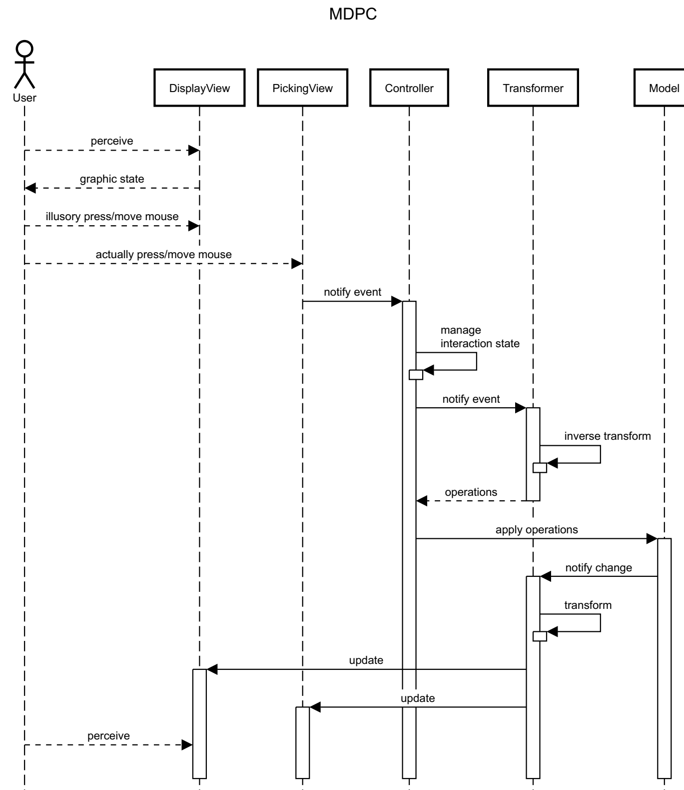


Graphical transformations and inverse transformations [6]. Graphical transformations are functions that transform the conceptual model into graphics. MDPC relies on two transformations: one for the display view and one for the picking view. The figure below shows the affine transforms applied to the model of a horizontal scrollbar (two values between 0 and 1) to generate the display view and the picking view. By computing the inverse transformations, graphical interactions (dragging the thumb) can be translated back into operations on the model (updating values).



4.5.4.Diagramme de séquence

Par rapport aux autres diagrammes de séquence, nous avons ajouté des éléments qui mettent en valeur le fait que l'utilisateur perçoit la vue, et récupère en retour un état graphique. C'est bien sur la vue que l'utilisateur clique, mais ce n'est pas le composant « Vue Affichée (Display View) » qui reçoit cet événement, mais bien une vue pour l'interaction (Picking View). Ces graphismes modifiés sont ensuite perçus par l'utilisateur, et la boucle de perception/action peut recommencer.



4.5.5.Code

Le code de la Vue Display est le même que celui de la vue de MVC. MDPC introduit une nouvelle vue, la vue Picking :

```
_define_
GraphicsPickingView() {
  NoFill _
  NoOutline _
  PickFill _

  Rectangle r (0,0,0,0) // center
  Rectangle left (0,0,0,0)
  Rectangle right (0,0,0,0)
  Rectangle top (0,0,0,0)
  Rectangle bottom (0,0,0,0)

  x aka r.x
```

```
  y aka r.y
  width aka r.width
  height aka r.height

  Double border (5)
  Double invtborder (0)

                 invtborder =:> top.height, bottom.height, left.width,  right.width

                        r.x =:> top.x,      bottom.x,      left.x
                        r.y =:> top.y,                     left.y,      right.y
                    r.width =:> top.width,  bottom.width
                   r.height =:>                            left.height, right.height

  r.x + r.width  - invtborder =:>                                       right.x
  r.y + r.height - invtborder =:>            bottom.y

}
```

Le code du contrôleur graphique est plus simple que celui de MVC (Annexe A.3) car il n'a pas besoin de déterminer les sous-vues sur lesquelles l'utilisateur agit : ce sont les vues picking qui se chargent de recevoir les événements et les transmettre.

```
_define_
GraphicsController(Process model, Process display_view_, Process picking_view_, Process frame)
{
  Spike about_to_delete
  view aka display_view_
  picking_view aka picking_view_

  // -- update the views whenever the model changes (subject/observer pattern)
  // and transform the model into a display and picking views
  model.{x,y,width,height} =:> view.{x,y,width,height}
  model.{x,y,width,height} =:> picking_view.{x,y,width,height}

  // -- transform user actions on the view into operations on the model

  // delta helpers
  px aka frame.move.x
  py aka frame.move.y
  Double lastx(0)
  Double sdx(0) // screen dx
  px -> {
    px - lastx =: sdx
            px =: lastx
  }
  Double lasty(0)
  Double sdy(0) // screen dy
  py -> {
    py - lasty =: sdy
            py =: lasty
  }

  // inverse transform user actions, from screen to model coordinates
  Double mdx(0) // model dx
  Double mdy(0) // model dy

  ScreenToLocal m (view.r)
  px =:> m.inX
  py =:> m.inY
```

```
ScreenToLocal m2 (view.r)
px-sdx =:> m2.inX
py-sdy =:> m2.inY

m.outX-m2.outX =:> mdx
m.outY-m2.outY =:> mdy

dx aka mdx
dy aka mdy

// actual interactions and model updates
move aka frame.move
center_press aka picking_view.r.press
left_press aka picking_view.left.press
right_press aka picking_view.right.press
top_press aka picking_view.top.press
bottom_press aka picking_view.bottom.press

// the FSM manages the interaction state by activating/deactivating dataflows
// interactions on picking views manage the FSM transitions
FSM drag {
  State idle
  State dragging_center {
      dx +=> model.x
      dy +=> model.y
  }
  State dragging_left {
      dx +=> model.x
    - dx +=> model.width
  }
  State dragging_right {
      dx +=> model.width
  }
  State dragging_top {
      dy +=> model.y
    - dy +=> model.height
  }
  State dragging_bottom {
      dy +=> model.height
  }
  idle -> dragging_center (center_press)
  idle -> dragging_left (left_press)
  idle -> dragging_right (right_press)
  idle -> dragging_top (top_press)
  idle -> dragging_bottom (bottom_press)
  { dragging_center, dragging_left, dragging_right, dragging_top, dragging_bottom } -> idle (frame.release)
}
```

5. ANALYSE DES PREOCCUPATIONS ET DES COMPOSANTS

Dans cette section, nous proposons une exégèse de ces travaux, c'est-à-dire « d'explication grammaticale et étymologique d'un texte ancien difficile, assortie d'un commentaire savant et d'une interprétation ». Nous discutions notamment de la version originale de MVC, de son évolution et du lien avec les patrons déjà analysés.

5.1.MVC Original (Model – View – Controller *+ Editor*)

L'auteur de MVC a lui-même commenté plus tard ses intentions originales (cf Annexe A.4)

I made the first implementation and wrote the original MVC note at Xerox PARC in 1978. The note defines four terms: Model, View, Controller and Editor. The Editor is an ephemeral component that the View creates on demand as an interface between the View and the input devices such as mouse and keyboard.

Il comment ensuite et ce qu'il est advenu de ses travaux :

Jim Althoff and others implemented a version of MVC for the Smalltalk-80 class library after I had left Xerox PARC; I was not involved in this work. Jim Althoff uses the term Controller somewhat differently from me.

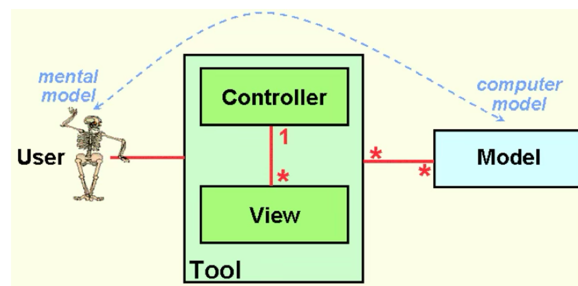
Puis il précise les rôles des différents composants :

An important aspect of the original MVC was that its Controller was responsible for creating and coordinating its subordinate views. Also, in my later MVC implementations, a view accepts and handles user input relevant to itself. The Controller accepts and handles input relevant to the Controller/View assembly as a whole, now called the Tool.

The Controller was here what I now call a Tool. The Smalltalk-80 Controller is here a fourth element called Editor.

A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen.

Some views provide a special controller, an editor, that permits the user to modify the information that is presented by the view. Such editors may be spliced into the path between the controller and its view, and will act as an extension of the controller. Once the editing process is completed, the editor is removed from the path and discarded."



Cet élément historique montre comment une architecture est inventée, détournée, et raffinée. Elle montre aussi que l'article qui est souvent cité (celui de Reenskaug) ne contient sans doute pas la description que nous connaissons

de MVC, qui est plutôt celle de MVC ST-80. La création éphémère de l'Editor de MVC original n'est pas sans rappeler celle du composant Vzm de MVzmC. Et le plus récent composant Tool qui rassemble Controller et View n'est pas sans rappeler la Présentation de PAC.

5.2. Séparer le Contrôleur de la Vue

Plusieurs articles mentionnent la difficulté apparente de séparer complètement la Vue et le Contrôle à bas niveau d'interaction. Les auteurs de Swing eux-mêmes, qui se sont basés sur MVC, le soulignent :

"The first Swing prototype followed a traditional MVC separation in which each component had a separate model object and delegated its look-and-feel implementation to separate view and controller objects. [...] We quickly discovered that this split didn't work well in practical terms because the view and controller parts of a component required a tight coupling (for example, it was very difficult to write a generic controller that didn't know specifics about the view). So we collapsed these two entities into a single UI (user-interface) object."
Java Swing Designers, http://java. sun.com/products/jfc/tsc/articles/architecture/, retrieved 2011-09-22.

Figure 4: Essential dependencies between model, view, and controller. (I call this essential because in fact the view and controller do link to each other directly, but developers mostly don't use this fact.)
- Martin Fowler [2] https://www.martinfowler.com/eaaDev/uiArchs.html

Le code du contrôleur de Scrollbar de ST-80 l'illustre :

```
scroll
    "Check to see whether the user wishes to jump, scroll up, or scroll down."

    | savedCursor regionPercent |
    savedCursor <- sensor currentCursor.
    [self scrollBarContainsCursor]
        whileTrue:
            [Processor yield.
            regionPercent <- 100 * (sensor cursorPoint x - scrollBar left) // scrollBar width.
            regionPercent <= 40
                ifTrue:  [self scrollDown]
                ifFalse: [regionPercent >= 60
                            ifTrue:  [self scrollUp]
                            ifFalse: [self scrollAbsolute]]].
    savedCursor show!
```

On voit comment le code teste la position du curseur par rapport à l'intérieur de la scollbar pour déterminer si l'interaction a lieu au-dessus, dans ou au-dessous du pouce.

C'est la raison pour laquelle PAC n'essaie pas de séparer Contrôle et Vue et l'assemble dans un seul composant Présentation. De même, MVzmC assemble dans un même composant Vzm la vue et sa manipulation par l'utilisateur.

Cependant MVzmC et MDPC identifient la gestion des zones comme étant une préoccupation du Contrôleur qui rend difficile la séparation Contrôleur-Vue. Avec les zones, ou les Display View, il est possible d'extraire cette préoccupation du contrôleur, et de le rendre plus générique/réutilisable.

En réifiant les transformations et les transformations inverses dans un composant Transformation, MDPC rend possible la transformation des interactions en opérations sur le modèle, et sépare encore davantage le contrôle de

la vue en éliminant le besoin d'effectuer des requêtes auprès de la Vue sur l'agencement des zones. Cette idée semble être présente dès l'origine des interactions graphiques, en témoigne une communication personnelle impliquant Alan Kay, leader sur Smalltalk (cf Annexe A.6) :

The good part [of MVC] was philosophical -- the idea to adapt the notion of "cameras" and "worlds" in the original 3D graphics stuff I participated in at Utah 45 years ago. […] We have never done a really satisfactory automatic inverter for dealing with the loss of "dimensions" that happen when a view is made (but we have done some experimental ones).

5.3.Séparation des Préoccupations

Rétrospectivement, nous pouvons analyser MVC, PAC, MVVM, MVzmC et MDPC comme une distribution, à un ensemble de composants, des préoccupations suivantes : la gestion de l'*état des données*, l'*affichage de la vue* et leur *synchronisation*, le *picking*, la *traduction des actions* de l'utilisateur en *opérations sur l'état* des données et la gestion de l'*état de l'interaction*.

Le composant Modèle de MVC, MVVM, MVzmC et MPDC, et le composant Abstraction de PAC traitent tous de l'état des données.

PAC n'a pas essayé de séparer l'affichage, le picking, la traduction et l'état interactif, puisqu'ils semblent si liés. Un seul composant de Présentation est chargé de ces préoccupations.

De façon similaire, le composant Vzm ne sépare pas ces préoccupations, mais le gère de façon plus modulaire au sein de la Vue grâce au composant zone, résultat d'une opération de *picking* par la Vue.

Le composant View de MVC s'occupe de l'affichage, laissant au contrôleur la responsabilité du picking, de la traduction et de l'état interactif. Il est donc difficile de découpler le contrôleur et la vue : souvent, le code qui en résulte mélange V et C (V+C) et ne remplit pas l'objectif de modularité. La composante V+C est presque l'addition des composantes P et C de PAC, ce qui signifie que les développeurs qui pensent architecturer avec MVC utilisent en fait une décomposition PAC.

La Picking View présente deux avantages. Premièrement, elles aident à gérer la dynamique des états de l'interaction (par exemple le triangle transitoire du menu hiérarchique), par opposition à l'état graphique de l'affichage. Deuxièmement, elles permettent d'éviter le calcul analytique des relations spatiales (par exemple, le mouvement avec une direction inférieure à 45°, ou la position d'un clic par rapport au pouce d'une barre de défilement) en utilisant les événements Enter/Leave générés par la boîte à outils graphique sous-jacente. Les vues Picking réifient les modes spatiaux d'interaction. Un mode spatial est l'équivalent spatial d'un mode temporel : un comportement différent en fonction de l'espace, contre un comportement différent en fonction du temps.

5.4.What's a controller anyway ? Qu'est-ce qu'un contrôleur finalement ?

Il existe un débat sur ce qu'est un Contrôleur. Le terme est utilisé pour définir un composant qui s'occupe de plusieurs préoccupations : l'aspect hiérarchique et de composition conceptuelle des interactions (haut-niveau), le contrôle de l'interaction au plus proche de la vue (C-Editor de MVC, C de PAC, Manipulateur de MVzmC, les Cm et Cv de MVzmC).

On peut appliquer le principe de séparation des préoccupations jusqu'au contrôleur. Si l'on délègue les préoccupations souvent implicites de gestion de l'état graphique conceptuel de l'interaction vers les Vues Pickings, et la transformation inverse des interactions vers un composant dédié, il ne reste plus que le contrôle de l'état de l'interaction. Dans MDPC, cette gestion consiste en une machine à états, qui gère les transitions d'après les

événements utilisateurs, et les changements de la vue Display et Picking, c'est-à-dire de l'état graphique pour l'interaction (vision+action).

5.5.Utilisabilité de la Conception et de la Programmation

Les patrons de d'architecture sont souvent présentés comme permettant la modularité, la réutilisabilité et l'extensibilité. Cependant, si l'on considère l'ingénieur logiciel comme un utilisateur, et ces patrons comme des outils pour la conception et la programmation de système interactif, il reste à démontrer que cette modularité, réutilisabilité et extensibilité sont effectivement plus utilisable dans le sens de l'ISO : efficience, efficacité, satisfaction. Par exemple, certains patterns (comme Factory) ont été montrés comme moins utilisables par des développeurs que les constructeurs simples dans certaines circonstances.

Pour ce qui est de l'efficience, les auteurs de MVzmC ont réussi à réarchitecturer les composants de Swing, tout en apportant les avantages que procurent MVzmC, montrant qu'il est possible de décrire des interactions avec ce patron. L'un des articles sur MDPC prétend que MDPC peut rendre plus utilisable la spécification des interactions [7]. Pourtant, ne serait-ce que deviser l'inverse d'une fonction de transformation n'a rien d'aisé.

Enfin, le principe de séparation des préoccupations entraine implicitement un principe de séparation du code correspondant, au détriment de la localité, ce qui peut rendre difficile la compréhension des interactions entre composants logiciels. C'est ce que suggère Alan Kay [Annexe 6]

Things seem to hang on in computing just because they work a little bit.
[…] The bad part of MVC was how we implemented it -- much too much machinery, etc.

6. CONCLUSION

Nous avons proposé une analyse de plusieurs patrons d'architecture sous forme d'intentions, d'exigences, de diagrammes de séquence et de code. Nous proposons de plus une exégèse sur les relations entre les composants des différents patrons, et sur leur préoccupation.

Cette analyse est loin d'être exhaustive, sur plusieurs plans : patrons manquants (p. ex. MVP [18], Garnet [15], Unidraw [21], ou les plus récentes du web), considérations manquantes (le cycle de vie, l'aspect hiérarchique, le bubbling, d'où viennent les événements depuis l'origine de leur émission, etc.), ou style d'interaction plus évolué (p. ex. l'Interaction Instrumentale [2], et un des modèles associés comme DoPIdom [3] ou Malai [4]).

On pourrait se poser la question du « patron qui les unifie tous ». Outre l'acronyme à rallonge qu'un tel patron engendrerait, il n'est pas sûr que cela soit souhaitable. Comme le préconise Slinky, c'est l'application particulière visée qui doit guider la conception et l'utilisation ou non d'un raffinement de patron. Pour terminer par une citation, Reenskaug lui-même a réalisé que ce travail n'était pas terminé :

The MVC problem has more facets than I realized in 1979. I started working on a pattern language to disentangle the different aspects, that last draft was dated August 20, 2003. The plan was that it should be improved by a group of authors, not just the current single one. Unfortunately, the project died at this point.

CONTRIBUTIONS DES AUTEURS

Stéphane Conversy a écrit l'article et est le seul responsable de son contenu. Il a participé à la programmation des exemples. Mathieu Magnaudet, Vincent Peyruqueou et Mathieu Poirier ont participé à la programmation des exemples.

ACKNOWLEDGMENTS

Acknowledgments are placed before the references. Add information about grants, awards, or other types of funding that you have received to support your research. Author can capture the **grant sponsor information**, by selecting the grant sponsor text and apply style 'GrantSponsor'. After this, select grant no and apply 'GrantNumber' from style panel. Example of Grant sponsor: Competitive Research Programme and example of Grant no: CRP 10-2012-03. ANR.... N°

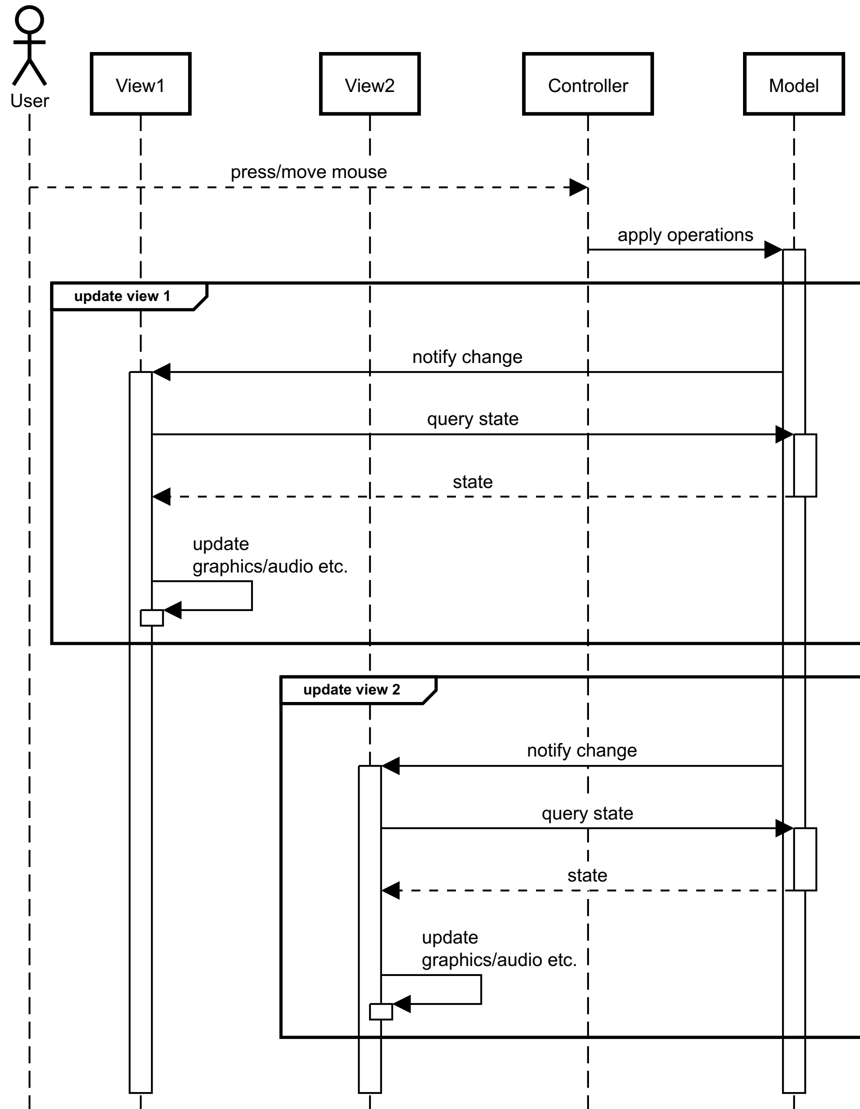
REFERENCES

1. Sherman R. Alpert, Kyle Brown, and Bobby Woolf. 1998. The design patterns Smalltalk companion. Addison-Wesley Longman Publishing Co., Inc., USA.
2. Michel Beaudouin-Lafon. 2000. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI '00). Association for Computing Machinery, New York, NY, USA, 446–453.
3. Olivier Beaudoux. 2006. DoPIdom: une approche de l'interaction et de la collaboration centrée sur les documents. In Proceedings of the 18th Conference on l'Interaction Homme-Machine (IHM '06). Association for Computing Machinery, New York, NY, USA, 19–26.
4. Arnaud Blouin and Olivier Beaudoux. 2009. Malai: un modèle conceptuel d'interaction pour les systèmes interactifs. In Proceedings of the 21st International Conference on Association Francophone d'Interaction Homme-Machine (IHM '09). Association for Computing Machinery, New York, NY, USA, 129–138.
5. Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, and Christophe Mertz. 2004. Revisiting visual interface programming: creating GUI tools for designers and programmers. In Proceedings of the 17th annual ACM symposium on User interface software and technology (UIST '04). Association for Computing Machinery, New York, NY, USA, 267–276.
6. S. Conversy. Improving usability of interactive graphics specification and implementation with picking views and inverse transformation. *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Pittsburgh, PA, USA, 2011, pp. 153-160
7. Stéphane Conversy, Eric Barboni, David Navarre, Philippe Palanque. Improving modularity of interactive software with the MDPC Architecture. *Joint Working Conference on Engineering Interactive Systems (EIS 2007)*, Mar 2007, Salamanca, Spain. pp.321-338,
8. Coutaz, J., 1987. PAC, an implementation model for dialog design. In: Interact'87, Stuttgart, September 1987, pp. 431-436.
9. Dijkstra. Edsger W. 1982. On the role of scientific thought. Selected writings on computing: a personal perspective. Springer-Verlag, Berlin, Heidelberg.
10. Fowler, Martin. Presentation Model. 2004. https://www.martinfowler.com/eaaDev/PresentationModel.html
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., USA.
12. John Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps. 2005.
13. Mathieu Magnaudet, Stéphane Chatty, Stéphane Conversy, Sébastien Leriche, Celia Picard, and Daniel Prun. 2018. Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming. Proc. ACM Hum.-Comput. Interact. 2, EICS, Article 12 (June 2018), 27 pages. https://doi.org/10.1145/3229094.
14. Bertrand Meyer. 1997. Object-oriented software construction (2nd ed.). Prentice-Hall, Inc., USA.
15. Brad A. Myers, Dario Giuse, Andrew Mickish, Brad Vander Zanden, David Kosbie, Richard McDaniel, James Landay, Matthew Golderg, and Rajan Pathasarathy. 1994. The garnet user interface development environment. In Conference Companion on Human Factors in Computing Systems (CHI '94). Association for Computing Machinery, New York, NY, USA, 457–458.
16. John Ousterhout. A Philosophy of Software Design, 2nd edition, 2021.
17. G. Pfaff and P.J.W. ten Hagen, editors. Seeheim Workshop on User Interface Management Systems, Berlin, 1985. Springer-Verlag.
18. Mike Potel. MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java. 1996.
19. Reenskaug, Trygve M. MODELS - VIEWS - CONTROLLERS. Xerox PARC technical note December 1979. https://folk.universitetetioslo.no/trygver/1979/mvc-2/1979-12-MVC.pdf
20. Reenskaug, .2003. The Model-View-Controller (MVC). Its Past and Present. Unfinished and unpublished.
21. John M. Vlissides and Mark A. Linton. 1990. Unidraw: a framework for building domain-specific graphical editors. ACM Trans. Inf. Syst. 8, 3 (July 1990), 237–268.

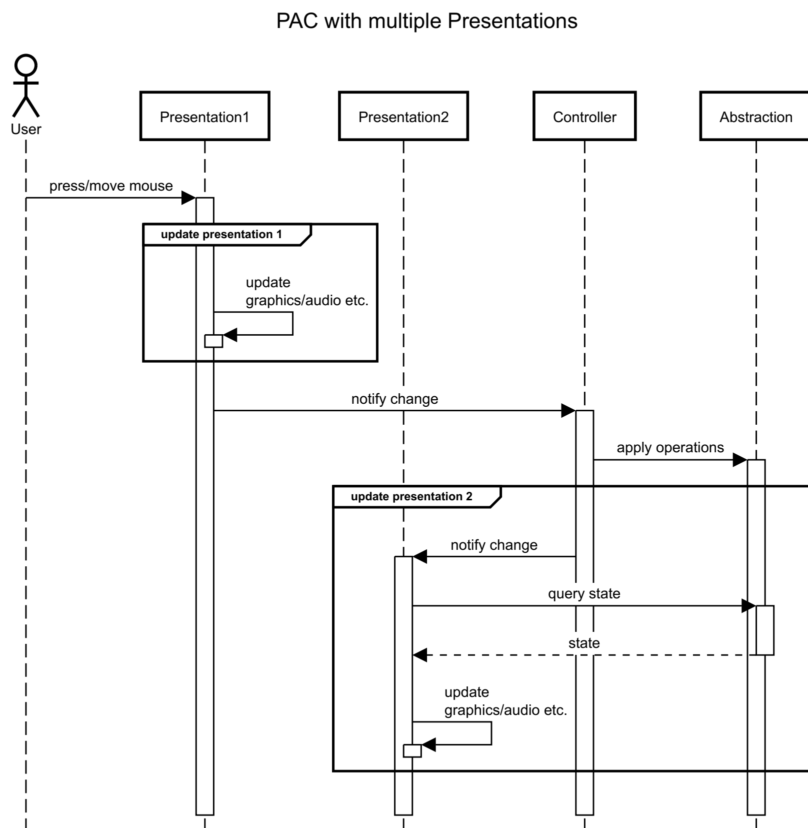
22. Edward Yourdon and Larry L. Constantine. 1979. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design (1st. ed.). Prentice-Hall, Inc., USA.
23. 1992. A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop. SIGCHI Bull. 24, 1 (Jan. 1992), 32–37. https://doi.org/10.1145/142394.142401.
24. ISO/IEC/IEEE 42010:2022 Software, systems and enterprise — Architecture description. 2022.

ANNEXES

A.1 MVC ST80 avec de multiples vues (ou observateurs)



A.2 PAC avec de multiples vues (ou observateurs)



A.3 Code MVC ST80 du contrôleur du rectangle graphique

```
use core
use base
use gui

_define_
GraphicsController(Process model, Process _view, Process frame)
{
  Spike about_to_delete
  view aka _view


  // -- update model from interactions on the view

  // abstract events away
  press aka view.r.press
  move aka frame.move
  release aka frame.release
  px aka frame.move.x
  py aka frame.move.y
```

```
// move delta helpers // FIXME should be in move?
Double lastx(0)
Double dx(0)
px -> {
  px - lastx =: dx
          px =: lastx
}
Double lasty(0)
Double dy(0)
py -> {
  py - lasty =: dy
          py =: lasty
}

// query layout of view internals: where on the rect did the user press? which border, or center?
Int border(5)

Bool center(0)
Bool left(0)
Bool right(0)
Bool top(0)
Bool bottom(0)

                  (abs(px - view.r.x) <= border) =:> left
  (abs(px - (view.r.x + view.r.width)) <= border) =:> right
                  (abs(py - view.r.y) <= border) =:> top
(abs(py - (view.r.y + view.r.height)) <= border) =:> bottom
          not (left || right || bottom || top) =:> center

// synthesize events upon press on borders or center
Spike center_press
Spike left_press
Spike right_press
Spike top_press
Spike bottom_press

FSM area_press {
  State idle
  State in_center { press -> center_press }
  State in_left   { press -> left_press }
  State in_right  { press -> right_press }
  State in_top    { press -> top_press }
  State in_bottom { press -> bottom_press }

  idle -> in_center  (center.true)
  in_center -> idle  (center.false)
  idle -> in_left    (left.true)
  in_left -> idle    (left.false)
  idle -> in_right   (right.true)
  in_right -> idle   (right.false)
  idle -> in_top     (top.true)
  in_top -> idle     (top.false)
  idle -> in_bottom  (bottom.true)
  in_bottom -> idle  (bottom.false)
}

// control interactions and update model
// 'control' means: 'control the state of the interaction' (the FSM) and 'activate data-flows' (inside states)
// 'update' is implemented with an addition connector '+=>'
FSM drag {
  State idle
```

```
    State dragging_center {
        dx +=> model.x
        dy +=> model.y
    }
    State dragging_left {
        dx +=> model.x
      - dx +=> model.width
    }
    State dragging_right {
        dx +=> model.width
    }
    State dragging_top {
        dy +=> model.y
      - dy +=> model.height
    }
    State dragging_bottom {
        dy +=> model.height
    }
    idle -> dragging_center (center_press)
    idle -> dragging_left   (left_press)
    idle -> dragging_right  (right_press)
    idle -> dragging_top    (top_press)
    idle -> dragging_bottom (bottom_press)
    { dragging_center, dragging_left, dragging_right, dragging_top, dragging_bottom } -> idle (release) // FIXME: why
{} ???
  }


  about_to_delete->(this) {
    delete this.control
    delete this.view
    delete this
  }
}
```

A.4 Histoire de MVC par Reenskaug

https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html

The following is the content of the above page, duplicated here for archival reasons. I did not get the permission from Reenskaug, as he died in 2024. Please cite the following as :

Reenskaug (October 29, 2018) MVC XEROX PARC 1978-79, https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html

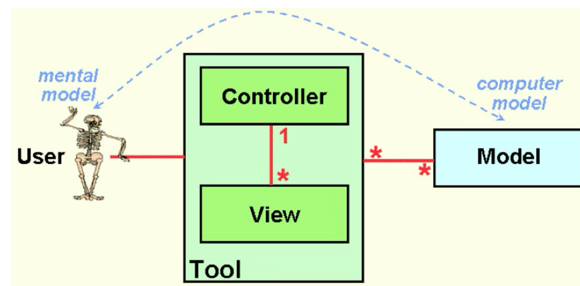
I spent a very happy and inspiring year as a visiting scientist with the Learning Research Gorup (LRG) at Xerox PARC from the summer og 1978 to the summer of 1979. This group was dedicated to Alan Kay's vision of the Dynabook; a portable computer that should contain all data of interest to its owner/user. Very importantly, these data included the programs the owner used to manipulate them. The owner/user should be able to understand and write the programs, thus gaining ascendancy over the computer.

The MVC notes should be read on this background. The user was the czar; everything done at LRG was done to support him. An earlier paper adds some background for MVC: (A note on DynaBook requirements 22 March 1979 (Partial scan, 11 pp).

I have sometimes been given more credit than is my due, so I should stress that I am not one of the original inventors of Smalltalk. I am only one of the very early and very enthusiastic users and contributors to this revolutionary innovation. I made the first implementation and wrote the original MVC note at Xerox PARC in 1978. The note defines four terms: Model, View, Controller and Editor. The Editor is an ephemeral component that the View creates on demand as an interface between the View and the input devices such as mouse and keyboard.

Jim Althoff and others implemented a version of MVC for the Smalltalk-80 class library after I had left Xerox PARC; I was in not involved in this work. Jim Althoff uses the term Controller somewhat differently from me. An important aspect of the original MVC was that its Controller was responsible for creating and coordinating its subordinate views. Also, in my later MVC implementations, a view accepts and handles user input relevant to itself. The Controller accepts and handles input relevant to the Controller/View assembly as a whole, now called the *Tool*.

The essential purpose of MVC is to bridge the gap between the human user's mental model and the digital model that exists in the computer. The ideal MVC solution supports the user illusion of seeing and manipulating the domain information directly. The structure is useful if the user needs to see the same model element simultaneously in different contexts and/or from different viewpoints. The figure below illustrates the idea.



MVC was conceived as a general solution to the problem of users controlling a large and complex data set. The hardest part was to hit upon good names for the different architectural components. Model-View-Editor was the first set: Thing-Model-View-Editor 12 May 1979 (11 pp).

After long discussions, particularly with Adele Goldberg, we ended with the terms Model-View-Controller: Models-Views-Controllers : 10 December 1979 (2 pp).

The Controller was here what I now call a *Tool*. The Smalltalk-80 Controller is here a fourth element called *Editor*. This was an ephemeral component that the View creates on demand as an interface between the View and the input devices such as mouse and keyboard.

The MVC problem has more facets than I realized in 1979. I started working on a pattern language to disentangle the different aspects, that last draft was dated August 20, 2003. The plan was that it should be improved by a group of authors, not just the current single one. Unfortunately, the projct died at his point : MVC Pattern Language.

A.5 MVVM by Gossman

https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps

The following is the content of the above page, duplicated here for archival reasons. I did not ask the permission from Gossman yet. The link above is a blog entry, but the images have disappeared. I got them back on a duplicated page here: https://www.cnblogs.com/shenfengok/archive/2011/10/18/2216135.html.

Please cite the following as:

Gossman, J. (10/08/2005(?)) Introduction to Model/View/ViewModel pattern for building WPF apps, https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps.

Model/View/ViewModel is a variation of Model/View/Controller (MVC) that is tailored for modern UI development platforms where the View is the responsibility of a designer rather than a classic developer. The designer is generally a more graphical, artistic focused person, and does less classic coding than a traditional developer. The design is almost always done in a declarative form like HTML or XAML, and very often using a WYSIWYG tool such as Dreamweaver, Flash or Sparkle. In short, the UI part of the application is being developed using different tools, languages and by a different person than is the business logic or data backend. Model/View/ViewModel is thus a refinement of MVC that evolves it from its Smalltalk origins where the entire application was built using one environment and language, into the very familiar modern environment of Web and now Avalon development.

Model/View/ViewModel also relies on one more thing: a general mechanism for data binding. More on that later.

The Model is defined as in MVC; it is the data or business logic, completely UI independent, that stores the state and does the processing of the problem domain. The Model is written in code or is represented by pure data encoded in relational tables or XML.

The View in Model/View/ViewModel consists of the visual elements, the buttons, windows, graphics and more complex controls of a GUI. It encodes the keyboard shortcuts and the controls themselves manage the interaction with the input devices that is the responsibility of Controller in MVC (what exactly happened to Controller in modern GUI development is a long digression...I tend to think it just faded into the background. It is still there, but we don't have to think about it as much as we did in 1979). The View is almost always defined declaratively, very often with a tool. By the nature of these tools and declarative languages some view state that MVC encodes in its View classes is not easy to represent. For example, the UI may have multiple modes of interaction such as "view mode" and "edit mode" that change the behavior of the controls or the look of the visuals, but these modes can't always be expressed in XAML (though triggers are a great start). We will solve this problem later.

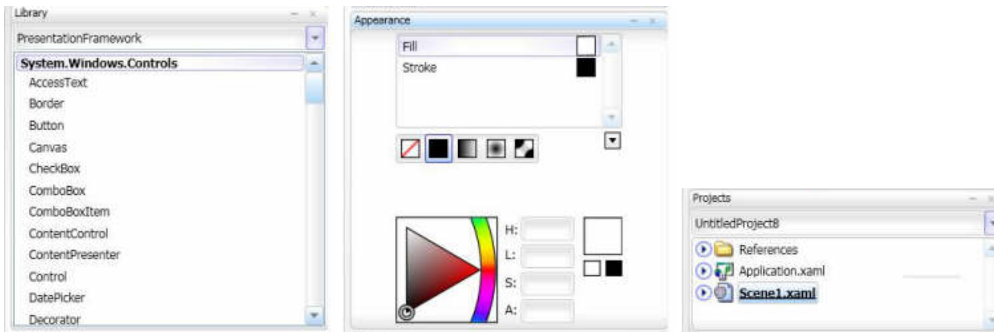
At this point data binding comes into play. In simple examples, the View is data bound directly to the Model. Parts of the Model are simply displayed in the view by one-way data binding. Other parts of the model can be edited by directly binding controls two-way to the data. For example, a boolean in the Model can be data bound to a CheckBox, or a string field to a TextBox.

In practice however, only a small subset of application UI can be data bound directly to the Model, especially if the Model is a pre-existing class or data schema over which the application developer has no control. The Model is

very likely to have a data type that cannot be mapped directly to controls. The UI may want to perform complex operations that must be implemented in code which doesn't make sense in our strict definition of the View but are too specific to be included in the Model (or didn't come with the pre-existing model). Finally we need a place to put view state such as selection or modes.

The ViewModel is responsible for these tasks. The term means "Model of a View", and can be thought of as abstraction of the view, but it also provides a specialization of the Model that the View can use for data-binding. In this latter role the ViewModel contains data-transformers that convert Model types into View types, and it contains Commands the View can use to interact with the Model.

I will develop these ideas, and describe in particular how to bind the View to Commands on the ViewModel in later posts. But the quickest way to clarify this pattern is to provide some examples:



The above picture shows three editing panels in the Sparkle UI. Each has been developed using the Model/View/ViewModel pattern. The simplest is the Library Panel at the top. The Model is a list of assemblies (each an instance of System.Reflection.Assembly), and associated with each list of assemblies is a list of controls. The View is one of our Panel chrome controls and a series of Styles and DataTemplates that expose the assembly list in a ComboBox and the list of controls in a ListBox. We data bind the caption of the ComboBox directly to the name of the Assembly and the items of the listbox take their text from the name of the Control. The ViewModel has as state the currently selected Assembly and exposes Commands for inserting a control into the scene. Selection is one of the most common components of a ViewModel. But since controls have selection, you may wonder why selection isn't left in the View instead. This is done because often many controls in the view need to coordinate based on a single selection. It is easier to bind to a single representation of selection in the ViewModel than coordinate all the different controls in the view. In the Library, the selected Assembly determines what is selected by the ComboBox and also what list data is displayed by the ListBox. Additionally, the designer could decide to switch to using a ListBox for the assemblies and a ComboBox for the control list without copying over selection coordination logic from his original view.

The Appearance panel has as its Model the selected shapes or controls in Sparkle's editing area. The View has a ListBox that displays the interesting properties on the selection (basically all the Pen and Brush properties), buttons that determine whether the Brush or Pen is solid, gradient etc. and a color spectrum for editing color components. The ViewModel includes what property is selected, what gradient stop is selected when editing a gradient, data converters for mapping colors to text values and to positions in the color spectrum, and commands for changing the Pens and Brushes being edited. In this case the Model was given to us by Avalon, the View could

easily be changed to something radically different, and the ViewModel provides an abstract representation for the reusable parts of this UI.

The final example is our Project panel. Here the Model is an MSBuild Project...again a Model class that was pre-existing. The View is a tree control, scrolling area, and contains context menus. The ViewModel adapts MSBuild concepts that were designed without Avalon in mind (and work perfectly fine from the command line) so we can data bind to them, and again contains selection and commands.

Once you've grokked the Model/View/ViewModel pattern any UI problem is quickly stated in its terms. In fact, the entire Sparkle UI is defined using the pattern. The "selected shapes or controls in the editing area" that is the Model for the Appearance panel is itself a ViewModel concept from our Scene editor. The layout of Panels inside Sparkle has a Model consisting of a list of all registered panels, a View made up of a Grid with splitters that position the views, and a ViewModel that includes what panels are currently visible and what logical containers they are in (editing area, right, left, bottom trays).

A.6 Personal communication between Don Hopkins and Alan Kay on MVC

https://news.ycombinator.com/item?id=8841428

I [Don Hopkins, qui a beaucoup travaillé sur les Pie-Menus] asked Alan Kay about his thoughts on MVC

Alan Kay:

Things seem to hang on in computing just because they work a little bit.

The good part was philosophical -- the idea to adapt the notion of "cameras" and "worlds" in the original 3D graphics stuff I participated in at Utah 45 years ago. The bad part of MVC was how we implemented it -- much too much machinery, etc.

We have never done a really satisfactory automatic inverter for dealing with the loss of "dimensions" that happen when a view is made (but we have done some experimental ones).

A.7 Personal communication between Kyle Brown and Joëlle Coutaz on PAC

https://wiki.c2.com/?WhatsaControllerAnyway

Joelle Coutaz kindly replied to an email I [Kyle Brown, from [1]] sent her a while back asking about the origins of the PAC architecture. She informed me that when she formulated PAC that she was *unaware* of MVC, and that she chose the word "control" independently. She later discovered MVC and was delighted to see the similarities, but also noted that her (independently coined) term "control" was similar to the term "controller" used with a very different meaning.