MVC

Separando el código de presentación del código de diminio

Código de presentación vs. Código de dominio

- Cualquier código que hace algo con la interfaz de usuario, solo debe estar relacionado a la interfaz de usuario. Le llamamos “código de presentación”
  - Si hace algo con la interfaz, no debe manipular los datos (salvo para formatearlos) o hacer cálculos.
- Si hace cálculos, transforma datos, hace validaciones, interactúa con otros sistemas, le llamamos “código de dominio”
  - Si es código de dominio, no debe tener ninguna referencia al “código de presentación”

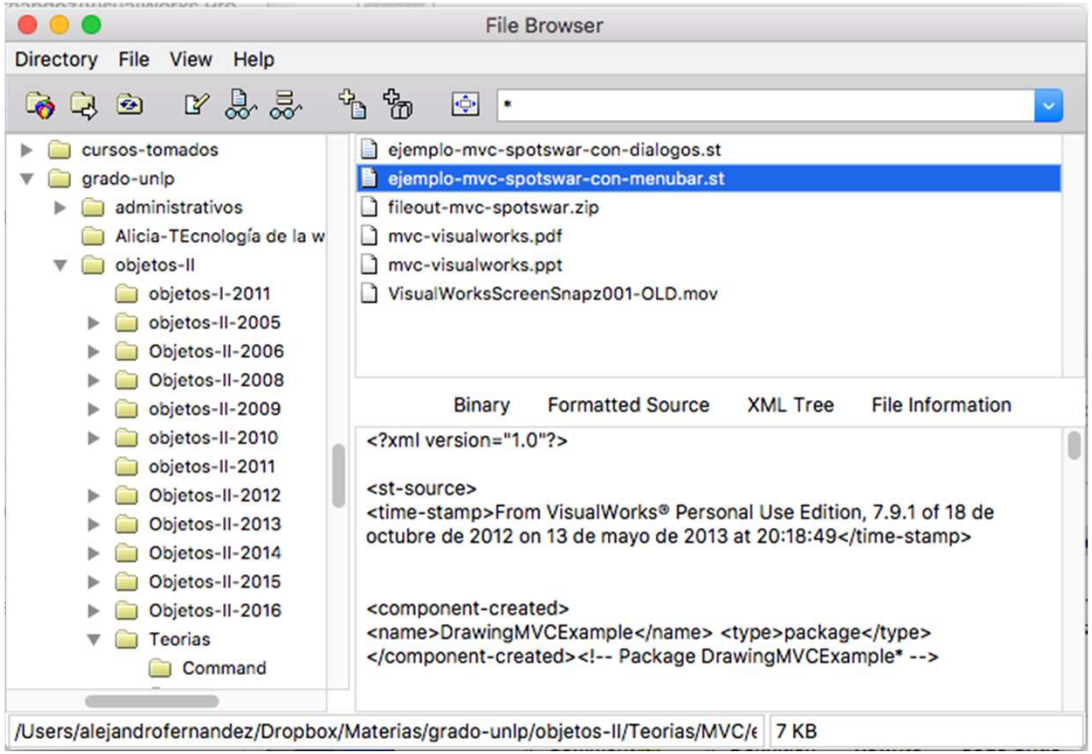
M. Fowler, "Separating user interface code," in *IEEE Software*, vol. 18, no. 2, pp. 96-97, March-April 2001.

¿Por que separarlos?

- El estilo con el que programamos el código de presentación, y la complejidad de este difiere del estilo y complejidad del código de dominio.
  - Mantenerlos separados permite que nos concentremos en una sola cosa a la vez
  - El código de presentación suele utilizar librerías que solo sirven para presentación.
  - Los tiempos con los que cambia uno y otro son diferentes
- Nos permite tener múltiples presentaciones para un mismo código de dominio
  - El código de dominio es mas fácil de portar

M. Fowler, "Separating user interface code," in *IEEE Software*, vol. 18, no. 2, pp. 96-97, March-April 2001.

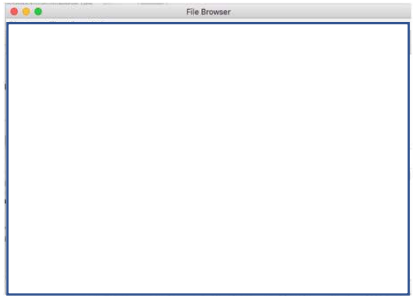
La interfaz de usuario

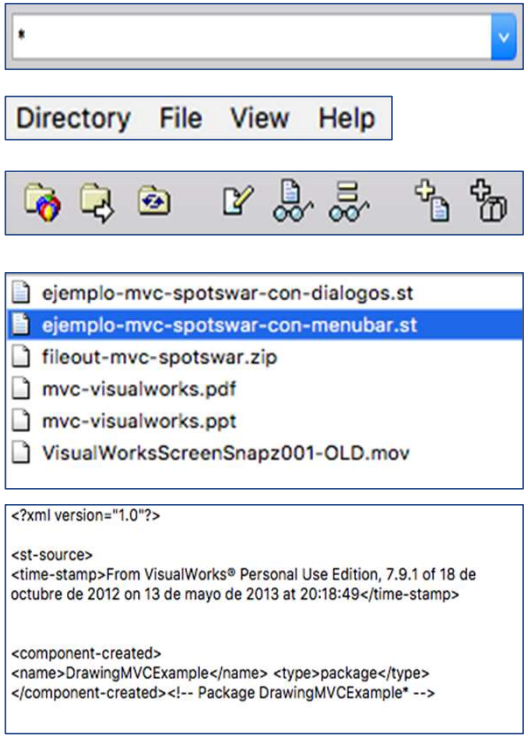


- Me permite explorar el modelo de objetos subyacente
- Me permite interactuar con el modelo de objetos
- Respeta un conjunto de convenciones que hemos aprendido con el tiempo (p.ej., metáfora de ventanas)

Visualworks (CINCOM)

El interfaz de usuario

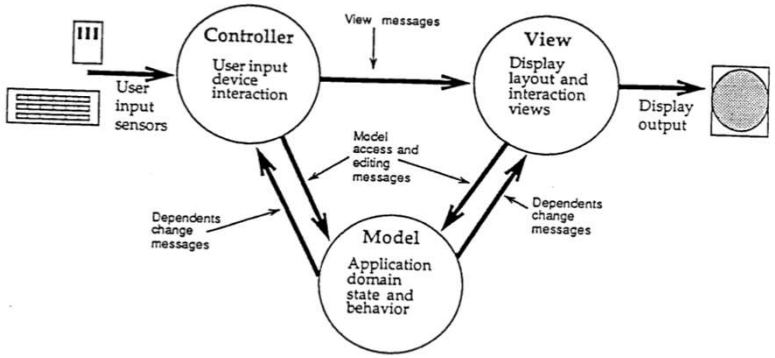




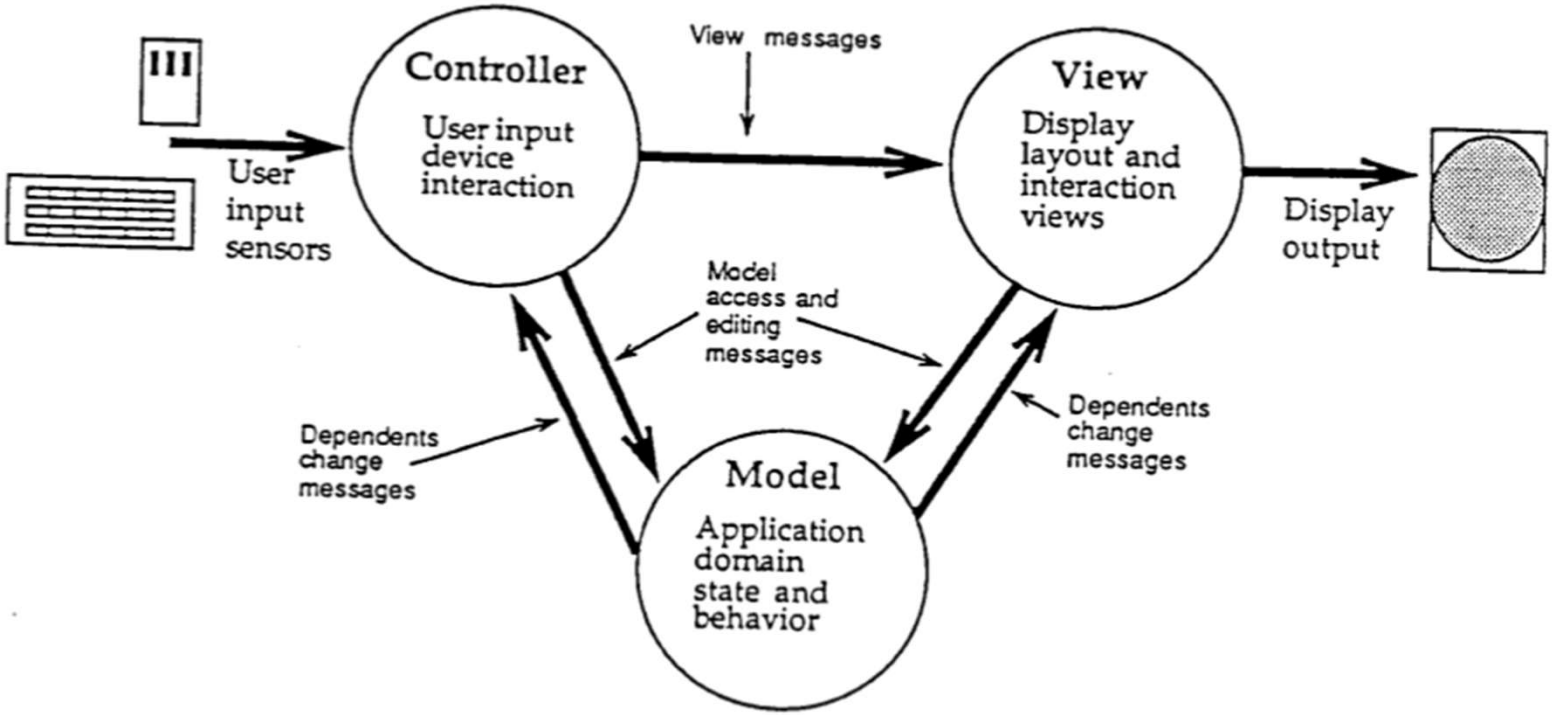
- Se construye componiendo elementos reutilizables (widgets)
  - Organizados en librerías
- Hacer widgets reutilizables implica:
  - Programar lo que se ve
  - Programar lo que se hace
  - Programar los objetos subyacentes

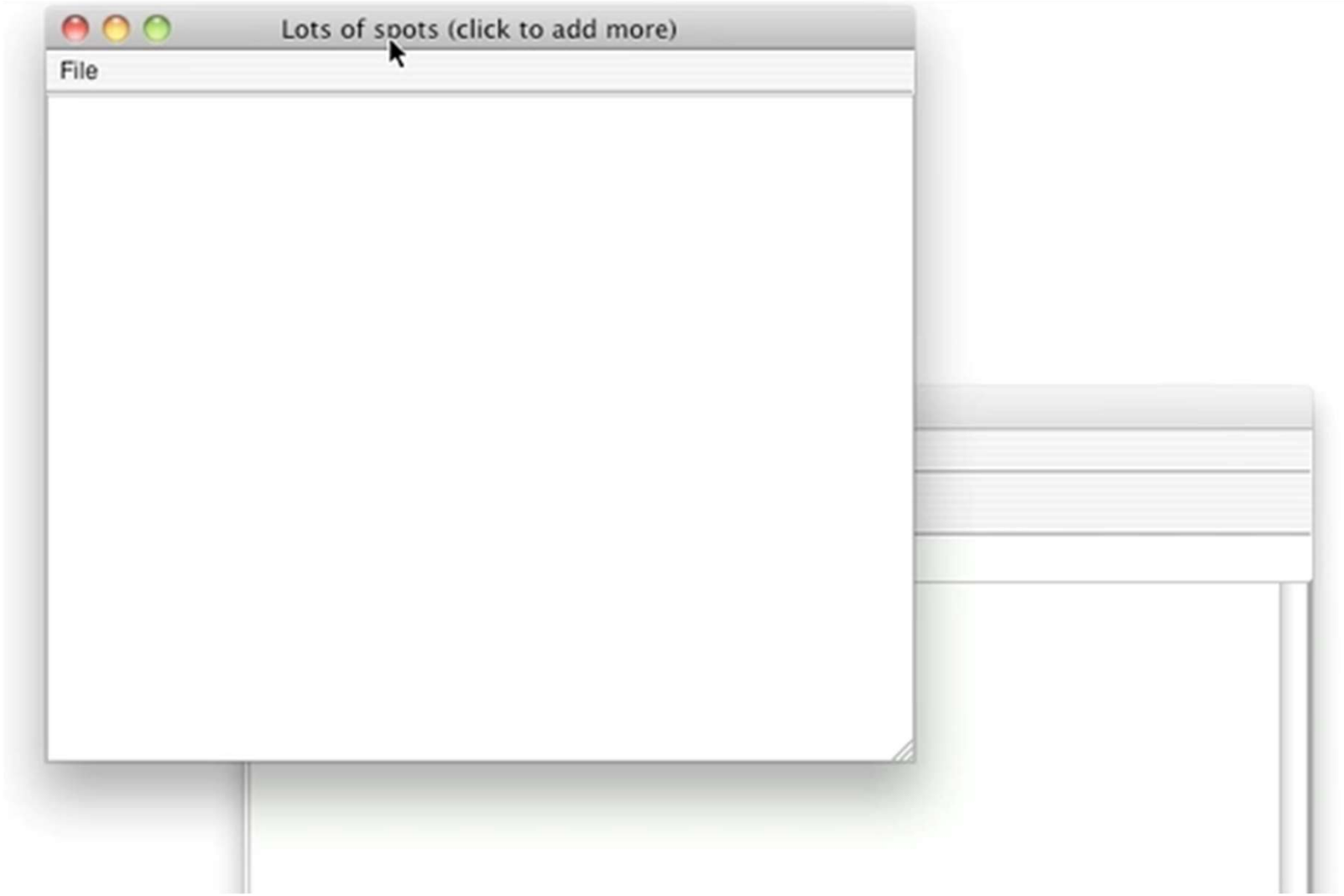
A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System

Glenn E. Krasner and Stephen T. Pope
ParcPlace Systems, Inc.
1550 Plymouth Street Mountain View, CA 94043 glenn@ParcPlace.com



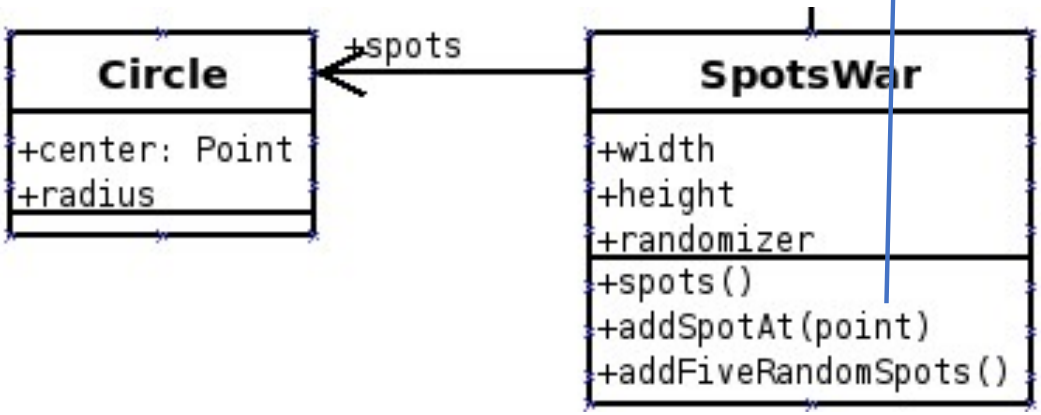
A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System

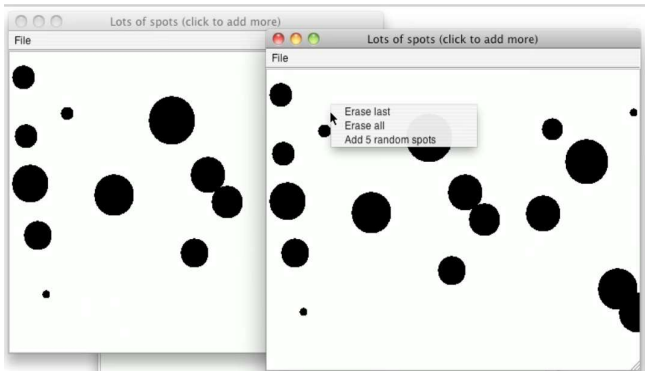



Lots of spots (click to add more)
File

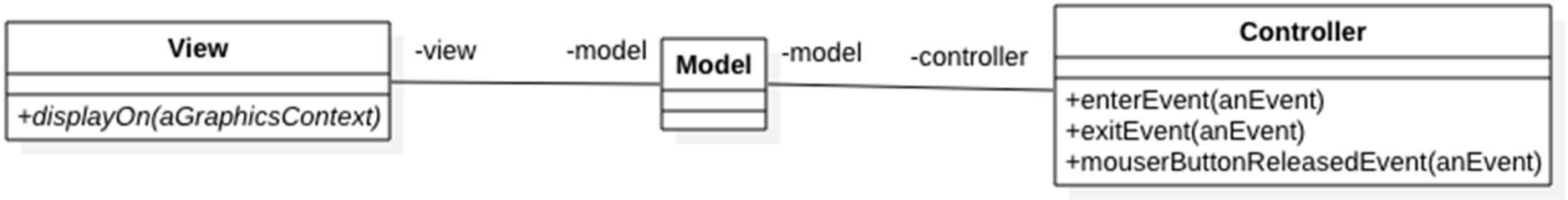
El modelo

```
addSpotAt: aPoint
    "Adds a new spot (a Circle) a the new coordinate,
    with random radius"

    | spot |
    spot := Circle
            center: aPoint
            radius: self generateRandomRadius.
    spots add: spot.
```

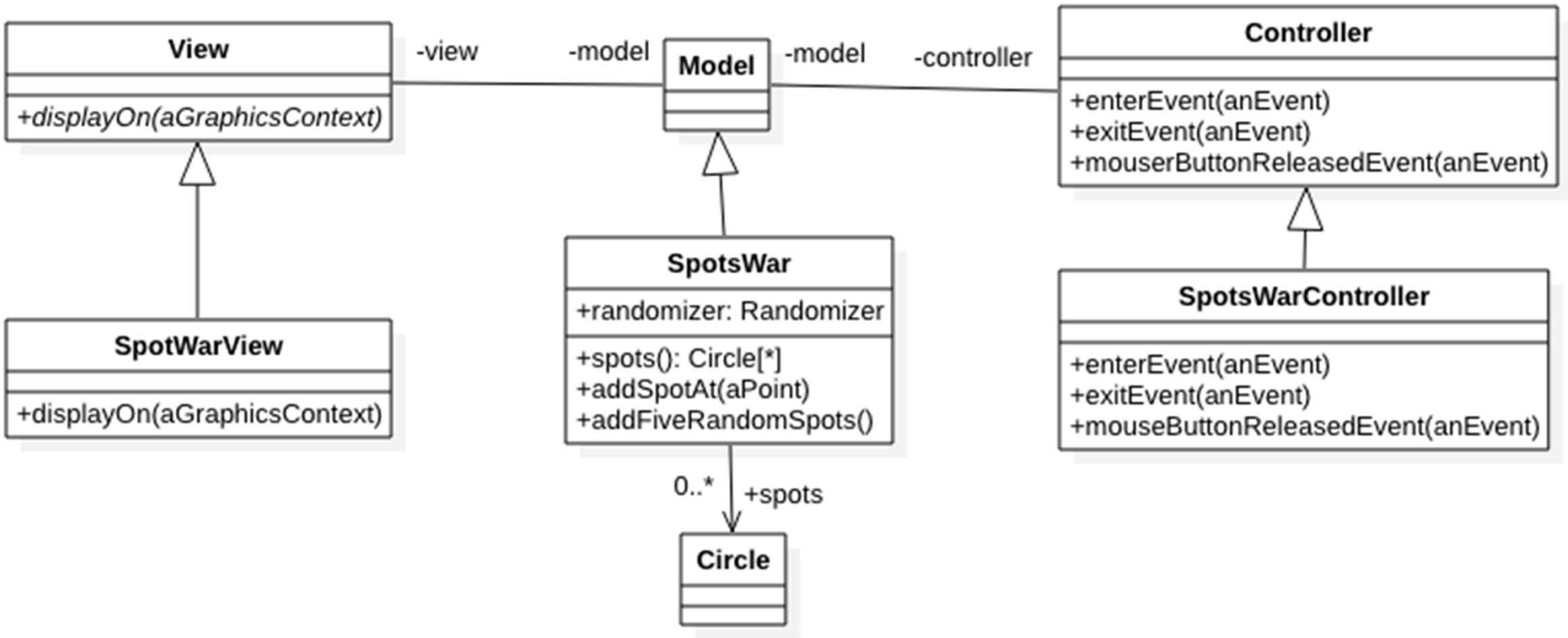




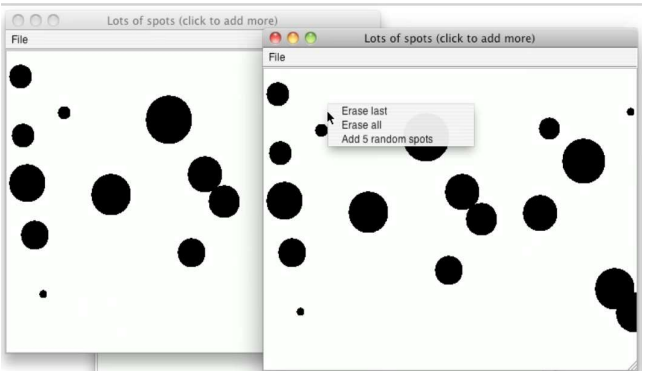
MVC: hotspots (en Visualworks Smalltalk)

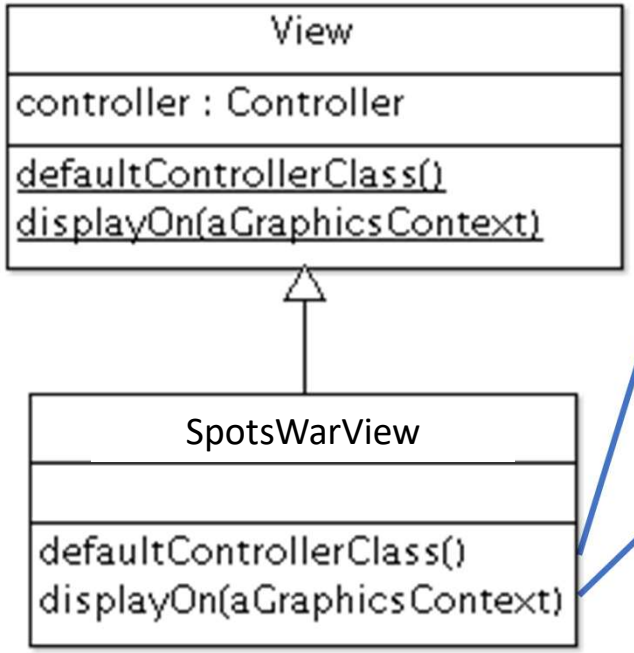


- Clases abstractas modelan cada uno de los tres elementos
- Subclasifico (hotspots de caja blanca) para agregar el comportamiento específico de mis MVCs
- #displayOn: es abstracto, mientras que #enterEvent: es concreto (aunque no hace nada)


View
+displayOn(aGraphicsContext)
-view
-model
Model
-model
-controller
Controller
+enterEvent(anEvent)
+exitEvent(anEvent)
+mouserButtonReleasedEvent(anEvent)
SpotWarView
+displayOn(aGraphicsContext)
SpotsWar
+randomizer: Randomizer
+spots(): Circle[*]
+addSpotAt(aPoint)
+addFiveRandomSpots()
SpotsWarController
+enterEvent(anEvent)
+exitEvent(anEvent)
+mouseButtonReleasedEvent(anEvent)
0..*
+spots
Circle

La Vista

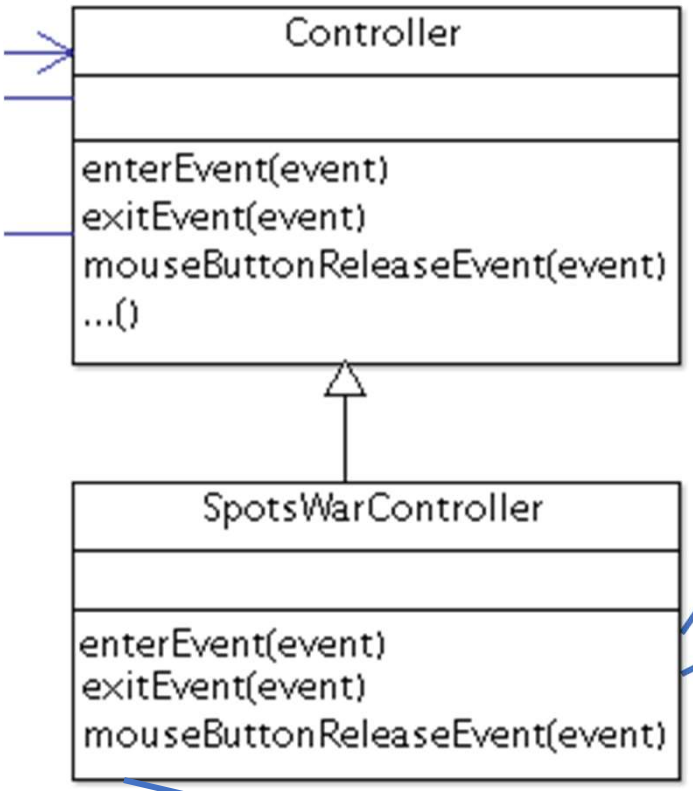





```
defaultControllerClass

    ^SpotWarController
```

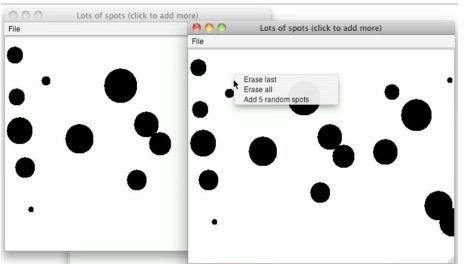
```
displayOn: aGraphicsContext

    self model isNil ifTrue: [^self].
    self model spots
        do: [:spot | aGraphicsContext displayFilledCircle: spot]
```

El controlador




```
enterEvent: anEvent
    Cursor bull show
```

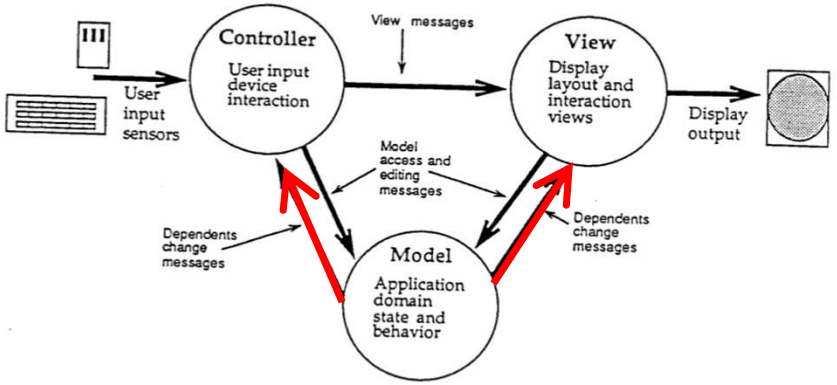
```
exitEvent: anEvent
    Cursor normal show
```



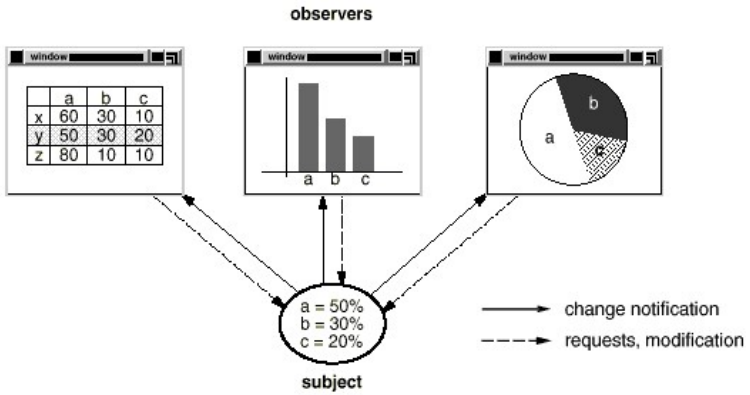

```
mouseButtonReleasedEvent: anEvent

    model addSpotAt: (self sensor cursorPointFor: anEvent)
```

Actualización de la vista …

- La vista conoce al modelo
  - Cuando se pinta, pide información al modelo
- El controlador conoce al modelo
  - Le envía mensajes y lo cambia
- Cuando se actualiza la vista?
  - ¿Le avisa el controlador?
  - ¿Le avisa el modelo? Si es así, ¿no queda muy acoplado?

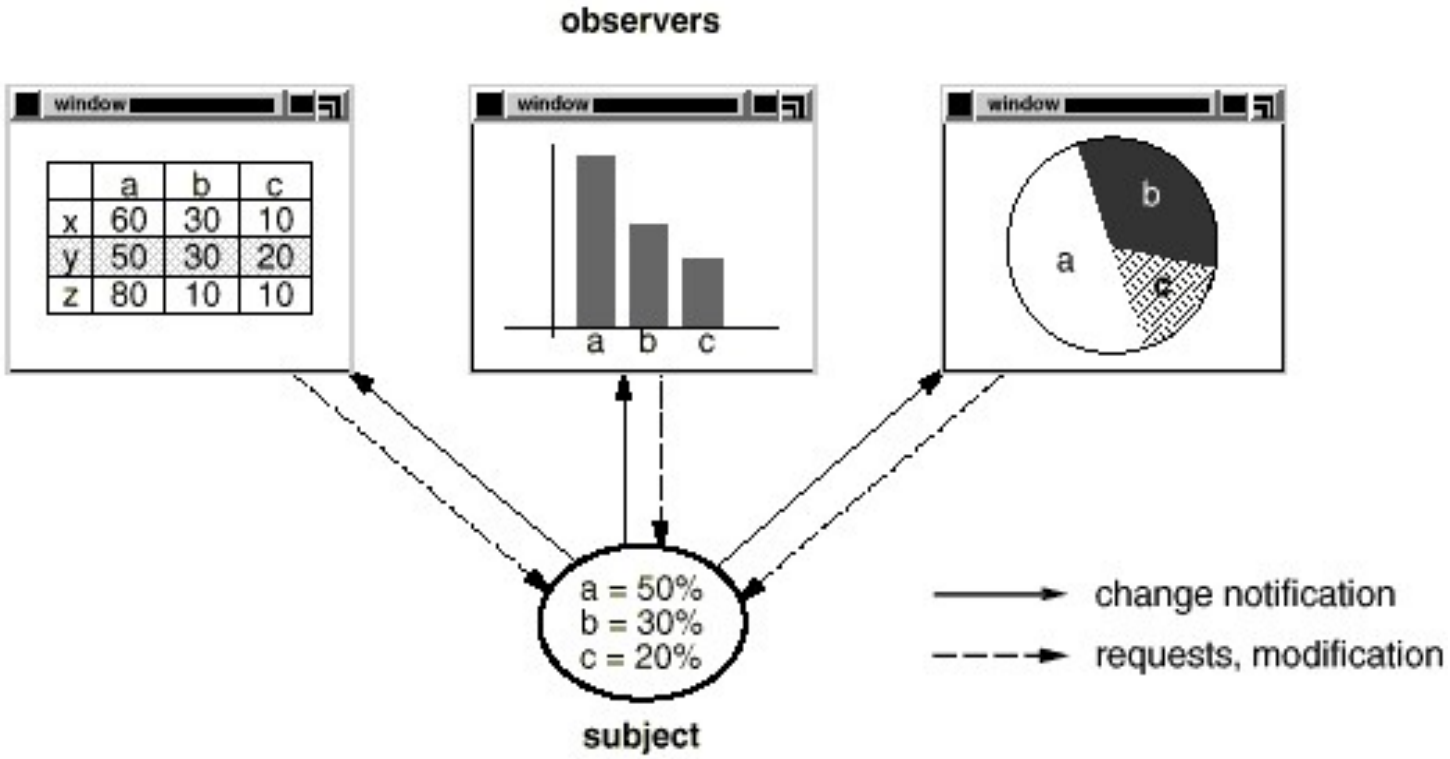


Modelos que avisan



- ¿Qué pasa si los cambios se originan fuera de la aplicación? Por ejemplo, en otra aplicación que muestra el mismo modelo
- Es necesario que el modelo notifique de cambios a sus vistas
  - Es “fundamental” que el modelo no se ensucie con código específico a la vista.
  - El modelo no debe saber si tiene ninguna, una o varias vistas, ni conocer sus protocolos

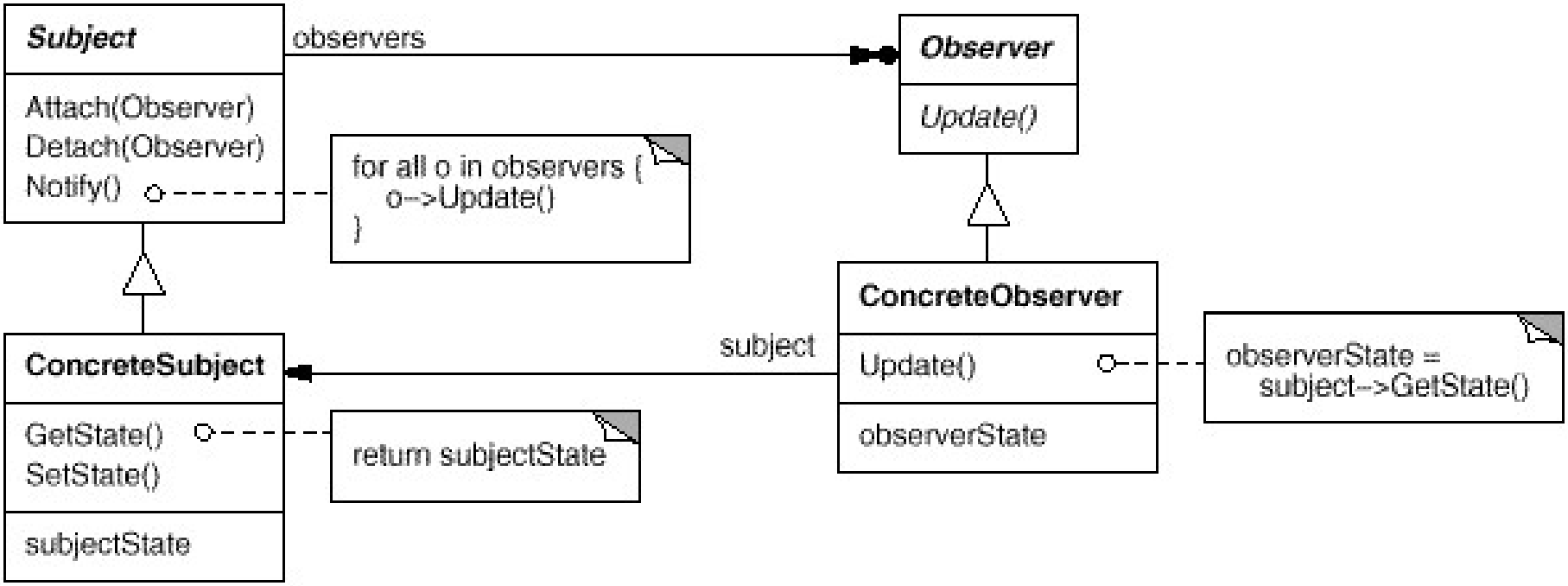
Pattern observer



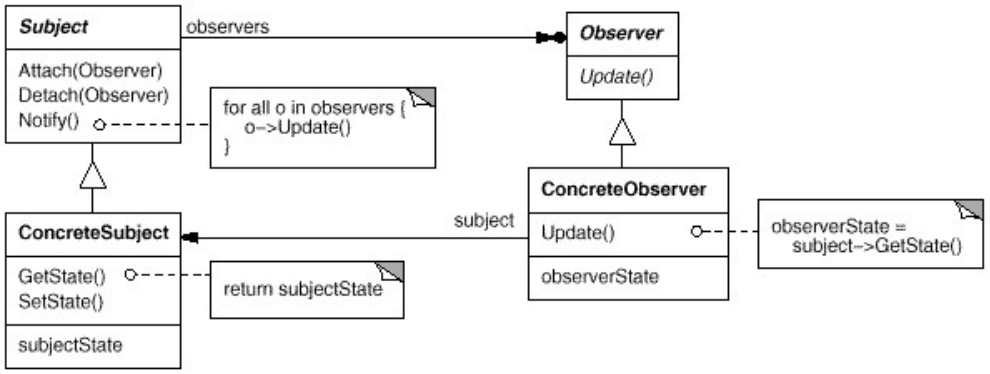
Patrón Observador

- Intención:
  - Definir una relación uno-a-muchos entre objetos de forma que cuanto un objeto cambia su estado, todo sus dependientes son notificados y actualizados automáticamente.
- Ejemplos comunes:
  - Las vistas de una aplicación son notificadas cuanto el modelo cambia de estado
  - Los tableros de instrumentos se actualizan cuando los sensores ven valores nuevos y avisan de cambios
  - Las alarmas se activan cuando alguno de los sensores detecta movimiento, humo, calor, etc. y avisa

Estructura


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

Participantes

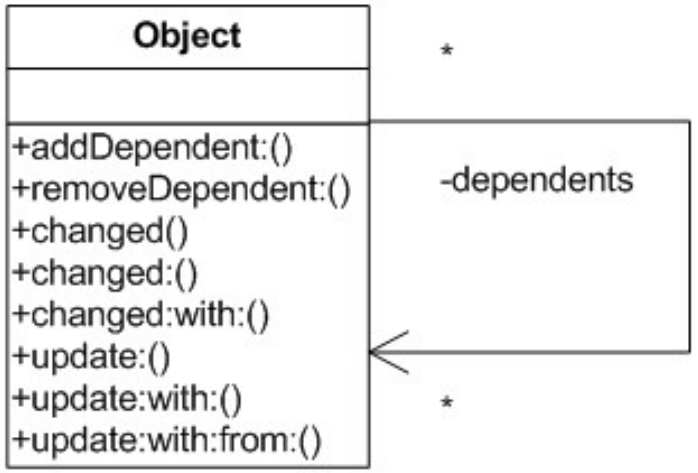


- Sujeto (Subject)
  - Conoce a sus observadores. Cualquier numero de observadores pueden observar a un sujeto.
  - Provee una interfase (mensajes) para registrar y des-registrar observadores.
- SujetoContreto (ConcreteSubject, p.e., SpotsWar)
  - Almacena el estado que les interesa a los ObservadoresConcretos.
  - Notifica a sus observadores cuando su estado cambia.
- Observador (Observer)
  - Define una interfaz de actualización de los objetos que serán notificado de cambia en un sujeto.
- ObservadorConcreto (ConcreteObserver, p.e., el SpotsWarView)
  - Mantiene una referencia a un SujetoConcreto.
  - Almacena estado que debe mantenerse consistente con el estado del SujetoConcreto.
  - Implementa la interfaz de actualización definida en la clase Observador para mantener su estado consistente con el de su SujetoConcreto.

Notificaciones con parámetros

- La notificación de cambio puede ir acompañada de argumentos, por ejemplo un indicador del aspecto que cambio (un string).
- El observador recibe el mensaje update() con los argumentos pasados por el sujeto, y lo utiliza para determinar la magnitud del cambio.
- Algunas implementaciones utilizan objetos complejos como argumentos. Eso objetos reciben el nombre general de anuncios (Announcements) y permiten implementar variantes mas ricas del patrón observador.
  - ¿Interesados en saber mas? Lean sobre announcement en Pharo.

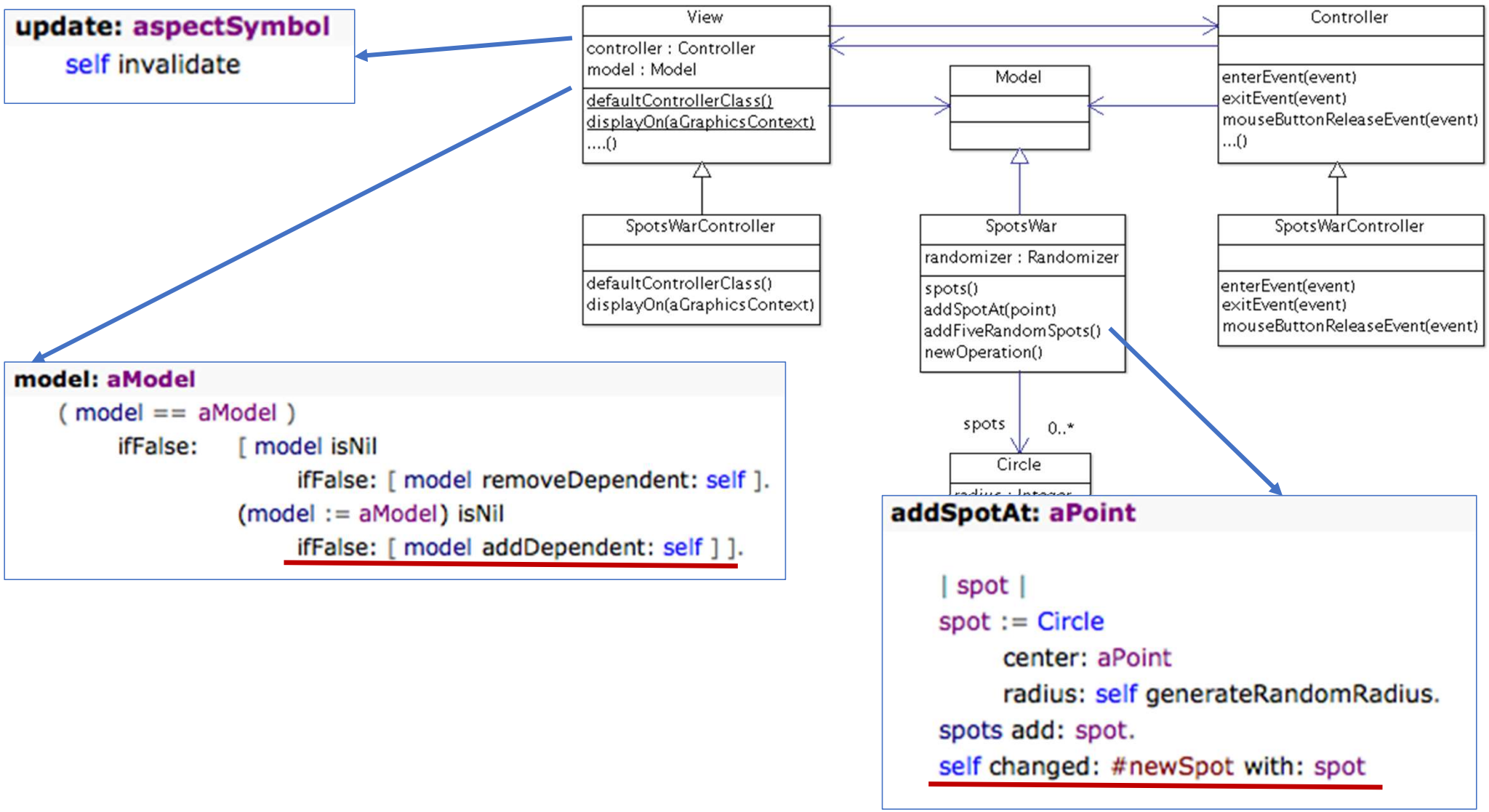
Observer en Pharo



- En Pharo todas las subclases de Object heredan la implementación del patern Observer (llamado dependencias).
- En principio podemos asumir que cada objeto maneja una colección de sus dependientes (aunque en realidad es algo mas complejo)
  - Para observar a un objeto le envío el mensaje #**addDependent**:
  - Cuando un objeto cambia se dice a si mismo #**changed**:
  - Cuando un sujeto cambia, el observador recibe el mensaje #**update**:

Utilizando dependencias / observer

- Nuestra vista quiere saber cuando algo cambia en el modelo por lo que se registra como dependiente del mismo enviándole el mensaje #addDependent: -> attach(observer)
- El modelo notifica cada vez que cambia enviándose a si mismo el mensaje #changed: -> notify()
- Como va a ser notificada, la vista implementa el método #update: -> update
  - En el #update:, la vista toma la información que necesita del modelo y “se invalida” para que el framework la repinte


update: aspectSymbol
self invalidate
View
controller : Controller
model : Model
defaultControllerClass()
displayOn(aGraphicsContext)
....()
Model
Controller
enterEvent(event)
exitEvent(event)
mouseButtonReleaseEvent(event)
...()
SpotsWarController
defaultControllerClass()
displayOn(aGraphicsContext)
SpotsWar
randomizer : Randomizer
spots()
addSpotAt(point)
addFiveRandomSpots()
newOperation()
SpotsWarController
enterEvent(event)
exitEvent(event)
mouseButtonReleaseEvent(event)
spots
0..*
Circle
model: aModel
(model == aModel)
ifFalse: [model isNil
ifFalse: [model removeDependent: self].
(model := aModel) isNil
ifFalse: [model addDependent: self]].
addSpotAt: aPoint
| spot |
spot := Circle
center: aPoint
radius: self generateRandomRadius.
spots add: spot.
self changed: #newSpot with: spot

Aplicabilidad

- Utilice el patrón observador en cualquiera de las siguientes situaciones:
  - Una abstracción tiene dos aspectos, uno dependiente del otro. Encapsular estos aspectos en objetos separados permite que los mismos varíen y sean reutilizados independientemente.
  - Cuando un cambio en un objeto requiere que otros objetos cambien, y usted no quiere saber cuantos otros objetos necesitan cambiar.
  - Cuando un objeto debe ser capaz de notificar a otros sin hacer conjeturas en relación a quienes son esos objetos. En otras palabras, usted no quiere a estos objetos fuertemente acoplados.

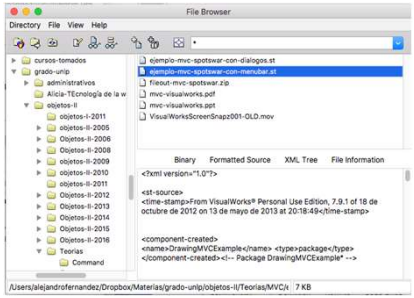
Colaboraciones (continuación)

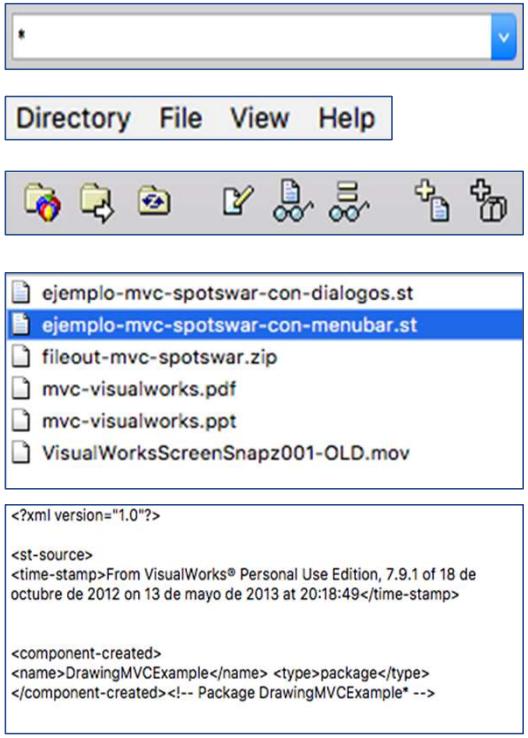
- El sujeto concreto notifica a sus observadores cuando ocurre un cambio que puede ser de interés para que sus observadores mantengan su estado consistente con el propio.
- Luego de ser informado de un cambo en el sujeto concreto, un observador concreto puede enviarle mensajes a sujeto para obtener mas información. El observador concreto utiliza esta información adicional para sincronizar su estado con el del sujeto.

Consecuencias

- El patrón Observador permite variar sujetos y observadores independientemente. Se puede reusar a los sujetos sin reusar los observadores y viceversa. Se pueden agregar nuevos observadores sin modificar el sujeto ni otros observadores.
- Acoplamiento abstracto entre Sujeto y Observador: Todo lo que un sujeto sabe es que tiene una lista de observadores, cada uno de los cuales implementa la interfaz de la clase abstracta Observador. El sujeto no conoce la clase concreta de ningún observador. Por tanto el acoplamiento entre observador y sujeto es abstracto y mínimo.
- Soporte para comunicación broadcast (uno a muchos): a diferencia que en un envío de mensaje tradicional, el sujeto no necesita indicar destinatario (receptor). Las notificaciones son distribuidas automáticamente a todos los objetos interesados que se subscribieron. El sujeto no se preocupa por cuantos interesados existen; su única responsabilidad es notificar cuando cambia. Esto permite agregar y eliminar observadores cuando uno quieren. Le corresponde al observador hacer algo con la notificación o ignorarla.

De vuelta al MVC

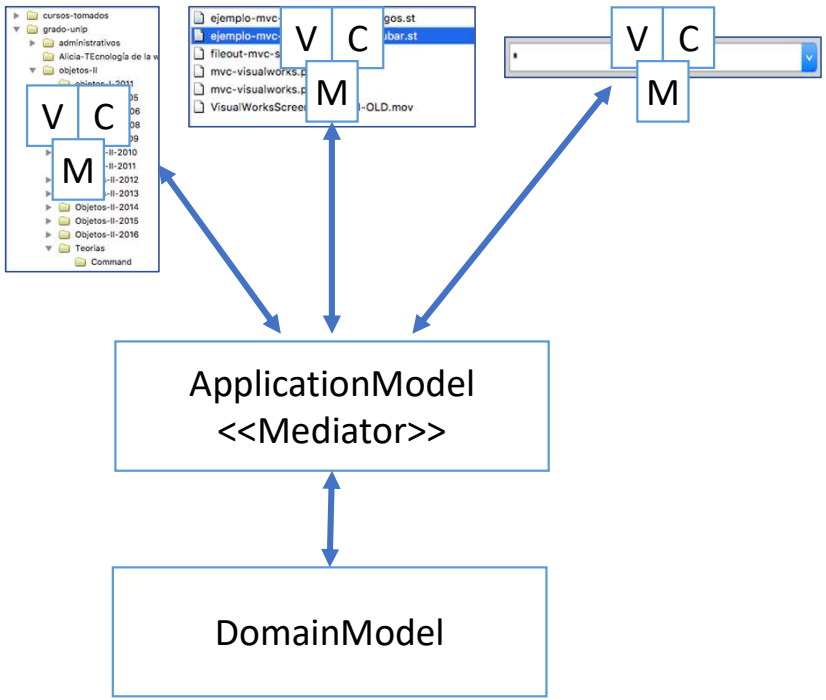




- Cada componente reutilizable implementa un MVC
- Se utilizan bloques y adaptadores para conectarlos a distintos modelos
- Nunca tienen comportamiento específico de la aplicación (por ejemplo, asociar dos listas)

¿Donde queda ese comportamiento específico de la aplicación?

Presenter/Component/ApplicationModel



- El comportamiento específico de la aplicación es responsabilidad del ApplicationModel
- Es un mediador (patrón Mediator)
- Maneja las relaciones entre los widgets
- Media entre los widgets y el modelo (a veces)