GoF: El patró Observer i l'arquitectura MVC

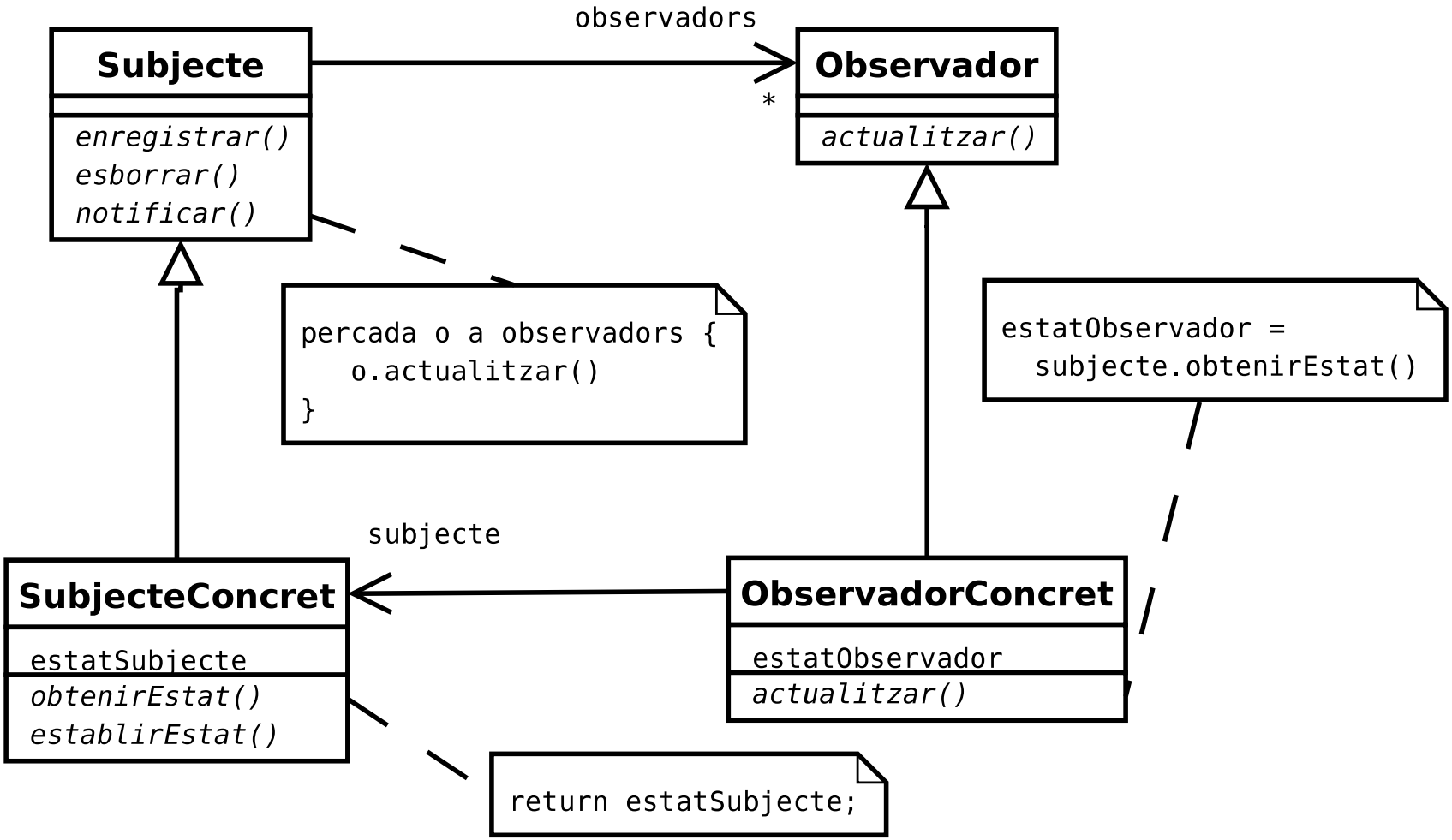
Juan Manuel Gimeno Illa
jmgimeno@diei.udl.cat

Curs 2008-2009

Índex

Observer (patró de comportament)

Defineix una dependència d'**un-a-molts** entre objectes de manera que, quan un objecte canvia d'estat, aquest canvi es notifica a tots els objectes que depenen d'ell i així poden actuar de la manera adequada.



Participants

Subjecte
- coneix els seus `Observadors` (qualsevol nombre d'ells)
- proporciona una interfície per enregistrar-los i esborrar-los

Observador
- defineix una interfície per actualitzar els objectes que han de ser notificats devant canvis en un `subjecte`

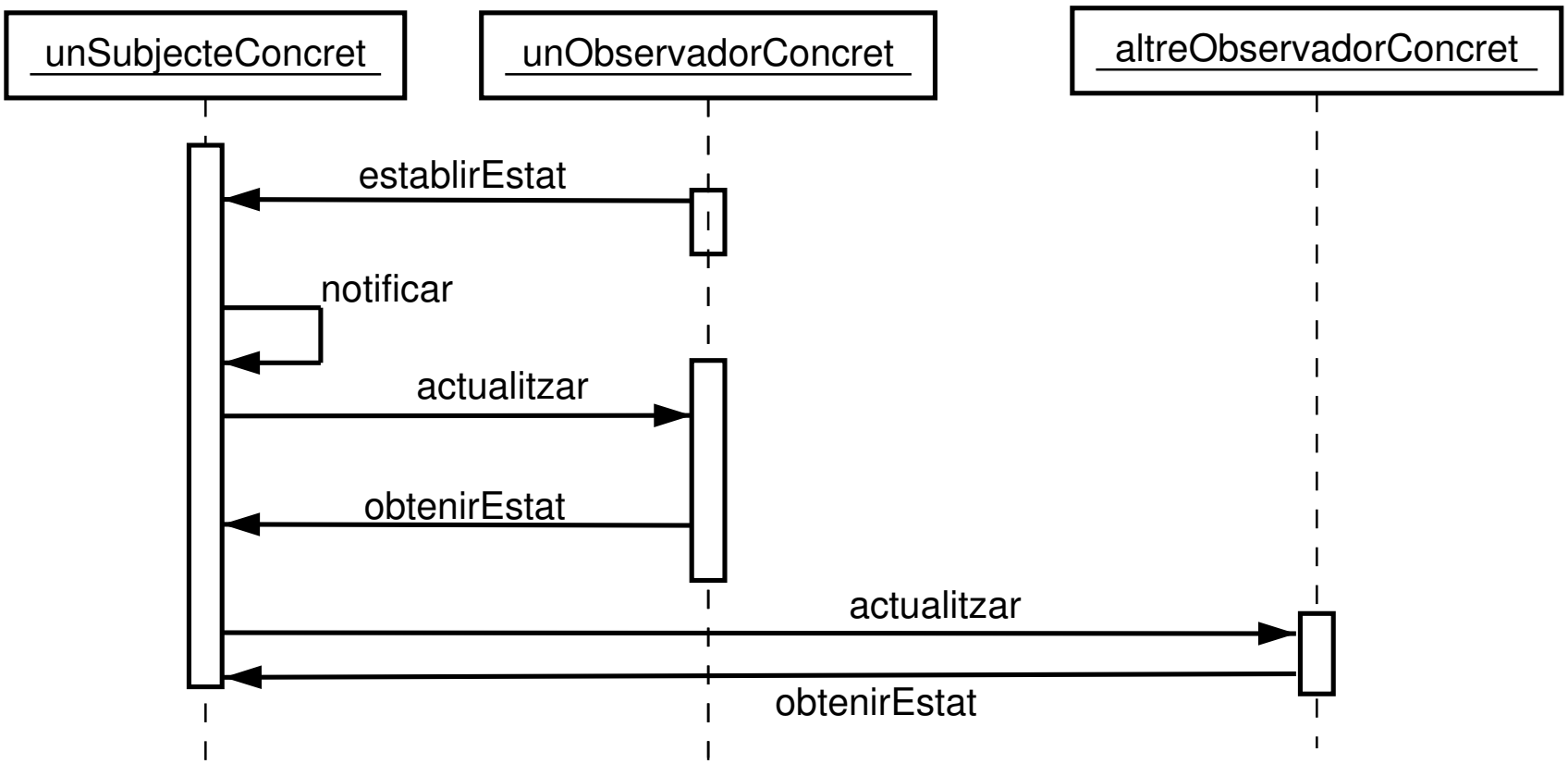
SubjecteConcret
- emmagatzema l'estat d'interés pels objectes `ObservadorConcret`
- envia una notificació als `Observadors` quan el seu estat canvia

ObservadorConcret
- manté una referència a un objecte `SubjecteConcret`
- guarda un estat que ha de ser consistent amb el del `SubjecteConcret`
- implementa la interfície d'actualització de l'`Observador`

Aplicabilitat

- Quan un canvi en un objecte requereix fer canvis a d'altres, i no se sap quants objectes han de ser canviats
- Quan un objecte ha de ser capaç de notificar a d'altres objectes sense fer cap mena d'assumpció sobre quins són. És a dir, no es vol que els objectes estiguin fortament acoblats
- Quan una abstracció té dos aspectes, un depenent de l'altre, però que interessa separar en objectes diferents per a reutilitzar-los independentment
- Com l'acoblament entre les parts és molt baix, aquests dos aspectes poden estar a capes diferents de l'aplicació (p.e. les dades que canvien i la part de la interfície que les mostra)

Col·laboracions



Fixeu-vos que l'`ObservadorConcret` (qui inicialitza el canvi) **postposa** la seva actualització fins a obtenir una notificació del subjecte.

Conseqüències

- Permet modificar els subjectes i els observadors de forma independent
- És possible reutilitzar subjectes sense els seus observadors i vice-versa
- Permet afegir observadors sense modificar els subjectes ni els altres observadors
- L'acoblament entre subjectes i observadors és mínim (poden pertànyer a capes diferents)
- Capacitat de comunicació mitjatçant difusió (*multicasting*): la notificació enviada per un subjecte no cal que especifiqui el seu receptor
- Problema de les *notificacions inesperades*: un observador no coneix el cost total d'un canvi en el subjecte ja que no coneix els seus observadors (i es poden produir canvis en cascada)

Variacions i aspectes d'implementació

- Utilitzar un registre central (possiblement un singleton) per a guardar la correspondència entre subjectes i observadors
- Permetre que un observador pugui observar més d'un subjecte: cal afegir un paràmetre al mètode `actualitzar` per a què l'observador sàpiga quin objecte examinar
- Qui dispara l'actualització?
  - les operacions que estableixen l'estat del subjecte després fan la notificació (problema amb les actualitzacions successives)
  - fer que els clients cridin a notificar (propens a errors ja que es poden oblidar)
- Si els subjectes poden esborrar-se: cal evitar que els observadors mantinguin referències a aquests objectes. Una forma de fer-ho es via notificació.

Variacions i aspectes d'implementació

- Assegurar-se de que l'estat del subjecte és consistent abans de fer la notificació

  El següent no funciona:

```
// A la classe MeuSubjecte
void operacio(int nouValor) {
  // aixo dispara la notificacio
  super.operacio(nouvalor);
  // actualitzem l'estat de la subclasse
  // (massa tard !!!)
  var += nouValor;
}
```

- Solució: enviar la notificació en mètodes plantilla

```
// A la classe Text
void tallar(SeleccioDeText t) {
  // substituirSeleccio redefinida a les subclasses
  substituirSeleccio(t);
  notificar();
}
```

Variacions i aspectes d'implementació

- Quantitat d'informació que envia el subjecte.
  Dos extrems:
  model pull no s'envia cap informació
  - enfatitza la ignorància del subjecte respecte els seus observadors
  - pot ser ineficient ja que les classes observador han de determinar què ha canviat sense ajuda per part del subjecte

  model push s'envia informació detallada sobre el canvi, independentment de que aquests la necessitin
  - assumeix que els subjectes saben alguna cosa sobre el que necessiten els observadors
  - els observadors poden perdre part de la seva capacitat de reutilització
  - pot millorar l'eficiència
- Especificació de quines modificacions són d'interés en el moment d'enregistrar-se
- ...

Patró observer a la biblioteca de Java

```
package java.util;
public class Observable {
    public void addObserver(Observer o) {...}
    public void deleteObserver(Observer o) {...}
    public int countObservers() {...}

    public void notifyObservers() {...}
    public void notifyObservers(Object arg) {...}

    public boolean hasChanged() {...}

    protected void clearChanged() {...}
    protected void setChanged() {...}
}
```

```
package java.util;
public interface Observer {
    public void update(Observable o, Object arg);
}
```

Exemple: Conversor de monedes

- Aquest exemple mostra com el patró Observador és usat per a connectar la capa presentació (vistes) i la capa domini (model).
- L'exemple consta de tres classes:

  `CModel` Conté la lògica del domini i rep, des de les vistes, les ordres del què cal fer. Juga el paper de `Observable` i notifica als `Observadors` els seus canvis.

  `RateView` Mostra una ràtio de canvi i notifica al model quan l'usuari l'ha modificat. Juga el paper d'`Observador` per tal d'estar sempre actualitzat respecte el `CModel`.

  `ValueView` El mateix que `RateView` però pels valors.
- També serveix com a introducció a l'arquitectura MVC.

Exemple: Conversor de monedes

```
enum Currency {DOLLAR, EURO, POUND};

class CModel extends Observable {

  private final EnumMap<Currency, Double> rates;
  private long value = 0;  // cents, euro cents, or pence
  private Currency currency = Currency.DOLLAR;

  public CModel() {
    rates = new EnumMap<Currency, Double>(Currency.class);
  }

  public void setRate(Currency currency, double rate) {
    rates.put(currency, rate);
    setChanged();
    notifyObservers(currency);
  }
  public double getRate(Currency currency) {
    return rates.get(currency);
  }
  ........
```

Exemple: Conversor de monedes

```
    .........
    public void setValue(Currency currency, long value) {
      this.currency = currency;
      this.value = value;
      setChanged();
      notifyObservers(null);
    }
    public long getValue(Currency currency) {
      if (currency == this.currency) {
        return value;
      } else {
        double ratio = getRate(currency)/getRate(this.currency);
        return Math.round(value*ratio);
      }
    }

    public void initialize(double... initialRates) {
      for (int i = 0; i < initialRates.length; i++) {
        setRate(Currency.values()[i], initialRates[i]);
      }
    }
}
```

Exemple: Conversor de monedes

```
class RateView extends JTextField implements Observer {
  private final CModel model;
  private final Currency currency;
  public RateView(final CModel model, final Currency currency) {
    this.model = model;
    this.currency = currency;
    addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         try {
           double rate = Double.parseDouble(getText());
           model.setRate(currency, rate);
           } catch (NumberFormatException x) {
           }
       }
    });
    model.addObserver(this);
  }
  public void update(Observable model, Object currency) {
    if (this.currency == currency) {
      double rate = ((CModel) model).getRate((Currency) currency);
      setText(String.format("%10.6f", rate));
    }
  }
}
```

Exemple: Conversor de monedes

```
class ValueView extends JTextField implements Observer {
  private final CModel model;
  private final Currency currency;
  public ValueView(final CModel model, final Currency currency) {
    this.model = model;
    this.currency = currency;
    addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          double entry = Double.parseDouble(getText());
          long value = Math.round(100.0*entry);
          model.setValue(currency, value);
          } catch (NumberFormatException x) {
          }
      }
    });
    model.addObserver(this);
  }
  public void update(Observable model, Object currency) {
    if (currency == null || currency == this.currency) {
      long value = ((CModel) model).getValue(this.currency);
      setText(String.format("%15d.%02d", value/100, value%100));
    }
  }
}
```

Exemple: Conversor de monedes

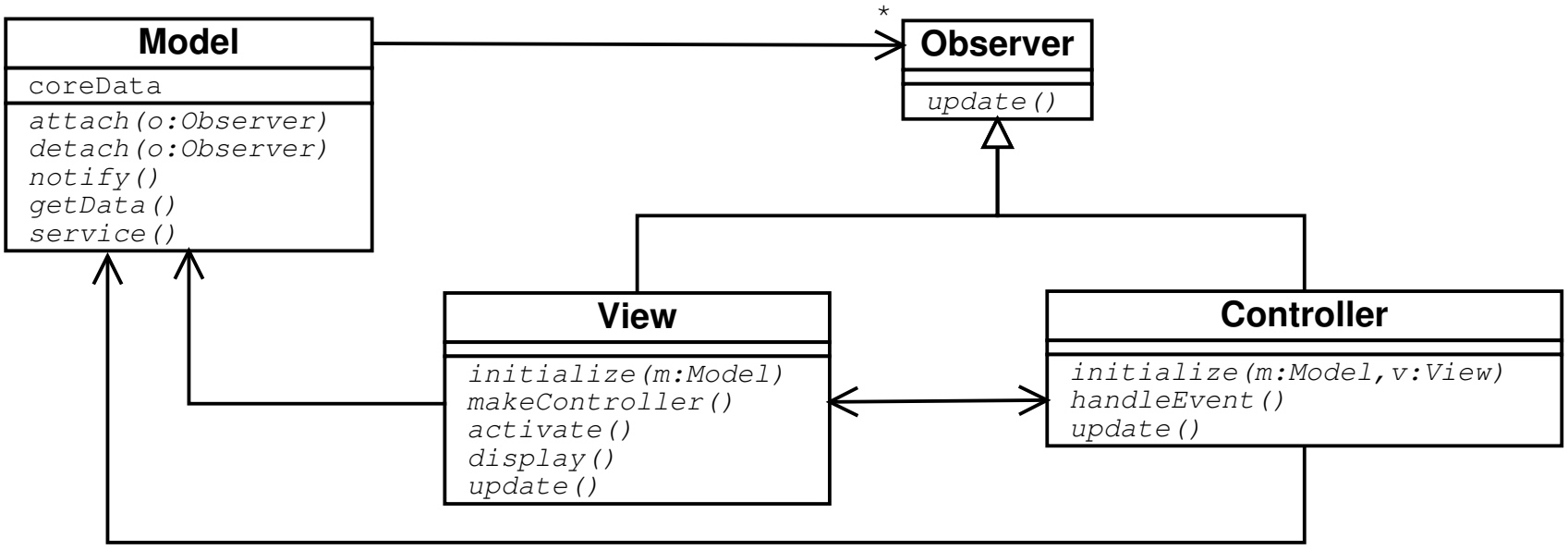
```
class Converter extends JFrame {
  public Converter() {
    CModel model = new CModel();
    setTitle("Currency converter");
    setLayout(new GridLayout(Currency.values().length + 1, 3));
    add(new JLabel("currency"));
    add(new JLabel("rate"));
    add(new JLabel("value"));
    for (Currency currency : Currency.values()) {
      add(new JLabel(currency.name()));
      add(new RateView(model, currency));
      add(new ValueView(model, currency));
    }
    model.initialize(1.0, 0.83, 0.56);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
  }
  public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new Converter().setVisible(true);
        });
  }
}
```

Model-Vista-Controlador

- MVC és un **patró arquitectural** que divideix una aplicació interactiva en tres parts:

  Model conté la funcionalitat central i les dades
  Vistes mostren informació a l'usuari
  Controladors maneguen les entrades dels usuaris
- *Vistes* i *Controladors* implementen la interfície d'usuari
- Un mecanisme de *notificació* de canvis assegura la consistència entre la interfície d'usuari i el model

Arquitectura



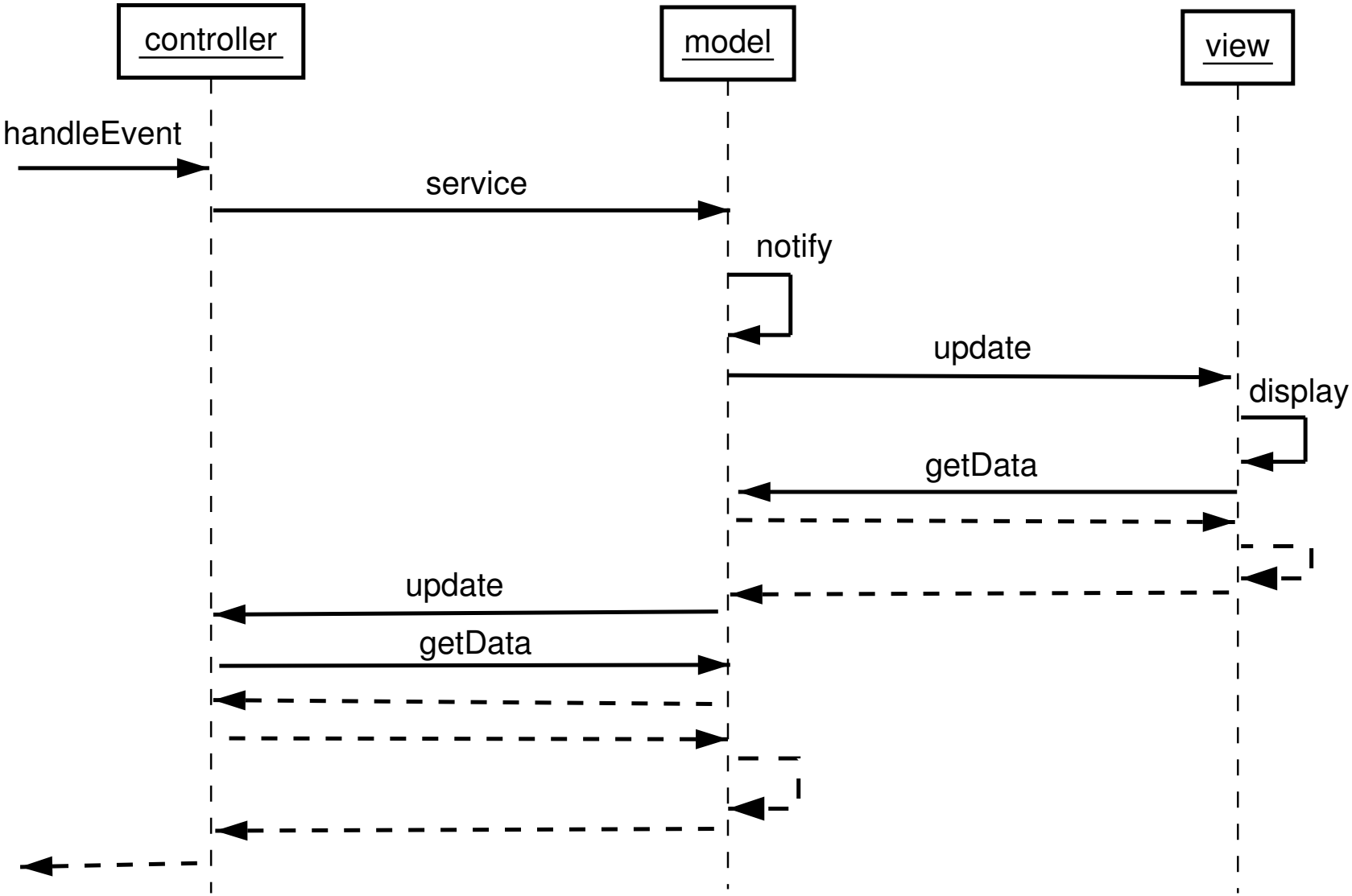
Responsabilitats dels participants

Model
- implementa la funcionalitat central de l'aplicació
- enregistrar les vistes i els controladors dependents
- notifica als components dependents sobre el canvis de les dades

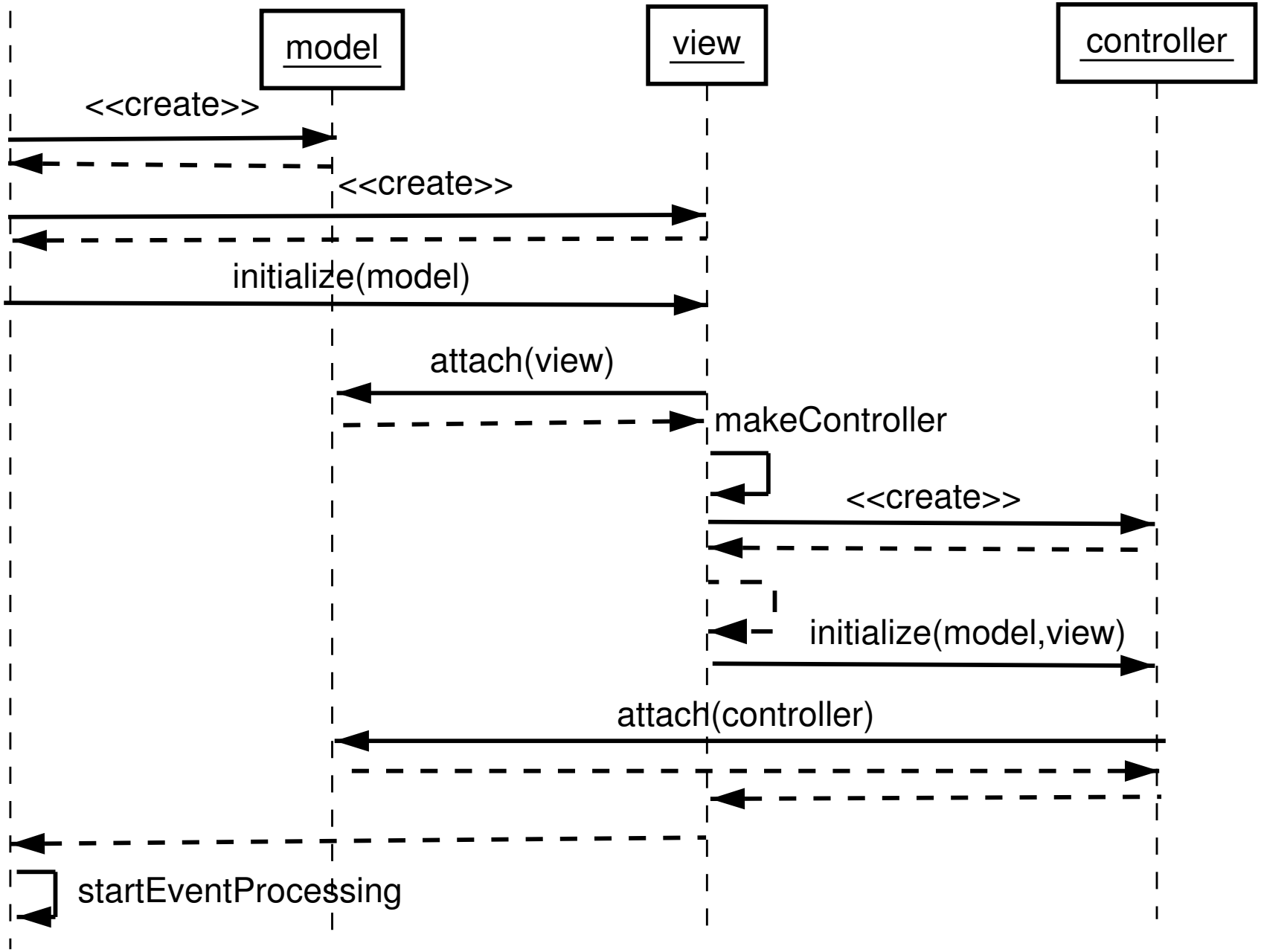
Vista
- crea i inicialitza el seu controlador associat
- mostra informació a l'usuari
- implementa el procediment d'actualització
- obté dades del model

Controlador
- accepta l'entrada de l'usuari mitjançant *events*
- tradueix els events en peticions al model o a la vista
- implementa el procediment d'actualització (si es necessita)

MVC: Processant events



MVC: Inicialització



Bibliografia

- Capítol 5 del llibre *Patrones de diseño*
- Capítol 2 del llibre *Pattern-Oriented Software Architecture. A System of Patterns. Vol.1*: F. Buschmann et al.
- Exemple d'observer: *M. Naftalin i P. Wadler*, Java Generics and Collections, O'Reilly 2006.

Llicència