Patrons

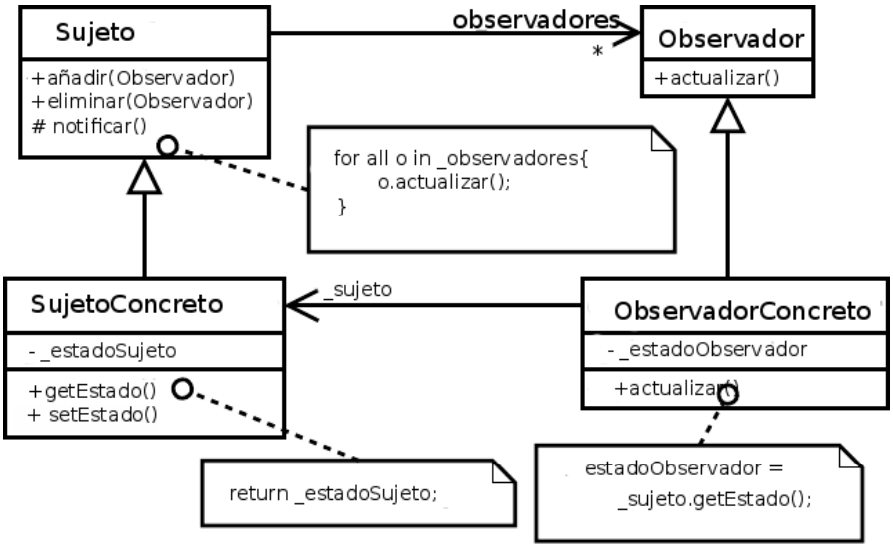
Observer

Patró Observer o observador → Patró de comportament

- Classificació: És un patró de comportament, ja que determina com s'ha de realitzar el intercanvi de missatges entre els diferents objectes per aconseguir realitzar una tasca.
- Propòsit: Definir una dependència del tipus d'un a molts entre objectes, de manera que quan un dels objectes canvia el seu estat, ho notifica a tots els objectes dependents.
- L'objecte de dades (Subjecte) disposa d'atributs mitjançant els quals qualsevol objecte observador o vista es pot subscriure a ell passant-li una referència a si mateix. El subjecte manté una llista de referències als seus observadors. Els observadors implementen uns mètodes determinats mitjançant els quals el subjecte és capaç de notificar als seus observadors "subscrits" els canvis que succeeixen perquè tots ells es puguin refrescar. De manera que quan es produeix un canvi en el subjecte, per exemple: executat, per algun dels observadors, el subjecte pot fer el recorregut per la llista dels observadors avisant a cadascun d'ells.
- Aquest patró es sol veure en els frameworks d'interfícies gràfiques orientades a objectes, en què la forma de capturar els esdeveniments és subscriure "listeners" als objectes que poden generar esdeveniments.

Patró Observer

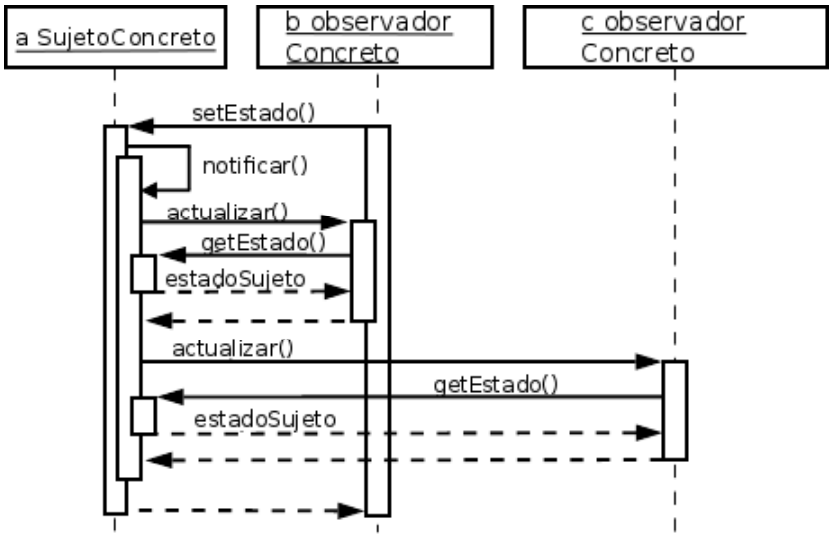
Estructura:



- Subjecte (Subject): Coneix als seus observadors. Proporciona una interfície per agregar, eliminar, notificar, .. Observadors.
- Observador (Observer): Defineix el mètode que usa el subjecte per notificar canvis en el seu estat (update/notify).
- Subjecte Concret (ConcreteSubject): Manté l'estat d'interès per als observadors concrets i els notifica quan canvia el seu estat. No tenen perquè ser elements de la mateixa jerarquia.
- Observador Concret (ConcreteObserver): Manté una referència al subjecte concret i implementa la interfície d'actualització, és a dir, guarden la referència de l'objecte que observen, així en cas de ser notificats d'algun canvi, poden preguntar sobre aquest canvi.

Patró Observer

Col·laboracions:



- Col·laboracions: La col·laboració més important en aquest patró és entre el subjecte i els seus observadors, ja que en el moment en que el subjecte pateix un canvi, aquest ho notifica als seus observadors
- El subjecte concret notifica als seus observadors cada vegada que es modifica el seu estat
- Després de ser informat d'un canvi en el subjecte concret, un observador concret interroga el subjecte per modificar la percepció que té del subjecte

Patró Observer

Conseqüències:

- EL patró Observer permet variar els subjectes i els observadors independentment. Es poden rebutjar subjectes sense el rebuix dels observadors i al revés. Permet agregar observadors sense modificar el subjecte o els observadors.
- Alguns avantatges i desavantatges del patró Observer són:
  1. Acoblament abstracte entre el subjecte i els observadors. Tot el que un objecte sap dels seus observadors és que té una llista d'objectes que satisfan la interfície Observer. Amb què podrien fins i tot pertànyer a dues capes diferents de l'arquitectura de l'aplicació.
  2. No s'especifica el receptor d'una actualització. S'envia a tots els objectes interessats
  3. Actualitzacions inesperades. Es podrien produir actualitzacions en cascada molt ineficients.

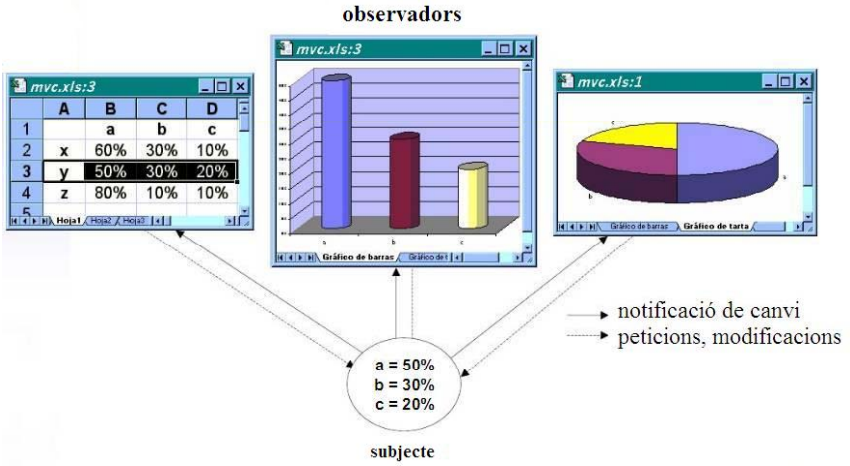
Patró Observer

Implementació:

- Implementació: associació subjecte-observadors
- Observació de més d'un subjecte
- Responsabilitat d'inici de la notificació
  - Subjecte invoca la notificació després d'executar-se un mètode que canviï el seu estat
  - Clients del subjecte (possiblement els propis observadors) són responsables de començar la notificació d'actualització
- Referències invàlides després d'esborrar un subjecte
- Assegurar la consistència de l'estat del subjecte abans d'iniciar la notificació
- Evitar actualitzacions dependent de l'observador
  - El subjecte envia informació detallada sobre el canvi (push model) (menys reusable)
  - El subjecte notifica els canvis; els observadors demanen la informació necessària (pull model) (ineficient)
- Especificació explícita de les modificacions

Exemple: una interfície d'usuari en la qual es posseeixen diferents representacions de determinades dades d'una aplicació. Les classes que representen les dades (el model) i les representacions gràfiques (les vistes) poden ser reutilitzades independentment.

- Un full de càlcul, un diagrama de barres i un diagrama de sectors poden mostrar diferents representacions de les dades d'una aplicació, però cada un d'aquests tres objectes es pot utilitzar independentment en altres aplicacions.
- Les modificacions sobre les dades han de repercutir en les tres presentacions, i fins i tot podria ocórrer que algun dels diagrames fos editable, i les modificacions també repercutirien a la resta de diagrames i en les pròpies dades.



- Separar les dades de la presentació.
- El full de càlcul i els gràfics mostren les mateixes dades amb diferents representacions.
- El full de càlcul i les gràfiques no es coneixen entre elles.

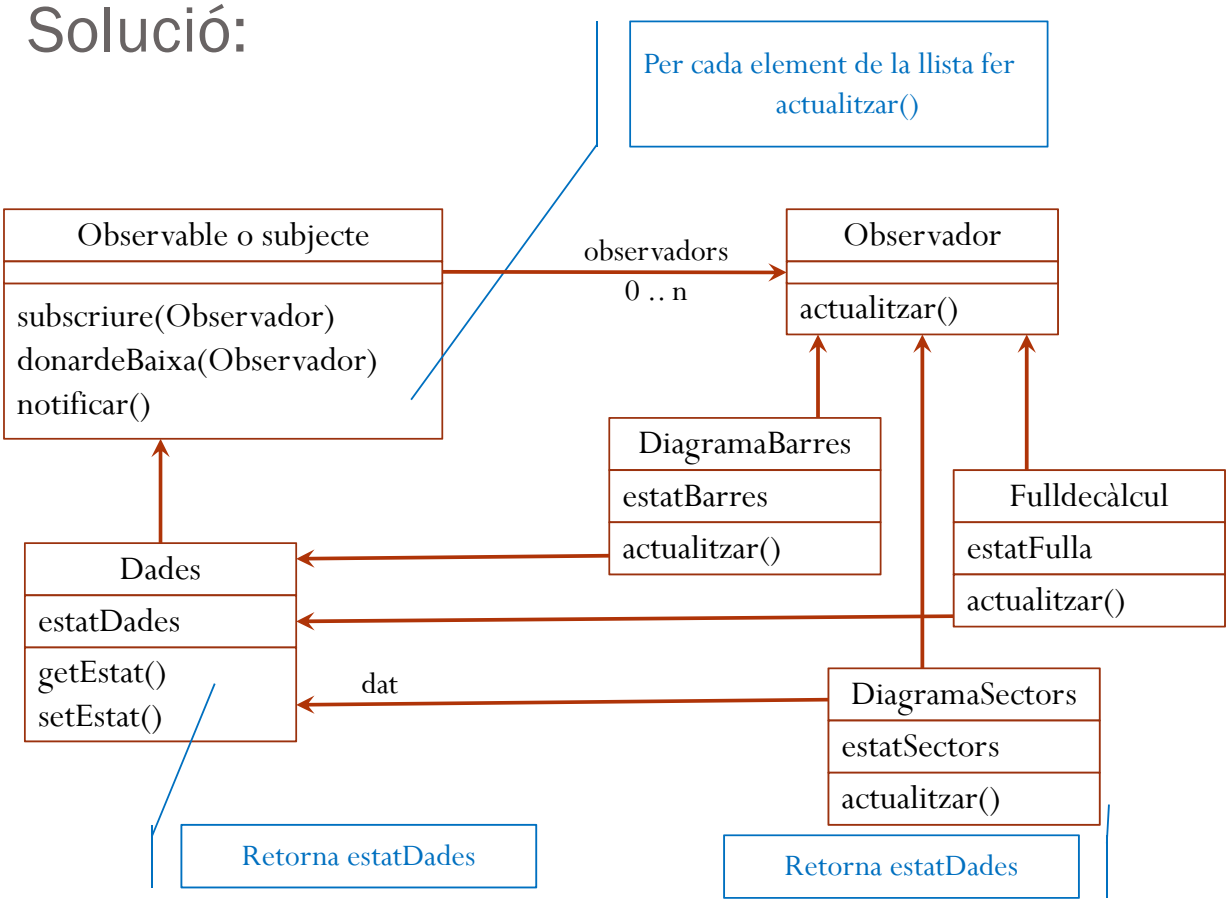
Motivació

- No es coneixen entre si, però es comporten com si es coneguessin: si l'usuari canvia dades en el full de càlcul llavors s'actualitzen els gràfics, i viceversa.
- Els gràfics i el full depenen de les dades.
- S'Han de notificar davant qualsevol canvi en les dades.

Solució

- El patró Observer descriu com establir les relacions.
- Defineix:
  - Subjects (subjectes): pot tenir diversos observadors.
  - Observers (observadors): són notificats quan un subjecte pateix un canvi d'estat. Preguntarà al subjecte pel nou estat per actualitzar-se.
  - Els subjectes publiquen notificacions, els observadors es subscriuen per rebre notificacions.
  - Les notificacions s'envien sense saber qui està subscrit.

Solució:



Exemple: Gestor d'esdeveniments genèric. (framework)

- Exemple d'un gestor d'esdeveniments que permet:
  - afegir i treure controladors en temps d'execució.
- El codi és completament genèric i independent dels controladors individuals.
- Els Gestors es poden implementar en mòduls separats i poden exportar una sèrie de funcions, funcions callback.
- Aquestes funcions poden ser cridades pel gestor d'esdeveniments.

Exemple: Gestor d'esdeveniments genèric

- Gestor d'esdeveniments: funcions del client
- start(Name, HandlerList)
  - Inicia un gestor d'esdeveniments genèric, registrat amb el nom *Name*. *HandlerList* és una llista de tuples de la forma {*Handler, Data*}, on *Handler* és el nom del mòdul i *Data* és l'argument passat al handler de la funció collback *init*.
- stop(Name)
  - Finalitzaran tots els controladors i aturarà el procés del gestor d'esdeveniments. Retorna una llista d'elements de la forma {*Handler, Data*}, on *Data* és el valor de retorn de la funció collback *terminate* dels controladors individuals.
- add_handler(Name, Handler, Data)
  - Afegir el handler *Handler*, passant *Data* com a argument de la collback *init* del handler.
- delete_handler(Name, Handler)
  - Treure el handler. Es crida el gestor de finalització de la funció *terminate* que retorna el valor de d'aquesta crida. Retorna la tupla {*error, instance*} si el Handler no existeix.
- get_data(Name, Handler)
  - Retorna el contingut de la variable d'estat del gestor (handler). Retorna la tupla {*error, instance*} si gestor (handler) no existeix.
- send_event(Name, Event)
  - Envia el contingut del esdeveniment a tots els handlers

Exemple: Gestor d'esdeveniments genèric

```
-module(event_manager).
-export([start/2, stop/1]).
-export([add_handler/3, delete_handler/2, get_data/2, send_event/2]).
-export([init/1]).

start(Name, HandlerList) ->
    register(Name, spawn(event_manager, init, [HandlerList])), ok.

init(HandlerList) ->
    loop(initialize(HandlerList)).

initialize([]) -> [];
initialize([{Handler, InitData} | Rest]) ->
    [{Handler, Handler:init(InitData)} | initialize(Rest)].
```

{Handler1, DadesIni_1}
{Handler2, DadesIni_2}
.
.
{HandlerN, DadesIni_N}

Seqüència:
1> event_manager:start(nom, LlistaHandlers).

Registra el 'nom' com el procés que gestiona tots els controladors

Inicialitza el Handler. Per exemple pot inicialitzar una impressora

Exemple: Gestor d'esdeveniments genèric

```
stop(Name) ->
    Name ! {stop, self()},
    receive
        {reply, Reply} ->
            Reply
    end.
```

```
terminate([]) -> [];
terminate([{Handler, Data} | Rest]) ->
    [{Handler, Handler:terminate(Data)} | terminate(Rest)].
```

```
loop(State) ->
    receive
        {request, From, Msg} ->
            {Reply, NewState} = handle_msg(Msg, State),
            reply(From, Reply),
            loop(NewState);
        {stop, From} ->
            reply(From, terminate(State))
    end.
```

Exemple: Gestor d'esdeveniments genèric

```
add_handler(Name, Handler, InitData) ->
    call(Name, {add_handler, Handler, InitData}).
```

```
> add_handlers(nom, Handler, IniDat)
```

```
nom!{request, From,{add_handler, Handler,IniDat}}
```

```
call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} -> Reply end.
```

```
reply(To, Msg) ->
    To ! {reply, Msg}.
```

```
loop(State) ->
    receive
        {request, From, Msg} ->
            {Reply, NewState} = handle_msg(Msg, State),
            reply(From, Reply),
            loop(NewState);
        {stop, From} ->
            reply(From, terminate(State))
    end.
```

Inicialitza el dispositiu i l'afegeix a la llista del subjecte. Retorna l'àtom 'ok' i la llista actualitzada

```
handle_msg({add_handler, Handler, InitData}, LoopData) ->
        {ok, [{Handler, Handler:init(InitData)} | LoopData]};
```

Exemple: Gestor d'esdeveniments genèric

```
delete_handler(Name, Handler) ->
    call(Name, {delete_handler, Handler}).
```

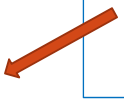
```
get_data(Name, Handler) ->
    call(Name, {get_data, Handler}).
```

```
handle_msg({delete_handler, Handler}, LoopData) ->
    case lists:keysearch(Handler, 1, LoopData) of
        false ->
            {{error, instance}, LoopData};
        {value, {Handler, Data}} ->
            Reply = {data, Handler:terminate(Data)},
            NewLoopData = lists:keydelete(Handler, 1, LoopData),
            {Reply, NewLoopData}
    end;
```

```
handle_msg({get_data, Handler}, LoopData) ->
    case lists:keysearch(Handler, 1, LoopData) of
        false ->
            {{error, instance}, LoopData};
        {value, {Handler, Data}} ->
            {{data, Data}, LoopData}
    end;
```

Exemple: Gestor d'esdeveniments genèric

```
send_event(Name, Event) ->
    call(Name, {send_event, Event}).
```



```
call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} -> Reply end.
```

Envia: nom ! {request, self(), {send_event, Event}} a loop/1 que crida a handle_msg(..)

```
handle_msg({send_event, Event}, LoopData) ->
    {ok, event(Event, LoopData)}.

event(_Event, []) -> [];

event(Event, [{Handler, Data} | Rest]) ->
    [{Handler, Handler:handle_event(Event, Data)} | event(Event, Rest)].
```

Exemple: Gestor d'esdeveniments genèric

Controladors

- Funcions callback dels controladors :
  - init(InitData):
    Inicialitza el controlador. Retorna l'identificador del controlador inicialitzat.
  - terminate(Data):
    Finalitza el controlador. Tanca tots els fitxers o sockets que s'han obert amb la funció d'inicialització. Retorna l'identificador del controlador.
  - handle_event(Event, Data):
    Es crida quant s'envia un esdeveniment al gestor d'esdeveniments amb la funció send_event/2. Retorna l'identificador del controlador.

Exemple: Gestor d'esdeveniments genèric

Controlador 1: Visualitza els esdeveniments a la consola

```
-module(io_handler).
-export([init/1, terminate/1, handle_event/2]).

init(Count) -> Count.

terminate(Count) -> {count, Count}.
```

```
handle_event({raise_alarm, Id, Alarm}, Count) ->
        print(alarm, Id, Alarm, Count),
        Count+1;
handle_event({clear_alarm, Id, Alarm}, Count) ->
        print(clear, Id, Alarm, Count),
        Count+1;
handle_event(Event, Count) -> Count.
```

```
print(Type, Id, Alarm, Count) ->
        Date = fmt(date()),
        Time = fmt(time()),
        io:format("#~w,~s,~s,~w,~w,~p~n",[Count, Date, Time, Type, Id, Alarm]).

fmt({AInt,BInt,CInt}) ->
        AStr = pad(integer_to_list(AInt)),
        BStr = pad(integer_to_list(BInt)),
        CStr = pad(integer_to_list(CInt)),
        [AStr,$:,BStr,$:,CStr].
```

```
pad([M1])  -> [$0,M1];
pad(Other) -> Other.
```

Exemple: Gestor d'esdeveniments genèric

Controlador 2: Emmagatzema els esdeveniments en un fitxer

```
-module(log_handler).
-export([init/1, terminate/1, handle_event/2]).

init(File) ->
    {ok, Fd} = file:open(File, write),
    Fd.

terminate(Fd) ->
    file:close(Fd).

handle_event({Action, Id, Event}, Fd) ->
    {MegaSec, Sec, MicroSec} = now(),
    Args = io:format(Fd, "~w,~w,~w,~w,~w,~p~n",[MegaSec, Sec, MicroSec, Action, Id, Event]),
    Fd;
handle_event(_, Fd) -> Fd.
```

Exemple: Us del gestor d'esdeveniments

1> **event_manager:start(alarm, [{log_handler, "FitxerAlarm"}]).**

Inicialitzem el gestor d'esdeveniments amb un controlador log_handler i la dada **FitxerAlarm**

```
start(Name, HandlerList) ->
    register(Name, spawn(event_manager, init, [HandlerList])), ok.
```

alarm

{log_handler, "FitxerAlarm"}

Crea el procés init/1 i el registre el nom "alarm"

```
init(HandlerList) ->
    loop(initialize(HandlerList)).
```

State:
[{log_handler, IdContr}]

```
initialize([]) -> [];
initialize([{Handler, InitData} | Rest]) ->
    [{Handler, Handler:init(InitData)} | initialize(Rest)].
```

```
loop(State) ->
    receive
        {request, From, Msg} ->
            {Reply, NewState} = handle_msg(Msg, State),
            reply(From, Reply),
            loop(NewState);
        {stop, From} -> reply(From, terminate(State))
end.
```

Crida a la funció log_handler:init("FitxerAlarm"). Retorna l'identificador del fitxer

Exemple: Us del gestor d'esdeveniments

```
2> event_manager:send_event(alarm, {raise_alarm, 10, armari_obert}).
```

Enviem l'alarma 10 que senyala que la porta de l'armari està oberta al gestor d'esdeveniments

```
send_event(Name, Event) ->
    call(Name, {send_event, Event}).
```

{"raise_alarm, 10, armari_obert"}

alarm

{send_event,{"raise_alarm, 10, armari_obert"}}

```
call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} -> Reply end.
```

```
loop(State) ->
    receive
        {request, From, Msg} ->
            {Reply, NewState} = handle_msg(Msg, State),
            reply(From, Reply),
            loop(NewState);
        {stop, From}  ->    reply(From, terminate(State))
end.
```

```
handle_msg({send_event, Event}, LoopData) ->
    {ok, event(Event, LoopData)}.
```

```
event(_Event, []) -> [];
event(Event, [{Handler, Data}|Rest]) ->
    [{Handler, Handler:handle_event(Event, Data)}|event(Event, Rest)].
```

FitxerAlarm: 1352,225766,640000,generar_alarma,10,armari_obert

Exemple: Us del gestor d'esdeveniments

```
3> event_manager:add_handler(alarm, io_handler, 1).
ok
4> event_manager:send_event(alarm, {raise_alarm, 10, armari_1_obert}).
#1,2012:11:06,20:25:19,alarm,10,armari_obert
ok
5> event_manager:send_event(alarm, {clear_alarm, 10, armari_obert}).
#2,2012:11:06,20:29:31,clear,10,armari_obert
ok
6> event_manager:get_data(alarm, io_handler).
{data,3}
7> event_manager:delete_handler(alarm, stats_handler).
{error,instance}
8> event_manager:delete_handler(alarm, io_handler).
{data,{count,3}}
9> event_manager:stop(alarm).
[{log_handler,ok}]
```

FitxerAlarm:
1352,230095,874000,raise_alarm,10,armari_obert
1352,230105,171000,raise_alarm,10,armari_obert
1352,230171,531000,clear_alarm,10,armari_obert