Design Patterns

MSc in Communications Software

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

GUI Synchronization

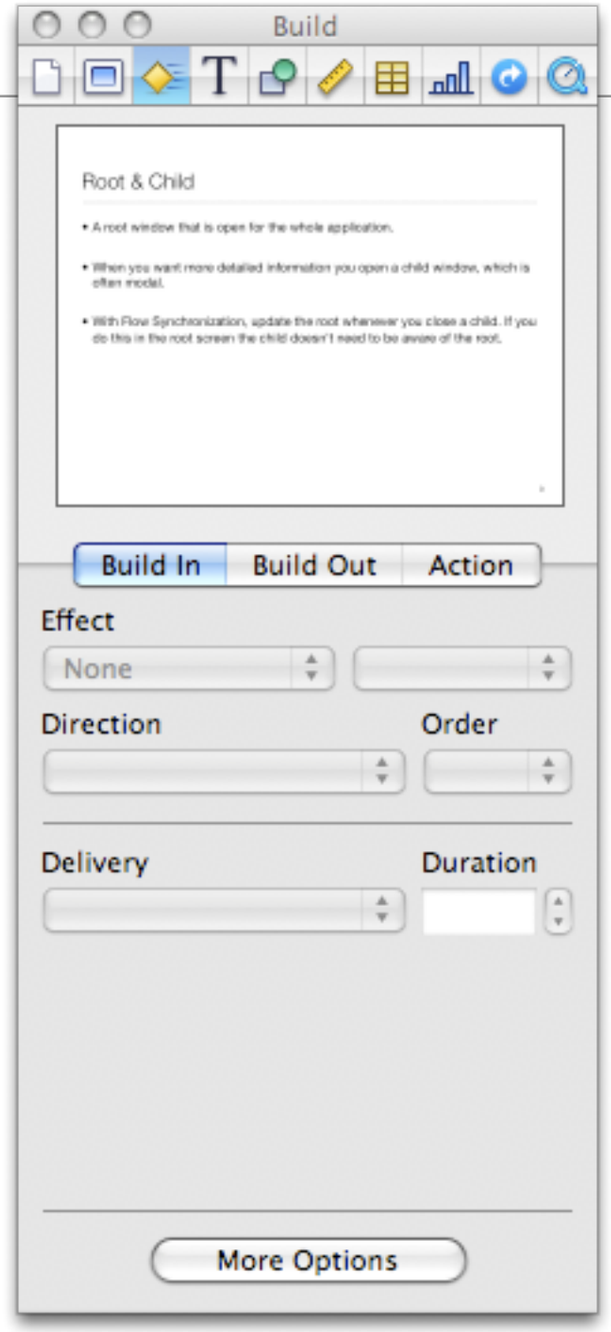
- Flow Synchronisation
- Observer Synchronisation

Flow Synchronization

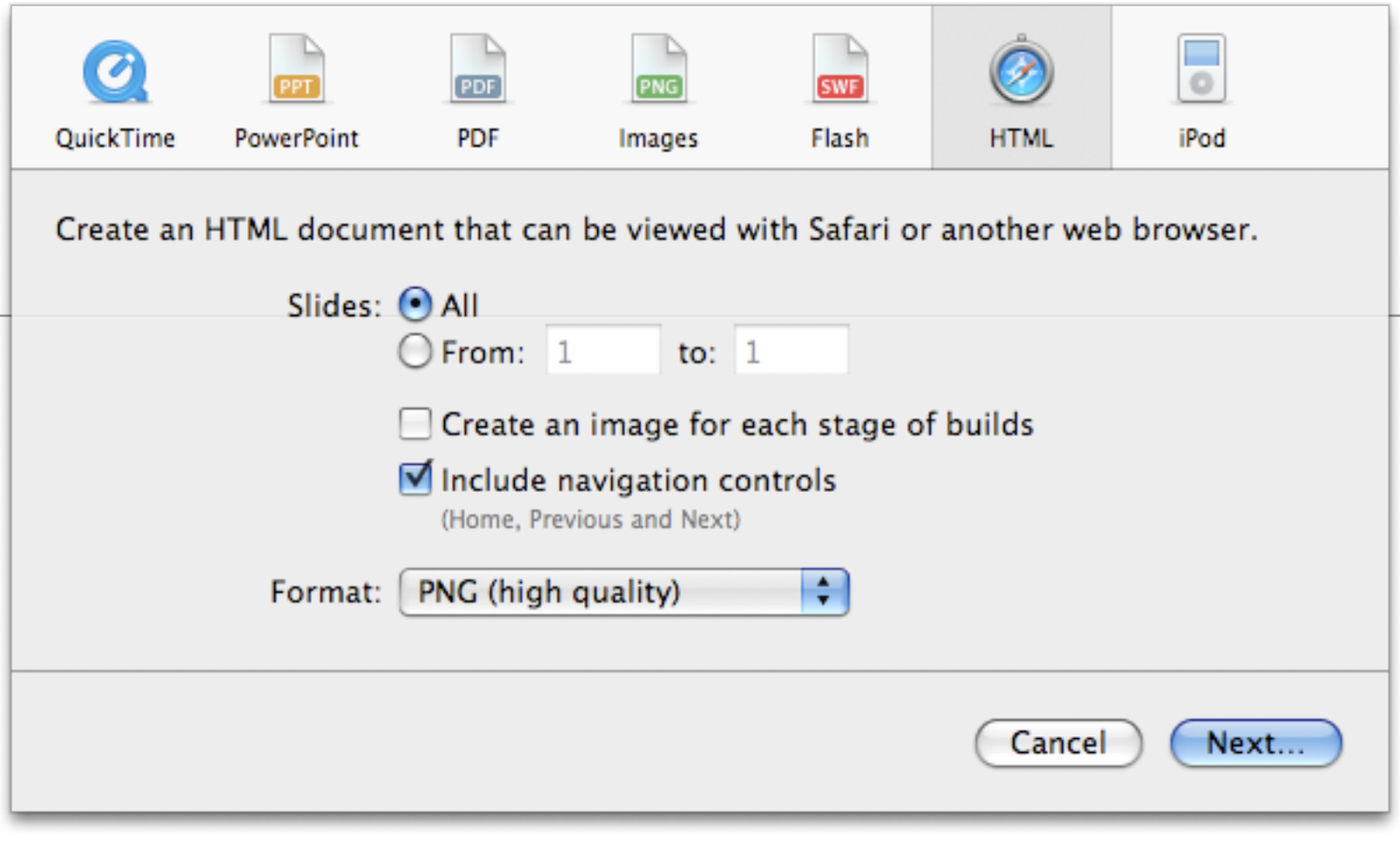
- Flow Synchronization is appropriate for simple navigation styles.
  - Each time you do something that changes state that's shared across multiple screens, you tell each screen to update itself.
  - Main problem: every screen is somewhat coupled to the other the other screens in the application.
- Two Common Styles UI where FlowSynchronization is appropriate:
  - Root & Child
  - Wizard

Root & Child

- A root window that is open for the whole application.
- When you want more detailed information you open a child window, which is often modal.
- With Flow Synchronization, update the root whenever you close a child. If you do this in the root screen the child doesn't need to be aware of the root.



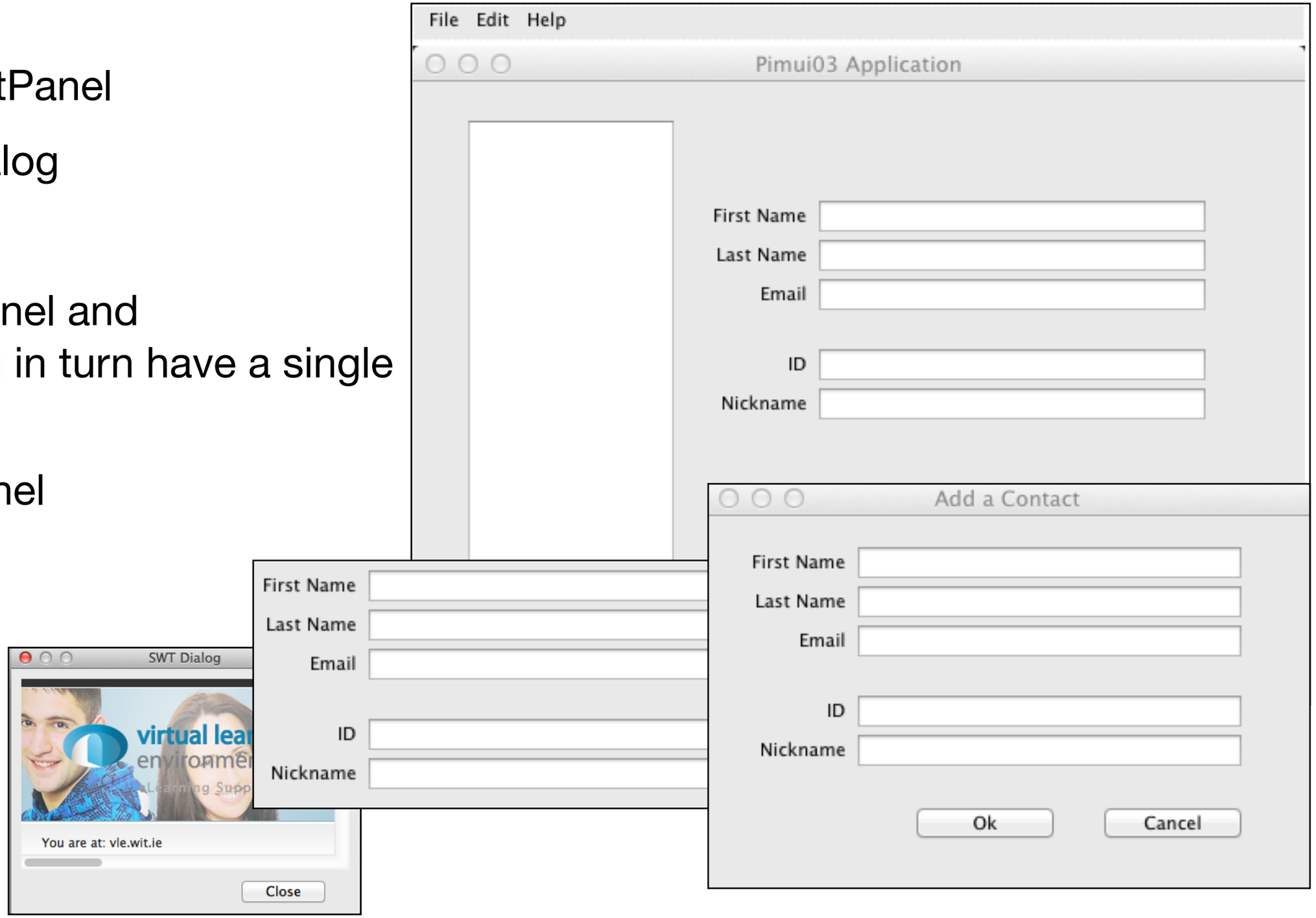
Wizard



- A sequence of screens in order.
- Each screen is modal and movement from one screen to another involved closing the old screen and opening another.
- Flow Synchronization:
  - update the data on closing a screen and load the fresh data when opening the next screen.
  - also possible to have a wizard sequence of modal children with a root screen where closing the last child updates the root.

Flow Synchronization in a Desktop App

- Main Window is the Root with three children:
  - ContactListPanel
  - ContactDialog
  - About
- ContactListPanel and ContactDialog in turn have a single child:
  - ContactPanel



Pacemaker

Login

Sign up

Signup

Sign up for the Pacemaker

Enter details below

First name

Last Name

Email

Password

Register

ActivitiesList

Activities

walk	fridge	20.0
walk	tv	19.0

Login

Login to Donation

You must be reigstered

Email

Password

Sign in

pacemaker

Enter Activity Details

walk

tv

Distance

18

19

20

Create Activity

Flow Synchronisation in pacemaker-android

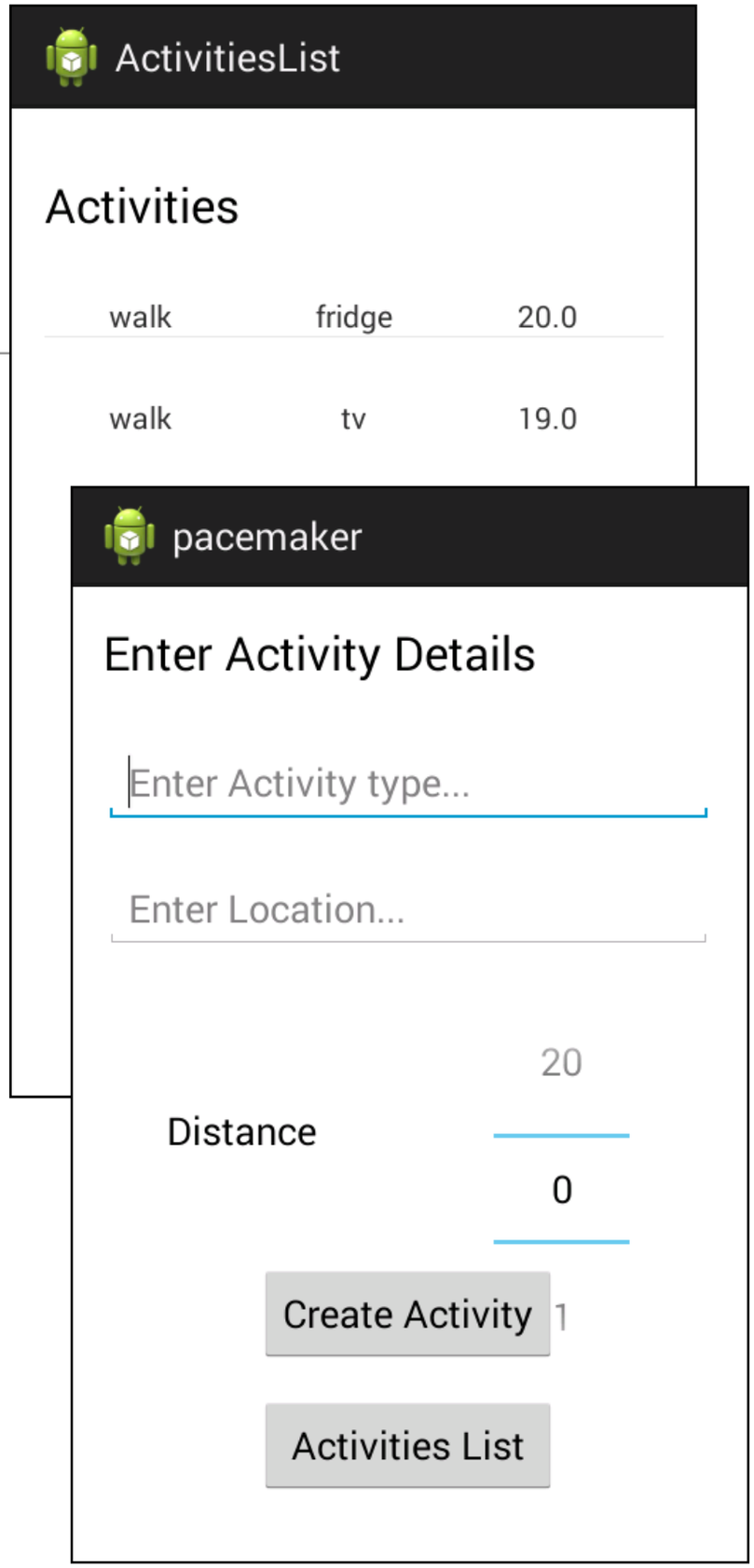
```
public class CreateActivity extends android.app.Activity
{
  private Button         createActivityButton;
  private TextView       activityType;
  private TextView       activityLocation;
  private NumberPicker   distancePicker;
  private ArrayList<Activity> activities = new ArrayList<Activity>();

  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    //…
  }

  public void createActivityButtonPressed (View view)
  {
    double distance = distancePicker.getValue();
    Activity activity
        = new Activity (activityType.getText().toString(),
                        activityLocation.getText().toString(), distance);

    activities.add(activity);
    Log.v("Pacemaker", "CreateActivity Button Pressed with " + distance);
  }
```

```
public void listActivityButtonPressed (View view)
{
  Log.v("Pacemaker", "List Activityies Button Pressed");
  Intent intent = new Intent(this, ActivitiesList.class);
  Bundle bundle = new Bundle();
  bundle.putParcelableArrayList("activities", activities);
  intent.putExtras(bundle);
  startActivity (intent);
}
```



Flow Synchronization : When to Use it

- With Flow Synchronization orchestrating views can become complex if there are an unconstrained number of windows associated with a model.
- Any change on any screen needs to trigger other screens to update. Doing this through explicit calls to other screens makes screens very interdependent.

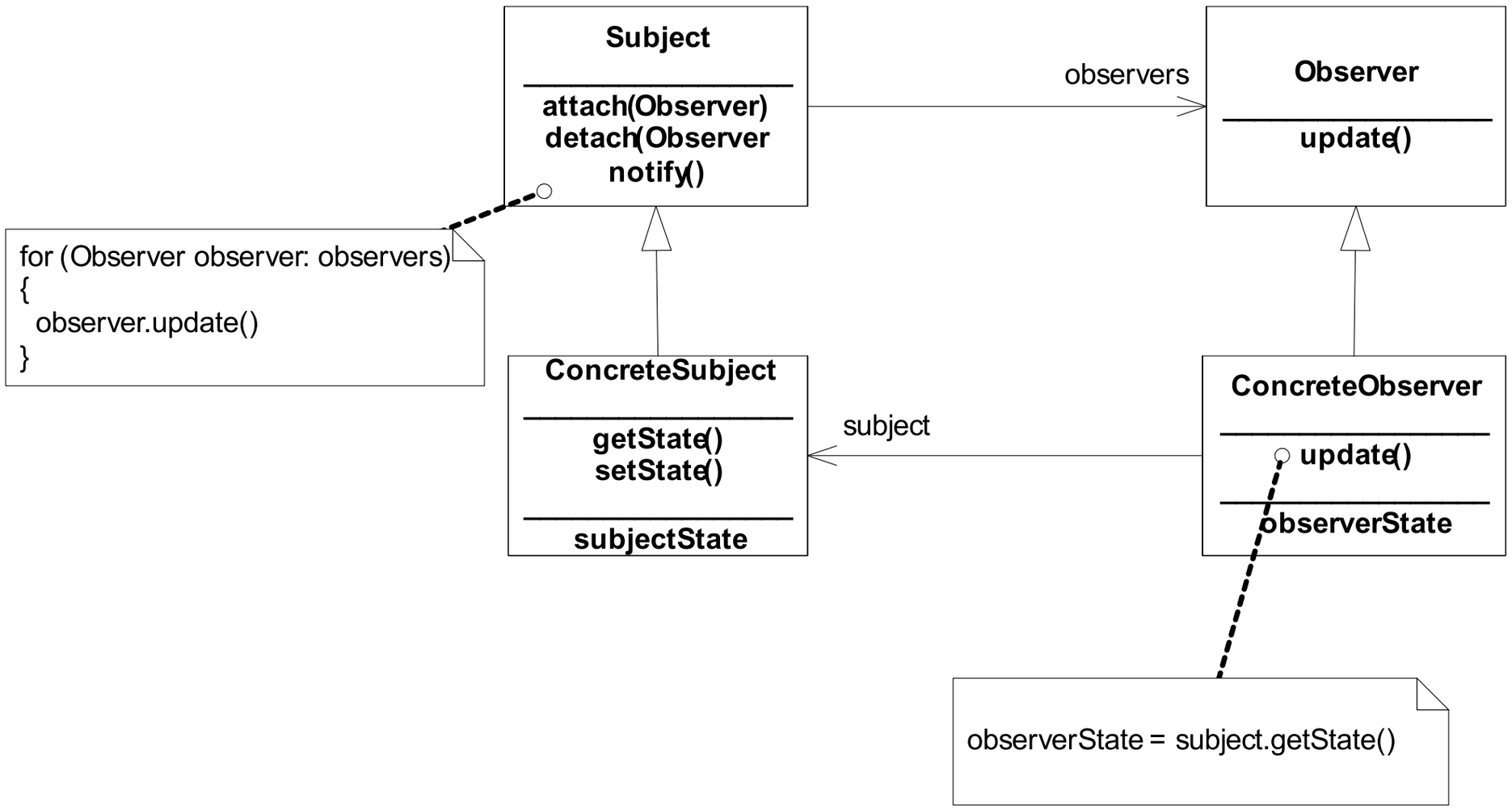
- Despite its limitations, Flow Synchronization does have its place. It works well providing the navigational flow of the user interface is simple and only one or two screen are active at the same time.
- Web user interfaces are effectively a sequence of screens and thus work effectively with Flow Synchronization.

Observer Synchronisation

- An application with multiple screens may display presentations of a common model.
- If a change is made to the model through one of these screens, all of the other screens must update correctly.
- However you don't want each screen to know about the others, because that would increase the complexity of the screens and make it harder to add new ones.
- Observer Synchronization allows the screens to “observe” a model, and have the model propagate changes to interested screens.
- Any change in one screen propagates to that domain oriented data and thence to the other screens.

Observer Pattern

- Have objects (Observers) that want to know when an event happens, attach themselves to another object (Subject) that is actually looking for it to occur.
- When the event occurs, the subject tells the observers that it occurred.



Observer Pattern

- Each screen, with its associated screen state, acts as an Observer on a common area of session data.

- All changes to the session data result in events which the screens listen to and respond by reloading from the session data.

- Once you have this mechanism set up, then you can ensure synchronization simply by coding each screen to update the session data.

- Even the screen that makes the change doesn't need to refresh itself explicitly, since the observer mechanism will trigger the refresh in the same way as if another screen made the change.

Granularity

- The biggest issue in this design is deciding what granularity of events to use and how to set up the propagating and observer relationships.
- At the very fine-grained level each bit of domain data can have separate events to indicate exactly what changed. Each screen registers for only the events that may invalidate that screen's data.
- The most coarse grained alternative is to use an Event Aggregator to funnel all events into a single channel. That way each screen does a reload if any piece of domain data changes, whether or not it would have affected the screen.
- The trade-off is between complexity and performance:
  - The coarse-grained approach is much simpler to set up and less likely to breed bugs (apart from serialization!)
  - However it leads to lots of unnecessary refreshes of the screen, which might impact performance.

Fowler Advice

- Start coarse grained and introduce appropriate fine-grained mechanisms only when needed after measuring an actual performance problem
- Events tend to be hard to debug since you can't see the chain of invocations by looking at the code.
  - As a result it's important to keep the event propagation mechanism as simple as you can, which is why Fowler favours coarse-grained mechanisms.
  - A good rule of thumb is to think of your objects as layered and to only allow observer relationships between layers.
  - So one domain object should not observe another domain object, only presentation objects should observe domain objects.

Fine-grained Approach: PropertyChangeSupport

java.beans

Class PropertyChangeSupport

java.lang.Object
  └ java.beans.PropertyChangeSupport

All Implemented Interfaces:
Serializable

Direct Known Subclasses:
SwingPropertyChangeSupport

```
public class PropertyChangeSupport
extends Object
implements Serializable
```

This is a utility class that can be used by beans that support bound properties. You can use an instance of this class as a member field of your bean and delegate various work to it. This class is serializable. When it is serialized it will save (and restore) any listeners that are themselves serializable. Any non-serializable listeners will be skipped during serialization.

See Also:
Serialized Form

PropertyChangeEvent

java.beans

Class PropertyChangeEvent

java.lang.Object
  └ java.util.EventObject
      └ **java.beans.PropertyChangeEvent**

All Implemented Interfaces:
Serializable

---

public class **PropertyChangeEvent**
extends EventObject

A "PropertyChange" event gets delivered whenever a bean changes a "bound" or "constrained" property. A PropertyChangeEvent object is sent as an argument to the PropertyChangeListener and VetoableChangeListener methods.

Normally PropertyChangeEvents are accompanied by the name and the old and new value of the changed property. If the new value is a primitive type (such as int or boolean) it must be wrapped as the corresponding java.lang.* Object type (such as Integer or Boolean).

Null values may be provided for the old and the new values if their true values are not known.

An event source may send a null object as the name to indicate that an arbitrary set of if its properties have changed. In this case the old and new values should also be null.

firePropertyChange

firePropertyChange

`public void` **`firePropertyChange`**`(PropertyChangeEvent evt)`

Fire an existing PropertyChangeEvent to any registered listeners. No event is fired if the given event's old and new values are equal and non-null.

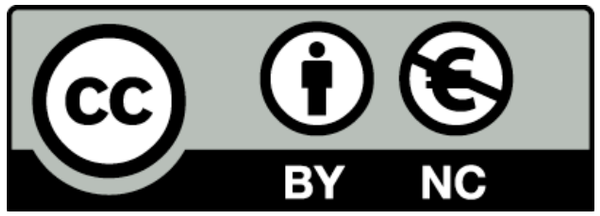
Parameters:

`evt` - The PropertyChangeEvent object.

Constructor Summary

`PropertyChangeEvent``(Object source, String propertyName, Object oldValue, Object newValue)`

Constructs a new `PropertyChangeEvent`.



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit