# 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

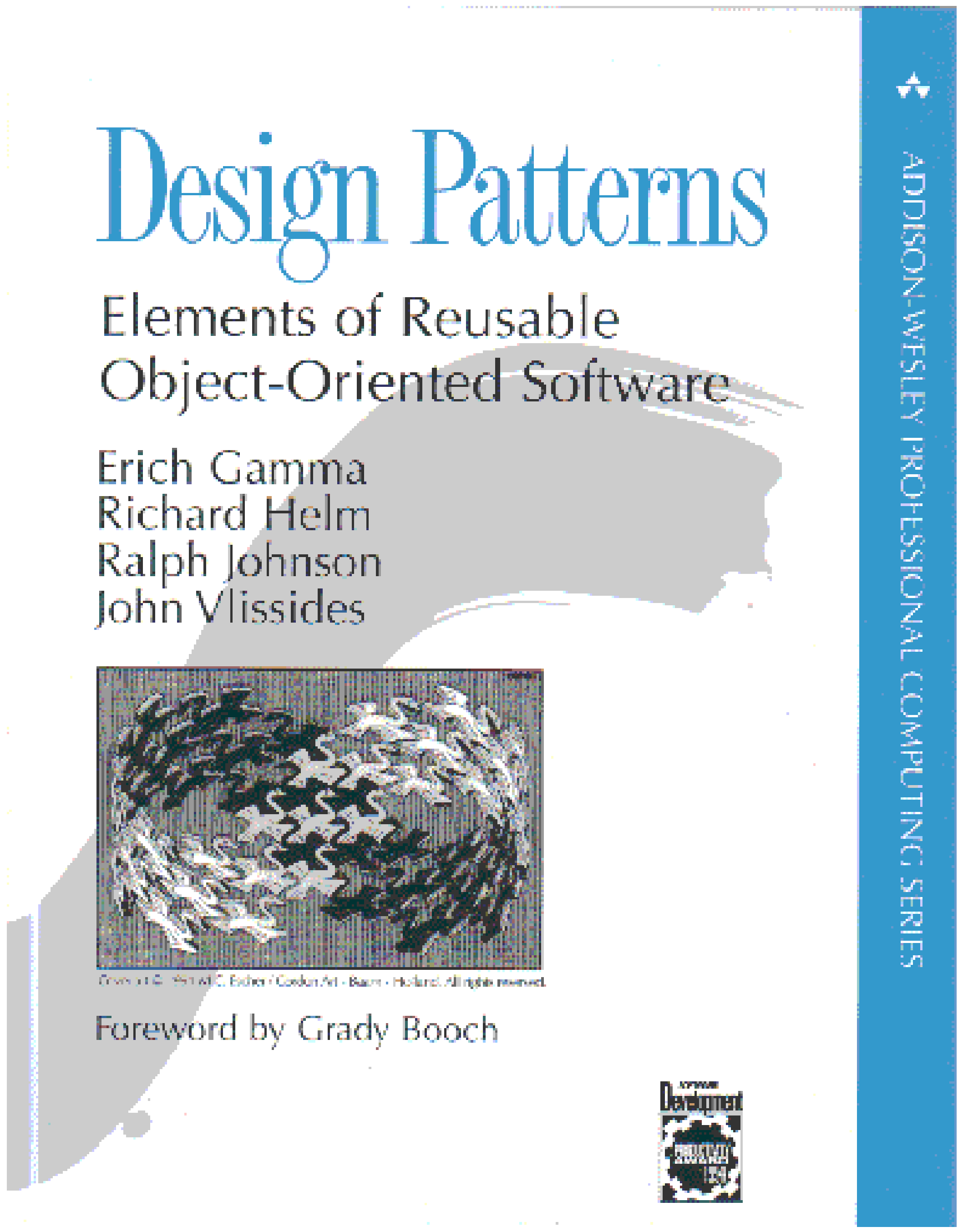
eLearning support unit

# Patterns for Graphic User Interface / Rich Client Development

Sources & Context

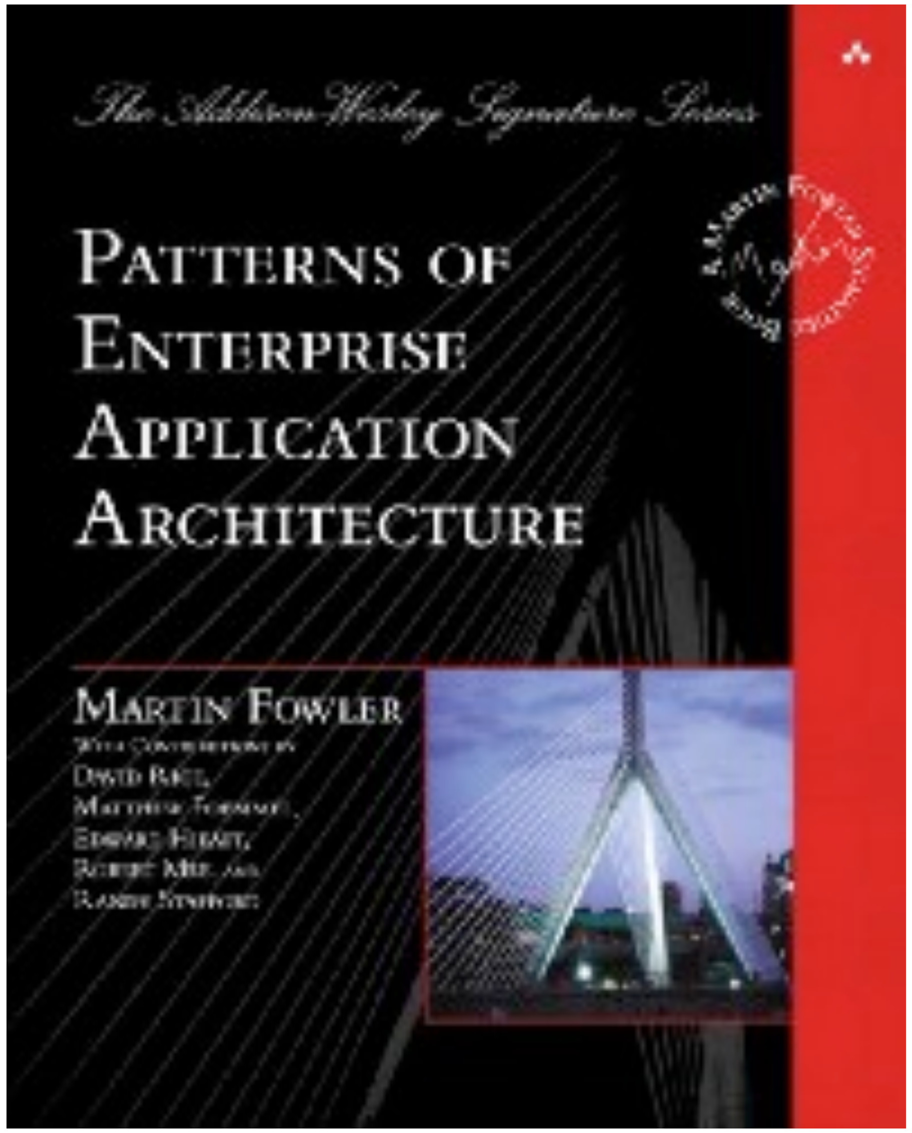
## Source: GoF

- So far - patterns have been part of the 'Classical' GoF reference.
- Canonical form - much referenced.
- Has had a strong influence on Java and on the JDK.
- "First generation" approach to GUIs


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

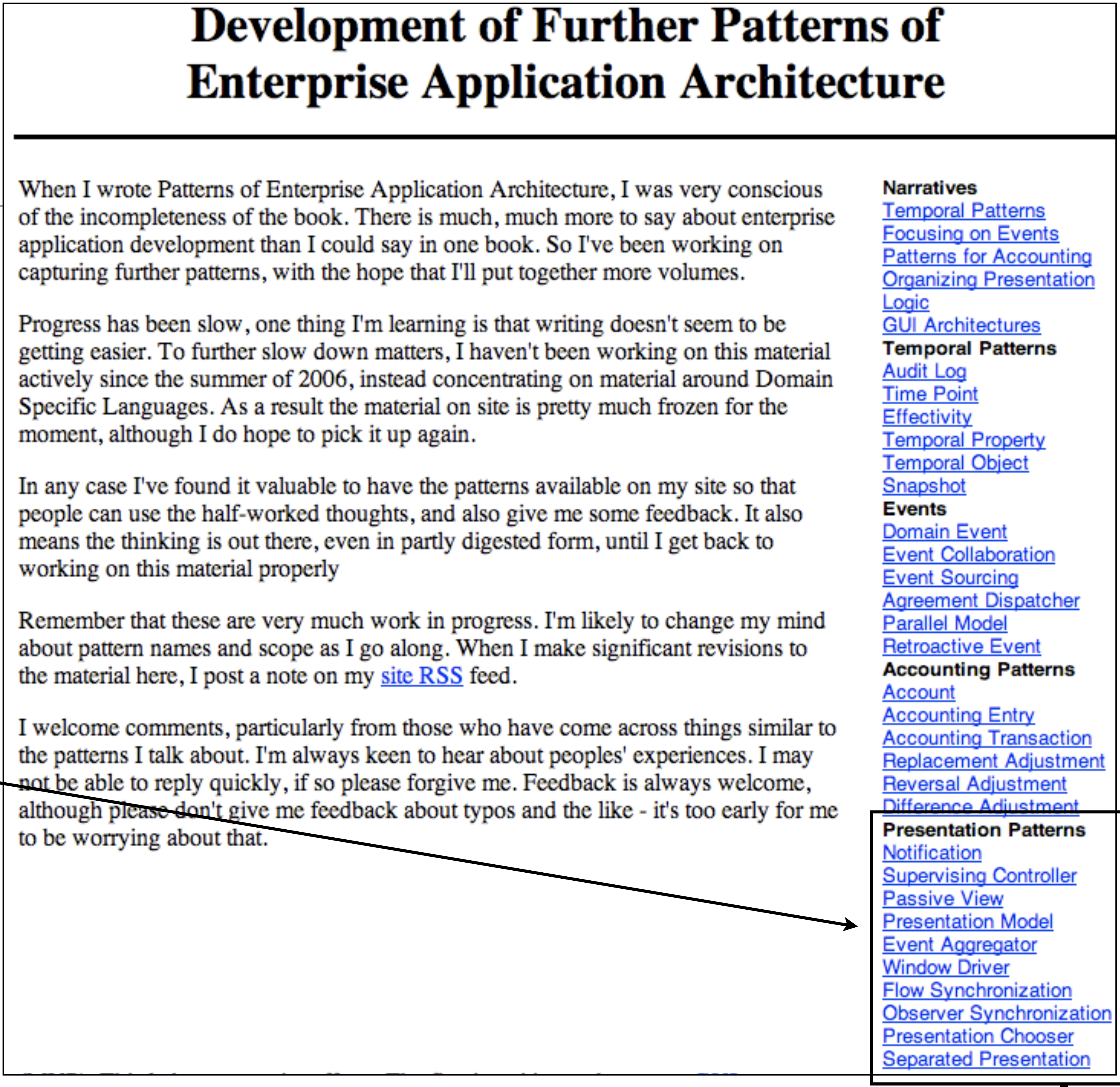
## Source: PoEAA: Patterns of Enterprise Application Development

- Catalogue of patterns for “Enterprise” development.
- Topics
  - how to layer an enterprise application
  - how to organize domain logic
  - how to tie that logic to a relational database,
  - how to design a web based presentation
- http://martinfowler.com/


The Addison-Wesley Signature Series
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
MARTIN FOWLER

# Eaa Work in Progress

- A web-only resource continuing to document further “enterprise” patterns.
- Considers GUI development patterns not discussed in P of Eaa text.



## Development of Further Patterns of Enterprise Application Architecture

When I wrote Patterns of Enterprise Application Architecture, I was very conscious of the incompleteness of the book. There is much, much more to say about enterprise application development than I could say in one book. So I've been working on capturing further patterns, with the hope that I'll put together more volumes.

Progress has been slow, one thing I'm learning is that writing doesn't seem to be getting easier. To further slow down matters, I haven't been working on this material actively since the summer of 2006, instead concentrating on material around Domain Specific Languages. As a result the material on site is pretty much frozen for the moment, although I do hope to pick it up again.

In any case I've found it valuable to have the patterns available on my site so that people can use the half-worked thoughts, and also give me some feedback. It also means the thinking is out there, even in partly digested form, until I get back to working on this material properly

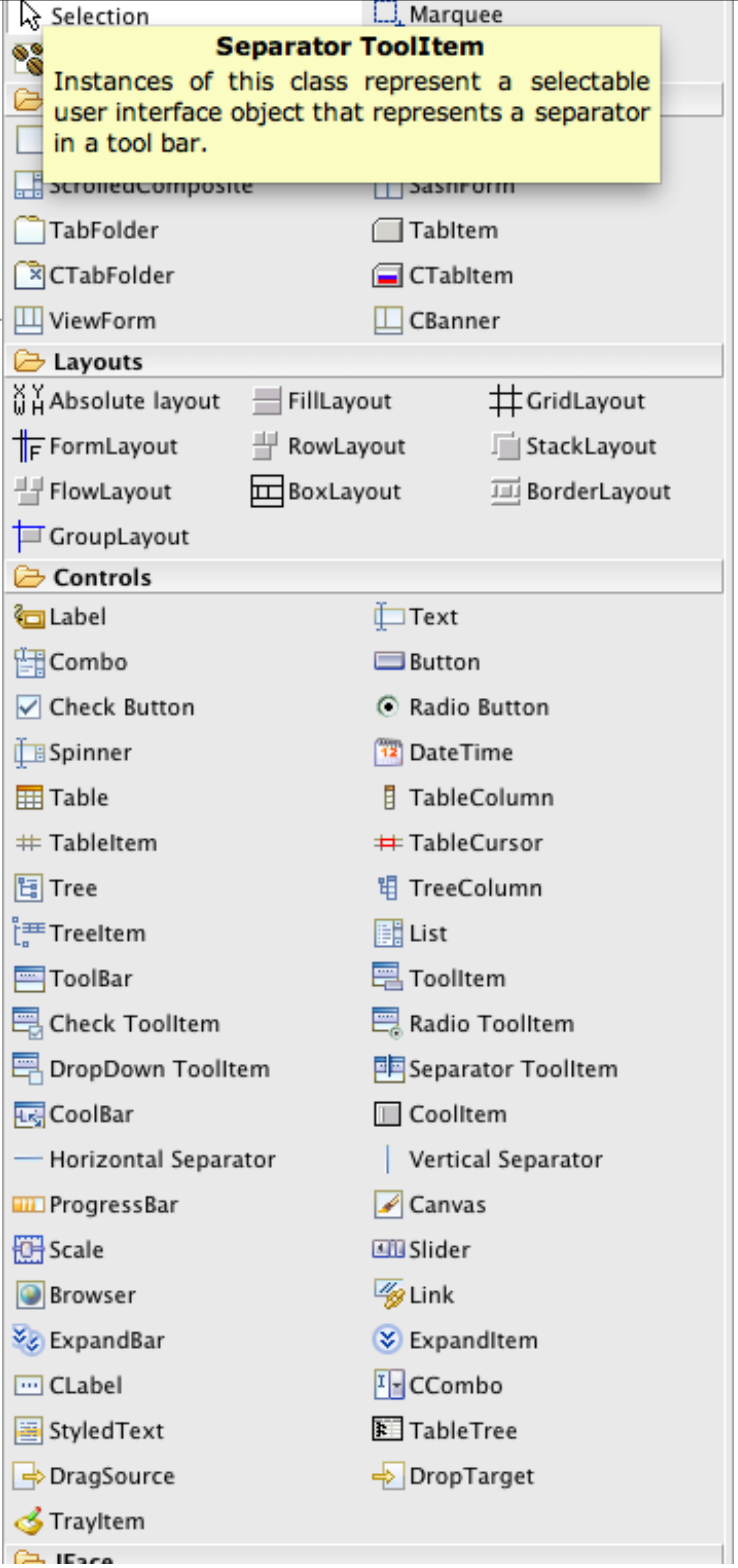
Remember that these are very much work in progress. I'm likely to change my mind about pattern names and scope as I go along. When I make significant revisions to the material here, I post a note on my site RSS feed.

I welcome comments, particularly from those who have come across things similar to the patterns I talk about. I'm always keen to hear about peoples' experiences. I may not be able to reply quickly, if so please forgive me. Feedback is always welcome, although please don't give me feedback about typos and the like - it's too early for me to be worrying about that.

**Narratives**
Temporal Patterns
Focusing on Events
Patterns for Accounting
Organizing Presentation Logic
GUI Architectures
**Temporal Patterns**
Audit Log
Time Point
Effectivity
Temporal Property
Temporal Object
Snapshot
**Events**
Domain Event
Event Collaboration
Event Sourcing
Agreement Dispatcher
Parallel Model
Retroactive Event
**Accounting Patterns**
Account
Accounting Entry
Accounting Transaction
Replacement Adjustment
Reversal Adjustment
Difference Adjustment
**Presentation Patterns**
Notification
Supervising Controller
Passive View
Presentation Model
Event Aggregator
Window Driver
Flow Synchronization
Observer Synchronization
Presentation Chooser
Separated Presentation

## Graphical User Interfaces

- GUIs - often called Rich Clients - can be notoriously complex.
- Inherent complexity: the GUI component set is at varying levels of abstraction with sophisticated event mechanisms:
  - Controls
  - Containers
  - Windows
  - Menus
- Accidental complexity: domain logic can easily become hopelessly intermingled with the GUI specific logic.

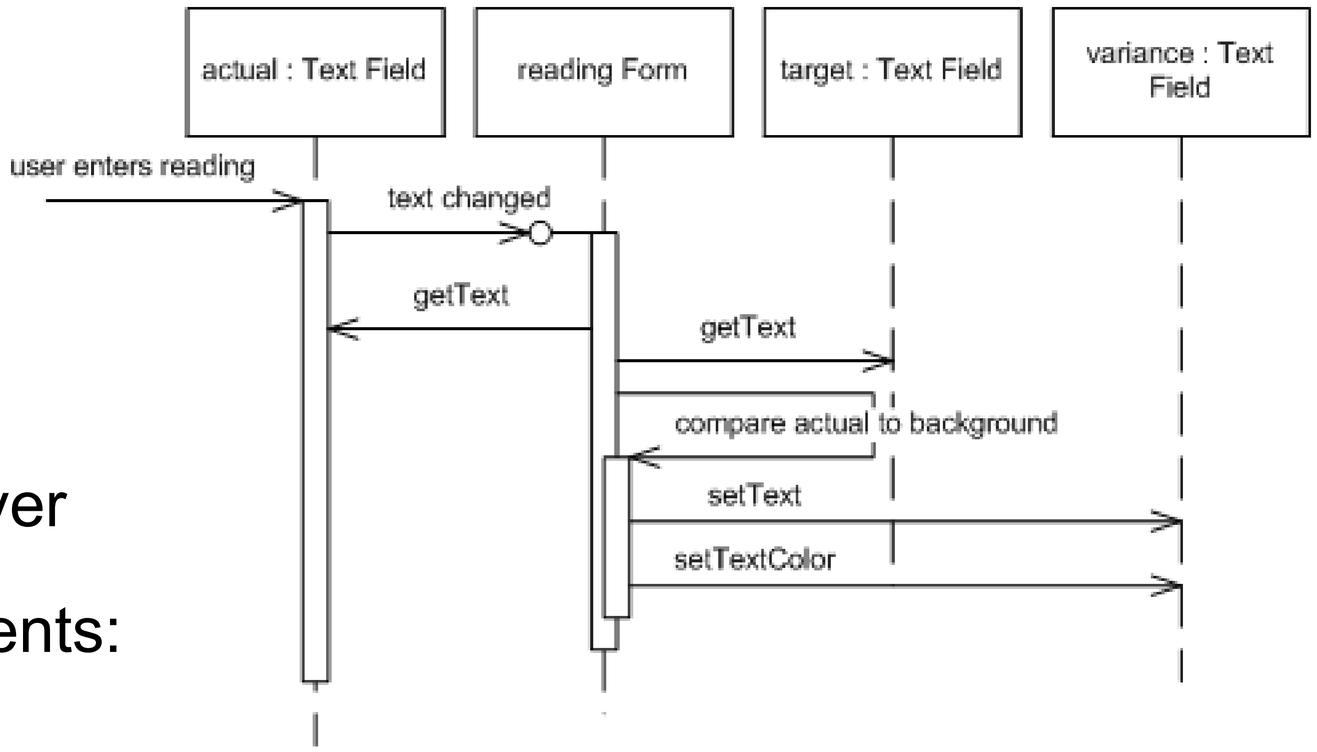


**Separator ToolItem**

Instances of this class represent a selectable user interface object that represents a separator in a tool bar.

## GUI Events

- A significant source of complexity
- Toolkit handles fine-grained events
  - Mouse entered, exited
  - Mouse pressed
  - Radio button pressed, armed, rollover
- Application handles coarse-grained events:
  - Radio button selected
  - Action performed
  - Domain property changed
- Managing the flow of these events requires careful consideration if design coherence is to be preserved.



## Patterns

- GoF Patterns weak in this area - tend to be too fine grained.
- One “Composed Pattern” is discussed:
    - Model View Controller
- Eaa Work in Progress (http://martinfowler.com/eaaDev/)
    - Notification
    - Supervising Controller
    - Passive View
    - Presentation Model
    - Event Aggregator
    - Window Driver
    - Flow Synchronization
    - Observer Synchronization
    - Presentation Chooser
    - Autonomous View
    - Model View Presenter

In particular, read
http://martinfowler.com/eaaDev/uiArchs.html
for background to these patterns

## Selected Patterns

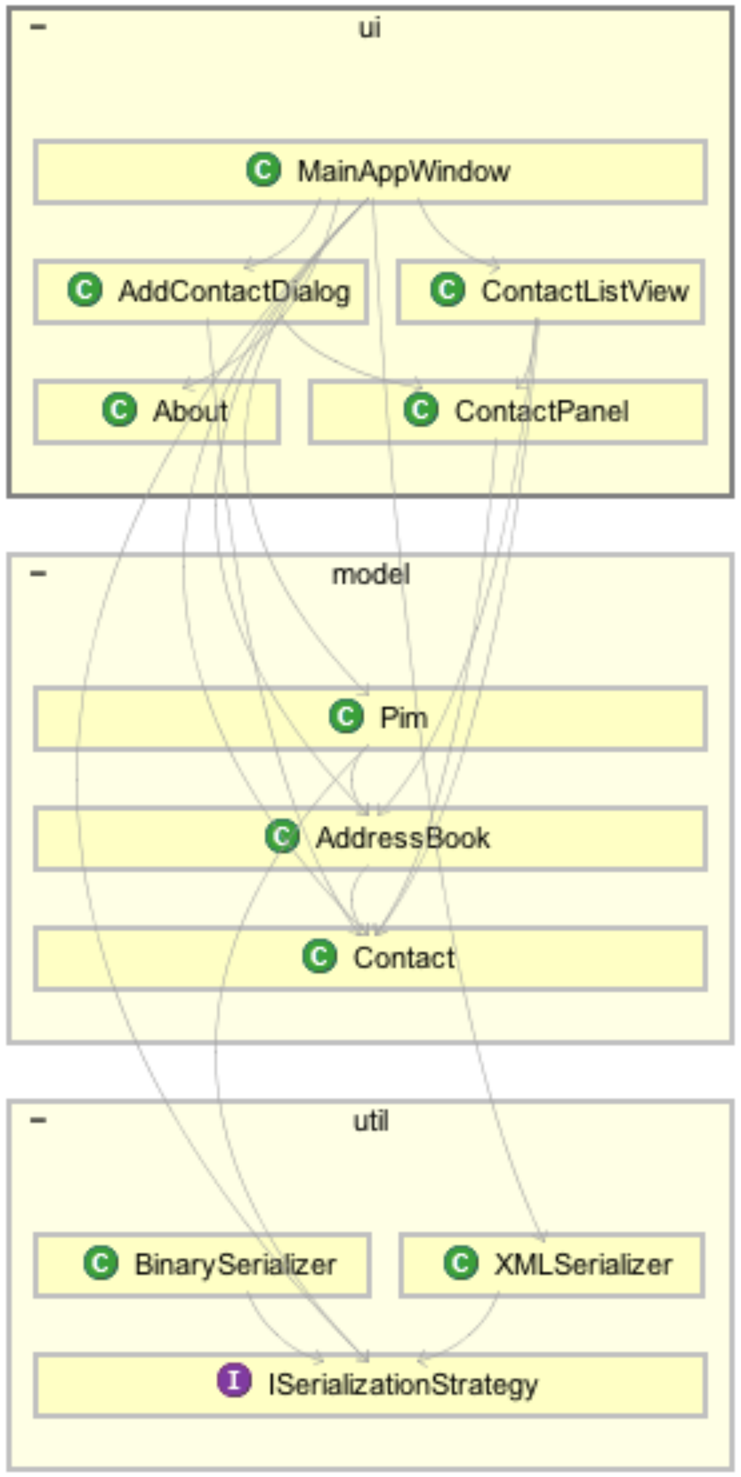
- Reasonable subset of fowler:
  - Separated Presentation
  - Flow Synchronization
  - Observer Synchronization
  - Passive View
  - Supervising Controller/Presenter
  - Model View Presenter
- + review Model View Controller in this context

## Separated Presentation

- A form of layering where presentation code and domain code in separate layers with the domain code unaware of presentation code.
    - Review all the data and behavior in a system identifying code involved in the presentation. Presentation code would manipulate GUI widgets and structures only.
    - We then divide the application into two logical modules with all the presentation code in one module and the rest in another module.
- The layers are a logical and not a physical construct. Java packages are a useful separation mechanism

# Separated Presentation - Dependencies

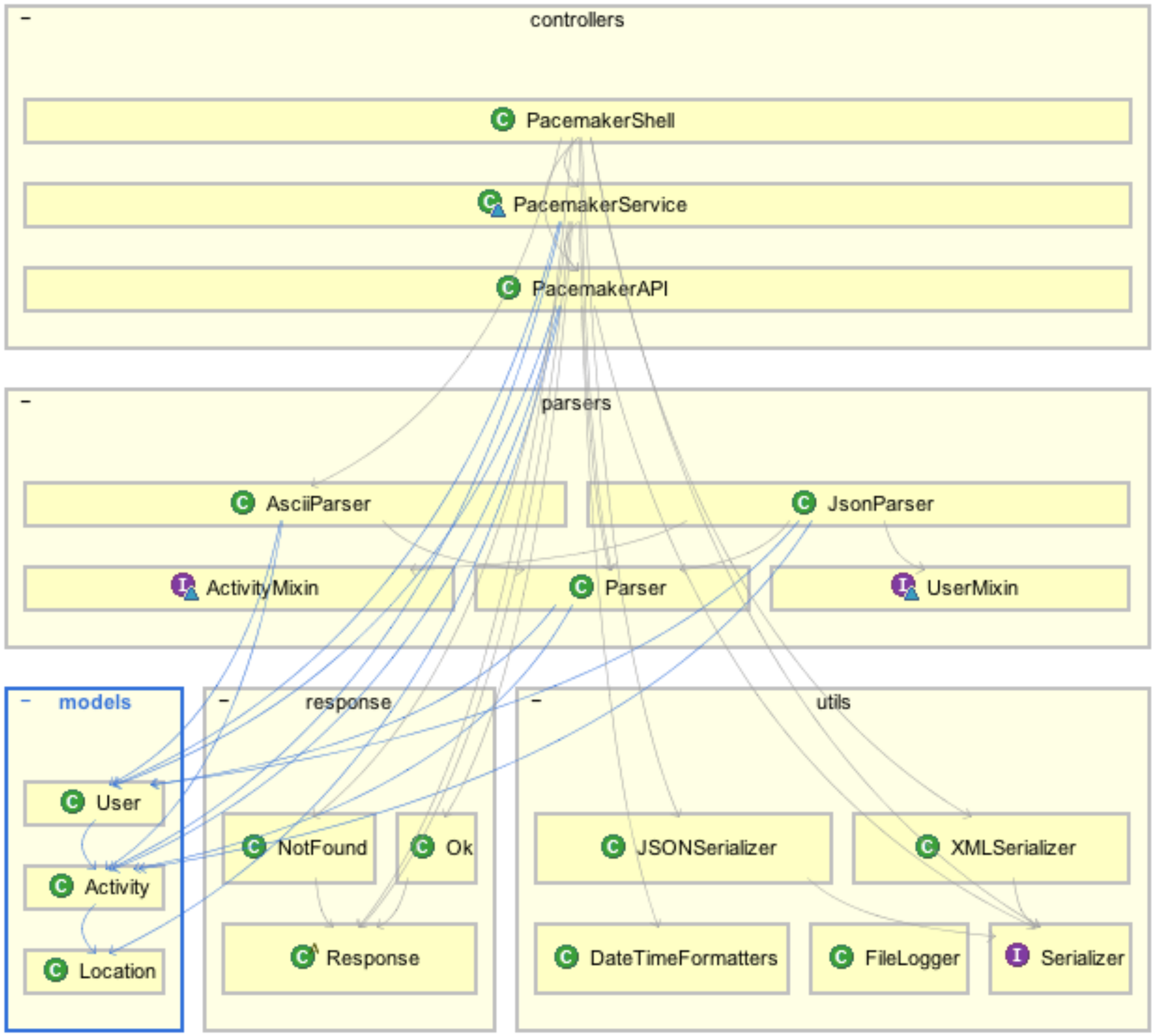
- Apply a strict visibility rule. The presentation is able to call the domain but not vice-versa.
    - This can be checked as part of a build with dependency checking tools. (Structure101)
- The domain should be utterly unaware of what presentations may be used with it.
- This both helps keep the concerns separate and also supports using multiple presentations with the same domain code.



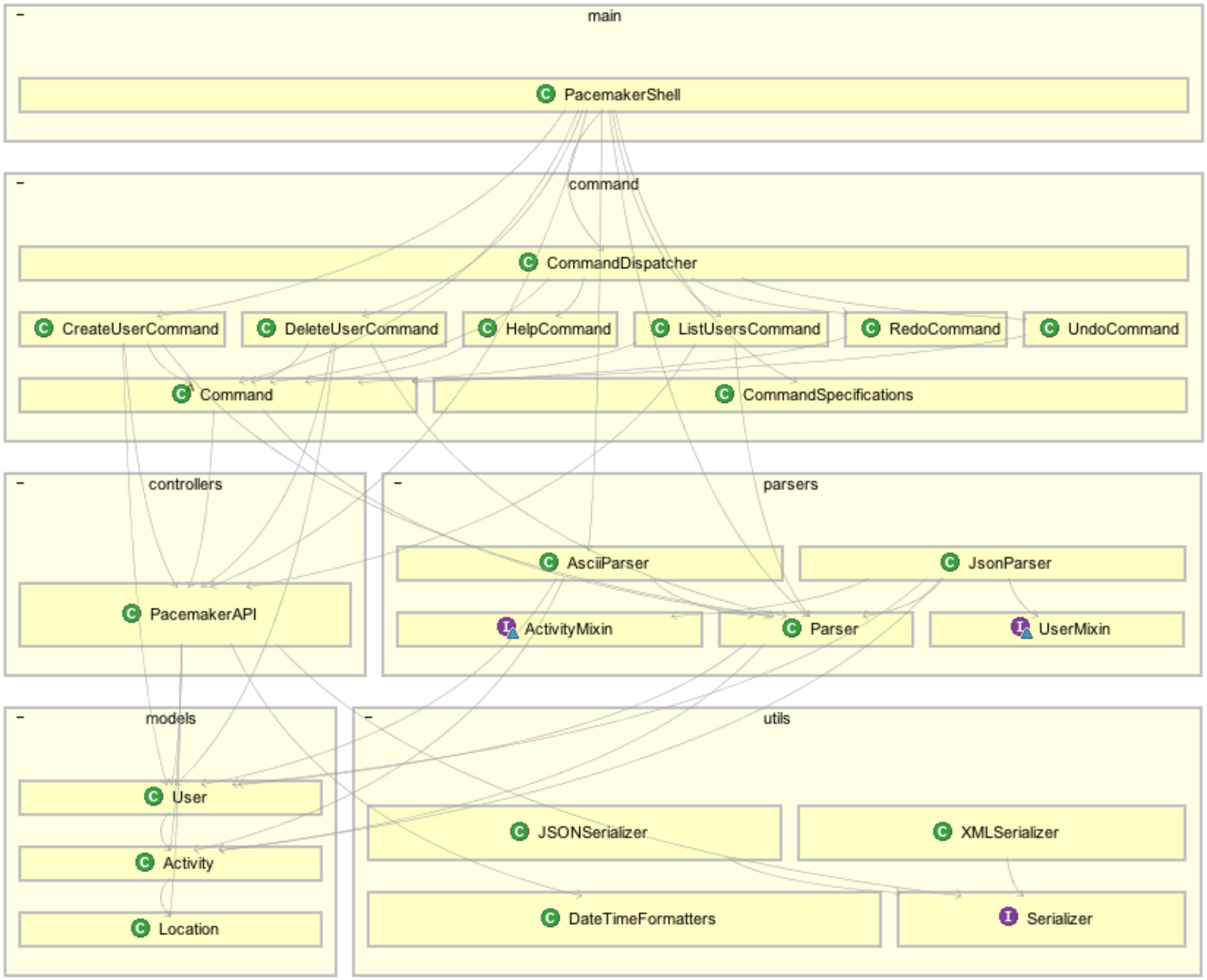
## Separated Presentation - UI Independent

- A good mental test to use to check you are using Separated Presentation is to imagine a completely different user interface.
    - If you are writing a GUI imagine writing a command line interface for the same application.
    - Ask yourself if anything would be duplicated between the GUI and command line presentation code - if it is then it's a good candidate for moving to the domain
- In Console & GUI version of PIM - the model package has been unchanged

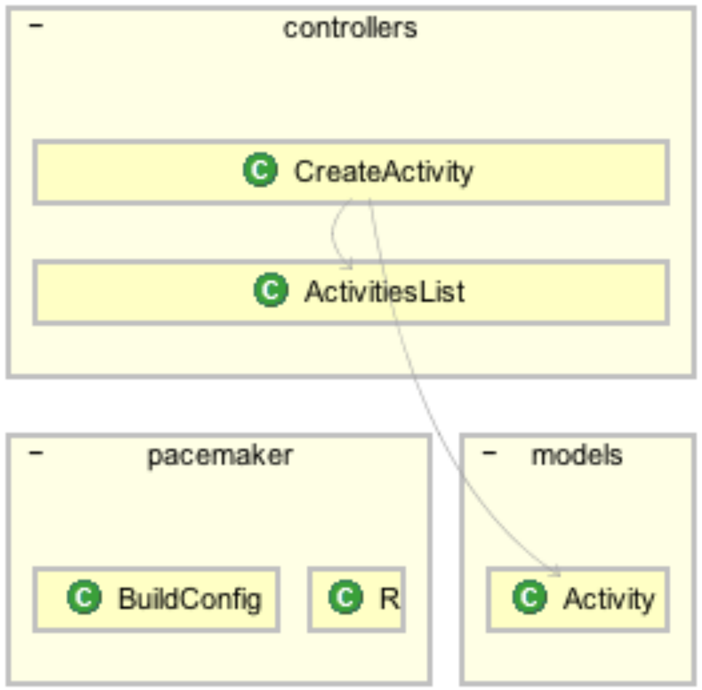
# pacemaker-console



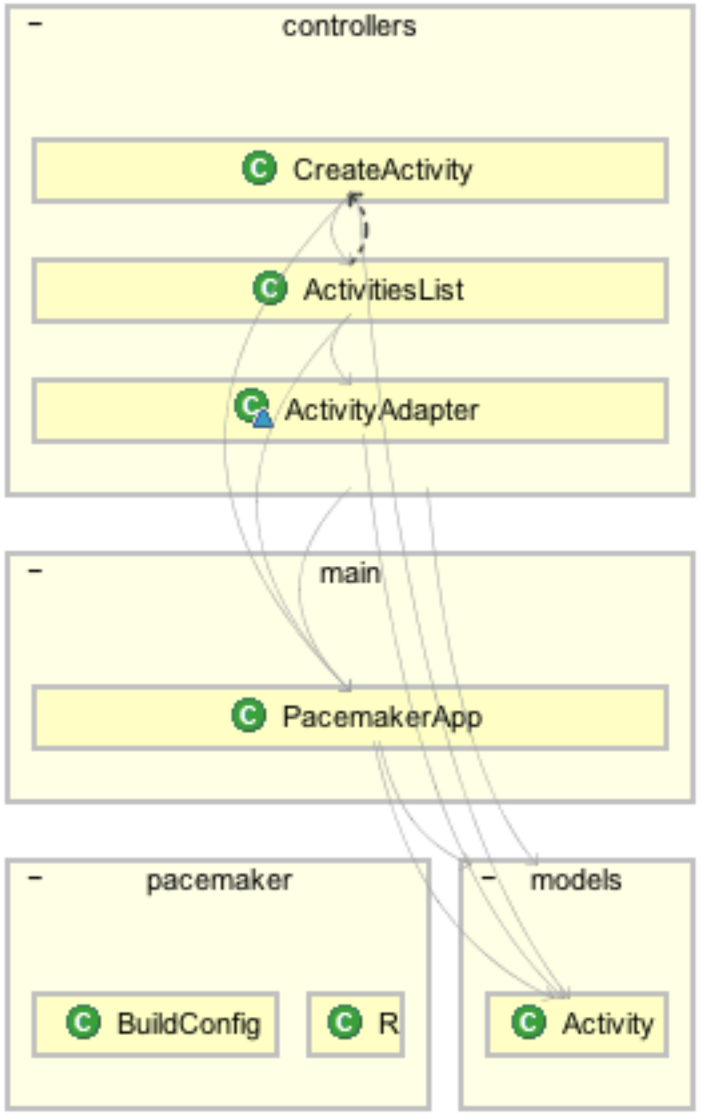
# pacemaker-console-command



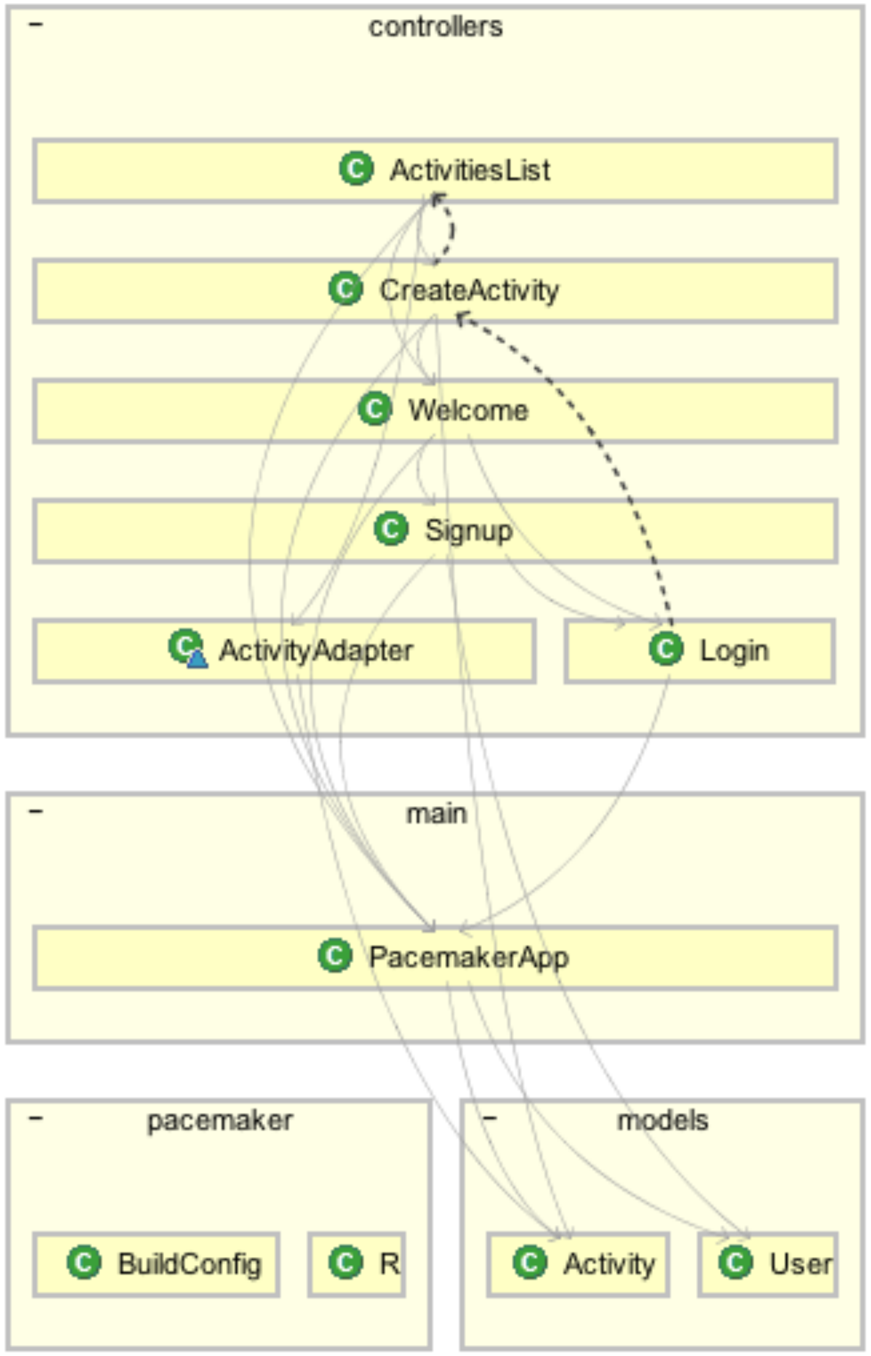
# pacemaker-android-v1



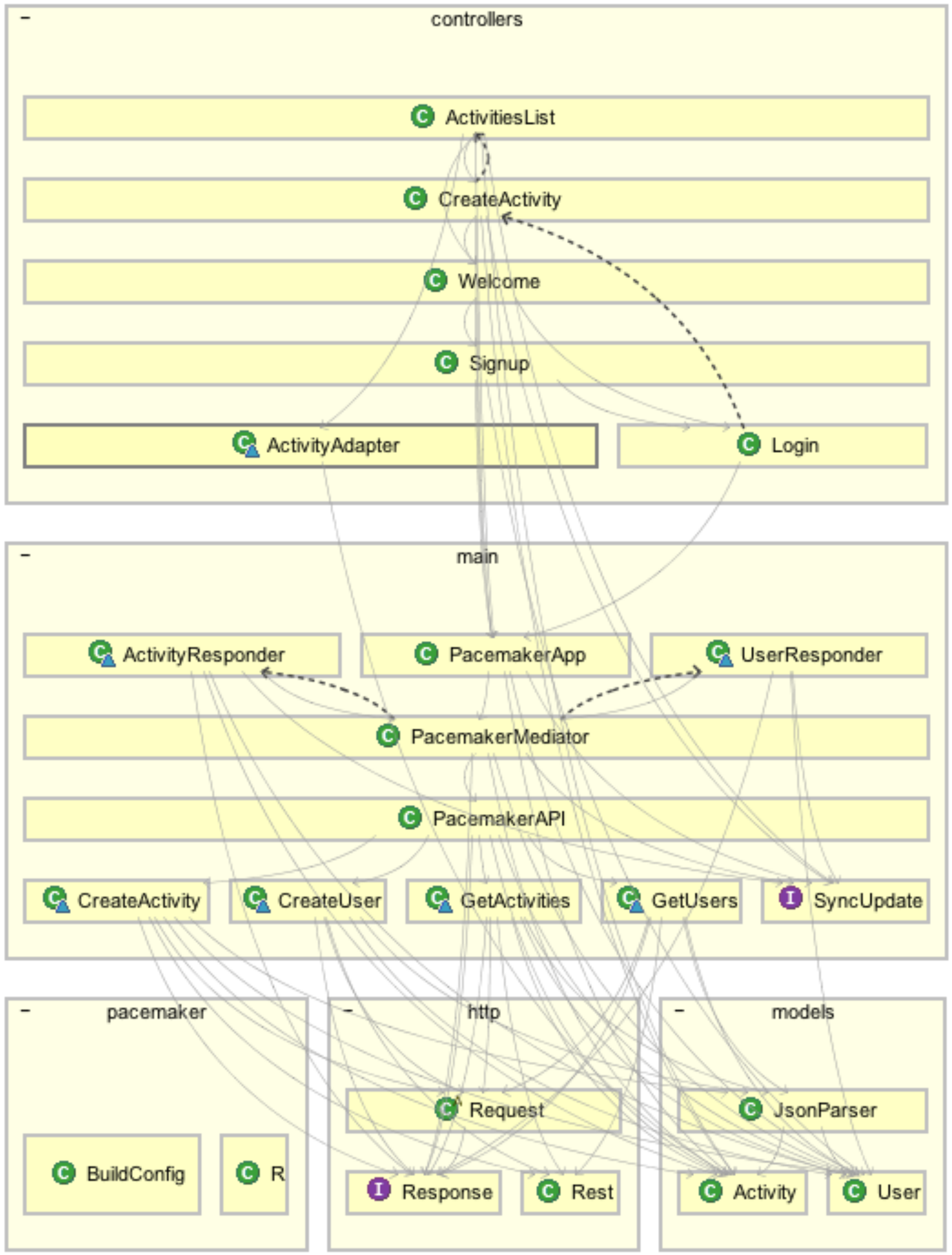
# pacemaker-android-v2



## pacemaker-android-v3

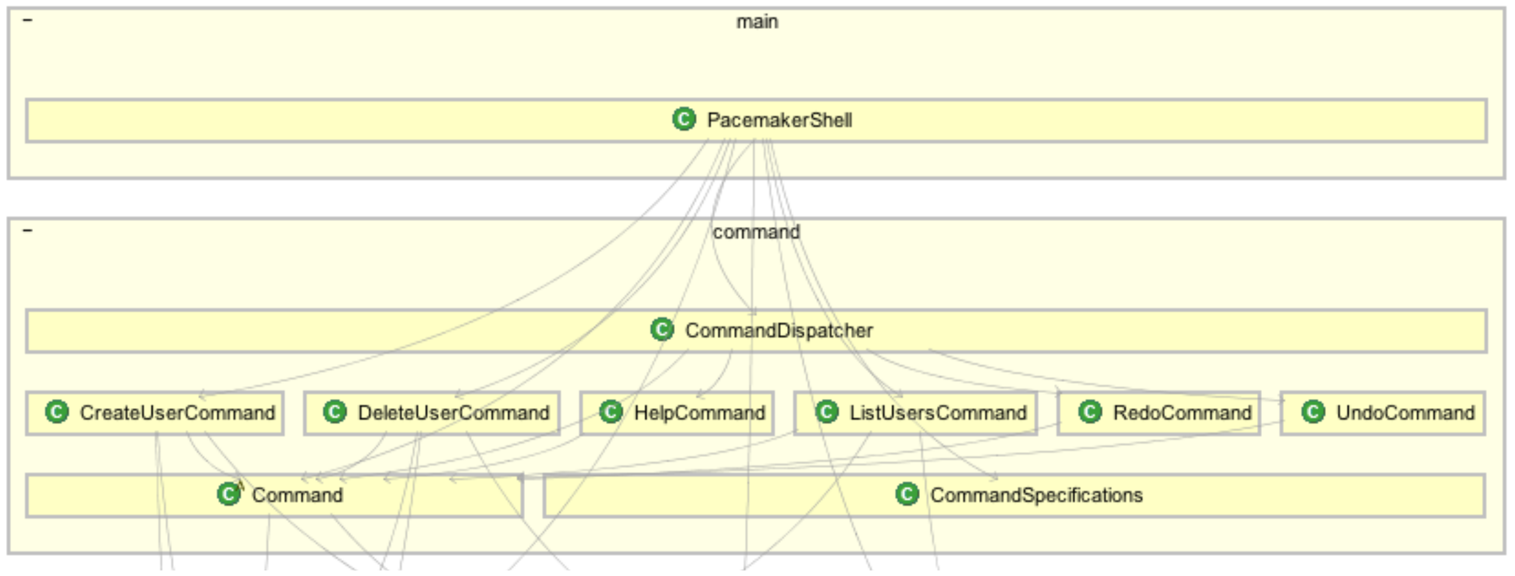


## pacemaker-android-v4

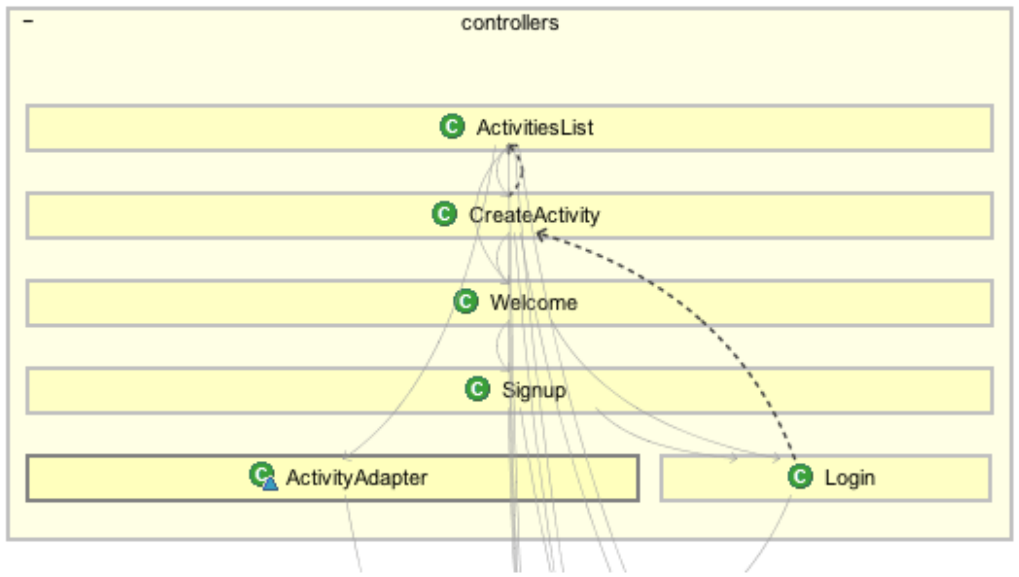


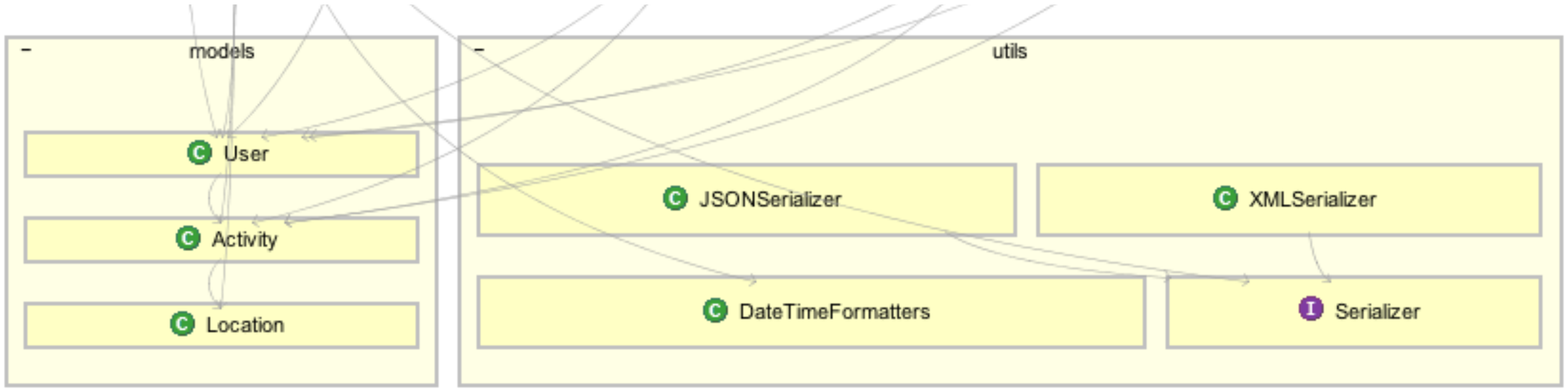
# Different UIs

Console



Android





Same Model + Persistence

## Separated Presentation - Synchronization

- It will be necessary for the domain to notify the presentation if any changes occur.
- 2 common solutions:
    - Flow Synchronization
    - Observer Synchronization
- Both of these solutions attempt will ensure that the model and the views are rendering the same information.
- However, both must ensure that Separated Presentation is preserved.