

# pide framework

Aras Bilgen  
Erman olak  
Mert İlhan  
Yaşar Tutuk  
Onur Vural

# Overview

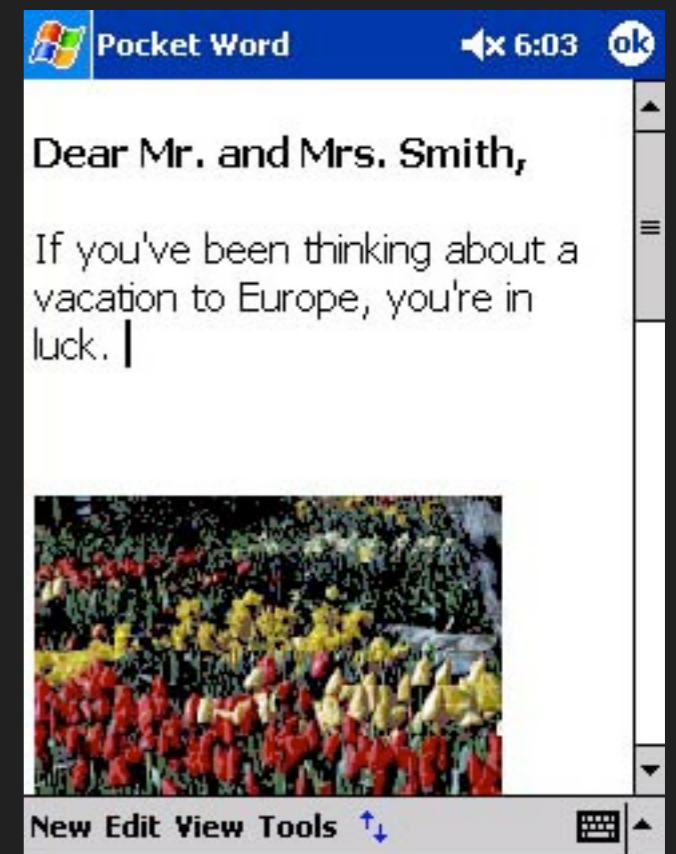
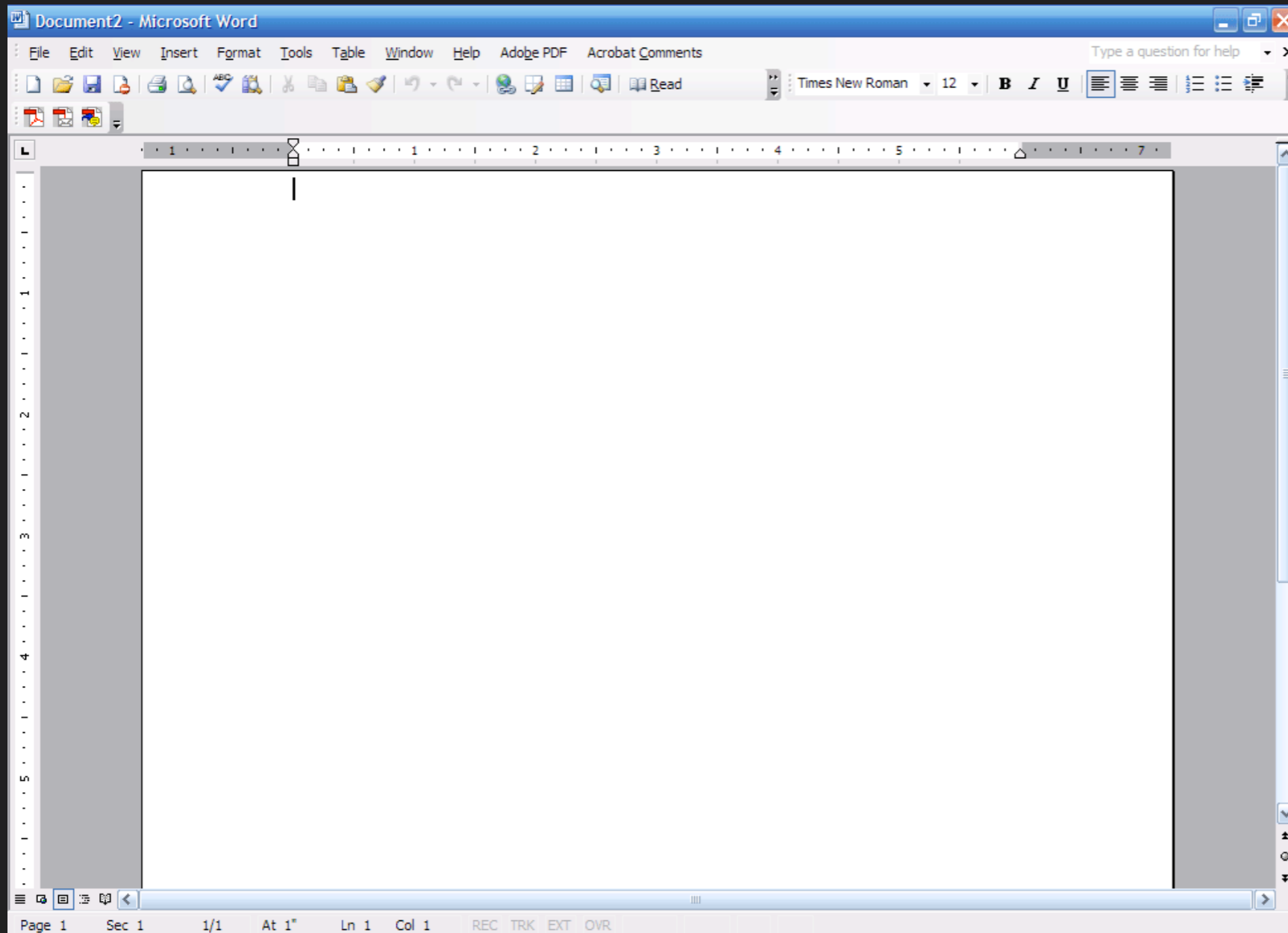
- Portable Interface Display Environment
  - An architecture/framework in which developers can develop device and interface independent applications
- Goal 1: To separate the interface from the application logic
- Goal 2: Provide tools and methods to implement this architecture

# What to expect?

- No GUI for the project
  - The programs you will see, except the web page, is done using the framework. What you see is their GUIs
- This is an architecture and the framework
  - Architecture: We came up with the idea, components, layout, protocols...
  - Framework: Reusable code we wrote to implement the architecture

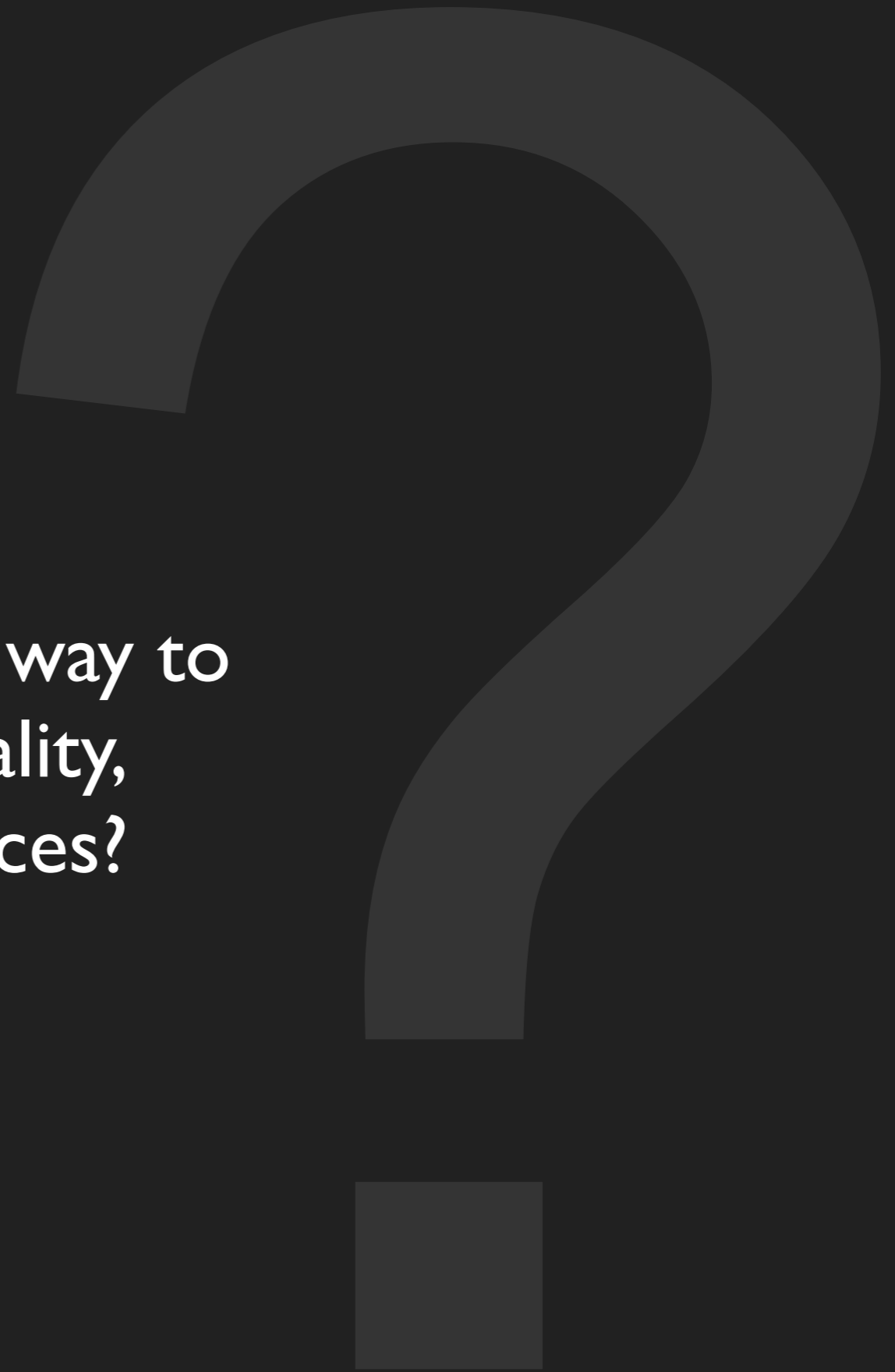
# Motivation

- Proliferation of new devices, new platforms, new modalities...
- They provide a different functionality, or they replicate an aspect of the desktop counterpart
  - Shopping list, PocketWord

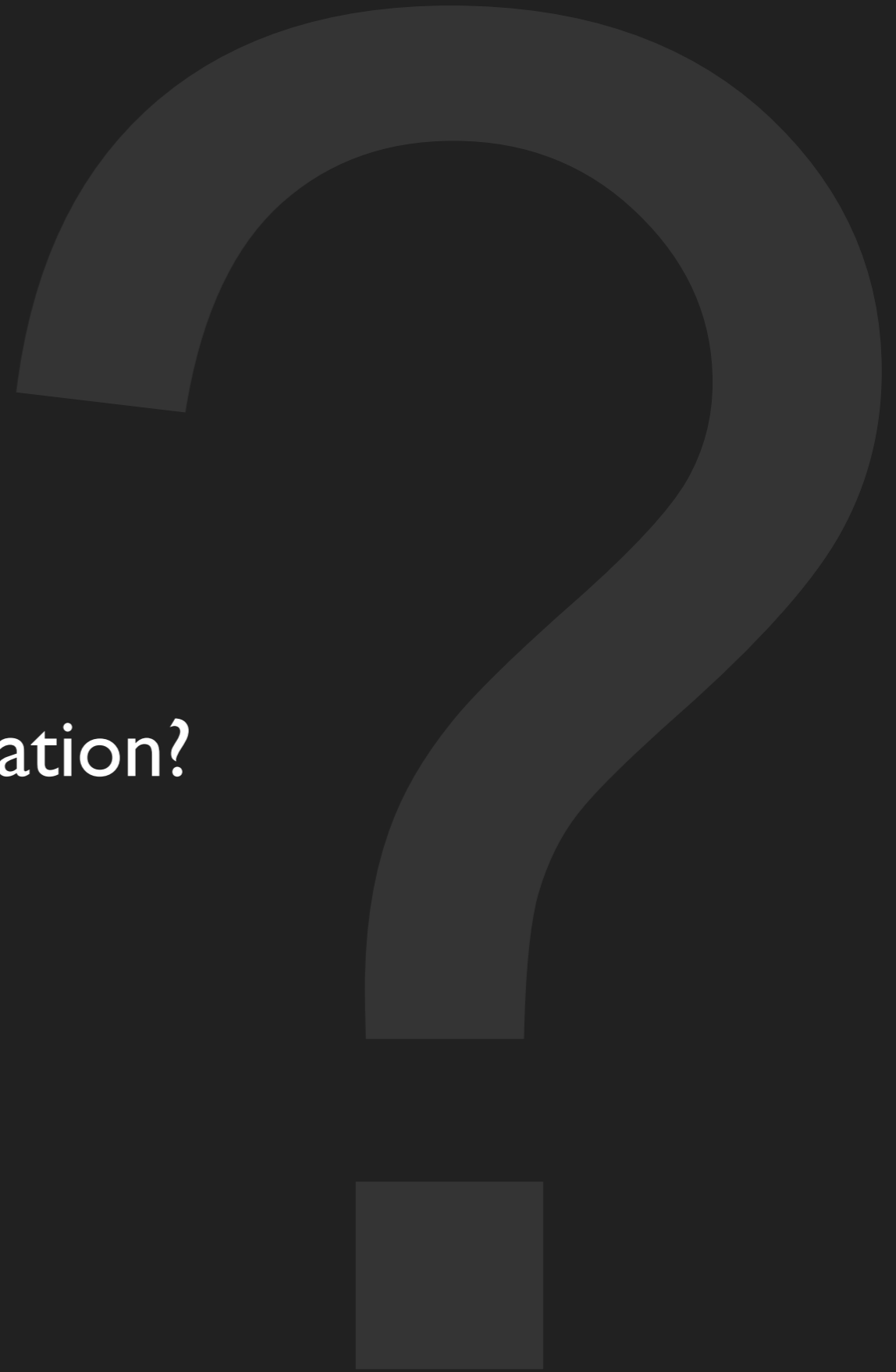


- Applications share functionalities
- Applications do not share user interfaces
- User interfaces are different, but the functionalities overlap
- But they are two, separately maintained applications...

Wouldn't it be nice to have a way to reuse the shared functionality, independent of the interfaces?



Could MVC provide a separation?



We have to configure the Controllers  
for each added interface, or changed  
models.

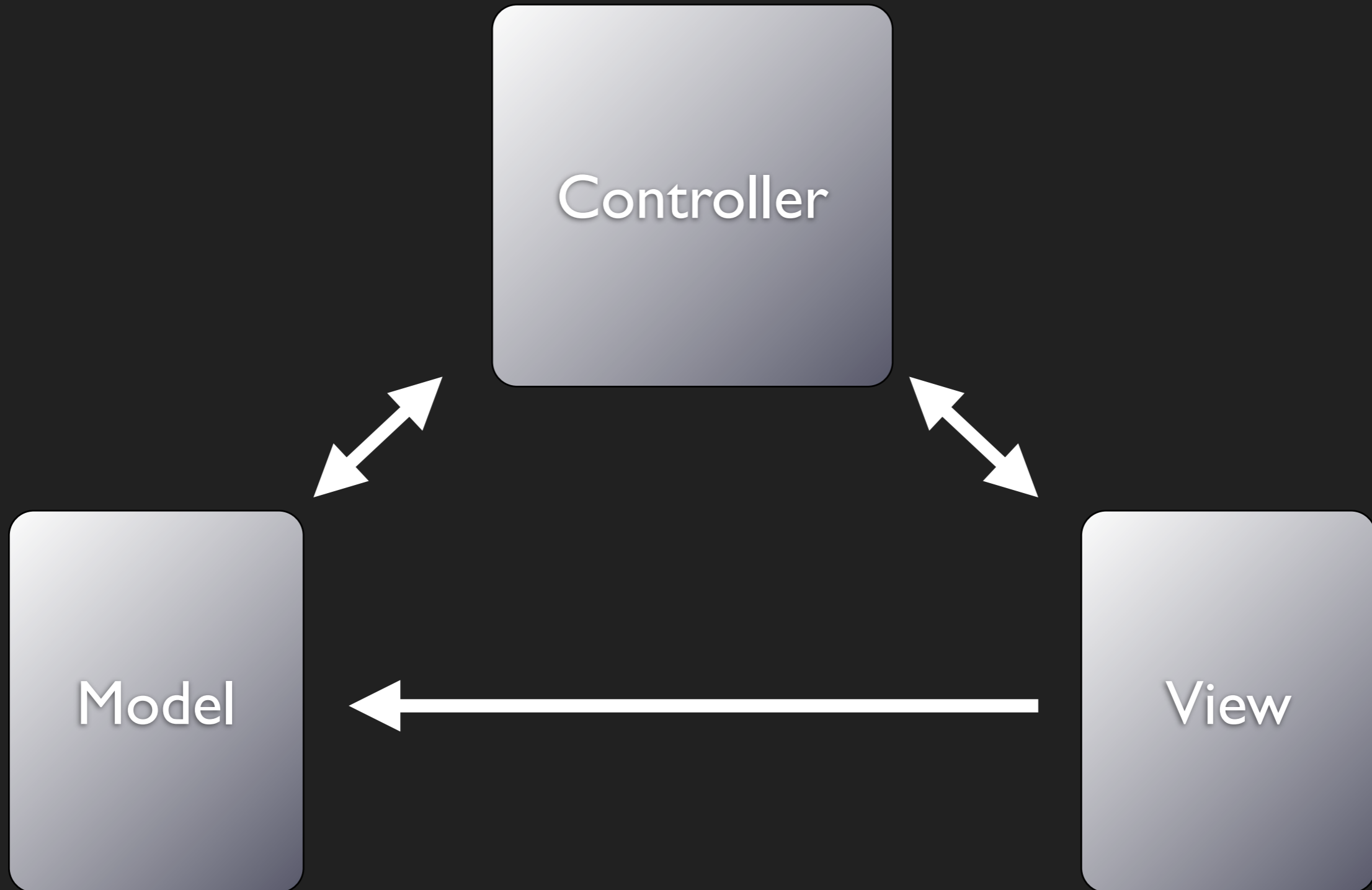
What other architectures and approaches achieve decoupled, or loosely coupled components?



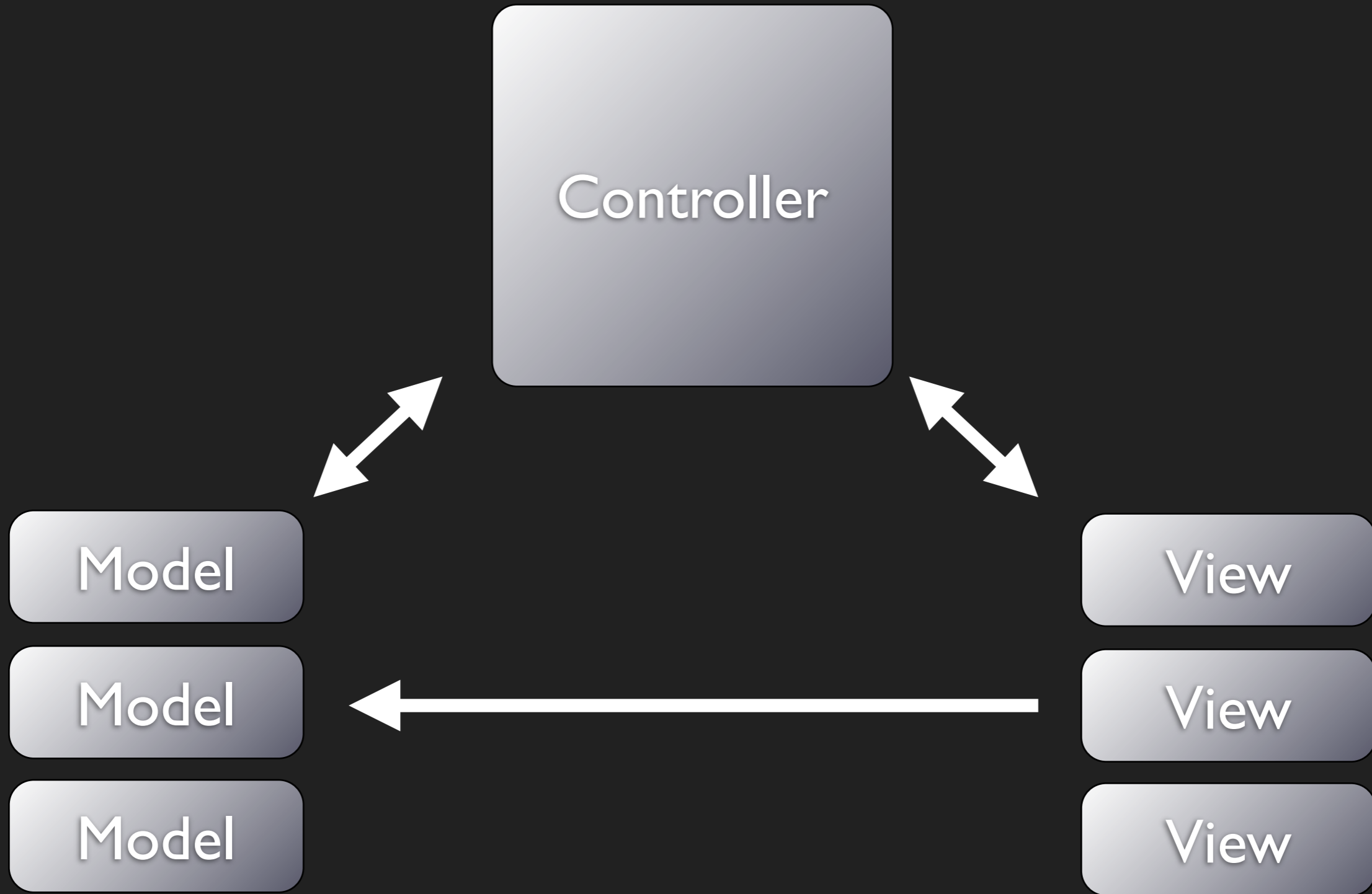
None of them completely address  
bundles of applications.

To be able to write applications that  
accept different interfaces with no, if not  
minimal configuration

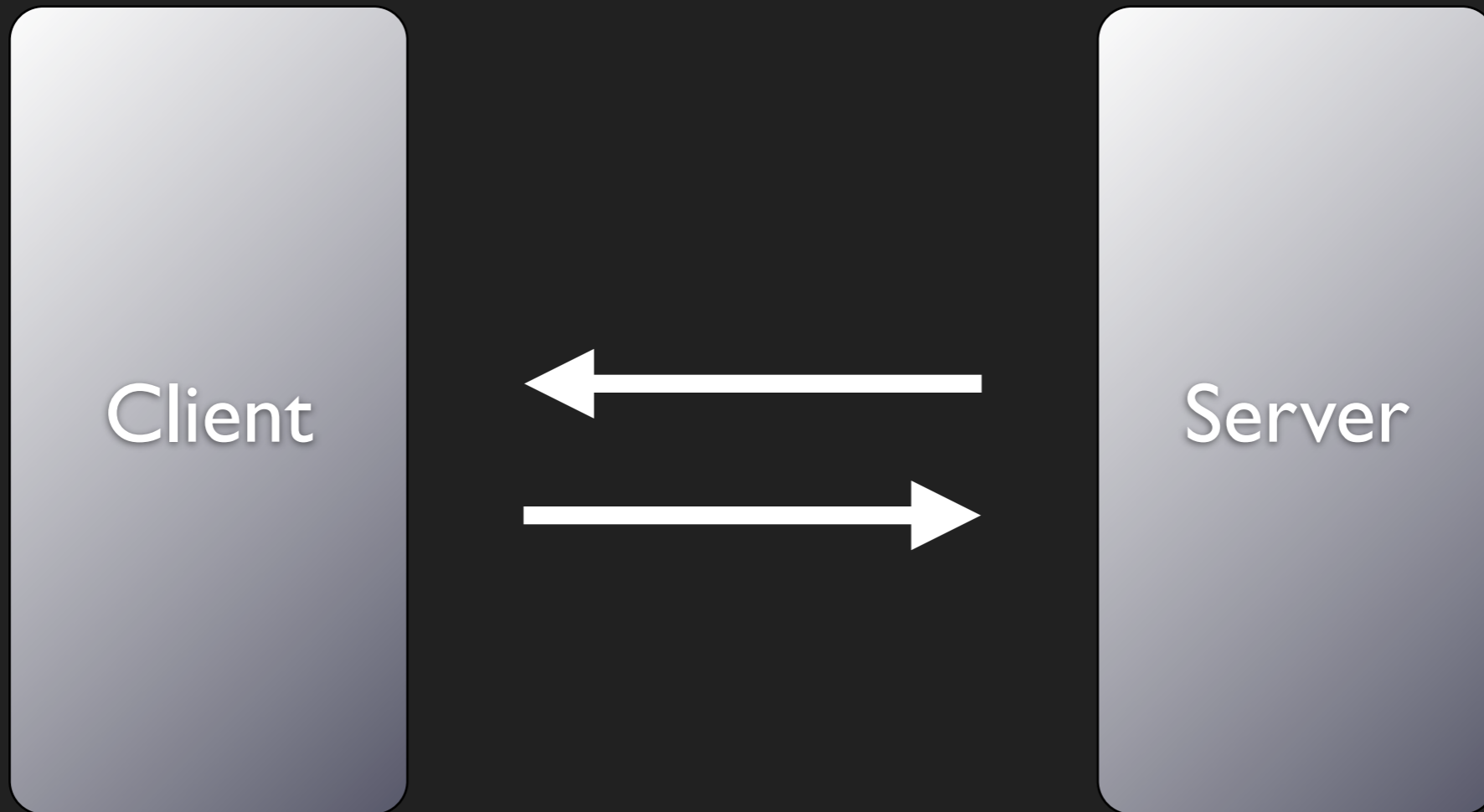
# Idea



# Model View Controller



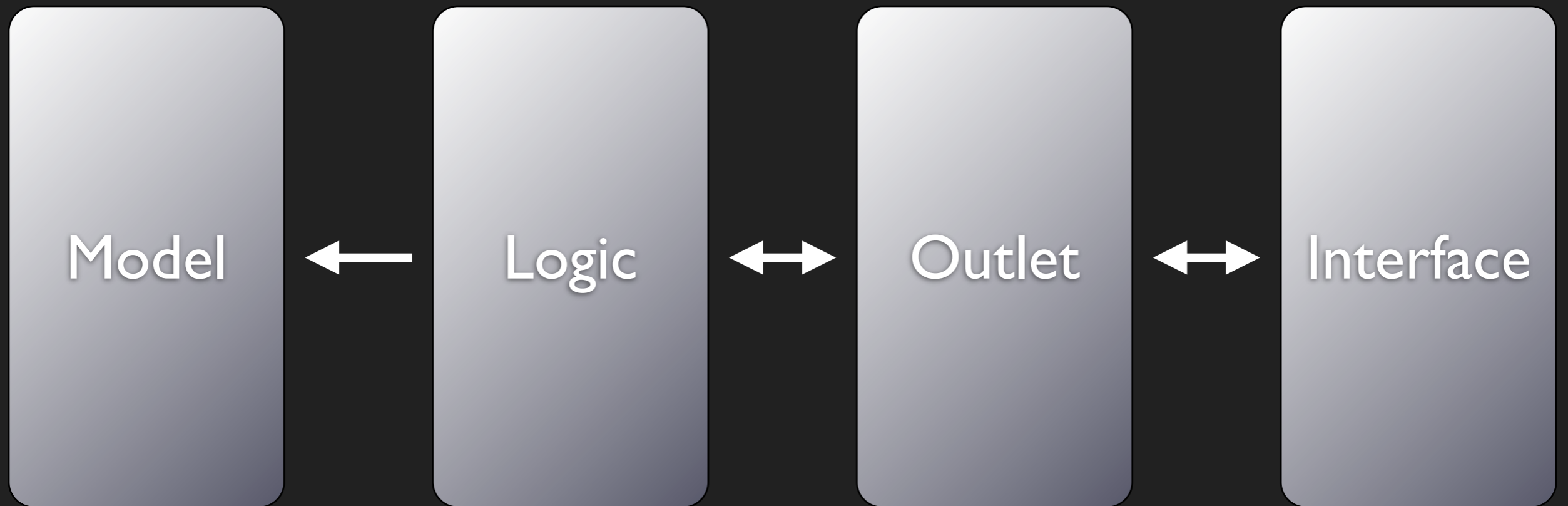
# Model View Controller



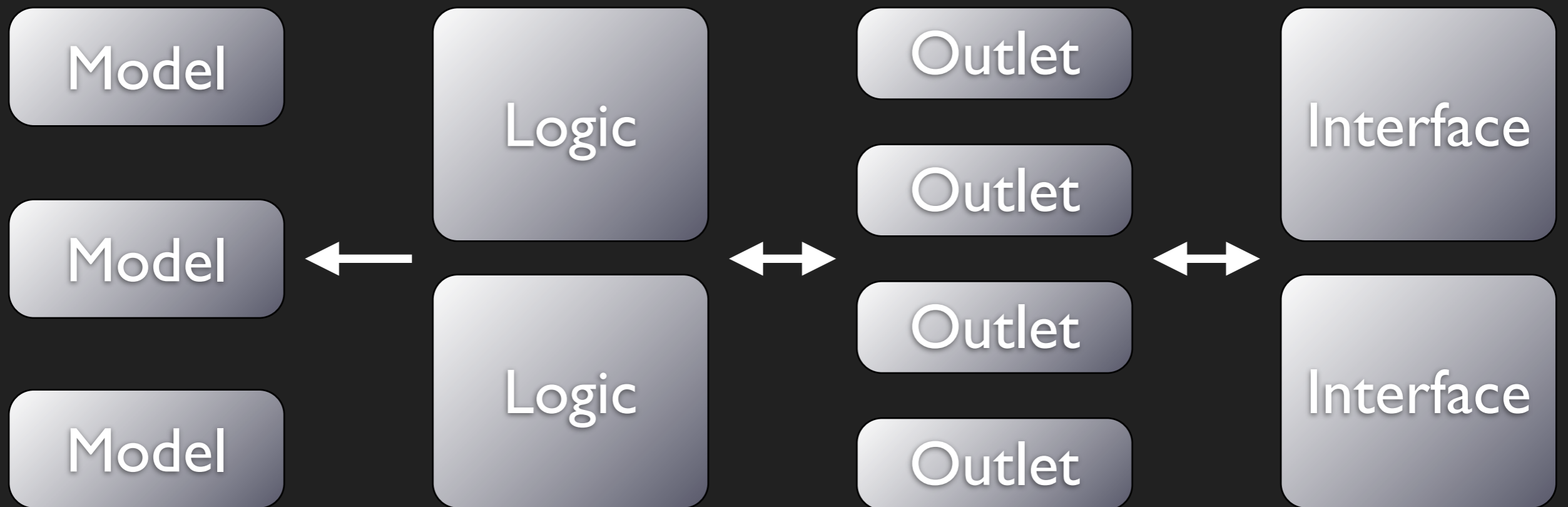
# Client Server



# Tiered Client Server



Model Logic Interface



# Model Logic Interface

# Application architecture

# Model

- Does the same job as in MVC.
- The operations on data is done by Logic
  - Model only provides a container for data and some for state.
  - Operations are done by Logics, not Model, as in MVC.

# Logic

- When an application is assembled, necessary functions are gathered to form “one” Logic.
- Form the basis of applications
- Communicate to Models, handle Interface requests, that are connected through Outlets
- Contain the necessary functions that Interfaces need for an application

# Outlet

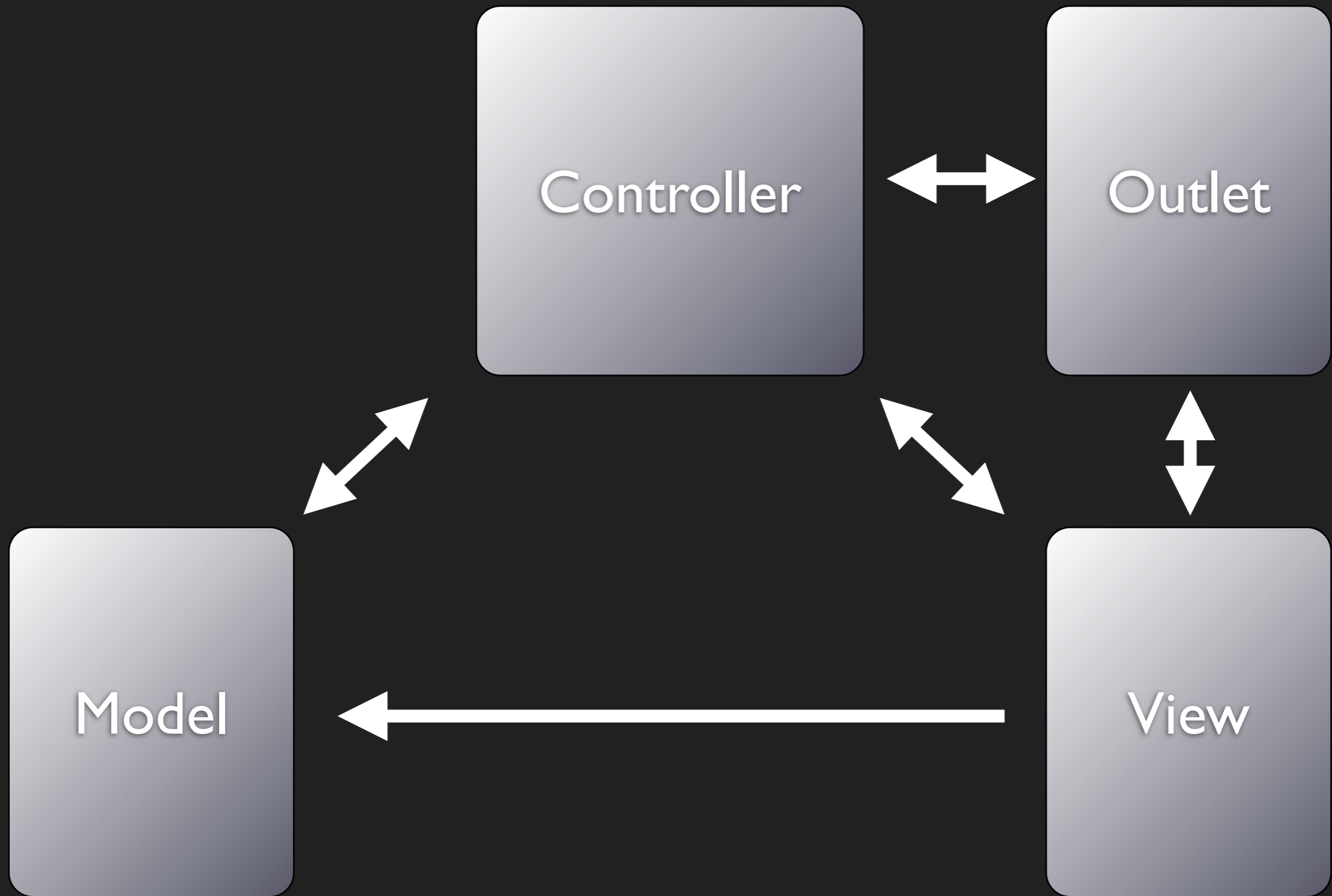
- Classes that process, aggregate and/or transform data to be used in interfaces
- Serve on the line of Model - Logic protocol
- Information about function dependencies and Logics stored in outlet configuration files.

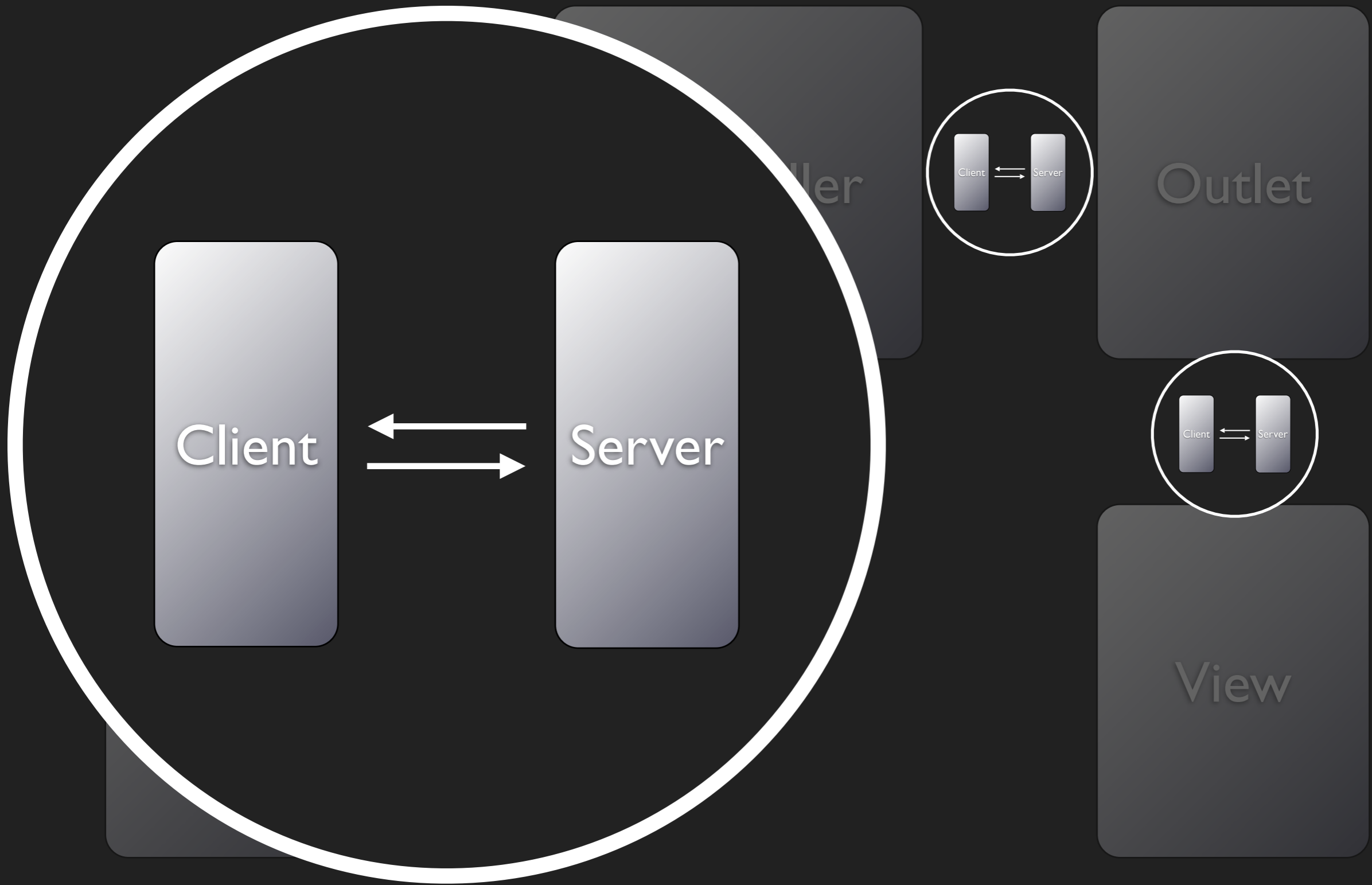
# Outlet

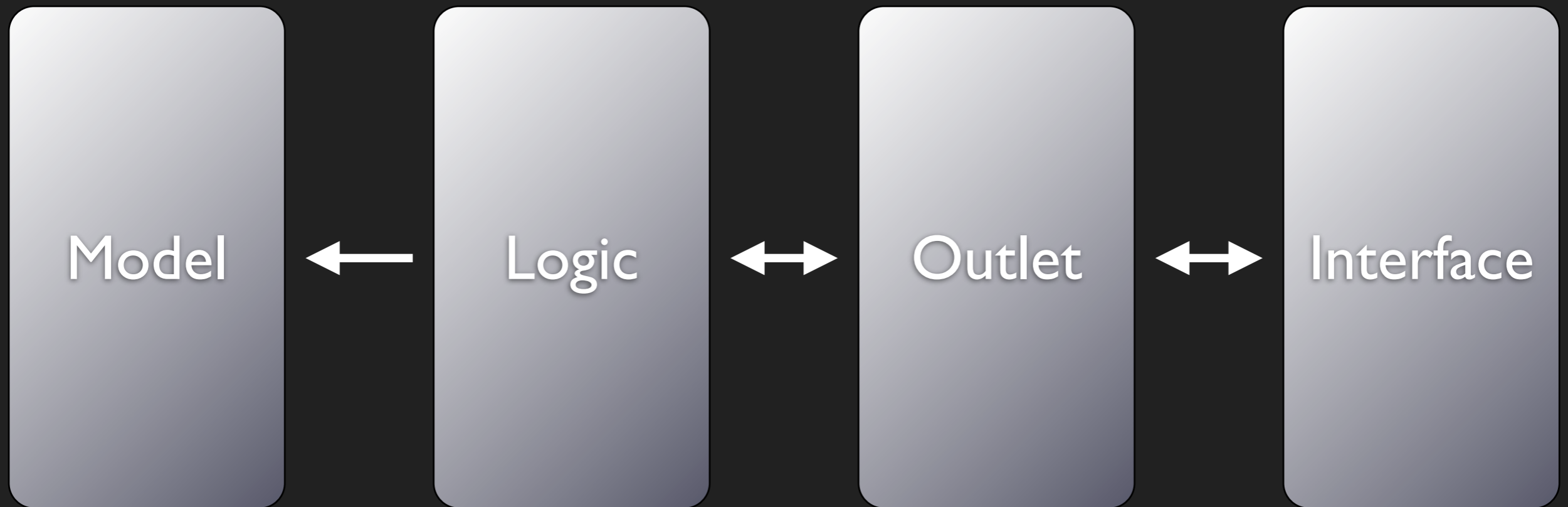
- Does not allow interfaces to communicate directly to Logics, which results in:
  - Does not populate Logics with export methods for Interface requests.
  - Outlets handle return type changes, if any.

# Interface

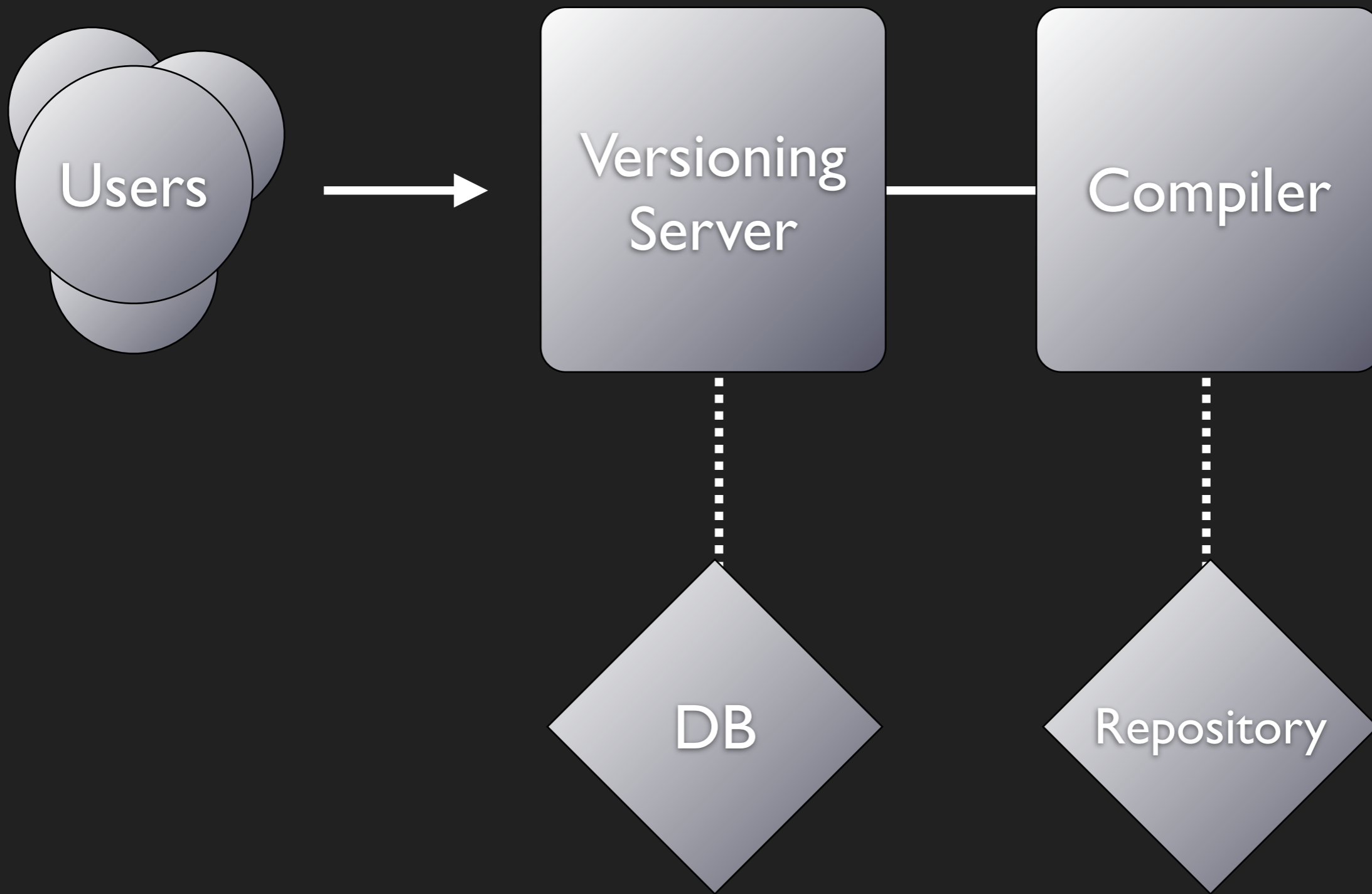
- Entities that are serviced by Outlets
- “Appearance” of applications
- May contain different parts for different operations
- Each different part is bound to different Outlets
- Changes in parts are relayed to Logics via Outlets

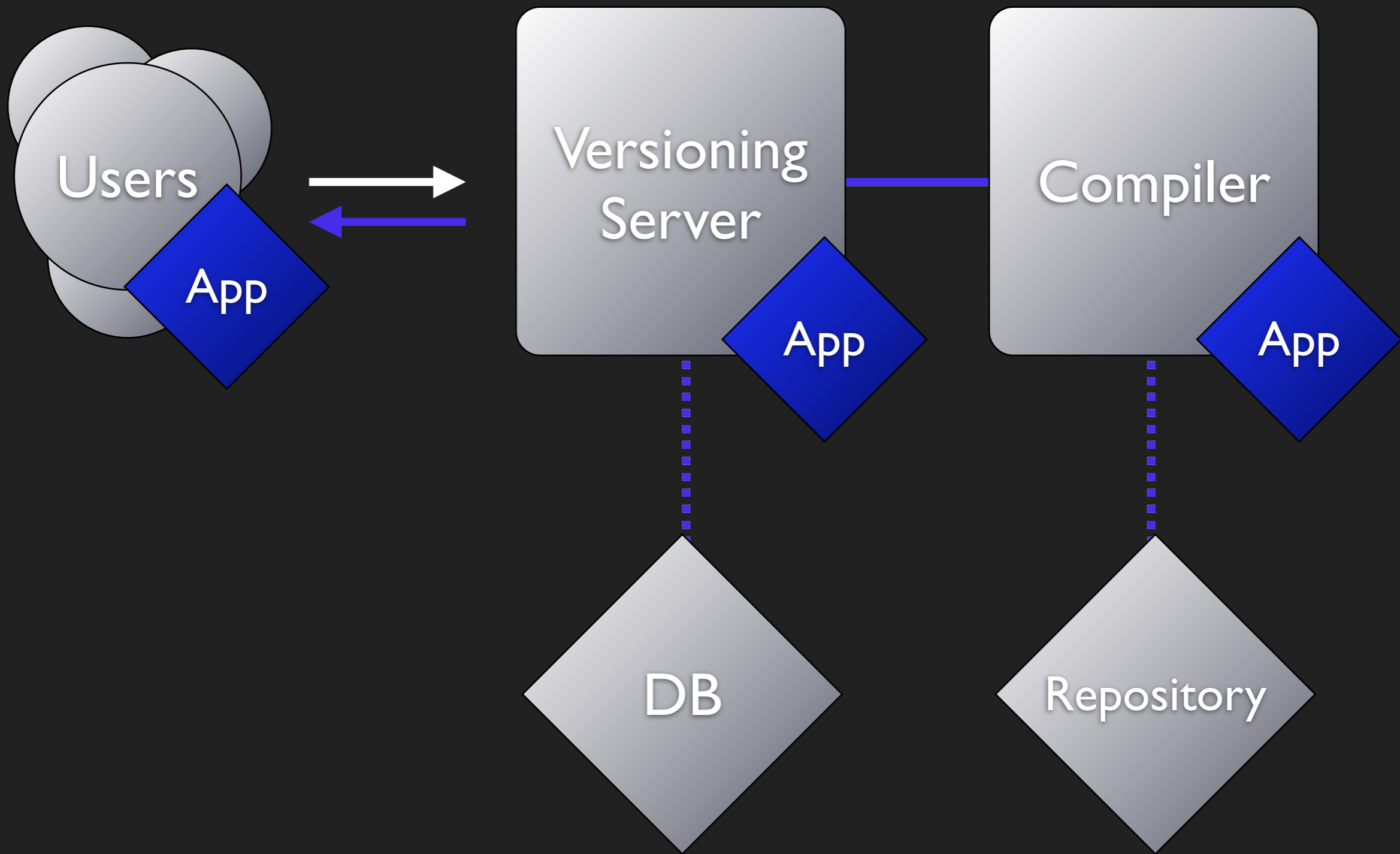






# Framework components





# Compiler

- Required methods in Logics are compiled into one single Logic file.
- Instantiates Models and Outlets to be used by Logic instances.

- Analyze Java file, look for tags
- Find imports and variable declarations
- Find class, inner class and method declarations
- Resolve dependencies between methods used
- Read configurations of components
- Consolidate required methods into one Logic

## Class block

Variables

Imports

Method

Inner Class

## Class block

Variables

Imports

Method

Inner Class

## Logic

Variables

Imports

Method

Inner Class

## Class block

Variables

Imports

Methods

Inner Class

## Class block

Variables

Imports

Method

Inner Class

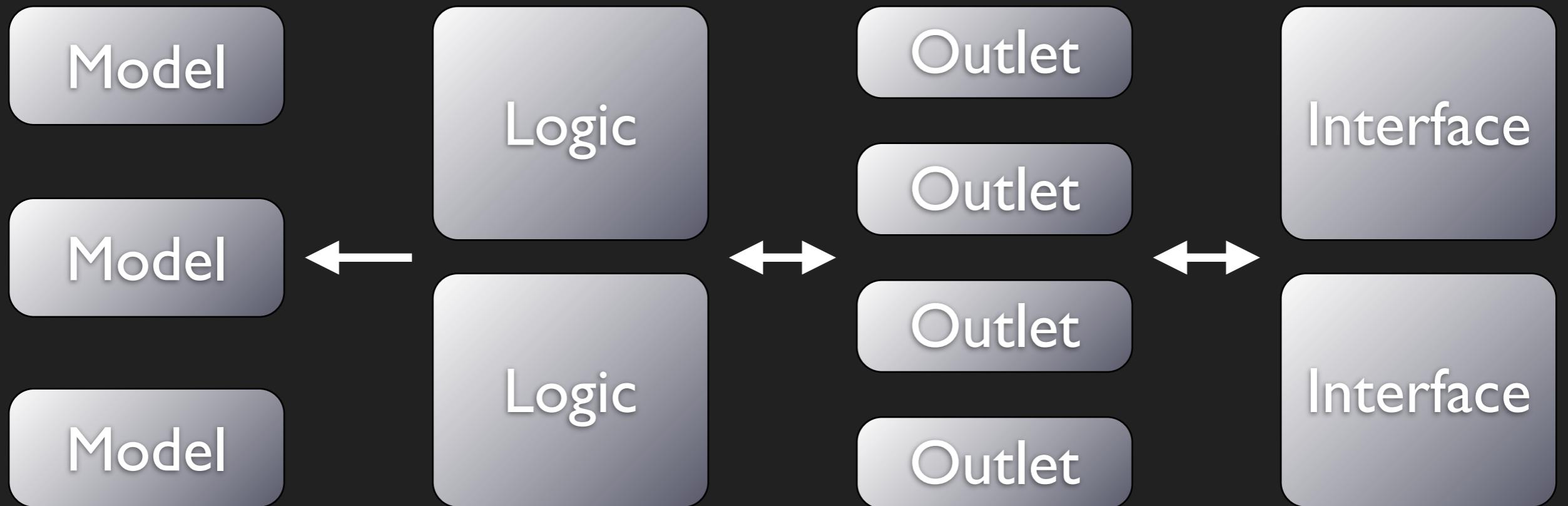
## Logic

Variables

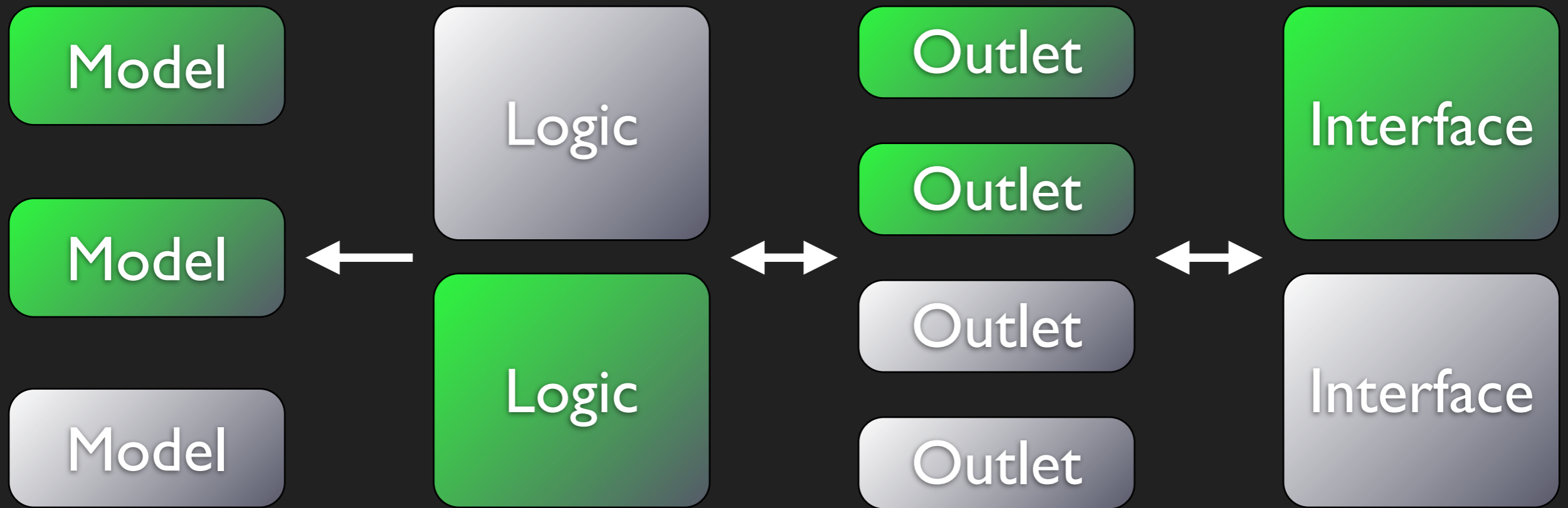
Imports

Method

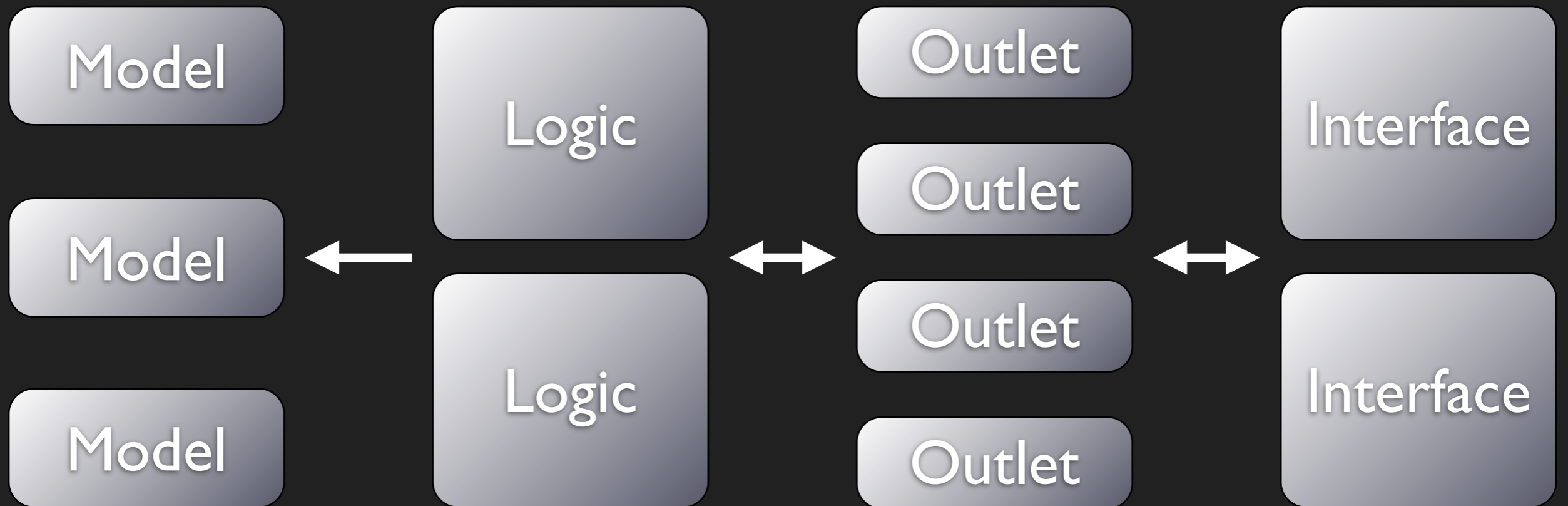
Inner Class



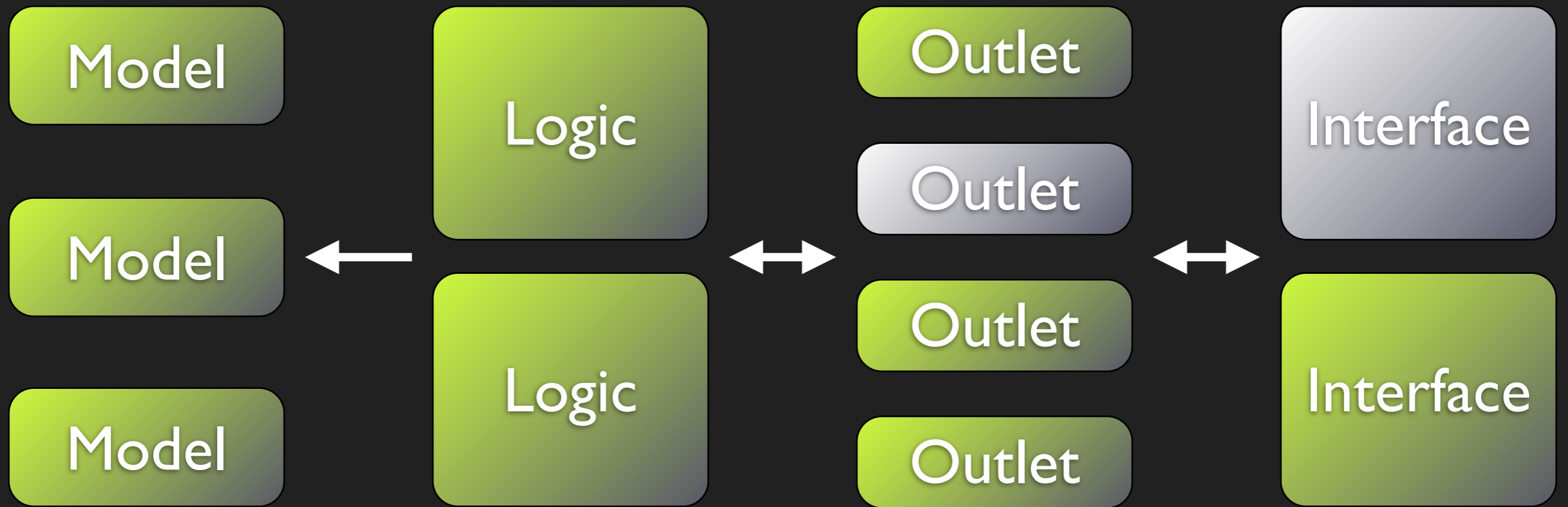
# Compile path



Compile path



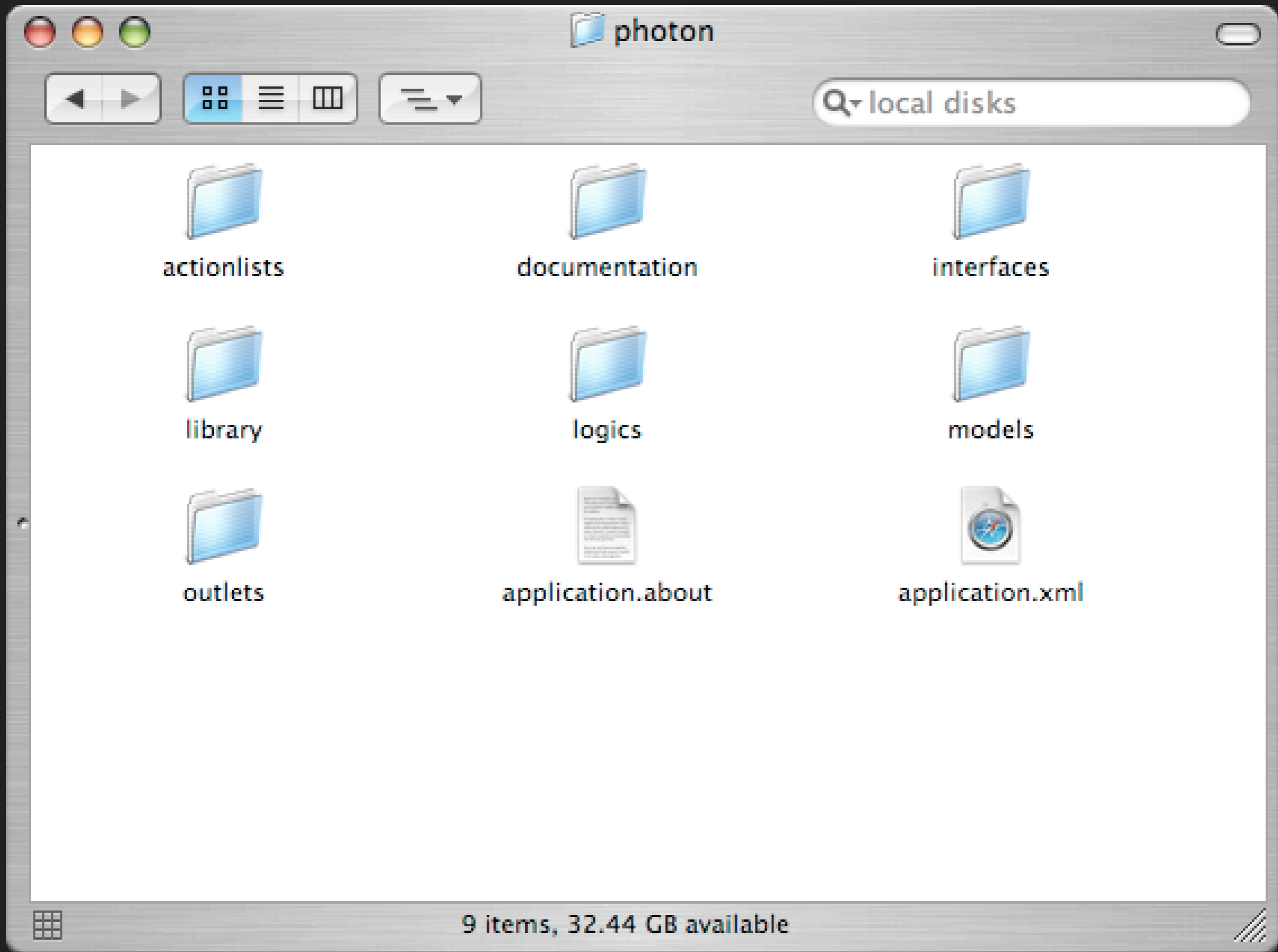
# Compile path



# Compile path

# Provisioning Server

- The components are stored separately until someone needs them. This is our delivery mechanism.
- Upon request
  - collect the required pieces
  - merge them together
  - serve the client a compiled application



# MVC vs. MLI

- Model in MVC does data operations, Model in MLI only holds data or provides a get/set layer.
- Views never access the Logic directly, Outlets are used.
- Applications use a subset of Interfaces, Interfaces use a subset of Outlets, Outlets use a subset of Logics, Logics use a subset of Models.

# Limitations

- Lack of yacc/lex kind of language specifications
- Extremely complex compile patterns
- Difficulties caused by previously adapted programming techniques
- Lack of platforms/tools that would aid in developing a new approach to programming

# Major challenges

- Binding the Outlets to the generated Logic
- Constructor aggregation for the Logic constituents
- Inheritance
  - Overrides, scope, variable shadowing
  - Multiple inheritance
- Data consistency due to static vs dynamic scope

# When to use Pide?

- Applications with many separate interfaces, doing the same things on each interface
- Interfacing with core libraries, other applications, services or legacy systems

# When not to use Pide

- The application has only one, maybe two interfaces
- There is an existing layer that already exposes the functionality of the application, directly or indirectly

# Demo



# Future work

- Provisioning server with distributed repositories
- Heuristic selection of components for optimal deployments
- Web compiler
- Integrated development environment
- A UI bridge toolkit for novel interaction