L14 - Introduction to architectural patterns (2)

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

UNIVERSITY OF ABERDEEN

14.1 Reminding past issues and mapping them to current topics

Where are we now?

Software development paradigms

⇒ The Unified Process (UP) paradigm

⇒ UP phases and UP disciplines (activities) within each phase

⇒ Elaboration (second UP phase)

⇒ Elaboration Design

⇒ Architectural design

⇒ Architectural patterns

⇒ Catalogue of architectural patterns (1)

⇒ Catalogue of architectural patterns (2)

What are architectural patterns? – a gentle reminder

- pattern = general model used as a starting point for system design
- Defines system decomposition, global control flow, handling of boundary conditions, inter-subsystem communication
- There are only a few architectural patterns around:
  - layered architectures (see previous lecture)
  - pipe and filter (see previous lecture)
  - shared-data (Word, Netbeans, . . .)
  - blackboard (a typical AI system pattern)
  - event-driven
  - model-view-controller (a typical Web IS pattern)

14.2 The shared-data architecture

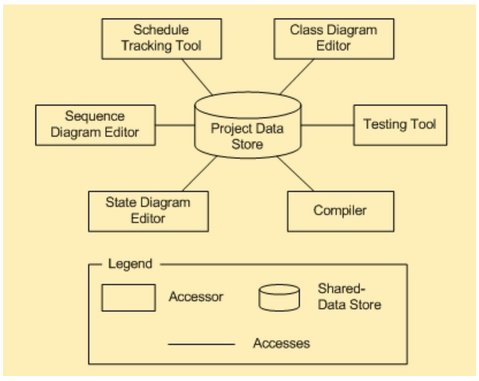
The shared-data architectural pattern

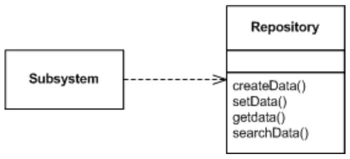
- Characterised by one or more *shared-data stores* used by one or more share-data accessors
- Shared-data accessors (i) store, delete, and modify data in shared-data stores and (ii) communicate through shared-data stores only
- Shared-data stores have no knowledge of accessors
- This pattern is used in all those cases where different units work off the same data (such as in the case of IDEs)

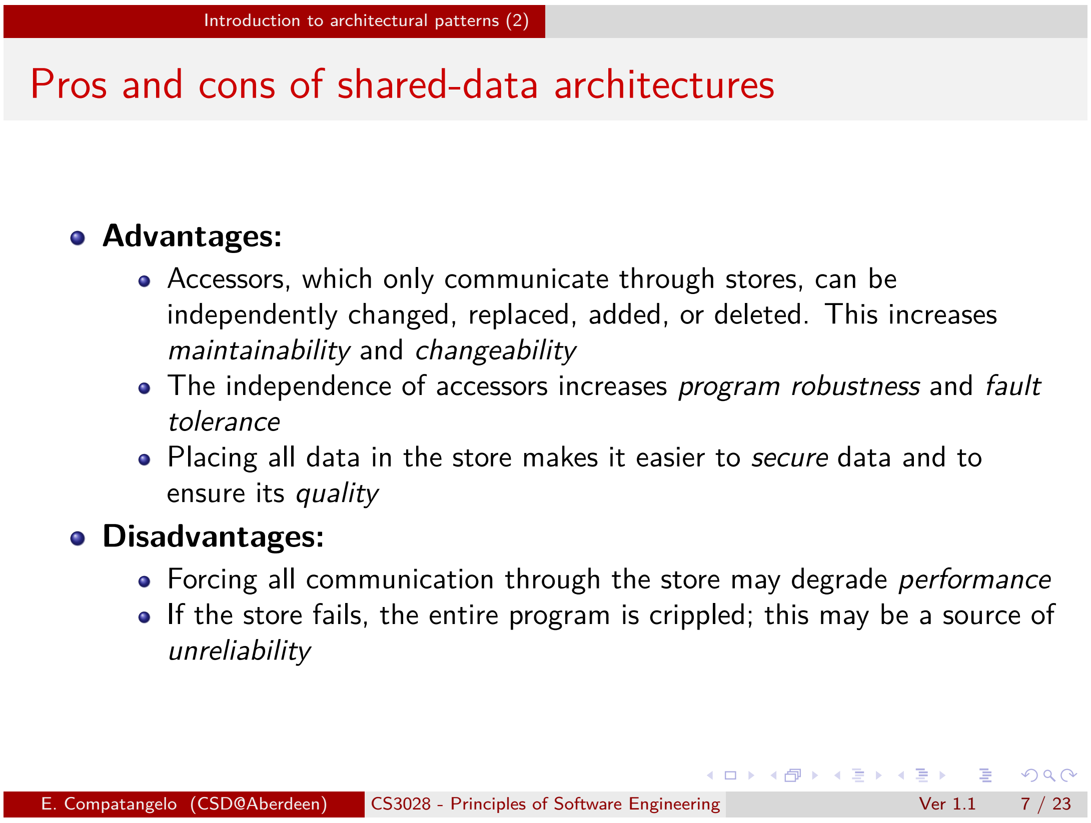
Types of shared-data architectures

- If shared-data stores activate the accessors when the shared-data stores change, then this gives rise to a **blackboard architectural pattern**, where shared-data stores are called *blackboards*
- If shared-data stores are passive and are queried by the accessors (which may run continuously or be controlled by some other component), then this gives rise to a **repository architectural pattern**, where the shared-data stores are called *repositories*

Shared-data architectural pattern examples







Introduction to architectural patterns (2)

Pros and cons of shared-data architectures

- **Advantages:**
  - Accessors, which only communicate through stores, can be independently changed, replaced, added, or deleted. This increases *maintainability* and *changeability*
  - The independence of accessors increases *program robustness* and *fault tolerance*
  - Placing all data in the store makes it easier to *secure* data and to ensure its *quality*
- **Disadvantages:**
  - Forcing all communication through the store may degrade *performance*
  - If the store fails, the entire program is crippled; this may be a source of *unreliability*

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 7 / 23

14.2.1 Shared-data architecture: pattern specification

Name: Shared-data

Application: Model several independent program components using persistent data.

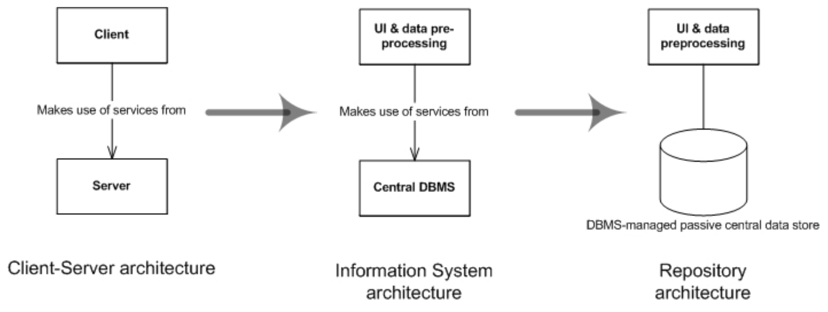
Form: This style does not specify a static form. Its dynamic form is one or more shared-data stores accessed by one or more shared-data accessors. The shared-data accessors do not directly interact. The accessors are activated by the shared-data stores when they change (the *Blackboard* style), or the shared-data stores are passive (the *Repository* style).

Consequences: It is easy to change, replace, remove, or add shared-data accessors in shared-data-style programs, making them changeable, maintainable, robust, and fault tolerant. Shared-data stores contribute to data security and quality. However, they are a single point of failure, which may decrease reliability.

14.3 Revisiting layered architectures

From client-server to shared data (repository)

- An IS with a central DB is an example of client-server architecture
- Clients receive user inputs, perform range checks, and initiate DB transactions when all needed data has been collected
- Server performs the transaction and guarantees data integrity
- This client-server architecture is a special case of *passive* shared data store (aka *repository*) architecture where the central data structure is managed by a process

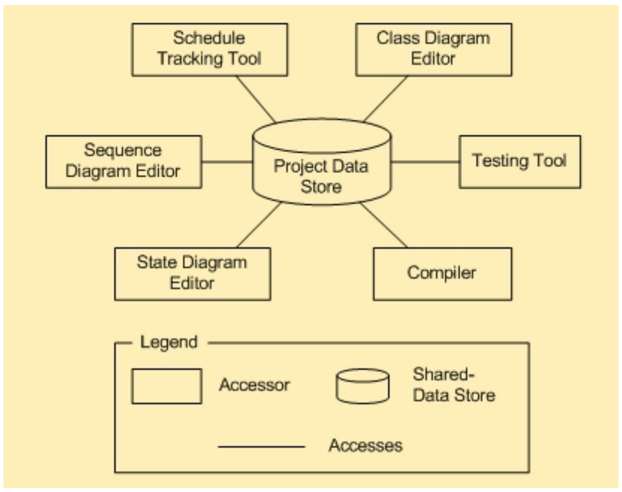

Client
Makes use of services from
Server
Client-Server architecture
UI & data pre-processing
Makes use of services from
Central DBMS
Information System architecture
UI & data preprocessing
DBMS-managed passive central data store
Repository architecture

14.4 The blackboard architecture

The blackboard pattern (specialising the shared data one)

- Characterised by a *blackboard* used by one or more accessors
- Blackboard accessors (i) store, delete, and modify data in the blackboard and (ii) communicate through the blackboard only
- The blackboard has no knowledge of accessors
- The blackboard activates the accessors when its data/knowledge content changes
- This pattern is used when different units work off the same knowledge, as in the case of autonomous robots or (hybrid) reasoners

Blackboard architectural pattern - shared data origin

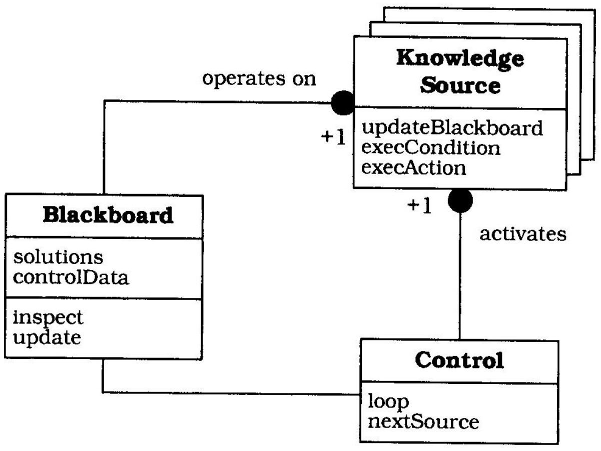


The blackboard architecture is not a client-server architecture!

Blackboard pattern components

- The blackboard, *i.e.*, the central data store
- A collection of *knowledge sources* (the accessors), *i.e.*, separate, independent subsystems that solve specific problem aspects
- A *control component*, which runs a loop that monitors the changes in the blackboard and decides what action to take next, scheduling knowledge sources evaluations and activations according to a strategy based on blackboard data

Blackboard architectural pattern - actual structure

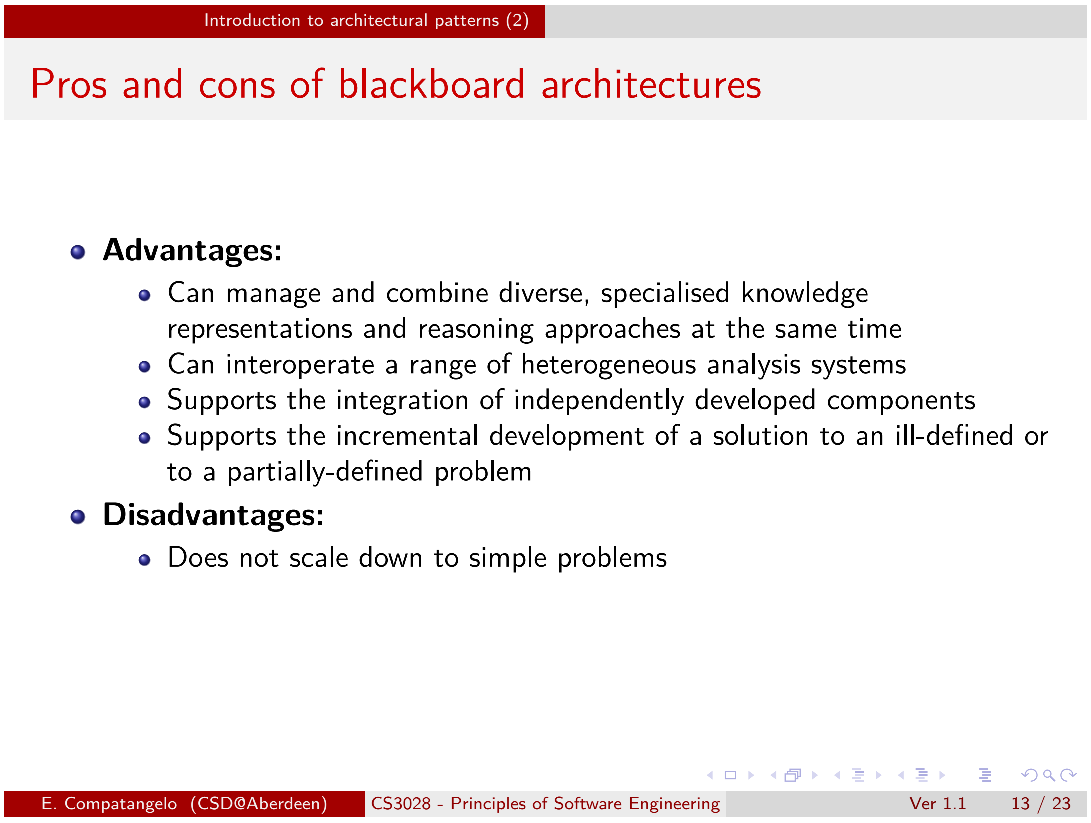


Blackboard-centric systems The blackboard pattern has been used in the following categories of intelligent systems:

- Sensorial interpreters (*e.g.*, as part of robotic systems, including computer vision, speech and sound analysis, etc.)

- Planners and schedulers

- Hybrid semantic reasoning engines
- Military Command, Control, & Coordination systems
- Machine learning systems
- Data fusion (integration, articulation, alignment) systems
- Distributed intelligent agents



Pros and cons of blackboard architectures

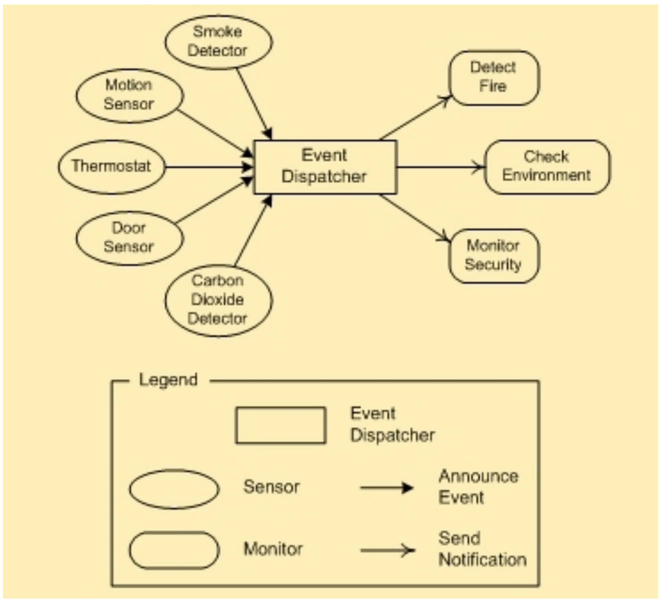
- **Advantages:**
  - Can manage and combine diverse, specialised knowledge representations and reasoning approaches at the same time
  - Can interoperate a range of heterogeneous analysis systems
  - Supports the integration of independently developed components
  - Supports the incremental development of a solution to an ill-defined or to a partially-defined problem
- **Disadvantages:**
  - Does not scale down to simple problems

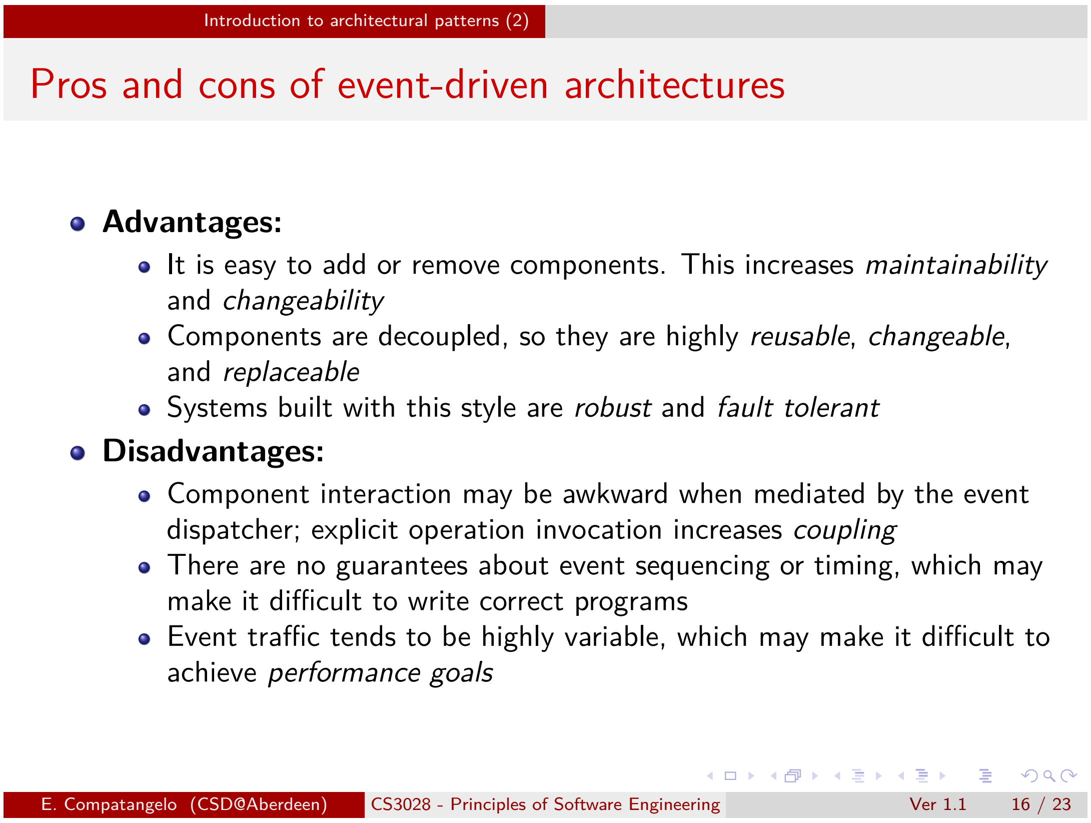
14.5 The event-driven architecture

The event-driven architectural pattern

- Also known as the *implicit-invocation* architectural pattern
- Has a special *event dispatcher* that mediates between components that (i) announce and (ii) are notified of events
- An event is an important occurrence that happens at a particular time
- Components register interest in events by requesting that the event dispatcher notify them when the events occur
- Components announce events to the event dispatcher, which then notifies all components registered for those event
- The event dispatcher is usually a distinct module separate from other program modules and often built into a system infrastructure
- This pattern is used in all those cases where asynchronous events may trigger certain actions (such as in the case of alarm systems)
- The Java GUI framework (AWT & Swing) has a built-in event dispatcher that programmers can assume will dispatch events

Event-driven architectural pattern example





Introduction to architectural patterns (2)

Pros and cons of event-driven architectures

- **Advantages:**
  - It is easy to add or remove components. This increases *maintainability* and *changeability*
  - Components are decoupled, so they are highly *reusable*, *changeable*, and *replaceable*
  - Systems built with this style are *robust* and *fault tolerant*
- **Disadvantages:**
  - Component interaction may be awkward when mediated by the event dispatcher; explicit operation invocation increases *coupling*
  - There are no guarantees about event sequencing or timing, which may make it difficult to write correct programs
  - Event traffic tends to be highly variable, which may make it difficult to achieve *performance goals*

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 16 / 23

14.5.1 Event-driven architecture: pattern specification

Name: Event-driven or *implicit-invocation*

Application: Model a program that must react to unpredictable sequences of inputs.

Form: This style does not specify a static form. In its dynamic form, one or more event announcers send events to an event dispatcher. When an event occurs, the event dispatcher invokes procedures of components that have registered interest in the event.

Consequences: Components that announce events are independent of other components that announce events and of components that respond to them, which are themselves independent of one another. Hence, components are easy to change, replace, remove, or add, making Event-Driven-style programs changeable, maintainable, robust, and fault tolerant.

Interacting entirely through the event dispatcher is awkward. The independence of components may make it hard to establish program correctness. Program performance may degrade when many events occur in quick succession.

14.5.2 Event-driven architecture: stylistic variations

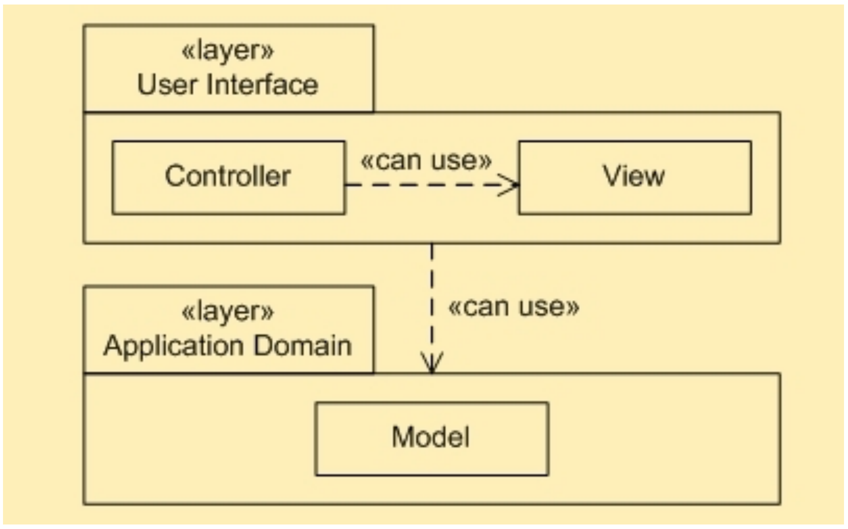
- Events may be notifications or they may carry data
- Events may have constraints honoured by the dispatcher, or the dispatcher may manipulate events
- Events may be dispatched synchronously or asynchronously
- Event registration may be constrained in various ways (*e.g.*, at compilation time or during run time)

14.6 The Model-View-Control architecture

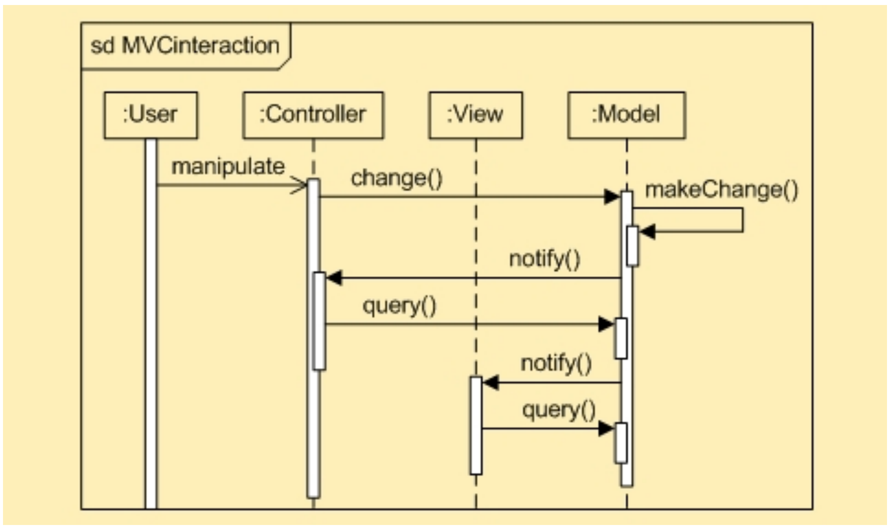
The Model-View-Controller architecture

- The MVC pattern shows how to set up the relationships between *user interface* and *problem-domain components*
- **Model**: a problem-domain component with data and operations for achieving program goals independent of the user interface
  - central architectural component
  - aware of each of its dependent views and controllers
- **View**: a data display component
  - corresponds to a particular style and format of information presentation
  - retrieves data from model and updates presentations when data is changed in one of the other views
  - creates its associated controllers
- **Controller**: a component that receives (and acts on) user input
  - accepts user input in the form of events which trigger execution of operations within the model
  - if these change information, then controller triggers updates in all views

MVC architecture: static structure



MVC architecture: component behaviour

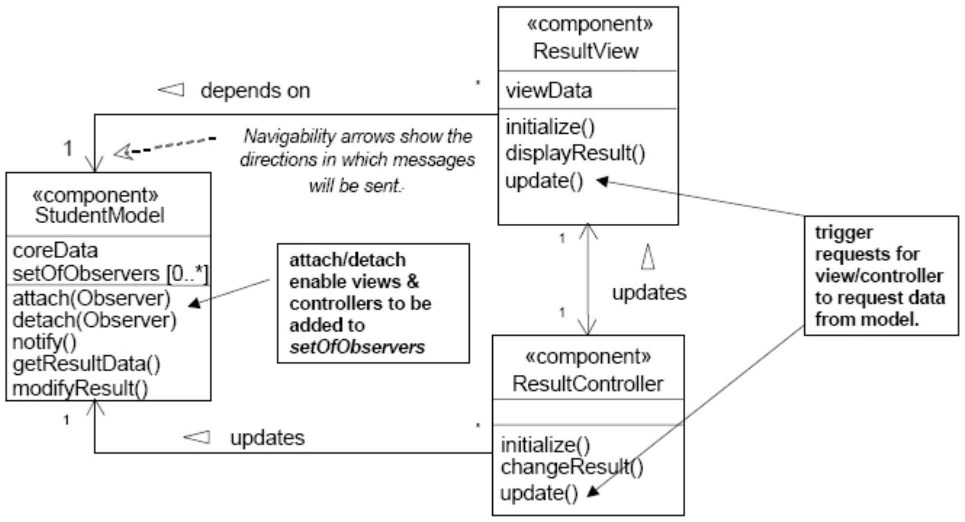


Model-View-Control separation principle

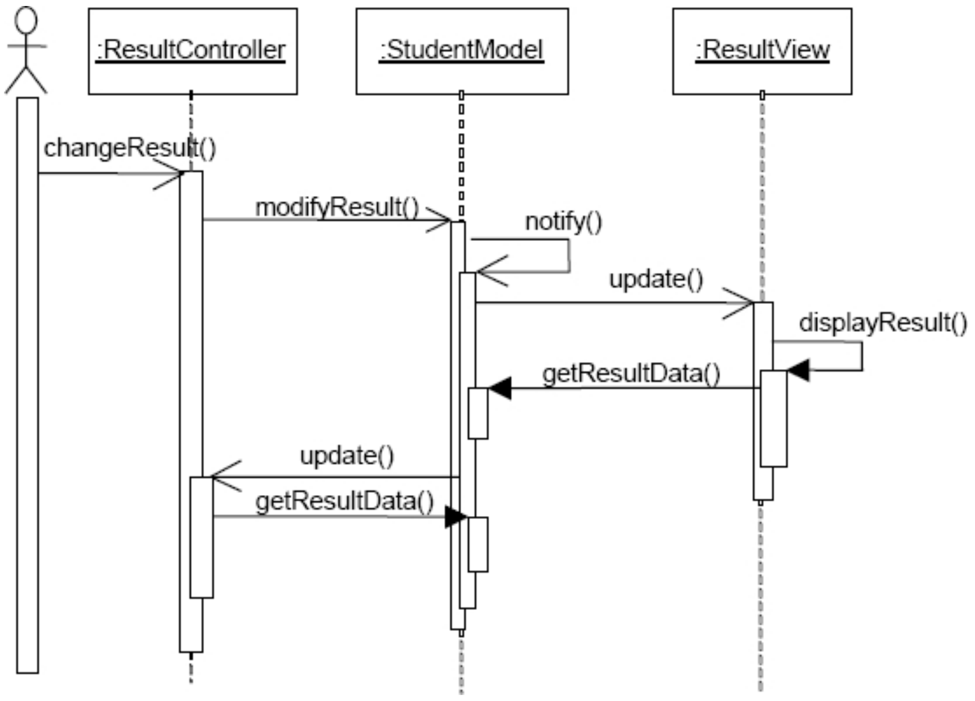
- Do not directly connect/couple elements in the user interface (presentation) layer (also known as the **View**) to objects in other layers , notably in the domain layer (also known as **Model**).

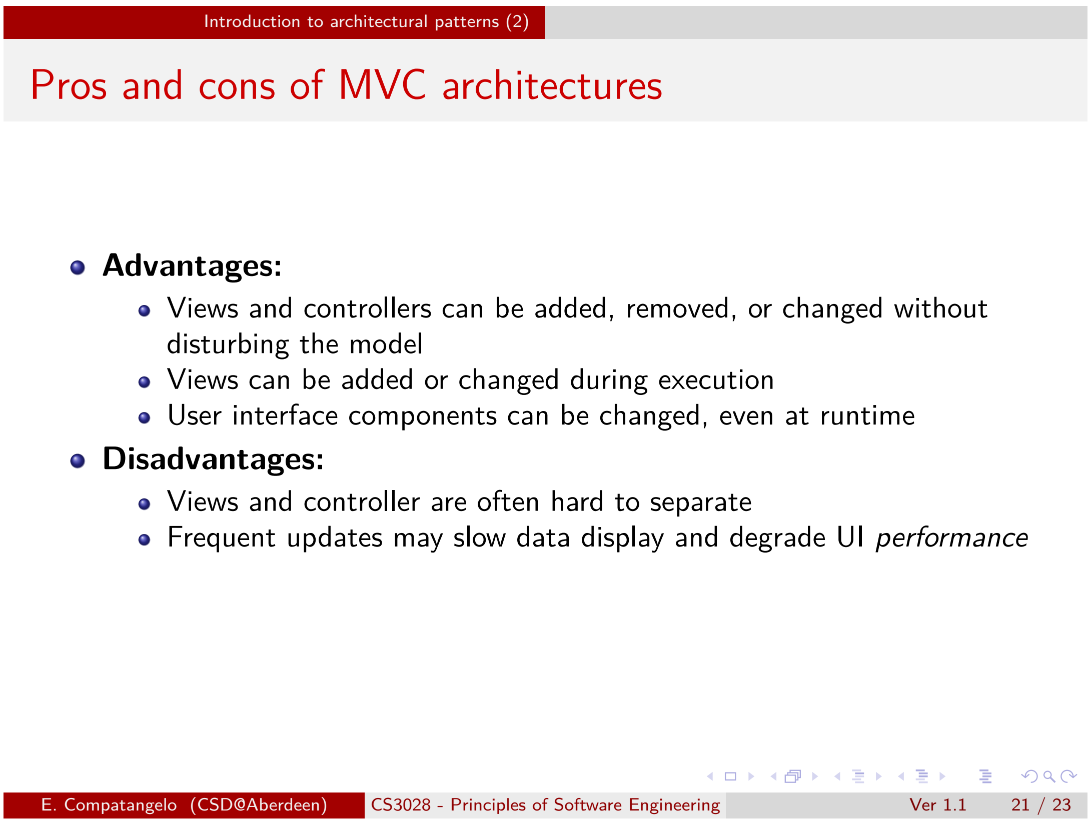
  Presentation layer objects are context-dependant and change frequently with time.
- Do not embed application logic (also known as **Control**) in View layer object methods.
  Same reason as above.

MVC application example - class diagram



MVC application example - sequence diagram





Introduction to architectural patterns (2)

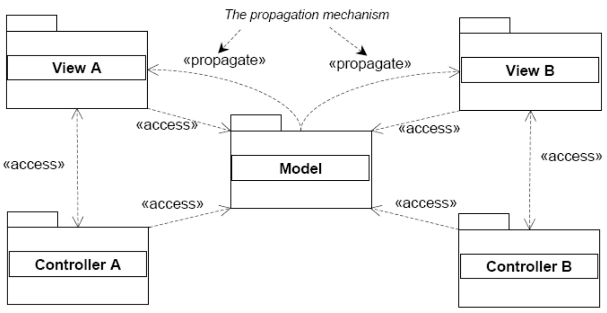
Pros and cons of MVC architectures

- **Advantages:**
  - Views and controllers can be added, removed, or changed without disturbing the model
  - Views can be added or changed during execution
  - User interface components can be changed, even at runtime
- **Disadvantages:**
  - Views and controller are often hard to separate
  - Frequent updates may slow data display and degrade UI *performance*

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 21 / 23

14.6.1 The MVC propagation mechanism

- **User** is given access to only that part of the functionality relevant to his/her role
- **Information** views differ according to user's role
- **Propagation mechanism** enables the model to inform each view that model data has changed



14.6.2 Model-View-Controller architecture: pattern specification

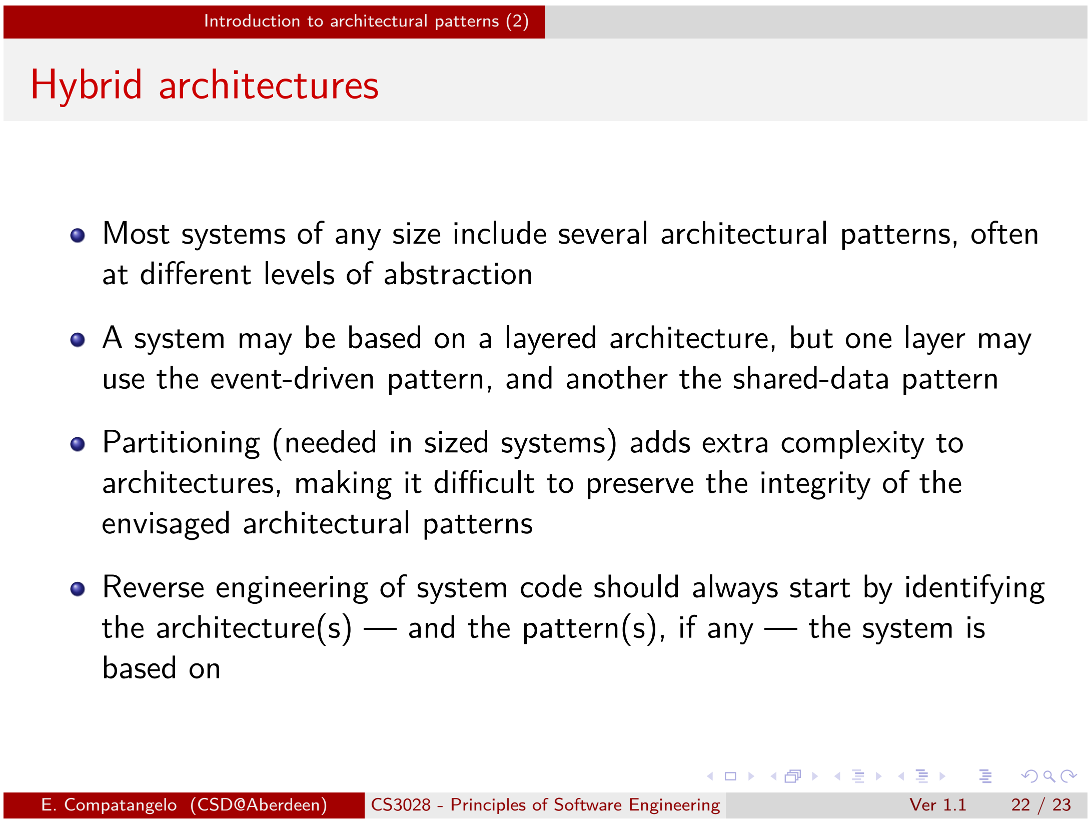
Name: Model-View-Controller (MVC)

Application: Model a program with interactive user interfaces.

Form: This style is a specialisation of the Strict Layered style. The top user interface layer contains view modules for data display and controller modules for user input. The bottom application domain layer contains model modules embodying the data and operations implementing core program function. Controllers may change views or models, and models may change on their own. When a model changes, it notifies views and controllers, which respond by querying the model and updating themselves.

Consequences: The user interface is decoupled from the models, so it may be changed independently, increasing changeability and maintainability. Views and controllers may be changed even at runtime, increasing flexibility and configurability. However, views and controllers are highly dependent on models, decreasing application domain layer changeability and maintainability. User interface update performance may be a problem.

14.7 Hybrid architectures



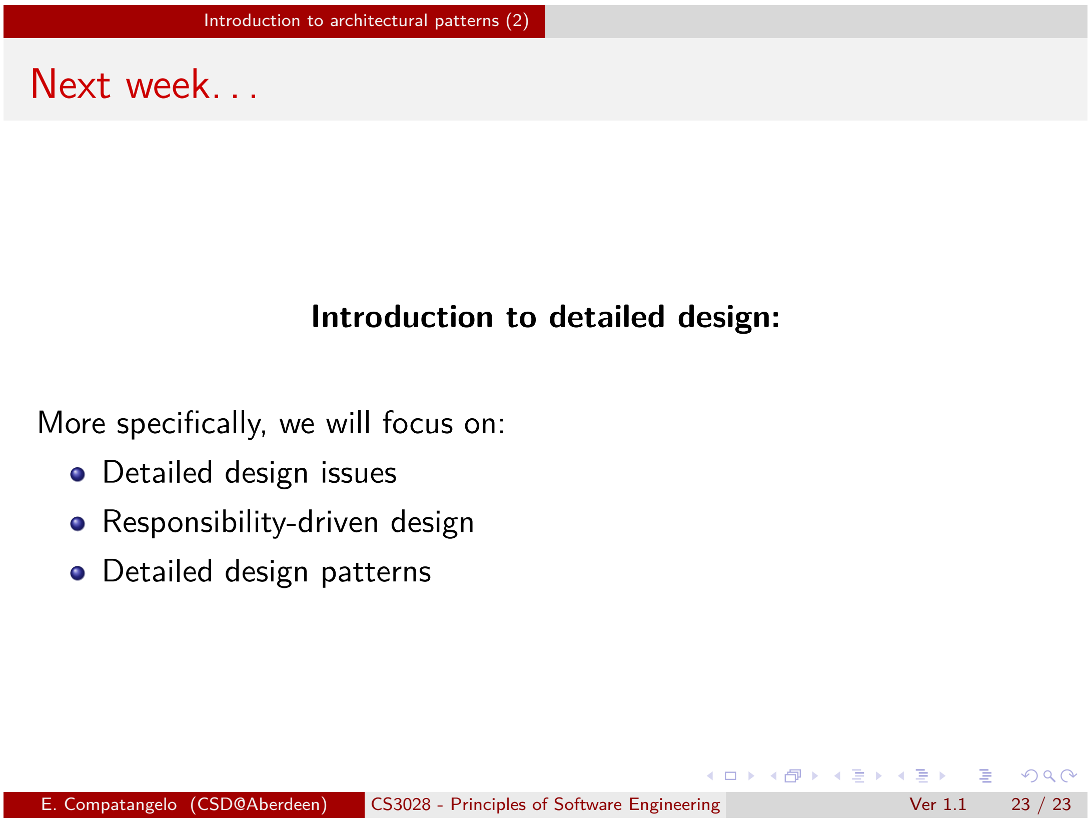
Introduction to architectural patterns (2)

Hybrid architectures

- Most systems of any size include several architectural patterns, often at different levels of abstraction
- A system may be based on a layered architecture, but one layer may use the event-driven pattern, and another the shared-data pattern
- Partitioning (needed in sized systems) adds extra complexity to architectures, making it difficult to preserve the integrity of the envisaged architectural patterns
- Reverse engineering of system code should always start by identifying the architecture(s) — and the pattern(s), if any — the system is based on

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 22 / 23

14.8 Preparing for the topic ahead



Next week. . .

Introduction to detailed design:

More specifically, we will focus on:

- Detailed design issues
- Responsibility-driven design
- Detailed design patterns