MVC design pattern

- It is common to divide our application into three main components:
  - Model – stores data
  - View – displays data
  - Controller – moves data
- Be aware that the MVC pattern has several different interpretations – especially when it comes to the different tasks of the view and the controller.
- MVC requires that we can divide the application into different parts – usually by using OOP.

MVC design pattern cont.

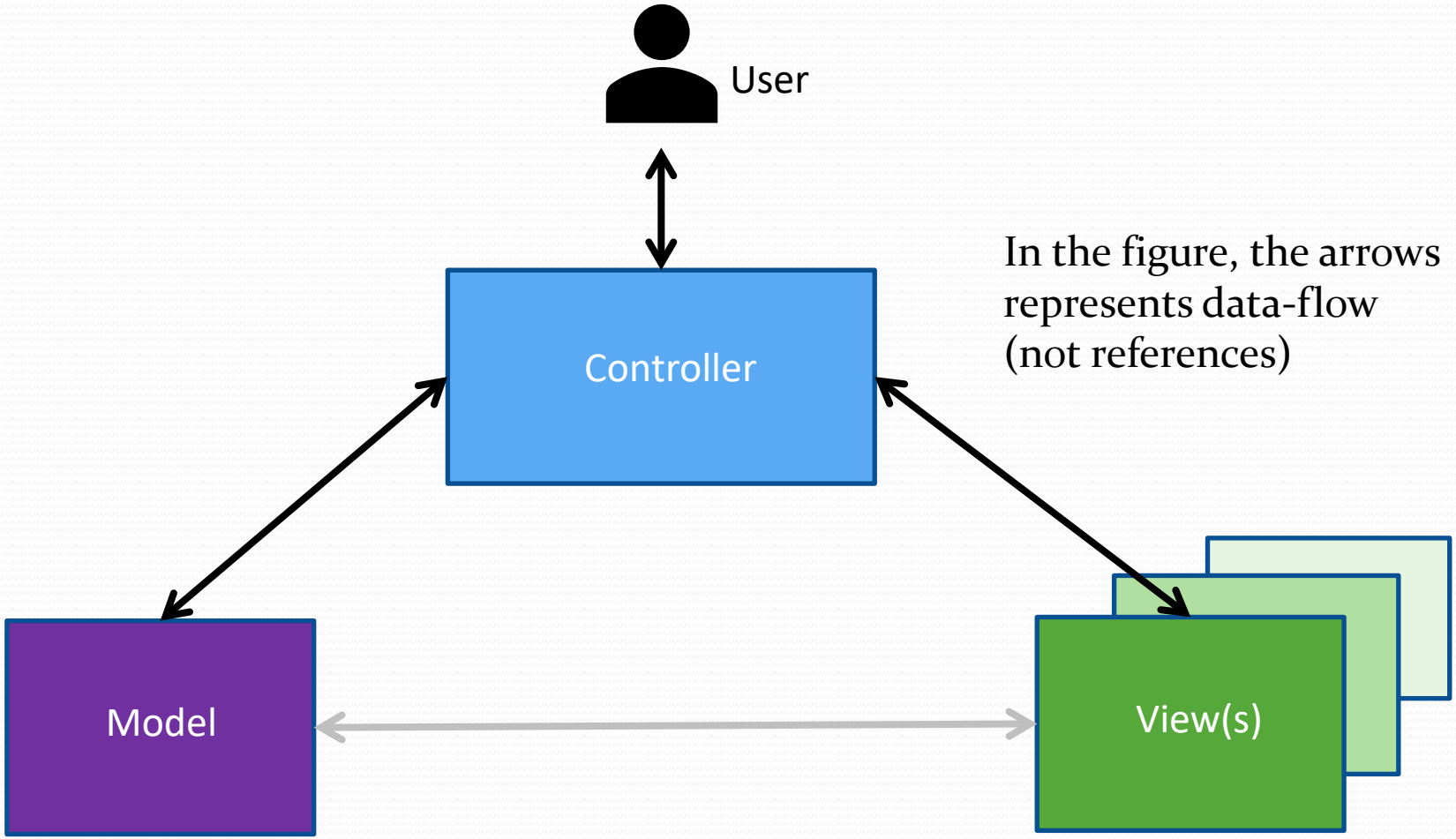
- *Model* - the model stores the data and should be independent with no references to other components.
- *Controller* - the controller must reference both the model and the view, and must retrieve data (input) and update the model. It works like a bridge between the two and translates and transfers data.
- *View* - dependent of the implementation, the view could reference the model to update itself, or only be updated through the controller

Note 1: even if the model and view don't reference other components, it can have it's own logic. The model can for example update itself at regular intervals

Note 2: a view may register user input and 'forward' it without doing anything more (dispatch an event).

Note 3: The *view* is defined as elements that display data related to the *model*. There may be other UI-components that have other types of functionality. These are usually implemented in the controller – or as separate GUI views/parts.

MVC design pattern

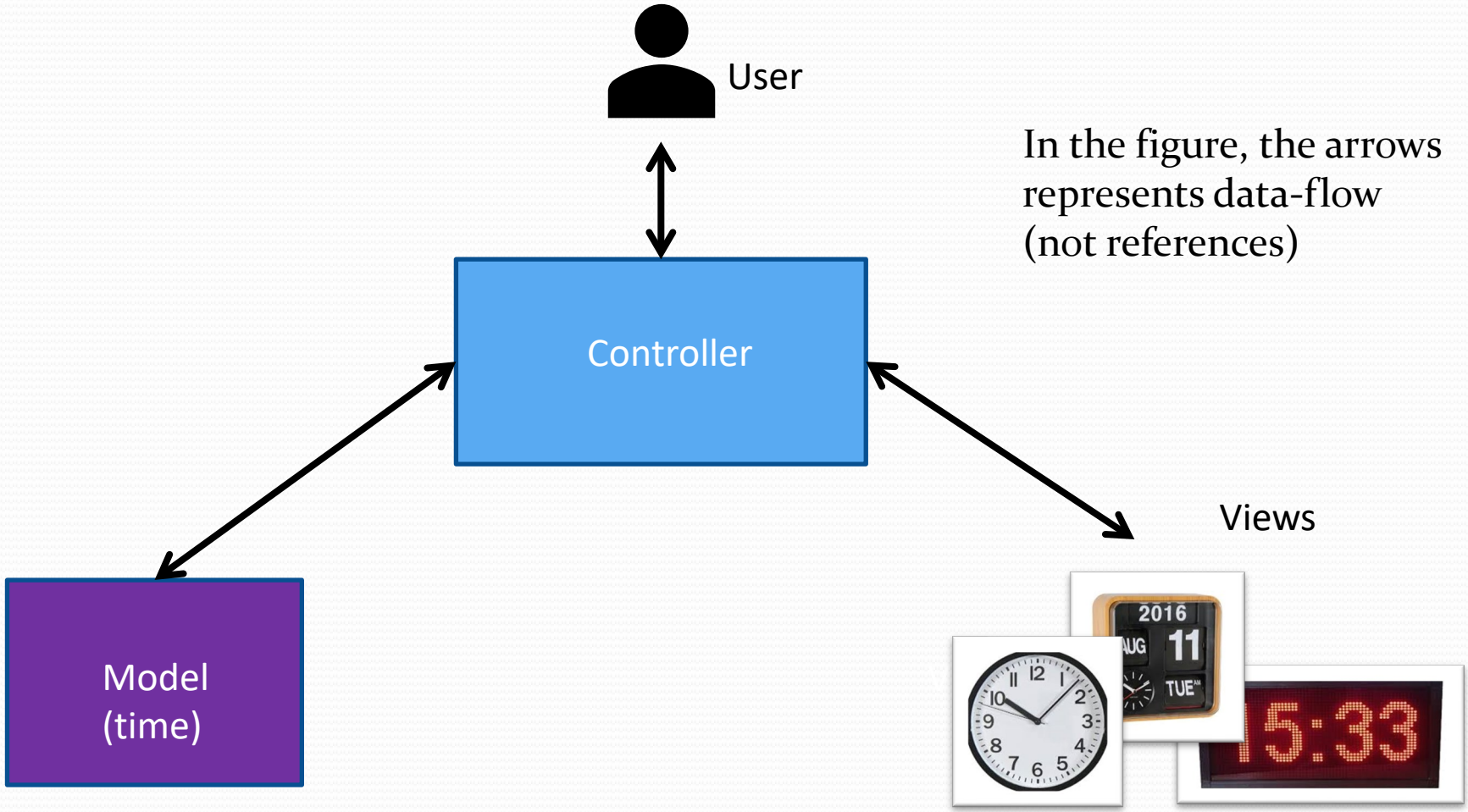


In the figure, the arrows represents data-flow (not references)

MVC design pattern cont.

- One of the advantages with the MVC pattern is that it makes it more easy to change how the data is displayed – we can for example have several versions of the view so that the data can be better understood by the user.
- The code is easier to maintain because of components with no or few references.
- This also means that the workload can be more easily assigned to multiple developers

MVC example - clock



In the figure, the arrows represents data-flow (not references)

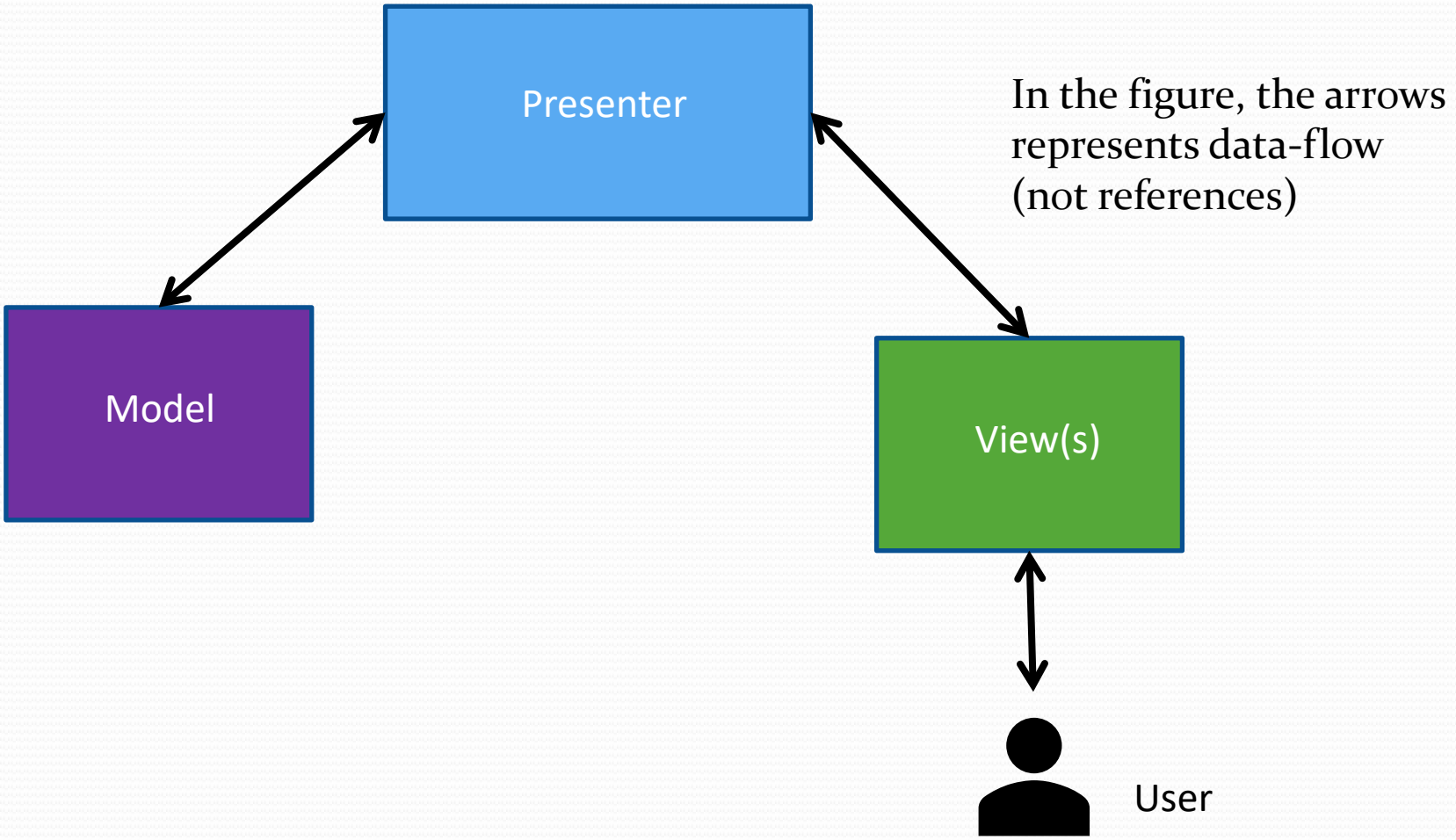
MVC in multi-layer web applications

- In a multi-layer web application, we have both a *client* and a *server*
- The tasks in MVC can be distributed between these two in mainly two ways:
  - *Thin client and webserver*: The server does all the tasks in MVC and returns fully formatted webpages so that the client doesn't need to do anything at all
  - *Client-server (using AJAX) and webservices*:
    - Usually we implement MVC both on the server and on the client:
      - On the **server**, the model is the database/storage, the view is the generated JSON or XML data, and the controller handles validation of data, user login etc.
      - On the **client**, the model represents the data/connection to the webservice, the view is the HTML/CSS that is shown in the browser window and the controller binds the model and the view two together - handling user input and parsing data between the model and the view.
    - The client may need to store data locally, e.g. keep data in the case the app goes offline. Then we often implement a model on the client that has the same API no matter if the client is online or offline. The model itself will take care of storing and retrieving data from the correct place (either server or local database), depending on if the client is online or offline.

MVP (model – view – presenter) design pattern

- The MVP pattern is very similar to the MVC pattern. The difference is that the view directly handles the interaction with the user. The presenter functions as the controller in MVC – it works like a bridge between the view and the model and translates and transfer data.
- In this type of pattern, it is more complicated to create different versions of the view, because of the complexity of user interaction elements.

MVP design pattern



In the figure, the arrows represents data-flow (not references)