

variety that exists in the original event types, events carry a topic object. The topic object carries a function type that is currently one of the following: data, state, action, or log. Additional function types can be added at a later point if a new function type is discovered. These function types correspond to the main event types that are divided in different ways through many different components in different languages. Further, a different topic interfaces exist to describe the type of data they carry. The interfaces that exist for topics currently include the following: definition, destruction, enable, instantiation, invocation, and value. These interfaces also serve as either a notification or a request. The definition topic type contains the metadata definition, which defines valid values that can occur in the event. Destruction events either notify of an object being destroyed or request that another component destroy an object. Enable allows view components or model components to be “disabled” to prevent changes. Instantiation is the opposite of destruction, relating to the creation of objects. Invocation requests a command be executed, or notifies that a command (or method) is being executed. Value notifies (or requests) the current value for a specific name.

[0530] For any event name, the combination of a function type, a topic type and the event serving as either a notification or a request allows a very small number of interfaces (4 function types, 6 topic types, and 2 topic roles) to replace all other events models. These interfaces lead to the “4.6.2” in HMVC 4.6.2.

[0531] The second part of this innovation is the use of a different structure for handling events between the model and view. The most popular approach is to use a model-view-controller pattern approach. In that approach, all view logic and view components are grouped into the view; all domain logic and business logic is grouped into the model; and all input handling (reacting to events to invoke methods on the model) is placed in the controller. Unfortunately, this does not allow for both small components and high reusability of code.

[0532] By placing all the component logic and display logic in the view, the view must handle the logic for getting and setting the data for each component in the view, as well as handling the interactions that occur between components. If selection of one item in a drop-down box causes the addition or removal of items in a different drop-down, the view must handle this. However, if the exact same components, arranged in the exact same layout for display, need to display a slightly different functionality, it may be impossible to reuse the view if the code is not moved into a separate component. If the view does move this logic into a separate component, in the MVC pattern, the view is still the responsibility of the view to setup and add the new component, which still prevents substitution of this logic, and still prevents the reuse of the view.

[0533] A solution is to create a new component in the architecture that is responsible for the interactions between components, while leaving the view responsible for individual components. This new object is the view controller. The view controller can be substituted without requiring any changes to the view, allowing complete reuse of a single component or a set of components. To allow substitution of smaller logical parts, a view controller consists of a hierarchy of small functional units, each handling one small group

of interactions between components. This allows componentization of display logic and reuse of common functionality even when applied to different sets of components. A view controller therefore meets the two goals of this invention: it reduces direct object connections (by connecting objects through event handling) and improves code reuse (by using a hierarchy of reusable components and moving small differences of display logic out of a view so the view can be reused without submodeling).

[0534] A similar situation and solution occurs in the model. By placing all the model logic and attributes in the model, the model must handle the logic for getting and setting the data for each attribute in the model, as well as handling the interactions that occur between attributes. In the case of an on-line order for products, the “order” model holds the “line items” attribute that can contain one or more items being purchased and the “total cost” attribute which is the sum of the cost of each line item. The order model is responsible for adding and removing items from the order as well as the logic to request a credit card number, calculate the total cost of the items, check for authorization, and bill the correct amount to the credit card. Even by separating the credit card and billing logic into a separate model, it is still the responsibility of the order model to setup and use the credit card authorization model. The replacement of credit card billing logic with logic for another method of payment (micro-transactions, electronic check, direct withdrawal) requires a rewrite of the order model to make it aware of these new methods. In the case of micro-transactions, even the calculations for the total cost of the order will be different.

[0535] A solution is to create a new component in the architecture that is responsible for interactions between attributes, while leaving the model responsible for the getting and setting attributes. This new object is the model controller. The model controller can be substituted without requiring any changes to the model, allowing complete reuse of a single attribute or a set of attributes. To allow substitution of smaller logical parts, a model controller consists of a hierarchy of small functional units, each handling one small group of interactions between attributes. This allows componentization of model logic and reuse of common functionality even when applied to different sets of attributes. A model controller therefore meets the two goals of this invention: it reduces direct object connections (by connecting objects through event handling) and improves code reuse (by using a hierarchy of reusable components and moving small differences of display logic out of a view so the view can be reused without model inheritance).

[0536] So the first adjustment for the HMV is to use a “model-view controller-view-input controller-model controller-model” pattern. Using the pattern of the first adjustment for the HMV, it becomes apparent that the original MVC pattern did not allow events from the model to be changed before being received by the view components, even though it did allow input events from the view to be altered. To give this flexibility, the input controller will be renamed “view out controller” and an equivalent “model out controller” are to be added in the same location on the model side to form a “model-model out controller-view in controller-view-view out controller-model in controller-model” pattern.