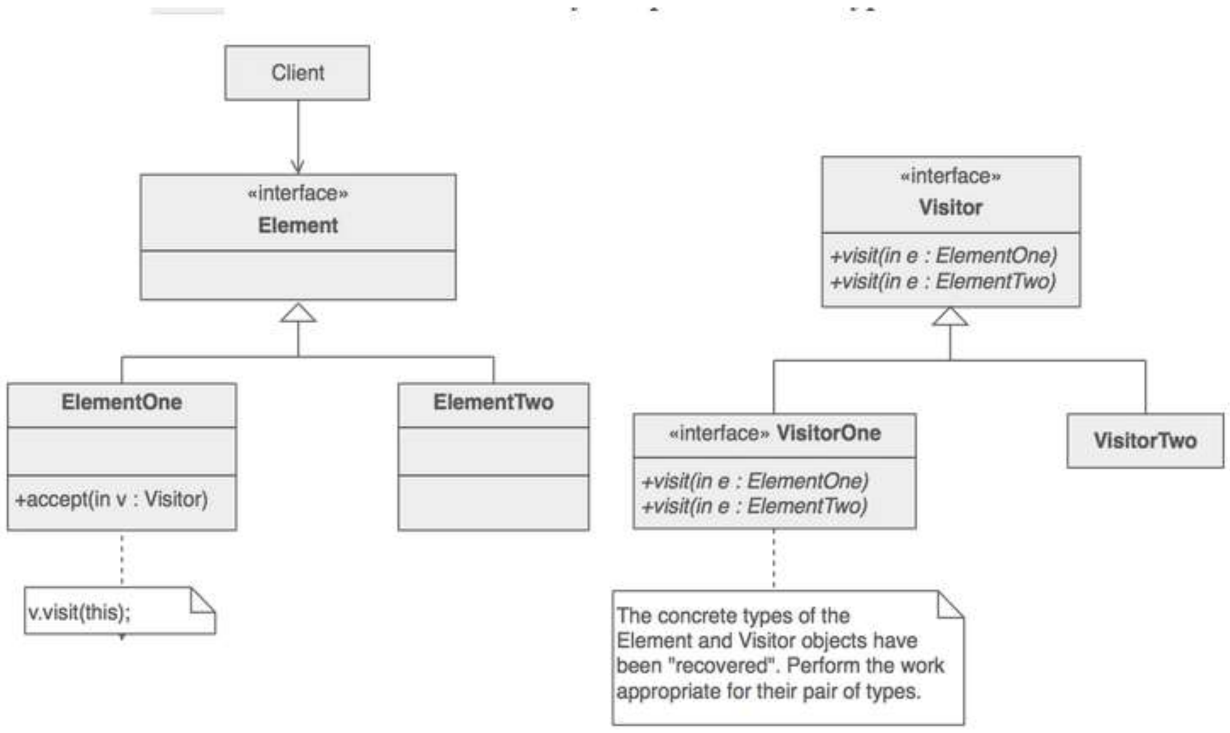
Game Architecture

In the architecture of the build2 we have used a design pattern which is called Visitor design pattern which below is explained in detail.

In many languages (including all of the C-style languages like Java, C#, C++, etc.), we only have single-dispatch directly available to us. In short, single-dispatch means that we can do polymorphism based on the specific type of object a method is called on. Sometimes, though, we want different behavior at run-time based on the specific type of two objects (for example: player card vs. "action type": a player card does something different depending on whether you're playing in the interrupt phase or in the turn phase). This is one of the situations where the Visitor pattern is typically useful. What many people call the "Visitor pattern" is actually the "Hierarchical visitor pattern", where we want to apply an algorithm on some data, with no knowledge on the structure of the data.

The basic layout for the Visitor pattern typically is that we have a "visitee" object and a "visitor" object. We call the "accept" method on the visitee, with the visitor as a parameter. This is usually the first dispatch, and we typically find visitee-specific work done here. Once the visitee is done preparing itself, it calls the "visit" method on the visitor, passing itself as a parameter. The specific visitor decides what to do with the visitee, sometimes having different overloads of some private method to do different logic depending on the specific type of visitee. A simpler version just determines what specific type the visitee is, and does stuff based on that.


Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
ElementTwo
v.visit(this);
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface» VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
VisitorTwo
The concrete types of the Element and Visitor objects have been "recovered". Perform the work appropriate for their pair of types.

As is typical in the Visitor pattern, we mainly work with the "visitees" and try not to touch the "visitor" too much. The interface is game.action.scaffold.IActionVisitee. On top of the required "accept", there are the following methods:

- **getType()** : helps identify the type of action (player card, player card icon/text, random event, etc.), so that the visitor knows what to do with a specific visitee.

- **getActions()** : the specific actions that must be executed. For example, the first icon on a card will be the sole return value of "getActions" on a player card, whereas a start-of-turn will offer all playable cards, all city cards, etc. as possible actions.

- **registerAction()** : tells the visitee that an action is completed, so that the visitee can update its data (for example, that the first icon on a card has been dealt with, so that the second icon is now offered as the action the next time "getActions" is called on that card.

The game.action.scaffold.IAction interface isn't that interesting by itself, it simply knows how to describe itself. It's mainly a placeholder for the sub-types: IExecutableAction, IOptionalAction and IConditionalAction. IExecutableAction offers an execute method; an example of this would be the "draw random card" action on player cards. IOptionalAction would allow the user to choose whether to execute the action or not (all other icons on player cards fall into this category). IConditionalAction must have their pre-conditions cleared before they can be executed.

The "execute" method on IAction returns a collection (Queue, if memory serves) of IActionDataGatherer. If no additional data is required, the collection is empty (NOT null). If we need more data to execute the action (for example, if we want to place a minion on the board, we'll need to know the area), this is how the information will get collected.

Some data-gathering actions affect other players (for example, taking money as part of the Mr.Boggis card). Such actions may be interrupted. For these situations, there is the IInterruptibleDataGatherer. Because the easiest way to represent the interruption is as a separate visitor, implementations of IInterruptibleDataGatherer must implement IActionDataGatherer (because that's where the action starts), IOptionalAction (because the targeted player may decide not to do anything) and IActionVisitee.

The Visitor part simply looks at the types and decides what to do entirely based on that. Once coded, it should not ever require changing, unless we add new types.

So here's how to represent a turn:

- ✓ Create a "StartOfTurn" visitee and a corresponding visitor; call the "accept" method (as part of it’s accept method, it will check to see whether winning conditions have already been met)

- ✓ List all possible sub-visitees: city area cards (if any), playable cards if any.

- ✓ Once we've completed the chain of actions thus started, the player draws cards to replenish, if required.

- ✓ Create a "StartOfTurn" visitee corresponding to the next player, and call its "accept" method.