Notes on UML

L. Laneve, A. Lopardo, R. Mencucci

March 2019

1 Effect

An `Effect` represents the atomic element of an interaction between a human and the game. Effects implement a *strategy* pattern to represent the gameplay interaction associated with them.

Multiple atomic effects may be combined to produce more complex effects. Such composition takes place using the opportune collections `EffectChoiche`, `EffectSequence` and `EffectSet`.

One use of `Effect` is to be part of `Weapon`s as a representation of the interactions they require. Interstingly, some weapons have peculiar functionality and interactions never used in any other part of gameplay. We have found inconvenient introducing many different atomic effects only to use them once. Our choiche is to introcude one peculiar effect for each one of these weapons and have it implement all the related fuctionality, regardless of the conceptual non-atomicity of it.

With these choiches, simple weapons like Heatseeker are implemented with atomic effects alone, more complex weapons like Plasma Gun are implemented with a composition of atomic effects and only a few problematic weapons like Vortex are implemented with their own peculiar `Effect`.

Another use of `Effect`s is to represent the individual components of a move in a turn, so that they can easily be composed together to represent the complete gameplay actions.

The modular nature of `Effect` allows us to encode cards in an asset file independent from the codebase. This gives us the flexibility to modify them or add new cards in the future, for example in a new expansion, with minimal code changes.

1.1 EffectVisitor

The `EffectVisitor` interface introduces an abstraction layer between `Action` and `Effect`, greatly reducing the coupling between the two classes.

2 Action

An `Action` expresses the complex and complete interaction with a player that eventually produces one or more `GameChange`s. It can be thought of as the main engine for interacting with human payers.

Actions are implemented using an *observer* pattern. They are part of the model, as they express foundamental mechanics strictly connected with the applicative domain of the application and are in charge of verifying that the interactions do not violate the integrity constraints of the model.

Actions use synchronus requests to interact with the player. This greatly simplifies the implementation of them and allows for a clearer representation of the real-game behaviour they express.