New Techno War

Development guide

Claudio Bonesana

Last revision: August 14, 2020

Contents

Chapter 1

Introduction

This is development guide is intended to contains diagrams and explanation that are difficult to explain using in code comments.

Although the majority of the code is covered by comments and follows a descriptive code-style, some diagrams are needed, hence the scope of this document.

Chapter 2

GameManager utility class

This is an utility class. Its goal is to update the `GameState` in a definitive or not way. To manipulate in a safe way the `GameState`, use an instance of this class. This class does not contains any information: everything change is an update to the `GameState` object.

For testing purposes, i.e. when an agent wants to see the consequences of an action, use the `activate()` method. This method will create *a deep copy* of the `GameState` object before apply the changes to it.

Otherwise, to apply in a definitive way the changes of an action to the state, use the `step()` method.

The methods `buildMovements()`, `buildShoots()`, `buildResponses()`, `buildActionsForFigure()`, and `buildActions()` can be used to create different kind of actions.

Other methods are used by the `MatchManager` class to control the development of a match.

Chapter 3

MatchManager class state-machine

A *match* is defined as following:

- The match is a collection of turns.
- In each turn the players activate all their units once and they can respond with their units at maximum once. First goes red player, then blue player.
- When all the figures are activated, the match goes to the next turn and the unit status is reset.

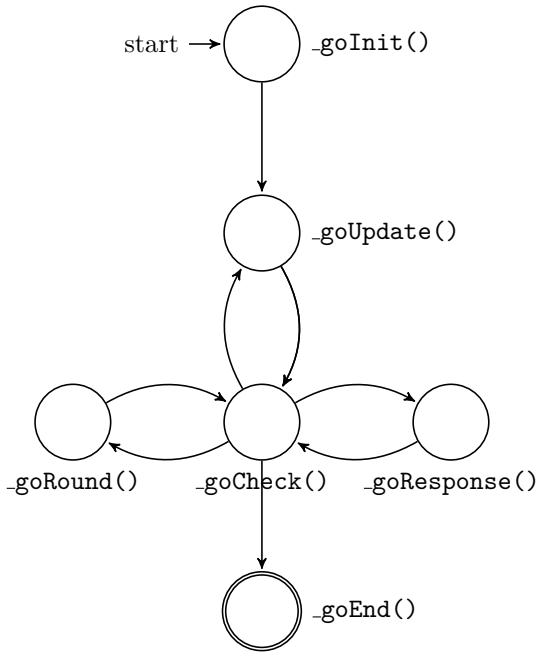

Figure 3.1: State transition diagram of the MatchManager class.

Figure 3.1 shows the state transition diagram of the `MatchManager` class. This class has the main purpose of manage a whole match. The structure of the match is decomposed into a state machine.

For the creation of this object, it is required to pass as argument the `string value` of the classes to instantiate. This is a consequence of the web interface and the interaction with humans. Check the documentation of the `__init__()` method for further details.

The state machine begin with an **initialization** step. Then the first action is an update action. After that, the state machine goes in *check* mode: it checks the current status and decide where to go next. The possible outcomes are four: if there was a *Shoot* or *Move* action (but not a response), a **response** is possible; if there are still figures that can be activated, there will be a **round** step; if the goal of the scenario is achieved, then the match will **end**; otherwise, the only possible remaining state is a turn **update**. All other states step back to **check** with the exception of the **end** state.

The class offers two utility methods:

- `nextStep()` will move the state-machine forward of one step.
- `nextTurn()` will move the state-machine forward until the turn change (an update happens).

Chapter 4

Players and Agents

The idea for the *Agents* and the *Players* is to use a common interface that can be easily used by the `MatchManager`. This interface is composed by two methods:

- `chooseAction()`,
- `chooseResponse()`.

Although they look similar, the need to be differentiated because the *response* state offers the possibility to skip the action. This possibility is something that the Player or Agent need to keep in mind during the design stage.

In the case that something is not doable, i.e. skip the *response* action or if a figure does not have action doable, please raise the `ValueError` exception in order to inform the *MatchManager* that there will not be an action available.

Please, check the `agents.players.PlayerDummy` class for an example of implementation of a random player.

Chapter 5

Bibliography