MecaTeam Framework: An Infrastructure for the Development of Soccer Agents for Simulated Robots

Orivaldo Vieira Santana Júnior,
Christina von Flach Garcia Chavez and Augusto Loureiro da Costa

***Abstract*— This paper presents the MecaTeam framework, a solution to reduce the effort on developing new soccer teams of robots for the 2D simulation category of the RoboCup. MecaTeam is an object-oriented framework based on features of two robot soccer teams: the MecaTeam 2006 and Uva Trilearn. The architecture of the proposed framework is presented and aspects of its use are discussed. Besides facilitating the development of new teams, the use of the MecaTeam framework may decrease the impact of changes in chunks of related code. Finally, the MecaTeam framework can be used by new researchers interested in simulated robots for soccer games.**

I. INTRODUCTION

In 1995, the Robot Soccer Games World Cup (**RoboCup**) was proposed as a new standard problem for Artificial Intelligence (AI), Robotics and related fields, in which soccer games are used for developing research in various branches such as Autonomous Agents, Multi-Agent Systems, Knowledge Acquisition, Real-Time Reasoning and Sensors Fusion. The most important goal of the *RoboCup* initiative is to promote technological advances to the society. In 1997, the first *RoboCup* was held in Nagoya, Japan, and since then, annual competitions were organized in different places [1], [2].

The implementation of intelligent agents for simulated robot soccer is not a trivial task. In order to track the progress of the oldest teams in the robot world cup, new teams in the league generally reuse code of previously successful teams. For instance, the UvA Trilearn Team [3], the winner of soccer competition (Simulation League) in RoboCup 2003, is a simulated robots soccer team that provides its source code on the Internet.

The MecaTeam Framework is an object-oriented (OO) infrastructure that defines the intra-agent architecture of autonomous agents for the RoboCup 2D soccer simulation category. OO frameworks organize collaborating classes with predefined cooperation among them and indicate extension points to adapt their behavior. Whereas programs built on top of reusable classes/libraries reuse only their source code, systems built on top of a framework exploit source code and architecture reuse. Thus, the MecaTeam framework promotes large-grain reuse of intra-agent architecture instead of simple reuse of code from another soccer team.

The base code for the MecaTeam framework is the source code from MecaTeam, a robot soccer team that falls into the RoboCup simulation 2D category. MecaTeam 2006 uses a multi-agent system in the implementation of the distributed control for a multi-robot system, in which each robot is controlled by an autonomous agent [4]. The agent's automatic reasoning is supported by a production rule-based system [5]. The MecaTeam Framework is the first OO framework developed for this application domain, with intense reuse of the UvA Trilearn team code. The UvA Trilearn team provides well-documented code, and therefore it has been reused in the lower layers of the MecaTeam agent architecture. The Expert-Coop++ [6] is responsible for supporting intelligence and reasoning for MecaTeam 2006 agents. Expert-Coop++ is an OO library that supports the development of multi-agent systems that work under real-time constraints [6]. This library is implemented in C++ and includes several classes that comprise the MecaTeam agent architecture as well as the support for knowledge-based systems.

This paper is organized as follows. Section II presents some important Software Engineering background. Section III presents design decisions concerning the construction of the MecaTeam Framework. Section IV illustrates the applicability and relevance of framework, by instantiating it into three different scenarios of increasing complexity and Section V discusses some results. Finally, some conclusions are drawn in section VI.

II. BACKGROUND

This section presents some background concepts about OO frameworks and design patterns, and explains their usage in the MecaTeam Framework construction.

A. Frameworks

The MecaTeam Framework consists of a collection of several components that have a pre-defined cooperation between them. The points where changes or adaptations can be made are called *hot-spots*, also known as refinement points or predefined extension points [7]. The components that form the fixed part are called *frozen-spots* [8] and define the invariable aspects for the soccer teams implementation of the simulated robots in 2D category. Fig. 1 shows the *frozen-spots* (the dark-gray area) and the *hot-spots* (the light-gray area) of the MecaTeam Framework.

The MecaTeam Framework has a generic design for a whole family of applications aimed at the *soccer server* simulator [9], which solves problems such as synchronization between the simulator and the robots soccer team, environment modeling. Moreover, it supports many ready

The authors are with the Department of Computer Science - Math Institute, Federal University of Bahia (UFBA). Salvador, 40170-110, BA, Brasil {orivajr, flach}@dcc.ufba.br e augusto.loureiro@ufba.br

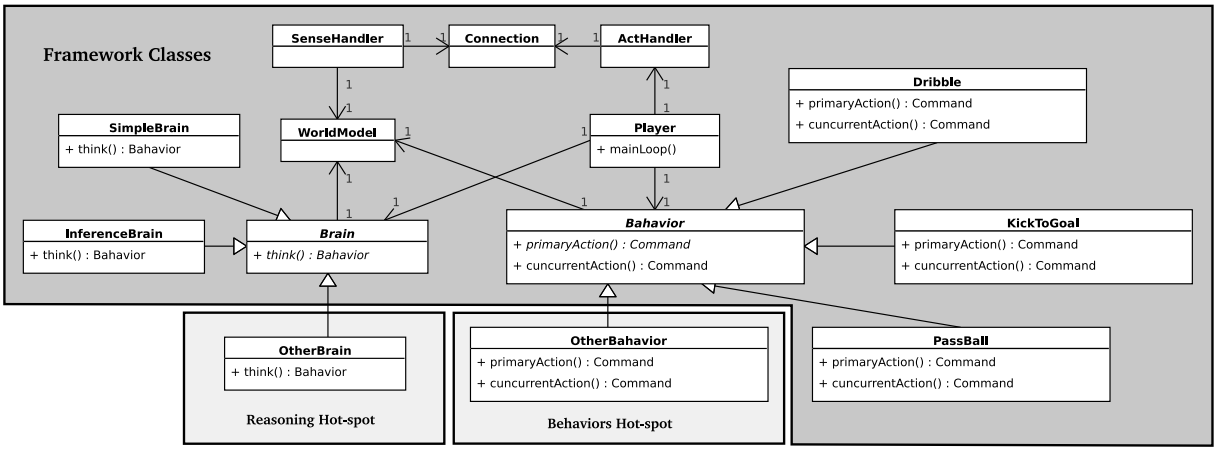

Fig. 1. The MecaTeam Framework: *frozen-spots* (the dark-gray area) and *hot-spots* (the light-gray area)

skills such as kicking, passing ball, dribble, and mark. This generic design pre-defines a general architecture, that is, the composition and interaction of components. New soccer teams can be generated by providing custom-behavior at the pre-defined *hot-spots* of the MecaTeam Framework.

The MecaTeam Framework instantiation, which is the process of implementing specific codes in *hot-spots* [7], can be achieved through two basic techniques: inheritance and composition. In instantiation by inheritance, the abstract class *Brain* is specialized by new subclasses. For the framework adaptation by composition, it is only necessary to know the external interfaces of the components and there is no need to know their implementation details. These components consists of classes that implement the behaviors of a soccer player.

The MecaTeam Framework development has become feasible due to three years of experience in research with simulated robots soccer of the student that developed this work. The costs are significantly higher when compared to developing a specific soccer team, so frameworks represent a long-term investment, which produce more effect when many teams are developed with use of this framework [7].

B. Design Patterns

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design [10]. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. An OO framework typically uses several kinds of design patterns.

Design patterns document several kinds of information and are organized in several parts or sections. The design pattern name identifies key aspects useful to create a reusable OO design. Each design pattern indicates the classes and participants instances, their roles and collaborations, and also the distribution of responsibility. Each design pattern focuses on a specific problem or characteristic of the OO design [10]. We present two design patterns used in the MecaTeam Framework – *Strategy* and *Singleton* – that make the MecaTeam Framework more reusable.

The intent of the *Strategy* pattern is to define a family of algorithms and encapsulate each of them, allowing that one may be replaced by another [10]. The MecaTeam Framework *Strategy* provides support for the reasoning strategy extension point. This enables the creation of several agents, each with its own reasoning strategy, by only extending and implementing the *Brain* class.

The intention of *Singleton* is to ensure that a particular class has only one instance and provide a global access point for this instance [10]. In the MecaTeam Framework, *Singleton* is applied, for example, in the class responsible for storing the world model, because there must be exactly one instance of this class accessible to various parts of MecaTeam Framework.

Other concepts related to OO frameworks and design patterns (a key to understanding this work), are presented in more detail in the section III: The MecaTeam Framework. This work deals with the reengineering of MecaTeam 2006, in order to transform it into an OO framework, thus promoting enhanced comprehension and reuse of the MecaTeam Agent by other developers. The *Strategy* pattern is taken as a basis for the MecaTeam Framework design, facilitating the change of reasoning strategies.

III. The MecaTeam Framework

This section presents the MecaTeam Framework, an OO infrastructure that defines the intra-agent architecture of soccer autonomous agents for the RoboCup 2D simulation category. It provides the documentation indicating extension points and procedures to assist reuse by others. First, it describes the problems identified in MecaTeam 2006, which motivated this work. Then, the framework is introduced in terms of its underlying architecture and extension points.

A. Problems in MecaTeam 2006

The problems identified made it difficult to understand, modify and reuse the MecaTeam 2006 Agent [4]. The main difficulties were to incorporate new behaviors, to incorporate

new reasoning strategies, to evaluate the impact of a change in MecaTeam related code chunks and to facilitate the reuse of MecaTeam code. These difficulties are explained below.

- Difficulty to incorporate new behaviors.
  The UvA code, despite of being well structured and object-oriented, offers some difficulties in the implementation of more elaborate behaviors. To implement a new behavior or a more complex skill on UvA code, such as mark opponent using potential fields, it is necessary to modify the *BasicPlayer* class. This class is a example of "large class" [11] which has 42 methods and 2903 lines. In a development as a team, after the implementation of some methods, is more difficult to manage the changes, since each member of the development team needs to know this large class.
- Difficulty to incorporate new strategies of reasoning.
  In UvA code, the reasoning strategy definition is codified in the *Player* class, into the method *deMeer5*, mixed with other functionalities. Implement a new strategy means changing a base system class. To modify the UvA code, the developer needs to know the entire code, since changes can affect the agent operation or even make it stop working. The method *deMeer5*, for example, deals with the addition of commands in the queue to be sent to the *soccer server*. In this queue, it can only be added at most a primary command. The *deMeer5* implementation has to deal with that restriction imposed by the *soccer server*, in each cycle.
- Difficulty in evaluating the impact of a change in chunks of related MecaTeam code.
  In particular, there is difficulty in understanding how a change in a class or method can demand a corresponding change in others chunks of MecaTeam code. This can be noticed in the very strong relationship between *BasicPlayer* and *Player*. The *Player* class inherits all the skills of *BasicPlayer* and combines these skills to get the desired behavior. However, this combination is mixed with the implementation of the reasoning strategy.
- Need to facilitate the reuse of MecaTeam.
  The MecaTeam 2006 is not structured as a framework for reuse. Thus, there is no indication of the places where developers should modify it or define new functionality, and many classes are arbitrarily exposed, without guidance on what can be reused as *black-box*, *white-box*, etc. For new developers in the MecaTeam research group, it is very important that the code is well organized and documented, because in addition to the inherent software complexity, the code is developed by students with variable permanence in the project. The OO framework promotes reuse in a larger scale, because it defines a semi-complete application, which only needs classes defined by the user, at pre-specified spots, in order to create a complete application.

B. The Framework Architecture

The architecture of the MecaTeam Framework, illustrated in Fig. 2, is based on the architecture of the UvA Trilearn [12]. The modules of the MecaTeam Framework can be divided into three types: core, incomplete and complete. The core modules are immutable and will be the same in all applications created by instantiating the MecaTeam Framework. The incomplete modules, which need to be defined by the framework user, correspond to *white-box* features in the MecaTeam Framework. The complete modules, ready for use, characterize the MecaTeam Framework *black-box* features. The following sections describe how these modules were built.

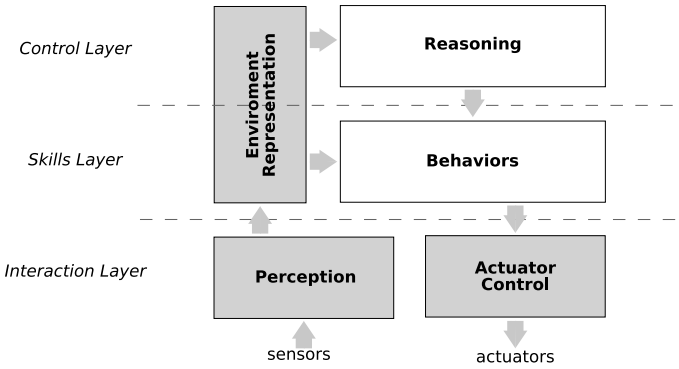

Fig. 2. The MecaTeam Framework Architecture

1) The Core Modules: The **core modules** are *frozen-spots* and constitute the invariable part of a soccer agent of simulated robots. They were extracted from the base of the *UvA Trilearn Team*. The functions of these modules are: to interact with the simulator and generate a representation of the environment. For this, they need be synchronized with the simulator, receiving and sending messages at the appropriate time.

The **perception module** is responsible for receiving the perceptions of the environment (coming from simulator in string messages), analyzing them and sending the results of this analysis for the **environment representation module**. The **actuators control module** is responsible for triggering the robot actuators, sending commands in string messages to the simulator. The classes of the environment representation module contain the most updated information of all objects in robots soccer field. Their operation is similar to human memory, which stores information about feelings (heard, seen, etc.).

The class that relates all these core modules of the framework is called *Agent*. It was created from the code contained in the *main* function of the original UvA Trilearn code. The *execute* method has a parameter of the *Brain* type class. This allows any reasoning strategy implemented by the user, from the specialization of *Brain*, to be incorporated to agents.

2) Incomplete Modules: The incomplete modules have the variation points (*hot-spots*) of the MecaTeam Framework.

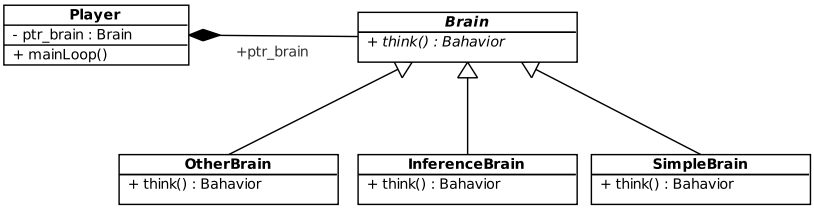

Fig. 3. Strategy Pattern used in the MecaTeam Framework

The framework has two variation points: one to implement reasoning strategies and other to implement behaviors.

The **reasoning module** has a semi-ready structure for a robot soccer agent, with which the user needs only to spend effort with the implementation of the reasoning strategy. The separation between reasoning strategy and the application core was implemented using polymorphism, more specifically the *Strategy* design pattern.

The *Strategy* implemented in MecaTeam Framework has basically three elements: *Player*, *Brain* and *ConcreteBrain*. The *Player* defines the agent operation algorithm, the way he feels, thinks and acts. A variation of this algorithm is in the way of thinking and acting; who defines how the agent thinks and acts is the *ConcreteBrain*. Through the reasoning strategy, the *ConcreteBrain* chooses the most appropriate behavior for a particular state of the environment. Thus, the method *think* of *ConcreteBrain* must return a behavior. Fig. 3 depicts this modeling, and *InferenceBrain*, *SimpleBrain and OtherBrain* are concrete classes (*ConcreteBrain*) that specialize the *Brain* class.

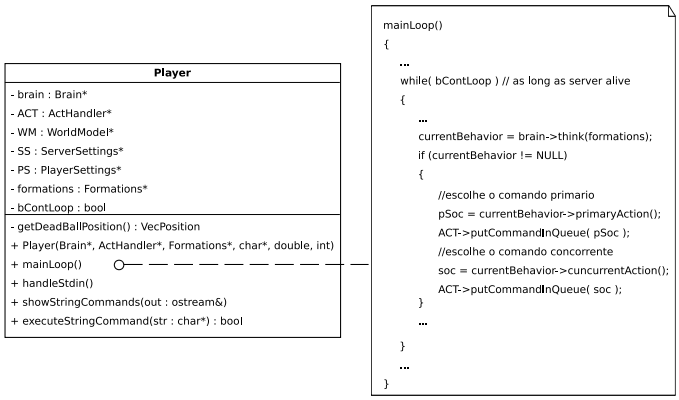

Fig. 4. UvA Trilearn Player class and a code chunk of mainLoop method.

The *Player* class (Fig. 4) was changed in the *mainLoop* and *deMeer5* methods to improve its structure. The method which contained the reasoning strategy, *deMeer5*, was replaced by a call to the *think* method of the *Brain* abstract class, featuring the *hot-spot* of the reasoning module. The *mainLoop* method of the *Player* class contains a loop that runs until the end of the connection with the simulator. It handles the basic algorithm of the agent (feel, think and act).

The **behavior** module consists of the *Behavior* class and all subclasses generated from it. The *Behavior* class is an adaptation of *BasicPlayer*, which contains all the skills of UvA Trilearn. In this class, two virtual methods were added – one to return the primary command and another to return a concurrent command. With these virtual methods, the *Player* class can receive any behavior from the *Brain* class. Thus, the *Player* class, at each cycle, performs the virtual methods of *Behavior* that return one primary and other concurrent command.

3) Completes Modules: The complete modules are those that have been generated from an incomplete module (section III-B.2). The framework **reasoning modules** are: *SimpleBrain, GoalieBrain* and *InferenceBrain*. The **behavior modules** are: *KickToGoal, HoldBall, InterceptBall, GoStrategicPosition, MarkOpponent, PassBall, SearchBall and Teleport*. These modules reinforce the *black-box* features of the MecaTeam Framework.

The *SimpleBrain* is a brain for a simple player, extracted from the *deMeer5* method of the *Player* class, which contains the implementation of the UvA Trilearn reasoning strategy. In *deMeer5*, the environment state identification and the behavior implementation are coded in the same scope. In this way, it is hard to distinguish what is a behavior from what is the environment state identification.

With *SimpleBrain* creation, the implementation of behaviors has been separated into different classes. Thus, the *think* method of *SimpleBrain* contains only the environment state identification and the association of this state to a behavior. In each state identified by *SimpleBrain*, *currentBehavior* is associated to a behavior and sent to the *Player* class. These behaviors (Fig. 5) are: *KickToGoal, InterceptBall, GoStrategicPosition, SearchBall and Teleport*. They were generated from the restructuring of the *deMeer5* method.

With the use of *Behavior* class, the skills of UvA Trilearn will always be encapsulated within behaviors. This facilitates the understanding, maintenance and evolution of the code, and the implementation of various behaviors with different techniques of artificial intelligence.

IV. Using the Framework

To illustrate the applicability and relevance of the framework, we present three different scenarios in which it can be instantiated, with increasing levels of complexity.

The simplest way to reuse the framework is using the existing components. Thus, about five lines of code will be needed to implement an agent. Also, there are various behaviors already implemented that can be reused by the user reasoning module so that users can implement their own reasoning module, or use the available one.

An interesting framework feature is the independence of how each extra module can be implemented. The user can implement a new behavior module using a particular Artificial Intelligence technique and implement the module using another completely different reasoning technique. For example, the user can implement a behavior using Neural Networks and the reasoning strategy using Knowledge Based Systems.

A. First scenario:Black-box Reuse

The simplest way to use the framework is the reuse of a complete module (*black-box* reuse), generated from the *Brain* class: *SimpleBrain*, *GoalieBrain* or *InferenceBrain*. To create a goalkeeper, for example, the user must implement a file with the function *main* and include the files *Agent.h* and *GoalieBrain.h*. Then, he should create objects from the *Agent* class and the *GoalieBrain* class (responsible for the goalkeeper reasoning strategy). The *execute* method of the *Agent* class receives as a parameter the *GoalieBrain* instance.

B. Second scenario: Implementing a New Brain

Suppose that the user wants to implement an agent controlled by fuzzy logic. To that end, the user needs to follow two steps to use the strategy reasoning *hot-spot* defined by the framework.

The first step is to prepare the file header containing the specifications of the *FuzzyBrain* class, following the instructions below:

- Include the *Brain* header file;
- Declare the *FuzzyBrain* concrete class that specializes *Brain*;
- Declare the attributes to the fuzzy reasoning;
- Declare the *think* method;
- Declare the constructor.

The second step is to generate a file with the implementation of the desired class, in this case *FuzzyBrain*, according to the following rules:

- The *FuzzyBrain* constructor should initialize its attributes and call the super class constructor, because this class contains the world model and behavior initializations;
- The implementation of the reasoning strategy with *fuzzy* logic in the *think* method should return a behavior;
- The world model must be referenced through the *WM* pointer, an attribute that is inherited from the *Brain* class.

C. Third scenario: Implementing a New Behavior

Suppose that the user wants to implement a opponent marking behavior using potential fields. For this, the user must follow two steps to use the behavior *hot-spot*.

The first step is to prepare the header file containing the class specifications, called *PotentialFieldsMark*. At this step, the user must follow the instructions below:

- Include the file *Behavior.h*;
- Declare the *PotentialFieldsMark* concrete class as *Behavior* specialization;
- Declare the *primaryAction* and *concurrenteAction* methods.

The implementation file for the *PotentialFieldsMark* class must contain at least two methods: *primaryAction* and *concurrentAction*. The *primaryAction* method uses the necessary skills for marking behavior that returns a primary command, whereas the *concurrentAction* method uses the necessary skills for marking behavior that returns a concurrent command. These two methods together form the behavior of marking on potential fields.

V. Results

In order to provide a preliminary evaluation of the results from instantiating the MecaTeam Framework, ten matches were processed with the three best teams of the Brazilian Robotics Competition in 2007 against MecaTeam. The new MecaTeam version, implemented with the MecaTeam Framework, played against his older version, developed without the framework support (called MecaTeam 2007); these two MecaTeam versions (the new and the older) also played against other teams. Thus, 40 matches were measured for each of the two versions. The MecaTeam Framework got 24 wins, 9 draws and 7 defeats and MecaTeam 2007 got 2 wins, 33 defeats and 5 draws. This shows that it is feasible implement new teams with the framework and obtain good results.

Comparing MecaTeam 2007 with applications developed with the MecaTeam Framework, we have the following results in favor of the MecaTeam Framework instantiations: 8 wins, 1 draw, 1 defeat, 62 goals made and 16 suffered. These good results come not only to the use of the framework, but also to an improvement in the MecaTeam Framework reasoning strategy and behaviors. Whereas the MecaTeam 2007 uses a simple reasoning strategy, with only a reasoning layer, the MecaTeam Framework uses one more reasoning layer than the MecaTeam 2007, allowing the use of plans. Besides a more sophisticated reasoning strategy, the MecaTeam Framework defines more behaviors and improve others.

With the framework, the developer will deal with a smaller amount of code, that is, a smaller number of classes and lines of code. This becomes more evident when we compare the size of MecaTeam 2007 and the MecaTeam Framework. The MecaTeam 2007 has about 57 classes and 19202 exposed lines of code, and the framework has about 68 classes and 18642 lines of code isolated. One MecaTeam Framework instantiation has 6 classes and 852 lines of code. Thus, the developer needs only to know some class interfaces of the framework without worrying about their internal implementation.

VI. Conclusions

This paper presented the MecaTeam Framework, an OO infrastructure that defines the intra-agent architecture of autonomous agents of simulated robots for soccer league in the RoboCup simulation 2D category. The MecaTeam Framework is concerned with solving some problems of the

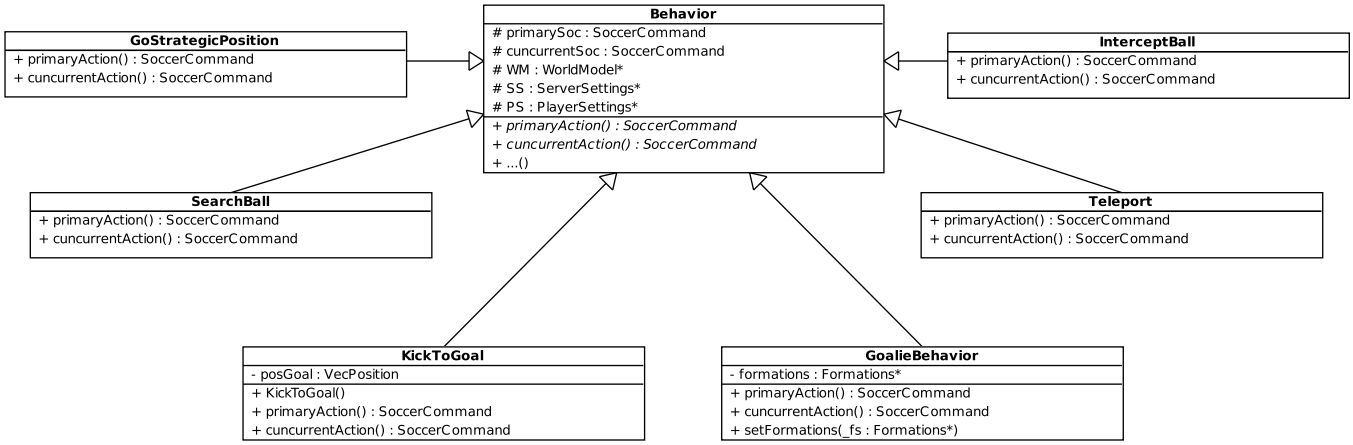

Fig. 5. Refining the behaviors of the MecaTeam Framework.

MecaTeam 2006 agent, such as: the need to facilitate the reuse of MecaTeam; difficulty in incorporating new reasoning strategies, difficulty in incorporating new behaviors and difficulty to assess the impact of a change in the related chunks of MecaTeam code.

A. Contributions

The modularity offered by the MecaTeam Framework decreases the impact of changes in related codes chunks. The framework extension points facilitate the reuse, separating the framework core from the reasoning strategies and behavior implementations. The framework provides a common base to all robot soccer agents. The framework also provides a documentation indicating extension points and procedures to help reuse.

The MecaTeam Framework can be reused by new researchers in the area of soccer simulated robots. The framework improves the quality of agents produced, because its core is composed of the UvA base team, champion in 2003 and which went through various tests and validations. With the reuse promoted by the MecaTeam Framework, the new teams will have their players ready faster and with decreased costs.

REFERENCES

[1] H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada, "The robocup synthetic agent challenge, 97," *International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997, nagoya, Japan.

[2] H. Kitano, "Robocup: The robot world cup initiative," *in Proc. of The First International Conference on Autonomous Agent (Agents-97))*, 1997, marina del Ray, The ACM Press.

[3] R. d. Boer and J. R. Kok, "The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team," Master's thesis, University of Amsterdam, The Netherlands, 2002.

[4] O. V. de Santana Júnior, J. P. R. P. Sousa, M. S. Linder, and A. L. Costa, "Mecateam: Um sistema multiagente para o futebol de robôs simulado baseado no agente autônomo concorrente," *Encontro de Robótica Inteligente / XXVI Congresso da Sociedade Brasileira de Computação*, pp. 146–152, 2006.

[5] O. V. de Santana Júnior and A. L. Costa, "Mecateam 2006: Um sistema multiagente reativo para futebol de robôs simulados," *VII Escola Regional de Computação Bahia-Alagoas-Sergipe*, p. midia digital, 2007.

[6] A. L. Costa, G. Bittencourt, E. M. N. Gonçalves, and L. R. Silva, "Expert-coop++: Ambiente para desenvolvimento de sistemas multi-agente," *IV ENIA Encontro Nacional de Inteligência Artificial*, pp. 597–606, Brasil, Campinas, 2 a 8 de agosto 2003, xXIII Congresso da Sociedade Brasileira de Computação.

[7] W. Pree, *Framework Patterns*. 71 West 23rd Street, New York: SIGS Books & Multimedia, 1996.

[8] I. M. Filho, "A documentação e a instanciação de frameworks orientados a objetos," Ph.D. dissertation, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, abril 2002.

[9] M. Chen, E. Foroughi, and F. H. at al., "Soccerserver manual," RoboCup Federation, Tech. Rep., 2002, http://sserver.sourceforge.net/docs/manual.ps.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley, 1998.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[12] J. R. Kok, N. Vlassis, and F. Groen, "Team description uva trilearn 2003." in *RoboCup 2003 Symposium*, March 2003.