DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Studiengang Informationstechnik

Studienarbeit II

6. Semester: 24. Dezember 2010 - 17. Juni 2011

Design and Implementation of the Botmanager and the Skillsystem of the Robocup Team Tigers Mannheim

von

Clemens Teichmann

- *Matrikelnr.: 293066* -

Lukas Gräser

- *Matrikelnr.: 280762* -

- *Kurs: TIT08A* -

Tigers Mannheim
DHBW Mannheim

Betreuer:
Dr. Jochem Poller

Author's Declaration

Unless otherwise indicated in the text or references, or acknowledged above, this thesis is entirely the product of my own scholarly work. Any inaccuracies of fact or faults in reasoning are my own and accordingly I take full responsibility.

Mannheim, July 25, 2011

Abstract

This seminar paper is a comprehensive documentation of the creation of a robot controller with serveral levels of abstraction for the centralsoftware Sumatra of the RoboCup-team Tigers Mannheim. The bot controller should be implemented in the modul system of Sumatra and use a qualitified partitionment of the program logic. Furthermore the controller should be an interface between the bots and the artificial intelligence modules. To do so, it is separated into Botmanager and Skillsystem, which have a connection among themselves and to the bots respectively the skillsystem. The result are two high-performance, clearly programmed modules with a succesful split of theirs tasks, which is a good beginning of the development. These modules need further implementations as Sumatras complexity increases and new function and options will be added to Sumatra or the bots.

Contents

List of Figures

list of abbreviations

AI	**a**rtificial **i**ntelligence
DHBW	**D**uale **H**ochschule **B**aden-**W**ürttemberg
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
XML	E**x**tensible **M**arkup **L**anguage

1 Introduction

The team Tigers Mannheim from the **D**uale **H**ochschule **B**aden-**W**ürttemberg (DHBW) Mannheim made it its task to develop a team of soccer playing robots with the aim to participate at the world championship of robot soccer, called RoboCup. The team of the Tigers Mannheim consists of about 40 students coming from different fields of study such as information technology, marketing, electronics and mechanics. These interdisciplinary competences are very important and necessary in order to lead this project to success, because the team has to face tasks such as:

- creating a corporate identity
- designing and constructing soccer playing robots
- developing a sophisticated software that controls the robots and enables them to play

The software that has been created is called Sumatra. The development of this software is still in progress. Just a few modules already reached the betastadium. That means that a lot of the sourcecode will be changed and extended. With regard to sustainability, it is important to keep the documentation of the code up to date, so that the next generation of team members are able to understand the software and to extend it. That is the aim of this thesis.
This thesis is concerned with just a certain part of the whole software: Skillsystem and Botmanager. Before you can understand what the Skillsystem and the Botmanager is and which functionality it provides you need to have an idea of the structure of the whole software.

1.1 Central Software Sumatra

The Sumatra software is the central program of the Tigers Mannheim. It is a object oriented programmed software completely written in Java. As it has to fulfill different tasks it consists of serveral modules. This structure simplifies the development by many programmers at the same time. Furthermore that guarantees expandability and maintainability. The tasks the Sumatra software has to deal with are inter alia:

- providing interfaces for incoming data
- processing incoming data

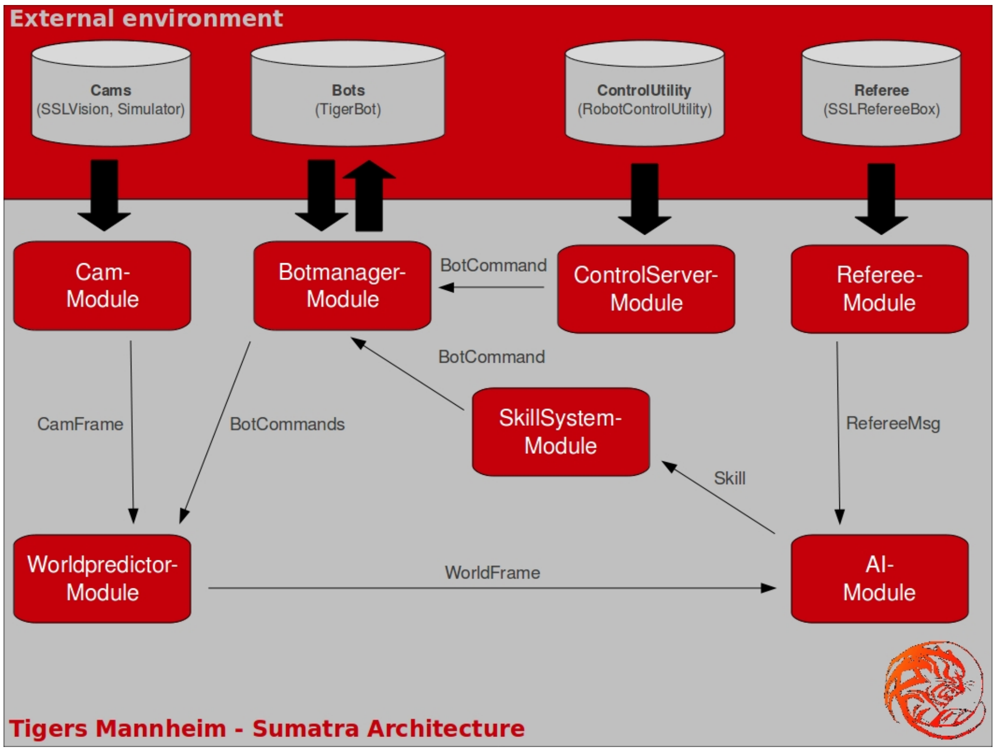


Figure 1: architecture of the Sumatra software

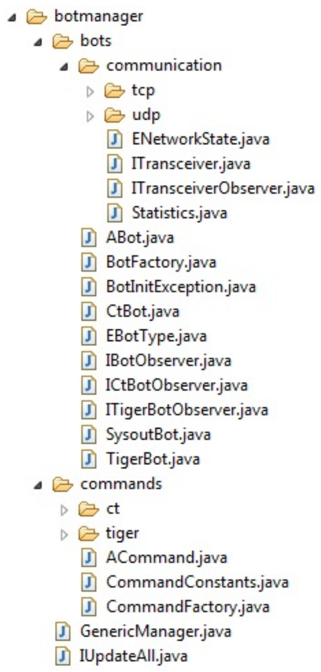
- compensating latencies
- making decisions on the basis of the processed data (**ar**tificial **i**ntelligence (AI))
- providing a simulator for testing
- controlling the robots

Figure 1 shows the rough architecture of the Sumatra software. Apparently the Skillsystem and the Botmanager are the connection between the AI-Module and the robots that have to be controlled. The Botmanager directly sends low level commands to the robot, such as: drive, rotate and shoot. In contrary, the AI-Module defines whole game moves containing interaction between several robots. Therefore an abstraction layer between Botmanager and the AI-Module is needed. That is the task of the Skillsystem.

2 Botmanager

The Botmanager is one module in the software Sumatra, the centralsoftware of the project ”Tigers Mannheim”. It controls the lowest layer between Sumatra and our robots. This includes bots and their commands as well as everything needed to communicate with the robots. Furthermore it provides information from the robots, e.g. power- or connectionstatus. The Botmanager is invoked from the skillsystem, which represents the next higher layer in the centralsoftware. There are implementation both for the tiger and the older ct bots, but only the part for the tiger bot is still in development. In the following sections will be a detailed look on how the botmanager is designed, how it operates and how it communicates with the tiger bots.

2.1 Design of the Botmanager



```
botmanager
  bots
    communication
      tcp
      udp
      ENetworkState.java
      ITransceiver.java
      ITransceiverObserver.java
      Statistics.java
    ABot.java
    BotFactory.java
    BotInitException.java
    CtBot.java
    EBotType.java
    IBotObserver.java
    ICtBotObserver.java
    ITigerBotObserver.java
    SysoutBot.java
    TigerBot.java
  commands
    ct
    tiger
    ACommand.java
    CommandConstants.java
    CommandFactory.java
  GenericManager.java
  IUpdateAll.java
```

Figure 2: package structure

In figure 2 you can see the design of the botmanager with its classes. The generic manager is like the main class, because from up there starts every command and the whole communication to the bots. Although it is divided into the packages bots and command. In the first package are all classes to set up the communication with the bots. This folder holds serveral abstract classes and interfaces to set up the bots, e.g. ABot.java and its subclasses TigerBot.java, CTBot.java and SysoutBot.java as well as transceivers for connections using **U**ser **D**atagram **P**rotocol (UDP) or **T**ransmission **C**ontrol **P**rotocol (TCP) (see section 2.3). The command package handles all commands from the skillsystem or the tiger/ct bots. Therefore we use an abstract class ACommand.java and a CommandFactory.java (see the sections 2.4 and 2.5). In the folders ct and tiger are all necessary commands for both bot types stored.

2.2 GenericManager

The class GenericManager handles the control of every single bot. All commands from the skillsystem go to its execute-method where the last steps before sending them to the robots are made. In addition it provides methods to add/remove and start/stop robots plus read a configuration file in E**x**tensible **M**arkup **L**anguage (XML) format. The GenericManager includes an innerclass UpdateAllTread, which extends the java interface Runnable. This thread is used to send all relevant command data (move, dribble, kick) via multicast to all robots. The process from a skill to a bytestream is illustrated in fig. 3 and will be explained in the following.

The Skillsystem sends a bot id and the corresponding command to the executeCommand-method in GenericManager.java. This method chooses the (tiger) bot by iterating over a hashmap and call its execute-method. Now the robot specifies the command whether it is a move-, dribble- or kickcommand with a switch-case statement. Each command needs other preconditions to be computed. At this point, the robot has two options available how to send the data. The first option is to use the updateAll function, so all data is send via UDP multicast. With the second option, the data is transfered to a single robot via UDP unicast. More to this options in section 2.3.1. The command is now in the sending process, which is also described in section 2.3.

2.3 Connection

The connection between the centralsoftware and our bots is organized with the protocol UDP. This protocol has many advantages compared to TCP for our system. First of

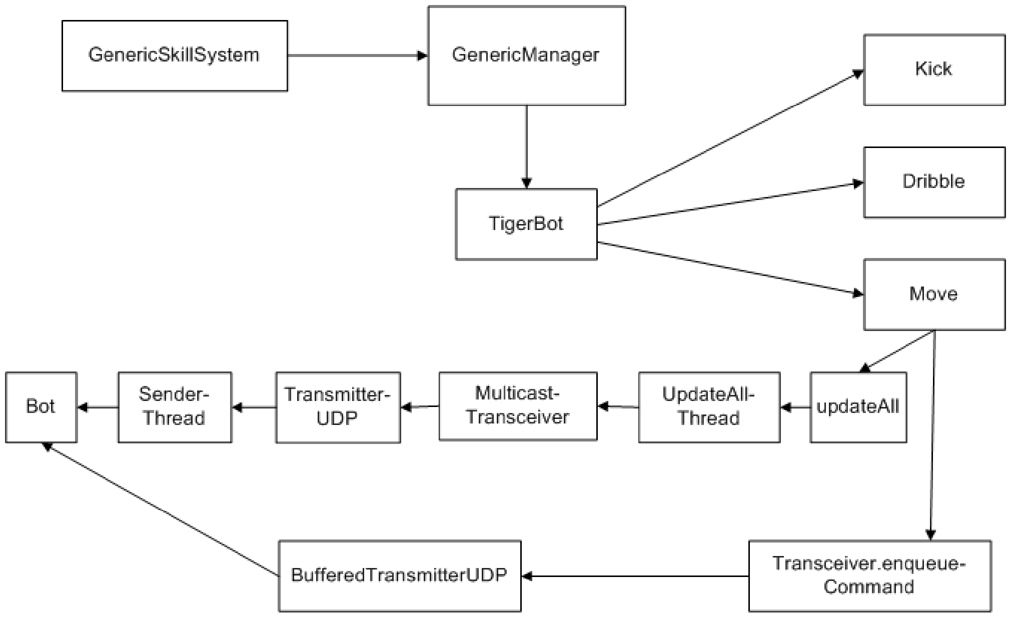


Figure 3: the flow of a command

all it's faster than TCP, because it works connectionless. This means that there is no transmission channel or handshake between two parties, it also relinquishs error correction. Another benefit is, that it uses much less bandwith than TCP, which is very important as the bandwith is limited. UDP supports two kind of delivery methods, unicast and multicast (see fig. 4(b)). To send data to exact one user, you use unicast. Multicast is used to send a message to many users in the network (but not to all → no broadcast). Those methods are useful to either send one command to one bot (like status update) or several commands grouped to many bots (like move and dribble).

As written in section 2.2, the centralsoftware use both systems. For unicast, the command is directly send to the UnicastTransceiver.java, which enqueue the command to the queue in the BufferedTransmitterUDP.java. The class BufferedTransmitterUDP has a so called SenderThread, which checks preconditions and transmits the command as a byte stream to the bot. One important precondition is the delay between the same commands. Every transmitted command will be saved into a buffer list with the command identifier and the sending time. If the bot is trying to send the same command again, before a minimum delay (1ms) is past, the command will be saved into another list. The SendThread will check this second list and if it waited long enough, the command

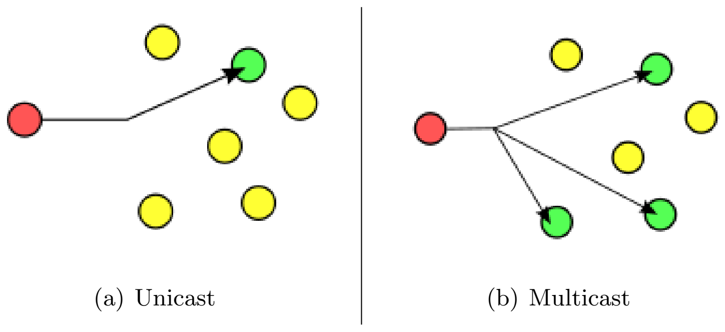

(a) Unicast (b) Multicast

Figure 4: delivrey methods unicast and multicast

will be enqueue again and send to the bot. But if the bot send the command a third time, the waiting command will be deleted.

A command send via multicast first goes to the MulticastDelegate.java. This class groups the command in three lists, each for a command type (move, kick and dribble). The Thread UpdateAllThread handles those grouped command. But first it checks, whether a bot is active or not. A bot is not active of it is turned off or currently out of the field. If such a deactivated bot is found, two commands are send to it, to stop the motor and dribble cylinder. Now with the list of all active bots on the field, the UpdateAllThread sets the commands to TigerMulticastUpdateAllV2.java. This class extends ACommand, therefore it can be handled as any other command, but it includes up to three commands. Then the thread sends the command to the MulticastTransceiver.java, which enqueue the command in the TransmitterUDP.java. There is no need to use the BufferedTransmitterUDP.java because the delay is implemented in the UpdateAllThread. The transmitter converts the command to a byte stream and send it over a socket to the bot.

2.3.1 Unicast vs. Multicast

As said earlier, a command can be sent via unicast or multicast. This differentiation is important for the amount of data, that is send to the bots. Every UDP packet has a header and the payload. The header stores information about the source/destination of the packet and the payload length. The payload holds the actual data and is variable in its length. If you send a command via unicast, you have only one command as payload. If you use multicast, Sumatra will pack the commands the one and send it out, so the payload includes many commands. This way the packet is only sent once. So multicast and the grouped commands reduce the used bandwidth to a minimum compared to multiple unicast transfers. Regardless, unicast is still used in Sumatra, e.g. to set motor parameters or request status updates. Most of the time, you just want information about one single bot, then multicast would become very difficult and produces much overhead.

2.3.2 State pattern

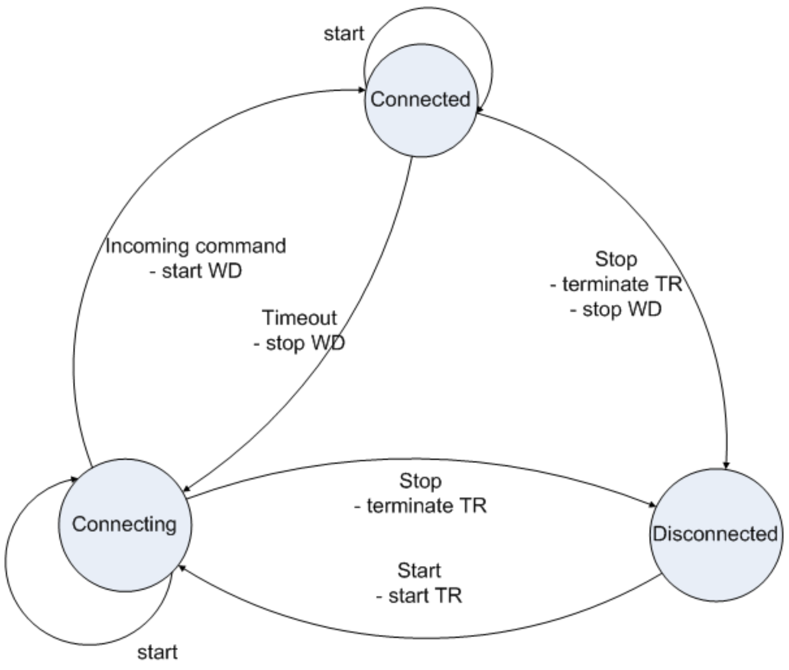


Figure 5: state machine

The "State Pattern" is a behavioral pattern among the design patterns. It defines so called objects for state. This means in practice, that every state has a its own class, which should extends an abstract superclass. External classes have acces through this abstract class to the several states, which are seperated from each other. The state pattern is often used to emulate finite state machines in programming.

In Sumatra, we have three states: disconnected, connecting and connected (see fig. 5). These can switch from all to all states, except from disconnected to connected. The whole setup of the connection between the centralsoftware and the bots is managed by this state pattern. In the following, the changes between the states shall be described. On default, a tigerbot is disconnected. If you turn it on, it will try to connect with sumatra, so the states changes to connecting. To do so, the bot transceiver gets all necessary information and starts. The terms transceiver and watchdog are declared at section 2.3.3. If the connecting failed, the state changes back to disconnected and terminate the transceiver. If the but succesfully connected to sumatra, the bot shift to connected. Thus will set saved parameters for the bot and starts the watchdog. Out of this state, the bot can switch to connecting, if for some reason the connection gets lost, or to disconnected, if the bot is turned off or just disconnected manually from the centralsoftware. If Sumatra lost the connection to one bot, the watchdog will be stopped and the software will try to connect the bot again. To disconnected a bot, the transceiver will be terminated and the watchdog stops.

2.3.3 Transmitter, Receiver and Watchdog

The centralsoftware Sumatra transmits and receives data using a transceiver (combination of transmitter and receiver) and controls the connection with the watchdog mentioned earlier. Every delivery method (see section 2.3) has its own transceiver. The transceiver handles the port to connect to a bot, opens or closes the link between bot and sumatra and hands over commands from bot to Sumatra and vice versa. The transmitter and receiver are both managed by the transceiver, like e.g. start and stop of sending/receiving thread. Thereby, the transceiver is like an interface to these to classes and external objects can only access them with the transceiver. The Watchdog observe the connection between Sumatra and the bots and triggers a system reset or other corrective actions. If a bot is turned off, there must be some method to recognize this action. Therefore is the watchdog implemented. It starts as soon as the connection is set to "connected" and runs in the background. The so called WatchdogRun thread checks, whether the bot sent a command or not. This thread sleeps a defined among of time and needs to be reseted after that time. If the bot sends a command to the centralsoftware, the transceiver automatically resets the watchdog . If the bot cannot transmit any more data, the watchdog recognize that as the period runs to zero without reseting. At this moment, Sumatra changes the state of the bot to "connecting" and tries so reconnect with the bot, meanwhile the watchdog is stopped. The bot remains "connecting" as long as the bot returns (e.g. turned on again) or the user disconnects the bot manually. This is very useful because the bot can just be turned on and is automatically connected to Sumatra.

2.4 ACommand

Every single command inherit from the abstract class ACommand. A command consists of a payload (the actual data) and a header (see next paragraph). ACommand includes global methods to convert byte arrays to different data types like (unsigned) int, short or float and the abstract methods setData, getData, getCommand and getDataLength. Those abstract methods are implemented by the specific command. The Method getData returns the byte data as a byte array, setData sets the bytestream for the command, getCommand returns the command identifier from the class CommandConstants and getDataLength return the length of the data in bytes.

Every command also has a header, a four byte long byte array. The first byte defines the command, the second the corresponding section and the last two bytes save the data length. To unterstand how a command is defined, you have to take a look at the class CommandConstants. This class contains all commands and assign them two bytes, one for the section and one for the command. The section byte groups the commands into e.g. movement, kicker, motor or multicast commands. With this information, the commandheader provides the necessary data to send and receive the command data.

2.5 CommandFactory

To build a command, that was send by a bot, the design pattern "factory method" is used. It's a creational pattern to generate objects without specifying the exact class. To explain the factory pattern, a short example will be given in the next paragraph.

Think of a simple pizzeria program, that implements the pizzeria and the pizzas. All pizzas will be created in the class Pizzeria and every pizza consists of the base and different toppings. Usually the class Pizzeria would call the constructor of the class Pizza with input parameters to create a pizza. But if you use a factory pattern, there will be another class (or at least a method) to do so. The Pizzafactory will be called to create pizzas with different toppings and a base. Therefore the factory is an abstraction of the constructor. The desicion, which exact class should be instantiated, is left to the subclasses.

The commandfactory in the botmanager is used to generate commands send from a bot, e.g. status updates or information about the kicker (loaded or not). If the receive thread pick up a command, the thread calls the commandfactory with the commandheader to create a new command. This new command can be hand-over to other methods to evaluate its information. By the information given in the commandheader, the factory knows which command to build.

3 Skillsystem

The Skillsystem is a module in the Sumatra software. It is the intermediary between the AI-module and the Botmanager. Its task is to transform complex highlevel skills given by the AI-module into simple lowlevel botcommands, that can be handled by the Botmanager. It also manages the assignment of the skills to the concerning robots, because every robot has its own Botmanager. Furthermore it assists in validating the performed skills.

3.1 Design of the Skillsystem

The design shown in figure 6 is the current design of the Skillsystem. The following sections describe this design in detail. In order to illustrate the improvements achieved during the progressing development a comparison between an older version of the skillsystem and this one is made in chapter 3.3. As one can see in figure 6 the three most important classes of the Skillsystem-module are:

ASkill abstract class that provides generic access to all implemented skills

SkillExecutor class with an own thread that transforms the current skills into botcommands

GenericSkillsystem intermediary class between Ares (class of AI-module) and Botmanager

3.1.1 Skills

A skill is a complex move of a robot that consists of couple of actions a robot is able to do at the same time. There are three distinct groups of skills:

move skill describing a certain movement, combination of translation and rotation

dribble skill concerning the dribbling device

kick skill for passes, straight shoots and chipkicks

This subdivision complies with the three forms of actions a robot is able to do at the same time.
All skills are implemented with regard to the reqirements of the AI-module. Most of the currently existing skills are move-skills:

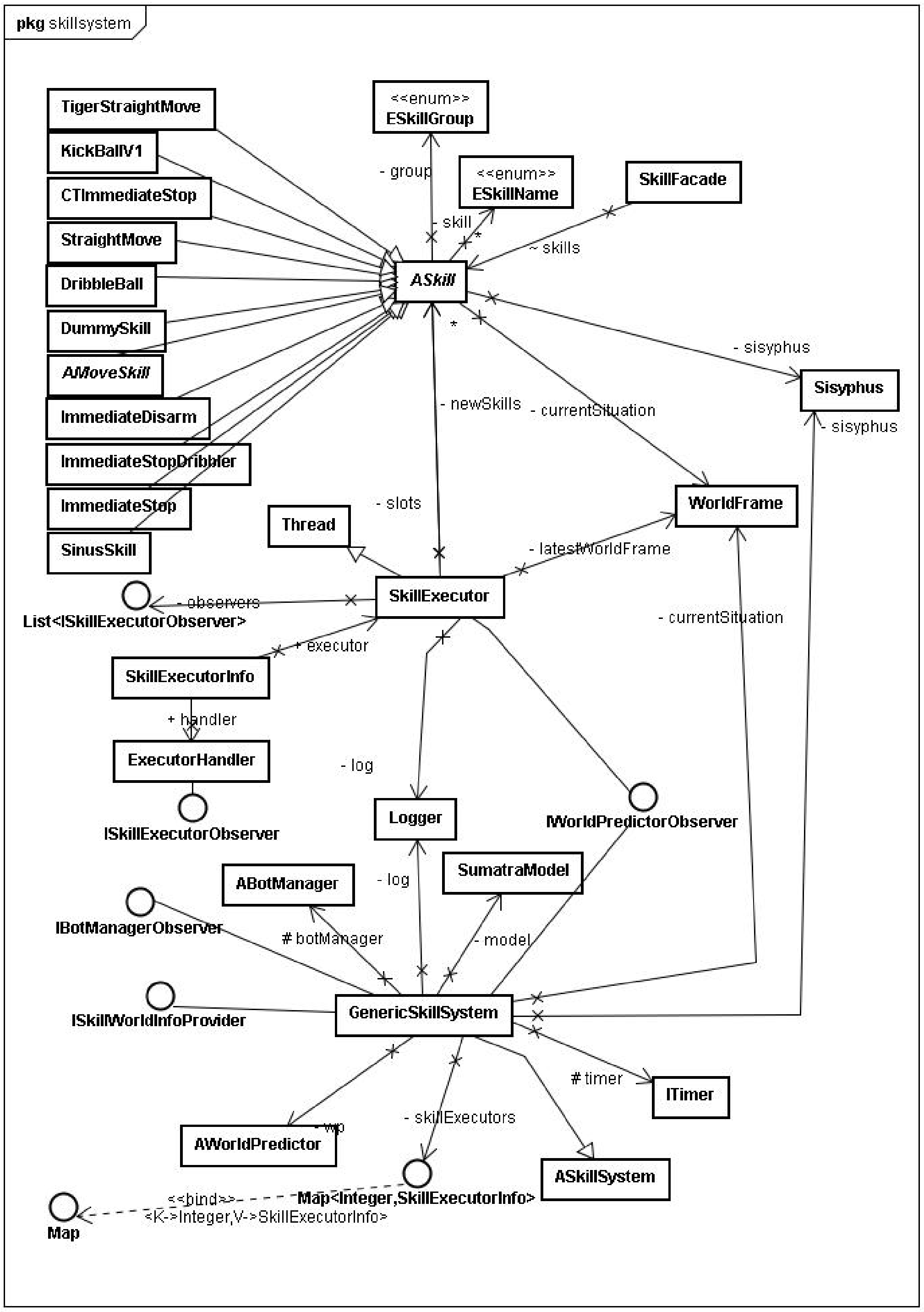


Figure 6: class diagram of the Skillsystem

- AimInCircle
- MoveAhead
- MoveDynamicTarget
- MoveFixedCurrentOrientation
- MoveFixedGlobalOrientation
- MoveFixedTarget
- SinusSkill
- TigerStraightMove

The following skills control the dribbling and kicking devices of the robots:

- DribbleBall
- ImmediateStop
- ImmediateDisarm
- ImmediateStopDribbler
- KickBalV1

All these skills inherit from the abstract class ASkill. As already mentioned this simplifies the generic handling of skills, because not all of the objects that are processing the skills need to know what skill it actually is.
The move skills do not directly inherit from ASkill. There is another abstract class, called AMoveSkill, that provides a generic access to all move skills. The figure 7 shows this concept of inheritance by the example of the skills ImmediateStop and MoveAhead. As one can see at the ASkill class every skill has to implement two functions on its own:

calcActions This function calculates and returns all the necessary commands that have to be executed in order to do this skill.

compareContent This function compares the values of the skills' attributes and returns the boolean value TRUE if they are equal, otherwise FALSE.

All the other functions are already implemeted by the ASkill class. The abstract subclass AMoveSkill implements the function calcMovement for all its subclasses and it declares two additional functions that have to be implemented by every move-skill. So every move-skill needs to have the three functions calcTargetOrientation, getTarget and compareContent implemented.

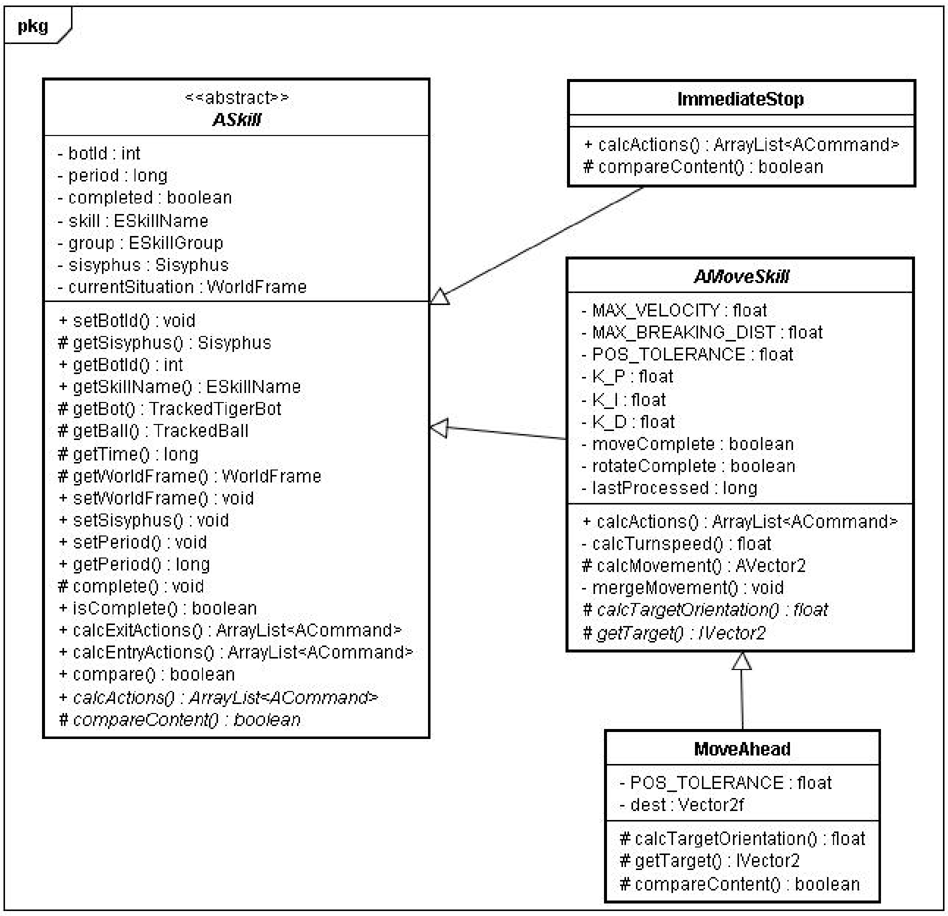

pkg
<<abstract>>
ASkill
- botId : int
- period : long
- completed : boolean
- skill : ESkillName
- group : ESkillGroup
- sisyphus : Sisyphus
- currentSituation : WorldFrame
+ setBotId() : void
# getSisyphus() : Sisyphus
+ getBotId() : int
+ getSkillName() : ESkillName
# getBot() : TrackedTigerBot
# getBall() : TrackedBall
# getTime() : long
# getWorldFrame() : WorldFrame
+ setWorldFrame() : void
+ setSisyphus() : void
+ setPeriod() : void
+ getPeriod() : long
# complete() : void
+ isComplete() : boolean
+ calcExitActions() : ArrayList<ACommand>
+ calcEntryActions() : ArrayList<ACommand>
+ compare() : boolean
+ calcActions() : ArrayList<ACommand>
# compareContent() : boolean
ImmediateStop
+ calcActions() : ArrayList<ACommand>
# compareContent() : boolean
AMoveSkill
- MAX_VELOCITY : float
- MAX_BREAKING_DIST : float
- POS_TOLERANCE : float
- K_P : float
- K_I : float
- K_D : float
- moveComplete : boolean
- rotateComplete : boolean
- lastProcessed : long
+ calcActions() : ArrayList<ACommand>
- calcTurnspeed() : float
# calcMovement() : AVector2
- mergeMovement() : void
# calcTargetOrientation() : float
# getTarget() : IVector2
MoveAhead
- POS_TOLERANCE : float
- dest : Vector2f
# calcTargetOrientation() : float
# getTarget() : IVector2
# compareContent() : boolean

Figure 7: excerpt of the class diagram as example for the inheritance among the skills

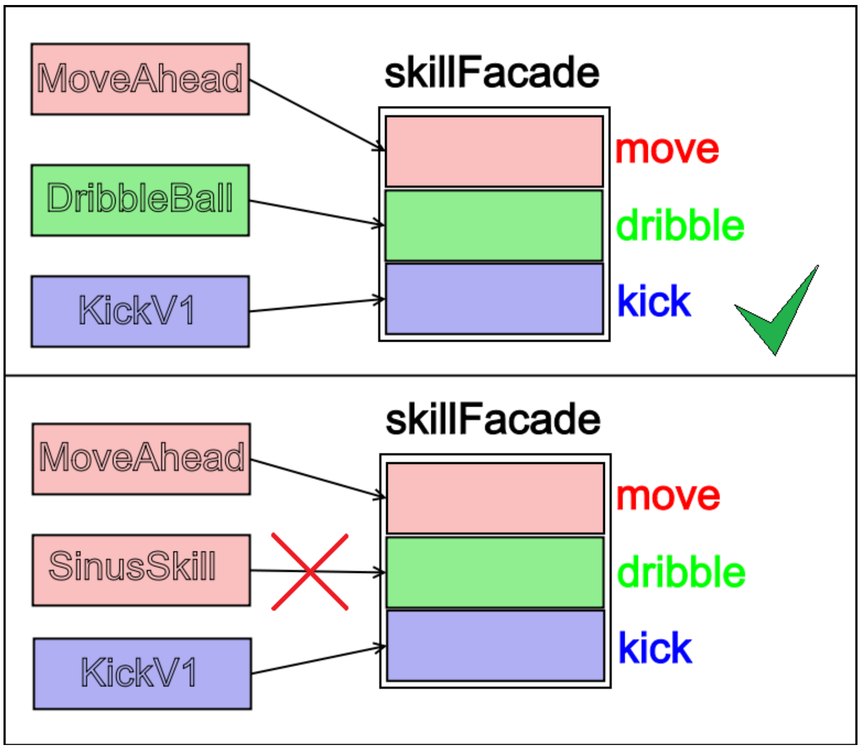


Figure 8: SkillFacade with three skill slots, one for each type

3.1.2 SkillFacade

Facade describes a design pattern that is often used to provide a simple interface for a complex subsystem. In this case the subsystem is all the different skills and the class SkillFacade is the interface. All objects that have to handle skills may use this interface. Its main task is to ensure the structured use of skills.
As mentioned there are three types of skills: move, dribble and kick. All skill triples containing one skill of each type are meant to be executed simultaneously. But it would not make sense to execute two different move skills, e.g. MoveAhead and SinusSkill, at the same time. Therefore the SkillFacade consists of three slots that may be filled with one skill of each type. As shown in figure 8 the SkillFacade guarantees that no skills of the same type can be combined.

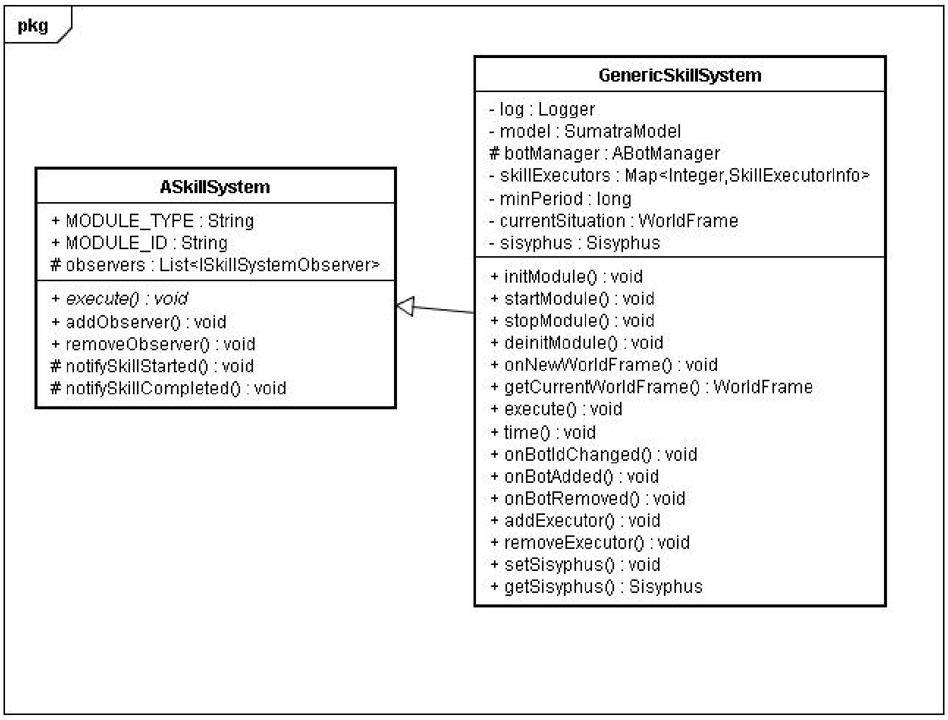


Figure 9: diagram of the GenericSkillsystem class

3.1.3 Generic Skillsystem

The Generic Skillsystem is the intermediary module between the AI-module and the Botmanager. It gets defined SkillFacades from the AI-module and passes them to the SkillExecutor. The SkillExecutor transforms the SkillFacade to several lowlevel commands and returns them to the Generic Skillsystem. Then the Skillsystem passes the commands right to the Botmanager. As one can see in figure 7 every Skill has the member botID (and so does SkillFacade). In order to ensure that a skill/skillFacade will be executed by the right robot, every robot has its own Botmanager and every Botmanager has its SkillExecutor. It is the task of the Generic Skillsystem to coordinate the SkillFacades properly. Therefore it has a couple of functions that react to events as changing, adding or removing a bot (fig. 9).
The only way the Generic Skillsystem may recognize when a bot has been changed, added or deleted is to "listen" to the Botmanager. This is realized by a design pattern called

observer. One object is the publisher, in this case the Botmanager, and several other objects are subscribers (here: inter alia the Generic Skillsystem) that want to be informed when something on the publisher's side has changed. The publisher maintains a list of all subscribers and every subscriber has to implement one or more special eventhandler functions that has been defined by an interface. In case of the Skillsystem these functions are: onBotIdChanged, onBotAdded, onBotRemoved (fig. 9). The Generic Skillsystem is not only subscriber but also publisher. It inherits the observer list from the abstract class ASkillsystem as well as the two functions notifySkillStarted and notifySkillCompleted to inform all observers when a skill has been started or stopped.

3.1.4 SkillExecutor

The SkillExecutor class receives skills or skillfacades and transforms them into lowlevel commands the Botmanager can deal with. According to the skill types it has three so called 'skill slots' that are filled with the currently executed skills. When one of these skills has been completed the slot is empty. When the Executor receives a new skill it will be assigned to its slot even if the slot still contains an "old" skill. This guarantees that no skill can get lost. If the new skill equals the currently executed skill, then the new one will be ignored and the execution of the old one continues.
For reasons of performance the transformation from skill to commands runs in an own thread. Therefore the SkillExecutor inherits the Thread class. The implemented run-function contains a loop that is running as long the thread is active. This is shown in figure 10. This flowchart represents the simplified workflow of the SkillExecutor thread. It shows the case. Inside the loop it iterates over the three slots (for simplicity neglected in figure 10), transforms them and returns the lowlevel commands to the GenericSkillsystem.

3.2 Lifecycle of a skill

This section shall illustrate the whole workflow of the Skillsystem-module starting from the instantiation of a skill up to the execution of its commands by the botmanager. In figure 11 this workflow is presented as a diagram.
The AI-module assigns different roles to the robots. Each of this role includes one SkillFacade that defines which actions a robot with this role has to execute. A certain class of the AI (Ares) handles these roles and invokes their function calculateSkills. This

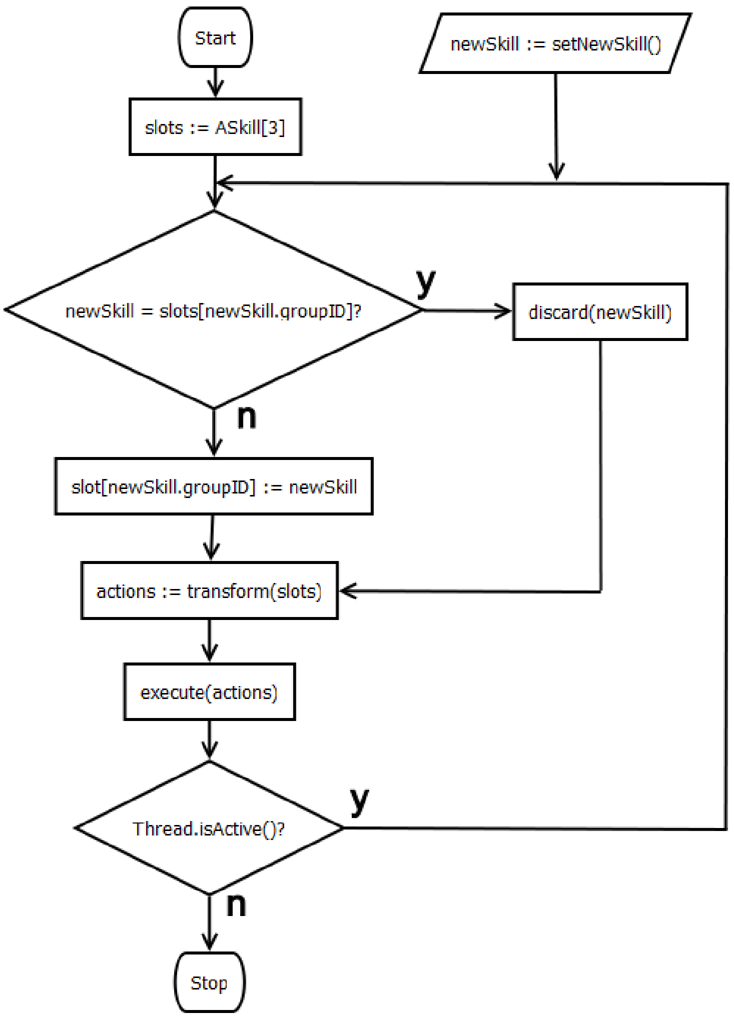

Start
newSkill := setNewSkill()
slots := ASkill[3]
newSkill = slots[newSkill.groupID]?
y
discard(newSkill)
n
slot[newSkill.groupID] := newSkill
actions := transform(slots)
execute(actions)
Thread.isActive()?
y
n
Stop

Figure 10: flowchart of the SkillExecutor thread

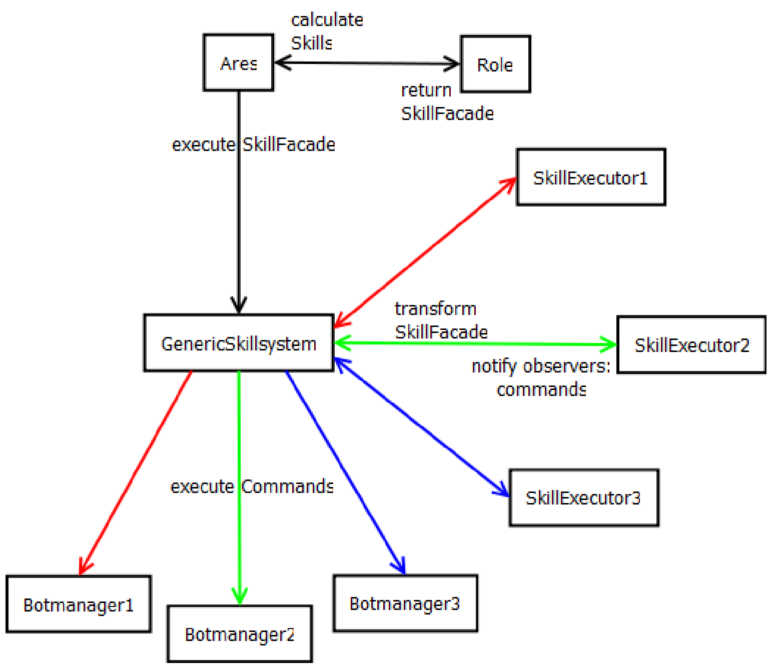


Figure 11: diagram of the skill-workflow

function instantiates the role's SkillFacade.
The Skillsystem is a member of the Ares class. This way Ares is able to pass the SkillFacade of the concerning role to the Skillsystem. The GenericSkillsystem passes the SkillFacade according to its robotID to the right SkillExecutor. The Skillexecutor divides the facade into the three skills and calculates their lowlevel commands. Then it sends all the commands to its observers, that means to the GenericSkillsystem. The GenericSkillsystem then passes the commands to the according Botmanager.

3.3 Comparison between old and new version

This chapter illustrates the development of the skillsystem. One of the biggest difference between the new and the old skillsystem is the structured handling of skills. The AI-module instantiated the skills according to the current role and passed them directly to the skillsystem. There was no distinction between the three different groups of skills. In order to combine skills there had been an ACombinedSkill-class. Objects of this class were handled as one skill, but these objects contained several skills. It was also possible

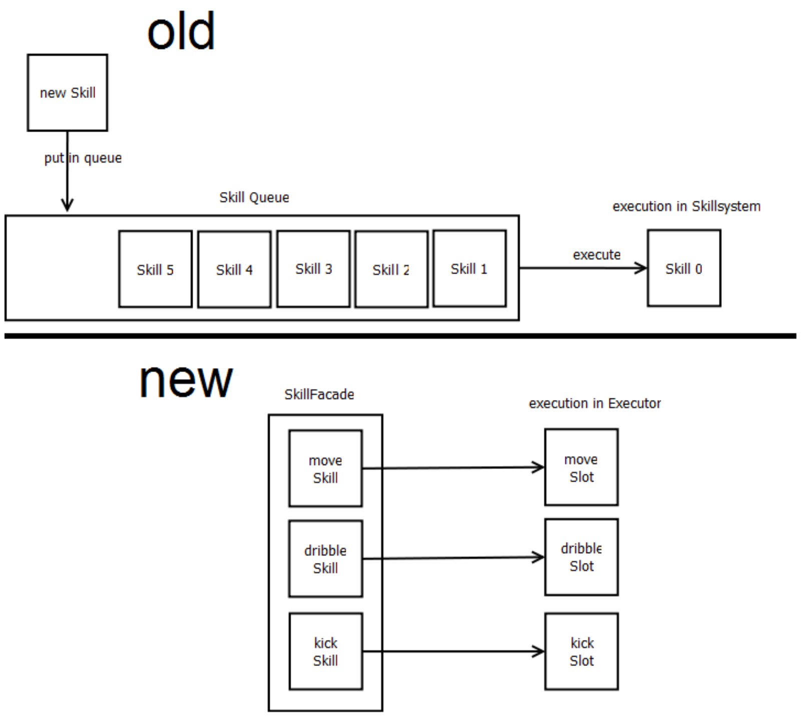


Figure 12: top: skillhandling of the old system; bottom: skillhandling of the new system

to put such a combines skills into a combined skill. There was no mechanism to control wether the skills that have been combined are executable in this way. So one could create skills that combine contrary move skills.
Another point is that the new GenericSkillsystem does not have a skill queue anymore. The old skillsystem put every incoming skill into a skillqueue where the skill waited until all the other skills in the queue were finished. The skillsystem also provided the option to flush the queue so that a new skill could be executed immediatly. The experience showed that this option was very useful and it was used every time. So the skill queue became rather senseless and instead the skillsystem now has only three slots that contain the currently executed skills (fig. 12).
Furthermore the skills have become more sophisticated. In the old system many skills were created for single specific action. Now the number of skills has decreased compared to the number of different actions the robots can execute. This was achieved by overloading some skills. So one skill covers different actions according to its different constructors.

Another big difference between the old and the new skillsystem is the use of skill executors. The old GenericSkillsystem is managing the execution of the skills by its own. The skill itself knows wich lowlevel commands it contains the skillsystem passes them one by one to the botmanager. The structure of the new skillsystem with its skill executors complies with the different botmanagers. This model is easier to understand and simplifies future adaptions of the skillsystem.

4 Conclusion

With the Botmanager and the Skillsystem two powerful modules have been created to communicate with the bots and to control them. Each module forms a layer to higher or lower program logic and can easily be altered through the module design of Sumatra. The interfaces are clear and allow the developer to modify e.g. the Botmanager without changing the Skillsystem.

Both layers are improved for the current system and need further development in the future as new features enlarge the command complexity. In the near future the bots will be upgraded with e.g. a chip kicker, which allows the bot to shot crosses (high pass). This needs to be implemented as a command in the Botmanager as well as a skill in the Skillsystem. Furthermore the Skillsystem needs to be enhanced as the ai-modules develop.