Software Design Document

Swinbots - Swinburne RoboCup Team
SEP Group 1, 2004

Author(s):	James Fraser - 1261371
	Shane Harvie - 1262246
	Michael Henderson - 1261487
	Luke Pitcher - 1262769
	David Reed - 1266659
	Andrew Walker - 1262106
	Nickolas Wanke - 1229338
	Dejan Zigic - 1265636
Last Modified:	October 4, 2004
Version:	*Revision* : 1.66

Modification History Table

Date	Version	Description	Author
22/04/04	1.1	Initial Creation	SH
22/04/04	1.2	Add in some of the Tactics Engine description of classes	LP
22/04/04	1.3	additions to the Tactics Engine of classes, methods, and data flow	LP
22/04/04	1.4	Removed errors in build and fixed references	LP
22/04/04	1.5	Started some of the image processing documentation	AW
22/04/04	1.6	Updated space requirements according to SQAP	DZ
22/04/04	1.7	Included embedded code descriptions and diagrams	NW
22/04/04	1.8	Added modification history for recording, updated mod history table	SH
07/07/04	1.9	added packet section to SDD and build process as appropriate	SH
09/07/04	1.10	more image processing	AW
10/07/04	1.11	added diagrams for all packet types for SDD	SH
13/07/04	1.12	added field-related class diagrams	SH
13/07/04	1.13	more image processing	AW
13/07/04	1.14	calibration tool	AW
13/07/04	1.15	added a section about the general layout for our MFC GUIs (based on tactics GUIs)	DR
13/07/04	1.16	changed layout of my part a little	DR
13/07/04	1.17	Tactics stuff on MapBuilder add to SDD	LP
13/07/04	1.18	Addition to tactics of LOSMap, and GoalMap classes	LP
13/07/04	1.19	Minor change - added Public to GoalMap Class description	LP
20/07/04	1.20	minor editing changes	SH
20/07/04	1.21	minor editing changes	SH
21/07/04	1.22	added outer comms diagrams	AW
21/07/04	1.23	added more embedded information	NW
21/07/04	1.24	Bibliography	AW
21/07/04	1.25	Changed order and section layout	DZ
21/07/04	1.26	shuffled some things around (tactics)	DR
21/07/04	1.27	changed mapbuilder image tags	AW
21/07/04	1.28	corrections to images	AW
22/07/04	1.29	added field to Tactics section and added verbatim tag to all tactics sections	DZ
22/07/04	1.30	Update History Table	DZ
22/07/04	1.31		
22/07/04	1.32	Revisions to update of tactics diagrams	DZ
22/07/04	1.33	Moved order of Tactics-field	DZ
22/07/04	1.34	Cleaned up beginverbatim endverbatim tags	DZ
22/07/04	1.35	Fixed up problems created by use of verbatim	DZ
22/07/04	1.36	Fixed Nick's picture	AW
22/07/04	1.37	Rewrote Tactics Debug Tool and Tactics Engine class descriptions	DZ
22/07/04	1.38	verb now consistent throughout sections, diagram titles now consistent, fixed many spelling errors and typos.	MH
22/07/04	1.39	Updated Modification History Table	MH
23/07/04	1.40	fixed for MH	AW
23/07/04	1.41	added tiny amount to tactics section (data structures)	DR
23/07/04	1.42	added verbatim tags to Tactics Debug Tool that Michael missed	DR

Modification History Table

Date	Version	Description	Author
23/07/04	1.43	added purpose, functionality and interface for Tactics Engine GUI	DR
23/07/04	1.44	added methods and description for GDI Field	DR
23/07/04	1.45	added descriptions for GDIMapBuilder and Tactics Structs	DR
23/07/04	1.46	added screen shots of UID for Tactics GUIs	DR
23/07/04	1.47	smart pointer utilities	AW
23/07/04	1.48	Updated modification history from for my revisions	DR
23/07/04	1.49	changes made to MapBuilder section. fig added, decription of methods changes	LP
23/07/04	1.50	Added GDIViewable and IGDIViewable method and class description	DZ
23/07/04	1.51	Added Abstract and a new page for mod table	MH
24/07/04	1.52	Started editing the SDD - mainly consistency changes, a little bit of content	SH
24/07/04	1.53	Editing changes	SH
24/07/04	1.54	Added Coach material and diagrams, minor editing changes	SH
24/07/04	1.55	Minor editing changes	SH
24/07/04	1.56	Minor editing changes	SH
24/07/04	1.57	Added to tactics section	LP
25/07/04	1.58	Minor editing changes to IP section.	SH
25/07/04	1.59	Updated method descriptions for some blue comm	NW
25/07/04	1.60	Minor fixes	NW
25/07/04	1.61	Editing changes (up to half way through tactics)	SH
25/07/04	1.62	Editing changes - now complete	SH
01/08/04	1.63	Fixed the easy things from yale's review	AW
20/09/04	1.64	Added second tier of tactics and IP modules as per review requirements	SH
22/09/04	1.65	Fixed remaining 'gaps' and performed final editing	JF
02/10/04	1.66	Added to abstract	SH

Abstract

The software design of the 2004 Swinburne RoboCup project is presented in this document. The document breaks down each of the software components into appropriate levels of abstraction for use by software developers.

The RoboCup project software is essentially a real-time control system, with associated configuration and calibration tools, for a team of six soccer-playing robots. The system's primary purpose is to analyse an overhead view of the robotic soccer game in real-time. Based on perceived game events, the system initiates intelligent and appropriate robot responses.

The Image Processing Server captures and processes real-time video data, acquired from the game environment, extracting useful information for analysis by the Tactics Engine. Communication between these two systems is via sockets.

An Image Calibration Tool has been designed to solve the problem of excessive manual calibration. The new tool will be able to calibrate the system using real-time data, in as little as 2 minutes.

To decide on team tactics and individual robot actions, a Tactics Engine has been designed, based on "potential map" theory. The Tactics Engine also provides a graphical representation of the software's 'knowledge' of the playing field. The Tactics Debug Tool has been designed to enable the developers to analyse the tactics decisions, which are made during a game, for debugging and tuning purposes.

Bluetooth provides two-way radio communications with the mobile robots. An RF Server has been designed to provide the interface between the Bluetooth subsystem and the Tactics Engine. Custom packet structures are also defined.

The system is based on the Microsoft Foundation Classes (MFC) to reduce the volume of code to be written; however, the embedded software is written in ANSI C. The Embedded Software covers all code running on the CPU core local to each robot, and has the primary functions of packet decoding and hardware control.

Audience

The target audience for this document is:

- The Client — Dr Ali Bab-Hadiashar
- Project Team Members
- Project Supervisors
- Past and Future Team Members

Contents

List of Figures

1 Introduction

1.1 Document Purpose

This document presents the software design of the Swinburne RoboCup project, 2004. It shows the breakdown of the system into its components, and describes the communication methods between each of these components, as per the "Architectural Design Document". It then breaks down each of the ADD components into appropriate levels of abstraction, such that the team members can fully understand the task at hand, and begin to develop the system. These will include class diagrams and/or data flow diagrams as appropriate.

1.2 Terminology Used

1.2.1 Definitions

Client: A piece of software which accesses a service

Dribbler/Roller: Device used on the robot, enabling it to acquire and control the ball, with a view to maintaining possession whilst in motion.

Friendly robot: A robot that is a part of the team. Not an opposition robot.

Ghost: A silhouette of a robot which can be dragged to a position on the GUI representing the desired position for the robot.

Image: A digital image, comprised of many pixels.

Kicker: Device used by the robot to effect a kick on the ball.

Overlay: A layer of data in two dimensional vector form used to create and build the potential field view of the game environment.

Potential Field: A map representation of the game environment whereby the field is divided into a grid, with each grid element having an associated 'potential', depending on the positioning of opposition and friendly robots, and the ball. A map is drawn for a particular robot, and the potentials represent the relative attraction that the robot has to each grid element. A high potential represents a large attraction.

RoboCup: International competition where entrants develop and build teams of robots to contest soccer matches.

Server: A piece of software which provides a service

Stream: A data conduit.

Streaming: The transmission of ordered data across a conduit.

Tactics: Strategies to be employed during soccer matches, incorporated into the AI system.

Tactics Engine: Module of the system which is responsible for analysing the Image Processing Data, and deciding on team and individual robot strategies.

1.2.2 Acronyms and Abbreviations

ADD Architectural Design Document

AI Artificial Intelligence

IP Image Processing

MFC Microsoft Foundation Classes

MVC Model-View-Controller

PC Personal Computer

RF Radio Frequency

SDD Software Design Document

SRS Software Requirement Specification

2 System Overview

2.1 RoboCup Requirements and System Components

The RoboCup software system is to be developed in order to satisfy various requirements of the RoboCup environment. The core requirements and functionality are listed below, alongside their corresponding system components (in brackets):

- Acquisition and analysis of overhead images, in order to derive as much information as possible about the state-of-play (Image Processing Server).
- Decision-making, based on interpretation of the sensed state-of-play and a pre-defined set of tactical rules (Tactics Engine).
- Transformation of decisions into discrete actions for each friendly robot on the field (Tactics Engine).
- Communication of actions to respective robots (RF Server/Embedded robot code).
- Physical realisation of desired actions by the robot hardware (Embedded robot code).

Auxiliary to this core set of components are tools to be employed ‘offline’ (i.e. not during gameplay), which are to facilitate the configuration and on-going improvement of the system:

- Calibration of the overhead camera and related image-processing parameters, allowing effective and timely response to varying ambient lighting conditions (Calibration Tool).
- Storage of all game data, which may then be replayed after a game via a graphical tool, giving the team insight into the performance of the Tactics Engine(Image Processing Server/Tactics Debug Tool).

2.2 Architectural Design

2.2.1 Architectural Style

The architectural style chosen for this system is known as the **pipe-and-filter** system, a specific form of **Dataflow** system.

The pipe-and-filter style has the following characteristics: [1]

- Each component has a set of inputs and outputs
- A component reads a stream of data on its input(s) and produces a stream of data on its output(s)

[1]CALVERT D., Software Architectural Styles, http://hebb.cis.uoguelph.ca/ dave/27320/new/architec.html

- Input is transformed both locally and incrementally so that output begins before input is consumed (a parallel system)
- Components are called *filters*
- Connectors serve as conduits for the information streams and are termed *pipes*

The four major components that constitute the RoboCup system are connected to form a 'closed-loop' arrangement, with each component feeding information to at least one successive component via a pipe. This is common practice in real-time hardware control, whereby plant is controlled with the assistance of constant feedback of state information. The two offline tools are also able to 'plug in' to this loop via pipes.

Alternative Styles
The pipe-and-filter style was chosen over other styles due to the continuous nature of the data to be manipulated. The "Call-and-return" style is not suitable because of the low response latency required of the system. The "Virtual Machine" or "Rule-based system" style is also not suitable for the overall architecture (but can be used in the internal design of components, such as the Tactics Engine).

2.2.2 Software Architecture

The UML component diagram presented in Figure 1 shows the architecture of the RoboCup software system. Please note that data storage symbols have been included to emphasise cases of simple data storage, as opposed to it being 'transformed' as part of the pipe-and-filter system.

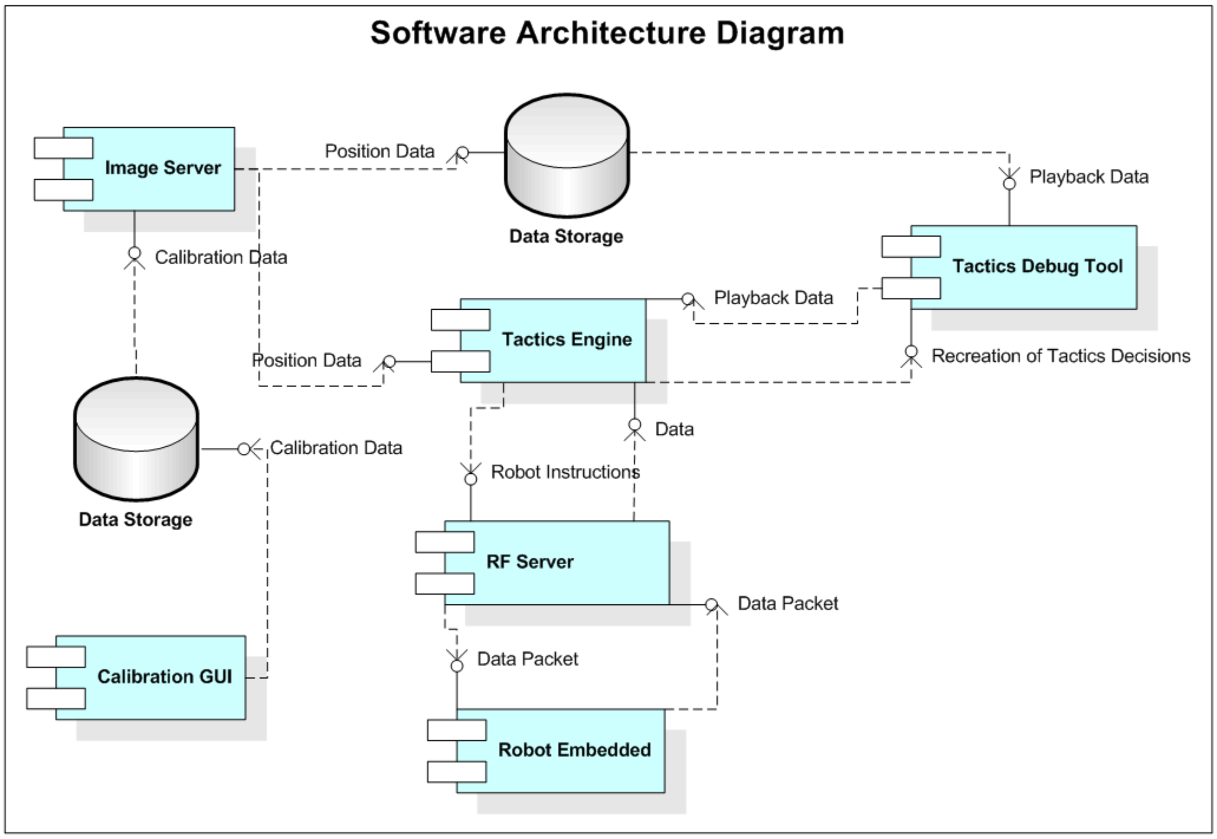


Figure 1: RoboCup System Architecture - UML Component Diagram

2.3 Environment Interface

Please refer to the Robocup 2004 SRS Appendix A

2.4 Human interface

Please refer to the Robocup 2004 SRS Section 9

3 Module Overview

This section provides conceptual diagrams for the two most complicated modules: Image Processing Server and Tactics Engine. Before delving into the detailed design for these modules, it is beneficial to describe the major concepts involved, the processing procedure, and the communication methods between these two modules.

3.1 Image Processing Server

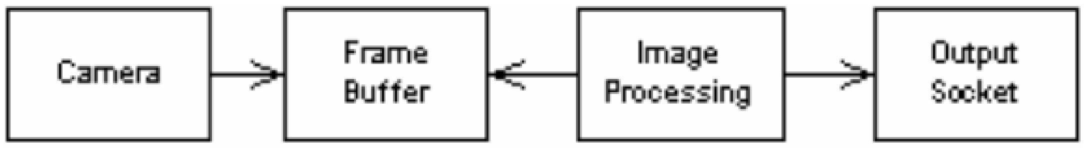


Figure 2: Module overview - Image Processing

Figure 2 shows the main sub-modules of the IP Server. The IP Server is responsible for starting the camera, redirecting the camera output to a circular buffer, and processing the image. It delegates these tasks to its two major components:

1. Camera
   - Provides the interface to DirectShow (for grabbing frames)
   - Provides configuration of DirectShow (hardware interface)
   - Creates and manages the video pipeline using COM Interfaces
   - Prevents resource leaks caused by DirectShow
   - Places image frames into a circular buffer
2. Image Processing
   - Reads images from the buffer
   - Processes images (finds the ball, finds the robots). Processing is delegated to a 3rd Party library: CMVision.
   - Writes IP packets (N.B. not Internet Protocol!), detailing ball and robot positions, to a socket

3.2 Tactics Engine

Figure 3 shows the major sub-modules of the Tactics Engine. It also shows the means of communication with the Image Processing Server. Arrows represent method calls. IP packets are 'pushed' into a socket on the IP Server end. The "Comms" sub-module is responsible for retrieving these packets and placing them into a circular buffer. Each time it retrieves one of these packets, a windows message is generated, to inform the IP packet handler (within the Tactics Engine) that a new packet has arrived, and is ready for processing. The IP packet handler uses the IP packet to update its view of the environment, stored within the "Field" sub-module. It sends each of the robots a "Current Position" packet informing them of their current position, via the RF Server. It then instructs Coach to update its tactics decision, based on the latest data. Coach uses MapBuilder to construct the appropriate potential map to determine the desired robot instruction. It then sends this instruction as a "Desired Position" packet via the RF Server.

]

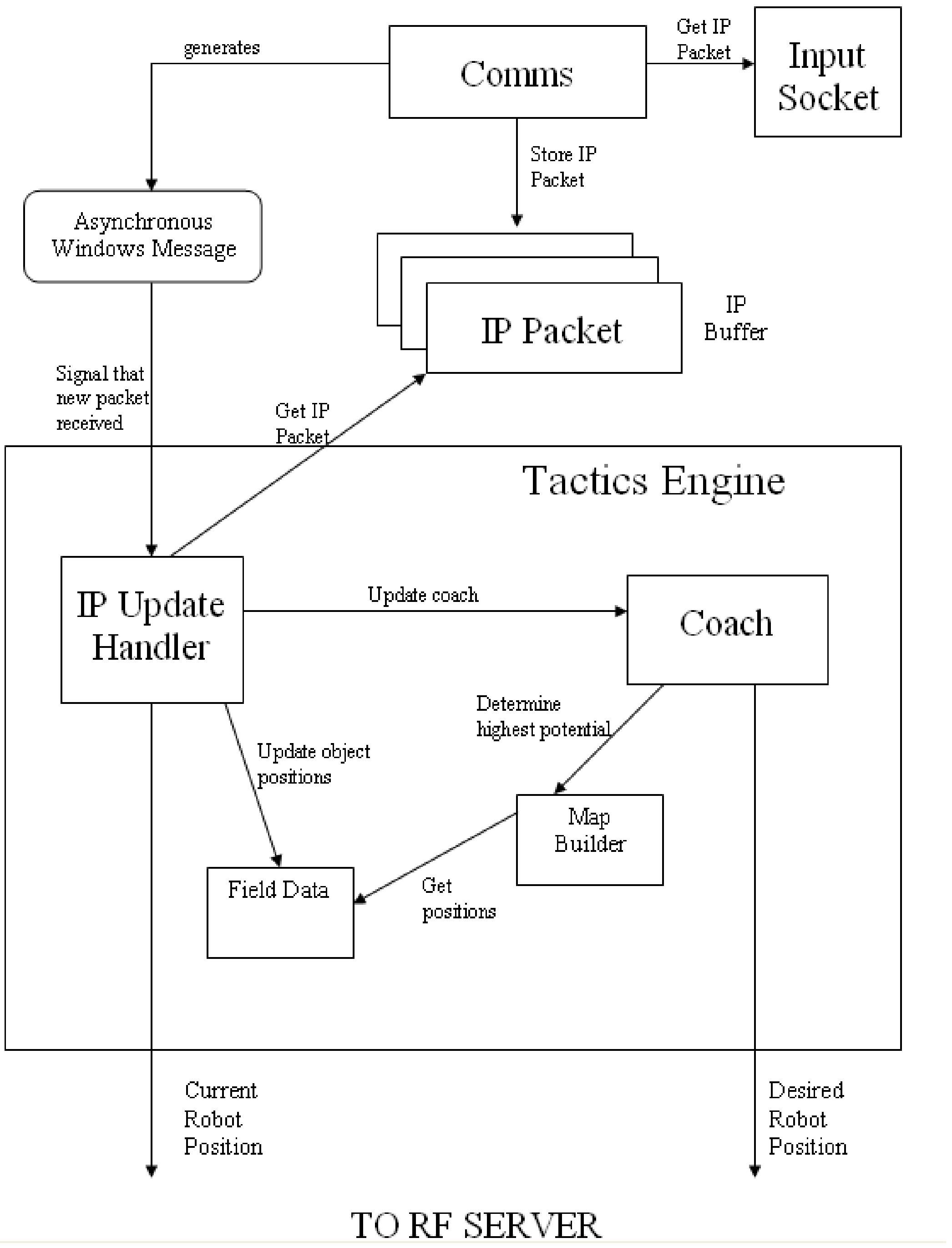


Figure 3: Module overview - Tactics Engine

4 Utility Modules

4.1 Graphical User Interfaces

The GUIs in our project will be made using the Microsoft Foundation Classes (MFC) Doc/View architecture, which is designed using the Model-View-Controller approach [2].

The Model-View-Controller (MVC) design pattern is a powerful way to separate the concerns of the user interface, the data model, and the state management of both. Events are used to achieve this decoupling. Both the model and view are well suited for declarative programming. The model manages the data, being a collection of fields with a known type, and the view is a representation of the user interface as expressed by a collection of controls. The controller is best constructed in code, as it contains the logic for responding to the events fired by the view and adjusting the model's state.

Advantages of using the MFC Application Framework:

- Much less code to write than a comparable Win32 application, which means that the application gets built faster and is more reliable. It also obviates the need for developers to write low-level and tedious code, much of which can be prone to errors.
- More reuse of code between applications, meaning that focus can go onto specialised code for the RoboCup applications.
- Standard interface and controls mean less frustration for users/customers

Disadvantages of using the MFC Application Framework

- Steep learning curve, especially compared to Java or C# Graphical User Interface development.
- The generalisations made by the library writers may produce a slower application.
- It can be harder to do exactly what you want, especially if you want a different look-and-feel, or if you want a new component.

4.1.1 Singleton and SharedData

SharedData provides a single place to store data such that each view can display this data in a different way. It is implemented as a single global variable which is a static object, in the template class `SingletonHolder`. `SingletonHolder` is based upon the singleton developed by Meyers in 'Effective C++', and described more fully by Alexandrescu in 'Modern C++ Design'.

4.1.2 IGDIViewable

This is an interface class used to avoid possible complications arising from the use of multiple inheritance in C++. It defines abstract methods that must be provided by subclasses if they are to be concrete.

The public member functions in the IGDIViewable class are:

```
virtual void  draw(CDC &dc) const = 0
```

- This method is used to represent the class' internal data in graphics. Must be overridden to provide customised drawing. Parameter `dc` is the device context to draw to.

[2]MVC is described in detail in Design Patterns: Elements of Reusable Object-Oriented Design

`virtual int  getDrawingHeight() const = 0`

- Returns the largest value that will be used in the Y direction. This value is used in the scaling calculations.

`virtual int  getDrawingWidth() const = 0`

- Returns the largest value that will be used in the X direction. This value is used in the scaling calculations.

`virtual void  setDrawingHeight(int height)= 0`

- Set the largest value that will be used in the Y direction. This value is used in the scaling calculations. Parameter `height` is the largest value that will be used in the Y direction.

`virtual void  setDrawingWidth(int width)= 0`

- Set the largest value that will be used in the X direction. This value is used in the scaling calculations. Parameter `width` is the largest value that will be used in the X direction.

`virtual double  getScaling() const = 0`

- Returns the scaling factor between the data and screen units.

`virtual CRect  getDrawingArea() const = 0`

- Returns the area available in which to draw.

`virtual void  setDrawingArea(CRect area)= 0`

- Sets the area available in which to draw. Override this method if you want to modify the area before it is set, then call the base class' version with the new area. Parameter `area` is the area in which to draw.

`virtual void  setViewer(CView *view) = 0`

- Sets the view object that the data will be drawn into. Parameter `view` is the `CView` derived class whose draw method will call this class' draw method.

`virtual CView *  getViewer() const = 0`

- Gets the view object that the data will be drawn into. Returns the `CView` derived class whose draw method calls this class' draw method.

`virtual int  scaleX(int x, bool toScreen=true) const = 0`

- This method converts the parameters between class data ordinates and screen ordinates. Parameter `x` is the ordinate to convert, and `toScreen` is true if converting class data to screen data and false if converting screen data to class data. Returns the ordinate in the appropriate scale.

`virtual int  scaleY(int y, bool toScreen=true) const = 0`

- This method converts the parameters between class data ordinates and screen ordinates. Parameter `y` is the ordinate to convert, and `toScreen` is true if converting class data to screen data and false if converting screen data to class data. Returns the ordinate in the appropriate scale.

`virtual void  updateDrawScaling() = 0`

- Updates the scaling parameters. This should be manually called whenever changes are made that affect the values that getHeight() or getWidth() return.

See Figures 14 and 19 for examples of usage.

4.1.3 GDIViewable

This is a semi-complete version of the above interface. Classes inheriting from `IGDIViewable` that wish to use the behavior defined in this class can simply create an instance variable and implement most of the required methods as a call to the corresponding method of the `GDIViewable` variable.

`draw()` is the only method left abstract. However, classes using this method must make sure to set the drawing width and height to non-zero values.

The public member functions in the `GDIViewable` class are:

`virtual  ~GDIViewable()`

- Destructor.

`virtual void  draw(CDC &dc) const`

- Implementation of the `IGDIViewable` class method.

`virtual int  getDrawingHeight() const`

- Implementation of the `IGDIViewable` class method.

`virtual int  getDrawingWidth() const`

- Implementation of the `IGDIViewable` class method.

`void  setDrawingHeight(int height)`

- Implementation of the `IGDIViewable` class method.

`void  setDrawingWidth(int width)`

- Implementation of the `IGDIViewable` class method.

`double  getScaling() const`

- Implementation of the `IGDIViewable` class method.

`CRect  getDrawingArea() const`

- Implementation of the `IGDIViewable` class method.

`void  setDrawingArea(CRect area)`

- Implementation of the `IGDIViewable` class method.

`void  setViewer(CView *view)`

- Implementation of the `IGDIViewable` class method.

`CView *  getViewer() const`

- Implementation of the `IGDIViewable` class method.

`int  scaleX(int x, bool toScreen=true) const`

- Implementation of the `IGDIViewable` class method.

`int  scaleY(int y, bool toScreen=true) const`

- Implementation of the `IGDIViewable` class method.

`void  updateDrawScaling()`

- Implementation of the `IGDIViewable` class method.

See Figures 14 and 19 for examples of usage.

4.2 Smart Pointers

Smart pointers are objects which store pointers to dynamically allocated (heap) objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners.

Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed. In this way they make some parts of C++ behave in a fashion similar to that of a Java program. However, rather than objects being deleted when a garbage collector runs, they are destroyed when the last reference goes out of scope.

Smart Pointers have been acknowledged by the C++ community as something easy to write, but almost impossible to write correctly [3]. To that end the RoboCup team have chosen to adopt the entire boost::smart_ptr library [4]

[3]Alexandrescu 'Modern C++ Design', Sutter 'Exceptional C++'

[4]The full documentation on the boost::smart_ptr library is available from the boost website `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`

4.3 Sockets

The sockets library is implemented on top of the winsock library, which is provided as a core part of the Windows operating system. It is designed to be a resource management and abstraction layer to make working with the API easier. Both TCP and UDP wrapper classes were written, although it is intended that only the `UDP` class will ever be used in the RoboCup project. The public interface of the library offers a subset of the functionality of the winsock API, which is more fully described in the Microsoft documentation `msdn.microsoft.com/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp`. The layout of the classes can be seen in figure 4.

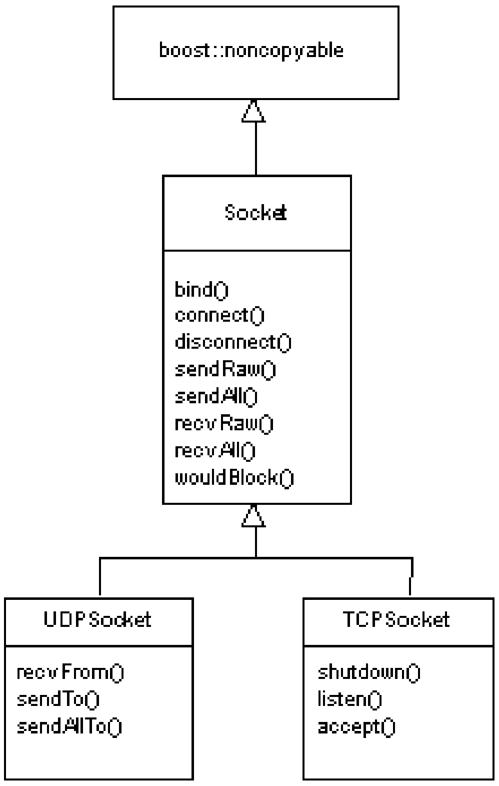


Figure 4: Frame Grabber Module - UML Class Diagram

4.4 Circular Buffer

Internally, the buffer acquires a critical section (a light-weight mutex) prior to adding the data to the buffer. In the case of adding images to the buffer, special care needs to be taken to ensure that images — which are stored as pointers to raw data — are dealt with in a thread safe fashion. This is implemented using memory copies. The following methods will be implemented in the `CircularBuffer` class:

```
virtual void CircularBuffer::write(T item )
```

- Adds an element to the circular buffer. If the buffer is full when write is called then the oldest data in the buffer is overwritten.

```
virtual void CircularBuffer::read(T& item )
```

- Retrieves an element from the circular buffer. If the buffer is empty when data is read the buffer will block until data is available.

```
virtual int CircularBuffer::size( )
```

- Retrieves the total number of elements in the buffer. This method is only relevant for debugging buffer capacity.

5 Image Processing Server

5.1 Image Server Overview

5.1.1 Purpose

To capture and process image data of the game environment, for analysis by the tactics engine.

5.1.2 Functionality

Pre Processing Stage:

1. Calibrate colours (off-line)
2. Calibrate field area (off-line)
3. Accept a client for data transfer

Normal Processing Procedure:

1. Capture image
2. Process image from camera
3. Analyse shapes from the image processing stage
4. Store position data
5. Send data to client

5.1.3 Interfaces

Interface with the Environment: A digital camera captures the environment data (an image of the field containing the robots, the ball, and the field characteristics).

Interface with the Tactics Engine: The Image Processing component acts as a server in that the Tactics Engine (the client) initiates a connection with the IP server, and the IP server begins streaming data to the Tactics Engine.

Interface with data storage: The IP Server writes the game environment data to a file.

Interface with the calibration GUI : The calibration GUI records a series of colour ranges, specified by a human from either live or prerecorded camera data. The thresholds are stored as files, which are read by the IP server.

5.2 Frame Grabber Module

To retrieve information flowing from the camera, a module is required to be involved in the video pipeline provided by windows. The most common way of achieving this is through the use of DirectShow. Software to interface with DirectShow needs to be implemented as COM objects supporting the appropriate interfaces and memory management systems. Once the software is compiled it will be linked as a Dynamic Library and registered with the Windows registry in a specific location. DirectShow can be a difficult API to work with; to prevent any potential problems, the Frame Grabbing filter is based on Microsoft's Grabber DirectShow Sample Filter, which ships with the DirectX SDK.

The frame grabber is triggered by a source when an image is ready to be processed. Once the trigger is received, the frame grabber transmits the image via a callback to an external piece of software, before resuming the standard processing behaviour. In most cases, the trigger for the frame grabber will be provided by a source filter which is provided by the camera manufacturer. The image will subsequently be deleted by a null renderer filter.

To make the filter easier to use with the rest of the image processing software some modifications were made to the callback system to allow it to interface in a more object-oriented manner. The only other change was to the transform method of the filter, which dispatches the callbacks. In all other respects, this code is identical to the Microsoft sample, so no significant design work was required to modify this code.

See SRS sections 6.1 and 6.2 for a detailed description of the requirements of this component

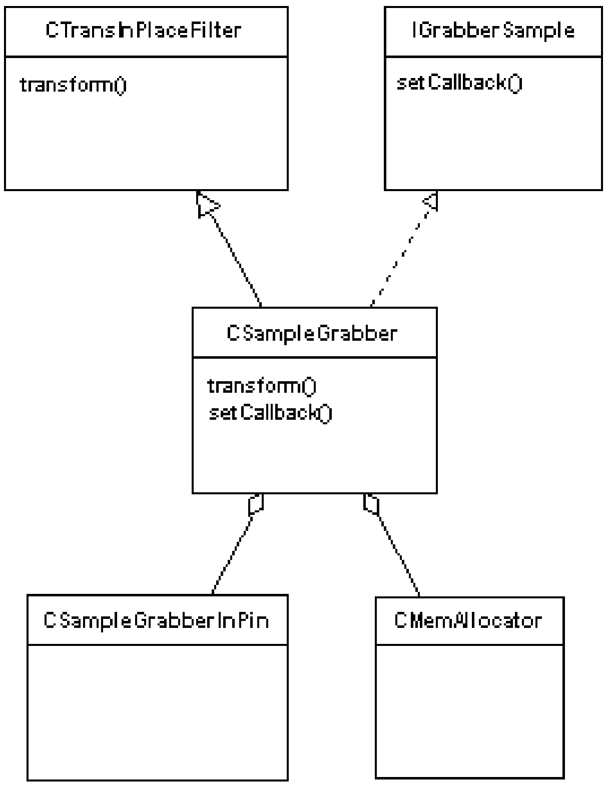


Figure 5: Frame Grabber Module - UML Class Diagram

5.3 Camera Module

The goal of the camera module is to provide a reusable interface to DirectShow via the Frame Grabber Module. The Camera Module initialises all of the components of the video pipeline, and starts the camera hardware transmitting data. The Frame Grabber Module exists within the video pipeline.

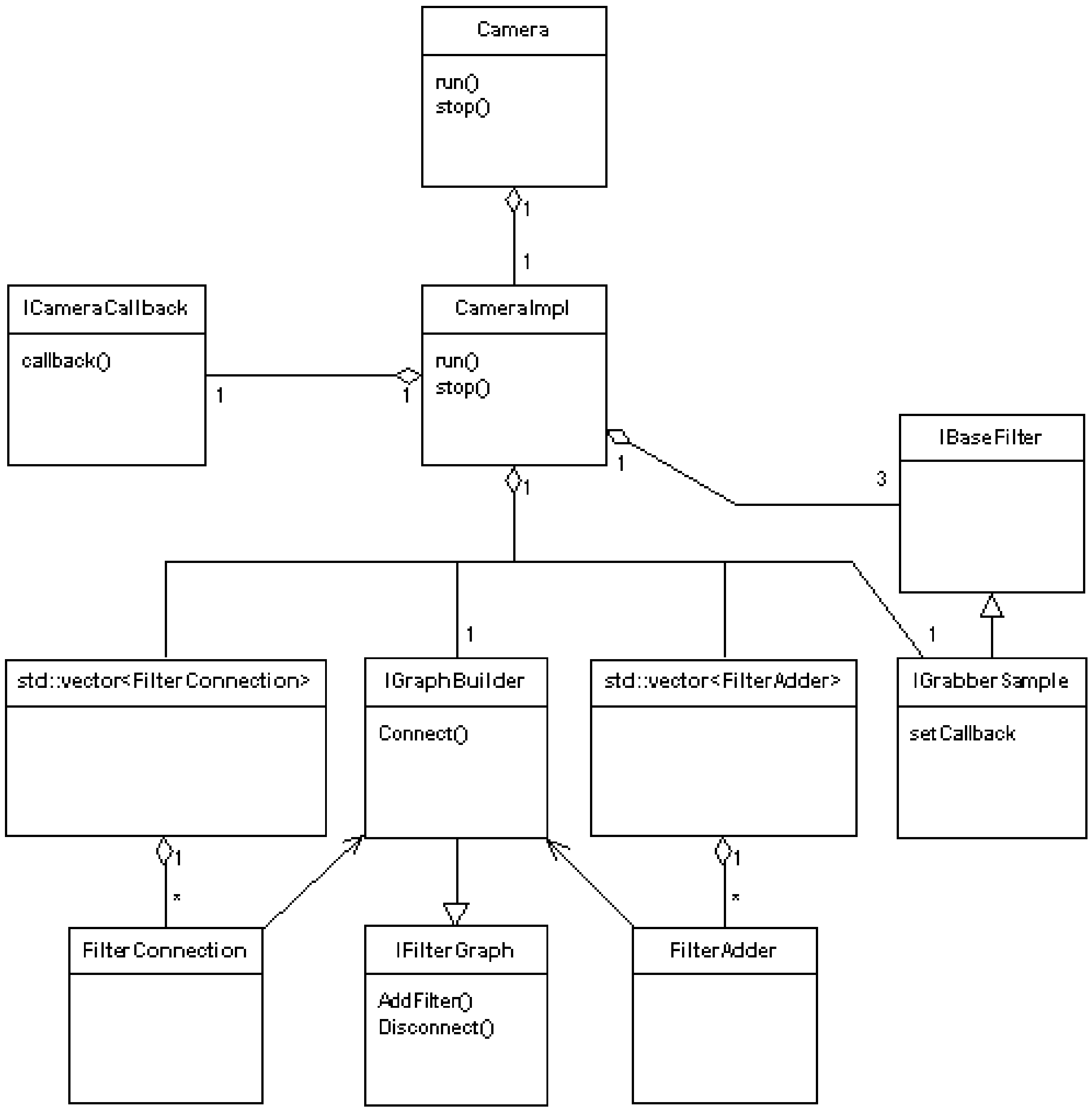


Figure 6: Camera Module - UML Class Diagram

The Camera and CameraImpl Classes
The design of the Camera module needs to ensure that cameras can be reused without conflict between components. This applies particularly to the Microsoft Foundation Classes (used for producing Graphical User Interface) and DirectShow. Previous work in this area by the 2003 RoboCup team and members of the 2004 team identified this requirement to be particularly challenging with regard to conflicting header files. The order of 'include' statements and the use of special header files control linker behaviour. The pointer-to-implementation approach [5] was chosen as a method to implement part of the design, with an outer `Camera` class delegating all responsibilities to the `CameraImpl` class. This allows `CameraImpl` to be fully independent of the usage patterns of `Camera`. The following section describes the `Camera` class:

```
Camera::Camera(int width,
```

[5]Described in 'Exceptional C++', by Herb Sutter

```
               int height,
               const GUID& subtype,
               ICameraCallback* pCallback)
```

- Builds a default camera.
- Parameters:
  - `width` - resolution width
  - `height` - resolution height
  - `subtype` - affects allowable widths and heights and colour depth. `MEDIASUBTYPE_Y411` should be the default.
  - `pCallback` - operation to perform on receiving a frame

```
HRESULT Camera::run( )
```

- Starts the filter graph running. After `run()` has been called the camera will start to collect frames and the callback will be triggered
- Exceptions: nothrow (ignores problems)
- Returns: DirectShow error code (see `IMediaControl::Run()`);

```
void Camera::setFormat(int width,
                       int height,
                       const GUID& subtype)
```

- Sets the media type and resolution to the desired settings, this allows some variation in the final format of the camera
- Parameters: `subtype` - default it to `MEDIASUBTYPE_Y411`
- Parameters: `width` and `height` - description of the resolution of a frame, typically 640 for width and 480 for height

```
HRESULT Camera::stop( )
```

- Stops the filter graph running. This method will prevent the camera from receiving frames until a corresponding run is called.
- Exceptions: nothrow (ignores problems)
- Returns: DirectShow error code (see `IMediaControl::Stop()`);

The FilterAdder and FilterConnection Classes
Leaks within DirectShow — especially if applications crash during development — can cause ongoing problems with release of resources, and degradation of the performance of the target machine. To combat this, the Camera module needs to ensure that resources are released properly, even in the presence of exceptions. The chosen approach was to consider each of the steps in starting the video pipeline to be a resource in itself, and to ensure that those operations could all be 'rolled-back' if any failed. `FilterAdder` and `FilterConnection` are the two resulting classes from this approach. A `FilterAdder` is instantiated, to control the adding of a single filter into the filter graph. A `FilterConnection` is instantiated to control the connection of a pair of filters. Both classes have no operations, and only exist to manage resources.

The ICameraCallback Interface
Although C++ does not directly support the concept of interfaces, they are used extensively by Microsoft's COM framework. The standard implementation is to use a class with no data and all pure virtual methods. The same concept was extended for use in specifying a callback. Any class that implements the interface and registers with the camera can receive a callback from the camera when an image arrives.

```
virtual HRESULT ICameraCallback::callback(BYTE* pData,
                                          __int64* StartTime,
                                          __int64* StopTime,
                                          BOOL TypeChanged)
```

- Called during the transform method of the sample grabber.
- Parameter `pData` - the media sample containing the image. See the DirectShow documentation for more details. The format of the raw data (a byte array) will be dependent upon the format of the camera for which the callback was generated.

5.4 Image Server

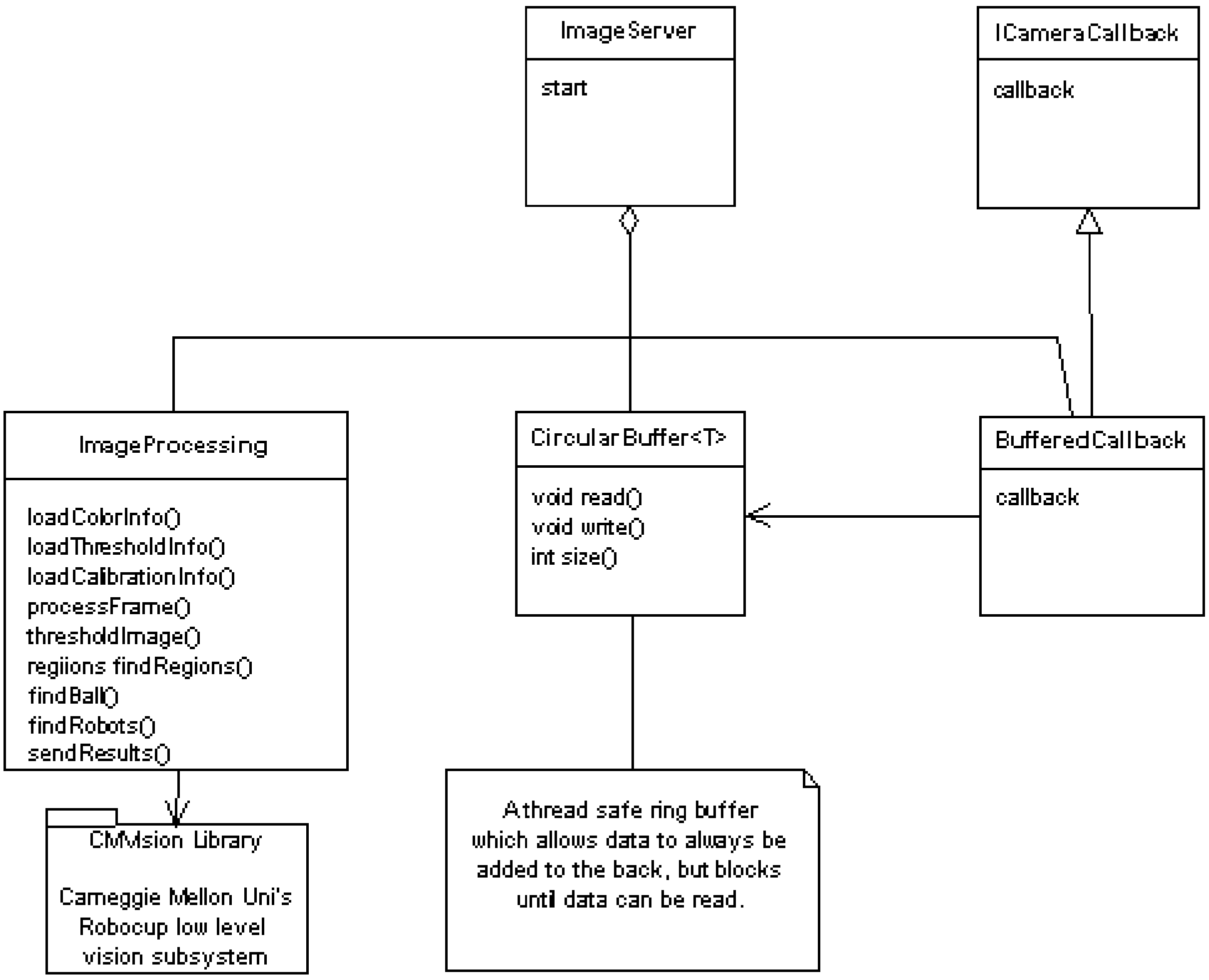


Figure 7: Image Server - UML Class Diagram

ImageServer Class
Provides an object oriented entry point to the application to make the project as easy to understand as possible for current and future developers, by maintaining software practices similar to those used by the Java community. This technique of providing an 'application' class also prevents a proliferation of global variables. It also provides a maintainable mechanism to separate recovery code from the main application in the case of exceptions.

```
virtual HRESULT ImageServer::start( )
```

- Called to start the application running (control should be transferred during the main function). After being called, the image server will run until a key is pressed. Although in a match situation the image server will need to run continuously, the software needs to be able to be demonstrated effectively without using the task manager to send a kill signal to the image server process.

BufferedCallback Class
A specialisation of the `ICameraCallback` class which places images received into a circular buffer for other code to deal with at a later time. This class is the last part of the process required to provide a video 'producer' (see figure 8).

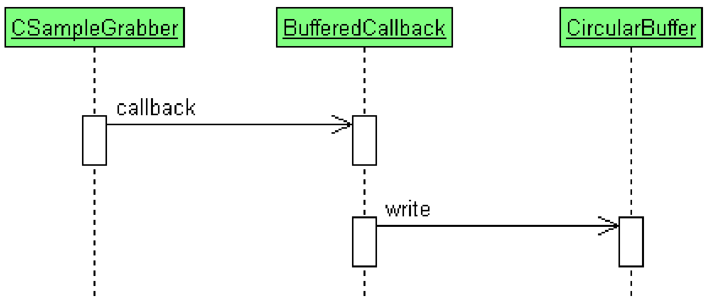


Figure 8: Video Producer - UML Sequence Diagram

ImageProcessing Class
The `ImageProcessing` class is the final stage of the video consumer process (see figure 9) which examines the images from the camera and processes the results. Additional facilities to those listed here will be added to this class as an aid for debugging, but will be removed for release builds. Debugging aids may include a visual display of the output from the image processing system. For example, a colour representation of the thresholded image with robots highlighted and marked with angle and id tags may be displayed.

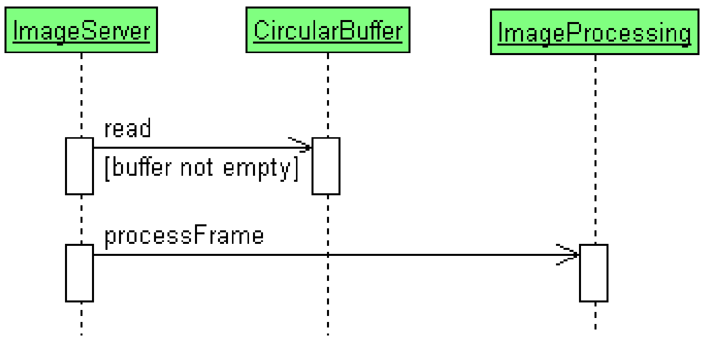


Figure 9: Video Consumer - UML Sequence Diagram

```
virtual void ImageProcessing::loadColourInfo(const char* filename)
```

- Loads the colour information from a file. The responsibility for loading colour information is implemented using the `CMVision` library function `CMVision::LoadColorInformation`. The results from the function will be stored into a structure provided by `CMVision`, for later use.

`virtual void ImageProcessing::loadThresholdInfo(const char* filename)`

- Loads the threshold information from a file. The output format of the thresholding information is a critical element for the performance of the entire image processing subsystem. Profiling of a naive implementation using an `if` statement to check entry conditions for each pixel showed that performance of the whole system dropped to 10 frames per second. One approach that avoids this problem is generating a very large (255^3 bytes) lookup table. Even with cache misses caused by a table of this size, performance returns to the theoretical maximum of 30 frames per second.

The file format for loading the threshold information is described below. `number_of_thresholds()` lists how many times the line below will be repeated. The values for Y, U and V represent upper and lower limits of the threshold. `Colour` represents the display colour necessary during debugging and the `name` string is to assist in situations where visual debugging of the image cannot be conducted.

```
number_of_thresholds(int)
name(string) Y_1(byte) Y_2(byte) U_1(byte) U_2(byte) V_1(byte) V_2(byte) Colour(DWORD)
```

`virtual void ImageProcessing::loadCalibrationInfo(const char* filename)`

- Loads the calibration information from a file. The format of the file will be determined as appropriate to ensure the best filtering of the results. It may include limiting sizes for significant features and the colour arrangement patterns (eg. the tops of the robots)

`virtual void ImageProcessing::ThresholdImage(unsigned char* pSrc, unsigned char* pDst)`

- Parameter: `pSrc` is the raw data of a Y411.
- Parameter: `pDst` is a pre allocated block of memory of sufficient size (1 byte per pixel) of the resultant thresholded image.
- Precondition: `loadThresholdInfo` has been called successfully.
- Loops through all of the pixels of the Y411 image, and uses the threshold information to convert each pixel from 3 bytes to 1 byte. Profiling showed that looping through the pixels of the image can be a bottleneck in some situations. Pointer arithmetic and reuse should be used where possible — function calls and `if` statements should be avoided in the inner loop.

`virtual void ImageProcessing::findBall(CMVision::color_class_state* pRegions)`

- Precondition: `loadCalibrationInfo` has been called successfully.
- Parameter: `pRegions` is the output from the sequence of `CMVision` library functions described in `ImageProcessing::ProcessFrame`.
- Scans through the list of regions, looking for the closest match to the ball, both by colour and by size. The resulting ball location and probability from this function will be stored into an internal structure which is accessible throughout the `ImageProcessing` class.

`virtual void ImageProcessing::findRobots(CMVision::color_class_state* pRegions)`

- Precondition: `loadCalibrationInfo` has been called successfully.
- Parameter: `pRegions` is the output from the sequence of `CMVision` library functions described in `ImageProcessing::ProcessFrame`.
- Scans through the list of regions, merging any overlapping black regions (the tops of robots). The list will be rescanned looking for the closest matches to a robot, both by colour and by size. Subsequently, `analyseRobot` will be called to retrieve more detailed information about the given robot (team, id and angle). The resulting robot location and probability from this function will be stored into an internal structure which is accessible throughout the `ImageProcessing` class.

```
virtual void ImageProcessing::ProcessFrame(unsigned char* pImg)
```

- Paramater: `pImg` is the raw data of a Y411.
- Precondition: `loadThresholdInfo` has been called successfully.
- Precondition: `loadCalibrationInfo` has been called successfully.
- Precondition: `loadColourInfo` has been called successfully.
- Precondition: that the size (width and height) of the `pImg` parameter is as expected.
- Runs a series of processing steps to produce the final output. Where possible, image operations should be performed 'in place' to minimise time spent copying images. Where a function should delegate responsibility to `CMVision`, it is noted with the function name. The processing steps are as follows:
  - Threshold the image
  - Clip the image
  - Run Length Encode the image (`CMVision::EncodeRuns`)
  - Connect the vertically connected runs (`CMVision::ConnectComponents`)
  - Extract the list of regions (`CMVision::ExtractRegions`)
  - Collect region information (`CMVision::SeparateRegions`)
  - Basic region filtering (`CMVision::SortRegions`)
  - High Level region filtering - find ball
  - High Level region filtering - find robots
  - High Level region measurement - analyse robots
  - Transmit Results via a socket

5.5 Image Calibration Tools

See the UID for more detailed information on the User Interface Design required for the threshold calibration tool. Software details of the implementation are deliberately unspecified due to the majority of the code being reliant upon a third party library (MFC). The one major requirement is that the file format matches the input format required by `ImageProcessing::loadThresholdInfo`.

6 Tactics

6.1 Tactics Overview

6.1.1 Purpose

To decide both team tactics and individual robot actions, and send these commands to the RF Server.

6.1.2 Functionality

1. Make a connection with the IP server
2. Grab environment data from the IP stream
3. Decide on team objective
4. Decide on individual robot actions based on team objective
5. Send robot instructions to RF server

6.1.3 Interfaces

Interface with the IP Server: The Tactics Engine acts as a client, in that it initiates a connection with the IP Server, and then begins to receive data over the socket connection.

RF Server: The RF Server will be passed the robot instruction by the Tactics Engine.

6.1.4 Data Flow

The `Coach` class is responsible for interpreting the image processing data, determining the current state of play (see figure 10) and the desired robot instructions. It then constructs packets to send to the robots via RF. The class structure of `Coach`, and its collaborators, can be seen in figure 11.

The sequence diagram shown in figure 12 better illustrates the interactions between the Image Processing module, the Tactics Engine and the RF Server. Packets may arrive from the IP module at any time. They trigger the `onImagePacket()` handler in the parent MFC class. The `onImagePacket()` method passes this data directly to `BlueCommsMgr` (for robot current position updates) and to `Coach` (for strategy updates). `Coach` then passes desired position packets to the robots via `BlueCommsMgr`.

6.2 Field

6.2.1 Field class

This class stores the information about all the known locations on the physical field. The data is constructed directly from IP data and is stored in the same units as that which is provided by the IP module. This data is used to construct a visual representation of the field, for use in the Tactics Engine (development mode) and the Tactics Debug Tool. The user will be able to drag-and-drop robots and the ball to simulate different setups (in the Debug Tool) or to specify desired positions (in the Tactics Engine). The latter of these two is achieved through the use of ‘ghosts’. When ghosts are enabled, the ball and robots have ghost versions which are used to indicate the desired position of the object. In the case of the ball, the ghost image can be used to specify ‘kick to’ commands. In the case of the robots, the ghost images can be used for ‘go to’ commands.

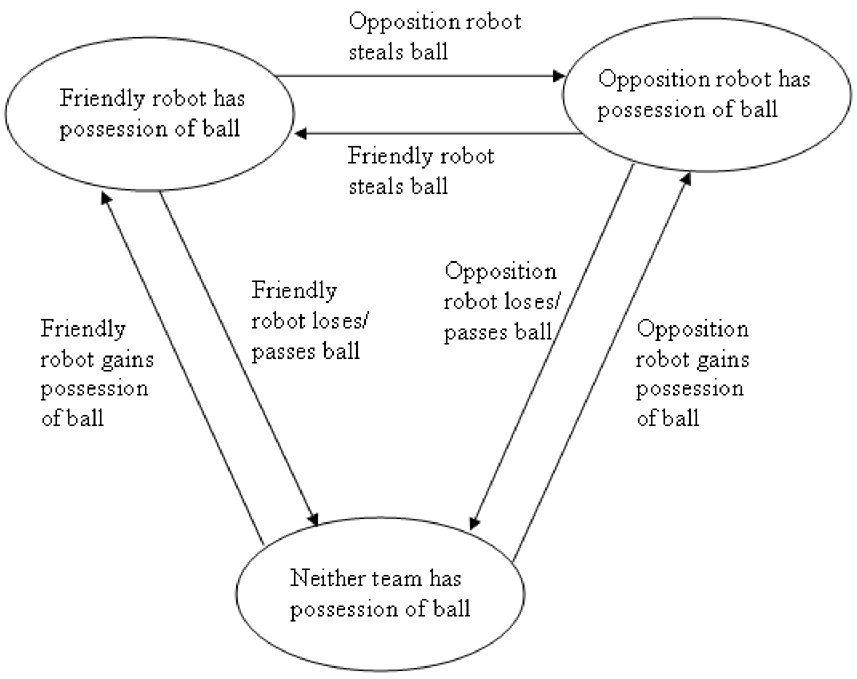

Opposition robot steals ball
Friendly robot has possession of ball
Opposition robot has possession of ball
Friendly robot steals ball
Friendly robot gains possession of ball
Friendly robot loses/ passes ball
Opposition robot loses/ passes ball
Opposition robot gains possession of ball
Neither team has possession of ball

Figure 10: Coach - UML State Transition diagram

Methods to be Used
The methods to be used in the Field class are:
Public

```
  Field(const std::string &fieldDataDir,
         bool ghostsEnabled=false,
         const std::string &filename="field_data.txt")
```

- Constructor which reads in the field setup from `field_data.txt` in the directory specified. Creates the Robots and the ball in default positions.

```
virtual ~Field()
```

- Destructor

```
BallPos const &  getBall(bool ghost=false) const
```

- Method which returns the position of the ball. The “ghost” parameter — if true — retrieves the actual position. If false, it retrieves the ghost (desired) position. Throws a `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

```
GoalPos const &  getGoal(ETeamColour type) const
```

- This method returns the position of the goal for the team indicated. The parameter type is either `BLUE` or `YELLOW`.

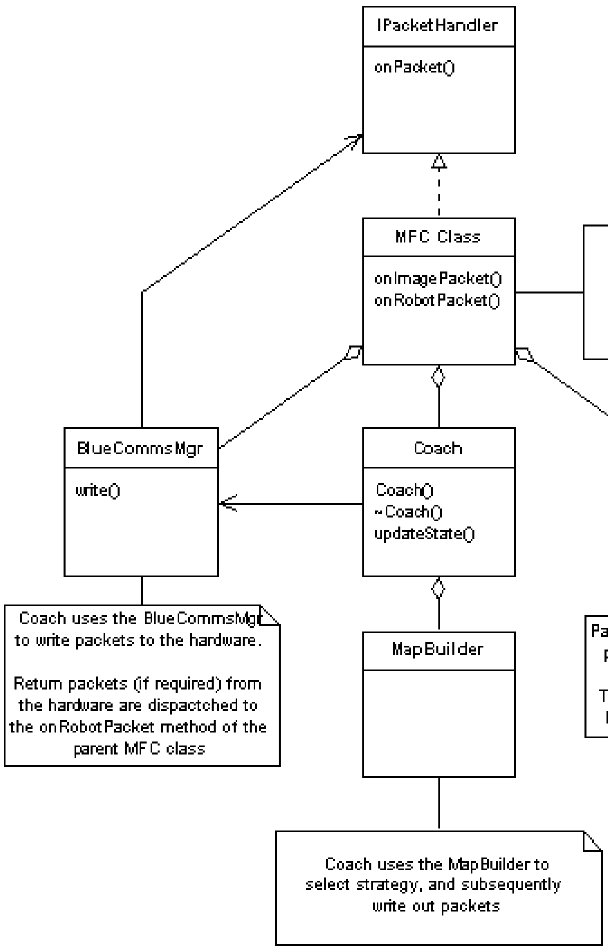

IPacketHandler
onPacket()
MFC Class
onImagePacket()
onRobotPacket()
onImagePacket passes data directly to BlueCommsMgr (robot position data updates) and to coach (for strategy outputs)
BlueCommsMgr
write()
Coach
Coach()
~Coach()
updateState()
ImageComms
Coach uses the BlueCommsMgr to write packets to the hardware.
Return packets (if required) from the hardware are dispacthed to the onRobotPacket method of the parent MFC class
MapBuilder
Packets may arrive from the image processing module at any time.
They trigger the onImagePacket() handler in the parent MFC class
Coach uses the MapBuilder to select strategy, and subsequently write out packets

Figure 11: Coach and its collaborators - UML Class Diagram

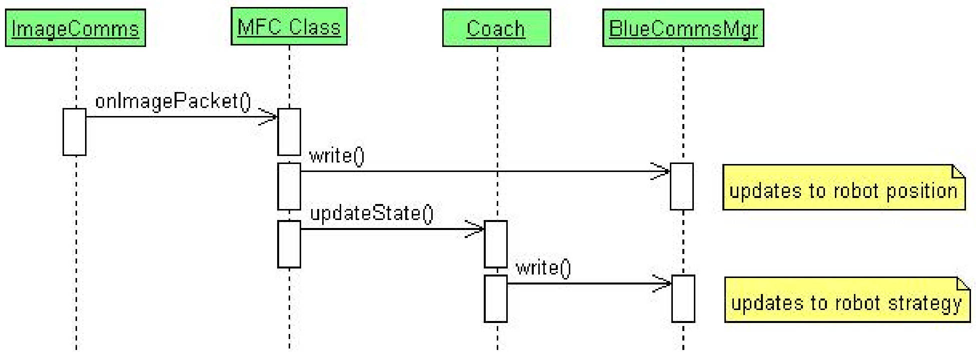


Figure 12: Communication between Image Processing, Tactics Engine and the RF Server - UML Sequence diagram

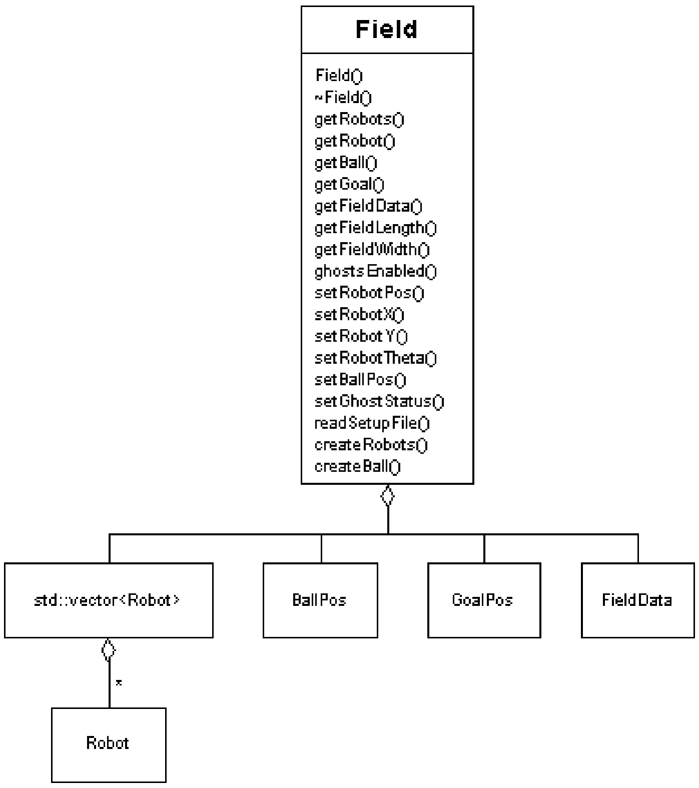


Figure 13: Field - UML Class Diagram

`RobotVector const &  getRobots(bool ghosts=false) const`

- The method returns a vector of the robot positions on the field. Parameter `ghosts` retrieves the actual positions if true. If false, it retrieves the ghost positions. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

`Robot const & getRobot(int whichOne, bool ghost=false) const`

- This method returns a robot's position on the field. Parameter `whichOne` is the index of the robot to return. Parameter `ghost` retrieves the actual position if true, or the ghost position if false. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

`int getFieldLength() const`

- This method returns the length of the field.

`int getFieldWidth() const`

- This method returns the width of the field.

`bool getWestGoalIsBlue() const`

- This method returns true if the west goal is the attacking goal for the blue team.

```
bool  ghostsEnabled() const
```

- This method returns true if ghost positions are being used.

```
void  setBallPos(int x, int y, bool ghost=false)
```

- This method updates the position of the ball. Parameter `x` is the x-ordinate of the ball, `y` is the y-ordinate of the ball. Parameter `ghost` sets the actual position if true, or the ghost position if false. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

```
void setRobotPos(int whichOne, int x, int y,
                 double theta, bool ghost=false)
```

- This method updates the position of a robot. Parameter `whichOne` is the index of the robot to set, `x` is the new x-ordinate of the robot, `y` is the new y-ordinate of the robot. `theta` is the new rotation angle of the robot and `ghost` sets the actual position if true or the ghost position if false. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

```
void  setRobotX(int whichOne, int x, bool ghost=false)
```

- This method updates the x-ordinate of a robot. Parameter `whichOne` is the index of the robot to set, `x` is the new x-ordinate of the robot and `ghost` sets the actual position if true, or the ghost position if false. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

```
void  setRobotY(int whichOne, int y, bool ghost=false)
```

- This method updates the y-ordinate of a robot. Parameter `whichOne` is the index of the robot to set, `y` is the new y-ordinate of the robot and `ghost` sets the actual position if true, or the ghost position if false. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

```
void  setRobotTheta(int whichOne, int theta, bool ghost)
```

- This method updates the theta ordinate of a robot. Parameter `whichOne` is the index of the robot to set, `theta` is the new theta ordinate of the robot and `ghost` sets the actual position if true, or the ghost position if false. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

```
void  setRobotHasBall(int whichOne, bool hasBall, bool ghost)
```

- This method updates whether the given robot has the ball. Parameter `whichOne` is the index of the robot to set, `hasBall` is true if the robot has gained possession of the ball and is false otherwise. `ghost` sets the possession for the actual robot if true or the possession for the ghost (desired pos) robot if false. Throws `std::out_of_range` exception if `ghost==true` and ghosts are not enabled.

```
void  setRobotRole(int whichOne, EPlayingPosition role)
```

- This method updates the playing role for a given robot (sets ghost as well). Parameter `whichOne` is the index of the robot to set and `role` is the new role for the robot.

```
void  setFieldPoints(const int data[], int points)
```

- This method sets the fixed location points on the field from IP data. Parameter `data` is the points as a series of x and y integers and `points` is the number of points in the array (should be (elements in data)/2).

`void setFieldPoints(const Point data[], int points)`

- This method sets the fixed location points on the field from IP data. Parameter `data` is the points as a series struct containing `x` and `y` as integers and `points` is the number of points in the array (should be the number of elements in data).

`void setGhostStatus(bool enabled)`

- This method sets whether ghosts (desired positions) are being used. Parameter `enabled` is true to enable ghosts or false to disable ghosts.

Protected

`FieldData const & getFieldData() const`

- This method returns a read-only version of the field data to be used by the class.

`void readSetupFile ()`

- This method reads the field data from the file specified in the constructor.

`virtual void createRobots()`

- This method accesses the fixed field positions to place robots in default positions.

`virtual void createBall()`

- This method uses the fixed field positions to place the ball in the default position.

6.2.2 GDIField

This class will be used in both the Tactics Engine GUI (section 6.4) and the Tactics Debug Tool (section 6.5). It inherits from `Field` and implements `IGDIViewable`. By overriding the `draw` method from `IGDIViewable` and passing through the other methods to a `GDIViewable` object, this class can be used to provide a visual representation of the data structures stored in Field.

See Sections 4.1.2 and 4.1.3 for descriptions of `IGDIViewable` and `GDIViewable`.

See Figure 14 for class relationships

Methods to be Used
Private (not shown in class diagram)
The following methods are suggestions to break up the implementation of the draw() method into logical parts.

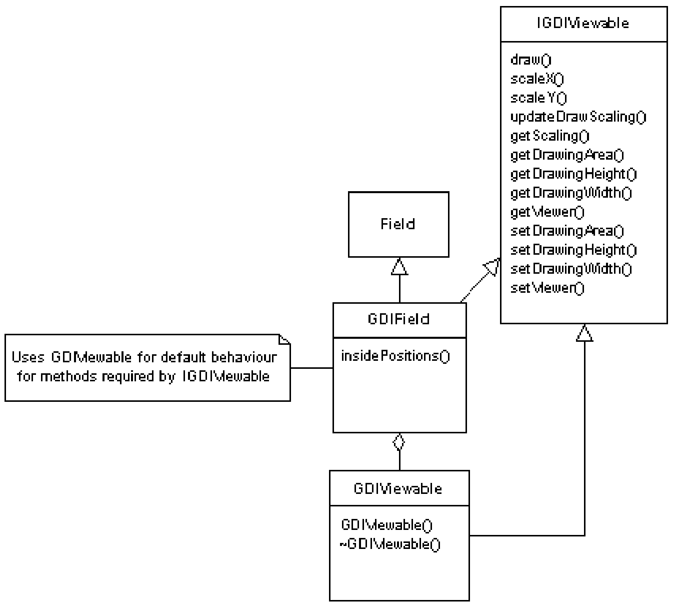


Figure 14: GDIField - UML Class Diagram

`void drawGrass(CDC & dc) const`

- Draw a green rectangle to represent the field's grass

`void drawWalls(CDC & dc) const`

- Draw white rectangles to represent the field boundary walls

`void drawLines(CDC & dc) const`

- Draw the white field lines (goal square, centre line and centre circle)

`void drawRobots(CDC & dc) const`

- Draw all the robots (and ghosts if enabled)

`void drawRobot(CDC & dc, Robot const &posOfRobot, bool ghost=false) const`

- Draw the given robot, draw it in lighter shades if it's a "ghost"

`void drawBall(CDC & dc) const`

- Draw the ball (and it's ghost if enabled)

`void drawBall(CDC & dc, BallPos const &posOfBall, bool ghost=false) const`

- Draw the ball at the given position, draw it in a lighter shade if it's the "ghost"

6.2.3 Data Structures

BallPos
See Figure 15
A defined structure which is essentially just a point. It is used to provide a concrete type which helps with understanding of the code.

GoalPos
See Figure 15
Two points to represent the boundary points of the goal mouth.

Robot
See Figure 15
Stores the following information about the robot:

- Location (x,y,theta)
- Team (colour)
- Identification (id) - related to id patches on top of physical robot
- Whether it has possession of the ball (hasBall)
- Playing Position (role) - e.g. Goalie

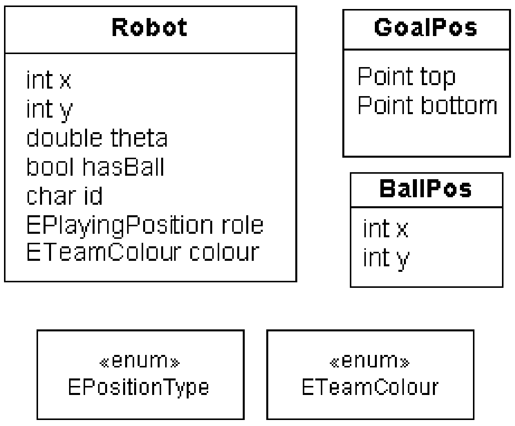


Figure 15: Structs for objects on field - UML Class Diagram

FieldData
See Figure 16
Provides storage for the following information:

- Location of the blue goal (west or east)
- Size of all the objects on the field (in image processing units).
- Co-ordinates of the calibration points that are provided by the image processing server.

6.3 MapBuilder

6.3.1 Potential Maps Construction

In order to decide on robot instructions, the RoboCup environment is represented as a potential field (or map). The construction of the potential maps is concentrated around the `MapBuilder` class. This

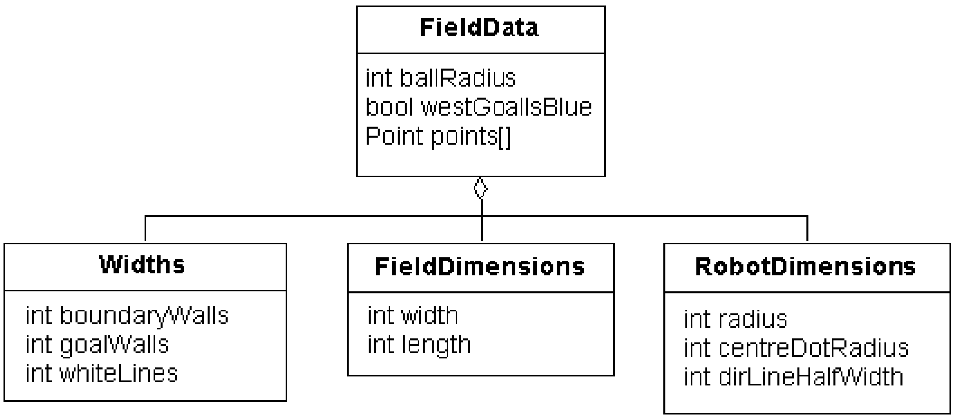


Figure 16: Structs for fixed data about field - UML Class Diagram

class calls the `LOSMap`, `DistanceMap` and `GoalMap` classes to aid it in performing the construction of the potential maps. The class structure of `MapBuilder` can be seen in figure 17.

The `GDIMapBuilder` class inherits from the `MapBuilder` class and serves the purpose of displaying the built map in the Tactics Debug Tool GUI.

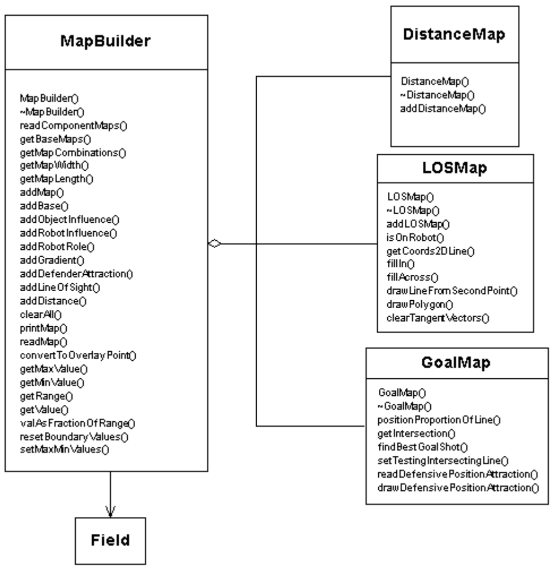


Figure 17: MapBuilder - UML class diagram

6.3.2 MapBuilder Class

The construction of the potential map is performed in this class. The potential map will be a two dimensional vector of integers. This will allow us to resize the potential map as required, and also resize the component maps, as not all will be of the same size. The components that are added to the potential map will be dependent on the type of component that is requested to be added by this class. The class itself will be required to read in from file and store information on the component maps, including their filename, description, grid and grid information. Not all of this is required for the actual construction of the potential maps, but is required by other areas of the system (such as the GUI).

The objects, structs and vectors to be used in the `MapBuilder` class are shown in Figure 18. They are used as follows:

- The `Field` object will allow `MapBuilder` access to all of the objects on the field, enabling it to draw a representation of these objects onto the potential map. The use of this object is vital to construction of the potential maps.
- The `PotentialMap` struct is used to store the information (two dimensional vector of integers, its width and length, highest and lowest values and the point at which these values occur) on the built potential map, as well as the components that go into creating the potential map.
- The `MapVector` is a vector of `ComponentMap` structs, which themselves inherit from the `PotentialMap` struct. These `ComponentMap` structs will be used to store further the information to the `PotentialMap` structs on filename and description. This will allow for easy storage and addition of the elements which combine to create the potential map.
- The `MapComboVector` is a vector of the `DefinedMapCombinations` struct which store information on the combinations of maps required to build particular types of potential maps.

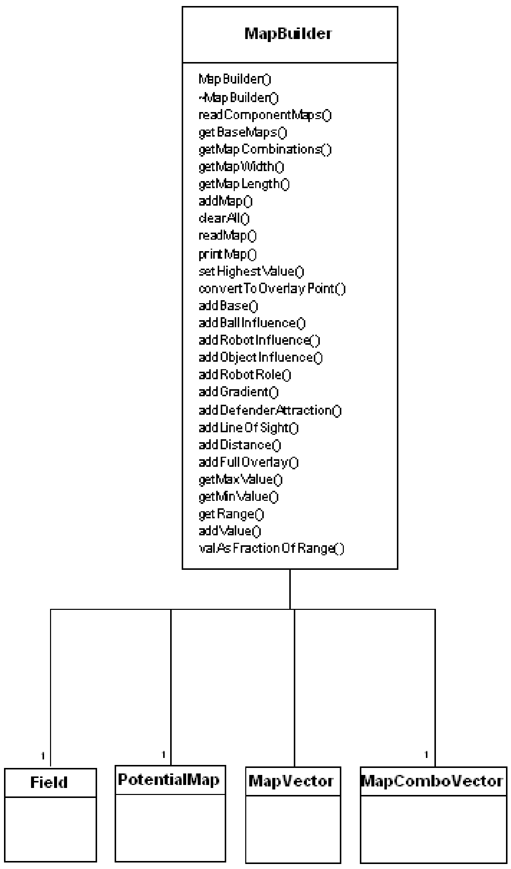


Figure 18: MapBuilder Class - Aggregation Diagram

Methods to be Used

Public

`MapBuilder(Field & theField, std::string mapDir)`

- Constructor for the creation of the `MapBuilder` object. It will set the required data (width, length, path, data file and print number) for the effective operation of this class. It will also need to call methods to create other remaining data needed in the class. Sizing the potential field two dimensional vector for representing the current situation of play will need to be completed here also.

`~MapBuilder()`

- Destructor used for the termination of the object.

`void readComponentMaps()`

- Method used for reading and storing all the component maps required for construction of the potential field. All information on the component maps (filename, grid, and description) will be read from file and stored in the vector of the `ComponentMap` structs.

`MapVector const &getBaseMaps()`

- Method to return the vector of the component maps.

`MapComboVector const &getMapCombinations()`

- Method to return the vector of defined map combinations.

`inline int getMapWidth() const`

- Method to return the width of the constructed potential map.

`inline int getMapLength() const`

- Method to return the length of the constructed potential map.

`void addMap(int indexOfMap, int selectedRobot)`

- Method called to add a component to the potential map. The method will determine the component to add to the map based on the index it is passed.

`void clearAll()`

- Method that resets all values in the constructed map to zero.

`void readMap(ComponentMap & map) const`

- Method to read the contents of a component map and store it into the `ComponentMap` vector. It reads from a file the values to be stored in the two dimensional vector for the component map, as well as its filename and description.

`void printMap()`

- Method to print the constructed potential map to file.

`void setHighestValue()`

- Method to find the highest and lowest values in the constructed map (used for colour scale in debug tool) and the position that they occur. The highest and lowest values found and the position of the highest value are set in the `PotentialMap` struct variables.

Protected

`void convertToOverlayPoint(int& x, int& y)`

- Method to convert the object coordinates (x and y) from image processing coordinates to potential field coordinates.

`void addBase(EComponentTypes indexOfMap)`

- Method used to add the base map overlay to the potential field. The base overlay is to provide the potential map with a weighting toward the team's attacking goal. The index passed determines which base map component (east or west) will be add to the potential map. This method calls the `addFullOverlay` method to add the overlay to the potential map.

`void addRobotInfluence(EComponentTypes indexOfMap, RobotVector const &theRobots)`

- Method to add all other robots to the potential field. The index passed to the method determines which component map will be used to represent the robots on the potential map, as there are several ways of representing them. Also, the robot for which the map is drawn is not added to the potential map — hence the need for the `selectedRobot` private integer for this class, which represents the index of the robot for which the potential map is created in the vector of robots. This method will call `addOjectInfluence` to add all of the robots.

`void addBallInfluence(EcomponentTypes indexOfMap, int x, int y)`

- Method to add the ball influence to the potential map. This method is needed to determine if it is necessary to add the ball influence. It checks if the ball is in possession and if it is not the ball will be added to the potential map. The method will call the addOjectInfluence method to add the ball.

`void addObjectInfluence(EComponentTypes indexOfMap, int x, int y)`

- Method to add an object (ball or robot) to the potential field. The index passed to the method represents the component map to add to the potential map. The addition to the potential map is based on the (x,y) coordinates of the object passed as parameters to the method. This allows the object to be represented accurately in the potential field.

`void addRobotRole(EComponentTypes indexOfMap, Robot const &theRobot)`

- Method used to add the current role of the robot to the potential map. Each robot will have a role to play at any given stage of the game. The index of the role overlay component map is passed to the method along with the robot who's role is to be added. The role will only be added if the role of the robot corresponds to the component map passed to the method. This method calls the `addFullOverlay` method to add the overlay to the potential map.

`void addGradient(EComponentTypes indexOfMap, RobotVector const &robots)`

- Method to add a gradient overlay to the potential map, which discourages a robot from being in the centre of the field if a team-mate has the ball. This will allow for a clearer shot on goal. The method checks to see whether the robot has a team-mate who is in possession of the ball. If this is true, then the gradient component map will be added to the potential map, based on the y-ordinate of the team-mate with the ball.

```
void addDefenderAttraction(EComponentTypes indexOfMap, RobotVector const &robots,
                           GoalPos const &goal, DefenderType type)
```

- Method to add attractive potential to the field at a desired point between an opposition robot and the goals. Used to defend robot with ball by staying to the defensive side of the robot, or to defend the robot who could receive a pass. Based on where the opposition robots are, the type of defender the robot is and where the goals are, place an area of attraction in this region in the potential field. This method collaborates with the `GoalPos` class.

```
void addLineOfSight(RobotVector const &robots)
```

- Method used to discourage a robot from trying to move to a position on the field which is blocked by another robot. The vector of robots is passed to the method. This method collaborates with the `LOSMap` class.

```
void addDistance(Robot const &theRobot)
```

- Method to encourage a robot to attempt to move to a position which is close by (and therefore more likely to remain a reachable position). It alters values in the potential field based on their distance from a robot. The closer they are, the more their potential will be increased. This method collaborates with the `DistanceMap` class.

```
void addFullOverlay(EComponentTypes indexOfMap)
```

- Method called to add an overlay which is of the same size as the potential map to the potential map.

```
inline int getMaxValue() const
```

- Method that returns the highest value in the constructed potential map.

```
inline int getMinValue() const
```

- Method that returns the lowest value in the constructed potential map.

```
inline int getRange() const
```

- Method that returns the range, ie the difference between the highest and lowest values in the constructed potential map.

```
inline void addValue(int length, int width, int value)
```

- Method called to add a value to the potential map. The value to be added and the location to add it are passed to the method.

`double valAsFractionOfRange(int length, int width) const`

- Return the fraction a value held in the potential map versus the overall range in the potential map. Used for colour range in other areas of the system.

`void resetBoundaryValues()`

- Method to reset the boundary values — highest and lowest — held in the `PotentialMap` struct.

6.3.3 DistanceMap Class

The `DistanceMap` class is used by the `MapBuilder` class to perform calculations based on distance from a robot. It is a simple class used only for the purpose of calculating distance. The class alters the potential map it receives to reflect the distance calculations.

Methods to be Used

The methods to be used in the `MapBuilder` class are:
Public

`DistanceMap(PotentialMap & theMap)`

- Constructor for the creation of the `DistanceMap` object. This constructor sets a reference to the potential map it is passed from the `MapBuilder` class and sets the class variables for width and length of this potential field.

`~DistanceMap()`

- Destructor, used for termination of the `DistanceMap` object.

`void addDistanceMap(int &x, int &y)`

- Method which alters values in the potential field based on distance from the x and y coordinates passed to it. The closer the coordinates in the potential map are to the coordinates received by the method the more they will be multiplied by.

6.3.4 LOSMap Class

This class is used to discourage a robot from trying to move to a position on the field which is blocked by another robot. Based on the position of another robot on the field, this method will alter the area behind this other robot to make it unattractive. This is to ensure the robot for which the potential map is constructed does not try to access an area it cannot directly get to. This class alters the potential map it receives to reflect the line of sight application.

Methods to be Used

The methods to be used in the `LOSMap` class are:
Public

```
LOSMap(PotentialMap & theMap)
```

- Constructor for the creation of the `LOSMap` object. Receives a reference to the two dimensional vector of the potential map and sets the relevant input data file for reading.

```
~LOSMap()
```

- Destructor, used for termination of the `LOSMap` object.

```
void addLOSMap(int x0, int y0, int x1, int y1)
```

- Method to add the line of sight to the potential map. The coordinates `x0` and `y0` are for the robot for which the map is being drawn, whilst `x1` and `y1` are the coordinates of the robot which is blocking the line of sight.

```
void addLOSMapForSmallGaps(int x0, int y0, std::vector robotPositions)
```

- Method to add the line of sight to the potential map for those gaps between robots through which a robot cannot fit. It uses the coordinates `x0` and `y0` for the robot who the potential map is for and a vector containing the other robot positions.

Private

```
bool isOnRobot(int robotX, int robotY, int x, int y)
```

- Method used to check whether the given coordinates fall within the envelope of the robot. Returns true if the coordinate is within the robot envelope. Takes the center coordinates of the robot (`robotX, robotY`) and the coordinates to check (`x, y`).

```
void getCoords2DLine(int x0, int y0, int x1, int y1,
                     std::vector<int> & xCoords,
                     std::vector<int> & yCoords)
```

- Method that returns the coordinates of a 2-dimensional line drawn between two specified points. Take starting (`x0,y0`) and ending (`x1,y1`) coordinates of a line as parameters as well as references to vectors in which to store the line coordinates. All `x` coordinates of the line are stored in the `xCoords` vector and `y` coordinates in the `yCoords` vector.

```
void fillIn(std::vector<int> leftX,
            std::vector<int> & rightX,
            int startY, int dy, int lastXOne, int lastXTwo)
```

- Method used to fill in the polygon when a line can be drawn between the two tangent lines in the x-direction. Used in blocking out the area behind a robot.

```
void fillAccross(std::vector<int> lineX, int startY, int dy, int dx)
```

- Method used to fill in a section of the grid to one side (in the x-direction) of a specified line. Used in blocking out area behind a robot, when a line cannot be drawn between the two tangent lines in the x-direction.

```
void drawLineFromSecondPoint(int x0, int y0, int x1, int y1,
                             int & lastX, std::vector<int> & xCoords)
```

- Method used to construct a line from a specified origin which passes through a specified way-point, and fill in this line from the way-point to the edge of the grid. Used in blocking out the area behind a robot.

```
void drawPolygon(int p1_1_x, int p1_1_y, int p1_2_x, int p1_2_y,
                 int p2x, int p2y, std::vector<int> & xOrds)
```

- Method used to fill in the area behind robot 2 which blocks the line of sight of robot 1. Calls some methods mentioned above to aid in performing the task of blocking the area behind a robot.

```
void clearTangentVectors()
```

- Method which clears the vectors which contain the ordinates for the most recently drawn LOS Map.

6.3.5 GoalMap Class

The GoalMap class draws maps and implements functions related to shots on goal and defending the goal. It determines the best area for a robot to shoot at when going for goal and also determines the best area for a robot to be placed when defending an opposition robot from scoring a goal.

Methods to be Used
The methods to be used in the `GoalMap` class are:
Public

```
GoalMap(PotentialMap & theMap)
```

- Constructor to create the `GoalMap` object. It receives a reference to the potential map being constructed and also sets relevant variables to suit the operation of the object.

```
~GoalMap()
```

- Destructor, used for the termination of the `GoalMap` object.

```
void positionProportionOfLine(int x0, int y0, int x1, int y1,
                              int percentAlongLine)
```

- Method to add the defensive position encouragement map, at a specified distance along a line between two specified points. Used to place a robot at a set distance from the opposition robot to the goal, 80 per cent to be used for attacking defender.

```
int getIntersection(int x0, int y0, int x1, int y1, int xToCheck)
```

- Method to determine the y-ordinate (if any) that a specified line (line 1) intersects a given line perpendicular to the x-axis (line 2) (such as the goal-line). This method could be used to determine whether the current trajectory of the ball is on target to score a goal, and if so, the coordinate to which the goal-keeper should move to block the shot.

```
int findBestGoalShot(int x0, int y0, int xToCheck,
                     int upperGoalY, int lowerGoalY,
                     std::vector<int> & robotXOrds,
                     std::vector<int> & robotYOrds)
```

- Method to determine the y-ordinate (if any) of the best possible shot on goal. Based on position of the robot in respect to the goal, find the best location to shoot to, if any exists (ie robots not blocking entire goal).

```
Coord findBestShootingPos(int selfX, int selfY, int xToCheck,
                          int upperGoalY, int lowerGoalY,
                          std::vector<int> robotXOrds,
                          std::vector<int> robotYOrds)
```

- Method to determine the coordinate which represents the best possible goal-shooting position. Based on position of the robot in respect to the goal, find the best location to move to for a shot on goal, if any exists.

```
void getTwoStagePosCommand(int selfX, int selfY,
                           int ballX, ballY,
                           int & x1, int & y1, int & theta1,
                           int & x2, int & y2, int & theta2)
```

- Method to determine the two sequential positions that the robot should go to in order to pick up the ball. The first position instruction will involve a rotation to align the front of the robot with the ball, to prepare for impact with the ball. The second position will encourage the robot to move 'through the ball' to gain possession.

Private

`void readDefensivePositionAttraction()`

- Method used to read the defensive position attraction map from a file.

`void drawDefensivePositionAttraction(int x, int y)`

- Method used to draw the defensive position map over the specified position.

`void readYGradientCentreBias()`

- Method used to read the y-gradient centre bias attraction map from a file.

`void readXGradientForwardBias()`

- Method used to read the x-gradient forward bias attraction map from a file.

`void addYGradientCentreBias()`

- Method used to add the y-gradient centre bias attraction map to the existing map.

`void addXGradientForwardBias(int goalX)`

- Method used to add the x-gradient forward bias attraction map to the existing map.

6.3.6 GDIMapBuilder Class

This class will be used in the Tactics Debug Tool (section 6.5).
It inherits from `MapBuilder` and implements `IGDIViewable`. By overriding the draw method from `IGDIViewable` and passing through the other methods to a `GDIViewable` object, this class can be used to provide a visual representation of the combined map built by the `MapBuilder`.

See Sections 4.1.2 and 4.1.3 for descriptions of `IGDIViewable` and `GDIViewable`.

See Figure 19 for class relationships

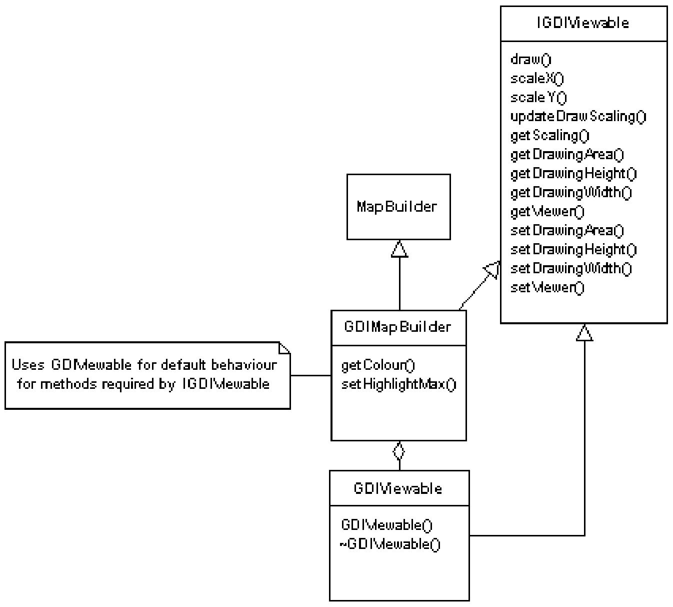


Figure 19: GDIMapBuilder - UML Class Diagram

Public Methods

`void setHighlightMax(bool highlight)`

- Set whether the highest values should be drawn in a different manner to the rest of the map

Protected Methods

`COLORREF GetColour(double position,double start,double end) const`

- Return a COLORREF for `position` in hot-cold representation
  Values of `position` near `start` should return 'cold' colours (blue)
  Values of `position` near `end` will return 'hot' colours (red)

6.3.7 Data Structures

6.3.7.1 PotentialMap

A struct which stores information on a potential map. This includes variables for the map width and length, the highest and lowest potentials, the point where the highest value occurs and the potential map grid itself. The struct will use a the following methods:

`inline int getValue(int x, int y) const`

- Method which returns the value in the potential map grid located at the x and y coordinates passed to it.

`void resizeGrid(int x, int y)`

- Method for resizing the potential map grid. Used to set up the grid for addition of values.

6.3.7.2 ComponentMap

This structure is used to store information on the component map which create the final constructed potential map. It inherits from `PotentialMap` and offers two extra variables for filename and description, which will be used for reading in from a file.

6.3.7.3 DefinedMapCombinations

Structure used to store a description of the types of potential maps which may be created, and a list of component map identifiers (integers) which go into creating the potential map.

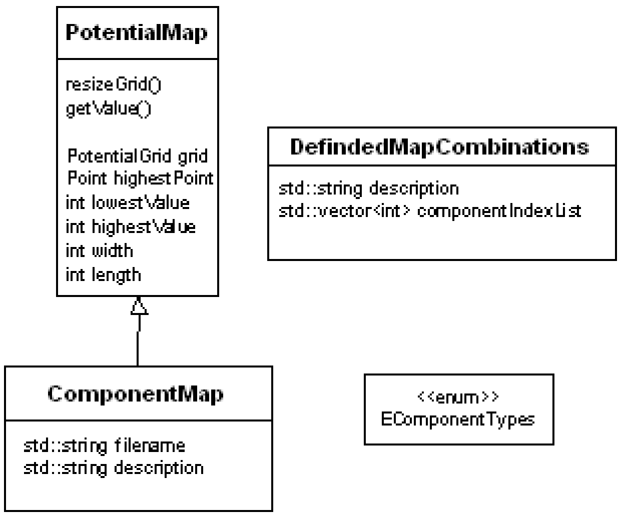


Figure 20: Structs for objects on the field - UML Class Diagram

6.4 Tactics Engine GUI

6.4.1 Purpose

The Tactics Engine GUI will provide:

- A graphical representation of the software's knowledge of the Field
- The ability to send high level robot commands to test robot behaviour
- The interface to start and stop a game

6.4.2 Functionality

1. Respond to new IP data
2. Update visual representation of environment
3. Work out team objective from data
4. Work out robot commands from data
5. Accept desired positions for a specified robot
6. Accept desired kicking power for a specified robot
7. Ability to send the following commands:
   - Move to a given position
   - Fire the kicking mechanism
   - Turn on/off the ball possession roller

6.4.3 Interfaces

Interface with Image Processing: The Tactics Engine GUI has a connection via a socket to the image processing, over which it receives a stream of object positions.

Interface with the RF Server: The Tactics Engine GUI has a bluetooth connection to each of the robots, over which it can send command packets.

6.4.4 Class Description

See figure 21 for class relationships

The following classes will be mostly generated by Visual Studio.
Changes that need to be made are listed under each class.

`CMainFrame`

- Provided by Single Document Interface (SDI) project wizard
- Add window splitter to have a GUI control view and a Field Visualisation View
- Place `CGuiView` and `FieldView` in splitter

`CTacticsEngineApp`

- Provided by Single Document Interface (SDI) project wizard
- Change initial active view to `CGuiView`

`CGameTab`

- Subclass of MFC class `CDialog` created from ‘dialog’ resource in Visual Studio
- Use external library (ColorControls) for red emergency button.
- Add event handlers for buttons

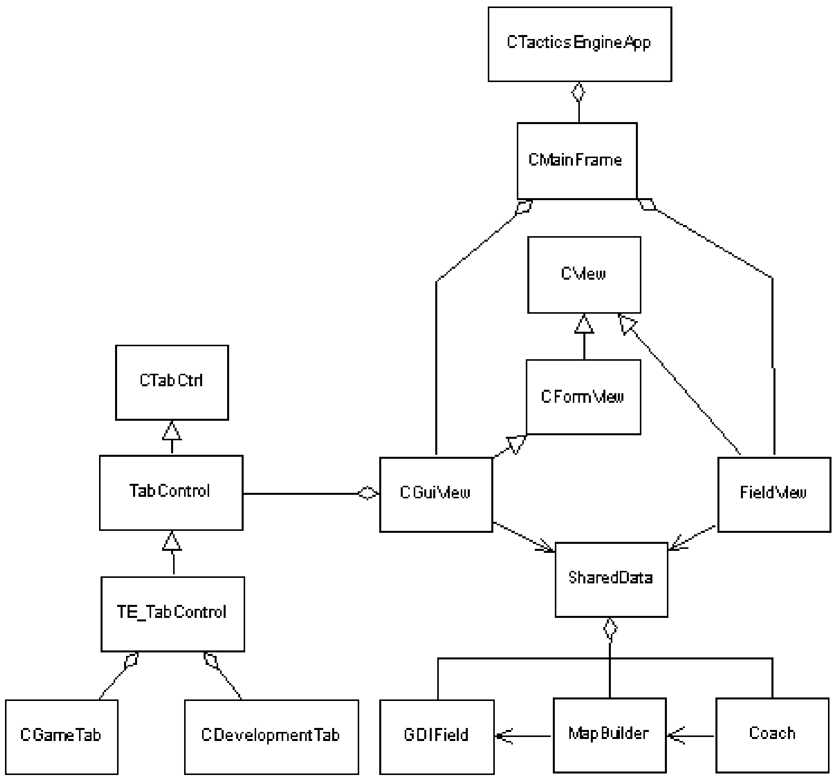


Figure 21: Tactics Engine - UML Class Diagram

- See figure 22 for control layout

`CDevelopmentTab`

- Subclass of MFC class `CDialog` created from 'dialog' resource in Visual Studio
- Use external library (ColorControls) for red emergency button.
- Add event handlers for buttons, text boxes and drop-down boxes.
- See 23 for control layout

`CGuiView`

- Subclass of MFC class `CFormView` created from 'dialog' resource in Visual Studio
- Add a `TE_TabControl` object and associated code

`TabControl`

- Subclass of MFC class 'CTabCtrl'
- Add code to dynamically resize to largest component tab

`TE_TabControl`

- Subclass of `TabControl`

- Add `CDevelopmentTab` and `CGameTab` objects for tabs

`FieldView`

- Modify from view provided by Single Document Interface (SDI) project wizard
- Subclass of MFC class `CView`
- Add event handlers for mouse clicks to have drag and drop functionality
- Forward draw method to `GDIField` object

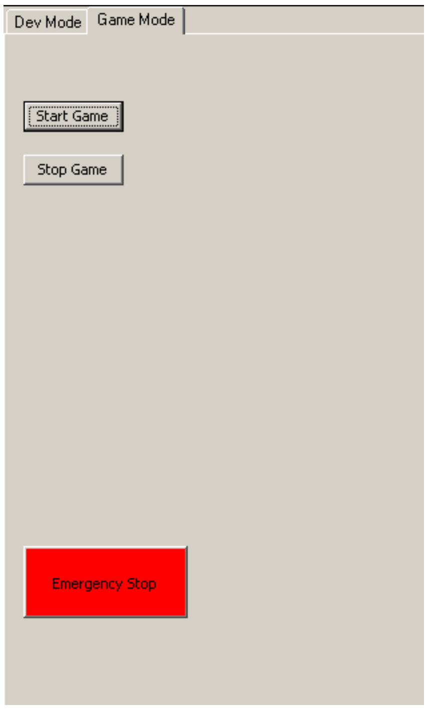


Figure 22: Control layout for GameTab in Tactics Engine GUI

6.5 Tactics Debug Tool

6.5.1 Purpose

To enable the developers to analyse the tactics decisions which are made during a game.

6.5.2 Functionality

1. Retrieve environment data
2. Display visual representation of environment
3. Display visual representation of team objective data
4. Display visual representation of individual robot decision data
5. Update to next/previous frame

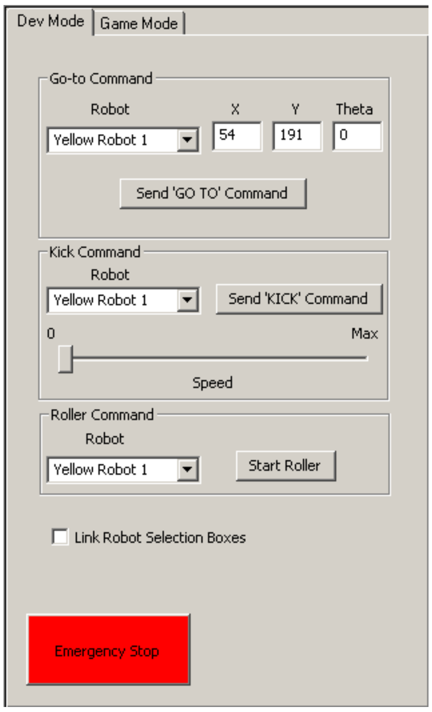


Figure 23: Control layout for DevelopmentTab in Tactics Engine GUI

6.5.3 Interfaces

Interface with data storage: The Tactics Debug tool reads the environment data from a file.

Interface with the Tactics Engine: The Tactics Debug Tool sends the environment data to the Tactics Engine, and receives the recreation of the tactics decision that was made during the game.

6.5.4 Class description

See figure 24 for class relationships

The following classes will mostly be generated by Visual Studio.
Changes that need to be made are listed under each class.

`CMainFrame`

- Provided by Single Document Interface (SDI) project wizard
- Add window splitter to have a GUI control view and two drawing views
- Place `CGuiView` and `DrawingFrames` in splitter

`CDebugToolApp`

- Provided by Single Document Interface (SDI) project wizard
- Change initial active view to `CGuiView`

`DrawingFrames`

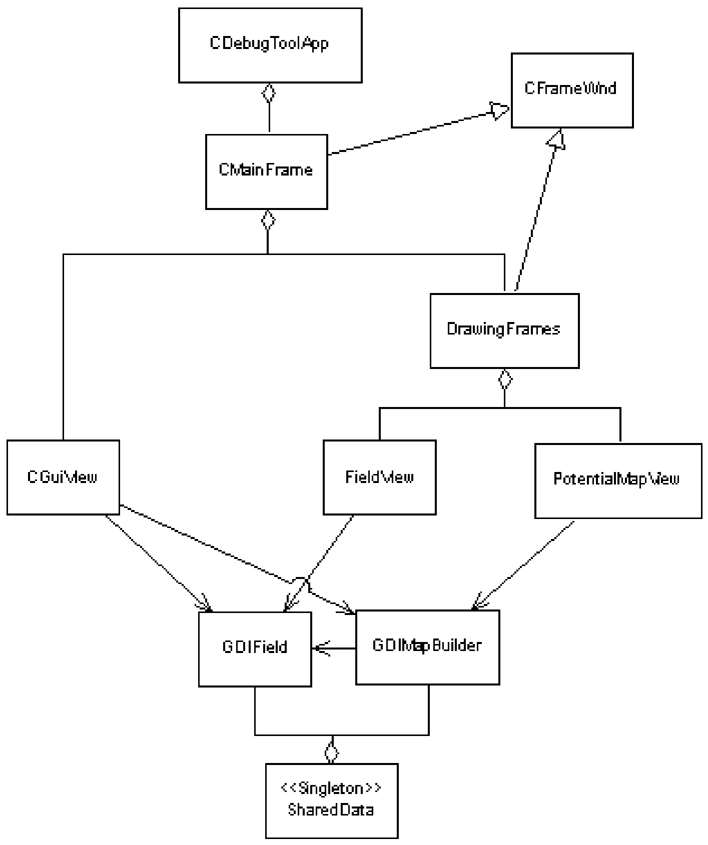


Figure 24: Tactics Debug Tool - UML Class Diagram

- Copy structure from `CMainFrame`
- Place `FieldView` and `PotentialMapView` in splitter

`CGuiView`

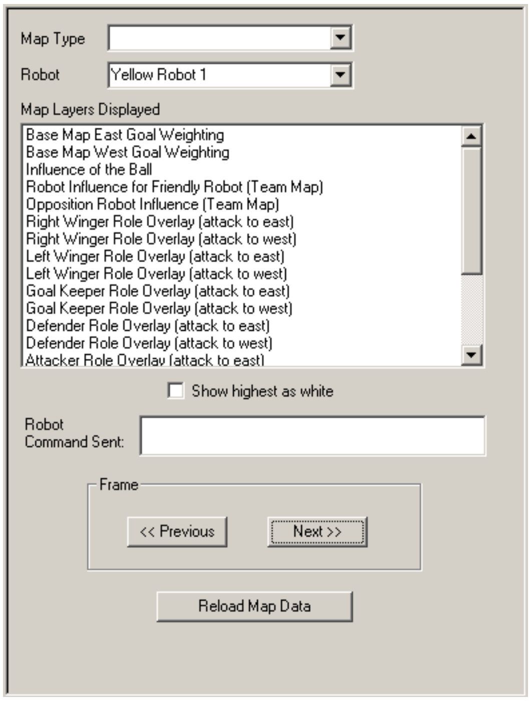
- Subclass of MFC class `CFormView` created from 'dialog' resource in Visual Studio
- Add event handlers for buttons, list boxes and drop-down boxes.
- See figure 25 for control layout

`FieldView`
This class should be the same as the one used for Tactics Engine and could be copied

- Modify from view provided by Single Document Interface (SDI) project wizard
- Subclass of MFC class `CView`
- Add event handlers for mouse clicks to have drag and drop functionality
- Forward draw method to `GDIField` object

`PotentialMapView`

- Modify from view provided by Single Document Interface (SDI) project wizard
- Subclass of MFC class `CView`
- Forward draw method to `GDIMapBuilder` object


Map Type
Robot: Yellow Robot 1

Map Layers Displayed
Base Map East Goal Weighting
Base Map West Goal Weighting
Influence of the Ball
Robot Influence for Friendly Robot (Team Map)
Opposition Robot Influence (Team Map)
Right Winger Role Overlay (attack to east)
Right Winger Role Overlay (attack to west)
Left Winger Role Overlay (attack to east)
Left Winger Role Overlay (attack to west)
Goal Keeper Role Overlay (attack to east)
Goal Keeper Role Overlay (attack to west)
Defender Role Overlay (attack to east)
Defender Role Overlay (attack to west)
Attacker Role Overlay (attack to east)
Show highest as white
Robot Command Sent:
Frame
<< Previous
Next >>
Reload Map Data

Figure 25: Control layout for GUIView in Tactics Debug Tool

7 RF Server

7.1 RF Server Overview

7.1.1 Purpose

To receive robot instructions from the Tactics Engine and Image Processing modules, and to transmit these instructions to the robots.
To receive diagnostic data from the robots.

7.1.2 Functionality

1. Receive robot instruction packet
2. Broadcast robot instruction packet
3. Receive current position instruction packet
4. Broadcase current position instruction packet
5. Receive robot data packet

7.1.3 Interfaces

Interface with the Tactics Engine: The RF Server will be passed the robot instruction by the Tactics Engine.

Interface with the IP Server: The RF Server will be passed the robot instruction by the IP Server.

Interface with the Robot Embedded Software : The RF Server will pass the instruction to the Robot Embedded Software. The Robot Embedded Software will pass a robot data packet to the RF Server.

These interfaces can be seen in figure 11.

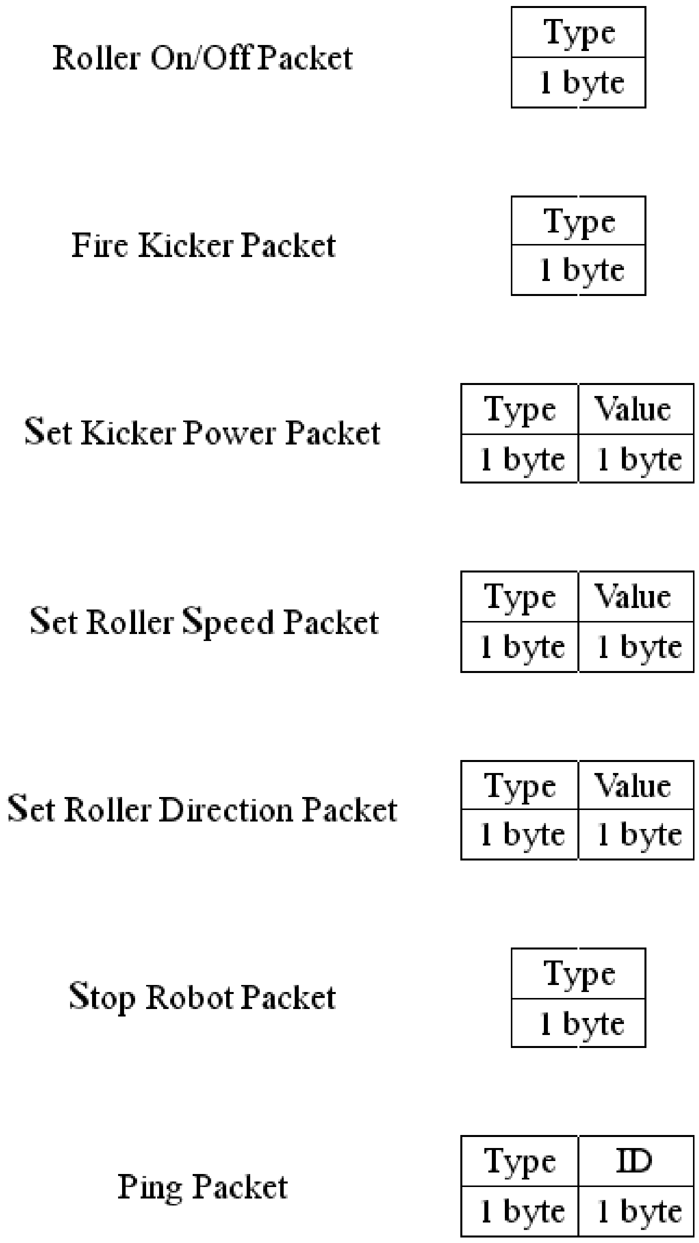
7.2 Packet Design

The Tactics Engine and IP Server will communicate to the robots via the RF Server to send robot instructions and updates. This will require packets to be sent to the robots which encode the data for the relevant command/update. The required packets will vary in length. Two options are available for dealing with different length packets:

- Pad out all packets to the size of the largest packet. This has the advantage of simpler code for encoding and decoding the packets — packets can be structs which can be converted to an array of bytes for sending via RF. The disadvantage with this method is that even small request or single-action command packets will have to be as big as the relatively large multi-action command packets.
- Have varying length packets. This has the disadvantage of being more difficult (read: time-consuming) to design and code, but has the advantage of increasing the message throughput over the RF.

The second option (varying length packets) has been chosen due to its increased efficiency at run-time.

The improvements to the robot communication system in 2004 will allow a higher-level protocol to be used to ensure that data is being accurately transmitted. This protocol will be implemented in two steps, allowing for flexibility and extensibility. The first step is to 'serialise', or encode a data structure into a series of bytes and attach a type field. Some examples of the layout of various types of data structures suitable for transmission can be seen in Figures 26 and 27.



Roller On/Off Packet

Type
1 byte

Fire Kicker Packet

Type
1 byte

Set Kicker Power Packet

Type	Value
1 byte	1 byte

Set Roller Speed Packet

Type	Value
1 byte	1 byte

Set Roller Direction Packet

Type	Value
1 byte	1 byte

Stop Robot Packet

Type
1 byte

Ping Packet

Type	ID
1 byte	1 byte

Figure 26: Packet Data - Structure Diagram

After serialising the raw data, a header, checksum and footer will be appended to the data. This ensures that new types of packets can be added with minimal effort in the future, and that old packets can be modified easily. The layout of the packet ready for transmission over the RF link can be seen in Figure 28.

Set PID (wheel) Velocities

Type	Velocity 1	Velocity 2	Velocity 3
1 byte	2 bytes	2 bytes	2 bytes

Desired position

Type	x-Ordinate	y-Ordinate	angle
1 byte	2 bytes	2 bytes	2 bytes

Current postion

Type	x-Ordinate	y-Ordinate	angle
1 byte	2 bytes	2 bytes	2 bytes

Two-stage Desired Position

Type	x1	y1	Theta1	x2	y2	Theta2
1 byte	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes

Figure 27: Packet Data - Structure Diagram(Continued)

STX	Header	Length	... Data ...	Checksum	EOT
1 byte	2 bytes	1 byte	Length Bytes	1 Bytes	1 byte

Figure 28: Sample Packet - Structure Diagram

By use of the `STX` and `EOT` chars, the software used to decode the packets can easily identify the start and finish of each packet, regardless of the length. By specifying the length of the actual data contained within the packet, the decoding software is able to look for the correct number of data bytes.

7.3 Packet Data

Since the length of the data, and the data itself, will vary from packet to packet, the design of the packet data creation software is very important. Inheritance has been used to take advantage of the fact that the creation of each packet type will involve the conversion of data into a string of bytes (via the `toString()` member function), and the decoding of a packet will involve the extraction of the same data from the string of bytes (via the `fromString()` member function). This also allows polymorphism to be used as a clean way of creating the packets in the calling software. Each packet data type has been implemented as a class which inherits from the base `PacketData` class, which has the abstract member functions `toString()` and `fromString()`. This can be seen in figure 29. The decoding method for extracting data from a packet will be implemented as a finite state machine, which will maintain state across several packet arrivals, ensuring that no packets are dropped without due cause. A simplified version of the state transition diagram without guard conditions can be seen in figure 30. The structure of the code required to support the communications is in figure 31

Description of Functions
The functions in figure 29 fall into seven categories:

1. Default Constructors (eg PIDVelData()): will be used to create default packet data objects. The data in these objects can be added using the relevant set-type function (defined below). This enables the calling software to create a single default object statically, and populate it dynamically. This avoids unnecessary calls to 'new', and the resultant memory management issues.

2. Constructors with parameters (eg PIDVelData(int v1, int v2, int v3)): can be used to create objects dynamically with the relevant data from the parameters. These will probably not be used this year.

3. Destructor: provided as a standard in all RoboCup C++ software.

4. `toString()` member functions: used to convert the member variables to the appropriately sized string of chars.

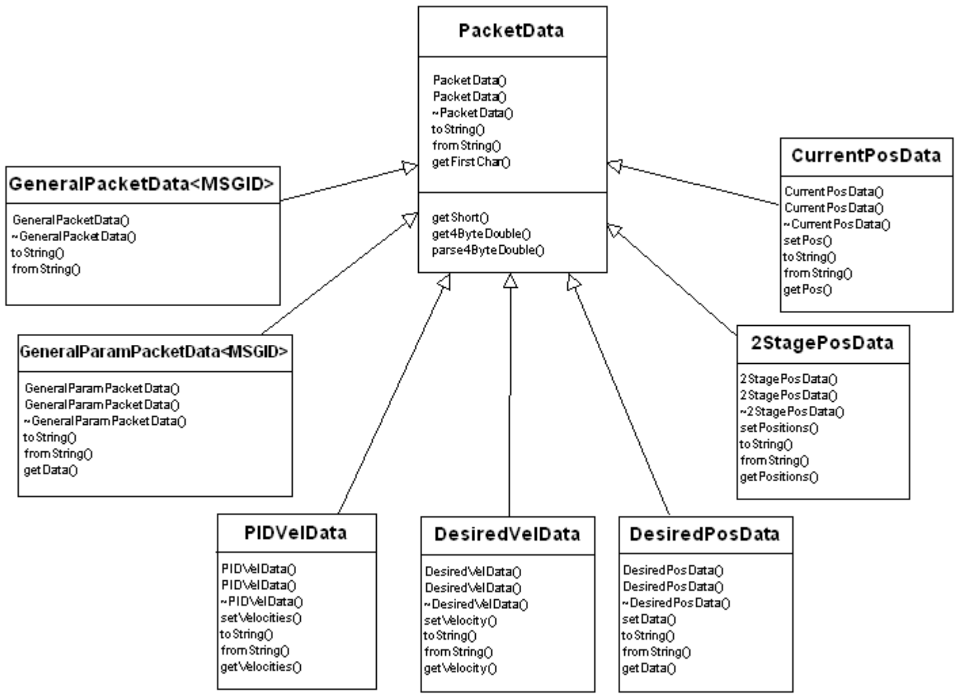


Figure 29: Tactics Engine Packet Data - UML Class Diagram

5. `fromString()` member functions: used to decode the string parameter into the appropriate member variables
6. Set-type functions: used to set the member variables for a statically created object (as described above) using the parameters of the function.
7. Get-type functions: used to retrieve the member variables by the calling software, using the reference parameters of the function.

The 2004 RoboCup team will not use all of these functions on both sides of the system. Some of them will be written to enable testing, and for easy conversion to C-code on the robot side.

7.4 Sending of Packets

The design of the software required to interface with the Bluetooth hardware is shown in the following section.

Public

```
void BluetoothComms::abortIO()
```

- Method to abort the attempted IO read/write
- Parameters: None.

```
DWORD BluetoothComms::read(char *  msg,
    DWORD  toRead)
```

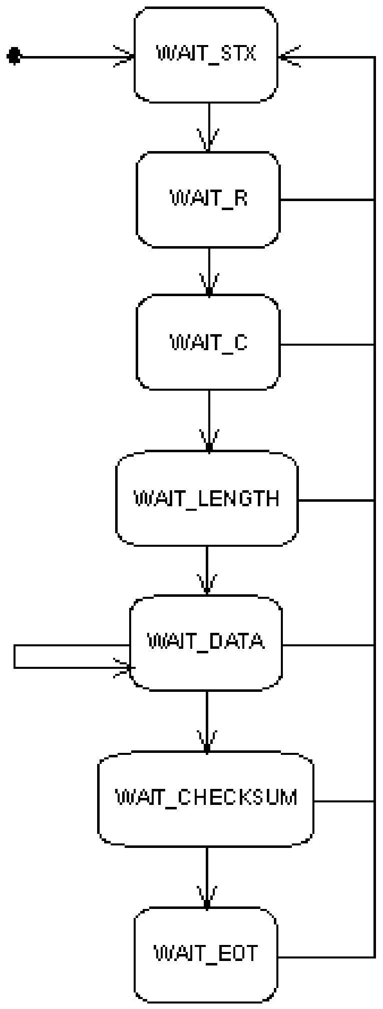


Figure 30: Decoding of Packets - UML State Transition Diagram

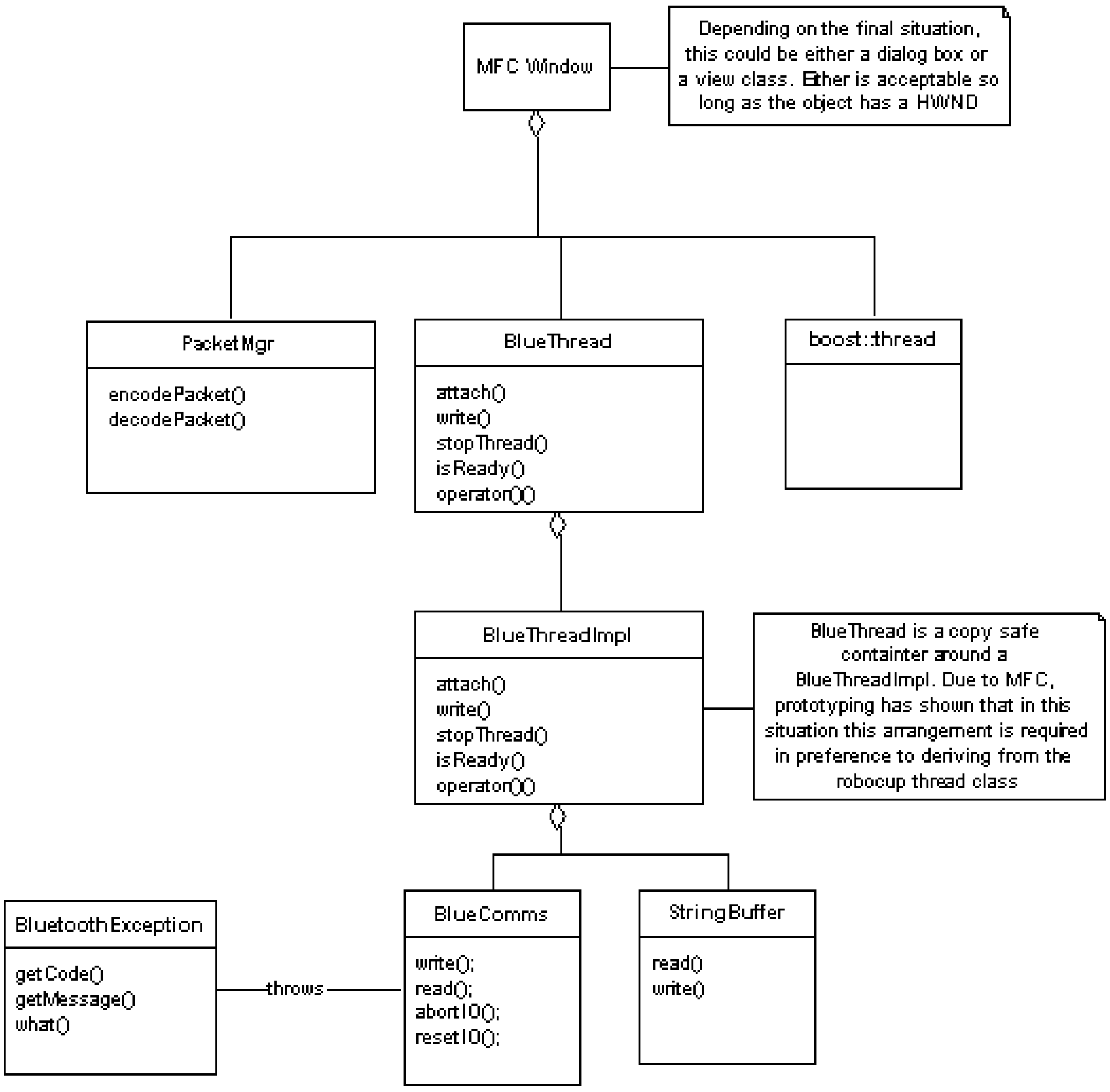

MFC Window
Depending on the final situation, this could be either a dialog box or a view class. Either is acceptable so long as the object has a HWND
PacketMgr
encodePacket()
decodePacket()
BlueThread
attach()
write()
stopThread()
isReady()
operator()()
boost::thread
BlueThreadImpl
attach()
write()
stopThread()
isReady()
operator()()
BlueThread is a copy safe containter around a BlueThreadImpl. Due to MFC, prototyping has shown that in this situation this arrangement is required in preference to deriving from the robocup thread class
BluetoothException
getCode()
getMessage()
what()
throws
BlueComms
write();
read();
abortIO();
resetIO();
StringBuffer
read()
write()

Figure 31: Bluetooth Communications - UML Class Diagram

- Read a string from the Bluetooth server
- Parameters:
  - `msg` - the char array in which to store the message
  - `toRead` - description

```
void BluetoothComms::resetIO()
```

- Reset the IO.
- Parameters: None.

```
DWORD BluetoothComms::write(const char *  msg,
   DWORD  toWrite)
```

- Write a string to the Bluetooth server
- Parameters:
  - `msg` - the message to write
  - `toWrite` - description

```
void BlueCommsMgr::attach(HWND  hWnd,
                           IPacketHandler *  pHandler)
```

- description
- Parameters:
  - `hWnd` - description
  - `pHandler` - description

```
void BlueCommsMgr::create(const std::string &  name,
                           DWORD  baudRate)
```

- description
- Parameters:
  - `name` - description
  - `baudRate` - description

```
void BlueCommsMgr::onPacket(WPARAM  wParam,  LPARAM  lParam)
```

- should be called from the attached HWND for on WM_USER_BLUETHREAD
- Parameters:
  - `wParam` - description
  - `lParam` - description

```
void BlueCommsMgr::startThreads()
```

- deferred launch of all threads
- Parameters: None.

```
void BlueCommsMgr::stopThreads()
```

- handler for WM_DESTROY
- Parameters: None.

```
void BlueCommsMgr::writeTo(size_t  i,
                           const std::string &  str)
```

- call this to write to a given robot
- Parameters:
  - `i` - the index of the robot, in the order of create
  - `str` - description

```
void BlueThread::attach(HWND  owner,
                        StringBuffer *  pStrBuffer,
                        const std::string &  commName,
                        DWORD  id,
                         DWORD  baudRate = CBR_57600)
```

- attach this thread to a window and pseudo construct this object
- Parameters:
  - `owner` - owner will receive a WM_USER_BLUETHREAD message on packet arrival
  - `pStrBuffer` - arriving packets will be written to this buffer
  - `commName` - the name of the com port to connect to
  - `id` - the lParam of the WM_USER_BLUETHREAD will contain this value
  - `baudRate` - speed of communications with the port

```
void BlueThread::operator()()
```

- the thread function for this object
- Parameters: None.

```
 DWORD BlueThread::write(const char *  msg,
                        DWORD  toWrite)
```

- write a message to the robot connected to the comms owned by this thread
- Parameters:
- `msg` - Data to write.
- `toWrite` - the length of msg in bytes

`virtual void IPacketHandler::onPacket(const std::string & str) [pure virtual]`

- description.
- Parameters: None.

`std::string PacketMgr::buildPkt(const std::string & pktstr) [static]`

- converts DataPacket string into a BlueTooth packet string
- Parameters: None.

`void PacketMgr::parsePacket(const std::string & input)`

- extracts the data from the packet
- Parameters: None.

`void PacketMgr::setPacketHandler(IPacketHandler * pHandler)`

- specify the destination of packets
- Parameters:
- `pHandler` - Packet method will be called when a complete packet arrives.

Private

`void BlueThread::run()`

- One cycle of the listening loop.
- Parameters: None.

`bool BlueThread::shouldStop()`

- return true if this thread is ready to stop
- Parameters: None.

8 Robot Embedded Software

The embedded software covers all code running on the CPU core (i.e. microcontroller) local to each robot. This core is programmed in ANSI C, which is not an object-oriented language; however, various OO design philosophies are used. Code is organised into 'modules', which are 'classes' in the standard object-oriented sense. However, that convention is not used in this document so as to not imply an OO language.

8.1 Robot Embedded Software Overview

8.1.1 Purpose

To convert the robot instruction packet into actions, such as activating the driving motors and/or the kicker and roller.

8.1.2 Functionality

1. Receive robot instruction packet
2. Determine required wheel velocities according to instruction
3. Activate driving motors according to instruction
4. Activate kicker and roller according to instruction
5. Send robot data packet to the RF Server

8.1.3 Interfaces

Interface with the RF Server — Incoming: The Robot Embedded Software will receive robot instructions and updates from the RF Server.

Interface with the RF Server — Outgoing: The Robot Embedded Software will send robot data to the RF Server

Interface with the Environment: The robot will change its relationship with the environment through actions (translation, rotation and ball interaction).

8.2 Code Layout

Individual modules are as follows:

- `main.c` - Call init. routines on 'boot-up' and endlessly loop, calling packet recv. function
- `uart.c` - Handle transmission and receiving of data bytes over Bluetooth, including a circular buffer
- `packet.c` - Packet parsing
- `pid.c` - PID closed-loop feedback control

- `pwm.c` - Pulse Width Modulation (for motor speed control)
- `tacho.c` - Tachometer (for registering encoders counts and calculating wheel speeds)
- `kicker.c` - Kicker control
- `roller.c` - Roller control
- `general.c` - Library routines such as reading switches etc.

Each module contains code unique to that 'process' (e.g. interfacing with particular port pins), and basic routines for utilising that code. Communication between modules is to be through shared variables, as it provides higher performance than function calls (the microcontrollers aren't very fast, so optimising code instruction length is very important in obtaining maximum plant performance). This practice is shunned in PC application programming, but the simpler, low-level structure of embedded code makes it easier to keep track of these variables (e.g. no threads to worry about). Any variables that need not be shared across modules are to be restricted to file scope by declaring them "static".

8.2.1 Processing

The code on this microcontroller will operate around interrupts. It is similar to rudimentary multi-tasking, but fully implemented in microcontroller hardware. Interrupts can call code in other modules, but all must relinquish control of the processor after completion. There are no eternal loops except for the `main()` function, used to iterate through a packet-receiving state machine.

Timer interrupts are used to periodically execute code for a certain task; these can almost be thought of as 'threads'. Other minor interrupts are used to maintain or keep track of system states, respond to an action etc.

Interrupts are used as follows:

- UART Byte Received - Executed when a single byte is received via Communications. Places byte into a Circular Buffer
- PID Timer - Updates the PID controller states (e.g. current speed, desired speed)
- TACHO Timer - Sample wheel speed values, resets counters
- Edge Triggers - Counts pulses for wheel encoders
- PWM Timer - Maintains 4 x hardware pulse width modulation waveforms

8.2.2 Functional Overview

The `main()` function is not discussed in detail as it has a simple task: to initialise various modules once at startup, and to continually call the packet-receiving state machine (which is practically identical to that shown in 30). This means that any CPU time not dedicated to servicing interrupts (i.e. keeping the robot running) is to be used for checking and decoding packets. The two main processing functions described here are the 'UART' module, and the 'PID' module.

The UART will process a single byte, placing it into a receive circular buffer (or transmit circular buffer if robot sends a packet). When that byte constitutes an end of a packet, the packet is parsed, interpreted, and the data acted upon. Once complete it releases control, and starts again when the next byte is received.

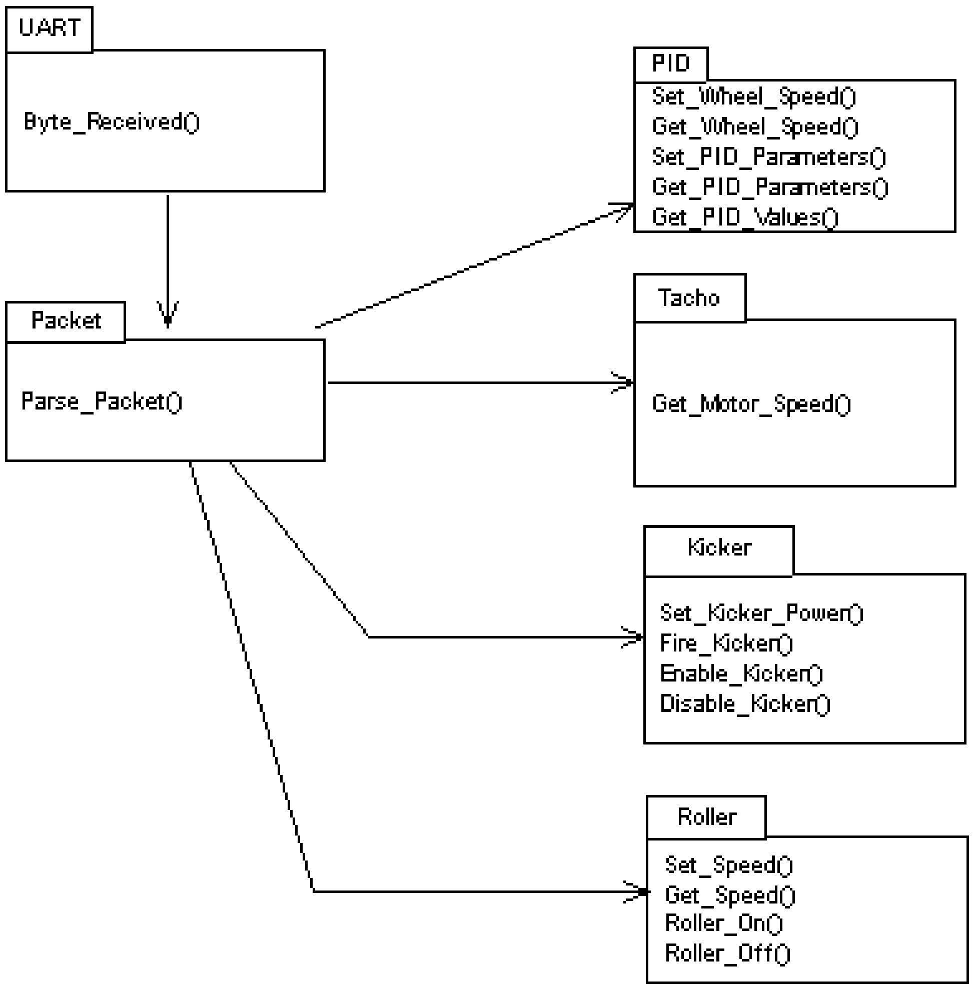


Figure 32: UART Module - UML Class Diagram

The PID closed-loop feedback controller executes approximately 140 times per second (i.e. a sampling rate of 140 Hz) Its job is to maintain wheel velocities by reading the current and previous velocities, applying the 'PID tuning' variables, and setting wheel motor voltages determined to be appropriate for obtaining the desired velocities.

8.2.3 Module Layout

The following lists explain all important module functions.

`PWM.c`

- `pwm_init()` - Initialises the PWM module

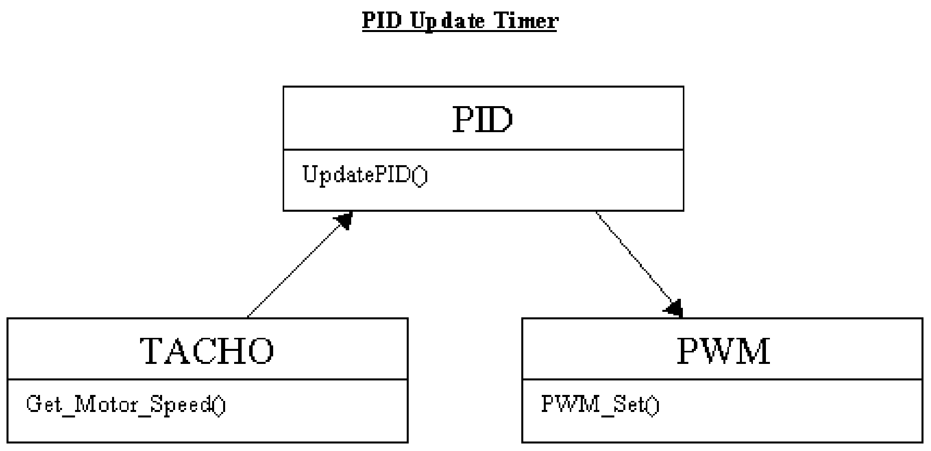


Figure 33: PID Closed Loop Feedback Controller - UML Class Diagram

- `pwm_set()` - Sets the duty cycle of a particular output

`TACHO.c`

- `tacho_init()` - Initialises the tacho module
- `tacho_getWheelSpeed()` - Retrieves current speed of a particular wheel
- `tacho_getWheelDirection()` - Retrieves current direction of a particular wheel

`PID.c`

- `pid_init()` - Initialises the PID module
- `pid_enable()` - Enables the PID
- `pid_disable()` - Disables the PID
- `pid_getP()` - Returns P parameter
- `pid_getI()` - Returns I parameter
- `pid_getD()` - Returns D parameter
- `pid_setParams()` - Sets P, I and D parameters
- `pid_setDesiredSpeed()` - Sets desired speed of a particular wheel

- `pid_getActualSpeed()` - Returns actual speed of a particular wheel
- `pid_getOutput()` - Returns output duty cycle of a particular wheel

`UART.c`

- `UART_init()` - Initialises the UART module
- `UART_SendByte()` - Sends a single character
- `UART_RecieveByte()` - Receives a single character
- `UART_RecievedByte()` - Determine if a character is available

`KICKER.c`

- `Kicker_init()` - Initialises the Kicker module
- `Kicker_power_set()` - Sets kicker fire power
- `Kicker_fire()` - Starts the firing sequence to kick
- `Kicker_enable()` - Turns the DC-DC converter on, which gets ready for a kick
- `Kicker_disable()` - Disables the DC-DC converter

`PACKET.c`

- `Parse_Packet()` - Retrieves a UART byte, interprets packet and acts on data

9 Bibliography

F180 Organising Committee Laws of the F180 League 2004 - Release v3.00a, http://www.itee.uq.edu.au/~wyeth/F180%20Rules/index.htm, 2004

FEHILY, C. Visual Quickstart Guide: Python, Berkely, Peachpit Press, 2002

AHO, A and SETHI R and ULLMAN J. Compilers: Principles, Techniques and Tools, Addison Wesley, 1988

WALLIN, D and NORBERG A. Luabind public beta documentation, http://luabind.sourceforge.net/docs.html, 2003

IERUSALIMSCHY, R et al. Lua 5.0 Reference Manual, http://www.lua.org/, 2003

BRUCE, J. Realtime Machine Vision Perception and Prediction, Thesis, Carnegie Mellon University

BRUCE, J. et al. Fast and Inexpensive Color Image Segmentation for Interactive Robots. Carnegie Mellon University, CORAL Research Group, 2000

BRUCE, J. and VELOSO M. Fast and Accurate Vision-Based Pattern Detection and Identification. Carnegie Mellon University, CORAL Research Group, 2003

VON HUNDELSHAUSEN F. and ROJAS R. Tracking Regions Free University of Berlin, Institute of Computer Science.

KARPATI T. et al. Detail Vision Documentation: Real-time Visual Perception for Autonomous Robotic Soccer Agents, http://RoboCup.mae.cornell.edu/documentation.php, 1999

LOOMIS J. et al. Performance Development of A Real-Time Vision System, http://RoboCup.mae.cornell.edu/documentation.php, 2003

NOLAN, C. Enhancements to a Global Vision System for F180 League Robotic Soccer, Thesis (B.Eng), The University of Queensland, 2002

JACK K. Video Demystified 3rd Edition - Chapter 3: Color Spaces http://www.video-demystified.com/mm/tutor/ch03.pdf

ISO/IEC 14882:1998(E) International Standard: Programming Languages - C++, 1998

C++ Standards committee, C++ Standard - unofficial list of revisions, 2003

SUTTER H., Exceptional C++, 1999

SUTTER H., More Exceptional C++, 2002

ALEXANDRESCU A., Modern C++ Design, 2001

JOSUTTIS N., The C++ Standard Library, 1999

STROSSTRUP B., The C++ Programming Language (Special Edition), 2000

MEYERS S., Effective C++ & More Effective C++ (CD Edition), 1996-1998

MUELLER J., Visual C++ .NET Developers Guide, Osborne, 2002

KRUGLINSKI D. et al., Programming Microsoft Visual C++ (Fifth Edition), Microsoft Press, 1998

MICKEY W., Sams Teach Yourself Visual C++ 6, Sams, 1998

EMBREE P., C++ Algorithm for Digital Signal Processing (Second Edition), Prentice Hall, 1999

COHEN A. and WOODRING M., Win32 Multithreaded Programming, O'Reilly, 1998

Microsoft Press, Desktop Applications with Microsoft Visual Studio C++ 6.0 - MSCD Training Kit for Exam 70-016, Microsoft Press, 1998

DUNLOP R. et al., Sams Teach Yourself DirectX 7, Sams, 2000

DEITEL & DEITEL, C++ How to Program, Prentice Hall, 1997

PRATA S., C Primer Plus (Third Edition), The Waite Group, 1999

OVERLAND B., C++ In Plain English (Second Edition), M&T Books, 1999

TEWS A. & WYETH G., University of Queensland, http://www.itee.uq.edu.au/~wyeth/publications/maps.pdf, 1998

Anderson, G et al., Cornell RoboCup Documentation Mechanical Group Final Documentation, http://RoboCup.mae.cornell.edu/documentation/RoboCup/2003/2003ME.pdf, 2003

Cornell electrical paper http://RoboCup.mae.cornell.edu/documentation/RoboCup/2003/2003EE.pdf

HALL B., Beej's Guide to Network Programming, Revision 2.3.1, http://www.ecst.csuchico.edu/~beej/guide/net/bgnet.pdf, 2001

KAPETANAKIS G., York RoboCup Vision Subsystem manual, Department of Computer Science University of York, http://www-users.cs.york.ac.uk/~georgios/vision/doc/visionmanual.pdf, 2003

Unknown author, Line Drawing Algorithms, http://www.cs.unc.edu/~mcmillan/comp136/Lecture6/Lines.html, 1996

CARTER, B et al., Mechanical Design and Modeling of an Omni-directional RoboCup Player, http://zen.ece.ohiou.edu/~RoboCup/papers/mech/37.pdf, 2001

ASADA M. et al., RoboCup-98: Robot Soccer World II, Springer Verlag, 1999

VELOSO M. et al., RoboCup-99: Robot Soccer World III, Springer Verlag, 2000

STONE P. et al., RoboCup-2000: Robot Soccer World IV, Springer Verlag, 2001

BIRK A. et al., RoboCup-2001: Robot Soccer World V, Springer Verlag, 2002

KAMINKA G. et al., RoboCup-2002: Robot Soccer World VI, Springer Verlag, 2003

CALVERT D., Software Architectural Styles, http://hebb.cis.uoguelph.ca/ dave/27320/new/architec.html

BONO C., University of Southern California — CSCI 201 Software Development — Intro To MFC, http://www-scf.usc.edu/~csci201/notes/MFCintroM.pdf

GAMMA E., HELM R. et al., Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995

DAVIES E. R., Machine Vision: Theory Algorithms Practice (2nd Edition), Academic Press, 1997