Duckburg
Users Guide

Jan Neuburger
jan.neuburger.jn@gmail.com

Erik Mayrhofer
erik.mayrhofer@liwest.at

Florian Schwarcz
florian.schwarcz@gmx.at

Maximilian Wahl
mexx.wale@gmail.com

December 10, 2018

Contents

1 Creating an Engine inside the Project

Engines are either shipped in individual .so files or as a group in one .so file. In general it is better to ship one Engine in one .so file, because this allows us to have more control over the final code when starting the robot.

1.1 Background knowledge

Duckburg itself does not contain any strategic, tactical or any other game relevant logic whatsoever. This is because we want to be able to quickly switch between scenarios like playing in the robocup and then presenting the Robot at a fair. In our current setup we only have to change the settings in one single configuration file, which specifies which components and engines will be used at startup. This implies multiple things:

- One engine does not know anything which engines are loaded.
- All engines have to communicate over a central message bus.
- One engine cannot have direct calls to other engines nor can include them.

1.2 How to: "Create an engine"

Let‘s assume we want to create a Communication-Engine. It‘s job is to listen for a TCP-Connection and transmit or receive information Let‘s use the following requirements:

- Receive Commands "Move forward", "Stop", "Sit", "Standup"
- Listen for Requests "Current Position", "Estimated Ball Position" and answer them
- Send Status information if the robot tips, starts to move or stops moving

1.2.1 Creating the class-file

Create files First we create two files `ExampleEngine.cpp` and `ExampleEngine.h`.

Register files in CMake Our compiler somehow has to know which files belong to which engine. Therefore we open our `CMakeLists.txt`. CLion automatically adds our files to a target of it's choice, most of the time this will be `Duckburg`. We search for our two files and delete them.

```
add_library(Duckburg SHARED
    core/Brain.cpp
    core/Brain.h
    ...
    core/Intent.h
    core/StandardIntents.h
    hightools/TimeUtil.h
    engines/ExampleEngine.cpp #Remove
    engines/ExampleEngine.h #Remove
    )
```

Figure 1: `CMakeLists.txt`

Engines **must** always be in a seperate target. Multiple Engines may be combined in one target, but this is discouraged and must have a good reason. Lets create a new target.

```
add_library(ExampleEngine SHARED
    engines/ExampleEngine.cpp
    engines/ExampleEngine.h)
target_link_libraries(ExampleEngine Duckburg)
```

Figure 2: `CMakeLists.txt`

1.2.2 Implementing the Engines

Now we have registered our class and it can be compiled. It's time that we take a close look at the class itself. Right now our two files are empty, or contain just some boilerplate code. Lets write the following things into them.

```cpp
#ifndef DUCKBURG_EXAMPLEENGINE_H
#define DUCKBURG_EXAMPLEENGINE_H

#include "../core/EngineBase.h"

class ExampleEngine : public EngineBase {
public:
    /***
    Returns the runtime-name of the engine.
    This can be whatever you want,
    but should be the same as the name of the class.
    */
    const char* getName() override;

    /***
    This is the first lifecycle-hook of all Engines.
    In onLoad() you should subscribe to all needed
    Intents and not do anything besides that.
    Setup-Code should be in the Constructor or in a
    Dispatcher of the "brain-run" method.
    */
    void onLoad() override;

    /***
    This method should always return
    the macro "ENGINE_BASE_VERSION".
    This is used to detect old Engines which need
    to be recompiled to run properly.
    */
    unsigned int getEngineBaseVersion() override;


    /***
    This is one of our Intenthandlers.
    */
    void doThing(void* data);
};
#endif //DUCKBURG_EXAMPLEENGINE_H
```

Figure 3: `ExampleEngine.h`

```cpp
#include <boost/signals2/signal.hpp>
#include <iostream>
#include "ExampleEngine.h"
#include "../hightools/Dyllo.h"
#include "../core/Brain.h"
#include "../core/Logger.h"

//All Log-calls after this Tag will be
//tagged accordingly.
//You can have more than one Log-Tag per File,
//but the name should be descriptive.
#define LOG_TAG LOGGER(ExampleEngine)

void ExampleEngine::onLoad() {

    //We want to register our doThing()
    //method to the tick-Intent.
    Brain::getInstance()->addDispatcher(
        INTENTID_ON_BRAIN_TICK,
        BINDTHIS(ExampleEngine::doThing)
    );
}

//The UNUSED macro is used to supress compiler
//warnings regarding unused parameters
void ExampleEngine::doThing(UNUSED void* data) {
    //LINFO writes an info-message to stdout
    LINFO("Thing is done");
}

const char *ExampleEngine::getName() {
    return "ExampleEngine";
}

unsigned int ExampleEngine::getEngineBaseVersion() {
    return ENGINE_BASE_VERSION;
}

//!!!!ATTENTION!!!!
//This line is crucial. If this is missing,
//our Engine cannot be loaded at runtime.
DL_CLASS_PUBLISH(ExampleEngine)
```

Figure 4: `ExampleEngine.cpp`

1.2.3 Start Duckburg and load our Engine

We now need to compile our Engine. If we use the shipped scripts, everything is compiled, including our Engine. We can verify that our Engine is present by looking in the `build` directory and checking if `libExampleEngine.so` exists. We will now tell Duckburg to load our Engine on startup, via the `conf.xml`.

```
<ENGINES>
    <ENGINE>
        <FILENAME>libExampleEngine.so</FILENAME>
        <NAME>ExampleEngine</NAME>
    </ENGINE>
</ENGINES>
```

Figure 5: `conf.xml`

Now everything is setup. We can now compile our engine and start duckburg using `upload.sh buildrun`

1.3 How to: "Subscribe to Intents"

```
    Brain::getInstance()->addDispatcher(...,
        BINDTHIS(...), SlotOrder::EXEC);
    Brain::getInstance()->addDispatcher(...,
        BINDTHIS(...), SlotOrder::PRE);
    Brain::getInstance()->addDispatcher(...,
        BINDTHIS(...), SlotOrder::POST);
```

Figure 6: `Advanced Intents`

Intent-Ids The first Argument to addDispatcher is the desired INTENT-ID. All INTENT-IDs are strings, which are defined using macros in a Header-File of an Engine. If you want to issue a new Intent, please define the INTENT-ID in the Header-File of your engine.

1.4 How to: ”Master the Logging-Framework”

The Logger is defined in `core/Logger.h`. To use the logger this file hast to be included and a log-tag has to be defined.

```
1 #include "../core/Logger.h"
2 #define LOG_TAG LOGGER(LoggingEngine)
```

Figure 7: `Use the Logger`

The log-tag indicates who will now log messages. You can use more than one log-tag in one file and then every log-call will use the last defined log-tag.

log-levels indicate how severe and important the following current message is. We decided to use the same logging-levels as journalctl so that we can maybe once stream our logs to the official linux-logging framework.

- LEVEL_EMERGENCY 0
- LEVEL_ALERT 1
- LEVEL_CRIT 2
- LEVEL_ERR 3
- LEVEL_WARN 4
- LEVEL_NOTICE 5
- LEVEL_INFO 6
- LEVEL_DEBUG 7

log-macros allow you to create logs. There are several:

```
1   LDEBUG("abc");
2   LINFO("abc");
3   LNOTICE("abc");
4   LWARN("abc");
5   LERR("abc");
6   LALERT("abc");
7   LCRIT("abc");
8   LEMERGENCY("abc");
9   int a = 3;
10  LFDEBUG("abc: " << a << " test");
11  LFINFO("abc" << a << " test");
12  LFNOTICE("abc" << a << " test");
13  LFWARN("abc" << a << " test");
14  LFERR("abc" << a << " test");
15  LFALERT("abc" << a << " test");
16  LFCRIT("abc" << a << " test");
17  LFEMERGENCY("abc" << a << " test");
```

Figure 8: `Log-Macros`

The `LF[LEVEL](msg)` calls are slower than the normal `L[LEVEL](msg)` calls but allow you to output variables and similar things.

The DEBUG_MODE macro will be defined if the log-level is DEBUG. This can be used to in #ifdefs around more complicated debug-code

1.4.1 Configure the Logging-Framework

You can specify several variables at compile time:

LOG_LEVEL If specified, all Logs that are more verbose than LOG_LEVEL will not be compiled into our library. We won't want DEBUG-Outputs in our library when we compile it for use at a fair or similar important events. These calls are completely dropped and will not slow down our code. If LOG_LEVEL is not specified **it defaults to INFO. Therefore no debug calls will be outputted to the console.**

LOG_LEVEL_CERR This indicates which logs will be outputted to cerr instead of cinfo. This defaults to `LEVEL_WARN`.