CodeSubWars v0.4.7b
Manual

written by Andreas Rose
2022-08-07

Contents

1 What is CodeSubWars? 2
2 Installation 2
3 Running the application 2
3.1 Application with graphical user interface 2
3.1.1 Quickstart 2
3.1.2 The user interface 2
3.1.3 New World 5
3.1.4 New Battle 5
3.1.5 Stop Battle 5
3.1.6 Replay 5
3.1.7 Options 6
3.2 Application with text output 6
4 Installing a new submarine 6
5 Creating a new submarine 6
5.1 Submarine construction 7
6 Background 9
6.1 The World 9
6.1.1 Object properties 9
6.1.2 Events 10
6.1.3 Commands 10

1 What is CodeSubWars?

CodeSubWars is a programming game environment, similar to robocode, roboships, etc. It bases on simple physics simulation in three dimensions.

The simulated world is assumed to be under water. There exists autonomous submarines which can fight against each other. The behavior of the submarines must be implemented in the programming language python. Its up to you if they are smart enough to survive. **The goal is to create a submarine that is alive at the end.**

Additionaly there can be objects with different properties: some large range force emitter (something like a black hole in space), magnetic objects (e.g. mines that moves automatically to the nearest object), passive rocks that moves around (something like asteroids in space), static objects (walls), etc.

2 Installation

For installation first get and install python 3.10 (available at http://www.python.org). Then extract the archive into a folder, e.g. `c:\program files\CodeSubWars`. That's it. You should now be able to start the application, e.g. located in `c:\program files\CodeSubWars\CodeSubWars.exe`.

3 Running the application

There are two modes available for running the application: with and without graphical output. The graphical application is designed to use for debug/study the behavior of your and other submarines. You can analyse submarines actions and reactions. This is fully passive. Your are only able to observe things that happen.

3.1 Application with graphical user interface

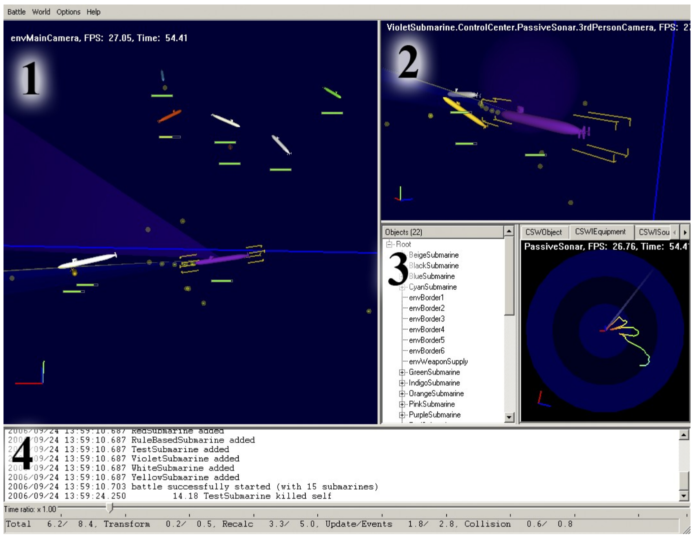
3.1.1 Quickstart

For quickly start a battle: start the application and choose Battle > New Battle. Select and add all submarines for new battle in the upcoming dialog and click „Ok“. Now a battle runs.

3.1.2 The user interface

A command shell is created in the background for showing errors or log outputs reported by python code. The main window is created (Fig. 1).

The user interface of the main window is divided into four main views:



```
2006/09/24 13:59:10.687 RedSubmarine added
2006/09/24 13:59:10.687 RuleBasedSubmarine added
2006/09/24 13:59:10.687 TestSubmarine added
2006/09/24 13:59:10.687 VioletSubmarine added
2006/09/24 13:59:10.687 WhiteSubmarine added
2006/09/24 13:59:10.687 YellowSubmarine added
2006/09/24 13:59:10.703 battle successfully started (with 15 submarines)
2006/09/24 13:59:24.250        14.18 TestSubmarine killed self
```

```
Total   6.2/  8.4, Transform   0.2/  0.5, Recalc   3.3/  5.0, Update/Events   1.8/  2.8, Collision   0.6/  0.8
```

Fig. 1: The applications user interface while running a battle mainly constists of the main view from top (1), a detail view of the selected object (2), a detailed more technically view to the properties of the selected object (3) and the log output (4)

1. The main camera view is initially watching from the top of the world. Here you can get an overview of what is happen globally (Fig. 2).

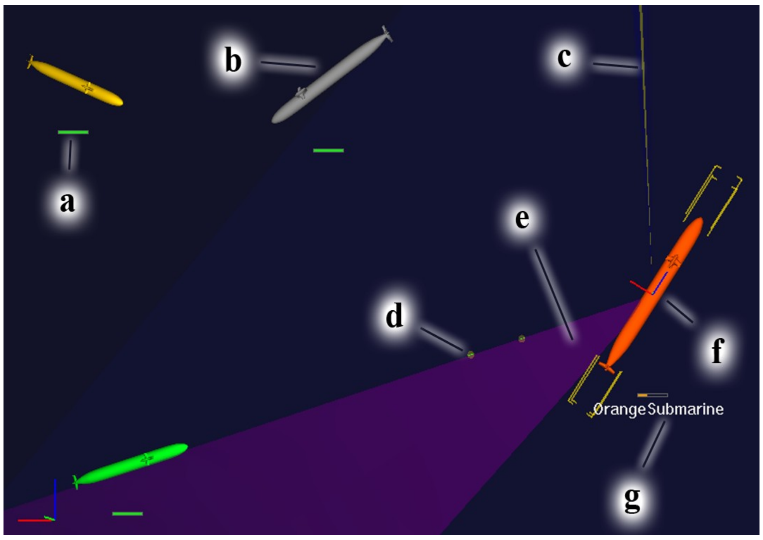
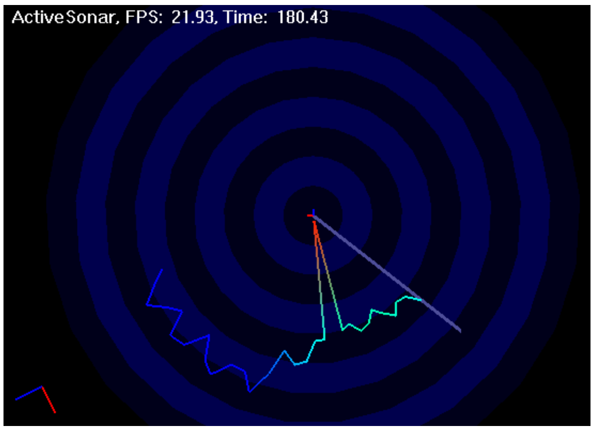
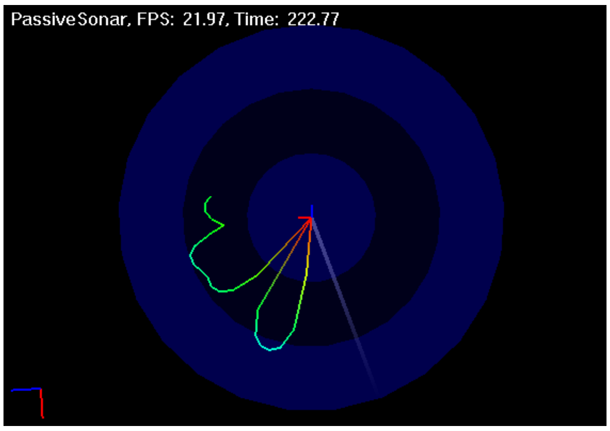


Fig. 2:A detailed region of a scene view is shown. Here four submarines and two torpedos are displayed. a) Health; b) Submarine; c) Visualization of active sonar; d) Torpedo; e) Visualization of passive sonar; f) Selected object; g) Name of the selected object

2. The 3rd person view of the currently selected object shows the local environment of the selected object.
3. The tree of existing objects in the world with the highlighted selected object is shown here. Some useful information regarding to the object properties are displayed (e.g. what command is currently processed, which objects are currently detected (Fig. 4 and Fig. 3), the current position and velocity, the state of equipment, etc.).



a)



b)

Fig. 3: a) A visualization of an active sonar is displayed. Here the current detected distance in meter is drawn over the current direction. The blue background cicles have the width of 500 m. b) A visualization of an passive sonar is displayed. Here the current detected sound pressure level (SPL) in decibel is drawn over the current direction. The blue background cicles have the width of 50 dB.

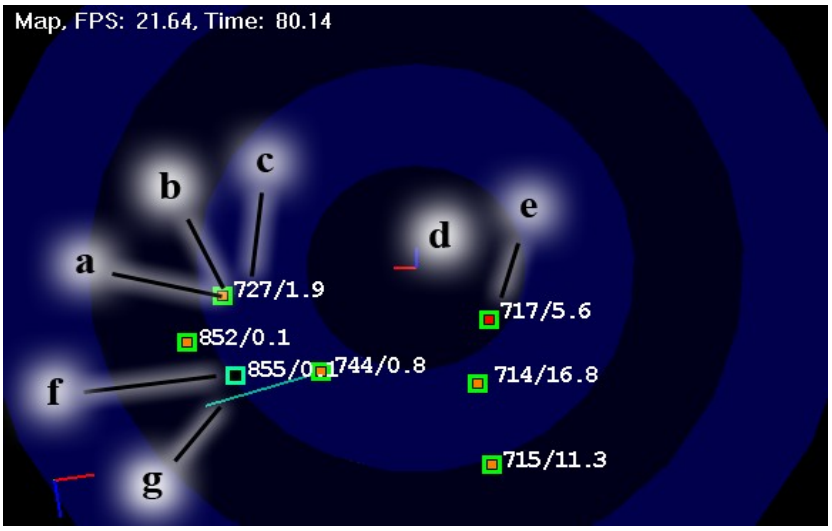


Fig. 4: A visualization of a map is displayed. The blue background cicles have the width of 500 m. Here different objects are inserted in this map: a) The dangerlevel of this object; b) The visualized userdefined data; c) The age of detection of the object in seconds;d) The origin of the submarine itself; e) Currently detected weapon (red inner square) which is about 480 m away; f) Visualized object inserted by the user (here: weapon supply); g) The approximated velocity with value and direction of this object

4. The log ouput shows important messages and current events. On finalization the statistics of the battle are shown.

The possible interactions are:

- On the graphical views you can move, zoom or rotate by clicking the left, middle or right mouse button in combination with moving or by using the mouse wheel.
- To select an object just click on it. Selected objects are displayed with its full name in the three dimensional views.

- Change the running speed by moving the slider at the bottom. This will cause the simulation to go faster or slower. If the running speed is set faster than real time, the collision simulation accuracy decreases. So the simulation results may be different from running in real time. Its not recommend to increase the speed higher than four times to real time. The second problem with high simulation speed is that the rotation and moving controller in submarines may become to flip-floping and dont reach the end condition.

3.1.3 New World

Here you can create a predefined world. By doing that the currently existing one including all objects are removed and the new selected is created. There will only be exist the world base elements. That means no user-defined submarine is placed. All selectable worlds are bordered by a cube and have an edge length of 5000 m. The possible worlds are:

- *Default 1:* This world is empty. Initially this world is constructed by default.
- *Default 2:* This world contains 20 randomly placed large rocks drawn as grey cubes.
- *Default 3:* This world contains a weapon supply initially placed at position $(0, 0, 0)^T$ [m] drawn as a green cube.
- *Default 4:* This world contains a not drawn „black hole“ placed at position $(0, 0, 0)^T$ [m].
- *Default 5:* This world contains 5 randomly placed active rocks drawn as light blue cubes.

3.1.4 New Battle

When starting a new battle all valid submarines in the sub folder `/submarine` are displayed in a dialog. Here the participating submarines and some other options for the new battle can be choosen. The selected submarines are constructed and placed at random positions in the world. The placing can be modified in the preferences: in one plane or everywhere.

Battles can be started in two modes:

- *Single:* Each submarine fight for its own. Its on the submarine how it handles enemy detection. Some can attack everything and others may do some pacts with other ones. The only thing that really counts is to be alive at last!
- *Team:* Each submarine is created more than once. The number of constructions can be selected. So a team is created. If a submarine is not designed to fight in teams, e.g. because it attacks everything it detects, the team mode makes no really sense for that. For this mode teammate submarines should communicate with each other to locate their enemies and concentrate their fire power. Teammates could negotiate its role in his team. So specializations of the same submarine at runtime are possible. And keep in mind: the goal is to keep the team alive as long as possible.

3.1.5 Stop Battle

This will stops a currently running battle and displays the battle statistics in the log ouput.

3.1.6 Replay

Here a CodesubwarsBattleRecord file (*.cbr) must be choosen. After loading, a previously recorded battle can be viewed (Fig. 5).

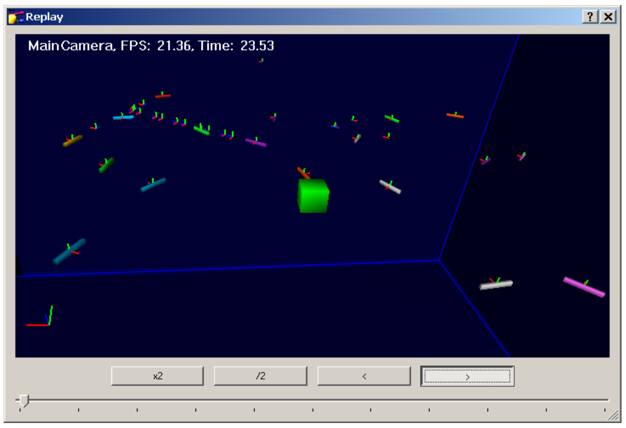


Fig. 5: The viewer that displays recorded battles.

3.1.7 Options

Here you can set preferences influencing what is drawn, how the battle starts, etc. By default submarines are initialized in a plane when starting a new battle.

3.2 Application with text output

In this mode only the log output is displayed. This is usable for faster simulations. The simulation runs with fixed time step (not in real time). Therefor all available processor power is used for simulation.

The usage is:

```
CodeSubWars [-silent] [-world=<1|2|3|4>] [-battle=<single|team>] [-teamsize=<3|5|10>] [-timestep=x]
```

`-silent`	Using this parameter makes the application starts without graphical output. The other parameters are only valid in silent mode.
`-world`	With this parameter the predefined world environment can be choosen. Default is 1.
`-battle`	This parameter indicates which battle mode should be run. Default is single.
`-teamsize`	This parameter defines the number of team member in team mode. Default is 3.
`-timestep`	Here the time step in seconds for each simulation step can be set. The given value must be in range [0.01, 0.1]. Default is 0.01.

4 Installing a new submarine

Installing a new submarine is really simple. Just copy the necessary files of the new submarine (the py-file and maybe some modules) into the subfolder `/submarines`.

5 Creating a new submarine

Create a new autonomous submarine means: arange a new behavior. This is done by collecting informations provided by the given sensors and events and control the actuators and the weapons. The programing language for doing that is the very nice and powerful python.

You have to create a py-file that is named as the submarine class name (Fig. 6). Then you have to inherit from the base class `CSWPySubmarine`. In the constructor you must set the name of the submarine and its length in range [30, 110] m (line 2-5). The name should be equal to the class name. Longer submarines are more powerful but slower in rotation manuvers. Long submarines are also better detected by sonar.

Additionally you have to define three methods: `initialize`, `update` and `processEvent`. These methods are called by the framework. The detailed meaning is as follows:

- `initialize` is called directly before the battle starts (line 7-8). This should be used for initializations (e.g. set color, store initial position). Setting the color (red, green, blue, alpha) is not neseccary, but submarines are better to differentiate. The given alpha value is ignored and is always set to 1. The default color is grey.
- `update` is called periodically about ervery 10 ms (line 10-11). Here you can execute code you want, e.g. checking sensors, change engine intensities.
- `processEvent` is called from the framework if an event has been arrived (line 13-14). Here you can check which type of message the event contains and apply some actions. Objects moved outside the world borders (which could not happen normally) will receive an event containing a system message as warning. If the object is more than 90 s outside the world, it will be kicked from the current battle.
- `finalize` (optional) is called directly before destruction. Here some finalizations can be done (e.g. release resources, removing references to itself).

But be aware to exceed excecution time of 6 ms in average in `update` and `processEvent`. If you do this your submarine will be kicked from the current battle.

```
(1)    class OrangeSubmarine(CSWPySubmarine):
(2)        def __init__(self):
(3)            CSWPySubmarine.__init__(self, \
(4)                                    self.__class__.__name__, \
(5)                                    110)
(6)
(7)        def initialize(self):
(8)            self.setColor(Vector4(1.0, 0.3, 0.0, 1.0))
(9)
(10)       def update(self):
(11)           pass
(12)
(13)       def processEvent(self, event):
(14)           return 1
```

Fig. 6: Example of a minimal submarine implementation that does nothing.

Watch out the html-documentation to get detail information about which functionality is available and how to use it. There exists some sample submarines in `/submarines`. These submarines may show some interesting implementations.

5.1 Submarine construction

All submarines are equiped with (see also Fig. 7):

- *Actuators:*
  - *MainEngine:* This engine is mounted at the back and is directed toward the front. Its function let the submarin move forward or backward.
  - *BuoyancyTank:* This "engine" is mounted in the center and is directed up. Its function let the submarin move up or down.

  - *BowsJetOar:* This engine is mounted at the front and is directed to the right. Its function let the submarin rotate leftside or rightside.
  - *InclinationJetOar:* This engine is mounted at the front and is directed up. Its function let the submarin rotate upward or downward.
  - *AxialInclinationJetOar:* This engine is mounted at the middle top and is directed to the right. Its function let the submarin rotate about his main axis.
- *Sensors:*
  - *ActiveSonar:* This sonar is mounted in the center. It moving behavior can be defined. By default it rotates horizontally about 360 degree.
  - *PassiveSonar:* This sonar is mounted in the center. It moving behavior can be defined. By default it rotates horizontally about 360 degree.
  - *GPS:* This global positioning system is mounted in the center.
  - *GyroCompass:* This compass is mounted in the center.
  - *MovingPropertiesSensor:* This sensor is mounted in the center.
- *Weapons:*
  - *FrontLeftWeaponBattery:* This weapon battery is mounted at the front left. It contains green torpedos (the main attack weapon).
  - *FrontRightWeaponBattery:* This weapon battery is mounted at the front right. It contains red (active sonar guided) and blue (passive sonar guided) torpedos.
  - *BackWeaponBattery:* This weapon battery is mounted at the back. It contains green and yellow mines.
- *Other:*
  - *Transceiver:* This Transceiver is mounted in the center.
  - *Map:* This map is mounted in the center. It functionality is to collect data from the sensors and provide useful accessing functions.

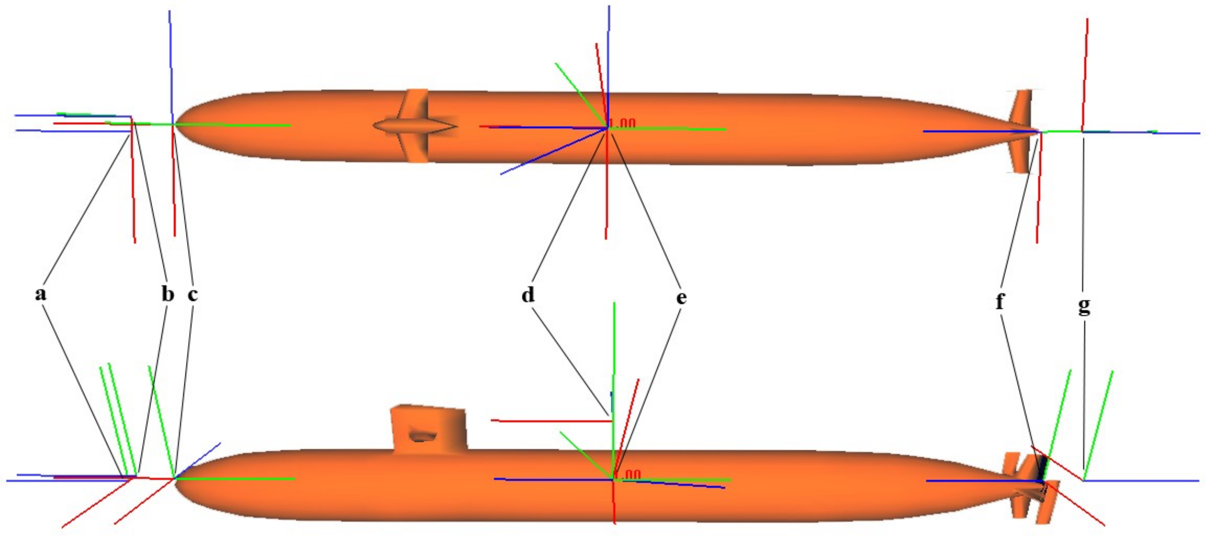


Fig. 7: Equipment of a submarine: a) FrontLeftWeaponBattery; b) FrontRightWeaponBattery; c) BowsJetOar / InclinationJetOar; d) AxialInclinationJetOar; e) BuoyancyTank and ControlCenter with ActiveSonar, PassiveSonar, GPS, Compass, Transceiver, MovingPropertiesSensor and Map; f) MainEngine; g) BackWeaponBattery

6 Background

6.1 The World

The world is a cube with edge length of 5000 m. Many objects which can have different combinations of properties can exist in that world. The world bases on simple physics. There exists masspoints, forces, impulses and friction. By manipulating these magnitudes objects can move around in the world.

6.1.1 Object properties

Objects can have different properties in the world:

- *Collideable:* Objets with this property can collide with other objects having this property.
- *Commandable:* Objects with this property are able to manage command executions.
- *Damageable:* Objects with this property can be damaged by collisions or by weapons. If the health is below zero the object is dead and will be removed from the world.
- *Dynamic:* Objects with this property are able to move/or rotate in the world. That means not that they have a machine and can move forward. This property only allows an object a possibility to move/rotate. If an object has no equipment for moving but this property it can be moved/rotated by global forces, by collisions with other objects, etc. Objects that have not this property are also called as "static".
- *EventDealable:* Objects with this property are able to handle events. These objects can have a method `processEvent()`. This method is called if an event has arived. In this method any code can be executed.
- *ForceEmitter:* Objects with this property are able to emit forces to other dynamic objects.
- *ImpulsEmitter:* Objects with this property are able to emit impulses to other dynamic objects.
- *Pythonable:* Objects with this property are able to be specialized in python.
- *Rechargeable:* Objects with this property are able to be recharged by a resource provider. For recharge at a resource provider, objects must be request the recharging. If the object is currently recharging a new request of recharge to the same provider will fail until the recharging ends.
- *ResourceProvider:* Objects with this property are able to provide resources to rechargeable objects.
- *Solid:* Objects with this property are visible. These will be drawn in the views.
- *SoundEmitter:* Objects with this property are able to emit sounds to sound receiver.
- *SoundReceiver:* Objects with this property receive sounds emitted by sound emitter.
- *Updateable:* Objects with this property have a method `update()`. In this method any code can be executed. Update is called about every 10 ms.

Examples for combination of these properties are:

- Submarine, which have properties Collideable, Commandable, Damageable, Dynamic, EventDealable, Impulsemitter, Pythonable, Solid, SoundEmitter and Updateable.
- Wall, which have properties Collideable and Solid.
- Rock (or asteroid), which have properties Collideable, Damageable, Dynamic and Solid.

- WeaponSupply, which have properties Collideable, Dynamic, Solid, ResourceProvider and Updateable.

6.1.2 Events

Events are delivered to objects that can handle events (EventDealable). An event consists of a sender, a receiver, a message and the time to life. There exists no specializations of events. Events are the container to transport information from one object to another. The information is the message. There exists specializations of messages:

- *CollisionMessage:* This message indicates a collision. It provides some detail information about the collision (e.g. the collision point and applied damage).
- *ExplosionMessage:* This message indicates an explosion. It provides some detail information about the explosion (e.g. the explosion point and applied damage).
- *TextMessage:* This message indicates an incoming text message. It contains the text.
- *TransceiverMessage:* This indicates an incoming message from a transceiver. It contains the text that was sent.
- *SystemMessage:* This message indicates a hint from the framework. Currently this is only the warning message if objects are outside the world.

6.1.3 Commands

Commands are used to control things on an object. Commands are executed one after another. If a command should be executed when another command is currently executed, the command is stored and will be executed immediately after the previous. So you can create and execute command stacks consisting of many commands (see example YellowSubmarine).

Commands can be also MacroCommands. Those macros can consist of commands and other macro commands. You can define some special macros that combine often used commands (see example BlueSubmarine, GreenSubmarine, CyanSubmarine).

The following predefined commands are available:

- Lowlevel commands
  - *PushCommand:* This command makes the command processor store the current context onto the top of a stack.
  - *PopCommand:* This command makes the command processor restore the topmost context of a stack to the current context.
  - *RepeatCommand:* This command makes the command processor repeat the previously executed commands of the current context.
  - *CleanupHistoryCommand:* This command makes the command processor clean up the container containing previously executed commands.
- Highlevel commands:
  - *CSWFireCommand:* This command makes the submarine launch a weapon.
  - *CSWSendEventCommand:* This command makes the submarine send an event.
  - *CSWSetEngineIntensityCommand:* This command makes an engine change its intensity.
  - *CSWSetEngineDirectionCommand:* This command makes an engine change its direction.
  - *CSWRechargeWeaponBatteryCommand:* This command makes a weapon battery request recharge to a weapon supply.

- *CSWWaitCommand:* This command makes the command execution wait a given time.
- *Predefined macro commands:* This is a collection of maybe often used combination of commands.

Create a new command

Combining available commands is good but if you want to create a completely new behavior you have to write your own command. This must be done by inheriting from the base class `CSWPyCommand`. In Fig. 8 a simple example is shown. This command does nothing. Other available examples are used by IndigoSubmarine.

You must define the following methods:

- `__init__`: The constructor must be able to handle copy constructions and ordinary constructions (see line 2-6). In the first case a construction from a given command of the same type must be done. In the second case an ordinary construction must be done.
- `initialize`: This method is called directly before the first time `step` is called. Here initializations should be done.
- `step`: This method is called periodically about every 10 ms. Here useful calculations should be done. With the method `setProgress` the current progress must be updated. If the command wishes to finish itself it must call `finished`. But before doing that the progress must be set to 1 (i.e. 100 percent) otherwise `wasExecuted` will never return true.
- `cleanup`: This method is called directly after finishing. Here the used resources (e.g. engines) should be released or reseted.
- `getName`: This method should return a useful command name (e.g. class name).
- `getDetails` (optional): This method should return some interesting information about the current command state.

The execution time of the methods `initialize`, `step` and `cleanup` is restricted. Do not exceed excecution time of 6 ms in average of each of these methods. If you do this the submarine that currently executes this command will be kicked from the current battle.

```
(1)    class NOPCommand(CSWPyCommand):
(2)        def __init__(self, param = None):
(3)            if isinstance(param, self.__class__):
(4)                CSWPyCommand.__init__(self, param)
(5)            else:
(6)                CSWPyCommand.__init__(self)
(7)
(8)        def initialize(self):
(9)            pass
(10)
(11)       def step(self):
(12)           self.setProgress(1)
(13)           self.finished()
(14)
(15)       def cleanup(self):
(16)           pass
(17)
(18)       def getName(self):
(19)           return self.__class__.__name__
```

Fig. 8: Example of a minimal command implementation that does nothing.