cs380su21-meta.sg

Grade

Reviewed on Thursday, 22 July 2021, 5:15 PM by Automatic grade
grade: 100.00 / 100.00

Assessment report [-]
[+]**Summary of tests**

Submitted on Thursday, 22 July 2021, 5:15 PM (Download)

functions.cpp

```
/*!*******************************************************************************
\file functions.cpp
\author Vadim Surov, Goh Wei Zhe
\par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
\par Course: CS380
\par Section: B
\par Programming Assignment 10
\date 07-22-2021
\brief
This file has declarations and definitions that are required for submission
********************************************************************************/
#include "functions.h"

namespace AI
{


} // end namespace
```

functions.h

```
/*!*****************************************************************************
\file functions.h
\author Vadim Surov, Goh Wei Zhe
\par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
\par Course: CS380
\par Section: B
\par Programming Assignment 10
\date 07-22-2021
\brief
This file has declarations and definitions that are required for submission
*******************************************************************************/
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <sstream>
#include <string>
#include <list>

#include "data.h"

#define UNUSED(x) (void)x;

namespace AI
{
    // Check the state of a task comparing it with given by parameter
    class CheckState : public Task
    {
        Task checktask;
        State checkstate;

    public:
        CheckState(Task checktask = {}, State checkstate = State::Success)
            : Task{ "CheckState" }, checktask{ checktask }
            , checkstate{ checkstate }{}

        /*!*************************************************************************
        \brief
        Function to check the stat of a task comparing it with given by
        parameter.

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        *************************************************************************/
        CheckState& operator()(Log* log = nullptr, std::string level = "")
        {
            if (this->checktask.getState() == this->checkstate)
                this->state = State::Success;
            else
                this->state = State::Failure;

            if (log)
                *log << level << "CheckState" << "(" << this->checktask.getId()
                << "," << STATES[this->state] << ")" << std::endl;

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

    class Selector : public Task
    {
        std::list<SMART> tasks;

    public:
        Selector(std::initializer_list<SMART> tasks = {})
            : Task{ "Selector" }, tasks{ tasks }{}

        /*!*************************************************************************
        \brief
        Selector composite function to return a success status code when one of
        its children runs successfully.

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        *************************************************************************/
        Selector& operator()(Log* log = nullptr, std::string level = "")
        {
            if (log)
                *log << level << "Selector()" << std::endl;

            this->state = State::Failure;

            for (auto& x : this->tasks)
            {
                this->state = x->operator()(log, level + "| ").getState();

                if (this->state != State::Failure)
                    break;
            }

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

    class Sequence : public Task
    {
        std::list<SMART> tasks;

    public:
        Sequence(std::initializer_list<SMART> tasks = {})
            : Task{ "Sequence" }, tasks{ tasks }{}

        /*!**********************************************************************
        \brief
        Sequence composite function to return a failure status code when one of
        its children fails.

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        ***********************************************************************/
        Sequence& operator()(Log* log = nullptr, std::string level = "")
        {
            if (log)
                *log << level << "Sequence()" << std::endl;

            this->state = State::Success;

            for (auto& x : this->tasks)
            {
                this->state = x->operator()(log, level + "| ").getState();

                if (this->state != State::Success)
                    break;
            }

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

    class RandomSelector : public Task
    {
        std::list<SMART> tasks;

    public:
        RandomSelector(std::initializer_list<SMART> tasks = {})
            : Task{ "RandomSelector" }, tasks{ tasks }{}

        /*!**********************************************************************
        \brief
        Random selector composite function tries a single child at random.

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        ***********************************************************************/
        RandomSelector& operator()(Log* log = nullptr, std::string level = "")
        {
            if (log)
                *log << level << "RandomSelector()" << std::endl;

            this->state = State::Failure;

            if (this->tasks.size())
            {
                int i = (rand() % this->tasks.size());

                int count = 0;

                for (auto& x : this->tasks)
                {
                    count++;

                    if(count == i)
                        this->state =
                        x->operator()(log, level + "| ").getState();
                }
            }

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

    // Decorators
    class Inverter : public Task
    {
        SMART task;

    public:
        Inverter(SMART task = {})
            : Task{ "Inverter" }, task{ task }{}

        /*!**********************************************************************
        \brief
        Inverter function to invert the value returned by a task.

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        *************************************************************************/
        Inverter& operator()(Log* log = nullptr, std::string level = "")
        {
            if (log)
                *log << level << "Inverter()" << std::endl;

            this->task->operator()(log, level + "| ");

            if (this->task->getState() == State::Success)
                this->state = State::Failure;
            else
                this->state = State::Success;

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

    class Succeeder : public Task
    {
        SMART task;

    public:
        Succeeder(SMART task = {})
            : Task{ "Succeeder" }, task{ task }{}

        /*!*************************************************************************
        \brief
        Succeeder function that always return success, irrespective of what the
        child node actually returned. It is useful in cases where you want to
        process a branch of a tree where a failure is expected or anticipated,
        but you don't want to abandon processing of a sequence that branch sits
        on

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        *************************************************************************/
        Succeeder& operator()(Log* log = nullptr, std::string level = "")
        {
            if (log)
                *log << level << "Succeeder()" << std::endl;

            this->state = State::Success;
            this->task->operator()(log, level + "| ");

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

    class Repeater : public Task
    {
        SMART task;
        int counter;

    public:
        Repeater(SMART task = {}, int counter = 0)
            : Task{ "Repeater" }, task{ task }, counter{ counter }{}

        /*!*************************************************************************
        \brief
        Repeater function that will reprocess its child node each time its child
        returns a result. It is often used at the very base of the tree to make
        the tree to run continuously. Repeaters may optionally run their
        children a set number of times before returning to their parent.

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        *************************************************************************/
        Repeater& operator()(Log* log = nullptr, std::string level = "")
        {
            if (log)
                *log << level << "Repeater(" << counter << ")" << std::endl;

            this->state = State::Success;

            while(this->task && ((this->counter--) > 0))
                this->task->operator()(log, level + "| ");

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;
                log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

    class Repeat_until_fail : public Task
    {
        SMART task;

    public:
        Repeat_until_fail(SMART task = {})
            : Task{ "Repeat_until_fail" }, task{ task }{}

        /*!***********************************************************************
        \brief
        Repeat_until_fail function. Like a repeaters, these decorators will
        continue to reprocess their child until a child finally returns a
        failure, at which point the repeater will return success to its parent.

        \param log
        An object pointer to Log class

        \param level
        A string to keep track of the level

        \return
        Returns this pointer
        ************************************************************************/
        Repeat_until_fail& operator()(Log* log = nullptr, std::string level ="")
        {
            if (log)
                *log << level << "Repeat_until_fail()" << std::endl;

            this->state = State::Success;

            while (this->task &&
                this->task->operator()(log, level + "| ").getState()
                == State::Success) {}

            if (log)
                *log << level << "L " << STATES[this->state] << std::endl;

            return *this;
        }
    };

} // end namespace

#endif
```

[VPL](#)

◄ Showcase: Battleship

Jump to...

Fuzzy Logic ►