

cs380su21-meta.sg

[Dashboard](#) / [My courses](#) / [cs380su21-meta.sg](#) / [12 July - 18 July](#) / [Assignment 10 \(Behavior Trees\)](#)

 [Description](#)


 [Submission](#)

 [Edit](#)

 Submission view

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

```
1   /*!*****  
2  \file functions.cpp  
3  \author Vadim Surov, Goh Wei Zhe  
4  \par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu  
5  \par Course: CS380  
6  \par Section: B  
7  \par Programming Assignment 10  
8  \date 07-22-2021  
9  \brief  
10 This file has declarations and definitions that are required for submission  
11 *****  
12 #include "functions.h"  
13  
14 namespace AI  
15  {  
16  
17  
18 } // end namespace
```

functions.h

```

1  /*!*****
2  \file functions.h
3  \author Vadim Surov, Goh Wei Zhe
4  \par DP email: vsurov@digipen.edu, weizhe.goh@digipen.edu
5  \par Course: CS380
6  \par Section: B
7  \par Programming Assignment 10
8  \date 07-22-2021
9  \brief
10 This file has declarations and definitions that are required for submission
11 *****/
12 #ifndef FUNCTIONS_H
13 #define FUNCTIONS_H
14
15 #include <sstream>
16 #include <string>
17 #include <list>
18
19 #include "data.h"
20
21 #define UNUSED(x) (void)x;
22
23 namespace AI
24 {
25     // Check the state of a task comparing it with given by parameter
26     class CheckState : public Task
27     {
28     public:
29         Task checktask;
30         State checkstate;
31
32         CheckState(Task checktask = {}, State checkstate = State::Success)
33             : Task{ "CheckState" }, checktask{ checktask }, checkstate{ checkstate }{}
34
35         /*!*****
36         \brief
37         Function to check the stat of a task comparing it with given by
38         parameter.
39
40         \param log
41         An object pointer to Log class
42
43         \param level
44         A string to keep track of the level
45
46         \return
47         Returns this pointer
48         *****/
49         CheckState& operator()(Log* log = nullptr, std::string level = "")
50         {
51             if (this->checktask.getState() == this->checkstate)
52                 this->state = State::Success;
53             else
54                 this->state = State::Failure;
55
56             if (log)
57                 *log << level << "CheckState" << "(" << this->checktask.getId()
58                 << "," << STATES[this->state] << ")" << std::endl;
59
60             if (log)
61                 *log << level << "L " << STATES[this->state] << std::endl;
62
63             return *this;
64         }
65     };
66
67     class Selector : public Task
68     {
69     public:
70         Selector(std::initializer_list<SMART> tasks = {})
71             : Task{ "Selector" }, tasks{ tasks }{}
72
73         /*!*****
74         \brief
75         Selector composite function to return a success status code when one of
76         its children runs successfully.
77
78         \param log
79         An object pointer to Log class
80
81         \param level
82         A string to keep track of the level
83
84         \return
85         Returns this pointer
86         *****/
87         Selector& operator()(Log* log = nullptr, std::string level = "")
88         {
89             if (log)
90                 *log << level << "Selector()" << std::endl;
91
92             this->state = State::Failure;
93
94             for (auto& x : this->tasks)
95             {
96                 this->state = x->operator()(log, level + "| ").getState();
97
98                 if (this->state != State::Failure)
99                     break;
100             }
101
102             if (log)
103                 *log << level << "L " << STATES[this->state] << std::endl;
104
105             return *this;
106         }
107     };
108 }

```

```

109     }
110 };
111
112 class Sequence : public Task
113 {
114     std::list<SMART> tasks;
115
116 public:
117     Sequence(std::initializer_list<SMART> tasks = {})
118         : Task{ "Sequence" }, tasks{ tasks }{}
119
120     /*!*****
121     \brief
122     Sequence composite function to return a failure status code when one of
123     its children fails.
124
125     \param log
126     An object pointer to Log class
127
128     \param level
129     A string to keep track of the level
130
131     \return
132     Returns this pointer
133     *****/
134     Sequence& operator()(Log* log = nullptr, std::string level = "")
135     {
136         if (log)
137             *log << level << "Sequence()" << std::endl;
138
139         this->state = State::Success;
140
141         for (auto& x : this->tasks)
142         {
143             this->state = x->operator()(log, level + "| ").getState();
144
145             if (this->state != State::Success)
146                 break;
147         }
148
149         if (log)
150             *log << level << "L " << STATES[this->state] << std::endl;
151
152         return *this;
153     }
154 };
155
156 class RandomSelector : public Task
157 {
158     std::list<SMART> tasks;
159
160 public:
161     RandomSelector(std::initializer_list<SMART> tasks = {})
162         : Task{ "RandomSelector" }, tasks{ tasks }{}
163
164     /*!*****
165     \brief
166     Random selector composite function tries a single child at random.
167
168     \param log
169     An object pointer to Log class
170
171     \param level
172     A string to keep track of the level
173
174     \return
175     Returns this pointer
176     *****/
177     RandomSelector& operator()(Log* log = nullptr, std::string level = "")
178     {
179         if (log)
180             *log << level << "RandomSelector()" << std::endl;
181
182         this->state = State::Failure;
183
184         if (this->tasks.size())
185         {
186             int i = (rand() % this->tasks.size());
187
188             int count = 0;
189
190             for (auto& x : this->tasks)
191             {
192                 count++;
193
194                 if(count == i)
195                     this->state =
196                         x->operator()(log, level + "| ").getState();
197             }
198         }
199
200         if (log)
201             *log << level << "L " << STATES[this->state] << std::endl;
202
203         return *this;
204     }
205 };
206
207 // Decorators
208 class Inverter : public Task
209 {
210     SMART task;
211
212 public:
213     Inverter(SMART task = {})
214         : Task{ "Inverter" }, task{ task }{}
215
216     /*!*****

```

```

217 \brief
218 Inverter function to invert the value returned by a task.
219
220 \param log
221 An object pointer to Log class
222
223 \param level
224 A string to keep track of the level
225
226 \return
227 Returns this pointer
228 *****/
229 Inverter& operator()(Log* log = nullptr, std::string level = "")
230 {
231     if (log)
232         *log << level << "Inverter()" << std::endl;
233
234     this->task->operator()(log, level + "| ");
235
236     if (this->task->getState() == State::Success)
237         this->state = State::Failure;
238     else
239         this->state = State::Success;
240
241     if (log)
242         *log << level << "L " << STATES[this->state] << std::endl;
243
244     return *this;
245 }
246 };
247
248 class Succeder : public Task
249 {
250     SMART task;
251
252 public:
253     Succeder(SMART task = {})
254         : Task{ "Succeder" }, task{ task }{}
255
256     /**/
257     \brief
258     Succeder function that always return success, irrespective of what the
259     child node actually returned. It is useful in cases where you want to
260     process a branch of a tree where a failure is expected or anticipated,
261     but you don't want to abandon processing of a sequence that branch sits
262     on
263
264     \param log
265     An object pointer to Log class
266
267     \param level
268     A string to keep track of the level
269
270     \return
271     Returns this pointer
272     *****/
273     Succeder& operator()(Log* log = nullptr, std::string level = "")
274     {
275         if (log)
276             *log << level << "Succeder()" << std::endl;
277
278         this->state = State::Success;
279         this->task->operator()(log, level + "| ");
280
281         if (log)
282             *log << level << "L " << STATES[this->state] << std::endl;
283
284         return *this;
285     }
286 };
287
288 class Repeater : public Task
289 {
290     SMART task;
291     int counter;
292
293 public:
294     Repeater(SMART task = {}, int counter = 0)
295         : Task{ "Repeater" }, task{ task }, counter{ counter }{}
296
297     /**/
298     \brief
299     Repeater function that will reprocess its child node each time its child
300     returns a result. It is often used at the very base of the tree to make
301     the tree to run continuously. Repeaters may optionally run their
302     children a set number of times before returning to their parent.
303
304     \param log
305     An object pointer to Log class
306
307     \param level
308     A string to keep track of the level
309
310     \return
311     Returns this pointer
312     *****/
313     Repeater& operator()(Log* log = nullptr, std::string level = "")
314     {
315         if (log)
316             *log << level << "Repeater(" << counter << ")" << std::endl;
317
318         this->state = State::Success;
319
320         while(this->task && ((this->counter-- > 0))
321             this->task->operator()(log, level + "| ");
322
323         if (log)
324             *log << level << "L " << STATES[this->state] << std::endl;

```

```
325
326         return *this;
327     }
328 };
329
330 class Repeat_until_fail : public Task
331 {
332     SMART task;
333
334 public:
335     Repeat_until_fail(SMART task = {})
336         : Task{ "Repeat_until_fail" }, task{ task }{}
337
338     /*!*****
339     \brief
340     Repeat_until_fail function. Like a repeaters, these decorators will
341     continue to reprocess their child until a child finally returns a
342     failure, at which point the repeater will return success to its parent.
343
344     \param log
345     An object pointer to Log class
346
347     \param level
348     A string to keep track of the level
349
350     \return
351     Returns this pointer
352     *****/
353     Repeat_until_fail& operator()(Log* log = nullptr, std::string level = "")
354     {
355         if (log)
356             *log << level << "Repeat_until_fail()" << std::endl;
357
358         this->state = State::Success;
359
360         while (this->task &&
361             this->task->operator()(log, level + "| ").getState()
362             == State::Success) {}
363
364         if (log)
365             *log << level << "L " << STATES[this->state] << std::endl;
366
367         return *this;
368     }
369 };
370
371 } // end namespace
372
373 #endif
```

[VPL](#)

[◀ Showcase: Battleship](#)

Jump to...

⬆

[Fuzzy Logic ▶](#)

You are logged in as [Wei Zhe GOH](#) ([Log out](#))
[cs380su21-meta.sg](#)
[Data retention summary.](#)
[Get the mobile app](#)