**Тема:** Поведенческие паттерны: «Цепочка обязанностей», «Посредник», «Стратегия».

**«Цепочка обязанностей»**

Описание:

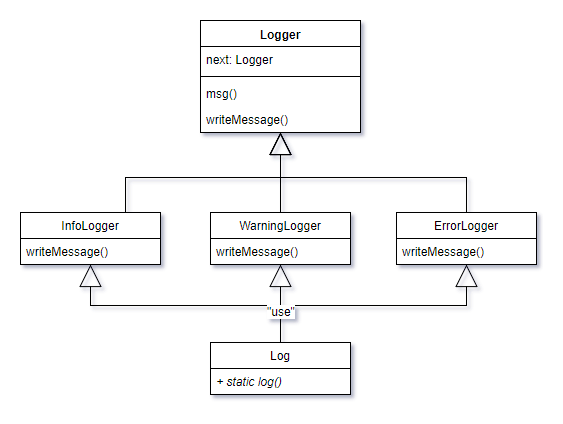
Шаблон рекомендован для использования в условиях:

* в разрабатываемой системе имеется группа объектов, которые могут обрабатывать сообщения определенного типа;
* все сообщения должны быть обработаны хотя бы одним объектом системы;
* сообщения в системе обрабатываются по схеме «обработай сам либо перешли другому», то есть одни сообщения обрабатываются на том уровне, где они получены, а другие пересылаются объектам иного уровня.

Пример использования:

Реализуем функционал «Логгера» в приложении. Сделаем это в соответствии с паттерном «цепочка обязанностей». Для этого нам понадобится абстрактный класс Logger, наследуемые от него InfoLogger, WarningLogger, ErrorLogger, а также класс Log, который будет использоваться клиентом.

Диаграмма классов:



Исходный код:

Logger.js

**class** Logger {  
 **static get** *INFO*() {  
 **return** 'Info';  
 }  
  
 **static get** *WARNING*() {  
 **return** 'Warning';  
 }  
  
 **static get** *ERROR*() {  
 **return** 'Error';  
 }  
  
 constructor(mask = 'Info') {  
 **switch** (mask) {  
 **case** 'Warning':  
 **case** 'Error':  
 **this**.mask = mask;  
 **break**;  
 **case** 'Info':  
 **default**:  
 **this**.mask = 'Info';  
 }  
 }  
  
 **get** Next() {  
 **return this**.next;  
 }  
  
 **set** Next(logger) {  
 **this**.next = logger;  
 **return** logger;  
 }  
  
 msg(message, mask = 'Info') {  
 **if** (mask === **this**.mask) {  
 **this**.\_writeMessage(message);  
 } **else** {  
 **this**.Next.msg(message, mask);  
 }  
 }  
  
 \_writeMessage() {  
 }  
}

InfoLogger.js

**class** InfoLogger **extends** Logger {  
 \_writeMessage() {  
 console.info(...arguments);  
 }  
}

WarningLogger.js

**class** WarningLogger **extends** Logger {  
 \_writeMessage() {  
 console.warn(...arguments);  
 }  
}

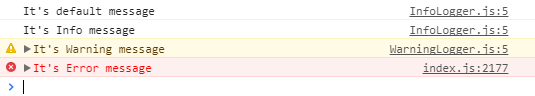
ErrorLogger.js

**class** ErrorLogger **extends** Logger {  
 \_writeMessage() {  
 console.error(...arguments);  
 }  
}

Log.js

**class** Log {  
 **static** *log*(message, mask = 'Info') {  
 Log.mainLogger = **new** InfoLogger(Logger.*INFO*);  
 Log.warningLogger = Log.mainLogger.Next = **new** WarningLogger(Logger.*WARNING*);  
 Log.errorLogger = Log.warningLogger.Next = **new** ErrorLogger(Logger.*ERROR*);  
  
 Log.mainLogger.msg(message, mask);  
 }  
}

Снимок экрана:



**«Стратегия»**

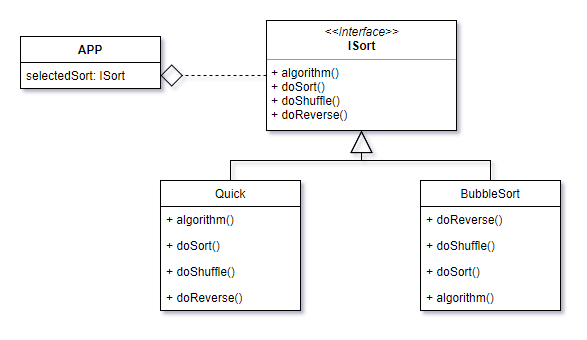
Описание:

Паттерн «стратегия» позволяет спроектировать взаимозаменяемые и надежные алгоритмы или стратегии, позволяющие выполнять модификацию алгоритмов без изменений на стороне клиента. «Стратегия» подразумевает определение для каждого алгоритма или стратегии отдельный класс со стандартным интерфейсом.

Пример реализации:

Создадим интерфейс сортировок ISort. Также создадим классы, реализующие саму логику алгоритмов сортировок.

Диаграмма классов:



Исходный код:

ISort.js

**class** ISort {  
 algorithm(shuffle = **false**) {  
 }  
  
 doSort() {  
 }  
  
 doShuffle() {  
 }  
  
 doReverse() {  
 }  
}

Quick.js

**class** Quick **extends** ISort {  
 constructor(scene) {  
 **super**();  
 **this**.scene = scene;  
 **this**.eventSorted = **new** Event('sorted', { bubbles: **true**, cancelable: **true** });  
 **this**.algorithm = **this**.algorithm.bind(**this**);  
 **this**.doSort = **this**.doSort.bind(**this**);  
 **this**.doShuffle = **this**.doShuffle.bind(**this**);  
 **this**.doReverse = **this**.doReverse.bind(**this**);  
 }  
  
 algorithm(direction = '') {  
 **const** timestamp = **new** Date();  
 **let** colorBlocks = **this**.scene.childNodes,  
 timerID,  
 baseCoefficient = 1,  
 ratio = 1;  
  
 **let** BlockSizeIterator = 1, BlockIterator = 0, LeftBlockIterator, RightBlockIterator, MergeIterator;  
  
 **let** LeftBorder, MidBorder, RightBorder;  
  
 **const** iteration = () => {  
 //Производим слияние с сортировкой пары блоков начинающуюся с элемента BlockIterator  
 //левый размером BlockSizeIterator, правый размером BlockSizeIterator или меньше  
 LeftBlockIterator = 0;  
 RightBlockIterator = 0;  
 LeftBorder = BlockIterator;  
 MidBorder = BlockIterator + BlockSizeIterator;  
 RightBorder = BlockIterator + 2 \* BlockSizeIterator;  
 RightBorder = (RightBorder < colorBlocks.length) ? RightBorder : colorBlocks.length;  
 **let** SortedBlock = [];  
  
 //Пока в обоих массивах есть элементы выбираем меньший из них и заносим в отсортированный блок  
 **while** (LeftBorder + LeftBlockIterator < MidBorder && MidBorder + RightBlockIterator < RightBorder) {  
 **let** condition;  
 **switch** (direction) {  
 **case** 'reverse':  
 condition = parseInt(colorBlocks[LeftBorder + LeftBlockIterator].dataset.hsv, 10) > parseInt(colorBlocks[MidBorder + RightBlockIterator].dataset.hsv, 10);  
 **break**;  
 **case** 'shuffle':  
 **case** 'random':  
 **case** 'rand':  
 condition = (Math.random() - 0.5) > 0;  
 **break**;  
 **default**:  
 condition = parseInt(colorBlocks[LeftBorder + LeftBlockIterator].dataset.hsv, 10) < parseInt(colorBlocks[MidBorder + RightBlockIterator].dataset.hsv, 10);  
 }  
 **if** (condition) {  
 SortedBlock[LeftBlockIterator + RightBlockIterator] = colorBlocks[LeftBorder + LeftBlockIterator];  
 LeftBlockIterator += 1;  
 }  
 **else** {  
 SortedBlock[LeftBlockIterator + RightBlockIterator] = colorBlocks[MidBorder + RightBlockIterator];  
 RightBlockIterator += 1;  
 }  
 }  
 //После этого заносим оставшиеся элементы из левого или правого блока  
 **while** (LeftBorder + LeftBlockIterator < MidBorder) {  
 SortedBlock[LeftBlockIterator + RightBlockIterator] = colorBlocks[LeftBorder + LeftBlockIterator];  
 LeftBlockIterator += 1;  
 }  
 **while** (MidBorder + RightBlockIterator < RightBorder) {  
 SortedBlock[LeftBlockIterator + RightBlockIterator] = colorBlocks[MidBorder + RightBlockIterator];  
 RightBlockIterator += 1;  
 }  
  
 **for** (MergeIterator = 0; MergeIterator < LeftBlockIterator + RightBlockIterator; MergeIterator++) {  
 **this**.scene.insertBefore(SortedBlock[MergeIterator], colorBlocks[LeftBorder + MergeIterator]);  
 }  
 BlockIterator += 2 \* BlockSizeIterator;  
 **if** (BlockIterator < colorBlocks.length - BlockSizeIterator) {  
 timerID = setTimeout(iteration, baseCoefficient \* ratio);  
 } **else** {  
 BlockSizeIterator \*= 2;  
 BlockIterator = 0;   
 **if** (BlockSizeIterator < colorBlocks.length) {  
 timerID = setTimeout(iteration, baseCoefficient \* ratio);  
 } **else** {  
 clearTimeout(timerID);  
 console.log('set event "sorted" with ' + (((**new** Date() - timestamp) / 1000) / 60\*60) + ' seconds');  
 **this**.scene.dispatchEvent(**this**.eventSorted);  
 }  
 }  
 };  
 **if** ((BlockSizeIterator < colorBlocks.length) && (BlockIterator < colorBlocks.length - BlockSizeIterator)) {  
 timerID = setTimeout(iteration, baseCoefficient \* ratio);  
 }  
 }  
  
 doSort() {  
 **this**.algorithm();  
 }  
  
 doShuffle() {  
 **this**.algorithm('shuffle');  
 }  
  
 doReverse() {  
 **this**.algorithm('reverse');  
 }  
}

BubbleSort.js

**class** BubbleSort **extends** ISort {  
 constructor(scene) {  
 **super**();  
 **this**.scene = scene;  
 **this**.eventSorted = **new** Event('sorted', { bubbles: **true**, cancelable: **true** });  
 **this**.algorithm = **this**.algorithm.bind(**this**);  
 **this**.doSort = **this**.doSort.bind(**this**);  
 **this**.doShuffle = **this**.doShuffle.bind(**this**);  
 **this**.doReverse = **this**.doReverse.bind(**this**);  
 }  
  
 algorithm(direction = '') {  
 **const** timestamp = **new** Date();  
 **let** colorBlocks = **this**.scene.childNodes,  
 i = 0, j = 0;  
 **const** iteration = () => {  
 **let** color1 = parseInt(colorBlocks[j].dataset.hsv, 10),  
 color2 = parseInt(colorBlocks[j + 1].dataset.hsv, 10),  
 condition;  
 **switch** (direction) {  
 **case** 'reverse':  
 condition = color1 < color2;  
 **break**;  
 **case** 'shuffle':  
 **case** 'random':  
 **case** 'rand':  
 condition = (Math.random() - 0.5) > 0;  
 **break**;  
 **default**:  
 condition = color1 > color2;  
 }  
 **if** (condition) {  
 **this**.scene.insertBefore(colorBlocks[j + 1], colorBlocks[j]);  
 }  
 **if** (j < colorBlocks.length - 2 - i) {  
 j++;  
 timerID = setTimeout(iteration, 1, direction);  
 } **else** {  
 **if** (i < colorBlocks.length - 2) {  
 i++;  
 j = 0;  
 timerID = setTimeout(iteration, 1, direction);  
 } **else** {  
 clearTimeout(timerID);  
 console.log('set event "sorted" with ' + (((**new** Date() - timestamp) / 1000) / 60\*60) + ' seconds');  
 **this**.scene.dispatchEvent(**this**.eventSorted);  
 }  
 }  
 };  
 **let** timerID = setTimeout(iteration, 1, direction);  
 }  
  
 doSort() {  
 **this**.algorithm();  
 }  
  
 doShuffle() {  
 **this**.algorithm('shuffle');  
 }  
  
 doReverse() {  
 **this**.algorithm('reverse');  
 }  
}