**Тема:** Структурные паттерны: «Адаптер», «Декоратор», «Итератор», «Композит».

**«Декоратор»**

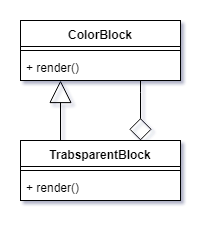
Описание:

Данный паттерн является альтернативой практики создания подклассов для расширения функциональности. Его суть заключается в следующем. Объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта. Для реализации это создается абстрактный класс, представляющий как исходный класс, так и новые, добавляемые в класс функции. В классах-декораторах новые функции вызываются в требуемой последовательности — до или после вызова последующего объекта. При желании остаётся возможность использовать исходный класс (без расширения функциональности), если на его объект сохранилась ссылка.

Пример реализации:

Класс TransparentBlock декорирует класс ColorBlock, заменяя сплошной цвет фона у блока на градиентый переход к прозрачному в центре.

Диаграмма классов:



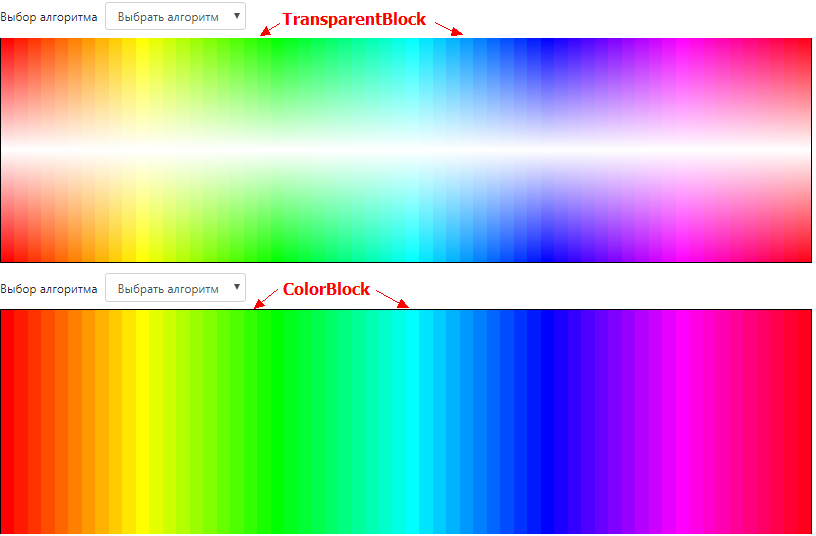
Исходный код:

ColorBlock.js

**class** ColorBlock **extends** React.Component {  
 render() {  
 **let** block;  
 block = <div  
 style={{  
 display: 'inline-block',  
 width: **this**.props.width,  
 height: '100%',  
 background: **this**.props.color[0],  
 }}  
 // eslint-disable-next-line  
 data-hsv={String(**this**.props.color[1] + ', 100' + ', 100')}  
 />;  
 **return** block;  
 }  
}

TransparentBlock.js

**class** TransparentBlock **extends** ColorBlock {  
 render() {  
 //Декорирование. Изменение цвета фона у блока.  
 **let** gradient = 'linear-gradient(to top, rgba(' + **this**.props.color[0].slice(4, -1) + ', 1), rgba(' + **this**.props.color[0].slice(4, -1) + ', 0), rgba(' + **this**.props.color[0].slice(4, -1) + ', 1))',  
 colorSet = [gradient,**this**.props.color[1]];  
 // Вызов функции render() у экземпляра класса ColorBlock:  
 **return** <ColorBlock color={colorSet} width={**this**.props.width} ref={(el) => **this**.block = el}/>;  
 }  
}



**«Итератор»**

Описание:

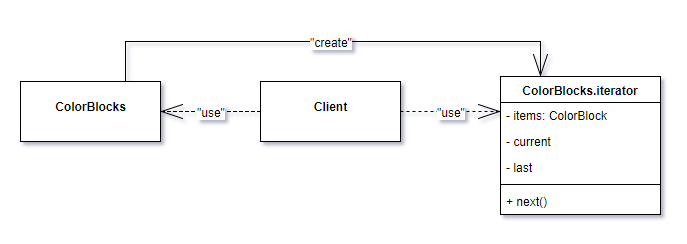
Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.

Например, такие элементы как дерево, связанный список, хэш-таблица и массив могут быть пролистаны (и модифицированы) с помощью объекта Итератор.

Пример реализации:

Класс ColorBlocks создаёт объект .iterator, имеющий публичный метод .next(), который при каждом обращении возвращает либо объект со следующим экземпляром коллекции и маркером конца коллекции done = false, либо объект, содержащий только маркер done = true – это будет сигналом о том, что коллекция закончилась.

Диаграмма классов:



Исходный код:

ColorBlocks.js

**class** ColorBlocks {  
 constructor(props) {  
 **super**(props);  
 **this**.colors = **this**.\_getColorSet(**this**.props ? **this**.props.colorsNumber || 60 : 60);  
 **this**.blocks = **this**.\_getColorBlocks();  
 **this**.iterator = {  
 blocks: **this**.blocks,  
 from: 0,  
 to: **this**.blocks.length - 1,  
 };  
 **this**.iterator[Symbol.iterator] = **function** () {  
 **let** items = **this**.blocks,  
 currIdx = **this**.from,  
 lastIdx = **this**.to;  
 **return** {  
 next() {  
 **if** (currIdx <= lastIdx) {  
 **return** {  
 done: **false**,  
 value: items[currIdx++],  
 }  
 } **else** {  
 **return** {  
 done: **true** }  
 }  
 }  
 }  
 };  
 }  
 ...  
}

Client:

...  
**this**.clrBlocks = [];  
**for** (**let** block **of** (**new** ColorBlocks()).iterator) {  
 **this**.clrBlocks.push(block);  
}  
...

**«Композит»**

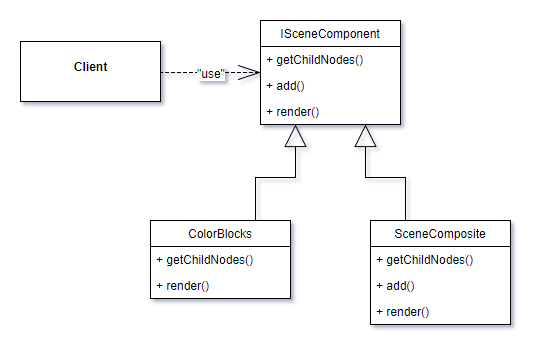
Описание:

Предположим, нам необходимо обрабатывать группу или композицию структур объектов одновременно. Для этого паттерн «композит» («компоновщик») предлагает определить классы для композитных и атомарных объектов таким образом, чтобы они реализовывали один и тот же интерфейс. В результате мы сможем реализовать иерархию объектов вида «часть-целое», чтобы клиенты единообразно обрабатывали составные и индивидуальные объекты. Это очень похоже на структуру «дерево». Мы одинаково работаем как с деревом, поддеревом, так и с листовым элементом.

Пример реализации:

Создадим абстрактный класс ISceneComponent, в который вынесем общую логику для листовых и композитных элементов. Также создадим классы SceneComposite и ColorBlocks, которые будут отвечать за композитные и листовые элементы соответственно.

Диаграмма классов:



Исходный код:

ISceneComponent.js

**class** ISceneComponent **extends** React.Component {  
 getChildNodes() {}  
  
 add(CompositeOrScene) {}  
  
 render(content) {  
 **return** (  
 <Grid>  
 <Row>  
 <Col xs={12}>  
 <Form inline={**true**}>  
 <FormGroup>  
 <ControlLabel>Выбор алгоритма</ControlLabel>  
 <FormControl  
 componentClass="select"  
 placeholder={"Выбор алгоритма"}  
 name="sorts"  
 className="sorts"  
 defaultValue={'null'}  
 // onChange={this.\_changeAlgorithm}  
 // ref={(el) => this.selectAlg = el}  
 style={{ margin: '10px' }}  
 >  
 <option value="null" disabled={**true**}>Выбрать алгоритм</option>  
 <option value="bubble">Пузырёк</option>  
 <option value="shaker">Шейкер</option>  
 <option value="merge">Слиянием (base 2)</option>  
 </FormControl>  
 </FormGroup>  
 </Form>  
 </Col>  
 </Row>  
 <Row>  
 <Col xs={12}>  
 <div  
 style={styles.scene}  
 >  
 {content}  
 </div>  
 </Col>  
 </Row>  
 </Grid>  
 );  
 }  
}

SceneComposite.js

**class** SceneComposite **extends** ISceneComponent {  
 constructor(props) {  
 **super**(props);  
 **this**.content = [(**this**.props.children || <ColorBlocks colorsNumber={**this**.props.colorsNumber} type={**this**.props.type}/>)];  
 }  
  
 getChildNodes() {  
 **return this**.content;  
 }  
  
 add(Node) {  
 **if** (**typeof** Node.length === 'undefined') {  
 **this**.content.push(Node);  
 } **else** {  
 **this**.content.push(...Node);  
 }  
 }  
  
 render() {  
 **return this**.content;  
 }  
}

ColorBlocks.js

**class** ColorBlocks **extends** ISceneComponent {  
 constructor(props) {...}  
  
 ...  
  
 getChildNodes() {  
 **return null**;  
 }  
  
 add() {  
 }  
  
 render() {  
 **return super**.render(**this**.blocks);  
 }  
}

Снимок экрана:

Композит сцен

