14. Учебный проект: разделяй и властвуй (Часть 1)

Рабочая ветка **module5-task1**

Задача

Абстрактный класс

В этом задании мы попрактикуемся в наследовании. Нам нужно выделить общие части компонентов в абстрактный класс `AbstractComponent`.

1. Изучите структуру существующих компонентов, чтобы понять, какая логика и какие данные повторяются. Не всегда ответ на этот вопрос очевиден, и порой придётся подумать, порисовать, порефакторить. Но если вы описали методы вроде `getElement` и `removeElement` и свойства вроде `_element`, как мы просили в предыдущих заданиях, то общими будут именно они.

2. Опишите абстрактный класс. Это точно такой же класс, как все остальные, только создавать объекты напрямую из него нельзя. От него можно только наследоваться. Поэтому добавьте в конструкторе проверку на `new.target === AbstractComponent` и бросьте исключение, если условие истинно.

3. Далее объявите в абстрактном классе общие свойства и методы, пока что пустые.

4. У всех наших компонентов-наследников обязательно должен быть реализован метод `getTemplate`. Абстрактный класс поможет нам об этом не забыть. Для этого опишите в абстрактном классе метод `getTemplate` внутри которого бросьте исключение. Теперь, если вы забудете в своём компоненте реализовать метод `getTemplate`, об этом вам напомнит ошибка в консоли.

5. А теперь унаследуем все наши компоненты от абстрактного класса `AbstractComponent` с помощью языковой конструкции `extends`.

6. В заключение перенесём реализацию общих методов из потомков (наших компонентов) в родителя (абстрактный класс). Теперь, благодаря ООП и наследованию, у всех наших потомков будут методы, объявленные в родителе, а не дублированные в каждом компоненте.

Не забудьте, что метод `getTemplate` должен остаться в компонентах, как и прочие частные методы.

Чтобы убедиться, что вы всё сделали правильно, запустите проект локально. Он должен сохранить свою полную работоспособность после всех ваших манипуляций.

Больше абстракций богу абстракций

На этом шаге мы максимально абстрагируемся от работы с DOM напрямую в пользу работы с нашими компонентами.

1. У нас появились вспомогательные функции по работе с DOM: `render`, `remove` и т. п. Вынесите их в отдельный модуль, например `utils/render.js`, чтобы не мешать их с другими вспомогательными функциями. А также измените реализацию этих функций, чтобы в них можно было передавать наши компоненты, а не DOM-элементы, насколько это возможно.

2. У нас осталось последнее место, где мы работаем с DOM напрямую — это подписка на события. Откажемся от прямого использования `addEventListener` в `main.js`, оставив его только в компонентах. Для этого добавьте в компоненты методы для установки обработчиков событий, а в тело этих методов перенесите использование `addEventListener`. Глобальные обработчики — на `document` и `window` — остаются как есть.

Было:

```
componentInstance.getElement().querySelector(`form
`).addEventListener(`submit`, () => {});
```
Стало:

```
// Объявление
class Component {
  setSubmitHandler(cb) {

this.getElement().querySelector(`form`).addEventLi
stener(`submit`, cb);
  }
}

// Использование
componentInstance.setSubmitHandler(() => {});
```
Теперь мы ничего не знаем о внутреннем устройстве компонентов и их реальных событиях. Важным для нас остаётся лишь их интерфейс — набор методов, которыми они обладают. Мы можем менять разметку и реальные события отдельных компонентов безболезненно для всего приложения. Главное, сохранять интерфейс компонентов.

Контроллер

Пришло время разгрузить `main.js` и вынести часть связанной логики в отдельную сущность — контроллер. Задача контроллера — создавать компоненты, добавлять их на страницу, навешивать обработчики. То есть реализовывать бизнес-логику и поведение приложения.

1. Заведите директорию для контроллеров. Например, `/src/controllers`.

2. Создайте класс `BoardController` с конструктором и методом `render`.

3. Конструктор должен принимать `container` — элемент, в который контроллер будет всё отрисовывать.

4. Перенесите из `main.js` в метод `render` всю логику по отрисовке задач и кнопки «Load more», а также по навешиванию на них обработчиков. В качестве параметров метод `render` должен принимать данные для отрисовки — задачи.

5. В `main.js` создайте экземпляр `BoardController`, а затем вызовите у него метод `render`, передав в него данные.

16. Личный проект: разделяй и властвуй (Часть 1)

Задача

Абстрактный класс

В этом задании мы попрактикуемся в наследовании. Нам нужно выделить общие части компонентов в абстрактный класс `AbstractComponent`.

1. Изучите структуру существующих компонентов, чтобы понять, какая логика и какие данные повторяются. Не всегда ответ на этот вопрос очевиден, и порой придётся подумать, порисовать, порефакторить. Но если вы описали методы вроде `getElement` и `removeElement` и свойства вроде `_element`, как мы просили в предыдущих заданиях, то общими будут именно они.

2. Опишите абстрактный класс. Это точно такой же класс, как все остальные, только создавать объекты напрямую из него нельзя. От него можно только наследоваться. Поэтому добавьте в конструкторе проверку на `new.target === AbstractComponent` и бросьте исключение, если условие истинно.

3. Далее объявите в абстрактном классе общие свойства и методы, пока что пустые.

4. У всех наших компонентов-наследников обязательно должен быть реализован метод `getTemplate`. Абстрактный класс поможет нам об этом не забыть. Для этого опишите в абстрактном классе метод `getTemplate` внутри которого бросьте исключение. Теперь, если вы забудете в своём компоненте реализовать метод `getTemplate`, об этом вам напомнит ошибка в консоли.

5. А теперь унаследуем все наши компоненты от абстрактного класса `AbstractComponent` с помощью языковой конструкции `extends`.

6. В заключение перенесём реализацию общих методов из потомков (наших компонентов) в родителя (абстрактный класс). Теперь, благодаря ООП и наследованию, у всех наших потомков будут методы, объявленные в родителе, а не дублированные в каждом компоненте.

Не забудьте, что метод `getTemplate` должен остаться в компонентах, как и прочие частные методы.

Чтобы убедиться, что вы всё сделали правильно, запустите проект локально. Он должен сохранить свою полную работоспособность после всех ваших манипуляций.

Больше абстракций богу абстракций

На этом шаге мы максимально абстрагируемся от работы с DOM напрямую в пользу работы с нашими компонентами.

1. У нас появились вспомогательные функции по работе с DOM: `render`, `remove` и т. п. Вынесите их в отдельный модуль, например `utils/render.js`, чтобы не мешать их с другими вспомогательными функциями. А также измените реализацию этих функций, чтобы в них можно было передавать наши компоненты, а не DOM-элементы, насколько это возможно.

2. У нас осталось последнее место, где мы работаем с DOM напрямую — это подписка на события. Откажемся от прямого использования `addEventListener` в `main.js`, оставив его только в компонентах. Для этого добавьте в компоненты методы для установки обработчиков событий, а в тело этих методов перенесите использование `addEventListener`. Глобальные обработчики — на `document` и `window` — остаются как есть.

Было:

```
componentInstance.getElement().querySelector(`form`).addEventListener(`submit`, () => {});
```

Стало:

```
// Объявление
class Component {
  setSubmitHandler(cb) {
```

```
this.getElement().querySelector(`form`).addEventListener(`submit`, cb);
  }
}

// Использование
componentInstance.setSubmitHandler(() => {});
```

Теперь мы ничего не знаем о внутреннем устройстве компонентов и их реальных событиях. Важным для нас остаётся лишь их интерфейс — набор методов, которыми они обладают. Мы можем менять разметку и реальные события отдельных компонентов безболезненно для всего приложения. Главное, сохранять интерфейс компонентов.

Контроллер

Пришло время разгрузить `main.js` и вынести часть связанной логики в отдельную сущность — контроллер. Задача контроллера создавать компоненты, добавлять их на страницу, навешивать обработчики. То есть реализовывать бизнес-логику и поведение приложения.

Киноман

1. Заведите директорию для контроллеров. Например, `/src/controllers`.

2. Создайте класс `PageController` с конструктором и методом `render`.

3. Конструктор должен принимать `container` — элемент, в который контроллер будет всё отрисовывать.

4. Перенесите из `main.js` в метод `render` всю логику по отрисовке фильмов и кнопки «Show more», а также по навешиванию на них обработчиков. В качестве параметров метод `render` должен принимать данные для отрисовки — фильмы и комментарии.

5. В `main.js` создайте экземпляр `PageController`, а затем вызовите у него метод `render`, передав в него данные.

Bigtrip

1. Заведите директорию для контроллеров. Например, `/src/controllers`.

2. Создайте класс `TripController` с конструктором и методом `render`.

3. Конструктор должен принимать `container` — элемент, в который контроллер будет всё отрисовывать.

4. Перенесите из `main.js` в метод `render` всю логику по отрисовке точек маршрута, а также по навешиванию на них обработчиков. В качестве параметров метод `render` должен принимать данные для отрисовки — точки маршрута, все возможные дополнительные опции и пункты назначения.

5. В `main.js` создайте экземпляр `TripController`, а затем вызовите у него метод `render`, передав в него данные.