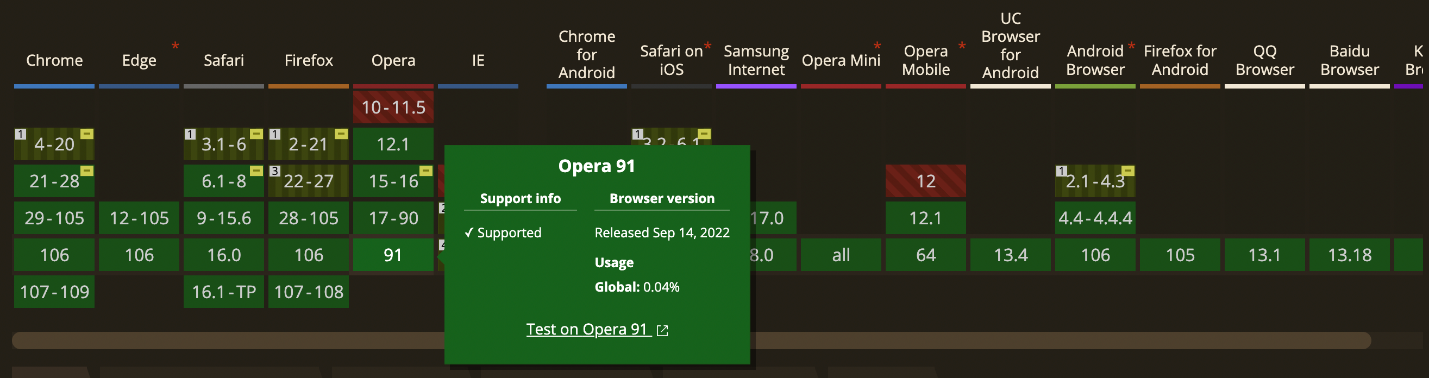
**Компонентный подход**

В процессе вёрстки более или менее сложного интерфейса вы быстро заметите, что в дизайне повторяются одни и те же элементы: кнопки, всплывающие окна, карточки, списки. Они выглядят одинаково на разных страницах сайта или просто в разных его зонах. Это и есть компонентный подход к интерфейсам.

Пользователи любят предсказуемый интерфейс. Когда они привыкнут к поведению какого-то элемента, им будет проще использовать аналогичный в другом месте.

Вот, надеемся, знакомый вам пример:

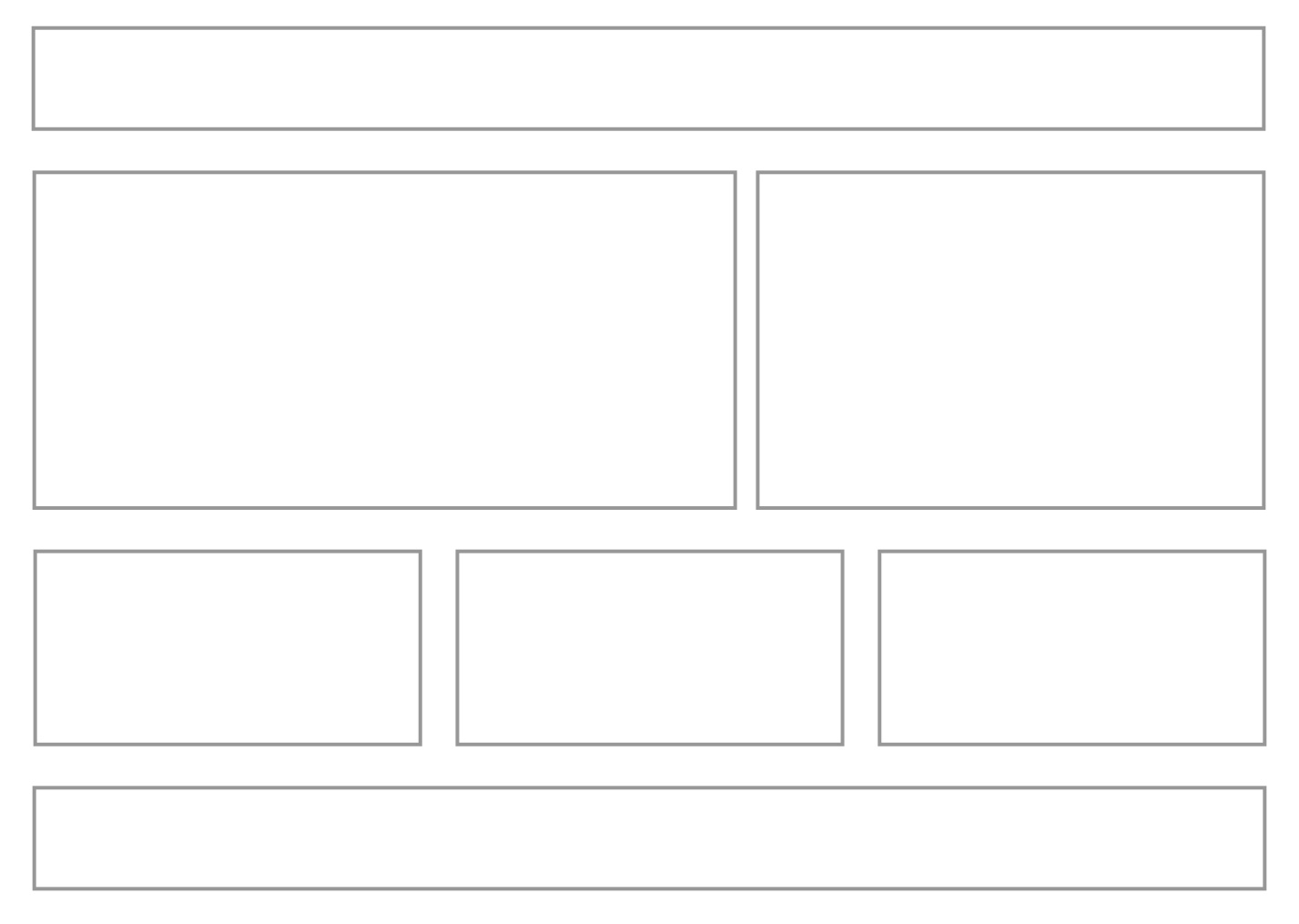


При наведению на каждую клетку сайта [caniuse.com](http://caniuse.com/) появляется тултип с информацией. Каждая клетка ведёт себя одинаковым образом. И каждый тултип похож на другой. И клетка, и тултип — это компоненты интерфейса.

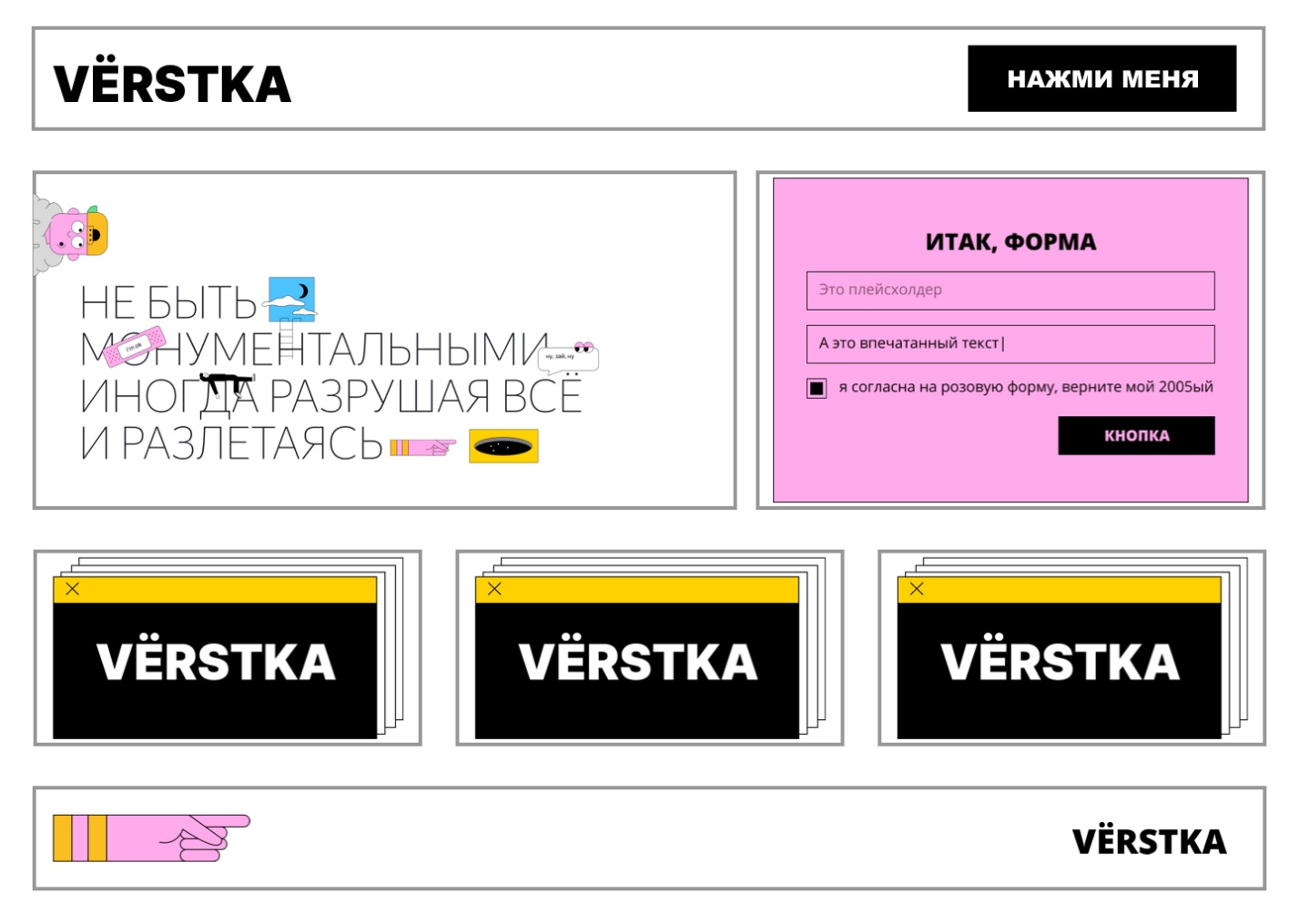
Понять разницу между HTML-элементом и компонентом проще всего, если считать, что компонент содержит в себе и структуру, и внешний вид, и функциональность. То есть компонент — это набор из HTML, CSS и иногда JavaScript кода, который может быть использован повторно в разных контекстах.

**Лейаут и компоненты**

Компонентный подход — самый распространённый взгляд на проектирование и разработку интерфейса. Он заключается в том, что при разработке страниц вы проводите мысленное разделение на лейаут и компоненты, наполняющие его. Лейаут — это каркас вашей страницы, её общая структура.



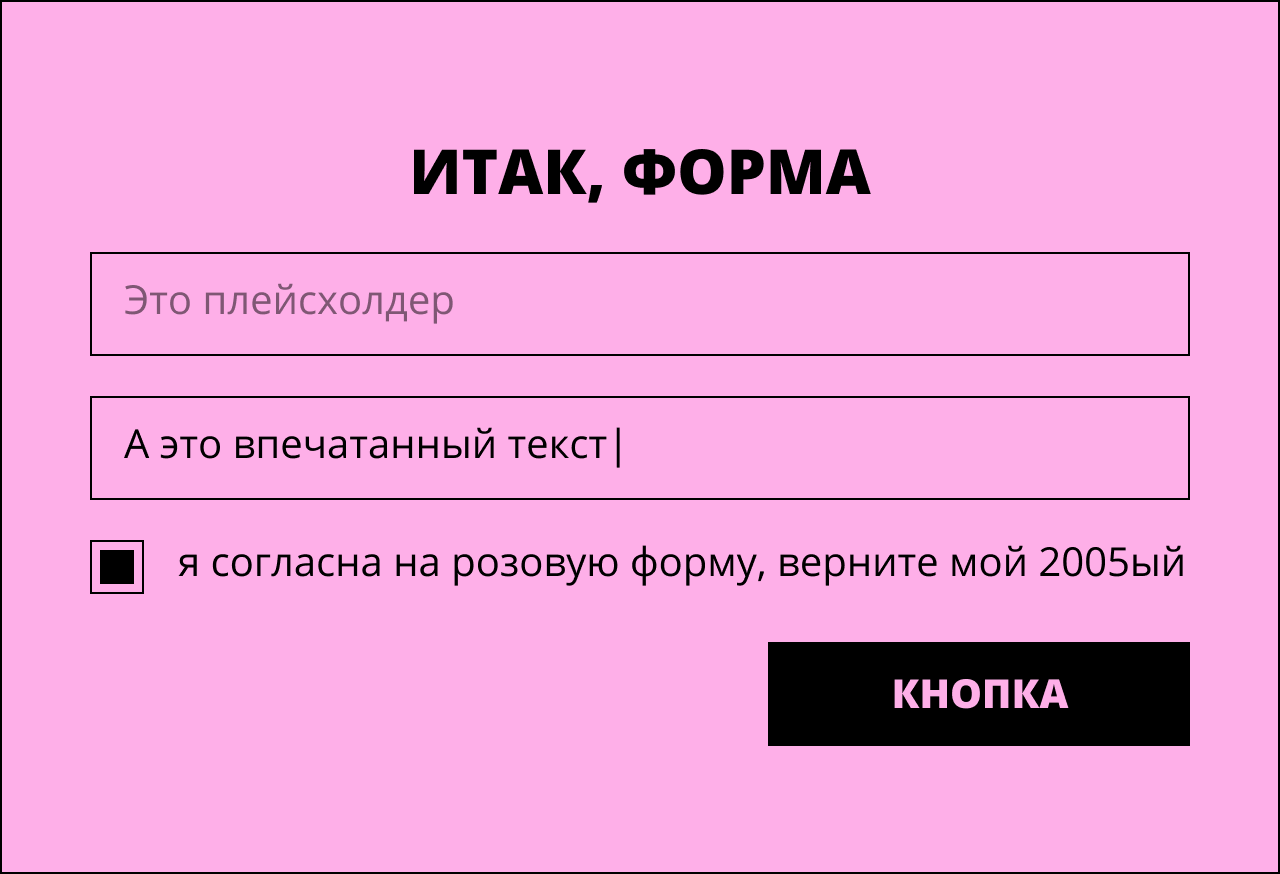
Схематическое изображение лейаута сайта



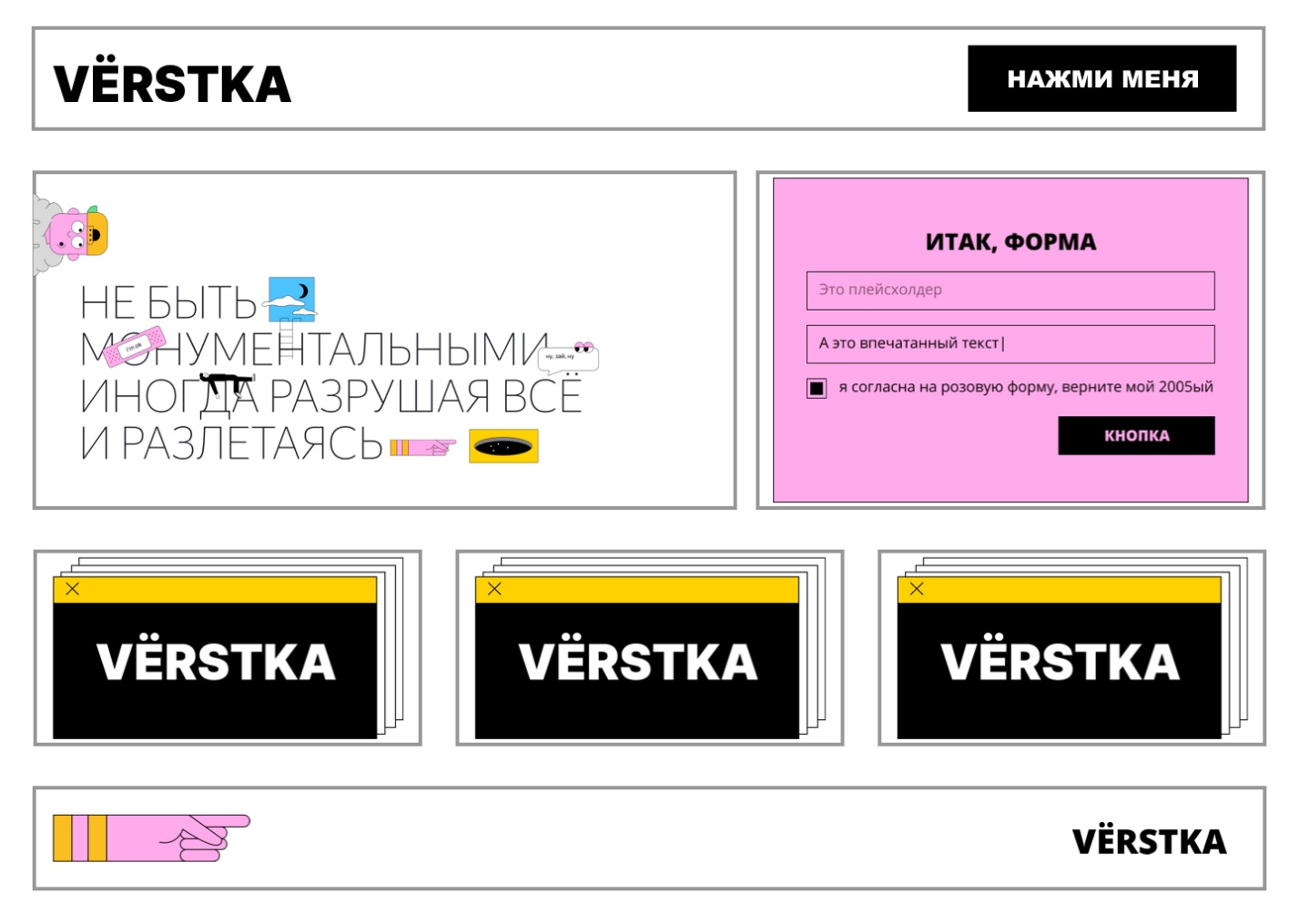
Лейаут, наполненный компонентами

**Компоненты и элементы**

Компоненты — это обычно некоторый набор элементов. Например, компонент «Форма» состоит из элементов: заголовок, поле ввода, текст лейбла и кнопка.



А теперь непростой момент: элементы внутри компонентов сами могут быть компонентами. Например, кнопка может повторно использоваться в разных частях сайта.



На странице две очень похожие друг на друга кнопки

Кнопка в шапке и кнопка в форме отличаются размерами и цветом текста. Скорее всего, у них разная функциональность. Но эти особенности уступают перед одной важной деталью — таких кнопок в интерфейсе могут быть десятки, и все они примерно одинаковые. Поэтому лучше сделать один компонент «Кнопка» со всеми общими свойствами, а особенности поведения как-то присвоить кнопкам-элементам, а не кнопкам-компонентам.

Но элементы большого компонента не всегда другие компоненты. Если какой-то элемент используется только в контексте какого-то конкретного компонента, и, скорее всего, не будет использован нигде больше — это просто элемент.



Логотип справа, скорее всего, встретится в интерфейсе не раз, а вот рука с указателем на него — это что-то специфичное для подвала сайта

**Объёмы компонентов**

Получается, что компоненты бывают разные по объёму. В одних компонентах может быть много элементов, а в других — всего один. Порой сложно бывает решить, как делить интерфейс: на мелкие детали или на крупные.

Как делить интерфейс — решение разработчика. Принцип один — вы будете быстрее строить интерфейс из крупных строительных блоков, поэтому как только видите что-то, что потенциально может быть объединено в конструкцию, подходящую для повторного использования, — объединяйте её в компонент. Мы умышленно пока не говорим, как это делать в коде. Сначала важно, чтобы вы поняли принцип стремления к повторному использованию разных частей вашего интерфейса.

Примеры комплексных компонентов:

* Шапка и подвал сайта будут использоваться на всех страницах. Как бы они ни были сложны — это компоненты.
* Форма обратной связи может появляться на разных страницах — это компонент.
* Карточка товара, статьи или услуги — это тоже компонент.
* Всплывающее окно с разным содержимым — это компонент.
* Входная секция с заголовком в статью — это компонент.

Примеры небольших компонентов:

* Кнопки одинакового вида;
* Поля ввода;
* Иконки с одинаковыми свойствами;
* Заголовки с повторяющимися свойствами;
* Абзацы.

Получается, что почти каждый отдельный HTML-элемент может быть компонентом, если его нужно повторно использовать с одинаковыми или очень похожими стилями.

**Почему компонентный подход?**

Разработка подавляющего числа продуктов ведётся в духе компонентного подхода. А самые популярные фреймворки и библиотеки (заготовки для программирования сайтов) строятся на принципах компонентного подхода. Разработка современного продуктового фронтенда в основном связана с такими технологиями, как React, Vue.js, Angular или Svelte, а это всё фреймворки с компонентным подходом. Чуть позже мы покажем, как там это реализовано. Но для начала ответим на вопрос из заголовка.

Компонентный подход к разработке интерфейса — путь экономии в разработке. Собрав систему компонентов однажды, разработчики уменьшают рутину. Одни и те же кнопки, контроллы (элементы управления) и даже крупные блоки сайта не приходится писать заново. Они просто лежат в библиотеке и ждут, когда разработчик скопирует заготовку и вставит в нужное место.

Компонентный подход — путь к осмысленному масштабированию проекта. Когда ваш сайт развивается, появляются новые разделы и функциональность. Собирать новый интерфейс гораздо проще из заготовок. С системой компонентов бизнес быстрее получает результат.

В конце концов, компонентный подход позволяет наладить цикл непрерывных улучшений. При появлении библиотеки компонентов вы получаете «источник правды». Вы развиваете библиотеку, учитывая всё новые требования: доступности, бизнес-логики, функциональности интерфейса. После подключения библиотеки к сайту все обновления компонентов будут поступать оттуда.

**За пределами вёрстки. Как компоненты работают в фреймворках**

Перед тем, как говорить о принципах вёрстки, позволяющих создавать компоненты, полезно увидеть практику применения компонентного подхода в его максимальном воплощении — в компонентных фреймворках. Это позволит сформировать картину результата, первым этапом получения которого является вёрстка.

Идеальный компонент — это набор разметки, стилей и функциональности, который можно переместить в любую новую зону одного проекта или даже в совершенно новый проект, без изменения кода самого компонента.

Вот пример из библиотеки React:

В видео показано, как из одного проекта с именем react-project-1 копируют папку с компонентом во второй проект с именем react-project-2. После этого остаётся только импортировать компонент в страницу, вставить в нужное место и дописать настройки, определённые в компоненте. На деле переиспользование в новом проекте выглядит похоже на HTML-теги, но вместо тегов используют целые компоненты. Вот такой код автор вставляет в новый проект:

Скопировать код

JSX

<Accordition buttonText="Кто автор?">

Кен Кизи

</Accordition>

<Accordition buttonText="Что за книга">

«Пролетая над гнездом кукушки»

</Accordition>

И на странице появляются два компонента с выпадающим по нажатию на кнопку текстом.

В других библиотеках повторное использование может быть ещё проще: во Vue.js или Svelte весь код компонента хранится в одном файле со специальным расширением. Этот файл содержит и стили, и разметку, и функциональность. Поэтому перетаскивать из проекта в проект приходится не папку с компонентом, а всего один файл.

А теперь представьте, что перетаскивать вообще ничего не нужно, весь код компонентов хранится где-то в сети, и вы только подключаете его оттуда в свои проекты. Обновления библиотеки компонентов также автоматически попадают к вам, вы только пользуетесь конструкциями вида:

Скопировать код

JSX

<Accordition buttonText="Кто автор?">

Кен Кизи

</Accordition>

После этого они превращаются в интерфейс. Это реальность современной разработки. Одни люди составляют библиотеки компонентов, другие их используют.

**Приземление в HTML и CSS**

Разделение кода на компоненты — опция, которая лежит за пределами вёрстки на HTML и CSS. Если CSS-код вы можете разделить на сколь угодно малые части и подключить последовательно к HTML-файлу вашей страницы, то HTML-код воспроизводится браузером как монолит. То есть вся страница с HTML-кодом должна быть загружена браузером разом. В HTML нет возможности создать свои теги и компоненты без программирования.

Получается, что реализовать компоненты на чистом HTML и CSS невозможно. Нужны дополнительные программы, способные вставить отдельный кусок HTML-кода в общее полотно страницы. Такие программы называются сборщиками или бандлерами. Кроме сборщика для реализации компонента, потребуется ещё одна программа — шаблонизатор. Она позволит создавать на HTML основу для контента, а сам контент помещать внутрь тегов при вызове компонента в нужном месте. Работает это примерно так:

Скопировать код

HTML

<div class="component">

<img src="{image}" alt="{altText}">

<h2>{title}</h2>

<p>{text}</p>

</div>

Когда программа-шаблонизатор увидит конструкции в фигурных скобках, она поймёт, что туда можно вставить содержимое с соответствующим именем. Часто это содержимое передаётся как атрибут компонента при вызове.

Скопировать код

HTML

<component

image="/images/img.png"

altText="альтернативный текст"

title="Текст заголовка"

text="Описание"

/>

Про настройку сборщиков и шаблонизаторов расскажем в других дополнительных материалах.

Сейчас вам понятны ограничения, но это не повод забыть о компонентном подходе. Потому что идеи компонентного подхода должны влиять на то, как вы именуете ваши классы в HTML и какие решения принимаете при написании CSS-кода.

**Компонентный подход в HTML и CSS**

Все принципы вёрстки в духе компонентного подхода исходят из основной идеи: повторяющиеся в интерфейсе блоки кода нужно максимально подготовить к повторному использованию в любом контексте. На деле нужно уделить особое внимание трём параметрам: независимости от окружения, отсутствию внешней геометрии и позиционирования, адаптивности размеров.

**Независимость от окружения**

Это означает, что в коде вашего компонента должно содержаться всё для его корректного отображения и работы. Предположим, вы описываете компонент карточки. В HTML он будет выглядеть примерно так:

Скопировать код

HTML

<article class="card">

<img src="/path-to-image.png" class="card-image" alt="текст">

<h2 class="card-title">Заголовок карточки</h2>

<p class="card-text">Абзац с описанием</p>

</article>

Принцип независимости от окружения диктует, что все стили этой карточки должны быть описаны для класса card и потомков, не рассчитывая, что какие-то из стилей придут снаружи. А это, в свою очередь, означает и переопределение всех дефолтных стилей браузера внутри.

Скопировать код

CSS

.card {

box-sizing: border-box;

font-family: Inter, sans-serif;

}

.card-image {

width: 100%;

}

.card-title {

margin: 0;

}

.card-text {

margin: 0;

}

В этом примере мы переопределили внутри компонента все стили, которые могли бы прийти снаружи, от браузера. Теперь компонент изолирован, он будет работать одинаково в любом контексте. Понятно, что стилей в карточке будет гораздо больше. Основное здесь то, что мы нивелировали влияние внешних факторов на этот компонент.

Несмотря на красоту идеи о полностью независимом компоненте, на деле писать так не всегда удобно. Разработчики часто сбрасывают браузерные стили глобально, задают шрифты для всего body и делают прочие, идущие в разрез с этой идеей вещи. Полная независимость от окружения — красивая концепция, вокруг которой стоит выстраивать диалог в команде. При этом вам самим нужно понимать границы доступного. Одно дело полагаться на семейство шрифтов, которое должно прийти извне, и совсем другое — на цвета, размеры анимации и другие важные для работы компонента свойства.

**Отсутствие внешней геометрии и позиционирования**

Компонент должен полностью определять своё внутреннее устройство, но не должен задавать никаких параметров за своими пределами. К внешней геометрии относят в первую очередь поля (margin) и границы (border). С границами всё просто: если назначить box-sizing: border-box компоненту, границы перестанут влиять на внешнюю геометрию. С полями немного сложнее.

Вернёмся к карточке. Теперь их будет две:

Скопировать код

HTML

<div class="container">

<article class="card">

<img src="/path-to-image.png" class="card-image" alt="текст">

<h2 class="card-title">Заголовок карточки</h2>

<p class="card-text">Абзац с описанием</p>

</article>

<article class="card">

<img src="/path-to-image.png" class="card-image" alt="текст">

<h2 class="card-title">Заголовок карточки</h2>

<p class="card-text">Абзац с описанием</p>

</article>

</div>

Мы хотим задать первой margin-bottom, чтобы отодвинуть вторую. Этого не стоит делать для селектора .card. Ведь в этом случае вторая карточка тоже получит этот нижний отступ.

Это очень простой пример, но с более сложными компонентами легко забыть, что все стили для компонента применятся в любом его контексте использования. Как же быть, если нужен отступ? Использовать ещё один класс или более сложный селектор, который конкретизирует контекст именно этой карточки.

Вот первый вариант:

Скопировать код

HTML

<div class="container">

<article class="card mb">

<img src="/path-to-image.png" class="card-image" alt="текст">

<h2 class="card-title">Заголовок карточки</h2>

<p class="card-text">Абзац с описанием</p>

</article>

<article class="card">

<img src="/path-to-image.png" class="card-image" alt="текст">

<h2 class="card-title">Заголовок карточки</h2>

<p class="card-text">Абзац с описанием</p>

</article>

</div>

Мы задали первой карточке класс mb, что значит margin-bottom. Для него будем описывать отступы. Некоторые разработчики любят такой подход и создают целые системы отступов вроде mb-1, mb-2 и т. д. Если классу mb-1 задать отступ в 8px, mb-2 в таком случае задают 16px. Получается система классов, созданная только для описания отступов. Такой подход практикуют в методологии “Atomic CSS”, о ней ещё поговорим в следующих статьях.

Вот ещё вариант:

Скопировать код

HTML

<div class="container">

<article class="card container-card--starter">

<img src="/path-to-image.png" class="card-image" alt="текст">

<h2 class="card-title">Заголовок карточки</h2>

<p class="card-text">Абзац с описанием</p>

</article>

<article class="card">

<img src="/path-to-image.png" class="card-image" alt="текст">

<h2 class="card-title">Заголовок карточки</h2>

<p class="card-text">Абзац с описанием</p>

</article>

</div>

Идея та же самая — добавить дополнительный класс. Но за счёт имени класса container-card--starter мы более понятно определяем, что эта карточка является не только самостоятельным компонентом, но и элементом блока container, причём тем, с которого он начинается. Этот подход ближе к методологии БЭМ, про неё тоже будет дальше. БЭМ намного жёстче подходит к именованию, но идея такая: именами классов показывать принадлежность компонентов друг к другу.

Ещё неплохой вариант на CSS:

Скопировать код

CSS

.container .card:first-of-type {

margin-bottom: 8px;

}

Таким кодом мы описали, что первому элементу с классом card, вложенному в контейнер container, нужно задать отступ. Этот вариант тоже связан с определением контекста использования компонента через его родительский блок, работа идёт с контекстом, а не с компонентом.

Плохой вариант:

Скопировать код

CSS

.card:first-of-type {

margin-bottom: 8px;

}

Этот вариант плох, потому что вредит повторному использованию. В другом месте, где карточки встанут в ряд, а не одна под другой, этот отступ будет не нужен, но он появится.

С позиционированием ситуация похожа на внешние поля. Если вам нужно фиксированно, абсолютно или липко спозиционировать компонент — добавьте для этого отдельный класс. Всегда существует контекст, где компоненту позиционирование не нужно, и очень редко — когда нужно. Вы можете использовать те же приемы: внедрить систему классов для управления позиционированием, описать контекст в имени дополнительного класса или использовать CSS для выбора элемента через родителя.

**Адаптивность размеров**

Правило простое: если ваш компонент блочный, вам всегда нужно оставить его ширину в 100%. Настройки размеров перенести в дополнительные классы. Это могут быть классы, описывающие контекст, как в примерах с полями и позиционированием, или дополнительные классы с модификацией размеров. Например, userpic\_size\_xl или userpic\_size\_s. Не задавайте размеры основным именам компонентов, размеры могут пригодиться разные.

Если ваш компонент ведёт себя строчно или блочно-строчно, управляйте размерами через внутренние свойства. Например, устанавливайте padding у кнопок вместо ширины и высоты. Если вам нужны модификации размеров — также пользуйтесь дополнительными классами, а не основным. Если ваш компонент строчный или блочно-строчный, отдельно задумайтесь, должен ли он быть таким. Ссылки, скорее всего, должны оставаться строчными, но вот поля ввода и изображения, исходя из нашей практики, удобнее сразу делать блочными, чтобы расширить их возможности использования в качестве компонентов. За взаимное расположение элементов чаще всего отвечает лейаут, а компоненты встраиваются в него в виде блоков.

**Файловые структуры в компонентном подходе**

Некоторые методологии, например, БЭМ, определяют, как раскладывать файлы. В реакте тоже есть принятые подходы. В команде вы можете просто следовать рекомендациям, а можете придумать своё разделение на компоненты.

Бывает полезно разделить вашу вёрстку на такие слои:

* Лейаут — общий каркас страниц.
* Компоненты — большие структурные части.
* UI — совсем небольшие строительные элементы.

В таком случае структура файлов CSS может приобретать подобный вид:

Скопировать код

|-styles

--|-layouts

----|-index.css

----|-about.css

----|-contacts.css

----|-...

--|-components

----|-header.css

----|-footer.css

----|-cover.css

----|-...

--|-UI

----|-buttons.css

----|-social-icons-list.css

----|-inputs.css

----|-...

--|-global.css

Это просто пример разветвлённой файловой структуры, в которой со временем может стать удобнее искать ваши стили.

Главная проблема при подобной сложной организации файловой структуры — это подключение стилей. При реализации сложных файловых структур вам придётся подключать все файлы в нужной последовательности внутрь тега <head> каждой конкретной страницы или импортировать файлы CSS друг в друга через директиву @import.

Через специальные программы-сборщики можно автоматизировать подобный процесс, но настройка такой инфраструктуры у вас ещё впереди. Сейчас мы только хотим проблематизировать эту ситуацию. Файлы компонентов в больших проектах принято правильно организовывать.

Если возвращаться к использованию стилей в компонентных фронтенд-библиотеках и фреймворках, таких как React, там принят подход, когда каждому компоненту выдают отдельную папку внутри директории components, и в отдельной папке хранят весь код компонента. Примерно так:

Скопировать код

|-components

--|-Accordition

----|-Accordition.jsx

----|-Accordition.css

В теме про БЭМ мы разберём один из сложных, но очень наглядных подходов к организации структуры файлов в проекте, чтобы расширить кругозор. В реальности файловых структур много и взглядов на архитектуру проектов тоже.

**Внешние библиотеки компонентов**

Создание компонентов и лейаута — это и есть основная задача верстальщика. Компоненты разнятся от проекта к проекту. Чтобы повторно их использовать, нужно их сначала создать. Но иногда бывают задачи, не требующие уникального дизайна, но требующие скорости разработки. Самый частый пример — это админ-панели сайтов. Их видят только сотрудники какой-либо организации, там часто нет потребности в красоте или уникальности, нужно просто, чтобы всё работало и выглядело приемлемо. Для таких задач используют готовые библиотеки компонентов. Их уже написали другие разработчики, вам остаётся только подключить и использовать.

Самая известная и популярная такая библиотека — [Bootstrap](https://getbootstrap.com/). К ней очень спорное отношение в сообществе, но это явный лидер по узнаваемости. Работает очень просто: подключаем файл CSS и cкрипт, заходим в раздел с компонентами, копируем HTML-код, всё работает, можно уделять внимание только контенту. Возможности достаточно богатые. Можно пользоваться сеткой, компонентами управления лейаутом, а можно просто брать компоненты поменьше. Вот короткое видео с примером подключения и использования компонента карточки.

Подобных библиотек много и работают они по похожему принципу. Вот ещё несколько, попроще, чем Bootstrap:

* [MUI](https://www.muicss.com/)
* [Skeleton](http://getskeleton.com/)
* [Semantic UI](https://semantic-ui.com/)

С любой внешней библиотекой компонентов возникают со временем одни и те же проблемы: вам в какой-то момент захочется изменить дизайн под себя. В этом случае нужно будет писать стили поверх стилей библиотеки, а это не всегда просто.

**Проблемы и тренды**

Компонентный подход и взгляд на интерфейс преобладает в современном вебе. CSS и браузеры адаптируются под веяния медленнее, чем рассчитывают многие разработчики. Поэтому у компонентного подхода есть «врождённые» проблемы, которые постепенно разрешаются.

Основная проблема при строительстве компонентов — уровни переопределения стилей. Подключив библиотеку с компонентами, даже созданную своими руками, разработчик попадает в ситуацию, когда дополнительная стилизация поверх готовых компонентов усложнена. Стили бывают написаны так, что сложно перезаписать их. Подобную проблему видят и пытаются решить новым стандартом [Cascade Layers](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Cascade_layers). Это совсем новая спецификация, которую только начинают использовать в браузерах. Через неё разработчики смогут вмешиваться в то, как браузер приоритизирует применение стилей из разных файлов. Спецификация непростая для новичков, но со временем появится много примеров использования в практике.

Вторая проблема — привязанность к контексту размера окна браузера. До последнего времени менять стили в компонентах можно было, только опираясь на размеры окна браузера, не на размеры блока, создающего контекст для компонента. Мы могли сказать «Когда окно браузера станет шире, чем 450 пикселей, поменяй направление флекс-контейнера у компонента». Это не соответствует идеи независимости от окружения. Ведь компонент ведёт себя определённым образом, опираясь на контекст браузера, а не тот, в который оказался помещён. Было бы здорово уметь говорить компоненту «Когда твой родитель станет меньше 600 пикселей в ширину, поменяй стили». Теперь так можно. Есть спецификация [Container Queries](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Container_Queries). С её помощью можно будет управлять стилями, опираясь на контекст, куда встроен компонент.

Обе эти спецификации очень молодые, пока не годятся для массового использования. Нужно проверить их временем, но это революционные инструменты CSS, связанные с компонентным подходом.

За пределами CSS, в мире JavaScript есть ещё одна технология, которая очень сильно связана с компонентным подходом — [Custom Elements](https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements). Это способ создавать свои элементы без каких-либо дополнительных инструментов (вроде React или Vue.js). С развитием и популяризацией этой технологии, компонентный подход станет возможным в HTML без шаблонизаторов и сборщиков. Разработчики смогут создавать свои теги-компоненты. Но пока готовность к повседневному использованию этой технологии вызывает вопросы. К тому же порог входа в разработку собственных компонентов достаточно высок.

**Зачем была нужна эта статья**

Вёрстка — это не только написание CSS и HTML, это система представлений об интерфейсе. Важно понимать, какой взгляд на интерфейс задаёт тон технологиям, полезно знать вектор, куда технологии развиваются. В этой статье мы разбирали компонентный подход. Мышление о компонентах — основа картины мира верстальщика. Тут, возможно, был перебор для новичка, некоторые разговоры забегали вперёд, за горизонт вёрстки. Тем не менее мы думаем, что нарисовать образ, показать цель так же важно, как сделать приятным путь. Надеемся, вы расширили кругозор, и это позволит взглянуть на макеты по-новому. Мы ещё будем возвращаться к компонентному подходу, в статье много отсылок к БЭМу, Atomic Design, элементам инфраструктуры. Эта статья очерчивает контекст для следующих статей, где будем разбирать более жёсткие принципы и внедрять непростые на первый взгляд инструменты.