Битва архитектур

 Максим Усачев

Перевод статьи Battle of the architectures с сайта mariano.io, опубликовано на css-live.ru с разрешения автора — Мариано Мигеля.

Сравнение двух популярных подходов в CSS, и чем они меня не устраивают

В последнее время в мире фронтенда много спорили о том, что такое «хороший CSS» и как нам проектировать код, чтобы он был поддерживаемым, масштабируемым, доступным и быстрым. И здесь я вижу две крайности:

1. **С одной стороны, есть «семантический» подход.** Те, кто считает, что код должен быть как можно проще, утверждая, что методологии вроде БЭМ и OOCSS недостаточно хороши, поскольку не уделяют должного внимания семантике и доступности.
2. **С другой стороны, есть «атомный/функциональный» подход.** Это те, кто возводит принцип единственной обязанности в абсолют, и штампуют сверхспециализированные служебные классы направо и налево, полностью наплевав при этом, за редкими исключениями, на читаемость и самодокументируемость кода.

Лично я думаю, что обе эти крайности, как бы это сказать, слегка хватают через край, и что ответ лежит где-то посередине.

Оговорка: впереди длинная статья! Если в какой-то момент покажется, что вы уже читали это раньше, посмотрите раздел «Использованные материалы» в конце статьи. Очень может быть, что так и есть.

О чём стоит помнить при написании CSS?

Прежде чем начать бросаться обвинениями, давайте определим основной критерий. Лично я уверен, что почти всегда можно оценить, хорош ли CSS или не очень, вот по каким показателям:

1. **Насколько код читаемый и самодокументированый:** если новичок в вашей команде взглянул на код, сколько времени ему/ей потребуется, чтобы понять для чего служит каждый элемент и что их связывает
2. **Насколько легко его поддерживать:** поддержка плохого CSS может превратиться в сущий кошмар. И напротив, хорошо продуманная архитектура CSS позволяет вносить изменения и новые функции, ничего не поломав (чаще всего).
3. **Насколько он масштабируемый:** созданные вами паттерны и компоненты должны быть достаточно абстрактными, чтобы использовать их повторно во многих случаях и для разных целей.

Например, понимает ли ваш код разницу между паттернами содержимого и отображения. Если нет, то он должен.

4. **Размер файла (после сжатия).** С размером файла всё ясно: чем меньше — тем лучше. Но стоит отметить, что из-за особенностей GZIP меньший из двух несжатых минифицированных файлов после сжатия может оказаться бОльшим.

Проблемы «семантического» подхода

Общая идея «семантического» подхода — не использовать классы без крайней необходимости. Хотя это и предполагает случаи, когда классы вполне оправданы, защитники этого подхода утверждают, что нужно всегда начинать с «семантики в первую очередь» (без классов). Давайте рассмотрим подводные камни этого подхода:

Недостаточно выразительный

В сегодняшнем HTML нет другого способа выделить столько разных пространств имен, как с подходом на основе названий классов вроде БЭМ. Вам не хватит тегов и атрибутов для всех объектов, компонентов и их вариантов — разве что если вы работаете над маленьким проектом (и не планируете его расширять). И когда все возможные комбинации закончатся, вам всё равно придётся воспользоваться классами, внося разнобой (почему у этого элемента есть класс, а у того нет?)

И пусть даже для всего нашлись комбинации, код вскоре всё равно пришлось бы усложнять (т.е. лишняя вложенность), делая поддержку вашего кода сущим геморроем, и всё это лишь чтобы обойтись без классов? Нате вам.

Поверьте, это СТАНЕТ сложным

Скажем, я хотел оформить все мои «текстообразные» input-ы с помощью семантического подхода:

```
[type="email"], [type="number"], [type="password"],
[type="search"], [type="text"], [type="tel"], [type="url"] { }
```

Отлично. Вышло семь селекторов и мне придется добавлять селектор каждый раз, если HTML вдруг введет очередной текстообразный input.

Гореть мне за такие слова в аду, но это можно упростить, перейдя на темную сторону:

```
.o-input-text { }
```

Жуть, правда?

Можно по-прежнему сделать HTML доступным

HTML без вариантов надо делать семантичным и доступным. Можно по-прежнему добавлять ARIA-роли и микроформаты, а также использовать

осмысленную разметку. Просто не ссылайтесь на них прямо в CSS. Понятно, что (плохие) разработчики скорее всего получат такой же визуальный дизайн и с какой угодно разметкой. **Быть плохим разработчиком недопустимо. Возможность делать глупости не дает права их делать.**

Никого не волнует

Ну, почти никого. Браузерам по барабану, используете ли вы классы (или нет), пользователям тоже. А вот новичку в вашей команде это наверняка важно. Насколько вы упростили ему жизнь?

Проблемы с «функциональным» подходом

А теперь зайдём с другой стороны: атомный/функциональный подход. Основная идея этого подхода — легко повторяемые классы, которые отвечают за что-то одно в стилях (по сути они похожи на встроенные стили) и у которых низкая специфичность, позволяющая легко переопределять эти классы. Преимущества есть: маленький, масштабируемый и простой CSS, плюс лёгкий способ экспериментировать с раскладкой даже не затрагивая CSS-файл.

Признаюсь, поначалу я прямо восхитился этим подходом. Поверьте, между этим подходом и «семантическим» я бы однозначно выбрал «функциональный». Но когда я более-менее успешно применил его на паре проектов, у него тоже нашлись свои изъяны:

Совершенно нечитаемый и непонятный код

С функциональным/атомным CSS не редкость видеть что-то вроде этого:

```
<div class="bg-white box-shadow pad-1 marg-1 fl-left">
</div>
```

Если я скажу вам, что это компонент карточки, смогли бы вы догадаться об этом сами, глядя только на код? Если вы не имеете отношение к этому коду, ответ скорее всего будет «нет».

И код не расскажет вам, какие классы критически важны для компонента, а какие необязательны или ситуативны. Всем ли карточкам требуется класс pad-1? А что насчёт fl-left?

А вот как мы могли бы сделать:

```
<div class="c-card u-float-left">
</div>
```

Здесь применяется синтаксис в стиле БЭМ вместе с понятными пространствами имён (в частности .c- для компонентов, а .u- для служебных классов). Теперь код говорит нам, что этот кусок DOM отвечает за компонент карточки, а также информирует, какие его части необязательны (.u-float-left). В вашем CSS появилось бы что-то вроде этого:

```
.c-card {
background: $white;
margin: $spacing-unit;
padding: $spacing-unit;
box-shadow: $box-shadow;
}
```

Просто, очевидно и последовательно. Что подводит к следующему моему пункту.

Слишком много места для несоответствий

Поскольку можно использовать какие угодно комбинации классов в любом месте, разработчику ничего не стоит как напортачить с чем-то, так и мимоходом сделать из вашего исходного компонента что-то совсем хитрое.

```
<!— Это исходная “карточка" —>
<div class="bg-white box-shadow pad-1 marg-1 fl-left">
</div>

<!— Кто-то забыл добавить box-shadow к вашей карточке и теперь она совсем не похожа на карточку. —>
<div class="bg-white pad-1 marg-1 fl-left">
</div>

<!— Разве не замечательно смотрелась бы эта карточка с чёрным фоном? —>
<div class="bg-black text-white box-shadow pad-1 marg-1 fl-left">
</div>
```

Ладно, может, с последним примером я слегка и перегнул, но суть вы поняли: чем больше классов, тем легче с чем-то напортачить.

Более БЭМ-подобный подход поможет не только сохранить ваши интерфейсы в порядке, давая возможность описывать новые варианты компонентов на более высоком уровне (поскольку пришлось бы заводить свой класс для каждого), но еще и избежать случайных ошибок, ведь компонентам уже не нужно столько классов.

Нет особой выгоды в размере файла

Чтобы файлы не разрастались, Атомный/Функциональный CSS велит нам избегать многократного определения одинаковых стилей для типовых свойств вроде margin или padding. Вместо этого для каждого свойства заводится один служебный класс, что избавляет от самоповторов и дает компактный CSS на выходе

На самом деле мы можем задавать у себя в CSS одни и те же стили для типичных свойств хоть тыщу раз, если только все они хранятся у нас в

одном месте (а не вводятся каждый раз вручную). Вот зачем избавляться от самоповторов: ради единого источника достоверных данных.

Насчёт остального не парьтесь: GZIP уберёт все повторяющиеся кусочки из скомпилированного CSS. Если хотите подробнее узнать о GZIP, посмотрите это видео от Кевина Хоу и Эрика Хиггинса.

Переносит поддержку в HTML

Если только вы не генерируете разметку динамически **очень продвинутым** движком, видно, как необходимость перелопатить все ваши шаблоны, чтобы всего лишь заменить везде .pad-1 на .pad-2, может стать сущим геморроем. А если это нужно на нескольких сайтах сразу, то вообще кошмар неминуем.

Так что же вы предлагаете?

Скоро намечается ещё одна длинная статья, где я подробно расскажу про собственный подход к написанию масштабируемого и поддерживаемого CSS для сайтов и приложений. А пока, рекомендую прочитать эту статью товарищей из BBC о том, как они переосмыслили их архитектуру CSS. Ребята, вы большие молодцы!

Использованные материалы

- «БЭМантика: пишите осмысленные стили без повторов» Мэтта Стоу
- «Может ли CSS быть слишком модульным?» Гарри Робертса
- «Более прозрачный код UI с пространствами имён» Гарри Робертса
- «CSS для большой команды» Хидде де Вриса