

Тема 3

«Тестирование по методу белого ящика»

Вспомним определение

Метод белого ящика (white-box testing, glass-box testing) – используется для тестирования программного кода **без запуска**.

Тестировщик **имеет доступ к исходному коду** ПС. Тесты основаны на **знании кода приложения и его внутренних механизмов**.

Метод белого ящика часто используется на стадии, **когда приложение ещё не собрано воедино**, но необходимо проверить каждый из его компонентов, модулей, процедур и подпрограмм.



Инструментальные средства тестирования веб-ориентированных приложений по методу белого ящика

Что можно тестировать...

В веб-ориентированном приложении по методу белого ящика можно тестировать:

1. Код:

- 1. HTML**
- 2. CSS**
- 3. JavaScript**
- 4. Код движка приложения (на том языке, на котором он написан)**

2. Базы данных и всё с ними связанное.

3. «Документацию»:

- 1. Непосредственно проектную документацию.**
- 2. Контекстную помощь.**
- 3. Тексты и надписи.**
- 4. Графические элементы.**

«Документация»

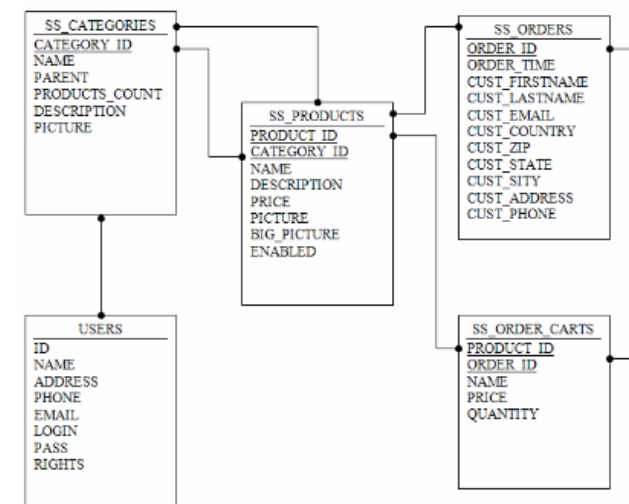
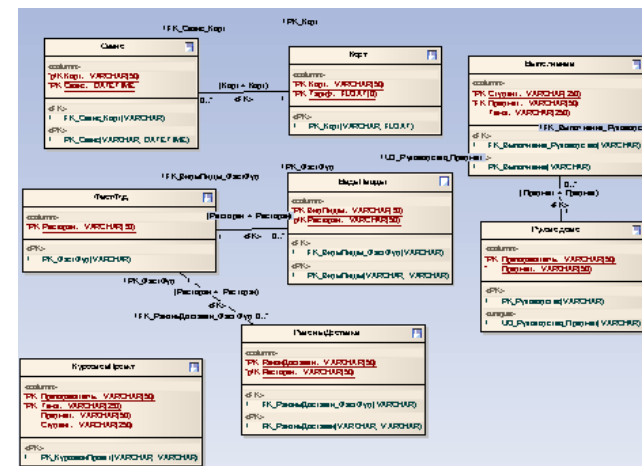
С тестированием документации всё просто:

- проверяем тексты с помощью средств проверки орфографии, синтаксиса, пунктуации (банально – можно сделать это в Word'e);
- мелкие надписи просто вычитываем;
- графические элементы проверяем на качество изображения, отсутствие повреждений, размер.


К слову, параллельно мы можем заниматься тестированием локализации, если проверяем конкретную языковую версию.



Тестирование баз данных (БД) по методу белого ящика сводится к анализу моделей БД и рассмотрению возможных сложностей, возникающих при использовании конкретной модели БД в тех или иных условиях.



Пример простого тестирования БД по методу белого ящика: итак, перед нами таблица; что здесь «не так»?

Persons 	
«column»	
*PK	<u>first_name</u> : VARCHAR(10) second_name: BIGINT birth_date: DATETIME
«PK»	
+	PK_Persons(VARCHAR)

Базы данных

Инструментальные средства тестирования БД по методу белого ящика **не являются некоей отдельной категорией ПО**. Для этой задачи **используются средства проектирования БД**, специфичные для каждой отдельной СУБД, или «универсальные» средства проектирования: Sparx EA, AllFusion ErWin Data Modeler и т.п.

В силу того факта, что основные проблемы с БД проявляются на стадии их эксплуатации, **дальнейшие исследования проводятся средствами динамического тестирования**.



HTML / CSS / JavaScript

HTML, CSS и JavaScript в контексте нашей дискуссии объединяет то, что:

- для их **первичного анализа** часто нет необходимости прибегать к сложным инструментальным средствам;
- они **доступны для «прямого восприятия глазами»**;
- **НО** иногда даже наличие хороших инструментальных средств не может помочь, если у тестировщика не хватает опыта и знаний в области тестирования совместимости или понимания стандартов оформления кода.



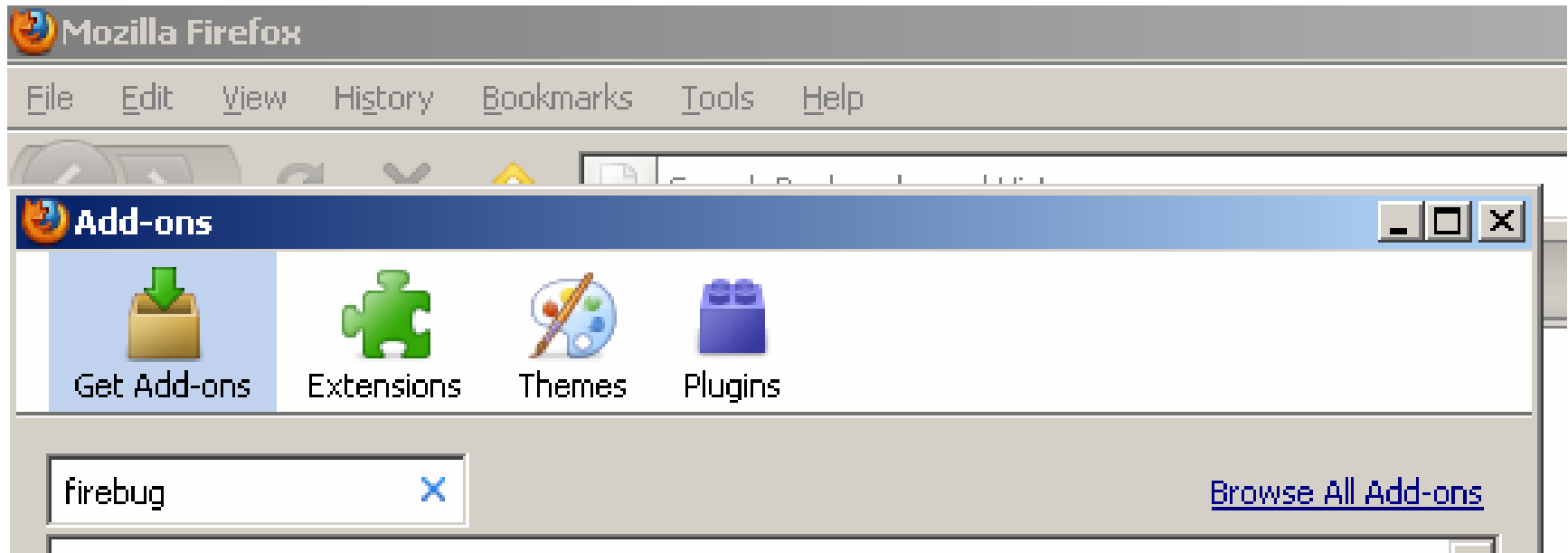
FireBug

В качестве основного средства быстрой проверки того, как работает веб-ориентированное приложение в браузере, можно использовать **расширение браузера Firefox – FireBug**.



FireBug – установка

FireBug легко устанавливается стандартными средствами Firefox по поиску и установке расширений:



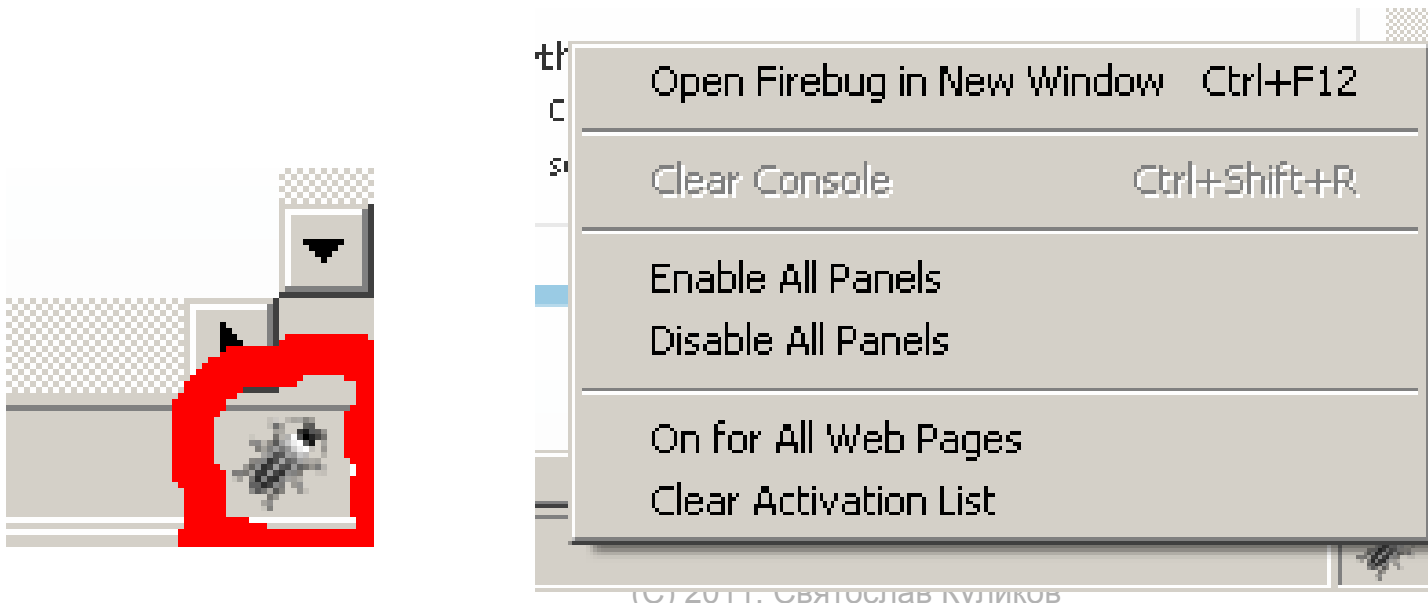
FireBug – установка

Другой способ – зайти на официальный сайт разработчиков (<http://getfirebug.com>) этого расширения, где всегда можно и скачать сам FireBug, и **узнать много интересного**:



FireBug – запуск

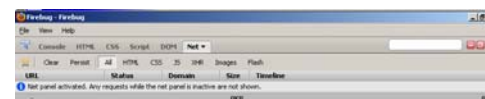
После успешной инсталляции в правом нижнем углу окна Firefox появится характерная иконка. Можно кликнуть по ней левой кнопкой мыши для запуска FireBug в «состоянии по умолчанию» или кликнуть правой кнопкой для выбора некоторых опций. Также можно просто нажать **F12**.



FireBug – режимы работы

FireBug может находиться в:

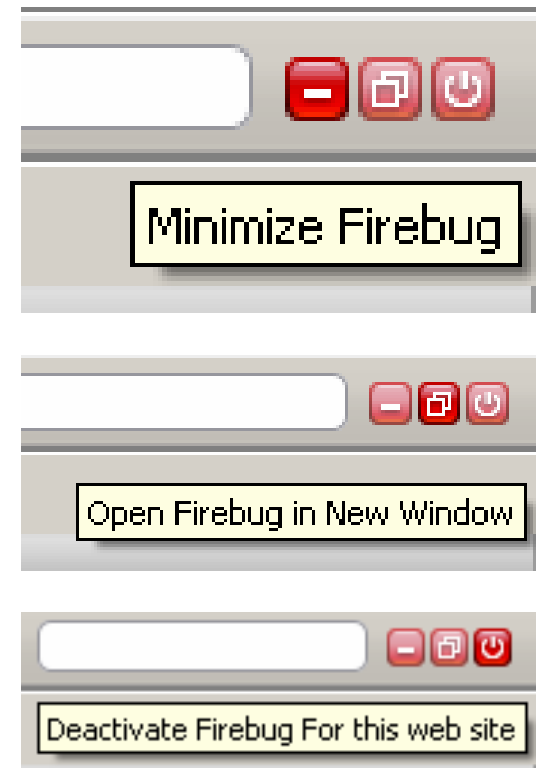
- развёрнутом состоянии;
- свёрнутом состоянии (продолжая работать);
- в виде отдельного окна.



FireBug – режимы работы

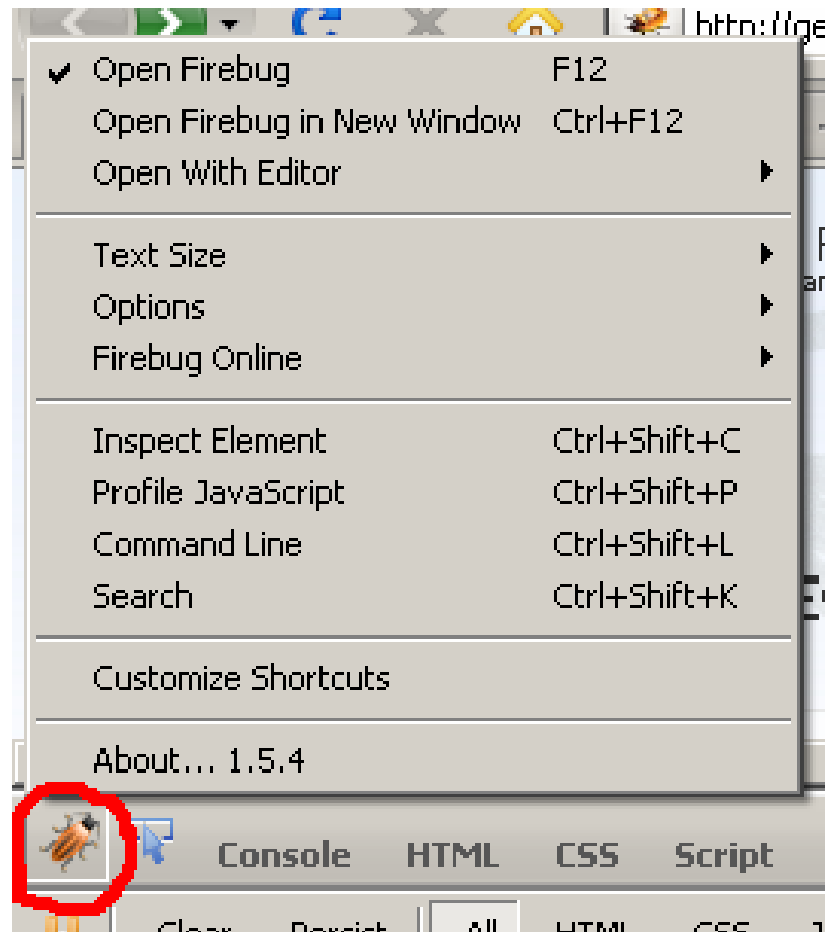
Для запуска FireBug сразу в режиме отдельного окна можно использовать комбинацию клавиш **Ctrl-F12**.

Для изменения режимов можно использовать «системные кнопки» в правом верхнем углу окна FireBug.



FireBug – режимы работы

Также много полезного можно увидеть в меню, вызываемом кликом по **иконке в левом верхнем углу** FireBug.



FireBug – основные элементы

Основные элементы FireBug отражены на главной панели инструментов.

Обратите внимание, что если рядом с названием инструмента есть «стрелочка вниз», это значит, что **этим инструментом или его поведением можно управлять**.



FireBug – консоль

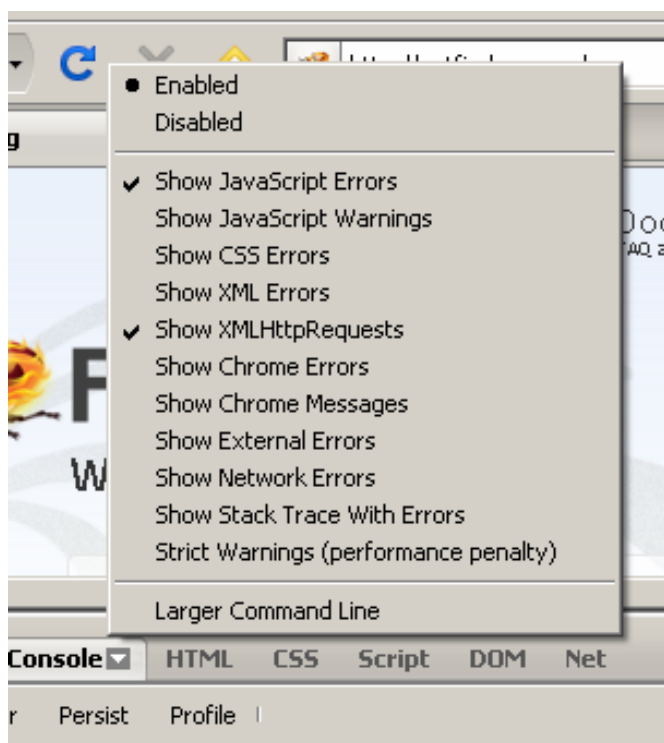
Консоль служит как для динамической отладки JavaScript (особенно **полезны такие особенности FireBug как встроенный профилировщик, возможность установки «точек остановки» (breakpoints) и пошагового выполнения**), так и для изучения сообщений обо всех проблемах, произошедших при анализе браузером страницы.

Да, это относится к динамическому тестированию, но **при анализе отдельных компонентов сайта это может помочь и при статическом тестировании по методу белого ящика.**

Рассмотрим на картинках...

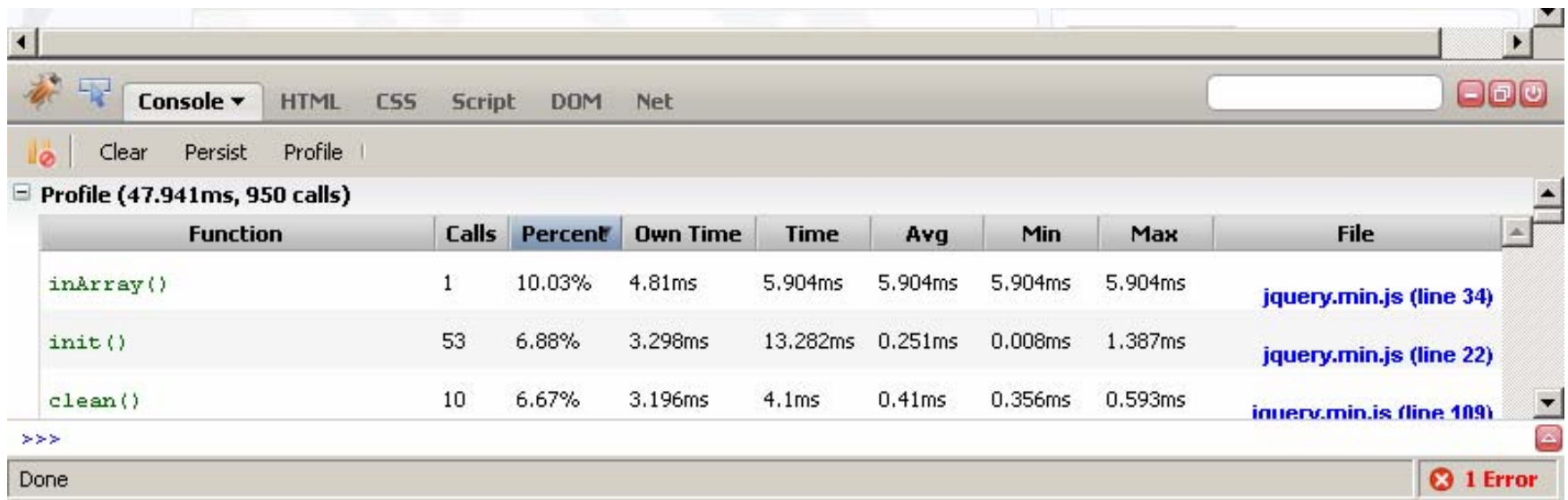
FireBug – консоль

Вы можете выбирать, какие сообщения отражать в консоли. Список сообщений достаточно обширен и включает не только сообщения, генерируемые движком JavaScript.



FireBug – консоль

Вы можете использовать профилировщик для анализа времени выполнения и частоты вызовов тех или иных JavaScript-функций.



The screenshot shows the FireBug console with the 'Console' tab selected. Below the tab bar, there are buttons for 'Clear', 'Persist', and 'Profile'. The main area displays a performance profile titled 'Profile (47.941ms, 950 calls)'. This profile contains a table with the following data:

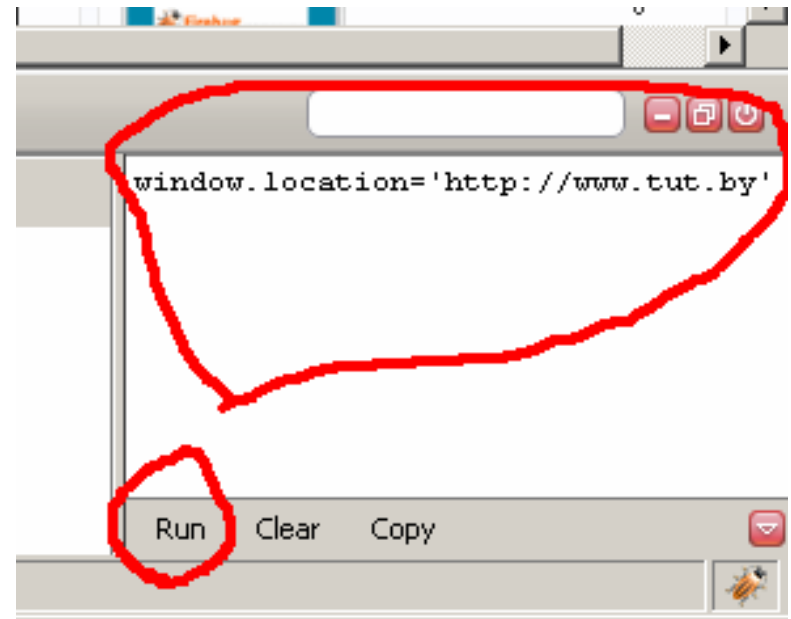
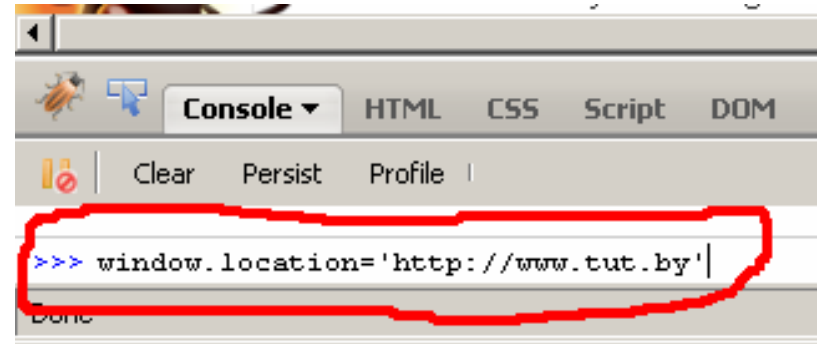
Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
<code>isArray()</code>	1	10.03%	4.81ms	5.904ms	5.904ms	5.904ms	5.904ms	jquery.min.js (line 34)
<code>init()</code>	53	6.88%	3.298ms	13.282ms	0.251ms	0.008ms	1.387ms	jquery.min.js (line 22)
<code>clean()</code>	10	6.67%	3.196ms	4.1ms	0.41ms	0.356ms	0.593ms	jquery.min.js (line 109)

At the bottom of the console, there is a status bar showing 'Done' and a red error icon with the text '1 Error'.

FireBug – консоль

Вы можете выполнять команды JavaScript как из всегда отображаемой «маленькой» командной строки...

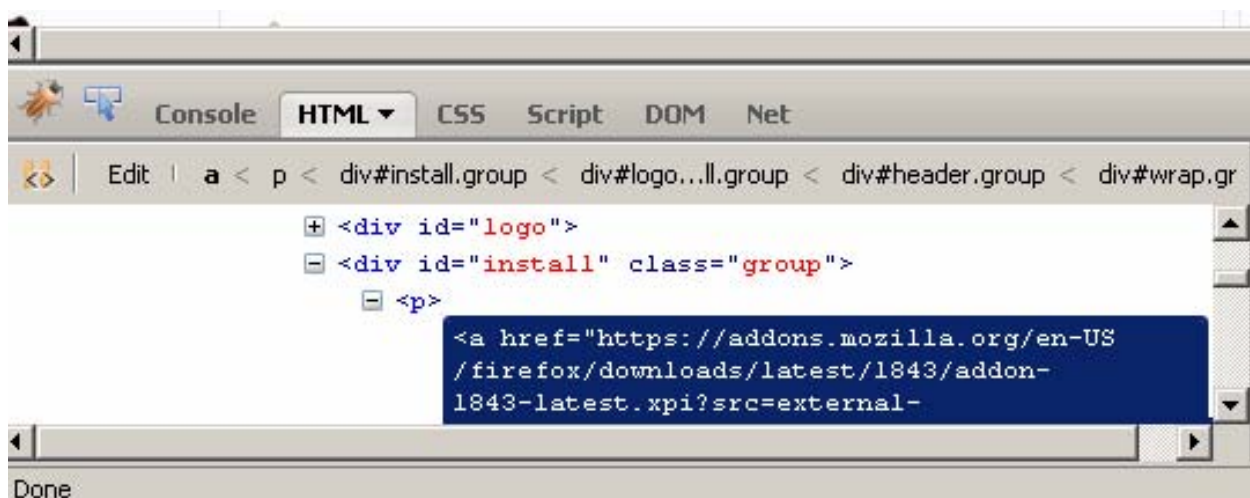
...так и переключиться в полноценный редактор JavaScript.



FireBug – HTML

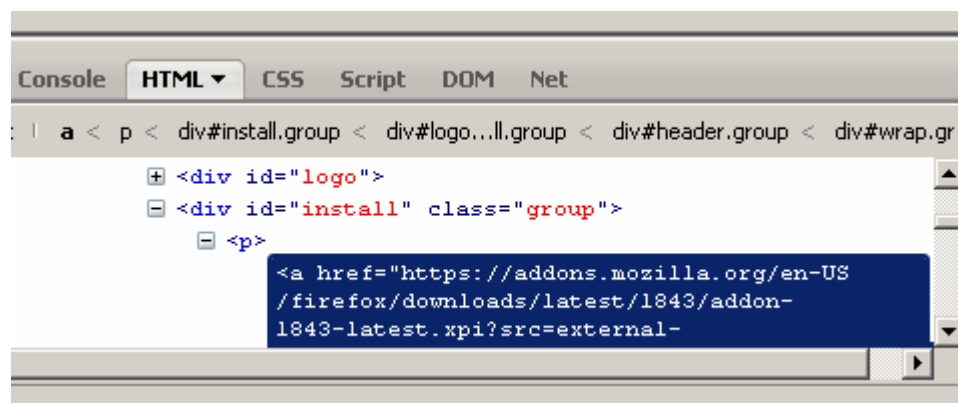
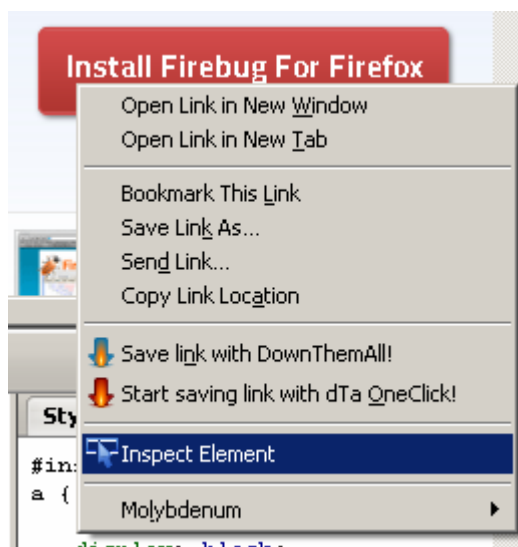
Для статического тестирования по методу белого ящика очень полезен инструмент «HTML».

Во-первых, **вы всегда можете увидеть красиво отформатированный HTML-код** (как бы криво его ни написал верстальщик).



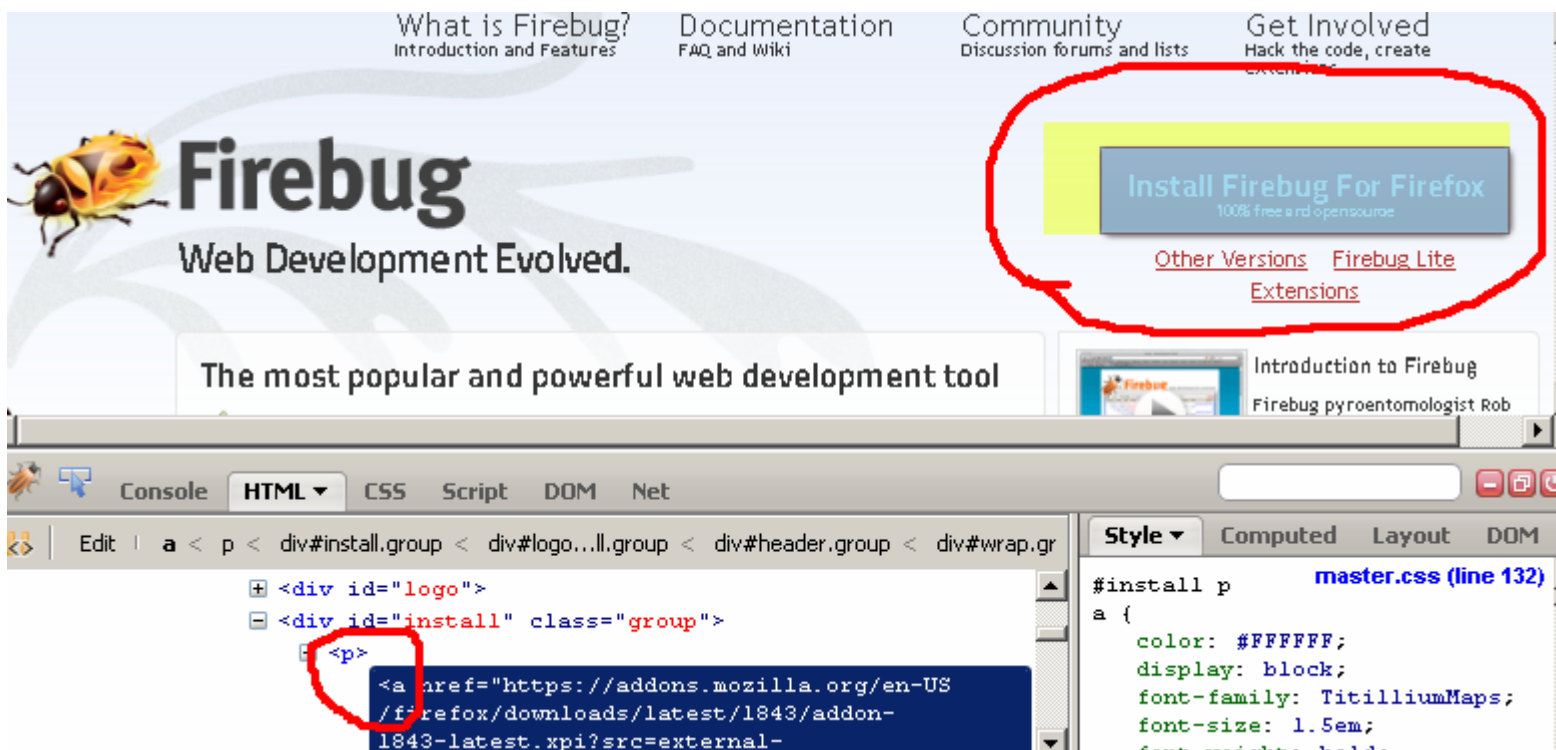
FireBug – HTML

Во-вторых, для любого элемента HTML-страницы достаточно вызвать команду «**Inspect element**» из контекстного меню, чтобы FireBug автоматически перевёл вас к коду этого элемента.



FireBug – HTML

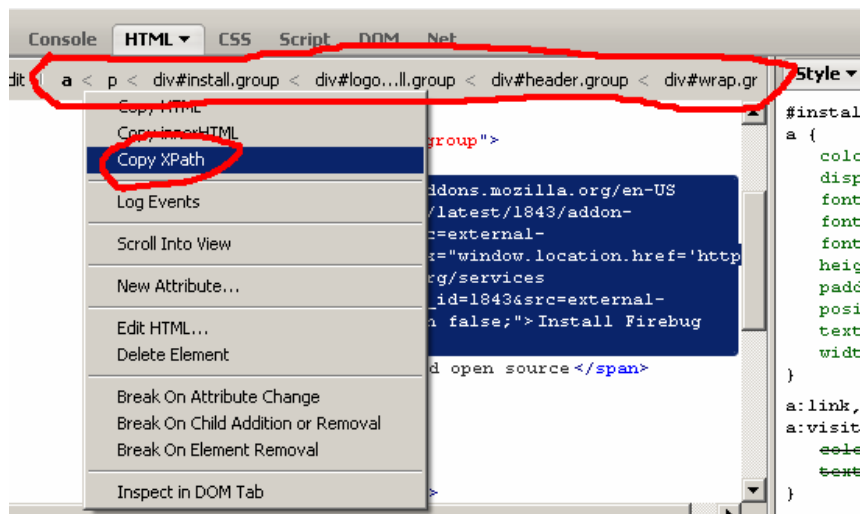
Когда вы наводите мышь на тот или иной элемент в окне исходного кода страницы, **FireBug автоматически показывает вам этот элемент на странице**: синим выделяется сам элемент (занимаемая им область), жёлтым – отступы от элемента до окружающих элементов.



FireBug – HTML

Вверху окна с исходным кодом страницы расположена панель, на которой отображается **полный XPath к анализируемому элементу и всем его родительским элементам**.

XPath можно скопировать в буфер. Это очень удобно, когда вы занимаетесь тестированием веб-ориентированного приложения с помощью Selenium или иного средства, умеющего определять элементы по XPath.



FireBug – HTML

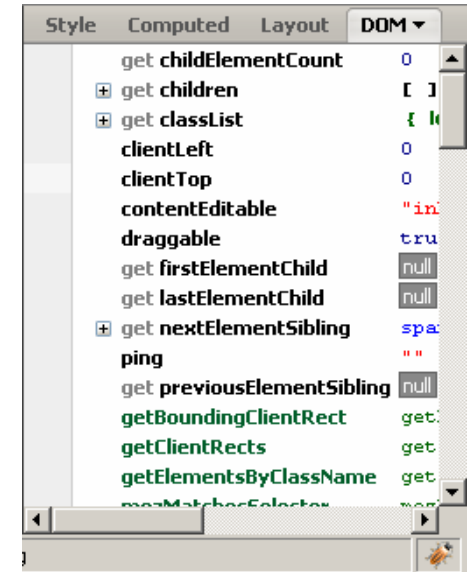
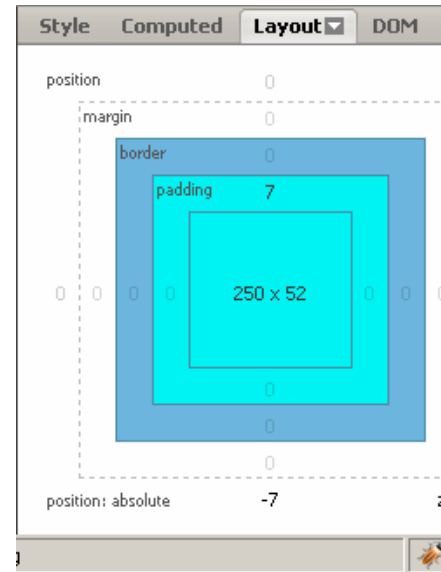
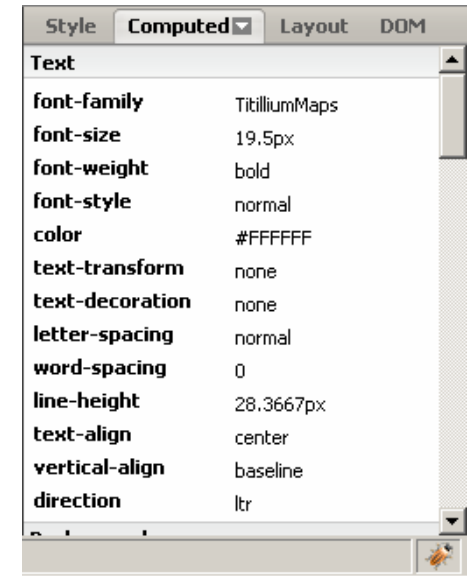
Для каждого HTML-элемента в правой части окна FireBug отображается:

1. **Список CSS-стилей**, применённых к элементу (закладка «Style»).

2. **Список свойств элемента**, вычисленных браузером на основе совокупности указанных CSS-стилей (закладка «Computed»).

3. **Вся информация о расположении элемента**, отступах и т.п. (закладка «Layout»).

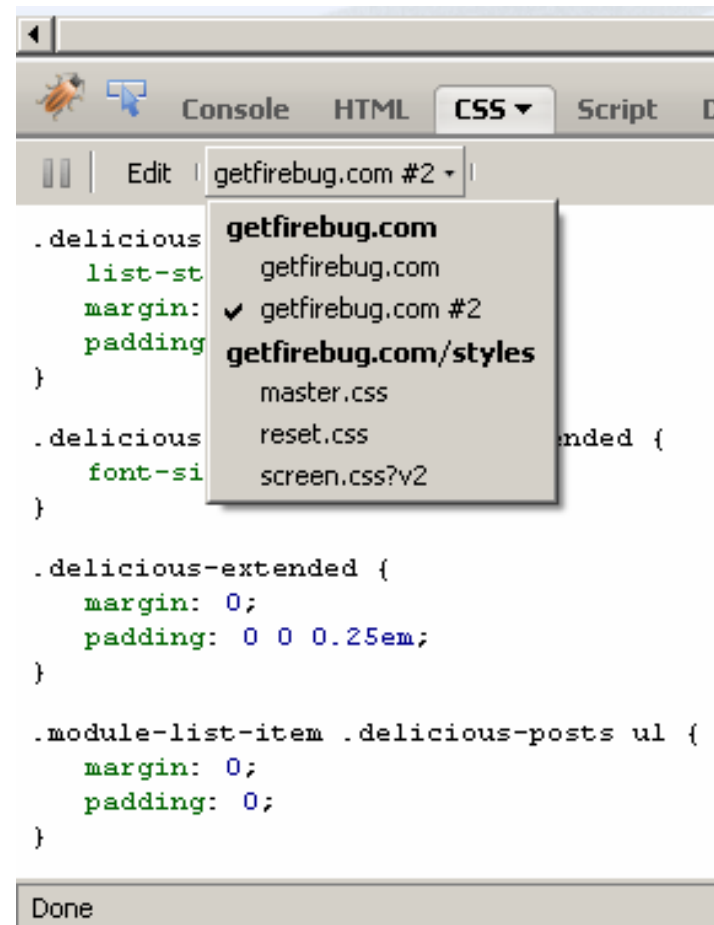
4. **Информация о DOM-свойствах элемента** (закладка «DOM» (document object model, объектная модель документа)).



FireBug – CSS

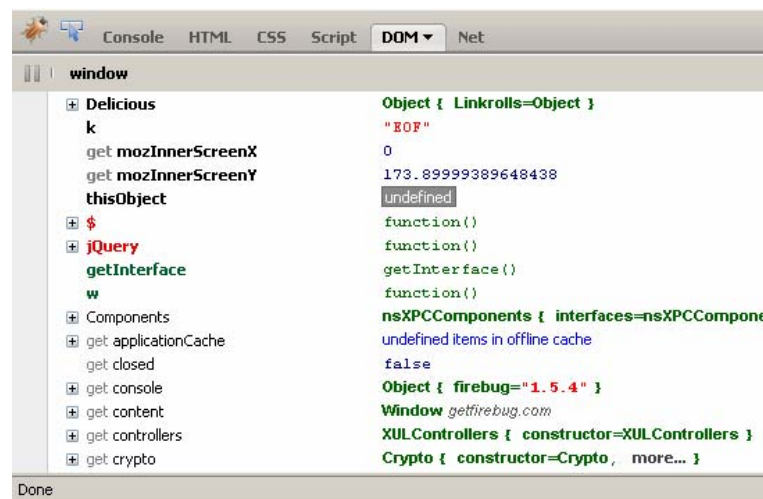
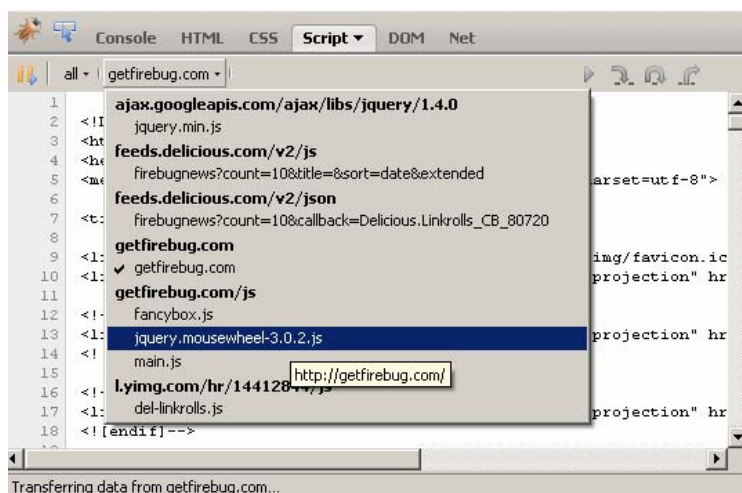
Инструмент CSS позволяет увидеть все инструкции применённые к данной странице (включая информацию о том, из какого файла взята та или иная инструкция).

Как и в случае с HTML, содержимое CSS можно редактировать, что приведёт к немедленным изменениям на странице.



FireBug – Script и DOM

Инструменты «Script» и «DOM» не относятся к тестированию по методу белого ящика, потому мы лишь скажем, что **они помогают заниматься отладкой всего того, что сделано на странице с помощью JavaScript**, а также лучше понять, почему тот или иной элемент ведёт себя так, а не иначе.



FireBug – Net

Одним из самых полезных и простых в использовании является инструмент «Net», отображающий информацию о всех запрошенных в процессе загрузки страницы файлах, всём затраченном времени (с учётом каждой операции), отосланных и принятых заголовках и т.п.

L	Status	Domain	Size	Timeline
GET getfirebug.com	200 OK	getfirebug.com	14.5 KB	1.63s
GET screen.css?v2	200 OK	getfirebug.com	198 B	563ms
GET main.js	200 OK	getfirebug.com	440 B	579ms
GET jquery.min.js	304 Not Modified	ajax.googleapis.com	68.2 KB	282ms
GET jquery.mouse	200 OK	getfirebug.com	872 B	563ms
GET fancybox.js	200 OK	getfirebug.com	20.5 KB	1.39s
GET reset.css	200 OK	getfirebug.com	696 B	312ms
GET master.css	200 OK	getfirebug.com	17.7 KB	1.49s
GET bg-grad.jpg	200 OK	getfirebug.com	517 B	78ms
GET 1281544202s	200 OK	getfirebug.com	5.2 KB	1.1s
GET screencast.jp	200 OK	getfirebug.com	27 KB	953ms
GET moogaloop.sv	200 OK	vimeo.com	201 B	609ms
GET Intro2FB.mp4	200 OK	getfirebug.com	2.5 KB	406ms
GET header-bg.pn	200 OK	getfirebug.com	187.7 KB	1.2s
GET firebug-logo.p	200 OK	getfirebug.com	17.6 KB	1.2s

+2.05s Started

0 0 DNS Lookup

0 0 Connecting

0 0 Sending

0 312ms Waiting

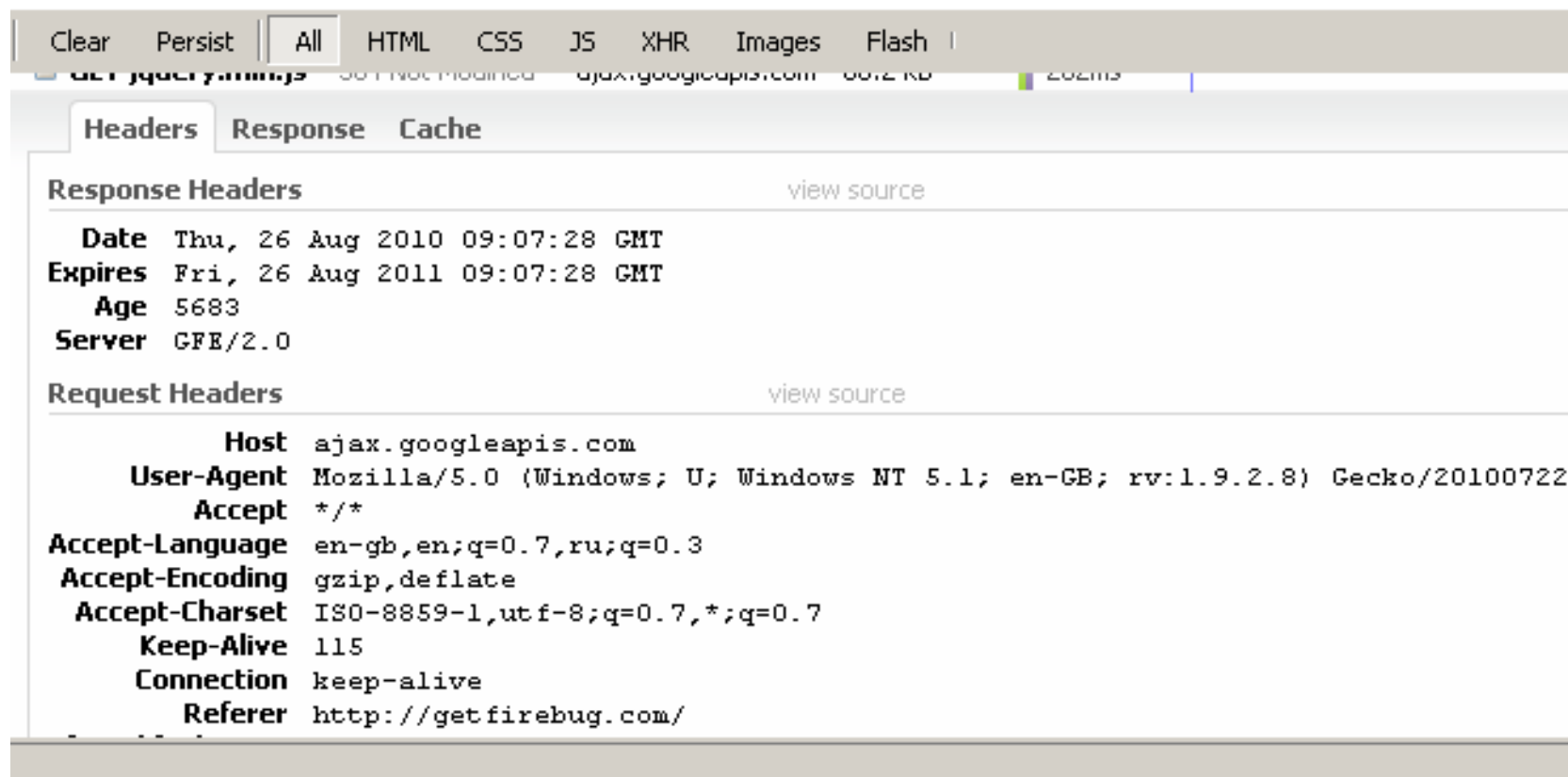
+312ms 0 Receiving

+3.63s 'DOMContentLoaded' (event)

+16.04s 'load' (event)

FireBug – Net

Анализ **заголовков запросов/ответов**, конечно, куда более полезен при динамическом тестировании, но эта информация в любом случае не может считаться лишней.



The screenshot shows the FireBug Net panel with the 'All' tab selected. The 'Response Headers' section is expanded, showing the following information:

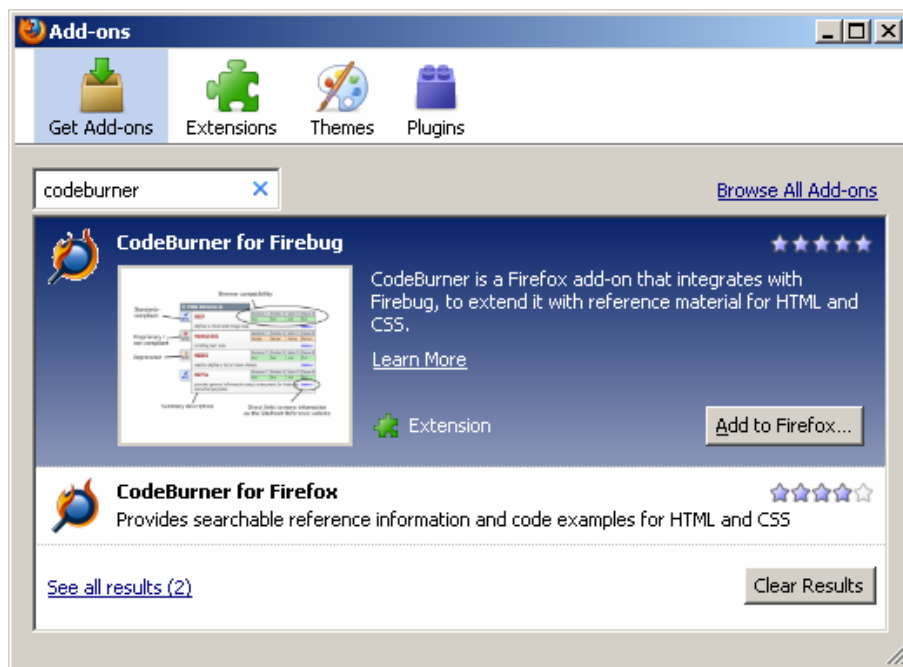
- Date:** Thu, 26 Aug 2010 09:07:28 GMT
- Expires:** Fri, 26 Aug 2011 09:07:28 GMT
- Age:** 5683
- Server:** GFE/2.0

Below the response headers, the 'Request Headers' section is also expanded, showing the following information:

- Host:** ajax.googleapis.com
- User-Agent:** Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.9.2.8) Gecko/20100722
- Accept:** */*
- Accept-Language:** en-gb,en;q=0.7,ru;q=0.3
- Accept-Encoding:** gzip,deflate
- Accept-Charset:** ISO-8859-1,utf-8;q=0.7,*;q=0.7
- Keep-Alive:** 115
- Connection:** keep-alive
- Referer:** http://getfirebug.com/

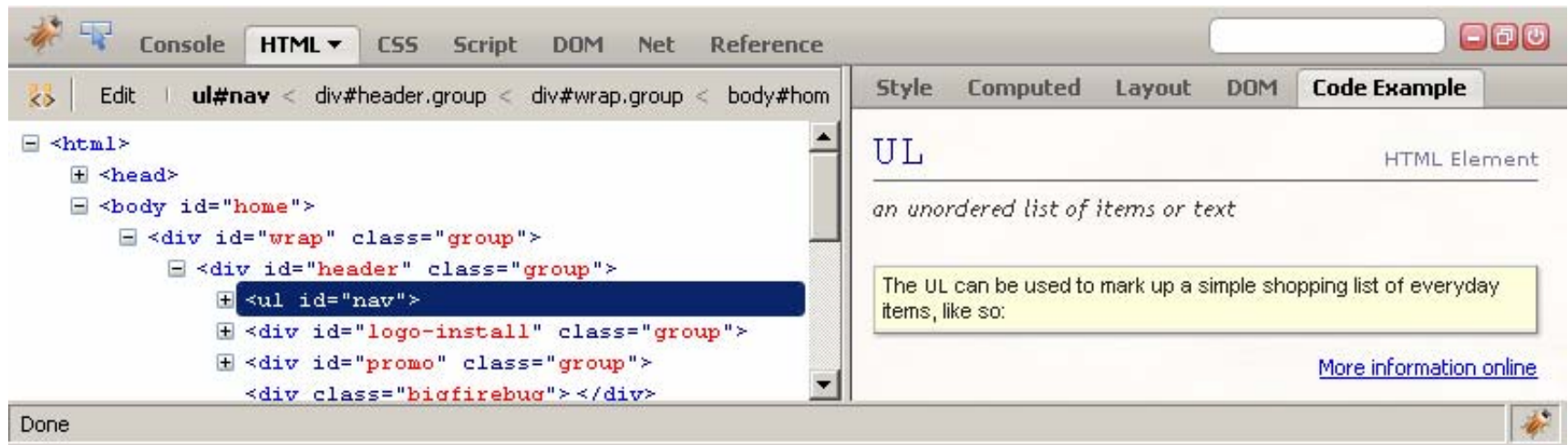
CodeBurner for FireBug/Firefox

О самом FireBug на этом – всё. Однако, к FireBug есть специальное расширение – **CodeBurner** – идеально приспособленное как раз для статического тестирования. Оно позволяет тем, кто не очень хорошо знаком с HTML/CSS, получить больше информации.



CodeBurner for FireBug


После установки «CodeBurner» вы увидите дополнительную закладку **«Code example»** в правом окне инструмента «HTML». Здесь будет присутствовать краткое описание выбранного элемента, пример его использования и ссылка на полноценное описание в стандартах.




CodeBurner for FireFox

Практически одноимённое расширение, работающее непосредственно с браузером («CodeBurner for FireFox») позволяет проводить такой же анализ без запуска FireBug.

По каждому анализируемому элементу также выводится информация о его совместимости с некоторыми браузерами.

 Attributes defined for this element
(1)

 W3C	<code>class</code>	Explorer 7	Firefox 3	Safari 3	Opera 9
		Full	Full	Full	Full

classifies this element into one or more subtypes (all elements) [more...](#)

Промежуточное заключение

Конечно, набор средств для тестирования веб-ориентированных приложений по методу белого ящика не ограничивается двумя только что рассмотренными.

Их много, но они — **узкоспецифичны** и не являются широкоизвестными.

Однако знание технологий, с помощью которых разработано приложение, всегда позволяет подобрать необходимый набор инструментальных средств.

Например, простые и удобные анализаторы кода на наличие в нём недопустимых символов или их последовательностей, лишних пробелов и т.п. вполне можно написать самостоятельно.

Тест для проверки изученного

1. Дайте определение тестирования по методу белого ящика.
2. Какие инструментальные средства тестирования веб-ориентированных приложений по методу белого ящика вы знаете?
3. Какие возможности предоставляет «Firebug» (расширение браузера «Firefox»)?
4. Как можно организовать тестирование по методу белого ящика базы данных?
5. Представим ситуацию: у нас есть доступ только к клиентской части веб-ориентированного приложения (т.е. тому, что можно запросить через браузер): что мы можем проверить методом белого ящика?
6. Можно ли автоматизировать тестирование по методу белого ящика?
7. Допустим, некоторый код является полностью корректным с точки зрения функционального тестирования (он надёжно выполняет то, что от него требуется). Означает ли это, что данный код можно считать качественным? Если да, то почему? Если нет, то в чём проблема?
8. Какие потенциальные проблемы может вызвать «чрезмерная оптимизация» базы данных в направлении уменьшения занимаемого записями объёма памяти?
9. Программист написал очень краткий код, использующий недокументированные особенности среды функционирования приложения. В чём здесь проблема?
10. Изучив код веб-ориентированного приложения, мы видим, что часть отладочных сообщений не может быть отключена, т.е. такие сообщения могут появиться в процессе работы приложения с конечными пользователями. Какие потенциальные угрозы несёт данная ситуация?

Валидаторы кода

Валидация HTML/CSS

Валидация – это проверка. Если точнее, несколько проверок:

1. Валидация синтаксиса – проверка на наличие синтаксических ошибок.

2. Проверка вложенности тегов – тэги должны быть закрыты в обратном порядке относительно их открытия.

3. Валидация DTD (document type definition) – проверка соответствия кода указанному Document Type Definition. Она включает проверку названий тэгов, атрибутов, и «встраивания» тэгов (тэги одного типа внутри тэгов другого типа).

4. Проверка на посторонние элементы – выявляет всё, что есть в коде, но отсутствует в DTD. Например, пользовательские теги и атрибуты.

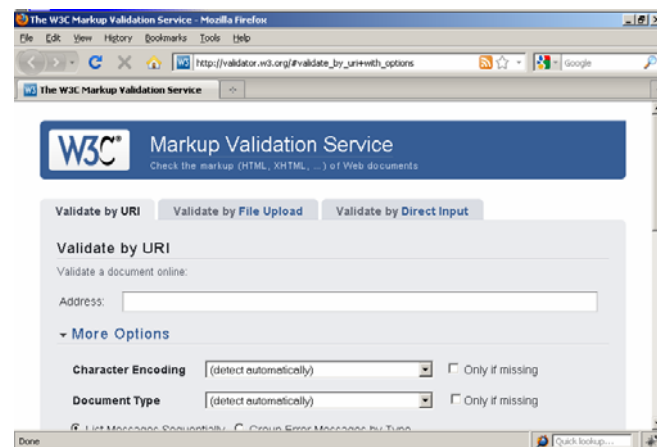
См.: <http://habrahabr.ru/blogs/webdev/101985/>

Валидаторы

Соответственно, **валидатор** – программа (или совокупность программ), выполняющая все или некоторые из только что перечисленных проверок.

В общем случае существует **три типа валидаторов**:

1. Встроенные в среды проектирования и разработки (например, в DreamWeaver).
2. Отдельное ПО (например, «CSE validator»).
3. Он-лайн (например, <http://validator.w3.org>)



Валидация – за и против

ЗА: обеспечение кроссбраузерности и совместимости со всевозможным ПО по анализу кода.



ПРОТИВ: валидация слишком строга и не соответствует тому, как на самом деле работают браузеры. Да, HTML может быть невалидным, но все браузеры могут обрабатывать некоторый невалидный код одинаково.



Валидация – за и против

Есть несколько важных вещей, которых не гарантирует валидный HTML:

- валидный HTML/CSS **не гарантирует accessibility** (читаемость и понятность кода и самого приложения, доступность); существует он-лайн тест доступности: <http://www.contentquality.com>;
- валидный HTML/CSS **не гарантирует хороший UX** (user experience, пользовательский опыт – термин, близкий к определению «удобства использования» как «меры пользовательского опыта»);
- валидный HTML/CSS **не гарантирует функциональность** сайта;
- валидный HTML/CSS **не гарантирует корректное отображение** сайта.

Валидация – за и против

Тогда, если есть столько «не гарантирует» и «против» – зачем этим заниматься?

Поясним на примере. Если вы пишете рассказ, то идеальные синтаксис, пунктуация, орфография, грамматика не гарантируют, что этот рассказ будет интересным, что он кому-то понравится, что его захочет кто-то читать.

Но даже самый потрясающий рассказ, написанный с кучей ошибок, оставит «неприятный осадок».

Валидация кода позволяет избежать этого «неприятного осадка».

Валидация гарантирует, что хотя бы на уровне примитивных правил, положенных в основу (X)HTML/CSS, с кодом всё «более менее в порядке».



Валидаторы – где взять

Элементарные запросы в Google «HTML validation», «XHTML validation», «CSS validation» выдают много полезных ссылок. Поскольку бесплатными являются в основном он-лайн валидаторы, приведём пару примеров:

1. HTML, XHTML

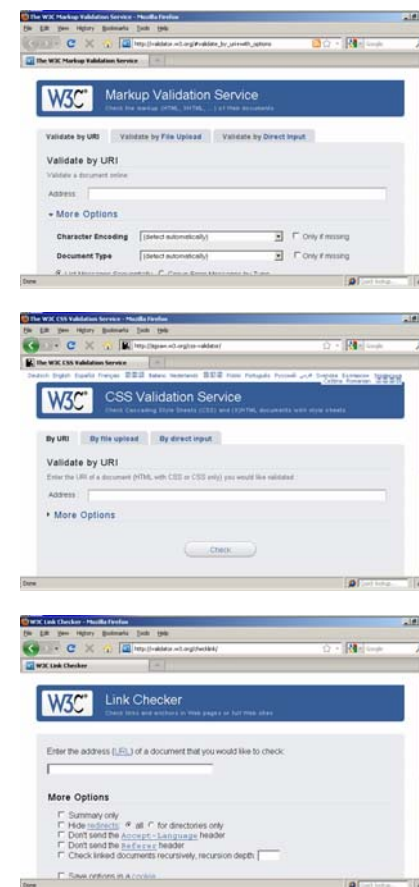
<http://validator.w3.org>

2. CSS

<http://jigsaw.w3.org/css-validator>

3. Проверка ссылок

<http://validator.w3.org/checklink>



Пример работы валидатора

Рассмотрим пример (забавный 😊). Итак, валидация `http://www.google.com` с точки зрения HTML5.

Как видите, не всё и не всегда так хорошо, как кажется.

Errors found while checking this document as HTML5!	
Result:	35 Errors, 2 warning(s)
Address :	<input type="text" value="http://www.google.com/"/>

Validation Output: 35 Errors

⚠ Using windows-1252 instead of the declared encoding iso-8859-1.

✖ Line 5, Column 2037: The bgcolor attribute on the body element is obsolete. Use CSS instead.

... () .src='/images/srpr/nav_logo14.png'" ><textarea id=csi style=display:none></t...

А теперь – о ссылках

Мы уже упомянули вскользь, что существует достаточное количество ПО, способного проверить корректность ссылок в веб-ориентированном приложении.

Однако хочется дать **несколько рекомендаций**, которые помогут **как разработчикам, так и тестировщикам**.

Итак...



Проверка ссылок

Когда разговор заходит о **проверке ссылок**, часто можно слышать слова в стиле «Да, пару раз проверим в процессе, потом ещё разок перед сдачей – и всё».

Так вот – не всё.

Да, бесспорно, проверять состояние ссылок нужно почаще ещё в процессе разработки приложения: всё же **«весь веб» базируется на ссылках**, так что не стоит их недооценивать.

Сейчас мы поговорим о том, когда, зачем и **как следует проверять ссылки**.



Написали – проверьте!

0. Написали – проверьте!

Да, именно **под номером ноль** идёт этот простой тезис.

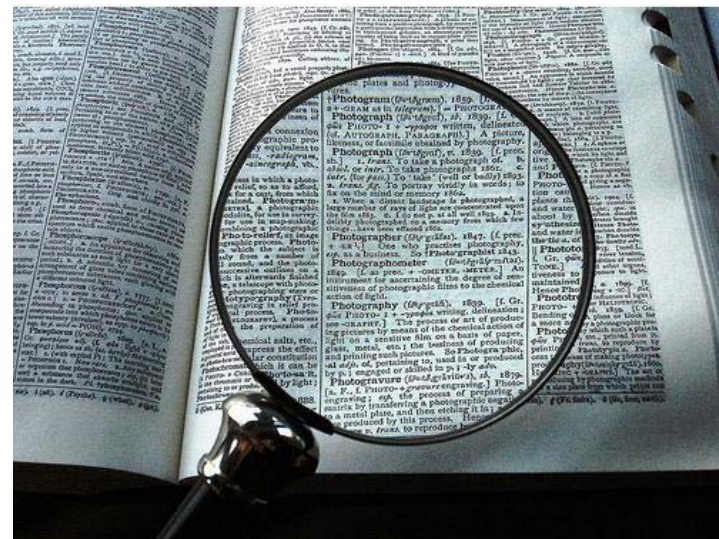
Добавлена новая страничка? Новая картинка? Вписана ссылка на CSS или JS?

Нужно проверить!

Нужно открыть эту **новую страницу**, убедиться, что **грузится картинка**, что **подхватились CSS и JS**, что **ссылка** на некий внешний источник **ведёт именно туда, куда должна**.

Известны случаи, когда верстальщики или программисты пропускали «какую-то совсем незаметную ссылочку», а потом...

Или долго искали, почему что-то не работает, или приходили к выводу, что «это в принципе не может работать» (для начинающих – нередкий случай) и реализовывали какой-то функционал или какие-то особенности интерфейса заново новым способом.



Используйте FireBug!

1. FireBug разработчику – друг, помощник и товарищ

Достаточно бывает просто поработать с разрабатываемым приложением, включив Firebug и поглядывая на его отчёт о запрошенных URL'ах и результатах выполнения этих запросов.



Появление там сообщений об ошибках должно насторожить.

Рекомендация начинающим: **ОБЯЗАТЕЛЬНО используйте Firebug.**

Иногда возникают вопросы в стиле: «Я вот скачал крутую библиотеку на JavaScript, все демо-примеры работают, а у меня – ничего не работает».

В 9 случаях из 10 – проблема в том, что неверно указаны ссылки на скрипты.

2. Самодиагностика – великая сила!

Да, не стоит писать каждый раз «линк-чекер». Его вообще не надо писать, т.к. есть готовые.

Но! Ваше приложение вполне способно хотя бы сообщать администратору о том, что что-то где-то идёт не так.

Современные движки, направленные на хитрую эмуляцию ссылок с использованием **mod_rewrite**, вынуждены сами отслеживать ситуации, когда, например, новость с номером 6451421875 не существует.



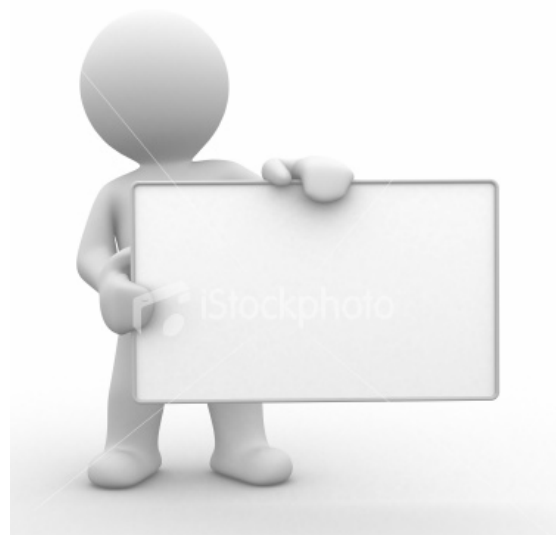
Самодиагностика

Лирическое отступление...

Итак, допустим, ваш интернет-магазин www.mycoolinetshop.com построен как раз с использованием **mod_rewrite** и ссылки внутри выглядят так:

[/notebooks/hp/new/](http://www.mycoolinetshop.com/notebooks/hp/new/)

Логично предположить, что страницы [/notebooks/hp/nuve/](http://www.mycoolinetshop.com/notebooks/hp/nuve/) не существует.



Но какой-то пользователь набрал такой URL.
Что он должен увидеть?

Самодиагностика



Как минимум — он должен увидеть **аккуратную, информативную** страницу «на мотив 404» с вариантами, по каким ссылкам пройти, чтобы увидеть **искомое** (что-то в стиле «Возможно, вы искали...»).



Ещё лучше — он должен увидеть **контекстно-зависимое сообщение** в стиле «Извините, в разделе Ноутбуки — HewlettPackard подраздела nиве нет.» + **всё те же варианты ссылок**, по которым может находиться **искомое**.

Самодиагностика

И, наконец, **чего явно НЕ должно быть**: пользователь не должен оказаться на «пустой белой странице».

Равно как не должен он оказаться и на странице вида «**вокруг какая-то вспомогательная информация, а в центре – пусто**».

Также **плохо**, когда в ответ на запрос пользователь получает стандартное сообщение браузера о том, что страница не найдена.

Возвращаемся к нашим ссылкам...



Самодиагностика

Итак, движок видит, что **запрошено несуществующее**. Это — повод **сформировать письмо администратору**.



Возможно, что-то где-то вышло из строя, или была допущена ошибка контент-менеджером, или (ещё веселее!) мы обнаружили **начальную стадию взлома**, когда кто-то «прощупывает» сайт.



Пометки в контенте

3. Создавайте пометки в генерируемом контенте

Итак, движок отследил ситуацию «404» («403» и т.п. – сейчас это не принципиально).

Он сообщил об этом администратору, а пользователю показал красивую страницу с подробными инструкциями.

Чудесно.

Но эта страница с инструкциями может идти либо вместе с «кодом ответа 404», либо с комментарием в первых же строках кода в стиле

`<!-- 404 -->`

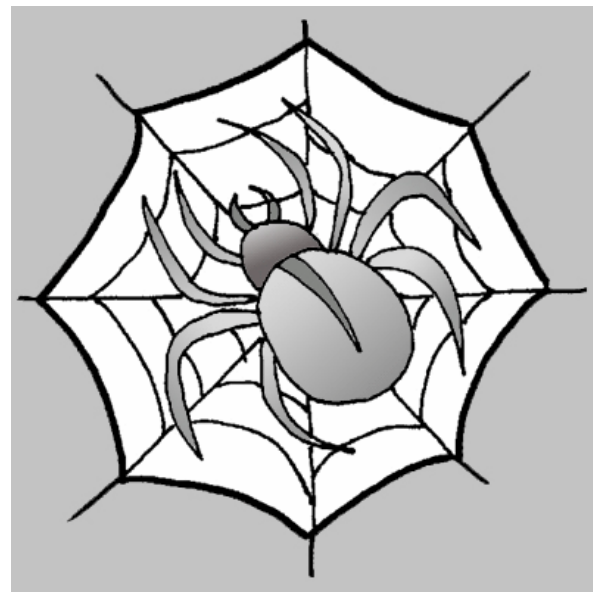
Зачем? А затем...



Пометки в контенте

А затем, что, возможно, мы пророчим нашему сайту долгую-предлогую жизнь и всё же планируем встроить в админку свой небольшенький «линк-чекер».

На основе этих данных наш **встроенный линк-чекер** сможет очень легко представить нам полный список страниц, где «что-то не так».



Ссылки исчезают

4. Ссылки исчезают

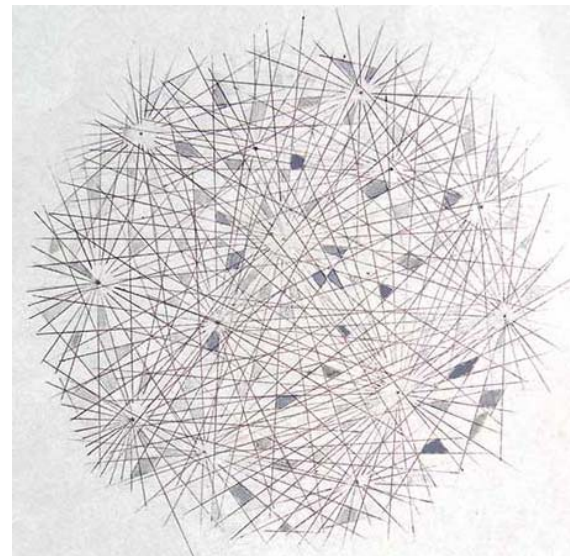
К сожалению, практика показывает, что **битые ссылки склонны размножаться со временем.**

Удалили какой-то раздел, статью, что-то переименовали и т.п. А ссылка где-то осталась.

Тогда возвращаемся к пункту 3 + приходим к выводу, что **проверку ссылок следует повторять раз за разом.**

Например, банально «по крону»(*) раз в полгода, месяц, неделю – в зависимости от масштаба проекта.

(*) <http://en.wikipedia.org/wiki/Cron> – планировщик задач в POSIX-совместимых ОС.

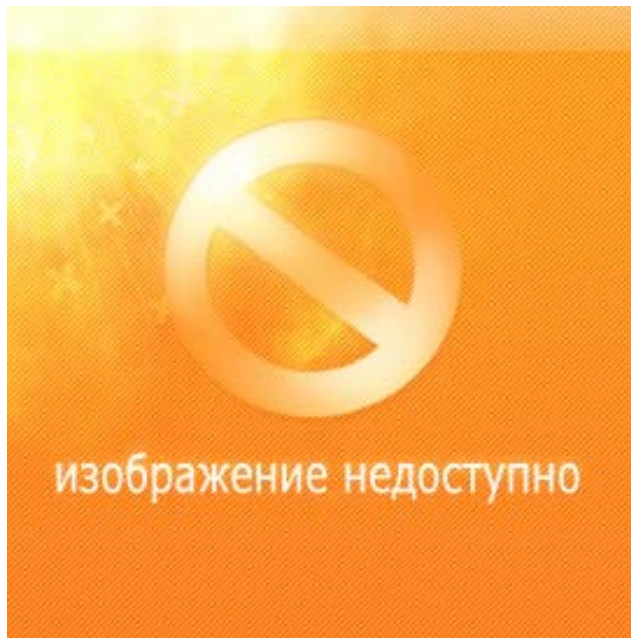


Обходной путь

5. Нормальные герои всегда идут в обход

Для некоторых проверок не нужен даже «линк-чекер».

Допустим, у нас есть новостной сайт, где в силу некоей специфики за новостью закреплена одна картинка (ещё пример – форум и аватары пользователей).



Имя картинки хранится в таблице базы данных. Можно просто пробегаться по базе данных и проверять, все ли файлы картинок на месте.

Ссылки на внешние ресурсы

6. Внешние ссылки исчезают ещё быстрее

Хорошо проверять «свои ссылки» (т.е. ссылки на свои же страницы и файлы). Но часто приходится указывать ссылки на внешние источники.

Система управления контентом должна анализировать добавляемый контент на предмет наличия в нём ссылок и проверять, не получено ли в ответ на запрос по ним сообщение «404» или ему подобное.

Это – первый шаг. В идеале можно извлечь запрошенный контент и сохранить где-то в отдельной таблице его хэш. При периодических проверках ссылок – снова извлекать контент, вычислять хэш и сравнивать с сохранённым. Отличие – повод для отправки сообщения администратору.



Хитрые битые ссылки

7. Внутренний враг

Сложнее всего автоматизировать самодиагностику битых ссылок в шаблонах, JavaScript и т.п.

Но спасает эту ситуацию то, что такие ссылки не склонны «умирать без причин», а потому тщательная проверка «линк-чекером» сайта перед его сдачей в эксплуатацию поможет сильно сэкономить нервы и время.



Ссылка «на самое себя»

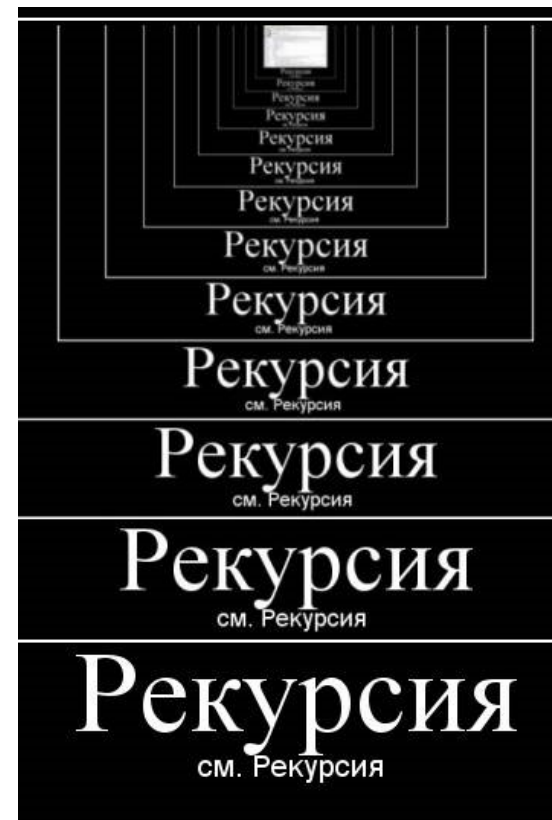
8. «Не самосошлись!»

Есть хорошее разумное правило – **«на странице не должно быть ссылок на саму себя».**

Обнаружение в процессе тестирования таких ссылок позволяет **избежать ситуаций, когда пользователь после клика окажется на той же самой странице.**

Это, конечно, больше вопрос юзабилити, но всё же стоит и ему уделить внимание.

Также, по мнению отдельных специалистов, поисковые системы не любят, когда в рамках сайта на одну и ту же страницу ведёт несколько ссылок, если при этом ссылки «называются по-разному» (например, «машины», «цветы», «нефть»).



Чек-лист по проверке ссылок

В заключение – полу-чек-лист-полу-план. Простенький, но даже такой – лучше, чем никакого.

1. Проверить **работоспособность всех ссылок** в добавленной или исправленной части сайта.
2. Загрузить страницу со включённым Firebug. **Убедиться, что Firebug не показал сообщений об ошибках.**
3. **Запускать механизм самодиагностики** после выхода каждого билда.
4. Проверить **наличие «меток» в страницах** с сообщениями об ошибках, генерируемых движком.

Чек-лист по проверке ссылок

- 5. Проверять все внешние ссылки. Хотя бы – один раз вручную сразу после добавления.
- 6. Убедиться в отсутствии на страницах ссылок «на самих себя».
- 7. Каждый релиз-кандидат полностью проверять «линк-чекером».

Ещё идеи?

Пример тестирования по методу Б/Я

Прежде чем приступить к практике тестирования по методу белого ящика, рассмотрим **несколько примеров того, как анализ кода приложения может помочь выявить множество проблем.**

И если ранее мы говорили больше о клиентской части веб-ориентированного приложения (HTML, CSS, JS), то **сейчас мы рассмотрим примеры тестирования по методу белого ящика серверной части приложения.**

Итак...

Пример тестирования по методу Б/Я

```
// авторизуем пользователя  
mysql_query("SELECT * from users where  
username=".$_POST[un]."          AND  
userpassword=".$_POST[up]);
```

1. **Нарушен синтаксис SQL.** Имена полей не взяты в обратные апострофы, значения не взяты в кавычки.
2. **Нарушен синтаксис PHP:** строковые ключи массива не взяты в кавычки.
3. **Нет проверки** на то, что в массиве \$_POST существуют элементы «un» и «up».
4. **Нет анализа и фильтрации** данных, полученных из массива \$_POST.

Пример тестирования по методу Б/Я

```
// читаем и выводим файл построчно  
$data = file($filename);  
foreach ($data as $line)  
{  
    echo $line;  
}
```

1. Нет проверки на существование файла.
2. Нет проверки на то, что массив `$data` не пустой.
3. Неопределённая ситуация с переносами строк (теоретически, они останутся «из исходного файла», но если надо вывести данные в формате HTML – получится одна длинная строка).

Пример тестирования по методу Б/Я

```
// считаем среднее значение
$sum = 0;
foreach ($values as $value)
{
    $sum += $value;
}
$average = $sum/count($values);
```

1. Нет проверки на то, что массив `$values` не пустой.
2. Нет проверки на равенство знаменателя нулю.
3. Использована медленная функция `count()` (функция `sizeof()` работает быстрее).

Пример тестирования по методу Б/Я

```
// ищем подстроку в тексте
if (strpos($text, $line))
{
    echo "OK";
}
```

1. Если текст начинается с искомой подстроки, **поведение программы будет некорректным** (функция `strpos()` вернёт 0, что будет интерпретировано как `FALSE`).
2. Для строковой константы **использованы двойных кавычки вместо одинарных**, что немного замедляет работу.

Пример тестирования по методу Б/Я

```
// сортируем числа
for ($i=0; $i<sizeof($arr); $i++)
{
    for($j=$i; $j<sizeof($arr); $j++)
    {
        if ($arr[$i]<$arr[$j])
        {
            $x=$arr[$j]; $arr[$j]=$arr[$i]; $arr[$i] = $x;
        }
    }
}
```

1. Не следует в цикле вызывать функцию sizeof().
2. Вместо всего этого кода достаточно было написать `sort($arr);`

Пример тестирования по методу Б/Я

```
// проверка статуса пользователя  
if ($status=76)  
{  
    ...  
}
```

1. Использовано «=» вместо «==», т.е. присваивание вместо сравнения.
2. Использование числа 76 – хардкодинг. Логичнее использовать переменную или константу.

Пример тестирования по методу Б/Я

```
// считаем сумму чисел  
foreach ($arr as $item)  
{  
    $num = $item;  
    $sum = $sum + $num;  
}
```

1. Нет инициализации значения суммы.
2. Использовано лишнее действие (\$num = \$item).

Пример тестирования по методу Б/Я

Фрагмент кода в движке сайта (C) bash.org.ru

```
function getRandomPassword()  
{  
    return 'aaaaaa';  
}
```

Тест для проверки изученного

1. Что такое валидация?
2. Какие виды валидации вы знаете?
3. Какие аргументы за и против валидации вы можете привести?
4. Какие показатели качества кода не гарантирует его валидность?
5. Вам нужно быстро провести «экспресс-валидацию» (хоть какую-то, в общем виде) кода некоторого сайта. Под рукой никаких инструментальных средств нет. Что вы используете?
6. Какие рекомендации по предотвращению проблем со ссылками на стадии их добавления вы можете дать?
7. Использование какого инструментального средства позволяет легко отслеживать битые ссылки в процессе навигации по приложению?

Тест для проверки изученного

8. Как можно повысить удобство работы с приложением в случае, если пользователь запрашивает несуществующую ссылку?
9. Какое функциональное дополнение веб-приложения в контексте диагностики неверных ссылок позволяет упростить работу администратора?
10. Какую пользу могут принести специальные пометки в контенте, генерируемом в ответ на ошибочную ситуацию?
11. Какой вывод следует из того факта, что количество битых ссылок со временем увеличивается?
12. Какую технику можно в некоторых случаях применить для проверки битых ссылок без непосредственного анализа самих ссылок?
13. Какую технику можно применить для проверки корректности ссылок на внешние источники?
14. Проверку каких ссылок сложнее всего автоматизировать (силами движка приложения) и почему?
15. Приведите несколько примеров из чек-листа по проверке ссылок.