HTML Sanity Checker Architecture Documentation

Table of Contents

Version {version} of 2022-01-14

Within the following text, the "Html Sanity Checker" shall be abbreviated with `HtmlSC`

Goals of this Documentation

This documentation is an example of arc42 documentation.

You may copy this documentation or parts of it for your own projects. In such cases you must include a link or reference to arc42 or aim42 (we regard this as *fair-use*).

For real-world projects, the relation of code and documentation is over-sized.

Disclaimer

We provide absolutely **no guarantee**, neither for the accuracy of this documentation nor for any property or feature of the software described here.

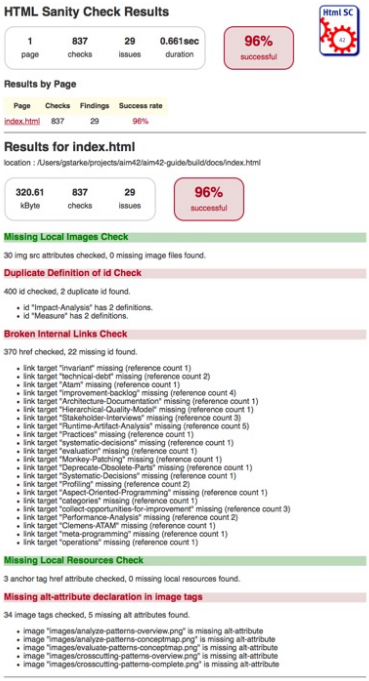
Do not use this software in critical situations or projects.

Chapter 1. Introduction and Goals

`HtmlSC` shall support authors creating digital formats with hyperlinks and integration of images and similar resources.

1.1. Requirements Overview

The overall goal of `HtmlSC` is to create neat and clear reports, showing errors within HTML files - as shown in the adjoining figure.



1.1.1. Basic Usage

1. A user configures the location (directory and filename) of one or more HTML file(s),
2. and the corresponding images directory.
3. `HtmlSC` performs various checks on the HTML and
4. reports its results either on the console or as HTML report.

`HtmlSC` can run from the command line or as Gradle-plugin.

Terminology: What Can Go Wrong in HTML Files?

Apart from purely syntactical errors, many things can go wrong in html, especially with respect to hyperlinks, anchors and id's - as those are often manually maintained.

Primary sources of problems are bad links (in technical terms: URIs). For further information, see the background information on URIs.

Broken Cross References:: Cross-references (internal links) can be broken, e.g. due to missing or misspelled link-targets.

See BrokenCrossReferencesChecker

Missing image files: Referenced image files can be missing or misspelled.

See MissingImageFilesChecker.

Missing local resources: Referenced local resources (other than images) can be missing or misspelled.

See MissingLocalResourcesChecker

Duplicate link targets: link-targets can occur several times with the same name - so the browser cannot know which is the desired target.

See DuplicateIdChecker.

Broken external links: External http links can be broken due to myriads of reasons: misspelled, link-target currently offline, illegal link syntax.

See BrokenHttpLinksChecker.

Missing Alt Attribute in Image Tags: Images missing an alt-attribute.

See MissingImgAltAttributeChecker.

Checking and reporting these errors and flaws is the central *business requirement* of `HtmlSC`.

Important terms (**domain terms**) of html sanity checking is documented in a (small) domain model.

1.1.2. General Functionality

Table 1. General Requirements

ID	Functionality	Description
G-1	read HTML file	HtmlSC shall read a single (configurable) HTML file
G-2	Gradle-plugin	HtmlSC can be run as Gradle-plugin.
G-3	command line usage	HtmlSC can be called from the command line with arguments and options
G-4	configurable output	output can be configured to console or file
G-5	free and open source	all required dependencies shall be compliant to the CC-SA-4 licence.
G-6	available via public repositories	like bintray or jcenter.
G-7	configurable to check multiple HTML files	configure a set of files to be processes in a single run and produce a joint report. (useful for e.g. API documentation with many HTML files referencing each other)

1.1.3. Types of Sanity Checks

Table 2. Required Checks

ID	Check	Description
R-1	missing image files	Check all image tags if the referenced image files exist. See MissingImageFilesChecker
R-2	broken internal links	Check all internal links from anchor-tags (href="#XYZ") if the link targets "XYZ" are defined. See BrokenCrossReferencesChecker
R-3	missing local files	either other html-files, pdf's or similar. See MissingLocalResourcesChecker
R-4	duplicate link targets	Check all bookmark definitions (... id="XYZ") whether the id's ("XYZ") are unique. See DuplicateIdChecker
R-5	malformed links	Check all links for syntactical correctness
R-6	missing alt-attribute	in image-tags. See MissingImgAltAttributeChecker
R-7	unused-images	Check for files in image-directories that are not referenced by any of the HTML files in this run
R-8	illegal link targets	Check for malformed or illegal anchors (link targets).

Table 3. Optional Checks

ID	Check	Description
Opt-1	missing external images	Check externally referenced images for availability
Opt-2	broken external links	Check external links for both syntax and availability

1.1.4. Reporting and Output Requirements

Table 4. Reporting Requirements

ID	Requirement	Description
R-1	various output formats	Checking output in plain text and HTML
R-2	output to stdout	HtmlSC can output results on stdout (the console)
R-3	configurable file output	HtmlSC can store results in file in configurable directories

1.2. Quality Goals

Table 5. Quality-Goals

Priority	Quality-Goal	Scenario
1	Correctness	Every broken internal link (cross reference) is found.
1	Correctness	Every missing local image is found.
2	Flexibility	Multiple checking algorithms, report formats and clients. At least Gradle, command-line and a graphical client have to be supported.
2	Safety	Content of the files to be checked is never altered.
2	Correctness	Correctness of every checker is automatically tested for positive AND negative cases
2	Correctness	Every reporting format is tested: Reports must exactly reflect checking results.
3	Performance	Check of 100kB html file performed under 10 secs (excluding gradle startup)

1.3. Stakeholder

Table 6. Stakeholder

Role	Description	Goal, Intention
Documentation author	writes documentation with Html output	wants to check that the resulting document contains good links, image references
arc42 user	uses the arc42 template for architecture documentation	wants a small but practical example of how to apply arc42.
aim42 contributor	contributes to aim42 methode-guide	check generated html code to ensure links and images are correct during (gradle-based) build process
software developer		wants an example of pragmatic architecture documentation and arc42 usage

1.4. Background Information on URIs

The generic structure of a Uniform Resource Identifier consists of the following parts: [type][://][subdomain][domain][port][path][file][query][hash]

An example, visualized:

The `java.net.URL` class contains a generic parser for URLs and URIs. See the following snippet, taken from the unit test class `URLUtilTest.groovy`:

Generic URI Structure

```
    @Test
    public void testGenericURISyntax() {
        // based upon an example from the Oracle(tm) Java tutorial:
        // http://docs.oracle.com/javase/tutorial/networking/urls/urlInfo.html
        def aURL =
                new
URL("http://example.com:42/docs/tutorial/index.html?name=aim42#INTRO");
        aURL.with {
            assert getProtocol() == "http"
            assert getAuthority() == "example.com:42"
            assert getHost() == "example.com"
            assert getPort() == 42
            assert getPath() == "/docs/tutorial/index.html"
            assert getQuery() == "name=aim42"
            assert getRef() == "INTRO"
        }
    }
```

URIs are used to **reference** other resources. For `HtmlSC` it is useful to distinguish between internal (== local)and external references:

- Internal references, a.k.a. Cross-References
- External references

1.4.1. Intra-Document URIs

a file... ref can be an internal link, or a URI without protocol...

1.4.2. References on URIs and HTML Syntax

- IETF RFC-2396 on URI Syntax: The fundamental reference!
- W3C HTML Reference
- Wikipedia on URI-Schemes

Chapter 2. Constraints

`HtmlSC` shall be:

- platform-independent and should run on the major operating systems (Windows™, Linux, and Mac-OS™)
- integrated with the Gradle build tool
- runnable from the command line
- developed under a liberal open-source license

Chapter 3. Context

3.1. Business Context

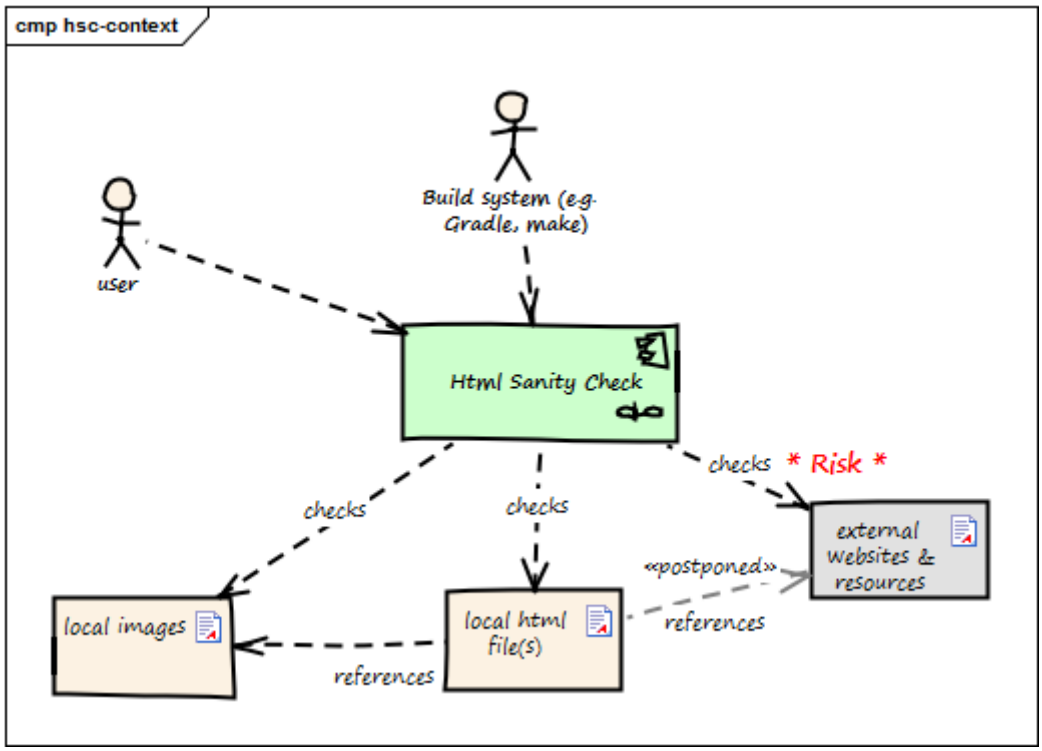


Figure 1. Business Context

Table 7. Business Context

Neighbor	Description
user	documents software with toolchain that generates html. Wants to ensure that links within this html are valid.
build system	
local html files	`HtmlSC` reads and parses local html files and performs sanity checks within those.
local image files	`HtmlSC` checks if linked images exist as (local) files.
external web resources	`HtmlSC` can be configured to optionally check for the existence of external web resources. Due to the nature of web systems, this check might need a significant amount of time and might yield invalid results due to network and latency issues.

3.2. Deployment Context

The following diagram shows the participating computers ({node}) with their technical connections plus the major {artifact} of `HtmlSC`, the hsc-plugin-binary.

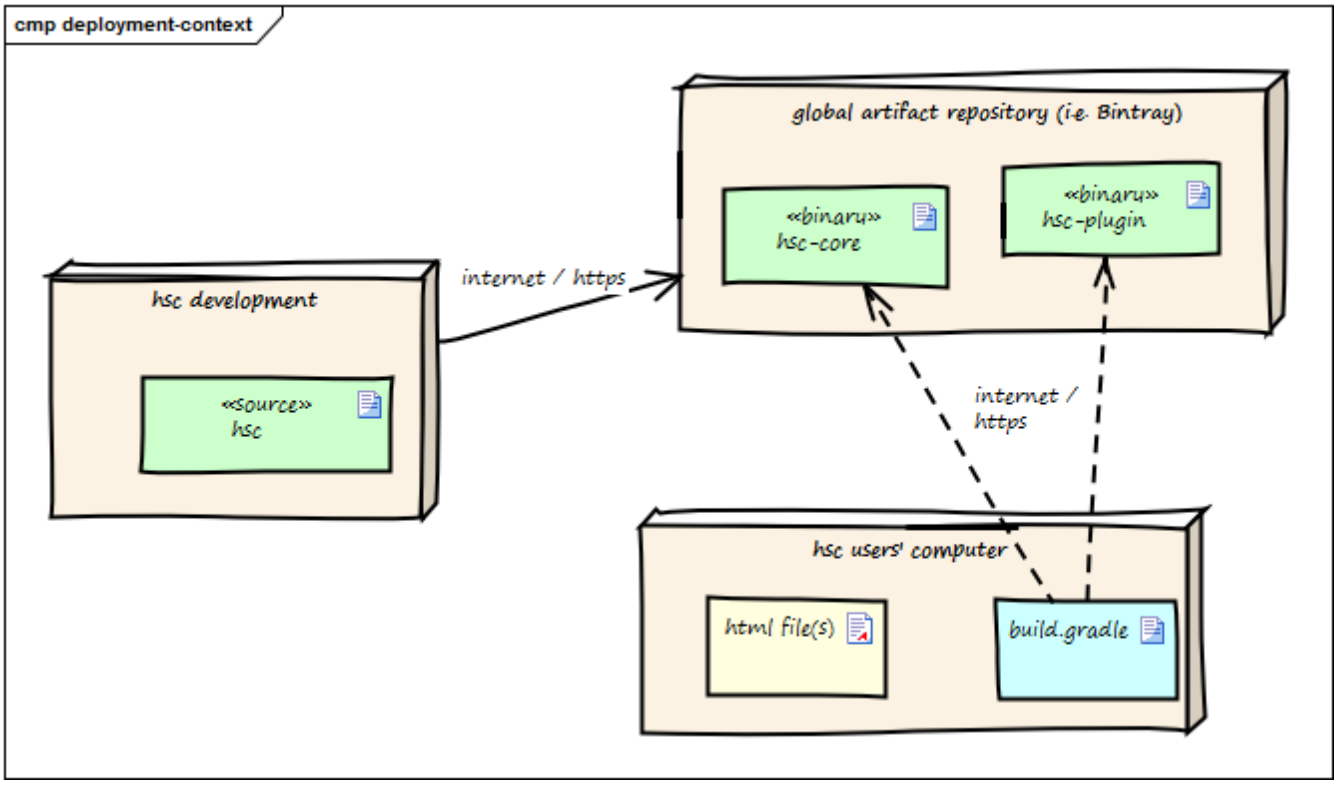


Figure 2. Deployment Context

Table 8. Deployment Context

Node / Artifact	Description
{node} hsc-development	where development of `HtmlSC` takes place
{artifact} hsc-plugin-binary	compiled and packaged version of `HtmlSC` including required dependencies.
{node} artifact repository (Bintray)	global public *cloud* repository for binary artifacts, similar to mavenCentral. `HtmlSC` binaries are uploaded to this server.
{node} hsc user computer	where arbitrary documentation takes place with html as output formats.
{artifact} build.gradle	Gradle build script configuring (among other things) the `HtmlSC` plugin to perform the Html checking.

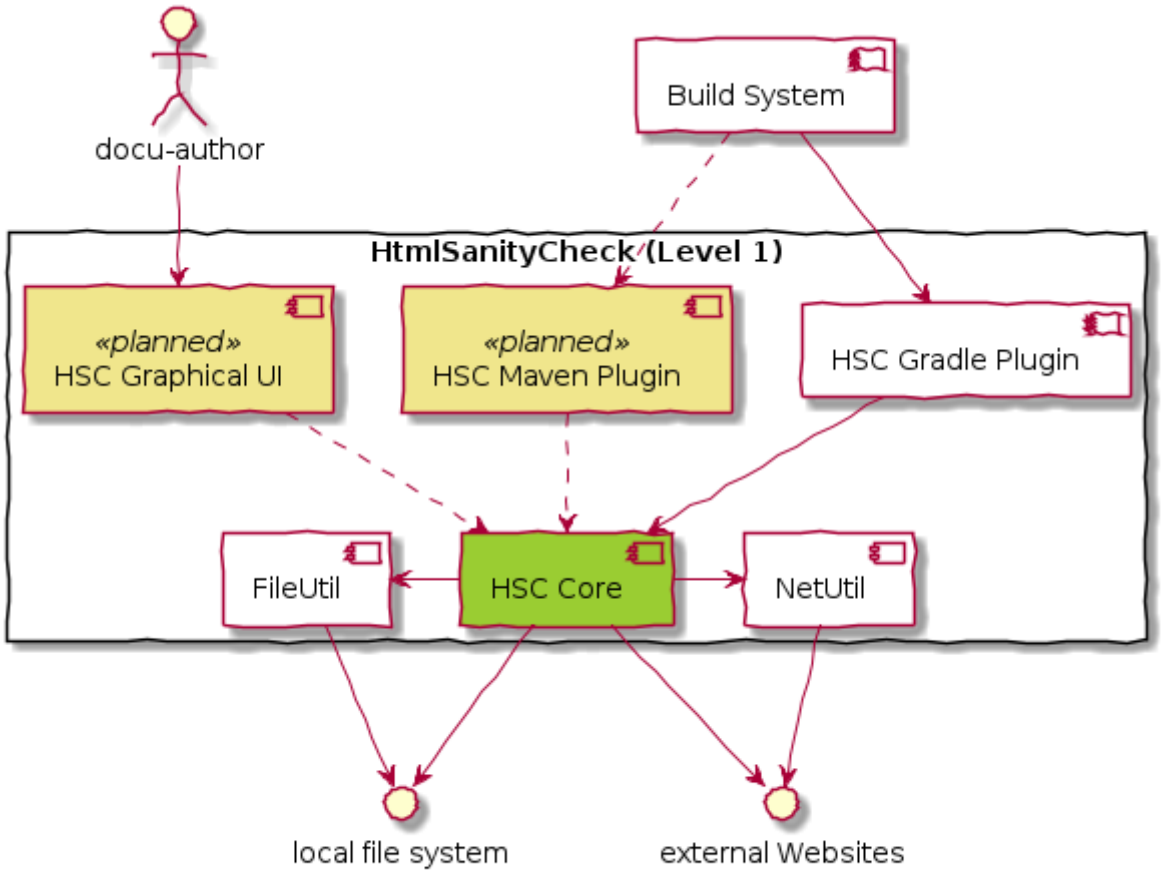
Details see deployment view.

Chapter 4. Solution Strategy

- Implement `HtmlSC` in Groovy and Java with minimal external dependencies. Wrap this implementation into a Gradle plugin, so it can be used within automated builds. Details are given in the Gradle plugin concept.
- Apply the *template-method-pattern* (see e.g. {template-method-url}) to enable:
  - multiple checking algorithms. See the concept for checking algorithms,
  - both HTML (file) and text (console) output. See the reporting concept.

Chapter 5. Building Block View

5.1. Whitebox HtmlSanityChecker



HtmlSanityCheck
https://github.com/aim42/htmlSanityCheck

Rationale

We used *functional decomposition* to separate responsibilities:

- CheckerCore shall encapsulate checking logic and Html parsing/processing.
- all kinds of outputs (console, html-file, graphical) shall be handled in a separate component (`Reporter`)
- Implementation of Gradle specific stuff shall be encapsulated.

Contained Blackboxes

Table 9. HtmlSanityChecker building blocks

HSC Core	hsc core: html parsing and sanity checking, configuration, reporting.
HSC Gradle Plugin	integrates the Gradle build tool with `HtmlSC`, enabling arbitrary gradle builds to use `HtmlSC` functionality.
HSC Maven Plugin	(planned, not yet implemented)
HSC Graphical Interface	(planned, not implemented)

Interfaces

Table 10. HtmlSanityChecker internal interfaces

Interface	Description
usage via shell	arc42 user uses a command line shell to call the `HtmlSC`
build system	currently restricted to Gradle: The build system uses `HtmlSC` as configured in the buildscript.
local-file system	`HtmlSC` needs access to several local files, especially the html page to be checked and to the corresponding image directories.
external websites	to check external links, `HtmlSC` needs to access external sites via http HEAD or GET requests.

5.1.1. *HSC Core* (Blackbox)

Intent/Responsibility

HSC_Core contains the core functions to perform the various sanity checks. It parses the html file into a DOM-like in-memory representation, which is then used to perform the actual checks.

Interfaces

Table 11. HSC_Core Interfaces

Interface (From-To)	Description
Command Line Interface → Checker	Uses the #AllChecksRunner class.
Gradle Plugin → Checker	Exposes `HtmlSC` via a standard Gradle plugin, as described in the Gradle user guide.

Files

- org.aim42.htmlsanitycheck.AllChecksRunner
- org.aim42.htmlsanitycheck.HtmlSanityCheckGradlePlugin

5.2. Building Blocks - Level 2

5.2.1. *HSC-Core* (Whitebox)

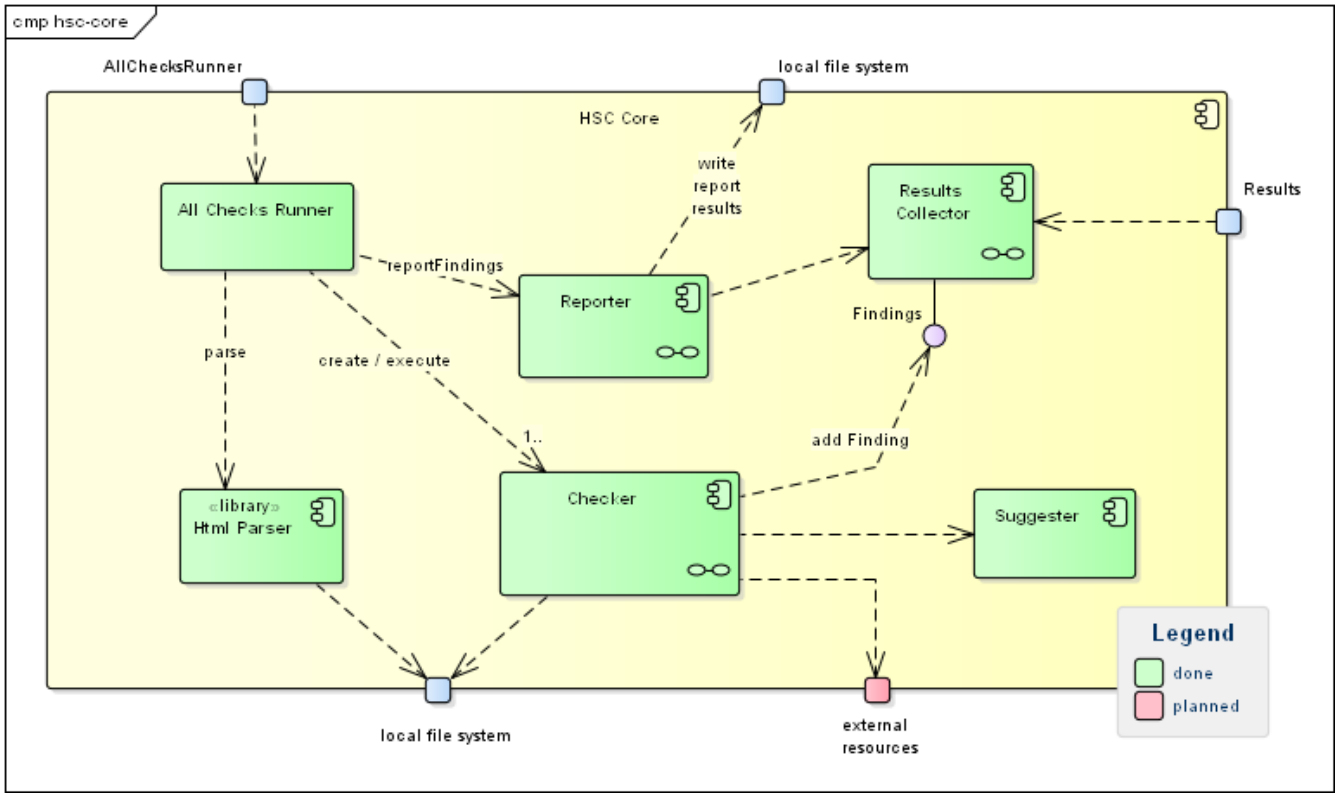


Figure 3. HSC-Core (Whitebox)

Rationale

This structures follows a strictly functional decomposition:

- parsing and handling html input
- checking
- collecting checking results

Contained Blackboxes

Table 12. HSC-Core building blocks

Checker	Abstract class, used in form of the template-pattern. Shall be subclassed for all checking algorithms.
AllChecksRunner	Facade to the different Checker instances. Provides a (parameter-driven) command-line interface.
ResultsCollector (Whitebox)	Collects all checking results. Its interface `Results` is contained in the whitebox description
Reporter	Reports checking results to either console or an html file.
HtmlParser	Encapsulates html parsing, provides methods to search within the (parsed) html.
Suggester	In case of checking issues, suggests alternatives by comparing the faulty element to the one present in the html file. Currently not implemented

5.2.2. Checker and xyzChecker Subclasses

The abstract Checker provides a uniform interface (`public void check()`) to different checking

algorithms. It is based upon the concept of extensible checking algorithms.

5.3. Building Blocks - Level 3

5.3.1. ResultsCollector (Whitebox)

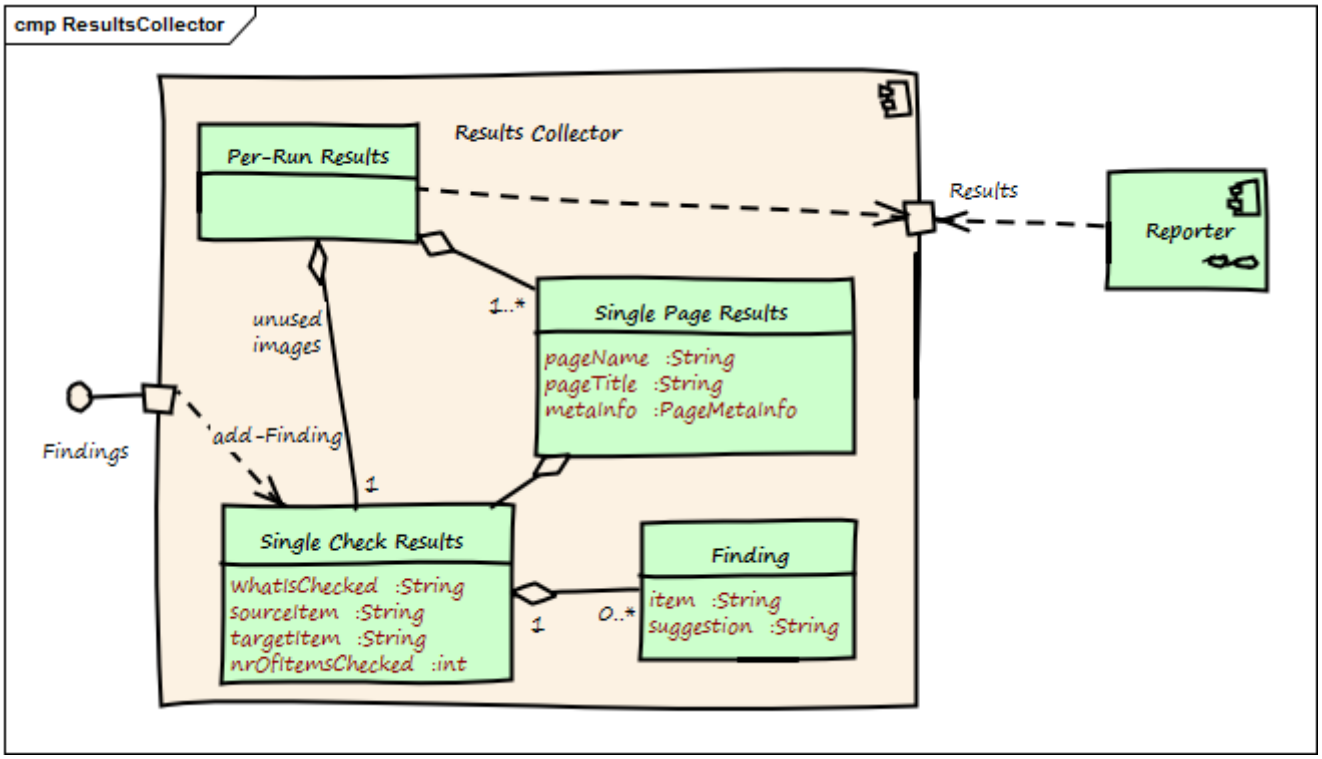


Figure 4. Results Collector (Whitebox)

Rationale

This structures follows the hierarchy of checks - namely managing results for:

1. a number of pages/documents, containing:
2. a single page, each containing many
3. single checks within a page

Contained Blackboxes

Table 13. ResultsCollector building blocks

Per-Run Results	results for potentially many Html pages/documents.
Single-Page-Results	results for a single page
Single-Check-Results	results for a single type of check (e.g. missing-images check)
Finding	a single finding, (e.g. "image 'logo.png' missing"). Can hold suggestions and (planned for future releases) the responsible html element.

Interface `Results`

The `Result` interface is used by all clients (especially `Reporter` subclasses, graphical and command-line clients) to access checking results. It consists of three distinct APIs for overall `RunResults`, single-page results (`PageResults`) and single-check results (`CheckResults`). See the interface definitions below - taken from the Groovy- source code:

Interface RunResults

```
public interface RunResults {

    // returns results for all pages which have been checked
    public ArrayList<SinglePageResults> getResultsForAllPages()

    // how many pages were checked in this run?
    public int nrOfPagesChecked()

    // how many checks were performed in all?
    public int nrOfChecksPerformedOnAllPages()

    // how many findings (errors and issues) were found in all?
    public int nrOfFindingsOnAllPages()

    // how long took checking (in milliseconds)?
    public Long checkingTookHowManyMillis()
}
```

Interface PageResults

```
public interface PageResults {

    // what's the title of this page?
    public String getPageTitle()

    // what's the filename and path?
    public String getPageFileName()
    public String getPageFilePath()

    // how many items have been checked?
    public int nrOfItemsCheckedOnPage()

    // how many problems were found on this page?
    public int nrOfFindingsOnPage()

    // how many different checks have run on this page?
    public int howManyCheckersHaveRun()
}
```

Interface CheckResults

```
public interface CheckResults {

    // return a description of what is checked
    // (e.g. "Missing Images Checker" or "Broken Cross-References Checker"
    public String description()

    // returns all findings/problems found during this check
    public  ArrayList<Finding> getFindings()
}
```

Chapter 6. Runtime View

Not appropriate for this system due to very simple implementation.

Chapter 7. Deployment View

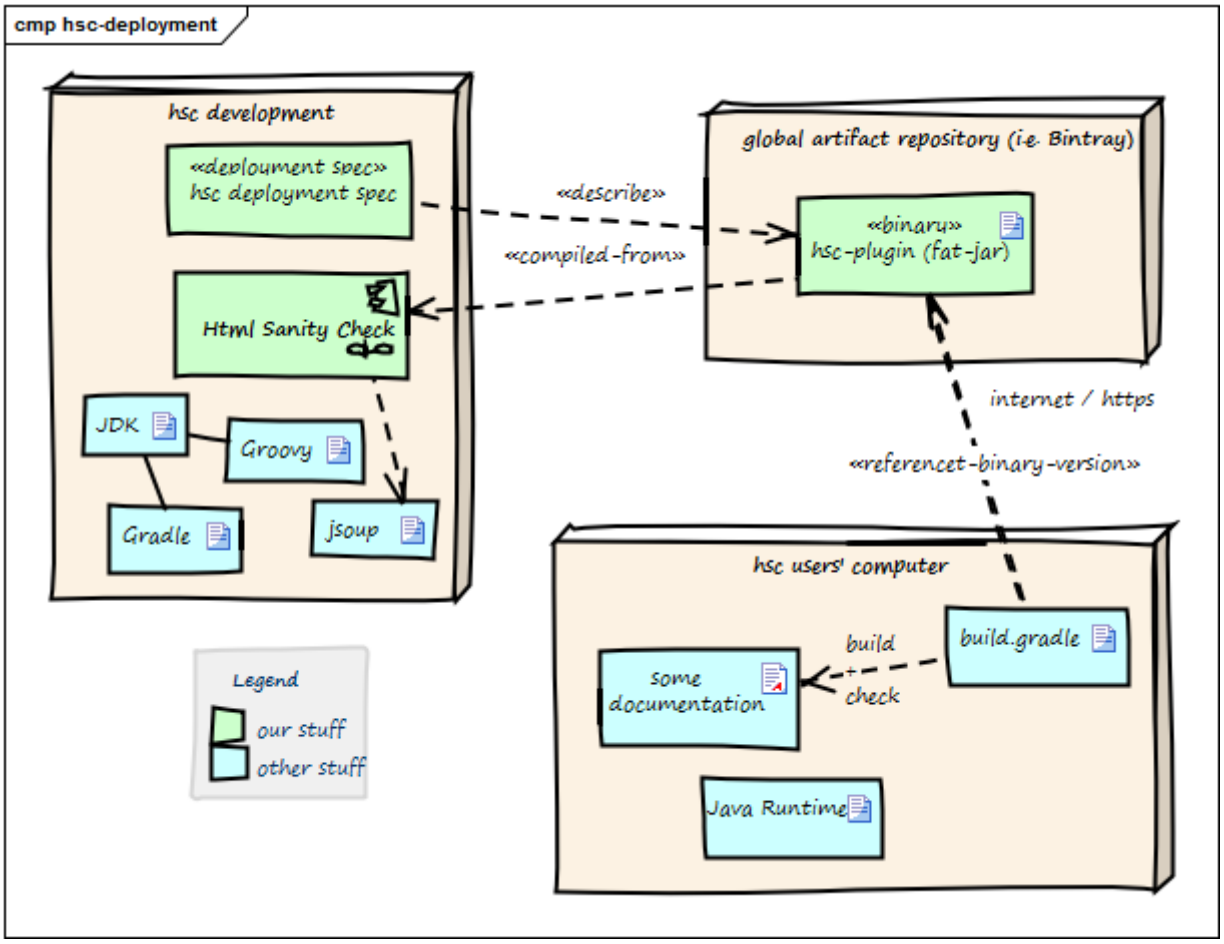


Figure 5. Deployment

Table 14. Deployment

Node / Artifact	Description
hsc plugin binary	compiled version of `HtmlSC`, including required dependencies.
hsc-development	where development of `HtmlSC` takes place
artifact repository (Bintray)	global public *cloud* repository for binary artifacts, similar to mavenCentral. `HtmlSC` binaries are uploaded to this server.
hsc user computer	where arbitrary documentation takes place with html as output formats.
build.gradle	Gradle build script configuring (among other things) the `HtmlSC` plugin to check *some documentation*.

The three nodes (*computers*) shown in Deployment are connected via Internet.

Sanity checker will:

1. be bundled as a single jar,
2. be uploaded to the Bintray repository,
3. referencable within a gradle buildfile,
4. provide a main method with parameters and options, so all checks can be called from the

command line.

Chapter 8. Technical and Crosscutting Concepts

8.1. HTML Checking Domain Model

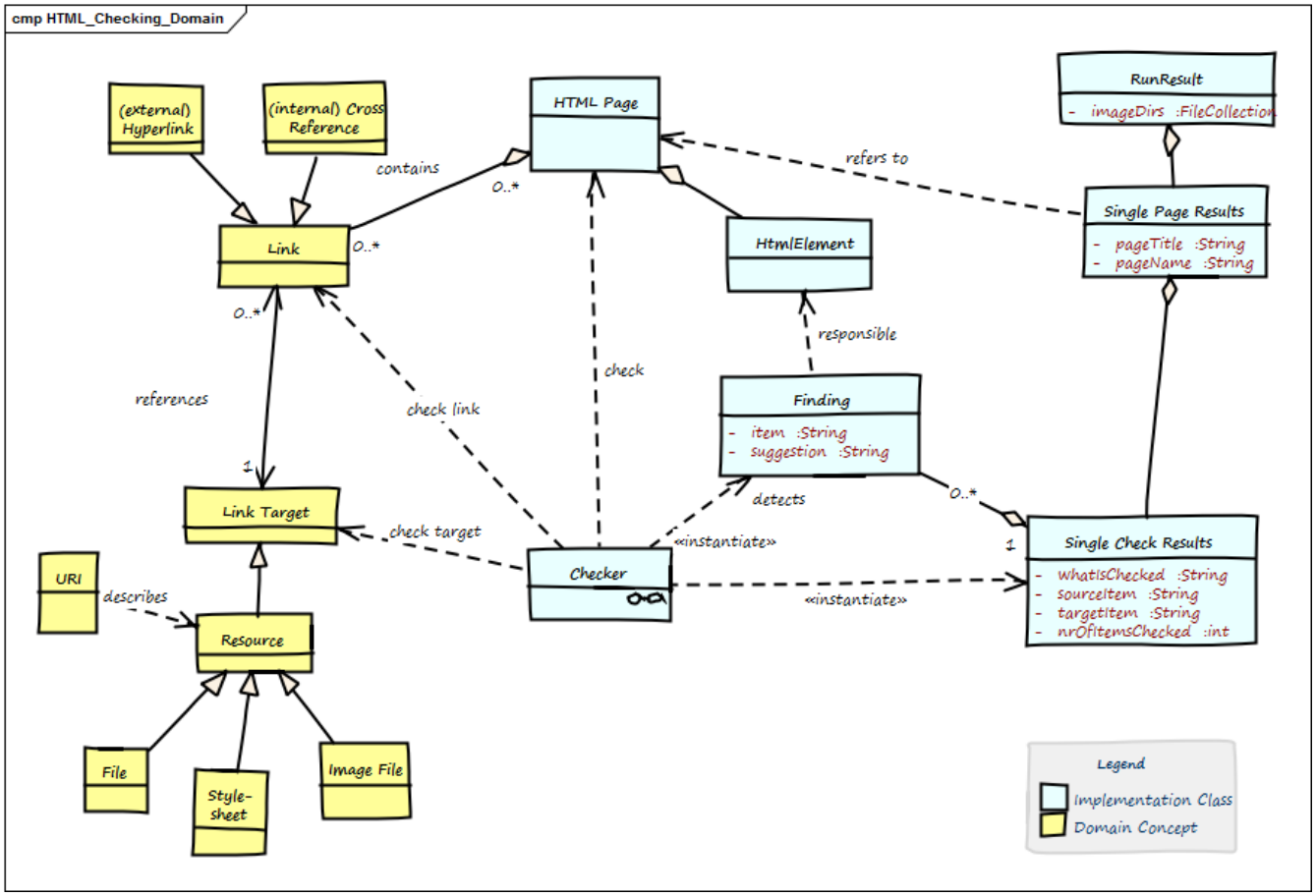


Figure 6. HTML Checking Domain Model

Table 15. Domain Model

Term	Description
Anchor	Html element to create →Links. Contains link-target in the form `<a href="link-target">`
Cross Reference	Link from one part of the document to another part within the same document. A special form of →Internal Link, with a →Link Target in the same document.
External Link	Link to another page or resource at another domain.
Finding	Description of a problem found by one →Checker within the →Html Page.
Html Element	HTML pages (documents) are made up by HTML elements .e.g., `<a href="link target">`, `<img src="image.png">` and others. See the W3-Consortium

Term	Description
Html Page	A single chunk of HTML, mostly regarded as a single file. Shall comply to standard HTML syntax. Minimal requirement: Our HTML parser can successfully parse this page. Contains →Html Elements. Also called *Html Document*.
id	Identifier for a specific part of a document, e.g. `<h2 id="#someHeader">`. Often used to describe →Link Targets.
Internal Link	Link to another section of the same page or to another page of the same domain. Also called *Local Link*.
Link	Any a reference in the →Html Page that lets you display or activate another part of this document (→Internal Link) or another document, image or resource (can be either →Internal (local) or →External Link). Every link leads from the *Link Source* to the *Link Target*
Link Target	The target of any →Link, e.g. heading or any other a part of a →Html Document, any internal or external resource (identified by URI). Expressed by →id
Local Resource	local file, either other Html files or other types (e.g. pdf, docx)
Run Result	The overall results of checking a number of pages (at least one page).
Single Page Result	A collection of all checks of a single →Html Page.
URI	Universal Resource Identifier. Defined in RFC-2396. The ultimate source of truth concerning link syntax and semantic.

8.2. Gradle Plugin Concept and Development

You should definitely read the original Gradle User Guide on custom plugin development.

To enable the required Gradle integration, we implement a lean wrapper as described in the Gradle user guide.

```
class HtmlSanityCheckPlugin implements Plugin<Project> {
    void apply(Project project) {
    project.task('htmlSanityCheck',
        type: HtmlSanityCheckTask,
        group: 'Check')
    }
}
```

8.2.1. Directory Structure and Required Files

```
|-htmlSanityCheck
   |  |-src
   |  |  |-main
   |  |  |  |-org
   |  |  |  |  |-aim42
   |  |  |  |  |  |-htmlsanitycheck
   |  |  |  |  |  |  | ...
   |  |  |  |  |  |  |-HtmlSanityCheckPlugin.groovy ①
   |  |  |  |  |  |  |-HtmlSanityCheckTask.groovy
   |  |  |  |-resources
   |  |  |  |  |-META-INF                          ②
   |  |  |  |  |  |-gradle-plugins
   |  |  |  |  |  |  |-htmlSanityCheck.properties  ③
   |  |  |-test
   |  |  |  |-org
   |  |  |  |  |-aim42
   |  |  |  |  |  |-htmlsanitycheck
   |  |  |  |  |  |  | ...
   |  |  |  |  |  |  |-HtmlSanityCheckPluginTest
   |
```

① the actual plugin code: `HtmlSanityCheckPlugin.groovy` and `HtmlSanityCheckTask.groovy` groovy files

② Gradle expects plugin properties in `META-INF`

③ property file containing the name of the actual implementation class: `implementation-class=org.aim42.htmlsanitycheck.HtmlSanityCheckPlugin`

8.2.2. Passing Parameters From Buildfile to Plugin

To be done

8.2.3. Building the Plugin

The plugin code itself is built with gradle.

8.2.4. Uploading to Public Archives

8.2.5. Further Information on Creating Gradle Plugins

Although writing plugins is described in the Gradle user guide, a clearly explained sample is given in a Code4Reference tutorial.

8.3. Flexible Checking Algorithms

`HtmlSC` uses the template-method-pattern to enable flexible checking algorithms:

> The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses.
>
> — https://sourcemaking.com/design_patterns/template_method

We achieve that by defining the skeleton of the checking algorithm in one operation, deferring the specific checking algorithm steps to subclasses.

The invariant steps are implemented in the abstract base class, while the variant checking algorithms have to be provided by the subclasses.

Template method "performCheck"

```
    /**
    ** template method for performing a single type of checks on the given @see
HtmlPage.
     *
     * Prerequisite: pageToCheck has been successfully parsed,
     * prior to constructing this Checker instance.
    **/
    public SingleCheckResults performCheck( final HtmlPage pageToCheck) {
        // assert non-null htmlPage
        assert pageToCheck != null

        checkingResults = new SingleCheckResults()

        // description is set by subclasses
        initCheckingResultsDescription()

        return check( pageToCheck ) // <1> delegate check() to subclass
    }
```

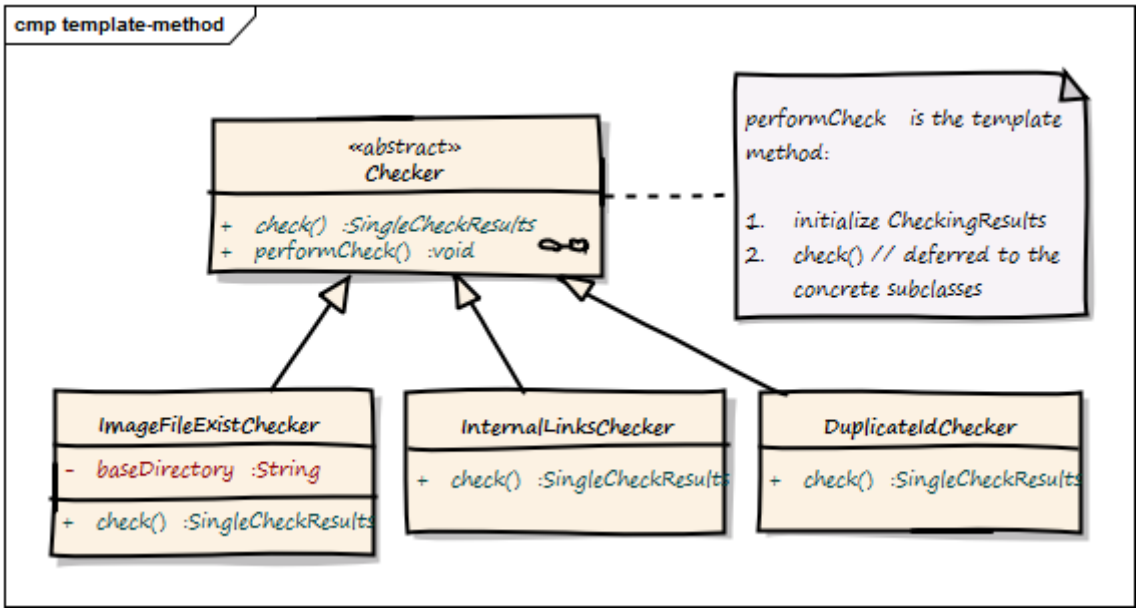


Figure 7. Template-Method Overview

Table 16. Template Method

Component	Description
Checker	*abstract* base class, containing the template method check() plus the public method performCheck()
MissingImageFilesChecker	checks if referenced local image files exist
MissingImgAltAttributeChecker	checks if there are image tags without alt-attributes
BrokenCrossReferencesChecker	checks if cross references (links referenced within the page) exist
DuplicateIdChecker	checks if any id has multiple definitions
MissingLocalResourcesChecker	checks if referenced other resources exist
BrokenHttpLinksChecker	checks if external links are valid
IllegalLinkChecker	checks if links do not violate HTML link syntax

8.3.1. MissingImageFilesChecker

Addresses requirement R-1.

Checks if image files referenced in <img src="someFile.jpg"> really exists on the local file system.

The (little) problem with checking images is their path: Consider the following HTML fragment (from the file testme.html):

```
<img src="./images/one-image.jpg">
```

This image file ("one-image.jpg") has to be located relative to the directory containing the corresponding HTML file.

Therefore the expected absolute path of the "one-image.jpg" has to be determined from the absolute path of the html file under test.

We check for existing files using the usual Java API, but have to do some *directory arithmetic* to get the absolutePathToImageFile:

```
File f = new File( absolutePathToImageFile );
if(f.exists() && !f.isDirectory())
```

8.3.2. MissingImgAltAttributeChecker

Addresses requirement R-6.

Simple syntactic check: iterates over all <img> tags to check if the image has an alt-tag.

8.3.3. BrokenCrossReferencesChecker

Addresses requirement R-2.

Cross references are document-internal links where the href="link-target" from the html anchor tag has no prefix like +http, https, ftp, telnet, mailto, file and such.

Only links with prefix # shall be taken into account, e.g. <a href="#internalLink">.

8.3.4. DuplicateIdChecker

Addresses requirement R-4.

Sections, especially headings, can be made link-targets by adding the id="#xyz" element, yielding for example html headings like the following example.

Problems occur if the same link target is defined several times (also shown below).

```
<h2 id="seealso">First Heading</h2>
<h2 id="seealso">Second Heading</h2>
<a href="#seealso">Duplicate definition - where shall I go now?</a>
```

8.3.5. MissingLocalResourcesChecker

Addresses requirement R-3.

Current limitations:

Does **NOT** deep-checking of references-with-anchors of the following form:

```
<a href="api/Artifact.html#target">GroupInit</a>
```

containing both a local (file) reference plus an internal anchor #target

See issues #252 (false positives) and #253 (deep links shall be checked)

8.3.6. BrokenHttpLinksChecker

Addresses requirement R-9.

Problem here are networking issues, latency and HTTP return codes. This checker is planned, but currently not implemented.

8.3.7. IllegalLinkChecker

Addresses requirement R-5.

This checker is planned, but currently not implemented. :jbake-menu: -

8.4. Encapsulate HTML Parsing

We encapsulate the third-party HTML parser (https://jsoup.org) in simple wrapper classes with interfaces specific to our different checking algorithms.

8.5. Flexible Reporting

`HtmlSC` allows for different output formats:

- formats (HTML and text) and
- destinations (file and console)

The reporting subsystem uses the template method pattern to allow different output formats (e.g. Console and HTML). The overall structure of reports is always the same:

Graphical clients can use the API of the reporting subsystem to display reports in arbitrary formats.

The (generic and abstract) reporting is implemented in the abstract Reporter class as follows:

```
/**
 * main entry point for reporting - to be called when a report is requested
 * Uses template-method to delegate concrete implementations to subclasses
*/
    public void reportFindings() {
        initReport()            ①
        reportOverallSummary()  ②
        reportAllPages()        ③
        closeReport()           ④
    }
//
    private void reportAllPages() {
        pageResults.each { pageResult ->
            reportPageSummary( pageResult ) ⑤
            pageResult.singleCheckResults.each { resultForOneCheck ->
               reportSingleCheckSummary( resultForOneCheck )  ⑥
               reportSingleCheckDetails( resultForOneCheck )  ⑦
               reportPageFooter()                             ⑧
        }
    }
```

① initialize the report, e.g. create and open the file, copy css-, javascript and image files.

② create the overall summary, with the overall success percentage and a list of all checked pages with their success rate.

③ iterate over all pages

④ write report footer - in HTML report also create back-to-top-link

⑤ for a single page, report the nr of checks and problems plus the success rate

⑥ for every singleCheck on that page, report a summary and

⑦ all detailed findings for a singleCheck.

⑧ for every checked page, create a footer, page break or similar to graphically distinguish pages between each other.

8.5.1. Styling the Reporting Output

- The `HtmlReporter` explicitly generates css classes together with the html elements, based upon css styling re-used from the Gradle JUnit plugin.
- Stylesheets, a minimized version of jQuery javascript library plus some icons are copied at report-generation time from the jar-file to the report output directory.
- Styling the back-to-top arrow/button is done as a combination of JavaScript plus some css styling, as described in https://www.webtipblog.com/adding-scroll-top-button-website/.

8.5.2. Copy Required Resources to Output Directory

When creating the HTML report, we need to copy the required resource files (css, JavaScript) to the output directory.

The appropriate copy method was re-used from the Gradle sources.

8.5.3. Attributions

Credits for the arrow-icon https://www.iconfinder.com/icons/118743/arrow_up_icon

Chapter 9. Design Decisions

9.1. Checking of external links postponed

In the current {revision} we won't check external links. These checks have been postponed to later versions.

9.2. HTML Parsing with jsoup

To check HTML we parse it into an internal (DOM-like) representation. For this task we use jsoup HTML parser, an open-source parser without external dependencies.

To quote from the jsoup website:

> jsoup is a Java library for working with real-world HTML. It provides a very convenient API for extracting and manipulating data, using the best of DOM, CSS, and jQuery-like methods.

Goals of this decision

Check HTML programmatically by using an existing API that provides access and finder methods to the DOM-tree of the file(s) to be checked.

Decision Criteria

- few dependencies, so the `HtmlSC` binary stays as small as possible.
- accessor and finder methods to find images, links and link-targets within the DOM tree.

Alternatives

- HTTPUnit: a testing framework for web applications and -sites. Its main focus is web testing and it suffers from a large number of dependencies.
- jsoup: a plain HTML parser without any dependencies (!) and a rich API to access all HTML elements in DOM-like syntax.

Find details on how `HtmlSC` implements HTML parsing in the HTML encapsulation concept.

9.3. String Similarity Checking with Jaro-Winkler-Distance

The small java-string-similarity library (by Ralph Allen Rice) contains implementations of several similarity-calculation algorithms. As it is **not available** as public binary, we use the sources instead, primarily:

```
net.ricecode.similarity.JaroWinklerStrategyTest
net.ricecode.similarity.JaroWinklerStrategy
```

The actual implementation of the similarity comparison has been postponed to a later release of `HtmlSC`

Chapter 10. Glossary

See the domain model for explanations of important terms.