Documentation

Introduction

What

★★☆☆ 🏆 Can explain the two types of developer docs

Developer-to-developer documentation can be in one of two forms:

1. **Documentation for *developer-as-user***: Software components are written by developers and reused by other developers, which means there is a need to document how such components are to be used. Such documentation can take several forms:
   - API documentation: APIs expose functionality in small-sized, independent and easy-to-use chunks, each of which can be documented systematically.
   - Tutorial-style instructional documentation: In addition to explaining functions/methods independently, some higher-level explanations of how to use an API can be useful.

- 📦 Example of API Documentation: String API.
- 📦 Example of tutorial-style documentation: Java Internationalization Tutorial.

2. **Documentation for *developer-as-maintainer***: There is a need to document how a system or a component is designed, implemented and tested so that other developers can maintain and evolve the code. Writing documentation of this type is harder because of the need to explain complex internal details. However, given that readers of this type of documentation usually have access to the source code itself, only *some* information needs to be included in the documentation, as code (and code comments) can also serve as a complementary source of information.

- 📦 An example: se-edu/addressbook-level4 Developer Guide.

Another view proposed by Daniele Procida in this article is as follows:

> **There is a secret that needs to be understood in order to write good software documentation: there isn't one thing called documentation, there are four. They are: tutorials, how-to guides, explanation and technical reference.** They represent four different purposes or functions, and require four different approaches to their creation. Understanding the implications of this will help improve most software documentation - often immensely. ...

TUTORIALS	HOW-TO GUIDES
A tutorial: • is learning-oriented • allows the newcomer to get started • is a lesson Analogy: teaching a small child how to cook	A how-to guide: • is goal-oriented • shows how to solve a specific problem • is a series of steps Analogy: a recipe in a cookery book
EXPLANATION	**REFERENCE**
An explanation: • is understanding-oriented • explains • provides background and context Analogy: an article on culinary social history	A reference guide: • is information-oriented • describes the machinery • is accurate and complete Analogy: a reference encyclopedia article

Software documentation (applies to both user-facing and developer-facing) is best kept in a text format for ease of version tracking. **A writer-friendly source format is also desirable** as non-programmers (e.g., technical writers) may need to author/edit such documents. As a result, formats such as Markdown, AsciiDoc, and PlantUML are often used for software documentation.

Exercises

Guidelines

Guideline: Go top-down, not bottom-up

What

★★☆☆ Can distinguish between top-down and bottom-up documentation

When writing project documents, a top-down breadth-first explanation is easier to understand than a bottom-up one.

Why

★★★☆ Can explain the advantages of top-down documentation

The main advantage of the top-down approach is that the document is structured like an upside down tree (root at the top) and **the reader can travel down a path she is interested in until she reaches the component she is interested to learn in-depth**, without having to read the entire document or understand the whole system.

How

★★☆☆ Can write documentation in a top-down manner

To explain a system called `SystemFoo` with two sub-systems, `FrontEnd` and `BackEnd`, start by describing the system at the highest level of abstraction, and progressively drill down to lower level details. An outline for such a description is given below.

[First, explain what the system is, in a black-box fashion (no internal details, only the external view).]

> `SystemFoo` is a ....

[Next, explain the high-level architecture of `SystemFoo`, referring to its major components only.]

> `SystemFoo` consists of two major components: `FrontEnd` and `BackEnd`.
>
> The job of `FrontEnd` is to ... while the job of `BackEnd` is to ...
>
> And this is how `FrontEnd` and `BackEnd` work together ...

[Now you can drill down to `FrontEnd`'s details.]

> `FrontEnd` consists of three major components: `A`, `B`, `C`
>
> `A`'s job is to ...
> `B`'s job is to...
> `C`'s job is to...
>
> And this is how the three components work together ...

[At this point, further drill down to the internal workings of each component. A reader who is not interested in knowing the nitty-gritty details can skip ahead to the section on `BackEnd`.]

> In-depth description of `A`
>
> In-depth description of `B`
>
> ...

[At this point drill down to the details of the `BackEnd`.]

> ...

Guideline: Aim for comprehensibility

What

★★☆☆ 🏆 Can explain the need for comprehensibility in documents

Technical documents exist to help others understand technical details. Therefore, **it is not enough for the documentation to be accurate and comprehensive; it should also be comprehensible**.

How

★★★☆ 🏆 Can write reasonably comprehensible developer documents

Here are some tips on writing effective documentation.

- **Use plenty of diagrams**: It is not enough to explain something in words; complement it with visual illustrations (e.g. a UML diagram).
- **Use plenty of examples**: When explaining algorithms, show a running example to illustrate each step of the algorithm, in parallel to worded explanations.
- **Use simple and direct explanations**: Convoluted explanations and fancy words will annoy readers. Avoid long sentences.
- **Get rid of statements that do not add value**: For example, 'We made sure our system works perfectly' (who didn't?), 'Component X has its own responsibilities' (of course it has!).
- **It is not a good idea to have separate sections for each type of artifact**, such as 'use cases', 'sequence diagrams', 'activity diagrams', etc. Such a structure, coupled with the indiscriminate inclusion of diagrams without justifying their need, indicates a failure to understand the purpose of documentation. Include diagrams when they are needed to explain something. If you want to provide additional diagrams for completeness' sake, include them in the appendix as a reference.

Exercises

Guideline: Document minimally, but sufficiently

What

★★☆☆ Can explain that documentation should be minimal yet sufficient

Aim for 'just enough' developer documentation.

- Writing and maintaining developer documents is an overhead. You should try to minimize that overhead.
- If the readers are developers who will eventually read the code, the documentation should complement the code and should provide only just enough guidance to get started.

How

★★★☆ Can write minimal yet sufficient documentation

Anything that is already clear in the code need not be described in words. Instead, **focus on providing higher level information that is not readily visible in the code or comments**.

Refrain from duplicating chunks of text. When describing several similar algorithms/designs/APIs, etc., do not simply duplicate large chunks of text. Instead, **describe the similarities in one place and emphasize only the differences in other places.** It is very annoying to see pages and pages of similar text without any indication as to how they differ from each other.

Tools

JavaDoc

What

★★☆☆ Can explain JavaDoc

JavaDoc is a tool for generating API documentation in HTML format from comments in the source code. In addition, modern IDEs use JavaDoc comments to generate explanatory tooltips.

An example method header comment in JavaDoc format (adapted from Oracle's Java documentation)

```java
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

📦 Generated HTML documentation:

getImage
`public Image getImage(URL url, String name)`

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

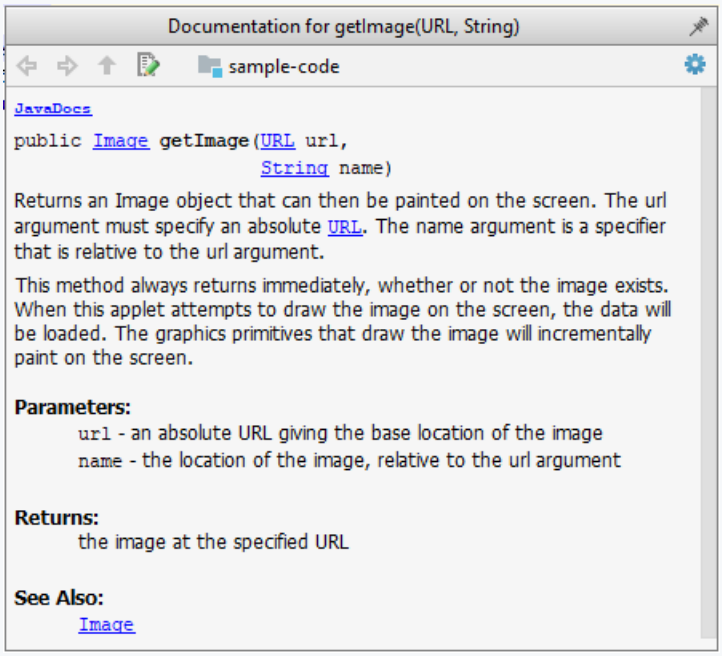
Parameters:
`url` - an absolute URL giving the base location of the image.

`name` - the location of the image, relative to the `url` argument.

Returns:
the image at the specified URL.

See Also:
`Image`

📦 Tooltip generated by Intellij IDE:



How

★★★☆ Can write JavaDoc comments

In the absence of more extensive guidelines (e.g., given in a coding standard adopted by your project), you can follow the two examples below in your code.

A minimal JavaDoc comment example for methods:

```
/**
 * Returns lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x X coordinate of position.
 * @param y Y coordinate of position.
 * @param zone Zone of position.
 * @return Lateral location.
 * @throws IllegalArgumentException If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
    throws IllegalArgumentException {
    // ...
}
```

A minimal JavaDoc comment example for classes:

```
package ...

import ...

/**
 * Represents a location in a 2D space. A <code>Point</code> object corresponds to
 * a coordinate represented by two integers e.g., <code>3,6</code>
 */
public class Point {
    // ...
}
```

Resources

- A short tutorial on writing JavaDoc comments -- from tutorialspoint.com
- A more detailed description -- from Oracle