**Topics in Software Engineering I**

**CSE 350**

**GROUP NAME: ALPHA IMPACT**

**NAME: DURR-E-SABEEH, HUMA NADIR, AQSA NAWAZ, AREEB AHMED**

**REG. NO: FA19-BSE-062, FA19-BSE-068, FA19-BSE-036, FA19-BSE-022**

**Assignment: 1**

1. Enable the project using any IDE for code review. It would be better to use Eclipse or Netbeans. You may use VS Code too.

Explore your project and find all options discussed in lecture 2 from Chapter 2. You need to prepare a google doc that highlights the potential issue that your code has. You may identify these issues with the help of

i) Identify deprecated technology or APIs. Just report the diagram.

ii) Potential issues missing in the documentation but available in the code

iii) missing of technical documentation (if applicable)

**Answer:**

These are the highlighted potential issue that our code has, and we identify these issues with the help of

deprecated technology or APIs:

* Not Understanding the User.
* Debugging.
* Keeping up with Technology.
* Communication.
* Time Estimation.
* Sitting for Hours.
* Security Threats.
* Working with Another Person’s Code.
* Group task issue.

In parallel with the access control logic for classes and members, the compiler looks for deprecated attributes of classes and members being accessed and issues warnings when deprecated classes or members are used.

The deprecation warning is suppressed if a compilation unit containing the deprecation is compiled at the same time as a compilation unit using a deprecated class or member. This allows you to build legacy APIs without warning. There is currently no other way to suppress the end of support notice.

When designing an API, carefully consider whether it replaces the old API. If so, and you want to encourage developers (API users) to migrate to the new API, then ditch the old API. Valid reasons for deprecating API support include:

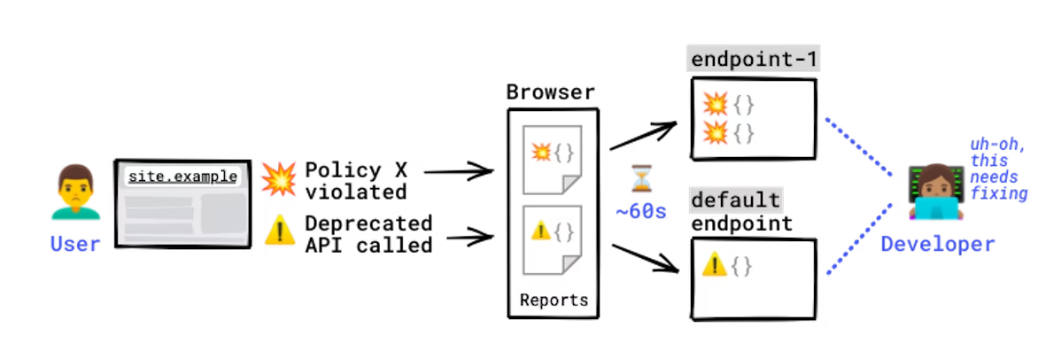
It’s insecure, buggy, or highly inefficient

It will disappear in a future version

It promotes bad coding practices

Ending support is a sensible choice in all these cases, as it preserves “backwards compatibility” while encouraging developers to move to the new API. End-of-support comments also help developers decide when to migrate to a new API, so they should briefly mention the technical reasons for end-of-support.

It is not necessary to drop individual member fields (properties) of a deprecated class, unless of course you want to make a specific point about the property.



**ii.**

Potential issues not come from audience analysis that comes from analysis of the claim which cause client might doubt claim**.** Potential issues missing in the code documentation are explain below:

**1.The code does not follow a step-by-step paradigm**

The code itself is non-linear. What appears at the top (eg variables) may not be implemented until the functions below work. Functions defined at the bottom can be executed inside other code blocks in the middle, and so on. When you get a piece of code to document, its build order is not at all obvious.

The central paradigm that most technical writers follow is the task-based model, where you start with step 1, 2, 3, and so on until you reach the end of the task. This is not the case with code documentation. Code is inherently non-linear. You cannot simply start at the top and work your way down. Although I tried to explain in parts in my code explanation, I had to skip some lines or notice that they were part of the logic explained in later parts.Overall, this non-linearity is very different from the procedural approach typically used in technical documents.

**2.The audience has a wide spread of technical levels**

Another challenge I faced was deciding what to explain and what to skip. Would the developers already be proficient in controllers in Lambda and Node JS? Or would it be new to them, like it was to me?

When documenting your code, you need to match the awareness and knowledge needs of your audience, even if their technical level varies dramatically. But when awareness and audience needs differ dramatically, we either overwrite for advanced developers by explaining the obvious, or alienate less experienced developers by assuming too much.

Implementing progressive disclosure models (where you reveal a bit of information and then let it grow with more details) can be tricky. Even if viewers are techies, there's no guarantee they have expertise in the specific technology you're documenting. As a result of all this obfuscation, we often think of ourselves as the audience.

**3.Code requires an understanding of a specific programming language**

Related to the above point about audience variability is the fact that we, the techies, often lack familiarity with the programming language. Or if we have some knowledge with programming, it is not always the right language for the project. So we're immediately at a disadvantage and have to jump into tutorials to understand the basics of what's going on in the code.

Additionally, we don't document the basics - we document how to implement the code in a specific context, often at an advanced level. Knowledge of how the code works is simply assumed. Basically, we have to enter an Advanced Calculus course and explain Legrange multipliers without taking anything more than the beginning of algebra.

**4.It takes a lot of maintenance to keep code samples working from release to release**

Another challenge is making sure the code samples work from release to release. I've included the full Lambda code to provide context, but over the course of a few weeks I've tweaked the code sample a bit. I then had to update the Lambda code as well as the section explanations. If you have dozens of code samples spread across your docs, maintaining that code from release to release will be difficult. How do you make it work? Are you separating the code from the narrative context so that it can be tested more regularly?Separating the code to allow testing sounds like a good idea, but once you separate the code from the conceptual explanation, you run the risk of someone updating the code in such a way that it no longer matches the explanation.

**5.Engineers need to know good versus bad code**

I should note that when I'm documenting code, I feel a bit like an outsider writing about a culture or country I don't belong to. Technical writers are often outside the engineering domain. If it wasn't for the developer, I might not even realize that the code is bad. Engineers live and breathe code, and many feel that code is poetry.

An efficient technique in code (eg recursive loops that expand resources as needed) can be beautiful and evoke an aesthetic in the engineer's mind. From my perspective as a technical writer, I'm unlikely to treat code with the same respect and awe. My mundane approach to code can make it difficult to resonate with developer users.

**iii.**

Effectively written documents help the intended audience by educating them on the details that are necessary.Creating software without documentation can lead to a loss of focus, underestimation of tasks, and in the worst cases, total project failure. Having such a document ensures that the development team is going in the right direction.

Missing of technical documentation cause potential issues in

* product documentation
* process documentation
* user documentation

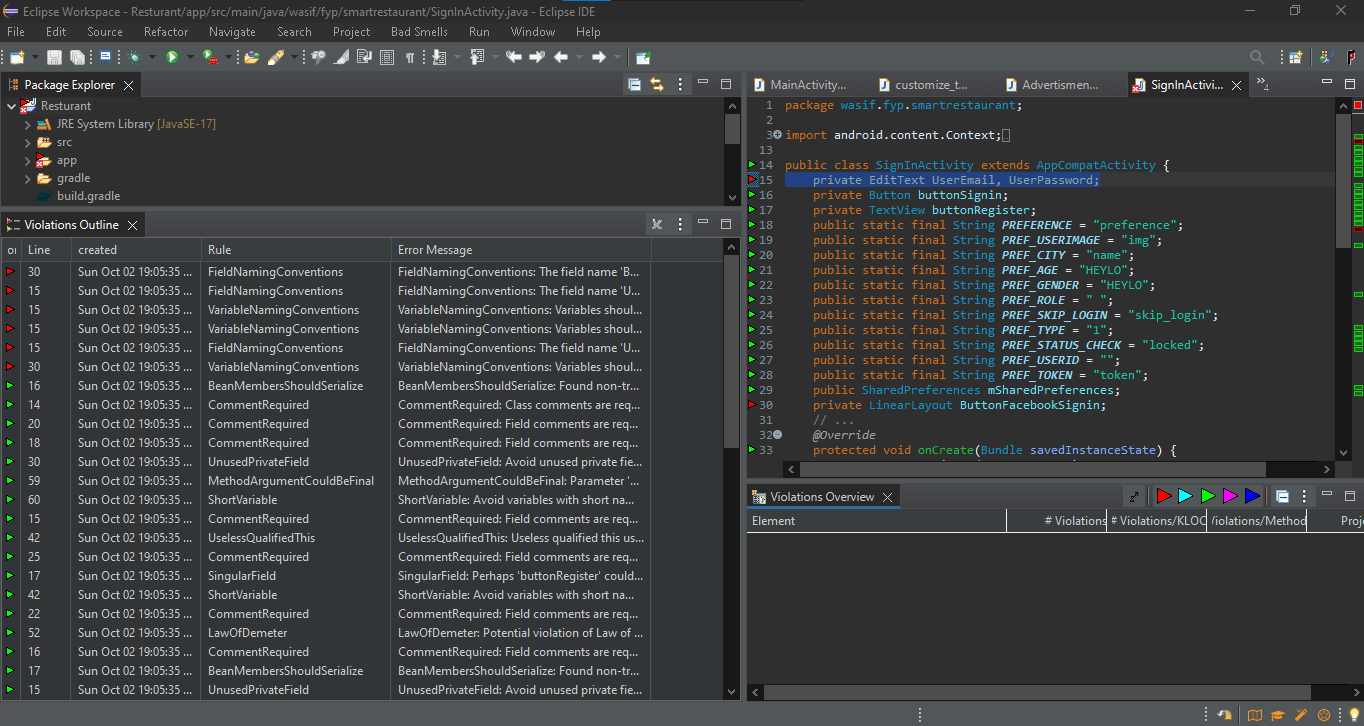
Whether we are talking about a small business or a large corporation, technical documentation is valuable to support the entire software development life cycle (SDLC). Contrary to what most people believe, creating a website or app is a complicated and complicated process. The time when developers were alone in a dark room producing something that people couldn't even read belongs to fictional stories. Currently, the developers are actively working with the design, marketing and sales departments.

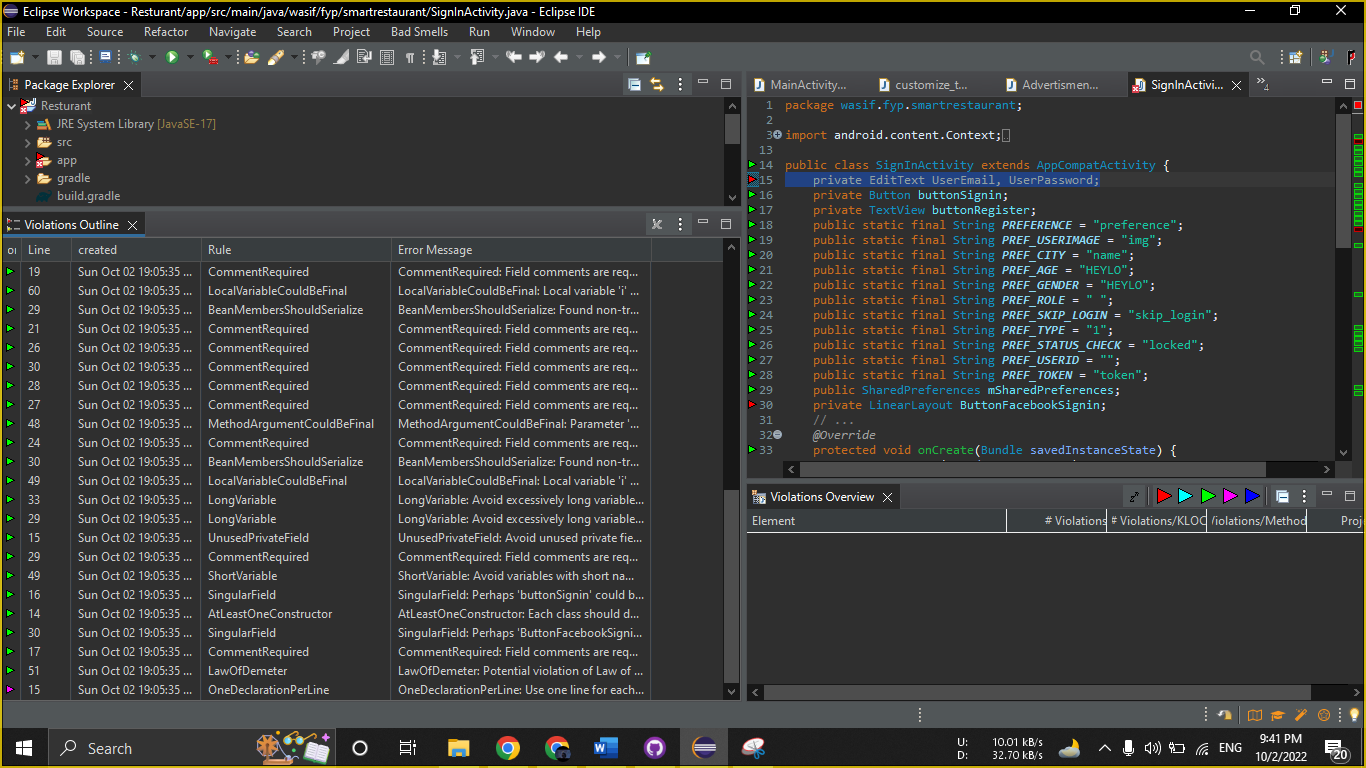
The technical documentation exists to make life easier for everyone throughout the process. It explains product functionality, unifies project-related information, and facilitates communication between developers, stakeholders, and team members.

b) Use PMD to help identify potential coding errors and customize the rules you use to make sure only pertinent rules are applied to your source code.

1. Create & view code issues directly from your editor
2. Track & prioritize code improvements like technical debt
3. Check your code quality
4. Apply at least 3 PMD rules with the help of tool
5. Generate the Abstract Syntax Tree of your source code suing PMD.

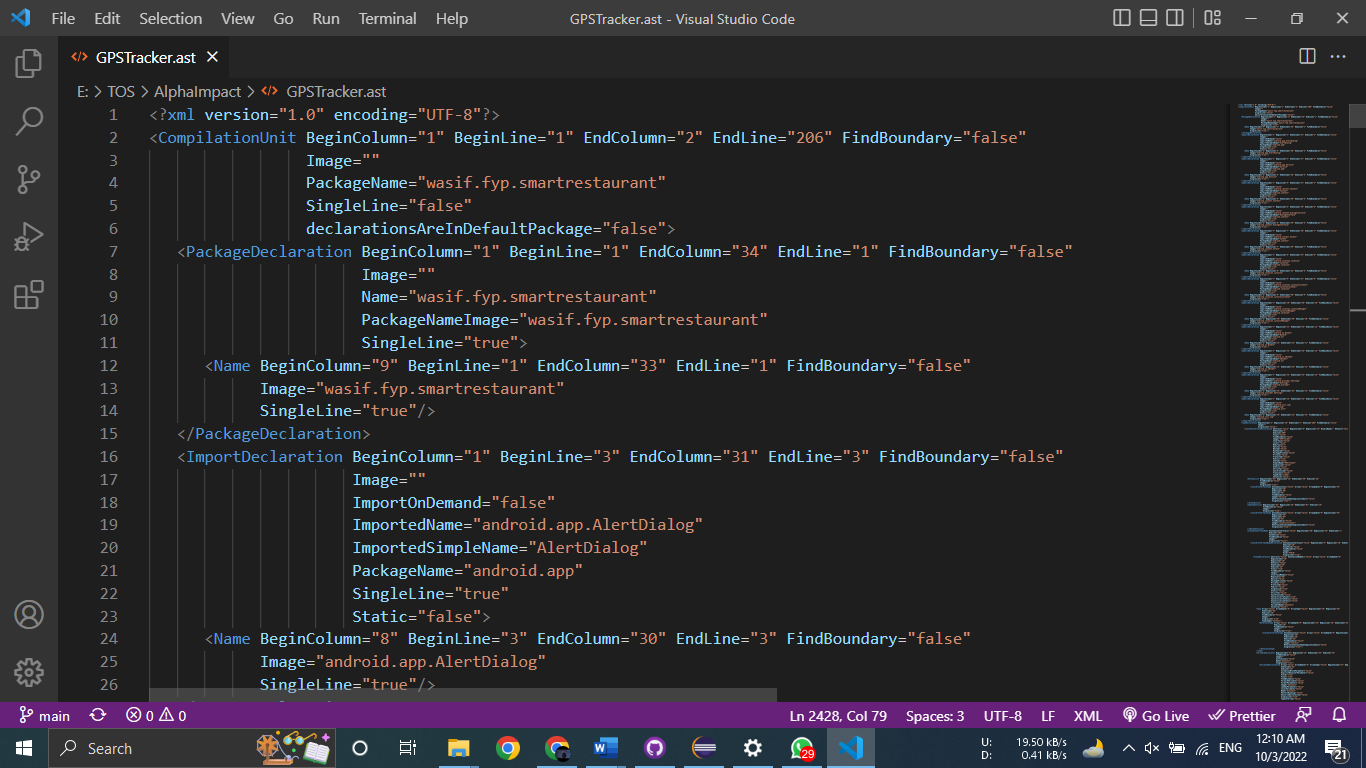
**Answer:**

****

****

* According to PMD rules, **blocker violation** occurs in this java file naming **FieldNamingConventions.**
* According to PMD rules, **urgent violation** occurs in this java file naming **UnusedPrivateField.**
* According to PMD rules, **Important violation** occurs in this java file naming **OneDeclarationPerLine.**

**Abstract Syntax Tree:**



C) Use the Check Style tool to review your code. At least apply 2 rules on your code using CHeckStyle.

**Answer:**

* Abbreviation in name 'isGPSEnabled' must contain no more than '1' consecutive capital letters.
* Location cannot be resolved to a type
* Member name 'mContext' must match pattern '^[a-z][a-z0-9][a-zA-Z0-9]\*$'.

