Regarding technical debt, students who are doing software development should be familiar with the following scenarios:

In order to dare to progress the project, detailed design, unit testing and other processes will not be written, and will be supplemented later.

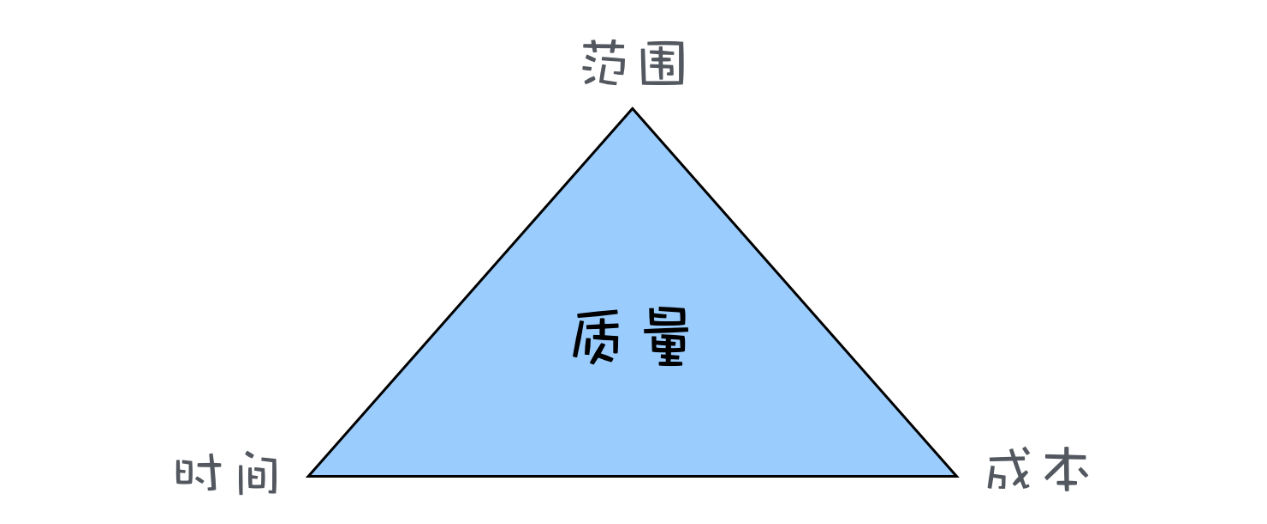
The demand has changed a lot. The original architecture design can't meet the new requirements, but I don't want to move the architecture, so I bypass the architecture design and add new code.

Old system, no documentation, no comments, difficult maintenance

As above, if we don't correct it in time, we will have more and more debts, which will lead to bloated code and inefficient system. We can use the term technical debt to describe the quality of the system.

 So what is technical debt?

  We know the triangle diagram, time, cost and scope of software quality when we are studying project management, as shown below:



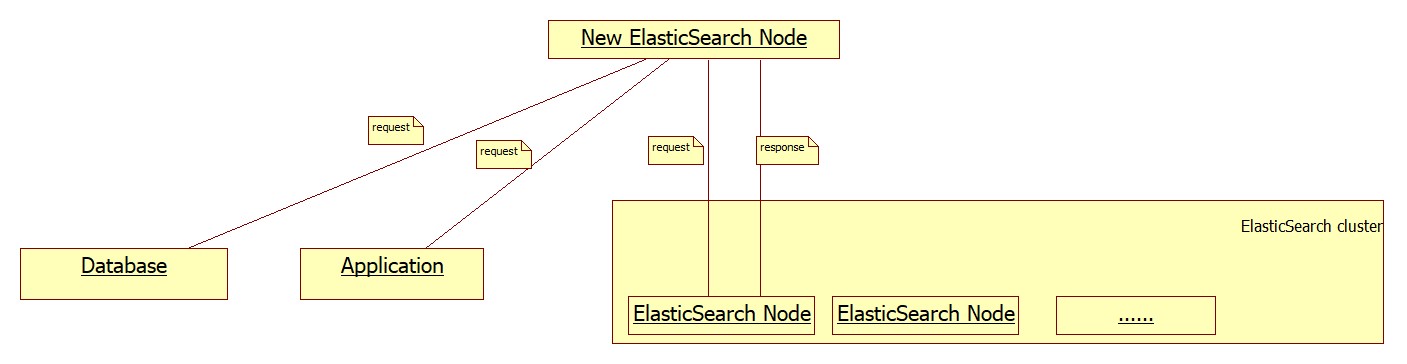
 Why is the quality placed in the middle of the triangle? Because quality is the result of the balance of the other three factors. For example, the scope is not reduced, the cost is not increased, and you want to save time and take shortcuts. Obviously, it will affect the quality. This quality is not only the product quality, but also the quality of the structure and the quality of the code. The overdraft of our quality is a kind of debt, and technical debt is the overdraft of the quality of the architecture and the quality of the code in the software project. Therefore, the technical debt is interest-bearing, and the interest on the debt is the additional time cost when the software is newly modified. Of course, technical debt is not necessarily bad. In software projects, technical debts are often deliberately owed, the development speed of short-term development is increased, software is quickly launched, and the market is seized. There are also models like rapid development, which owe technical debt. Quick verification of the method, even if the verification is not feasible, the technical debt does not need to be repaid.

Then we transfer the problem to elasticsearch. First, let's take a look at the basic content of elasticsearch.

 Elasticsearch is an open source, highly scalable distributed full-text search engine that can store and retrieve data in near real-time. It is very scalable and can be extended to hundreds of servers to process PB-level data. Elasticsearch also uses Java to develop and use Lucene as its core to implement all indexing and search functions, but its purpose is to hide the complexity of Lucene through a simple RESTful API, making full-text search simple.

ES mainly solves three problems: 1) Retrieving relevant data; 2) Returning statistical results; 3) Retrieving faster.

When ElasticSearch's node is started, it uses multicast (or unicast if the user changes the configuration) to find and connect to other nodes in the cluster. This process is shown below:



In the development of Elasticsearch, many technical debts were inevitably generated. Technical debt needs to be managed and maintained at a reasonable level to ensure that developers can continue to contribute to the project. Solving, avoiding technical debt can maintain alternative maintenance costs, improve prevention and prevent accidental accidents. In this section, the technical debt that exists in the Arduino project will be analyzed.

SonarQube is a platform for continuous software quality monitoring. This platform is able to analyse a large software project like Gradle in a matter of minutes. It provides the user with crucial insights about bugs, security and technical debt within the project. We put the project on SonarQube for analysis and got the following results: SonarQube reported a large number of errors (162), vulnerabilities (263) and code smells (8400), the number of duplicate codes was 1.6% of the code, and the repeating block 493. SonarQube identified 280 days of technical debt, meaning that fixing all of the mentioned violations would take 280 working days, approximately 9 months. But after inquiry, we found that some of the debts monitored by SonarQube are actually the code style choices that the Elasticsearch team made during its development cycle, and therefore do not constitute actual technical debt. or example, the biggest contributor to violations in Elasticsearch is the Boolean comparison, which results in 1380 code smells. The reason for the detection is that the developer compares all Boolean comparisons to the format a==true instead of if(a). The latter is a format we usually use, because it is more concise. But using the former makes the code more readable, which is a reasonable choice for the Elasticsearch development team.

After browsing the code we found that there is a lot of extra code in Elasticsearch. SonarQube found 117 unused method parameters and 237 unused import statements. This extra code makes the code less readable. But cleaning up such code doesn't take much time. The Elasticsearch developers have a tendency to name String literals with an all-caps name, such as String FOO. These literals mostly represent the name of a certain field in a class, e.g. String FOO = "foo". These literals are used in so-called xContent methods, which deal with different types (X types) of content that needs to be written to an output stream. Instead of hardcoding the actual strings in such an xContent method, the all-caps String definitions are used. This causes a number of classes to contain both the normal field foo, which is named in lower-case, as well as the xContent field FOO, named in upper-case. Although this is a deliberate choice made by the developers, SonarQube flags this as a blocker type issue, meaning an immediate fix is preferred.

Lack of instrumentation is also a form of technical debt. We found that Elasticsearch does miss some important static code analysis tools, such as PMD and FindBugs, and does not make optimal use of CheckStyle. Elasticsearch uses Gradle as their build system and allows users to execute the static code checks using ./gradlew precommit. They employ CheckStyle to verify code style conventions, although their CheckStyle configuration is not very extensive. They only check a small number of rules, such as avoiding star imports, naming files in compliance with the classes they contain, and avoiding empty Javadoc comments. However, they do not check if there are any Javadoc comments at all, nor do they check method length, class length, method and class complexity, or proper indentation. What is even more interesting is their check for line length: their CheckStyle configuration defines a maximum line length of 140 characters, but the colocated CheckStyle suppressions file suppresses this check for every file that does not pass it until these files start to pass the check, as stated in the documentation.

Also, as we explore the code base, we noticed that many methods and classes are missing Javadoc. Through the survey, we found an Elasticsearch member, and he responded that he is happy with the current situation regarding Javadoc, also seeing as they had "almost no Javadoc 4 years ago". He makes it clear that they have recently started being "much more diligent about adding Javadocs to public methods whenever we add them or touch existing ones, also enforcing that habit as part of the review process". While this does show that there are some efforts being made to reduce Javadoc debt, it also becomes clear that the main focus is to keep the number of uncommented methods as small as possible in the public API, indicating that the Elasticsearch developers focus on the user experience first and on the developer experience second.

Especially for large and distributed projects such as Elasticsearch, it is important that software is properly tested. Weak tests or low test coverage are therefore also part of the technical debt of a system. The testing documentation stated until recently that a test coverage report could be generated using mvn -Dtests.coverage test jacoco:report. However, this is a Maven command, while Elasticsearch migrated to Gradle in October 2015. As this documentation was not updated since, this may indicate that the developers do not pay much (if any) attention to their test coverage. After more experimenting, we concluded that RandomizedTestingTask would need a significant rewrite by someone with sufficient experience so as to keep the required existing functionalities, though we did update the outdated documentation.+ Having a build configuration unable to support testing instrumentation such as JaCoCo is a major technical debt, as it also causes problems upon integrating other instrumentation, such as SonarQube or mutation testing tools.