Before you start developing small or big software, you should have answer for the following questions:

- Where is the **Requirements** Specification?
- Where is the **Impact Analysis** Document?
- Where is the **Design** Document?
- Have you documented all the **assumptions**, **limitations** properly?
- Have you done review of all the documents?
- Did you get sign off on all the documents from all the stakeholders?

Once you have positive answers for all the above questions, you are safe and ready to proceed for the coding.

What you learn today, prepares you for tomorrow!

So, again it is one of the best practices to have documentation as much as possible. Few important documents, which will prepare you for future are:

- Design Approaches
- Tips and Tricks
- Special functions, commands and instructions
- Lessons learnt
- Peculiar situations
- Debugging methods
- Best Practices
- Anything which can help you in future

Keeping documents electronically does not cost you.

Code should be written to be reviewed

While writing your software code, keep in mind that someone is going to review your code and you will have to face criticism about one or more of the following points but not limited to:

- Bad coding
- Not following standard
- Not keeping performance in mind
- History, Indentation, Comments are not appropriate.
- Readability is poor
- Open files are not closed
- Allocated memory has not been released
- Too many global variables.
- Too much hard coding.
- Poor error handling.
- No modularity.
- Repeated code.

Software Design Principles

Software design has always been the most important phase in the development cycle. The more time you put into designing a resilient and flexible architecture, the more time will save in the future when changes arise.
Requirements always change — software will become legacy if no features are added or maintained on regular basis — and the cost of these changes are determined based on the structure and architecture of the system. In this article, we'll discuss the key design principles that help in creating easily maintainable and extendable software.

A Practical Scenario

Suppose that your boss asks you to create an application that converts Word documents to PDFs. The task looks simple — all you have to do is to look up a reliable library that converts Word documents to PDFs and plug it in inside your application. After doing some research, say you ended up using the *Aspose.words* framework and created the following class:

```
/**
 * A utility class which converts a word document to PDF
 * @author Hussein
 *
 */
public class PDFConverter {
    /**
     * This method accepts as input the document to be converted and
     * returns the converted one.
     * @param fileBytes
     * @throws Exception
     */
    public byte[] convertToPDF(byte[] fileBytes) throws Exception {
        // We're sure that the input is always a WORD. So we just use
        //aspose.words framework and do the conversion.
        InputStream input = new ByteArrayInputStream(fileBytes);
        com.aspose.words.Document wordDocument = new com.aspose.words.Document(input);
        ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
        wordDocument.save(pdfDocument, SaveFormat.PDF);
        return pdfDocument.toByteArray();
    }
}
```

Life is easy and everything is going pretty well until now!

Requirements Change, as Always

After a few months, some client asks you to support Excel documents as well. So you did some research and decided to use *Aspose.cells*. Then, you go back to your class, add a new field called *documentType,* and modify your method like the following:

```
public class PDFConverter {
    // we didn't mess with the existing functionality, by default
    // the class will still convert WORD to PDF, unless the client sets
    // this field to EXCEL.
    public String documentType = "WORD";
    /**
     * This method accepts as input the document to be converted and
     * returns the converted one.
     * @param fileBytes
     * @throws Exception
     */
    public byte[] convertToPDF(byte[] fileBytes) throws Exception {
        if (documentType.equalsIgnoreCase("WORD")) {
            InputStream input = new ByteArrayInputStream(fileBytes);
            com.aspose.words.Document wordDocument = new
com.aspose.words.Document(input);
            ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
            wordDocument.save(pdfDocument, SaveFormat.PDF);
            return pdfDocument.toByteArray();
        } else {
            InputStream input = new ByteArrayInputStream(fileBytes);
            Workbook workbook = new Workbook(input);
            PdfSaveOptions saveOptions = new PdfSaveOptions();
            saveOptions.setCompliance(PdfCompliance.PDF_A_1_B);
            ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
            workbook.save(pdfDocument, saveOptions);
            return pdfDocument.toByteArray();
        }
    }
}
```

This code will work perfectly for the new client (and will still work as expected for the existing clients), but some bad design smells are starting to appear in the code. That means we're not doing this the perfect way, and we will not be able to modify our class easily when a new document type is requested.

1. **Code repetition:** As you see, similar code is being repeated inside the if/else block, and if we managed, someday, to support different extensions, then we will have a lot of repetitions. Also, if we decided later on, for example, to return a file instead of byte[], then we have to make the same change in all the blocks.
2. **Rigidity:** All the conversion algorithms are being coupled inside the same method, so there is a possibility that, if you change some algorithm, others will be affected.
3. **Immobility:** The above method depends directly on the *documentType* field. Some clients will forget to set the field before calling *convertToPDF()*, so they will not get the expected result. Also, we're not able to reuse the method in any other project because of its dependency on the field.
4. **Coupling between the high-level module and the frameworks:** If we decide later on, for some purpose, to replace the Aspose framework with a more reliable one, we will end up modifying the whole *PDFConverter* class — and many clients will be affected.

Doing It the Right Way

Normally, developers are not able to predict future changes, so most would implement the application exactly as we implemented it the first time. However, after the first change, the picture becomes clear that similar future changes will arise. So, instead of hacking it with an if/else block, good developers will do it the right way in order to minimize the cost of future changes. So, we create an abstract layer between our exposed tool *(PDFConverter)* and the low-level conversion algorithms, and we move every algorithm into a separate class as follows:

```
/**
 * This interface represents an abstract algorithm for converting
 * any type of document to a PDF. @author Hussein
 */
public interface Converter {
    public byte[] convertToPDF(byte[] fileBytes) throws Exception;
}
/**
 * This class holds the algorithm for converting Excel
 * documents to PDFs.
 * @author Hussein
 *
 */
public class ExcelPDFConverter implements Converter {
    public byte[] convertToPDF(byte[] fileBytes) throws Exception {
        InputStream input = new ByteArrayInputStream(fileBytes);
        Workbook workbook = new Workbook(input);
        PdfSaveOptions saveOptions = new PdfSaveOptions();
        saveOptions.setCompliance(PdfCompliance.PDF_A_1_B);
        ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
        workbook.save(pdfDocument, saveOptions);
        return pdfDocument.toByteArray();
    };
}
/**
 * This class holds the algorithm for converting Word
 * documents to PDFs. * @author Hussein
 */
public class WordPDFConverter implements Converter {
    @Override
    public byte[] convertToPDF(byte[] fileBytes) throws Exception {
        InputStream input = new ByteArrayInputStream(fileBytes);
        com.aspose.words.Document wordDocument = new com.aspose.words.Document(input);
        ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
        wordDocument.save(pdfDocument, SaveFormat.PDF);
        return pdfDocument.toByteArray();
    }
}
public class PDFConverter {
    /**
     * This method accepts the document to be converted as an input and
     * returns the converted one.
     * @param fileBytes
     * @throws Exception
     */
    public byte[] convertToPDF(Converter converter, byte[] fileBytes) throws Exception {
        return converter.convertToPDF(fileBytes);
    }
}
```

We force the client to decide which conversion algorithm to use when calling *convertToPDF()*.

What Are the Advantages of Doing It This way?

1. **Separation of concerns (high cohesion/low coupling)**: The *PDFConverter* class now knows nothing about the conversion algorithms used in the application. Its main concern is to serve the clients with the various conversion features regardless how the conversion is being done. Now that we're able to replace our low-level conversion framework, no one would even know as long as we're returning the expected result.
2. **Single responsibility:** After creating an abstract layer and moving each dynamic behavior to a separate class, we actually removed the multiple responsibilities that the *convertToPDF()* method previously had in the initial design. Now it just has a single responsibility, which is delegating client requests to the abstract conversion layer. Also, each concrete class of the *Converter* interface has now a single responsibility related to converting some document type to a PDF. As a result, each component has one reason to be modified, hence no regressions.
3. **Open/Closed application:** Our application is now opened for extension and closed for modification. Whenever we want to add support for some document type, we just create a new concrete class from the *Converter* interface and the new type will become supported without the need to modify the *PDFConverter* tool, since our tool now depends on abstraction.

Design Principles Learned from This Article

The following are some best design practices to follow when building your application's architecture.

1. **Divide** your application into **several modules** and **add** an **abstract layer** at the top of each module.
2. **Favor abstraction** over implementation: **Always** make sure to **depend on the abstraction layer**. This will make your application open for future extensions. The **abstraction** should be applied on the **dynamic parts of the application** *(*which are most likely to be changed regularly*)* and not necessarily on every part, since it complicates your code in case of overuse.
3. **Identify** the aspects of your application that vary and **separate** them from what stays the same.
4. **Don't repeat yourself**: Always put duplicate functionalities in some utility class and make it accessible through the whole application. This will make your modification a lot easier.
5. **Hide** low-level implementation through the abstract layer: Low-level modules have a very high possibility to be changed regularly, so separate them from high-level modules.
6. Each class/method/module should have one reason to be changed, so always give a single responsibility for each of them in order to minimize regressions.
7. Separation of concerns: Each module knows what another module does, but it should never know how to does it.

A Checklist for setting up a Java-based Software Architecture

I recently had the opportunity to propose the architecture for a new large-scale software project. Finding the technologies that best fit to the customer's requirements, may come up with a solution that is future-proof and scales with expectations.
To ease this task in the future, I compiled a list of concerns you should think about when setting up a new software project.
Note that this list is not complete and weighs heavily towards Java technologies, so you should use this list with the caution that the task of selecting the best architecture for your customer deserves.
I hope this helps any readers who are setting up a new Java-based software architecture.

Architecture Style
Which should be the basic architecture style of the application? There are of course more styles that are listed here. However, monoliths and micro-services seem to be the most discussed architecture styles these days.

Monolithic	A monolithic architecture contains all of its functionality in a single deployment unit. Might not support a flexible release cycle but doesn't need potentially fragile distributed communication.
(Micro-) Services	Multiple, smaller deployment units that make use of distributed communication to implement the application's functionality. May be more flexible for creating smaller and faster releases and scales with multiple teams but comes at the cost of distributed communication problems.

Back-End Concerns

What things should you think about that concern the back-end of the application you want to build?

Logging	Use SLF4J with either Logback or Log4J2 underneath. Not really much to think on, nowadays. You should however think about using a central log server (we will come to that later).
Application Server	Where should the software be hosted? A distributed architecture may work well with Spring Boot while a monolithic architecture might better be served from a full-fledged application server like Wildfly. Choice of application server is often predetermined for you, since corporate operations like to define a default server for all applications they have to run.
Job Execution	Almost every medium-to-large sized application will need to execute scheduled jobs

	like cleaning up a database or batch-importing third-party data. Spring offers basic job scheduling features. For more sophisticated needs, you may want to use Quartz, which integrates into a Spring application nicely as well.
Database Refactoring	You should think about how to update the structure of your relational database between two versions of your software. In small projects, manual execution of SQL scripts may be acceptable, in medium-to-large projects you may want to use a database refactoring framework like Flyway or Liquibase (see my previous blog post. If you are using a schemaless database you don't really need a database refactoring framework (you should still think about which changes you can do to your data in order to stay backwards-compatible, though).
API Technology	Especially when building a distributed architecture, you need to think about how your deployment units communicate with each other. They may communicate asynchronously via a messaging middleware like Kafka or synchronously, for example via REST using Spring MVC and Feign.
API Documentation	The internal and external APIs you create must be documented in some form. For REST APIs you may use Swagger's heavy-weight annotations or use Spring Rest Docs for a more flexible (but more manual) approach (see my previous blog post). When using no framework at all, document your APIs by hand using a markup format like Markdown or Asciidoctor.
Measuring Metrics	Are there any metrics like thoughput that should be measured while the application is running? Use a metric framework like Dropwizard Metrics (see my previous blog post) or the Prometheus Java Client.
Authentication	How will users of the application prove that they are who they claim to be? Will users be asked to provide username and password or are there additional credentials to check? With a client-side single page app, you need to issue some kind of token like in OAuth or JWT (also see this blog post about OpenID). In other web apps, a session id cookie may be enough.
Authorization	Once authenticated, how will the application check what the user is allowed to do and what is prohibited? On the server side, Spring Security is a framework that supports implementation of different authorization mechanisms.
Database Technology	Does the application need a structured, schema-based database? Use a relational database. Is it storing document-based structures? Use MongoDB. Key-Value Pairs? Redis. Graphs? Neo4J.

Persistence Layer	When using a relational database, Hibernate is the de-facto default technology to map your objects into the database. You may want to use Spring Data JPA on top of Hibernate for easy creation of repository classes. Spring Data JPA also supports many NoSQL databases like Neo4J or MongoDB. However, there are alternative database-accessing technologies like iBatis and jOOQ.

Frontend Concerns

What concerns are there to think about that affect the frontend architecture?

Frontend Technology	Is the application required to be hosted centrally as a web application or should it be a fat client? If a web application, will it be a client-side single page app (use Angular) or a server-side web framework (I would propose using Apache Wicket or Thymeleaf / Spring MVC over frameworks like JSF or Vaadin, unless you have a very good reason). If a fat client, are the requirements in favor of a Swing or JavaFX-based client or something completely different like Electron?
Client-side Database	Do the clients need to store data? In a web application you can use Local Storage and IndexedDB. In a fat client you can use some small-footprint database like Derby.
Peripheral Devices	Do the clients need access to some kind of peripheral devices like card readers, authentication dongles or any hardware that does external measurements of some sort? In a fat client you may access those devices directly, in a web application you may have to provide a small client app which accesses the devices and makes their data available via a http server on localhost which can be integrated into the web app within the browser.
Design Framework	How will the client app be layouted and designed? In a HTML-based client, you may want to use a framework like Bootstrap. For fat clients, the available technologies may differ drastically.
Measuring Metrics	Are there any events (errors, client version, ...) that the client should report to a central server? How will those events be communicated to the server?
Offline Mode	Are the clients required to work offline? Which use cases should be available offline and which not? How will client side data be synchronized with the server once the client is online?

Operations Concerns

What you should discuss with the operations team before proposing your architecture to anyone.

Servers	Will the application be hosted on real hardware or on virtual machines? Docker is a popular choice for virtualization nowadays.
Network Infrastructure	How is the network setup? Are there any communication obstacles between different parts of the application or between the application and third party applications?
Load Balancing	How will the load on the application be balanced between multiple instances of the software? Is there a hardware load balancer? Does it have to support sticky sessions? Does the app need a reverse proxy that routes requests to different deployment units of the application (you may want to use Zuul)?
Monitoring	How is the health of the server instances monitored and alarmed (Icinga may be a fitting tool)? Who will be alarmed? Should there be a central dashboard where all kinds of metrics like thoughput etc. are measured (Prometheus + Grafana may be the tools of choice).
Service Registry	When building a (Micro-)Service Architecture, you may need a central registry for your services so that they find each other. Eureka and its integration in Spring Boot may be a tool to look into.
Central Log Server	Especially in a distributed architecture with many deployment units, but also in a monolithic application (which also should have at least two instances running), a central log server may make bug hunting easier. The Elastic Stack (Elastic Search, Logstash, Kibana) is popular, but heavy to set up. Graylog 2 is an alternative.
Database Operations	What are the requirements towards the database? Does it need to support hot failover and / or load balancing between database instance? Does it need online backup? Oracle RAC is a pretty default (but expensive) technology here, but other databases support similar requirements.

Development Concerns

Things that the whole development team must deal with every day. Discuss these points with the development team before starting development.

IDE	What's the policy on using IDE's? Is each developer allowed to use his/her IDE of choice? Making a specific IDE mandatory may reduce costs for providing several parallel solutions while letting each developer use his favorite IDE may reduce training costs. I'm a follower of IntelliJ and would try to convert all Eclipsians when starting a new project ;).
Build Tool	Which tool will do the building? Both Maven and Gradle are popular choices, while I would chose Gradle for it's customizability in form of Groovy Code.
Unit Testing	Which parts of the code should be unit tested? Which frameworks will be used for this? JUnit4 and Mockito are a reasonable starting point (note that JUnit 5 is currently on the way).
End-to-End Tests	Which parts of the code should be tested with automated end-to-end tests? Selenium is a popular choice to remote control a browser. When working on a single page application with Angular and angular-cli, Protractor is setup by default. Have a look at this blog post for a proposal on how to create end-to-end tests with Selenium while still having access to the database internals.
Version Control	Where will the source code be hosted? Git is quickly becoming the de-facto standard, but Subversion has a better learning curve.
Coding Conventions	How are classes and variables named? Is code and javadoc in english or any other language? I would propose to choose an existing code formatter and a Checkstylerule set and include it as a build breaker into the build process to make sure that only code that adheres to your conventions are committed to the code base.
Code Quality	How will you measure code quality? Are the coding conventions enough or will you run additional metrics on the code? How will those metrics be made visible? You may want to setup a central code quality server like SonarQube for all to access.
Code Reviews	Will you perform code reviews during development (I highly recommend this)? How will thos code reviews be supported by software? There are code review tools like Review Board. Some version control tools like GitLab support workflows in which each user works on his own branch until he is ready to merge his changes. A merge request is a perfect opportunity for code reviews.
Continuous Integration	How will the build process be executed on a regular basis? There are cloud providers

	like CircleCI or Travis or you may install a local Jenkins server.
Continuous Deployment	Are there automatic tasks that deploy your application to a development, staging or production environment? How will these tasks be executed?
Logging Guidelines	Which information should be logged when? You should provide a guideline for developers to help them include the most valuable information in the log files.
Documentation	Which parts of the application should be documented how? What information should be documented in a wiki like Confluence and what should be put into Word documents? If there is a chance, use a markup format like Markdown ord AsciiDoc instead of Word.