Software Reuse

Ruben Teimas

Universidade de Évora, Portugal
m47753@alunos.uevora.pt

Abstract. This essay starts by explaining what *software reuse* is and its motivation. It also defines what services are and establishes a bridge between the appearance of *service-based software* and *software-reuse*. As it is impossible to make good use of *software reuse* without *software testing*, this last subject is also approached in the essay. To finish, the essay describes the steps to assure the consistency of a *service-oriented architecture*.

Keywords: Software Engineering · Service-based Software · Software Reuse · Software Testing

1 Introduction

The appearance of the web changed the world, it made everything accessible at the distance of the click. This phenomenon has been making the world "move faster" with massive breakthroughs happening in shorter periods of time therefore making the people more impatient.

This has also affected software, with people being more impatient the deadlines tend to be shorter. Besides that, software's complexity has been growing exponentially over the years.

The increasing complexity and shorter deadlines created the need to reuse software. Software reuse can happen at different scales: from reusing a function to reusing an application.

This notion powered an architecture, which was one of the main architectures adopted between 2002-2014, the *Service-oriented architecture*.

2 Software Reuse

2.1 What is it?

Software Reuse is the process of creating new software from existing one, rather than building all of it from scratch. There can be software reuse at different scales [Som16]. It can go from simply reusing a function or a class, to reusing a whole application or system.

2.2 Emergence and Motivation

Like many computer-science concepts, *Software Reuse* appeared for the first time around the *70's*. Back then it was not a need, because the programs were much less complex than they are now.

Not only the software complexity was much smaller than it is now but the high level languages at the time, like *C*, *Lisp* and *ML*, were not very friendly for software reuse practices, as they had not been designed for it.

With the complexity of software growing exponentially, this concept continued to be developed throughout the time but it wasn't until the *90's* that *Software Reuse* started to establish. For this to happen, the concept of *Object-Oriented* was crucial. With the appearance of programming languages like *Java*, *C++*, *Perl* and even *Common Lisp* it was now possible to put in practice the ideas behind *software-reuse* because all of these languages had been designed with that in mind.

The increasingly use of *Software Reuse* practices showed a increase in software engineer's productivity [MC07], from which Software Companys would benefit a lot. At this point these companys realized *Software Reuse* saved them time, by having smaller development cycles, and consequently, saved them money.

2.3 Benefits

The emergence and permanence of software reuse is only possible due to the benefits it brings.

By using a *reuse-based architecture* the development team can expect a reduced development time and effective use of resources. This is possible because the team does not need to spend time developing every part of the system, instead, they will use well tested and suited pieces of software. By doing so, they can also focus on their prime goal which usually are the features they want to add on top of well know and solid pieces of software.

A practical example of *Software Reuse* usage happens with the Linux Distributions. All of them use the *Linux Kernel*, instead of writing their own, and focus on their personal vision for the user.

Another, more specific, benefit from software reuse are the standards compliance. This is particularly useful when building user interfaces, where the developer can reuse components that will be laid out throughout the experience, therefore making it easier for the developer and more familiar to the user.

2.4 Disadvantages

Even though there are many advantages, *reuse-based architectures* are not perfect.

Problems might arise when designing the architecture and integration of reusable components, because this type of tasks are not always straight forward. [Som16]

When the software reused is external to a company/team there might also be lack of support tools. This can happen when the source code of the software is not public (the system is being integrated via an *API*) and some specification changes throughout the project or the software being integrated suffers changes that do not go along with the the company's project.

2.5 In today's world

Reusable software pushed software engineers and architects to learn and apply *Design Patterns*. Nowadays it is impossible to talk about one without mentioning the other. It's not for nothing that *Clean Code* is one of the most recommended books to software engineers and developers.

There are many examples of Software Reuse in today's world but one the most present are *Software Frameworks*. A framework can be defined as an abstraction of components providing generic functionalities that can be extend in order to suit the developer's needs.

3 Software development based on services

3.1 Services, what are they?

"Services are reusable components that are independent and loosely coupled." [Som16]

3.2 Motivation

This development style appeared as a need to re-utilize software that had would appear in different systems.

Until recently, most software systems were built following a *Monolithic Architecture*, which forced the different components to be tightly coupled. For that reason it would be impossible to use a component outside of its system, creating the need for the developers to re-write the same code as they had previously written.

For instance, if we had 2 applications and both needed a scheduling component, if we had a monolithic architecture we would need to write the component

on both applications. However, if we opted for a *Services Oriented Architecture* (*SOA*) we could simply reuse our scheduling service on both applications.

A *Service-Oriented Architecture* should be confused with a *Micro-services Architecture* (*MSA*. They both share some similarities, but they also differ a lot. For instance, in *MSA* the focus tends to be on the service design itself while on *SOA* the focus is on the integration between services.[Com]

3.3 Benefits

Besides the obvious reasons, which are inherited from *Software Reuse*, like reduced costs and development time, there are other advantages.

Since these services often communicate over external data representation (*XDR*) the services can be written using different programming languages, which can be very convenient.

Since the services are loosely-coupled, they do not need to run on the same machine as another service, therefore, they can be deployed in another machine from another provider, according to the need.

4 Software Testing

Software is developed to complete a task/group of tasks and for it to be useful we need to know that it functions properly, which means, it has to be tested.

In order to properly test software, the tests must be exhaustive and try to validate as many cases as possible, however, there are countless possibilities which causes bugs to occasionally appear, even when different types of tests are being well applied.

Usually, there are several testing stages. The first one is the development testing where the system is tested by the developers in order to discover bugs. The second stage happens when the developing teams understands that the product is viable and is ready to be released to the user, in this stage the testing team tests the software and assures if the software is, in fact, ready to be released. The final stage is the *user testing* stage, in which a sample of users test the software in order to analyse its performance and consistency.

4.1 Development Testing

The first stage of development is carried out by the development team and can be done, mainly in 2 different ways, either by writing the tests after writing the software or by writing the test before.

When writing tests after the code there are 3 stages of testing. The first one is called *Unit testing*, in it methods and classes are tested. The second stage is the *Component testing*, where the units, tested previously, that form a component are tested together in order to be validated. The final stage is the *System testing* where the components are tested together.

The second way of writing development tests is very common and is called *Test-Driven Development* (*TDD*). In it, the tests are written before the code and passing the tests is the driver of development. The code is developed incrementally and the new functionalities of the software are only written after the existing ones have passed the tests. This test methodology ensures a better code coverage by ensuring that each functionality has at least one test associated to it. It also simplifies the process of debugging because most likely the problems will come from the last part of written software. Another interesting characteristic of this methodology is the *regression testing*, which consists of testing software to assure that the changes do not affect previously working code.

Regression testing can be expensive when made manually, however, most of the testing these days is automated.

4.2 Release testing

It is performed by a specialized testing team which should not be involved in the development stage.

While the development testing should be concerned in assuring that there are no bugs and the software does what the team expects to, the release testing focus on verifying if the software meets the specified requirements, rather than find bugs.

This type of tests consist in identifying requirements from the users specifications, create scenarios for those requirements and then tests the respective features for those scenarios. This process is also known as *Behaviour-Development Testing* (*BDD*).

4.3 User testing

In the final testing stage the software is provided to a sample of user and those user provide feedback about the software.

This stage of testing is essential because the user's working environment can affect the system's performance, reliability and experience, for that reason it is important to try it in many environment as possible, which means picking a good user sample.

Even inside this stage there are other stages: the *Alpha*, *Beta* and *Acceptance* testing stages. In the *alpha* stage the users test the system in the company infrastructure. The *beta* stage already allows the users to test the product on their own environment and raise any problems they find. The final (*acceptance*) stage let the "user decides" weather the software is ready to be deployed or not.

5 Service-based systems and Testing

All the concepts presented in this paper walk side-by-side. Service-based systems are an application of *Software Reuse* and *Software Reuse* would not be possible without *Software Testing*, otherwise we could be potently propagating errors across multiple systems, which would increase the development type and the costs, defeating some of the *Software Reuse* benefits.

However, testing a *SOA* can be hard for several reasons: it is difficult to simulate testing environments, we might want to test services from other vendors and many other reasons. [Gur]

Because of that, *SOA* testing revolves around 3 layers in the architecture: the service level, process level and consumer level. The service level consists mainly in service's testing, functional testing, security testing and performance testing for each service. The process level tests the service the form each process and assure they work properly together. At last, the consumer level focus in testing the whole system.

6 SOA life-cycle consistency

In order to assure service's consistency during its life-cycle, some steps must be taken. [Erl04]

6.1 Service-oriented Analysis

It is in this initial stage that service layers are mapped out, and individual services are modeled as service candidates that comprise a preliminary SOA.

6.2 Service-oriented design

When we know what it is we want to build, we need to determine how it should be constructed. Service-oriented design is the process of designing the services and interaction between them.

6.3 Service development

After having our services designed we must implement them according to the design.

6.4 Service testing

In order to assure that our services work properly and do what we expected them to do, they must be tested following the *SOA* testing methodologies.

6.5 Service deployment

This is the stage where are our architecture really starts taking form. This is a very important step because there are many things to be concerned, like how will the services be distributed, how will we integrate new services to the already existing one, how should services used by multiple solutions be deployed...

6.6 Service administration

After the services are deployed we need to keep track of what is happening with them. This can be done by monitoring the services. Monitoring can be useful not only to minimize failures, by creating alerts, but also to evaluate performance (for this to happen, metrics must be added to the services before deploying them).

7 Conclusion

Service-based software has been essential to software development and allowed us to come to where we are now. This type of software development would not have been possible without making use of software reuse and software testing.

In order to take all the benefits from this concepts we must spend some time thinking about our architecture, infrastructure and delivery strategy. Otherwise we might not really having more problems than the one we try to solve.

Nonetheless, if well applied *SOA* proves to be a solid and robust solution to today's systems even though some statistics show it losing space to another architecture: *Micro-services*.

References

[Erl04] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. 2004. ISBN: 9780131858580.

[MC07] Parastoo Mohagheghi and Reidar Conradi. "Quality, productivity and economic benefits of software reuse: A review of industrial studies". In: *Empirical Software Engineering* 12 (Oct. 2007), pp. 471–516. DOI: 10.1007/s10664-007-9040-x.

[Som16] Ian Sommerville. *Software Engineering*. 10th Edition. Addison-Wesley, 2016. ISBN: 9780133943030.

[Com] Software Development Community. *What Is Service-Oriented Architecture?* URL: https://medium.com/@SoftwareDevelopmentCommunity/what-is-service-oriented-architecture-fa894d11a7ec. (accessed: 03.06.2021).

[Gur] Guru99. *Integration Testing: What is, Types, Top Down Bottom Up Example*. URL: https://www.guru99.com/integration-testing.html. (accessed: 04.06.2021).