1.4 I can appreciate (describe) the need for testing and project-based development

Types of manual testing

**Manual Testing**

**What is manual testing?**

[Manual testing](https://www.globalapptesting.com/manual-testing-best-practices) is defined as software testers manually executing test cases without the use of any automation tools. They play the role of the end-user and try to find as many bugs in the application as quickly as possible. The bugs are collated into a bug report, which is passed on to the developers to review and fix them. Manual testing often focuses on usability, performance testing, and assessing the overall software quality.

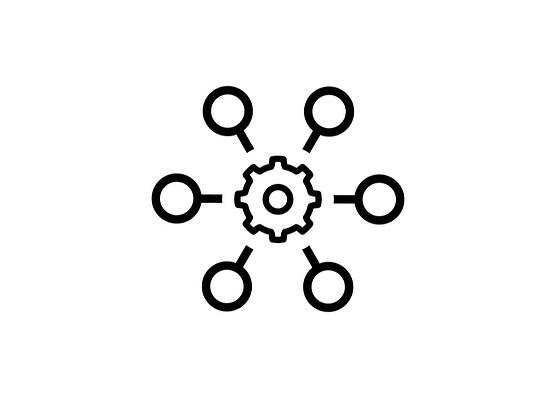
An application cannot be tested using automation exclusively so manual testing plays a vital role in software testing. It requires a certain mindset; patience, creativity & open-minded amongst them.

**Automated Testing**

**What is automated testing?**

[Automated testing](https://www.globalapptesting.com/best-practices-automated-testing) is a process in which an automation tool is used to execute pre-scripted test cases. The objective of test automation is to simplify and increase efficiency in the testing process.

If a particular form of testing consumes a large percentage of quality assurance, it could be a good candidate for automation. Acceptance testing, integration testing, and functional testing are all well suited to this type of software test. For instance, checking login processes is a good example of when to use automation testing.

  
Using automated testing is undoubtedly quicker than manual testing. If you’re looking to speed up the software development life cycle, it can be a worthwhile investment. In terms of testing execution, it will increase productivity and reduce testing time for the majority of apps/websites. Even though set up costs are high, automated testing can save money in the long-term.

Repetitive tasks are inefficient when done manually, especially when they reoccur. There is also an increased chance of human error. Automated testing can eradicate this, depending on the quality and scope of the test cases.

While automated testing is great for types of testing like stress testing and smoke testing, it’s not suited for everything. Looking at user interface, documentation, installation, compatibility, and recovery are often better suited to manual tests. Even if you choose to automate, some form of manual testing will be needed.

Initial set up costs (automation tool purchase, training and tutorials, maintenance of test scripts) are expensive. Also, if your app or website changes regularly, the cost and time associated with script maintenance will increase considerably.

**Black box testing**

**What is black box testing?**

Black box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation and without seeing the source code. The testers are only aware of what the software is supposed to do, not the logic of how it actually does this.



|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Unbiased tests | Test repetition |
| No programming language knowledge needed | Complex test cases |
| End-user point-of-view | Can be time-consuming |
| Faster test case creation | Cannot be used for complex code |

**Pros**

Black box testing offers unbiased tests because the designer and tester work independently. The tester doesn’t need to know any specific programming languages to test the reliability and functionality of an app/website.

Black box testing is performed from an end-user point-of-view rather than a developer standpoint. Test cases can be designed immediately after the completion of specifications.

**Cons**

Testing every possible input stream is not possible because it is too time-consuming and would eventually leave many program paths untested. Black box testing isn’t intended to test complex segments of code.

**White Box Testing**

**What is white box testing?**

White box testing is the opposite of black box testing. It tests the internal structure of an application to test the code itself, as opposed to the functionality exposed to the end-user. This type of testing is used by both developers and testers. It helps them to understand which line of code is actually executed and which isn’t.



|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Transparency of the internal coding structure | Complex and expensive |
| Maximum test coverage | Regular updates to test script |
| Easily automated | Time-consuming |

**Pros**

Transparency of the internal coding structure is helpful to understand the type of input data that is needed to test effectively. White box testing covers all possible paths of code which can motivate developers to write better code. Test cases can be easily automated with an abundance of tools available to do this.

**Cons**

White box testing is a complex and expensive procedure which requires a mix of extensive programming knowledge and a deep understanding of the internal code structure. The complexity is prolonged if it’s a large application. Updates to the test script are required when the implementation is changing too often.

The necessity to create a full range of inputs to test each path and condition make white box testing extremely time-consuming. It means some conditions might be untested as it is not realistic to test every one.

**The need for testing**

Why software testing is important

Few can argue against the need for quality control when developing software. Late delivery or software defects can damage a brand’s reputation — leading to frustrated and lost customers. In extreme cases, a bug or defect can degrade interconnected systems or cause serious malfunctions.

Consider Nissan having to recall over 1 million cars due to a software defect in the airbag sensor detectors. Or a software bug that caused the failure of a USD 1.2 billion military satellite launch. [2](https://www.ibm.com/topics/software-testing#citation2)The numbers speak for themselves. Software failures in the US cost the economy USD 1.1 trillion in assets in 2016. What’s more, they impacted 4.4 billion customers. [3](https://www.ibm.com/topics/software-testing#citation3)

Though testing itself costs money, companies can save millions per year in development and support if they have a good testing technique and QA processes in place. Early software testing uncovers problems before a product goes to market. The sooner development teams receive test feedback, the sooner they can address issues such as:

* Architectural flaws
* Poor design decisions
* Invalid or incorrect functionality
* Security vulnerabilities
* Scalability issues

When development leaves ample room for testing, it improves software reliability and high-quality applications are delivered with few errors. A system that meets or even exceeds customer expectations leads to potentially more sales and greater market share.

**Catch bugs early**

Why bug tracking is important

It is estimated that software developers make 100 to 150 errors for every thousand lines of code.[4](https://www.ibm.com/topics/bug-tracking#citation4)  According to a report by the Consortium for IT Software Quality (CISQ): “Even if only a small fraction — say 10 percent — of these errors are serious, then a relatively small application of 20,000 lines of code will have roughly 200 serious coding errors." [5](https://www.ibm.com/topics/bug-tracking#citation5)

Software testing is essential for isolating and mitigating errors. A good QA process can uncover hundreds or even thousands of defects, and testing teams need to manage all of them. Integrating bug tracking into the testing workflow improves efficiency by helping testers prioritize, monitor and report on the status of each error.

“Defect tracking helps ensure that bugs found in the system actually get fixed,” says Agile consultant, Yvette Francino. “Tracking tools not only provide a way to ensure follow-through but also provide valuable metrics. Depending on the tool being used, the team can tie defects to changed code, tests or other data that will allow for traceability or analysis on defect trends. If a certain module is riddled with defects, it may be time to review and rewrite that module." [6](https://www.ibm.com/topics/bug-tracking#citation6)

Ideally, testing should be done as soon as possible — when bugs are easier and far less costly to fix. An earlier study by IBM found that defects found post-production or after release can cost 15 times more to fix compared to errors resolved early in development.

Many teams are now using a methodology known as [continuous testing](https://www.ibm.com/cloud/learn/continuous-testing). In this case, quality testing and feedback are conducted at all stages of development, from design and coding to deployment. Modern technologies like artificial intelligence (AI) too can aid the testing process by detecting and analyzing bugs early in the lifecycle.

**Save costs of late bugs**

Key features of bug tracking

Quality control is critical for developing robust applications. Software testing, change management and bug tracking tools allow teams to uncover defects, measure their scope and impact — and resolve them.

In the Harvard Business Review, Nicholas Bowen outlines a process for managing defects. The first step is to classify and prioritize: “Generally, teams will prioritize two types of bugs: those that cause a system to crash and those that are less severe but could be pervasive....Next, decide your target response time for each level of severity. If the quality management system is new, then the initial focus should be on fixing the most severe bugs within hours or days. As you use your system, you can gather data on two key metrics, incoming bug rates and the productivity of the bug fixers, and adjust your targets as needed.” He says organizations also need to create a system where defects and the time required to resolve them can be reviewed at all levels, from the CEO on down.[7](https://www.ibm.com/topics/bug-tracking#citation7)

A good bug tracking system can assist this process by providing a single workflow for defect monitoring, reporting and lifecycle traceability. It should further link with other management systems for shared visibility and feedback both within software development and the larger organization. [IBM Rational ClearQuest](https://www.ibm.com/products/rational-clearquest), for example, provides a centralized platform for error tracking and reporting. It integrates with other IBM development and change management systems, and helps improve communication and collaboration among developers, operations and broader teams.

As well, look for testing and tracking systems that use AI to detect errors early in the development process. It can optimize the number and types of tests that teams run, automate the testing process and use AI to analyze past defects and prevent them in the future.

**Having working software**

Software testing best practices

Software testing follows a common process. Tasks or steps include defining the test environment, developing test cases, writing scripts, analyzing test results and submitting defect reports.

Testing can be time-consuming. Manual testing or ad-hoc testing may be enough for small builds. However, for larger systems, tools are frequently used to automate tasks. Automated testing helps teams implement different scenarios, test differentiators (such as moving components into a cloud environment), and quickly get feedback on what works and what doesn't.

A good testing approach encompasses the application programming interface (API), user interface and system levels. As well, the more tests that are automated, and run early, the better. Some teams build in-house test automation tools. However, vendor solutions offer features that can streamline key test management tasks such as:

* **Continuous testing:** Project teams test each build as it becomes available. This type of software testing relies on test automation that is integrated with the deployment process. It enables software to be validated in realistic test environments earlier in the process – improving design and reducing risks.
* **Configuration management:** Organizations centrally maintain test assets and track what software builds to test. Teams gain access to assets such as code, requirements, design documents, models, test scripts and test results. Good systems include user authentication and audit trails to help teams meet compliance requirements with minimal administrative effort.
* **Service virtualization:** Testing environments may not be available, especially early in code development. Service virtualization simulates the services and systems that are missing or not yet completed, enabling teams to reduce dependencies and test sooner. They can reuse, deploy and change a configuration to test different scenarios without having to modify the original environment.
* **Defect or**[**bug tracking**](https://www.ibm.com/topics/bug-tracking)**:** Monitoring defects is important to both testing and development teams for measuring and improving quality. Automated tools allow teams to track defects, measure their scope and impact, and uncover related issues.
* **Metrics and reporting:** Reporting and analytics enable team members to share status, goals and test results. Advanced tools integrate project metrics and present results in a dashboard. Teams quickly see the overall health of a project and can monitor relationships between test, development and other project elements.

**Meet requirements**

Types of software testing

There are many different types of software tests, each with specific objectives and strategies:

* **Acceptance testing:**Verifying whether the whole system works as intended.
* **Integration testing:** Ensuring that software components or functions operate together.
* **Unit testing:** Validating that each software unit performs as expected. A unit is the smallest testable component of an application.
* **Functional testing:** Checking functions by emulating business scenarios, based on functional requirements. Black-box testing is a common way to verify functions.
* **Performance testing:** Testing how the software performs under different workloads. Load testing, for example, is used to evaluate performance under real-life load conditions.
* **Regression testing:**Checking whether new features break or degrade functionality. Sanity testing can be used to verify menus, functions and commands at the surface level, when there is no time for a full regression test.
* **Stress testing:** Testing how much strain the system can take before it fails. Considered to be a type of non-functional testing.
* **Usability testing:** Validating how well a customer can use a system or web application to complete a task.

In each case, validating base requirements is a critical assessment. Just as important, exploratory testing helps a tester or testing team uncover hard-to-predict scenarios and situations that can lead to software errors.

Even a simple application can be subject to a large number and variety of tests. A test management plan helps to prioritize which types of testing provide the most value – given available time and resources. Testing effectiveness is optimized by running the fewest number of tests to find the largest number of defects.

**Information from IBM**

**Project based development**

**What are Microservices?**

Microservice architecture, or simply microservices, is a distinctive method of developing software systems that tries to focus on building single-function modules with well-defined interfaces and operations. The trend has grown popular in recent years as Enterprises look to become more Agile and move towards a DevOps and continuous testing.

Microservices have many benefits for Agile and [DevOps](https://smartbear.com/solutions/devops/) teams - as Martin Fowler [points out](http://martinfowler.com/articles/microservices.html), Netflix, eBay, Amazon, Twitter, PayPal, and other tech stars have all evolved from monolithic to microservices architecture. Unlike microservices, a monolith application is built as a single, autonomous unit. This make changes to the application slow as it affects the entire system. A modification made to a small section of code might require building and deploying an entirely new version of software. Scaling specific functions of an application, also means you have to scale the entire application.

Microservices solve these challenges of monolithic systems by being as modular as possible. In the simplest form, they help build an application as a suite of small services, each running in its own process and are independently deployable. These services may be written in different programming languages and may use different data storage techniques. While this results in the development of systems that are scalable and flexible, it needs a dynamic makeover. Microservices are often connected via APIs, and can leverage many of the same tools and solutions that have grown in the RESTful and web service ecosystem. [Testing these APIs](https://smartbear.com/solutions/api-testing/) can help validate the flow of data and information throughout your microservice deployment.

**Benefits Of Microservices**

|  |  |
| --- | --- |
| Simpler To Deploy | Deploy in literal pieces without affecting other services. |
| Simpler To Understand | Follow code easier since the function is isolated and less dependent. |
| Reusability Across Business | Share small services like payment or login systems across the business. |
| Faster Defect Isolation | When a test fails or service goes down, isolate it quickly with microservices. |
| Minimized Risk Of Change | Avoid locking in technologies or languages - change on the fly without risk. |

**Understanding Microservice Architecture**

Just as there is no formal definition of the term microservices, there’s no standard model that you’ll see represented in every system based on this architectural style. But you can expect most microservice systems to share a few notable characteristics.

[See How SwaggerHub Helps Teams Get Started With Microservices](https://swagger.io/tools/swaggerhub/)

**The Six Characteristics Of Microservices**

1. **Multiple Components**

Software built as microservices can, by definition, be broken down into multiple component services. Why? So that each of these services can be deployed, tweaked, and then redeployed independently without compromising the integrity of an application. As a result, you might only need to change one or more distinct services instead of having to redeploy entire applications. But this approach does have its downsides, including expensive remote calls (instead of in-process calls), coarser-grained remote APIs, and increased complexity when redistributing responsibilities between components.

1. **Built For Business**

The microservices style is usually organized around business capabilities and priorities. Unlike a traditional monolithic development approach—where different teams each have a specific focus on, say, UIs, databases, technology layers, or server-side logic—microservice architecture utilizes cross-functional teams. The responsibilities of each team are to make specific products based on one or more individual services communicating via message bus. In microservices, a team owns the product for its lifetime, as in Amazon’s oft-quoted maxim “[You build it, you run it.](http://www.strehle.de/tim/weblog/archives/2010/11/09/1320)

1. **Simple Routing**

Microservices act somewhat like the classical UNIX system: they receive requests, process them, and generate a response accordingly. This is opposite to how many other products such as ESBs (Enterprise Service Buses) work, where high-tech systems for message routing, choreography, and applying business rules are utilized. You could say that microservices have smart endpoints that process info and apply logic, and dumb pipes through which the info flows.

1. **Decentralized**

Since microservices involve a variety of technologies and platforms, old-school methods of centralized governance aren’t optimal. Decentralized governance is favored by the microservices community because its developers strive to produce useful tools that can then be used by others to solve the same problems. Just like decentralized governance, microservice architecture also favors decentralized data management. Monolithic systems use a single logical database across different applications. In a microservice application, each service usually manages its unique database.

1. **Failure Resistant**

Like a well-rounded child, microservices are designed to cope with failure. Since several unique and diverse services are communicating together, it’s quite possible that a service could fail, for one reason or another (e.g., when the supplier isn’t available). In these instances, the client should allow its neighboring services to function while it bows out in as graceful a manner as possible. However, [monitoring microservices](https://smartbear.com/en-us/learn/performance-monitoring/monitoring-microservices/) can help prevent the risk of a failure. For obvious reasons, this requirement adds more complexity to microservices as compared to monolithic systems architecture.

1. **Evolutionary**

Microservices architecture is an evolutionary design and, again, is ideal for evolutionary systems where you can’t fully anticipate the types of devices that may one day be accessing your application.. Many applications start based on monolithic architecture, but as several unforeseen requirements surfaced, can be slowly revamped to microservices that interact over an older monolithic architecture through APIs.

**Examples of Microservices**

Netflix has a widespread architecture that has evolved from monolithic to SOA. It receives more than one billion calls every day, from more than 800 different types of devices, to its streaming-video API. Each API call then prompts around five additional calls to the backend service.

Amazon has also migrated to microservices. They get countless calls from a variety of applications—including applications that manage the web service API as well as the website itself—which would have been simply impossible for their old, two-tiered architecture to handle.

The auction site eBay is yet another example that has gone through the same transition. Their core application comprises several autonomous applications, with each one executing the business logic for different function areas.

**Microservice Pros and Cons**

Microservices are not a silver bullet, and by implementing them you will expose communication, teamwork, and other problems that may have been previously implicit but are now forced out into the open. But [API Gateways in Microservices](https://smartbear.com/learn/api-design/api-gateways-in-microservices/) can greatly reduce build and QA time and effort.

One common issue involves sharing schema/validation logic across services. What A requires in order to consider some data valid doesn’t always apply to B, if B has different needs. The best recommendation is to apply versioning and distribute schema in shared libraries. Changes to libraries then become discussions between teams. Also, with strong versioning comes dependencies, which can cause more overhead. The best practice to overcome this is planning around backwards compatibility, and accepting [regression tests](https://smartbear.com/learn/automated-testing/what-is-regression-testing/) from external services/teams. These prompt you to have a conversation *before* you disrupt someone else’s business process, not after.

As with anything else, whether or not microservice architecture is right for you depends on your requirements, because they all have their pros and cons. Here’s a quick rundown of some of the good and bad:

**Pros**

* Microservice architecture gives developers the freedom to independently develop and deploy services
* A microservice can be developed by a fairly small team
* Code for different services can be written in different languages (though many practitioners discourage it)
* Easy integration and automatic deployment (using open-source continuous integration tools such as Jenkins, Hudson, etc.)
* Easy to understand and modify for developers, thus can help a new team member become productive quickly
* The developers can make use of the latest technologies
* The code is organized around business capabilities
* Starts the web container more quickly, so the deployment is also faster
* When change is required in a certain part of the application, only the related service can be modified and redeployed—no need to modify and redeploy the entire application
* Better fault isolation: if one microservice fails, the other will continue to work (although one problematic area of a monolith application can jeopardize the entire system)
* Easy to scale and integrate with third-party services
* No long-term commitment to technology stack

**Cons**

* Due to distributed deployment, testing can become complicated and tedious
* Increasing number of services can result in information barriers
* The architecture brings additional complexity as the developers have to mitigate fault tolerance, network latency, and deal with a variety of message formats as well as load balancing
* Being a distributed system, it can result in duplication of effort
* When number of services increases, integration and managing whole products can become complicated
* In addition to several complexities of monolithic architecture, the developers have to deal with the additional complexity of a distributed system
* Developers have to put additional effort into implementing the mechanism of communication between the services
* Handling use cases that span more than one service without using distributed transactions is not only tough but also requires communication and cooperation between different teams

**How Microservice Architecture Works**

**1) Monoliths and**[**Conway’s Law**](http://en.wikipedia.org/wiki/Conway%27s_law)

To begin with, let’s explore Conway’s Law, which states: *“Organizations which design systems…are constrained to produce designs which are copies of the communication structures of these organizations.”*

Imagine Company X with two teams: *Support and Accounting*. Instinctively, we separate out the high risk activities; it’s only difficult deciding responsibilities like customer refunds. Consider how we might answer questions like “Does the Accounting team have enough people to process both customer refunds and credits?” or “Wouldn’t it be a better outcome to have our Support people be able to apply credits and deal with frustrated customers?” The answers get resolved by Company X’s new policy: Support can apply a *credit*, but Accounting has to *process a refund* to return money to a customer. The roles and responsibilities in this interconnected system have been successfully split, while gaining customer satisfaction and minimizing risks.

Likewise, at the beginning of designing any software application, companies typically assemble a *team* and create a *project*. Over time, the team grows, and multiple projects on the same codebase are completed. More often than not, this leads to competing projects: two people will find it difficult to work at cross purposes in the same area of code without introducing tradeoffs. And adding more people to the equation only makes the problem worse. As [Fred Brooks](http://en.wikipedia.org/wiki/Brooks%27s_law) puts it, nine women can’t make a baby in one month.

Moreover, in Company X or in any dev team, priorities frequently shift, resulting in management and communication issues. Last month’s highest priority item may have caused our team to push hard to ship code, but now a user is reporting an issue, and we no longer have time to resolve it because of *this* month’s priority. This is the most compelling reason to adopt SOA, including the microservices variety. Service-oriented approaches recognize the frictions involved between change management, domain knowledge, and business priorities, allowing dev teams to explicitly separate and address them. Of course, this in itself is a tradeoff—it requires coordination—but it allows you to centralize friction and introduce efficiency, as opposed to suffering from a large number of small inefficiencies.

Most importantly, smartly implementing an SOA or microservice architecture forces you to apply the [Interface Separation Principle](http://en.wikipedia.org/wiki/Interface_segregation_principle). Due to the connected nature of mature systems, when isolating issues of concern, the typical approach is to find a seam or communication point and then draw a dotted line between two halves of the system. Without careful thought, however, this can lead to accidentally creating two smaller but growing monoliths, now connected with some kind of bridge. The consequence of this can be marooning important code on the wrong side of a barrier: Team A doesn’t bother to look after it, while Team B needs it, so they reinvent it.

**2) Microservices: Avoiding the Monoliths**

We’ve named some problems that commonly emerge; now let’s begin to look at some solutions.

How do you deploy relatively independent yet integrated services without spawning accidental monoliths? Well, suppose you have a large application, as in the sample from our Company X below, and are splitting up the codebase and teams to scale. Instead of finding an entire section of an application to split off, you can look for something on the *edge* of the application graph. You can tell which sections these are because nothing depends on them. In our example, the arrows pointing to Printer and Storage suggest they’re two things that can be easily removed from our main application and abstracted away. Printing either a Job or Invoice is irrelevant; a Printer just wants printable data. Turning these—Printer and Storage—into external services avoids the monoliths problem alluded to before. It also makes sense as they are used multiple times, and there’s little that can be reinvented. Use cases are well known from past experience, so you can avoid accidentally removing key functionality.

Diagram

Description automatically generated

**3) Service Objects and Identifying Data**

So how do we go from monoliths to services? One way is through *service objects*. Without removing code from your application, you effectively just begin to structure it as though it were completely external. To do that, you’ll first need to differentiate the *actions* that can be done and the *data* that is present as inputs and outputs of those actions. Consider the code below, with a notion of *doing something useful* and a *status of that task*.

# A class to model a core transaction and execute it

class Job

def initialize

@status = 'Queued'

end

def do\_useful\_work

....

@status = 'Finished'

end

def finished?

return @status == 'Finished'

end

def ready?

return @status == 'Queued'

end

end

Ruby

To prepare this to begin looking like a microservice, what’s next?

# Service to do useful work and modify a status

class JobService

def do\_useful\_work(job\_status)

....

job\_status.finish!

return job\_status

end

end

# A model of our Job's status

class JobStatus

def initialize

@status = 'Queued'

end

def finished?

return @status == 'Finished'

end

def ready?

return @status == 'Queued'

end

def finish!

@status = 'Finished'

end

end

Ruby

Now we’ve distinguished two distinct classes: one that models the data, and one that performs the operations. Importantly, our JobService class has little or no state—you can call the same actions over and over, changing only the data, and expect to get consistent results. If JobService somehow started taking place over a network, our otherwise monolithic application wouldn’t care. Shifting these types of classes into a library, and substituting a network client for the previous implementation, would allow you to transform the existing code into a scalable external service.

This is [Hexagonal Architecture](http://alistair.cockburn.us/Hexagonal+architecture), where the core of your application and the coordination is in the center, and the external components are orchestrated around it to achieve your goals.

Diagram

Description automatically generated

(You can read more about service objects and hexagonal architecture [here](http://blog.mattwynne.net/2012/05/31/hexagonal-rails-objects-values-and-hexagons/) and [here](https://blog.engineyard.com/2014/keeping-your-rails-controllers-dry-with-services).)

**4) Coordination and Dumb Pipes**

Now let’s take a closer look at what makes something a microservice as opposed to a traditional SOA.

Perhaps the most important distinction is *side effects*. Microservices avoid them. To see why, let’s look at an older approach: Unix pipes.

ls | wc -l

Above, two programs are chained together: the first lists all of the files in a directory, the second reads the number of lines in a stream of input. Imagine writing a comparable program, then having to modify it into the below:

ls | less

Composing small pieces of functionality relies on repeatable results, a standard mechanism for input and output, and an exit code for a program to indicate success or lack thereof. We know this works from observational evidence, and we also know that a Unix pipe is a “dumb” interface because it has no control statements. The pipe applies SRP by pushing data from A to B, and it’s up to members of the pipeline to decide if the input is unacceptable.

Let’s go back to Company X’s Job and Invoice systems. Each controls a transaction and can be used together or separately: Invoices can be created for jobs, jobs can be created without an invoice, and invoices can be created without a job. Unlike Unix shell commands, the systems that own jobs and invoices have their own users working independently. But without falling back to a *policy*, it’s impossible to enforce rules for either system globally.

Say we want to extract out the key operations that can be repeatedly executed—the services for sending an invoice, mutating a job status and mutating an invoice status. These are completely separate from the task of *persisting* data.

Diagram

Description automatically generated

Here this allows us to wire together the discrete components into two pipelines:

**User creates a manual invoice**

* Adds data to invoice, status *created*

— Invokes BillingPolicyService to determine when an invoice is payable for a given customer

* Invoice is issued to customer
* Persists to the invoice data service, status *sent*

**User finishes a job, creating an invoice**

* Validates job is completable
* Adds data to invoice, status *created*

— Invokes BillingPolicyService to determine when an invoice is payable for a given customer

* Invoice is issued to customer
* Persists to the invoice data service, status *sent*

The invoice calculation related steps are idempotent, and it’s then trivial to compose a *draft invoice* or preview the amounts payable by the customer by leveraging our new dedicated microservices.

Unlike traditional SOA, the difference here is that we have low-level details exposed via a simple interface, as compared to a high-level API call that might execute an entire business action. With a high-level API, in fact, it becomes difficult to rewire small components together, since the service designer has removed many of the seams or choices we can take by providing a one-shot interface.

By this point, the repetition of business logic, policy and rules leads many to traditionally push this complexity into a service bus or singular, centralized workflow orchestration tool. However, the crucial advantage of microservice architecture is not that we *never* share business rules/processes/policies, but that we push them into discrete packages, aligned to business needs. Not only does this mean that policy is distributed, but it also means that *you can change your business processes without risk*.

**SOA vs. Microservices**

“Wait a minute,” some of you may be murmuring over your morning coffee, “isn’t this just another name for SOA?” Service-Oriented Architecture (SOA) sprung up during the first few years of this century, and microservice architecture (abbreviated by some as MSA) bears a number of similarities. Traditional SOA, however, is a broader framework and can mean [a wide variety of things](http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html). Some microservices advocates reject the SOA tag altogether, while others consider microservices to be simply an ideal, refined form of SOA. In any event, we think there are clear enough differences to justify a distinct “microservice” concept (at least as a special form of SOA, as we’ll illustrate later).

The typical SOA model, for example, usually has more dependent ESBs, with microservices using faster messaging mechanisms. SOA also focuses on imperative programming, whereas microservices architecture focuses on a responsive-actor programming style. Moreover, SOA models tend to have an outsized relational database, while microservices frequently use NoSQL or micro-SQL databases (which can be connected to conventional databases). But the real difference has to do with the architecture methods used to arrive at an integrated set of services in the first place.

Since everything changes in the digital world, [agile development techniques](https://smartbear.com/learn/software-testing/what-is-agile-testing/) that can keep up with the demands of software evolution are invaluable. Most of the practices used in microservices architecture come from developers who have created software applications for large enterprise organizations, and who know that today’s end users expect dynamic yet consistent experiences across a wide range of devices. Scalable, adaptable, modular, and quickly accessible cloud-based applications are in high demand. And this has led many developers to change their approach.

**The Future of Microservice Architecture**

Whether or not microservice architecture becomes the preferred style of developers in future, it’s clearly a potent idea that offers serious benefits for designing and implementing enterprise applications. Many developers and organizations, without ever using the name or even labeling their practice as SOA, have been using an approach toward leveraging APIs that could be classified as microservices.

We’ve also seen a number of existing technologies try to address parts of the segmentation and communication problems that microservices aim to resolve. SOAP does well at describing the operations available on a given endpoint and where to discover it via WSDLs. UDDI is theoretically a good step toward advertising what a service can do and where it can be found. But these technologies have been compromised by a relatively complex implementation, and tend not to be adopted in newer projects. REST-based services face the same issues, and although you can use WSDLs with REST, it is not widely done.

Assuming discovery is a solved problem, sharing schema and *meaning* across unrelated applications still remains a difficult proposition for anything other than microservices and other SOA systems. Technologies such as [RDFS](http://www.w3.org/TR/rdf-schema/), [OWL](http://www.w3.org/TR/owl2-overview/), and [RIF](http://www.w3.org/standards/techs/rif#w3c_all) exist and are standardized, but are not commonly used. [JSON-LD](http://json-ld.org/) and [Schema.org](http://schema.org/) offer a glimpse of what an entire open web that shares definitions looks like, but these aren’t yet adopted in large private enterprises.

The power of shared, standardized definitions are making inroads within government, though. The results are visible through in data.gov and data.gov.uk, and you can explore the large number of data sets available as well-described linked data [here](http://lod-cloud.net/). If a large number of standardized definitions can be agreed upon, the next steps are most likely toward *agents:* small programs that orchestrate microservices from a large number of vendors to achieve certain goals. When you add the increasing complexity and communication requirements of SaaS apps, wearables, and the Internet of Things into the overall picture, it’s clear that microservice architecture probably has a very bright future ahead.

**Information from Smartbear**

**Agile Scrum**

**Agile scrum methodology is a sprint-based project management system with the goal of delivering the highest value to stakeholders.**

* Agile and scrum are two similar project management systems with a few key differences.
* Agile is more flexible and promotes leadership teams, while scrum is more rigid and promotes cross-functional teams.
* Agile lets teams develop projects in small increments called “sprints” and allows for more effective collaborations among teams working on complex projects.
* **This article is for business owners and project managers who want to learn more about agile scrum methodology and how to implement it as a management process.**

Agile scrum methodology is used by companies of all sizes for its ability to provide high-end collaboration and efficiency for project-based work. Agile and scrum are two different methods and can be used separately; however, their combined benefits make the agile scrum methodology the most popular use of agile. Here’s the complete guide to agile scrum methodology.

**Did you know?**Agile and scrum can be used separately, but their combined benefits make the methodology popular.

How does agile scrum work?

Agile scrum methodology is the combination of the agile philosophy and the scrum framework. Agile means “incremental, allowing teams to develop projects in small increments. Scrum is one of the many types of agile methodology, known for breaking projects down into sizable chunks called “sprints.” Agile scrum methodology is good for businesses that need to finish specific projects quickly.

Agile scrum methodology is a [project management system](https://www.businessnewsdaily.com/9977-best-online-project-management-software.html) that relies on incremental development. Each iteration consists of two- to four-week sprints, where the goal of each sprint is to build the most important features first and come out with a potentially deliverable product. More features are built into the product in subsequent sprints and are adjusted based on stakeholder and customer feedback between sprints.

Whereas other project management methods emphasize building an entire product in one operation from start to finish, agile scrum methodology focuses on delivering several iterations of a product to provide stakeholders with the highest business value in the least amount of time.

Agile scrum methodology has several benefits. First, it encourages products to be built faster, since each [set of goals](https://www.businessnewsdaily.com/11225-set-achievable-business-goals.html) must be completed within each sprint’s time frame. It also requires frequent planning and goal setting, which helps the scrum team focus on the current sprint’s objectives and increase productivity.

What is agile?

Agile is a process that allows a team to more efficiently manage a project by breaking it down into several stages, each of which allows for consistent collaboration with stakeholders to promote steady improvements at every stage.

**Key takeaway:**Agile lets a team manage a project more efficiently by breaking it down into several stages.

What are the values of agile?

Agile was first described in the Agile Manifesto in 2000 by a group of developers who sought out a new method of writing software. The manifesto cites four values:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

What are the 12 principles of agile?

The Agile Manifesto also enacted 12 principles in reference to software development and was later reconfigured to fit a wider perspective of users:

1. Customer satisfaction
2. Early and continuous delivery
3. Embrace change
4. Frequent delivery
5. Collaboration of businesses and developers
6. Motivated individuals
7. Face-to-face conversation
8. Functional products
9. Technical excellence
10. Simplicity
11. Self-organized teams
12. Regulation, reflection and adjustment

What is scrum?

In short, scrum is a framework for effective collaborations among teams working on complex products. Scrum is a type of agile technology that consists of meetings, roles, and tools to help teams working on complex projects collaborate and better structure and manage their workload. Although it is most often used by software development teams, scrum can be beneficial to any team working toward a common goal.

Who can benefit from scrum?

While scrum can be useful for a wide variety of businesses and projects, these are the most likely beneficiaries:

* **Complicated projects:**Scrum methodology is ideal for projects that require teams to complete a backlog. Scrum breaks down each process into bite-sized chunks that can make a complex project easier.
* **Companies that value results:** Scrum is also beneficial to companies that value results over the documented progress of the process. This is because scrum is focused on efficiency and innovation to drive results, rather than a detailed, rigid process.
* **Companies that cater to customers:**Scrum can help companies that develop products in accordance with customer preferences and specifications. Scrum is adaptable to change, making it key when responding to customer requests.

What are the benefits of agile scrum methodology?

These are some of the collective benefits of agile scrum methodology:

* [Flexibility and adaptability](https://www.businessnewsdaily.com/5696-pivot-business-strategy-shift.html)
* Creativity and innovation
* Lower costs
* Quality improvement
* Organizational synergy
* Employee satisfaction
* Customer satisfaction

The greatest benefit of agile scrum methodology is its flexibility. With the sprint-based model, the scrum team typically receives feedback from stakeholders after each sprint. If there are any problems or changes, the scrum team can easily and quickly adjust product goals during future sprints to provide more valuable iterations. This way, stakeholders are happier because they get exactly what they want after being involved every step of the way.

Compare this with traditional project management systems, in which stakeholders do not provide frequent feedback and time is wasted making changes to the product halfway through development – or worse, such as the teams needing to start from scratch after the product has already been built.

To implement agile scrum methodology, there must be either a scrum expert in the company or an outside consultant to ensure scrum principles are being applied correctly. Agile scrum methodology involves precise execution and could result in serious problems if not done properly.

**Tip:** To implement agile scrum, you’ll need an expert in your company or an [outside consultant](https://www.businessnewsdaily.com/4610-business-consultant.html).

What are the different roles in agile scrum methodology?

Agile scrum methodology consists of two sets of roles: core roles, known as “pigs,” and ancillary roles, known as “chickens.”

There are three core roles: scrum master, product owner and scrum team. All of these people are committed to the scrum project.

1. **Scrum master:** The scrum master is the facilitator of the scrum development process. In addition to holding daily meetings with the scrum team, the scrum master makes certain that scrum rules are being enforced and applied as intended. The scrum master’s responsibilities also include coaching and [motivating the team](https://www.businessnewsdaily.com/4782-how-to-motivate-workers.html), removing impediments to sprints, and ensuring that the team has the best possible conditions to meet its goals and produce deliverable products.
2. **Product owner:** The product owner represents stakeholders, who are typically customers. To ensure the scrum team is always delivering value to stakeholders and the business, the product owner determines product expectations, records changes to the product and administers a scrum backlog, a detailed and constantly updated to-do list for the scrum project. The product owner is also responsible for prioritizing goals for each sprint, based on their value to stakeholders, such that the most important and deliverable features are built in each iteration.
3. **Scrum team:**The scrum team is a self-organized group of three to nine individuals who have the business, design, analytical and development skills to carry out the actual work, solve problems and produce deliverable products. Members of the scrum team self-administer tasks and are jointly responsible for meeting each sprint’s goals.

Ancillary roles, on the other hand, are other stakeholders who are involved in, but not committed to, the scrum project. Typically, ancillary roles consist of customers, management and members of the executive team who are involved for the purpose of consulting, reporting progress and gathering feedback to better work toward delivering the highest value possible.

What is the training for scrum and agile?

Managers and employees can [enroll in training](https://www.businessnewsdaily.com/15839-new-hire-training.html) for both agile and scrum through various online and in-person courses. Many educational training courses result in certification in agile or scrum methodologies. Agile training provides the trainee with the basic knowledge of agile and how to implement it to the rest of their team. Scrum provides similar training, including the basic agile overview; however, the training caters to the scrum framework.

To become a certified scrum master (CSM) or certified scrum product owner (CSPO), you must first prepare and learn the basic details of scrum through videos or a simple internet search. Next, find a suitable CSM or CSPO course, either through your workplace or another internet search. Once you’ve completed the course, you usually have to pass an exam to become certified. After certification, you’re able to lead your team through the scrum process or provide scrum product details.

What are the differences between scrum and agile?

Although scrum and agile are similar, they have some key differences:

* Scrum values rigidity, whereas agile is more flexible.
* Agile leaders play a vital role, while scrum promotes a cross-functional team that is self-functioning.
* Agile involves face-to-face interactions between cross-functional team members, while scrum involves daily stand-up meetings.
* Agile is meant to be kept simple, while scrum can be innovative and experimental.
* Scrum delivers shorter, separate projects, while agile delivers everything at the end of the process.

***Sara Angeles contributed to the writing and research in this article.***

**V-modeling testing**

**Software Engineering | SDLC V-Model**

* Difficulty Level : [Basic](https://www.geeksforgeeks.org/basic/)
* Last Updated : 03 Mar, 2022

The V-model is a type of SDLC model where process executes in a sequential manner in V-shape. It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage. Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it. 



**Verification:** It involves static analysis technique (review) done without executing code. It is the process of evaluation of the product development phase to find whether specified requirements meet.

**Validation:** It involves dynamic analysis technique (functional, non-functional), testing done by executing code. Validation is the process to evaluate the software after the completion of the development phase to determine whether software meets the customer expectations and requirements.

So V-Model contains Verification phases on one side of the Validation phases on the other side. Verification and Validation phases are joined by coding phase in V-shape. Thus it is called V-Model.

**Design Phase:** 

* **Requirement Analysis:** This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.
* **System Design:** This phase contains the system design and the complete hardware and communication setup for developing product.
* **Architectural Design:** System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.
* **Module Design:** In this phase the system breaks down into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD).

**Testing Phases:** 

* **Unit Testing:** Unit Test Plans are developed during module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.
* **Integration testing:** After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.
* **System Testing:** System testing test the complete application with its functionality, inter dependency, and communication.It tests the functional and non-functional requirements of the developed application.
* **User Acceptance Testing (UAT):** UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user’s requirement and system is ready for use in real world.

**Industrial Challenge:** As the industry has evolved, the technologies have become more complex, increasingly faster, and forever changing, however, there remains a set of basic principles and concepts that are as applicable today as when IT was in its infancy. 

* Accurately define and refine user requirements.
* Design and build an application according to the authorized user requirements.
* Validate that the application they had built adhered to the authorized business requirements.

**Principles of V-Model:** 

* **Large to Small:** In V-Model, testing is done in a hierarchical perspective, For example, requirements identified by the project team, create High-Level Design, and Detailed Design phases of the project. As each of these phases is completed the requirements, they are defining become more and more refined and detailed.
* **Data/Process Integrity:** This principle states that the successful design of any project requires the incorporation and cohesion of both data and processes. Process elements must be identified at each and every requirements.
* **Scalability:** This principle states that the V-Model concept has the flexibility to accommodate any IT project irrespective of its size, complexity or duration.
* **Cross Referencing:** Direct correlation between requirements and corresponding testing activity is known as cross-referencing.
* **Tangible Documentation:** This principle states that every project needs to create a document. This documentation is required and applied by both the project development team and the support team. Documentation is used to maintaining the application once it is available in a production environment.

**Why preferred?** 

* It is easy to manage due to the rigidity of the model. Each phase of V-Model has specific deliverables and a review process.
* Proactive defect tracking – that is defects are found at early stage.

**When to use?** 

* Where requirements are clearly defined and fixed.
* The V-Model is used when ample technical resources are available with technical expertise.

**Advantages:** 

* This is a highly disciplined model and Phases are completed one at a time.
* V-Model is used for small projects where project requirements are clear.
* Simple and easy to understand and use.
* This model focuses on verification and validation activities early in the life cycle thereby enhancing the probability of building an error-free and good quality product.
* It enables project management to track progress accurately.

**Disadvantages:** 

* High risk and uncertainty.
* It is not a good for complex and object-oriented projects.
* It is not suitable for projects where requirements are not clear and contains high risk of changing.
* This model does not support iteration of phases.
* It does not easily handle concurrent events.