MONASH University

FIT5032 - Internet Applications Development

Testing, Deployment & Evolution

Prepared by - Jian Liew

Last Updated - 1st October 2018

Monash University (Caulfield Campus)

Outline

1. What is Testing?
2. Errors, Defects and Failures.
3. Functional Testing vs Non-Functional Testing.
4. The Seven Testing Principles
5. Human Psychology and Testing.
6. Test Techniques
7. TDD in ASP.NET MVC
8. Software Development & Deployment
9. Types of Deployment Environments
10. Modern Development Practices
11. Summary

What is Testing?

- Software systems are an integral part of life, from business applications to consumer products.
- Most people have had an experience with software that **did not work as expected.**
- Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death.
- Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.
- It is common for software testers have an ISTQB certification. ISTQB® is a non-profit organization responsible for defining various guidelines such as examination structure and regulations, accreditation, certification etc. Materials for this lecture are derived from ISTQB foundation level materials.

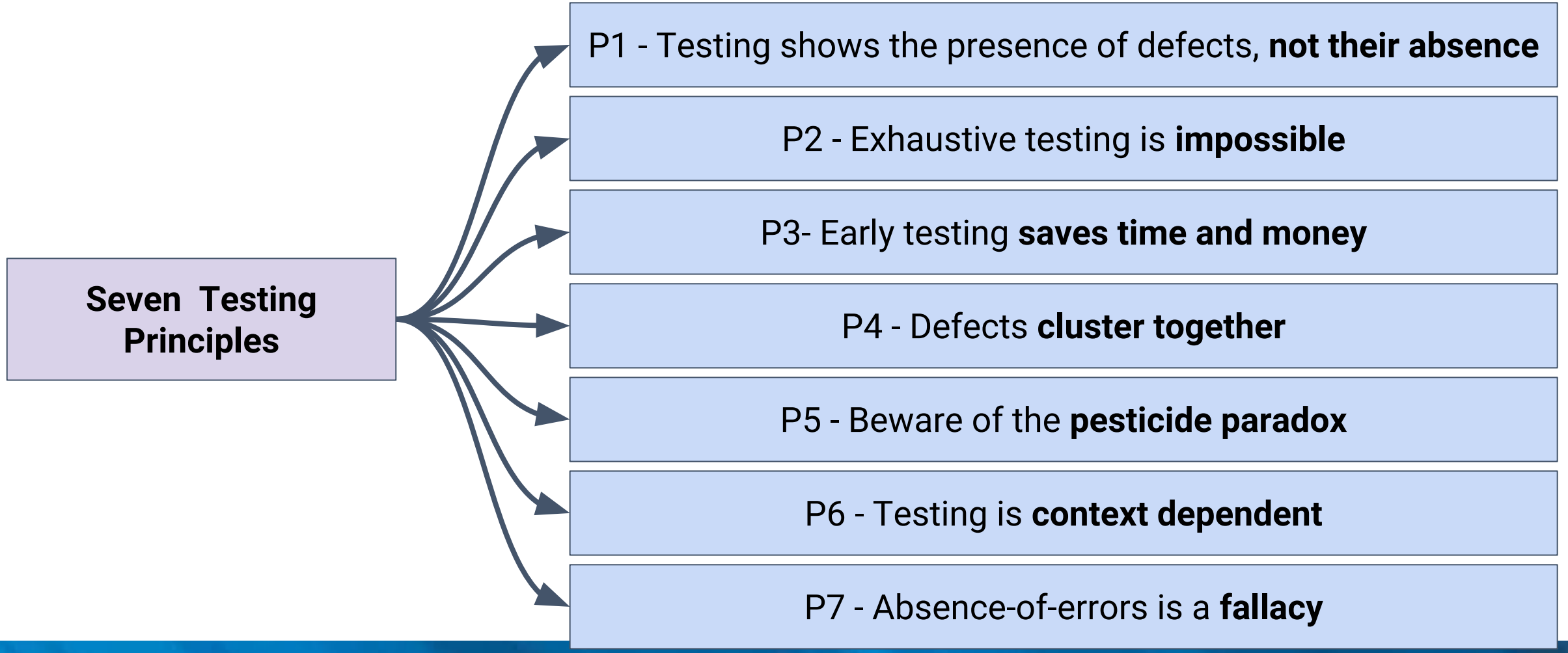
Errors, Defects & Failures

- A person can make an error (mistake), which can lead to the **introduction of a defect** (fault or bug) in the software code or in some other related work product.
- An error that leads to the introduction of a defect in one work product can trigger an error that leads to the introduction of a defect in a related work product.
- For example, a requirements elicitation error can lead to a requirements defect, which then results in a programming error that leads to a defect in the code.
- If a defect in the code is executed, this may cause a failure, but not necessarily in all circumstances.
- For example, some **defects require very specific inputs or preconditions to trigger a failure, which may occur rarely or never.**

Functional vs Non-Functional Testing

- Functional testing involves tests that evaluate functions that the system should perform.
- Functional requirements may be described in work products such as business requirements specifications, epics, user stories, use cases, or functional specifications, or they may be undocumented. The functions are “what” the system should do.
- Non-functional testing of a system evaluates characteristics of systems and software such as **usability, performance efficiency or security**. Non-functional testing is the testing of “how well” the system behaves. This is often called quality requirements.
- It is common to consider “login” features to be under non-functional requirements, however at the end of the day everything depends on the context.
- It is important to understand and map out functional requirements at all times.

The Seven Testing Principles (ISTQB)



Testing shows the presence of defects, not their absence

- Dijkstra (1969) said that testing merely shows the presence of defects, but it does **not mean there are no defects.**
- Testing can show that defects are present.
- It cannot prove that there are no defects.
- Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, **testing is not a proof of correctness**.

Exhaustive testing is impossible

- In this type of testing all probable data combinations are used for test execution.
- Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases.
- For instance, consider an application with **10 input fields where each can have six possible values**. In order to perform exhaustive testing, number of combinations are more than what you think it is. Can you estimate the number of test cases in this scenario?
- Rather than attempting to test exhaustively, risk analysis, test techniques, and priorities should be used to focus test efforts. In other words, there exist strategies like Pair Wise testing and etc..

Early testing saves time and money

- To find defects early, both static and dynamic test activities should be started as early as possible in the software development lifecycle.
- Early testing is sometimes referred to as shift left testing. (Shifting the testing stage to an earlier time)
- Testing early in the software development life cycle helps reduce or eliminate costly changes.

Defects cluster together

- A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures.
- Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort.
- Defect Clustering in Software Testing means that the **majority of the defects are caused by a small number of modules**, i.e. the distribution of defects are not across the application but rather centralized in limited sections of the application.
- Most of the defects are clustered in particular sections of the application of an application e.g. Product Details, Reviews and Search Results Page for an e-commerce website and the remaining defects in other areas.

Beware of the pesticide paradox

- If the same tests are repeated over and over again, eventually these tests no longer find any new defects.
- To detect new defects, existing tests and test data may need changing, and new tests may need to be written. (Tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while.)
- In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects.

Testing is context dependent

- Testing is done differently in different contexts.
- For example, safety-critical industrial control software is tested differently from an e-commerce mobile app.
- As another example, testing in an Agile project is done differently than testing in a sequential lifecycle project.
- Testing focus will depend on what is more important for that type of application.

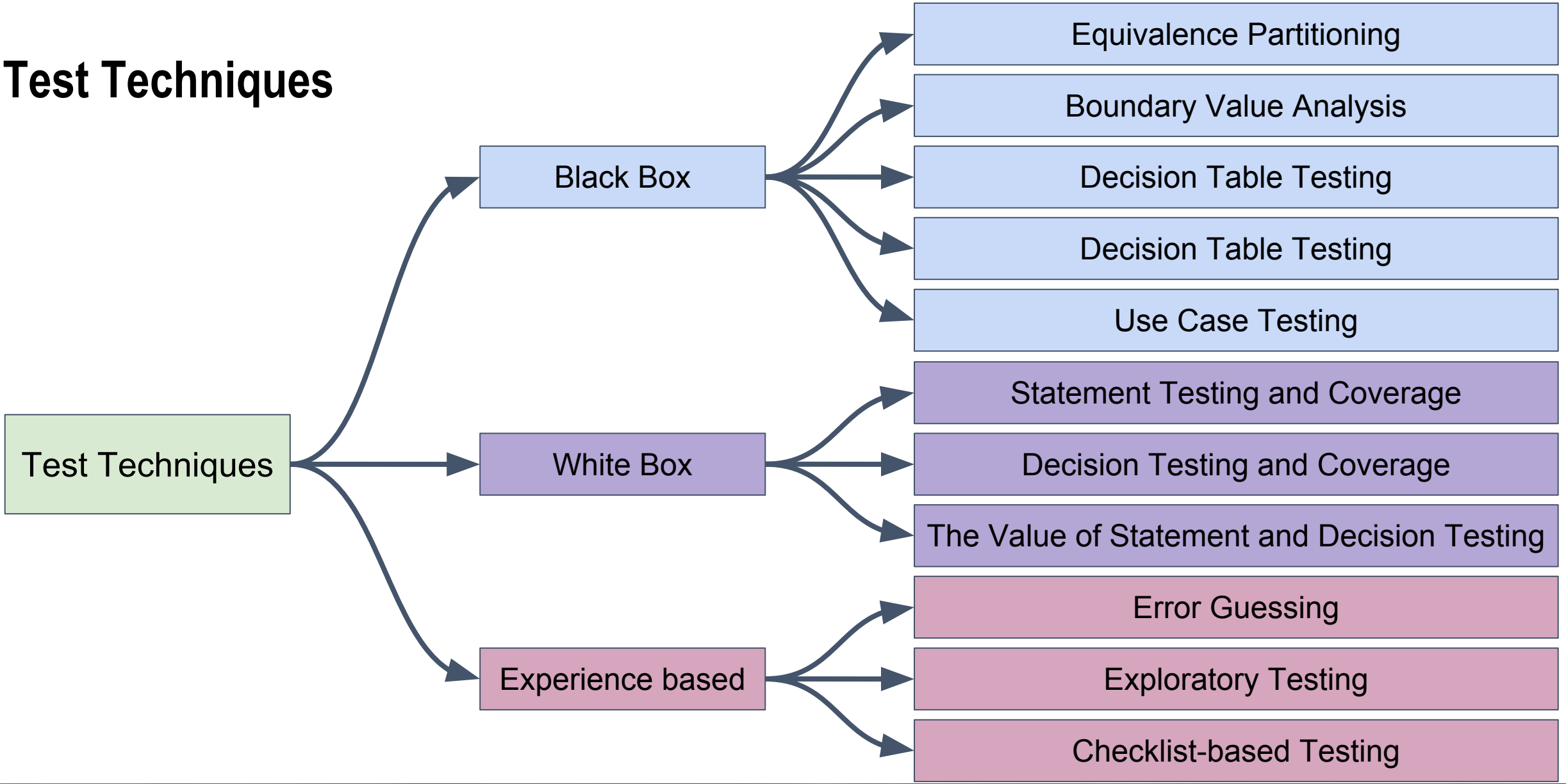
Absence-of-errors is a fallacy

- Some organizations expect that testers can run all possible tests and find all possible defects, but principles 2 and 1, respectively, tell us that this is impossible.
- Further, it is a fallacy (i.e., a mistaken belief) to expect that just finding and fixing a large number of defects will ensure the success of a system.
- For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfill the users' needs and expectations, or that is inferior compared to other competing systems.

Human Psychology and Testing

- Some people may perceive testing as a **destructive activity**, even though it contributes greatly to project progress and product quality.
- Since developers expect their code to be correct, they have a **confirmation bias** that makes it difficult to accept that the code is incorrect.
- It is a common human trait to **blame the bearer of bad news**, and information produced by testing often contains bad news.
- An element of human psychology called confirmation bias can make it difficult to accept information that disagrees with currently held beliefs.
- This is the main reason it is said that software developers do not like testers. (Please do not hate your tutor if he or she breaks your web application.)

Test Techniques



Blackbox

In black box test, it is **common not to look** into the source codes, however it is not wrong if it is done. As long as there are inputs and outputs.

- Black-box test techniques (also called behavioral or behavior-based techniques) are based on an analysis of the appropriate test basis (e.g., **formal requirements documents, specifications, use cases, user stories, or business processes**).
- These techniques are applicable to both **functional and nonfunctional testing**.
- Black-box test techniques concentrate on the **inputs** and **outputs** of the test object **without reference to its internal structure.**

Common Characteristics of Black Box Testing

- Test conditions, test cases, and test data are derived from a test basis that may include **software requirements, specifications, use cases, and user stories**.
- Test cases may be used to detect gaps between the requirements and the implementation of the requirements, as well as deviations from the requirements.
- Coverage is measured based on the items tested in the test basis and the technique applied to the test basis.
- Examples of black box test techniques are **Equivalence Partitioning, Boundary Value Analysis, Decision Table Testing, State Transition Testing and Use Case Testing.**

Whitebox

- White-box test techniques (also called structural or structure-based techniques) are based on an analysis of the architecture, detailed design, internal structure, or the code of the test object.

- Unlike black-box test techniques, white-box test techniques **concentrate on the structure and processing within the test object.**

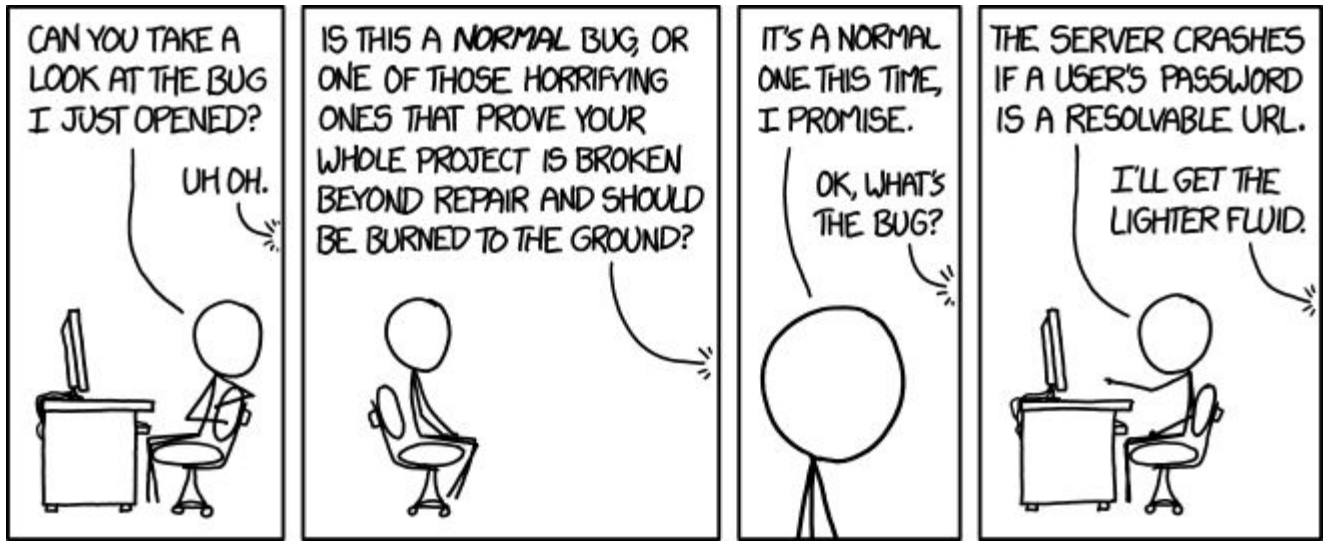
Common Characteristics of Whitebox

- Test conditions, test cases, and test data are derived from a test basis that may include **code, software architecture, detailed design, or any other source of information regarding the structure of the software.**
- Coverage is measured based on the items tested within a selected structure (e.g., the code or interfaces)
- Specifications are often used as an additional source of information to determine the expected outcome of test cases.
- White-box test techniques can be used at all test levels.
- Examples of white-box test techniques are **Statement Testing and Coverage, Decision Testing and Coverage, and The Value of Statement and Decision Testing.**

Experience-based test techniques

- Test cases are derived from the tester's skill and intuition, and their experience with similar applications and technologies.
- These techniques can be helpful in identifying tests that were not easily identified by other more systematic techniques.
- Depending on the tester's approach and experience, these techniques may achieve widely varying degrees of coverage and effectiveness.
- Coverage can be difficult to assess and may not be measurable with these techniques.
- Examples of experienced based test techniques are **Error Guessing, Exploratory Testing, and Checklist-based Testing.**

Relevant XKCD

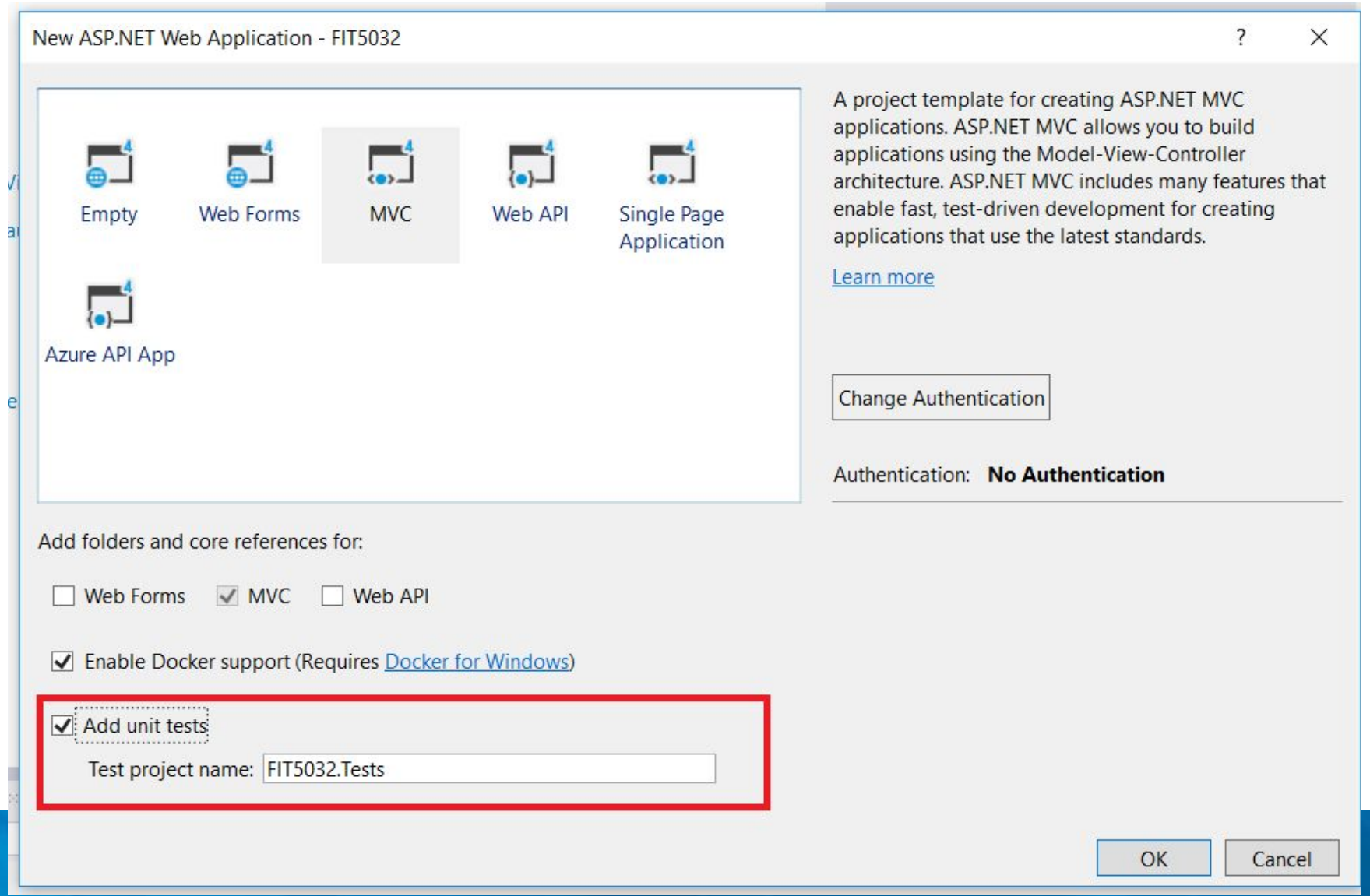


Bugs happen in all projects. Some bugs are hard to reproduce, others stem from developers taking shortcuts and their lack of understanding.

At the end of the day, bugs needs to be fixed in a proper manner and not dodged or burnt with a fire.

TDD in ASP.NET MVC

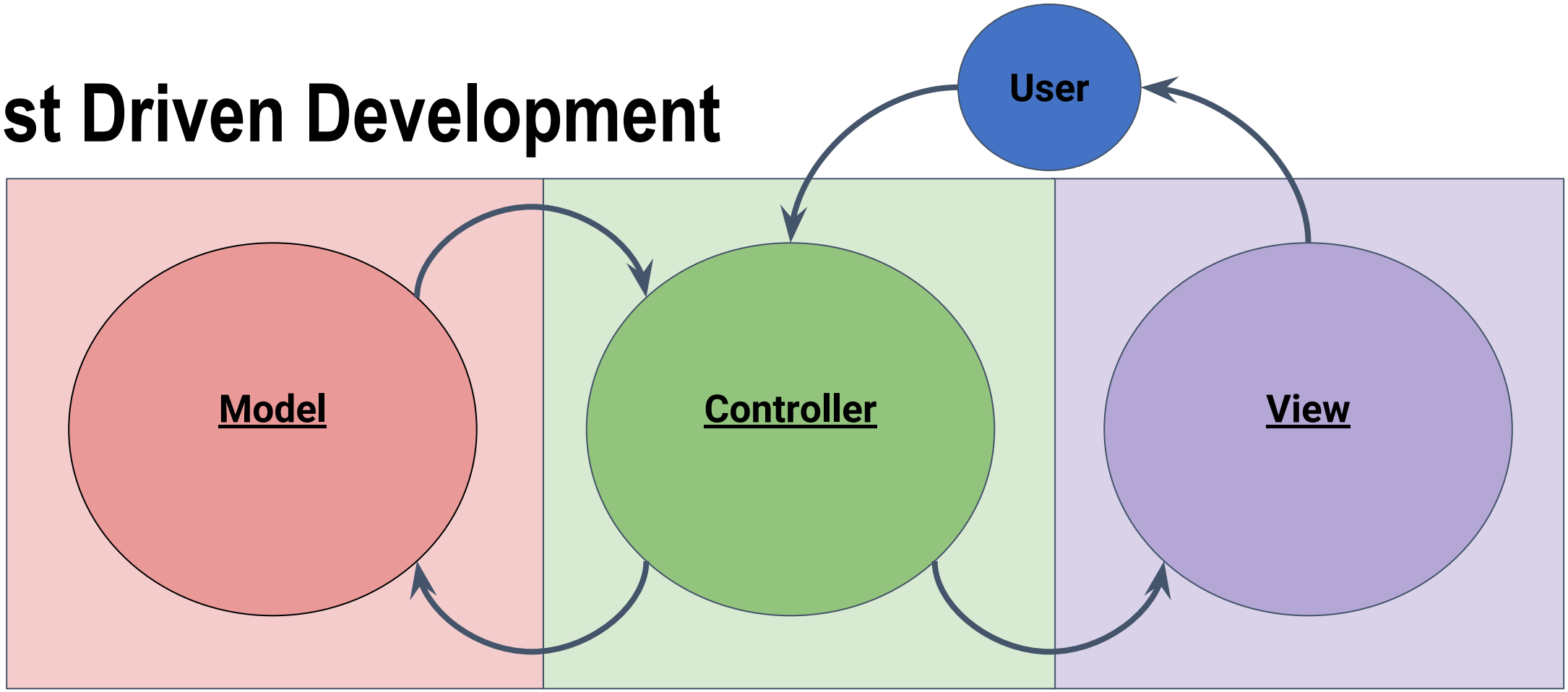
- One of the benefits of using ASP.NET MVC is that it supports TDD.
- Test-driven development (TDD) is a software development process that relies on the **repetition of a very short development cycle:**
- First the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.
- In other words, in TDD test cases are written first.



By default, you can create the project with the default template for Unit Testing.

At the end of the day, Test cases must be manually written even though there are tools that might help with it.

Test Driven Development



Model Testing.

Most common approach is to test controllers.

Front End Testing

In TDD, test cases are written first.

Integration Test

Integration Test

Testing Controllers

- Controllers are a central part of any ASP.NET Core MVC application.
- Automated tests can provide you with this confidence and can detect errors before they reach production. **(Automated test does not mean we do not have to write test cases!)**
- It's important to avoid placing unnecessary responsibilities within your controllers and ensure your tests focus only on controller responsibilities.
- Controller logic should be **minimal** and **not be focused on business logic** or **infrastructure concerns** (for example, data access).
- Test controller logic, not the framework. Test how the controller behaves based on valid or invalid inputs. Test controller responses based on the result of the business operation it performs. **(We do not test library functions!)**

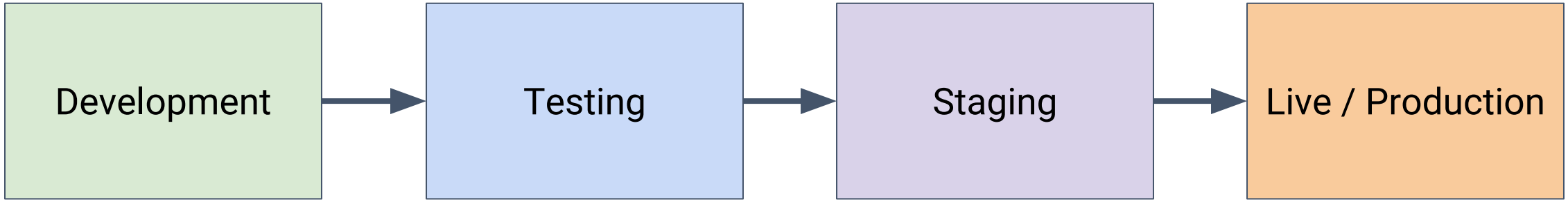
Testing Models

- Model represents domain specific data and business logic in MVC architecture.
- It maintains the data of the application. Model objects retrieve and store model state in the persistence store like a database.
- Our objective for this unit is to create **“fat models” and “thin controllers”**. This means that the majority of our business logic should be in the model layer, while the controllers should be kept thin only handling input and output.
- The models in our MVC architecture can have its own architecture pattern. For example, you can include a repository design pattern or also introduce a service layer for a service oriented architecture pattern. At the end of the day, it is important for this layer to be tested.

Software Development

- It is a common misconception that whenever a software or website is build and delivered to the client the process of software development ends.
- This is often not the case, because modern software especially web applications should be properly **deployed and maintained continuously** by the team.
- The process of deployment and maintenance is often times more complicated than the development process itself.
- The act of shipping the software with the required tools to run it is often coined software deployment. For this subject, it is important to understand that a website should reside on the remote server and not the local development environment. So, it must be deployed. There are different deployment environments.

Types of Deployment Environment



Different companies would have different deployment environments. Some companies may just move straight to production from development. (Skipping testing and staging stages).

In many subjects, you are only working in your own development environment however this is not the case in real life. (The real difficulty only comes in the live and production environments.) :(

Development Environment

- The development environment (dev) is the environment in which changes to software are developed, most simply an individual developer's workstation.

- This differs from the ultimate target environment in various ways – the target may not be a desktop computer (it may be a smartphone, embedded system, headless machine in a data center, etc.), and even if otherwise similar, the developer's environment will include development tools like a compiler, integrated development environment, different or additional versions of libraries and support software, etc., which are not present in a user's environment.

- In ASP MVC.NET we will aim to deploy our solution to the IIS Server. (Remote server, or the cloud server) but our development environment would be our own computer. It is important to note that the remote server might have different configurations in comparison to our development environment.

Testing Environment

- The purpose of the test environment is to allow human testers to exercise new and changed code via either automated checks or non-automated techniques.
- After the developer accepts the new code and configurations through unit testing in the development environment, the items are moved to one or more test environments.
- Upon test failure, the test environment can remove the faulty code from the test platforms, contact the responsible developer, and provide detailed test and result logs. If all tests pass, the test environment or a continuous integration framework controlling the tests can automatically promote the code to the next deployment environment.
- Different types of testing suggest different types of test environments, some or all of which may be virtualized to allow rapid, parallel testing to take place.

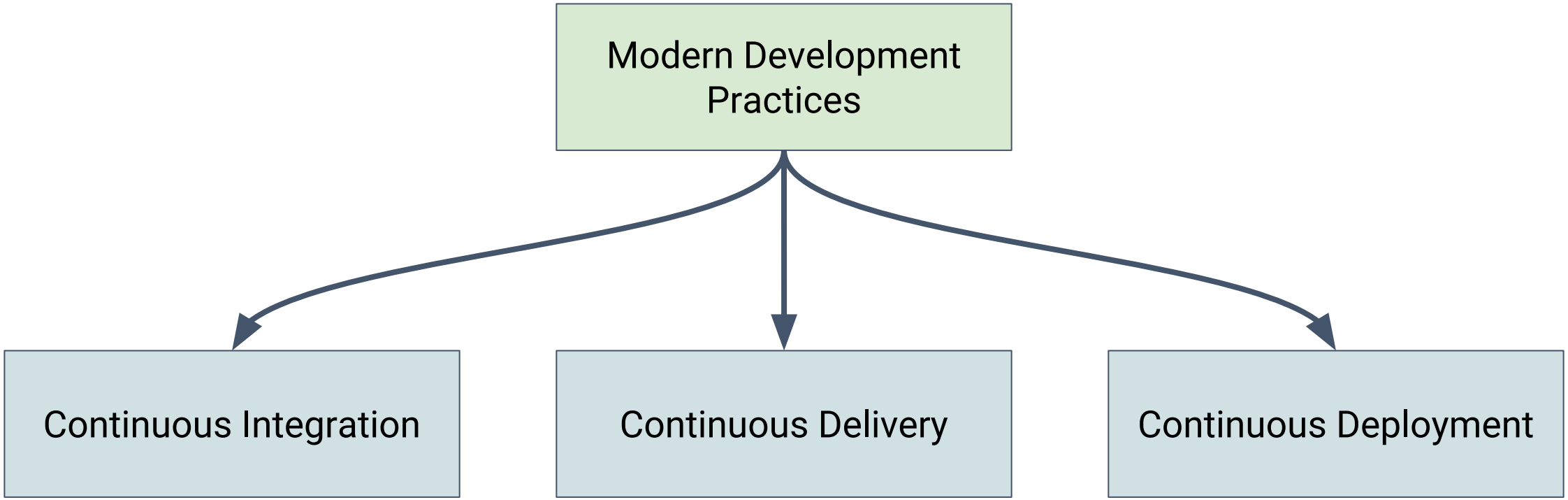
Staging Environment

- Staging is an environment for final testing immediately prior to deploying to production.
- It seeks to mirror the actual production environment as closely as possible and may connect to other production services and data, such as databases.
- For example, servers will be run on remote machines, rather than locally (as on a developer's workstation during dev, or on a single test machine during test), which tests the effect of networking on the system.
- The primary use of a staging environment is to test all installation/configuration/migration scripts and procedures, before they are applied to production environment. This ensures that all major and minor upgrades to the production environment will be completed reliably without errors, in minimum time.

Live / Production Environment

- The production environment is also known as live, particularly for servers, as it is the environment that users directly interact with.
- Deploying to production is the most sensitive step; it may be done by deploying new code directly (overwriting old code, so only one copy is present at a time), or by deploying a configuration change.
- This can take various forms: deploying a parallel installation of a new version of code, and switching between them with a configuration change; deploying a new version of code with the old behavior and a feature flag, or by deploying separate servers (one running the old code, one the new) and redirecting traffic from old to new with a configuration change at the traffic routing level.
- These in turn may be done all at once or gradually, in phases.

Modern Development Practices



These 3 terms are often used to describe modern development practices. It is important to understand the difference between them.

Continuous Delivery

- Continuous delivery (CD) is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time.
- It aims at building, testing, and releasing software with greater speed and frequency. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production.
- Continuous Delivery is the ability to get changes of all types—including new features, configuration changes, bug fixes and experiments—into production, or into the hands of users, safely and quickly in a sustainable way.
- A straightforward and repeatable deployment process is important for continuous delivery.
- Google's Chrome web browser is largely hailed as a feat of continuous delivery engineering. Google Chrome was released in 2011.

Continuous Integration

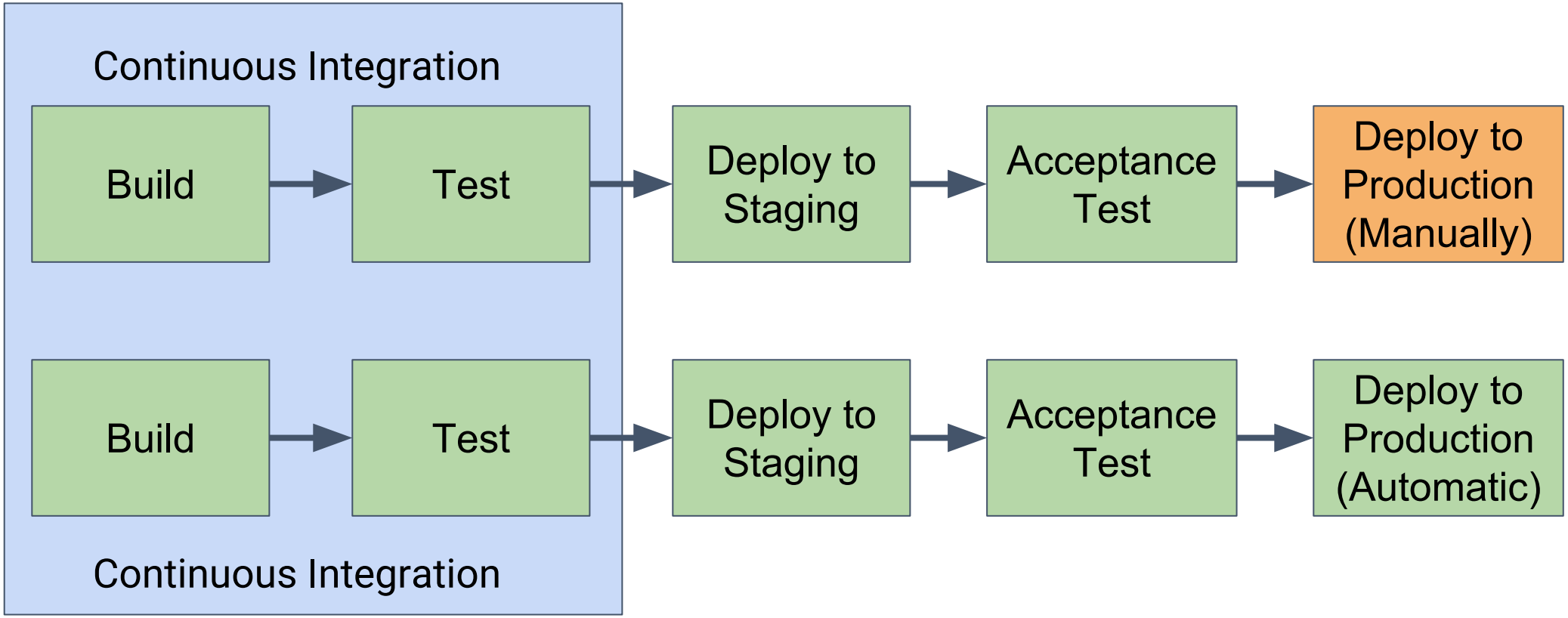
- Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day.
- Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.
- Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Continuous Deployment

- Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

- Continuous deployment is an excellent way to accelerate the feedback loop with your customers and take pressure off the team as there isn't a Release Day anymore. Developers can focus on building software, and they see their work go live minutes after they've finished working on it.

Continuous Delivery



Continuous Deployment

The major difference here is that in continuous delivery, the deploy to production stage happens manually while the other is automated.

Benefits of Continuous Integration

- Less bugs get shipped to production as regressions are captured early by the automated tests.
- Building the release is easy as all integration issues have been solved early.
- Less context switching as developers are alerted as soon as they break the build and can work on fixing it before they move to another task.
- Testing costs are reduced drastically – your CI server can run hundreds of tests in the matter of second.
- Your QA team spend less time testing and can focus on significant improvements to the quality culture.

Benefits of Continuous Delivery

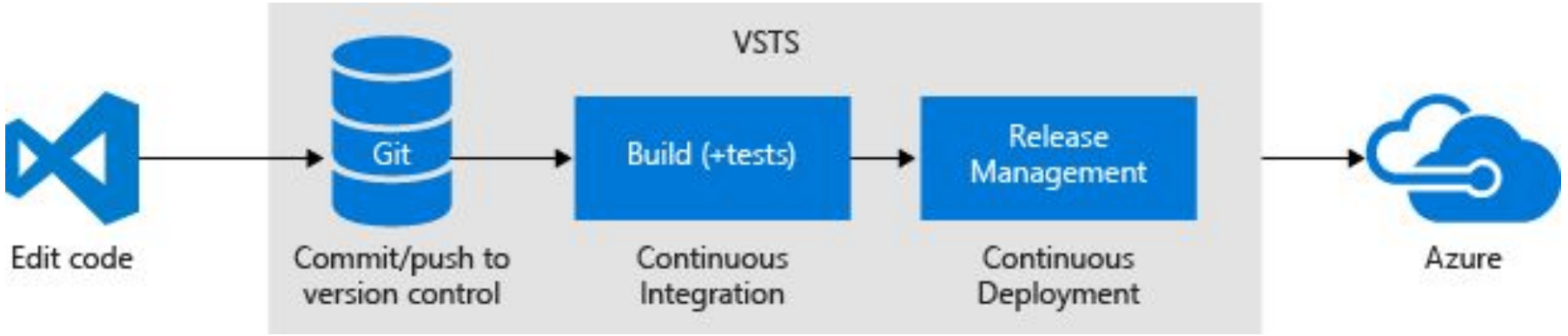
- The complexity of deploying software has been taken away. Your team doesn't have to spend days preparing for a release anymore.
- You can release more often, thus accelerating the feedback loop with your customers.
- There is much less pressure on decisions for small changes, hence encouraging iterating faster.

Benefits of Continuous Deployment

- You can develop faster as there's no need to pause development for releases. Deployments pipelines are triggered automatically for every change.
- Releases are less risky and easier to fix in case of problem as you deploy small batches of changes.
- Customers see a continuous stream of improvements, and quality increases every day, instead of every month, quarter or year.

Visual Studio Team Services

Modern applications are developed using **Continuous Integration** & **Continuous Development.**



Docker

- Docker is a platform built on top of lightweight containers.
- It handles much of the work around handling containers for you. In Docker, you create and deploy apps, which are synonymous with images in the VM world, albeit for a container-based platform.
- Docker manages the container provisioning, handles some of the networking problems for you, and even provides its own registry concept that allows you to store and version Docker applications.
- Visual Studio 2017 supports the use of Docker and Azure supports deployment via Docker.

It is honestly hard to cover everything important in one lecture, so the general idea is that these days, deployment is done via containers like Docker.

In the labs

You will learn how to

- deploy your web application to Azure server.

Summary

1. What is Testing?
2. Errors, Defects and Failures.
3. Functional Testing vs Non-Functional Testing.
4. The Seven Testing Principles
5. Human Psychology and Testing.
6. Test Techniques
7. TDD in ASP.NET MVC
8. Software Development & Deployment
9. Types of Deployment Environments
10. Modern Development Practices
11. Visual Studio Team Services.
12. Docker