

SENG 275

# SOFTWARE TESTING

DR. NAVNEET KAUR POPLI

DEPT. OF ELECTRICAL AND COMPUTER  
ENGINEERING



# TDD & BDD



University  
of Victoria



TDD

DEVELOP CODE  
DRIVEN BY TESTS



University  
of Victoria

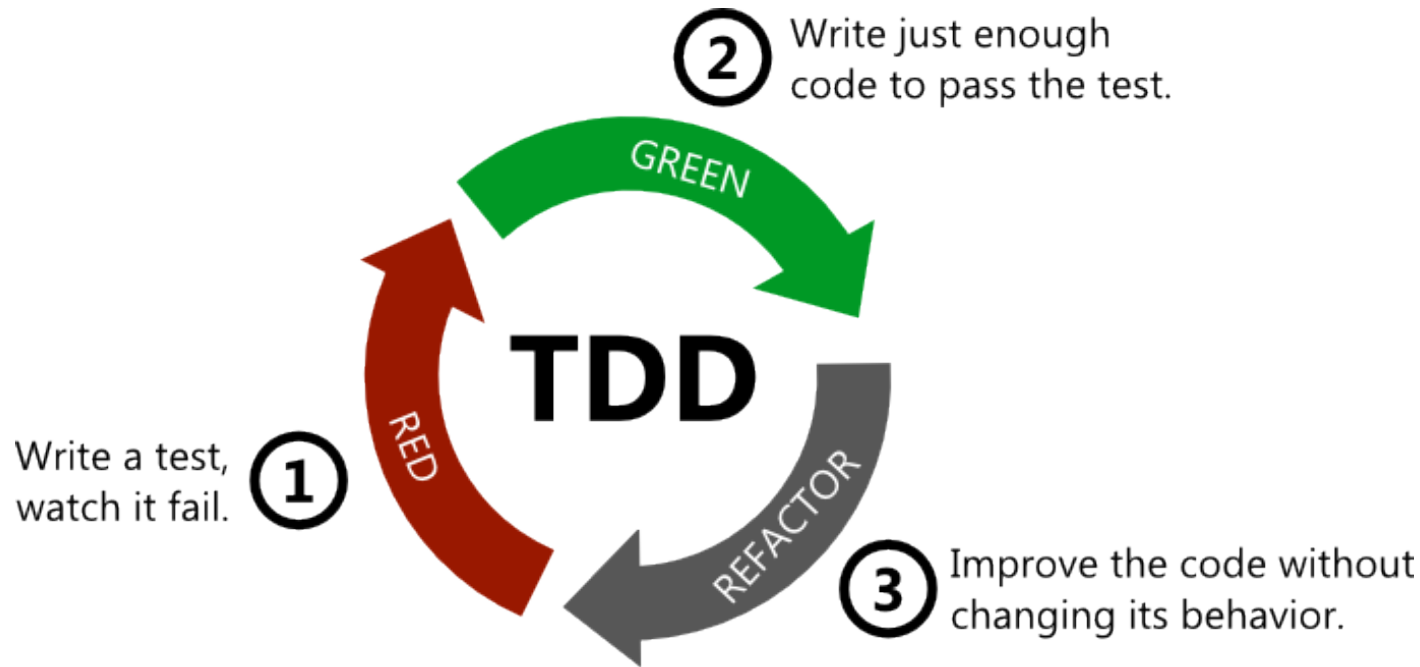
# Rules of the game

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

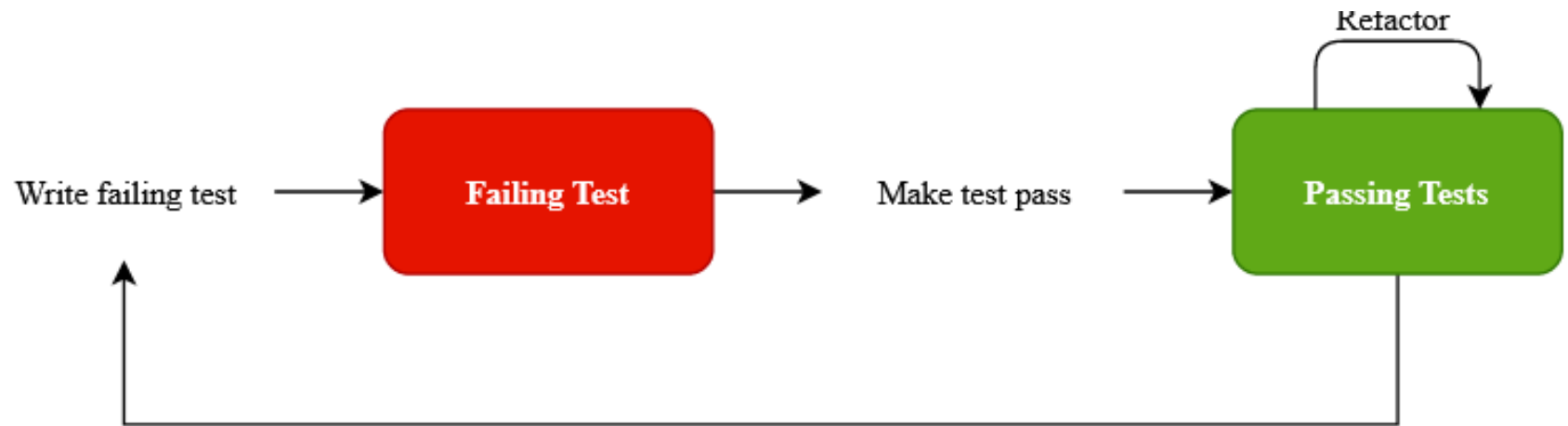


# TDD-Test then Code

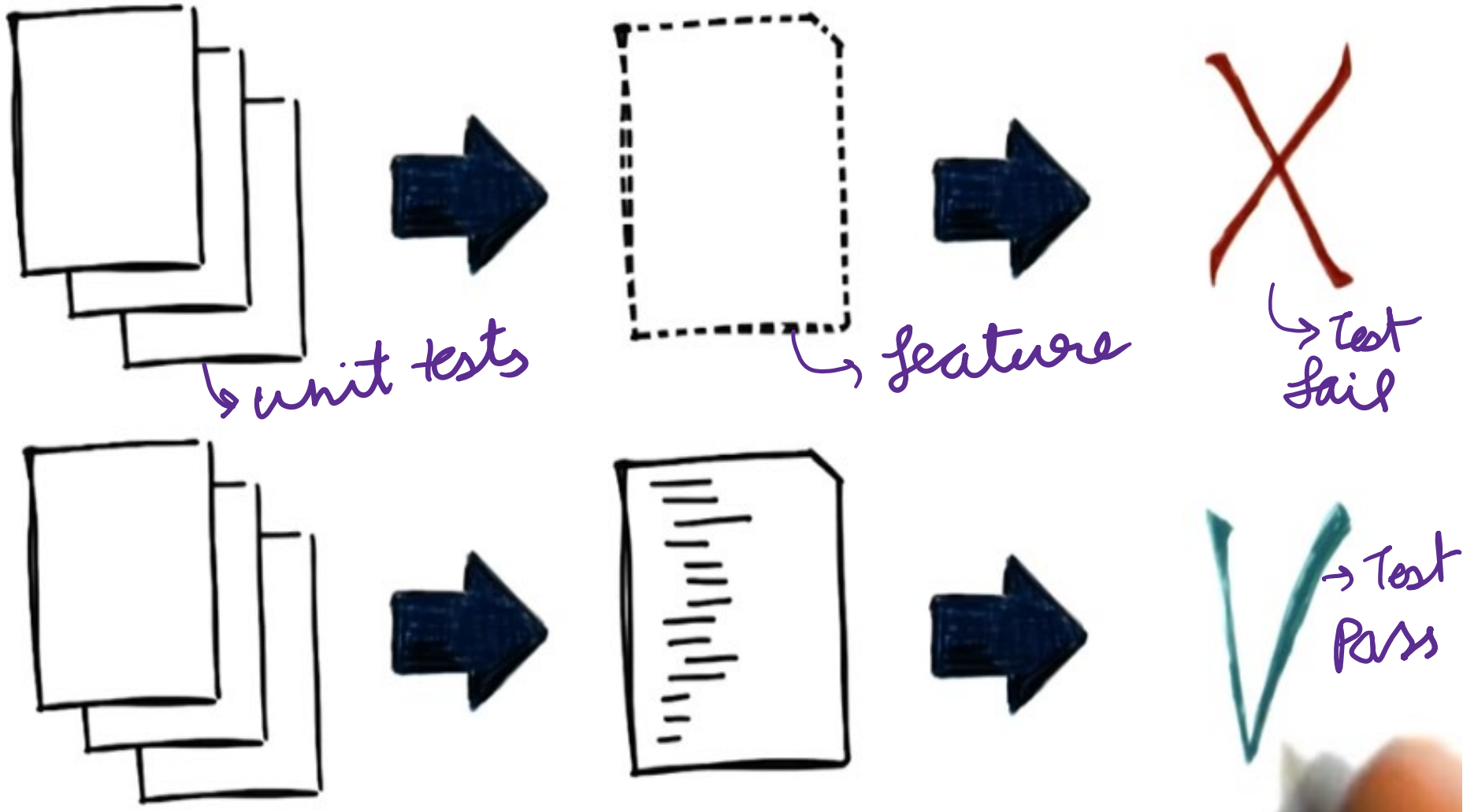
- **TDD (Test-driven development)** is a software development technique that entails writing automated test cases earlier than writing functional pieces of the code.



# TDD



# TEST-FIRST DEVELOPMENT



# Let's play the game

## Returning Customer

I am a returning customer

E-Mail Address

Password

[Forgotten Password](#)


Login




University  
of Victoria



## Play the game



Create test case to verify that if the correct email id and password is added and Login button is clicked, the user is navigated to the home page.



Implement just enough code to pass this test, no more.



Refractor and improve the code to say making the navigation speed better.



## Play the game

1. First, you act like you're a **demanding user** who wants to use the code that's about to be written in the simplest possible way. **You must write a test that uses a piece of code as if it were already implemented.**
2. Do not write a bunch of functions/classes that you think you may need. Concentrate on the feature you are implementing and on what is really needed. Writing something the feature doesn't require is over-engineering.



# Advantages

1



• Code Quality

2



• Code Coverage



# Why TDD?

- TDD developers use test cases before they write a single line of code.
- This approach encourages them to consider **how the software will be used** and what design it needs to have to provide the expected usability.
- Once a test fails, developers understand what needs to be changed and they refactor the code, that is, they rewrite it to improve it without altering its function.
- TDD focuses on the code necessary to pass the test, **reducing it to essentials**.
- It takes the **test unit**, or the smallest bit of functionality, as its basis.
- It's a bit like **building a house brick by brick**—no brick is laid down before it's tested, to make sure it's sound and that it's an integral part of the design.
- In TDD, you achieve **100% coverage test**. Every single line of code is tested, unlike traditional testing.



## Other advantages

1. Focus on the requirements
2. Controlling your pace
3. Testable code
4. Faster feedback
5. Revealing design problems early



# Focus on the requirements

Starting by the test means **starting by the requirements**. It makes us think more about:

- what we expect from the class.
- how the class should behave in specific cases.

We do not write "useless code"

- Go to your codebase right now. How much code have you written that is never used in real world?



# Controlling your pace

- Having a failing test, give us a clear focus: **make the test pass.**
- I can write whichever test I want:
  - If I feel insecure, I can write a simpler test.
  - If I feel safe, I can write a more complicated test.
  - If there's something I do not understand, I can take a tiny baby step.
  - If I understand completely, I can take a larger step.



## Testable code

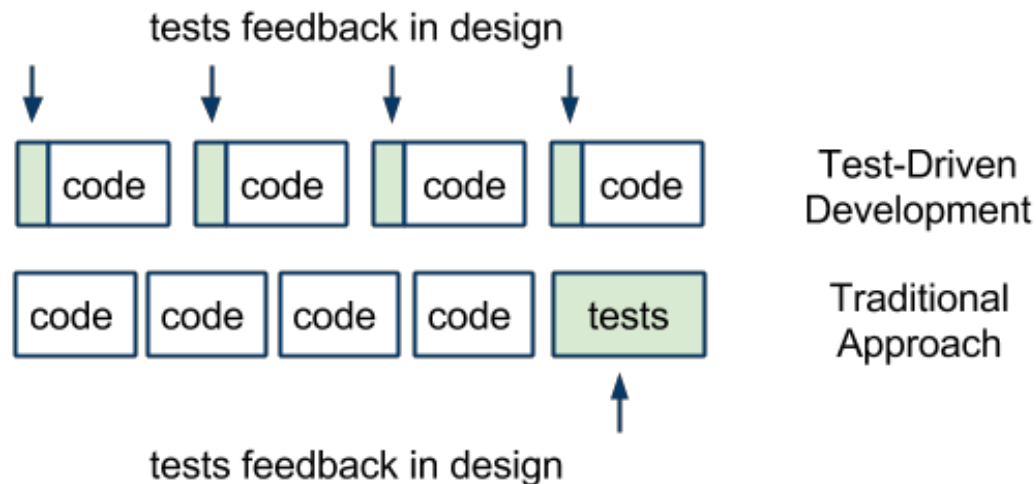
- TDD makes you think about tests from the beginning.
  - This means you will be enforced to **write testable classes**.
- A testable class is also an easy-to-use class.
- Some people call TDD as *Test-Driven Design*.





# Faster feedback

- You are writing tests frequently. This means you will find the problem sooner.
- Tests at the end also work. But maybe the feedback is just too late.



# Listen to your test- reveal design problems early

- The test may reveal design problems.
- You should **"listen to it"**.
- **Too many tests?**
  - Maybe your class **does too much**.
- **Too many mocks?**
  - Maybe your class is **too coupled**.
- **Complex set up** before calling the desired behavior?
  - Maybe rethink the **pre-conditions**.



# Create a junit test



```
import static org.junit.Assert.*;
import org.junit.Test;
// REQUIREMENTS
//0+0=0
//100+1=101
//1+100=101
//-19+19=0
//-19-19=-38
//2147483647+1=2147483648
```

```
public class CalculatorTest {

@Test
public void addTwoNumbers() {
Calculator obj=new Calculator();
assertEquals(0, obj.add(0,0));

}

}
```



This test fails because nothing is implemented as such.  
When you implement, the test passes.



```
public class Calculator {  
    public int add(int a, int b)  
    {  
        return a+b;  
    }  
  
}
```



I'm demonstrating that the acceptance criterion have been met, not writing any more code



```
import static org.junit.Assert.*;
import org.junit.Test;
// REQUIREMENTS
//0+0=0
//100+1=101
//1+100=101
//-19+19=0
//-19-19=-38
//2147483647+1=2147483648

public class CalculatorTest {

@Test
public void addTwoNumbers() {
    Calculator obj=new Calculator();
    assertEquals(0, obj.add(0,0));
    assertEquals(101, obj.add(100,1));
    assertEquals(101, obj.add(1,100));
}

}
```



University  
of Victoria

I am checking all my acceptance criterion and my tests are passing without further changes to my code.

```
import static org.junit.Assert.*;
import org.junit.Test;
// REQUIREMENTS
//0+0=0
//100+1=101
//1+100=101
//-19+19=0
//-19-1=-20
//2147483647+1=2147483648

public class CalculatorTest {
@Test
public void addTwoNumbers() {
Calculator obj=new Calculator();
assertEquals(0, obj.add(0,0));
assertEquals(101, obj.add(100,1));
assertEquals(101, obj.add(1,100));
assertEquals(0, obj.add(-19,19));
assertEquals(-20, obj.add(-19,-1));}}
```



# Checking the domain now. Integers can have max value 2147483647



```
public class CalculatorTest {  
  
    @Test  
    public void addTwoNumbers() {  
        Calculator obj=new Calculator();  
        assertEquals(0, obj.add(0,0));  
        assertEquals(101, obj.add(100,1));  
        assertEquals(101, obj.add(1,100));  
        assertEquals(0, obj.add(-19,19));  
        assertEquals(-20, obj.add(-19,-1));  
        assertEquals(2147483648, obj.add(2147483647,1));  
    }  
}
```

Markers Properties Servers Data Source Explorer Snippets Console JUnit Coverage

ed after 0.038 seconds

s: 1/1 Errors: 1 Failures: 0

CalculatorTest [Runner: JUnit 4] (0.000 s)

addTwoNumbers (0.000 s)

Failure Trace

java.lang.Error: Unresolved compilation problem:

# Checking last condition



```
assertEquals(2147483648L, obj.add(2147483647,1));
```

```
}
```

```
}
```



Markers Properties Servers Data Source Explorer Snippets Console JUnit Coverage

shed after 0.033 seconds

ns: 1/1

Errors: 0

Failures: 0



CalculatorTest [Runner: JUnit 4] (0.000 s)

Failure Trace



University  
of Victoria



Now you have to refactor the code  
and all the tests pass.



```
public class Calculator {  
    public long add(long a, long b)  
    {  
        return a+b;  
    }  
  
}
```



# HOW TDD FITS INTO DEVOPS, AGILE AND CI/CD



## WATERFALL

# DEPLOYMENT FREQUENCY



## AGILE



## DEVOPS



Company	Deployment Frequency
Amazon	23,000 per day
Google	5,500 per day
Netflix	500 per day
Facebook	1 per day
Twitter	3 per week
Typical enterprise	1 every 9 months

# Waterfall>agile>DevOps

- Waterfall-annual
- In the **agile** age, most companies would deploy/ship software in **monthly, quarterly, bi-annual, or even annual** releases (remember those days?).
- Now, however, in the **DevOps** era, **weekly, daily, and even multiple times a day** is the norm.
- Development teams have adapted to **the shortened delivery cycles by embracing automation** across their software-hardware delivery pipeline.



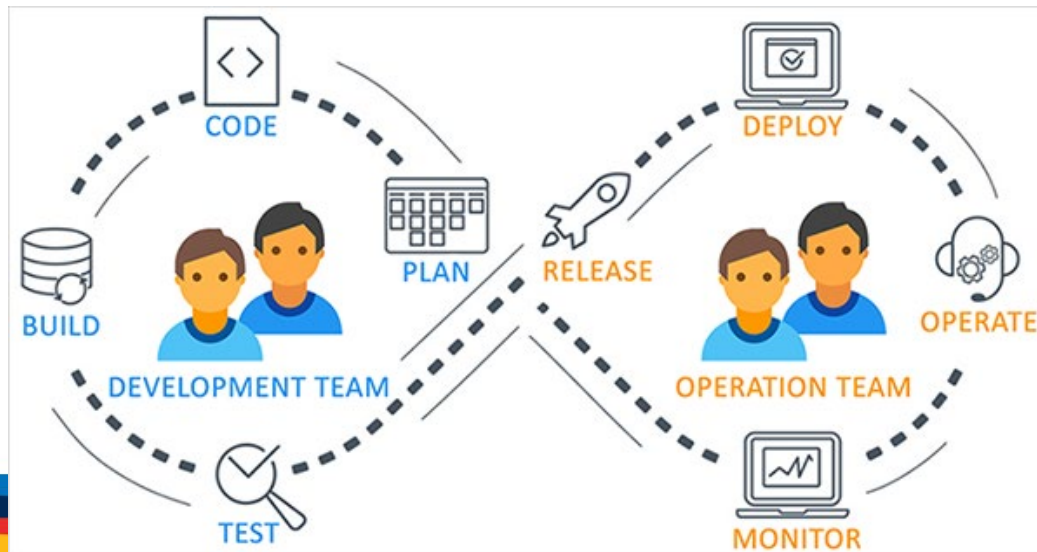
# DevOps = Dev + Ops

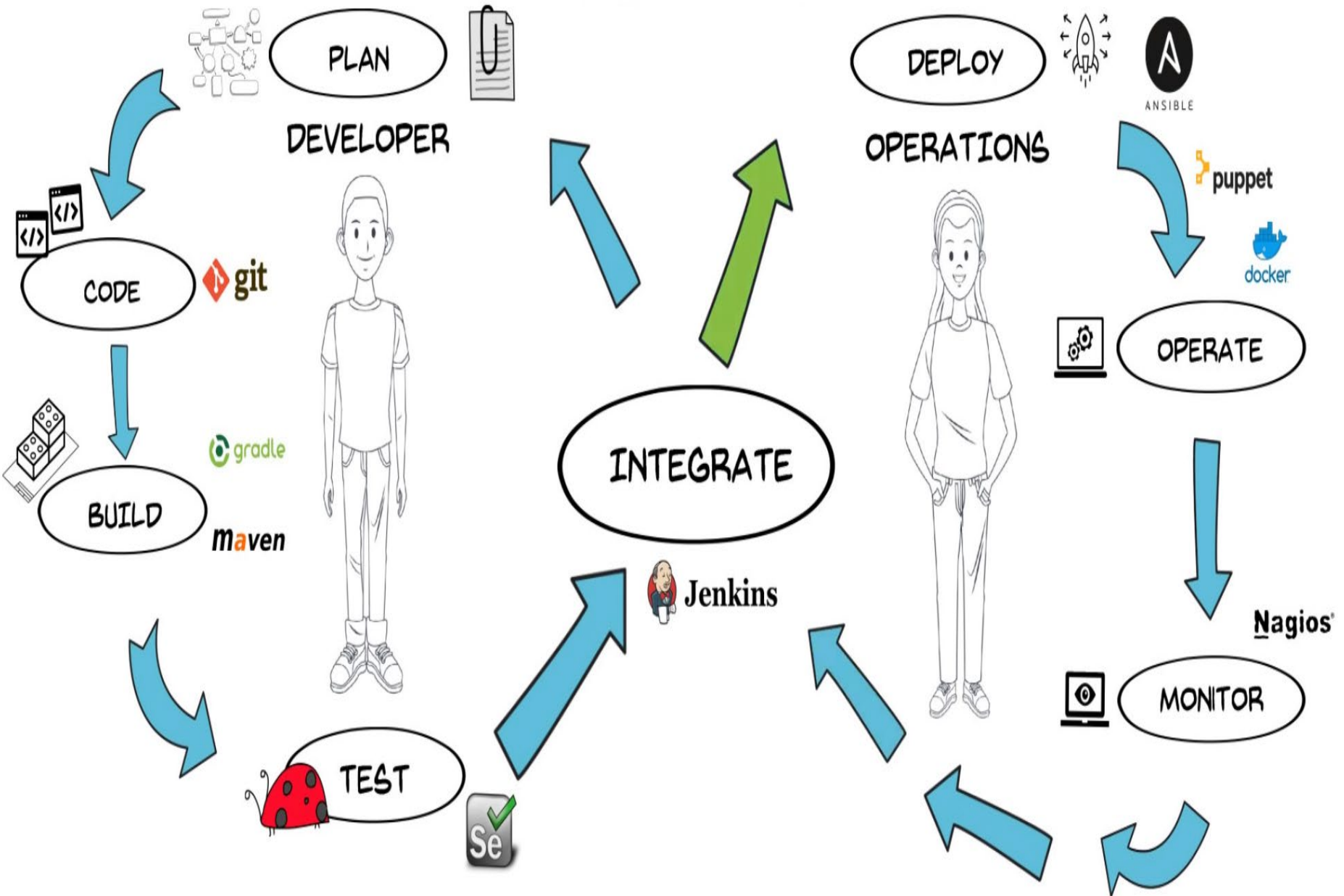
- **Developers** come from a mindset where **change is what they're paid to accomplish**.
- The business depends on them to respond to changing needs, so they're usually encouraged to create, innovate, and generate as much change as possible.
- By contrast, **operations see change as the enemy**.
- The business depends on operations to keep the lights on and deliver the services that generate money for the business today.
- Operations is motivated to resist change, because it undermines **stability and reliability**.



# DevOps

- DevOps is the **combination** of cultural philosophies, practices, and tools that increases an organization's ability to **deliver applications and services at high velocity**: evolving and improving products at a faster pace than organizations using traditional software-hardware development and infrastructure management processes.
- This speed enables organizations to **better serve their customers** and compete more effectively in the market.







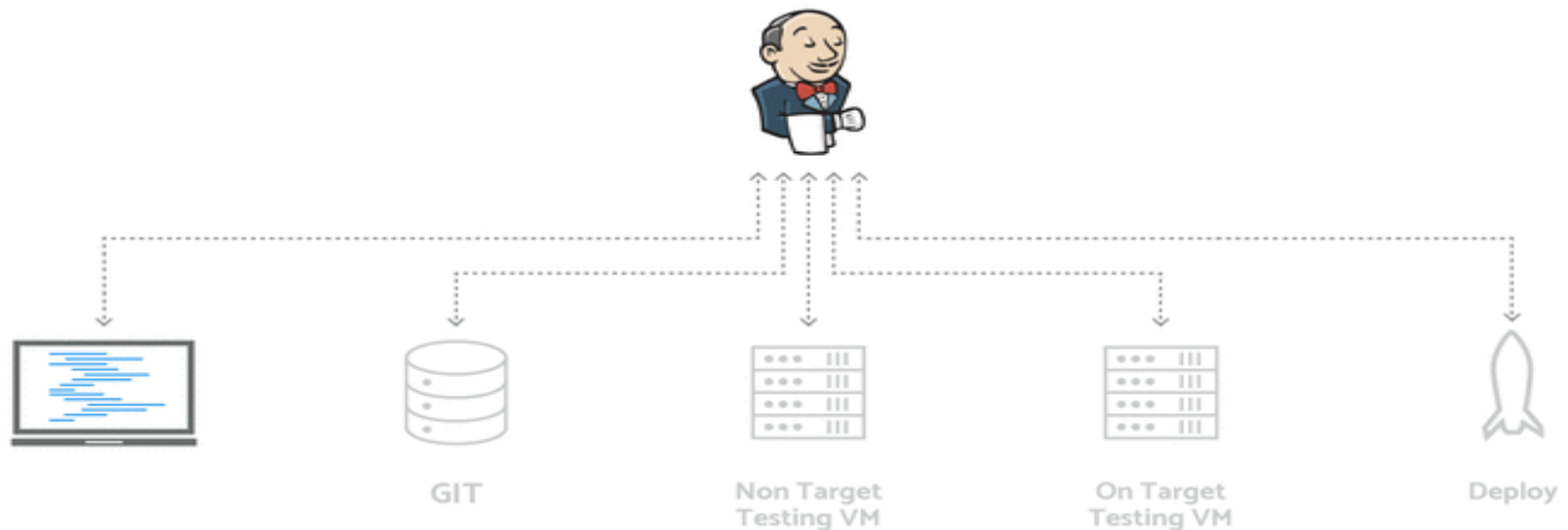
# CI/CD/CD as DevOps tools

- To foster a DevOps culture, implementing the right DevOps tools with the right DevOps process is essential.
- Continuous integration/continuous delivery/continuous deployment (CI/CD/CD) help developers and testers ship the software **faster and safer** in a structured environment.

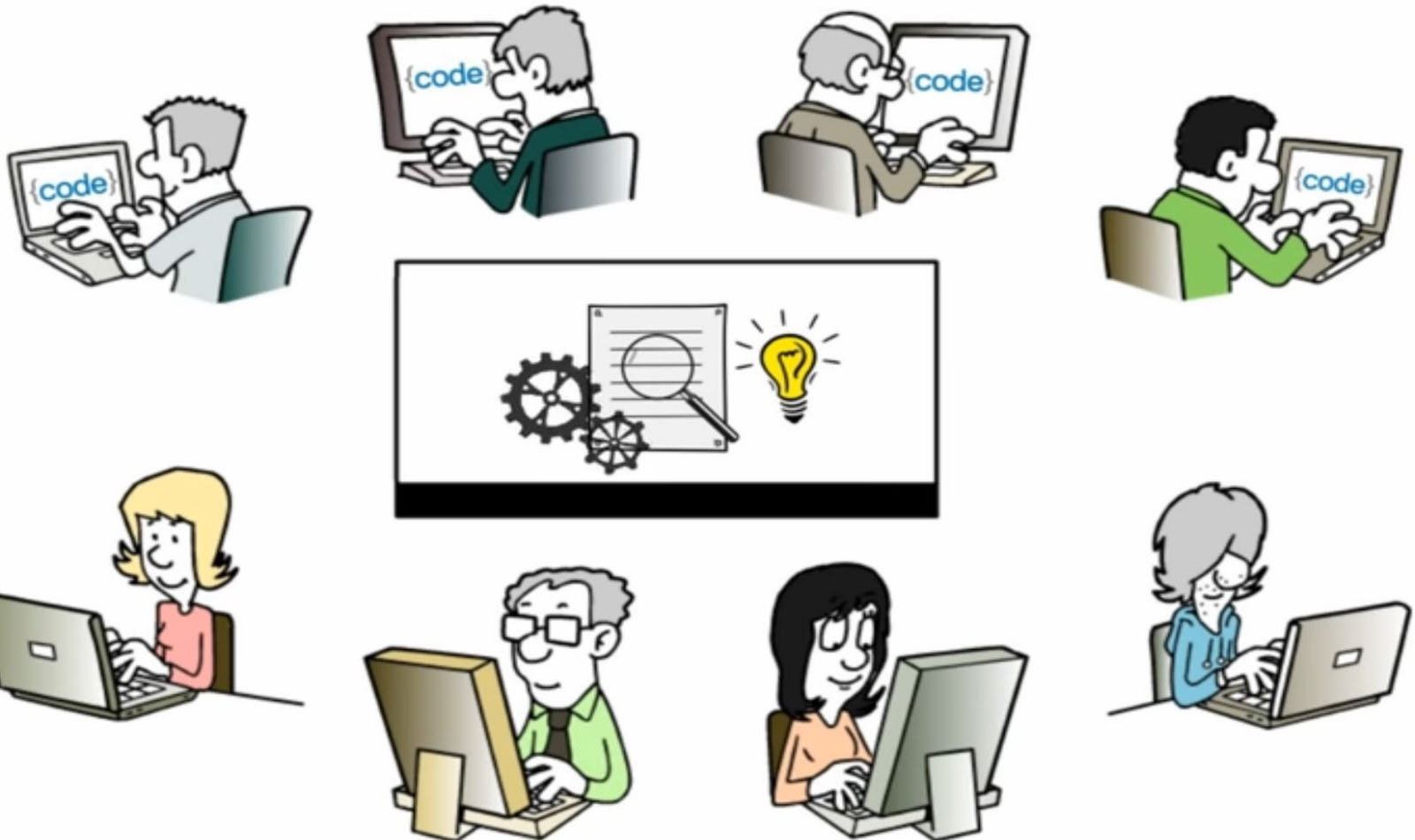


# How CI/CD works

In essence, a CI/CD pipeline is all about **automation**, enabling **developers to release changes and new features more frequently**. Simultaneously, it gives the **operations team confidence** that they can get those features in front of customers faster, while **maintaining quality and stability**.

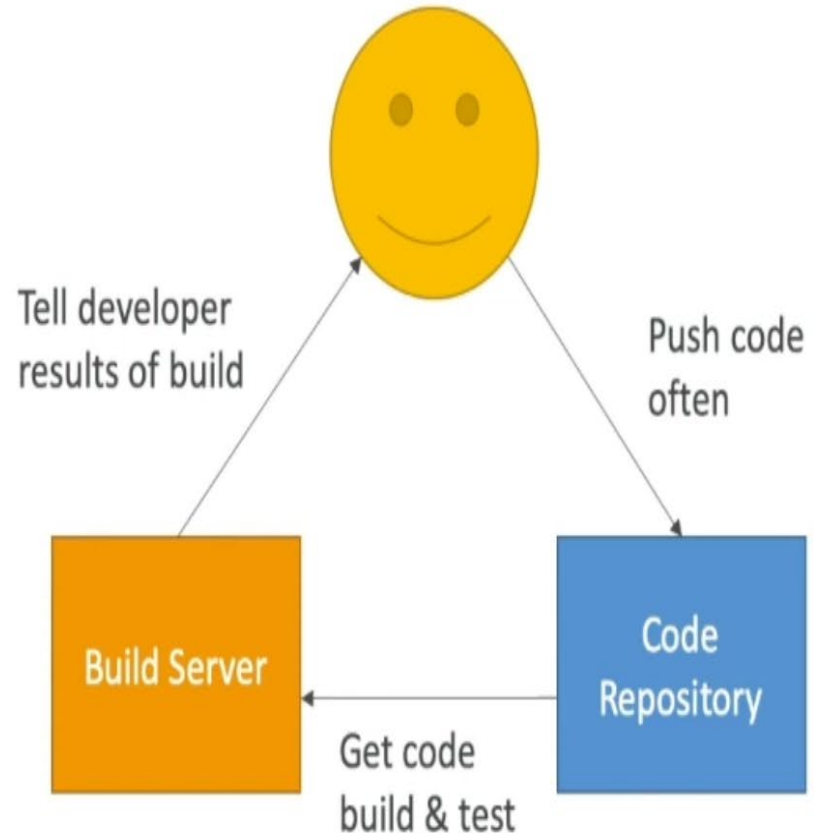


# CI-commit your code often



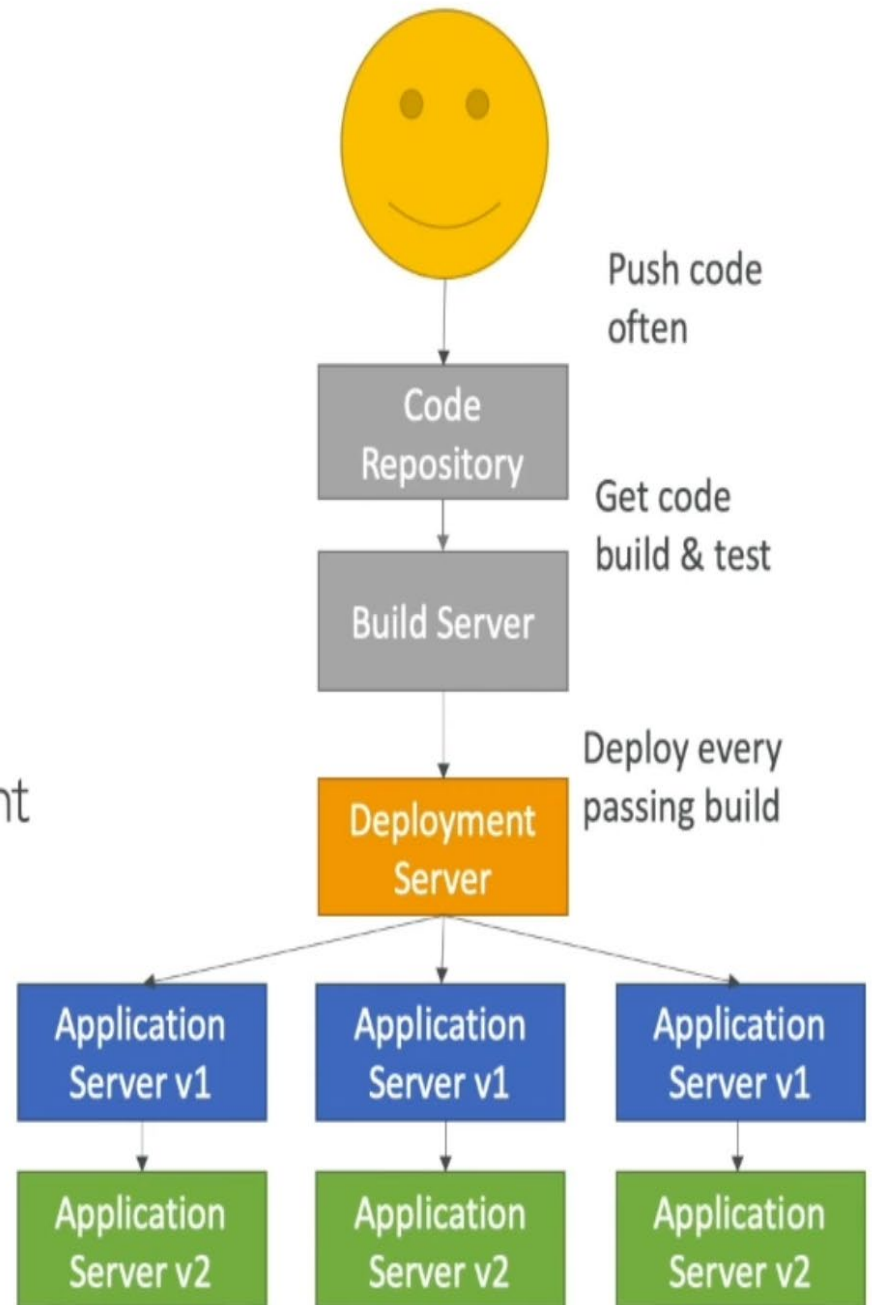
# Continuous Integration

- Developers push the code to a code repository often (GitHub / CodeCommit / Bitbucket / etc...)
- A testing / build server checks the code as soon as it's pushed (CodeBuild / Jenkins CI / etc...)
- The developer gets feedback about the tests and checks that have passed / failed
- Find bugs early, fix bugs
- Deliver faster as the code is tested
- Deploy often
- Happier developers, as they're unblocked

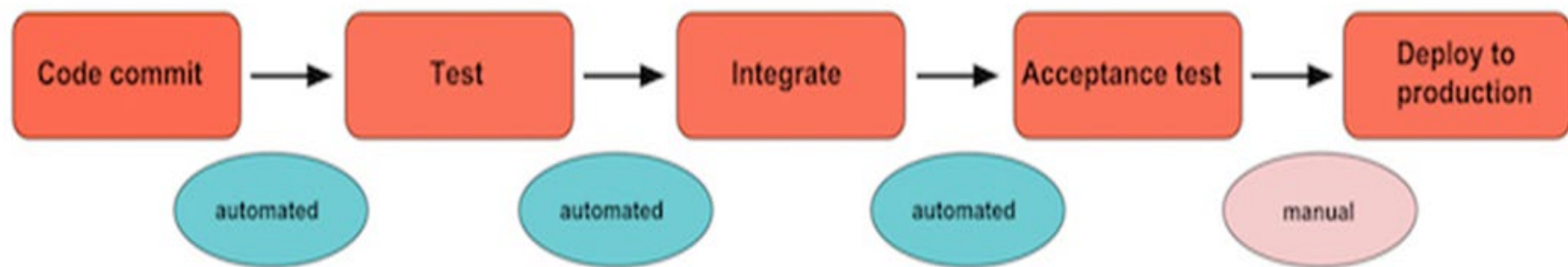


# Continuous Delivery

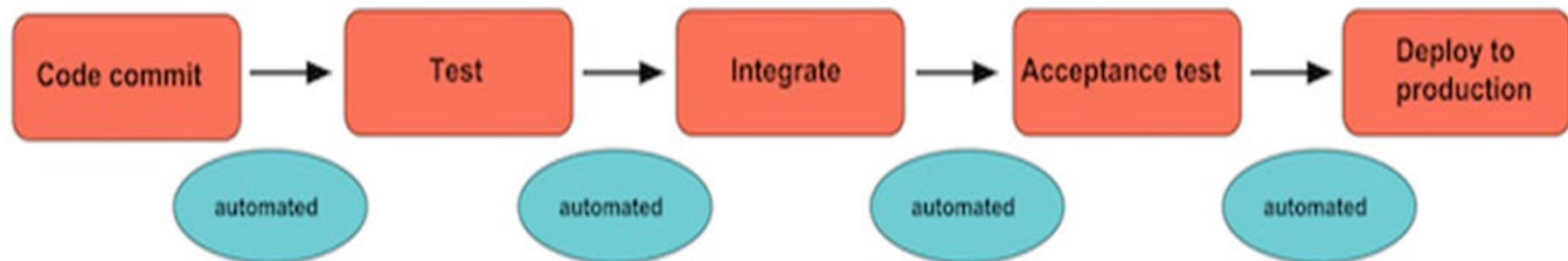
- Ensure that the software can be released reliably whenever needed.
- Ensures deployments happen often and are quick
- Shift away from “one release every 3 months” to “5 releases a day”
- That usually means automated deployment
  - CodeDeploy
  - Jenkins CD
  - Spinnaker
  - Etc...



# Continuous delivery



# Continuous deployment





# Continuous Delivery vs Continuous Deployment

- Continuous Delivery:
  - Ability to deploy often using automation
  - May involve a manual step to “approve” a deployment
  - The deployment itself is still automated and repeated!
- Continuous Deployment:
  - Full automation, every code change is deployed all the way to production
  - No manual intervention of approvals

# DevOps and CI/CD Case Study- Tesla





# Some cars today boast more than 300M lines of code!

- DevOps for software embedded in vehicles is not trivial. The automotive industry faces unique challenges when it comes to delivering software -- due to the complex testing matrix and deployment processes, and its **strict safety, regulation and compliance rules**.
- **DevOps and CI/CD automation** enables automotive manufacturers to **accelerate their releases** while ensuring security and mitigating the risk of failed/recalled software releases.
- The **ISO/IEC 15504 standard**, also referred to as **Automotive SPICE (Software Process Improvement and Capability Determination)**, defines a set of principle documents for the software development process, and this is just one such standard for the automotive industry. There are others, too, such as **Automotive Open System Architecture (AUTOSAR), IEC 61508, ISO 26262 and MISRA C**, etc.



## Traditional car vendors

- Over the years, car vendors have worked hard to build a **better piece of hardware**.
- They've built **stronger engines, tweaked car designs** and piled on the **comforts**.
- They've integrated **software** into the mix – **fuel sensors, anti-lock braking systems, cruise control, back-up collision avoidance systems**, etc. – but once developed and incorporated into the car, they are essentially **one-time innovations burned into the car at production time**.



# Tesla car- sophisticated computer on wheels

- **Tesla cars** are connected to the **company's software labs** the way apps on mobile phones are tied to Facebook and iTunes.
- This means Tesla can **push fixes and add new features** to their cars **continuously**, without the user having to go to a **repair shop to get the changes physically applied**.
- This wouldn't be possible without an **agile delivery system**, stoked by **continuous integration, continuous delivery and DevOps practices**.



# Tesla brought major changes to the car market

- Tesla's delivery mechanism changes the economics of the car market.
- **Conventional wisdom** holds that a **new car depreciates in value by at least 10 percent the minute the buyer drives it off the lot.**
- Tesla vehicles, on the other hand, are getting new features pumped into them regularly, so they, in theory, should not only **hold their value but could increase in value over a period of time.**



# Tesla OTA Updates



University  
of Victoria

# Power of software-hardware vertical integration, AI and data.

OTA updates general

<https://www.youtube.com/watch?v=zKu4Pg0HH4Y>

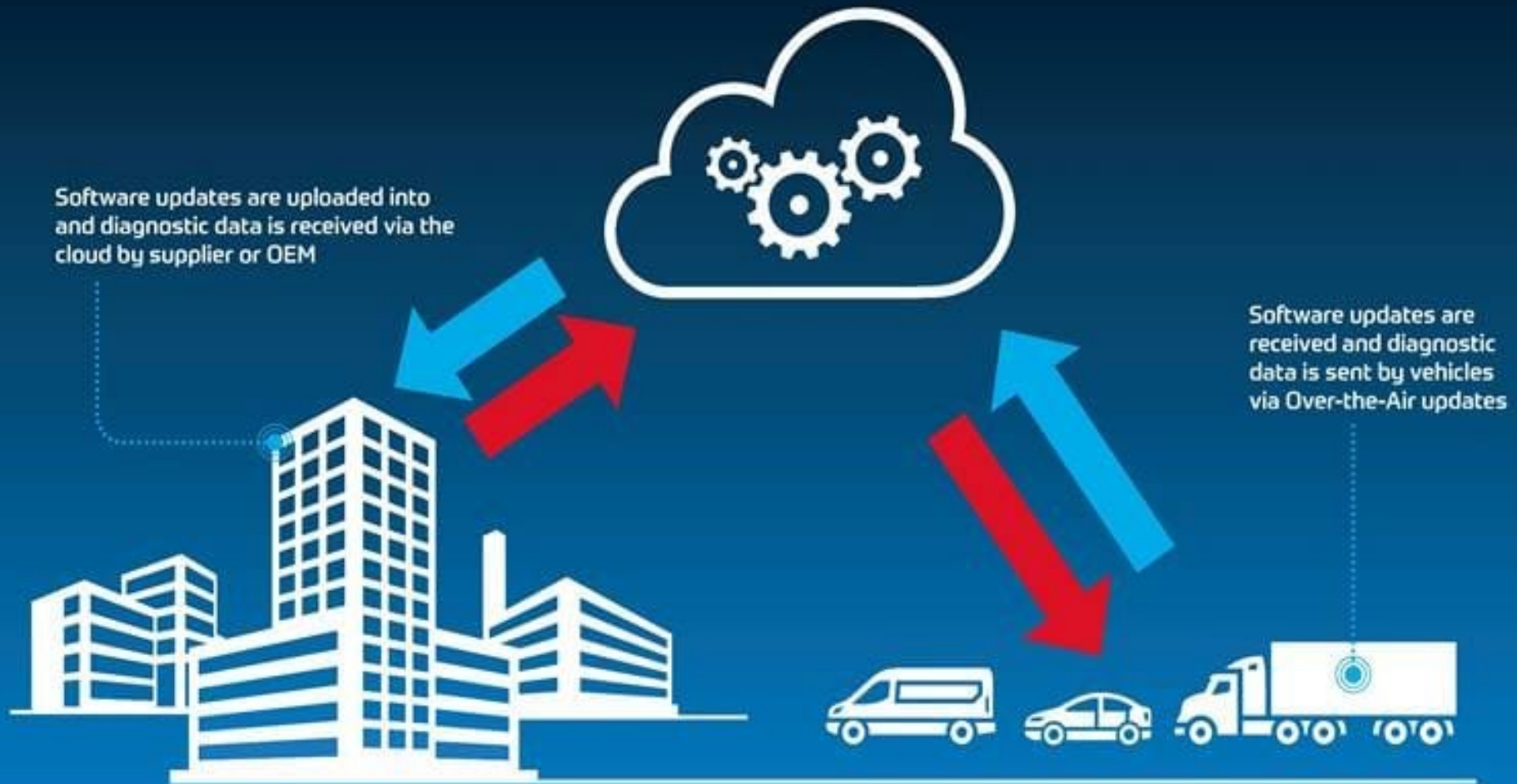
OTA updates safety

- <https://twitter.com/i/status/1360105772592467969>



University  
of Victoria

# Over-the-Air Updates



**OTA updates through satellite**

<https://www.youtube.com/watch?v=EETwoyTO7F4>



## OTA - liquid software solves safety issue

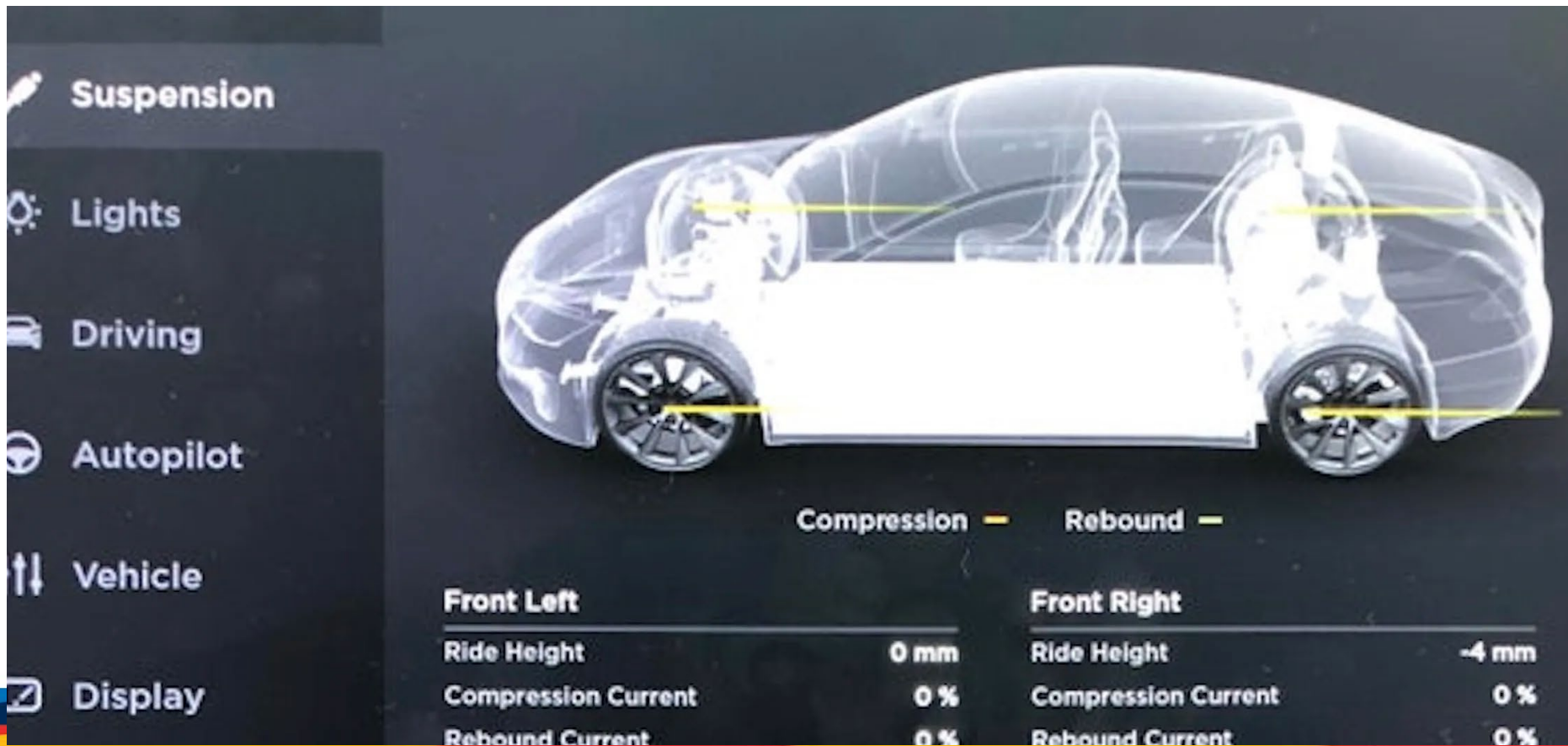
- The delivery mechanism also helped Tesla solve a **safety issue a couple of years ago.**
- Cars had experienced a series of **fires originating in the battery case**, and through the feedback system Tesla had set up, technicians determined that the fires were caused by **punctures in the cars' undercarriages.**





# OTA - liquid software solves safety issue

- Tesla delivered a **software update** that **automatically raised the chassis high enough** that road debris would no longer puncture the **battery case**.
- **Fast action**, enabled by a **tuned-up feedback system**, may have helped Tesla avoid a **major scandal or expensive physical recall**.



## Other companies like Jaguar weren't so well-prepared

- NHTSA announced that **3,083 Jaguar I-PACE** (2019 and 2020 model year) are potentially affected by a **regenerative braking system issue**. These were then **recalled**.
- According to the description, "if the electrical regenerative brake system fails, there will be **an increased delay between when the driver brakes and when the vehicle decelerates.**"
- The remedy could have been an over the air software update.



## Dog feature-A recent OTA update



## Dog feature-A recent OTA update

- This little feature, which gets activated as driver exits the car, keeps the **interior air conditioning running, maintaining an interior temperature safe for dogs.**
- At the same time, a dashboard display **alerts passers-by** that they don't need to be concerned about the animal's wellbeing.
- Now, dogs and electric cars might feel far removed from the mission-critical business, but this feature is a great example and tangible demonstration of the principle of **Continuous Integration and Continuous Delivery (CI/CD).**





# WRITING TESTS IN BDD

**“We can’t just start building features before:**

- we have actually articulated how we expect people to use them,**
- what purpose they are meant to serve, and**
- what value they are intended to help us deliver.”**

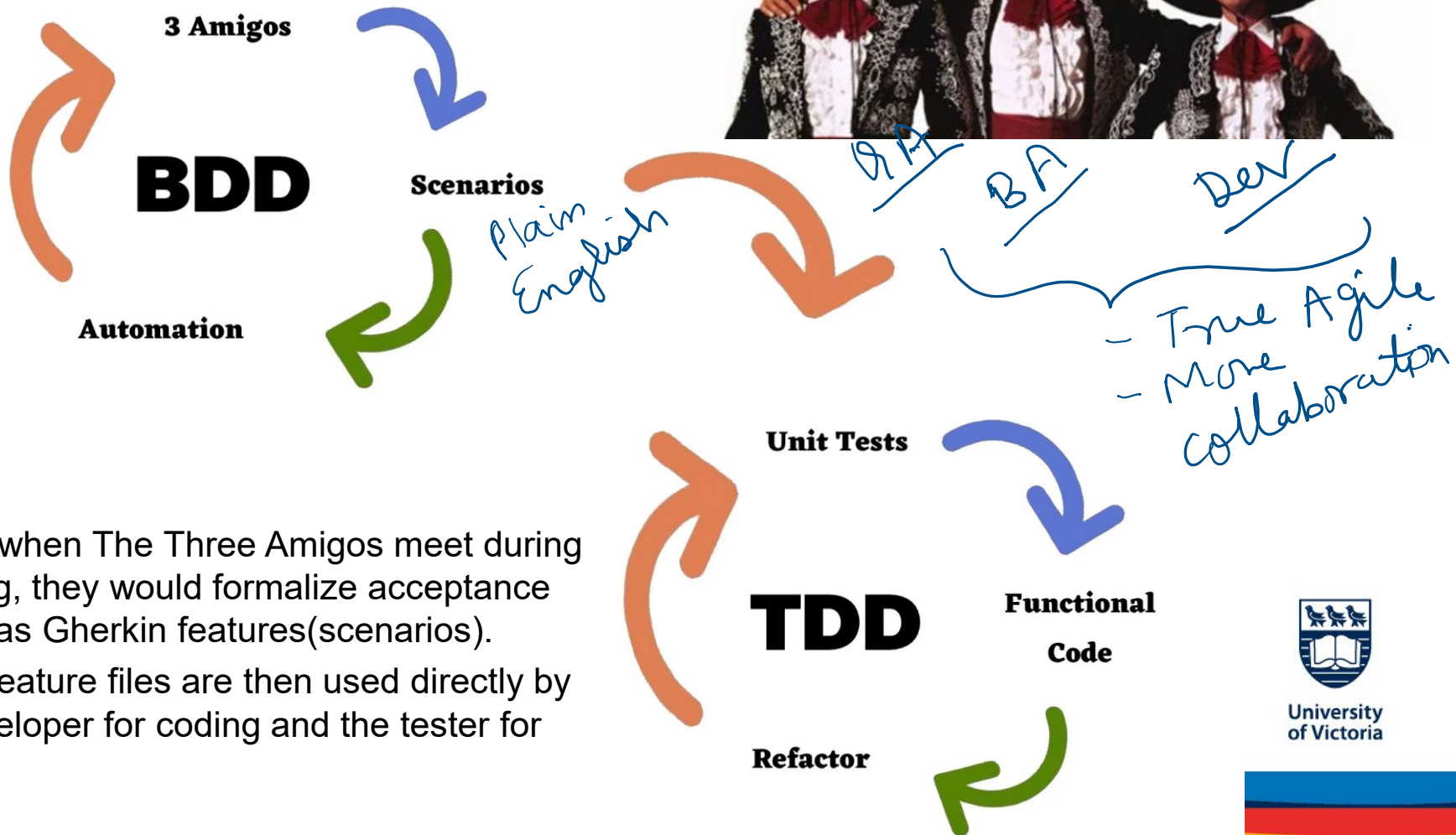


# BDD

- **Behavior Driven Development (BDD)** is a branch of Test-Driven Development (TDD).
- BDD uses **human-readable descriptions of software user requirements** as the basis for software tests.
- An early step in BDD is the definition of **a shared vocabulary between stakeholders, domain experts, and engineers**.
- This process involves the definition of entities, events, and outputs that the **users care about**, and giving them names that everybody can agree on.



# BDD



# BDD Acceptance Criteria

**Scenario:** Log in valid credentials

**Given** I have entered correct values for both username and password

UserName	Password
abc	123

**When** I click on login

**Then** User is navigated to dashboard page

**Scenario:** Log in Invalid credentials validation empty fields

**Given** I have left username or password fields empty

UserName	Password

**When** I click on login

**Then** error message should be displayed

**And** Login attempt should fail

**Scenario:** Log in Incorrect username

**Given** I have entered an invalid value for username

UserName	Password
ert	123

**When** I click on login

**Then** error message should be displayed

**And** Login attempt should fail

**Scenario:** Log in Correct username and incorrect password

**Given** I have entered correct username and incorrect password

UserName	Password
abc	456

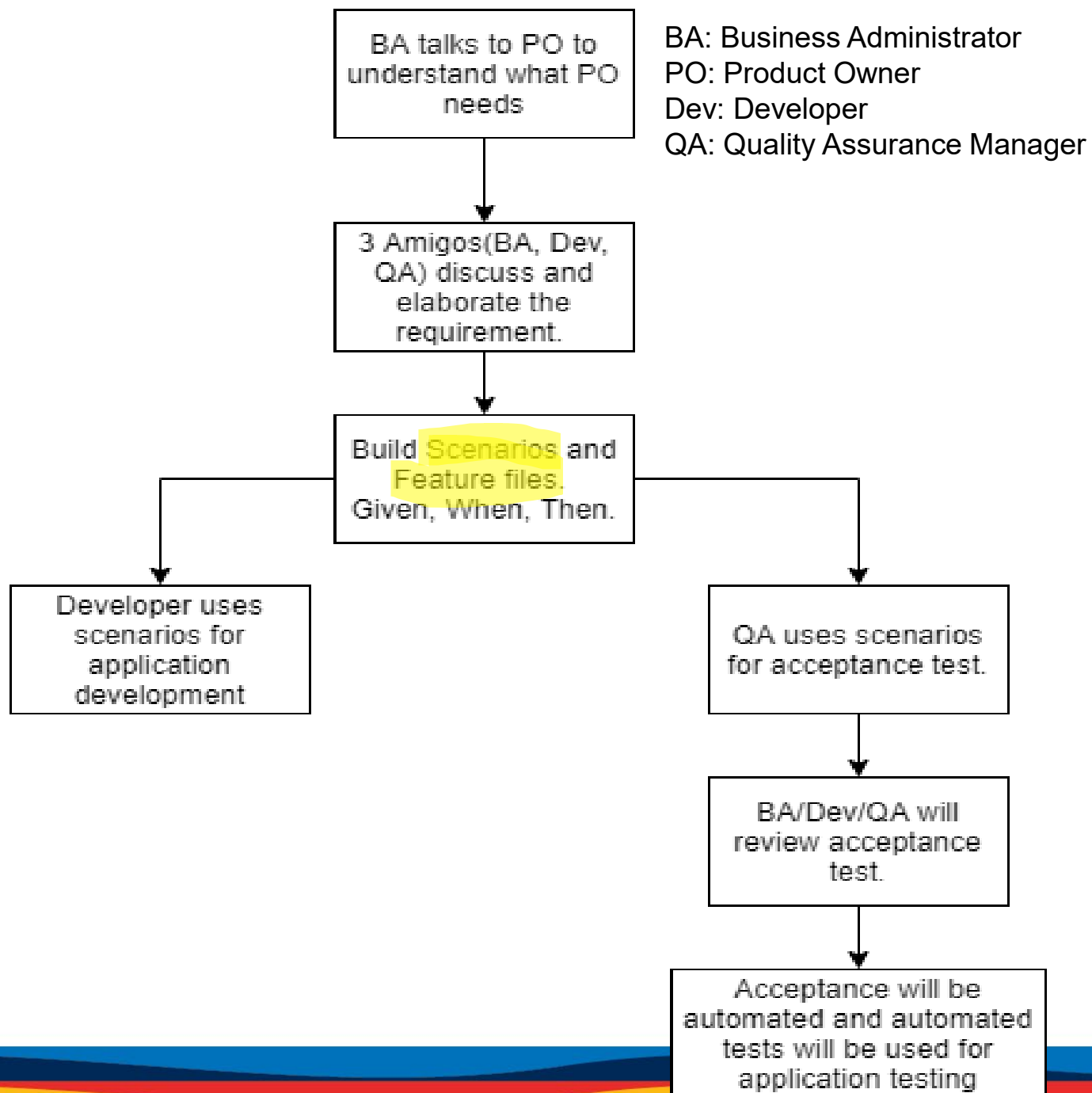
**When** I click on login

**Then** error message should be displayed

**And** Login attempt should fail



# BDD



# Gherkin

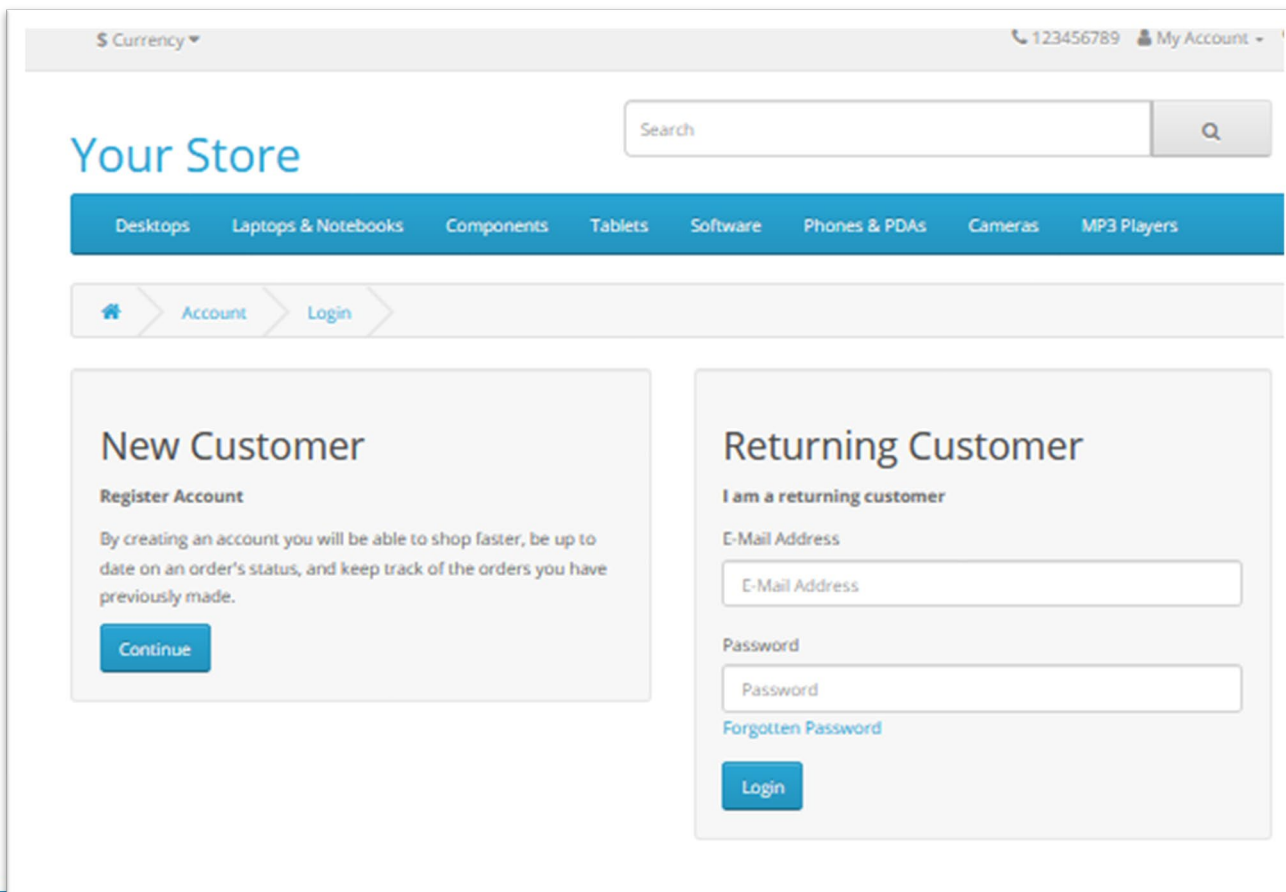
- **Gherkin** is a domain specific language that captures the conversations in a human readable format.
- We can define our acceptance criteria for a user story (the most atomic form of requirement) is Gherkin format.
- **BDD tools like Cucumber** are frameworks that parse Gherkin syntax to generate automation tests.



# Example-Writing test scenarios in BDD

**Feature:** Login

**User Story:** **As a** returning customer, **I want** to login the application, **So that** I can go to the home page



The screenshot shows a web application interface for a store. At the top, there's a header with 'Currency' and a phone number '123456789'. Below the header is a search bar. The main navigation bar lists categories: Desktops, Laptops & Notebooks, Components, Tablets, Software, Phones & PDAs, Cameras, and MP3 Players. A breadcrumb trail shows 'Home' > 'Account' > 'Login'. The page is divided into two main sections: 'New Customer' and 'Returning Customer'. The 'New Customer' section has a 'Register Account' heading and a 'Continue' button. The 'Returning Customer' section has a heading 'I am a returning customer' and fields for 'E-Mail Address' and 'Password', with a 'Login' button and a 'Forgotten Password' link.

Currency 123456789 My Account

Your Store

Search

Desktops Laptops & Notebooks Components Tablets Software Phones & PDAs Cameras MP3 Players

Home Account Login

**New Customer**

Register Account

By creating an account you will be able to shop faster, be up to date on an order's status, and keep track of the orders you have previously made.

Continue

**Returning Customer**

I am a returning customer

E-Mail Address

E-Mail Address

Password

Password

[Forgotten Password](#)

Login



University  
of Victoria

# Gherkin Test Scenario from user story

- **Feature:** Login Functionality
- **User Story:**

**As a** returning customer  
**I want** to login the application  
**So that** I can go to the home page

- **Given** : prerequisite
  - **When**: action
    - **And**: more action
  - **Then**: expected output
- } **Scenario**



# Gherkin Test Scenario from user story

- **Feature**: Login Functionality

- **User Story**:

**As a** returning customer

**I want** to login the application

**So that** I can go to the home page

- **Scenario**: Logging with **valid** credentials

**Given** user is on Login screen

**When** user enters valid username and password

**And** clicks on Login button

**Then** user is navigated to the home page.



# Gherkin Test Scenario from user story

- **Feature**: Login Functionality

- **User Story**:

**As a** returning customer

**I want** to login the application

**So that** I can go to the home page

- **Scenario**: Logging with **invalid** credentials

**Given** user is on Login screen

**When** user enters invalid username and password

**And** clicks on Login button

**Then** user is not navigated to the home page.



University  
of Victoria

# Advantages of BDD

- Easy to understand by all business stakeholders. Everybody can read and understand feature files and the resulting test cases.
- The given, when, then flow is very detailed and unambiguous.
- Easy to automate tests.
- Tests written in Cucumber directly interact with the development code.



# Here is a sample Gherkin document for withdrawal of money from account

**Feature:** Account Holder withdraws cash (Account has sufficient funds)

**User story:**

As an account holder

I want to request money

So that I am able to withdraw money

**Scenario1: The card is valid, and the machine contains enough money**

Given the account balance is \$100

And the card is valid

And the machine contains enough money

When the Account Holder requests \$20

Then the ATM should dispense \$20

And the account balance should be \$80

And the card should be returned





## Other test scenarios

**Scenario 2:** The card is valid, and the machine does not contain enough money

**Scenario 3:** The card is invalid, and the machine contains enough money.

**Scenario 4:** The card is valid, and the machine does not contain enough money



## Scenario: Signup process in Facebook

- **Given** Fred is signing up for Facebook,  
**When** he enters the required details,  
**And** submits his request,  
**Then** a Facebook account is created,  
**And** account name is set as Fred's email address,  
**And** a confirmation email is sent to Fred.



# Write test scenarios for these user stories using given, when, then format

1. As a customer of Air Canada I should be able to buy a flight ticket so that I should be able to fly to my destination.
2. As a customer of Air Canada I should be able to cancel my flight ticket so that I should be able to able to get my refund.
3. As a user of onedrive.com I should be able to upload my documents so that my team members may be able to see them.



# JPacman game

- The JPacman game is played on a rectangular board. A square on the board can be empty, or can contain the Pacman itself, one of the several ghosts, a pellet (worth 10 points), or a wall. Moveable characters such as the Pacman and the ghosts can make single-step horizontal or vertical moves. Tunnels on the border make it possible to move from one border to the opposite border. When the Pacman moves over a square containing a pellet, the player earns points and the pellet disappears. If a player and a ghost meet at the same square, the the game is over. The player wins the game once he or she has eaten all pellets.



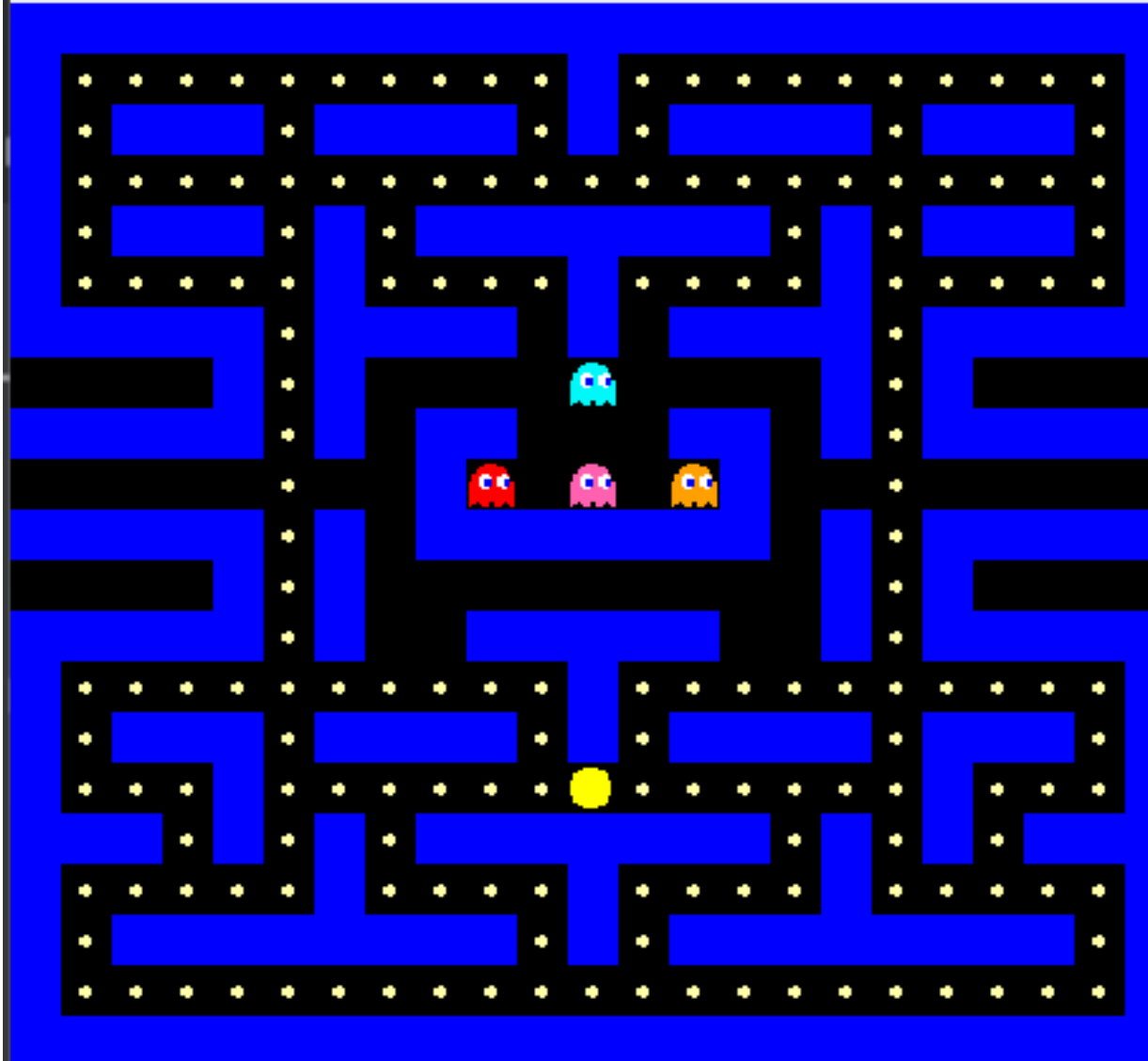


JPacman



Player 1

Score: 0



Start

Stop



University  
of Victoria

# Play the game and write stories and scenarios

- Start the game
- Move the JPacman
- Consume the pellet
- Think about 3 more scenarios.....



# Story 1 and Scenario1 : Startup

As a player

I want to start the game  
so that I can actually play

Given the user has launched the JPacman GUI;  
When the user presses the "Start" button;  
Then the game should start.



University  
of Victoria

## Story 2: Move the Player

- As a player,
- I want to move my Pacman around on the board;
- So that I can earn all points and win the game.





## Scenario S2.1: The player consumes the pellet

- Given the game has started, and my Pacman is next to a square containing a pellet;
- When I press an arrow key towards that square;
- Then my Pacman can move to that square, and I earn the points for the pellet, and the pellet disappears from that square.



## Scenario S2.2: The player moves on empty square

- Given the game has started, and my Pacman is next to an empty square;
- When I press an arrow key towards that square;
- Then my Pacman can move to that square and my points remain the same.



# Tools for BDD-creating and automating BDD user stories

- Cucumber
- JBehave
- Behat

