# Migration and Porting in Software Reengineering

- Migration and porting are common activities in software reengineering when dealing with legacy systems or when seeking to take advantage of newer technologies to improve scalability, performance, and maintainability.

# Why Migration and Porting

- The motivation behind migration and porting a legacy software is improving performance, scalability, maintainability, or addressing end-of-life technology issues.

# *What is Software Migration and Porting*

- Software migration and porting are processes in software reengineering that involve transferring a software system from one environment to another.

- These terms are often used interchangeably, but there are subtle differences:

# *Do Migration & Porting have the same meaning*

- **Migration:** In the context of software, migration involves moving a system or application from one environment or platform to another.

- This process includes not only the transfer of the software but also the associated data, configurations, and potentially changes to the overall architecture.

# *Do Migration & Porting have the same meaning?*

- **Porting:** Porting specifically refers to the process of adapting the software so that it can run on a different platform or environment. This may involve making changes to the code, addressing platform-specific dependencies, and ensuring compatibility.

- Porting is a subset of the broader migration process. It focuses on the software aspect of the migration, ensuring that the code is adjusted to work efficiently on the new platform.

- Addressing Platform-Specific Dependencies (If a game developed for a PlayStation console is being ported to run on a PC, the code might need to be adjusted to use PC-compatible graphics libraries and input devices instead of the PlayStation-specific ones.)

- Ensuring Compatibility (Platforms can vary in terms of hardware architecture (intel to AMD), operating system features, or even processor instruction sets. Porting aims to ensure that the software is compatible with these platform differences.)

# Types of Migrations

- **Platform Migration**: Moving from one hardware or operating system platform to another.

✔ Windows to Linux Migration

✔ Desktop Application to Web Application Migration

# Types of Migrations

- **Language Migration:** Transitioning from one programming language to another.

✔ Java to Kotlin Migration

✔ Python 2 to Python 3 Migration

✔ PHP to Node.js Migration

# Types of Migrations

- **Architecture Migration:** Shifting from one architectural style to another, e.g., monolithic to microservices.

✔ Monolithic to Microservices Migration

✔ Client-Server to Serverless Migration

✔ Single-Tier to Three-Tier Migration (UI, Data and Logic layers)

# Migration Process

- Analysis and Assessment: Understanding the existing system, its dependencies, and challenges associated with migration.
- Planning and Design: Defining the target architecture, setting migration goals, and creating a migration plan.

# Migration Process

- Implementation: Executing the migration plan, including code modifications, data migration, and integration with the new environment.

- Testing and Validation: Ensuring the migrated software functions correctly and meets the desired performance and quality standards.

- Deployment and Post-Migration Support: Rollout of the migrated system and providing ongoing support.

- **Example: Platform Migration - On-Premises to Cloud**

- **Scenario:** An on-premises web application hosted in a local data center.

- **Motivation:** Migrating to the cloud to benefit from scalability, elasticity, and reduced infrastructure maintenance.

- **Approach:**

- Choose a cloud provider and set up the necessary infrastructure.
- Modify the application to be cloud-compatible (e.g., configuring load balancers, auto-scaling, and cloud storage).
- Migrate data and ensure seamless connectivity between the application and cloud services.

- **Benefits:** Scalability, reduced infrastructure costs, and improved disaster recovery options.

- **Example: Language Migration - Python 2 to Python 3**

- **Scenario:** A legacy Python 2 application.

- **Motivation:** Python 2 reached its end-of-life, and Python 3 offers improved features and performance.

- **Approach:**

- Identify Python 2 specific code and libraries.
- Update the code to be compatible with Python 3, addressing language syntax differences and library changes.
- Test the application thoroughly to ensure functionality is maintained after migration.

- **Benefits:** Access to newer Python features, security patches, and community support.

- **Example: Monolithic to Microservices Migration**

- **Scenario:** A large monolithic e-commerce application.

- **Motivation:** The monolithic architecture hinders/reduces scalability and deployment flexibility.

- **Approach:**
- **(1) Identify distinct functionalities within the monolith.**

✔ In a monolithic application, different business functionalities are tightly coupled within the same codebase. The first step in the migration process is to identify these distinct functionalities and determine which ones can be separated into individual microservices.

✔ For example, consider an e-commerce monolithic application that handles user authentication, product catalog, order processing, and payment processing. These are distinct functionalities that can be separated into microservices.

## • (2) Create Mini-Websites for Each Job/task:

✔ Now that we know the different jobs, we create small, separate parts (microservices) for each job.

✔ Instead of one giant website, we now have small websites that each do one thing really well.

✔ So, we create a mini-website for user logins, one for showing products, another for processing orders, and one more for dealing with payments.

## (3) Let Them Talk to Each Other:

✔ But now, these mini-websites need to talk to each other. They do this by having special ways (APIs) to communicate.

✔ It's like each mini-website has its own phone number, and they can call each other when they need something.

✔ For example, when the orders website needs to know about a product, it calls the products website using its special phone number (API).

- **Result**

- Now, we have a bunch of small websites (microservices) that work together like a team.

- Each one does its own job well, and they can be updated or improved without messing up the whole website. This makes the website easier to manage, faster, and more flexible when we want to change or add something.

```
project-root/
|-- login-microservice/
|   |-- src/
|   |   |-- main/
|   |       |-- java/
|   |           |-- Login.java
|   |
|   |-- pom.xml (if using Maven for dependencies)
|
|-- payment-microservice/
|   |-- src/
|   |   |-- main/
|   |       |-- java/
|   |           |-- Payment.java
|   |
|   |-- pom.xml
|
|-- other-microservices/
|   |-- ...
```

Microservices are often developed as separate projects, each with its own codebase, dependencies, and potentially even its own deployment pipeline.
Each microservice is independently developed, tested, and deployed.
This independence is a key characteristic of microservices architecture.

- **Benefits:** Improved scalability, independent deployment of services, and better fault isolation.

# Challenges in Migration and Porting

- Compatibility issues between the old and new platforms, languages, or architectures.
- Data migration and transformation challenges.
- Performance differences between the old and new environments.
- Testing complexities during migration.
- Risk management and handling unforeseen issues.

# Assignment

- Find out how we can identify the code complexity (cyclometic complexity) in popular IDE's like Visual Studio, Eclipse etc. Find if we need some special plug-ins for that..

- **Thank You**