Integration

Introduction

What

★☆☆☆ 🏆 Can explain integration

Combining parts of a software product to form a whole is called *integration*. It is also one of the most troublesome tasks and it rarely goes smoothly.

Approaches

'Late and One Time' vs 'Early and Frequent'

★★☆☆ 🏆 Can explain how integration approaches vary based on timing and frequency

In terms of timing and frequency, there are two general approaches to integration: *late and one-time*, *early and frequent*.

***Late and one-time*: wait till all components are completed and integrate all finished components near the end of the project.**

✖ This approach is not recommended because integration often causes many component incompatibilities (due to previous miscommunications and misunderstandings) to surface which can lead to delivery delays i.e. Late integration → incompatibilities found → major rework required → cannot meet the delivery date.

***Early and frequent*: integrate early and evolve each part in parallel, in small steps, re-integrating frequently.**

📦 A *walking skeleton* can be written first. This can be done by one developer, possibly the one in charge of integration. After that, all developers can flesh out the skeleton in parallel, adding one feature at a time. After each feature is done, simply integrate the new code into the main system.

Here is an animation that compares the two approaches:

Big-Bang vs Incremental Integration

★★★☆ 🏆 Can explain how integration approaches vary based on amount merged at a time

***Big-bang integration*: integrate all components at the same time.**

✖ Big-bang is not recommended because it will uncover too many problems at the same time which could make debugging and bug-fixing more complex than when problems are uncovered incrementally.

***Incremental integration*: integrate a few components at a time.** This approach is better than big-bang integration because it surfaces integration problems in a more manageable way.

Here is an animation that compares the two approaches:

Exercises

Top-Down vs Bottom-Up Integration

★★★★ Can explain how integration approaches vary based on the order of integration

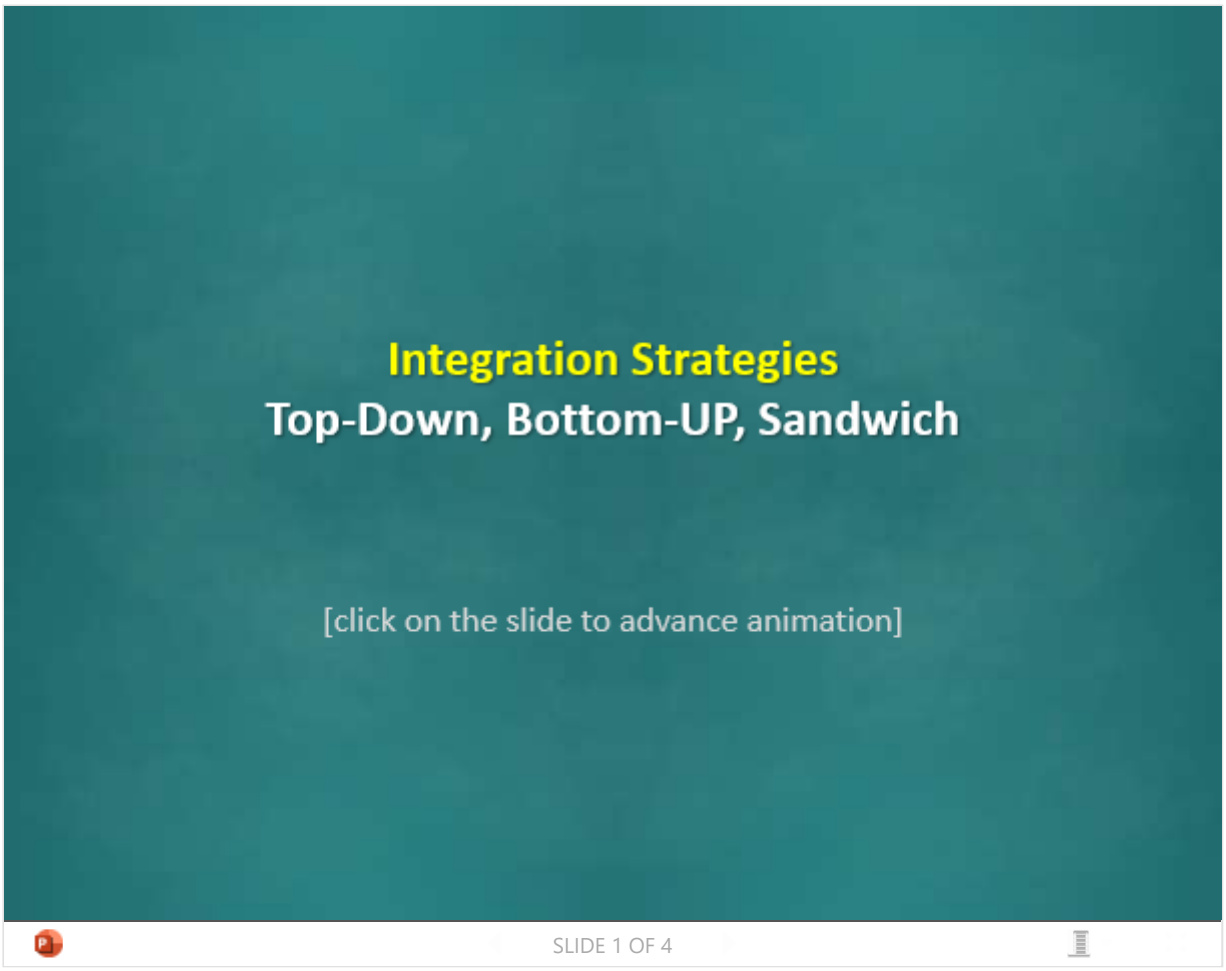
Based on the order in which components are integrated, incremental integration can be done in three ways.

***Top-down* integration: higher-level components are integrated before bringing in the lower-level components.** One advantage of this approach is that higher-level problems can be discovered early. One disadvantage is that this requires the use of stubs in place of lower level components until the real lower-level components are integrated into the system. Otherwise, higher-level components cannot function as they depend on lower level ones.

***Bottom-up* integration: the reverse of top-down integration.** Note that when integrating lower level components, drivers may be needed to test the integrated components because the UI may not be integrated yet, just like how top-down integration needs stubs.

***Sandwich* integration: a mix of the top-down and bottom-up approaches.** The idea is to do both top-down and bottom-up so as to 'meet' in the middle.

Here is an animation that compares the three approaches:



Exercises

Build Automation

What

★☆☆☆ Can explain build automation tools

Build automation tools automate the steps of the build process, usually by means of build scripts.

In a non-trivial project, building a product from its source code can be a complex multi-step process. For example, it can include steps such as: pull code from the revision control system, compile, link, run automated tests, automatically update release documents (e.g. build number), package into a distributable, push to repo, deploy to a server, delete temporary files created during building/testing, email developers of the new build, and so on. Furthermore, this build process can be done 'on demand', it can be scheduled (e.g. every day at midnight) or it can be triggered by various events (e.g. triggered by a code push to the revision control system).

Some of these build steps such as compiling, linking and packaging, are already automated in most modern IDEs. For example, several steps happen automatically when the 'build' button of the IDE is clicked. Some IDEs even allow customization of this build process to some extent.

However, most big projects use specialized build tools to automate complex build processes.

Some popular build tools relevant to Java developers: Gradle, Maven, Apache Ant, GNU Make

Some other build tools: Grunt (JavaScript), Rake (Ruby)

Some build tools also serve as *dependency management tools*. Modern software projects often depend on third party libraries that evolve constantly. That means developers need to download the correct version of the required libraries and update them regularly. Therefore, dependency management is an important part of build automation. Dependency management tools can automate that aspect of a project.

Maven and Gradle, in addition to managing the build process, can play the role of dependency management tools too.

Resources

- **Getting Started with Gradle** -- documentation from Gradle
- **Gradle Tutorial** -- from tutorialspoint.com

Working With Gradle in Intellij IDEA (6 minutes)

Exercises

Continuous Integration and Continuous Deployment

★★☆☆ Can explain continuous integration and continuous deployment

An extreme application of build automation is called *continuous integration (CI)* in which integration, building, and testing happens automatically after each code change.

A natural extension of CI is *Continuous Deployment (CD)* where the changes are not only integrated continuously, but also deployed to end-users at the same time.

Some examples of CI/CD tools: Travis, Jenkins, Appveyor, CircleCI, GitHub Actions