[L7P3]

How to Avoid a Big Bang: Integrating Software Components

Integration

Timing and frequency: 'Late and one time' vs 'early and frequent'

Integrating parts written by different team members is inevitable in multi-person projects. It is also one of the most troublesome tasks and it rarely goes smoothly.

In terms of timing and frequency, there are two general approaches to integration:

1. *Late and one-time*: In an extreme case of this approach, developers wait till all components are completed and integrate all finished components just before the release. 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: Late integration → incompatibilities found → major rework required → cannot meet the delivery date.

2. *Early and frequent*: The other approach is to integrate early and evolve in parallel in small steps, re-integrating frequently. For example, a working skeleton[9] can be written first (i.e. it compiles and runs but does not produce any useful output). 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 to the main system.

Whether using frequent integration or one-time late integration, there is still a need to decide the order in which components are to be integrated. There are several approaches to doing this, as explained next.

The order of integration: Big bang vs incremental

Big-bang integration

In the *big-bang integration* approach, all components are integrated at the same time. This approach is not recommended since it will uncover too many problems at the same time which could make debugging and bug-fixing more complex than when problems are uncovered more gradually.

Incremental integration

For non-trivial integration efforts, the following three incremental integration approaches are more suitable.

- *Top-down* integration: In *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 dummy or skeletal components (i.e. stubs) in place of lower level components until the real lower-level components are integrated to the system. Otherwise, higher-level components cannot function as they depend on lower level ones.

[9] Some call it a '*walking* skeleton'

- *Bottom-up* integration: This is the reverse of top-down integration. Advantages and disadvantages are simply the reverse of those of the top-down approach.
- *Sandwich* integration: This is a mix of the top-down and the bottom-up approaches. The idea is to do both top-down and bottom-up so as to "meet up" in the middle.

Build automation

In a non-trivial project, building a product from source code can be a complex multi-step process. For example, it can include steps such as to 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 to compile, link and package 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 to this build process to some extent.

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

GNU *Make* (http://www.gnu.org/software/make/) and ***Apache Ant*** (http://ant.apache.org/) are two build tools that used to be very popular about a decade ago and still being used. Two popular build tools at the moment are **Maven** (http://maven.apache.org/) and **Gradle** (https://gradle.org/)

Dependency Management

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. Dependency Management tools can automate that aspects of a project. Maven and Gradle, in addition to managing the build process, are dependency management tools too.

Continuous Integration

An extreme application of build automation is called *continuous integration* (CI) in which integration, building, and testing happens automatically after each code change. **Travis** (https://travis-ci.org/) and **Jenkins** (http://jenkins-ci.org) are examples of popular CI tools.

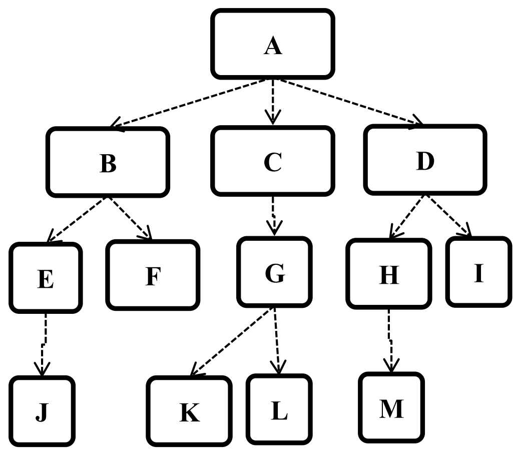
Worked examples

[Q1]

Consider the architecture given below. Describe the order in which components will be integrated with one another if the following integration strategies were adopted.

(a) big-bang
(b) top-down
(c) bottom-up
(d) sandwich

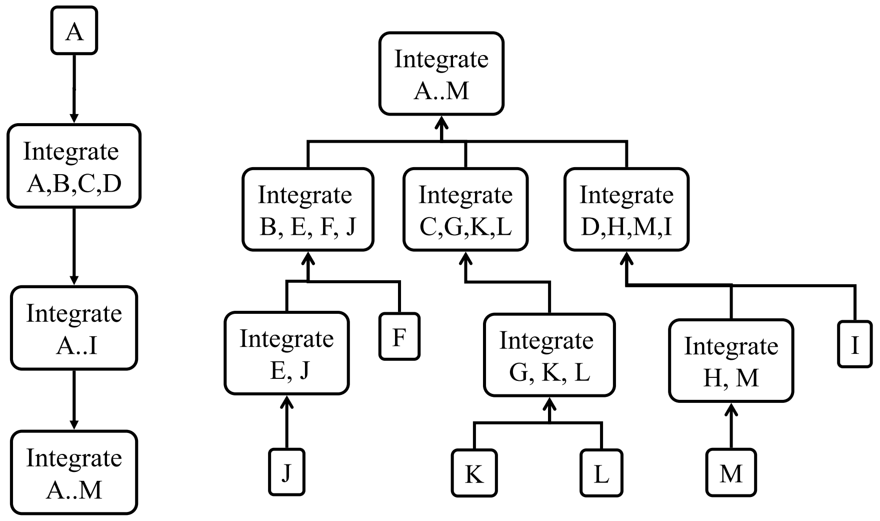
Note that dashed arrows show dependencies (e.g. A depend on B, C, D and therefore, higher-level than B, C and D).



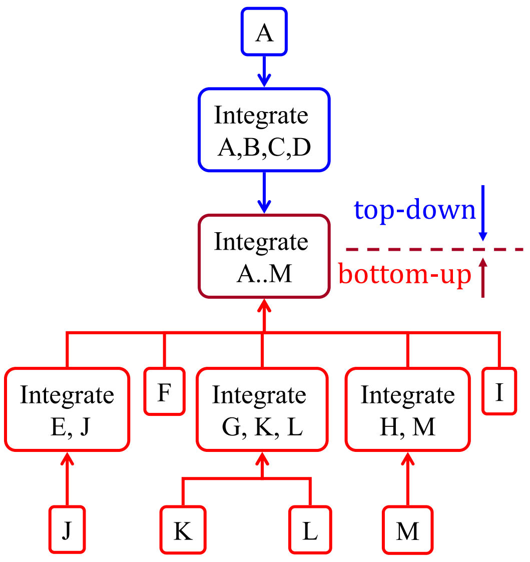
[A1]

(a) Big-bang approach: integrate A-M in one shot.

(b) Top-down approach and (c) bottom-up approach [side by side comparison]



(d) Sandwich approach



[Q2]
Give two arguments in support and two arguments against the following statement.

Because there is no external client, it is OK to use big bang integration for the CS2103 module project.

[A2]
Arguments for:

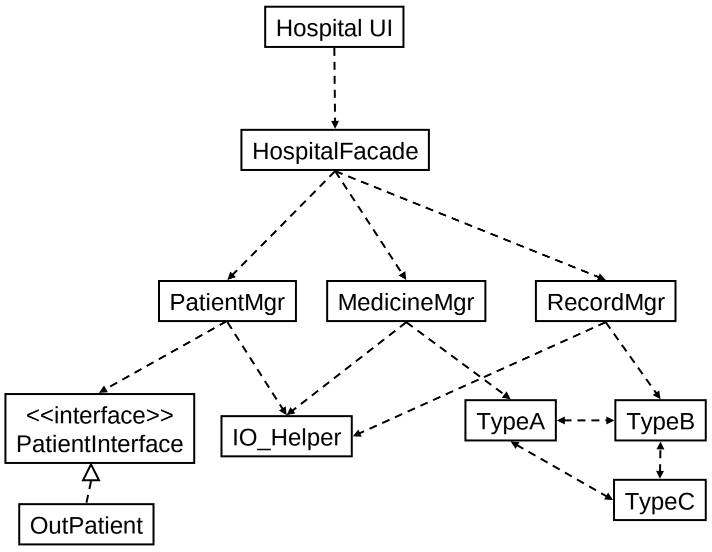
- It is relatively simple; even big-bang can succeed.
- Project duration is short; there is not enough time to integrate in steps.
- The system is non-critical, non-production (demo only); the cost of integration issues is relatively small.

Arguments against:

- Inexperienced developers; big-bang more likely to fail
- Too many problems may be discovered too late. Submission deadline (fixed) can be missed.
- Team members have not worked together before; increases the probability of integration problems.

[Q3]
Suggest an integration strategy for the system represented by following diagram. You need not follow a strict top-down, bottom-up, sandwich, or big bang approach. Dashed arrows represent dependencies between classes.
Also take into account the following facts in your test strategy.

- **HospitalUI** will be developed early, so as to get customer feedback early.
- **HospitalFacade** shields the UI from complexities of the application layer. It simply redirects the method calls received to the appropriate classes below
- **IO_Helper** is to be reused from an earlier project, with minor modifications
- Development of **OutPatient** component has been outsourced, and the delivery is not expected until the 2nd half of the project.

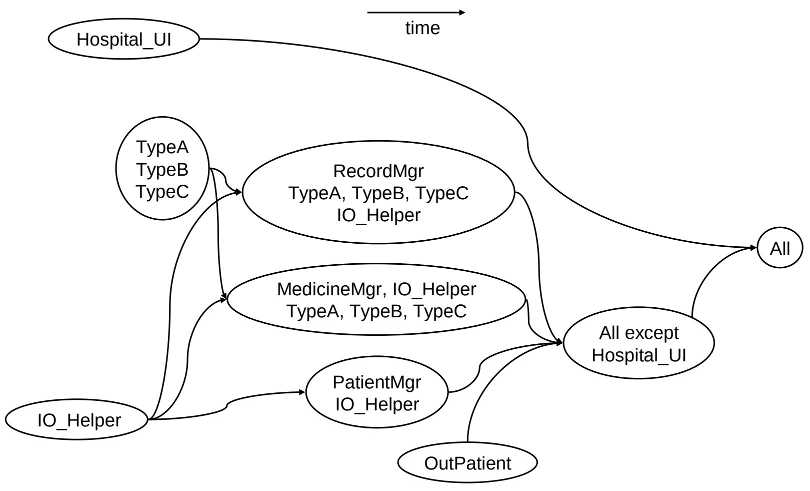


[A3]
There can be many acceptable answers to this question. But any good strategy should consider at least some of the below.

- Since **HospitalUI** will be developed early, it's OK to integrate it early, using stubs, rather than wait for the rest of the system to finish. (i.e. a top-down integration is suitable for **HospitalUI**)

- Because **HospitalFacade** is unlikely to have a lot of business logic, it may not be worth to write stubs to test it (i.e. a bottom-up integration is better for **HospitalFacade**).
- Since **IO_Helper** is to be reused from an earlier project, we can finish it early. This is especially suitable since there are many classes that use it. Therefore **IO_Helper** can be integrated with the dependent classes in bottom-up fashion.
- Since **OutPatient** class may be delayed, we may have to integrate **PatientMgr** using a stub.
- **TypeA**, **TypeB**, and **TypeC** seem to be tightly coupled. It may be a good idea to test them together.

Given below is one possible integration test strategy. Relative positioning also indicates a rough timeline.



-- End of handout --