[Handout for L5P3]
How to Avoid a Big Bang: Integrating Software Components

Integration

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

Integrating parts written by different team members is inevitable in multi-person projects. It is also one of the most troublesome tasks and 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 continuous*: The other approach is to integrate early and evolve in parallel in small steps, re-integrating frequently. For example, we can write a working skeleton[3] first (i.e. it compiles and runs but does not produce any useful output). This can be done by one developer, 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, we can integrate the new code to the main system. This approach is called *continuous integration*. There are build tools e.g. CruiseControl (http://cruisecontrol.sourceforge.net/) that specifically cater for continuous integration.

Whether we use continuous integration or we wait till the end to integrate, we still have to decide the order in which we integrate components. 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, we integrate all components at the same time. This approach is not recommended since it will surface too many problems at the same time which could make debugging and bug-fixing more complex than necessary. The other three more 'incremental' approaches explained next are more suitable for non-trivial integration efforts.

Top-down integration

In *top-down integration*, we integrate higher-level components before we bring in the lower-level components. One advantage of this approach is that we can discover higher-level problems early. One disadvantage is that this requires us to use 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.

[3] 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 that we meet in the middle.

Revision control (team)

In a previous handout we briefly mentioned the use of Revision control systems (RCS) for managing revisions of your own files. RCS are even more useful to manage revisions when you are working as a team[4].

When using an RCS in a team setting, in addition to the local repository on your machine, you also need a remote repository ('remote repo' for short) that other team members can access remotely.

There are two models we can follow when using an RCS in a team setting: the centralized model and the decentralized model.

1. **Centralized RCS** (CRCS for short): In this model, there is a central remote repo shared by the team. Team members download ('pull') and upload ('push') changes between their own local repositories and the central repository. A team member's local repository cannot share changes directly with other members' local repositories. This is illustrated in Figure 16. Older RCS tools such as CVS and SVN support only this model. Furthermore, they do not support the notion of a local repo. Instead, they force users to do all the versioning with the remote repo. This situation is illustrated in Figure 17.

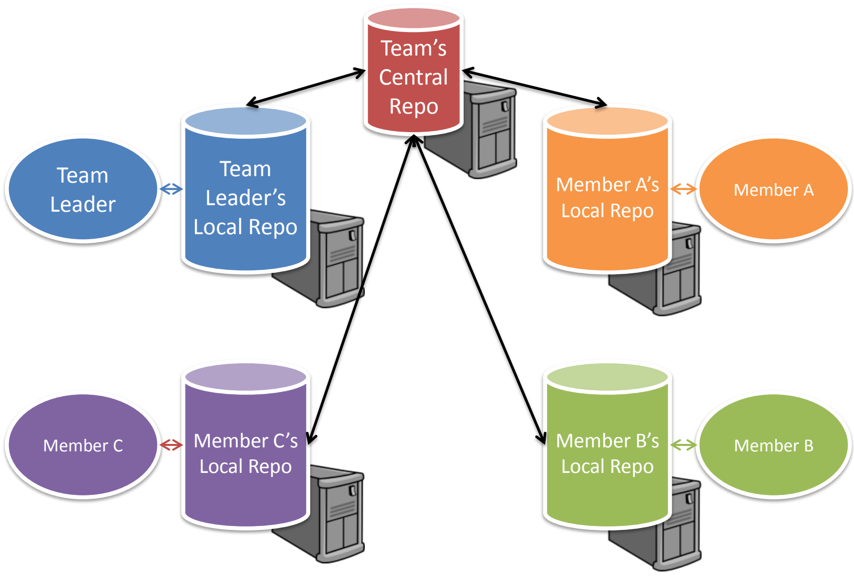


Figure 16. The centralized RCS approach using both local and remote repos

[4] Some parts of this section, including the three diagrams, were adapted from tutor Steve Teo's work.

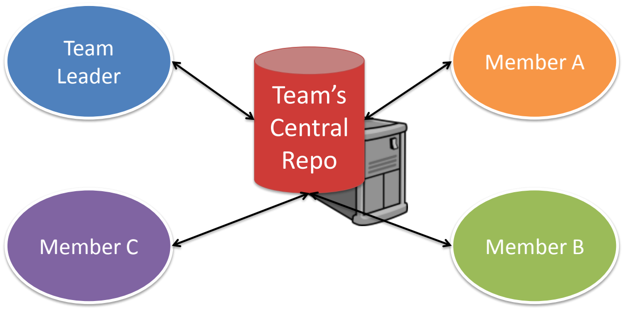


Figure 17. The centralized RCS approach without any local repos (e.g., CVS, SVN)

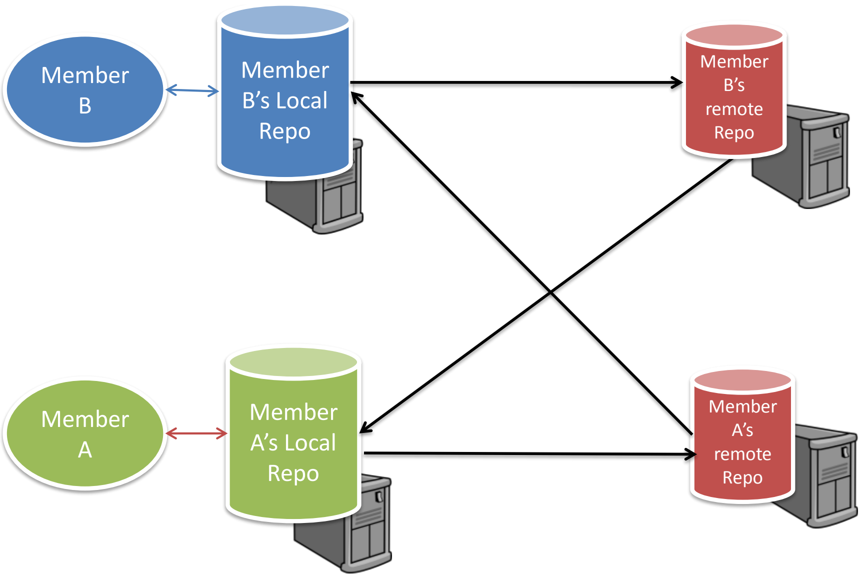


Figure 18. The decentralized RCS approach

2. **Distributed RCS** (DRCS for short, also known as *Decentralized* RCS): In this model, there can be multiple remote repos and pulling and pushing can be done among them in arbitrary ways. Hence the workflow can vary differently from team to team. For example, every team member can have his/her own remote repository in addition to their own local repository, as shown in Figure 18. Mercurial, Git, and Bazzar are some prominent RCS tools that support distributed RCS.

Branching and merging are two terms that are often used in team-based RCS usage. Conceptually, branching is the process of evolving multiple versions of the software in parallel. For example, one team member can create a new branch and add an experimental feature to it while the rest of the team keep working on the *trunk*. 'Trunk' refers to the main line of development while 'branches' are other variants of the software being developed in parallel. A branch can contain many revisions. Once the experimental feature is stable, that branch can be *merged* to the main trunk. Branching and merging is illustrated in Figure 19.

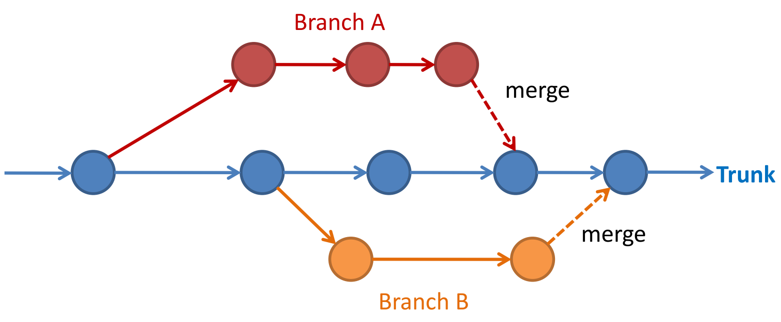


Figure 19. Branching and merging

When using the DRCS approach, there are multiple repositories (local and remote) each one containing different parallel versions. They are also called *clones*. DRCS users have to merge quite often, whenever they pull a change set from another repo.

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 check out code from the revision control system, compile, link, run automated tests, automatically update release documents (e.g. build number), package into a distributable, upload 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. everyday at midnight) or it can be triggered by various events (e.g. triggered by a code check-in to the revision control system).

Some of these build steps such as compile, link and package are already automated in most modern IDEs. For example, when you press the 'build' button in your IDE, several steps happen automatically. Some IDEs even allow you to customize this build process into some extent.

However, most big projects use specialized build tools to automate complex build processes. GNU *Make* (http://www.gnu.org/software/make/) is one such powerful build tool that has been used since long ago and still popular among practitioners. To quote its home page,

> *Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. Make gets its knowledge of how to build your program from a file called the makefile, which lists each of the non-source files and how to compute it from other files. When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program.*

As you can see from the above description, all build instructions are written in a script file called *makefile*. Such a makefile is especially useful when you distribute your software as source code for expect users to build the product by themselves.

Apache Ant (http://ant.apache.org/) is a similar build tool popular in the Java world. Makefiles uses its own syntax while Ant 'buildfiles' are formatted as xml.

Sample Makefile (extract)

```
edit : main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
             cc -o edit main.o kbd.o command.o display.o \
                        insert.o search.o files.o utils.o

     main.o : main.c defs.h
             cc -c main.c
     kbd.o : kbd.c defs.h command.h
             cc -c kbd.c
     command.o : command.c defs.h command.h
             cc -c command.c
     display.o : display.c defs.h buffer.h
             cc -c display.c
     insert.o : insert.c defs.h buffer.h
             cc -c insert.c
     clean :
             rm edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o
```

Sample Ant file (extract)

```
<target name="myTarget" depends="myTarget.check"
if="myTarget.run">
    <echo>Files foo.txt and bar.txt are present.</echo>
</target>

<target name="myTarget.check">
    <condition property="myTarget.run">
        <and>
            <available file="foo.txt"/>
            <available file="bar.txt"/>
        </and>
    </condition>
</target>
```

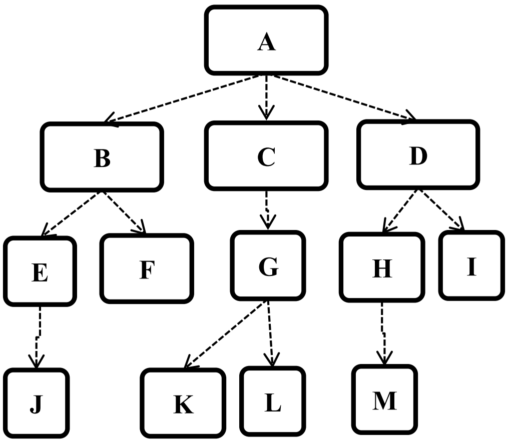
Worked examples

[Q1]

Consider the architecture given below. Give which components will be integrated with other components in which order if we were using the following integration strategies.

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

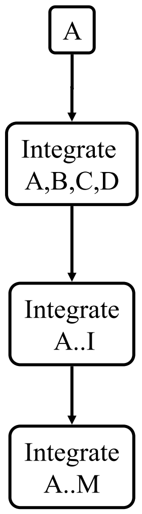
Note that dashed arrows show dependencies (e.g. A depend on B, C, D and therefore, higher-level than B, C and D).

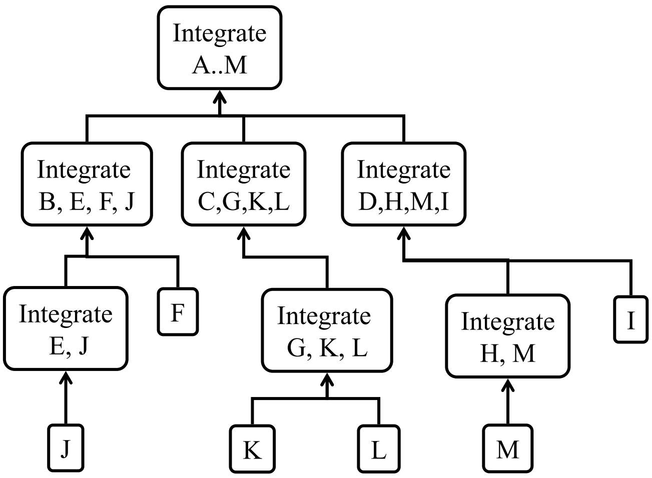


[A1]

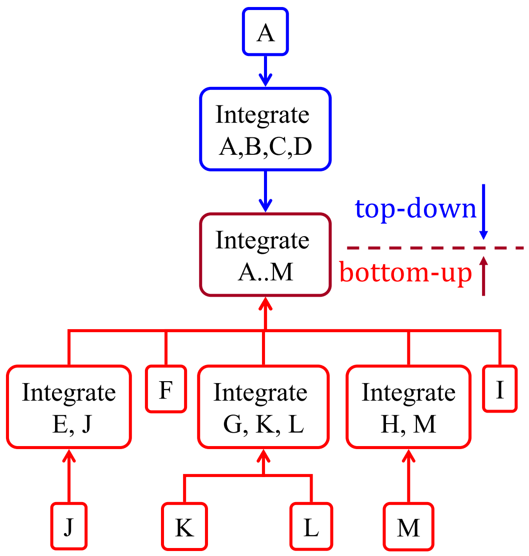
(a) Big-bang approach: integrate A-M in one shot.

(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.

Assuming there is no external client, it is OK to use big bang integration for CS2103 course project

[A2]
Arguments for:

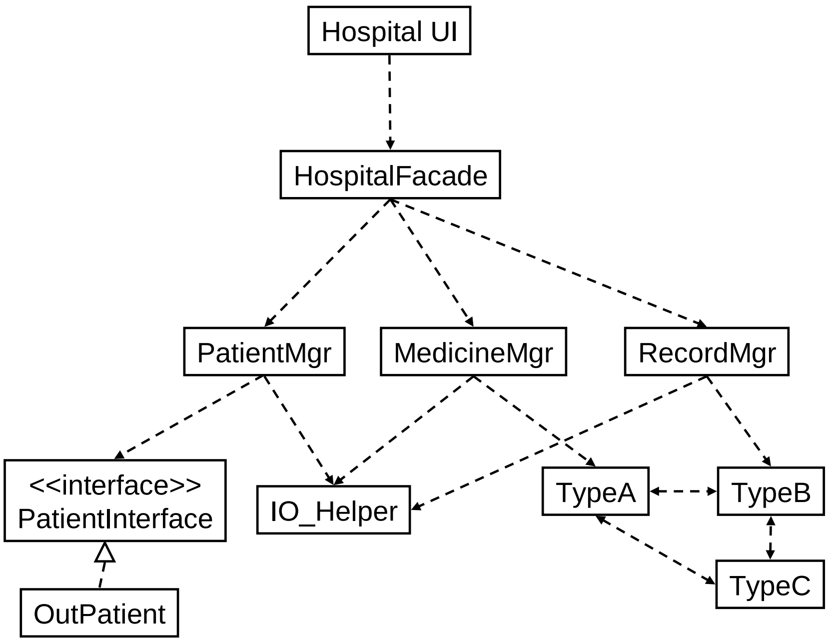
- It is relatively simple; even big-bang can succeed.
- Project duration is short; there is not enough time to integrate in steps.
- The system is non-critical, non-production (demo only); the cost of integration issues is relatively small.

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.

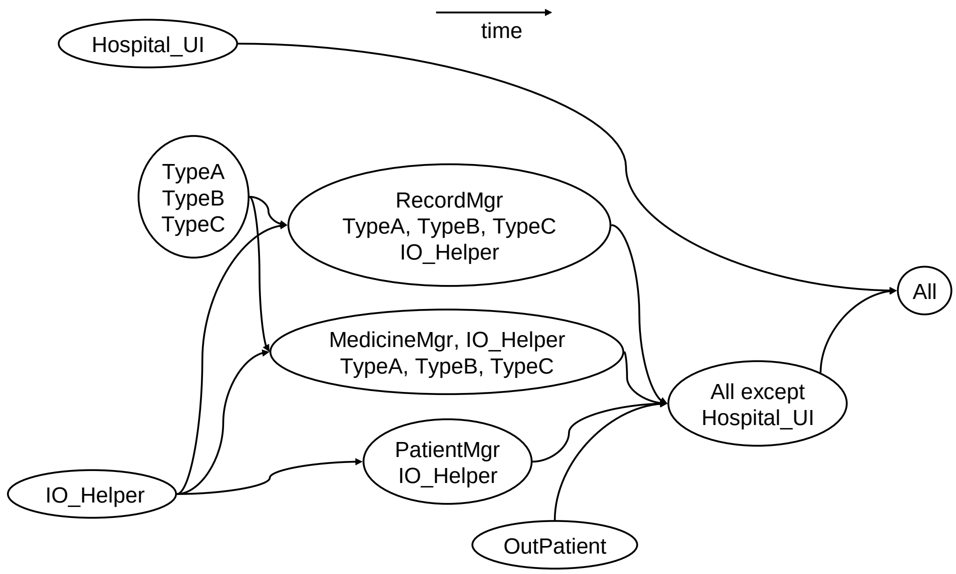


[A3]

There can be many 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 lot of business logic. Instead of using a stub to integrate HospitalUI, we can use Therefore 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 Document---