Software Engineering Concepts (COMP3003)

1. Architecture

In the distant reaches of the first year of your computing degree, you learnt about **Non-Functional Requirements (NFRs)**. And you probably hated them, because, even if you could see their theoretical value, what exactly are you supposed to you *do* with them? How do you take one and *implement* it?

Functional requirements are easy, by contrast. You think of them as menu options (or at least options that exist *somewhere*, if not in a menu) that can simply be chosen or activated at an appropriate time. You can code, test and debug them one-by-one.

But you cannot simply sit down at your PC and code up an NFR. NFRs need to be built-in to everything else. They are "cross-cutting concerns". But there must be something specific we can do to implement them, or else what's the point?

In large part, NFRs are implemented through **software architecture** – the very high-level design choices we need to make right at the start of a project. Architecture is decided early because it underpins everything else. For that reason, it's also very difficult (i.e. expensive) to change afterwards.

When developing software architecture, we seek to answer questions like this:

- What operating systems (or virtual machines) will it run on?
- Will it be web-based?
- Will it be distributed (having multiple parts that communicate over a network)?
- Will it be separated into multiple processes running on the same machine?
- Will it be multithreaded?
- Will it have a plugin system, or scripting capabilities?
- What framework (if any) will it use?
- What other software will it require to run? (Which database system? Which authentication system? Which payment system?)

A lot of these questions *could* be answered in the requirements document. Even if they are not, the client should be aware of (and comfortable with) architectural decisions, because they have significant consequences!

1 Platforms and Portability

As users, we tend to expect (or at least hope) that our favourite software will run on whatever device we have. As software developers, we would (if needed) have a non-functional requirement for this:

"The system must run on Android 6.0 and Windows 10."

We call such software **cross-platform** (or **platform-independent**). A **platform** can represent a few different concepts: the operating system, the CPU instruction set, the set of APIs you're using, or all of the above. A platform is everything that software developers assume to already exist on the user's machine.

The main challenge to creating cross-platform software is the difference in APIs. Each operating system – Windows, MacOS, Linux, Android, iOS and others – comes bundled with different sets of libraries, with different interfaces. The same problem arises for different *versions* of OSs; for instance, what works on Windows 10 may not necessarily work on Windows 7. Linux, moreover, can be customised in so many different ways that it's difficult to define precisely what it even is[a].

Operating systems are not the only things we refer to as "platforms" though. For instance:

Virtual Machines. The Java Virtual Machine (JVM), the .NET Common Language Runtime (CLR), the CPython virtual machine, etc. are designed to run across different operating systems[b], but they are also platforms in their own right. Instead of writing software "for Linux" (for instance), one can write software "for the JVM" (even in non-Java languages).

Web Browsers. JavaScript (technically ECMAScript) and its associated web-based APIs provide an increasingly sophisticated platform for web applications. The various popular web browsers today – Chrome, Firefox, Edge (or Internet Explorer), Safari, etc. – have numerous small differences, but are similar enough that (with some care and testing) they can collectively be treated as one platform. One need not even write JavaScript, as other languages can be compiled to it (as if it were a kind of virtual machine code).

Hybrid mobile frameworks. Mobile app development is dominated by the Android and iOS operating systems. Mobile developers must contend with different languages *and* different APIs *and* different hardware. However, there are frameworks such as Cordova/PhoneGap and Xamarin that provide an abstraction layer over the top, allowing you to write a single set of code that will run on (more or less) any device. These systems can either:

- Compile your code into separate Android and iOS distributions; or
- Provide separate Android and iOS wrappers to sit underneath your code at runtime.

So what does platform independence really mean? True platform independence is technically impossible, as you must make at least some assumptions about the user's machine. Using one of the above approaches only narrows those assumptions. In practice, what we really mean by "platform independence" is *operating system independence*.

Right away, you can see that the virtual machines, web browsers, etc. tick this box. They can provide an easy way to implement our cross-platform NFRs. However, they are *not always* the right choice.

What if the thing you're making is itself a VM, or a web browser? You can't have a JVM running on a JVM running on a JVM *ad infinitum*[c]. What if you have high performance or

[a] Technically "Linux" is just the OS kernel, and "GNU/Linux" is – according to some[4] – the more correct name for the broader OS, but the point remains that essentially every component of it can be swapped out for something else, and doing so will inevitably create a somewhat different platform.

[b] .NET is still playing catch-up to Java when it comes to running on different operating systems, but the Mono Project is helping on this front.

[c] Unless you're a mathematician.

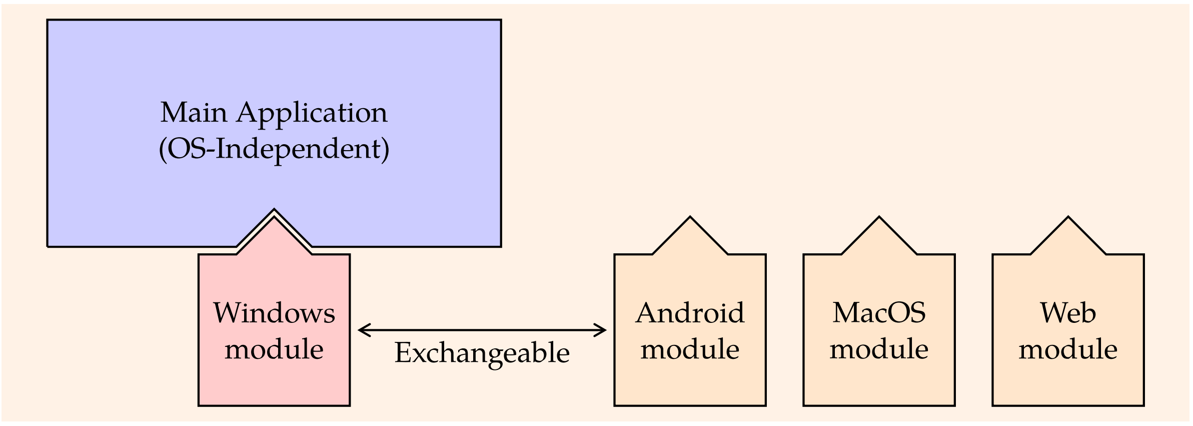

Figure 1: Basic cross-platform architecture.

functionality requirements that cannot be met by a web browser or hybrid mobile framework, and you can't (or don't want to) rely on the user already having a particular VM installed?

How do you engineer platform independence *yourself*? Figure 1 shows the general architectural approach. The idea is simple, and has a lot in common with the Strategy Pattern in OO design. You create a series of platform-specific modules, each of which implements a common interface. The interface should provide all the functionality your application relies on, but each module should be as "lightweight" as possible to avoid duplication.

Generally you would create a separate distribution of your application at compile time, with one corresponding to each platform-specific module. (In principle, you could arrange for the correct module to be loaded at runtime, but this will just make things more complicated than needed.)

We'll get back to build engineering in Lecture 4.

2 Frameworks

Frameworks are like libraries or APIs, but they go further and define how your application will be structured (to some extent). They take care of a lot of otherwise tedious and repetitive stuff, so you can focus on the core functionality. Certain platforms tend to be associated with certain frameworks, and (in some cases) they are not easily separable. Here are some examples:

- Django (djangoproject.com) is a framework for creating Python-based web applications.
- Spring (spring.io) is a framework for creating Java-based web applications.
- The standard Android APIs and development tools form a framework that governs the structure of Android apps.

Whereas an ordinary API provides a set of functions and classes that can be invoked as needed, frameworks weave themselves much more tightly into your application, almost as if they were part of the language:

- They tend to provide an overarching inheritance hierarchy and event handling system, so that parts of your application are expected to extend parts of the framework.
- They tend to define what the component parts of your application are, or at least what "types of things" your application can be made up of. Many frameworks are characterised as having a "Model-View-Controller" structure. Normally, MVC would be

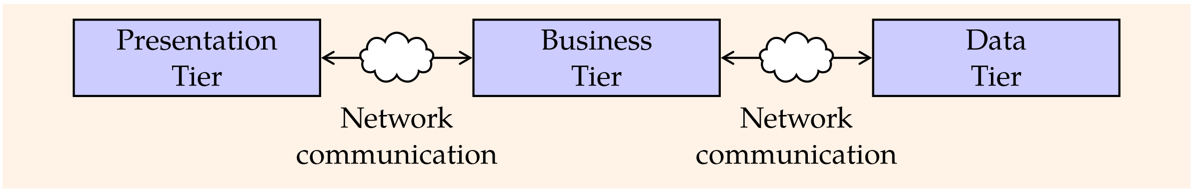

Figure 2: Outline of a distributed system.

something application developers themselves choose to adopt (or not), but an MVC framework forces the issue. Here, the framework itself separates out the UI, the business logic, and the data model.

- They often specify naming and organising conventions. For instance, Ruby on Rails arranges for class and field names to be the same as database table and column names (whereas they could, in principle, be completely different), as part of its stated doctrine of "Convention over Configuration".[1]
- They sometimes provide a build system – a distinct mechanism for compiling your application (as we'll see later).

Frameworks are an architectural concern because the choice must be made up-front, and is very expensive to change. Applications are tightly coupled to whatever framework they are using. Though coupling should usually be minimised, in this case it is a sacrifice made in order to reduce the quantity of code needed, and in turn reducing the cost, time, and number of defects.

Frameworks also contribute towards usability. They take care of much of the UI design, or at least provide structures and conventions that steer developers towards consistent UI practices. You can always develop an application to light up any combination of pixels on the screen (and how else could you write a full-screen video player?) and yet, on any given OS, most applications look mostly the same. This is the result of frameworks. Consequently, users can generally find their way around new apps based on their prior experience. Moreover, the UI choices made by the framework designers are (frankly) based on a great deal more UI expertise than the average software developer has.

Frameworks can create other problems, though, if not well-chosen. Some frameworks may not scale well with large volumes of data, or otherwise not have the performance you need. Some frameworks may not work well (or at all) on certain platforms, or in conjunction with certain services. For some frameworks, you may have difficulty hiring developers who understand them! You can change frameworks, but this tends to involve redoing a lot of your work, which can be painful.

3 Distributed Components

As seen in Distributed Computing (COMP3008), software can be made to be **distributed**. That is, an application can be split up into several distinct components that communicate over a network, as depicted in Figure 1.

There are several important consequences of splitting up a system this way:

- Communication between the components is much slower than between ordinary objects, since it travels across networks. (In practice, the network in question is often a small, fast local-area-network, rather than the Internet, but even a LAN is orders of magnitude slower than the CPU's function call instruction[d].)

[d] This is a physical inevitability due to the speed of light, and the relative distance between networked computers compared to the distance between hardware components within a computer.

- Communication between the components is more complex, due to the need for serialisation and deserialisation. There are "remote procedure calls" that can be made across network links, but these do not work like ordinary method calls[e]. There is a big difference between (a) passing an object reference to a method, and (b) transmitting an object to a remote server.
- With some care, and depending on where the network connections actually are, these systems can be more secure than the non-distributed variety. Even if an attacker compromises one component, the others are not necessarily affected. In Figure 1, for instance, it may only be the presentation tier that is directly accessible over the Internet. The other two tiers may be hidden behind firewalls or intrusion detection systems.

  This is no security silver bullet of course, and it won't help unless the different components perform their own separate data validation. The business tier should not assume that the presentation tier is secure, nor should the data tier assume the business tier has validated data correctly.
- Components are swappable, to some extent. If communication is kept reasonably simple, it may be possible to design, implement and deploy drop-in replacements for any of the components, while the other components remain running.
- System administrators have more immediate and fine-grained control over how computing resources are allocated. If the business tier is particularly CPU-intensive, for instance, administrators could "throw more hardware" at it themselves without any software changes required, and perhaps even without restarting the system. They would just need to duplicate that component onto a number of different machines, and configure their routers accordingly.

Git, the version control system created by Linus Torvolds, illustrates how a distributed system can significantly affect performance for the better.[2] In Git (and other distributed version control systems, like Mercurial), much of the time-consuming processing is done on developers' local repositories, sparing the central server. In this particular case, the distributed architecture even manages to *reduce* overall network traffic too, compared to centralised version control systems.

4 Client-Server Architecture and "The Cloud"

Applications involving a multiple clients connecting to a central server are a special case of distributed components. Web applications and many mobile apps are a common case. This has some significant benefits:

- For web applications in particular, the user doesn't need to install or update anything, except for the web browser itself. All they need is the URL to the web application. This is possible because the client code (the JavaScript) is simply transmitted from the server.
- Not only can users use (practically) any device, but they can effortlessly access the same data across multiple devices.

  Non-server-client architectures just can't match this. While there are several obvious ways users might transfer files between devices – Windows file sharing, NFS, external

[e] RPCs can often *look* a bit like, or even exactly like, ordinary method calls, but this is a superficial resemblance. One must be aware of what is actually happening underneath, or else risk blundering into huge performance bottlenecks or other weird defects.

drives – these all require the user to keep track of what they are transferring. (The user's own brain is a resource whose usage should be optimised!)

- Implementation of "sharing" features between users is significantly easier

However, as with all architectural decisions, there are tradeoffs:

- Client-server architectures require some sort of authentication system. The server must be able to keep track of user accounts, and passwords, or interact with another system that does this (Google, Facebook, etc.).
- Security and privacy are significantly affected, to the extent that entirely new NFRs probably need to be written to deal with this architectural decision. User data is not entirely safe on users' own machines/devices, of course, but users themselves are assumed to be responsible for anything stored on their own physical computing property. Where users' data is stored on a central server, it is an inherently more interesting target for attackers (it's all there together), and it is the application's responsibility to secure it.

  For some context, there are already known to be *billions* of compromised accounts from various sites across the Internet.[3]

5 Plugins and Scriptability

Extensibility is a common theme in many popular pieces of software. With the ability to write plugins or scripts, technically knowledgeable users can extend the software without needing to access or modify the original code. It's a way to future-proof your application – to provide a way to satisfy possible future requirements. Ultimately, it can be the basis for a *community* to form around the software.

Extensibility mechanisms are known by various names: plugins, scripts, add-ons, extensions, etc.

There are many examples of them, especially for large and widely-distributed software packages:

- Modern web browsers allow third-party developers to write extensions. Browsers already provide a platform to web applications, and a lot of the same software infrastructure (JavaScript and its APIs) is reused to support browser extensions.
- Desktop office productivity software – Microsoft Office and Libre/OpenOffice – have long supported scripting through a variant of Basic.
- Many games can be extended through Lua scripts.

Scriptability is, of course, an architectural decision. You need an embedded interpreter to handle the execution of the scripts. But you also need to think carefully about what events in your application will trigger script execution, and what parts of your application will serve as an API for the script. In a sense, you are creating a mini-platform all of your own, and a whole new class of users: script writers.

You must start to consider NFRs relevant to script writers. How easy is it to script your application? Is the API you've created easily understandable to script writers? Will it allow them to write clean, concise scripts with minimal effort?

Scriptability poses significant security challenges as well, and you may need to think very carefully about it if security features prominently amongst your requirements. If you encourage your users to develop and use third-party add-ons, you're also providing an attack vector through which your software, and thus the user's machine, can be compromised. Once you allow third-party code to run, you have to expect that it could (and will) do everything allowed by your API, and also everything allowed by the underlying scripting language's API.

There are approaches to sanitise or lock-down certain features of the language (e.g. file IO or network access) to avoid abuse, but this is extremely difficult to do properly. There are often ways for malicious code the "escape", at which point an attacker could have complete access to the user's files and network connection.

6 Multithreading

Multithreading can help applications achieve performance requirements. In the simplest sense, your software can utilise multiple CPU cores simultaneously. It can also schedule CPU-intensive tasks (computations) to run alongside IO-intensive tasks (e.g. disk/network transfers), rather than one needing to wait for the other.

Multithreading can also benefit *usability*, by allowing the user interface to remain operational while a long-running task is performed in the background.

We'll discuss this in a lot more depth in the next two lectures.

7 From Here On

Throughout the semester we will look at software architecture, and also some lower-level implications of it. For instance, implementing **concurrency** requires a great deal of care. And if we're developing a system with a complex architecture, **build engineering** becomes important as a way of managing dependencies, compilation, testing and packaging.

Further Reading

- Simon Brown (2010), *Are You a Software Architect?*, www.infoq.com/articles/brown-are-you-a-software-architect.
- Martin Fowler (2003), *Errant Architectures*, www.drdobbs.com/errant-architectures/184414966.

References

[1] David Heinemeier Hansson. The Rails Doctrine. http://rubyonrails.org/doctrine/, 2016. Accessed: 2017-07-28. **[See section (document).]**

[2] Giancarlo Lionetti. What is version control: centralized vs. DVCS. https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs, 2012. Accessed: 2017-07-27. **[See section (document).]**

[3] Lily Hay Newman. Inside LeakedSource and its Database of 3 Billion Hacked Accounts. https://www.wired.com/2016/12/inside-leakedsource-database-3-

billion-hacked-accounts/, 2016. Accessed: 2017-07-28. **[See section (document).]**

[4] Richard Stallman. Linux and the GNU System. https://www.gnu.org/gnu/linux-and-gnu.en.html, 2017. Accessed: 2017-07-27. **[See section a.]**