Software Engineering Concepts (COMP3003)

6a. API Design

Updated: 07/10/2019.

Software architecture concerns a great deal of system-to-system interaction. This happens where you divide a system into components (which act like mini-systems), and also where your system must use existing systems. System-to-system interaction requires **Application Programming Interfaces** (APIs).

You've used them already. It's virtually impossible to write useful code without the **standard API** of your chosen language. But there are many more different APIs out there too, and at some point you'll probably have to make one.

There are two sides to an API: the system that *uses* it (the client), and the system that *implements* it (the provider[a]). Using an API is just a matter of consulting its documentation, and using its features as needed. Implementing an API involves making a system that provides those features, as documented, so that another developer can use it – potentially anyone, anywhere in the world.

What we're really interested in here is how to *design* an API in the first place, to make this possible. How do we decide how multiple systems – written by different people in different organisations at different points in time – can and should interact?

1 Types of APIs

There are various generic ways to get different systems to interact. APIs can be built on top of any of them (thus providing a *more specific* way for systems to interact), but what an API is built on determines a lot about how it must work.

We'll briefly summarise three different interaction mechanisms that lead to different types of APIs:

- Compile-time or runtime linking. As shown in Figure 1, we could be trying to use a library, or load plugins.

  For now, we'll consider libraries and plugins to be separate systems from the main application, even though at runtime they are combined into the same running process[b]. For libraries, we may even combine them with the main application at compile-time. However, from a project management point of view, they are still very much separate software *projects*.

  In these cases, the API can use the complete range of programming language declarations in whatever language you're using[c]. In C, you can use anything you would typically find in

[a] "Provider" may not actually be a widely-used term, but it's helpful to have a sufficiently-generic term in this discussion.

[b] To recap, a **process** is a running instance of a program (just as an object is an instance of a class, in object orientation). We can't just say "program", because you can run the same program multiple times simultaneously, each instance having its own separate data and state.

[c] To expand on *why* this is the case: it's basically because it's the compiler/linker that joins the two systems together, and it knows the language.

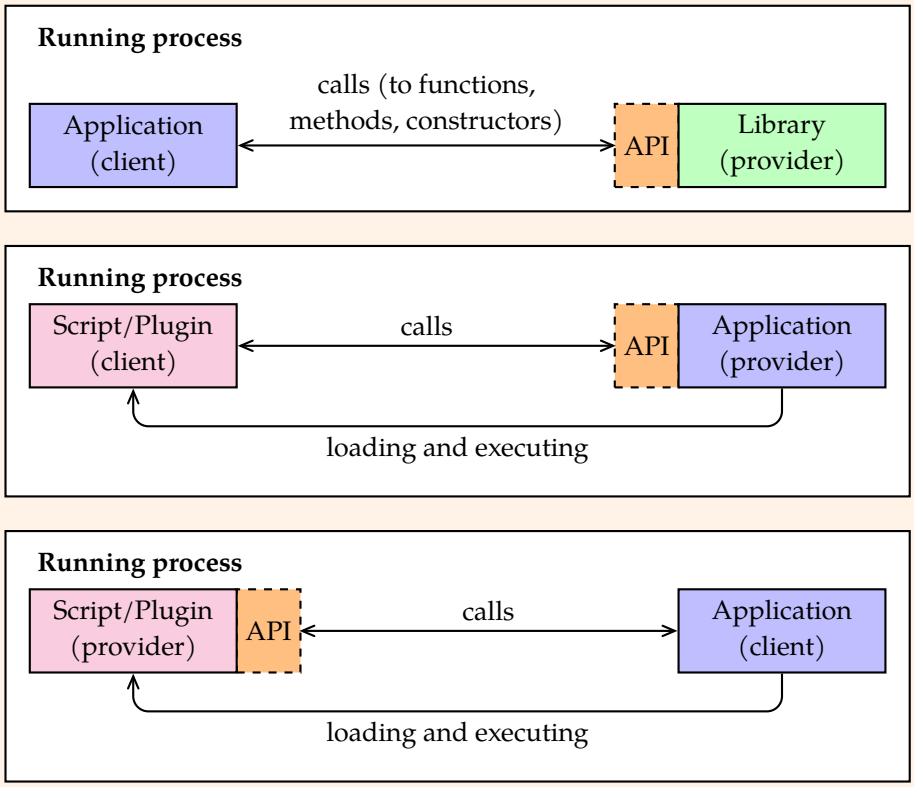


Figure 1: APIs based on programming languages, using classes, objects, functions, methods and whatever else the language may provide.

header files: function declarations, typedefs, structs, enums and constants[d]. In Java, you can use packages, classes, interfaces, methods and constant fields.

- Interprocess communication, as show in Figure 2. Here, the client and the provider exist on the same machine, but in different processes.

  Unfortunately, language-level concepts like classes, interfaces, methods, etc. don't work between different processes. Processes run concurrently (like threads), so a method/function call from one to the other doesn't make sense. They also have independent memory, so transferring object references makes no sense either. Finally, two processes may be running code based on different languages, and there's no guarantee that any two languages share compatible concepts of classes, interfaces and methods.

  What we *do* have – what all languages and operating systems support – is the ability to send and receive streams of bytes. API design can work with this. Just be aware that this leads to quite a different *kind* of API.

- Network communication, as shown in Figure 3. This involves two systems on entirely separate machines, and here APIs are typically built on top of HTTPS (HyperText Transfer Protocol Secure). As with interprocess communication, we cannot use language constructs to mediate between the two systems.

  Web APIs also have performance-related limitations:

  - Internet access is potentially billions of times slower than the inside of a CPU and RAM, and also relatively unreliable.
  - On mobile devices, internet access can consume a lot of battery charge.

[d] You can *technically* have global variables as part of your API, but this is a monumentally bad idea.

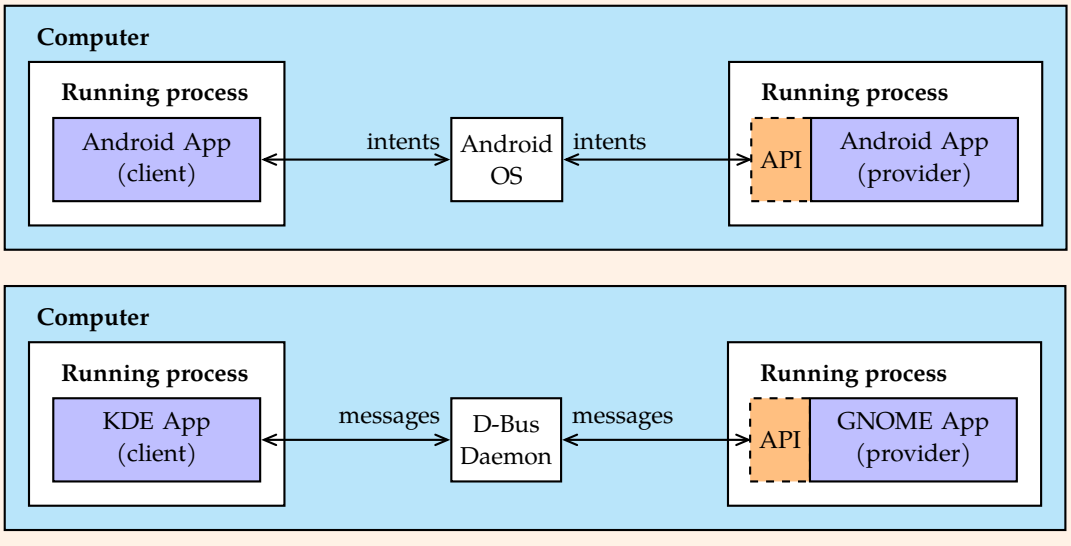


Figure 2: APIs based on inter-process communication.

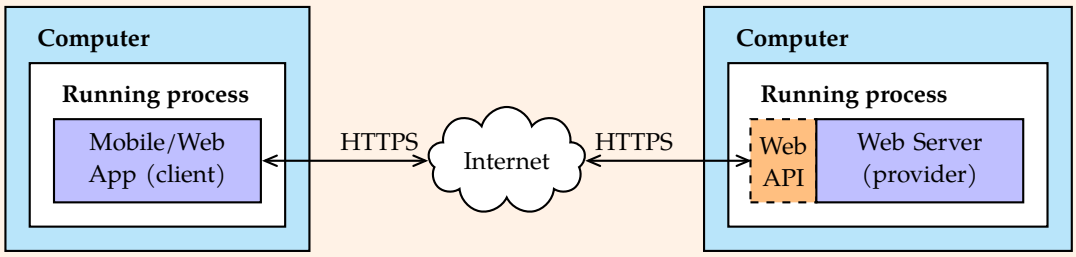


Figure 3: APIs based on network communication.

2 Parts of an API

For the most part, we'll be looking at library/plugin APIs, designed to work within a running process. Since these APIs have access to the full range of programming language constructs, they are generally the most complex ones.

This section will look at the role that different language constructs can play in APIs. (However, a note of caution: you're not designing or using an API just because you're designing or using any of the constructs listed below. They can *all* also form part of the internal structure of a software system.)

2.1 Functions and Static Methods

Very simple APIs may consist solely of functions or static methods (depending on the language). It is actually quite difficult to find examples of such APIs, though, as there's typically good reason to take advantage of more sophisticated constructs too.

Parts of some APIs (particularly a language's standard API) are designed this way, though:

- Much of the C standard API (though not all of it), particularly math and string functions.

Listing 1: Standalone functions in the standard C API.

```
/* string.h */
size_t strlen(const char *s);
char *strcpy(char *dest, const char *src);
char *index(const char *s, int c);
...
```

- The `Math` class in standard Java.

Listing 2: Standalone static methods in the standard Java API.

```
package java.lang;
public final class Math
{
    public static double abs(double a) {...}
    public static double atan2(double y, double x) {...}
    public static double random() {...}
    ...
}
```

- The `IOUtils` class in the Apache Commons IO library.[2]

Listing 3: Standalone static methods in the Apache Commons IO library.

```
package org.apache.commons.io;
public class IOUtils
{
    public static void copy(InputStream in, Writer out, Charset enc) {...}
    public static List<String> readLines(InputStream in, Charset enc) {...}
    ...
}
```

Keep in mind that the API stops where the public (or protected) declarations stop. I've omitted the code here, because that *isn't* part of the API. Rather, it's just part of the API provider's implementation. The API itself is essentially just declarations (and their documentation), though not all declarations are necessarily part of the API.

2.2 New Datatypes

The vast majority of library APIs go beyond just a set of functions or static methods. They almost always define a set of new concepts, best represented by new datatypes. They bundle together related information into structs (in C) or classes (in OO languages). They assign "enum" constants to represent certain API options or states.

For instance:

- The C-based Simple DirectMedia Layer (SDL) library[1] (which provides cross-platform graphics, audio, keyboard, mouse, etc. functionality) has a number of structs and enums for bundling together related data and representing options:

Listing 4: Selected datatypes defined in the API for Simple DirectMediaLayer.

```
/* SDL_surface.h */
typedef struct SDL_Surface   /* Represents a "canvas" to be drawn on. */
{
    SDL_PixelFormat *format;
    int w, h;
    ...
} SDL_Surface;
```

```
/* SDL_mouse.h */
typedef enum { SDL_MOUSEWHEEL_NORMAL,
               SDL_MOUSEWHEEL_FLIPPED } SDL_MouseWheelDirection;
```

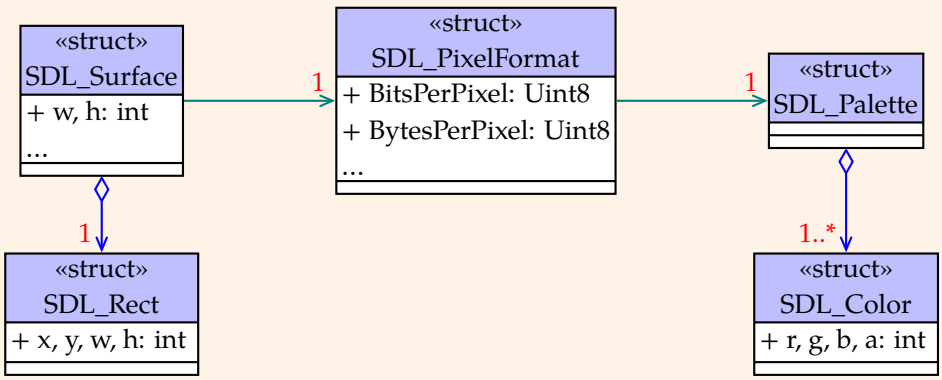


Figure 4: UML overview of some of the structures in the API for Simple DirectMedia Layer.

These types are then used as parameters to and returns from various SDL functions.

- GUI libraries like JavaFX[4] include numerous classes for representing parts of the GUI itself (buttons, checkboxes, etc.), but also other kinds of abstractions. For instance:

Listing 5: Selected JavaFX classes.

```
package javafx.scene.control;
public class CheckBox extends ButtonBase
{
    public Checkbox() {...}
    public boolean isSelected() {...}
    ...
}
```

```
package javafx.application;
public enum ConditionalFeature
{
    CONTROLS, EFFECT, FXML, GRAPHICS, INPUT_METHOD, INPUT_MULTITOUCH, ...;
}
```

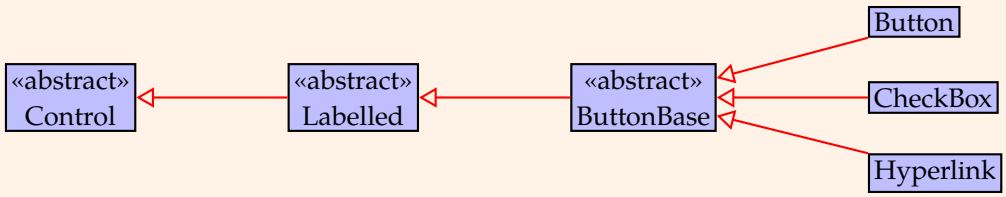


Figure 5: UML overview of selected classes from the JavaFX API.

Here too, there is an important dividing line between what's part of the API itself and what's part of the implementation. API providers often have "implementation" classes that, while needed to make things work, are outside the API specification. Even for classes that *are* part of the API, in order to implement them properly, the API provider must add in the actual code, as well as certain private fields, private methods, private nested classes, and private constructors, and these things are *not* part of the API.

Not having access modifiers, C has other tricks to accomplish something similar:

- Declaring a function `static`, and omitting it from the header file, effectively makes it "private" to the current .c file.
- Using "incomplete types" can allow client code to *refer* to a struct, and pass it around, but without actually being able to access it. Thus, you could define a struct pointer type as part of the API, but leave the struct fields to the implementation.

2.3 Service Classes

As in OO generally, classes are not just for data; often they represent tasks. Many APIs are there to help you perform a complex or highly configurable task. While the API *could* technically do this via a single static method, sometimes this is the wrong tool for the job. If the client needs more control over the process, it may be desirable for the API to have an entire class (or interface) to represent the action.

The Apache Commons CSV library[3] is a simple example of this. It helps read and write CSV files, for which it contains the classes `CSVParser` and `CSVPrinter`. Objects of these classes represent "services" for getting data out of a CSV file or putting it in[e]. They contain methods that help perform these actions incrementally, as well as providing additional information. (If Apache Commons CSV instead only had static `parse()` and `print()` methods, it would be difficult to preserve the same level of functionality.)

Service classes/objects have advantages:

- Your API can have state without having *global* state. In other words, your API can remember things (like the client's settings), but still let multiple parts of the client system deal with the API independently of one another. This can happen because you can create multiple objects of a given service class.

  (As an analogy, it's nice if the staff at your local restaurant remember your favourite order, and not useful if they only remember the most popular order overall.)

  Keeping state can simplify the way that the client interacts with the API. Global state (which is created by global variables, and also by static class fields, even if private), can be limiting and have unexpected side-effects. Moreover, in a multi-threaded situation, global state must be frequently locked. This is both messy and error-prone compared to, say, using a different service object in each thread, in which case threading issues don't arise.

- You can configure several different service objects to behave in different ways, and use them all at the same time for slightly different purposes.

- A future version of the API may create a subclass that overrides methods in the service class. This can be done to provide new features or functionality, but without the need to make any significant changes to the client code.

  (There are limits to what you can do this way. You can't expect *all* new API functionality to magically get absorbed by the client without modification.)

2.4 Callbacks and Back-References

Notice in Figure 1 that the "call" arrows go *both ways*. If it were only one way – client to API provider – we'd be limited in what we could do. The client could use the API to perform actions, or return data, but receiving notifications would be trickier.

The client could **poll** the API; i.e. make periodic calls to determine whether an event has happened yet. This is often considered wasteful, though it is the way some APIs operate. In particular Simple DirectMedia Layer has a `SDL_PollEvent()` function that checks for various kinds of input – keyboard/mouse/touch, etc. It reports the details, if any such input has been received since the last time the function was called.

Polling isn't a good general solution. It's wasteful, because you're using CPU resources even when nothing is happening. Also, the client may not always be able to perform regular (or regular *enough*) checks. For instance, it might be important that the notifications happen *during* some

[e] While lists of comma-separated values may not *seem* complicated at first, there are actually many variants of the format with many subtle differences.

other call to the API, or during some long-running process on the client. Polling facilitates event handling from within a single thread, so if we're going to have multiple threads anyway, its rationale is not very strong.

So, other APIs typically deal with **callbacks**. A callback is a function or method that exists in the *client* code, and is called by the API provider; i.e. the reverse of the normal calling direction. However, the API provider doesn't know anything about the client, so the client must first specify *which* function(s) or method(s) should be called.

In C, callbacks imply pointers-to-functions. The API would define a type for the callback, a function to register it, and various other functions that perform various tasks in which the callback might need to be called:

Listing 6: Generic example C API that accepts callbacks.

```c
/* my-api.h */
typedef void (* MyCallbackType)(void* data, int param1, int param2);
void registerCallback(MyCallbackType cb, void* data);
void doImportantThings();
```

Then the client code would define and register its callback:

Listing 7: Generic example C client that registers a callback.

```c
/* client.c */
#include <my-api.h>

void actualCallback(void* data, int x, int y)
{
    printf("%d, %d", x, y); /* Do something important */
}

int main(void)
{
    registerCallback(actualCallback, NULL);
    doImportantThings();
    return 0;
}
```

This is a relatively trivial case. The client expects that a call to `doImportantThings()` could (or will) result in another call "back" to the client's own code in `actualCallback()`. If/when this happens, the API may provide some information (two `int`s in this case). The *timing* of the callback itself is often of interest. It may (implicitly) tell the client that a particular event has just happened, so that the client can respond.

The API also typically provides a `void*` parameter, which gives the client a way to indirectly pass information to its own callback. The API simply takes in that pointer when the callback is registered, and passes it on to the callback when needed. This is particularly important in C, since there's no other "nice" way (short of global variables) for the callback to access the client's own data.

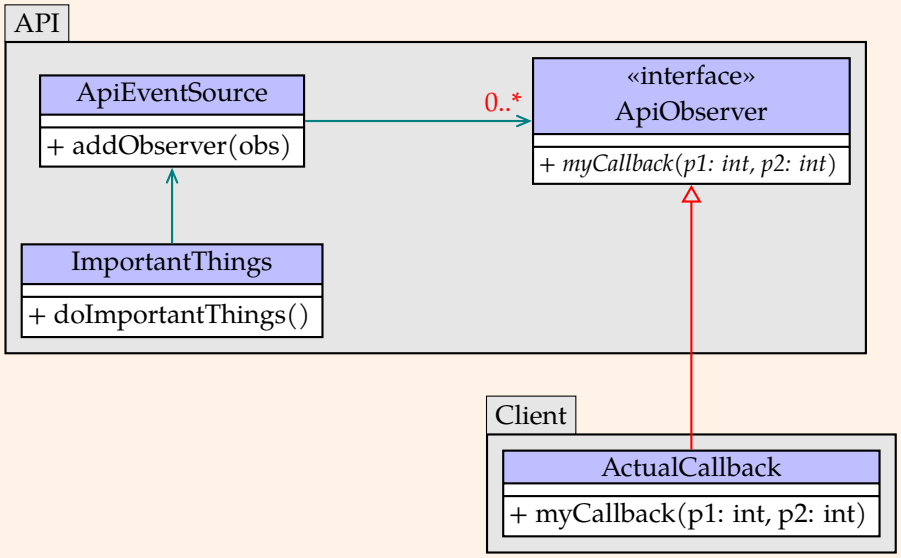
In fact, Simple DirectMedia Layer *also* supports callbacks in this way, and so contains the following:

Listing 8: Callback support in the Simple DirectMedia Layer API.

```
/* SDL_events.h */
typedef int (* SDL_EventFilter)(void *userdata, SDL_Event *event);
void SDL_AddEventWatch(SDL_EventFilter filter, void* userdata);
...
```

OO APIs generally use the Observer Pattern (or some other polymorphic design) to represent callbacks. In this case, the API would define a callback as an "observer" interface, with a separate class keeping track of the observers, and various other classes that perform various tasks in which the observers might need to be notified:

Figure 6: UML overview of the Observer Pattern, as might be used in an API. ("ImportantThings" is just a placeholder for any part of the API that might trigger an event.)



Listing 9: Example Java API that accepts callbacks.

```
package org.example.myapi;
public interface ApiObserver
{
    void myCallback(int param1, int param2);
}
```

```
package org.example.myapi;
public class ApiEventSource
{
    public void addObserver(ApiObserver obs) {...}
    ... // There would *also* be a list of observers, and a notifyObservers()
        // method to actually call myCallback(), but those would be
        // considered part of the implementation, not the API.
}
```

```
package org.example.myapi;
public class ImportantThings
{
    public void doImportantThings();
}
```

Then the client code would implement the interface, create a new instance, and register it:

Listing 10: Example callback implementation in the client.

```
import org.example.myapi;
public class ClientMain
{
    // Define the callback
    private static class ActualCallback implements ApiObserver
    {
        @Override
        public myCallback(int x, int y)
        {
            System.out.println(x + ", " + y); // Do something important
        }
    }

    public static void main(String[] args)
    {
        // Instantiate API objects as needed
        ApiEventSource es = new ApiEventSource();
        ImportantThings it = new ImportantThings(es);

        // Instantiate the callback object
        ActualCallback ac = new ActualCallback();

        // Register callback
        es.addObserver(ac);

        // Make API call(s) that may result in the callback being called.
        it.doImportantThings();
    }
}
```

The Observer Pattern is perhaps the most *usual* way of doing this in OO languages. However, if the API provides any interfaces or abstract classes for the client to implement/extend, then you're basically dealing with callbacks of one sort or another. For instance, this could happen using the Strategy Pattern, the Template Method Pattern, the Decorator Pattern, or the Composite Pattern too, if API allows the client to define a subclass and override methods.

References

[1] Simple DirectMedia Layer. https://wiki.libsdl.org/FrontPage. Accessed: 2019-10-07. [**See section 2.2.**]

[2] The Apache Software Foundation. `org.apache.commons.io.IOUtils`. https://commons.apache.org/proper/commons-io/javadocs/api-release/org/apache/commons/io/IOUtils.html, 2017. Accessed: 2019-10-07. [**See section 2.1.**]

[3] The Apache Software Foundation. Commons CSV. https://commons.apache.org/proper/commons-csv/, 2019. Accessed: 2019-10-07. [**See section 2.3.**]

[4] Gluon. JavaFX. https://openjfx.io/. Accessed: 2019-10-07. [**See section 2.2.**]

[5] Martin Reddy. *API Design in C++*. Elsevier, 2011. [**See section (document).**]